RANCANG BANGUN VOIP BERBASIS P2P MENGGUNAKAN DHT KADEMLIA

SKRIPSI

Untuk memenuhi sebagian persyaratan
memperoleh gelar Sarjana Komputer

Disusun oleh:
I Wayan Vendy Wiranatha
NIM: 125150207111008

PROGRAM STUDI INFORMATIKA/ILMU KOMPUTER
FAKULTAS ILMU KOMPUTER
UNIVERSITAS BRAWIJAYA
MALANG
2016

PENGESAHAN

RANCANG BANGUN VOIP BERBASIS P2P MENGGUNAKAN DHT KADEMLIA

SKRIPSI

Diajukan untuk memenuhi sebagian persyaratan
memperoleh gelar Sarjana Komputer

Disusun Oleh :
I Wayan Vendy Wiranatha
NIM: 125150207111008

Skripsi ini telah diuji dan dinyatakan lulus pada
28 April 2016
Telah diperiksa dan disetujui oleh:

Dosen Pembimbing I	Dosen Pembimbing II
Adhitya Bhawiyuga, S.Kom, M.S	Gembong Edhi Setyawan, S.T, M.T
NIK: 201405 890720 1 001	NIK: 201208 761201 1 001

Mengetahui
Ketua Program Studi NamaProgramStudi

Issa Arwani, S.Kom, M.Sc
NIP: 19830922 201212 1 003

PERNYATAAN ORISINALITAS

Saya menyatakan dengan sebenar-benarnya bahwa sepanjang pengetahuan saya, di dalam naskah skripsi ini tidak terdapat karya ilmiah yang pernah diajukan oleh orang lain untuk memperoleh gelar akademik di suatu perguruan tinggi, dan tidak terdapat karya atau pendapat yang pernah ditulis atau diterbitkan oleh orang lain, kecuali yang secara tertulis disitasi dalam naskah ini dan disebutkan dalam daftar pustaka.

Apabila ternyata didalam naskah skripsi ini dapat dibuktikan terdapat unsur-unsur plagiasi, saya bersedia skripsi ini digugurkan dan gelar akademik yang telah saya peroleh (sarjana) dibatalkan, serta diproses sesuai dengan peraturan perundang-undangan yang berlaku (UU No. 20 Tahun 2003, Pasal 25 ayat 2 dan Pasal 70).

Malang, 13 April 2016

I Wayan Vendy Wiranatha

NIM: 125150207111008

KATA PENGANTAR

Puji dan syukur penulis panjatkan kepada Tuhan Yang Maha Esa karena berkat rahmat dan karunia-Nya, penulis dapat menyelesaikan laporan skripsi yang berjudul "Rancang Bangun VoIP Berbasis P2P Menggunakan DHT Kademlia" ini dapat terselesaikan.

Penulis sangat menyadari bahwa skripsi ini tidak dapat terselesaikan tanpa bantuan dari beberapa pihak. Oleh karena itu, penulis ingin menyampaikan rasa hormat dan terima kasih yang sebesar-besarnya kepada:

1. Bapak Adhitya Bhawiyuga, S.Kom., M.S dan Bapak Gembong Edhi Setyawan, S.T., M.T selaku dosen pembimbing skripsi yang telah membimbing dan mengarahkan penulis untuk dapat menyelesaikan skripsi ini.
2. Bapak Issa Arwani, S.Kom, M.Sc selaku ketua Program Studi Informatika.
3. Bapak Eko Sakti P., S.Kom., M.Kom dan Bapak Aswin Suharno, S.T., M.Sc selaku dosen penguji I dan dosen penguji II yang telah memberikan arahan, saran, dan masukan sehingga skripsi ini dapat diselesaikan dengan lebih baik.
4. Keluarga serta saudara yang telah memberi dukungan berupa nasehat, kasih sayang, semangat, dan doa untuk dapat menyelesaikan skripsi ini, serta kesabarannya dalam membesarkan dan mendidik penulis.
5. Siti Azza Amira, Rangga Dinata B., Zata Ismah, I Gde Yogi M, I Made Yoga A., RM. Ragiel Sakti B, I Made Candra Girinata, Siwi R. Januar serta teman – teman asisten sistem operasi 2015/2016 atas dukungan, saran dan kesediannya membantu pengadaan *smartphone android* sebagai sarana pengembangan aplikasi dan pengujian sistem dalam penelitian ini. Sehingga skripsi ini dapat terselesaikan dengan baik.
6. Seluruh civitas akademika Informatika Universitas Brawijaya yang telah banyak memberi bantuan dan dukungan selama penulis menempuh studi di Informatika Universitas Brawijaya dan selama penyelesaian skripsi ini.

Dalam penyusunan skripsi ini, penulis menyadari bahwa masih banyak kekurangan, sehingga penulis sangat mengharapkan adanya kritik dan saran yang membangun. Akhir kata penulis berharap skripsi ini dapat berguna dan bermanfaat bagi semua pihak yang membaca.

Malang, 13 April 2016

Penulis

vendywira@gmail.com

ABSTRAK

VoIP pada umumnya dibangun di atas arsitektur *client-server*. Arsitektur ini kerap mengalami permasalahan *single failure.* Untuk mengatasi hal tersebut dipilih arsitektur *peer-to-peer* sebagai gantinya. Arsitektur *peer*-to-*peer* memiliki keunggulan dalam bidang skalabilitas dan pendistribusian beban sumber daya. Untuk dapat menemukan calon penerima panggilan dilakukan mekanisme pencarian *peer*. Perncarian *peer* ini menggunakan metode *DHT Kademlia*. *DHT Kademlia* dipilih karena memiliki tingkat skalabilitas yang tinggi dan merupakan salah satu *DHT* dengan perfoma terbaik dibandingkan *CAN, Chord, Pastry* dan *Tapestry* pada kasus *VoIP*. Dalam pencariannya *DHT Kademlia* menggunakan prinsip *binary search tree*. Setelah *peer* dapat ditemukan kemudian dilakukan mekanisme pertukaran pesan untuk memulai panggilan *VoIP*. Dari hasil penelitian dapat disimpulkan bahwa *VoIP* berbasis *peer*-to-*peer* dengan menggunakan *DHT Kademlia* terbukti memiliki skalabilitas sistem yang tinggi dan performa yang baik serta kualitas *QoS* yang dihasilkan telah memenuhi standar *ITU-T* dengan predikat baik.

Kata kunci: *VoIP, P2P, DHT, Kademlia.*

ABSTRACT

Generally VoIP is built on client-server architecture. But the client-server architecture has a problem of single failure. peer-to-peer architecture can be handle it. peer-to-peer architecture has advantage of high scalability and distributed resource. To find candidates for calls use find node mechanism. DHT Kademlia was selected for find node mechanism. Kademlia was chosen becouse it have a high level for scalability and performs better than other DHT such as CAN, Chord, Pastry dan *Tapestry on VoIP case. On DHT Kademlia use binary search tree to find a node. If the node can be found, both a peer can make call with exchange some message. The result on this research concluded VoIP system on peer-to-peer use DHT Kademlia shown to have high system scalability, performance as well as and the quality of the resulting QoS meets the standards ITU-T with a good predicate.*

Keyword: VoIP, P2P, *DHT*, *Kademlia*.

DAFTAR ISI

PENGESAHAN ii
PERNYATAAN ORISINALITAS iii
KATA PENGANTAR iv
ABSTRAK v
ABSTRACT vi
DAFTAR ISI vii
DAFTAR TABEL x
DAFTAR GAMBAR xi
BAB 1 PENDAHULUAN 1
1.1 Latar Belakang 1
1.2 Rumusan Masalah 2
1.3 Tujuan 2
1.4 Manfaat 2
1.5 Batasan Masalah 3
1.6 Sistematika Pembahasan 3
BAB 2 LANDASAN KEPUSTAKAAN 5
2.1 Kajian Pustaka 5
2.2 Landasan Teori 6
2.1.1 *VoIP (Voice over Internet Protocol)* 6
2.1.2 Arsitektur *client-server* 6
2.1.3 Arsitektur *peer-to-peer* 7
2.1.4 *DHT Kademlia* 7
2.1.5 *QoS* (Quality of Service) 8
2.1.6 Skalabilitas 9
2.1.7 *Codec* 10
2.1.8 Protokol 10
2.1.9 *Bandwidth* 10
BAB 3 METODOLOGI 11
3.1 Identifikasi Masalah 11
3.2 Studi Literatur 12

3.3 Ruang Lingkup Jaringan 12
3.4 Perancangan Sistem 12
3.5 Implementasi 12
3.6 Pengujian dan Analisis 13
3.6.1 Pengujian *Query Time* 13
3.6.2 Pengujian Skalabilitas Sistem 14
3.6.3 Pengujian *QoS (Quality of Service)* 15
3.6.4 Pengujian Penggunaan *Bandwidth* 16
3.6.5 Pengujian Penggunaan *Codec* 16
3.7 Penarikan Kesimpulan. 17
BAB 4 PERANCANGAN SISTEM 18
4.1 Arsitektur Umum Sistem 18
4.2 Gambaran Umum Sistem 19
4.3 Perancangan *Peer* 21
4.4 Perancangan Kademlia 21
4.4.1 Perancangan Pesan *RPC* (*Remote Procedure Call*) 21
4.4.2 Perancangan *KBucket* 23
4.4.3 Perancangan Bootstraping 24
4.4.4 Perancangan Pencarian *Node* 26
4.5 Perancangan *VoIP* (Voice over Internet Protocol) 27
4.5.1 Perancangan Panggilan Suara 27
4.5.2 Perancangan Pemilihan dan Penggunaan *Codec* 32
4.5.3 Perancangan Transmisi Data 33
BAB 5 IMPLEMENTASI 34
5.1 Implementasi *Peer* 34
5.2 Implementasi Kademlia 34
5.2.1 Implementasi Pesan *RPC* 35
5.2.2 Implementasi *KBucket* 44
5.2.3 Implementasi Mekanisme *Bootstrapping*. 49
5.2.4 Implementasi Mekanisme Pencarian *Node* 55
5.3 Implementasi *VoIP*. 62
5.3.1 Implementasi Mekanisme Panggilan Suara 62

5.3.2 Implementasi Pemilihan dan Penggunaan *Codec* 73
5.3.3 Implementasi Transmisi Data 77
BAB 6 PENGUJIAN DAN ANALISIS 86
6.1 Hasil dan Analisis Pengujian *Query Time* 86
6.1.1 *Join Time* 86
6.1.2 *Find Time* 89
6.2 Hasil dan Analisis Pengujian Skalabilitas 92
6.2.1 Pengujian Skalabilitas *Join Node.* 92
6.2.2 Pengujian Skalabilitas *Leave Node.* 93
6.3 Hasil dan Analisis Pengujian *QoS* 96
6.3.1 *Packet loss* 96
6.3.2 *Jitter* 96
6.3.3 *Delay* 97
6.4 Hasil dan Analisis Pengujian *Bandwidth* 98
6.5 Hasil dan Analisis Pengujian Penggunaan *Codec* 99
BAB 7 PENUTUP 102
7.1 Kesimpulan 102
7.2 Saran 103
DAFTAR PUSTAKA 104

DAFTAR TABEL

Tabel 3.1 Tanda Pengenal *Node* Pada Topologi Pengujian Skalabilitas. 14
Tabel 6.1 *Join* Time Skenario Ke-1 86
Tabel 6.2 *Join* Time Skenario Ke-2. 86
Tabel 6.3 *Join Time* Skenario Ke-3 87
Tabel 6.4 *Join* Time Skenario Ke-4. 87
Tabel 6.5 Hasil Pengukuran Proses *Join Time*. 88
Tabel 6.6 *Find Time* Skenario Ke-1. 89
Tabel 6.7 *Find Time* Skenario Ke-2. 89
Tabel 6.8 *Find Time* Skenario Ke-3. 89
Tabel 6.9 *Find Time* Skenario Ke-4. 90
Tabel 6.10 Hasil Pengukuran Proses *Find Time*. 91
Tabel 6.11 Hasil Pengujian Skalabilitas Pada Skenario *Join Node*. 92
Tabel 6.12 Hasil Pengujian Skalabilitas Leave *Node*. 93
Tabel 6.13 Pengujian *Packet loss* 96
Tabel 6.14 Kategori *Pakect Loss* 96
Tabel 6.16 hasil Pengujian *Jitter*. 96
Tabel 6.15 Standard ITU-T 96
Tabel 6.17 Hasil Pengujian *Delay* 97
Tabel 6.18 Standard *Delay* ITU-T 97
Tabel 6.19 Pengujian *Bandwidth* Terhadap *Packet loss*. 98
Tabel 6.20 Pengujian *Bandwidth* Terhadap *Jitter*. 98
Tabel 6.21 Pengujian *Bandwidth* Terhadap *Jitter*. 98
Tabel 6.22 Pengujian *Bandwidth* Terhadap *Delay*. 99
Tabel 6.23 Pengujian Penggunaan Codec Terhadap Packet Loss 99
Tabel 6.24 Pengujian Penggunaan Codec Terhadap Jitter 100
Tabel 6.25 Pengujian Penggunaan Codec Terhadap Delay 100

DAFTAR GAMBAR

Gambar 2.1 Arsitektur *Client-Server* 6
Gambar 2.2 Arsitektur *Peer-To-Peer Overlay*. 7
Gambar 2.3 Mekanisme Pencarian *Node* Pada *DHT Kademlia* 8
Gambar 3.1 Diagram Metodologi Penelitian 11
Gambar 3.2 Topologi Pada Pengujian Skalabilitas 14
Gambar 3.3 Diagram Alir Pengujian Skalabilitas 15
Gambar 4.1 Gambaran Umum Arsitektur Sistem 18
Gambar 4.2 Gambaran Umum Sistem 19
Gambar 4.3 Diagram Alir Kerja Sistem. 20
Gambar 4.4 Perancangan *Peer* 21
Gambar 4.5 Perancangan *KBucket* 23
Gambar 4.6 Perancangan KBucket *Cache* 24
Gambar 4.7 Perancangan Bootstrapping 25
Gambar 4.8 Perancangan Mekanisme *Finding Node* 26
Gambar 4.9 Perancangan Sistem *Voip* Secara Umum 28
Gambar 4.10 Kondisi Pada Saat Melakukan Panggilan Suara Dari Sisi Penelpon. 29
Gambar 4.11 Kondisi Pada Saat Terjadi Panggilan Suara Dari Sisi Penerima 31
Gambar 5.1 Kode Implementasi Kelas *PeerDescriptor.* 34
Gambar 5.2 Kode Implementasi Kelas *KadBasicMessage.* 35
Gambar 5.3 Kode Implemetasi Pengiriman Pesan *RPC.* 35
Gambar 5.4 Kode Implementasi Penangkap Pesan *RPC.* 36
Gambar 5.5 Kode Implementasi Pesan *Bootstrapping.* 36
Gambar 5.6 Kode Implementasi Pesan *Join*. 37
Gambar 5.7 Kode Implementasi Pesan *Join Success.* 38
Gambar 5.8 Kode Implementasi Pesan Find *Node*. 38
Gambar 5.9 Kode Implementasi Pesan *Store Node* 39
Gambar 5.10 Kode Implementasi *Ping Bucket*. 40
Gambar 5.11 Kode Implementasi *Ping Call.* 40
Gambar 5.12 Kode Implementasi Pesan *Pong Bucket*. 41
Gambar 5.13 Implementasi Pesan *Pong Call.* 41

Gambar 5.14 Kode Implementasi *Call Request*..... 42
Gambar 5.15 Kode Implementasi Pesan *Call Accept*. 43
Gambar 5.16 Kode Implementasi Pesan *Call Reject.* 43
Gambar 5.17 Kode Implementasi Pesan *Call Finish*..... 44
Gambar 5.18 Kode Implementasi pada kelas *Triplet.* 45
Gambar 5.19 Kode Implemetasi *KBucket.* 45
Gambar 5.20 Kode Implementasi Pemetaan *Triplet* ke Dalam *KBucket*..... 47
Gambar 5.21 Kode Implementasi *KBucket Cache*..... 49
Gambar 5.22 Implementasi *join node* menggukan infomasi *KBucket cache*..... 50
Gambar 5.23 implementasi Aktivasi Metode *Broadcast*. 51
Gambar 5.24 Kode Implementasi Kelas *GetBroadcast.* 52
Gambar 5.25 kode Implementasi Kelas *ListeningBroadcast.*..... 53
Gambar 5.26 Kode Implementasi *Bootstrapping* secara *Manual.* 54
Gambar 5.27 Implementasi Penerimaan Pesan Pada *bootstrapping.*..... 55
Gambar 5.28 Kode Implementasi Kelas *Kademlia Key*. 56
Gambar 5.29 Kode Implementasi Fungsi Pencarian *Node*..... 58
Gambar 5.30 Kode Implementasi Fungsi Pemilihan *Node* Terdekat. 59
Gambar 5.31 Kode Implementasi Kelas Matrik. 60
Gambar 5.32 Kode Implementasi Penangkapan Pesan *Find Node.*..... 61
Gambar 5.33 Kode Implementasi Penangkapan Pesan *StoreNode.* 62
Gambar 5.34 Tampilan Pada Menu Panggilan..... 62
Gambar 5.35 Tampilam Panggilan Keluar dan Masuk Pada Aplikasi *VoIP.* 63
Gambar 5.36 Kode Implementasi Aturan Panggilan Suara..... 63
Gambar 5.37 Kode Implementasi *Node* Tidak DItemukan. 65
Gambar 5.38 Kode Implementasi *Node* Tidak Aktif..... 67
Gambar 5.39 Kode Implementasi Fungsi *Peer* Penelpon Bagian 1..... 68
Gambar 5.40 Kode Implementasi Fungsi *Peer* Penelpon Bagian 2..... 70
Gambar 5.41 Kode Implementasi Fungsi *Peer* Penerima Panggilan..... 72
Gambar 5.42 Kode Implementasi Fungsi *Call Accept*. 73
Gambar 5.43 Kode Implementasi Pemilihan *Codec.*..... 74
Gambar 5.44 Tampilan Menu *Settings* Untuk Memilih *Codec*. 74
Gambar 5.45 Kode Implementasi Kelas *Codec.* 76

Gambar 5.46 Kode Implementasi Penyeragaman *Codec*. 77
Gambar 5.47 Kode Implementasi Kelas *VoiceStart.* 78
Gambar 5.48 kode Implementasi Kelas *RTPAudiRecord.* 79
Gambar 5.49 Kode Implemetasi Kelas *Encoder*. 81
Gambar 5.50 Kode Implementasi Kelas *RTPAudioStream.* 83
Gambar 5.51 Kode Implementasi Kelas *Decoder.* 85
Gambar 6.1 Grafik Pengujian *Join Time.* 88
Gambar 6.2 Grafik Pengujian *Find Time.* 91
Gambar 6.3 Grafik Hasil Pengujian *Packet loss* 96
Gambar 6.4 Grafik Hasil Pengujian *Jitter*. 97
Gambar 6.5 Grafik Hasil Pengujian *Delay* 97
Gambar 6.6 Grafik Hasil Pengujian *Bandwidth* Terhadap *Packet loss* 98
Gambar 6.7 Pengujian *Bandwidth* Terhadap *Delay.* 99
Gambar 6.8 Pengujian Penggunaan Codec Terhadap Packet Loss 99
Gambar 6.9 Pengujian Penggunaan Codec Terhadap Jitter 100
Gambar 6.10 Pengujian Penggunaan Codec Terhadap Delay 100

BAB 1 PENDAHULUAN

1.1 Latar Belakang

Saat ini sistem *VoIP* (*Voice over Internet Protocol*) sebagian besar dibangun diatas arsitektur *client-server* (Lin at al., 2010). arsitektur *client-server* dipilih karena mudah untuk diterapkan. Dibalik keuntungan tersebut, arsitektur *client-server* memiliki beberapa kekurangan. Arsitektur *client-server* dianggap tidak memadai saat digunakan dalam komunikasi *VoIP* karena dapat meningkatkan beban penggunaan dan konsumsi *bandwidth* yang berlebih (Lin et al., 2010). Selain itu arsitektur ini juga dapat menimbulkan permasalahan *single failure*. Permasalahan ini terjadi ketika *server* tidak mampu melayani permintaan *client* yang melebihi kapasitasnya sehingga dapat mempengaruhi performa komunikasi data dalam jaringan (Chen at al., 2014; Wu at al., 2007).

Permasalahan pada arsitektur *client-server* dapat diatasi dengan menerapkan arsitektur *peer-to-peer overlay* sebagai penggantinya. *peer-to-peer* dipilih karena memiliki kelebihan skalabilitas dan utiilitas sumber daya yang lebih tinggi dibanding arsitektur *client-server* (Lin et al., 2010). Pada sistem komunikasi *VoIP* memerlukan skalabilitas jaringan yang tinggi. Hal ini dikarenakan pertumbuhan pengguna *VoIP* yang semakin meningkat sehingga sistem dituntut dapat mengikuti perkembangan jumlah pengguna (Rocha & Ruiz, 2008). Oleh karena itu *VoIP* akan menjadi lebih optimal jika dibangun diatas arsitektur *peer-to-peer*.

Terdapat empat topologi yang dikenal dalam arsitektur *peer-to-peer (P2P).* Topologi tersebut meliputi topologi jaringan terpusat, topologi jaringan terdistribusi penuh tidak terstruktur, topologi jaringan semi-terdistribusi dan topologi jaringan terdistribusi penuh tersetruktur. Dalam topologi jaringan terdistribusi penuh tidak terstruktur mekanisme pengenalan *node* menggunakan mekanisme *flooding.* Mekanisme ini melakukan pemetaan *node* secara acak. Dengan pola pemetaan seperti itu jaringan *peer-to-peer* menjadi kurang efisien dari segi performa.

Kekurangan yang terdapat pada topologi jaringan terdistribusi penuh tidak terstruktur kemudian dibenahi oleh topologi jaringan terdistribusi penuh terstruktur. Pada topologi ini menggunakan mekanisme pendistribusian tabel *hash* dalam melakukan pencarian *node.* Protokol yang digunakan dalam pencarian node ini disebut *DHT (Distributed Hash Table)*. *DHT* ini mampu mengatasi permasalahan dalam topologi jaringan terdistribusi penuh yang tidak terstruktur. Dengan menggunakan *DHT*, pengenalan *node* dapat dikoordinir oleh dirinya sendiri melalui mekanisme *join* dan *leave node*. Selain itu *DHT* juga menawarkan kelebihan seperti skalabilitas yang tinggi, keamanan yang lebih baik, *load balancing, robustness, reability* serta adanya toleransi akan kesalahan (Wang at al., 2010; Wu et al., 2007).

Terdapat banyak jenis *DHT* yang telah dikenal sampai saat ini, *DHT* yang sering digunakan meliputi *CAN, Chord, Pastri, Tapestry, Kademlia* dan *TomP2P*.

Dalam penelitian ini menggunakan *DHT* kademlia sebagai algoritma pengenalan *node*. Hal ini dilakukan karena *DHT Kademlia* merupakan salah satu *DHT* dengan perfoma terbaik dibandingkan *CAN, Chord, Pastry* dan *Tapestry* pada kasus *VoIP. DHT Kademlia* juga mampu melakukan *self-learning* dan *self-organizing* (Wu et al., 2007).

Dari hasil penelitian Rocha & Ruiz dengan judul "*A Study of the Effect of Using Kademlia as an Alternative to Centralized User Location Servers in SIP-based IP Telephony Systems*" mendapati bahwa *DHT Kademlia peer-to-peer* dapat menjadi solusi untuk mengatasi permasalahan pada arsitektur *client-server*. *DHT kademlia* juga dikatakan menjadi solusi yang efektif dalam membangun sistem pengenalan *IP* telepon pada komunikasi *VoIP*. Dalam penelitian tersebut juga menyimpulkan bahwa arsitektur peer-to-peer telah terbebas dari permasalahan *single failure* seperti yang terjadi pada arsitektur *client-server*.

1.2 Rumusan Masalah

Berdasarkan paparan latar belakang tersebut, maka rumusan masalah yang dapat dikaji dalam penelitian ini sebagai berikut:

1. Bagaimana melakukan pencarian *peer* sebelum melakukan komunikasi *VoIP*?
2. Bagaimana mekanisme koordinasi antar *peer* saat melakukan komunikasi *VoIP*?
3. Bagaimana performa DHT Kademlia dan kualitas komunikasi *VoIP* yang dihasilkan berdasarkan skalabitas, *query time* serta *QoS*?
4. Faktor-faktor apakah yang berpengaruh dalam komunikasi *VoIP*?

1.3 Tujuan

Adapun tujuan yang ingin dicapai dalam penelitian di skripsi ini yaitu :

1. Dapat merancang dan menerapkan *DHT* kademlia sebagai metode pengenalan *peer* pada sistem komunikasi *VoIP*.
2. Dapat membangun aplikasi *VoIP* diatas jaringan *peer-to-peer*.
3. Dapat mengetahui performa dari sistem yang dibangun.
4. Dapat mengetahui faktor-faktor yang berpengaruh dalam komunikasi *VoIP.*

1.4 Manfaat

Manfaat yang diperoleh dari penelitian yang sudah dilakukan yaitu :

1. Bagi Penulis
   - Mampu dalam merancang dan mengimplementasikan *VoIP* yang berjalan diatas arsitektur P2P dengan menggunakan *DHT Kademlia* sebagai algoritma pengenalan *node*nya.

- Memberikan pengetahuan dan pemahan tentang bagaimana komunikasi data pada *VoIP* P2P.
- Mampu melakukan perhitungan terhadap performa dari sistem yang di bangun dari segi kecepatan pengenalan dan pencarian *node*, *QoS*, penggunaan sumber daya serta skalabilitas sistemnya.

2. Bagi masyarakat
   - Mampu memberikan solusi terkait masalah *single failure* ketika menggunakan arsitektur *client-server*.
   - Dapat mengefisiensikan sumber daya perangkat yang dimiliki dengan menggunakan arsitektur *peer*-to-*peer*.
   - Dapat menyediakan komunikasi yang bersifat *realtime* dengan skalabilitas yang tinggi.
3. Bagi fakultas
   - Memberikan pengabdian untuk membantu institusi atau pemerintahan dalam bidang teknologi komunikasi yang terbarukan.

1.5 Batasan Masalah

Sesuai dengan terpaparnya permasalahan yang terdapat pada latar belakang agar tidak memperluas area bahasan maka diperlukan suatu batasan sebagai berikut :

1. Aplikasi *VoIP* dalam penelitian ini diimplementasikan pada OS Android, dengan spesifikasi minimal versi *SDK* 16 atau *Ice Cream Sandwich*.
2. Penulis memanfaatkan beberapa *library java* yang bersifat *open source* untuk memudahkan dalam membangun sistem komunikasi *VoIP peer-to-peer DHT Kademlia.*
3. Aplikasi yang diimplementasikan sementara hanya mencakup satu jaringan dengan subnet yang sama.

1.6 Sistematika Pembahasan

Penelitian ini diuraikan dengan sistematika penulisan yang terbagi menjadi 7 bab bahasan seperti berikut.

BAB I PENDAHULUAN
Dalam bab ini dijelaskan latar belakang serta tujuan dilakukannya penelitian tentang rancang bangun *VoIP* berbasis P2P menggunakan *DHT Kademlia* termasuk juga perumusan masalah, manfaat penelitian, batasan penelitian, sistematika pembahasan, dan jadwal penelitian.

BAB II LANDASAN KEPUSTAKAAN
Bab ini berisi tentang teori-teori pendukung serta penelitian-penelitian terdahulu yang dapat menguatkan dan menjadi dasar penelitian rancang bangun *VoIP* berbasis P2P menggunakan *DHT* kademlia ini.

BAB III METODOLOGI

Dalam bab ini dijelaskan metode-metode penelitian yang digunakan untuk perancangan *VoIP* berbasis P2P menggunakan *DHT Kademlia* dan langkah kerja yang dilakukan dalam penulisan tugas akhir.

BAB IV PERANCANGAN SISTEM

Membahas bagaimana merancang arsitektur umum sistem, gambaran umum sistem dan peran *DHT Kademlia* sebagai metode pengenalan dan pencarian *node* pada *VoIP* serta merancang cara pembuatan komunikasi *VoIP*.

BAB V IMPLEMENTASI

Membahas bagaimana cara penerapan *DHT Kademlia* dan sistem komunikasi *VoIP* sesuai dengan apa yang telah dirancang pada bab perancangan sistem. yang menjadi pokok bahasan dalam bab implementasi ini adalah penerapan bagian-bagian *DHT Kademlia* seperti pembuatan *peer*, penerapan KBucket, penerapan mekanisme pengenalan *node* dengan cara bootstrapping, mekanisme pencarian *node* dalam jaringan *peer-to-peer DHT Kademlia* dan juga menerapkan bagaimana pembangunan sistem komunikasi *VoIP* mulai dari cara komunikasi, penggunaan *codec* serta penerapan transmisi data agar antar *peer* dapat berbincang-bincang secara realtime.

BAB VI PENGUJIAN DAN ANALISIS

Pada bab ini membahas mengenai bagaimana performa dan skalabilitas dari sistem yang dibangun. Adapun yang menjadi parameter uji pada bab ini meliputi *Delay*, *Jitter*, packet lost, *query time* meliputi *join* time dan find time, skalabilitas sistem, penggunaan semberdaya memori dan *CPU*, penggunaan *codec* serta *bandwidth*.

BAB VII PENUTUP

Bab ini membahas tentang kesimpulan yang dapat diambil berdasarkan rumusan masalah yang telah ditetapkan sebelumnya. Serta memberikan saran terkait hal-hal yang perlu dikembangkan dalam penelitian ini. Agar penelitian ini dapat menjadi lebih baik lagi kedepannya.

BAB 2 LANDASAN KEPUSTAKAAN

Pada pembahasan ini terdapat kajian pustaka yang merupakan peninjauan kembali pustaka – pustaka pendukung yang memiliki keterkaitan dengan penelitian ini. Dasar teori juga dibutuhkan sebagai teori dasar yang harus dipahami agar pembaca dapat mengerti terkait permasalahan yang dibahas dalam penelitian.

2.1 Kajian Pustaka

Penelitian yang pertama yaitu dengan judul "*Bandwidth Constrained Tree Construction for Live Streaming Systems in P2P Networks*" menyebutkan dalam penelitiannya bahwa pada saat ini arsitektur *client-server* adalah arsitektur jaringan yang paling banyak digunakan di internet. Akan tetapi dibalik kemudahan dalam pengimplentasiannya banyak terdapat permasalahan seperti *single failure* dan juga komunikasi data yang kurang effisien serta menjelaskan juga mengenai penggunaan *bandwidth* yang berlebih merupakan permasalahan yang didapat ketika menggunakan arsitektur *client-server*. Didalam penelitian tersebut juga dikatakan bahwa teknologi *peer-to-peer* dapat mengatasi permasalahan yang terdapat di dalam arsitektur *client-server*. Dimana disana disebutkan pula arsitektur *peer-to-peer* memiliki beberapa kelebihan diantaranya skalabilitas dan *utilitas resource* yang tinggi (Lin et al., 2010).

Pada penelitian Singh (2014) yang berjudul "*VoIP: State of art for global connectivity—A critical review*" menyebutkan bahwa *VoIP* akan lebih efesien dibandingkan konikasi *PSTN* karena *VoIP* ini memiliki kehandalan seperti (1) *low cost* yang artinya memerlukan biaya yang sedikit dalam pengimplementasiannya karena berjalan pada jaringan sudah ada (internet). (2) *Integrated service* yang dimaksud disini ialah *VoIP* dapat terintegrasi terhadap telephone *PSTN* dengan menggunakan protokol *SIP* untuk komunikasinya. (3) Skalabilitas yang tinggi serta kemudahan dalam melakukan perubahan. (4) Keamanan, disini dimaksudkan paket data yang dikirimkan oleh *VoIP* dapat dienkripsi dengan metode VPN sehinga data yang dikirimkan bisa lebih aman.

Dalam penelitian yang berjudul "*An Improved Kademlia Protocol In a VoIP System*" menjelaskan arsitektur *peer-to-peer* sangat berbeda dengan arsitektur *client-server* dimana *peer-to-peer* terbebas dari permasalahan *single failure* yang kerap kali terjadi pada arsitektur *client-server*. Arsitektur *peer-to-peer* menawarkan kelebihan-kelebihan seperti skalabilitas, reliabilitas, toleransi kesalahan, *self-organizing* dan resistensi terhadap serangan DOS. Dalam penelitian ini juga dijelaskan bahwa *DHT Kademlia* merupakan salah satu algoritma *DHT* yang lebih baik jika dibandingkan dengan algoritma *DHT* lainnya seperti *CAN, Chord, Pastry* dan *Tapestry* pada kasus *VoIP* (Wu et al., 2007).

Dalam publikasinya yang berjudul "I*mproving Lookup Performance Based on Kademlia Chunzhi*" Wang (2010) mengatakan bahwa *peer-to-peer* terdiri dari beberapa jenis topologi seperti topologi jaringan terpusat, topologi jaringan

terdistribusi penuh tidak terstruktur, topologi jaringan semi-terdistribusi dan topologi jaringan terdistribusi penuh tersetruktur. Pada topologi terdistribusi tidak terstruktur menggunakan *flooding* untuk mengetahui *node* yang terdapat di skitarnya. Dan cara itu menurut Wang tidak efektif dan ada metode yang lebih baik untuk menggantikan metode *flooding* tersebut yaitu dengan cara memanfaatkan algoritma *DHT Kademlia*, dengan *DHT Kademlia* ini pengenalan *node* akan lebih terstruktur. Dengan cara mengirimkan pesan PING untuk mengetahui *node* yang aktif serta terdapat metode-metode lain seperti STORE, FIND VALUE, dan FIND *NODE* untuk membuat pencarian *node* lebih effisien lagi.

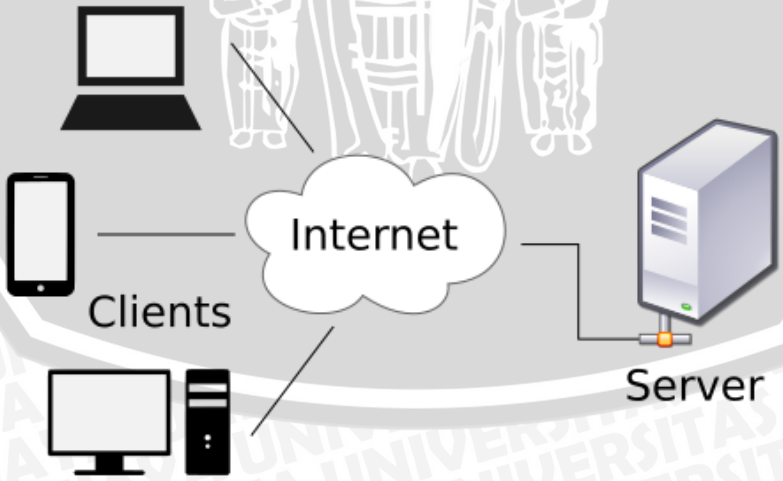
2.2 Landasan Teori

Berikut ini merupakan teori-teori yang digunakan dalam penelitian membangun *VoIP* menggunakan *DHT Kademlia*. Adapun penjelasan-penjelasan teori tersebut seperti berikut :

2.1.1 *VoIP (Voice over Internet Protocol)*

VoIP adalah suatu alternatif untuk menggantikan komunikasi telepon *PSTN*. Teknologi ini memanfaatkan jaringan internet sebagai transmisi datanya. Hal tersebut mengakibatkan *VoIP* memiliki beberapa kelebihan dibandingkan teknologi *PSTN* meliputi rendahnya biaya operasional, layanannya dapat terintegrasi dengan layanan *PSTN*, skalabilitas tinggi dan mudah dilakukan perubahan serta keamanan datanya dapat ditingkatkan dengan menggabungkannya dengan teknologi *VPN*. Kelebihan-kelebihan tersebut menjadikan *VoIP* sangat populer sehingga beberapa perusaan dibidang teknologi berlomba-lomba menawarkan produk layanan *VoIP* mereka (Singh et al., 2014).

2.1.2 Arsitektur *client-server*

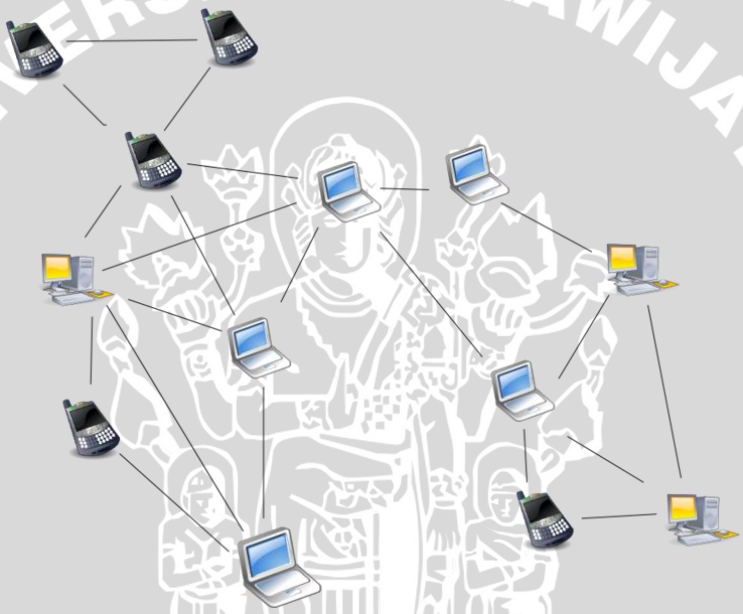


Gambar 2.1 Arsitektur *Client-Server*
[sumber: wikipedia]

Client adalah sebuah system atau proses yang meminta layanan pada *server*. Dimana *server* ialah system atau proses yang sedianya melayani semua permintaan dari *client*. Arsitektur *client-server* ialah sebuah arsitektur jaringan yang yang dimana terdapat komunikasi data, berupa permintaan dari *client* dan akan mendapatkan respon dari sebuah *server*. (Microsystem, 2009)

2.1.3 Arsitektur *peer-to-peer*

Peer-to-*peer* adalah arsitektur yang proses perhitungannya didistribusikan ke dalam masing-masing *peer*. Semua host yang tergabung dalam arsitektur *peer-to-peer* disebut sebagai *peer* atau *node*. Setiap *peer* dapat berbagi sumber daya mereka dalam pemrosesan dan berbagi penyimpanan kepada *peer* lain tanpa harus berkordinasi terhadap *server* pusat (Schollmeier, 2001).



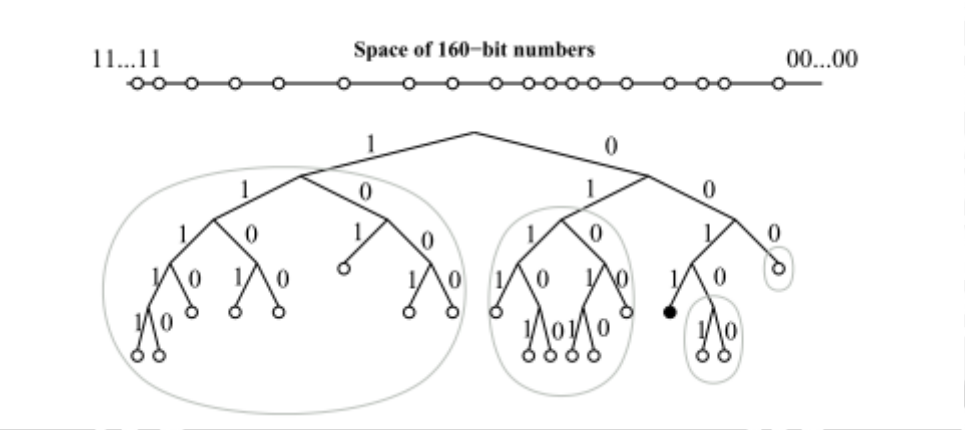
Gambar 2.2 Arsitektur *Peer-To-Peer Overlay*.
[Sumber : Wikipedia]

Peer dapat menjadi *server* maupun *client* dalam berbagi sumber daya, bukan seperti halnya pada arsitektur *client-server* host yang bertindak sebagai penyedia dan pemakai berada pada *host* yang berbeda. *Peer*-to-*peer* dapat membentuk jaringan komunitas berbagi dokumen antar *peer* yang tergabung di dalamnya. Setiap *peer* dapat berbagi dokumen yang mereka ingin bagikan terhadap *peer* lain, dan *peer* lain dapat mencari dokumen yang mereka butuhkan (Bandara & Jayasumana, 2013).

2.1.4 *DHT Kademlia*

Kademlia adalah protokol yang pertama kali diperkenalkan pada tahun 2002 dalam penelitian yang berjudul "*Kademlia*: A *peer-to-peer* information system based on the XOR metric" yang dipublikasikan oleh Petar P. Maymounkov dan David Mazieres di Amerika (Petar Maymonkov, 2002). Algoritma ini telah

digunakan pada aplikasi Bittorent pada tahun 2005. Dan setelah bittorrent terdapat beberapa aplikasi yang telah mengimplementasikannya seperti BitComet, BitSpirit dan eMule, dengan perbedaan metode dalam memperoleh *key, value* dan *node id*.



Gambar 2.3 Mekanisme Pencarian *Node* Pada *DHT Kademlia*
[sumber : Petar Maymonkov, 2002]

Pada kademlia, setiap *node* id merupakan sebuah angka unik terdiri dari 160 *byte string* yang mana terbentuk secara acak oleh system. Disaat yang sama *key* dan *value* dipasangkan dan dikirimkan ke *node* yang terdekat. Dasar dari *DHT* kademlia ialah proses pencariannya menggunakan *binary search tree*, yang mana *leave* berupa *node kademlia*. Semua *node* memiliki tanda pengenal berupa *node id*. Informasi dari *node* yang telah dikenal dimasukan dalam sebuah wadah yang bernama *KBucket. Node* yang tersimpan dalam *KBucket* menandakan *node* tersebut saling berdekatan hal itu diketahuai berdasarkan kode unik dari setiap *node id*-nya. (Guangmin, 2009)

2.1.5 *QoS* (Quality of Service)

Pada tahun 1994 *ITU-T* membuat sebuah rekomendasi E.800 terkait kualitas komunikasi telepon yang baik. Rekomenasi tersebut lebih dikenal dengan istilah *QoS* (Quality of service) . *QoS* ini membandingkan semua aspek yang dibutuhkan pada sebuah jaringan komunikasi seperti *delay, packet loss, jitter*, perbandingan gangguan sinyal, *crosstalk, echo* dsb (Menychtas, Kyriazis, & Tserpes, 2009). Yang menjadi fokus pada penelitian ini ialah *delay, jitter* dan *packet loss.* Untuk lebih jelasnya akan dijelaskan pada sub bagian berikut.

2.1.5.1 *Packet loss*

Packet loss terjadi ketika satu atau lebih paket yang dikirimkan melalui jaringan komputer tidak dapat diterima oleh komputer tujuan. *packet loss* ini akan diukur menggunakan perbandingan prosentase paket yang gagal dengan jumlah paket yang dikirimkan. *Packet loss* disebabkan karena terjadinya tabrakan paket pada jaringan yang padat ataupun karena faktor lain seperti paket yang datang pada router melebihi kemampuan penyangga dari router tersebut sehingga paket akan langsung dihapus (Kurose & Ross, 2013).

2.1.5.2 Jitter

Dalam kontek jaringan komputer, *jitter* adalah variasi keterlambatan paket yang diukur dalam variabel waktu saat paket dikirimkan melalui jaringan. Pada jaringan dengan *delay* paket yang tetap maka nilai dari *jitter* akan nol karena tidak adanya variasi keterlambatan paket (Comer, 2009).

Jitter dapat terjadi karena beberapa faktor seperti beban trafik dan juga besarnya potensi tabrakan paket pada jaringan. beban trafik dan potensi tabrakan yang besar akan meningkatkan waktu *jitter*. Tingginya nilai *jitter* akan menurunkan nilai dari *QoS* sehingga juga mempengaruhi kualitas komunikasi *VoIP* (Suryawan at al., 2012).

2.1.5.3 *Delay*

Delay adalah waktu tunda yang diperlukan dalam pengiriman paket dari *host* pengirim ke *host* tujuan. pada rekomendasi *ITU-T* agar komunikasi *VoIP* dapat dikategorikan baik, *delay* harus berada dibawah 150 ms (Saiful & Nrp, 2006).

2.1.6 Skalabilitas

Skalabilitas adalah kemampuan mengatasi pertumbuhan yang dinamis dari sebuah sistem, jaringan atau proses (Bondi, 2005). Sebagai contohnya pada penelitian ini sistem *VoIP* diimplementasikan pada perangkat bergerak. Maka kemungkinan besar jumlah *node* yang tergabung akan berubah-ubah sehingga system dituntut harus mampu mengatasi perubahan tersebut tanpa harus mempengaruhi kinerja dari *VoIP* itu sendiri. Terdapat beberapa tipe skalabilitas (El-Rewini & Abd-El-Barr, 2005) seperti berikut ini :

2.2.1.1 Skalabilitas Adminitratif

Skalabilitas administrative adalah kemampuan untuk mengatasi meningkatnya pengguna dalam berbagi sumber daya pada sistem terdistribusi.

2.2.1.2 Skalabilitas Fungsional

Skalabiltas fungsional adalah kemampuan untuk meningkatkan fungsionalitas sistem tanpa harus melakukan perubahan yang berarti pada sistem jaringan tersdistribusi.

2.2.1.3 Skalabilitas Geografi

Skalabilitas geografi adalah kemampuan menyesuaikan dan membenahi performa, kenyamanan penggunaan pengguna dan usabilitinya ketika terjadi perluasan wilayah dari sistem terdistribusi tersebut.

2.2.1.4 Skalabilitas beban

Kemampuan sebuah sistem terdistribusi secara mudah menyesuaikan perubahan ketika terjadi penambahan beban *node* dan pengurangan beban *node*.

2.2.1.5 Skalabilitas Generasi

Skalabilitas yang tertuju pada kemampuan sebuah sistem untuk memperluas skala terhadap komponen yang baru. Contohnya seperti kasus skalabilitas *heterogen* dimana jaringan dapat menerima komponen jaringan dari berbagai vendor.

2.1.7 *Codec*

Codec merupakan cara standar yang digunakan untuk melakukan komunikasi antar perangkat. *Codec* ini dapat mempengaruhi keefetifan dalam melakukuan komunikasi digital. Didalam *codec* didefinisikan acuan terhadap bitstream yang digunakan, serta algoritma *encode*r dan *decoder* dalam berkomunikasi. Algoritma pemampatan akan dimplementasikan pada proses *decoder* dan *encode*r sehingga data yang dihasilkan menjadi lebih kecil tanpa mengilangkan informasi data multimedia (Turaga & Chen, n.d.).

2.1.8 Protokol

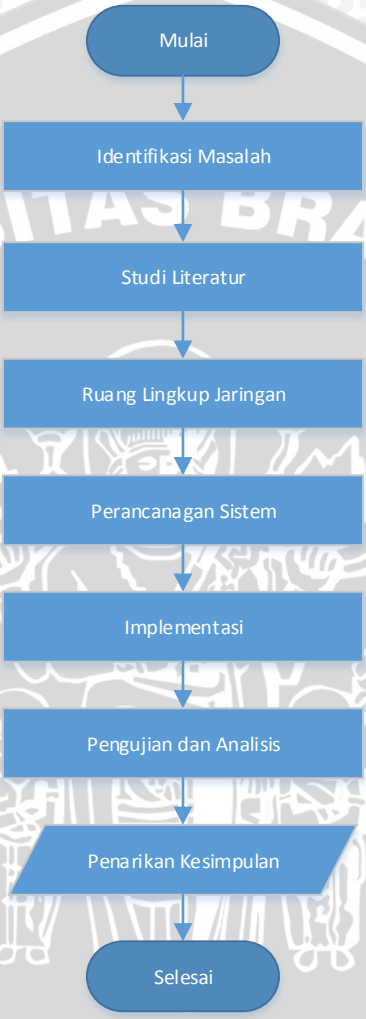
Protokol didefinisikan sebagai himpunan aturan yang mengatur bagaimana data dapat ditransmisikan pada jaringan komputer (Kurose & Ross, 2013). Dalam penelitian ini menggunakan beberapa jenis protokol untuk menjalin komunikasi antar perangkat. Protokol-protokol yang digunakan seperti protokol *IP, UDP, RTP, SIP.* Setiap kali terhubung dalam jaringan akan mendapatkan *IP* sebagai pengenal sebuah *host* di dalam jaringan. *Ip* didapat melalui proses *dhcp* maupun diatur secara *manual* menggunakan *IP static*. Protokol *UDP* merupakan protokol yang terdapat pada layer *transport* dalam struktur *OSI Layer*. Protokol *UDP* digunakan ketika kebutuhan akan informasi data yang relevan dan mentoleransi adanya kegagalan paket (Kurose & Ross, 2013). Protokol *UDP* yang dibalut oleh header yang berupa infomasi terkait data nomor *sequence* packet tipe *codec* dalam komunikasi *VoIP* akan menjadi paket *RTP*. Protokol *SIP* kerap kali digunakan dalam komunikasi *VoIP*. Protokol *SIP* merupakan protokol yang membawa sinyal ketika terjadi komunikasi *VoIP.* Protokol *SIP* ini memiliki alamat *SIP* tersendiri yaitu terdiri dari *username@hostname:port*.

2.1.9 *Bandwidth*

Bandwidth merupakan suatu perhitungan terhadap konsumsi data yang digunakan dalam komunikasi jaringan. *Bandwidth* dihitung dengan menggunakan satuan *bits* per detik (Chou, 2006). Semakin besar data yang dikirimkan maka *bandwidth* yang dibutuhkan semakin besar. Gangguan komunikasi akan terjadi ketika *bandwidth* yang ditawarkan pada jaringan lebih kecil disbandingkan *bandwidth* yang diperlukan dalam komunikasi.

BAB 3 METODOLOGI

Metodologi penelitian berisi tentang tata cara yang digunakan untuk melakukan penelitian atau dapat juga diartikan tindakan yang digunakan untuk teknik pemilihan rancangan di dalam penelitian ini. Adapun tahapan penelitian yang ditunjukkan pada Gambar 3.1 sebagai berikut :



Gambar 3.1 Diagram Metodologi Penelitian
Sumber: [Perancangan]

3.1 Identifikasi Masalah

Masalah yang diteliti pada penelitian ini adalah bagaimana mengatasi kekurangan *single failure* yang dimiliki oleh arsitektur *client-server* pada kasus *VoIP.* Dari permasalahan tersebut kemudian diidentifikasi cara-cara yang dapat digunakan untuk mengatasi permasalahan tersebut. dari identifikasi permasalahan ini kemudian dijabarkan solusi-solusi yang memungkinkan untuk diambil dan diterapkan. Sehingga akhirnya penelitian ini memililih arsitektur *peer-*

to-*peer* sebagai solusi penanganan. Arsitektur peer-to-peer dipilih karena skalabilitasnya yang tinggi dan sifatnya yang tidak mengenal adanya *single failure.*

Setelah menentukan permasalahan yang diteliti kemudian dilanjutkan dengan mencari tujuan penelitian. Adapun tujuan yang diharapkan setelah melakukan penelitian ini adalah arsitektur *peer*-to-*peer* mampu membuat komunikasi *VoIP* menjadi lebih efisien dengan menggunakan metode pengenalan *node DHT Kademlia*. Efisiensi yang diukur dengan meliputi kecepatan proses pencarian *node*, kualitas suara yang didapatkan serta skalabilitas yang ditawarkan.

3.2 Studi Literatur

Untuk dapat mendalami permasalahan yang diteliti maka peneliti memerlukan informasi-informasi dan solusi terkait permasalahan tersebut. Informasi-informasi tersebut diperoleh melalui buku, *journal,* situs internet, penjelasan dari dosen dan juga rekan-rekan mahasiswa. Adapun hal-hal yang dipelajari dalam studi literature ini meliputi:

1. Membaca dan memahami teori-teori terkait komunikasi *VoIP*.
2. Mendalami arsitektur *client-server* dan melakukan pengamatan terkait pengaruh arsitektur ini dalam komunikasi *VoIP*.
3. Mendalami arsitektur *peer-to-peer* beserta algoritma *DHT* yang sering digunakan dalam arsitektur tersebut.
4. Memdalami *DHT Kademlia* dan mencari tahu terkait proses-proses yang terjadi didalamnya dan bagaimana cara penerapannya.
5. Membaca literatur-literatur terkait faktor-faktor yang mempengaruhi komunikasi *VoIP.*

3.3 Ruang Lingkup Jaringan

Peneliti membuat sebuah jaringan lokal yang terdiri dari sebuah *access point* dan beberapa perangkat *android* sebagai sarana untuk menjalankan aplikasi yang telah dibangun. Pada penelitian ini menggunakan arsitektur *peer-to-peer* sehingga tidak memerlukan pengadaan *server* didalamnya.

3.4 Perancangan Sistem

Pada tahap perancangan sistem peneliti melakukan beberapa perancangan meliputi peliputi perancangan *DHT Kademlia* dan sistem VoIP yang dibangun. Dalam perancangan *DHT Kademlia* dibagi menjadi beberapa bagiann perancangan seperti perancangan pesan *RPC,* prancangan *KBucket* dan *Cache*, perancangan mekanisme *bootstrapping* dan pencarian *node*. Sedangkan pada perancangan *VoIP* yang dirancang meliputi panggilan suara, pemilihan dan penggunaan *codec* serta transmisi data yang dilakukan.

3.5 Implementasi

Implementasi merupakan tindakan lanjutan atau penerapan dari perancangan yang sudah disusun sebelumnya. Dalam penelitian ini yang

diimplementasikan berupa fase-fase yang terjadi pada *DHT Kademlia* serta komunikasi *VoIP*. Pengimplementasiannya dilakukan pada perangkat *Android*.

3.6 Pengujian dan Analisis

Dalam tahap Pengujian dilakukan beberapa pengukuran performa dari sistem yang telah diimplentasikan. Adapun yang menjadi alat ukur dalam proses uji coba ini meliputi *QoS*, skalabilitas, penggunaan *resource* dan *Query time* atau kecepatan proses *join* dan pencarian *node* pada *DHT Kademlia*. Untuk prosedur masing-masing pengujian sebagai berikut.

3.6.1 Pengujian *Query Time*

3.6.1.1 Tujuan Pengujian

Pengujian *query time* merupakan pengujian waktu yang diperlukan oleh sebuah *peer* untuk dapat berkomunikasi dengan *peer* lainnya. Komunikasi tersebut dapat berupa permintaan *join* ataukah permintaan pencarian *node*. Dengan dilakukannya pengujian *query time* ini diharapkan dapat mengukur performa dari *DHT Kademlia* dari segi efisiensi waktu terhadap jumlah *peer* yang telah tergabung pada jaringan tersebut. Sehingga pada akhirnya dapat ditarik kesimpulan mengenai keterkaitan antara performa yang dihasilkan *DHT Kademlia* terhadap jumlah *peer* yang bergabung dalam jaringan.

3.6.1.2 Prosedur Pengujian

Pengujian *query time* memiliki 4 skenario pengujian. Setiap skenario pengujian dibedakan dengan jumlah *node* yang tergabung didalam jaringan tersebut. Untuk skenario 1 menggunakan 5 *node* dan untuk selanjutnya akan bertambah menjadi 10, 15 dan 20 *node*.

Dalam pengujian *query time* terdapat dua hal yang diuji yaitu *join time* dan *find time*. Dalam pengujiannya kedua hal tersebut menggunakan skenario yang serupa. Dari sejumlah *peer* yang tergabung dalam jaringan *DHT* dipilih sebuah *peer* secara bergiliran.

Untuk pengujian *join* time, *peer* yang telah dipilih kemudian melakukan permintaan *join* terhadap *peer* lain hingga semua *peer* saling terhubung satu dengan yang lainnya. Ketika melakukan permintaan *join* terdapat jeda waktu yang dibutuhkan dari proses pengiriman permintaan *join* hingga permintaan tersebut disetujui. Waktu inilah yang diteliti dalam pengujian ini. Hal ini dilakukan untuk mengetahui seberapa cepat proses *join time* terjadi. Dari masing-masing *peer* dapat diukur kecepatan proses *join timenya*. Kemudian ukuran tersebut dirata-ratakan untuk dijadikan nilai dari setiap skenarionya.

Pada pengujian *find time* diasumsikan bahwa *peer* yang berada dalam jaringan *DHT Kademlia* telah saling terhubung baik secara langsung maupun terhubung atas perantara *peer* lain. *Peer* dipilih secara bergilir dan melakukan pencarian *node* secara acak. Waktu yang diperlukan untuk melakukan pencarian *node* dijadikan sebagai tolak ukur dan dirata-ratakan untuk setiap skenarionya.

3.6.2 Pengujian Skalabilitas Sistem

3.6.2.1 Tujuan Pengujian

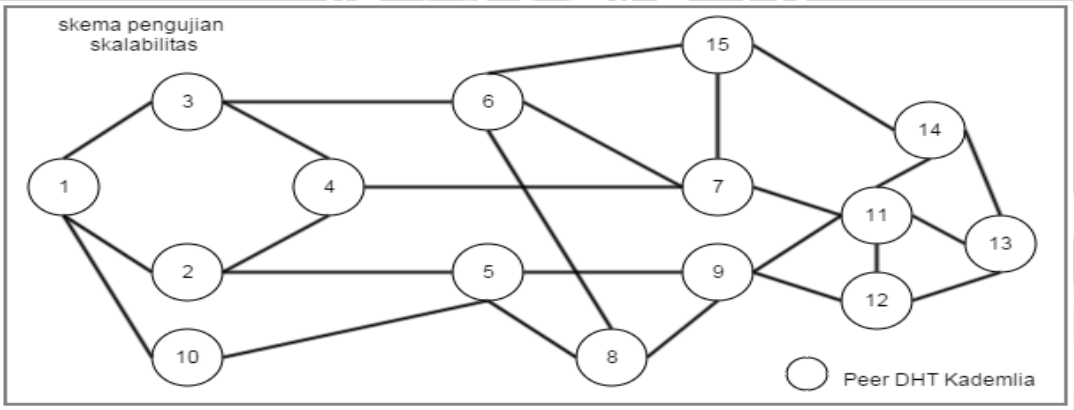
Dalam pengujian skalabilitas yang menjadi fokus uji disini adalah kemampuan sistem untuk menangani masalah perubahan skala jaringan pada mekanisme keluar dan masuknya *peer* dalam jaringan *DHT Kademlia*. Dari hasil pengujian ini diharapkan dapat ditarik kesimpulan terkait tingkat skalabilitas dari jaringan *DHT Kademlia* tersebut.

3.6.2.2 Prosedur Pengujian

Dalam pengujian ini terdapat dua skenario yang dilakukan. Skenario pertama dilakukan uji coba terkait pertumbuhan jaringan yang disebabkan oleh bertambahnya *node* (*join node)* dalam jaringan *DHT* kademlia. Skenario kedua yang diukur adalah kemampuan sistem jika terjadi penurunan jumlah *node (leave node)* pada jaringan *DHT Kademlia*.

Tabel 3.1 Tanda Pengenal *Node* Pada Topologi Pengujian Skalabilitas.
Sumber: [Perancangan]

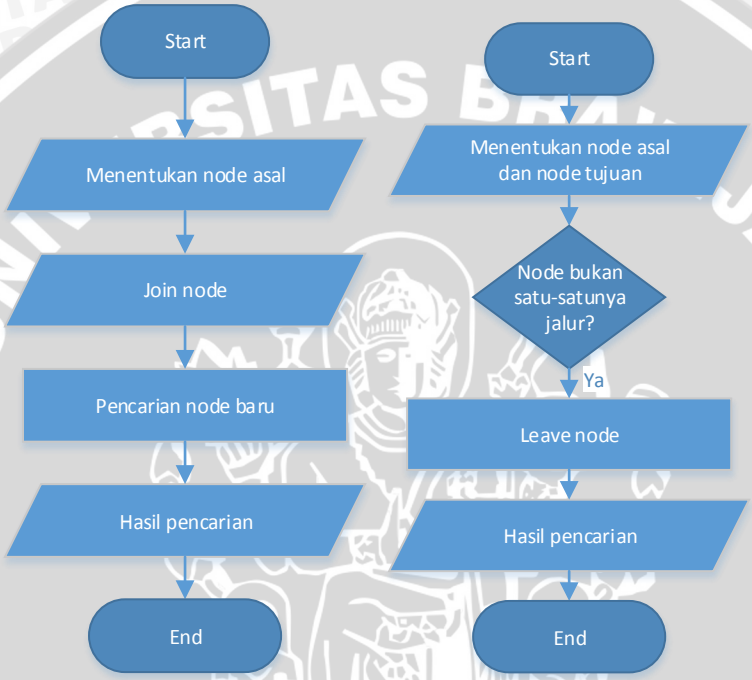
Node	Nama *Node*	*Node*	Nama *Node*	*Node*	Nama *Node*
1	085338584841	6	085338584846	11	085338584851
2	085338584842	7	085338584847	12	085338584852
3	085338584843	8	085338584848	13	085338584853
4	085338584844	9	085338584849	14	085338584854
5	085338584845	10	085338584850	15	085338584855



Gambar 3.2 Topologi Pada Pengujian Skalabilitas
Sumber: [Pengujian]

Pada skenario pertama dilakukan Langkah-langkah sebagai berikut. Pertama-tama ditetap sebuh *node* sebagai *node* yang akan melakukan interaksi dengan *node* yang lainnya. Setelah itu dilakukan penambahkan *node* satu persatu kedalam jaringan *DHT Kademlia*. *Node* yang baru ditambahkan kemudian dicoba untuk ditemukan oleh *node* yang telah ditetapkan sebelumnya. Dari percobaan ini menghasilkan prosentase keberhasilan *node* baru untuk ditemukan. Untuk skenario yang kedua menyerupai skenario pertama. Yang membedakan skenario

kedua ini adalah jumlah *node* akan mengalami penurunan kuantitasnya. Pada skenario ini akan menjadikan sebuah *node* yang dipilih tidak aktif. Dalam pentidak aktifan *node* menggunakan persyaratan bahwa *node* yang dihapus adalah *node* yang bukan sebagai jalur tunggal. Sehingga ketika *node* ditiadakan jaringan *DHT Kademlia* masih dapat tetap terhubung. Dalam pengujian skalabilitas ini menggunakan topologi jaringan seperti pada **gambar 3.2**. Dalam topologi tersebut masing-masing lingkaran mewakili sebuah *node*. Dan setiap garis yang ada mewakili jaringan komunikasi antar *node*. Pada **tabel 3.1** ini menerangkan secara lengkap informasi dari masing-masing *node* pada gambar dengan nama sebernarnya dalam pengujian.



Gambar 3.3 Diagram Alir Pengujian Skalabilitas
Sumber: [Pengujian]

3.6.3 Pengujian *QoS (Quality of Service)*

3.6.3.1 Tujuan Pengujian

Pengujian *QoS* adalah pengujian terkait kualitas dari jaringan *VoIP Kademlia*. Dalam pengujian *QoS* terdapat 3 parameter yang diuji yaitu *packet loss*, *jitter* dan *delay*. Tujuan dari pengujian ini untuk mengukur bagaimana kualitas dari komunikasi *VoIP peer-to-peer* menggunakan *DHT Kademlia ini*. Dari pengukuran ketiga parameter tersebut. Dari hasil pengujian ini kemudian diketahui kualitas sistem *VoIP* yang dibangun termasuk dalam kategori baik atau buruk.

3.6.3.2 Prosedur Pengujian

Pada pengujian *QoS* ini menggunakan perantara *access point TP-Link TD-W8151N* dengan *bandwidth* yang didukung 150 *Mbps*. Terdapat 8 *node* yang

tergabung dalam jaringan *peer-to-peer DHT Kademlia*. Dimana masing-masing *node* adalah *smarth phone android* yang telah terpasang aplikasi *VoIP Kademlia.*

Terdapat 4 skenario yang digunakan yaitu skenario yang terkait dari jumlah pasangan *peer* yang berkomunikasi pada saat yang sama. Pada skenario 1 terdapat sepasang *peer* yang melakukan komunikasi *VoIP* dan selanjutnya 2 pasang, 3 pasang hingga 4 pasang *peer*.

Pengujian ini menggunakan bantuan aplikasi *wireshark* dalam melakukan pengamatan terkait paket yang keluar dan masuk melalui jaringan. Parameter yang diamati melalui aplikasi *wireshark* adalah *packet loss, jitter* dan *delay*. Dari hasil tersebut kemudian dibandingakan dengan kualitas yang direkomendasikan oleh *ITU-T*.

3.6.4 Pengujian Penggunaan *Bandwidth*

3.6.4.1 Tujuan Pengujian

Pengujian *Bandwidth* adalah pengujian terkait kualitas dari jaringan *VoIP Kademlia* yang diukur perdasarkan skenario perbedaan *bandwidth* yang diberikan. Sama halnya dengan pengujian *QoS* terdapat 3 parameter yang dapat diuji yaitu *packet loss*, *jitter* dan *delay.* Tujuan dari pengujian ini untuk dapat mengetahui seberapa besar pengaruh *bandwidth* akan kualitas komunikasi *VoIP* yang dihasilkan.

3.6.4.2 Prosedur Pengujian

Pada pengujian *bandwidth* ini menggunakan *router mikrotik* bertipe RB750. Terdapat 5 jenis *bandwidth* yang di uji. Setiap pengujiannya dipilih *codec* dengan tipe g711a. terdapat 4 kali percobaan yaotu menggunakan *bandwidth* 128 *Kbps*, 256 *Kbps*, 384 *Kbps*, dan 512 *Kbps*

Pengujian ini menggunakan bantuan aplikasi *wireshark* dalam melakukan pengamatan terkait paket yang keluar dan masuk melalui jaringan. Parameter yang diamati melalui *wireshark* adalah *packet loss, jitter* dan *delay*. Dari hasil terbut akan dibandingakan dengan kualitas yang di rekomendasikan oleh *ITU-T*.

3.6.5 Pengujian Penggunaan *Codec*

3.6.5.1 Tujuan Pengujian

Dalam pengujian *codec* terdapat 4 *codec* yang di uji. Codec-*codec* tersebut meliputi *speex, g711u, g711a,* dan *g722.* Tujuan dilakukannya pengujian ini adalah untuk mengetahui apakah perbedaan codec yang digunakan akan mempengaruhi kualitas komunikasi VoIP tersebut.

3.6.5.2 Prosedur Pengujian

Codec diuji dengan bantuan aplikasi *wireshark*. Sebelum memulai komunikasi VoIP aplikasi wireshark harus dijalankan sebelumnya. setelah itu kedua peer melakukan komunikasi VoIP dengan menggunakan 4 jenis codec yang disediakan secara bergantian. Pada pengujian ini menghasilkan nilai *packet loss,*

jitter dan *delay*. Kemudian parameter-parameter tersebut dianalisis untuk mengetahui seberapa besar pengaruh penggunaan codec terhadap kualitas komunikasi *VoIP* yang dilakukan.

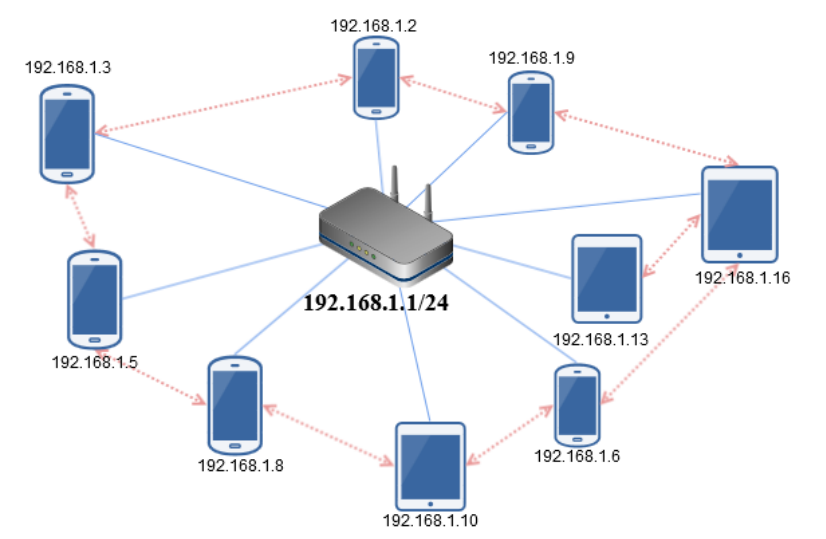
3.7 Penarikan Kesimpulan.

Tahap ini dilakukan ketika sistem selesai dibangun dan tahap pengujian telah dilakukan. Pada tahapan ini melakukan penarikan kesimpulan terkait rumusan masalah yang telah terpecahkan dalam penelitian ini.

BAB 4 PERANCANGAN SISTEM

Pada bab 4 ini membahas terkait arsitektur sistem, gambaran umum sistem serta perancangan sistem yang dibangun. Perancangan sistem tersebut meliputi perancangan *peer*, perancangan *Kademlia* dan perancangan *VoIP*.

4.1 Arsitektur Umum Sistem



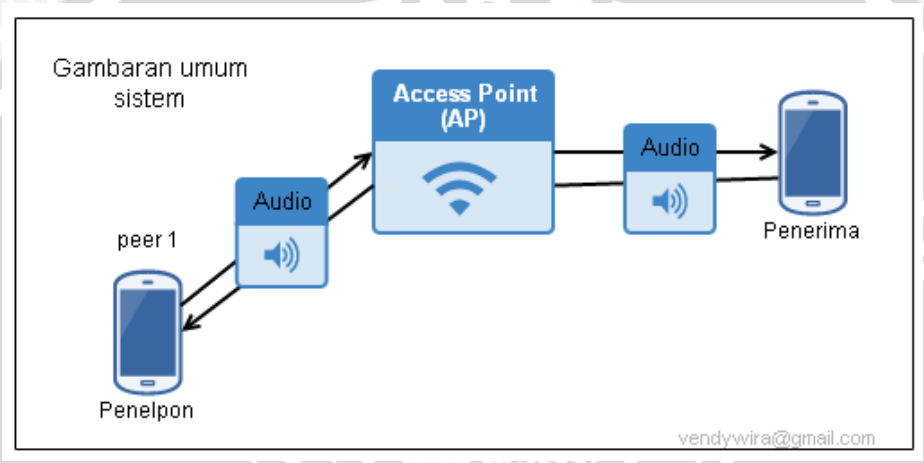
Gambar 4.1 Gambaran Umum Arsitektur Sistem
Sumber: [Perancangan]

Arsitektur yang digunakan pada penelitian ini ialah *peer-to-peer*. Arsitektur ini tidak memerlukan *server* untuk menjembatani komunikasi antar *peer* di dalam sistem. *Acces point (AP)* digunakan sebagai media penghubung komunikasi antar *peer.* Pada **gambar 4.1** dapat dilihat bahwa masing-masing perangkat telah saling mengenal satu dengan yang lainnya. hal tersebut ditunjukan oleh garis panah merah yang menghubungkan satu perangkat dengan perangkat lainnya. Perangkat tersebut telah saling mengenal baik secara langsung maupun melalui perantara perangkat lain. *Access point* pada gambar tersebut dianalogikan sebagai satu jaringan yang sama dengan *network id* 192.168.1.1 dan *subnet mask* 255.255.255.0 dan garis panah biru menggambarkan bahwa seluruh perangkat telah tergabung dalam jaringan tersebut.

Masing-masing perangkat yang terhubung dalam jaringan AP dapat disebut sebagai *peer* atau *node*. Setiap *peer* yang terhubung kedalam jaringan dapat melakukan panggilan suara terhadap *peer* lainnya. Setiap *peer* diidentifikasi menggunakan nomor telepon dari masing-masing perangkat tersebut. Nomor telepon kemudian dikonversikan menjadi bilangan biner berupa kombinasi angka 0 dan 1. Bilangan binari itulah menjadi identitas masing-masing perangkat yang disebut sebagai *node Id*. Selain *node id* masing-masing perangkat juga memiliki parameter berupa alamat *ip* dan *udp port.* Dimana informasi dari alamat *ip* dan *udp port* digunakan sebagai alamat pertukaran data antar *peer*.

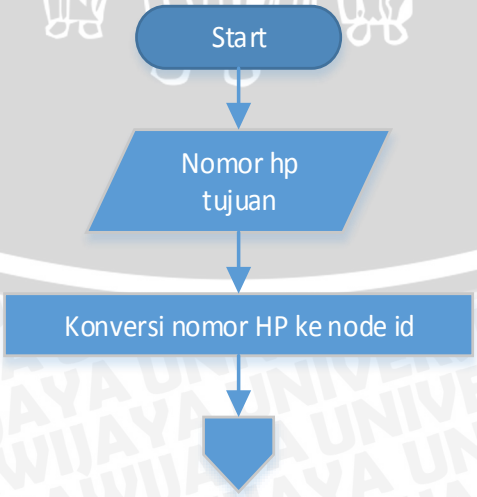
Dalam penelitian ini menggunakan algoritma *Distributed Hash Table (DHT) Kademlia* sebagai mekanisme pencarian dan pengenalan *node* tetangganya. Dengan menggunakan mekanisme ini *peer* dapat menemukan *peer* lain yang tidak dikenalnya secara langsung. Pada umumnya *DHT Kademlia* melakukan pencarian *node* dengan menggunakan metode pecarian *binary tree*. Dalam proses pencariannya sebuah *peer* akan memilih dua *node* yang dikenalnya berdasarkan perhitungan jarak kedekatan menggunakan metode *XOR distance*. Dimana kedua *peer* tersebut kemudian diminta untuk melakukan pencarian terhadap *node* yang telah dikenalnya. Ketika *peer* tujuan ditemukan, peer penemu akan mengirimkan pesan kepada peer pencari terkait informasi dari *peer* tujuan.

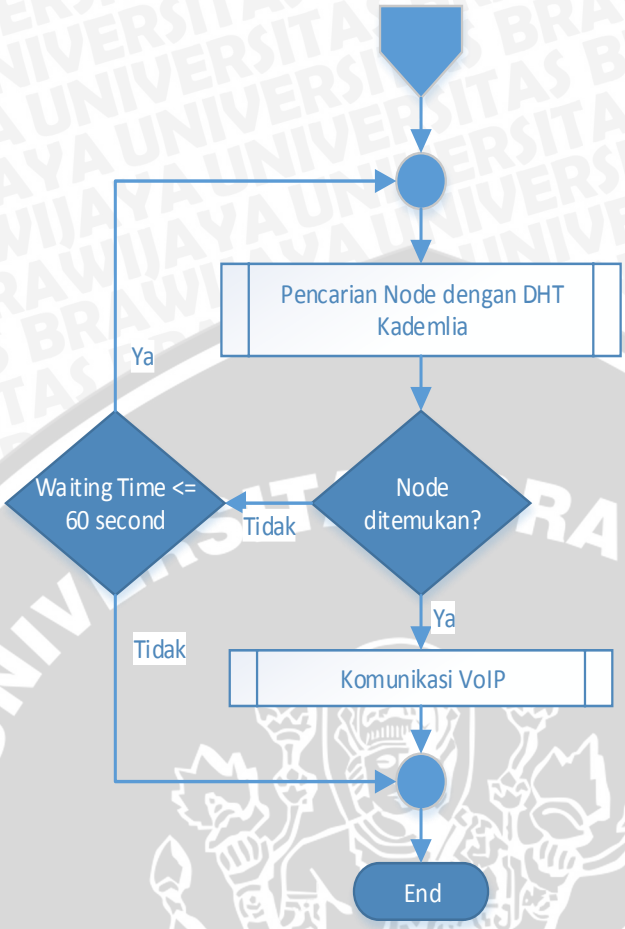
4.2 Gambaran Umum Sistem



Gambar 4.2 Gambaran Umum Sistem
Sumber: [Perancangan]

Gambaran umum sistem ini menjelaskan secara umum bagaimana sistem *VoIP Kademlia* bekerja. Dalam Sistem ini terdapat tiga komponen utama yaitu *Access Point (AP)* dan dua *peer* yang masing-masing memiliki peranan sebagai penelepon dan penerima telepon.



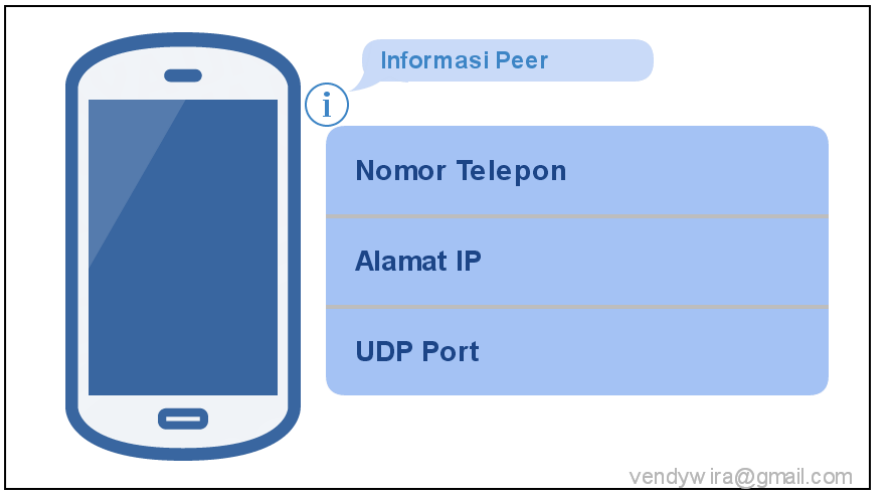


Gambar 4.3 Diagram Alir Kerja Sistem.
Sumber: [Perancangan]

Sesuai dengan diagaram alir pada **gambar 4.3** dapat dilihat alur proses yang dilakukan peer sesaat sebelu melakukan panggilan suara hingga mengkahiri panggilan tersebut. Untuk melakukan panggilan suara *peer* penelepon harus memasukan sederetan nomor telpon yang digunakan sebagai identitas dari *peer* tujuan. Deretan nomor telepon tersebut kemudian diproses oleh sistem dan dijadikan *node id*. Kemudian *node id* tersebut dijadikan patokan pencarian node tujuan. Saat *node* tujuan ditemukan peer penemu akan mengirimkan informasi berupa alamat ip dan udp port node tujuan. informasi tersebutlah yang menjadi alamat pengiriman pesan panggilan suara oleh peer pencari.

Setelah *peer* penelepon mengetahui alamat calon penerima, *peer* penelepon kemudian mengirimkan pesan panggilan kepada *peer* tujuan. Saat *peer* tujuan menerima pesan tersebut, kemudian *peer* tujuan berdering sebagai tanda terdapat permintaan panggilan masuk. Panggilan ini dapat diterima maupun ditolaknya. Jika panggilan diterima maka proses dilanjutkan dengan komunikasi suara antar kedua *peer* tersebut. Dan jika panggilan tersebut ditolak maka *peer* penerima telepon kemudian mengirimkan pesan penolakan sebagai sinyal untuk mengakhiri panggilan.

4.3 Perancangan *Peer*



Gambar 4.4 Perancangan *Peer*
Sumber: [Perancangan]

Seperti yang telah dibahas sebelumnya *kademlia* adalah mekanisme yang digunakan dalam pengenalan dan pencarian *node-node* yang terhubung dalam jaringan *peer-to-peer*. *Peer* merupakan hal yang paling mendasar sebagai *host* penyusun sistem *peer-to-peer*. Dalam perancangan *peer* ini diterangkan informasi yang diperlukan dan harus dimiliki oleh sebuah *peer*.

Informasi yang tersimpan pada masing-masing *peer* digunakan sebagai tanda pengenal dari *peer* tersebut. Adapun Informasi-informasi tersebut ialah nomor telepon, alamat ip serta udp port. Yang mana nomor telepon digunakan sebagai nama dari *peer* tersebut. Hal ini dimaksudkan agar masing-masing *peer* dapat memiliki nama yang unik sehingga tidak terdapat kesamaan identitas dalam jaringan *peer-to-peer*.

4.4 Perancangan Kademlia

Dalam perancangan ini membahas bagaimanan *DHT Kademlia* dapat digunakan sebagai metode pengenalan *peer* dalam sistem *VoIP peer-to-peer*. Adapun yang menjadi pokok perancangan meliputi perancangan pertukaran pesan *RPC (remote procedure call), KBucket,* mekanisme *bootstrapping* dan mekanisme pencarian *node*. Untuk lebih jelasnya dapat dilihat dalam sub bagian berikut.

4.4.1 Perancangan Pesan *RPC* (*Remote Procedure Call*)

RPC (remote procedure call) adalah mekanisme yang digunakan dalam *DHT Kademlia* untuk dapat berinteraksi dengan *peer* lain yang tergabung dalam jaringan *peer-to-peer*. Pesan *RPC* dapat membangkitkan aksi pada *peer* penerimanya sesuai dengan jenis pesan *RPC* yang diterimanya.

Dalam penelitian ini proses pengiriman dan penerimaan pesan *RPC* menggunakan bantuan pustaka yang telah ada yaitu *sip2peer*. Pustaka ini bersifat *open source* sehingga dapat disesuaikan dengan keperluan penelitian tanpa melanggar hak cipta pendirinya. Pustaka ini dirancang untuk menangani proses komunikasi antar *peer* dalam jaringan *peer-to-peer*. Komunikasi antar *peer* dilakukan dengan mekanisme pertukaran pesan menggunakan *format json*. Karena pustaka ini dirancang berdasarkan protokol *SIP* maka alamat yang dikirimkan menggunakan format penulisan alamat *SIP*. Yang mana format alamat tersebut diawali dengan identitas nomor telepon kemudian diikuti oleh alamat ip beserta udp portnya. Dengan memanfaatkan pustaka ini sebuah *peer* dapat mengetahui pesan *RPC* yang dikirimkan berhasil maupun mengalami kegagalan.

Pada penelitian ini terdapat beberapa pesan yang digunakan untuk berkomunikas. Pesan-pesan tersebut meliputi pesan *bootstrap, ping, join node, find node, store node, call request, call accept, call reject, call finish.* Berikut ini merupakan fungsi-fungsi dari masing-masing pesan tersebut.

a. *Bootstrap* pesan ini dikirimkan ketika *peer* untuk pertama kali aktif di dalam jaringan *peer-to-peer DHT*. Pesan ini kemudian diikuti dengan pesan *join* sebagai pesan permintaan untuk dapat bergabung dalam jaringan *peer-to-peer DHT* tersebut.

b. *Join Node* pesan ini dikirimkan ketika *peer* hendak mengenal *peer* tetangganya. Penerimaan pesan ini menimbulkan aksi berupa penambahan informasi *Triplet* dari *peer* tetangga kedalam ruang penyimpanan *KBucket* miliknya.

c. Pesan *ping* digunakan untuk mendeteksi sebuah *peer* masih dalam keadaan aktif atau tidak. Pesan ini digunakan sebelum penambahan *Triplet* kedalam *KBucket* ketika *KBucket* telah mencapai batas maksimal. Serta digunakan juga ketika sebelum melakukan panggilan *VoIP.*

d. *Find Node* adalah pesan yang digunakan untuk membangkitkan pencarian *node* pada *peer* penerus. Dimana penerimaan pesan ini dapat membangkitkan aksi pencarian *node* dalam KBuket *peer* penerimanya.

e. *Store Node* adalah pesan balasan dari pesan find *node*. Pesan ini dikirimkan ketika *node* yang dicari telah ditemukan.

f. *Call request* adalah pesan yang dikirimkan ketika sebuah *peer* ingin melakukan panggilan suara. Pesan call request ini dapat membangkitkan nada dering pada perangkat penerima, hal ini sebagai tanda bahwa terdapat panggilan masuk dari *peer* penelepon.

g. *Call Accept* adalah pesan yang dikirimkan jika *peer* penerima bersedia untuk melakukan panggilan suara dengan penelepon. Jika pesan ini telah dikirim maka dimulailah proses komunikasi antar kedua *peer* tersebut.

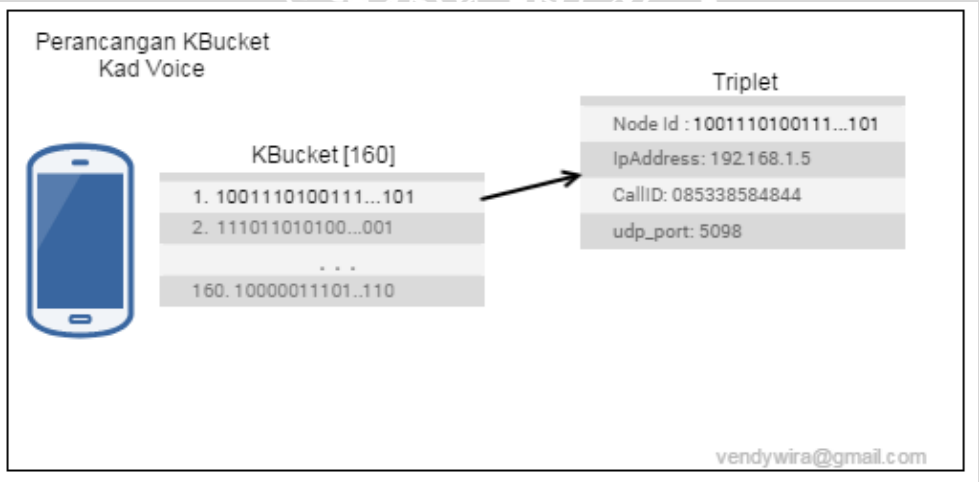
h. *Call Reject*, ada kalanya *peer* tujuan dalam keadaan sibuk sehingga tidak dapat melakukan komunikasi *VoIP*. Untuk melakukan penolakan terhadap panggilan masuk *peer* penerima kemudian mengirimkan pesan *call reject* kepada *peer*

penelepon. Setelah penelepon menerima pesan ini panggilan akan langsung dibatalkan.

i. Call finish adalah pesan yang dikirimkan untuk mengakhiri percakapan panggilan suara. Ketika pesan ini di terima oleh *peer* dapat membangkitkan penutupan koneksi *socket* dan menghentikan panggilan yang sedang berlangsung.

4.4.2 Perancangan *KBucket*

Seperti yang telah dibahas pada perancangan *peer* dapat diketahui bahwa masing-masing *peer* menyimpan informasi yang unik. Dimana informasi-informasi tersebut kemudian disimpan didalam sebuah wadah yang bernama *KBucket. KBucket* ini menyimpan informasi-infomasi terkait *peer* yang telah dikenalnya. Wadah ini memiliki kapasitas yang terbatas yaitu 160 ruang penyimpanan. Hal tersebut dimaksudkan agar tidak membebani penggunaan sumber daya penyimpanan pada *peer* dan juga dapat mempercepat proses pencarian *node* dalam *KBucket*. Dikatakan dapat mempercepat pencarian karena proses pencarian *node* akan membandingkan *node* yang dicari dengan *node* yang tersimpan dalam *KBucket*. Dengan dibatasinya jumlah *node* dalam *KBucket* akan mempercepat proses pencarian *node*.



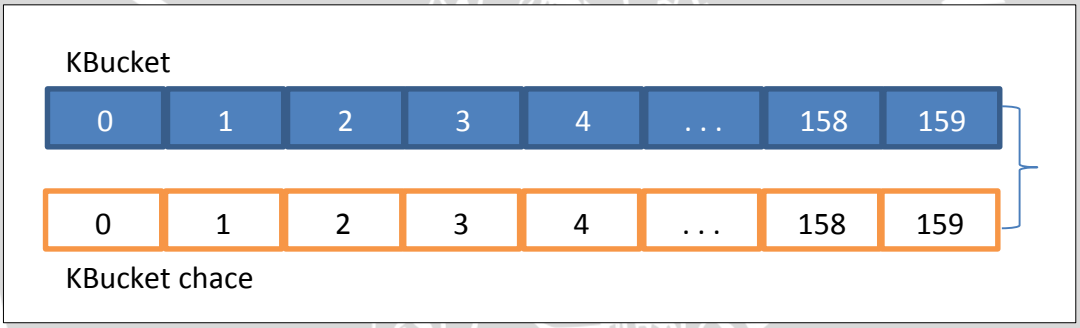
Gambar 4.5 Perancangan *KBucket*
Sumber: [Perancangan]

Pada masing-masing ruang penyimpanan *KBucket* akan disimpan sebuah kelas *Triplet*. *Triplet* inilah yang menyimpan informasi-informasi mengenai *peer* tetangganya. Informasi-informasi yang terdapat pada *Triplet* meliputi informasi *node id*, alamat *ip, udp port, call id* dan *call ip*. Informasi yang tersimpan dalam informasi alamat *ip* merupakan informasi alamat dengan format alamat *SIP.* Penulisan alamat *SIP* selalu diawali dengan identitas *node id* dan kemudian diikuti oleh alamat *ip* dan *udp port*. Peubah *Call id, call ip* dan *udp port* menyimpan informasi terkait nomor telepon, alamat *ipv4*, dan *port* yang digunakan.

Infomasi yang tersimpan didalam *Triplet* kemudian dipetakan kedalam *KBucket* saat terdapat penambahan *Triplet* baru. Penambahan *Triplet* ini dapat

terjadi ketika proses *join node* dan proses *store* saat *node* ditemukan. Untuk melakukan pemetaan tersebut, *Triplet* akan memenuhi *KBucket* dari bagian kiri (*head*) terlebih dahulu kemudian beranjak ke bagian kanan (*tail*). Setiap kali *Triplet* digunakan, *Triplet* tersebut kemudian dipindahkan kebagian *tail KBucket*. Saat *KBucket* mencapai batas maksimum, proses penambahan *Triplet* harus melaui mekanisme pemeriksaan *peer* terlebih dahulu. Pemeriksaan ini dilakukan dengan prosedur, *Triplet* yang menempati bagian *head* akan dikirimkan pesan ping. Saat *peer* tersebut tidak membalas maka *peer* itu dinyatkan tidak aktif dan *Triplet* bersangkutan kemudian dihapuskan. Setelah itu *Triplet* yang baru kemudian di petakan pada bagian *tail KBucket*. Jika yang terjadi sebaliknya maka *Triplet* yang baru akan di tolak dan dibuang.

Untuk penerapannya, *KBucket* disimpan dalam bentuk *array*. Hal ini dilakukan agar proses penggunaan *KBucket* bisa lebih efisien dari sisi waktu dan juga penggunaan sumber daya dibandingkan penyimpanan dalam *file* ataupun *database*. Dengan penggunaan *array* performa kecepatan dari sistem tersebut dapat meningkatkan. Peningkatan performansi sistem ini dapat diamati saat terjadi peningkatan jumlah *node* dalam sistem.



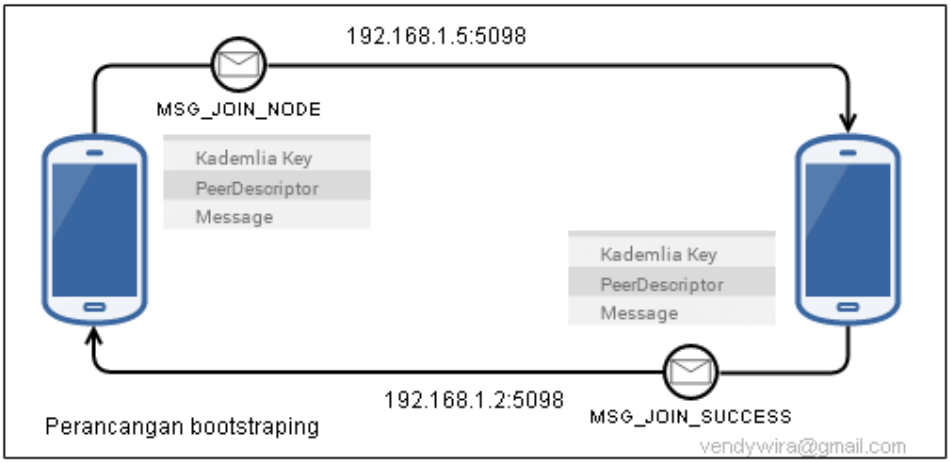
Gambar 4.6 Perancangan KBucket *Cache*
Sumber: [Perancangan]

Dibalik sisi positif pemakaian *array* memiliki sisi negatif. *Array* hanya dapat menyimpan nilai ketika aplikasi dalam keadaan berjalan. Sehingga saat pengguna menghentikan aplikasi informasi yang tersimpan dalam K*Bucket* akan hilang. Untuk mengatasi hal tersebut peneliti membuat sebuah mekanisme untuk menjaga nilai yang terdapat dalam *KBucket*. Hal tersebut diatasi dengan cara membuat salinan *KBucket* dan menyimpannya kedalam *Cache*. *Cache* ini disimpan ke dalam database. Yang membedakan fungsi *KBucket* dengan *Cache* ialah *Cache* tidak digunakan dalam proses pencarian *node* akan tetapi hanya sebagai penyimpan informasi yang diperlukan ketika proses *Bootstraping*. Untuk menjaga nilai yang tersimpan dalam *KBucket* sama dengan nilai yang disimpan dalam *Cache* maka dibuat sebuah mekanisme pembaharuan nilai *Cache*. Nilai *Cache* akan mengalami perubahan ketika terjadi perubahan nilai pada *KBucket*.

4.4.3 Perancangan Bootstraping

Pada saat pertama kali *peer* bergabung dalam jaringan *peer-to-peer kademlia, peer* saling berdiri sendiri tanpa mengenal *peer* lain. Masing-masing *peer* dapat berkomunikasi antara satu *peer* dengan *peer* yang lain jika hanya jika

peer tersebut telah saling mengenal *peer* secara langsung maupun dengan perantara *peer* tetangga. Oleh karena itu, maka dibuat sebuah mekanisme pertukaran pesan untuk mengenal *peer* lain yang masih aktif dalam jaringan. mekanisme pengenalan *node* ini disebut dengan mekanisme *bootstrapping.* Dimana mekanisme ini terjadi saat *KBucket* bernilai *null* atau kosong.



Gambar 4.7 Perancangan Bootstrapping
Sumber: [Perancangan]

Dalam penelitian ini menerapkan tiga cara yang dapat dilakukan dalam mekanisme bootstrapping. Cara pertama yaitu mengirimkan permintaan join kepada peer yang telah tersimpan di dalam KBucket cahe. Cara yang kedua mengirimkan pesan bootstrap secara broadcast. Dan cara yang terakhir adalah menambahkan sendiri peer yang masih aktif.

Mekanisme pertama ini mengirimkan pesan *bootrap* kepada setiap alamat yang tersimpan dalam *Cache*. Cara ini merupakan prioritas utama yang dilakukan sebelum kedua mekanisme lainnya dijalankan. Mekenisme ini dilakukan saat penyimpanan *Cache* tidak dalam keadaan kosong. Setelah terpenuhinya persyaratan itu kemudian *peer* mengirimkan pesan *bootstrap* kepada seluruh kontak yang terdapat dalam *Cache*. *Peer* yang membalas pesan tersebut kemudian ditambahkan ke dalam *KBucket*.

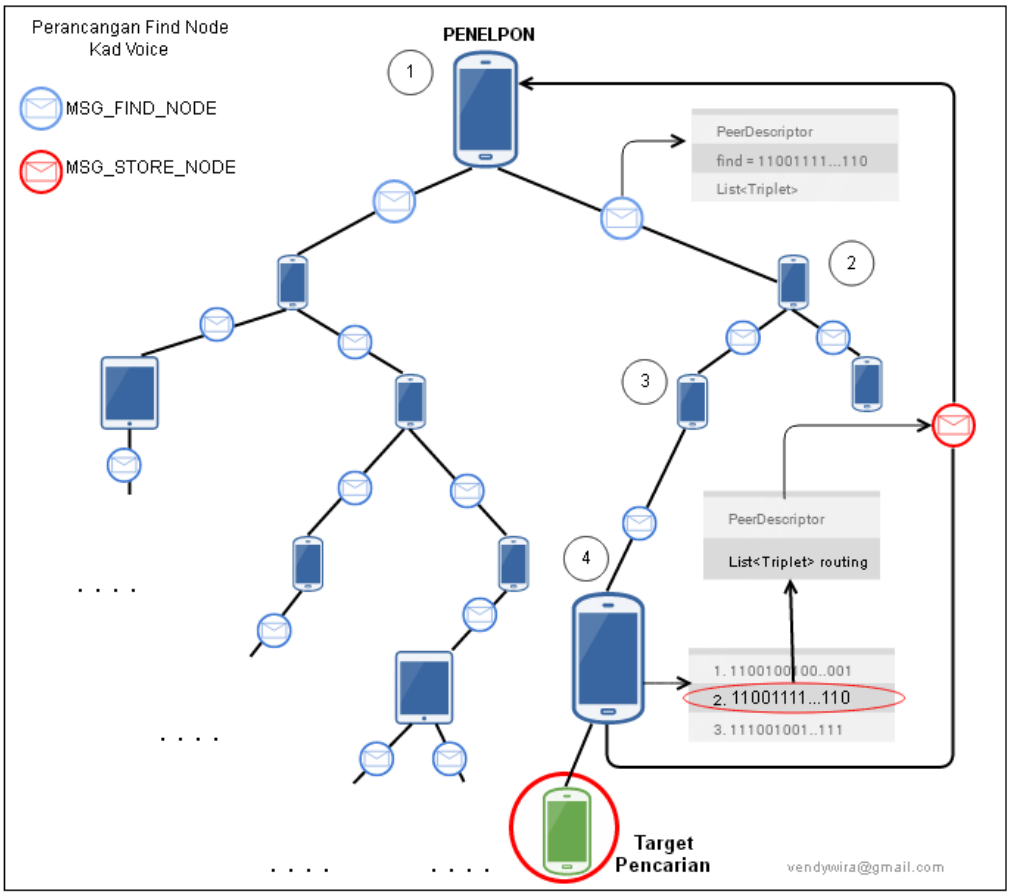
Cara kedua adalah pengenalan *node* menggunakan metode *broadcast*. Mekanisme ini dapat dilakukan ketika *Cache* dalam kondisi kosong atau *KBucket* masih kosong setelah melakukan mekanisme yang pertama. *Peer* akan melakukan pengiriman pesan secara *broadcast* ke jaringan dengan harapan terdapat sebuah *peer* yang telah bergabung dalam jaringan tersebut. Saat pesan *broadcast* di jawab maka *peer* baru tersebut kemudian mengirimkan permintaan *bootstrapping* kepada *peer* yang aktif.

Jika cara pertama dan kedua tidak berhasil atau *KBucket* masih dalam keadaan kosong maka Mekanisme yang dapat dilakukan adalah mekanisme pengiriman pesan *bootstrap* secara *manual*. Untuk cara ketiga ini alamat ip dan udp port harus diketahui karena infomasi alamat tersebut yang manjadi alamat

pengiriman permintaan *bootrapping* agar dapat bergabung dalam jaringan kademlia *peer-to-peer*.

4.4.4 Perancangan Pencarian *Node*

Dalam jaringan *Kademlia* sebuah *peer* dapat berkomunikasi dengan *peer* lain yang belum dikenalnya melalui perantara peer tetangga. Hal tersebut dapat terjadi karena adanya mekanisme pencarian *node* yang dimiliki oleh *DHT Kademlia*. Pada **gambar 4.8** mendeskripsikan alur proses yang dilalui untuk dapat menemukan *peer* lain yang tidak dikenal oleh *peer* pencari.



Gambar 4.8 Perancangan Mekanisme *Finding Node*
Sumber: [Perancangan]

Mekanisme pencarian *node* ini berawal ketika penelepon menghubungi nomor telepon *peer* tujuan. Nomor tersebut kemudian diubah menjadi bilangan biner dengan panjang data 160 *bit* yang disebut sebagai *node id*. *Node id* inilah yang kemudian dijadikan patokan dalam pencarian *node*.

Terdapat dua mekanisme pencarian *node* yang dilakukan dalam penelitian ini. Mekanisme pertama ialah mencari *node* yang telah dikenal dalam *KBucket*. Untuk itu dilakukan perbandingan nilai *node id* yang dicari dengan *node id* yang tersimpan dalam *KBucket* miliknya. Jika *node* yang dicari telah ditemukan pada *KBucket* miliknya mekanisme kedua tidak lagi dijalankan dan informasi terkait *peer*

yang telah ditemukan tersebut kemudian digunakan sebagai alamat pengiriman pesan *RPC* agar dapat memulai panggilan suara.

Mekanisme kedua adalah pencarian *node* yang dilakukan ketika *node id* tidak ditemukan dalam *KBucket*. Hal ini berarti *peer* tersebut tidak dikenal oleh *peer* pencari. Kemudian pencarian dilanjutkan dengan cara menunjuk dua *peer* dalam *KBucket* untuk untuk meneruskan pencarian *node* seperti yang terlihat pada urutan 1 dalam **gambar 4.8**.

Untuk memilih dua *peer* tersebut menggunakan metode matrik *XOR distance*. Matrik *XOR distance* ini membandingkan jarak kedekatan *node id* yang dicari dengan *node id peer* yang tersimpan dalam *KBucket*. Yang dimaksudkan Kedekatan disini, bukan merupakan kedekatan jarak pisikal antara *peer* yang ditunjuk dengan *peer* yang dicari, melainkan berdasarkan kedekatan jarak kemiripan *node id* tersebut.

Setelah kedua *peer* penerus pencarian ditentukan maka tahap selanjutnya melakukan pengiriman pesan pencarian *node*. Pesan pencarian *node* yang diterima oleh *peer* penerus kemudian membangkitkan aksi pencarian *peer* dalam *KBucket* miliknya. Jika dalam *KBucket* tersebut ditemukan *node id peer* yang dicari maka selanjutnya *peer* tersebut mengirimkan pesan *store node* kepada *peer* pencari seperti yang ditunjukan pada urutan ke-4 pada gambar. Jika belum ditemukan, *peer* tersebut memiliki kewajiban meneruskan pencarian *node* kepada dua *peer* yang terdapat pada *KBucket* miliknya seperti yang terlihat pada urutan ke-2 dan ke-3 pada gambar.

Untuk meningkatkan efiensi pencarian *node* maka dibuat sebuah aturan dimana sebuah *peer* hanya dapat dikunjungi sekali dalam satu proses pencarian *node*. *Peer* selanjutnya tidak dapat mengirimkan pesan pencarian *node* kepada *peer* yang pernah dikunjungi sebelumnya dan harus memilih dua *peer* yang berbeda dari daftar *peer* sebagai *peer* penerus. Jika semua *peer* yang terdapat dalam *KBucket* masuk kedalam daftar yang telah dikunjungi maka *peer* tersebut akan menghentikan pencariannya.

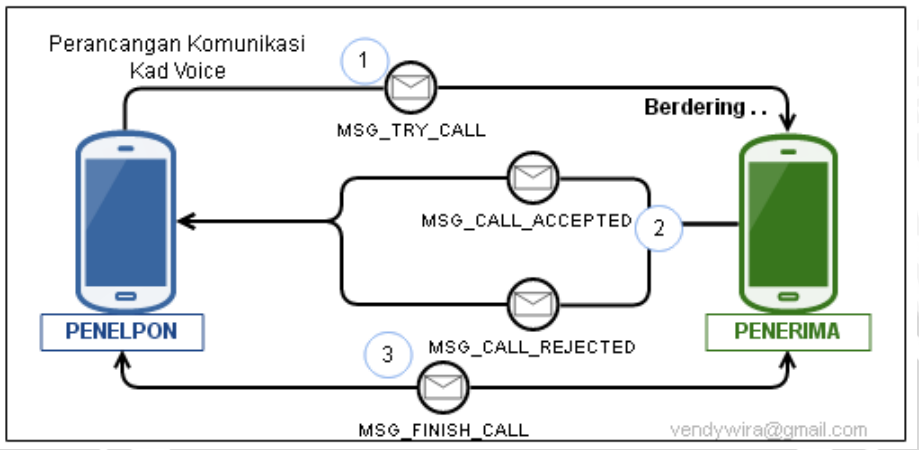
4.5 Perancangan *VoIP* (Voice over Internet Protocol)

Perancangan *VoIP* ini membahas mengenai bagian-bagian yang diimplementasikan dalam sistem *VoIP* kademlia *peer-to-peer*. *VoIP* dalam sistem kademlia *peer-to-peer* berfungsi sebagai proses yang dilalui sebuah *peer* penelpon agar dapat melakukan komunikasi suara dengan *peer* penerima. Adapun yang dirancang pada sub bagian ini meliputi perancangan mekanisme panggilan suara, pemilihan dan penggunaan *codec* dan transmisi suara.

4.5.1 Perancangan Panggilan Suara

Pada sub bagian perancangan panggilan suara ini mambahas terkait tahapan yang dilakukan saat sebelum terjadi komunikasi *VoIP* hingga komunikasi tersebut berakhir. Dalam melakukan panggilan suara terdapat dua *peer* yang terlibat didalamnya yaitu *peer* penelpon dan *peer* penerima telepon seperti yang

terlihat pada **gambar 4.9**. Masing-masing *peer* ini memiliki aksi yang berbeda saat terjadi panggilan suara. Oleh karena itu maka pada perancangan ini membahas dua pokok bahasan yaitu mengenai aksi-aksi yang dilakukan oleh penelpon dan penerima telepon.



Gambar 4.9 Perancangan Sistem *Voip* Secara Umum
Sumber: [Perancangan]

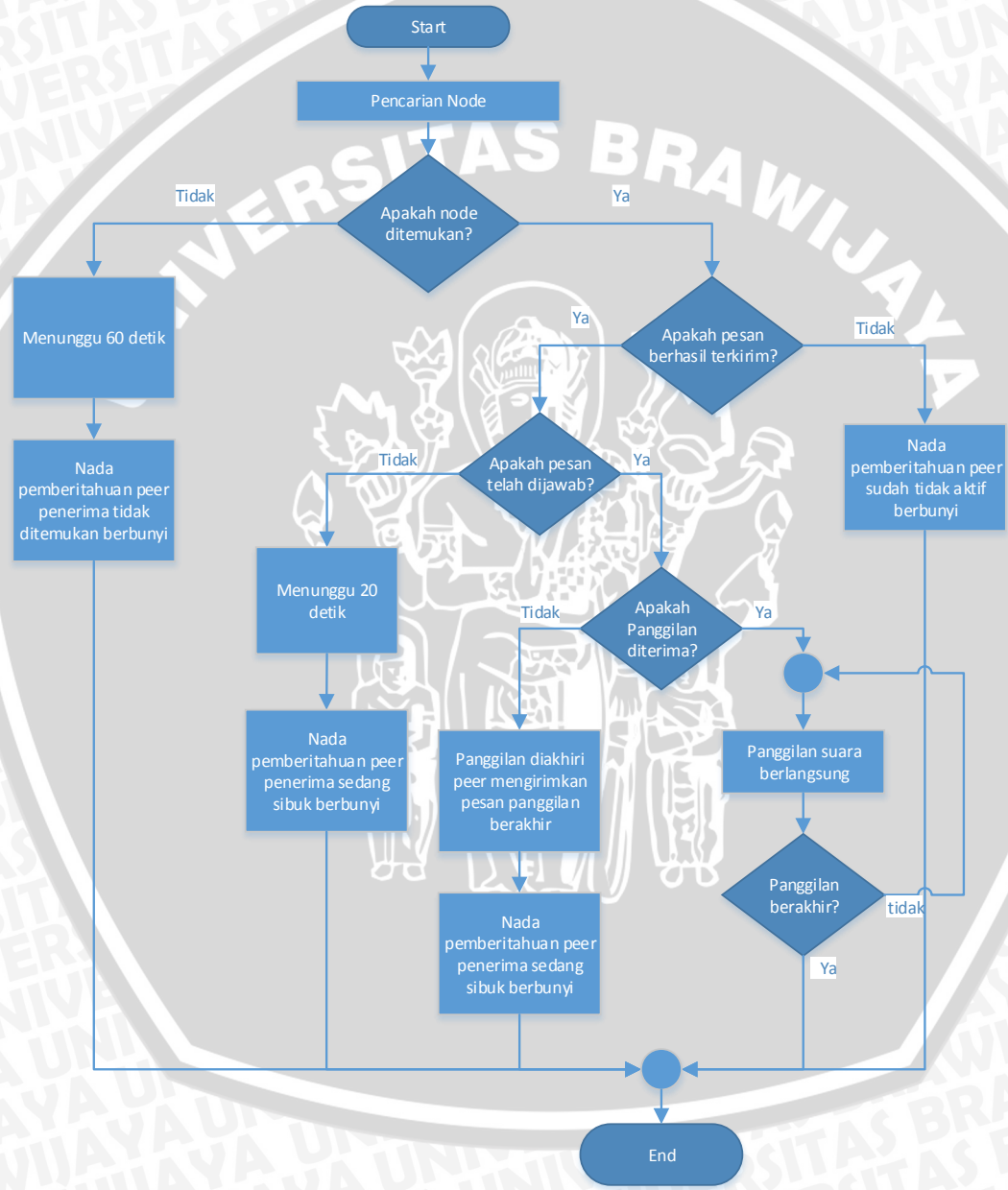
Nomor yang terdapat dalam gambar **gambar 4.9** menandakan tahapan-tahapan yang dilakukan saat melakukan komunikasi. Seperti yang terlihat pada tahap pertama, *peer* penelpon mengirimkan pesan *call request* untuk memberitahukan bahwa dirinya hendak melakukan komunikasi. Saat *peer* tujuan menerima pesan tersebut *peer* tujuan dapat membalasnya dengan pesan *call reject* atau *call accept* seperti yang ditandai oleh nomor 2 tersebut. Tahapan yang ketiga terjadi jika Kedua *peer* telah menjalin komunikasi suara. Untuk dapat mengakhiri komunikasi tersebut salah satu *peer* dapat mengirimkan pesan *finish call* sebagai tanda panggilan diakhiri.

Dalam melakukan panggilan ada sebuah aturan umum yang harus dipenuhi oleh setiap *peer* yaitu ketika sedang melakukan panggilan tidak dapat menerima panggilan dari *peer* lain. Dan jika terjadi permintaan panggilan pada saat *peer* tujuan sedang melakukan panggilan maka permintaan panggilan tersebut akan secara langsung ditolak.

Terkait dua bahasan pokok yang telah dibahas sebelumnya maka berikut ini merupakan deskripi singkat yang dilakukan dalam perancangan pada *peer* penelpon dan penerima telepon. Pada sisi penelpon dirancang bagaimana cara melakukan permintaan panggilan hingga aksi-aksi yang dapat dilakukan ketika terjadi kondisi-kondisi tertentu pada saat melakukan panggilan. Sedangkan pada sisi penerima dirancang aksi-aksi yang dapat dilakukan ketika menerima permintaan pemanggilan suara dari *peer* penelpon. Untuk lebih jelasnya dapat dilihat pada sub bagian perancangan pada sisi penelepon dan perancangan pada sisi penerima telepon sebagai berikut.

4.5.1.1 Perancangan Pada Sisi Penelepon

Secara keseluruhan proses yang terjadi ketika melakukan panggilan dari *peer* penelepon dapat dilihat pada **gambar 4.10**. Proses diawali dengan pencarian *node* tujuan yang diidentifikasi oleh nomor telepon penerima. Pada proses pencarian *node* ini memanfaatkan mekanisme pencarian *node* pada *DHT Kademlia* yang telah dijelaskan sebelumnya. Terdapat beberapa kemungkinan kondisi yang dapat terjadi pada saat pencarian *node*, kondisi itu meliputi *peer* tujuan ditemukan ataukah tidak.



Gambar 4.10 Kondisi Pada Saat Melakukan Panggilan Suara Dari Sisi Penelpon
Sumber: [Perancangan]

Seperti yang telah dijelaskan pada mekanisme pencarian *node Kademlia*, *peer* mengirimkan pesan pencarian terhadap dua *peer* yang telah dikenalnya. hal

tersebut terus dilakukan hingga *peer* yang dicari ditemukan. *Peer* dikatakan ditemukan saat *peer* pencari menerima pesan *store node* dari *peer* penerus pencarian. Suatu permasalahan timbul ketika *peer* yang dicari tak kunjung ditemukan. Ketika permasalahan itu terjadi *peer* penelpon akan terus menunggu hingga pesan *store* diterima. Jika hal itu terjadi maka proses untuk menunggu panggilan suara tak kunjung berhenti. Untuk mengatasi permasalahan tersebut, sehingga dibuat sebuah mekanisme pembatasan waktu tunggu. Lamanya waktu tunggu ditetapkan selama 60 detik. Ketika waktu tunggu tersebut berakhir dan pesan *store* tidak kunjung diterima maka *peer* penelpon kemudian mendapat pemberitahuan bahwa *peer* tujuan tidak ditemukan dan kemudian proses pemangilanpun dibatalkan.

Kondisi yang kedua pesan *store* diterima oleh *peer* penelpon hal ini menyatakan bahwa *peer* yang dicari telah pernah bergabung dalam jaringan tersebut. Ketika mengetahui *peer* tujuannya ditemukan kemudian *peer* penelpon mengirimkan pesan *RPC* permintaan panggilan suara atau *call request*. Dalam mengirimkan pesan *call request* terdapat dua kemungkinan yang terjadi. Kemungkinan pertama pesan berhasil dikirimkan dan kemungkinan kedua pesan gagal terkirim.

Kegagalan pengiriman dapat terjadi jika *peer* tujuan sudah tidak aktif dan keluar dari jaringan *kademlia*. Ketika pesan *RPC* yang dikirimkan gagal maka *peer* penelpon akan mendapat pemberitahuan tentang *peer* yang dituju tengah berada diluar jangkauan area. Kemudian proses pemanggilan suarapun berakhir. Kemungkinan berikutnya ialah pesan berhasil sampai pada tujuan dan *peer* penelepon akan menunggu balasan terkait permintaan panggilannya. Permasalahan akan terjadi ketika pengguna tidak mengetahui ada panggilan masuk pada *smartphone* milikinya. Yang terjadi ialah *peer* penelpon akan selalu menunggu dalam keadaan ketidak pastian. Hal seperti ini akan mengakibatkan ketidak nyamanan pengguna. Ketika pada saat yang sama terdapat *peer* yang ingin melakukan panggilan suara dengan *peer* penelpon, maka panggilan tersebut secara langsung ditolak karena *peer* penelpon masih dalam proses melakukan panggilan. Untuk mengatasi permasalahan tersebut maka diterapkan aturan untuk membatasi waktu tunggu *peer* penelepon selama 20 detik. Saat waktu tunggu telah habis dan *peer* penelpon belum mendapatkan jawaban dari pesan *call requestnya,* maka secara sepihak panggilan suara akan dibatalkan. Dan pada *peer* penelpon mendapat pemberitahuan bahwa *peer* tujuan dalam keadaan sibuk.

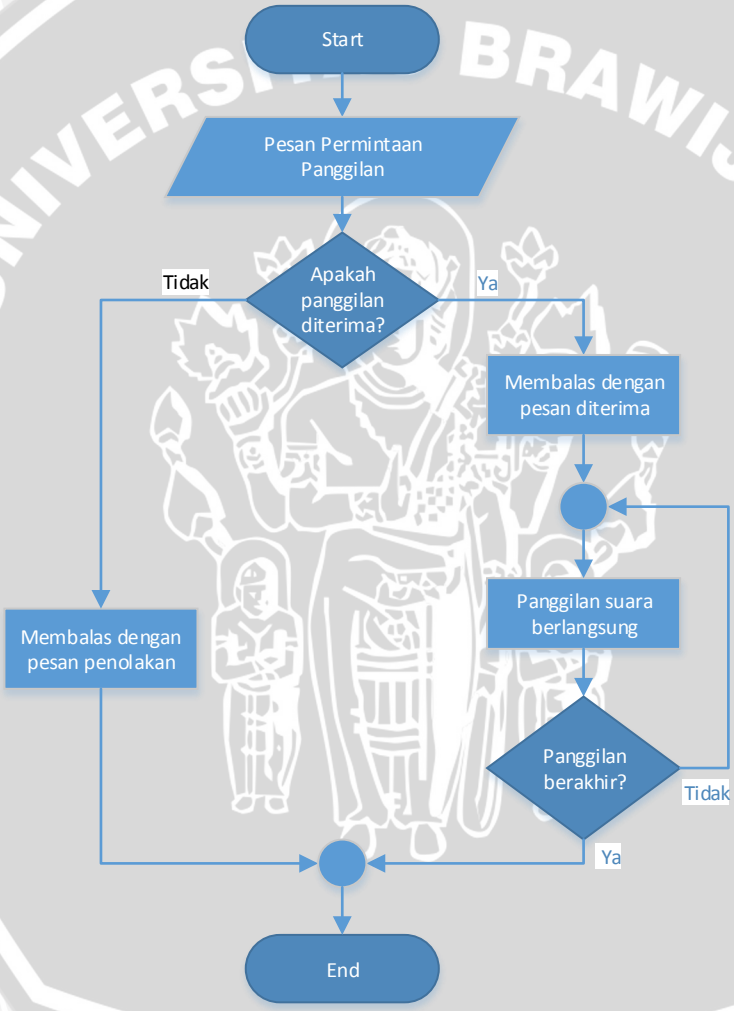
Balasan pesan *call request* dapat berupa *call reject* atau *call accept*. Jika pesan *call reject* yang diterima maka permintaan panggilan ditolak oleh penerima. Ketika panggilan ditolak maka *peer* penelpon kemudian mendapat pemberitahuan bahwa *peer* tujuannya dalam keadaan sibuk. Dan jika sebaliknya pesan yang diterima adalah pesan *call accept* maka akan dilanjutkan pada tahap komunikasi suara atara *peer* penelpon dan *peer* penerima telepon.

Setelah dirasa cukup melakukan panggilan suara salah satu *peer* dapat mengakhirinya. Jika diakhiri oleh *peer* penelpon maka panggilan suara pada *peer*

penelpon akan dihentikan dan *peer* penelpon akan mengirimkan pesan *call finish* kepada *peer* penerima. Dan jika sebaliknya yang menyudahi panggilan dari pihak penerima maka *peer* penelpon akan menerima pesan *call finish.* Pada saat pesan tersebut diterima maka proses penghentian panggilan dibangkitkan pada *peer* penerima.

4.5.1.2 Perancangan Pada Sisi Penerima Telepon

Perancangan pada sisi perima telepon tidak serumit perancangan pada sisi penelpon. Karena hanya terdapat dua kondisi yang dapat terjadi pada *peer* penerima telpon yaitu menerima atau menolak permintaan panggilan tersebut. Penjelasan visualnya dapat dilihat pada **gambar 4.11**.



Gambar 4.11 Kondisi Pada Saat Terjadi Panggilan Suara Dari Sisi Penerima
Sumber: [Perancangan]

Sebuah *peer* dapat dikatakan sebagai *peer* penerima telpon ketika *peer* tersebut menerima pesan permintaan panggilan atau *call request*. Pada saat pesan *call request* diterima maka selanjutnya pesan tersebut dapat membangkitkan nada panggilan pda *peer* penerimanya. Sehingga pengguna dapat mengetahui jika terdapat panggilan masuk.

Pada saat *peer* penerima berdering terdapat dua kondisi yang dapat dipilih oleh pengguna yaitu menolak panggilan tersebut ataukah menerimanya. Saat pengguna menolak maka *peer* penerima telpon akan mengirimkan pesan *RPC call reject* kepada *peer* penelpon dan panggilanpun dihentikan.

Kemungkinan yang kedua adalah panggilan tersebut diterima. Saat pengguna memutuskan untuk menerima permintaan telepon tersebut maka *peer* penerima telepon mengirimkan pesan *RPC call accept.* Dan proses komunikasipun dapat berlangsung. Ketika pengguna sudah merasa cukup dalam melakukan panggilan suara, pengguna dapat mengakhiri panggilan tersebut. Pada saat panggilan telah diakhiri pesan *RPC call finish* yang dikirimkan kepada *peer* penelpon.

4.5.2 Perancangan Pemilihan dan Penggunaan *Codec*

Komunikasi *VoIP* menghasilkan data yang cukup besar jika dibandingkan komunikasi teks biasa. Ukuran data yang besar tersebut dapat menimbulkan potensi paket *loss, jitter* dan *delay* yang besar pula. Untuk menghindari permasalahan tersebut diperlukan sebuah algoritma *codec* yang mampu meminimalkan data yang dikirimkan melalui jaringan. Codec ini dapat memperkecil ukuran data suara tanpa menghilangkan informasi yang tersimpan dalam data suara tersebut.

Terdapat beberapa *codec* yang direkomendasikan oleh ITU. Pada penelitian ini menyediakan 4 *codec* yang siap untuk digunakan. *Codec-codec* tersebut meliputi *Speex, G711u, G711a,* dan *G722*. Dalam penerapannya peneliti tidak membuat *codec* dari awal melainkan menggunakan kembali *codec* yang telah tersedia. *Codec* yang akan digunakan adalah *codec* dari project *Sipdroid* dengan lisensi *open source*.

Pada aplikasi *VoIP* pengguna dapat memilih secara bebas *codec* yang mereka inginkan. Komunikasi suara akan jelas ketika masing-masing *peer* menggunakan *codec* yang sejenis. Hal tersebut terjadi karena masing-masing *codec* memiliki algoritma yang berbeda-beda. Sehingga sebuah *codec* memiliki mekanisme *encode* dan *decode*nya tersendiri. Proses *encode* adalah proses dimana data suara akan dikonversikan menjadi data yang lebih kecil dan siap untuk dikirimkan melalui jaringan. Untuk mengubah data *encode* menjadi informasi suara diperlukan proses *decode*. Proses *decode* adalah proses kebalikan dari proses *encode* dengan melalui proses *decode* ini informasi suara dapat didengar dengan jelas.

Untuk dapat menjamin *codec* yang digunakan penelpon dan penerima merupakan *codec* yang sejenis maka harus ada mekanisme tersendiri yang dapat mengontrolnya. Cara untuk mengontrolnya yaitu dengan cara memberitahukan pada calon penerima terkait *codec* yang digunakan oleh penelpon. Sehingga saat berkomunikasi, *codec* pada penerima dapat disamakan dengan *codec* penelpon. Untuk mekanisme penerapannya pemberitahuan *codec* dilakukan saat pengiriman pesan *call request*. Dan pada saat penerima menyetujui permintaan panggilan suara maka *codec* dari *peer* perima disesuaikan dengan informasi *codec* pada

pesan call request. Dengan demikian *codec* antar *peer* penelpon dan *peer* penerima dapat disamakan dan komunikasipun dapat dilakukan.

4.5.3 Perancangan Transmisi Data

Agar dapat melakukan komunikasi antar *peer* penelpon dan penerima maka suara harus dapat disampaikan pada lawan bicara. Untuk dapat meyampaikan informasi suara tersebut diperlukan mekanisme transmisi data dari sebuah *peer* menuju *peer* yang menjadi lawan bicaranya. Untuk itu maka pada bagian ini dirancang proses pengiriman data tersebut.

Dalam penelitian ini komunikasi *VoIP* bersifat *real time.* Data suara yang dihasilkan pada salah satu pengguna dapat didengar oleh lawan bicaranya dalam waktu yang bersamaan. Karena sifatnya yang *real time* ini maka diperlukan protokol yang mampu mengirimkan data secara *realtime* juga. Oleh karena itu peneliti menggunakan protokol *RTP (Real Time Protocol)* dimana protokol tersebut telah dirancang untuk keperluan komunikasi yang bersifat *realtime*.

Saat permintaan panggilan suara disetujui oleh penerima, seketika itu juga dibentuk sebuah *socket* datagram untuk komunikasi kedua *peer*. S*ocket* tersebut berisikan alamat *ip* dan *udp port* dari lawan bicaranya. Suara yang dihasilkan oleh pengguna kemudian disimpan dalam bentuk data *i/o stream.* Data suara ini kemudian melalui mekanisme *encode* sebelum dikirimkan. Hasil dari proses *encode* kemudian dimasukan ke dalam paket-paket *RTP*. Dan paket-paket tersebut telah siap untuk dilewatkan pada *socket*. Pada saat lawan bicara menerima paket *RTP,* paket tersebut akan melalui proses *decode*. Data hasil *decode* kemudian dibaca secara bertahap sehingga informasi suara dapat terdengar. Untuk mengurangi pemakian memori maka data yang telah didengar akan secara langsung dihapus. Setelah percapakapan selesai dan panggilan diakhiri *socket* kemudian ditutup.

Dalam komunikasi *VoIP*, terjadi komunikasi dua arah. Dimana ketika suatu *peer* mengirimkan data pada saat yang sama *peer* tersebut juga mampu menerima data dari *peer* lain. Sehingga terdapat dua proses yang berjalan secara bersamaan dalam waktu yang sama. Oleh karena itu dalam penerapannya dibutuhkan mekanisme *multi thread* agar dapat mengkoordinir proses-proses tersebut. Dalam multi thread sangat rentan terjadi *deadlock.* Sehingga harus dilakukan proses sinkonisisasi supaya sumberdaya yang masih terpakai tidak dapat digunakan oleh proses lain sampai sumber daya tesebut dilepaskan.

BAB 5 IMPLEMENTASI

5.1 Implementasi *Peer*

Seperti yang telah dirancang pada bab 4 setiap *peer* menyimpan informasi terkait nama, *ip address* dan juga *udp port.* Maka dalam penerapannya semua informasi pada *peer* tersebut disimpan dalam kelas yang bernama *PeerDescriptor.* Untuk lebih jelasnya dapat dilihat pada **gambar 5.1** dibawah ini.

```
1  public class PeerDescriptor implements Serializable {
2
3     private String name;
4     private String address;
5     private int udpPort;
6
7  public PeerDescriptor(String name, String address,int udpPort){
8     this.setName(name);
9     this.setAddress(address);
10    this.setUdpPort(udpPort);
11 }
12 public String getName() {
13    return name;
14 }
15 public void setName(String name) {
16    this.name = name;
17 }
18 public String getAddress() {
19    return address;
20 }
21 public void setAddress(String address) {
22    this.address = address;
23 }
24 public void setUdpPort(int udpPort) {
25    this.udpPort = udpPort;
26 }
27 public int getUdpPort(){
28    return udpPort;
29 }
30
```

Gambar 5.1 Kode Implementasi Kelas *PeerDescriptor.*
Sumber: [Implementasi]

Kelas *PeerDescriptor* menyimpan informasi umum pada sebuah *peer*. Kelas ini diintansiasi pada saat pertama kali aplikasi dijalankan. Informasi yang tersimpan dalam kelas *PeerDescriptor* yang akan dipertukarkan ketika sebuah *peer* melakukan komunikasi dengan *peer* lain. Kelas *PeerDescriptor* terdiri dari metode *set* untuk melakukan penyimpanan nilai dan metode *get* untuk pengambilan informasi. Hal tersebut dapat dilihat pada baris ke-12 hingga baris ke-29.

5.2 Implementasi Kademlia

Terkait dengan proses pengenalan *node* menggunakan *DHT Kademlia* terdapat beberapa bagian yang harus diimplementasikan agar sesuai dengan perancangan pada bab sebelumnya. Adapun yang diimplementasikan meliputi *KBucket, Cache*, pesan *RPC*, proses *bootstrapping*, serta mekanisme pencarian *node.*

5.2.1 Implementasi Pesan *RPC*

Hal pertama yang diimplementasikan pada pesan *RPC* adalah membuat sebuah kelas dasar yang bernama *KadBasicMessage*. Kelas ini berisikan konten dasar dari setiap jenis pesan *RPC*. Dalam kelas *KadBasicMessage* diimplentasikan tipe pesan yang dikirimkan. Sehingga setiap pesan *RPC* memiliki identitas tipe tersendiri. Untuk lebih jelasnya bisa dilihat pada **gambar 5.2** dibawah ini.

```
01 public class KadBasicMessage {
02    private String type;
03    public String getType() {
04       return type;
05    }
06    public KadBasicMessage(String type) {
07       this.type = type;
08    }
09 }
10
```

Gambar 5.2 Kode Implementasi Kelas *KadBasicMessage*.
Sumber: [Implementasi]

Pada baris ke-6 terdapat konstruktor yang berfungsi sebagai penginisialisasi jenis pesan dari setiap kelas turunannya. Dan untuk mengetahui jenis pesan *RPC* yang dikirimkan dapat dapat menggunakan metode *getType()* seperti kode pada baris ke-3 hingga 5.

```
01 @Override
02 public void sendMessage(Address toAddress,Address toContactAddress,
03             Address fromAddress,String msg, String contentType){
04    super.sendMessage(toAddress,toContactAddress,fromAddress,msg,
05             contentType);
06 }
```

Gambar 5.3 Kode Implemetasi Pengiriman Pesan *RPC*.
Sumber: [Implementasi]

Untuk melakukan pengiriman pesan *RPC* menggunakan fungsi *sendMessage* seperti yang terlihat pada kode implementasi pada **gambar 5.3**. Fungsi ini merupakan fungsi yang dipanggil kembali dari fungsi yang telah disediakan oleh pustaka *sip2peer*. Fungsi inilah yang bertugas dalam hal pengiriman pesan *RPC*.

Pustaka *Sip2peer* telah menyediakan fungsi-fungsi yang dapat mendeteksi pesan yang terkirim dan pesan yang mengalami kegagalan pengiriman. Fungsi-fungsi tersebut kemudian digunakan kembali dan dimodifikasi untuk disesuaikan dengan keperluan penelitian ini seperti yang terlihat pada **gambar 5.4**.

```
01 @Override
02 protected void onDeliveryMsgFailure(String peerMsgSended,
03                       Address receiver, String contentType) {
04    JsonParser parser = new JsonParser();
05    JsonObject msg = (JsonObject) parser.parse(peerMsgSended);
06    String msgType = msg.get("type").getAsString();
07    . . .
08 }
09 @Override
10 protected void onDeliveryMsgSuccess(String peerMsgSended,
11                     Address receiver, String contentType) {
12
```

```
13          String messageType = peerMsg.get("type").getAsString();
14          . . .
15  }
16  @Override
17  protected void onReceivedJSONMsg(JsonObject peerMsg,
18                                                  Address sender) {
19          String messageType = peerMsg.get("type").getAsString();
20          . . .
21  }
22
```

Gambar 5.4 Kode Implementasi Penangkap Pesan *RPC.*
Sumber: [Implementasi]

Ketika pesan *RPC* datang pesan tersebut akan terdeteksi oleh fungsi *on Receive JSON Message*. Dalam fungsi *on Receive JSON Message* ditempatkan fungsi yang dapat membangkitkan aksi pada masing-masing pesan *RPC* yang diterima nantinya. Selain fungsi *on Receive JSON Message* terdapat pula fungsi *on Delivery Message Failure*. Fungsi ini akan dijalankan ketika pengiriman pesan *RPC* gagal karena alasan-alasan tertentu seperti *node* telah tidak aktif maupun karena terjadi gangguan pada jaringan. Dan yang terakhir adalah fungsi *on Delivery Message Success*. Fungsi tersebut dieksekusi saat pesan *RPC* berhasil terkirim.

5.2.1.1 Implementasi Pesan Bootstrapping.

Pengiriman pesan *bootstrapping* bertujuan untuk mendapatkan informasi dari *peer* tetangganya. Untuk kode implementasi pesan bootstrapping dapat dilihat pada **gambar 5.5** di bawah ini.

```
01  public class BootstrapMessage extends KadBasicMessage {
02      public static final String MSG_BOOTSTRAP = "bootstrap";
03      PeerDescriptor peerDescriptor;
04      KademliaKey ID;
05      public BootstrapMessage(PeerDescriptor pd, KademliaKey key) {
06          super(MSG_BOOTSTRAP);
07          this.peerDescriptor = pd;
08          this.ID = key;
09      }
10      public PeerDescriptor getPeerDescriptor() {
11          return peerDescriptor;
12      }
13      public KademliaKey getID() {
14          return ID;
15      }
16      public String getJSONString() {
17          Gson gson = new Gson();
18          return gson.toJson(this);
19      }
20  }
21
```

Gambar 5.5 Kode Implementasi Pesan *Bootstrapping.*
Sumber: [Implementasi]

Pada baris ke-6 dilakukan penetapan bahwa pesan ini merupakan pesan *bootstrap.* Jika diamati pada baris ke 3 dan 4 menjelaskan bahwa pesan *bootstrap* membawa informasi mengenai *PeerDescriptor* dan *node id* miliknya. Informasi *PeerDescriptor* menyimpan identitas alamat *ip, port* serta nomor telepon sebuah

peer. Informasi alamat *ip* dan *port* digunakan sebagai alamat pengiriman balasan pesan oleh penerima.

5.2.1.2 Implementasi Pesan *Join.*

Pesan *join* dikirimkan ketika sebuah *peer* hendak bergabung dalam jaringan *peer-to-peer Kademlia*. Informasi-informasi yang tersimpan dalam pesan *join* dapat dilihat pada kode sesuai **gambar 5.6** dibawah ini.

```
01  public class JoinNodeMessage extends KadBasicMessage{
02
03      public static final String MSG_JOIN_NODE = "join_message";
04      PeerDescriptor peerDescriptor;
05      KademliaKey ID;
06
07      public JoinNodeMessage(PeerDescriptor pd, KademliaKey key) {
08          super(MSG_JOIN_NODE);
09          this.peerDescriptor = pd;
10          this.ID = key;
11      }
12      public PeerDescriptor getPeerDescriptor() {
13          return peerDescriptor;
14      }
15      public KademliaKey getID() {
16          return ID;
17      }
18      public String getJSONString() {
19          Gson gson = new Gson();
20          return gson.toJson(this);
21      }
22  }
23
```

Gambar 5.6 Kode Implementasi Pesan *Join*.
Sumber: [Implementasi]

Pesan *Join* memiliki kemiripan dengan pesan *bootstrapping* yang membedakannya hanya pada jenis pesan tersebut. Ketika sebuah *peer* mengirimkan pesan *join* kepada *peer* tujuan. *Peer* penerima akan mejawabnya dengan pesan *join success.* Untuk kode implementasi dari pesan *join success* dapat diamati pada **gambar 5.7**.

```
01  public class JoinNodeSuccess extends KadBasicMessage {
02      public static final String MSG_JOIN_SUCCESS =
03                                      "success_join_message";
04
05      PeerDescriptor peerDescriptor;
06      KademliaKey ID;
07
08      public JoinNodeSuccess(PeerDescriptor pd, KademliaKey key) {
09          super(MSG_JOIN_SUCCESS);
10          this.peerDescriptor = pd;
11          this.ID = key;
12      }
13      public PeerDescriptor getPeerDescriptor() {
14          return peerDescriptor;
15      }
16      public KademliaKey getID() {
17          return ID;
18      }
19      public String getJSONString() {
20          Gson gson = new Gson();
```

```
21          return gson.toJson(this);
22      }
23  }
24
```

Gambar 5.7 Kode Implementasi Pesan *Join Success.*
Sumber: [Implementasi]

Pesan *join* success ini menjadi sebuah tanda bahwa permintaan *node* untuk bergabung telah diterima. Pesan *join* success ini membawa informasi balasan terkait identitas dari *peer* penerima pesan *join* seperti terlihat pada baris ke 5 dan ke 6.

5.2.1.3 Implementasi Pesan *Find Node.*

Pesan *find node* dikirimkan ketika sebuah *peer* hendak berkomunikasi dengan *peer* lain yang belum dikenalnya. Pesan ini menyimpan informasi terkait informasi *PeerDescriptor* dari *peer* pencari, *node* id yang dicari, daftar *node* yang telah dikunjungi serta rute yang telah dilewati. Hal tersebut dapat dilihat pada baris ke-5 hingga baris ke-8. Untuk lebih jelasnya dapat dilihat pada kode implemtasi pesan *find node* pada **gambar 5.8** berikut.

```
01  public class FindNodeMessage extends KadBasicMessage{
02
03     public static final String MSG_FIND_NODE="find_node_message";
04
05     private PeerDescriptor peerDescriptor;
06     private KademliaKey nodeId;
07     private ArrayList<Triplet> list;
08     private ArrayList<Triplet> routing;
09
10    public FindNodeMessage(ArrayList<Triplet> list,
11                           ArrayList<Triplet> routing,
12                           PeerDescriptor pd, KademliaKey ID) {
13        super(MSG_FIND_NODE);
14        this.peerDescriptor = pd;
15        this.key = ID;
16        this.list = list;
17        this.routing= routing;
18     }
19     public PeerDescriptor getPeerDescriptor() {
20        return peerDescriptor;
21     }
22     public KademliaKey getKey() {
23        return key;
24     }
25     public ArrayList<Triplet> getList() {
26        return list;
27     }
28     public ArrayList<Triplet> getRouting(){
29        return routing;
30     }
31     public String getJSONString() {
32        Gson gson = new Gson();
33        return gson.toJson(this);
34     }
35  }
36
```

Gambar 5.8 Kode Implementasi Pesan Find *Node*.
Sumber: [Implementasi]

Ketika melakukan pencarian terdapat beberapa *node* yang dilewati sebelum menemukan *node* tujuan. *Node-node* yang dilewati ini kemudian disimpan kedalam sebuah *list* dengan peubah *routing*. Setiap *node* yang dilewati membuat sebuah pesan find *node* menambah daftar *routing*-nya. Hal ini dapat dilihat pada baris ke-17. Dalam efisiensi waktu *node* yang telah dilewati tidak akan dikunjungi untuk kedua kalinya pada rute yang sama pada setiap kata kunci pencarian. Sehingga informasi mengenai daftar *node* yang telah dikunjungi disimpan kedalam variabel yang bernama list (dapat dilihat pada baris ke – 7).

5.2.1.4 Implementasi Pesan *Store Node.*

Pesan *store node* ini merupakan jawaban dari pesan *find node* dalam mekanisme pencarian *node*. Pesan ini dikirimkan ketika sebuah *peer* mengetahui informasi dari *peer* yang dicari. Informasi yang tersimpan dalam pesan ini berupa informasi *PeerDescriptor* dan *node id* dari *peer* yang dicari serta informasi rute yang telah dilalui. Untuk kode implementasi mengenai informasi yang tersimpan dalam pesan ini dapat dilihat pada baris ke-4 hingga baris ke-6 dalam **gambar 5.9** berikut.

```
01  public class StoreNodeMessage extends KadBasicMessage{
02      public static final String MSG_FOUND_NODE="found_node_message";
03
04      private PeerDescriptor peerDescriptor;
05      private KademliaKey NodeID;
06      private ArrayList<Triplet> routing;
07
08      public StoreNodeMessage(ArrayList<Triplet> routing,
09                              PeerDescriptor pd, KademliaKey ID) {
10          super(MSG_FOUND_NODE);
11          this.peerDescriptor = pd;
12          this.key = ID;
13          this.routing = routing;
14      }
15      public PeerDescriptor getPeerDescriptor() {
16          return peerDescriptor;
17      }
18      public KademliaKey getKey() {
19          return key;
20      }
21      public ArrayList<Triplet> getRouting() {
22          return routing;
23      }
24      public String getJSONString() {
25          Gson gson = new Gson();
26          return gson.toJson(this);
27      }
28  }
29
```

Gambar 5.9 Kode Implementasi Pesan *Store Node*
Sumber: [Implementasi]

5.2.1.5 Implementasi Pesan *Ping*

Pesan *ping* digunakan untuk mengetahui sebuah *node* dalam keadaan aktif atau tidak. Pesan ini dikirimkan saat terjadi penambahan *KBucket* yang *telah penuh* serta sesaat sebelum komunikasi *VoIP* berlangsung. Pesan *ping* dibedakan

menjadi 2 jenis yaitu *ping bucket* dan *ping call.* Untuk kode implementasinya dapat dilihat pada gambar kode implementasi *ping bucket* dan *ping call*.

```
01  public class BucketPingMessage extends KadBasicMessage{
02      public static final String MSG_PING_BUCKET =
03                                          "ping_bucket_message";
04      PeerDescriptor peerDescriptor;
05      public BucketPingMessage(PeerDescriptor pd) {
06          super(MSG_PING_BUCKET);
07          this.peerDescriptor = pd;
08      }
09      public PeerDescriptor getPeerDescriptor() {
10          return peerDescriptor;
11      }
12      public String getJSONString() {
13          Gson gson = new Gson();
14          return gson.toJson(this);
15      }
16  }
17
```

Gambar 5.10 Kode Implementasi *Ping Bucket*.
Sumber: [Implementasi]

```
01  public class CallPingMessage extends KadBasicMessage {
02     public static final String MSG_KAD_PING = "kad_ping_message";
03
04  PeerDescriptor peerDescriptor;
05
06     public CallPingMessage(PeerDescriptor pd) {
07        super(MSG_KAD_PING);
08        this.peerDescriptor = pd;
09     }
10     public PeerDescriptor getPeerDescriptor() {
11        return peerDescriptor;
12     }
13     public String getJSONString() {
14        Gson gson = new Gson();
15        return gson.toJson(this);
16     }
17  }
```

Gambar 5.11 Kode Implementasi *Ping Call.*
Sumber: [Implementasi]

Pesan *ping bucket* dan *ping call* membawa informasi yang sama yaitu informasi terkait *peer* pengirim. Hal ini dapat dilihat pada baris ke 4 pada kode impelementasi *ping bucket* maupun *ping call*. Saat *peer* tujuan masih aktif dan menerima pesan ping maka per tersebut akan membalasnya dengan pesan *pong*. Untuk implementasi pesan ping dapat dilihat pada **gambar 5.10** dan **gambar 5.11**.

Pesan pong dikirimkan untuk memberitahukan *peer* pengirim bahwa dirinya masih dalam keadaan aktif. Untuk kode implementasi pesan pong dapat dilihat pada gambar berikut.

```
01  public class BucketPongMessage extends KadBasicMessage {
02
03      public static final String MSG_PONG_BUCKET =
04                                      "kad_pong_message";
05
06      PeerDescriptor peerDescriptor;
07
```

```
08      public BucketPongMessage(PeerDescriptor pd) {
09          super(MSG_PONG_BUCKET);
10          this.peerDescriptor = pd;
11      }
12      public PeerDescriptor getPeerDescriptor() {
13          return peerDescriptor;
14      }
15      public String getJSONString() {
16          Gson gson = new Gson();
17          return gson.toJson(this);
18      }
19  }
20
```

Gambar 5.12 Kode Implementasi Pesan *Pong Bucket*.
Sumber: [Implementasi]

```
01  public class CallPongMessage extends KadBasicMessage {
02
03    public static final String MSG_KAD_PONG = "kad_pong_message";
04
05     PeerDescriptor peerDescriptor;
06     KademliaKey ID;
07    public CallPongMessage(PeerDescriptor pd, KademliaKey key) {
08        super(MSG_KAD_PONG);
09        this.peerDescriptor = pd;
10     }
11     public PeerDescriptor getPeerDescriptor() {
12        return peerDescriptor;
13     }
14     public String getJSONString() {
15        Gson gson = new Gson();
16        return gson.toJson(this);
17     }
18  }
19
```

Gambar 5.13 Implementasi Pesan *Pong Call*.
Sumber: [Implementasi]

Pesan *ping bucket* dibalas dengan *pong backet* dan *ping call* dibalas dengan pesan *pong call*. Untuk implementasi *pong baucke*t dapat dilihat pada kode implementasi pesan pong bucket dalam **gambar 5.12**. Sedangkan kode pesan pong call dapat dilihat pada **gambar 5.13**.

Pada implementasi *ping bucket, pong backet, ping call* maupun *pong call* memiliki kesamaan informasi yang dibawanya yang membedakanya ialah tipe dari pesan tersebut. Informasi yang dibawa oleh pesan *ping* dan *pong* yaitu informasi *PeerDescriptor* seperti yang terlihat pada baris ke-4 di setiap gambar kode implementasi pesan *ping* dan *pong*.

5.2.1.6 Implementasi Pensinyalan Pesan Pada Komunikasi *VoIP.*

Ketika sebuah *peer* hendak melakukan komunikasi *VoIP, peer* tesebut harus terlebih dahulu memberitahukan *peer* tujuan terkait maksudnya tersebut. Sehingga dibuat pesan pensinyalan agar kedua *peer* dapat berkordinasi satu sama lain. Pesan-pesan pensinyalan berupa pesan *call request, call accept, call reject* dan *call finish*.

Saat sebuah *peer* penelpon hendak melakukan komunikasi *VoIP* dengan penerima. *Peer* penelepon tersebut harus mengirimkan pesan *call request* terlebih dahulu sebagai tanda untuk mengajak komunikasi. Informasi yang dikirimkan dalam pesan ini berupa informasi *PeerDescriptor* dan *node id* dari *peer* penelepon serta *codec* yang digunakan selama proses komunikasi berlangsung. Keterangan tersebut dapat dilihat pada implentasi baris ke-4 hingga baris ke-6 seperti yang terlihat pada **gambar 5.14**. Setelah *peer* penerima menerima pesan *call request* ini maka *peer* penerima kemudian membalas pesan tersebut dengan pesan *call accept* maupun *call reject*.

```
01  public class CallRequestMessage extends KadBasicMessage {
02
03    public static final String MSG_TRY_CALL_REQUEST = "TRY_CALL";
04      PeerDescriptor peerDescriptor;
05      KademliaKey ID;
06      Data codec;
07      public CallRequestMessage(PeerDescriptor pd,
08                                KademliaKey key,Data codec) {
09          super(MSG_TRY_CALL_REQUEST);
10          this.peerDescriptor = pd;
11          this.ID = key;
12          this.codec=codec;
13      }
14      public PeerDescriptor getPeerDescriptor() {
15          return peerDescriptor;
16      }
17      public KademliaKey getID() {
18          return ID;
19      }
20      public Data getCodec(){
21          return this.codec;
22      }
23      public String getJSONString() {
24          Gson gson = new Gson();
25          return gson.toJson(this);
26      }
27  }
28
```

Gambar 5.14 Kode Implementasi *Call Request*.
Sumber: [Implementasi]

Ketika pesan *call Accept* yang diterima oleh *peer* penelpon, hal ini menandakan bahwa *peer* penerima bersedia untuk melakukan komunikasi *VoIP* dengan *peer* tersebut. Informasi yang tersimpan dalam pesan ini berupa informasi dari *PeerDescriptor* serta *node id* dari *peer* penerima. Hal tersebut dapat dilihat pada baris ke-4 dan ke-5 pada **gambar 5.15**.

```
01  public class CallAcceptedMessage extends KadBasicMessage{
02      public static final String MSG_CALL_ACCEPTED = "CALL_ACCEPT";
03
04      PeerDescriptor peerDescriptor;
05      KademliaKey ID;
06
07      public CallAcceptedMessage(PeerDescriptor pd,KademliaKey key){
08          super(MSG_CALL_ACCEPTED);
09          this.peerDescriptor = pd;
10          this.ID = key;
11      }
12      public PeerDescriptor getPeerDescriptor() {
```

```
13          return peerDescriptor;
14      }
15      public KademliaKey getID() {
16          return ID;
17      }
18      public String getJSONString() {
19          Gson gson = new Gson();
20          return gson.toJson(this);
21      }
22  }
23
```

Gambar 5.15 Kode Implementasi Pesan *Call Accept*.
Sumber: [Implementasi]

Pesan *call reject* merupakan kebalikan dari pesan *call accept*. Sehingga ketika pesan ini yang diterima menandakan *peer* penerima tidak besedia untuk melakukan komunikasi *VoIP*. Sama halnya dengan pesan *call accept*, pesan *call reject* juga membawa informasi *PeerDescriptor* serta *node id* dari *peer* calon penerima telepon.

```
01  public class CallRejectedMessage extends KadBasicMessage{
02      public static final String MSG_CALL_REJECTED = "CALL_REJECT";
03
04      PeerDescriptor peerDescriptor;
05      KademliaKey ID;
06
07      public CallRejectedMessage(PeerDescriptor pd,KademliaKey key){
08          super(MSG_CALL_REJECTED);
09          this.peerDescriptor = pd;
10          this.ID = key;
11      }
12      public PeerDescriptor getPeerDescriptor() {
13          return peerDescriptor;
14      }
15      public KademliaKey getID() {
16          return ID;
17      }
18      public String getJSONString() {
19          Gson gson = new Gson();
20          return gson.toJson(this);
21      }
22  }
23
```

Gambar 5.16 Kode Implementasi Pesan *Call Reject.*
Sumber: [Implementasi]

Pesan call finish dikirimkan ketika salah satu *peer* hendak mengakhiri percakapan mereka. Informasi yang tersimpan dalam pesan c*all finish* ini sama seperti informasi pada pesan call accept dan juga *call reject* yaitu informasi *peer* Descriptor dan juga *node* id *peer* pengirim. Hal itu dapat dilihat pada baris ke-4 dan baris ke-5. Sehingga ketika pesan ini dikirimkan, *peer* pengirim akan menutup *socket* yang digunakan dan hal tersebut juga terjadi pada *peer* penerima ketika telah menerima pesan ini.

```
01  public class FinishCallMessage extends KadBasicMessage {
02      public static final String MSG_FINISH_CALL = "FINISH_CALL";
03
04      PeerDescriptor peerDescriptor;
05      KademliaKey ID;
06
07      public FinishCallMessage(PeerDescriptor pd, KademliaKey key) {
08          super(MSG_FINISH_CALL);
09          this.peerDescriptor = pd;
10          this.ID = key;
11      }
12
13      public PeerDescriptor getPeerDescriptor() {
14          return peerDescriptor;
15      }
16
17      public KademliaKey getID() {
18          return ID;
19      }
20      public String getJSONString() {
21          Gson gson = new Gson();
22          return gson.toJson(this);
23      }
24  }
```

Gambar 5.17 Kode Implementasi Pesan *Call Finish*.
Sumber: [Implementasi]

5.2.2 Implementasi *KBucket*

Dalam pembahasan perancangan *KBucket* terdapat beberapa inti bahasan yang harus diimplementasikan. Yang diimplementasikan dalam fungsi *KBucket* ini meliputi penerapan *Triplet* sebagai media penyimpanan informasi dan *KBucket* sebagai wadahnya, kemuadian penerapan proses pemetaan *Triplet* kedalam *KBucket*, serta membuat *Cache* sebagai media salinan dari isi *KBucket.* Untuk lebih jelasnya dapat dijelaskan pada sub-sub bagian berikut.

5.2.2.1 Implementasi *Triplet* dan *KBucket.*

Informasi-informasi dari sebuah *peer* tetangga disimpan dalam sebuah kelas *Triplet.* Kelas *Triplet* ini berisikan informasi-informasi mengenai alamat *ip, udp port, node id, call id* dan *ip call.* Hal ini dapat dilihat pada kode implementasi kelas *Triplet* pada baris ke-3 hingga baris ke-7. Saat kelas *Triplet* diinstansiasi data-data informasi terkait *peer* tentangga disimpan dalam kelas ini. Hal itu dapat dilihat pada baris 9 sampai dengan 15. Untuk mengambil informasi yang tersimpan dalam kelas *Triplet* ini dapat memanfaatkan metode get yang telah diterapkan. Metode-metode get yang dapat digunakan meliputi *getIpAddress, getUdpPort, getNodeID* dan *getCallID* dapat dilihat pada baris 16 hingga baris ke-32.

```
01  public class Triplet {
02
03     private String ipAddress;
04     private int udpPort;
05     private KademliaKey nodeID;
06     private String callID;
07     private String ipCall;
08
09     public Triplet(String ipAddress, int udpPort, KademliaKey ID) {
```

```
10          this.ipAddress = ipAddress;
11          this.udpPort = udpPort;
12          this.nodeID = ID;
13          this.callID = ipAddress.substring(0,ipAddress.indexOf("@"));
14          this.ipCall = ipAddress.substring(ipAddress.indexOf("@")+1);
15      }
16      public String getIpAddress() {
17          return ipAddress;
18      }
19      public int getUdpPort() {
20          return udpPort;
21      }
22      public KademliaKey getNodeID() {
23          return nodeID;
24      }
25      public String getIpCall(){
26          this.ipCall = ipAddress.substring(ipAddress.indexOf("@")+1);
27          return ipCall;
28      }
29      public String getCallID(){
30          this.callID = ipAddress.substring(0,ipAddress.indexOf("@"));
31          return callID;
32      }
33  }
34
```

Gambar 5.18 Kode Implementasi pada kelas *Triplet.*
Sumber: [Implementasi]

KBucket merupakan wadah penyimpanan kelas *Triplet* hal ini dapat dilihat pada gambar implementasi *KBucket* baris 1 pada **gambar 5.19**. Pada baris 1 dan 2 juga menjelaskan bahwa dalam penerapan *KBucket* menggunakan *array list* dengan panjang 160 ruang penyimpanan. Untuk mengetahui jumlah data yang tersimpan dalam *KBucket* dapat menggunakan metode *getKBucketSize()* dan untuk mengambil semua data yang tersimpan dalam *KBucket* dapat memanfaatkan fungsi *getBuckets()* seperti pada baris 7 hingga 9.

```
01  private static ArrayList<Triplet> KBucket;
02  final private int MAX_BUCKET_SIZE = 160;
03
04  public KademliaPeer(String pathConfig, String key) {
05      KBucket = new ArrayList<>(MAX_BUCKET_SIZE);
06  }
07  public ArrayList<Triplet> getBuckets() {
08      return this.KBucket;
09  }
10  public int getKBucketSize(){
11      return KBucket.size();
12  }
13
```

Gambar 5.19 Kode Implemetasi *KBucket.*
Sumber: [Implementasi]

5.2.2.2 Implementasi Pemetaan *Triplet ke* Dalam *KBucket.*

Pada mekanisme pemetaan *Triplet* kedalam *KBucket* terdapat 3 hal penting yang harus diterapkan. 3 hal tersebut berupa langkah-langkah yang dilakukan ketika dilakukan penambahan *Triplet* ke dalam *KBucket*, pemindahan *Triplet* kebagian *tail KBucket*, dan yang terakhir ialah mekanisme penghapusan *Triplet* pada bagian *head KBucket*. Untuk implementasinya dapat dilihat pada

gambar 5.20 diibawah ini. Dalam gambar tersebut diambil potongan-potongan kode program yang terkait dalam pembuatan dan proses-prose yang terjadi dalam *KBucket*.

```
01  public void addNewNode(PeerDescriptor pd, KademliaKey key) {
02    String contactAddress = pd.getContactAddress();
03    int colonPos = contactAddress.indexOf(':');
04    String sipAddress = contactAddress.substring(0, colonPos);
05    int port = Integer.parseInt(
06               contactAddress.substring(colonPos + 1));
07    Triplet t = new Triplet(sipAddress, port, key);
08    for(Triplet Triplet : KBucket){
09       if(Triplet.getNodeID().getBitString()
10             .equals(t.getNodeID().getBitString())) {
11          found=true;
12          break;
13       }
14    }
15    if(found){
16       moveToTail(KBucket,t);
17    }else{
18       if(KBucket.size()<MAX_BUCKET_SIZE){
19          KBucket.add(t);
20       }else{
21          this.Triplet = t;
22          String address = KBucket.get(0).getIpAddress()
23             +":"+KBucket.get(0).getUdpPort();
24          pingBucket(address);
25       }
26
27    }
28  }
29  private void moveToTail(ArrayList<Triplet> list, Triplet t) {
30     for(int i=0; i<list.size(); i++) {
31        if(list.get(i).getNodeID().getBitString()
32                  .equals(t.getNodeID().getBitString())) {
33           list.remove(i); //menghapus list posisi sekarang
34           list.add(t);       }
35     }
36  }
37  private void pingBucket(String address){
38     BucketPingMessage bpm =
39                 new BucketPingMessage(PeerDescriptor, getKadKey());
40     sendMessage(new Address(address), new Address(address),
41                 getAddress(), bpm.getJSONString(),
42                 "application/json");
43  }
44  private void removeBucket(String address){
45     int colonPos = address.indexOf(':');
46     String ipAddress = address.substring(0, colonPos);
47     int udpPort = Integer.parseInt(address.substring(colonPos+1));
48     for(Triplet t : KBucket) {
49        if(t.getIpAddress().equals(ipAddress) &&
50           t.getUdpPort() == udpPort) {
51                KBucket.remove(t);
52                KBucket.add(addnode);
53                }
54           }
55        }
56  protected void onReceivedJSONMsg(JsonObject peerMsg,
57  Address sender) {
58     String messageType = peerMsg.get("type").getAsString();
59     if (messageType.equals(JoinNodeMessage.MSG_JOIN_NODE)){
60
```

```
61            Gson gson = new Gson();
62            PeerDescriptor pd = gson.fromJson(
63                  peerMsg.get("peerDescriptor").toString(),
64                  PeerDescriptor.class);
65            KademliaKey key = gson.fromJson(peerMsg.get("ID").
66                  toString(), KademliaKey.class);
67            this.addNewNode(pd, key);
68         }
69      els if(messageType.equals(StoreNodeMessage.MSG_FOUND_NODE)){
70         if(recievedFoundMessage){
71            notfound=false;
72            Gson gson = new Gson();
73            PeerDescriptor pd = gson.fromJson(
74                        peerMsg.get("peerDescriptor").toString(),
75                        PeerDescriptor.class);
76            KademliaKey key = gson.fromJson(peerMsg.get("key")
77                        .toString(), KademliaKey.class);
78            addNewNode(pd, key);
79         }
80      }
81   protected void onDeliveryMsgFailure(String peerMsgSended,
82                           Address receiver, String contentType) {
83      JsonParser parser = new JsonParser();
84      JsonObject msg = (JsonObject) parser.parse(peerMsgSended);
85      String msgType = msg.get("type").getAsString();
86
87      if(msgType.equals(BucketPingMessage.MSG_PING_BUCKET)){
88         removeBucket(receiver.getURL());
89      }
90   }
```

Gambar 5.20 Kode Implementasi Pemetaan *Triplet* ke Dalam *KBucket.*
Sumber: [Implementasi]

Pada saat pesan *join* dan *store* diterima maka kemudian terjadi penambahan *node* kedalam *KBucket.* Hal tersebut diterapkan pada baris ke-56 hingga baris ke-80. Untuk proses penambahan *node* ini dapat dilihat pada baris ke-1 hingga baris ke-28. Penambahan *node* diawali dengan proses pemeriksaan informasi yang tersimpan dalam *Triplet,* hal ini dapat dilihat pada baris 8 s/d 14. Dari hasil pemeriksaan itu kemudian diketahui bahwa *Triplet* tersebut merupakan *Triplet* baru ataukah lama. Jika *Triplet* tersebut merupakan *Triplet* lama maka akan dipindahkan kebagian *tail KBucket* sesuai kode pada baris ke 16. Dan jika *Triplet* tersebut merupakan *Triplet* baru maka proses dilanjutkan dengan pemeriksaan ketersediaan ruang penyimpanan dalam *KBucket*. Saat ruang masih tersedia, *Triplet* baru kemudian disimpan dan ditempatkan pada bagian *tail KBucket.* hal ini dapat dilihat pada baris ke-19.

Saat *KBucket* dalam kondisi penuh, proses penambahan *Triplet* harus melalui mekanisme pemeriksaan keaktifan *node* terlebih dahulu. *Node* yang diperiksa merupakan *node* yang berada pada bagian *head KBucket,* proses ini dapat dilihat pada baris ke-21 hingga baris ke-23. Jika pengiriman pesan tersebut gagal karena *peer* tujuan telah tidak aktif maka *Triplet* yang bersangkutan kemudian dihapus. Penghapusan *node* yang tidak aktif dapat dilihat pada baris 88. Kemudian *Triplet* yang baru ditambahkan pada bagian *tail KBucket,* sesuai kode baris ke-51 dan 52.

5.2.2.3 Implementasi *KBucket Cache.*

Cache merupakan salinan informasi dari *KBucket. Cache* memiliki fungsi yang sama dengan fungsi-fungsi yang terdapat dalam *KBucket*. Perbedaannya ialah *Cache* disimpan didalam *database sqlite*. Untuk kode penerapan *Cache* dapat dilihat pada **gambar 5.21** berikut.

```
01  public class dbContact {
02      Schema schema;
03      private final int MAX_BUCKET = 160;
04      public ArrayList<Hashtable> getData(){
05          String[] columns = {schema.UID, schema.CONTACT,
06                              schema.IP, schema.PORT};
07          Cursor cursor = db.query(schema.TBNAME, columns, null,
08                               null, null, null, schema.UID, "10");
09          ArrayList <Hashtable> contactAddress = new ArrayList();
10          try {
11              while (cursor.moveToNext()){
12                  Hashtable data = new Hashtable();
13                  data.put("id",id);
14                  data.put("contact",contact);
15                  data.put("ipaddress",ipaddress);
16                  data.put("port",port);
17                  contactAddress.add(data);
18              }
19          }catch (Exception e){
20              e.printStackTrace();
21          }
22          return contactAddress;
23      }
24      public int checkDataKontak(Hashtable contact){
25          Boolean found = false;
26          SQLiteDatabase db;
27          String kontak = contact.get("contact").toString();
28          try {
29              db = schema.getWritableDatabase();
30          }catch (NullPointerException e){
31              schema = new Schema(MainActivity.activity);
32              db = schema.getWritableDatabase();
33          }
34          return update(contact);
35      }
36      public int update(Hashtable hash){
37          Boolean found=false;
38          SQLiteDatabase db = schema.getWritableDatabase();
39          ArrayList<Hashtable> data = getData();
40          ArrayList<Hashtable> cahеData = new ArrayList<>();
41          for(int j=0; j<data.size(); j++){
42              if(hash.get("contact").toString()
43                              .equals(data.get(j).get("contact")
44                              .toString())){
45                  data.remove(j);
46                  found=true;
47                  break;
48              }
49          }
50          cahеData.add(hash);
51          cahеData.addAll(data);
52          ArrayList<ContentValues> cahedata = new ArrayList<>();
53          for(int i=0; i<=data.size(); i++){
54              ContentValues cahe = new ContentValues();
55              cahe.put(schema.CONTACT,contact);
56              cahe.put(schema.IP,ipaddress);
57              cahe.put(schema.PORT,port);
```

```
58              cahedata.add(cahe);
50          }
60          if(!found){
61              long id = db.insert(schema.TBNAME,
62                                  null,cahedata.get(0));
63          }
64          for(int j=0; j<=data.size(); j++){
65              int count = db.update(schema.TBNAME,
66                                  cahedata.get(j),schema.UID+"=?",
67                                  new String[]{1+j+""});
68              realcount +=count;
69          }
70          if(data.size()>MAX_BUCKET){
71              for (int i =MAX_BUCKET; i<data.size(); i++){
72                  delete(i);
73              }
74          }
75          return realcount;
76      }
77      public int delete(int i){
78          int count = db.delete(Schema.TBNAME,
79                                  Schema.UID + "=?",
80                                  new String[]{i +""});
81          return count;
82      }
83  }
84
```

Gambar 5.21 Kode Implementasi *KBucket Cache*
Sumber: [Implementasi]

Fungsi *update* dieksekusi ketika terjadi perubahan nilai pada *KBucket*. Fungsi *update* ini diterapkan pada baris ke-36 hingga baris ke-76. Data yang masuk kemudian ditempatkan pada bagian *tail Cache*. Jika terjadi penambahan *node* ketika *Cache* telah mencapai batas maksimum 160 maka data *Triplet* pada bagian *head* kemudian dihapus. Untuk kode implemetasinya dapat dilihat pada baris 77 hingga baris 83.

5.2.3 Implementasi Mekanisme *Bootstrapping*.

Seperti yang telah diterangkan pada bab perancangan mekanisme bootstrapping adalah mekanisme untuk mengenal informasi sebuah *peer* dengan *peer* yang lain. Dalam melakukan pengiriman pesan *bootstrapping* mekanisme ini menawarkan tiga cara alternatif. Yang pertama dengan memanfaatkan informasi pada *KBucket cache* untuk menjadi tujuan pengiriman pesan *bootstrap*. Yang kedua ialah dengan cara mengirimkan pesan *bootstrap* secara *broadcast* kepada seluruh perangkat aktif dalam jaringan *peer-to-peer*. Dan cara terakhir melakukan pengiriman pesan *join* secara *manual* oleh pengguna. Untuk Implementasi-implementasi ketiga cara tersebut dibahas pada sub bagian berikut.

5.2.3.1 Implementasi *Bootstrapping* menggunakan Informasi *KBucket* Cache.

Pada sub bagian sebelumnya telah diimplementasikan *KBucket cache.* Informasi yang tersimpan dalam *KBucket cache* tersebut dimanfaatkan dalam mekanisme *bootstrapping* pada tahap pertama ini. Dimana setiap infomasi *peer* yang tersimpan dalam KBucket cache akan dikirimkan pesan *join*. Untuk implementasi pengambilan informasi dari KBucket cache dapat dilihat pada kode

baris ke 6. Dan pengiriman pesan *join* ke setiap *peer* yang tersimpan dapat dilihat pada bari ke 9 hingga baris ke 12 dalam **gambar 5.22** berikut.

```
01
02  . . .
03
04  if(peer.getBuckets().size()==0){
05      dbContact dbcontact = new dbContact(activity);
06      ArrayList<Hashtable> data = dbcontact.getData();
07      new KeteranganTask().execute("cahe bootstrap");
08      if(data.size()!=0){
09          for (int i=0; i<data.size(); i++){
10              new JoinTask().execute(data.get(i).
11              get("ipaddress").toString()+":"+
12              data.get(i).get("port").toString());
13              new KeteranganTask().execute(data.get(i).
14              get("contact").toString()+"@"+
15              data.get(i).get("ipaddress").toString()+":"+
16              data.get(i).get("port").toString());
17          }
18      }else {
19          new Thread(new GetBroadcast(true)).start();
20      }
21
22  . . .
23
24  public void JoinNode(String address) {
25      peer.signalJoinNode(address);
26  }
27  . . .
28  public void signalJoinNode(String address) {
29     BootstrapMessage bm = new BootstrapMessage(
30                         peerDescriptor,getKadKey());
31     sendMessage(new Address(address), new Address(address),
32                 getAddress(), bm.getJSONString(),
33                 "application/json");
34  }
35  . . .
36
37  public class JoinTask extends AsyncTask<String, Void, Void> {
38      JoinInterface join;
39
40      public JoinTask(){
41          join = MainActivity.activity;
42      }
43      protected Void doInBackground(String... params) {
44          join.JoinNode(params[0]);
45          return null;
46      }
47  }
48  . . .
```

Gambar 5.22 Implementasi *join node* menggukan infomasi *KBucket cache*.
Sumber: [Implementasi]

Setiap pengiriman pesan *join* ditangani oleh kelas *JoinTask.* Untuk implementasi kodenya dapat dilihat pada baris ke 29 sampai dengan baris ke 39 sesuai sintak pada **gambar 5.22** diatas. Fungsi yang bertugas dalam mengirikan pesan *join* adalah fungsi *joinNode* metode ini bertugas dalam mengirimkan pesan *join* kepada *peer* dengan alamat tujuan yang telah didapat dari KBucket cache.

5.2.3.2 Implementasi *Bootstrapping* Menggunakan Metode *Broadcast.*

Dengan metode broadcast pesan *join* dapat dikirimkan kesemua *peer* aktif dalam jaringan *DHT Kademlia*. *Broadcast bootstrapping* diawali dengan diaktifkannya metode broadcast karena kondisi KBucket masih dalam keadaan kosong. Seperti yang terlihat pada baris 7 dan 8 pada gambar berikut.

```
01  . . .
02
03  if(peer.getBuckets().size()==0){
04  . . .
05  }else {
06      . . .
07      new Thread(new GetBroadcast()).start();
08      new Thread(new ListeningBroadcast()).start();
09      . . .
10  }
```

Gambar 5.23 implementasi Aktivasi Metode *Broadcast*.
Sumber: [Implementasi]

Proses *broadcast* dalam penerapannya menggunakan *Thread.* Dengan *Thread* proses broadcast dapat dilakukan dalam sebuah proses tersendiri sehingga tidak harus menunggu proses sebelumnya terselesaikan. Selain mengirim pesan broadcast diimplementasikan juga proses yang selalu siaga untuk menerima pesan *broadcast* proses itu bernama *ListeningBroadcast* seperti yang terlihat pada baris ke-6.

```
01  public class GetBroadcast extends Thread{
02      private WifiManager mWifi;
03      private Context context;
04      private MainActivity main;
05      private KademliaPeer peer;
06      private Boolean cahe;
07
08      public GetBroadcast(){
09          main = MainActivity.getInstance();
10          context = MainActivity.activity;
11      }
12
13      public GetBroadcast(Boolean cahe){
14          this.cahe = cahe;
15          main = MainActivity.getInstance();
16          context = MainActivity.activity;
17      }
18
19      @Override
20      public void run() {
21          try {
22              if(cahe==null){
23                  Thread.sleep(15000);            }
24              if(MainActivity.broadcast){
25                  new KeteranganTask().execute("get broadcast");
26                  final String msg= ".";
27                  String port = "1025";
28                  int pnum = Integer.parseInt(port);
29                  main.socketM3 = new DatagramSocket();
30                  main.socketM3.setBroadcast(true);
31                  DatagramPacket packet = new
32                              DatagramPacket(msg.getBytes(),
33                              msg.length(),getBroadcastAddress(),
34
```

```
35                                  pnum);
36                      main.socketM3.send(packet);
37                  }
38              } catch (InterruptedException e) {
39                  e.printStackTrace();
40              } catch (SocketException e) {
41                  e.printStackTrace();
42              } catch (IOException e) {
43                  e.printStackTrace();
44              }
45          }
46
47          private InetAddress getBroadcastAddress() throws IOException {
48              mWifi = (WifiManager)context.getSystemService
49                                      (context.WIFI_SERVICE);
50              DhcpInfo dhcp = mWifi.getDhcpInfo();
51              if (dhcp == null) {
52                  return null;
53              }
54
55              int broadcast = (dhcp.ipAddress & dhcp.netmask) |
56                              ~dhcp.netmask;
57              byte[] quads = new byte[4];
58              for (int k = 0; k < 4; k++)
59                  quads[k] = (byte) ((broadcast >> k * 8) & 0xFF);
60              return InetAddress.getByAddress(quads);
61          }
62      }
63
```

Gambar 5.24 Kode Implementasi Kelas *GetBroadcast.*
Sumber: [Implementasi]

Sebelum mengirimkan pesan broadcast sebuah *peer* harus mengetahui alamat broadcast dari jaringannya. Cara mencari alamat broadcast diimplementasikan pada fungsi *getBroadcastAddress().* Alamat groadcast didapat melalui informasi *dhcp* pada saat bergabung dalam jaringan. Fungsi tersebut dapat dilihat pada baris ke 46 hingga baris ke-61. Setelah mengetahui alamat broadcast kemudian dilanjutkan dengan pengiriman pesan *join* melalui media *broadcast*. Tetapi sebelum pesan tersebut dikirimkan proses pengiriman akan ditunda selama 15000 mili detik sejak pertama kali aplikasi dijalankan. Setelah 15000 mili detik dan KBucket masih dalam kondisi kosong maka dilanjutkan dengan proses pengiriman pesan broadcast. Dalam mengirimkan pesan melalui jaringan harus melakukan implementasi terhadap *socket*. Socket yang diterapkan adalah *socket* datagram yang menggunakan protokol udp dengan port 1025. Implementasi port dapat dilihat pada baris ke-28 hingga ke-33. Setelah implementasi *socket* kemudian paket dikirimkan melalui *socket* tersebut dapat dilihat pada baris 34.

```
01  public class ListeningBroadcast extends Thread {
02      private MainActivity main;
03      private Boolean broadcast_join = true;
04      final private int PORT = 5098;
05
06      public ListeningBroadcast(){
07          main = MainActivity.getInstance();
08      }
09
10      public void run() {
11          try {
12              final String msg=main.getLocalIpAddress();
```

```
13              final String exnomor = "085338584844123"; //15 digit
14
15              String port = "1025";
16              int pnum = Integer.parseInt(port);
17              main.socketM4=new DatagramSocket(pnum);
18
19              while (true) {
20                  byte[] bufin = new byte[exnomor.getBytes().length];
21                  DatagramPacket packet3 = new DatagramPacket(bufin,
22                                                  bufin.length);
23                  main.socketM4.receive(packet3);
24                  String recmessage = new String(packet3.
25                                                 getAddress().toString());
26                  String ip= recmessage.substring(1).trim();
27                  if (!msg.equalsIgnoreCase(ip)) {
28                      String paket = new String(packet3.getData());
29                      String nomor = paket.split("\u0000")[0];
30                      String contact = nomor+"@"+ip+":"+PORT;
31                      if(broadcast_join){
32                          new JoinTask().execute(contact);
33                          MainActivity.broadcast=false;
34                          broadcast_join = false;
35                      }
36                  }
37              }
38
39          } catch (SocketException e1) {
40              e1.printStackTrace();
41          } catch (IOException e) {
42              e.printStackTrace();
43          }
44      }
45  }
46
```

Gambar 5.25 kode Implementasi Kelas *ListeningBroadcast.*
Sumber: [Implementasi]

Penerimaan pesan broadcast ditangani oleh kelas *ListeningBroadcast*. Kelas ini selalu siaga dengan perulangan terus menerus seperti yang terlihat pada kode baris ke-19. Saat pesan broadcast diterima kemudian dilanjutkan dengan proses pengiriman pesan *join* kepada *peer* pengirim pesan. Hal ini dapat diamati pada baris kode ke- 32.

5.2.3.3 Implementasi *Bootstrapping* Dengan Cara *Manual*.

Jika cara pertama dan kedua tidak berhasil maka cara yang terakhir dalah dengan cara mengirimkan pesan *bootstrapping* secara *manual*. Syarat yang harus diketahui untuk dapat menjalankan cara ketiga ini adalah *peer* pengirim harus mengetahui alamat *ip* serta *udp port* yang digunakan oleh *peer* penerima. Sehingga dalam pengaplikasiannya pengguna akan memasukan alamat *ip* dan *udp port* setelah itu pesan *join* kemudian dikirimkan melalui alamat tersebut. Penerapannya terdapat pada baris kode ke-8 hingga 10.

```
01   . . .
02
03  if(ipBox.getText().length()!=0){
04      String number="";
05      if(numberBox.getText().length()!=0){
06          number = numberBox.getText().toString()+"@";
07
```

```
08      }
09      String contact = number+ipBox.getText().toString()+
10                       ":"+portBox.getText().toString();
11      MainActivity.peer.signalJoinNode(contact);
12      new KeteranganTask().execute("Sending join request");
13  }
14
15  . . .
16
17  public void signalJoinNode(String address) {
18     BootstrapMessage bm = new BootstrapMessage
19                               (peerDescriptor,getKadKey());
20     sendMessage(new Address(address), new Address(address),
21                              getAddress(), bm.getJSONString(),
22                              "application/json");
23  }
24  . . .
```

Gambar 5.26 Kode Implementasi *Bootstrapping* secara *Manual.*
Sumber: [Implementasi]

5.2.3.4 Implementasi Mekanisme Penerimaan Pesan *Bootstrapping*

Ketiga alternative yang telah dibahas sebelumnya adalah cara bagaimana pesan *bootstrap* dikirimkan oleh sebuah *peer*. Pesan *bootstrap* tersebut terarah pada salah satu *peer* yang telah bergabung di dalam jaringan *peer*-to-*peer*. *Peer* yang menerima pesan akan melakukan aksi tertentu terkait permintaan *bootstrapping* tersebut. Untuk implementasi kodenya dapat dilihat pada **gambar 5.27**.

```
01  . . .
02  protected void onReceivedJSONMsg(JsonObject peerMsg,Address sender){
03
04     String messageType = peerMsg.get("type").getAsString();
05     KeteranganTask keterangan = new KeteranganTask();
06
07     if(messageType.equals(BootstrapMessage.MSG_BOOTSTRAP)){
08        Gson gson = new Gson();
09        PeerDescriptor pd = gson.fromJson(peerMsg
10                 .get("peerDescriptor").toString(),
11                  PeerDescriptor.class);
12           JoinNodeMessage jnm = new JoinNodeMessage(peerDescriptor,
13                 getKadKey());
14           sendMessage(new Address(pd.getContactAddress()),
15                 new Address(pd.getContactAddress()), getAddress(),
16                 jnm.getJSONString(), "application/json");
17     }
18    else if (messageType.equals(JoinNodeMessage.MSG_JOIN_NODE)){
19        Gson gson = new Gson();
20        PeerDescriptor pd = gson.fromJson(peerMsg
21                 .get("peerDescriptor").toString(),
22                 PeerDescriptor.class);
23        KademliaKey key = gson.fromJson(peerMsg.get("ID")
24                 .toString(), KademliaKey.class);
25        this.addNewNode(pd, key);
26        MainActivity.broadcast = false;
27
28        JoinNodeSuccess jns = new JoinNodeSuccess(peerDescriptor,
29                 getKadKey());
30        sendMessage(new Address(pd.getContactAddress()),
31                 new Address(pd.getContactAddress()), getAddress(),
32
```

```
33                jns.getJSONString(), "application/json");
34        }
35        else if (messageType.equals(JoinNodeSuccess.MSG_JOIN_SUCCESS)){
36            Gson gson = new Gson();
37
38            PeerDescriptor pd = gson.fromJson(peerMsg
39                  .get("peerDescriptor").toString(),PeerDescriptor.class);
40            KademliaKey key = gson.fromJson(peerMsg
41                  .get("ID").toString(), KademliaKey.class);
42
43            this.addNewNode(pd, key);
44            MainActivity.broadcast =false;
45        }
46    }
47    . . .
```

Gambar 5.27 Implementasi Penerimaan Pesan Pada *bootstrapping*.

Sumber: [Implementasi]

Dalam **gambar 5.27** dapat dilihat tiga kondisi proses penerimaan pesan. Pada baris kode ke-7 hingga ke 17 terdapat kondisi penerimaan pesan *bootstrapping*. Ketika sebuah *peer* menerima pesan ini *peer* tersebut kemudian menanggapinya dengan pengiriman pesan *join*, hal ini dapat diamati pada baris ke-14 hingga 16.

Saat *peer* mengirimkan pesan *join* disisi lain terdapat *peer* yang menerima pesan *join* tersebut. Kemudian pesan *join* ini membangkitkan fungsi *addNewNode()* untuk menambahkan *peer* pengirim kedalam *KBucket* miliknya. Pemanggilan fungsi *addnewnode()* ini dapat dilihat pada baris ke-25. Setelah *peer* berhasil ditambahkan kemudian proses dilanjutkan dengan membalas pesan *join* tersebut dengan pesan *join success*.

Sama seperti yang terjadi saat *peer* menerima pesan *join node* pada penerimaan pesan *join success* juga membangkitkan fungsi *addNewNode().* Pembangkitan penambahan *node* pada kondisi penerimaan pesan *join success* dapat dilihat pada baris ke-42. Setelah melewati proses ini maka kedua *peer* yang terlibat dalam mekanisme *bootstrapping* ini telah mengenal satu dengan yang lainnya.

5.2.4 Implementasi Mekanisme Pencarian *Node*

Node id sebagai identitas dari masing-masing *peer* terdiri dari nilai binari. *Node id* ini disimpan dalam peubah *biginteger* dengan tipe data *BigInteger* seperti yang terlihat pada baris ke-3. Tipe data *BigInteger* dipilih karena panjang *node id* yang mencapai 160 digit. *Node id* ini diciptakan oleh kelas *Kademlia key*. Pembentukan *node id* berasal dari nomor telepon yang dienkripsi menggunakan algoritma *SHA-1.* Pengenkripsian nomor telepon tersebut dilakukan pada konstruktor *KademliaKey* yang terdapat pada baris ke-8 hingga baris 19.

```
01  public class KademliaKey {
02
03      private BigInteger bigInt;
04
05      public KademliaKey(String bitString, int nothing) {
06          this.bigInt = new BigInteger(bitString, 2);
07      }
```

```
08      public KademliaKey(Object somethingToHash) {
09          try {
10              MessageDigest md = MessageDigest.getInstance("SHA-1");
11              md.update(somethingToHash.toString().getBytes());
12              byte[] digest = md.digest();
13
14              BigInteger bi = new BigInteger(1, digest);
15              this.bigInt = bi;
16          } catch(NoSuchAlgorithmException e) {
17              this.bigInt = null;
18          }
19      }
20      public BigInteger getBigInt() {
21          return bigInt;
22      }
23      public String getBitString() {
24          StringBuilder sb = new StringBuilder();
25          byte[] bytes = this.bigInt.toByteArray();
26
27          if(bytes[0]==0) bytes = Arrays.
28                          copyOfRange(bytes, 1, bytes.length);
29
30          for(int i=0; i<bytes.length; i++) {
31              sb.append(String.format("%8s",
32              Integer.toBinaryString(bytes[i] & 255))
33              .replace(' ', '0'));
34          }
35          return sb.toString();
36      }
37      @Override
38      public String toString() {
39          return this.getBitString();
40      }
41      @Override
42      public boolean equals(Object obj) {
43          if(obj.getClass().equals(KademliaKey.class)) {
44              if(this.getBigInt().compareTo(((KademliaKey)
45                      obj).getBigInt())==0) return true;
46              else return false;
47          }
48          return false;
49      }
50      @Override
51      public int hashCode() {
52          return this.getBitString().hashCode();
53      }
54  }
55
```

Gambar 5.28 Kode Implementasi Kelas *Kademlia Key*.
Sumber: [Implementasi]

Setelah melakukan enkripsi untuk mendapatkan nilai binari hasil enkripsi tersebut kemudian dijadikan *byte array* dan dikonversi menjadi nilai binari. *Byte array* yang kosong kemudian diberi nilai 0. Sehingga hasil dari *node id* dapat berupa kombinasi nilai 1 dan 0. Implementasi tersebut dapat dilihat pada fungsi *getBitString()* dalam **gambar 5.28**.

Fungsi pencarian diterapkan pada metode *findNode.* Ketika sebuah *peer* hendak melakukan pencarian *node, peer* tersebut harus mengetahui nomor telepon *peer* tujuan. Dalam proses pencarian diawali oleh proses pencarian pada KBucket milikinya. Pencarian dalam KBucket ini dapat dilihat pada potongan

program baris ke-13 hingga baris ke-35. Proses pencarian KBucket ini membandingkan *node id* yang dicari dengan masing-masing *node id* pada *Triplet* yang telah tersimpan dalam KBucket. Implementasi perbandingan *node* id tersebut dapat dilihat pada **gambar 5.29** dalam baris ke 23 hingga baris ke 34.

```
01  public void findNode(KademliaKey ID, String nomor) {
02     keyIamLookingFor = ID;
03     findTime=System.currentTimeMillis();
04
05     this.nomor = nomor;
06     Boolean Triplet = true;
07     Hashtable<String, PeerDescriptor> noID = new Hashtable();
08     if(KBucket != null) {
09        if (containsNode(ID)) {
10           Triplet = false;
11        }
12     }
13     if(Triplet == false){
14        for (Triplet x : KBucket) {
15           PeerDescriptor pd = new PeerDescriptor(x.getCallID(),
16                          x.getIpAddress()+":"+x.getUdpPort(),
17                          x.getNodeID().getBigInt().toString());
18           String IpLooking = x.getIpAddress();
19           String noHp = IpLooking.substring(0,
20                               IpLooking.indexOf("@"));
21           noID.put(noHp, pd);
22        }
23        if (noID != null) {
24           if (noID.containsKey(nomor)) {
25              PeerDescriptor pd = noID.get(nomor);
26              String ip = pd.getAddress().
27                             split("@")[1].split(":")[0];
28              this.IpFounded = ip;
29              this.found=true;
30              pingToPeer(pd);
31           }else{
32              this.found = false;
33           }
34        }
35     }else{
36        String address = getPeerDescriptor().
37                          getAddress().split(":")[0];
38        int port = Integer.parseInt(getPeerDescriptor().
39                                    getAddress().split(":")[1]);
40
41        Triplet self = new Triplet(address,port,getKadKey());
42        ArrayList<Triplet> node = new ArrayList<>();
43        node.add(self);
44        ArrayList<Triplet> nodes =
45                        this.getClosestNodesToID(node,ID, ALPHA);
46        if(nodes.size() == 0 && nodesYetToAnswer == 0) {
47           queriedNodes.remove(keyIamLookingFor);
48           LookupCompleteMessage lcm =
49                        new LookupCompleteMessage();
50           sendMessage(new Address(
51                        peerDescriptor.getContactAddress()),
52                        new Address(
53                        peerDescriptor.getContactAddress()),
54                        getAddress(), lcm.getJSONString(),
55                        "application/json");
56        }else {
57           recievedFoundMessage=true;
58           notfound=true;
59           new Thread(new AutoNotFound()).start();
```

```
60          sendFindNodeMessage(nodes, ID);
61        }
62     }
63  }
64  private void sendFindNodeMessage(ArrayList<Triplet> list,
65                                                  KademliaKey ID){
66      String address = getPeerDescriptor().
67                       getAddress().split(":")[0];
68      int port = Integer.parseInt(getPeerDescriptor().getAddress().
69                  split(":")[1]);
70      Triplet self = new Triplet(address,port,getKadKey());
71      ArrayList<Triplet> routing = new ArrayList<>();
72      routing.add(self);
73      FindNodeMessage fnm = new FindNodeMessage(routing,routing,
74                            getPeerDescriptor(), ID);
75      for(Triplet t : list) {
76         Address toContactAddress = new Address(t.getIpAddress(),
77                                    t.getUdpPort());
78         sendMessage(toContactAddress, toContactAddress,getAddress(),
79                            fnm.getJSONString(),"application/json");
80      }
81   }
82
```

Gambar 5.29 Kode Implementasi Fungsi Pencarian *Node*.

Sumber: [Implementasi]

Saat *node id* tidak dapat ditemukan dalam KBucket, kemudian proses berlanjut dengan memilih dua *node* dalam KBucket seperti yang terlihat pada baris ke 45 yang terdapat pada **gambar 5.29**. Setelah mendapat 2 *node* terdekat kemudian *node* tersebut dijadikan tujuan pengiriman pesan *find node.* Implementasi pengiriman pesan find *node* dapat dilihat pada baris ke-60. Untuk proses pemilihan *node* sebagai penerus pencarian dalam implementasi program pada **gambar 5.30.**

```
01  private ArrayList<Triplet> getClosestNodesToID(ArrayList<Triplet>
02                       queriedNodes, final KademliaKey ID, int k) {
03     ArrayList<Triplet> list = new ArrayList<Triplet>();
04     for(Triplet t : KBucket) {
05        list.add(t);
06     }
07     Collections.sort(list, new Comparator<Triplet>() {
08        @Override
09        public int compare(Triplet t1, Triplet t2) {
10           if(Metrics.xorDistanceNumber(ID, t1.getNodeID()).
11                       compareTo(Metrics.xorDistanceNumber(ID,
12                       t2.getNodeID())) < 0) return -1;
13           if(Metrics.xorDistanceNumber(ID, t1.getNodeID()).
14                       compareTo(Metrics.xorDistanceNumber(ID,
15                       t2.getNodeID())) > 0) return 1;
16           return 0;
17        }
18     });
19     subtractTripletLists(list, alreadyQueried);
20
21     if(k < list.size()) {
22        for(int i=k; i<list.size(); i++) {
23           list.remove(i);
24        }
25     }
26     return list;
27  }
28  public static void subtractTripletLists(ArrayList<Triplet>
```

```
29                              list1, ArrayList<Triplet> list2) {
30    if(list2 != null && list2.size() > 0) {
31       for(int i=0; i<list2.size(); i++) {
32          BigInteger bigInt = list2.get(i).
33                            getNodeID().getBigInt();
34          for(int j=0; j<list1.size(); j++) {
35             if(list1.get(j).getNodeID().
36             getBigInt().compareTo(bigInt) == 0){
37             list1.remove(j);
38             }
39          }
40       }
41    }
42 }
43
44
```

Gambar 5.30 Kode Implementasi Fungsi Pemilihan *Node* Terdekat.
Sumber: [Implementasi]

Proses penentuan *node* penerus pesan pencarian diawali dengan proses pengurutan informasi KBucket berdasarkan jarak terdekat dengan *node* pencari. Proses pengurutan tersebut dapat dilihat pada potongan program baris ke 7 hingga ke 18. Proses perhitungan jarak tersebut menggunakan bantuan dari kelas Matrik yang telah diimplementasikan seperti yang terlihat pada **gambar 5.31** dibawah ini. Setelah *Triplet* dalam *KBucket* diurutkan selanjutnya dilakukan proses subtraksi pengurangan *list node*. Proses pengurangan ini berdasarkan *node* yang sudah pernah dikunjungi. Pemanggilan proses pengurangan dapat dilihat pada baris kode ke-19 dan proses substraksi secara lengkap dapat dilihat pada fungsi *substractTripletLists()*. Setelah mendapat daftar *node* yang memungkinkan untuk dikirimkan pesan *find node*. Kemudian dipilih 2 *node* dengan urutan teratas untuk meneruskan pencarian *node*.

```
01 public class Metrics {
02
03    public static String xorDistance(KademliaKey key1,
04                                      KademliaKey key2) {
05       StringBuilder sb = new StringBuilder();
06
07       for(int i=0; i<key1.getBitString().length(); i++) {
08          if(key1.getBitString().charAt(i) ==
09                   key2.getBitString().charAt(i)) sb.append('0');
10          else sb.append('1');
11       }
12       return sb.toString();
13    }
14
15    public static BigInteger xorDistanceNumber
16                   (KademliaKey key1, KademliaKey key2) {
17       String dist = xorDistance(key1, key2);
18       return new BigInteger(dist, 2);
19    }
20
21    public static int findWeight(String bitString) {
22       for(int i=0; i<bitString.length(); i++) {
23          if(bitString.charAt(i) == '1') return (159-i);
24       }
25       return 0;
26    }
27    public static int findDistanceWeight
28
```

```
29                      (KademliaKey key1, KademliaKey key2) {
30          String dist = xorDistance(key1, key2);
31          return findWeight(dist);
32      }
33
34      public static boolean isCloserToTarget(KademliaKey target,
35                                  KademliaKey key1, KademliaKey key2) {
36          BigInteger dist1 = xorDistanceNumber(key1, target);
37          BigInteger dist2 = xorDistanceNumber(key2, target);
38          if(dist1.compareTo(dist2) < 0) return true;
39          else return false;
40      }
41  }
```

Gambar 5.31 Kode Implementasi Kelas Matrik.

Sumber: [Implementasi]

Kelas matrik adalah kelas yang diimplementasikan untuk membantu pemilihan *node* dengan mode *XOR Distance.* Pada fungsi *xorDistance()* dibandingkan *node* id yang dicari dengan *node* id yang tersimpan dalam Triplet KBucket. *Node* id dengan panjang 160 *byte* tersebut dibandingkan pada masing-masing *byte*-nya. Nilai yang memiliki kesamaan nilai diberi nilai 1 dan yang berbeda mendapat nilai 0. Kemudian pada fungsi *findWeight()* nilai 1 dan 0 yang diperoleh sebelumnya kemudian dihitung nilai bobotnya. Nilai yang memiliki tingkat kesamaan yang terbanyak akan menjadi prioritas utama sebagai *node* yang terpilih.

```
01  if(messageType.equals(FindNodeMessage.MSG_FIND_NODE)) {
02      Boolean found = false;
03      System.out.println("\nReceived FIND_NODE_MESSAGE");
04
05      Gson gson = new Gson();
06      PeerDescriptor pd = gson.fromJson(peerMsg
07              .get("peerDescriptor").toString(), PeerDescriptor.class);
08      KademliaKey key = gson.fromJson(peerMsg.get("key").toString(),
09              KademliaKey.class);
10
11      Type listOfTriplets = new TypeToken<ArrayList<Triplet>>(){}
12              .getType();
13      ArrayList<Triplet> received = gson.fromJson(peerMsg.get("list"),
14              listOfTriplets);
15      ArrayList<Triplet> routing = gson.fromJson(peerMsg
16              .get("routing"), listOfTriplets);
17
18      String address = getPeerDescriptor().getAddress().split(":")[0];
19      int port = Integer.parseInt(getPeerDescriptor()
20                  .getAddress().split(":")[1]);
21      Triplet self = new Triplet(address,port,getKadKey());
22      routing.add(self);
23
24
25      for (Triplet x : KBucket) {
26          if(x.getNodeID().getBigInt().equals(key.getBigInt())){
27              System.out.println("found on bucket");
28              PeerDescriptor pdf = new PeerDescriptor(x.getCallID(),
29                  x.getIpAddress()+":"+x.getUdpPort(),
30                  x.getNodeID().getBigInt().toString());
31              StoreNodeMessage fnm =
32                  new StoreNodeMessage(routing,pdf,key);
33              sendMessage(new Address(pd.getContactAddress()),
34                  new Address(pd.getContactAddress()), getAddress(),
35
```

```
36                fnm.getJSONString(), "application/json");
37            found = true;
38         }
39      }
40
41      if(!found){
42         received.add(self);
43         ArrayList<Triplet> nodes =
44                this.getClosestNodesToID(received,key, ALPHA);
45         if(nodes.size() == 0) {
46            LookupCompleteMessage lcm = new LookupCompleteMessage();
47            sendMessage(getAddress(), new Address(getPeerDescriptor()
48                .getContactAddress()), getAddress(),
49                lcm.getJSONString(), "application/json");
50         }else {
51            System.out.println("Nodes to query: " + nodes.size());
52            FindNodeMessage fnm =
53                new FindNodeMessage(received,routing,pd, key);
54            for(Triplet t : nodes) {
55               Address toContactAddress =
56                 new Address(t.getIpAddress(), t.getUdpPort());
57               sendMessage(toContactAddress, toContactAddress,
58                etAddress(), fnm.getJSONString(), "application/json");
50            }
60         }
61      }
62   }
```

Gambar 5.32 Kode Implementasi Penangkapan Pesan *Find Node*.
Sumber: [Implementasi]

Pesan find *node* yang dikirimkan kepada dua *node* terdekat kemudian membangkitkan fungsi pencarian pada *peer* penerimanya. Pencarian yang dilakukan meliputi pencarian pada KBucket miliknya, seperti yang terlihat dalam baris ke 26 hingga 39 pada **gambar 5.32**. Saat target pencarian *node* ditemukan dalam KBucket Kemudian *peer* tersebut mengirimkan pesan *store node* kepada *peer* pencari. Proses pengiriman pesan *store node* dapat dilihat pada baris ke-34 hingga 36. Terdapat kemungkinan bahwa target pencarian *node* tidak ditemukan dalam KBucket miliknya. Jika hal ini terjadi maka proses pencarian *node* dilanjutkan dengan melakukan pemilihan dua *peer* terdekat dalam *KBucket* miliknya untuk dapat meneruskan pencarian. Seperti yang terlihat pada potongan kode baris ke 51 hingga baris ke 58.

```
01   If(messageType.equals(StoreNodeMessage.MSG_FOUND_NODE)){
02      if(recievedFoundMessage){
03         notfound=false;
04         Gson gson = new Gson();
05         PeerDescriptor pd = gson.fromJson(peerMsg
06             .get("peerDescriptor").toString(), PeerDescriptor.class);
07         KademliaKey key = gson.fromJson(peerMsg
08             .get("key").toString(), KademliaKey.class);
09         Type listOfTriplets = new TypeToken<ArrayList<Triplet>>(){}
10             .getType();
11         ArrayList<Triplet> routing = gson.fromJson(peerMsg
12             .get("routing"), listOfTriplets);
13         String routingNode="";
14         int i=0;
15         for(Triplet t : routing){
16            String temp = routingNode;
17            if(i==0){
18               routingNode=temp+t.getCallID()+" (My phone)"+"->";
```

```
19              }else{
20                 routingNode=temp+t.getCallID()+"->";
21              }
22              i++;
23          }
24          routingNode=routingNode+pd.getName();
25          addNewNode(pd, key);
26          recievedFoundMessage=false;
27          new PerformanceTask().execute("Routing :" + routingNode);
28          call = MainActivity.activity;
29          call.call(pd);
30          OnGoingCallActivity.caller=true;
31          new Thread(new AutoCanceledCall(pd)).start();
32      }
33  }
34
```

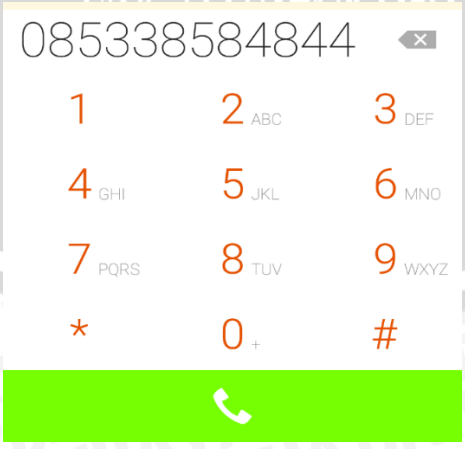
Gambar 5.33 Kode Implementasi Penangkapan Pesan *StoreNode.*
Sumber: [Implementasi]

Saat *node* yang cari ditemukan, *peer* pencari akan menerima pesan *store* dari *peer* yang telah mengenal target pencarian *node*. Pesan *store node* memungkinkan diterima oleh *peer* pencari beberapa kali karena setiap *peer* yang mengenal *node* target mengirimkan pesan *store node.* Oleh karena itu maka terdapat persyaratan bahwa *store node* pertama yang datang akan diterima dan selain itu akan ditolak. Persyaratan tersebut dapat dilihat pada **gambar 5.33** dalam baris ke-2. Pesan *store node* yang diterima kemudian membangkitkan fungsi *addNewNode()* untuk menambahkan *node* yang telah ditemukan tersebut kedalam KBucket *peer* pencari. Hal ini dapat dilihat pada baris ke-25.

5.3 Implementasi *VoIP.*

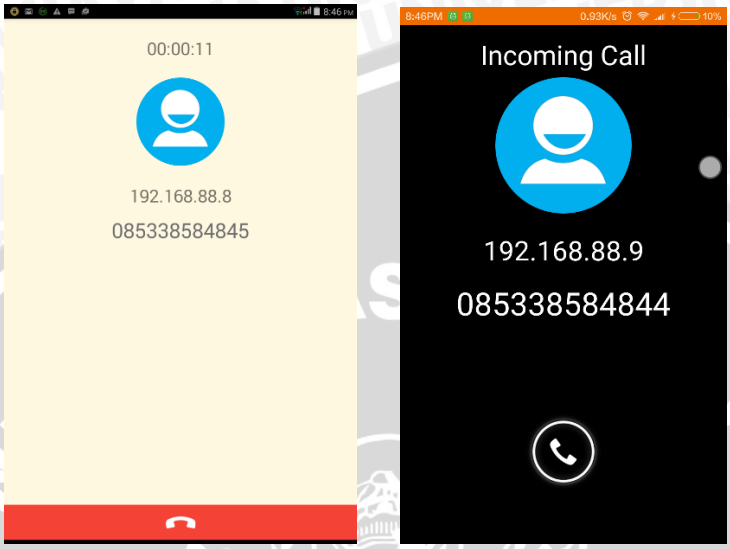
Dalam implementasi *VoIP* terdapat 3 mekanisme yang diterapkan. Mekanisme-mekanisme tersebut terdiri dari mekanisme proses pemanggilan suara, mekanisme pemilihan dan penggunaan *codec* dan mekanisme transmisi data suara.

5.3.1 Implementasi Mekanisme Panggilan Suara.



Gambar 5.34 Tampilan Pada Menu Panggilan.
Sumber: [Implementasi]

Saat *peer* penelpon hendak melakukan panggilan suara, *peer* penelpon wajib mengmasukan nomor tujuannya seperti yang tertera pada **gambar 5.34**. setelah nomor tujuan lengkap kemudian dilanjutkan dengan menekan tombol *call* yang berwarna hijau untuk melakukan panggilan.



Gambar 5.35 Tampilam Panggilan Keluar dan Masuk Pada Aplikasi *VoIP.*

[Sumber : Implementasi]

Setelah melakukan panggilan maka tampilan pada *peer* penelpon terlihat seperti pada **gambar 5.35** bagian kiri. Ketika sebuah *peer* mendapatkan permintaan panggilan masuk akan terlihat seperti pada gambar bagian kanan.

Saat terjadi mekanisme pemanggilan suara terdapat aturan umum yang harus terpenuhi. Adapun aturan tersebut ialah tidak memperkenankan sebuah *peer* untuk melakukan komunikasi ganda. Ketika *peer* penelpon menghubungi *peer* penerima, kedua *peer* tersebut tidak dapat menerima telepon dari *peer* lain. untuk implementasi aturan tersebut diimplementasikan dengan kode program seperti yang terlihat **gambar 5.36** berikut.

```
01  . . .
02  if(!onComingCall && !MainActivity.onCalling){
03      TryCallkey = gson.fromJson(peerMsg.get("ID").toString(),
04                   KademliaKey.class);
05      new CommingCall().execute(pd);
06      onComingCall = true;
07      OnGoingCallActivity.caller=false;
08  }else{
09      call = MainActivity.activity;
10      call.CallRejected(pd);
11  }
12  . . .
```

Gambar 5.36 Kode Implementasi Aturan Panggilan Suara.

Sumber: [Implementasi]

Pada baris 2 menjelaskan bahwa *peer* yang dapat menerima telepon adalah *peer* yang tidak sedang menerima telepon lain dan juga tidak aktif

melakukan panggilan suara. Ketika *peer* tujuan dalam keadaan menerima panggilan suara maka permintaan panggilan suara akan secara langsung ditolak dengan cara mengirimkan pesan *call reject*. Hal itu dapat dilihat seperti pada baris ke-10.

Dalam proses pemanggilan suara terdapat dua *peer* yang berperan aktif yaitu *peer* penelpon dan *peer* penerima. Pada masing-masing *peer* memiliki proses yang berbeda terkait panggilan suara yang dilakukannya. Sehingga implementasi panggilan suara dibedakan menjadi dua yaitu mekanisme panggilan suara pada *peer* penelpon dan mekenisme panggilan suara pada *peer* penerima.

5.3.1.1 Implementasi Pada Sisi Penelepon

Saat *peer* penelpon hendak melakukan panggilan suara, *peer* penelpon wajib memasukan nomor tujuannya seperti yang tertera pada **gambar 5.34** dan melakukan panggilan. kemudian proses pencarian *node* dijalankan untuk mencari *node* yang sesuai dengan nomor tujuan. saat terjadi prose pencarian *node* terdapat peluang bahwa *node* tujuan tidak ditemukan. Sehingga pesan *store node* juga tak kunjung diterima. Disisi penelpon masih dalam proses menunggu. Agar tidak terlalu lama menunggu dibuat mekanisme pembatasan waktu tunggu pada *peer* penelpon. Kode implementasi pembatasan waktu tunggu tersebut dapat dilihat pada kode implementasi berikut ini.

```
01  public class AutoNotFound extends Thread {
02      CallRejectedInterface reject;
03      PeerDescriptor pd;
04      long start;
05      public AutoNotFound(){
06          reject = (CallRejectedInterface)
07                   OnGoingCallActivity.activity;
08          start = System.currentTimeMillis();
09          KademliaPeer.notfound=true;
10      }
11      public void run(){
12          while (true){
13              if(KademliaPeer.notfound){
14                  if((System.currentTimeMillis() - start)>=60000){
15                      OnGoingCallActivity.notfound=true;
16                      reject.closeConnectionCall();
17                      interrupt();
18                      break;
19                  }
20              }else{
21                  interrupt();
22                  break;
23              }
24          }
25      }
26  }
27  . . .
28  @Override
29  public void closeConnectionCall() {
30      KademliaPeer.onComingCall = false;
31      MainActivity.onCalling = false;
32
33      if(MainActivity.voicestart != null){
34          try {
35              MainActivity.voicestart.join();
36              MainActivity.voicestart=null;
```

```
37              voicestop = new Thread(new VoiceStop());
38              voicestop.start();
39              voicestop.join();
40              voicestop = null;
41              MainActivity.onCalling=false;
42          } catch (InterruptedException e) {
43              e.printStackTrace();
44          }
45      }
46      . . . .
47      try {
48          tone.setPeerDescriptor(pd);
49      }catch (NullPointerException e){
50          e.printStackTrace();
51      }
52
53      try {
54          new Thread(tone).start();
55      }catch (IllegalThreadStateException e) {
56          e.printStackTrace();
57      }
58  }
59
60  . . .
61  public class ReplayTone extends Thread {
62      . . .
63      private final MediaPlayer notfoundtone = MediaPlayer.create
64                    (MainActivity.activity.getApplicationContext(),
65                     R.raw.not_found_tone);
66      . . .
67      public void stopTone(){
68          . . .
69          if(notfoundtone.isPlaying()){
70              notfoundtone.stop();
71              interrupt();
72          }
73          . . .
74      }
75      public void run(){
76          . . .
77          if(OnGoingCallActivity.notfound){
78              notfoundtone.start();
79              while (System.currentTimeMillis()-start<=11000 &&
80                            OnGoingCallActivity.notfound);
81              notfoundtone.stop();
82             Intent main = new Intent(OnGoingCallActivity.activity,
83                            MainActivity.class);
84              OnGoingCallActivity.activity.startActivity(main);
85              interrupt();
86          }
87          . .   .
88      }
89  }
90
```

Gambar 5.37 Kode Implementasi *Node* Tidak DItemukan.
Sumber: [Implementasi]

Implementasi pembatasan waktu tunggu ketika *node* tidak ditemukan ditangani oleh kelas *AutoNotFound*. Kelas ini merupakan turunan dari kelas *Thread.* Penurunan sifat *thread* diterapkan agar kelas tesebut dapat berjalan dalam sebuah proses tersendiri. Dengan menggunakan *thread* ini suatu proses dapat dieksekusi tanpa harus dipengaruhi oleh proses lain. Kelas *AutoNotFound*

dijalankan ketika melakukan proses pemanggilan. Proses pada kelas tersebut akan menunggu selama 60 detik dan ketika pesan *store* tidak kunjung diterima maka dilakukan pembatalan panggilan suara. Kode implementasinya dapat ditemukan pada baris ke 14 dan 16.

Kode baris ke 16 membangkitkan fungsi *closeConnectionCall().* Pada fungsi ini *socket* yang telah dibuka saat melakukan pemanggilan suara kemudian ditutup itu terlihat pada baris ke 33 hingga pada baris ke 45. Dan kemudian nada pemberitahuan terkait *node* yang tidak ditemukan dibangkitkan dengan cara menjalankan thread pada kelas *tone*. Hal ini dapat dilihat pada kode baris ke 54. Dengan dijalankannya thread tersebut maka nada pemberitahuanpun berbunyi. Pemutaran nada pemberitahuan dapat dilihat pada baris ke-78 dalam **gambar 5.37** diatas. Setelah nada pemberitahuan selesai diputar kemudian thread dihentikan dengan memanggil fungsi *interrupt()*.

Dalam kondisi yang berbeda pesan *store node* telah diterima oleh *peer* penelpon akan tetapi *node* tujuan telah tidak aktif. Dalam kondisi tersebut mengakibatkan *peer* penelpon menunggu panggilan yang tak kunjung terjawab. Oleh Karena hal tersebut maka dilakukan mekanisme ping call sebelum melakukan pengiriman pesan call request. Implementasi pemanggilan proses ping call dapat dilihat pada baris ke 4 dalam gambar 5.38. Ketika *node* tujuan tidak aktif dapat dipastingan pengiriman pesan *ping call* mengalami kegagalan. Kegagalan tersebut kemudian ditangkap oleh fungsi *onDeliveryMSGFailure()*. Saat mengalami kegagalan, panggilan kemudian dibatalkan dengan menjalankan fungsi *callrejected()* pada baris ke-31. Fungsi tersebut kemudian membangkitkan nada pemberitahuan bahwa *node* yang tengah dihubungi berada diluar jangkauan. Pembangkitan nada tersebut dapat dilihat pada potongan kode baris ke 46.

```
01  if(messageType.equals(StoreNodeMessage.MSG_FOUND_NODE)){
02     if(recievedFoundMessage){
03          . . .
04          pingToCall(pd);
05          . . .
06     }
07  }
08  . . .
09  @Override
10  public void pingToCall(PeerDescriptor pd) {
11     CallPingMessage kpm = new CallPingMessage(peerDescriptor,
12                           getKadKey());
13     sendMessage(new Address(pd.getAddress()),
14                           new Address(pd.getAddress()),
15                           getAddress(), kpm.getJSONString(),
16                           "application/json");
17  }
18  . . .
19  @Override
20  protected void onDeliveryMsgFailure(String peerMsgSended,
21                     Address receiver, String contentType) {
22     JsonParser parser = new JsonParser();
23     JsonObject msg = (JsonObject) parser.parse(peerMsgSended);
24     String msgType = msg.get("type").getAsString();
25  . . .
26     else if(msgType.equals(CallPingMessage.MSG_KAD_PING)){
27       if(OnGoingCallActivity.caller){
28          OnGoingCallActivity.deactiveNode = true;
```

```
29          try {
30             new CallRejected().execute();
31          } catch (InterruptedException e) {
32             e.printStackTrace();
33          }
34       }
35    }
36 . .
37 }
38 public class ReplayTone extends Thread {
39    private final MediaPlayer deactivetone = MediaPlayer.create
40                  (MainActivity.activity.getApplicationContext(),
41                  R.raw.deactive_node_tone);
42    public void run(){
43          . . .
44          if(OnGoingCallActivity.deactiveNode){
45              deactivetone.start();
46              while (System.currentTimeMillis()-start<=12000 &&
47                           OnGoingCallActivity.deactiveNode);
48              deactivetone.stop();
49              callTime();
50              Intent main = new Intent(OnGoingCallActivity.activity,
51                           MainActivity.class);
52              OnGoingCallActivity.activity.startActivity(main);
53              interrupt();
54          }
55          . . .
56       }
57 }
58
```

Gambar 5.38 Kode Implementasi *Node* Tidak Aktif.
Sumber: [Implementasi]

Kondisi selanjutnya yang dapat terjadi ialah *node* tujuan dapat dihubungi dan membalas dengan pesan *pong call.* Kemudian proses dilanjutkan dengan fungsi *call* yang menjalankan pengiriman pesan *call request*. Pembangkitan fungsi *call* terdapat pada baris ke 11 dan pengiriman pesan call request dapat dilihat pada baris ke 22 hingga 24 dalam **gambar 5.39** dibawah ini.

```
01 . . .
02 if(messageType.equals(CallPongMessage.MSG_KAD_PONG)){
03          Gson gson = new Gson();
04          PeerDescriptor pd = gson.fromJson(peerMsg
05                              .get("peerDescriptor").toString(),
06                              PeerDescriptor.class);
07          KademliaKey key = gson.fromJson(peerMsg.get("ID")
08                              .toString(), KademliaKey.class);
09          . . .
10          call = MainActivity.activity;
11          call.call(pd);
12          OnGoingCallActivity.caller=true;
13          new Thread(new AutoCanceledCall(pd)).start();
14       }
15 }
16 . . .
17 @Override
18 public void call(PeerDescriptor pd) {
19     Data codec = new Data(codectype, codecRTP);
20     CallRequestMessage crm = new CallRequestMessage(peer
21                 .getPeerDescriptor(), peer.getKadKey(), codec);
22     peer.sendMessage(new Address(pd.getAddress()),
23                 new Address(pd.getAddress()), peer.getAddress(),
24
```

```
25                  crm.getJSONString(), "application/json");
26      . . .
27      Intent i = new Intent(this,OnGoingCallActivity.class);
28      i.putExtra("pd", pd);
29      startActivity(i);
30  }
31  . . .
32  public void run(){
33      while (KademliaPeer.cancel){
34         if((System.currentTimeMillis() - start)>=20000){
35             String addressDest = pd.getAddress();
36             Address myAddress = MainActivity.peer.getAddress();
37             FinishCallMessage fcm = new FinishCallMessage
38                             (MainActivity.peer.getPeerDescriptor(),
39                              MainActivity.peer.getKadKey());
40             MainActivity.peer.sendMessage(new Address(addressDest),
41                             new Address(addressDest), myAddress,
42                             fcm.getJSONString(), "application/json");
43              OnGoingCallActivity.rejected=true;
44              reject.closeConnectionCall();
45              interrupt();
46              break;
47          }
48       }
49    }
50  }
51  . . .
52  @Override
53  public void closeConnectionCall() {
54      . . . .
55      try {
56          new Thread(tone).start();
57      }catch (IllegalThreadStateException e) {
58          e.printStackTrace();
50      }
60  }
61
```

Gambar 5.39 Kode Implementasi Fungsi *Peer* Penelpon Bagian 1.
Sumber: [Implementasi]

Setelah mengirimkan pesan *call request peer* penelpon kemudian menunggu hingga pesan balasan diterima. Permasalah terjadi ketika *peer* penerima tidak kunjung membalas pesan tersebut. Oleh karena itu dibuat mekanisme pembatalan panggilan selama 20 detik. Mekanisme pembatalan tersebut dapat dilihat pada baris ke 33 hingga baris ke 43. Dengan dijalankannya fungsi *close connection call* kemudian membangkitkan nada pemberitahuan yang menyatakan bahwa *peer* tujuan sedang dalam keadaan sibuk. Pembangkitan nada pemberitahuan dapat dilihat pada baris ke 55 seperti pada gambar 5.39.

Nada pemberitahuan sibuk, juga dijalankan saat pesan penolakan *call reject* yang diterima. Seperti yang terlihat dalam baris ke 28 pada **gambar 5.40**. Hal yang berbeda terjadi, ketika *peer* tujuan menjawab dengan pesan call accept. Diterimanya pesan ini kemudian membangkitkan fungsi thread pada kelas *voice start* seperti pada baris ke-14 sehingga komunikasi *VoIP* dapat berlangsung. Aksi-aksi yang dijalankan pada kelas voice start itu berupa membentuk *socket* pada baris ke-35. Kemudian mengaktifkan aksi perekaman suara dan pemutaran suara pada baris ke-47 dan 48.

Untuk menutup pembicaraan, salah satu *peer* dapat menekan tombol tutup panggilan. Tombol ini berwarna merah seperti yang terlihat pada **gambar 5.35** bagian kiri. Dengan ditekannya tombol tersebut kemudian fungsi *onClick* dijalankan. Fungsi ini mengatur penutupan panggilan serta pengiriman pesan *finish call* kepada lawan bicaranya. Proses penutupan panggilan dapat dilihat pada baris ke-58 dan pengiriman pesan finish terdapat pada baris ke-60 hingga 68. Saat proses penutupan panggilan kelas yang dijalankan adalah kelas *voice stop*. Kelas ini menjalankan aksi penghentian fungsi perekaman dan pemutaran suara seperti yang terlihat pada baris ke-88 dan 89.

```
01  if(messageType
02        .equals(CallAcceptedMessage.MSG_CALL_ACCEPTED)){
03        Gson gson = new Gson();
04        cancel = false;
05        if(OnGoingCallActivity.caller){
06           OnGoingCallActivity.calltime=System.currentTimeMillis();
07        }
08        OnGoingCallActivity.waittone = false;
09        PeerDescriptor pd = gson.fromJson(peerMsg
10                              .get("peerDescriptor").toString(),
11                              PeerDescriptor.class);
12        MainActivity.voicestart = new Thread(new VoiceStart
13                        (pd.getAddress().split("@")[1].split(":")[0]));
14        MainActivity.voicestart.start();
15        MainActivity.onCalling = true;
16        KademliaKey key = gson.fromJson(peerMsg.get("ID").toString(),
17                         KademliaKey.class);
18        callee = pd;
19        this.addNewNode(pd, key);
20     }
21     else if(messageType
22           .equals(CallRejectedMessage.MSG_CALL_REJECTED)){
23        Gson gson = new Gson();
24        OnGoingCallActivity.rejected=true;
25        MainActivity.onCalling=false;
26        cancel=false;
27        onComingCall = false;
28        new CallRejected().execute();
29     }
30  . . .
31  public class VoiceStart extends Thread {
32      . . .
33      public void run(){
34          try {
35              udp_socket = new DatagramSocket(56434);
36              System.out.println(udp_socket.getLocalPort());
37          } catch (SocketException e) {
38              e.printStackTrace();
39          }
40          if(main.m_iRecord==null && main.m_iPlay==null){
41              try {
42                  main.m_iRecord = new RTPAudioRecord(udp_socket,ip);
43              } catch (UnknownHostException e) {
44                  e.printStackTrace();
45              }
46              main.m_iPlay = new RTPAudioStream(udp_socket);
47              main.m_iPlay.start();
48              main.m_iRecord.start();
49              main.m_ec.setCancelling(true);
50              new Thread(MainActivity.m_ec).start();
51          }
52
```

```
53      }
54  }
55  . . .
56  @Override
57  public void onClick(View v) {
58    . . .
50    voicestop = new Thread(new VoiceStop());
60    . . .
61    if(pd !=null){
62      String addressDest = pd.getAddress();
63      Address myAddress = main.peer.getAddress();
64      FinishCallMessage fcm = new FinishCallMessage(
65              main.peer.getPeerDescriptor(), main.peer.getKadKey());
66      MainActivity.peer.sendMessage(new Address(addressDest),
67              new Address(addressDest), myAddress,
68              fcm.getJSONString(), "application/json");
69    }
70  }
71  . . .
72  if(messageType.equals(FinishCallMessage.MSG_FINISH_CALL)){
73      Gson gson = new Gson();
74      PeerDescriptor pd = gson.fromJson(peerMsg
75            .get("peerDescriptor").toString(), PeerDescriptor.class);
76      if(callee.getName().equals(pd.getName())||
77         callee.getName().equals(getPeerDescriptor().getName())) {
78         OnComeCallActivity.choosen = true;
79         MainActivity.onCalling = false;
80         onComingCall = false;
81         cancel = false;
82         OnGoingCallActivity.rejected=false;
83         new CallRejected().execute();
84      }
85  }
86  public class VoiceStop extends Thread {
87  . . .
88      public void run(){
89          main.m_iPlay.free();
90          main.m_iRecord.free();
91          main.m_ec.free();
92          if(main.m_iPlay!=null)
93              main.m_iPlay = null;
94          if(main.m_iRecord!=null)
95              main.m_iRecord = null;
96          if(main.m_ec!=null)
97              main.m_ec=null;
98      }
99  }
```

Gambar 5.40 Kode Implementasi Fungsi *Peer* Penelpon Bagian 2.
Sumber: [Implementasi]

5.3.1.2 Implementasi Pada *Peer* Penerima

Sebuah *peer* pada sistem *VoIP* dalam penelitian ini dapat dikatakan sebagai *peer* penerima panggilan ketika *peer* tersebut menerima pesan *call request* dari *peer* penelpon. Pesan call request yang datang kemudian ditangani oleh fungsi onReceivedJSONMsg(). Saat pesan ini diterima maka fungsi *coming call* kemudian dibangkitkan sperti yang terlihat pada baris ke-16. Fungsi tersebut kemudian memanggil kelas *on coming call activity.* Dan saat *kelas on coming call activity* dijalankan kelas tersebut membangkitkan nada dering sebagai tanda terdapat

panggilan masuk. Kode untuk membangkitkan nada dapat dilihat pada baris ke-29 dan 30 dalam **gambar 5.41** dibawah ini.

```
01  @Override
02  protected void onReceivedJSONMsg(JsonObject peerMsg,
03                                              Address sender) {
04     if(messageType.equals(CallRequestMessage.MSG_TRY_CALL_REQUEST)){
05        Gson gson = new Gson();
06        PeerDescriptor pd = gson.fromJson(peerMsg
07                           .get("peerDescriptor").toString(),
08                           PeerDescriptor.class);
09        Data codec = gson.fromJson(peerMsg.get("codec").toString(),
10                        Data.class);
11        MainActivity.codectype = codec.getCodec();
12        MainActivity.codecRTP = codec.getRTPcodec();
13        if(!onComingCall && !MainActivity.onCalling){
14           TryCallkey = gson.fromJson(peerMsg.get("ID")
15                        .toString(), KademliaKey.class);
16           new CommingCall().execute(pd);
17           onComingCall = true;
18           OnGoingCallActivity.caller=false;
19        }else{
20           call = MainActivity.activity;
21           call.CallRejected(pd);
22        }
23     }
24  }
25  . . .
26  public class OnComeCallActivity extends AppCompatActivity
27                 implements CallRejectedInterface, ServiceInterface {
28      . .  .
29      ringtone = new Thread(new Ringtone());
30      ringtone.start();
31      . . .
32     @Override
33      public void onTrigger(View v, int target) {
34        if(target == 0) {
35          main.CallAccepted(pd);
36          KademliaPeer.setCallee(pd); //set id caller
37          startActivity(ongoingcall);
38          choosen = true;
39          MainActivity.onCalling = true;
40          KademliaPeer.onComingCall = true;
41          MainActivity.peer.addNewNode(pd,KademliaPeer.TryCallkey);
42          try {
43              ringtone.join();
44          } catch (InterruptedException e) {
45              e.printStackTrace();
46          }
47       }else if (target == 2) {
48          main.CallRejected(pd);
49          startActivity(mainactivity);
50          choosen = true;
51          MainActivity.onCalling = false;
52          KademliaPeer.onComingCall = false;
53          try {
54              ringtone.join();
55          } catch (InterruptedException e) {
56              e.printStackTrace();
57          }
58      }
59      glowPad.reset(true);
60    }
61  . . .
62
```

```
63  }
64  public class Ringtone extends Thread{
65      Uri notification = RingtoneManager.getDefaultUri
66                         (RingtoneManager.TYPE_RINGTONE);
67      final android.media.Ringtone ringtone = RingtoneManager
68                         .getRingtone(OnComeCallActivity.activity
69                         .getApplicationContext(), notification);
70      @Override
71      public void run() {
72          while (!OnComeCallActivity.choosen){
73              ringtone.play();
74          }
75          ringtone.stop();
76          interrupt();
77      }
78  }
79  . . .
```

Gambar 5.41 Kode Implementasi Fungsi *Peer* Penerima Panggilan.
Sumber: [Implementasi]

Terdapat dua pilihan yang dapat dipilih saat terjadi panggilan. Pilihan pertama berupa penolakan dan yang kedua berupa penerimaan panggilan. Untuk penerapan pilihan tersebut dapat dilihat pada fungsi *onTriger()*. Jika pilihan yang dilakukan adalah penolakan maka kode pada baris 47 hingga 58 yang kemudian dieksekusi. Pada baris ke-48 menjelaskan bahwa panggilan ditolak dengan menjalankan fungsi *call reject.* Pada fungsi tersebut berfungsi untuk mengirimkan pesan *call reject* kepada *peer* penelpon. Dan pada baris ke-50 peubah *choosen* di beri nilai *true* sehingga nada dering kemudian dihentikan oleh kelas *thread ringtone.*

Jika pilihan yang diambil oleh penerima adalah menerima panggilan tersebut maka baris ke-34 hingga 46 yang dijalankan seperti yang terlihat pada gambar 5.41 diatas. Pada baris ke 37 pemanggilan *startActivity()* mengakibatkan tampilan aktiviti berganti pada tampilan *ongoingcall*. Dan pada baris ke-38 nada dering panggilan dihentikan. Inti dari penerimaan panggilan ini terdapat dalam baris ke-35 yaitu pemanggilan fungsi *call accept.* Untuk lebih jelasnya aksi-aksi yang dapat dijalankan oleh fungsi *call accept* dapat dilihat pada kode implementasi pada **gambar 5.42** dibawah ini.

```
01  . . .
02  @Override
03  public void CallAccepted(PeerDescriptor pd) {
04      if(OnGoingCallActivity.voicestop != null){
05          try {
06              OnGoingCallActivity.voicestop.join();
07              OnGoingCallActivity.voicestop = null;
08          } catch (InterruptedException e) {
09              e.printStackTrace();
10          }
11      }
12      if(voicestart != null){
13          try {
14              voicestart.join();
15              voicestart = null;
16          } catch (InterruptedException e) {
17              e.printStackTrace();
18          }
19
```

```
20      }
21      voicestart = new Thread(new VoiceStart(pd.getAddress()
22                  .split("@")[1].split(":")[0]));
23      voicestart.start();
24      this.onCalling = true;
25      KademliaKey key= new KademliaKey(pd.getAddress()
26                  .split("@")[0]);
27      peer.addNewNode(pd, key);
28      CallAcceptedMessage ca = new CallAcceptedMessage(peer
29                  .getPeerDescriptor(), peer.getKadKey());
30      peer.sendMessage(new Address(pd.getAddress()),
31                  new Address(pd.getAddress()), peer.getAddress(),
32                  ca.getJSONString(), "application/json");
33  }
34  . . .
```

Gambar 5.42 Kode Implementasi Fungsi *Call Accept*.
Sumber: [Implementasi]

Saat penerima bersedia melakukan komunikasi *VoiP* maka yang dijalankan adalah fungsi *call accepted*. Fungsi ini berfungsi untuk mengaktifkan fungsi yang berkaitan dengan komunikasi suara dan juga mengirimkan pesan *call accept* kepada *peer* penelpon sebagai tanda bukti kesediannya untuk melakukan komunikasi *VoIP*. Pada baris ke 22 ditunjukan bahwa fungsi yang berkaitan dengan panggilan suara dijalankan yaitu fungsi *voice start*. Dan pada baris ke-27 hingga baris ke-31 menunjukan proses pengiriman pesan *call accept* kepan *peer* penelpon agar komunikasi dapat berlangsung antar kedua *peer* tersebut.

5.3.2 Implementasi Pemilihan dan Penggunaan *Codec*

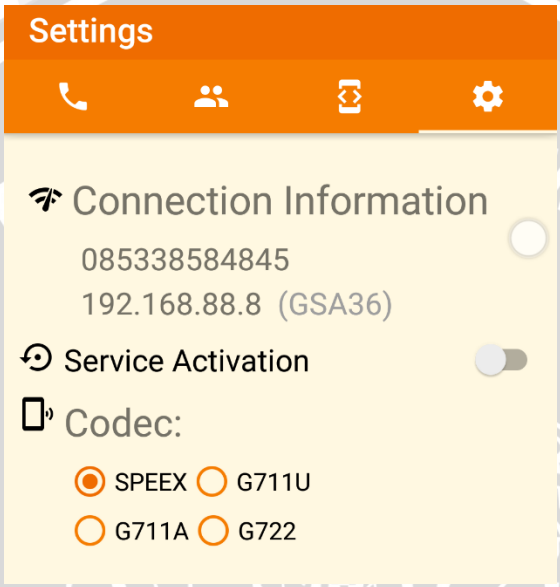
Dalam penelitian ini pengguna dibebaskan untuk memilih *codec* sesuai keperluannya. Terdapat 4 *codec* yang telah disediakan yaitu *codec* speex, g711a, g711u, g722. Untuk implementasi pemilihan *codec* dapat dilihat pada gambar 5.43 dibawah ini.

```
01  . . .
02  btn_speex.setOnTouchListener(new View.OnTouchListener() {
03      public boolean onTouch(View v, MotionEvent event) {
04          MainActivity.codectype = 1;
05          MainActivity.codecRTP = 15;
06          . . .
07      }
08  });
09  btn_g711u.setOnTouchListener(new View.OnTouchListener() {
10      public boolean onTouch(View v, MotionEvent event) {
11          MainActivity.codectype = 2;
12          MainActivity.codecRTP = 0;
13          . . .
14      }
15  });
16  btn_g711a.setOnTouchListener(new View.OnTouchListener() {
17      public boolean onTouch(View v, MotionEvent event) {
18          MainActivity.codectype = 3;
19          MainActivity.codecRTP = 8;
20          . . .
21      }
22  });
23  btn_g722.setOnTouchListener(new View.OnTouchListener() {
24      public boolean onTouch(View v, MotionEvent event) {
25          MainActivity.codectype = 4;
26          MainActivity.codecRTP = 9;
```

```
27              . . .
28          }
29      });
30      . . .
```

Gambar 5.43 Kode Implementasi Pemilihan *Codec.*
Sumber: [Implementasi]

Pada kode pada **gambar 5.43** dapat dilihat terdapat 4 aksi pada tombol yang kemudian mewakili masing-masing *codec*. Saat tombol tersebut disentuh *codec* yang digunakan akan menyesuaikannya. Untuk tampilan saat pemilihan *codec* dapat dilihat pada **gambar 5.44** berikut.



Gambar 5.44 Tampilan Menu *Settings* Untuk Memilih *Codec*.
Sumber: [Implementasi]

Saat *codec* tersebut dipilih maka ketika terjadi komunikasi *VoIP* proses *decode* dan *encode* untuk melakukan penyesuaian terhadap jenis *codec* yang dipilih oleh pengguna. Penyesuaian trsebut diimplemnetasikan pada kelas yang bernama *codec*. Isi dari kelas *codec* dapat dilihat pada **gambar 5.45** dibawah ini.

```
01  public  class Codec {
02     protected Speex speex;
03     protected ulaw g711u;
04     protected alaw g711a;
05     protected G722 g722;
06     protected static int codeccode;
07     protected final int framesize=160;
08     protected final int DEFAULT_BITRATE = 64000;
09
10     public Codec(int codeccode) {
11        this.codeccode = codeccode;
12     }
13     public void init() {
14        switch(this.codeccode){
15        case 0: break;
16        case 1:
17           speex=new Speex();
18           speex.init();
19           break;
20        case 2:
```

```
21            g711u = new ulaw();
22            g711u.init();
23            break;
24         case 3:
25            g711a = new alaw();
26            g711a.init();
27            break;
28         case 4:
29            g722 = new G722();
30            g722.init();
31            break;
32         }
33      }
34
35      public int open(int compression){
36         switch(this.codeccode){
37         case 0: return 0;
38         case 1: return speex.open(compression);
39         case 2: return 0;
40         case 3: return 0;
41         case 4: return g722.open(DEFAULT_BITRATE);
42         default: return 0;
43         }
44      }
45
46      public int getFrameSize()
47      {
48         switch(this.codeccode){
49         case 0: return framesize;
50         case 1: return speex.getFrameSize();
51         case 2: return 0;
52         case 3: return 0;
53         case 4: return 0;
54         default: return 0;
55         }
56      }
57
58      public int decode(byte[] encoded, short[] lin, int size)
50      {
60         switch(this.codeccode){
61         case 0: return size/2;
62         case 1: return speex.decode(encoded, lin, size);
63         case 2: return g711u.decode(encoded, lin, size);
64         case 3: return g711a.decode(encoded, lin, size);
65         case 4: return g722.decode(encoded, lin, size);
66         default: return 0;
67         }
68      }
69
70      public int encode(short[] lin, int offset,
71                        byte[] encoded, int size){
72         switch(this.codeccode){
73         case 0: return size*2;
74         case 1: return speex.encode(lin, offset, encoded, size);
75         case 2: return g711u.encode(lin, offset, encoded, size);
76         case 3: return g711a.encode(lin, offset, encoded, size);
77         case 4: return g722.encode(lin, offset, encoded, size);
78         default: return 0;
79         }
80      }
81
82      public void close(){
83         switch(this.codeccode){
84         case 0: break;
85         case 1: speex.close();
```

```
86              break;
87          case 4: g722.close();
88          default: break;
89          }
90      }
```

Gambar 5.45 Kode Implementasi Kelas *Codec.*
Sumber: [Implementasi]

Dari kode implementasinya setiap *codec* memiliki ciri khas masing-masing. Ciri khas setiap *codec* terdapat pada besar *framesize,* jenis *decode dan encode.* Untuk mendapat ukuran framesize dari masing-masing *codec* menggunakan fungsi *getFrameSize().* Dan untuk proses *encode* ditangani oleh fungsi *encode* dan proses *decode* ditangani oleh fungsi *decode*. Untuk inisialisasi *codec* digunakan fungsi init seperti yang terlihat pada baris ke-14 hingga baris ke-33.

Telekomunikasi suara menjadi jelas ketika *codec* yang digunakan oleh kedua *peer* merupakan *codec* yang sama. Oleh karena hal tersebut maka diimplementasikan mekanisme untuk melakukan keseragaman *codec* kepada kedua *peer* tersebut. Saat mengirimkan pesan call request dalam pesan tersebut disisipkan informasi yang digunakan oleh penelpon. Seperti yang terlihat pada kode implementasi pada **gambar 5.46.**

```
01  public class Data {
02      private int codec;
03      private int RTPcodec;
04
05      public Data(int codec, int RTPcodec){
06          this.codec = codec;
07          this.RTPcodec = RTPcodec;
08      }
09      public void setCodec(int codec){
10          this.codec = codec;
11      }
12      public int getCodec(){
13          return codec;
14      }
15      public void setRTPcodec(int RTPcodec){
16          this.RTPcodec = RTPcodec;
17      }
18      public int getRTPcodec(){
19          return RTPcodec;
20      }
21  }
22  . . .
23  public class CallRequestMessage extends KadBasicMessage {
24      public static final String MSG_TRY_CALL_REQUEST = "TRY_CALL";
25
26      . . .
27      Data codec;
28
29      public CallRequestMessage(. . ,Data codec) {
30          . . .
31          this.codec=codec;
32      }
33      . . .
34      public Data getCodec(){
35          return this.codec;
36      }
37      . . .
38  }
```

```
39  . . .
40  if(messageType.equals(CallRequestMessage.MSG_TRY_CALL_REQUEST)){
41      . . .
42      Data codec = gson.fromJson(peerMsg.get("codec").toString(),
43                  Data.class);
44      MainActivity.codectype = codec.getCodec();
45      MainActivity.codecRTP = codec.getRTPcodec();
46      . . .
47  }
48  . . .
49
```

Gambar 5.46 Kode Implementasi Penyeragaman *Codec*.
Sumber: [Implementasi]

Informasi yang dibawa dalam pesan *call request* disimpan dalam kelas yang bernama kelas data. Kelas tersebut menyimpan informasi terkait jenis *codec* yang digunakan oleh penelpon. Pada baris 1 dalam gambar 5.46 terdapat peubah dengan nama *codec*. Peubah ini berfungsi menyimpan kode nilai yang dimiliki oleh setiap *codec* kemudian kode inilah yang digunakan untuk melakukan penyesuaian terkait *codec* yang digunakan oleh penelpon. Pada baris ke-2 terdapat peubah dengan nama *codec RTP,* peubah ini digunakan untuk pemberian informasi sesuai kode *codec* yang telah diakui oleh *ITU-T*. C*odec RTP* tersebut dijadikan *codec* pengenal ketika mengirimkan paket *RTP* melalui jaringan. dengan menyertakan *codec RTP* tersebut saat melakukan penangkapan paket menggunakan aplikasi *wireshark* dapat diketahui *codec* yang digunakan beserta nama *codec* tersebut.

Untuk penyisipan informasi *codec* kedalam pesan bisa dilihat pada baris kode ke-31. Sehingga ketika penerima panggilan menerima pesan *call request*. *Peer* tersebut dapat mengenali *codec* yang digunakan penelpon dengan membaca informasi yang disematkan. Pembacaan informasi tersebut dapat dilihat pada baris kode ke-42 hingga baris ke-45.

5.3.3 Implementasi Transmisi Data

Saat terjadinya persetujuan antar dua *peer* untuk saling berkomunikasi maka proses *VoiceStart* dijalankan. Dalam proses ini terjadi penginisialisasian *socket* yang digunakan sebagai pintu masuk dari paket-paket yang dikirimkan. *Socket* yang diterapkan pada proses ini adalah *socket datagram* hal ini dapat dilihat pada baris ke-12 seperti pada **gambar 5.47**. Dalam baris tersebut dibuat sebuah *socket udp* dengan *port* 56434. Jika dilihat pada baris ke 18 terdapat inisialisasi paket *RTP* yang digunakan sebagai *socket* tujuan pengiriman data suara. Dan pada baris ke 22 merupakan inisialisasi *socket* yang digunakan sebagai jalur masuk paket dari jaringan.

```
01  public class VoiceStart extends Thread {
02      String ip;
03      DatagramSocket udp_socket;
04      MainActivity main;
05
06      public VoiceStart(String ip){
07          this.ip = ip;
08          main = MainActivity.getInstance();
09      }
10      public void run(){
11          try {
```

```
12              udp_socket = new DatagramSocket(56434);
13          } catch (SocketException e) {
14              e.printStackTrace();
15          }
16          if(main.m_iRecord==null && main.m_iPlay==null){
17              try {
18                  main.m_iRecord = new RTPAudioRecord(udp_socket,ip);
19              } catch (UnknownHostException e) {
20                  e.printStackTrace();
21              }
22              main.m_iPlay = new RTPAudioStream(udp_socket);
23              main.m_iPlay.start();
24              main.m_iRecord.start();
25          }
26      }
27  }
28
```

Gambar 5.47 Kode Implementasi Kelas *VoiceStart.*
Sumber: [Implementasi]

Pada baris ke-18 membuat instansiasi dari kelas *RTPAudioRecord*. Kelas ini merupakan kelas yang mengatur bagaimana data yang berupa suara *analog* kemudian dioalah menjadi data *digital* dan ditransmisikan melalui jaringan. Secara lengkap tahap-tahap yang dilalui dapat diamati melalui kode implementasi pada **gambar 5.47** dibawah ini.

```
01  public class RTPAudioRecord extends Thread{
02
03      private int frameRate;
04      protected android.media.AudioRecord m_in_rec;
05      protected int m_in_buf_size;
06      protected short[] m_in_bytes;
07
08      protected AudioTrack m_out_trk;
09      protected int m_out_buf_size;
10      protected byte[] m_out_bytes;
11      protected boolean m_keep_running;
12
13      protected DatagramSocket udp_socket;
14      protected DataOutputStream dout;
15      protected LinkedList<byte[]> m_in_q;
16      protected int SampleRate;
17      protected String destip;
18      protected int destport;
19      private final int framesize = 160;
20      private RTPSocket RTPSocket;
21      private final int RTPHeader = 12;
22
23      private Encoder encoder ;
24
25      public RTPAudioRecord(DatagramSocket socket, String destip)
26                                          throws UnknownHostException {
27          this.destip = destip;
28          this.destport = socket.getLocalPort();
29          SampleRate = 8000;
30          this.frameRate = SampleRate / framesize;
31          m_in_buf_size = android.media.AudioRecord
32                          .getMinBufferSize(SampleRate,AudioFormat
33                          .CHANNEL_CONFIGURATION_MONO,AudioFormat
34                          .ENCODING_PCM_16BIT);
35          m_in_rec = new android.media.AudioRecord(MediaRecorder
36                          .AudioSource.MIC,SampleRate,
37                          AudioFormat.CHANNEL_CONFIGURATION_MONO,
```

```
38                          AudioFormat.ENCODING_PCM_16BIT,
39                          m_in_buf_size*10);
40          m_in_bytes = new short [framesize];
41          m_in_q = new LinkedList<byte[]>();
42          m_keep_running = true;
43          muteflag = false;
44          udp_socket = socket;
45          RTPSocket = new RTPSocket(udp_socket, InetAddress
46                          .getByName(this.destip), this.destport);
47          encoder = new Encoder(MainActivity.codectype);
48      }
49      public void run(){
50          try{
51              int seq = 0;
52              short[] bytes_pkg = new short[framesize];
53              byte[] buffer = new byte[framesize + RTPHeader];
54              RTPPacket packet = new RTPPacket(buffer, 0);
55              packet.setPayloadType(MainActivity.codecRTP);
56              Thread encodeThread = new Thread (encoder);
57              encoder.setRecording(true);
58              encodeThread.start();
59              m_in_rec.startRecording();
60              while(m_keep_running){
61                  int bufferReadResult = m_in_rec.read(m_in_bytes, 0,
62                          framesize);
63                  bytes_pkg = m_in_bytes.clone();
64                  if(encoder.isIdle()){
65                      encoder.putData(System.currentTimeMillis(),
66                              bytes_pkg , bufferReadResult);
67                  }else {
68
69                  }
70                      if(bufferReadResult >0){
71                          if(encoder.isGetData())
72                          {
73                              byte[] temp_getdata = encoder.getData()
74                              .clone();
75                              System.arraycopy(temp_getdata, 0,
76                              buffer,packet.getHeaderLength(),
77                              temp_getdata.length);
78                              packet.setPayloadLength(temp_getdata.length);
79                              packet.setSequenceNumber(seq++);
80                              RTPSocket.send(packet);
81                          }
82                      }
83              }
84
85          }
86          catch (Exception e) {
87              e.printStackTrace();
88          }
89      }
90      public int getport(){ return udp_socket.getLocalPort();}
91  }
92
93
```

Gambar 5.48 kode Implementasi Kelas *RTPAudiRecord.*

Sumber: [Implementasi]

Pada konstruktor kelas *RTPAudiRecord* dibuat inisialisasi awal mengenai peubah-peubah yang digunakan dalam melakukan proses *encoding* data. Seperti yang terlihat pada baris ke-31 terdapat peubah yang bernama *m_in_buffer_size*. Peubah ini digunakan untuk menentukan ukuran penyangga yang diterima saat

melakukan perekaman suara dari perangkat. Kemudian pada baris ke-35 terdapat pula inisialisasi m_in_rec pada peubah ini data hasil perekaman suara disimpan. Dan pada baris 45 melakukan pembentukan soket yang digunakan sebagai pintu transmisi data. Dalam konstruktor terdapat fungsi untuk menentukan jenis *codec* yang digunakan dalam komunikasi suara. Hal ini dapat dilihat pda baris ke-47.

Kelas *RTPAudioRecord* merupakan kelas turunan dari kelas *thread.* Sehingga kelas ini dapat berjalan tanpa dipengaruhi oleh kelas lainnya. Pada fungsi *run()* melakukan pembacaan suara yang terdeteksi pada *microphone* perangkat android dan kemudian mengaktifkan fungsi *encoder*. Hal ini dapat dilihat pada baris ke- 57 dan 59. Pada baris ke-54 dibuat sebuah paket dengan berprotokol *RTP* seukuran besar nilai *buffer.* Pada baris ke 61 dilakukan pembacaan terus-menerus terkait suara yang dapat ditangkap oleh microphone perangkat. Jika *encoder* dalam posisi tersedia maka suara yang telah didapat dari proses pembacaan *microphone* kemudian diproses ke tahap *encode*. Untuk kode implementasi dapat dilihat pada baris ke 64 hingga baris ke 66. Data yang telah menyelesaikan proses *encode* kemudian dikirimkan melalui jaringan. hal ini diterapkan pada baris ke-80.

```
01  public class Encoder implements Runnable {
02
03      private volatile int leftSize = 0;
04      private final Object mutex = new Object();
05      private Codec codec ;
06      private int frameSize = 160;
07      private long ts;
08      private byte[] processedData = new byte[frameSize*2];
09      private short[] rawdata = new short[frameSize];
10      private volatile boolean isRecording;
11      protected LinkedList<byte[]> m_in_q=new LinkedList<byte[]>();
12      private EchoCancellation m_ec;
13      static public int num_send;
14
15      public Encoder(int codeccode) {
16          super();
17          codec =new Codec(codeccode);
18          codec.init();
19          frameSize = codec.getFrameSize();
20      }
21      public void run() {
22          android.os.Process.setThreadPriority(android.os
23          .Process.THREAD_PRIORITY_URGENT_AUDIO);
24          int getSize = 0;
25          while (this.isRecording()) {
26              synchronized (mutex) {
27                  while (isIdle()) {
28                      try {
29                          mutex.wait();
30                      } catch (InterruptedException e) {
31                          e.printStackTrace();
32                      }
33                  }
34              }
35              synchronized (mutex) {
36                  short output[]=rawdata.clone();
37                  this.m_ec= MainActivity.m_ec;
38                  if(m_ec!=null){
39                      if(Decoder.num_recv>0){
40                          if(num_send<20)
41                          {
```

```
42                                  num_send++;
43                              }
44                              else{
45                                  m_ec.putData(true, output,
46                                   output.length);
47                                  if(m_ec.isGetData())
48                                  {
49                                      output=m_ec.getshortData();
50                                  }
51                              }
52                          }
53                      }
54                      getSize = codec.encode(output, 0, processedData,
55                                leftSize);
56                      byte tempdata[] =new byte[getSize];
57                      System.arraycopy(processedData, 0, tempdata,
58                               0, getSize);
59                      m_in_q.add(tempdata);
60                      setIdle();
61                  }
62              }
63          }
64          public void putData(long ts, short[] data, int size) {
65              synchronized (mutex) {
66                  this.ts = ts;
67                  System.arraycopy(data, 0, rawdata, 0, size);
68                  this.leftSize = size;
69                  mutex.notify();
70              }
71          }
72          public byte[] getData(){
73              if(m_in_q.size()>1) return m_in_q.removeFirst();
74              else return m_in_q.getFirst();
75          }
76          public boolean isGetData(){ return m_in_q.size() == 0 ?false :
77                                  true;}
78          public boolean isIdle() {return leftSize == 0 ? true : false;}
79          public void setIdle() { leftSize = 0; }
80          public void setRecording(boolean isRecording) {
81              synchronized (mutex) {
82                  this.isRecording = isRecording;
83                  if (this.isRecording) {
84                      mutex.notify();
85                  }
86              }
87          }
88          public boolean isRecording() {
89              synchronized (mutex) {
90                  return isRecording;
91              }
92          }
93          public void free(){
94              num_send=0;
95              codec.close();
96          }
97      }
98
```

Gambar 5.49 Kode Implemetasi Kelas *Encoder*.
Sumber: [Implementasi]

Proses *encode* ditangani oleh kelas yang bernama *encoder*. Kelas ini mengatur data suara yang telah direkam oleh microphone menjadi data *digital* sesuai dengan *codec* yang telah disepakati kedua *peer*. Pada baris ke-54 terdapat

pemanggilan fungsi *encode* berdasarkan jenis *codec* yang telah diinisialisasi sebelumnya. Pada baris ke 72 menerapkan *method getData()* yang berfungsi untuk mengembalikan data yang telah dihasilkan oleh proses *encode* ke kelas *RTPAudioRecord.* Untuk mengurangi penggunaan memori maka setiap data yang telah diambil dan digunakan kemudian dihapuskan. Hal ini dapat dilihat pada baris ke- 73.

Kelas *RTPAudioStream* adalah kelas yang digunakan untuk menangkap paket *RTP* pada jaringan dan menjadikannya data informasi suara. Dalam kelas ini terdapat mekanisme *decode* yang berguna untuk mengembalikan data suara dari proses *encode*. Proses yang terjadi pada kelas ini merupakan kebalikan dari kelas *encoder.*

```
01  public  class RTPAudioStream extends Thread{
02     protected AudioTrack m_out_trk;
03     protected int m_out_buf_size;
04     protected byte[] m_out_bytes;
05     protected DatagramSocket udp_socket;
06     protected boolean m_keep_running;
07     protected int SampleRate;
08     protected int listenport;
09     protected LinkedList<byte[]> m_out_q;
10     private final int framesize = 320;
11     private RTPSocket RTPSocket;
12     private final int RTPHeader = 12;
13     private Decoder decoder;
14
15      public RTPAudioStream(DatagramSocket socket){
16         try{
17            SampleRate=8000;
18            m_out_buf_size = AudioTrack.getMinBufferSize(SampleRate,
19                             AudioFormat.CHANNEL_CONFIGURATION_MONO,
20                             AudioFormat.ENCODING_PCM_16BIT);
21            m_out_trk = new AudioTrack(AudioManager
22                             .STREAM_VOICE_CALL,SampleRate,
23                             AudioFormat.CHANNEL_CONFIGURATION_MONO,
24                             AudioFormat.ENCODING_PCM_16BIT,
25                             m_out_buf_size*5,
26                             AudioTrack.MODE_STREAM);
27            m_out_bytes = new byte[framesize+RTPHeader];
28            udp_socket = socket;
29            RTPSocket = new RTPSocket(udp_socket);
30            m_keep_running = true;
31            m_out_q = new LinkedList<byte[]>();
32            decoder = new Decoder(MainActivity.codectype);
33         }
34         catch (Exception e) {
35           // TODO: handle exception
36            e.printStackTrace();
37        }
38      }
39
40      public void run(){
41        RTPPacket packet = new RTPPacket(m_out_bytes,0);
42        Thread decoderthread = new Thread (decoder);
43        decoder.setPlaying(true);
44        decoderthread.start();
45        m_out_trk.play();
46
47         while(m_keep_running){
48            try{
```

```
49              RTPSocket.receive(packet);
50               byte [] bytes_pkg= packet.getPayload();
51
52               if(decoder.isIdle()){
53                 decoder.putData(System.currentTimeMillis(),
54                  bytes_pkg ,packet.getPayloadLength());
55              }else {
56              }
57               if(decoder.isGetData()==true){
58                  short[] s_bytes_pkg = decoder.getData().clone();
59                     m_out_trk.write(s_bytes_pkg, 0,
60                      s_bytes_pkg.length);}
61           }
62           catch (Exception e) {
63             e.printStackTrace();
64          }
65        }
66     }
67
68     public void free(){
69        m_keep_running=false;
70        m_out_trk.stop();
71        m_out_trk.release();
72        udp_socket.close();
73        decoder.free();
74        m_out_trk = null;
75        m_out_bytes=null;
76    }
77 }
78
79
```

Gambar 5.50 Kode Implementasi Kelas *RTPAudioStream.*
Sumber: [Implementasi]

Pada baris ke-50 dalam kelas *RTPAudioStream* menjelaskan bahwa paket *RTP* yang telah diterima kemudian diambil isi *payload* packet tersebut. *Payload packet* ini menyimpan data suara dari lawan bicaranya. Pengambilan data *payload* dapat lihat pada baris ke 51. Selanjutnya data tersebut kemudian dimasukan dalam proses *decode* saat proses *decode* dalam keadaan *idle.* Implementasi tersebut dapat dilihat pada baris ke 55. Kemudian data hasil *decode* disimpan dalam peubah bertipe *track audio* untuk dapat diputar dan menghasilkan suara. Penyimpanan data hasil *decode* kedalam peubah *track audio* dapat diamati pada baris ke-60.

```
01 public class Decoder implements Runnable {
02
03    private volatile int leftSize = 0;
04    private final Object mutex = new Object();
05    private Codec codec;
06    private int frameSize =160;
07    private long ts;
08    private short[] processedData = new short[frameSize];
09    private byte[] rawdata = new byte[frameSize*2];
10    private volatile boolean isPlaying;
11    protected LinkedList<short[]> m_out_q=new LinkedList<short[]>();
12    private EchoCancellation m_ec;
13    static public int num_recv;
14
15    public Decoder(int codeccode) {
16       super();
17       codec = new Codec(codeccode);
```

```
18          codec.init();
19      }
20
21      public void run() {
22
23          android.os.Process.setThreadPriority(android.os
24                  .Process.THREAD_PRIORITY_URGENT_AUDIO);
25
26          int getSize = 0;
27          while (this.isPlaying()) {
28
29              synchronized (mutex) {
30                  while (isIdle()) {
31                      try {
32                          mutex.wait();
33                      } catch (InterruptedException e) {
34                          e.printStackTrace();
35                      }
36                  }
37              }
38              synchronized (mutex) {
39                  this.m_ec= MainActivity.m_ec;
40                  byte[] raw_temp = new byte[leftSize];
41                  System.arraycopy(rawdata, 0, raw_temp, 0,
42                          raw_temp.length);
43                  getSize = codec.decode(raw_temp, processedData,
44                          leftSize);
45                  short[]buffer = processedData.clone();
46                  if(m_ec!=null){ //&& m_ec.isCancelling()==true){
47                      if(num_recv<100)
48                          num_recv++;
49                      else{
50                          m_ec.putData(false, buffer, buffer.length);
51                      }
52                  }
53                  m_out_q.add(buffer);
54                  setIdle();
55              }
56          }
57      }
58      public void putData(long ts, byte[] data, int size) {
59          synchronized (mutex) {
60              this.ts = ts;
61              System.arraycopy(data, 0, rawdata, 0, size);
62              this.leftSize = size;
63              mutex.notify();
64          }
65      }
66
67      public boolean isGetData()
68      {
69          return m_out_q.size() == 0 ?false : true;
70      }
71      public short[] getData(){
72          if(m_out_q.size()>0) return m_out_q.removeFirst();
73          else return m_out_q.getFirst();
74      }
75      public boolean isIdle() {
76          return leftSize == 0 ? true : false;
77      }
78      public void setIdle() {
79          leftSize = 0;
80      }
81      public void setPlaying(boolean isPlaying) {
82          synchronized (mutex) {
```

```
83              this.isPlaying = isPlaying;
84              if (this.isPlaying) {
85                  mutex.notify();
86              }
87          }
88      }
89      public boolean isPlaying() {
90          synchronized (mutex) {
91              return isPlaying;
92          }
93      }
94      public void free(){
95          num_recv=0;
96          codec.close();
97      }
98  }
99
```

Gambar 5.51 Kode Implementasi Kelas *Decoder.*

Sumber: [Implementasi]

Pada kelas *decode* merupakan kebalikan dari kelas *encode* dimana pada kelas ini diproses dari data yang sudah di*encode* agar dapat dibaca oleh *track Audio* dari perangkat *android.*

BAB 6 PENGUJIAN DAN ANALISIS

6.1 Hasil dan Analisis Pengujian *Query Time*

6.1.1 *Join Time*

Terdapat 4 skenario percobaan untuk mengukur waktu *join* pada jaringan *DHT Kademlia*. Berikut ini merupakan hasil yang diperoleh dalam pengujian *join time*.

Tabel 6.1 *Join* Time Skenario Ke-1
Sumber: [Pengujian]

No	*Node* Asal	*Node* Tujuan	Waktu (s)
1	085338584841	085338584842	0.19
2	085338584842	085338584843	0.068
3	085338584843	085338584831	0.036
4	085338584831	085338585832	0.015
5	085338585832	085338584841	0.093
Rata-rata *Join* Time			0.0804

Pada **tabel 6.1** merupakan hasil yang didapat ketika melakukan pengujian dengan skenario satu. Pengujian tersebut menggunakan 5 *node* dan 5 kali percobaan *bootstrapping*. Dari waktu yang dihasilkan selama 5 kali melakukan mekanisme bootsrtraaping didapatkan nilai rata-rata 0.0804 detik.

Tabel 6.2 *Join* Time Skenario Ke-2.
Sumber: [Pengujian]

No	*Node* Asal	*Node* Tujuan	Waktu (s)
1	085338584841	085338584842	0.21
2	085338584842	085338584843	0.08
3	085338584843	085338584844	0.082
4	085338584844	085338585845	0.085
5	085338585845	085338584831	0.208
6	085338584831	085338584832	0.196
7	085338584832	085338584833	0.218
8	085338584833	085338584834	0.227
9	085338584834	085338585835	0.195
10	085338584835	085338584841	0.015
Rata-rata *Join* Time			0.1516

Untuk skenario kedua dihasilkan nilai sesuai dengan isi **tabel 6.2**. Pada pengujian ini melakukan mekanisme bootstrapping sebanyak 10 kali dengan 10 *node* yang tergabung dalam jaringan. Dari 10 kali percobaan tersebut didapatkan waktu rata-rata yang diperlukan melakukan proses bootrapping selama 0.1516 detik.

Tabel 6.3 *Join Time* Skenario Ke-3
Sumber: [Pengujian]

No	*Node* Asal	*Node* Tujuan	Waktu (s)
1	085338584841	085338584842	0.196
2	085338584842	085338584843	0.085
3	085338584843	085338584844	0.078
4	085338584844	085338584845	0.078
5	085338584845	085338584846	0.08
6	085338584846	085338584847	0.09
7	085338584847	085338584848	0.074
8	085338584848	085338584831	0.212
9	085338584831	085338584832	0.218
10	085338584832	085338584833	0.227
11	085338584833	085338584834	0.215
12	085338584834	085338584835	0.203
13	085338584835	085338584836	0.211
14	085338584836	085338584837	0.219
15	085338584837	085338584841	0.039
Rata-rata *Join* Time			0.14833333

Pada skenario ketiga dilakukan percobaan mekanisme bootstrapping sebanyak 15 kali percobaan dengan jumlah 15 *node* yang bergabung dalam jaringan tersebut. Waktu yang diperlukan setiap kali percobaan dapat dilihat pada isi **tabel 6.3** diatas. Ketika dirata-ratakan waktu yang dibutuhkan untuk melakukan 15 kali mekanisme bootstrapping ini adalah 0.1483333 detik.

Tabel 6.4 *Join* Time Skenario Ke-4.
Sumber: [Pengujian]

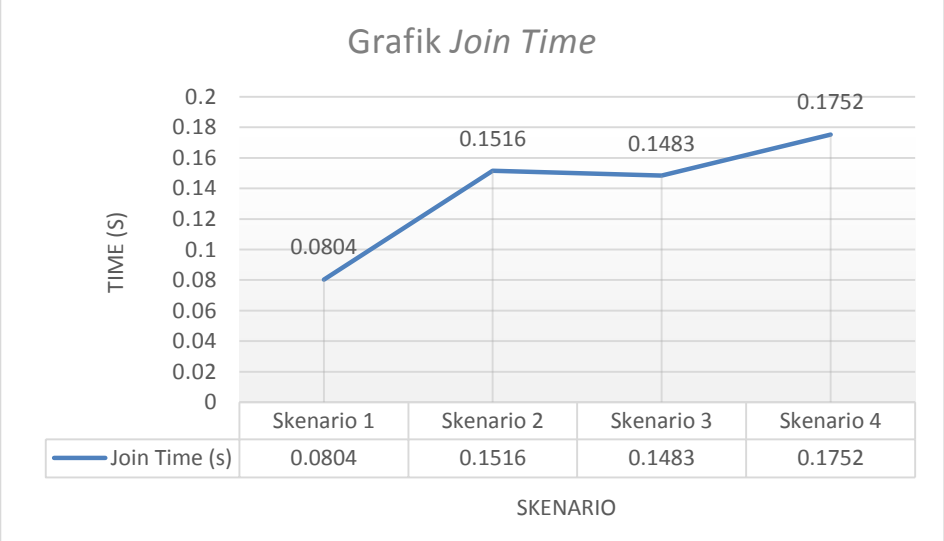
No	*Node* Asal	*Node* Tujuan	Waktu (s)
1	085338584841	085338584842	0.218
2	085338584842	085338584843	0.091
3	085338584843	085338584844	0.088
4	085338584844	085338584845	0.89
5	085338584845	085338584846	0.072
6	085338584846	085338584847	0.08
7	085338584847	085338584848	0.084
8	085338584848	085338584849	0.09
9	085338584849	085338584850	0.09
10	085338584850	085338584831	0.022
11	085338584831	085338584832	0.021
12	085338584832	085338584833	0.203
13	085338584833	085338584834	0.219
14	085338584834	085338584835	0.211
15	085338584835	085338584836	0.219

No	*Node* Asal	*Node* Tujuan	Waktu (s)
16	085338584836	085338584837	0.242
17	085338584837	085338584838	0.211
18	085338584838	085338584839	0.211
19	085338584839	085338584840	0.211
20	085338584840	085338584841	0.031
Rata-rata *Join* Time			0.1752

Dan pada skenario yang terakhir dilakukan mekanisme bootstrapping sebanyak 20 kali percobaan dengan jumlah *node* yang tergabung dalam jaringan sebanyak 20 buah. Nilai rata-rata yang diperlukan untuk mekanisme bootstrapping adalah 0.1752 detik. Dari keempat skenario yang telah dilaksanakan didapat nilai rata-rata seperti pada **tabel 6.5** dibawah ini.

Tabel 6.5 Hasil Pengukuran Proses *Join Time*.
Sumber: [Pengujian]

No	Skenario	Jumlah *Node*	Waktu *Join*(s)
1	Skenario 1	5 *Node*	0.0804
2	Skenario 2	10 *Node*	0.1516
3	Skenario 3	15 *Node*	0.1483
4	Skenario 4	20 *Node*	0.1752



Gambar 6.1 Grafik Pengujian *Join Time.*
Sumber: [Pengujian]

Dari **gambar 6.1** dapat dilihat bahwa saat skenario pertama dan kedua waktu *bootstrapping* mengalami peningkatan. Sedangkan pada skenario ketiga cenderung mengalami penurunan. Peningkatan waktu *bootstrapping* kembali dialami ketika pengujian skenario keempat. Melihat pola dari grafik tersebut dapat diambil kesimpulan bahwa proses *join node* tidak di pengaruhi oleh banyaknya *node* yang tergabung dalam jaringan *DHT Kademlia* tersebut.

6.1.2 *Find Time*

Sama halnya dengan pengujian *join time* dalam pengujian *find time* terdiri dari 4 skenario percobaan. berikut ini merupakan hasil yang diperoleh dalam pengujian *find time* pada jaringan *DHT Kademlia*.

Tabel 6.6 *Find Time* Skenario Ke-1.
Sumber: [Pengujian]

No	*Node* Asal	*Node* Tujuan	Waktu (s)
1	085338584841	085338584842	0.113
2	085338584841	085338584843	0.026
3	085338584841	085338584831	0.015
4	085338584841	085338584832	0.0165
5	085338584841	085338584833	0.022
Rata-rata Find Time			0.0385

Pada **tabel 6.6** merupakan hasil yang didapat ketika melakukan pengujian dengan skenario satu. Pengujian tersebut menggunakan 5 *node* dan 5 kali percobaan pencarian *node*. Dari waktu yang dihasilkan selama 5 kali percobaan tersebut didapatkan nilai rata-rata 0.0385 detik.

Tabel 6.7 *Find Time* Skenario Ke-2.
Sumber: [Pengujian]

No	*Node* Asal	*Node* Tujuan	Waktu (s)
1	085338584841	085338584835	0.365
2	085338584841	085338584834	0.024
3	085338584841	085338584833	0.04
4	085338584841	085338584832	0.046
5	085338584841	085338584831	0.076
6	085338584835	085338584841	0.266
7	085338584835	085338584842	0.031
8	085338584835	085338584843	0.039
9	085338584835	085338584844	0.078
10	085338584835	085338584845	0.078
Rata-rata Find Time			0.1043

Untuk skenario kedua dihasilkan nilai sesuai dengan isi **tabel 6.7**. pada skenario ini dilakukan percobaan sebanyak 10 kali dengan 10 *node* yang tergabung dalam jaringan *peer*-to-*peer*. Dari 10 kali percobaan tersebut waktu rata-rata yang diperlukan adalah 0.1043 detik.

Tabel 6.8 *Find Time* Skenario Ke-3.
Sumber: [Pengujian]

No	*Node* Asal	*Node* Tujuan	Waktu (s)
1	085338584841	085338584837	0.022

No	*Node* Asal	*Node* Tujuan	Waktu (s)
2	085338584841	085338584836	0.0173
3	085338584841	085338584835	0.08
4	085338584841	085338584834	0.084
5	085338584841	085338584833	0.038
6	085338584841	085338584832	0.022
7	085338584841	085338584831	0.002
8	085338584837	085338584841	0
9	085338584837	085338584842	0.024
10	085338584837	085338584843	0.07
11	085338584837	085338584844	0.055
12	085338584837	085338584845	0.109
13	085338584837	085338584846	0.141
14	085338584837	085338584847	0.156
15	085338584837	085338584848	0.141
Rata-rata Find Time			0.06408667

Pada skenario ketiga dilakukan percobaan pencarian *node* sebanyak 15 kali percobaan dengan jumlah 15 *node* yang bergabung dalam jaringan tersebut. Waktu yang diperlukan setiap kali percobaan dapat dilihat pada **tabel 6.8** dibawah ini. Ketika dirata-ratakan waktu yang dibutuhkan untuk melakukan 15 kali mekanisme *bootstrapping* ini adalah 0.06408667 detik.

Tabel 6.9 *Find Time* Skenario Ke-4.
Sumber: [Pengujian]

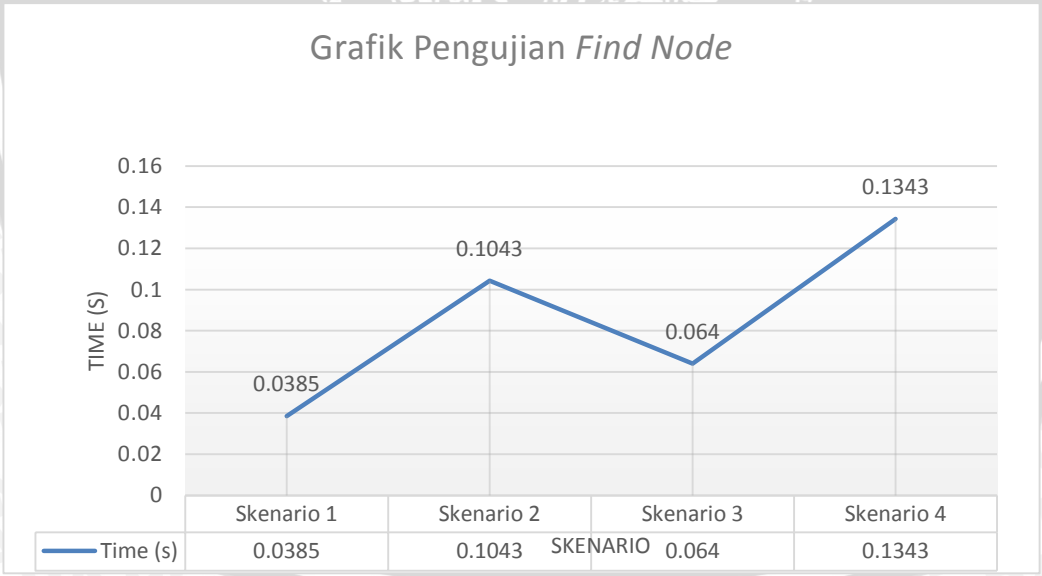
No	*Node* Asal	*Node* Tujuan	Waktu (s)
1	085338584841	085338584840	0
2	085338584841	085338584839	0.036
3	085338584841	085338584838	0.071
4	085338584841	085338584837	0.1
5	085338584841	085338584836	0.142
6	085338584841	085338584835	0.178
7	085338584841	085338584834	0.206
8	085338584841	085338584833	0.262
9	085338584841	085338584832	0.206
10	085338584841	085338584831	0.206
11	085338584831	085338584841	0.227
12	085338584831	085338584842	0.219
13	085338584831	085338584843	0.195

No	*Node* Asal	*Node* Tujuan	Waktu (s)
14	085338584831	085338584844	0.148
15	085338584831	085338584845	0.133
16	085338584831	085338584846	0.117
17	085338584831	085338584847	0.164
18	085338584831	085338584848	0.062
19	085338584831	085338584849	0.015
20	085338584831	085338584850	0
Rata-rata Find Time			0.13435

Dan pada skenario yang terakhir dilakukan pencarian *node* sebanyak 20 kali percobaan dengan jumlah *node* yang tergabung dalam jaringan sebanyak 20 buah. Nilai rata-rata yang diperlukan untuk mekanisme pencarian *node* ini adalah 0.13435 detik. Dari percobaan-percoban skenario 1 hingga 4 didapatkan nilai seperti pada **tabel 6.10** di bawah ini.

Tabel 6.10 Hasil Pengukuran Proses *Find Time.*
Sumber: [Pengujian]

No	Skenario	*Node*	Waktu (s)
1	Skenario 1	5 *Node*	0.0385
2	Skenario 2	10 *Node*	0.1043
3	Skenario 3	15 *Node*	0.064
4	Skenario 4	20 *Node*	0.1343



Gambar 6.2 Grafik Pengujian *Find Time.*
Sumber: [Pengujian]

Saat melihat grafik pengujian *find node* hasilnya tidak jauh berbeda dengan hasil pengujian *join node*. Sehingga dapat ditarik kesimpulan bahwa mekanisme *find node* juga tidak dipengaruhi oleh banyaknya jumlah *node* yang tergabung dalam jaringan *DHT Kademlia*.

6.2 Hasil dan Analisis Pengujian Skalabilitas

6.2.1 Pengujian Skalabilitas *Join Node.*

Dalam pengujian skalabilitas pada kasus *join node* dapat diketahui hasilnya seperti pada **table 6.11** dibawah ini.

Tabel 6.11 Hasil Pengujian Skalabilitas Pada Skenario *Join Node.*

Sumber: [Pengujian]

No.	*Node* Pencari	*Node* Baru	Rute *Node*	Keterangan Ditemukan
1	085338584841	085338584842	085338584841-> 085338584841	YA
2	085338584841	085338584843	085338584841-> 085338584843	YA
3	085338584841	085338584850	085338584841-> 085338584850	YA
4	085338584841	085338584844	085338584841-> 085338584842-> 085338584844	YA
5	085338584841	085338584845	085338584841-> 085338584850-> 085338584845	YA
6	085338584841	085338584846	085338584841-> 085338584843-> 085338584846	YA
7	085338584841	085338584847	085338584841-> 085338584843-> 085338584846-> 085338584847	YA
8	085338584841	085338584848	085338584841-> 085338584850-> 085338584845-> 085338584848	YA
9	085338584841	085338584849	085338584841-> 085338584842-> 085338584845-> 085338584849	YA
10	085338584841	085338584851	085338584841-> 085338584843-> 085338584846-> 085338584847-> 085338584851	YA
11	085338584841	085338584852	085338584841-> 085338584850-> 085338584845->	YA

No.	*Node* Pencari	*Node* Baru	Rute *Node*	Keterangan Ditemukan
			085338584849-> 085338584852	
12	085338584841	085338584853	085338584841-> 085338584842-> 085338584844-> 085338584847-> 085338584851-> 085338584853	YA
13	085338584841	085338584855	085338584841-> 085338584843-> 085338584846-> 085338584855	YA
14	085338584841	085338584844	085338584841-> 085338584843-> 085338584846-> 085338584855-> 085338584854	YA

Dari **table 6.12** diatas dapat disimpulkan semua *node* baru yang bergabung dalam jaringan dapat ditemukan. Dan rute yang dilalui memgambarkan alur pesan yang dilewati hingga *node* tujuan. Jika diamati lebih lanjut rute yang dilalui adalah rute terpendek dari *node* pencari ke *node* tujuan atau *node* baru pada kasus ini. Dari data pada tabel dapat ditarik kesimpulan bahwa permasalahan Bertambahnya skala *node* pada jaringan *peer-to-peer DHT Kademlia* dapat diatasi dengan baik oleh sistem *VoIP peer-to-peer* ini. Terbukti dengan setiap *node* yang baru bergabung dalam jaringan dapat ditemukan oleh *node* yang tidak saling mengenal secara langsung. Dan hal ini juga membuktikan bahwa *DHT Kademlia* memiliki skalabilitas yang tinggi pada kasus perluasan skala *node* pada jaringan *peer-to-peer*.

6.2.2 Pengujian Skalabilitas *Leave Node*.

Dari skenario yang dijalankan sebelumnya telah menghasilkan sebanyak 15 *node* yang telah terhubung pada jaringan seperti pada gambar topologi yang telah dijelaskan pada bagian prosedur pengujian. Dari topologi tersebut pada pengujian kali ini akan dilakukan pentidak aktifan *node*. Sehingga dapat diperoleh hasil seperti pada **tabel 6.12** dibawah ini.

Tabel 6.12 Hasil Pengujian Skalabilitas Leave *Node*.

Sumber: [Pengujian]

No.	Pencarian *Node*	*Node* Tidak Aktif	Rute *Node*	Keterangan Ditemukan
1	085338584841 ->	-	085338584841-> 085338584842->	

No.	Pencarian *Node*	*Node* Tidak Aktif	Rute *Node*	Keterangan Ditemukan
	085338584853		085338584844->085338584847->085338584851->085338584853	YA
2	085338584841 -> 085338584853	085338584842	085338584841->085338584843->085338584844->085338584847->085338584851->085338584853	YA
3	085338584841 -> 085338584853	085338584844	085338584841->085338584843->085338584846->085338584855->085338584854->085338584853	YA
4	085338584841 -> 085338584853	085338584854	085338584841->085338584843->085338584846->085338584847->085338584851->085338584853	YA
5	085338584841 -> 085338584853	085338584851	085338584841->085338584850->085338584845->085338584849->085338584852->085338584853	YA
6	085338584841 -> 085338584853	085338584845	085338584841->085338584843->085338584846->085338584848->085338584849->085338584852->085338584853	YA
7	085338584841 -> 085338584853	085338584857	085338584841->085338584843->085338584846->085338584848->085338584849->085338584852->085338584853	YA

No.	Pencarian *Node*	*Node* Tidak Aktif	Rute *Node*	Keterangan Ditemukan
8	085338584841 -> 085338584853	085338584850	085338584841-> 085338584843-> 085338584846-> 085338584848-> 085338584849-> 085338584852-> 085338584853	YA
9	085338584841 -> 085338584853	085338584855	085338584841-> 085338584843-> 085338584846-> 085338584848-> 085338584849-> 085338584852-> 085338584853	YA
10	085338584841 -> 085338584853	085338584849	-	TIDAK

Dari **table 6.13** diatas dapat dilihat bahwa *node* yang dinonaktifkan merupakan *node* yang tidak menjadi satu-satunya jalur penghubung antara *node* 085338584841 dengan *node* 085338584853. Sehingga ketika *node* dimatikan terdapat *node* lain yang mampu meneruskan pesan pencarian. Sehingga *node* 085338585853 dapat ditemukan melalui rute yang masih terhubung dengan *node* 085338584841. Dari hasil pada tabel 6.13 dapat disimpulkan bahwa *node* akan dapat ditemukan jika masih terdapat jalur yang tersedia antara *node* pencari dan *node* tujuan hal ini dapat dilihat pada kasus 1 hingga 9. Pada kasus no. 10 *node* yang masih terhubung adalah *node* 085338584841 -> 085338584843 -> 085338584846 -> 085338584848 -> 085338584849 -> 085338584852 -> 085338584853 sehingga ketika *node* 085338584849 dimatikan maka *node* 085338584841 tidak dapat menemukan *node* o85338584853. hal ini disebabkan karena *node* 085338584849 merupakan *node* pada jalur utama yang menghubungkan kedua *node* tersebut. Dan hal hasil tidak ada *node* lain yang dapat menggantikannya. Hal ini mengakibatkan jaringan *DHT Kademlia* tidak terhubung dengan sempurna lagi. Hal tersebut yang menyebabkan *node* 085338584853 tidak dapat ditemukan. Dengan melihat dan menganalisa hasil pengujian ini dapat diterik kesimpulan bahwa *DHT Kademlia* memiliki tingkat skalabilitas yang tinggi.

6.3 Hasil dan Analisis Pengujian *QoS*

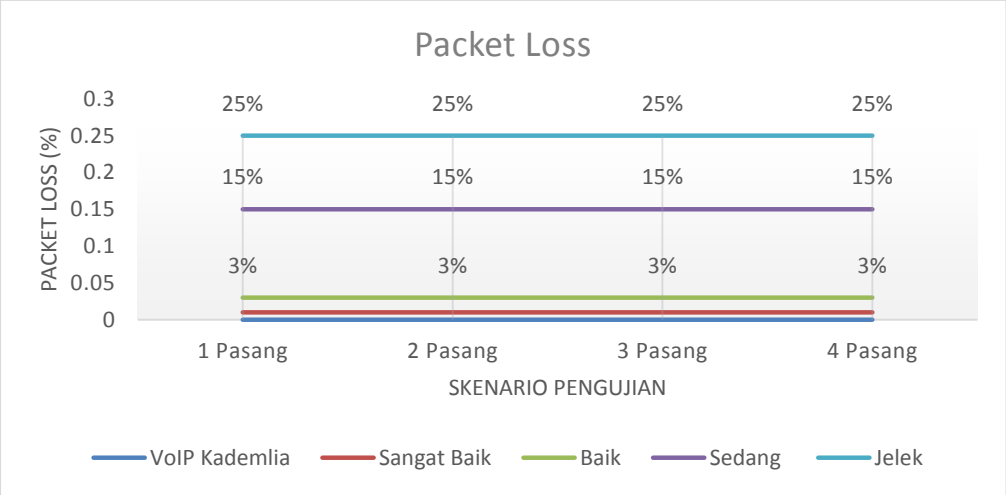
6.3.1 *Packet loss*

Tabel 6.13 Pengujian *Packet loss*
[Sumber: Pengujian]

Skenario	*Packet loss*
1 Pasang	0
2 Pasang	0
3 Pasang	0
4 Pasang	0

Tabel 6.14 Kategori *Pakect Loss* .
[Sumber: Suryawan et al., 2012]

Kategori Degradasi	*Packet loss*
Sangat Baik	0
Baik	3%
Sedang	15%
Jelek	25%



Gambar 6.3 Grafik Hasil Pengujian *Packet loss*

Sumber: [Pengujian]

Dari hasil pengujian *packet loss* didapati hasil seperti yang tertera pada **tabel 6.17** dan jika dimasukan dalam bentuk grafik akan menghasilkan keluaran grafik seperti **gambar 6.14** diatas ini. Jika diamati hasil pengujian *packet loss* pada aplikasi *VoIP Kademlia*, *packet loss* yang dihasilkan masuk kedalam ketegori sangat baik karena dari keempat skenario *packet loss* menjukan nilai 0.

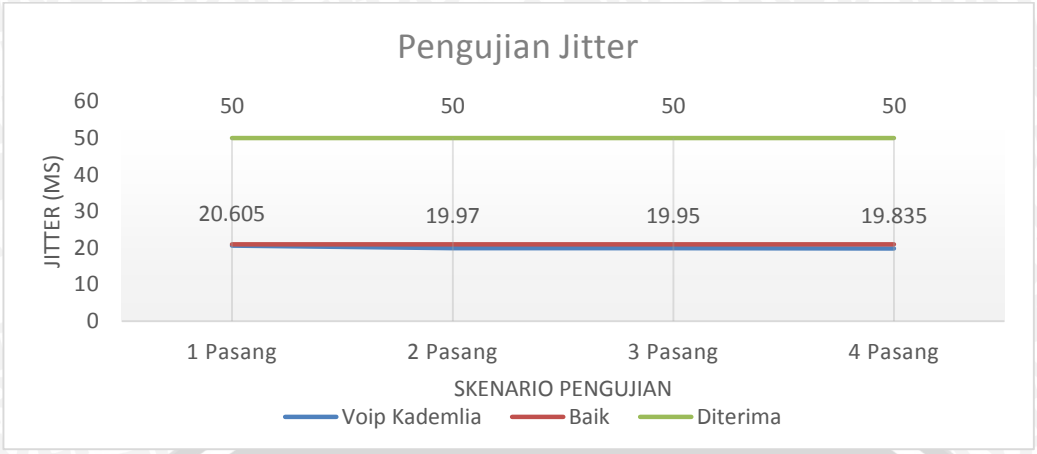
6.3.2 *Jitter*

Tabel 6.16 hasil Pengujian *Jitter.*
Sumber: [Pengujian]

Skenario	*Jitter* (ms)
1 Pasang	20.605
2 Pasang	19.97
3 Pasang	19.95
4 Pasang	19.835

Tabel 6.15 Standard ITU-T.
[Sumber: Suryawan et al., 2012]

Kategori	*Jitter* (ms)
Baik	20
Diterima	20 - 50
Buruk	> 50



Gambar 6.4 Grafik Hasil Pengujian *Jitter*.
Sumber: [Pengujian]

Dari grafik pada **gambar 6.15** dapat dilihat bahwa *jitter* pada aplikasi *VoIP Kademlia* terhimpit dengan garis berkategori Baik dalam kategori *ITU-T*. Oleh karena hal tersebut dapat diambil kesimpulan bahwa *jitter* yang dihasilkan oleh aplikasi *VoIP* kademlia memenuhi syarat dikatakan baik dalam komunikasi *VoIP* dari sudut pandang *jitter* yang dihasilkan.

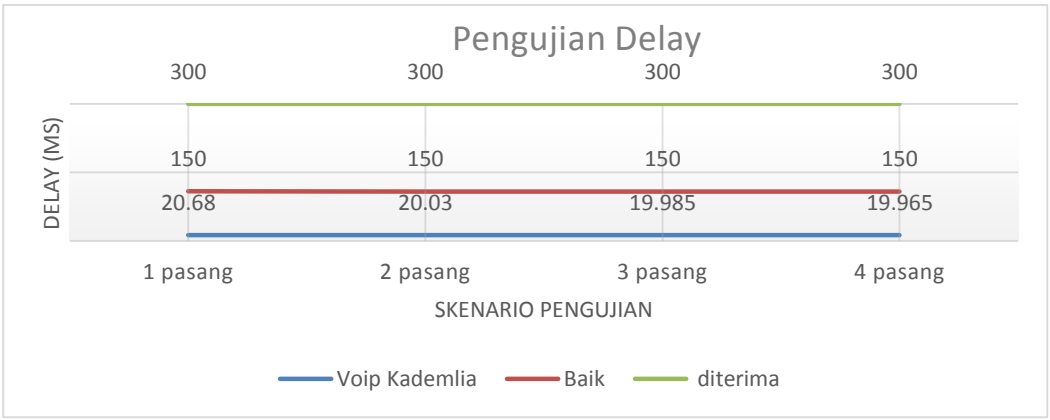
6.3.3 *Delay*

Tabel 6.17 Hasil Pengujian *Delay*
[Sumber: Pengujian]

Skenario	*Delay*
1 Pasang	20.68
2 Pasang	20.03
3 Pasang	19.985
4 Pasang	19.965

Tabel 6.18 Standard *Delay* ITU-T.
[Sumber: Suryawan et al., 2012]

Kategori	*delay* (ms)
Baik	<150
Diterima	150 s/d 300
Buruk	300 s/d 400



Gambar 6.5 Grafik Hasil Pengujian *Delay*
[Sumber: Pengujian]

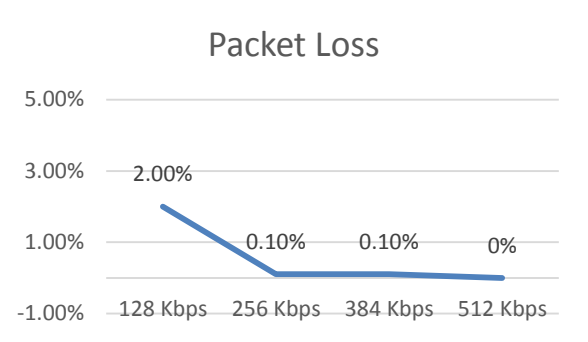
Dalam grafik pada **gambar 6.6** dapat dilihat bahwa nilai *delay* yang dihasilkan aplikasi *VoIP Kademlia* berada dibawah garis kategori baik. Oleh karena hal tersebut maka dapat diambil kesimpulan *delay* yang dihasilkan aplikasi *VoIP* memenuhi standarisasi *ITU-T* dan berkategori baik.

6.4 Hasil dan Analisis Pengujian *Bandwidth*

saat melakukan 4 kali percobaan dengan menggunakan 4 *bandwidth* yang berbeda didapatkan hasil sebagai berikut.

Tabel 6.19 Pengujian *Bandwidth* Terhadap *Packet loss.*
Sumber: [Pengujian]

Bandwidth	*Packet loss (%)*	Kategori
128 *Kbps*	2.0 %	Baik
256 *Kbps*	0.1 %	Baik
384 *Kbps*	0.1 %	Baik
512 *Kbps*	0 %	Sangat Baik

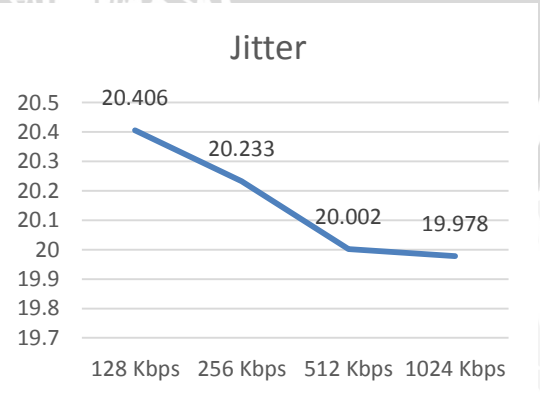


Gambar 6.6 Grafik Hasil Pengujian *Bandwidth* Terhadap *Packet loss*
Sumber: [Pengujian]

Dari grafik pengujian bandwidth terhadap packet loss dapat dilihat bahwa terjadi penurunan packet loss pada saat bandwidth mengalami peningkatan. Semakin besar bandwidth yang diberikan packet loss yang dihasilkan semakin menurun. Hal tersebut secara tidak langsung membuktikan bahwa peningkatan bandwidth yang diberikan juga meningkatakan kualitas komunikasi VoIP.

Tabel 6.20 Pengujian *Bandwidth* Terhadap *Jitter.*
Sumber: [Pengujian]

Bandwidth	*Jitter (ms)*	Kategori
128 *Kbps*	20.406	Baik
256 *Kbps*	20.233	Baik
384 *Kbps*	20.002	Baik
512 *Kbps*	19.978	Sangat Baik



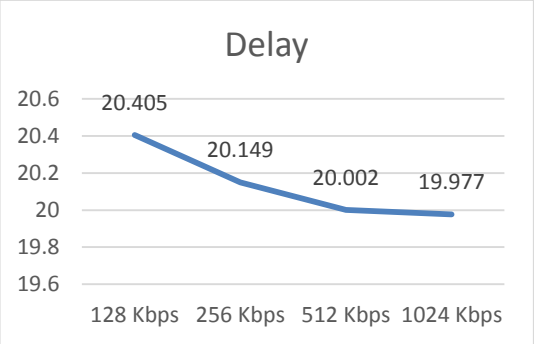
Tabel 6.21 Pengujian *Bandwidth* Terhadap *Jitter.*
Sumber: [Pengujian]

Dari grafik pengujian bandwidth terhadap jitter dapat dilihat bahwa terjadi penurunan jitter pada saat bandwidth mengalami peningkatan. Semakin besar

bandwidth yang diberikan nilai *jitter* yang dihasilkan semakin menurun. Hal tersebut secara tidak langsung membuktikan bahwa peningkatan bandwidth yang diberikan juga meningkatakan kualitas komunikasi *VoIP*. Hal tersebut juga terjadi pada parameter *delay*. Pengujian parameter *delay* dapat dilihat pada **tabel 6.25** dan **gambar 6.7** berikut.

Tabel 6.22 Pengujian *Bandwidth* Terhadap *Delay.*
Sumber: [Pengujian]

Bandwidth	*Delay (ms)*	Kategori
128 *Kbps*	20.405	Baik
256 *Kbps*	20.149	Baik
384 *Kbps*	20. 002	Baik
512 *Kbps*	19.977	Baik



Gambar 6.7 Pengujian *Bandwidth* Terhadap *Delay.*
Sumber: [Pengujian]

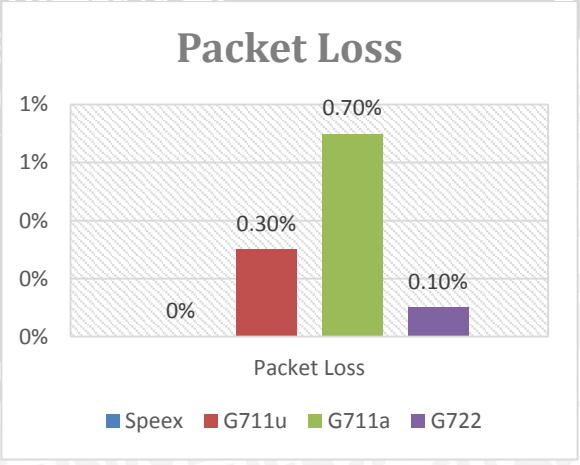
Jika dilihat dari hasil pengujian ketiga parameter *QoS* terhadap besarnya *bandwidth* yang berikan dapat ditarik sebuah kesimpulan bahwa besarnya *bandwidth* yang tersedia mempengaruhi kualitas dari komunikasi *VoIP* tersebut. Hal ini terbukti dari ketiga grafik tersebut mengalami penurunan ketika *bandwith* yang diberikan bertambah.

6.5 Hasil dan Analisis Pengujian Penggunaan *Codec*

Dari hasil pengukuran dan analisis keluaran *wireshark* didapatkan hasil sesuai tabel berikut.

Tabel 6.23 Pengujian Penggunaan Codec Terhadap Packet Loss
Sumber: [Pengujian]

Bandwidth	*Packet loss (%)*	Kategori
Speex	0 %	SangatBaik
G711u	0.3 %	Baik
G711a	0.7 %	Baik
G722	0.1 %	Baik

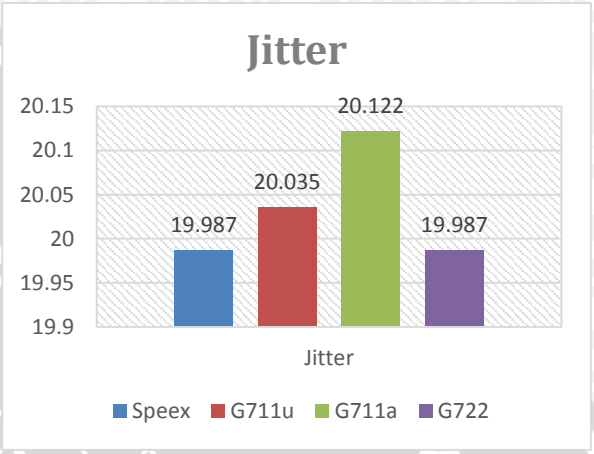


Gambar 6.8 Pengujian Penggunaan Codec Terhadap Packet Loss
Sumber: [Pengujian]

Dari grafik pengujian penggunaan *codec* terhadap packet loss dapat dilihat bahwa terjadi perbedaan kualitas packet loss pada setiap jenis codec yang digunakan. Hal tersebut secara tidak langsung membuktikan bahwa pemilihan jenis codec dapat mempengaruhi kualitas komunikasi VoIP pada parameter *packet loss*.

Tabel 6.24 Pengujian Penggunaan Codec Terhadap Jitter
Sumber: [Pengujian]

Bandwidth	*Jitter (ms)*	Kategori
Speex	19.987	Baik
G711u	20.035	diterima
G711a	20.122	diterima
G722	19.987	Baik

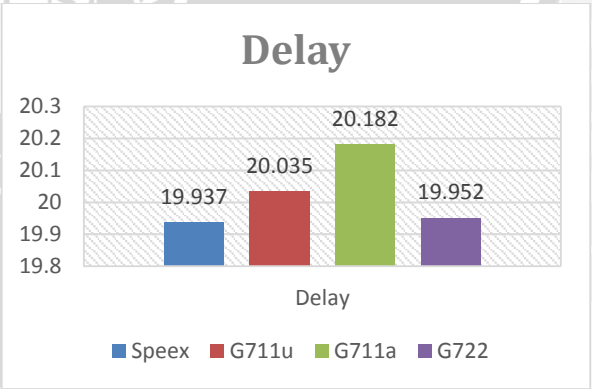


Gambar 6.9 Pengujian Penggunaan Codec Terhadap Jitter
Sumber: [Pengujian]

Dari grafik pengujian penggunaan *codec* terhadap juga dapat dilihat bahwa terjadi perbedaan kualitas jitter pada setiap jenis codec yang digunakan. Hal tersebut secara tidak langsung membuktikan bahwa pemilihan jenis codec dapat mempengaruhi kualitas komunikasi VoIP pada parameter *jitter*. Hal serupa juga dialami pada pengujian penggunaan *codec* terhadap kualitas delay yang dihasilkan. Hasil pengujian berikut dapat dilihat sebagai berikut.

Tabel 6.25 Pengujian Penggunaan Codec Terhadap Delay
Sumber: [Pengujian]

Bandwidth	*Delay (ms)*	Kategori
Speex	19.937	Baik
G711u	20.035	Baik
G711a	20.182	Baik
G722	19.952	Baik



Gambar 6.10 Pengujian Penggunaan Codec Terhadap Delay
Sumber: [Pengujian]

Jika dilihat dari hasil pengujian ketiga parameter *QoS* terhadap jenis codec yang digunakan dapat ditarik sebuah kesimpulan bahwa jenis codec yang dipilih akan mempengaruhi kualitas dari komunikasi *VoIP* tersebut. Hal ini terbukti dari

ketiga grafik tersebut setiap jenis codec menghasilkan kualistas QoS yang berbeda-beda. Hal ini membuktikan bahwa jenis codec yang digunakan menjadi faktor penting yang harus diperhitungkan ketika melakukan komunikasi VoIP.

BAB 7 PENUTUP

Berdasarkan perancangan, implementasi, pengujian, dan analisis hasil yang telah dilakukan, maka dapat diambil kesimpulan sebagai berikut.

7.1 Kesimpulan

1. *DHT Kademlia* digunakan dalam pencarian *peer* pada sistem *VoIP peer-to-peer*. Dengan menggunakan *DHT Kademlia peer* yang tidak saling mengenal dapat berkomunikasi melalui perantara *peer* tetangganya yang telah mengenal *peer* tujuan tersebut terlebih dahulu. *DHT Kademlia* melakukan pencarian *node* berdasarkan algoritma *binary search tree.* Dengan memilih 2 *node* yang telah dikenal untuk meneruskan pencarian. Hal tersebut akan terus berlanjut hingga *peer* yang dimaksud dapat ditemukan.
2. Koordinasi yang dilakukan saat melakukan komunikasi *VoIP* menggunakan mekanisme pertukaran pesan. Pesan yang dikirimkan ketika hendak melakukan panggilan adalah pesan *call request*. Pesan ini kemudian dibalas oleh *peer* penerima permintaan. Pesan balasan tersebut dapat berupa pesan *call accept* maupun *call reject.* Pesan *call accept* menandakan kesedian *peer* penerima untuk melakukan komunikasi *VoIP*. Sedangkan pesan *call reject* dikirimkan untuk menolak permintaan panggilan tersebut. Pesan yang terakhir adalah pesan *call finish*. Pesan ini dikirimkan untuk mengakhiri komunikasi *VoIP* tersebut.
3. Dari pengukuran performa sistem yang telah dibangun dapat di simpulkan sebagai berikut. (1) Performa *query time* tidak dipengaruhi oleh banyaknya *peer* yang ikut bergabung didalam jaring *DHT Kademlia*. Hal ini terbukti dengan penurunan waktu tempuh pada mekanisme pengenalan dan pencarian *node* saat jumlah *node* pada jaringan bertambah dari 10 menjadi 15 *node*. Waktu yang ditempuh oleh mekanime pencarian dengan jumlah 10 *node* adalah 0.1043s dan mengalami penurunan ketika jumlah *node* bertambah menjadi 15 *node* menghasilkan waktu temu 0.064s. (2) *DHT Kademlia* memiliki tingkat skalabilitas yang tinggi. Hal ini dibuktikan dengan pengujian *join node* dan *leave node*, keseluruhan *node* yang dicari tetap dapat ditemukan walaupun mengalami perubahan pada jumlah *node* yang tergabung dalam jaringan DHT. (3) Kualitas *VoIP* yang dihasilkan telah memenuhi standar komunikasi yang ditetapkan oleh *ITU-T* dengan kualitas baik. Hal ini didapat dari nilai *delay*, *jitter* dan *packet loss* masuk dalam jangkauan nilai dari parameter yang ditetapkan *ITU-T*. Pada parameter *delay* nilai pengujian yang dihasilkan aplikasi memiliki nilai rata-rata 20 ms dan *ITU-T* menetapkan nilai *delay* dibawah 150 ms mendapat predikat baik. Pada parameter *jitter* nilai pengujian yang dihasilkan aplikasi memiliki rata-rata 20 ms dan standar *ITU-T* menetapkan predikat baik jika nilai *jitter* sama dengan 20 atau dibawahnya. Sedangkan pada parameter *packet loss* mendapatkan predikat sangat baik karena dari 4 kali percobaan packet lost yang dihasilkan nilai 0.

4. Faktor-faktor yang mempengaruhi kuliatas Komunikasi *VoIP* yaitu *codec dan bandwidth.* Dalam percobaan penggunaan *codec*, membuktikan bahwa jenis codec yang dipilih akan mempengaruhi kualitas dari komunikasi *VoIP* tersebut. hal tersebut dapat dilihat dari masing-masing jenis codec yang digunakan memiliki kulitas *QoS* yang berbeda. Dan pada pengujian pengujian *bandwidth* dapat dibuktikan bahwa peningkatan *bandwidth* dari 128 *Kbps* menjadi 256 *Kbps* mengakibatkan menurunkan nilai *delay*, *jitter* dan *packet loss.* Pada parameter *delay* mengalami penurunan dari angka 20.405 menjadi 20.149. Pada parameter *jitter* juga mengalami penurunan nilai dari 20.406 menjadi 20.233. Dan terjadi pula pada parameter *packet loss* dari angka 2% menjadi 0.1%. Dengan demikian dapat diambil kesimpulan bahwa peningkatan *bandwidth* akan meningkatkan kualitas dari komunikasi *VoIP* yang dilakukan.

7.2 Saran

Berikut beberapa saran yang diharapkan dalam pengembangan sistem ini.

1. Sistem masih mencakup luas area satu jaringan, suatu saat diharapkan sistem *VoIP DHT Kademlia* ini dapat dikembangkan dalam area yang lebih luas.
2. Pengembangan aplikasi ini sebatas pada aplikasi *mobile* pada *OS Android.* Dikemudian hari aplikasi ini diharapkan dapat dikembangkan pada mode *Dekstop* dengan sistem operasi *linux, windows* dan *OSX* serta juga dapat dekembangkan dalam versi *mobile* dalam *IOS* dan *windows phone.*

DAFTAR PUSTAKA

Bandara, H. M. N. D., & Jayasumana, A. P. (2013). Collaborative applications over peer-to-peer systems-challenges and solutions. *Peer-to-Peer Networking and Applications*, *6*(3), 257–276. http://doi.org/10.1007/s12083-012-0157-3

Bondi, A. B. (2005). Characteristics of scalability and their impact on performance. In *Proceedings of the second international workshop on Software and performance - WOSP '00* (pp. 195–203). New York, New York, USA: ACM Press. http://doi.org/10.1145/350391.350432

Chen, L., Soh, W. S., & Hu, A. (2014). An improved Kademlia protocol with double-layer design for P2P voice communications. In *Communications Security Conference (CSC 2014), 2014* (pp. 1–8). http://doi.org/10.1049/cp.2014.0727

Chou, C. Y. (2006). *Modeling Message Passing Overhead*. *Advances in Grid and Pervasive Computing: First International Conference, GPC 2006*.

Comer, D. (2009). *Computer Networks and Internets*. Prentice Hall. Retrieved from https://books.google.com/books?id=tm-evHmOs3oC&pgis=1

de Castro Louback Rocha, F., & Ruiz, L. B. (2008). A Study of the Effect of Using Kademlia as an Alternative to Centralized User Location Servers in SIP-based IP Telephony Systems. In *Telecommunication Networks and Applications Conference, 2008. ATNAC 2008. Australasian* (pp. 349–354). http://doi.org/10.1109/ATNAC.2008.4783349

El-Rewini, H., & Abd-El-Barr, M. (2005). *Advanced Computer Architecture and Parallel Processing*. John Wiley & Sons. Retrieved from https://books.google.com/books?id=7JB-u6D5Q7kC&pgis=1

Guangmin, L. (2009). An Improved Kademlia Routing Algorithm for P2P Network. In *International Conference on New Trends in Information and Service Science, 2009. NISS '09* (pp. 63–66). http://doi.org/10.1109/NISS.2009.172

Kurose, J. F., & Ross, K. W. (2013). *Computer Network: a top-down approach - 6th ed. Pearson* (Vol. 1). http://doi.org/10.1017/CBO9781107415324.004

Lin, Y., Wu, Q., Lu, X., & Gu, Y. (2010). Bandwidth Constrained Tree Construction for Live Streaming Systems in P2P Networks. In *2010 IEEE 16th International Conference on Parallel and Distributed Systems* (pp. 516–523). IEEE. http://doi.org/10.1109/ICPADS.2010.39

Menychtas, A., Kyriazis, D., & Tserpes, K. (2009). Real-time reconfiguration for guaranteeing QoS provisioning levels in Grid environments. *Future Generation Computer Systems*, *25*(7), 779–784. http://doi.org/10.1016/j.future.2008.11.001

Petar Maymonkov, D. M. (2002). Kademlia: A peer-to-peer information system based on the XOR metric. *PhD Proposal*, *1*. http://doi.org/10.1017/CBO9781107415324.004

Saiful, F., & Nrp, H. A. Q. M. (2006). Analisis Implementasi Aplikasi Video Call pada Sinkronisasi Learning Management System berbasis Moodle sebagai Metode Distance Learning dalam Institusi Pendidikan. *Jurusan Teknik Elektro, Institut Teknologi Sepuluh Nopember, Surabaya.*, 1–6.

Schollmeier, R. (2001). A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. *Proceedings First International Conference on Peer-to-Peer Computing*, 101–102. http://doi.org/10.1109/P2P.2001.990434

Singh, H. P., Singh, S., Singh, J., & Khan, S. A. (2014). VoIP: State of art for global connectivity—A critical review. *Journal of Network and Computer Applications*, *37*, 365–379. http://doi.org/10.1016/j.jnca.2013.02.026

Sun Microsystem. (2009). Distributed Application Architecture. Sun Microsystem.

Suryawan, K. D., Husni, M., & Anggraini, E. L. (2012). Analisis Layanan Kinerja Jaringan VoIP Pada Protokol SRTP Dan VPN. *Jurusan Teknik Elektro, Institut Teknologi Sepuluh Nopember, Surabaya.*, 1–6.

Turaga, D., & Chen, T. (n.d.). ITU-T Video Coding Standards.

Wang, C., Yang, N., & Chen, H. (2010). Improving Lookup Performance Based on Kademlia. In *2010 Second International Conference on Networks Security Wireless Communications and Trusted Computing (NSWCTC)* (Vol. 1, pp. 446–449). http://doi.org/10.1109/NSWCTC.2010.111

Wu, X., Fu, C., & Chang, H. (2007). An Improved Kademlia Protocol In a VoIP System. *2007 IFIP International Conference on Network and Parallel Computing Workshops (NPC 2007)*, 920–928. http://doi.org/10.1109/NPC.2007.168