

**IMPROVISASI CRAWLING PADA PETA WEB
MENGGUNAKAN ALGORITMA TERDISTRIBUSI DENGAN
MODEL KOORDINASI BERBASIS SOCKET PROGRAMMING**

Skripsi

**Disusun untuk memenuhi salah satu syarat
memperoleh gelar Sarjana Komputer**



*Mencerdaskan dan
Memartabatkan Bangsa*

Oleh:
Muhammad Ridho Rizqillah
1313619033

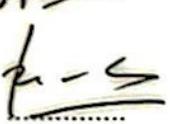
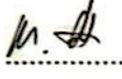
**PROGRAM STUDI ILMU KOMPUTER
FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM
UNIVERSITAS NEGERI JAKARTA**

2024

LEMBAR PERSETUJUAN HASIL SIDANG SKRIPSI
IMPROVISASI CRAWLING PADA PETA WEB
MENGGUNAKAN ALGORITMA TERDISTRIBUSI DENGAN
MODEL KOORDINASI BERBASIS SOCKET PROGRAMMING

Nama : Muhammad Ridho Rizqillah

No. Registrasi : 1313619033

Penanggung Jawab	Nama	Tanda Tangan	Tanggal
Dekan	: Prof. Dr. Muktiningsih N. M.Si.
	NIP. 196405111989032001		
Wakil Penanggung Jawab			
Wakil Dekan I	: Dr. Esmar Budi, S.Si., MT.
	NIP. 197207281999031002		
Ketua	: Drs. Mulyono, M.Kom.		25/01/2024
	NIP. 196605171994031003		
Sekretaris	: Ari Hendarno S.Pd., M.Kom.		24/01/2024
	NIP. 198811022022031002		
Penguji	: Dr. Ria Arafiyah, M.Si.		24/01/2024
	NIP. 197511212005012004		
Pembimbing I	: Muhammad Eka Suryana, M.Kom.		24/01/2024
	NIP. 198512232012121002		
Pembimbing II	: Med Irzal, M.Kom..		24/01/2024
	NIP. 197706152003121001		

Dinyatakan lulus Ujian Skripsi tanggal: 23 Januari 2024

LEMBAR PERNYATAAN

Saya menyatakan dengan sesungguhnya bahwa skripsi dengan judul "**Improvisasi Crawling pada Peta Web Menggunakan Algoritma Terdistribusi dengan Model Koordinasi Berbasis Socket Programming**" yang disusun sebagai syarat untuk memperoleh gelar Sarjana Komputer dari Program Studi Ilmu Komputer Universitas Negeri Jakarta adalah karya ilmiah saya dengan arahan dari dosen pembimbing.

Sumber informasi yang diperoleh dari peneliti lain yang telah dipublikasikan yang disebutkan dalam teks Skripsi ini, telah dicantumkan dalam Daftar Pustaka sesuai dengan norma, kaidah dan etika penulisan ilmiah.

Jika dikemudian hari ditemukan sebagian besar skripsi ini bukan hasil karya saya sendiri dalam bagian-bagian tertentu, saya bersedia menerima sanksi pencabutan gelar akademik yang saya sanding dan sanksi-sanksi lainnya sesuai dengan peraturan perundang-undangan yang berlaku.

Jakarta, 12 Januari 2024



Muhammad Ridho Rizqillah



KEMENTERIAN PENDIDIKAN DAN KEBUDAYAAN
UNIVERSITAS NEGERI JAKARTA
UPT PERPUSTAKAAN

Jalan Rawamangun Muka Jakarta 13220
Telepon/Faksimili: 021-4894221
Laman: lib.unj.ac.id

**LEMBAR PERNYATAAN PERSETUJUAN PUBLIKASI
KARYA ILMIAH UNTUK KEPENTINGAN AKADEMIS**

Sebagai sivitas akademika Universitas Negeri Jakarta, yang bertanda tangan di bawah ini, saya:

Nama : Muhammad Ridho Rizqillah
NIM : 1313619033
Fakultas/Prodi : MIPA / Ilmu Komputer
Alamat email : mridhor08@gmail.com

Demi pengembangan ilmu pengetahuan, menyetujui untuk memberikan kepada UPT Perpustakaan Universitas Negeri Jakarta, Hak Bebas Royalti Non-Ekslusif atas karya ilmiah:

Skripsi Tesis Disertasi Lain-lain (.....)

yang berjudul :

Improvisasi Crawling Pada Peta Web Menggunakan Algoritma Terdistribusi

Dengan Model Koordinasi Berbasis Socket Programming

Dengan Hak Bebas Royalti Non-Ekslusif ini UPT Perpustakaan Universitas Negeri Jakarta berhak menyimpan, mengalihmediakan, mengelolanya dalam bentuk pangkalan data (*database*), mendistribusikannya, dan menampilkan/mempublikasikannya di internet atau media lain secara *fulltext* untuk kepentingan akademis tanpa perlu meminta ijin dari saya selama tetap mencantumkan nama saya sebagai penulis/pencipta dan atau penerbit yang bersangkutan.

Saya bersedia untuk menanggung secara pribadi, tanpa melibatkan pihak Perpustakaan Universitas Negeri Jakarta, segala bentuk tuntutan hukum yang timbul atas pelanggaran Hak Cipta dalam karya ilmiah saya ini.

Demikian pernyataan ini saya buat dengan sebenarnya.

Jakarta , 12 Januari 2024

Penulis

(Muhammad Ridho Rizqillah)
nama dan tanda tangan

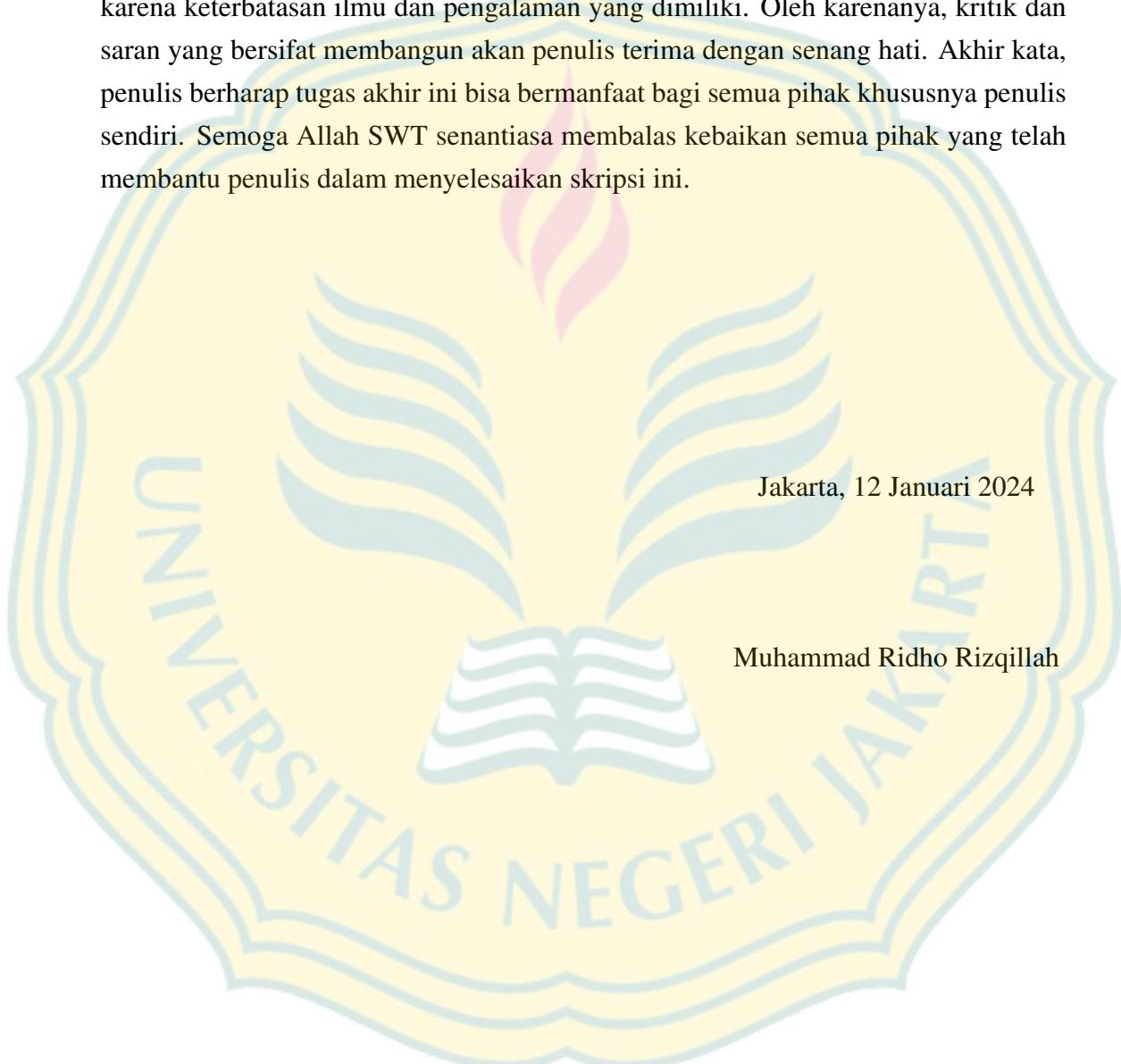
KATA PENGANTAR

Puji syukur penulis panjatkan ke hadirat Allah SWT, karena dengan rahmat dan karunia-Nya, penulis dapat menyelesaikan skripsi yang berjudul *Improvisasi Crawling pada Peta Web Menggunakan Algoritma Terdistribusi dengan Model Koordinasi Berbasis Socket Programming*.

Keberhasilan penyusunan skripsi ini merupakan hasil dari kerja sama yang erat dengan berbagai pihak yang dengan tulus dan ikhlas memberikan kontribusi berharga untuk menyempurnakan penyusunan skripsi ini. Penulis merasa perlu menyampaikan rasa terima kasih dengan kerendahan hati kepada semua pihak yang telah turut serta memberikan bantuan dan masukan dalam proses penyusunan skripsi ini. Dalam kesempatan ini, penulis ingin mengekspresikan rasa syukur yang mendalam atas dukungan dan kerja sama yang telah diberikan oleh berbagai individu dan lembaga yang turut berperan dalam keberhasilan skripsi ini.

1. Yth. Para petinggi di lingkungan FMIPA Universitas Negeri Jakarta.
2. Yth. Ibu Dr. Ria Arafiyah, M.Si selaku Koordinator Program Studi Ilmu Komputer.
3. Yth. Bapak Muhammad Eka Suryana, M.Kom selaku Dosen Pembimbing I yang telah membimbing, mengarahkan, serta memberikan saran dan koreksi terhadap skripsi ini.
4. Yth. Bapak Med Irzal, M.Kom selaku Dosen Pembimbing II yang telah membimbing, mengarahkan, serta memberikan saran dan koreksi terhadap skripsi ini.
5. Kedua orang tua, kakak, dan adik penulis yang telah mendukung dan memberikan semangat serta doa untuk penulis.
6. Teman - teman Program Studi Ilmu Komputer 2019 yang telah memberikan dukungan dan dorongan untuk menyelesaikan skripsi ini.
7. Teman - teman Warteg yang bersedia memberikan dukungan dan kebersamaan dalam menyelesaikan skripsi ini.
8. Salah satu adik tingkat yang serta merta menemani, mendengarkan keluh kesah, memberikan doa, dan semangat.

Penulis menyadari bahwa penyusunan skripsi ini masih jauh dari sempurna karena keterbatasan ilmu dan pengalaman yang dimiliki. Oleh karenanya, kritik dan saran yang bersifat membangun akan penulis terima dengan senang hati. Akhir kata, penulis berharap tugas akhir ini bisa bermanfaat bagi semua pihak khususnya penulis sendiri. Semoga Allah SWT senantiasa membela kebaikan semua pihak yang telah membantu penulis dalam menyelesaikan skripsi ini.



*Mencerdaskan dan
Memartabatkan Bangsa*

ABSTRAK

Muhammad Ridho Rizqillah. Improvisasi *Crawling* pada Peta Web Menggunakan Algoritma Terdistribusi dengan Model Koordinasi Berbasis *Socket Programming*. Skripsi. Program Studi Ilmu Komputer. Fakultas Matematika dan Ilmu Pengetahuan Alam, Universitas Negeri Jakarta. Januari 2024. Di bawah bimbingan Muhammad Eka Suryana, M. Kom. dan Med Irzal, M. Kom.

Mesin pencari memerlukan jumlah data yang besar untuk berjalan dengan optimal. Data yang besar diperlukan proses *crawling* yang masif. Penelitian ini mencoba meningkatkan efisiensi *crawling* dengan mengimplementasikan *crawler* terdistribusi menggunakan *socket* sebagai media komunikasi antar perangkat. Sistem arsitektur *crawler* terdistribusi mencakup *tracker*, *manager*, dan *client*. Tujuan utamanya adalah untuk meningkatkan efisiensi dan efektivitas *crawling* dengan cara terdistribusi. Hasil akhir dari improvisasi ini menunjukkan peningkatan data sebanyak 30% secara terdistribusi dengan dua *crawler*, dibanding dengan *crawler* individual. Serta tidak adanya data yang terduplikasi.

Kata kunci : *search engine, distributed crawler, socket programming, crawling.*

*Mencerdaskan dan
Memartabatkan Bangsa*

ABSTRACT

Muhammad Ridho Rizqillah. Improvisasi *Crawling* pada Peta Web Menggunakan Algoritma Terdistribusi dengan Model Koordinasi Berbasis *Socket Programming*. Mini Thesis. Computer Science. Faculty of Mathematics and Natural Sciences, State University of Jakarta. January 2024. Under the guidance of Muhammad Eka Suryana, M. Kom. and Med Irzal, M. Kom.

Search engines require large amounts of data to run with optimal. Big data requires a massive crawling process. This research trying to improve crawling efficiency by implementing crawlers distributed using sockets as a communication medium between devices. System Distributed crawler architecture includes tracker, manager, and client. Objective The main thing is to increase the efficiency and effectiveness of crawling by means distributed. The final result of this improvisation shows an increase in data as much as 30% distributed with two crawlers, compared to a crawler individual. And there is no duplicate data.

Keyword : search engine, distributed crawler, socket programming, crawling.

*Mencerdaskan dan
Memartabatkan Bangsa*

DAFTAR ISI

ABSTRAK	vi
ABSTRACT	vii
DAFTAR ISI	ix
DAFTAR GAMBAR	xi
DAFTAR TABEL	xii
I PENDAHULUAN	1
1.1 Latar Belakang Masalah	1
1.2 Rumusan Masalah	4
1.3 Pembatasan Masalah	4
1.4 Tujuan Penelitian	5
1.5 Manfaat Penelitian	5
II KAJIAN PUSTAKA	6
2.1 Jaringan Komputer	6
2.2 Protokol Jaringan	6
2.3 Protocol Layering	7
2.4 <i>TCP/IP Model</i>	9
2.4.1 <i>Link Layer</i>	11
2.4.2 <i>Internet Layer</i>	13
2.4.3 <i>Transport Layer</i>	15
2.4.4 <i>Application Layer</i>	25
2.5 <i>Hypertext Transfer Protocol</i>	30
2.5.1 <i>HTTP/2</i>	30
2.5.2 <i>HTTP/3</i>	31
2.6 <i>Socket Programming</i>	31
2.7 <i>Bittorrent</i>	37
2.7.1 Arsitektur Bittorrent	38
2.7.2 <i>Distributed Hash Tables (DHTs)</i>	40
2.7.3 Proses Komunikasi BitTorrent	40
2.8 Algoritma pada Bittorrent	41
2.8.1 <i>Piece Selection</i>	41
2.8.2 <i>Strict Policy</i>	42
2.8.3 <i>Rarest First</i>	42
2.8.4 <i>Random Piece First</i>	42
2.8.5 <i>Endgame Mode</i>	42
2.9 <i>Resource Allocation Bittorrent</i>	43

2.9.1	<i>Choking</i>	43
2.9.2	<i>Optimistic Unchoking</i>	44
2.10	<i>Network Address Translation (NAT)</i>	44
2.10.1	Klasifikasi NAT	45
2.10.2	<i>NAT Traversal</i>	47
2.10.3	<i>UDP Hole Punching</i>	48
2.10.4	<i>Timeouts</i>	52
2.11	<i>Distributed Web Crawler Architecture</i>	52
2.11.1	<i>Workbench</i>	54
2.11.2	<i>Distributor</i>	56
III METODOLOGI PENELITIAN		58
3.1	Tahapan Penelitian	58
3.2	Arsitektur <i>Crawler</i> Terdistribusi	58
3.3	Proses Komunikasi <i>Peer-to-peer</i>	61
3.4	Penyeimbang Antrian <i>URL</i>	64
3.5	Skema Pendistribusian	66
3.6	Alat dan Bahan Penelitian	67
3.7	Tahapan Pengembangan	67
3.7.1	Improvisasi <i>crawling</i> secara terdistribusi	67
3.7.2	Skenario Eksperimen	68
3.8	Skema Uji	69
3.8.1	Sumber Data	69
3.8.2	Metrik Pengujian	69
3.8.3	Rancangan Eksperimen	70
IV HASIL DAN PEMBAHASAN		71
4.1	Implementasi	71
4.1.1	<i>Tracker</i>	72
4.1.2	<i>Manager</i>	75
4.1.3	Klien	76
4.2	Pengujian	88
4.2.1	Proses Komunikasi	89
4.2.2	Konsistensi Data	90
4.2.3	Efisien Sumber Daya dan Optimalisasi Data	90
4.3	Analisis Hasil	91
V KESIMPULAN DAN SARAN		92
5.1	Kesimpulan	92
5.2	Saran	93
DAFTAR PUSTAKA		94

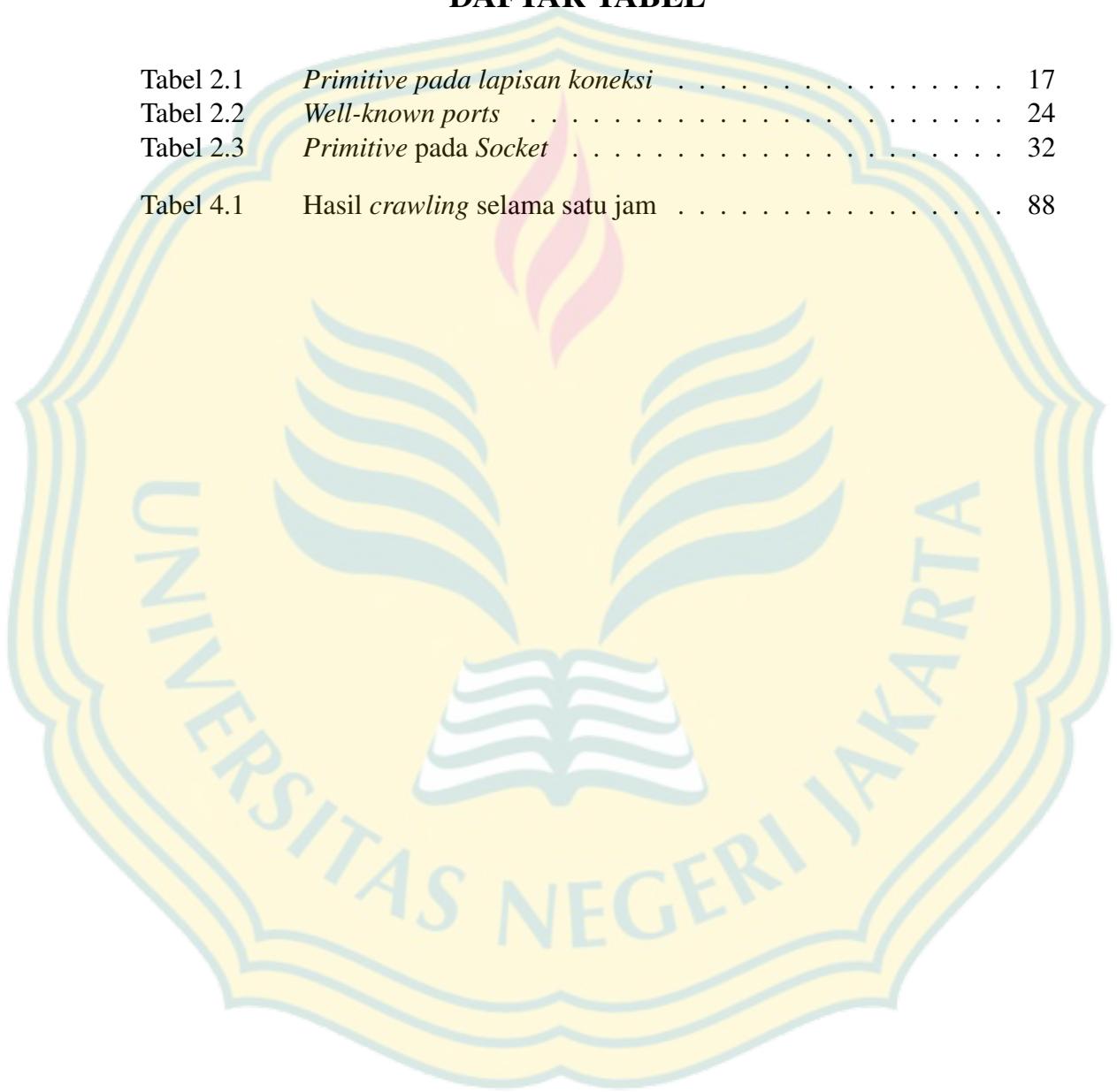
DAFTAR GAMBAR

Gambar 1.1	<i>Analisis Database Penelitian Crawling</i>	2
Gambar 2.1	Five-Layer Network	8
Gambar 2.2	OSI dan TCP/IP Model	10
Gambar 2.3	TCP/IP Model dan Protokolnya	11
Gambar 2.4	Jaringan Transportasi dan Lapisan Aplikasi	16
Gambar 2.5	Segmen Paket dan <i>Frame</i>	18
Gambar 2.6	Koneksi Transportasi TSAP dan NSAP	19
Gambar 2.7	(a) Multiplexing, (b) Inverse Multiplexing	21
Gambar 2.8	UDP Header	22
Gambar 2.9	IPv4 Pseudoheader di UDP Checksum	23
Gambar 2.10	TCP Header	25
Gambar 2.11	CDN Tree	27
Gambar 2.12	Bittorent	29
Gambar 2.13	Skema Koneksi Manajemen Sederhana	32
Gambar 2.14	Kode Socket Sisi Klien	36
Gambar 2.15	Kode Socket Sisi Server	37
Gambar 2.16	(a) <i>Centralized Tracker</i> , (b) <i>Decentralized Tracker</i>	39
Gambar 2.17	<i>Full-cone NAT</i>	45
Gambar 2.18	<i>Address-restricted-cone NAT</i>	46
Gambar 2.19	<i>Port-restricted-cone NAT</i>	46
Gambar 2.20	<i>Symmetric NAT</i>	47
Gambar 2.21	Sebelum <i>Hole Punching</i> di Belakang Satu NAT yang Sama.	49
Gambar 2.22	Setelah <i>Hole Punching</i> di Belakang Satu NAT yang Sama.	49
Gambar 2.23	Sebelum <i>Hole Punching</i> di Belakang NAT yang Berbeda.	50
Gambar 2.24	Setelah <i>Hole Punching</i> di Belakang NAT yang Berbeda.	50
Gambar 2.25	Sebelum <i>Hole Punching</i> di Belakang Berbagai Level NAT.	51
Gambar 2.26	Setelah <i>Hole Punching</i> di Belakang Berbagai Level NAT.	51
Gambar 2.27	Arsitektur BUbiNG	53
Gambar 2.28	BUbiNG <i>Workbench</i>	54
Gambar 2.29	BUbiNG <i>Distributor</i>	57
Gambar 3.1	<i>Flowchart</i> Tahapan Penelitian Crawler Terdistribusi	58
Gambar 3.2	Arsitektur <i>Crawler Master-Slave</i>	59
Gambar 3.3	Arsitektur <i>Crawler Peer to Peer</i> Melalui NAT	60
Gambar 3.4	<i>Flowchart</i> Skema Pendistribusian	66
Gambar 4.1	<i>Flowchart</i> Tahapan Penelitian yang Berhasil Dibuat	71
Gambar 4.2	Arsitektur <i>Crawler Terdistribusi</i>	72
Gambar 4.3	<i>Flowchart</i> <i>tracker</i>	73
Gambar 4.4	Potongan kode <i>tracker</i> untuk klien baru terhubung	73

Gambar 4.5	Potongan kode <i>tracker</i> merespons <i>manager</i> yang meminta kumpulan klien	74
Gambar 4.6	Potongan kode <i>tracker</i> merespons <i>manager</i> sudah siap membuka koneksi	74
Gambar 4.7	Potongan kode <i>tracker</i> memberitahu klien <i>private</i> untuk <i>connect</i>	75
Gambar 4.8	<i>Flowchart manager</i>	75
Gambar 4.9	Potongan kode <i>manager</i> yang menerima data klien dan menginformasikan <i>tracker</i>	76
Gambar 4.10	Potongan kode <i>manager</i> untuk membuka koneksi agar klien <i>public</i> dapat terhubung	76
Gambar 4.11	<i>Flowchart client</i>	77
Gambar 4.12	Struktur direktori kode klien	78
Gambar 4.13	Fungsi untuk mendapatkan IP <i>address</i> apakah <i>public</i> atau <i>private</i>	79
Gambar 4.14	Pendeklarasian kelas <i>Client</i>	80
Gambar 4.15	Pendeklarasian kelas <i>LoadBalancer</i>	80
Gambar 4.16	Potongan kode klien <i>private</i>	81
Gambar 4.17	Potongan kode klien <i>public</i> mencoba terhubung dengan <i>manager</i>	81
Gambar 4.18	Potongan kode komunikasi antar klien	82
Gambar 4.19	Potongan kode klien <i>public</i> menerima data hasil <i>crawling</i>	83
Gambar 4.20	Fungsi untuk pengecekan duplikasi url yang didapatkan dari hasil <i>crawling</i>	83
Gambar 4.21	Potongan kode klien <i>public</i> mengirimkan url yang akan di- <i>crawling</i>	84
Gambar 4.22	Fungsi untuk pengecekan "kesehatan" dari setiap klien yang terhubung	85
Gambar 4.23	Fungsi untuk klien <i>private</i> terhubung ke klien <i>public</i>	85
Gambar 4.24	Fungsi untuk klien <i>private</i> menerima url yang akan di- <i>crawling</i> dari klien <i>public</i>	86
Gambar 4.25	Potongan kode dari fungsi untuk klien <i>private</i> mengirim url hasil <i>crawling</i> ke klien <i>public</i>	87
Gambar 4.26	Potongan kode untuk klien <i>private</i> melakukan <i>scraping</i> halaman web	88
Gambar 4.27	Proses komunikasi pada <i>tracker</i>	89
Gambar 4.28	Proses komunikasi pada <i>manager</i>	89
Gambar 4.29	Proses komunikasi pada klien <i>public</i>	89
Gambar 4.30	Proses komunikasi pada klien <i>private</i>	90

DAFTAR TABEL

Tabel 2.1	<i>Primitive pada lapisan koneksi</i>	17
Tabel 2.2	<i>Well-known ports</i>	24
Tabel 2.3	<i>Primitive pada Socket</i>	32
Tabel 4.1	Hasil <i>crawling</i> selama satu jam	88



*Mencerdaskan dan
Memartabatkan Bangsa*

BAB I

PENDAHULUAN

1.1 Latar Belakang Masalah

Mesin pencari atau *search engine* adalah sebuah teknologi yang digunakan untuk mencari sebuah informasi yang tersedia di internet. Dengan memasukkan *keyword* terkait hal yang ingin diketahui dan akan terlihat berbagai macam situs *web* yang memiliki informasi sesuai. Perkembangan *search engine* melahirkan penelitian yang berkelanjutan, meneliti bagaimana menciptakan algoritma yang efektif untuk menjalankan *search engine* agar menampilkan pencarian yang relevan kepada pengguna.

Penelitian *search engine* yang dilakukan sebelumnya lebih terfokus untuk mengembangkan *crawler* yang dapat berjalan secara *multi-threaded* (Qoriiba 2021). Sedangkan pada penelitian selanjutnya mengarah pada *refactoring crawler* dari penelitian sebelumnya dan mencari *similarity score* dari setiap halaman situs *web* yang di-*crawling* (Khatulistiwa 2023). Dan penelitian lainnya saat melakukan pengumpulan data menggunakan *crawler* yang sudah ada agar dapat dikelola dalam bentuk *user interface* (Asmara 2024).

Kelemahan yang terdapat pada penelitian tersebut adalah *crawling* hanya dapat berjalan pada satu perangkat saja. Hal itu dapat menghambat hasil dari *crawling*, karena membutuhkan waktu lebih lama untuk mendapatkan hasil *crawling* yang masif. Oleh karena itu, diperlukan cara agar *crawler* dapat berjalan pada berbagai macam perangkat atau terdistribusi satu dengan lainnya. Serta kelemahan yang didapatkan bahwa proses *crawling* dengan *public IP address* dapat menyebabkan *IP public* perangkat terblokir oleh Cloudflare atau layanan penyedia internet, sebab dianggap perilaku mencurigakan.

Terdapat juga kelemahan lainnya pada *database*. *Wasting storage* yang cukup besar terhadap kolom - kolom yang tidak terlalu diperlukan untuk *similarity scoring*. Maka, peneliti melakukan analisis terkait *database*-nya dengan melakukan *crawling* selama 48 jam, didapatkan hasil laporan *database* pada gambar 1.1. Total dari keseluruhan *database*-nya adalah 8.7 GB.

Table	Action	Rows	Type	Collation	Size	Overhead
crawling		1	InnoDB	latin1_swedish_ci	16.0 Kib	-
page_forms		~104,404	InnoDB	latin1_swedish_ci	84.6 MiB	-
page_images		~1,944,622	InnoDB	latin1_swedish_ci	812.0 MiB	-
page_information		61,904	InnoDB	latin1_swedish_ci	353.8 MiB	-
page_linking		~7,826,012	InnoDB	latin1_swedish_ci	1.7 GiB	-
page_list		~5,065,296	InnoDB	latin1_swedish_ci	2.5 GiB	-
page_scripts		~2,307,602	InnoDB	latin1_swedish_ci	3.0 GiB	-
page_styles		~127,348	InnoDB	latin1_swedish_ci	250.6 MiB	-
page_tables		~67,019	InnoDB	latin1_swedish_ci	121.6 MiB	-
9 tables	Sum				~17,504,208	InnoDB latin1_swedish_ci 8.7 GiB 0 B

Gambar 1.1: Analisis Database Penelitian Crawling

Terdapat *wasting storage* pada *table page_list*, mengandung kumpulan *tag *. *page_scripts*, mengandung skrip *javascript*. *page_styles*, mengandung *style* dari CSS (*Cascading Style Sheet*). *page_tables*, mengandung *tag <table>*. Karena memiliki ukuran yang besar dan informasi yang didapatkan kurang berguna untuk melakukan *page ranking*. Bahkan ukuran dari *tables* yang disebutkan di atas dapat memenuhi setidaknya lebih dari 60% dari total ukuran *database*.

Sedangkan *tables* yang berguna untuk hasil dari *crawling* adalah: *Table crawling*, untuk menjelaskan *starting url*, *total page*, durasi *crawl*, dan tanggal *crawling* berakhir. Lalu, pada *table page_linking* untuk mengetahui *outgoing link*, yang nantinya akan digunakan untuk *page ranking*. Pada *table page_information* diperlukan untuk mendapatkan *initial page rank*, yang hasilnya didapatkan dari 1 / total baris *table page_information*. Dan juga berguna untuk *document ranking tf-idf* dengan mengambil setiap kata yang ada. Melalui hasil analisis *database* berikut, bahwa data penting yang digunakan untuk *page rank* dan *tf idf* adalah pada *table crawling*, *page_information*, dan *page_linking*.

Diperlukan improvisasi pada algoritma dan menambahkan cara untuk membagi tugas *crawling* supaya menjadi versi terdistribusi. *Crawling* terdistribusi adalah melakukan *crawling* secara terpisah ke setiap perangkat yang tersedia, guna membagi beban kerja dan dapat menghasilkan hasil *crawler* dalam jumlah yang masif secara efisien serta efektif. Sebelum melakukan improvisasi *crawler* yang sekarang menjadi terdistribusi, diperlukan pemahaman terkait sistem arsitekturnya.

Sistem arsitektur *crawler* terdistribusi memiliki dua komponen penting, yaitu *crawling system* dan *crawling application*. *Crawling System* meliputi *crawl manager*, *downloader*, dan *DNS resolvers*. Sedangkan *crawling application* yang melakukan

proses *crawling* yang berjalan pada perangkat.

Berdasarkan arsitektur sederhana didapatkan bahwa proses distribusi yang terjadi adalah pada sisi *downloader*. Setelah *crawl manager* melakukan *crawling* pada suatu *web* yang akan menghasilkan *urls*, kemudian *url* tersebut dilanjutkan kepada masing - masing *downloader*. *Downloader* ini terletak pada *server* atau perangkat yang berbeda (Shkapenyuk dkk., 2002).

Komunikasi yang dilakukan dalam arsitektur tersebut dilakukan dalam dua cara yaitu, via *socket* dan *file system* untuk data yang lebih besar. Untuk improvisasi yang akan diterapkan dalam proyek ini akan lebih tertuju kepada penerapan *socket*.

Socket dapat dianggap sebagai *endpoint* dalam *two-way communication channel*. *Socket* dapat melakukan komunikasi antara dua proses, baik dalam satu perangkat atau perangkat yang berbeda. Untuk dapat melakukan komunikasi dua arah, terlebih dahulu dipahami arsitektur *client-server*. *Client* dapat berupa PC, laptop, telepon seluler, dll. Dan setiap *client* memiliki IP *address*-nya masing - masing. Untuk dapat melakukan komunikasi atau pertukaran data diperlukan setiap *client* untuk terhubung ke sebuah *server*. *Server* pun juga memiliki IP *address*. Agar *client* dapat terhubung ke *server*, maka ia harus mengetahui IP *server*. Dan setiap *client* yang terhubung ke *server* akan dapat berkomunikasi via *server* dalam hal ini akan menggunakan *socket* (IBMCorporation 2021).

Pada penelitian lain terkait *query processing* dan alokasi komputasional *crawler* terdistribusi yang dilakukan oleh Cambazoglu terdapat beberapa jenis arsitektur berdasarkan cara penyimpanan *index*-nya. Secara *centralized* (SE-C), *replicated* (SE-R), dan *partitioned* (SE-P). Ada juga arsitektur gabungan antara *partial replication* dan *query forwarding* (SE-H). Arsitektur *centralized* masih banyak digunakan sampai sekarang, apalagi dalam skala yang lebih kecil (Cambazoglu dkk., 2009).

Pada penelitian yang dilakukan oleh (Orlando dkk., 2002) menjelaskan bahwa dalam *web search engine* terdiri dari *spidering system*. Kesatuan dari sistem ini berjalan secara paralel, yang mana *crawler* akan mengunjungi setiap *web* dan mengumpulkan informasi yang diperlukan. Inti utama dalam *Information Retrieval* (IR) adalah *indexer* dan *query analyzer*.

Pada tahapan *query analyzer* melakukan proses paralel terhadap *Query Brokers* dan *Local Searchers*. Dilakukan secara terdistribusi menggunakan teknik *document partitioning*. Strategi untuk membuat paralel *web search engine* harus memenuhi dua hal, *task parallel* dan *data parallel*. Untuk menghindari eksplorasi

processor secara berlebihan, maka diperlukan *load balancer*, agar setiap *query* dapat berjalan secara efisien dan efektif. Dengan melakukan pembagian *task* secara paralel. Oleh karena setiap *query* diproses secara terpisah, maka juga memiliki partisi *database*-nya tersendiri. Untuk itu diperlukan kombinasi antara *task* dan *data parallel*, agar hasil akhir dari proses *search engine* lebih relevan (Orlando dkk., 2002).

Pada penelitian lain dijelaskan juga mengenai distribusi algoritma, walaupun diterapkan pada perhitungan *page rank* agar lebih efisien memori dalam melakukan perhitungan. Karena untuk perhitungannya diperlukan memori yang besar. Oleh karena itu, diperlukan proses efisiensi memori (Pradana 2023). Walaupun begitu data perhitungan akan berkurang karena efisiensi memori, memang ada hal yang menjadi timbal baliknya. Nanti hasil dari penelitian tersebut juga akan diterapkan untuk penelitian lebih lanjut setelah penelitian yang penulis lakukan.

Berdasarkan beberapa rujukan sebelumnya, hasil dari penelitian ini adalah sebuah *crawler* yang berjalan secara terdistribusi yang akan mengimplementasikan penggunaan *socket* dalam mendistribusikan tugas ke setiap perangkat, serta peningkatan jumlah data yang terkumpul.

1.2 Rumusan Masalah

Berdasarkan uraian pada latar belakang yang diutarakan di atas, maka perumusan masalah pada penelitian ini adalah "**bagaimana meningkatkan efisiensi *crawling* dengan cara terdistribusi?**".

1.3 Pembatasan Masalah

Pembatasan masalah pada penelitian ini antara lain:

1. Pengembangan arsitektur *crawler* menjadi versi terdistribusi dengan skala kinerja 2 hingga 5 perangkat.
2. *Crawler* terdistribusi yang dikembangkan menggunakan *socket* sebagai media komunikasi antar perangkat.
3. Pengujian terhadap *crawler* terdistribusi dan tidak terdistribusi yang akan mempertimbangkan faktor-faktor seperti kecepatan crawling dan efisiensi penggunaan sumber daya.

1.4 Tujuan Penelitian

Tujuan penelitian pada penelitian ini antara lain:

1. Mengembangkan arsitektur *distributed crawler*.
2. Membuat implementasi *socket* untuk memenuhi kebutuhan improvisasi *crawler*.

1.5 Manfaat Penelitian

1. Bagi penulis

Menambah pengetahuan di bidang sistem terdistribusi mengenai *search engine* dan *crawling*, mengasah kemampuan programming, dan memperoleh gelar sarjana di bidang Ilmu Komputer. Selain itu, penulisan ini juga merupakan media bagi penulis untuk mengaplikasikan ilmu yang didapat di kampus ke kehidupan masyarakat.

2. Bagi Universitas Negeri Jakarta

Menjadi pertimbangan dan evaluasi akademik khususnya Program Studi Ilmu Komputer dalam penyusunan skripsi sehingga dapat meningkatkan kualitas akademik di program studi Ilmu Komputer Universitas Negeri Jakarta serta meningkatkan kualitas lulusannya.

*Mencerdaskan dan
Memartabatkan Bangsa*

BAB II

KAJIAN PUSTAKA

2.1 Jaringan Komputer

Jaringan komputer adalah sekumpulan perangkat komputasi yang saling terhubung satu sama lain. Seperti dua komputer yang saling terhubung dan dapat bertukar informasi. Interkoneksi dapat berlangsung pada berbagai macam media transmisi, seperti kawat tembaga, kabel *fiber optic*, dan gelombang radio. Internet adalah contoh yang sering dijumpai sebagai jaringan dari banyak jaringan.

Peran jaringan komputer yang sangat umum adalah untuk mendapatkan akses terhadap suatu informasi. Cara pengaksesan informasi ini pun dapat melalui berbagai cara, menggunakan internet dengan *web browser*, mengakses informasi melalui media sosial. Bahkan dengan menggunakan *smartphone* sudah dapat mengakses banyak informasi dengan internet yang terhubung. Banyak informasi yang diakses melalui internet menggunakan model *client-server*, klien meminta informasi kepada server melalui jaringan, dan server mengembalikan informasi yang dibutuhkan oleh klien. Model ini sudah banyak dan secara luas diterapkan dalam proses mendapatkan informasi. Model lain yang cukup populer adalah *peer-to-peer*, dalam proses komunikasi *peer-to-peer* dapat langsung berhubungan dengan klien atau server dan tidak pasti siapa yang akan menjadi klien atau server. Jadi, keduanya dapat bertukar informasi secara langsung.

2.2 Protokol Jaringan

Protokol jaringan adalah aturan yang ada pada jaringan. Kegunaannya untuk menjaga agar jaringan yang digunakan dapat berjalan dengan baik. Beberapa hal yang perlu diperhatikan pada protokol jaringan yaitu:

1. *Reliability*

Agar suatu jaringan dapat beroperasi dengan normal, perlu adanya *reliability* atau keandalan. Sebab, sering dijumpai beberapa kesalahan dalam transmisi informasi, seperti *electrical noise*, *random wireless signal*, dll. Cara mengatasinya hal tersebut terjadi dengan menerapkan *error detection*. Dengan demikian lebih mudah untuk mengetahui kemungkinan *error* yang terjadi dan

dapat diperbaiki dengan cepat. Alangkah baiknya jika diterapkan juga *error correction*. Ketika terjadi *error*, maka dengan otomatis akan teratasi. Sebagai contoh ketika terdapat gangguan koneksi di suatu rute transmisi, maka akan teratasi dengan mengubah jalur transmisi ke rute yang lain.

2. *Resource Allocation*

Semakin besar dan berkembang suatu jaringan, akan timbul permasalahan baru. Seperti *resource* atau sumber data yang semakin terbatas. Ketika ingin meminta informasi, dari klien kepada server, karena semakin besar jaringan yang ada dalam suatu lingkup. Maka akan terjadi *traffic* yang tinggi sebab banyaknya klien yang meminta informasi dalam waktu yang bersamaan. Untuk mengatasi hal ini, diperlukan pembatasan terhadap permintaan atau menerapkan *bandwidth*. Dengan adanya *bandwidth*, setiap klien memiliki batasan untuk meminta sebuah informasi pada waktu yang ditentukan.

3. *Evolvability*

Berkembangnya jaringan dari waktu ke waktu memerlukan adaptasi terhadap jaringan yang ada sebelumnya. Perubahan struktur jaringan dan mekanisme yang digunakan perlu diselaraskan dengan jaringan yang terdahulu sudah ada. Karena tidak mungkin perubahan dilakukan secara drastis dan menghapus total jaringan terdahulu. Perlu ada masa peralihan dan penyesuaian.

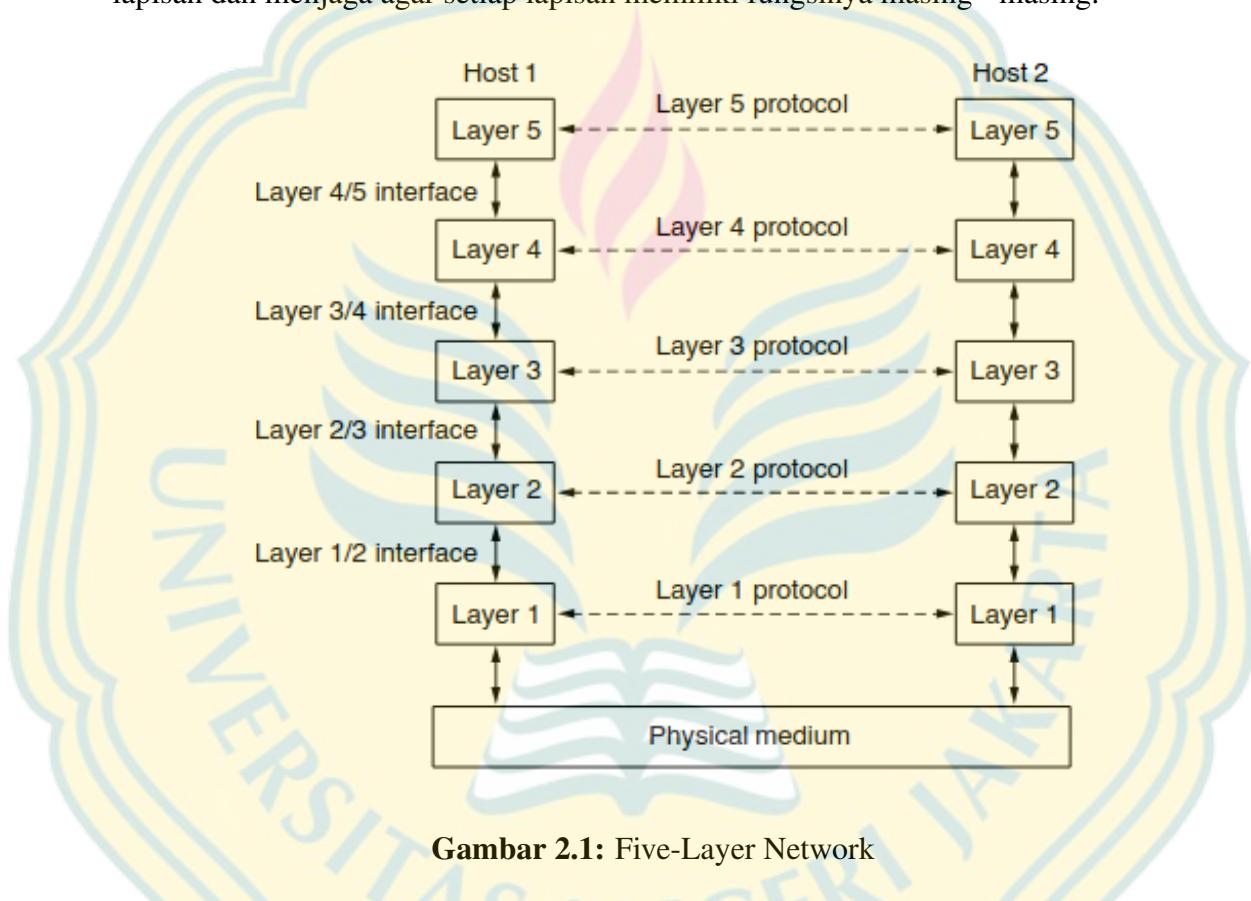
4. *Security*

Keamanan adalah faktor penting dalam jaringan. Untuk menghadapi berbagai macam ancaman yang ada diperlukan keamanan yang memadai. Salah satu mekanisme yang diterapkan adalah dengan menggunakan autentikasi, dengan adanya autentikasi menjadikan seseorang menjadi lebih jelas identitasnya dan tidak akan mudah untuk melakukan penyamaran.

2.3 Protocol Layering

Untuk mengurangi kompleksitas dalam jaringan, dibentuk sebuah *layer* atau lapisan. Masing - masing lapisan terbentuk berdasar lapisan yang ada di bawahnya. Dengan pembuatan lapisan ini menyediakan layanan dan kebutuhan untuk lapisan berikutnya yang lebih tinggi, sementara melindungi detail penting dari setiap lapisan dan implementasinya. Hal ini mirip seperti konsep pada *Object Oriented*

Programming (OOP) yang mana terdapat pilar - pilar yang terdiri dari *abstraction* dan *encapsulation*. Berguna untuk hanya menampilkan info yang penting dari setiap lapisan dan menjaga agar setiap lapisan memiliki fungsinya masing - masing.



Gambar 2.1: Five-Layer Network

Pada Gambar 2.1 adalah contoh dari *five-layer network*. Entitas yang terdiri dari setiap lapisan itu disebut dengan *peers*. *Peers* dapat berupa proses perangkat lunak, perangkat keras, atau bahkan manusia. Setiap *peers* berkomunikasi menggunakan protokol. Proses transmisi data yang terjadi tidak serta merta langsung antara satu lapisan pada satu perangkat menuju lapisan yang sama pada perangkat yang lain. Melainkan prosesnya terjadi dari lapisan paling atas menuju ke lapisan paling bawah sampai mencapai physical medium, setelah itu baru komunikasi berlanjut pada lapisan itu. Garis putus - putus menunjukkan *virtual communication* melalui protokol dan garis tidak putus - putus menunjukkan *physical communication* antara tiap lapisan.

Diantara setiap lapisan terdapat *interface*. *Interface* bertujuan untuk mendefinisikan operasi dan layanan apa saja yang disediakan lapisan bawah untuk lapisan atasnya. Hal penting yang harus diperhatikan dalam mendesain jaringan

adalah clean *interface* agar menjadi lebih jelas apa fungsi dari setiap lapisannya.

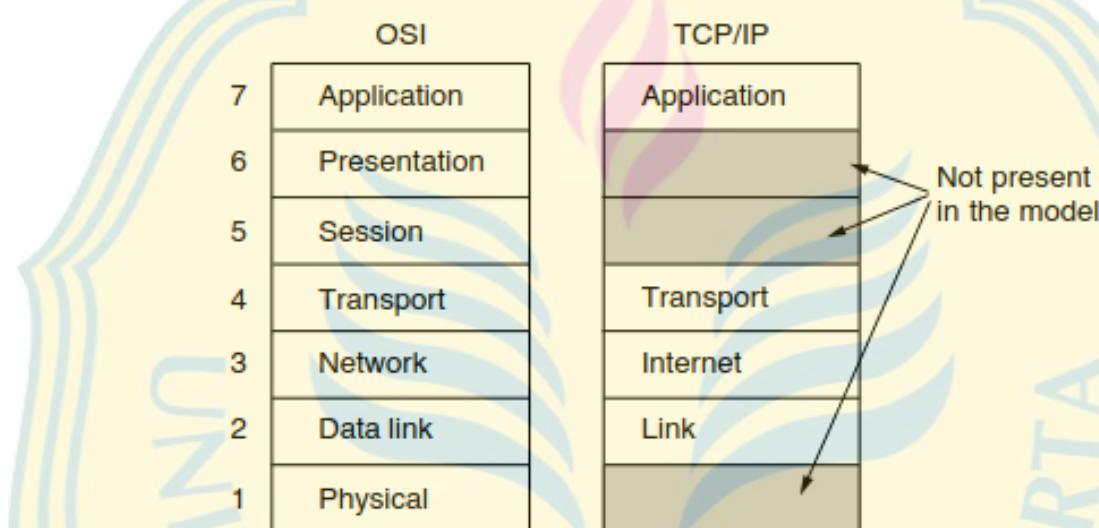
Berbagai lapisan menawarkan dua opsi layanan koneksi: *connection-oriented* dan *connectionless*. Pada *connection-oriented*, seperti halnya koneksi telepon. Untuk menggunakan telepon, pemanggil memasukan nomor tujuan dan menelponnya. Pada pihak penerima mendapatkan panggilan dan mengangkat telepon, barulah koneksi terhubung dan terjadi. pemanggil memberikan sebuah informasi dan diterima oleh penerima dan begitupun sebaliknya. Pada *connectionless*, seperti halnya surat fisik pada layanan pos. Pengirim mengirim surat kepada penerima dengan alamatnya. Pesan tersebut akan diterima dalam kurun beberapa hari, bisa sampai dengan tepat waktu atau lebih lambat. Jika penerima juga mengirim surat yang berbeda kepada pengirim pada waktu berikutnya, bisa saja surat dari penerima yang sampai duluan. Tidak semua aplikasi membutuhkan koneksi antara dua belah pihak. Seperti halnya seorang penipu bisa saja mengirimkan satu email kepada banyak penerima. Tanpa peduli pesan tersebut dibaca oleh penerima. Layanan *connectionless* biasa disebut sebagai *datagram service*.

Kedua layanan tersebut dapat dikategorikan berdasarkan *reliability*. Penggunaan *connection-oriented* menjamin pengirim mengirimkan data kepada penerima, data yang dikirimkan dapat diterima secara lengkap, tidak ada kekurangan dan dapat menghindari duplikasi data. Terkadang memiliki *cost* yang besar untuk mengimplementasikan layanan ini. Sedangkan *connectionless* tidak memerlukan koneksi yang stabil antara pengirim dan penerima. Pengiriman data yang dilakukan berjalan secara lebih sederhana. Pengirim hanya perlu mengirim data, tidak memperdulikan data yang tidak sampai dengan jelas, duplikasi, dan tidak berurutan. Penggunaan *connectionless* lebih menguntungkan ketika terjadi *online meeting*, karena video yang tetap dapat terlihat dan diterima walaupun ada beberapa *pixels* yang salah. Dibandingkan jika harus menunggu keseluruhan video terlihat jelas menggunakan *connection-oriented*, tapi perlu menunggu lebih lama untuk menampilkannya.

2.4 TCP/IP Model

TCP/IP Model banyak digunakan oleh jaringan komputer secara menyeluruh dari internet di seluruh dunia. Ketika satelit dan jaringan radio hadir, protokol yang ada memiliki masalah dalam beroperasi dengannya, jadi diperlukan model arsitektur

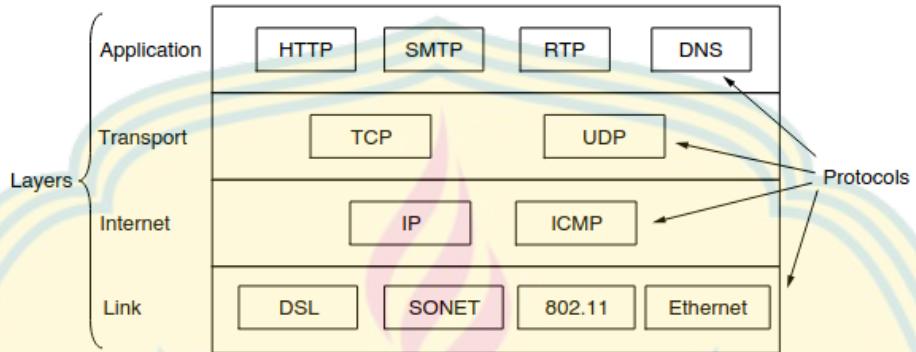
yang baru. Oleh karena itu, kemampuan TCP/IP dari awal adalah untuk dapat menghubungkan beberapa jaringan dengan lancar yang merupakan salah satu tujuan utama dari model ini. Arsitektur ini kemudian dikenal sebagai *TCP/IP Reference Model*.



Gambar 2.2: OSI dan TCP/IP Model

Pada Gambar 2.2 terdapat dua model, yaitu OSI dan TCP/IP. OSI (*Open Systems Interconnection*) dinamakan seperti itu karena berkaitan dengan menghubungkan *open system* atau sistem terbuka untuk berkomunikasi dengan sistem lainnya. Pada OSI model terdiri dari tujuh lapisan dan setiap lapisannya memiliki beberapa layanan untuk lapisan di atasnya. OSI model memiliki tiga konsep penting: *services*, *interfaces*, dan *protocols*. Perbedaan dari OSI dan TCP/IP adalah perubahan pada lapisannya, TCP/IP tidak memiliki lapisan physical, session, dan presentation. TCP/IP merupakan pembaruan dari OSI model. Terdapat empat lapisan pada TCP/IP dan contoh protokolnya pada Gambar 2.3.

*Mencerdaskan dan
Memartabatkan Bangsa*



Gambar 2.3: TCP/IP Model dan Protokolnya

2.4.1 *Link Layer*

Sebagai lapisan paling bawah dalam model TCP/IP lapisan *link* sering dibandingkan dengan kombinasi dari lapisan data *link* dan *physical*. Fungsi dari lapisan ini adalah untuk menyediakan layanan kepada lapisan jaringan atau internet agar dapat mentransfer data dari lapisan jaringan yang berada pada *source machine* menuju *destination machine*. Dan menghubungkan antara *physical* dan *logical network*.

Pada lapisan ini terdapat tiga protokol sederhana saat melakukan transmisi data, yaitu:

1. *Utopia: No Flow Control or Error Detection*

Pada protokol ini dibuat sederhana, dengan anggapan bahwa tidak ada kemungkinan terjadinya kesalahan. Data hanya ditransmisikan satu arah saja. Lapisan jaringan selalu siap untuk mentransmisikan dan menerima. Waktu pemrosesan dapat diabaikan. Ruang *buffer* yang tak terbatas. Dan saluran komunikasi antara lapisan *link* tidak pernah rusak atau kehilangan *frame*. Oleh karena itu disebut sebagai ‘Utopia’. Protokol terdiri dari dua prosedur yang berbeda, pengirim dan penerima. Itu pengirim berjalan di lapisan *link* dari *source machine*, dan penerima berjalan masuk lapisan *link* dari *destination machine*. Tidak ada *sequence numbers* atau *acknowledgements* yang digunakan. Pengirim berada dalam *while loop* yang tak terbatas yang hanya bertugas untuk memompa data keluar ke saluran secepat mungkin. *Body* dari *loop* ini terdiri dari tiga tindakan: *fetch packet* dari lapisan jaringan, buat *frame* keluar, dan mengirim *frame* dalam perjalanan. Protokol *utopia* tidak realistik

karena tidak menangani *flow control* atau *error detection*. Pengolahannya mirip dengan layanan *unacknowledged connectionless* yang bergantung pada lapisan yang lebih tinggi untuk menyelesaikan masalah ini, bahkan layanan *unacknowledged connectionless* akan melakukan beberapa deteksi kesalahan.

2. Adding Flow Control: Stop-and-Wait

Sekarang mengatasi masalah dengan mencegah pengirim membanjiri penerima dengan *frame* lebih cepat daripada kemampuan penerima untuk memprosesnya. Situasi ini bisa mudah terjadi dalam praktek sehingga mampu mencegahnya itu sangat penting. Saluran komunikasi masih dianggap bebas kesalahan dan lalu lintas data masih sederhana. Salah satu solusinya adalah membangun penerima yang cukup kuat untuk memproses aliran *frame back-to-back* yang cepat, harus memiliki *buffering* yang cukup dan kemampuan pemrosesan untuk berjalan pada *line rate* dan harus dapat melewati kumpulan *frame* yang diterima ke lapisan jaringan dengan cukup cepat. Namun, ini adalah solusi terburuk. Ini membutuhkan perangkat keras khusus dan dapat membuang-buang sumber daya jika penggunaan *link* sebagian besar rendah. Apalagi hanya menggeser masalah bertransaksi dengan pengirim yang terlalu cepat di tempat lain; dalam hal ini ke lapisan jaringan.

Solusi yang lebih umum dalam masalah ini adalah dengan membuat penerima menyediakan umpan balik kepada pengirim. Setelah paket dikirimkan ke lapisan jaringan, penerima mengirim *dummy frame* kepada pengirim, yang mengartikan bahwa pengirim dapat mengirim *frame* selanjutnya. Hal ini adalah contoh penggunaan dari protokol *flow control*. Protokol yang mana pengirim mengirim satu *frame* dan kemudian menunggu *acknowledgement* sebelum melanjutkan pengiriman disebut *stop-and-wait*.

3. Adding Error Correction: Sequence Numbers and ARQ

Sekarang pertimbangkan situasi normal dari saluran komunikasi membuat kesalahan. *Frame* mungkin rusak atau hilang sama sekali. Namun, asumsikan bahwa jika *frame* rusak saat transit, perangkat keras penerima akan mendektrinya. Jika *frame* yang rusak tiba di penerima, itu akan dibuang. Setelah beberapa saat, pengirim akan *time out* dan mengirim *frame* lagi. Proses ini akan diulang sampai *frame* tersebut akhirnya tiba dengan utuh. Tetapi, skema ini memiliki kesalahan fatal. Karena tujuan dari lapisan *link*

adalah untuk menyediakan komunikasi yang transparan dan bebas kesalahan antara proses lapisan jaringan. Lapisan jaringan pada perangkat A memberikan serangkaian paket ke lapisan *link*-nya, yang harus memastikan bahwa rangkaian paket yang identik dikirimkan ke lapisan jaringan pada perangkat B oleh lapisan *link*-nya. Secara khusus, lapisan jaringan pada B tidak memiliki cara untuk mengetahui bahwa suatu paket telah hilang atau digandakan, demikian lapisan *link* harus menjamin bahwa tidak ada kombinasi kesalahan transmisi, betapapun kecil kemungkinannya, yang dapat terjadi menyebabkan paket duplikat dikirim ke lapisan jaringan.

LAN banyak digunakan untuk interkoneksi, tetapi kebanyakan infrastruktur *wide area network* dibangun dari *point-to-point*. Pada LAN, protokol hubungan data yang ditemukan pada jalur *point-to-point* di internet dalam tiga situasi umum. Situasi pertama adalah ketika paket dikirim melalui sambungan *optical fiber* SONET di *wide area networks*. Sambungan ini banyak digunakan, misalnya, untuk menghubungkan *router* di lokasi yang berbeda jaringan ISP (*Internet Service Provider*). Situasi kedua adalah untuk sambungan ADSL (*Asymmetric Digital Subscriber Loop*) yang berjalan di *local loop* jaringan telepon. Situasi ketiga adalah untuk sambungan DOCSIS (*Data Over Cable Service Interface Specification*) di *local loop* jaringan kabel. Baik ADSL dan DOCSIS menghubungkan jutaan individu dan bisnis ke internet.

2.4.2 *Internet Layer*

Lapisan ini bertugas untuk mengizinkan *host* untuk menginjeksi paket ke jaringan dan dikirim ke destinasi tujuan yang juga memungkinkan berada pada jaringan yang berbeda. Pada lapisan ini menjelaskan *official packet format* dan protokol yang disebut dengan *Internet Protocol* (IP) dan protokol pendamping *Internet Control Message Protocol* (ICMP). Perutean paket yang dikirim pada lapisan ini terkadang menjadi kendala dalam sisi manajemen yang menyebabkan kemacetan atau lalu lintas yang padat pada saat pengiriman paket. Hal ini dapat ditangani dengan bantuan lapisan yang lebih tinggi, yakni lapisan *transport*.

2.4.2.1 *Quality of Services*

Solusi mudah untuk memberikan kualitas layanan yang baik adalah dengan membangun jaringan kapasitas yang cukup untuk lalu lintas apa pun yang akan dilemparkan padanya. Nama untuk solusi ini adalah *overprovisioning* atau

penyediaan yang berlebihan. Jaringan yang dihasilkan akan membawa lalu lintas aplikasi tanpa kerugian yang signifikan, dengan asumsi skema perutean yang layak, akan mengirimkan paket dengan latensi rendah. Performa tidak menjadi lebih baik dari ini. Sampai batas tertentu, sistem telepon *overprovisioned* karena jarang mengangkat telepon dan tidak mendapatkan nada panggil secara instan. Ada begitu banyak kapasitas yang tersedia sehingga tuntutan hampir selalu dapat dipenuhi. Empat masalah harus ditangani untuk memastikan kualitas layanan: aplikasi apa yang dibutuhkan dari jaringan, bagaimana mengatur lalu lintas yang masuk ke jaringan, cara memesan sumber daya di *router* untuk menjamin kinerja, dan apakah jaringan dapat dengan aman menerima lebih banyak lalu lintas.

Mampu mengatur bentuk lalu lintas yang ditawarkan adalah awal yang baik. Namun, untuk memberikan jaminan kinerja, harus mencadangkan sumber daya yang cukup di sepanjang rute yang diambil paket melalui jaringan. Untuk melakukan ini, mengasumsikan paket aliran mengikuti rute yang sama. Menyemprotkannya ke *router* secara acak membuatnya sulit untuk menjamin apa pun. Akibatnya, sesuatu yang mirip dengan virtual-circuit harus diatur dari sumber ke tujuan, dan semua paket yang termasuk aliran harus mengikuti rute ini. Algoritma yang mengalokasikan sumber daya *router* di antara paket-paket aliran dan diantara aliran yang bersaing disebut algoritma penjadwalan (scheduling) paket. Tiga berbeda jenis sumber daya berpotensi dicadangkan untuk aliran yang berbeda: *bandwidth*, *buffer space*, dan CPU *cycles*.

2.4.2.2 *Internet Protocols*

Kegiatan yang dilakukan pada jejaring internet erat kaitannya dengan transfer data, data yang dimaksudkan dapat berupa informasi yang ingin disampaikan ke tempat tujuan. Data dapat terkirim dari satu tempat ke tempat lainnya memerlukan sebuah aturan supaya dapat sampai ke tujuan yang tepat. Oleh karena itu, dikenalkan *internet protocol*. *Internet Protocol* (IP) adalah sebuah protokol internet yang membuat berbagai aturan yang bertujuan untuk mengirim data dari satu tujuan ke tujuan yang sesuai. Data yang ditransfer melalui jejaring internet dibagi menjadi beberapa bagian kecil yang disebut packets. Setiap perangkat yang terhubung ke internet memiliki sebuah alamat yang disebut sebagai *IP Address*. *IP Address* bertujuan untuk menetapkan bahwa masing - masing perangkat memiliki alamatnya tersendiri, guna kelancaran dan ketepatan dalam pengiriman data.

Internet Protocol Version 4 (IPv4) dikembangkan pada awal tahun 1980-an. IPv4 banyak digunakan oleh banyak perangkat saat ini. Pada awal pengembangannya IPv4 ditetapkan memiliki 32-bit panjang alamat dapat menampung sekitar 4,3 miliar alamat yang berbeda, dengan anggapan bahwa tidak mungkin memerlukan lebih banyak total alamat ke depannya. Tetapi, pada awal tahun 2011 dengan meningkatnya total pengguna teknologi seperti smartphone dan laptop yang terhubung dengan internet menjadikan jumlah alamat IPv4 yang tersedia mulai menipis. Beberapa cara dilakukan untuk mengatasi hal ini, seperti menggunakan kembali *address block* menggunakan *multiple layers* dengan *Network Address Translation (NAT)*, seperti yang dilakukan oleh *Internet Service Provider (ISP)*. NAT menjadikan setiap routers yang terhubung dengan berbagai perangkat hanya memiliki satu IP saja, dengan anggapan mewakili dari semua perangkat yang terhubung. Dengan demikian ISP tidak perlu untuk menetapkan IP *address* yang berbeda untuk masing - masing perangkat yang menggunakan layanan internet dari provider tersebut.

Internet Protocol Version 6 (IPv6) adalah versi terbaru dari IPv4, dikembangkan sebagai penerus IPv4. Pengembangan IPv6 dimulai pada tahun 1991 dan rampung pada tahun 1997 oleh *Internet Engineering Task Force (IETF)*. Digunakan pertama kali oleh *Internet Corporation for Assigned Names and Numbers (ICANN)* pada tahun 2004. Kapasitas dari alamat IPv6 lebih besar sampai 128-bit, dapat menampung sekitar 2^{128} alamat yang berbeda. IPv6 dikembangkan agar setiap pengguna dapat memiliki alamat IP sendiri. Alasan utama penggunaan IPv6 ini karena berkurangnya kapasitas IPv4 semakin hari. Dengan penggunaan IPv6 seperti halnya memiliki kapasitas IP *address* yang tidak terbatas untuk jumlah perangkat yang kian meningkat setiap tahunnya.

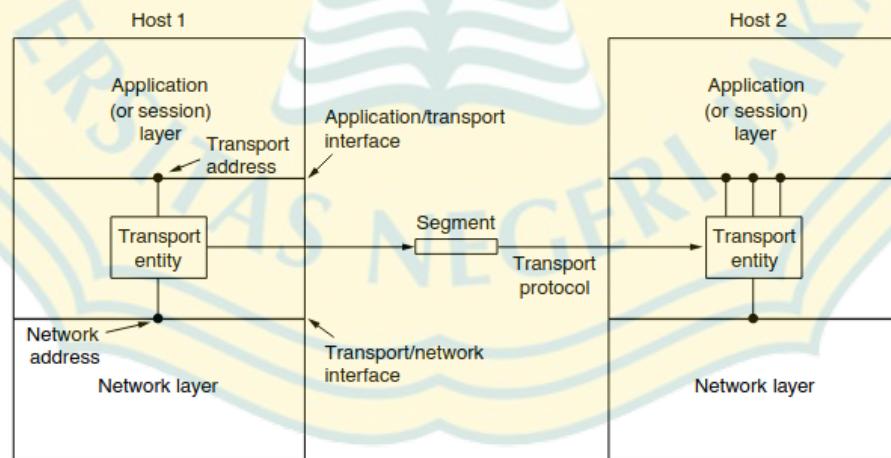
2.4.3 Transport Layer

Lapisan ini didesain agar peer pada source dan destination host dapat berkomunikasi. Terdapat dua *end-to-end transport protocol* pada lapisan ini: TCP (*Transmission Control Protocol*) yang menggunakan connection-oriented, UDP (*User Datagram Protocol*) yang menggunakan connectionless. Tujuan utama dari lapisan ini adalah untuk menyediakan layanan yang *efficient*, *reliable*, dan *cost-effective* data transmission kepada penggunanya yang nanti akan diproses lebih lanjut pada lapisan *application*.

Perangkat keras atau lunak yang melakukan tugas pada lapisan ini disebut dengan *transport entity* atau entitas transportasi. Entitas transportasi dapat terletak pada kernel operasi sistem, *library package* yang terikat ke jaringan aplikasi, proses pengguna yang terpisah, atau *network interface card*.

2.4.3.1 Services and Connections

Lapisan *transport* terletak di antara lapisan *application* dan network atau internet. Pada lapisan internet terdapat dua layanan yang sama dengan lapisan *transport* yaitu *connection-oriented* dan *connectionless* kedua lapisan ini memiliki kesamaan. Tetapi perlu ada pemisahan antara lapisan internet dan *transport* padahal memiliki layanan yang sama. Hal itu disebabkan oleh lapisan *transport* berjalan sepenuhnya pada perangkat pengguna, sedangkan lapisan internet berjalan pada *routers*. Akan terjadi permasalahan ketika terdapat sesuatu seperti packets lost yang menyebabkan *routers* mengalami kerusakan. Sementara pengguna tidak memiliki kendali untuk mengontrol hal tersebut yang berada pada lapisan internet sebab berada pada *routers*.



Gambar 2.4: Jaringan Transportasi dan Lapisan Aplikasi

Oleh karena itu, langkah yang baik untuk mencegah hal ini adalah dengan membuat satu lapisan baru di atas lapisan internet yaitu lapisan transportasi yang hanya khusus untuk mengelola sistem transportasi data. Jadi, ketika terjadi packets lost masalah tersebut dapat teratasi dengan entitas transportasi yang mentransmisikan kembali packet yang hilang itu. Dengan adanya lapisan transportasi, para programmer tidak perlu cemas ketika berhadapan dengan jaringan

internet yang berbeda. Karena programmer dapat mengkonfigurasikan kodennya sesuai dengan standar yang diperlukan.

Untuk memungkinkan pengguna mengakses layanan transportasi, lapisan transportasi harus menyediakan beberapa operasi ke program aplikasi, yaitu *transport service interface*. Setiap layanan transportasi memiliki *interface* sendiri. Layanan transportasi mirip dengan layanan jaringan, tetapi ada juga beberapa perbedaan penting. Perbedaan utama adalah bahwa layanan jaringan dimaksudkan untuk memodelkan layanan yang ditawarkan oleh *real network*. *Real network* dapat kehilangan paket, sehingga layanan jaringan umumnya tidak dapat diandalkan.

Layanan transportasi *connection-oriented*, sebaliknya, dapat diandalkan. Tentu saja, *real network* tidak bebas dari kesalahan, tetapi justru itulah tujuan dari lapisan transportasi untuk menyediakan layanan yang dapat diandalkan di atas jaringan yang tidak dapat diandalkan. Layanan transportasi *connection-oriented* adalah tentang menyembunyikan ketidak sempurnaan layanan jaringan sehingga proses pengguna hanya dapat menganggap adanya aliran *error-free bit* bahkan ketika mereka berada di perangkat yang berbeda. Lapisan transportasi juga dapat menyediakan layanan (datagram) yang tidak dapat diandalkan atau unreliable. Ada beberapa aplikasi, seperti komputasi *client-server* dan *streaming multimedia*, yang dibangun di atas layanan transportasi *connectionless*. Perbedaan antara layanan jaringan dan layanan transportasi adalah kepada siapa layanan tersebut ditujukan.

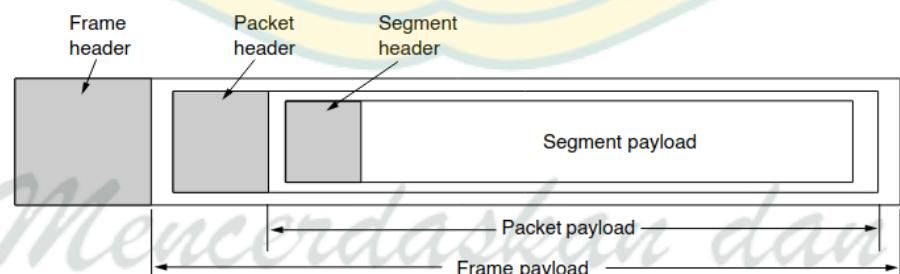
Tabel 2.1: Primitive pada lapisan koneksi

Primitive	Packet sent	Berarti
<i>LISTEN</i>	(none)	Memblokir sampai ada proses yang terhubung
<i>CONNECT</i>	<i>CONNECTION REQ.</i>	Secara aktif mencoba membuat koneksi
<i>SEND</i>	<i>DATA</i>	Mengirim informasi
<i>Routing</i>	Setiap paket diarahkan secara independen	Route dipilih ketika VC sudah siap dan semua paket akan mengikuti
<i>RECEIVE</i>	(none)	Memblokir sampai paket DATA tiba

<i>Quality of service</i>	Sulit	Mudah, jika memiliki sumber daya yang cukup untuk dialokasikan secara advance untuk setiap VC
<i>DISCONNECT</i>	<i>DISCONNECT REQ.</i>	Meminta untuk memutuskan koneksi

Primitive adalah tipe data sederhana yang menjadi dasar dari semua tipe data lain. Untuk melihat bagaimana *primitives* digunakan dengan server dan *remote clients*. Diawali dengan server mengeksekusi *LISTEN*. Memblokir sampai ada klien yang terhubung. Ketika klien mencoba untuk terhubung, maka akan mengeksekusi *CONNECT*. Entitas transportasi melakukan primitif ini dengan memblokir caller dan mengirim paket ke server. Terbungkus dalam muatan paket ini adalah pesan lapisan transportasi untuk entitas transportasi server. Terminologi *segment* untuk pesan yang dikirim dari entitas transportasi ke entitas transportasi. TCP, UDP, dan protokol Internet lainnya menggunakan istilah ini.

Dengan demikian, segmen terkandung dalam paket. Pada gilirannya, paket-paket ini terkandung dalam *frame* (dipertukarkan oleh lapisan link). Ketika sebuah *frame* tiba, lapisan link memproses *header frame*, jika alamat tujuan cocok untuk pengiriman lokal, maka meneruskan konten *frame payload* ke entitas jaringan. Entitas jaringan juga memproses *header* paket dan kemudian meneruskan konten muatan paket ke entitas transportasi.



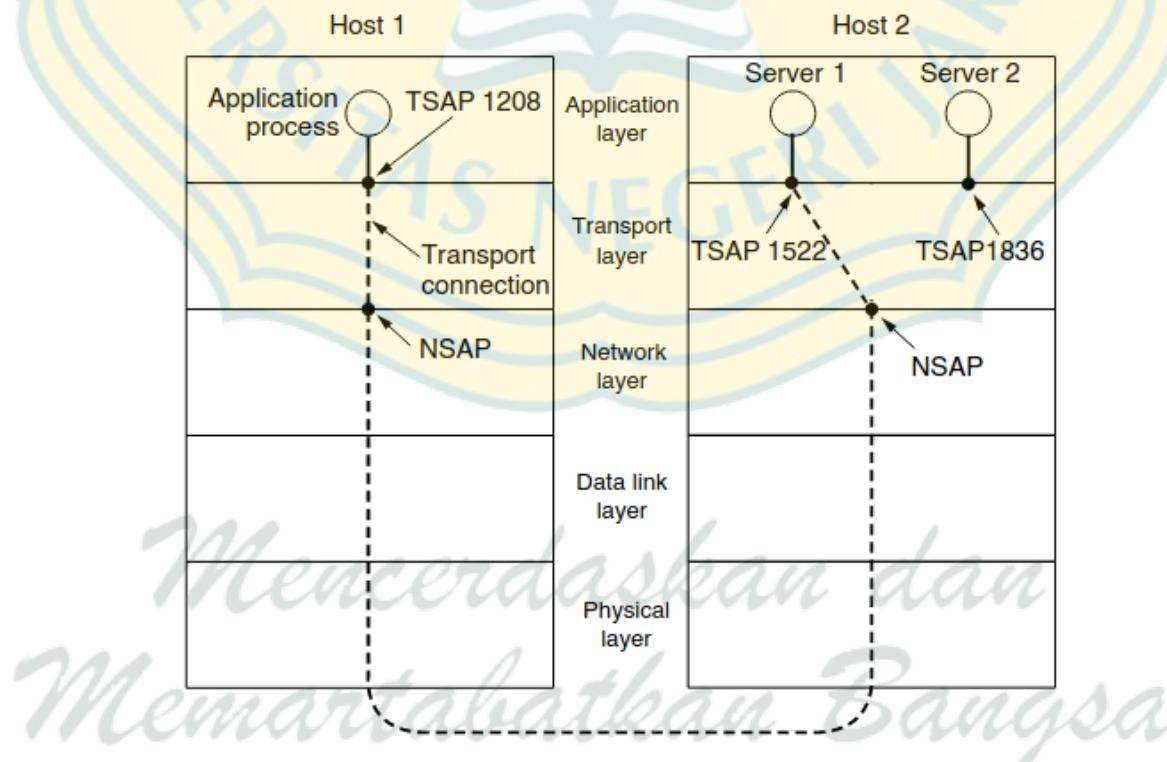
Gambar 2.5: Segmen Paket dan *Frame*

Memanggil *CONNECT* dari sisi klien menyebabkan segmen *CONNECTION REQUEST* untuk dikirim ke server. Ketika tiba, entitas transportasi memeriksa untuk

melihat bahwa server diblokir pada *LISTEN*. Kemudian membuka blokir server dan mengirimkan segmen *CONNECTION ACCEPTED* kembali ke klien. Saat segmen ini tiba, klien akan membuka blokirnya dan sambungan dibuat.

Data sekarang dapat dipertukarkan menggunakan *SEND* dan *RECEIVE* primitif. Dalam bentuk paling sederhana, salah satu pihak dapat melakukan (blocking) *RECEIVE* untuk menunggu pihak lain untuk melakukan *SEND*. Saat segmen tiba, penerima membuka blokirnya. Itu bisa memproses segmen dan mengirim balasan. Selama kedua belah pihak dapat melacak giliran siapa yang mengirim, skema ini berfungsi dengan baik.

Ketika sebuah proses aplikasi ingin mengatur koneksi ke proses aplikasi jarak jauh, itu harus menentukan proses mana pada titik akhir jarak jauh yang akan dihubungkan. Metode yang biasanya digunakan adalah untuk menentukan alamat transportasi yang dapat digunakan oleh proses mendengarkan permintaan koneksi. Di Internet, titik akhir ini disebut port. Istilah TSAP (*Transport Service Access Point*) untuk mengartikan titik akhir spesifik pada lapisan transportasi. Titik akhir analog dalam jaringan lapisan (yaitu, alamat lapisan jaringan) tidak mengherankan disebut NSAPs (*Network Service Access Points*). Alamat IP adalah contoh NSAP.

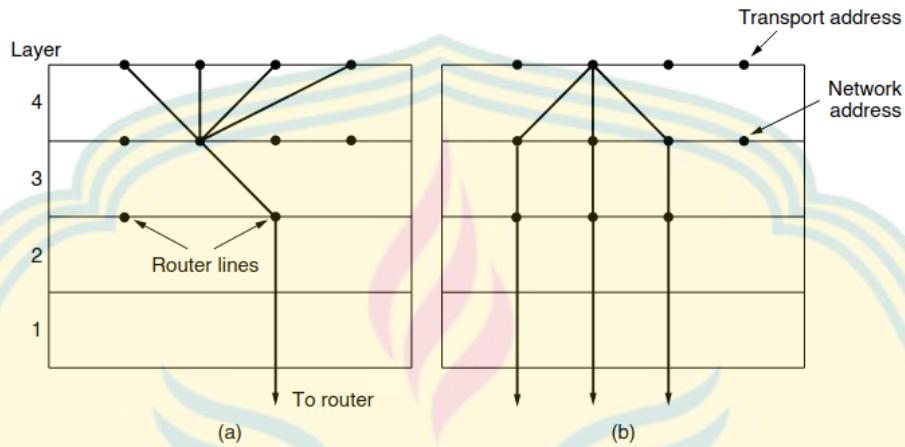


Gambar 2.6: Koneksi Transportasi TSAP dan NSAP

Application process, baik klien maupun server, dapat melampirkan dirinya sendiri ke TSAP lokal untuk membuat sambungan ke TSAP jarak jauh. Koneksi ini dijalankan melalui NSAP pada setiap *host*. Tujuan dari memiliki TSAP adalah bahwa di beberapa jaringan, setiap komputer memiliki NSAP tunggal, jadi beberapa cara diperlukan untuk membedakan beberapa titik akhir transportasi yang berbagi itu NSAP. Skenario yang mungkin untuk koneksi transportasi adalah sebagai berikut:

1. Proses server email menempel pada TSAP 1522 di *host 2* untuk menunggu panggilan masuk atau *incoming call*. Bagaimana suatu proses menempel pada TSAP berada di luar model jaringan dan sepenuhnya bergantung pada operasi sistem lokal. Panggilan seperti *LISTEN* dapat digunakan.
2. *Application Process* pada *host 1* ingin mengirim pesan email, jadi itu menempel pada TSAP 1208 dan mengeluarkan permintaan *CONNECT*. Permintaan yang menentukan TSAP 1208 pada *host 1* sebagai sumber dan TSAP 1522 pada *host 2* sebagai tujuan. Tindakan ini pada akhirnya menghasilkan koneksi transportasi yang dibuat antara *application process* dan server.
3. *Application process* mengirim *mail message*.
4. *Mail server* merespon kalau pesan itu akan dikirimkan.
5. Koneksi transportasi dilepaskan.

Multiplexing, atau berbagi beberapa percakapan melalui koneksi, *virtual circuits*, dan *physical link* berperan dalam beberapa lapisan arsitektur jaringan. Pada lapisan transportasi, kebutuhan akan *multiplexing* dapat muncul dalam beberapa cara. Jika hanya satu alamat jaringan yang tersedia di sebuah *host*, semua koneksi transportasi pada perangkat harus menggunakaninya. Ketika sebuah segmen masuk, ada caranya diperlukan untuk mengetahui proses mana yang harus diberikan. Situasi ini, disebut *multiplexing*.

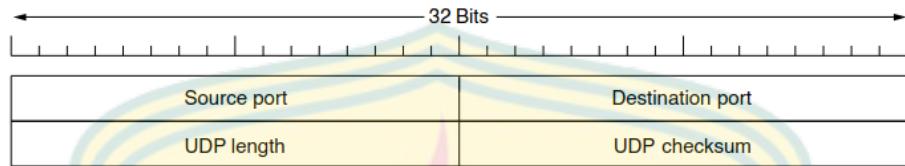


Gambar 2.7: (a) Multiplexing, (b) Inverse Multiplexing

Multiplexing juga berguna di lapisan transport karena alasan lain. Misalkan, sebuah *host* memiliki banyak jalur jaringan yang dapat digunakannya. Jika pengguna membutuhkan lebih banyak *bandwidth* atau lebih banyak *reliability* daripada yang dapat disediakan oleh salah satu jalur jaringan, jalan keluarnya adalah memiliki koneksi yang mendistribusikan lalu lintas di antara banyak jalur jaringan dengan basis *round-robin*, seperti ditunjukkan pada Gambar 2.7 (b). Modus operandi disebut *inverse multiplexing*. Dengan k koneksi jaringan terbuka, *bandwidth* efektif dapat ditingkatkan dengan faktor k. Contoh *inverse multiplexing* adalah SCTP (*Stream Control Transmission Protocol*) yang dapat menjalankan koneksi menggunakan beberapa antarmuka jaringan. Sebaliknya, TCP menggunakan titik akhir jaringan tunggal. *Inverse multiplexing* juga ditemukan pada lapisan *link*, ketika beberapa *low-rate links* digunakan secara paralel sebagai satu tautan cepat.

2.4.3.2 User Datagram Protocol

Paket protokol internet mendukung protokol transportasi *connectionless* yang disebut UDP. UDP menyediakan cara bagi aplikasi untuk mengirim enkapsulasi datagram IP tanpa harus membuat koneksi. UDP mentransmisikan segmen yang terdiri dari *header* 8-byte diikuti oleh *payload*. Pada Gambar 2.8, ada *port* yang berfungsi untuk mengidentifikasi titik dalam perangkat sumber dan tujuan. Ketika paket UDP tiba, *payload* diserahkan ke proses yang melekat pada *port* tujuan. Lampiran ini terjadi ketika primitif *BIND* atau yang serupa digunakan.

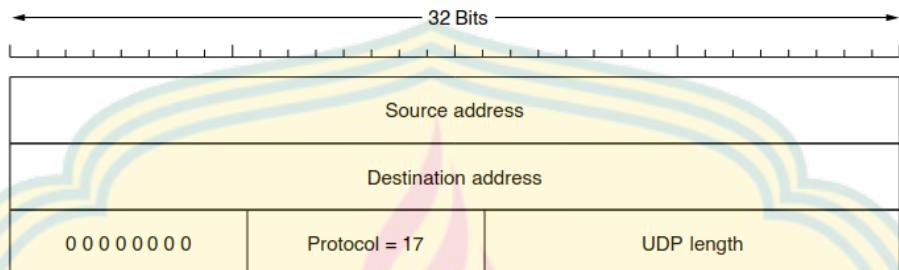


Gambar 2.8: UDP Header

Port sumber sangat dibutuhkan ketika balasan harus dikirim kembali ke sumber. Dengan menyalin bidang *port* sumber dari *incoming segment* ke bidang *port* tujuan dari *outgoing segment*, proses pengiriman balasan dapat menentukan proses mana pada perangkat pengirim yang akan menerimanya. Bidang panjang UDP menyertakan *header* 8-byte dan data. Minimal panjangnya 8 byte, untuk menutupi *header*. Panjang maksimumnya adalah 65.515 byte, yang lebih rendah dari jumlah terbesar yang muat dalam 16-bit karena batas ukuran pada paket IP.

Checksum opsional juga disediakan untuk keandalan ekstra. Melakukan *checksum header*, data, dan *conceptual IP pseudoheader*. Saat melakukan perhitungan ini, *Checksum field* disetel ke nol dan bidang data diisi dengan byte nol tambahan jika panjangnya adalah angka ganjil. Algoritma *checksum* hanya menjumlahkan semua kata 16-bit dalam komplemen satu dan mengambil komplemen dari jumlah tersebut. Konsekuensinya, saat penerima melakukan perhitungan pada seluruh segmen, termasuk *checksum field*, hasilnya harus 0. Jika *checksum* tidak dihitung, disimpan sebagai 0, 0 yang benar dihitung disimpan sebagai semua 1.

Pseudoheader untuk kasus IPv4 ditunjukkan pada Gambar 2.9. Berisi alamat IPv4 32-bit dari perangkat sumber dan tujuan, nomor protokol untuk UDP (17), dan jumlah byte untuk segmen UDP (termasuk header). Menyertakan *pseudoheader* dalam komputasi *checksum* UDP membantu mendeteksi paket yang salah kirim, tetapi menyatakannya juga melanggar hierarki protokol karena alamat IP di dalamnya milik lapisan IP, bukan milik lapisan UDP. TCP menggunakan *pseudoheader* yang sama untuk *checksum*-nya.



Gambar 2.9: IPv4 Pseudoheader di UDP Checksum

2.4.3.3 User Datagram Protocol

TCP dirancang khusus untuk menyediakan aliran byte end-to-end melalui *unreliable internetwork*. Sebuah jaringan internet berbeda dari satu jaringan karena bagian yang berbeda mungkin sangat berbeda topologi, *bandwidth*, *delays*, ukuran paket, dan parameter lainnya. TCP dulu dirancang untuk secara dinamis beradaptasi dengan properti dari *internetwork* dan menjadi kuat menghadapi berbagai macam kegagalan. Layanan TCP diperoleh oleh pengirim dan penerima yang menciptakan akhir poin, disebut socket. Setiap socket memiliki nomor socket (alamat) yang terdiri dari alamat IP *host* dan nomor 16-bit lokal ke *host* tersebut, yang disebut *port*. *Port* adalah nama TCP untuk TSAP. Agar layanan TCP menjadi diperoleh, koneksi harus dibuat secara eksplisit antara socket pada satu perangkat dan di perangkat yang lain. Socket dapat digunakan untuk beberapa koneksi sekaligus.

Nomor *port* di bawah 1024 dicadangkan untuk layanan standar yang biasanya bisa hanya dapat dimulai oleh *privileged users*. Disebut sebagai *well-known ports*. Berikut adalah contoh dari beberapa *port* yang sudah digunakan.

*Mencerdaskan dan
Memartabatkan Bangsa*

Tabel 2.2: Well-known ports

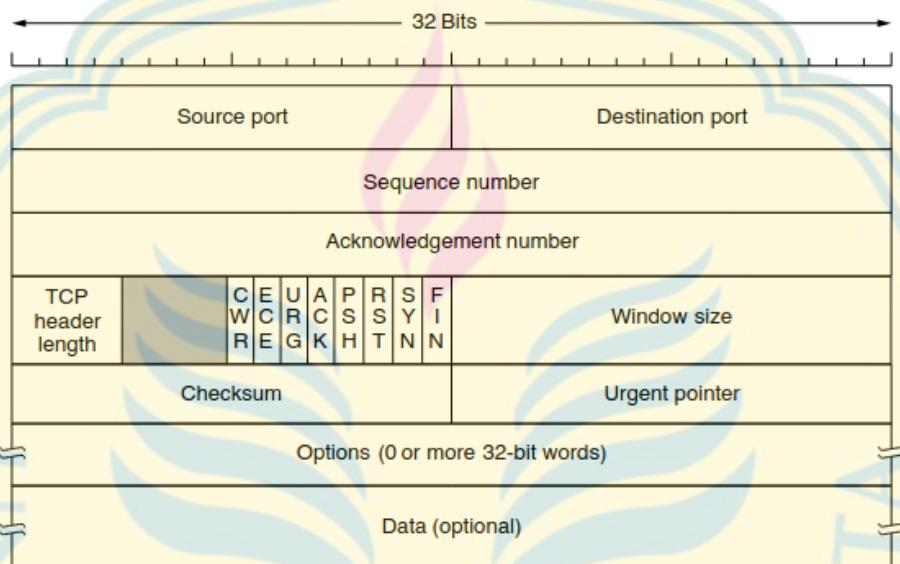
Port	Protokol	Kegunaan
20, 21	FTP	File transfer
22	SSH	Remote login, pengganti untuk Telnet
25	SMTP	Email
80	HTTP	World Wide Web
110	POP-3	Remote email access
143	IMAP	Remote email access
443	HTTPS	Secure Web (HTTPS)

Semua koneksi TCP adalah *full duplex* dan *point-to-point*. *Full duplex* berarti bahwa lalu lintas dapat berjalan di kedua arah secara bersamaan. *Point-to-point* artinya setiap koneksi memiliki tepat dua titik akhir. TCP tidak mendukung *multicasting* atau *broadcasting*. Koneksi TCP adalah *byte stream*, bukan *message stream*. *Message boundaries* tidak dipertahankan dari ujung ke ujung.

Fitur utama dari TCP, dan yang mendominasi desain protokol, adalah bahwa setiap byte pada koneksi TCP memiliki nomor urut 32-bit sendiri. Ketika internet dimulai, jalur antar router sebagian besar adalah 56-kbps *leased line*, jadi *host* membutuhkan waktu lebih dari 1 minggu untuk menggilir nomor urut. Pada kecepatan jaringan modern, nomor urut dapat dikonsumsi dengan *alarming rate*. Nomor urut 32-bit yang terpisah dibawa pada paket untuk posisi *sliding window* dalam satu arah dan untuk *acknowledgement* dalam arah sebaliknya.

Entitas TCP pengirim dan penerima bertukar data dalam bentuk segmen. Segmen TCP terdiri dari header 20-byte tetap (ditambah bagian opsional) yang diikuti dengan nol atau lebih byte data. Perangkat lunak TCP memutuskan seberapa besar seharusnya segmen menjadi. Itu dapat mengumpulkan data dari beberapa penulisan menjadi satu segmen atau dapat membagi data dari satu penulisan ke beberapa segmen. Dua batasan membatasi ukuran segmen. Pertama, setiap segmen, termasuk header TCP, harus sesuai dengan muatan IP 65.515 byte. Kedua, setiap link memiliki MTU (*Maximum Transfer Unit*). Setiap segmen harus muat di MTU di

pengirim dan penerima sehingga dapat dikirim dan diterima di paket tunggal yang tidak terfragmentasi. Dalam praktiknya, MTU umumnya berukuran 1500 byte.



Gambar 2.10: TCP Header

2.4.4 Application Layer

Pada lapisan ini mengandung *higher-level protocol*: *virtual terminal* (TELNET), file transfer (FTP), *electronic mail* (SMTP), dan *Domain Name System* (DNS). Hilangnya lapisan *session* dan *presentation* disatukan pada lapisan application, jadi tidak semata - mata hilang begitu saja. Karena kedua lapisan tersebut tidak banyak berguna untuk sebagian besar aplikasi, oleh karena itu dihilangkan. Pada lapisan ini secara langsung memberikan layanan kepada pengguna seperti menggunakan internet, melakukan operasi pada perangkat lunak, dan kegiatan lainnya.

2.4.4.1 World Wide Web

World Wide Web (WWW) adalah arsitektur kerangka kerja untuk mengakses konten tertaut yang tersebar lebih dari jutaan mesin melalui internet. Web dimulai pada tahun 1989 di CERN (Conseil Européen pour la Recherche Nucléaire), Pusat Riset Nuklir Eropa. Ide awalnya adalah untuk membantu tim besar, seringkali dengan anggota yang banyak atau banyak negara dan zona waktu, berkolaborasi menggunakan koleksi yang terus berubah laporan, cetak biru, gambar, foto, dan

dokumen lain yang dihasilkan oleh eksperimen dalam fisika partikel. Proposal untuk Web dokumen tertaut berasal fisikawan CERN Tim Berners-Lee. Prototipe pertama (berbasis teks) telah beroperasi 18 bulan kemudian. Demonstrasi publik yang diberikan pada konferensi Hypertext '91 menarik perhatian peneliti lain, yang membuat Marc Andreessen di University of Illinois mengembangkan browser grafis pertama. Itu disebut Mosaic dan dirilis pada Februari 1993.

Dari sudut pandang pengguna, Web terdiri dari koleksi yang luas dan mendunia dalam bentuk halaman web. Setiap halaman biasanya berisi tautan ke ratusan objek lain, yang dapat di-hosting di server mana pun di internet, di mana pun di dunia. Objek - objek ini mungkin berupa teks dan gambar lain, tetapi saat ini juga menyertakan berbagai macam objek, termasuk iklan dan skrip pelacakan. Halaman juga dapat ditautkan ke halaman web lain; pengguna dapat mengikuti tautan dengan mengkliknya, yang kemudian membawa mereka ke halaman yang ditunjuk. Proses ini dapat diulang tanpa batas. Gagasan memiliki satu halaman menunjuk ke yang lain, sekarang disebut hypertext, ditemukan oleh M.I.T. profesor teknik elektro, Vannevar Bush, pada tahun 1945. Ini jauh sebelum internet ditemukan.

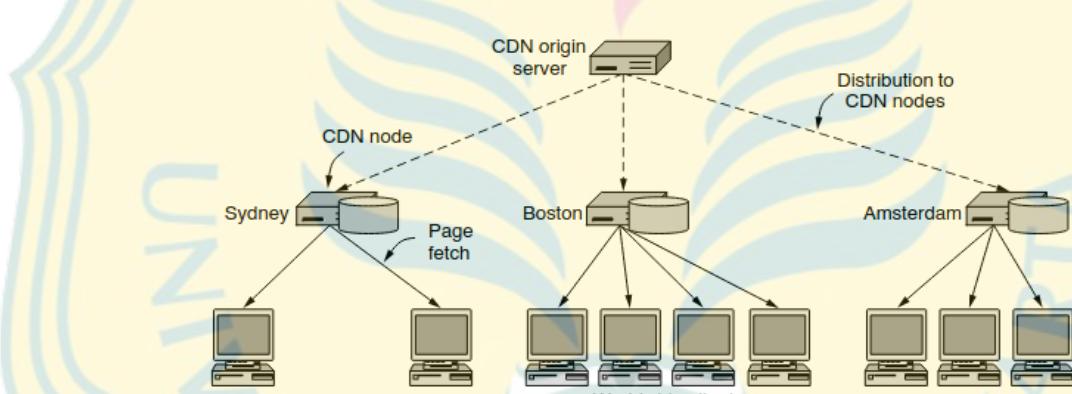
2.4.4.2 Content and Internet Traffic

Internet digunakan untuk semua tentang komunikasi *point-to-point*, seperti jaringan telepon. Banyak orang telah menggunakan email untuk berkomunikasi satu sama lain sejak lama, dan sekarang menggunakan video dan *voice over IP*. Namun, sejak web tumbuh, internet telah menjadi lebih dari sekadar tentang konten daripada komunikasi. Banyak orang menggunakan Web untuk mencari informasi, dan ada banyak sekali pengunduhan musik, video, dan materi lainnya. Karena tugas mendistribusikan konten berbeda dengan *point-to-point* komunikasi, itu menempatkan persyaratan yang berbeda pada jaringan.

Teknik yang digunakan untuk distribusi konten telah berkembang dalam waktu ke waktu. Di awal pertumbuhan web, popularitasnya hampir berakhir. Lagi permintaan konten menyebabkan server dan jaringan sering kelebihan beban. Banyak orang mulai menyebut WWW sebagai *World Wide Wait*. Untuk mengurangi penundaan tidak berujung, para peneliti mengembangkan arsitektur yang berbeda untuk menggunakan *bandwidth* untuk mendistribusikan konten. Arsitektur umum untuk mendistribusikan arsitektur konten adalah CDN (*Content Delivery Network*), terkadang juga disebut *Content Distribution Network*. CDN secara efektif adalah

kumpulan *cache* terdistribusi yang sangat besar, yang biasanya berfungsi konten langsung ke klien. Cara lain untuk mendistribusikan konten adalah melalui jaringan P2P (*Peer-to-Peer*), di mana komputer saling menyajikan konten, biasanya tanpa penyediaan terpisah pada server atau titik pusat kontrol apa pun.

CDN mengubah gagasan *caching web* tradisional. Alih-alih meminta klien mencari salinan halaman yang diminta di *cache* terdekat, provider menempatkan salinan halaman dalam satu set node di lokasi yang berbeda dan mengarahkan klien untuk menggunakan node terdekat sebagai server.



Gambar 2.11: CDN Tree

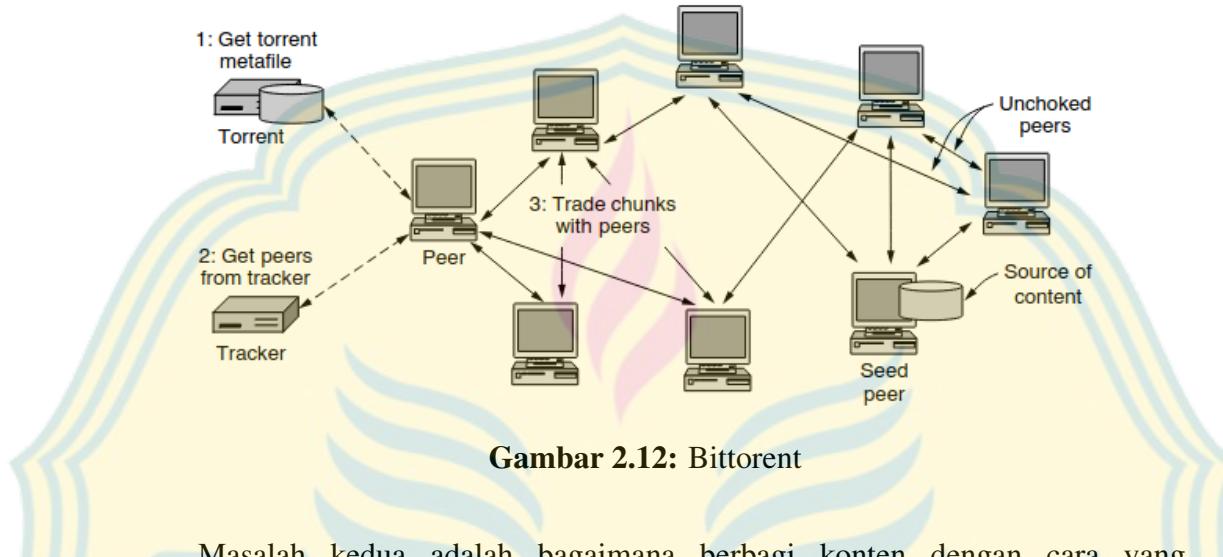
Menggunakan *tree structure* memiliki tiga keuntungan. Pertama, distribusi konten bisa ditingkatkan ke klien sebanyak yang diperlukan dengan menggunakan lebih banyak *node* di CDN, dan lebih banyak level di pohon saat distribusi di antara *node* CDN menjadi hambatan. Tidak peduli berapa banyak klien yang ada, struktur pohnnya efisien. *Origin server* tidak kelebihan beban karena berbicara ke banyak klien melalui pohon *node* CDN, tidak harus menjawab setiap permintaan untuk sebuah halaman dengan sendirinya. Kedua, setiap klien mendapatkan kinerja yang baik dengan mengambil halaman dari server terdekat dibanding server yang jauh. Ini karena waktu bolak-balik untuk menyiapkan koneksi lebih pendek, lambat laun TCP meningkat lebih cepat karena perjalanan bolak-balik dengan waktu yang lebih pendek, dan jalur jaringan yang lebih pendek cenderung melewati daerah kemacetan di internet. Terakhir, total beban yang ditempatkan di jaringan juga disimpan secara minimal. Jika *node* CDN ditempatkan dengan baik, lalu lintas untuk halaman tertentu harus melewati setiap bagian dari jaringan hanya sekali.

2.4.4.3 Peer-to-Peer Network

Ide dasar dari jaringan berbagi file P2P adalah bahwa banyak komputer bersatu dan menyatukan sumber daya mereka untuk membentuk sistem distribusi konten. Komputer seringkali hanya komputer rumahan. Mereka tidak perlu seperti mesin di pusat data internet. Komputer disebut rekan atau *peers* karena masing - masing seseorang dapat secara bergantian bertindak sebagai klien ke *peer* lain, mengambil kontennya, dan sebagai server, menyediakan konten ke *peer* lain. Apa yang membuat sistem *peer-to-peer* menarik adalah bahwa tidak ada infrastruktur khusus, tidak seperti di CDN. Semua orang berpartisipasi dalam tugas mendistribusikan konten, dan seringkali tidak ada titik pusat kendali. Banyak orang yang tertarik dengan teknologi P2P. Alasannya bukan karena dibutuhkan perusahaan besar untuk menjalankan CDN, sementara siapa pun yang memiliki komputer dapat bergabung dengan jaringan P2P. Jaringan P2P yang memiliki kapasitas yang luar biasa untuk mendistribusikan konten.

Protokol BitTorrent dikembangkan oleh Bram Cohen pada tahun 2001 untuk memungkinkan satu set *peer* berbagi file dengan cepat dan mudah. Ada banyak klien yang tersedia secara bebas yang menggunakan protokol ini, sama seperti banyak *browser* yang menggunakan protokol HTTP ke *web server*. Terdapat tiga masalah yang harus selesaikan dalam hal membagikan konten ini: bagaimana caranya agar *peer* dapat mencari *peer* lainnya ketika ingin mengunduh konten?, bagaimana konten dapat direplikasi oleh *peer* untuk menyediakan kecepatan mengunduh untuk semua orang?, dan bagaimana untuk mendorong sesama *peer* untuk mengunggah konten ke orang lain seperti mengunduh konten untuk diri mereka sendiri?

Masalah pertama ada karena tidak semua *peer* akan memiliki semua konten. Itu pendekatan yang diambil di BitTorrent adalah untuk setiap penyedia konten untuk membuat konten deskripsi disebut torrent. Torrent jauh lebih kecil dari kontennya, dan memang demikian digunakan oleh *peer* untuk memverifikasi integritas data yang diunduh dari *peer* lain. Pengguna lain yang ingin mengunduh konten harus mendapatkan torrent terlebih dahulu.



Gambar 2.12: BitTorrent

Masalah kedua adalah bagaimana berbagi konten dengan cara yang memberikan unduhan cepat. Kumpulan *peer* disebut *swarm*. Saat *swarm* pertama kali terbentuk, beberapa *peer* harus memiliki semua bongkahan itu buat isinya. *Peer* ini disebut *seeder*. *Peer* lainnya yang bergabung dengan *swarm* tidak akan memiliki *chunks*, *peer* yang mengunduh konten. Sementara *peer* yang berpartisipasi dalam *swarm*, secara bersamaan mengunduh *chunks* yang hilang dari *peer* lain, dan mengunggah potongan yang dimilikinya ke *peer* lain yang membutuhkan. *Trading* ini ditunjukkan pada tahap ketiga dari Gambar 2.12. Seiring waktu, *peer* mengumpulkan lebih banyak *chunk* hingga semuanya berhasil diunduh. *Peer* dapat meninggalkan *swarm* kapan saja. Biasanya *peer* akan tinggal untuk waktu yang singkat setelah menyelesaikan unduhannya sendiri. Dengan *peers* yang datang dan pergi, tingkat *chunk* dalam *swarm* bisa sangat tinggi.

Masalah ketiga melibatkan insentif. *Node* CDN diatur secara eksklusif untuk memberikan konten kepada pengguna. *Node* P2P tidak. Mereka adalah komputer pengguna, dan pengguna mungkin lebih tertarik untuk mendapatkan konten daripada membantu unduhan pengguna lain, dengan kata lain, terkadang ada insentif bagi pengguna untuk menipu sistem. *Node* yang mengambil sumber daya dari suatu sistem tanpa memberikan kontribusi dalam bentuk apa pun adalah disebut *free-rider* atau *leechers*. Jika jumlahnya terlalu banyak, sistem tidak akan berfungsi dengan baik. BitTorrent berupaya mengatasi masalah ini dengan memberi penghargaan kepada *peers* yang menunjukkan perilaku pengunggahan yang baik. Setiap peer secara acak mengambil sampel peer lainnya, mengambil *chunk* dari mereka saat mengunggah potongan ke mereka. Peer terus *trading chunk* dengan hanya sejumlah

kecil peer yang memberikan kinerja unduhan tertinggi, sementara juga secara acak mencoba peer lain untuk menemukan mitra yang baik. Secara acak mencoba peer juga memungkinkan pendatang baru untuk mendapatkan chunk awal yang dapat mereka *trading* dengan peer lainnya. Peers yang saat ini bertukar chunk disebut unchoked.

2.5 *Hypertext Transfer Protocol*

Hypertext Transfer Protocol (HTTP) adalah sebuah protokol yang terdapat pada lapisan *application* yang terdapat pada TCP/IP model. Melalui HTTP, data klien dapat meminta data kepada server dan server dapat mengirimkan kembali datanya. Data yang dikirim dapat berupa text, text layout, gambar, dll.

2.5.1 *HTTP/2*

Sejak awal pembuatan HTTP/2 pada tahun 2012 dilakukan karena HTTP/1.0 dan HTTP/1.1 sudah terbentuk sejak 2007 dan sudah cukup lama. Oleh karena itu, *Internet Engineering Task Force* (IETF) membentuk grup untuk membuat HTTP/2. Grup ini memiliki beberapa tujuan: mengizinkan klien dan server untuk memilih versi HTTP yang akan digunakan, menjaga kompatibilitas dengan HTTP/1.1 sebaik mungkin, meningkatkan performa dengan *multiplexing*, *pipelining*, *compression*, dll, serta mendukung *browsers*, *servers*, *proxies*, *delivery*, jaringan yang sudah ada agar dapat menggunakan versi terbaru dari HTTP.

HTTP/2 lebih unggul dalam hal kecepatan browsing web dikarenakan ketika klien meminta data dari server, data tersebut dapat dikirim dalam bentuk *binary* dan sistem koneksi yang dibuat dapat meminta banyak data dan dapat dikirimkan dengan urutan yang diinginkan, karena berjalan secara *multiplexing*. Tanpa harus meminta data satu persatu, seperti yang terdapat pada HTTP/1.1.

Kendala enkripsi menjadi suatu perdebatan pada HTTP/2. Dikarenakan banyak pihak yang sangat menginginkannya, tetapi banyak juga pihak yang menolaknya. Sebagian besar yang menolak hal tersebut adalah yang berkaitan dengan *Internet-of-Things* (IoT). Karena pada bidang IoT, enkripsi tidak dibutuhkan. Tetapi pada semua *browsers* memerlukan enkripsi, pada akhirnya tetap ada enkripsi untuk keperluan web *browsing*.

2.5.2 HTTP/3

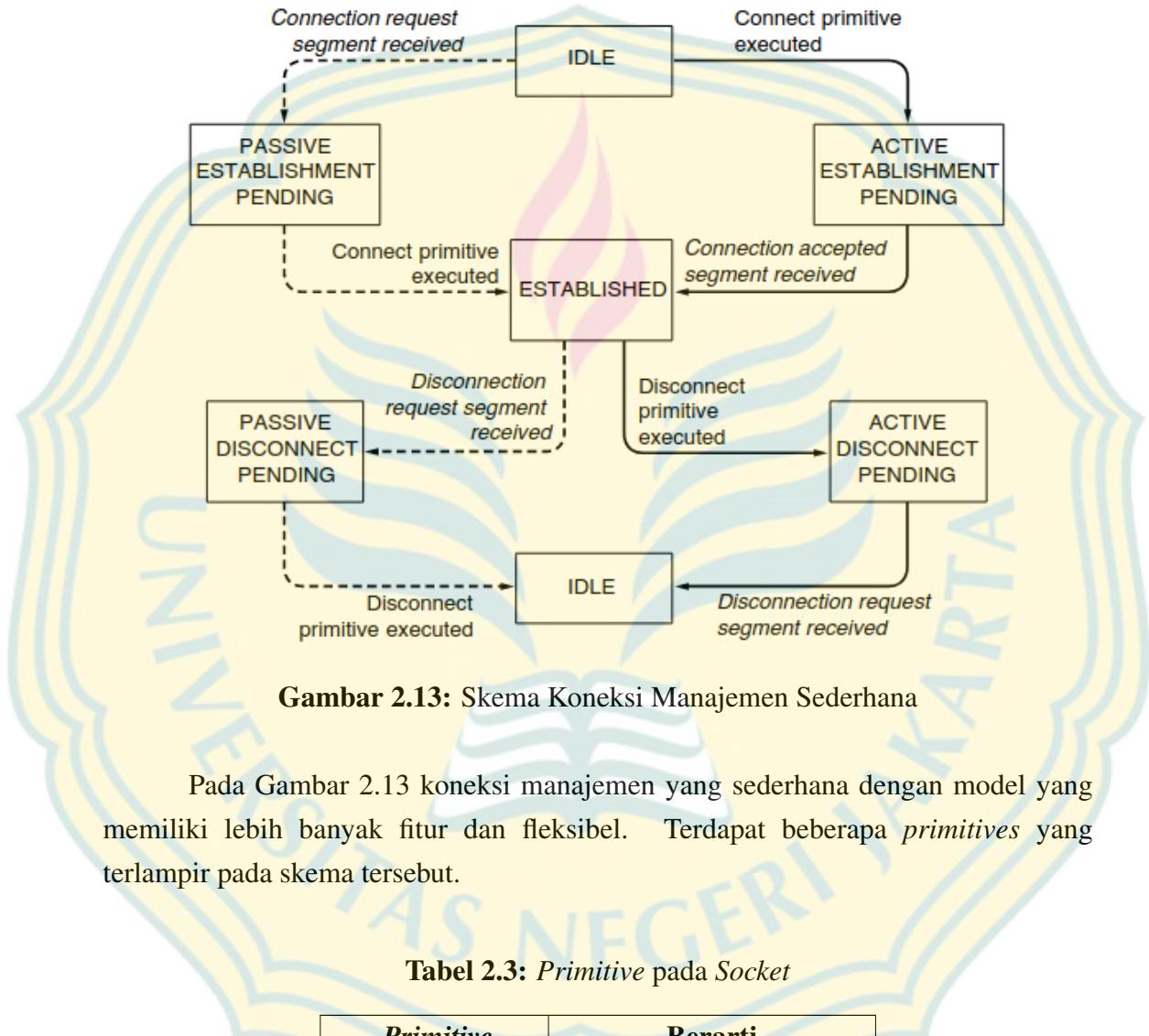
HTTP/3 adalah revisi terbesar ketiga dari HTTP, didesain sebagai penerus dari HTTP/2. Perbedaan terbesar dari HTTP/3 adalah pada lapisan transportasi yang mendukung HTTP *messages* ketimbang bergantung pada TCP. Karena sekarang mendukung Quick UDP Internet Connections (QUIC) yang merupakan versi tambahan dari UDP. Sudah banyak bahasa pemrograman yang mendukung penggunaan QUIC dan HTTP/3 seperti C, C++, Python, Rust, dan Go. *Web server* nginx juga sudah memadai untuk HTTP/3.

Pada HTTP/3 juga masih menggunakan layanan yang sama dengan HTTP/2 seperti *multiplexing*. QUIC memiliki pengaturan koneksi *zero-round-trip*, ketika koneksi sudah tersedia di *server endpoint* atau sebelumnya klien sudah pernah terhubung ke suatu server, HTTP/3 dapat mengizinkan klien untuk menggunakan kembali koneksi tersebut dengan berbagai macam URLs yang berbeda dan secara cepat. Karena HTTP/3 berjalan di atas QUIC yang merupakan lapisan transportasi akan meningkatkan performa yang sangat signifikan. Dengan kata lain, QUIC adalah generasi selanjutnya dari TCP.

2.6 *Socket Programming*

Socket pertama kali dirilis sebagai bagian dari Berkeley UNIX 4.2BSD distribusi perangkat lunak pada tahun 1983. Socket dengan cepat menjadi populer. *Primitives* itu sekarang digunakan untuk pemrograman internet pada banyak sistem operasi, terutama sistem berbasis UNIX, dan ada API (*Application Programming Interface*) bergaya socket untuk Windows disebut ‘*winsock*’.

Mencerdaskan dan
Memartabatkan Bangsa



Tabel 2.3: Primitive pada Socket

Primitive	Berarti
<i>SOCKET</i>	Membuat sebuah komunikasi endpoint baru
<i>BIND</i>	Mengasosiasi alamat lokal dengan socket
<i>LISTEN</i>	Bersedia untuk menerima koneksi dan membuat antrian

<i>ACCEPT</i>	Mengizinkan koneksi masuk dan terbentuk secara pasif
<i>CONNECT</i>	Melakukan percobaan untuk mendirikan koneksi secara aktif
<i>SEND</i>	Mengirimkan beberapa data melalui koneksi
<i>RECEIVE</i>	Menerima beberapa data dari koneksi
<i>CLOSE</i>	Melepaskan koneksi

Empat *primitives* pertama dalam daftar dieksekusi dalam urutan oleh server. *SOCKET primitive* membuat titik akhir baru dan mengalokasikan ruang tabel untuknya di dalam entitas transportasi. Parameter panggilan menentukan format pengalamatan yang akan digunakan, jenis layanan yang diinginkan (misalnya *reliable byte stream*), dan protokol. Panggilan socket yang berhasil mengembalikan *file descriptor* biasa untuk digunakan dalam panggilan yang berhasil, sama seperti panggilan *OPEN* pada file.

Socket yang baru dibuat tidak memiliki alamat jaringan. Oleh karena itu, menggunakan *BIND*. Setelah server mengikat alamat ke socket, klien jarak jauh dapat terhubung ke sana. Berikutnya adalah panggilan *LISTEN*, yang mengalokasikan ruang untuk mengantri panggilan, jika terdapat beberapa klien yang mencoba untuk terhubung pada waktu yang sama. Ketika segmen yang meminta koneksi tiba, entitas transportasi membuat socket baru dengan properti yang sama dengan yang asli dan mengembalikan *file descriptor* untuk itu. Server kemudian dapat memotong proses atau utas untuk menangani koneksi pada socket baru dan kembali menunggu koneksi berikutnya pada socket asli. *ACCEPT* mengembalikan *file descriptor*, yang dapat digunakan untuk membaca dan menulis dengan cara standar, sama seperti untuk file.

CONNECT memblokir pemanggilan dan memulai proses koneksi. Ketika segmen yang sesuai diterima dari server, proses klien dibuka blokirnya dan koneksi dibuat. Kedua belah pihak sekarang dapat menggunakan *SEND* dan *RECEIVE* untuk

mengirim dan menerima data melalui koneksi full-duplex. Panggilan sistem UNIX *READ* dan *WRITE* standar juga dapat digunakan jika tidak ada opsi khusus *SEND* dan *RECEIVE* yang diperlukan. Pelepasan koneksi dengan socket bersifat simetris. Ketika kedua belah pihak telah mengeksekusi *CLOSE*, sambungan dilepaskan atau dihentikan.

Berikut adalah penjelasan terkait bagaimana penerapan sederhana socket dilakukan. Kodenya akan memiliki beberapa batasan, tetapi telah menerapkan prinsip dasar socket dan dapat dikompilasi oleh berbagai sistem UNIX yang terhubung ke internet. Kode klien dapat dieksekusi dengan parameter yang sesuai untuk mengambil file apapun pada server yang memiliki akses pada mesinnya.

Kode server dimulai dengan memasukkan beberapa standar *header*, tiga yang terakhir berisi definisi terkait internet utama dan struktur data. Berikutnya adalah definisi *SERVER_PORT* sebagai 8080. Angka ini dipilih secara bebas. Angka apa pun antara 1024 dan 65535 akan berfungsi sama baik, selama tidak digunakan oleh beberapa proses lain, *port* di bawah 1023 dicadangkan untuk *privileged users*. Dua baris berikutnya di server mendefinisikan konstanta. Yang pertama menentukan ukuran potongan dalam byte yang digunakan untuk transfer file. Yang kedua menentukan bagaimana banyak koneksi yang tertunda dapat ditahan sebelum koneksi tambahan dibuang.

Setelah deklarasi variabel lokal, kode server dimulai. Dimulai dengan menginisialisasi struktur data yang akan menyimpan alamat IP server. Struktur data ini akan segera terikat ke socket server. Memanggil *memset* mengatur data struktur untuk semua menjadi 0s. Tiga tugas berikutnya mengisi tiga bidangnya. Terakhir berisi *port* server. Fungsi *htonl* dan *htons* harus dilakukan dengan mengkonversi nilai ke format standar sehingga kode berjalan dengan benar pada mesin *little-endian* (mis., Intel x86) dan mesin *big-endian* (mis., SPARC).

Selanjutnya, server membuat socket dan memeriksa kesalahan ($s < 0$). Di dalam versi produksi kode, pesan kesalahan bisa menjadi sedikit lebih jelas. Panggilan ke *setsockopt* diperlukan agar *port* dapat digunakan kembali, jadi server dapat berjalan tanpa batas, memberikan permintaan secara terus menerus. Sekarang alamat IP terikat socket dan pemeriksaan dilakukan untuk melihat apakah panggilan untuk mengikat berhasil. Langkah terakhir dalam inisialisasi adalah panggilan untuk *listen* untuk *announce* kesediaan server untuk terima panggilan masuk dan memberitahu sistem untuk menahan hingga *QUEUE_SIZE* jika permintaan baru tiba saat server masih memproses yang sekarang. Jika antrian penuh dan permintaan

tambahan tiba, maka secara perlahan akan disingkirkan.

Pada titik ini, server memasuki *loop* utamanya. Satu-satunya cara untuk menghentikan *loop*-nya adalah dengan memberhentikannya dari luar. Panggilan untuk menerima memblokir server sampai beberapa klien mencoba membuat koneksi dengannya. Jika panggilan terima berhasil, maka akan mengembalikan socket descriptor yang dapat digunakan untuk membaca dan menulis, *file descriptor* dapat digunakan untuk membaca dari dan menulis ke *pipe*. Namun, tidak seperti *pipe*, yang searah, socket berlaku secara dua arah, jadi *sa* (diterima socket) dapat digunakan untuk membaca dari koneksi dan juga untuk menulis ke sana. Setelah koneksi dibuat, server membaca nama file darinya. Jika namanya belum tersedia, server menunggunya. Setelah mendapatkan nama file, server membuka file dan memasukkan *loop* yang secara bergantian membaca blok dari file dan menulisnya ke socket sampai seluruh file telah disalin. Kemudian server menutup file dan koneksi dan menunggu koneksi berikutnya ditampilkan ke atas. Itu mengulangi *loop* ini selamanya.

Sedangkan, kode klien dimulai dengan beberapa penyertaan dan deklarasi. Eksekusi dimulai dengan memeriksa untuk melihat apakah itu telah dipanggil dengan jumlah argumen yang tepat, di mana argc = 3 artinya program dipanggil dengan namanya ditambah dua argumen. Selanjutnya, socket dibuat dan diinisialisasi. Setelah itu, klien mencoba buat koneksi TCP ke server, menggunakan connect. Jika server berjalan pada perangkat dan terpasang ke SERVER_PORT dan memiliki ruang dalam antrean *listen*-nya, koneksi akan dibuat. Menggunakan koneksi, klien mengirimkan nama file dengan menulis di socket. Jumlah byte yang dikirim adalah satu lebih besar dari nama yang seharusnya, karena 0 byte yang mengakhiri nama juga harus dikirim untuk memberi tahu server tempat berakhirnya nama tersebut.

Sekarang klien memasuki *loop*, membaca file blok secara satu persatu dari socket dan menyalinnya ke standar *output*. Prosedur fatal mencetak pesan kesalahan dan keluar. Karena klien dan server dikompilasi secara terpisah dan biasanya berjalan di komputer yang berbeda, mereka tidak dapat berbagi kode fatal. Server juga dapat memiliki prosedur ini.

Pada server kesalahan pemeriksannya sedikit dan pelaporan kesalahannya biasa saja. Karena menangani semua permintaan secara berurutan (karena hanya memiliki satu utas), kinerjanya rendah. Jelas tidak pernah tau tentang keamanan, dan menggunakan panggilan sistem UNIX kosong bukanlah cara untuk mendapatkan

independensi *platform*. Itu juga membuat beberapa asumsi bahwa secara teknis ilegal, seperti mengasumsikan bahwa nama file cocok dengan *buffer* dan memang demikian ditransmisikan secara atomik. Terlepas dari kekurangan ini, tetapi berfungsi layaknya server file internet. Kode klien disertakan setelah ini.

```
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 8080
#define BUF_SIZE 4096

int main(int argc, char **argv)
{
    int c, s, bytes;
    char buf[BUF_SIZE];
    struct hostent *h;
    struct sockaddr_in channel;

    if (argc != 3) {printf("Usage: client server-name file-name0); exit(-1);}
    h = gethostbyname(argv[1]); /* look up host's IP address */
    if (!h) {printf("gethostbyname failed to locate %s0, argv[1]); exit(-1);}

    s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (s < 0) {printf("socket call failed0); exit(-1);}
    memset(&channel, 0, sizeof(channel));
    channel.sin_family= AF_INET;
    memcpy(&channel.sin_addr.s_addr, h->h_addr, h->h_length);
    channel.sin_port= htons(SERVER_PORT);
    c = connect(s, (struct sockaddr *) &channel, sizeof(channel));
    if (c < 0) {printf("connect failed0); exit(-1);}

    /* Connection is now established. Send file name including 0 byte at end. */
    write(s, argv[2], strlen(argv[2])+1);

    /* Go get the file and write it to standard output. */
    while (1) {
        bytes = read(s, buf, BUF_SIZE); /* read from socket */
        if (bytes <= 0) exit(0); /* check for end of file */
        write(1, buf, bytes); /* write to standard output */
    }
}
```

Gambar 2.14: Kode Socket Sisi Klien

```

#include <sys/types.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/fcntl.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 8080
#define BUF_SIZE 4096
#define QUEUE_SIZE 10

int main(int argc, char *argv[])
{ int s, b, l, fd, sa, bytes, on = 1;
  char buf[BUF_SIZE];
  struct sockaddr_in channel;

  /* Build address structure to bind to socket. */
  memset(&channel, 0, sizeof(channel));           /* zero channel */
  channel.sin_family = AF_INET;
  channel.sin_addr.s_addr = htonl(INADDR_ANY);
  channel.sin_port = htons(SERVER_PORT);

  /* Passive open. Wait for connection. */
  s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP); /* create socket */
  if (s < 0) {printf("socket call failed0"); exit(-1);}
  setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *) &on, sizeof(on));

  b = bind(s, (struct sockaddr *) &channel, sizeof(channel));
  if (b < 0) {printf("bind failed0"); exit(-1);}

  l = listen(s, QUEUE_SIZE);                      /* specify queue size */
  if (l < 0) {printf("listen failed0"); exit(-1);}

  /* Socket is now set up and bound. Wait for connection and process it. */
  while (1) {
    sa = accept(s, 0, 0);                         /* block for connection request */
    if (sa < 0) {printf("accept failed0"); exit(-1);}

    read(sa, buf, BUF_SIZE);                      /* read file name from socket */

    /* Get and return the file. */
    fd = open(buf, O_RDONLY);                     /* open the file to be sent back */
    if (fd < 0) {printf("open failed");}
    if (fd < 0) {printf("open failed");}

    while (1) {
      bytes = read(fd, buf, BUF_SIZE); /* read from file */
      if (bytes <= 0) break;          /* check for end of file */
      write(sa, buf, bytes);        /* write bytes to socket */
    }
    close(fd);                                /* close file */
    close(sa);                                /* close connection */
  }
}

```

Gambar 2.15: Kode Socket Sisi Server

2.7 BitTorrent

Awal mula BitTorrent dicetuskan karena masalah ketika klien mengunduh file dari suatu server yang dapat memungkinkan prosesnya melambat. Jika pada satu waktu, banyak klien yang mengunduh. Hal tersebut menyebabkan menurunnya *bandwidth* ketika mengunggah file ke setiap klien. Perbedaan kecepatan internet juga

menjadi masalah. Dikarenakan proses unduh akan memiliki kecepatan yang lebih tinggi dibanding dengan unggah. Ketika server memiliki kecepatan unggah yang rendah, maka akan menjadi penghambat. BitTorrent menjadi sebuah solusi untuk menyelesaikan permasalahan tersebut.

BitTorrent adalah teknologi atau protokol yang membuat distribusi file, terutama file berukuran besar, lebih mudah dan hemat bandwidth. Hal ini dicapai dengan memanfaatkan kapasitas unggah *peers* yang mendownload file. Peningkatan pengunduh yang cukup signifikan hanya akan menghasilkan sedikit peningkatan beban pada server yang meng-hosting file tersebut. *Peers* adalah sekumpulan dari klien yang melakukan proses unduh dan unggah. Jadi, *peer* tidak hanya mendapatkan hasil unduhan dari server saja, tetapi juga dari *peer* lain yang melakukan proses unggah file. Proses unggah yang dilakukan dalam protokol BitTorrent dengan memotong file menjadi beberapa kepingan yang disebut pieces. Lalu setiap *peer* yang terhubung akan mendapatkan kepingan yang berbeda. Lalu masing - masing *peer* akan bertukar kepingan untuk melengkapi data dari file tersebut, menjadi satu kesatuan file yang utuh. Pada protokol ini, *peer* bertindak melakukan unduh dan unggah.

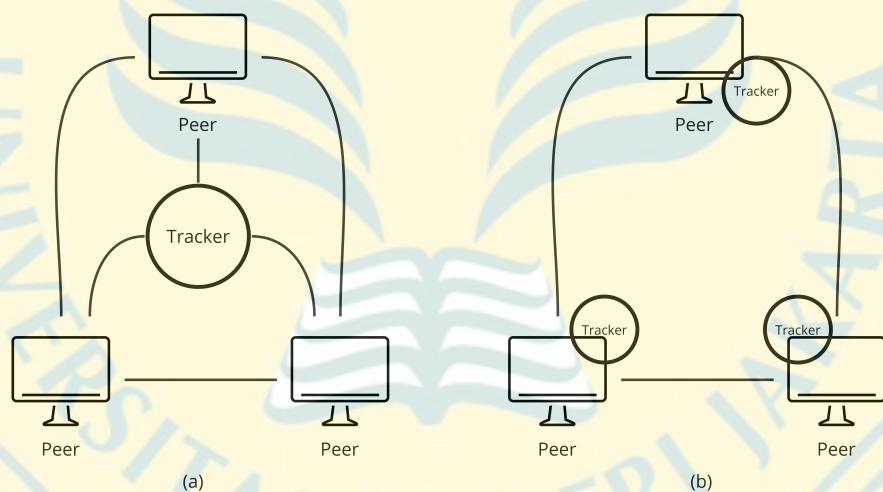
2.7.1 Arsitektur BitTorrent

Langkah pertama untuk merilis file menggunakan BitTorrent adalah membuat metadata info dari file tersebut. Metadata ini disebut dengan torrent dan ditetapkan dalam file dengan extension ‘.torrent’. Pada metadata ini, terdapat info yang terdiri dari url dari *tracker*, *filename*, *size* , *path*, dan *hashing information*. Pembuatan torrent dapat dilakukan dengan aplikasi penyedia layanan torrent.

Proses pembuatan torrent baru berasal dari satu komputer yang memiliki file secara utuh. Lalu file tersebut dibuatkan torrentnya, dengan metadata yang diperlukan. Setelah itu komputer tersebut akan melakukan proses unggah kepada *peer* yang ingin mengunduh. Proses tersebut disebut *seeding* dan yang melakukan proses itu disebut ‘*seeder*’. Sedangkan *peer* lain yang menambahkan torrent tersebut dapat mengunduhnya. *Peer* yang belum memiliki keseluruhan file atau kepingan dari file disebut ‘*leecher*’. Semakin banyak *peer* yang terhubung, maka proses mengunduh semakin cepat. Sekumpulan *peer* yang membagikan file yang sama disebut ‘*swarm*’. Setelah *peer* tersebut memiliki file yang utuh, maka dapat sepenuhnya menjadi *seeder*.

Saat membuat file torrent dari file asli, file asli dipotong menjadi kepingan kecil, biasanya berukuran 512 kb atau 256 kb. Kode hash SHA-1 dari setiap kepingan disertakan dalam berkas torrent. Data yang diunduh diverifikasi dengan menghitung kode hash SHA-1 dan membandingkannya dengan kode SHA-1 dari bagian yang sesuai di file torrent. Dengan cara ini data diperiksa untuk kesalahan dan menjamin pengguna bahwa mereka mengunduh data yang benar. Setiap kali sebuah kepingan diunduh dan diverifikasi, *peer* yang mengunduh melaporkan ke *peer* lain di dalam *swarm* tentang kepingan barunya. Kepingan ini sekarang tersedia untuk *peer* lainnya.

Bittorrent memiliki dua jenis konsep *peer to peer*, *centralized tracker* dan *decentralized tracker*.



Gambar 2.16: (a) *Centralized Tracker*, (b) *Decentralized Tracker*

Pada Gambar 2.16 (a) adalah *centralized tracker* dan Gambar 2.16 (b) adalah *decentralized tracker*. *Tracker* atau pelacak menyimpan log peers yang saat ini mengunduh file, dan membantu mereka menemukan satu sama lain. *Tracker* tidak terlibat langsung dalam transfer data dan tidak memiliki salinan file. *Tracker* dan *peer* yang melakukan proses mengunduh bertukar informasi menggunakan protokol sederhana di atas HTTP. Pertama, pengguna memberikan informasi kepada *tracker* mengenai file mana yang diunduh, port yang digunakan, dll. Kemudian *tracker* merespon mengenai daftar *peer* lain yang mengunduh file yang sama dan informasi tentang cara menghubungi mereka.

Pada kedua konsep itu, *tracker* memiliki tugas yang sama. *Decentralized tracker* menjadi sebuah solusi untuk mengatasi kegagalan pada bittorrent dengan

central tracker. Jika *tracker* tersebut mengalami kegagalan, maka proses distribusi data tidak dapat dilakukan. Dengan adanya *decentralized tracker* menjadi pemecah masalah tersebut. Karena setiap *peer* memiliki trackernya sendiri atau dapat berperilaku sebagai *tracker*. Solusi ini dikembangkan pada Mei 2005 dengan bittorrent versi 4.1 dan berkonsep pada *Distributed Hash Tables* (DHT).

2.7.2 *Distributed Hash Tables (DHTs)*

DHT adalah sistem distribusi *decentralized*. Sistem ini terdiri dari sekumpulan *node* yang berpartisipasi dan sekumpulan kunci (*keys*). DHT melakukan fungsi tabel *hash*. Pasangan *key* dan *value* dapat disimpan dan *value* dapat dicari jika *key* yang benar disediakan. Apa yang membedakan DHT dari tabel *hash* biasa adalah penyimpanan dan pencarian didistribusikan di antara *node* (perangkat) dalam jaringan. Semua *node* adalah *peer* yang dapat bergabung dan meninggalkan jaringan secara bebas. DHT membuat jaminan tentang kinerja meskipun terlihat cukup kacau dengan adanya *peer* yang bergabung dan meninggalkan secara acak.

Struktur DHT terdiri dari dua bagian, partisi *keyspace* dan jaringan *overlay*. Partisi *keyspace* menangani partisi *key* di antara *node* dalam jaringan. Jaringan *overlay* menghubungkan *node* dan membiarkan mereka menemukan pemilik *key*. Pada bittorrent menggunakan DHT protokol yang disebut dengan Kademia.

2.7.3 Proses Komunikasi BitTorrent

Proses komunikasi dan pertukaran data pada protokol bittorrent berlangsung secara *peer to peer*. Mengarah langsung kepada klien, tanpa memerlukan sebuah server. Informasi mengenai *peer* yang terhubung pada jaringan, dengan mendapatkan informasi dari *tracker*. Tetapi *tracker* tidak ikut andil dalam transfer data.

Pertama kali *peer* menambahkan torrent file untuk diunduh, saat itu juga *peer* tersebut akan terhubung ke *tracker*, melalui *tracker* url yang tercantum pada torrent file. *Tracker* mendapatkan informasi terkait *peer* yang baru saja terhubung, seperti *IP address* dan *port* bittorrent klien yang digunakan. Jika klien berada pada private IP, maka dengan sendirinya bittorrent akan melakukan *port forwarding*, yang populer dengan menggunakan UPnP (*Universal Plug and Play*). *Port forwarding* berguna agar *peer* tersebut dapat terkoneksi dengan *peer* lain, karena IP dan *port* dari *peer* sudah di-forward. Setelah itu, *tracker* memberikan sekumpulan *peer* yang terhubung

dengan torrent yang sama.

Peer yang baru terhubung memiliki informasi tersebut untuk meminta *piece* file yang dibutuhkan dengan algoritma dan kebijakan yang ada. Sebelum meminta *piece* file, terlebih dahulu *peer* melakukan ‘*handshake*’ atau proses untuk membuat suatu koneksi dengan *peer* lain yang memiliki *piece*. *Handshake* dilakukan dengan terhubung melalui *IP address* dan *port* dari masing - masing *peer*. Setelah *handshake* berhasil, *peer* tersebut dapat berbagi data.

2.8 Algoritma pada BitTorrent

Di banyak protokol *file sharing peer-to-peer* lainnya, pertukaran file terjadi satu-ke-satu, artinya pengguna sendiri yang memilih rekan lain untuk mengunduh. Konsep BitTorrent adalah dapat mengunduh dari banyak *peer* lainnya secara bersamaan. Ini membutuhkan cara untuk mengetahui *peer* mana yang akan mengunduh dari bagian file yang mana, tujuannya untuk menerima file secara utuh secepat mungkin. Memilih *peer* untuk terhubung adalah masalah dua sisi. Pertama, membutuhkan cara untuk menemukan yang terbaik urutan unduh kepingan file. Kedua, *peer* yang memiliki kepingan yang diinginkan, mungkin tidak mengizinkan *peer* lain mengunduhnya. Strategi untuk *peer* yang tidak mengizinkan *peer* lain mengunduh dari mereka dikenal sebagai *choking*, dan menyangkut *resource allocation*.

2.8.1 Piece Selection

Tujuan dari algoritma ini adalah untuk mereplikasi kepingan yang berbeda pada *peer* yang berbeda sesegera mungkin. Ini akan tingkatkan kecepatan pengunduhan, dan juga pastikan semua bagian file ada di suatu tempat di jaringan jika *seeder* utama meninggalkan jaringan. BitTorrent menggunakan TCP dan karenanya penting untuk selalu mentransfer data atau kecepatan transfer akan drop karena mekanisme mulai lambat. Kepingan atau *pieces* tersebut selanjutnya dipecah menjadi sub-kepingan atau sub-*piece*, biasanya berukuran sekitar 16kb. Terdapat beberapa kebijakan atau policy yang diterapkan pada bittorrent.

2.8.2 Strict Policy

Setelah sub-*piece* diminta, sub-*piece* yang tersisa untuk bagian tertentu itu diminta sebelum sub-*piece* dari bagian lain. Ini membantu mendapatkan bagian utuh dari file secepat mungkin. Jadi setiap *peer* yang terhubung memungkinkan untuk memiliki setiap sub-*piece* yang berbeda dari *peer* utama pemilik file yang utuh.

2.8.3 Rarest First

Suatu *peer* memilih *piece* berikutnya untuk diunduh, memilih bagian yang paling sedikit dimiliki oleh *peer* yang berada di dalam satu swarm. Bertujuan agar setiap *piece* tersebar secara merata. Mempercepat proses unduh, karena *piece* tersebut tersebar di *peer* lain. Menjadikan *peer* lain memiliki opsi lain untuk mendapatkan *piece* tersebut. Dengan mengumpulkan *piece* paling langka, akan menjadikan *piece* yang common ditempatkan pada urutan terakhir, karena pasti banyak *peer* yang memiliki *piece* tersebut. Kebijakan ini juga mencegah *piece* yang langka ini hilang. Apabila terjadi kegagalan pada seeder utama dan *piece* yang langka ini belum dimiliki.

2.8.4 Random Piece First

Ketika *peer* pertama kali terhubung, *peer* tersebut tidak memiliki *piece* apapun untuk diunggah dan dibagikan ke *peer* lain. Oleh karena itu, penting untuk mendapatkan *piece* secepat mungkin dengan cara mengunduh *piece* secara random. Kebijakan ini lebih efisien dibanding *rarest first*. Jika semua *peer* mengambil *piece* paling langka, maka kecepatan unduh akan berkurang, karena sama - sama menginginkan *piece* yang sama. Jadi, untuk langkah pertama yang dilakukan oleh *peer* ketika terhubung adalah dengan mengambil *piece* secara acak.

2.8.5 Endgame Mode

Terkadang *piece* yang diunduh dari *peer* memiliki kecepatan transfer yang lambat. Ini bisa berpotensi menunda penyelesaian unduhan. Untuk mencegah hal ini, bittorrent memiliki "endgame mode". Pada prinsip pipelining, yang memastikan bahwa *peer* selalu memiliki sejumlah permintaan (sub-*piece*) tertunda, jumlahnya sering ditetapkan menjadi lima. Ketika semua sub-*piece* pada *peer* kekurangan diminta oleh *peer* lain, permintaan ini disiarkan ke semua *peer*. Ini membantu untuk

mendapatkan *piece* terakhir file secepat mungkin. Setelah *sub-piece* tiba, *peer* mengirimkan pesan pembatalan menunjukkan bahwa *peer* tersebut telah memperolehnya dan *peer* lain dapat mengabaikan permintaan tersebut. Beberapa bandwidth tentu saja disia-siakan oleh penyiaran ini, tetapi dalam praktiknya hal ini tidak terlalu banyak karena periode singkat dari mode endgame.

2.9 Resource Allocation BitTorrent

Tidak ada alokasi sumber daya terpusat di BitTorrent. Setiap *peer* bertanggung jawab atas memaksimalkan tingkat unduhannya. Seorang *peer*, secara alami, akan mencoba mengunduh dari siapa pun yang mereka bisa. Untuk memutuskan *peer* mana yang akan diunggah, *peer* menggunakan varian dari algoritma ‘tit-for-tat’. ‘tit-for-tat’ adalah strategi yang berasal dari teori permainan berulang, dan merupakan strategi berbasis kerjasama pada timbal balik. Intinya adalah lakukan pada yang lain seperti yang mereka lakukan.

2.9.1 Choking

Choking adalah penolakan sementara untuk mengunggah ke *peer* lain, tetapi *peer* itu masih dapat mengunduh dari *peer* yang ditolak. Prinsipnya unggah ke *peer* yang sudah unggah kepada kita sebelumnya. Tujuannya adalah untuk memiliki beberapa koneksi dua arah kapan saja, dan mencapai ‘efisiensi Pareto’.

Peer selalu *unchokes* sejumlah tetap dari *peer* (standarnya adalah empat). Memutuskan *peer* mana yang akan *unchokes* hanya ditentukan oleh tingkat unduh saat ini. Telah dipilih untuk menggunakan 20 detik rata - rata untuk memutuskan ini. Karena penggunaan TCP, tidak diatur untuk *choke* dan *un-choke* juga dengan cepat. Jadi, ini dihitung setiap sepuluh detik.

Hasilnya adalah setiap rekan akan mengunggah ke *peer* yang memberikan tingkat unduhan terbaik. Itu sebaliknya; jika tingkat unggahan kita tinggi, lebih banyak *peer* yang memungkinkan untuk kita unduh dari mereka. Ini berarti kita bisa mendapatkan tingkat unduhan yang lebih tinggi jika kita memiliki banyak unggahan. Ini fitur paling penting dari protokol BitTorrent. Ini melarang sejumlah besar “penumpang gratis” atau ‘freerider’ yang merupakan *peer* yang hanya mengunduh dan tidak mengizinkan pengunggahan. Agar *peer-to-peer* jaringan berjalan secara efisien, semua peer harus berkontribusi ke jaringan. Pembatasan ini tidak hadir di

sebagian besar protokol dan aplikasi *peer-to-peer* lainnya, dan merupakan salah satu alasannya BitTorrent telah menjadi sangat populer.

2.9.2 *Optimistic Unchoking*

Optimistic Unchoking adalah variasi dari choking. Setiap beberapa waktu tertentu, *peer* melakukan *optimistic unchoke*. Proses tersebut yang dilakukan *peer*, memilih *peer* lain yang terlihat kurang aktif pada swarm, *peer* tersebut akan disebut sebagai *peer unchoked*. *Peer* yang memilih *peer unchoked* akan mengirimkan *piece* kepadanya. Membuat *peer unchoked* kembali aktif dalam transfer data. Hal ini bertujuan untuk memberikan kesempatan kepada *peer unchoked* untuk meningkatkan koneksi dengan *peer* lain, dan dapat berbagi *piece* dengan *piece* lainnya. Dapat menghindari beberapa *peer* yang dominan dan mengendalikan transfer data secara masif.

2.10 *Network Address Translation (NAT)*

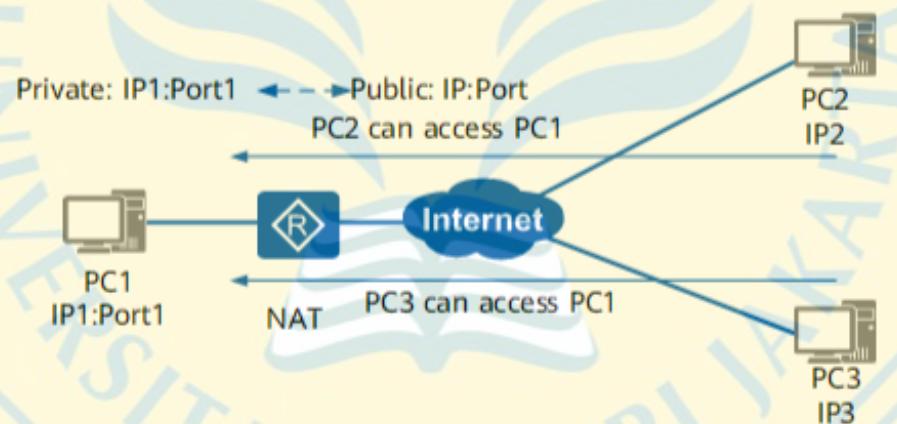
Network address translation (NAT) adalah metode pemetaan ruang alamat IP ke yang lain dengan memodifikasi informasi alamat jaringan di header IP paket saat mereka transit melintasi perangkat perutean lalu lintas atau *router*. Teknik ini awalnya digunakan untuk melewati kebutuhan untuk menetapkan alamat baru ke setiap *host* saat jaringan dipindahkan, atau saat penyedia layanan internet diganti, tetapi tidak dapat merutekan ruang alamat jaringan. Ini telah menjadi alat yang populer dan penting dalam melestarikan ruang alamat *global* dalam menghadapi keterbatasan alamat IPv4. Satu alamat IP yang dapat dirutekan internet dari gateway NAT dapat digunakan untuk seluruh jaringan pribadi. Karena NAT memodifikasi informasi alamat IP dalam paket, implementasi NAT dapat bervariasi dalam perilaku spesifiknya dalam berbagai kasus pengalamatan dan pengaruhnya terhadap lalu lintas jaringan.

Perangkat NAT memungkinkan penggunaan alamat IP pribadi pada jaringan pribadi di belakang *router* dengan satu alamat IP publik yang menghadap ke internet. Perangkat jaringan internal berkomunikasi dengan *host* di jaringan eksternal dengan mengubah alamat sumber permintaan keluar ke perangkat NAT dan menyampaikan balasan kembali ke perangkat asal.

2.10.1 Klasifikasi NAT

Network address dan *translation port* dapat diimplementasikan dalam beberapa cara. Beberapa aplikasi yang menggunakan informasi alamat IP mungkin perlu menentukan alamat eksternal dari penerjemah alamat jaringan. Ini adalah alamat yang dideteksi oleh *peer* komunikasinya di jaringan eksternal. Selain itu, mungkin perlu untuk memeriksa dan mengkategorikan jenis pemetaan yang digunakan, misalnya bila diinginkan untuk mengatur jalur komunikasi langsung antara dua klien yang keduanya berada di belakang *gateway* NAT yang terpisah. NAT juga diklasifikasikan menjadi empat tipe:

1. *Full-cone NAT*

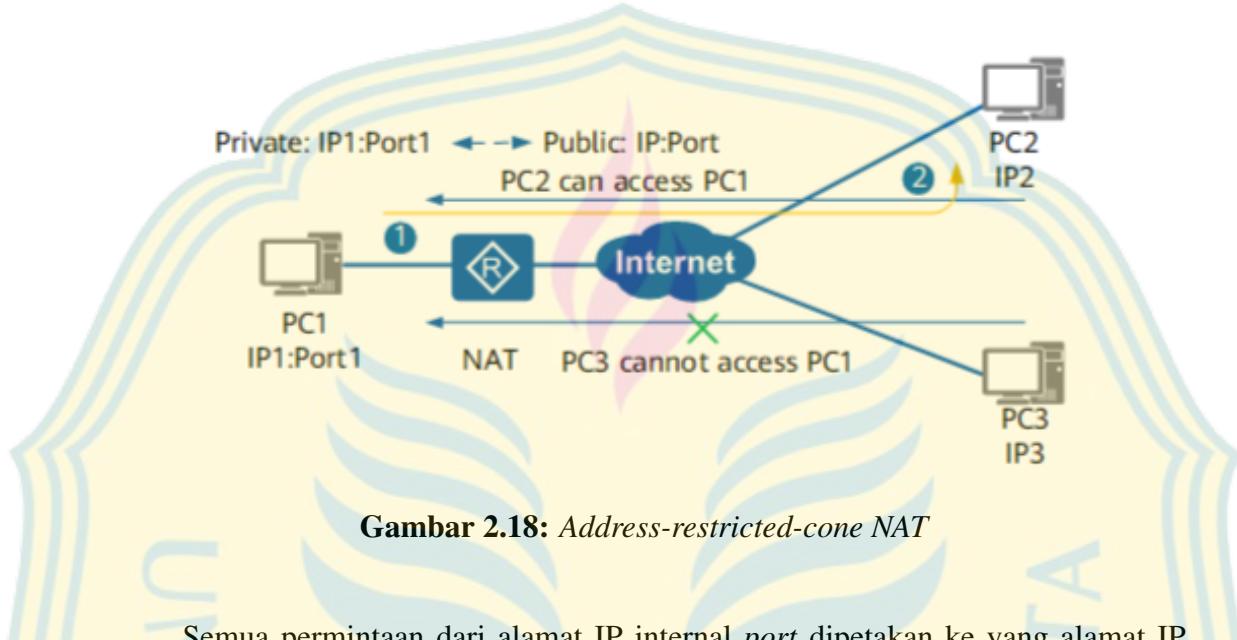


Gambar 2.17: *Full-cone NAT*

Semua alamat berasal dari satu internal alamat IP dan *port* yang sama akan dipetakan pada satu eksternal alamat IP dan *port*. Setiap *host* eksternal dapat mengirim paket ke *host* internal, dengan mengirimkan paket ke yang dipetakan alamat eksternal.

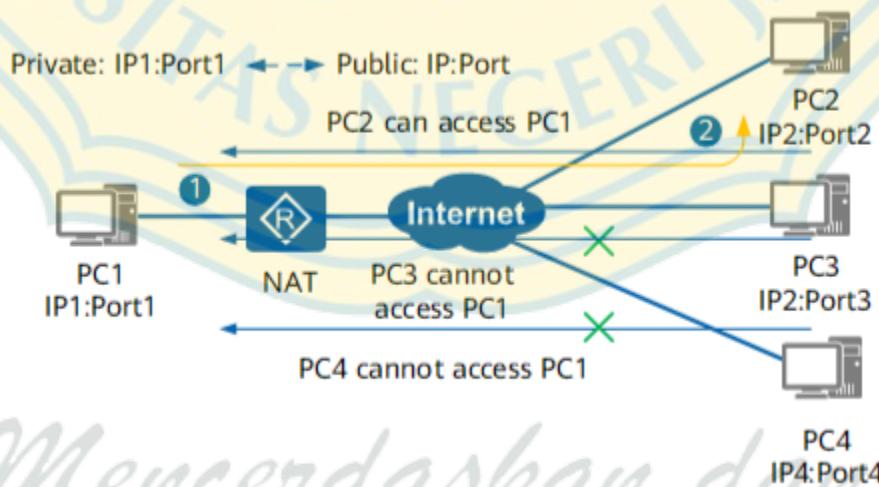
*Mencerdaskan dan
Memartabatkan Bangsa*

2. Address-restricted-cone NAT



Semua permintaan dari alamat IP internal *port* dipetakan ke yang alamat IP dan *port* eksternal yang sama. Berbeda dari *Full-cone*, eksternal *host* (dengan alamat IP A) dapat mengirim paket ke *host* internal hanya jika *host* internal sebelumnya mengirim paket ke IP alamat A.

3. Port-restricted-cone NAT

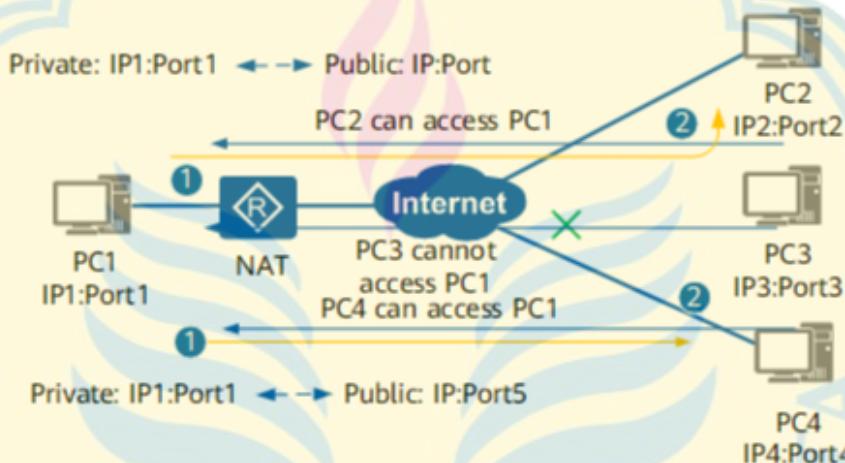


Gambar 2.19: Port-restricted-cone NAT

Mirip seperti *Address-restricted-cone*, tetapi pembatasan tersebut mencakup nomor *port*. Secara khusus, *host* eksternal dapat mengirim paket, dengan IP

sumber alamat A dan *port* sumber P, ke *host* internal hanya jika *host* internal sebelumnya telah mengirim paket ke alamat IP A dan *port* P.

4. symmetric NAT



Gambar 2.20: Symmetric NAT

Semua permintaan dari alamat IP internal dan *port* yang sama, ke alamat IP dan *port* yang spesifik, dipetakan ke alamat IP eksternal dan *port* yang sama. Jika *host* yang sama mengirimkan paket dengan sumber yang sama alamat dan *port*, tetapi ke tujuan yang berbeda, maka akan menggunakan pemetaan yang berbeda. Hanya *host* eksternal yang menerima paket dapat mengirim paket UDP kembali ke *host* internal.

2.10.2 NAT Traversal

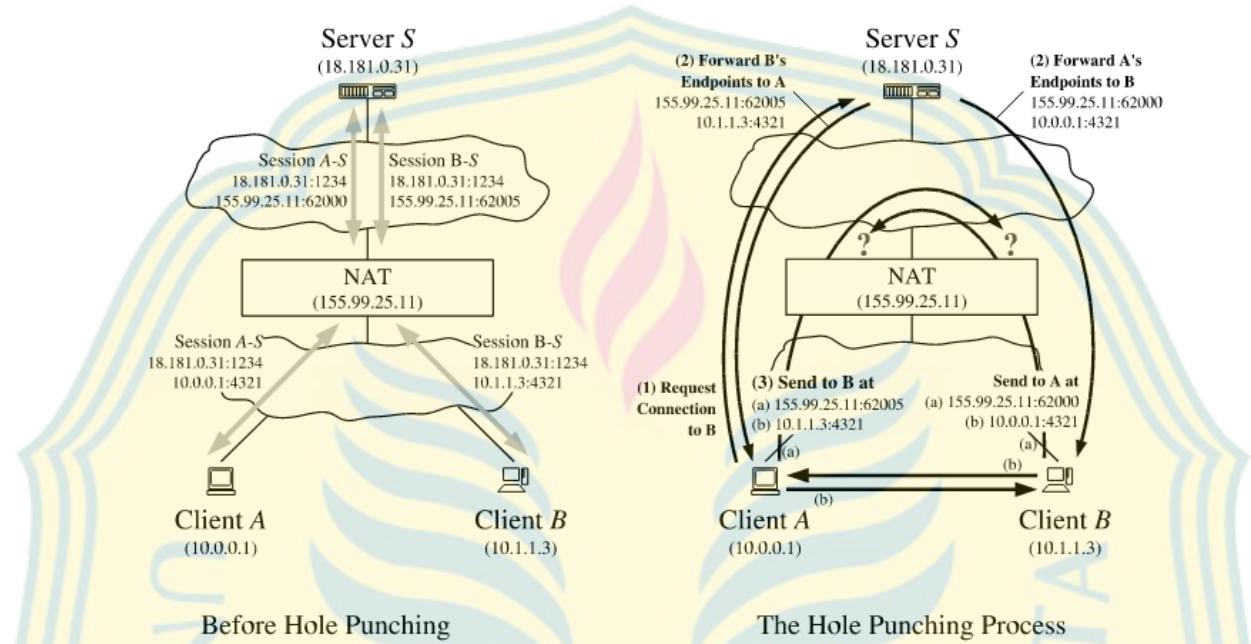
NAT *traversal* adalah teknik jaringan komputer untuk membangun dan menjaga koneksi Protokol internet di seluruh *gateway* yang menerapkan *network address translation* (NAT). Teknik NAT *traversal* diperlukan untuk banyak aplikasi jaringan, seperti berbagi file *peer-to-peer*. Masalah NAT *traversal* muncul ketika *peer* di belakang NAT yang berbeda mencoba untuk berkomunikasi. Salah satu cara untuk mengatasi masalah ini adalah dengan menggunakan *port forwarding*. Cara lain adalah dengan menggunakan berbagai teknik *traversal* NAT. Teknik yang paling populer untuk *traversal* UDP NAT adalah UDP *hole punching*.

2.10.3 *UDP Hole Punching*

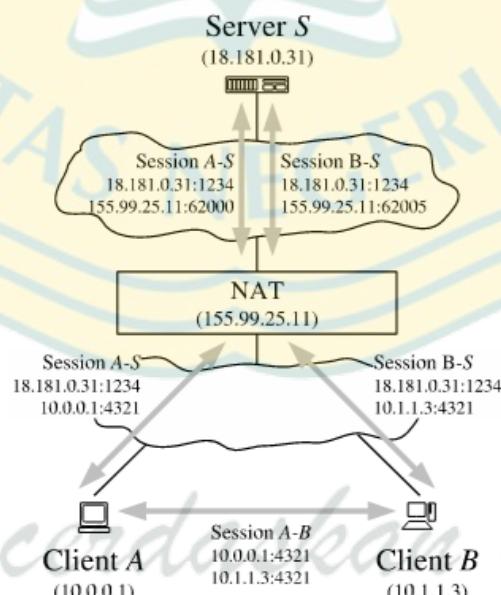
UDP hole punching membangun koneksi antara dua *host* yang berkomunikasi melalui satu atau lebih penerjemah alamat jaringan. Biasanya, *host* pihak ketiga di jaringan publik digunakan untuk menetapkan status *port* UDP yang dapat digunakan untuk komunikasi langsung antara *host* yang berkomunikasi. Setelah status *port* berhasil dibuat dan *host* berkomunikasi, status *port* dapat dipertahankan baik oleh lalu lintas komunikasi normal, atau jika tidak ada lagi, dengan paket *keep-alive*, biasanya terdiri dari paket UDP kosong atau paket dengan minimal, non-intrusive konten yang mengganggu.

Salah satu cara untuk melakukan *UDP hole punching* memerlukan *intermediary* atau perantara. Perantara ini disebut sebagai *rendezvous server* atau server pertemuan. Guna dari server pertemuan ini adalah agar masing - masing klien terhubung dan mendapatkan alamat IP dan *port* publik yang diberikan *router* kepada proses komunikasi yang berada di belakang NAT dengan alamat IP dan *port private*. Server pertemuan ini harus berada pada publik IP. Langkah yang terjadi ketika dua klien ingin melakukan komunikasi, yang mana kedua klien tersebut berada di belakang NAT adalah dengan mengirim udp packet ke server pertemuan. Lalu server akan mendapatkan alamat IP dan *port address* publik dari klien tersebut. Lalu melakukan pertukaran alamat IP dan *port* publik dari klien A kepada klien B, dan sebaliknya. Setelah mendapatkan masing - masing alamat IP dan *port* dari masing - masing klien. Klien A dapat mengirim packet data ke klien B melalui alamat IP dan *port* publiknya, dan sebaliknya. Setelah itu, kedua klien terhubung, walaupun server pertemuan dimatikan atau koneksi dari server terputus, kedua klien tersebut tetap dapat terhubung dan berkomunikasi, karena telah membuat ‘lubang’ pada NAT.

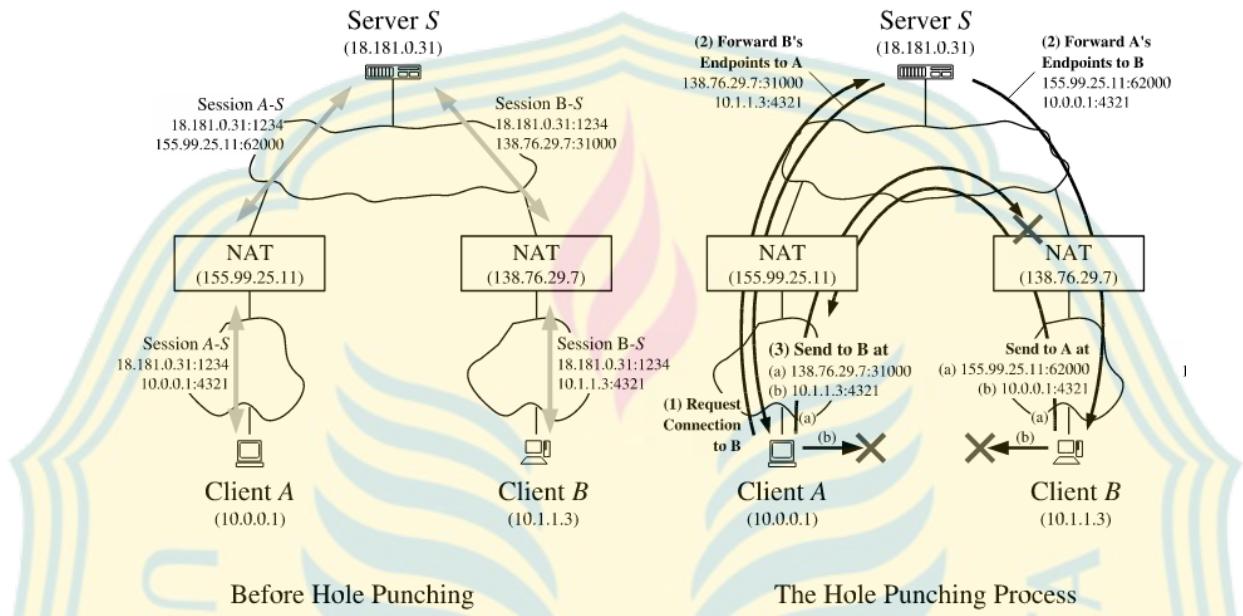
Mencerdaskan dan
Memartabatkan Bangsa



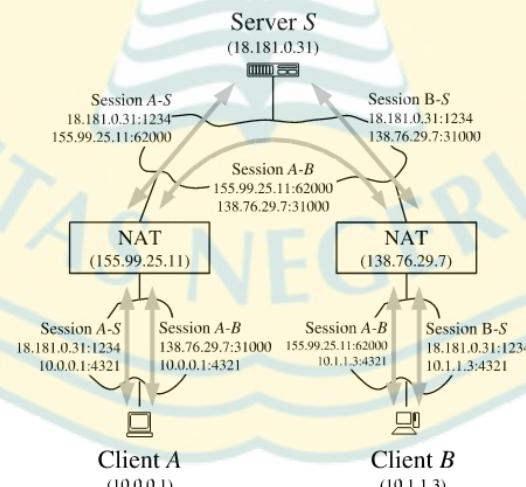
Gambar 2.21: Sebelum *Hole Punching* di Belakang Satu NAT yang Sama.



Gambar 2.22: Setelah *Hole Punching* di Belakang Satu NAT yang Sama.

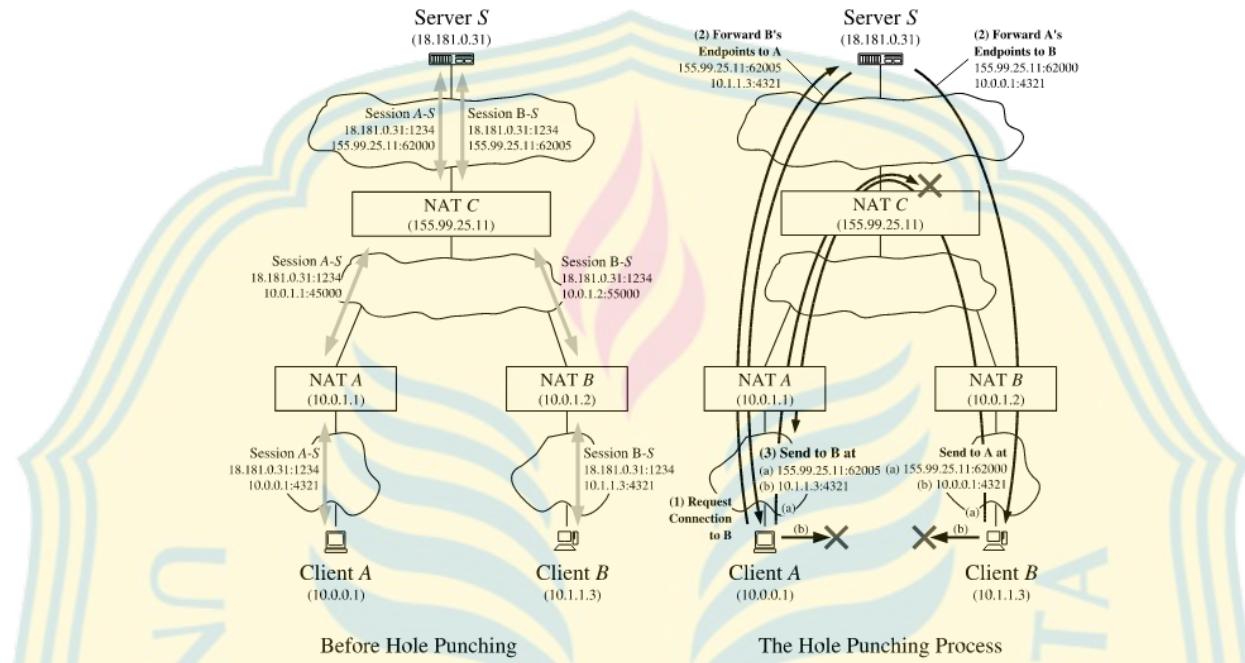


Gambar 2.23: Sebelum *Hole Punching* di Belakang NAT yang Berbeda.

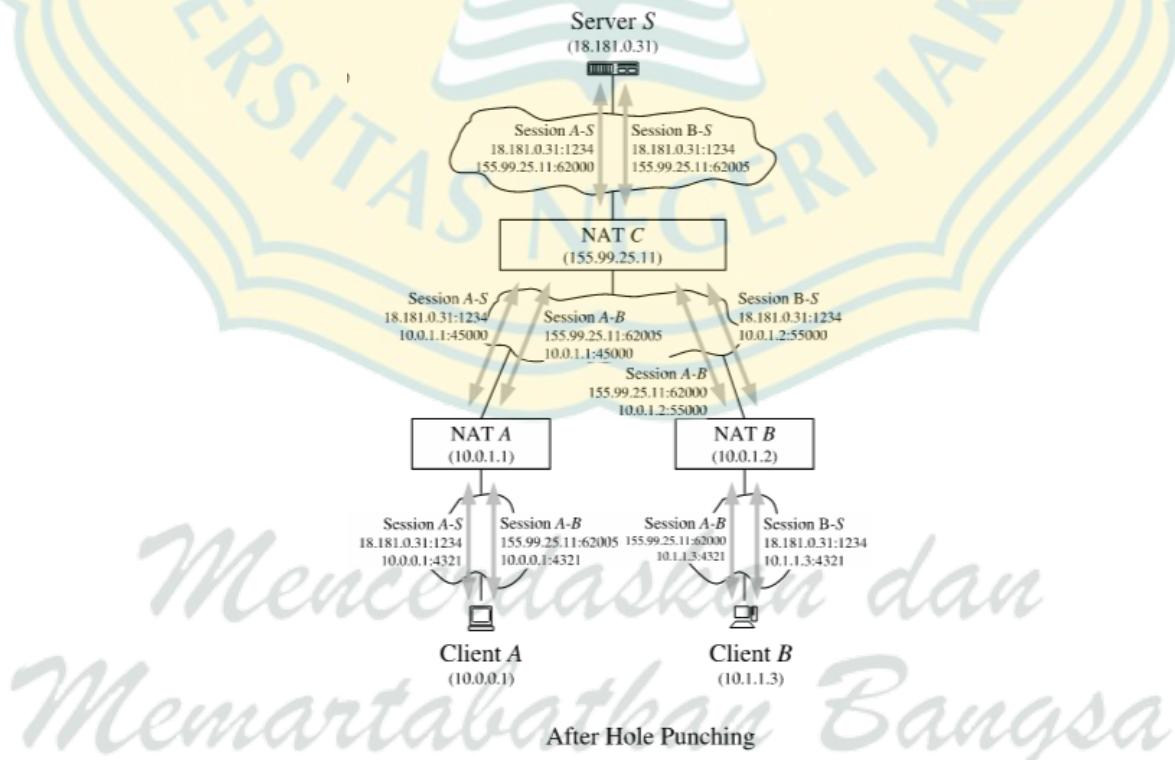


After Hole Punching

Gambar 2.24: Setelah *Hole Punching* di Belakang NAT yang Berbeda.



Gambar 2.25: Sebelum Hole Punching di Belakang Berbagai Level NAT.



Gambar 2.26: Setelah Hole Punching di Belakang Berbagai Level NAT.

2.10.4 *Timeouts*

NAT melakukan pemetaan alamat eksternal dan *port* untuk setiap klien yang ingin melakukan koneksi ke internet. Banyak perangkat NAT mematikan pemetaan tersebut jika tidak ada aktivitas yang terjadi dalam rentang waktu tertentu, beberapa NAT memiliki *timeouts* yang pendek sekitar 20 detik. Yang akan bervariasi tergantung dari perangkat dan konfigurasinya. Akan menjadi sebuah masalah jika pemetaan NAT untuk komunikasi antara klien berubah. Cara untuk mengatasi hal ini adalah dengan mengirim paket *keep alive* setiap beberapa rentang waktu. Untuk menjaga serta memastikan koneksi tetap terhubung antara klien. Paket *keep alive* biasanya berupa paket dengan ukuran kecil.

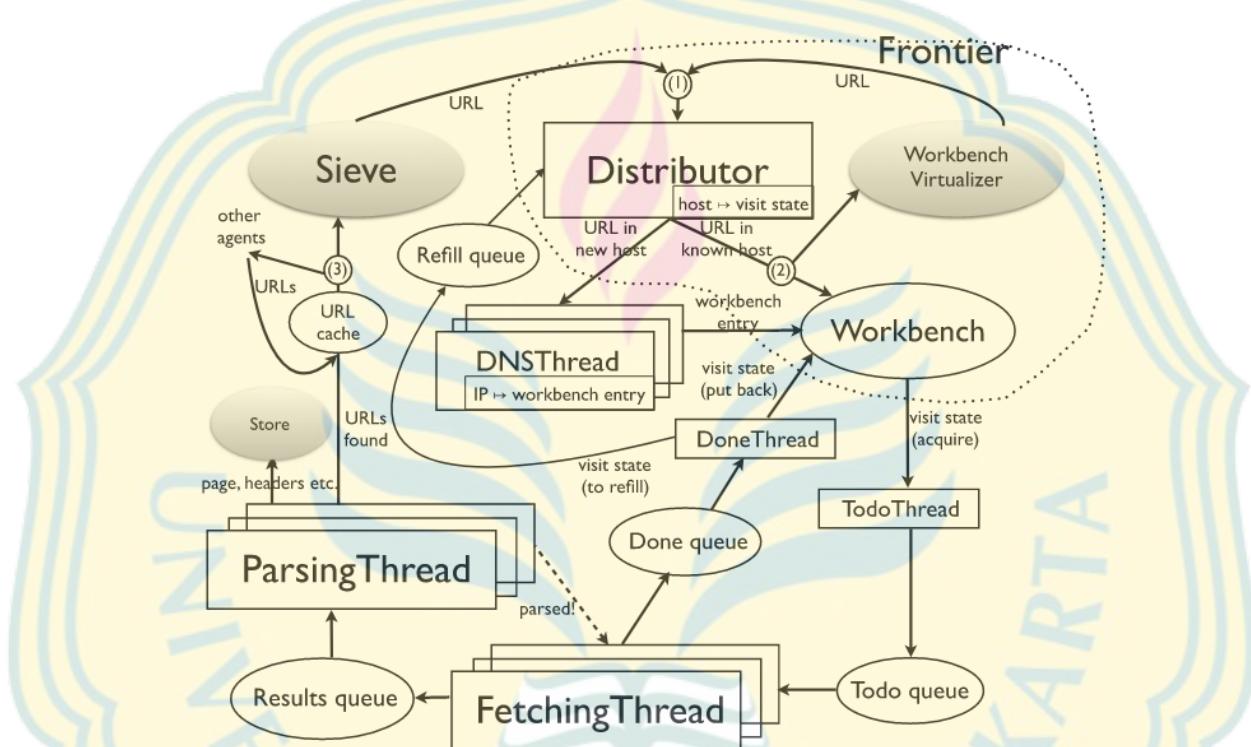
Tetapi, banyak NAT mengasosiasikan *idle timer* UDP dengan sesi UDP individual, jadi mengirim *keep-alive* pada satu sesi tidak akan membuat sesi lain tetap aktif meskipun semua sesi berasal dari titik akhir *private* yang sama. Alih-alih mengirim *keep-alive* di berbagai sesi P2P, aplikasi dapat menghindari lalu lintas *keep-alive* yang berlebihan dengan mendeteksi kapan sesi UDP tidak lagi berfungsi, dan menjalankan kembali prosedur *hole punching* lagi ‘*on demand*’.

Koneksi TCP dapat tetap dalam fase yang ditetapkan tanpa batas waktu bertukar paket apapun. Beberapa host akhir dapat dikonfigurasi untuk mengirim paket *keep-alive* pada koneksi yang *idle*; secara *default*, paket *keep-alive* dikirim setiap 2 jam jika diaktifkan. Akibatnya, NAT yang menunggu lebih dari 2 jam dapat mendeteksi koneksi *idle* dengan paket *keep-alive* dikirim secara *default*. Koneksi TCP dalam fase *partially open* atau *closing*, di sisi lain, bisa *idle* paling lama 4 menit sambil menunggu paket dalam pengiriman yang akan dikirimkan.

2.11 *Distributed Web Crawler Architecture*

Web crawler dikembangkan dari awal dikenalkannya web. Beberapa generasi pertama *web crawler* pada awal tahun 90-an yaitu: *World Wide Worm*, *RBSE Spider*, *MOMspider*, dan *WebCrawler*. Berawal dari generasi ini yang memiliki kontribusi besar terhadap algoritma utama dan *design issues* dari *web crawler*. Beberapa komersial *search engine* memiliki dan mengembangkan *crawler*-nya masing - masing. Seiring berkembangnya *crawler*, sampai pada generasi sekarang, yang mana diharapkan mampu mengunduh miliaran halaman web dengan cepat. Dan semua itu tidak ada yang tersedia secara gratis dan *open source*. BUbiNG adalah *open source crawler* pertama yang didesain untuk menjadi *crawler* yang *fast*,

scalable, dan *Runnable* di perangkat keras secara luas. Arsitektur dari *web crawler* BUbiNG akan menjadi referensi dari penelitian ini untuk menentukan distribusinya.



Gambar 2.27: Arsitektur BUbiNG

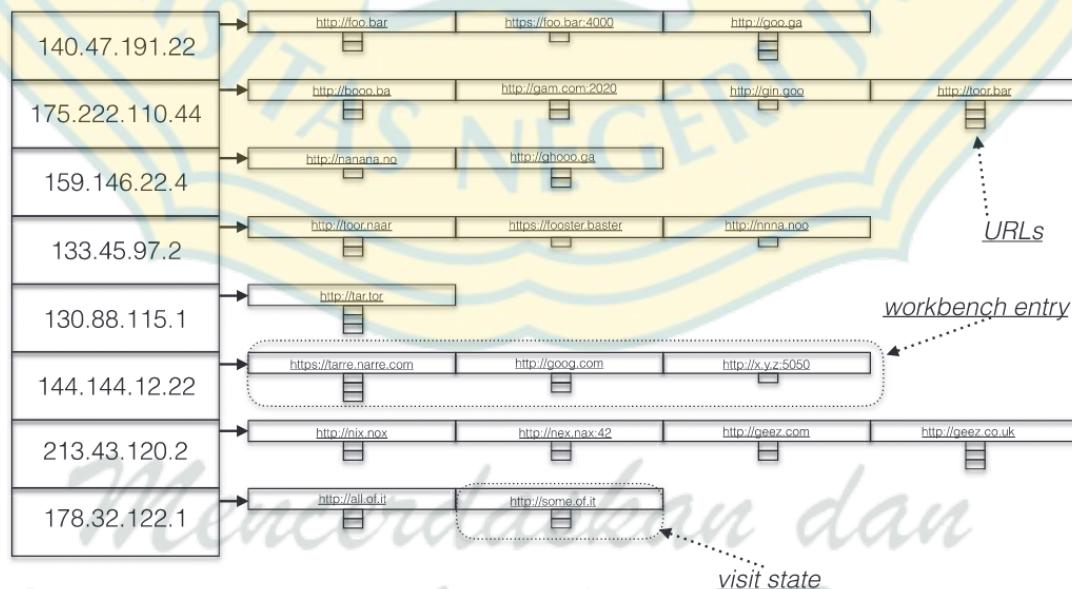
Proses *crawling* berawal dari *fetching thread*, pada tahap ini menghabiskan banyak waktu untuk menunggu data jaringan. Biasanya mengalokasikan *parsing threads* sebanyak jumlah *core* yang tersedia, karena aktivitasnya sebagian besar terikat CPU. Saat *parsing threads* menemukan URL baru, dimasukkan ke *sieve* yang melacak URL yang telah ditemukan. *Sieve* adalah struktur data yang mirip dengan antrean dengan memori: setiap elemen yang *di-queue* (masuk dalam antrean) akan *di-dequeued* (dikeluarkan dari antrean) di lain waktu, dengan jaminan bahwa suatu elemen yang diantrekan beberapa kali akan *di-dequeued* sekali saja. Dengan kata lain, *sieve* bertugas untuk menyaring URL yang ditemukan. URL ditambahkan ke *sieve* saat ditemukan dengan *parsing*. URL yang berada di-*sieve* sudah siap untuk dikunjungi dan akan diatur oleh *frontier*. *Frontier* dapat disebut sebagai kumpulan URL yang sudah ditemukan, tetapi belum di-*crawl*. Pada *frontier*, struktur data terpenting pada adalah *workbench*. Karena pada tahap tersebut, akan mengetahui *visit state* setiap *host* atau perangkat yang melakukan *crawling*. Informasi ini akan dikumpulkan untuk nantinya didistribusikan kepada perangkat lain yang

menjalankan *crawler*. Setelah itu, URL yang didapatkan akan disaring (agar tidak ada duplikasi) dan kembali ditempatkan di *workbench*.

Komponen yang aktif pada *frontier* adalah *distributor*. Tugas dari *distributor* adalah untuk mengeluarkan URL dari antrean yang berasal dari *sieve*. Melakukan pengecekan terhadap *visit state* dari masing - masing *host*, apakah sudah dikunjungi oleh *host* tersebut. Jika belum, maka akan membuat *visit state* yang baru atau mengantrekan URL ke yang sudah *visit state* yang sudah ada. Jika *visit state* yang baru diperlukan, diteruskan ke kumpulan DNS threads yang melakukan DNS resolution dan kemudian memindahkan *visit state* ke *workbench*.

2.11.1 Workbench

URL yang terkait dengan *host* tertentu disimpan dalam struktur yang disebut *visit state*, berisi antrean FIFO (*First In First Out*) dari URL berikutnya yang akan di-*crawl* untuk *host* tersebut bersama dengan *next-fetch field* yang menentukan saat pertama URL dari antrean dapat diunduh, menurut ke konfigurasi politenesss setiap *host*. *Visit state* hanya menyimpan *byte-array* representasi jalur dan kueri URL; pendekatan ini secara signifikan mengurangi pembuatan objek dan menyediakan bentuk kompresi sederhana dengan penghilangan awalan atau *prefix omission*.



Gambar 2.28: UBING Workbench

Visit state selanjutnya dikelompokkan ke dalam entri *workbench* berdasarkan

alamat IP mereka; setiap kali URL pertama untuk *host* tertentu ditemukan, *visit state* baru dibuat, lalu alamat IP ditentukan (oleh salah satu *DNS threads*): *visit state* baru dimasukkan ke dalam entri *workbench* baru (jika tidak ada *host* yang diketahui dikaitkan dengan alamat IP itu) atau yang sudah ada. *Workbench* bertindak sebagai menunda antrian (*delay queue*): operasi untuk mengeluarkan dari antrian menunggu, jika perlu, sampai *host* siap untuk dikunjungi. Pada saat itu, entri teratas dihapus dari *workbench* dan *visit state* teratas dihapus dari entri teratas. Kedua penghapusan terjadi dalam waktu logaritmik (dalam jumlah *visit state*).

Akses ke *workbench* terhimpit di antara dua antrean *lock-free* (pendekatan ini sangat berguna untuk menghindari akses langsung ke struktur data yang disinkronkan dengan waktu modifikasi logaritmik, seperti prioritas antrian, sebagai pertentangan antara *fetching thread* bisa menjadi sangat signifikan): *to-do queue* dan *done queue*. Antrian tersebut dikelola oleh dua *thread* prioritas: *to-do thread* mengekstrak *visit state* yang *host*-nya dapat dikunjungi tanpa melanggar batasan politeness dan memindahkannya ke *to-do queue*, di mana akan diambil oleh *fetching thread*; di sisi lain, *done thread* mengambil *visit state* setelah digunakan oleh *fetching thread* dan mengembalikannya ke *workbench*.

Tujuan dari pengaturan ini adalah untuk menghindari pertentangan oleh ribuan *threads* pada struktur yang relatif lambat (karena mengekstraksi dan memasukkan elemen di *workbench* membutuhkan waktu logaritmik dalam jumlah *host*). Membuat jumlah *visit state* yang siap untuk diunduh mudah diukur: ini hanya ukuran antrean tugas. Sisi negatifnya adalah pada prinsipnya menggunakan penundaan politeness per-*host* atau per-IP yang dapat menyebabkan urutan *to-do queue* tidak mencerminkan prioritas sebenarnya dari *visit state* yang terkandung di dalamnya; fenomena ini mungkin mendorong urutan global visit lebih jauh dari breath-first visit.

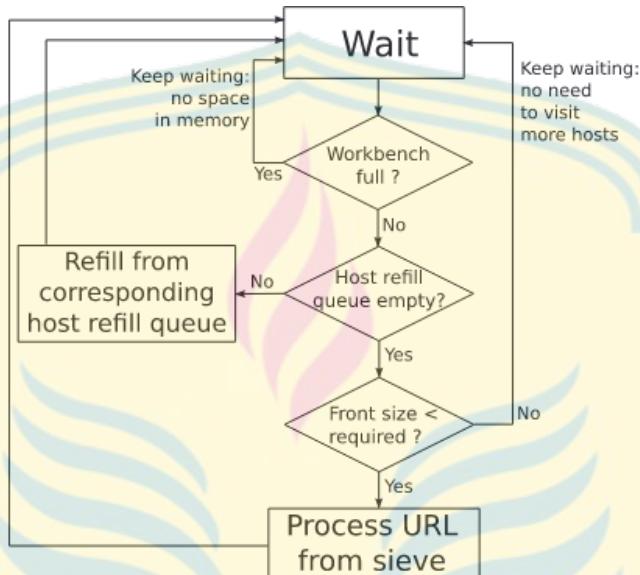
Workbench virtualizer mempertahankan pemetaan pada disk dari *host* ke FIFO antrian *virtual URL*. Secara konseptual, semua URL yang telah diekstraksi dari *sieve* tetapi belum diambil akan diantrekan di *visit state workbench* tempatnya berada, dalam urutan yang tepat saat keluar *sieve*. Karena, bertujuan melakukan *crawling* dengan jumlah memori yang konstan jumlah URL yang ditemukan, bagian dari antrian harus ditulis di disk. Setiap antrian *virtual* berisi sebagian kecil URL dari setiap *visit state*, sedemikian rupa sehingga keseluruhan urutan URL sesuai, per-*host*, berdasar breadth-first yang asli.

2.11.2 Distributor

Distributor adalah *high-priority* thread yang mengatur pergerakan URL keluar dari *sieve* dan memuat URL dari antrian *virtual* ke *workbench* seperlunya. Saat *crawling* berlangsung, URL terakumulasi dalam *visit state* dengan kecepatan berbeda, baik karena *host* memiliki daya tanggap koneksi yang berbeda dan karena situs web memiliki ukuran dan faktor percabangan yang berbeda. Selain itu, *workbench* memiliki ukuran batas yang dapat dikonfigurasi agar tidak dapat melampaui batas jumlah ukurannya. Jumlah memori utama yang ditempati tidak dapat bertambah tanpa batas dalam jumlah URL yang ditemukan, tetapi hanya dalam jumlah *host* yang ditemukan. Dengan demikian, mengisi *workbench* secara terus - menerus dengan URL yang keluar dari *sieve* akan segera menghasilkan *workbench* yang hanya berisi URL milik sejumlah *host*.

Bagian depan (front) *crawling*, pada waktu tertentu, adalah jumlah status kunjungan yang siap untuk diunduh. Ukuran front menentukan *throughput* atau *bandwidth* keseluruhan *crawler*, jumlah *host* berbeda yang sedang dikunjungi adalah yang terpenting untuk menentukan seberapa cepat atau lambat *crawling* akan dilakukan.

Salah satu dari dua kekuatan yang menggerakkan *distributor* adalah *front* harus selalu cukup besar sehingga tidak ada *fetching thread* yang harus menunggu. Untuk mencapai tujuan ini, *distributor* diperbesar secara dinamis sesuai ukuran *front* yang dibutuhkan, yang merupakan perkiraan jumlah *host* yang harus dikunjungi secara paralel untuk membuat semua *fetching thread* sibuk: setiap kali *fetching thread* harus menunggu, meskipun demikian ukuran *front* saat ini lebih besar dari ukuran *front* yang dibutuhkan saat ini, yang terakhir ditingkatkan. Setelah fase *warm-up*, ukuran *front* yang diperlukan distabilkan ke nilai yang bergantung pada jenis *host* yang dikunjungi dan pada jumlah sumber daya yang tersedia. Pada saat itu, tidak mungkin untuk memiliki *crawl* lebih cepat mengingat sumber daya yang tersedia, karena semua *fetching thread* terus mengunduh data. Meningkatkan jumlah *fetching thread*, tentu saja, dapat menyebabkan peningkatan ukuran *front* yang dibutuhkan.



Gambar 2.29: BUbiNG Distributor

Kekuatan kedua yang menggerakkan *distributor* adalah persyaratan yang dimiripkan dengan pendekatan *breadth-first visit*. Dengan catatan, kekuatan ini bekerja berlawanan arah dengan memperbesar ukuran *front* URL yang sudah ada dalam *visit state* yang seharusnya, pada prinsipnya, URL akan dikunjungi jika sudah berada pada *sieve*, tetapi memperbesar *front* memerlukan lebih banyak URL yang dikeluarkan dari antrean *sieve* untuk menemukan *host* baru.

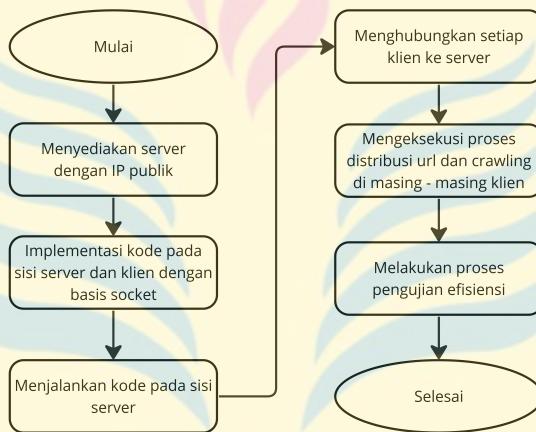
Distributor bertanggung jawab untuk mengisi *workbench* dengan URL yang akan datang, yang keluar dari *sieve* atau virtual queue. Hak istimewa *distributor* untuk mengisi ulang antrean *workbench* menggunakan URL dari *virtualizer* (lingkaran bermotor (1) pada Gambar 2.27) karena ini membuat kunjungan lebih dekat ke *breadth-first*. Nafmun, jika tidak ada isi ulang yang harus dilakukan dan *front* tidak cukup besar, *distributor* akan membaca dari *sieve*, berharap menemukan *host* baru untuk membuat *front* lebih besar.

Saat *distributor* membaca URL dari *sieve*, URL tersebut dapat diletakkan di *workbench* (lingkaran bermotor (2) pada Gambar 2.27) atau ditulis dalam antrian virtual, tergantung ada tidaknya sudah URL pada disk untuk *host* yang sama, dan jumlah URL per alamat IP yang seharusnya di *workbench* agar tetap penuh, tetapi tidak *overflowing*, bila *front* sesuai dengan ukuran yang dibutuhkan.

BAB III

METODOLOGI PENELITIAN

3.1 Tahapan Penelitian



Gambar 3.1: Flowchart Tahapan Penelitian Crawler Terdistribusi

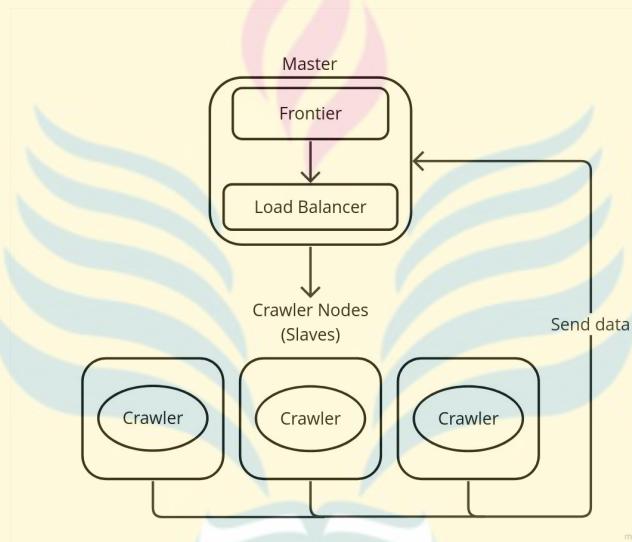
Socket berperan penting dalam proses distribusi *url* untuk melakukan crawling pada setiap klien. Karena socket sebagai pokok transportasi *url* dari satu perangkat ke perangkat yang lain melalui media internet.

Kode program crawler yang ada saat ini hanya menerima masukan *url* dari perangkat yang menjalankan crawler tersebut. Sedangkan, pada penelitian ini *url* yang diterima didapatkan dari pembagian dari perangkat lain. Menjadikan klien yang terhubung dapat menjalankan crawling secara bersamaan dan tidak melakukan crawling dengan *url* yang sama. Jadi tidak ada duplikasi data dari setiap kliennya.

3.2 Arsitektur Crawler Terdistribusi

Sistem yang akan dibuat pada penelitian ini adalah pengembangan *crawler* menjadi secara terdistribusi yang dapat berjalan atau melakukan *crawling* pada banyak perangkat sekaligus secara terkontrol. Pengembangan ini menggunakan koneksi socket untuk komunikasi. Gambaran awal arsitektur ini adalah *master-slave*. *Master* bertugas untuk mengelola semua *slave* yang tersedia, melakukan *write*

operation ke dalam database, dan membagi serta menyeimbangkan tugas ke masing - masing *slave*. Sedangkan *slave* bertugas untuk melakukan *crawling* dari url yang diberikan oleh *master* dan akan mengembalikan data hasil *crawling* kembali ke *master* untuk dimasukkan ke database, lalu menyeimbangkan kembali url untuk di-*crawling*.

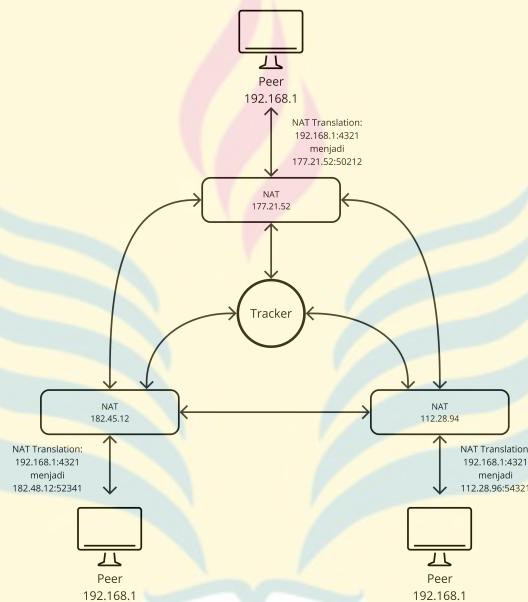


Gambar 3.2: Arsitektur *Crawler Master-Slave*

1. *Frontier* adalah antrian dari sekumpulan url yang belum dikunjungi dan sudah dikunjungi oleh *crawler*. Url yang terdapat pada *Frontier* sudah terkuras agar tidak ada url yang terduplikat, guna menghindari melakukan *crawling* pada halaman web yang sama. Kemudian *crawler* akan menerima url dari *frontier* dan melakukan *crawling* pada halaman dari url tersebut.
2. Load balancer bertugas sebagai scheduler untuk mengatur urutan antrian dari url yang akan didistribusikan kepada *crawler*. Mendistribusikan url ke setiap *crawler nodes* agar setiap *crawler* memiliki beban kerja yang seimbang. *Load balancer* memiliki sekumpulan antrian url dari masing - masing *crawler*.
3. *Crawler nodes* adalah perangkat yang akan melakukan *crawling*, url untuk melakukan *crawling* didapatkan dari *frontier* dan hasil dari *crawling* disimpan pada database.

Berdasarkan arsitektur pada Gambar 3.2, terdapat beberapa pengembangan pada tahapan komunikasi antara *slave* atau klien atau *peer*. Proses komunikasi

dilakukan secara langsung antara *peer*, disebut sebagai *peer-to-peer*. Karena komunikasi ini terjadi antara *peer* yang memiliki *private IP* dapat saling berkomunikasi.



Gambar 3.3: Arsitektur Crawler Peer to Peer Melalui NAT

1. *Tracker* atau *master* bertugas untuk mengetahui eksternal atau publik IP address dan port dari masing - masing *peer* yang terhubung. *Tracker* adalah sebuah server yang memiliki publik IP. Setelah mengetahui eksternal IP address dan port setiap *peer*, maka *peer* dapat terhubung dengan satu sama lain. Mengoleksi kumpulan *peer* yang terhubung. Setup initial url untuk *crawling*. Melakukan broadcast secara berkala untuk update *peer* yang terhubung dan keluar dari swarm.
2. *Peer* dapat disebut klien. Setiap *peer* memiliki duplikasi dari kumpulan *peer* yang terhubung dengan *tracker*. Melakukan proses *crawling*. Mengoleksi sekumpulan url untuk di-*crawl*. Mengambil bagian url yang akan di-*crawl* untuk setiap *peer*, saat mengambil url juga mengeluarkannya dari sekumpulan url, sekali mengambil url langsung cukup banyak, bergantung dengan jumlah *peer* yang terhubung, baru di-broadcast kepada semua *peer* yang terhubung atau melalui perantara *tracker* mengenai sekumpulan url yang baru. Minimal menyisakan url lebih banyak dari jumlah *peer* yang terhubung. Agar semua

peer mendapatkan bagian url yang seimbang. Memasukkan data hasil *crawling* ke database.

3.3 Proses Komunikasi *Peer-to-peer*

Proses komunikasi menggunakan metode UDP *hole punching*, dengan membuat ‘lubang’ pada NAT, agar antara *peer* dapat saling terhubung antara satu sama lain melalui ‘lubang’ tersebut. Menggunakan socket protokol untuk mengirim packet kepada *tracker* untuk mengetahui publik IP address dari *peer* yang mengirim packet tersebut. Karena *peer* berada di belakang NAT atau router. Jadi IP address dan portnya akan diwakilkan oleh publik IP address dan port dari NAT. Pentingnya untuk mengetahui publik IP address dan portnya, agar *peer* lain dapat terhubung melalui publik IP address dan port dari NAT tersebut. Langkah tersebut dapat dilakukan karena sudah ada lubang pada NAT. Ketika *tracker* suatu saat mati dengan sendirinya, proses komunikasi tetap dapat berlanjut, karena transfer data tidak melalui *tracker*. Melainkan menuju langsung ke *peer*.

Algorithm 1 Algoritma Tracker

```
sock.bind('0.0.0.0', port_number)                                ▷ Bind socket
clients[]                                                        ▷ Inisiasi clients
```

```
function LISTENCONNECTEDPEERS
    while True do
        data, address ← sock.recv()
        APPEND(clients[], address)
    end while
end function
```

```
listener ← ListenConnectedPeers()
listener.start()

for client ∈ clients[..] do
    address, port ← EXTRACT(client)
    sock.sendto(address, port, destination_address)
end for
```

Tracker menjadi sebagai perantara atau *rendezvous* antara klien untuk dapat berkomunikasi di belakang NAT. Tracker akan mengembalikan masing - masing alamat IP dan port dari klien atau *peer* yang terhubung. Dan alamat tersebut bersifat private, karena diwakilkan oleh NAT dari router.



*Mencerdaskan dan
Memartabatkan Bangsa*

Algorithm 2 Algoritma Peer

```

peers[address : value, port : value]                                ▷ Inisiasi peers
rendezvous ← (server_ip, server_port)
sock.bind('0.0.0.0', port_number)
sock.sendto(b'0', rendezvous)                                         ▷ Bind socket
                                                               ▷ Mengirim pesan kosong ke server

function LISTENPEERSFROMTRACKER
    while True do
        data ← sock.recv()
        APPEND(peers[], data)
    end while
end function
listenerPeers ← ListenPeersFromTracker()
listenerPeers.start()

function LISTENTOHOLEPUNCH
    for peer ∈ peers[..] do
        address, port ← EXTRACT(peer)
        sock.sendto(b'0', port, address)
    end for
end function
listenerHolePunch ← ListenToHolePunch()
listenerHolePunch.start()

function LISTENCONNECTIONFROMPEERS
    while True do
        data ← sock.recv()                                         ▷ Mendapatkan data dari peer
    end while
end function
listenerConn ← ListenConnectionFromPeers()
listenerConn.start()

while True do
    msg ← input()
    sock.sendto(msg, port, address)                               ▷ Mengirim data ke peer secara spesifik
end while

```

Terdapat proses mengirim pesan kosong ke server, berguna untuk mengetahui alamat IP dan port dari klien yang diwakilkan oleh NAT, karena klien bisa saja memiliki *private IP* dan menyimpannya. Pada fungsi *ListenPeersFromTracker* untuk mendapatkan data alamat dan port dari klien lain, yang nantinya akan digunakan untuk melakukan *hole punching*. Setelah proses itu, kedua atau lebih klien dapat bertukar data secara langsung antara satu sama lain, walaupun klien memiliki *private IP*.

3.4 Penyeimbang Antrian URL

Tahapan untuk mengatur urutan antrian url pada *load balancer* terhadap sejumlah *peer* yang tersedia. Setiap *peer* memiliki antriannya sendiri. Gunanya antrian yang berada pada *load balancer* adalah untuk mengatur beban kerja dari *peer*. Agar setiap *peer* mendapatkan tugas yang seimbang untuk melakukan crawling. Ketika salah satu *peer* memiliki antrian url yang banyak dan tidak seimbang dengan *peer* yang lain. Maka url yang berasal dari frontier akan diantrikan ke *peer* yang lain dengan jumlah antrian yang lebih sedikit.

Algorithm 3 Struktur Data

```

struct PEER queue: list, socket_address: str
    self.queue ← queue
    self.socket_address ← socket_address
end struct

struct LOADBALANCER queues: list, p: Peer
    self.queues ← queues
    self.peers.append(p)
end struct

```

*Mencerdaskan dan
Memartabatkan Bangsa*

Algorithm 4 Algoritma Penyeimbang Antrian

balancer = LoadBalancer

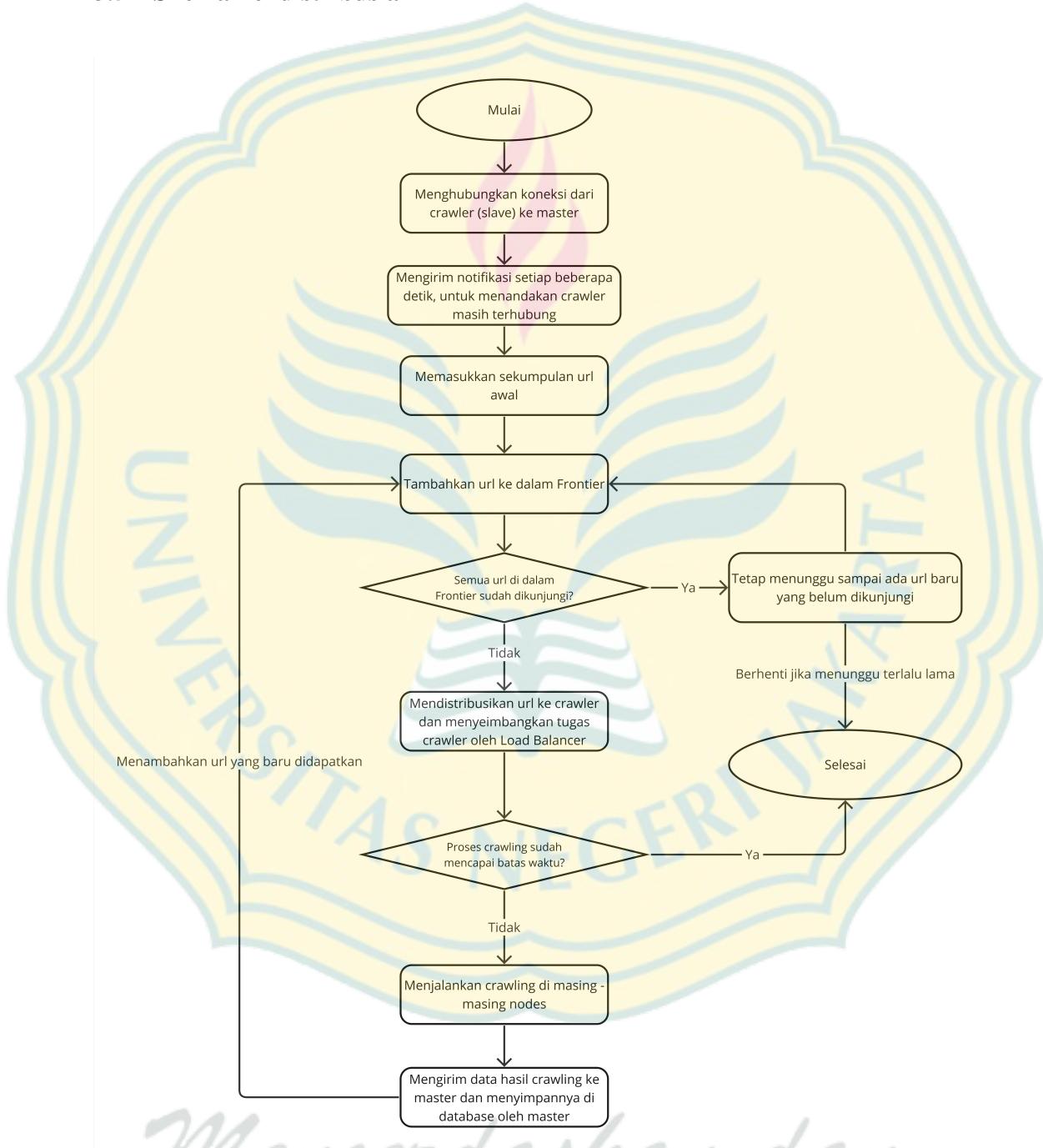
queues = balancer.queues

```
function BALANCINGQUEUE(list_of_url)
    for url ∈ list_of_url[..] do
        min_length = INFINITY
        min_queue_index = None

        for i, current_queue ∈ enumerate(queues[..]) do
            current_length = current_queue.length()
            if current_length == 0 then
                min_queue_index = i
                break
            else if current_length < min_length then
                min_length = current_length
                min_queue_index = i
            end if
        end for
    end for
    queues[min_queue_index].enqueue(url)
end function
```

Mencerdaskan dan
Memartabatkan Bangsa

3.5 Skema Pendistribusian



Gambar 3.4: Flowchart Skema Pendistribusian

Proses dari sistem *crawler* terdistribusi dimulai dengan memasukkan sekumpulan url awal yang ingin di-*crawling* ke dalam *frontier*. Lalu, melakukan pengecekan terhadap url yang sudah dikunjungi di *frontier*. Jika kondisinya tidak,

maka akan melanjutkan untuk mendistribusikan url ke masing - masing *crawler* dan menyeimbangkan beban kerjanya. Setelah itu melakukan *crawling* oleh *crawler* dan data disimpan di database. Data yang didapatkan dari *crawling* akan dikembalikan ke *frontier* yang berbentuk outgoing url dan akan dilakukan penyaringan url yang belum dikunjungi. Ketika kondisi semua url pada *frontier* sudah dikunjungi maka akan menunggu sampai ada url baru yang belum dikunjungi. Akan *suspend* ketika terlalu lama menunggu.

3.6 Alat dan Bahan Penelitian

Pada penelitian ini, terdapat beberapa alat yang digunakan sebagai penunjang dalam pembuatan sistem terdistribusi dengan rincian sebagai berikut:

- Dua laptop dengan konfigurasi Intel Core i7-8650U, 16GB RAM dan Intel Core i5-8265U, 20GB RAM
- Sistem operasi *Linux*
- *VSCode* sebagai *code editor*
- Database *SQLite3* untuk menyimpan data
- *Python 3*
- *Virtual private server* untuk menjalankan program yang memerlukan public IP address

3.7 Tahapan Pengembangan

3.7.1 Improvisasi *crawling* secara terdistribusi

Proses improvisasi akan dilakukan dengan memodifikasi *crawler* agar dapat berjalan secara terdistribusi, serta meningkatkan konsistensi data dan efisiensi sumber daya. Penelitian ini akan mengembangkan versi *crawler* terdahulu (Khatulistiwa 2023).

Hasil penelitian tersebut menghasilkan *crawler* yang dapat menghimpun data daribagai halaman *web* dengan tambahan *multi-threading* yang dapat memaksimalkan proses kerja dengan membagi pekerjaan ke *thread* yang lain. Karena hasil penelitian sebelumnya masih berjalan dalam satu perangkat saja. Dan

tidak dapat berkoordinasi dengan perangkat lain. Maka, penelitian ini akan mengembangkan menjadi terdistribusi dengan mekanisme *socket programming*.

3.7.2 Skenario Eksperimen

Berikut adalah skenario eksperimen agar perangkat dapat berkomunikasi satu sama lain dengan *socket* dan melakukan *crawling*.

1. Menjalankan program tracker pada server dengan public IP
2. Menjalankan program manager pada server dengan public IP. Dan terhubung ke tracker
3. Menjalankan program klien, pada server dengan public IP dan ada juga yang berjalan di private IP nantinya klien akan terkurasai yang memiliki IP public dan private. Dan terhubung ke tracker
4. Manager melakukan binding koneksi, agar klien public dapat terhubung kepadanya dengan mengirim info ke tracker
5. Tracker menginformasi klien public untuk mencoba terhubung ke manager dengan memberikan IP address dan portnya
6. Klien public menerima informasi tersebut dan menghubungkan koneksi ke manager
7. Setelah berhasil, klien public akan melakukan binding koneksi agar semua klien private dapat terhubung dengannya. Lalu memberikan informasi kepada tracker terkait hal itu
8. Tracker menginformasi semua klien private untuk mencoba terhubung ke klien public dengan memberikan IP address dan portnya
9. Klien private menerima informasi tersebut dan menghubungkan koneksi ke klien public
10. Manager mengirim starting url yang diperlukan kepada klien public yang terhubung
11. Klien public menerima starting url dan mengirimkan kepada klien private (yang aka melakukan *crawling*)

12. Klien private melakukan *crawling* dan setiap selesai satu url. Data hasilnya (konten dari web yang di-*crawl* dan sekumpulan url baru yang ditemukan) akan dikirimkan ke klien public
13. Klien public akan menyimpan data konten hasil *crawling* di database. Dan sekumpulan url yang didapat akan dicek apakah ada duplikasi atau tidak
14. Klien public kembali mengirim url yang akan di-*crawling* kepada semua klien private dengan mempertimbangkan "kesehatan" dari klien tersebut dan jumlah url yang berada di klien itu. Agar pembagian menjadi seimbang

3.8 Skema Uji

Pengujian yang akan dilakukan pada penelitian ini adalah untuk menguji konsistensi data dan efisiensi dengan crawler terdahulu.

3.8.1 Sumber Data

Sumber data yang akan digunakan dalam penelitian ini adalah situs web berita dan data yang diambil itu mencakup konten teks dari halaman web serta tautan yang berada di web tersebut.

3.8.2 Metrik Pengujian

Performa crawler terdistribusi akan dinilai menggunakan metrik berikut:

1. **Proses Komunikasi:** Proses komunikasi antara masing - masing perangkat dapat berjalan dengan sistematis.
2. **Konsistensi Data:** Konsistensi data mengacu pada kemampuan mengirimkan data lengkap tanpa kehilangan atau duplikasi informasi selama proses *crawling*. Konsistensi data penting untuk memastikan keakuratan dan kelengkapan informasi yang dikumpulkan di Internet. Duplikasi atau hilangnya data dapat menyebabkan analisis yang tidak akurat dan berdampak pada keputusan berdasarkan data tersebut. Dengan pengujian analisis data tersimpan dengan membandingkan data yang dikumpulkan dengan sumber atau data yang diharapkan untuk memastikan tidak ada duplikasi.

3. **Efisiensi dan Optimalisasi:** Efisien sumber daya dan optimalisasi jumlah data yang terkumpul dari proses *crawling* terdistribusi dengan *crawling* tidak terdistribusi. Ini mencakup seberapa cepat sistem dapat mengumpulkan data. Karena proses crawling yang lambat atau penggunaan sumber daya yang tidak efisien dapat menghambat kinerja *crawler*. Dengan pengujian penyimpanan data, data yang dihasilkan oleh *crawler* terdistribusi dapat memiliki jumlah data yang lebih banyak dalam kurun waktu yang sama, yaitu satu jam.

3.8.3 Rancangan Eksperimen

1. Menjalankan kedua versi crawler secara bersamaan dengan *initial url web* yang sama.
2. Catat jumlah data yang terkumpul dalam jangka waktu tertentu

Skema uji ini akan membantu dalam membandingkan kinerja antara crawler terdistribusi dan terdahulu (tanpa terdistribusi).

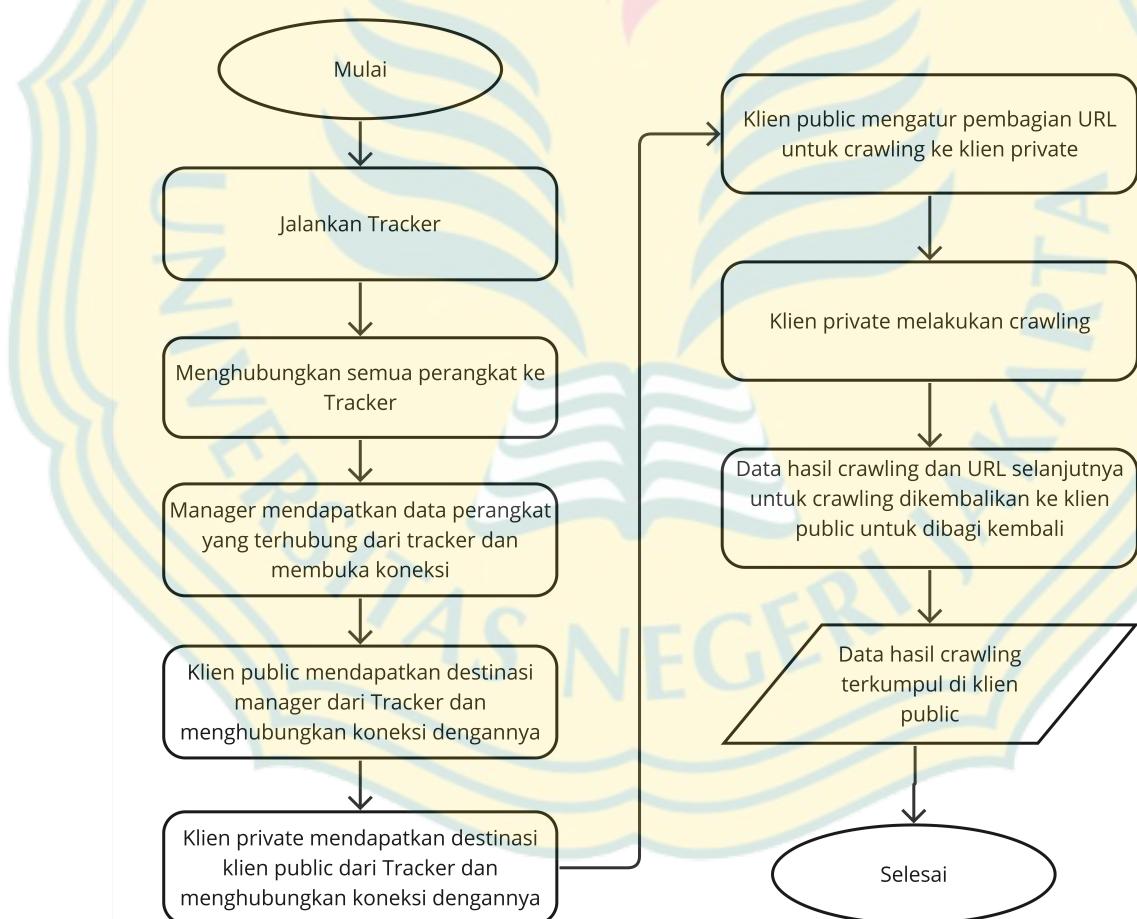
*Mencerdaskan dan
Memartabatkan Bangsa*

BAB IV

HASIL DAN PEMBAHASAN

4.1 Implementasi

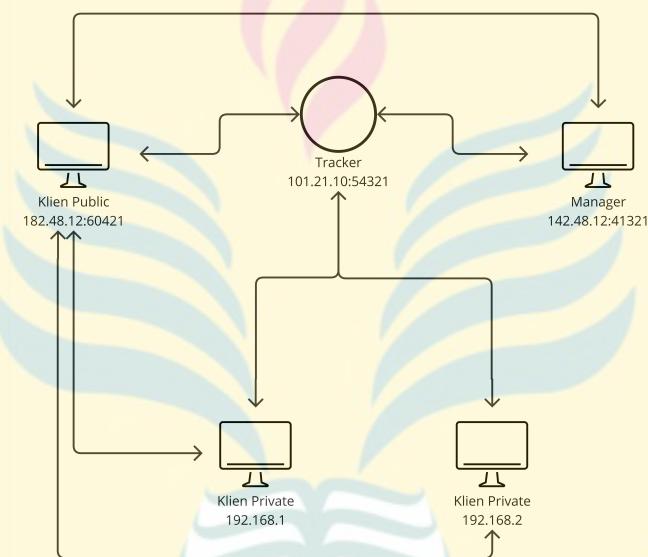
Berikut ini adalah keterangan *flowchart* terkait tahapan penelitian yang sudah berhasil dibuat. Penyempurnaan dari *flowchart* pada Gambar 3.1.



Gambar 4.1: *Flowchart* Tahapan Penelitian yang Berhasil Dibuat

Arsitektur *crawler* terdistribusi dalam penelitian ini terdiri dari lima perangkat lunak dengan konfigurasi tiga perangkat memiliki *public IP address* dan dua perangkat memiliki *private IP address*. Dengan pembagian perannya masing-masing, satu *Tracker*, satu *Manager*, dan tiga *Klien*. Pada penerapannya setiap

program berjalan dengan bahasa pemrograman Python dan setiap proses dapat berjalan secara paralel dengan menggunakan *threading*. Arsitektur ini telah mengalami modifikasi dari yang sebelumnya pada Gambar 3.2. Setelah melakukan percobaan untuk mendapatkan kondisi *peer to peer*, maka dihasilkan arsitektur seperti Gambar 4.2.

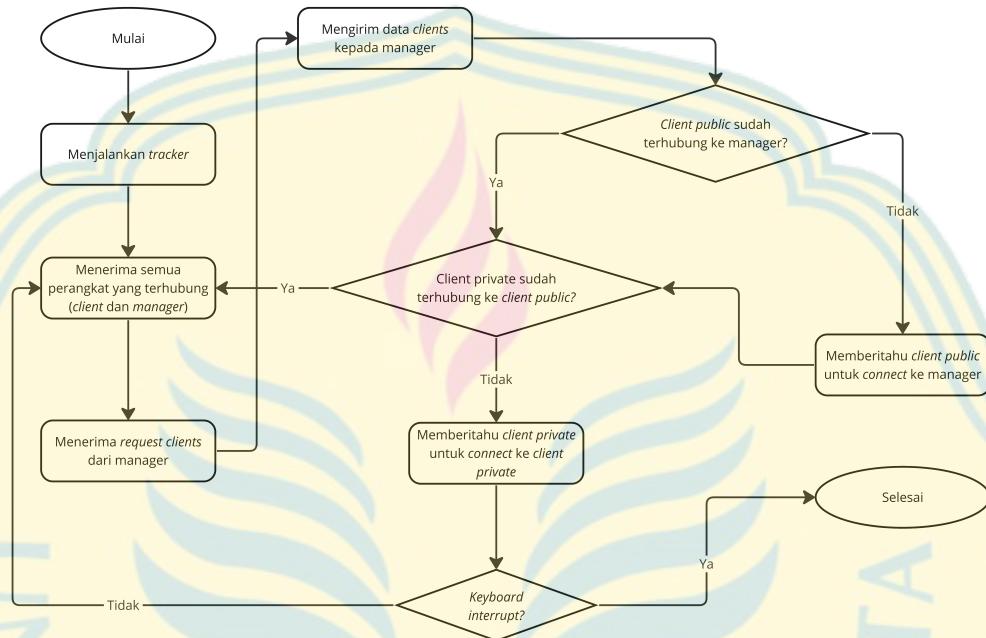


Gambar 4.2: Arsitektur Crawler Terdistribusi

4.1.1 *Tracker*

Tracker akan memiliki *public IP address*, dikarenakan *tracker* memiliki tugas sebagai jalur informasi antara perangkat yang terhubung. Setiap perangkat harus terlebih dahulu untuk terhubung melalui *tracker* dan juga mengirimkan informasi apakah perangkat tersebut memiliki *public IP address* atau tidak dan tipe dari perangkat tersebut, apakah dia *manager* atau klien. Lalu, *tracker* akan menampung semua perangkat yang terhubung.

*Mencerdaskan dan
Memartabatkan Bangsa*



Gambar 4.3: Flowchart tracker

- Potongan kode untuk mengatur perangkat atau klien yang baru terhubung

```

def handle_client(client):
    while True:
        data = client.client_socket.recv(1024)
        if not data:
            break # Connection closed by client
        data = data.decode('utf-8')

        # Handle the data received from the client
        data_json = parse_json(data)
        if data_json:
            if client.type is None:
                client.update_info(data_json['type'], data_json['is_private'])
                print(f'Received from {client.address}: {data_json}')
                global client_public_ready, client_public_destination
                if client_public_ready == True:
                    for client_private in filter(lambda c: c.is_private, clients):
                        client_private.client_socket.send(json.dumps(
                            {
                                "msg": "CLIENT_PUBLIC_IP_READY", "client_public_destination": client_public_destination
                            }).encode())

```

Gambar 4.4: Potongan kode tracker untuk klien baru terhubung

Setiap klien yang terhubung akan diterima oleh *tracker* dan pesannya akan disimpan.

- Potongan kode untuk merespons *manager* yang meminta kumpulan klien

```
# If the client is a manager and requests the list of clients
if client.type == 'manager':
    if data_json.get("msg") == "REQUEST_CLIENTS":
        # Send the list of connected clients to the manager
        client_info_list = [
            {"address": c.address, "type": c.type, "is_private": c.is_private}
            for c in clients
        ]
        print("clients:", client_info_list)
        response_data = {
            "msg": "CLIENT_LIST",
            "data": client_info_list
        }
        client.client_socket.send(json.dumps(response_data).encode())
```

Gambar 4.5: Potongan kode *tracker* merespons *manager* yang meminta kumpulan klien

Setiap kali manager meminta data klien kepada *tracker*. *Tracker* akan merespons pesan tersebut dengan mengirim data klien kepada manager.

- Potongan kode untuk merespons *manager* sudah siap membuka koneksi

```
elif data_json.get("msg") == "MANAGER_READY":
    global manager_ready
    manager_ready = True
    print('manager ready?', manager_ready)
    client_destination = tuple(data_json.get("client_destination", ()))
    client_info_list = [
        {
            "address": client.address,
            "type": client.type,
            "is_private": client.is_private,
        }
    ]

    response_data = {
        "msg": "MANAGER_READY",
        "data": client_info_list
    }

    print('notify client')
    # Search for the client with the target address
    target_client = next((client for client in clients if client.address == client_destination), None)
    target_client.client_socket.send(json.dumps(response_data).encode())
```

Gambar 4.6: Potongan kode *tracker* merespons *manager* sudah siap membuka koneksi

Manager memberitahukan *tracker* bahwa manager sudah siap membuka koneksi untuk dihubungkan oleh klien *public*.

- Potongan kode untuk memberitahu klien *private* bahwa dapat terhubung ke *manager*

```

if data_json.get("msg") == "CLIENT_CONNECTED_TO_THE_MANAGER":
    client_public_ready = True
    print("received data:", data_json)
    client_public_destination = client.address
    for client_private in filter(lambda c: c.is_private, clients):
        client_private.client_socket.send(json.dumps({"msg": "CLIENT_PUBLIC_IP_READY", "client_public_destination": client.address}).encode())

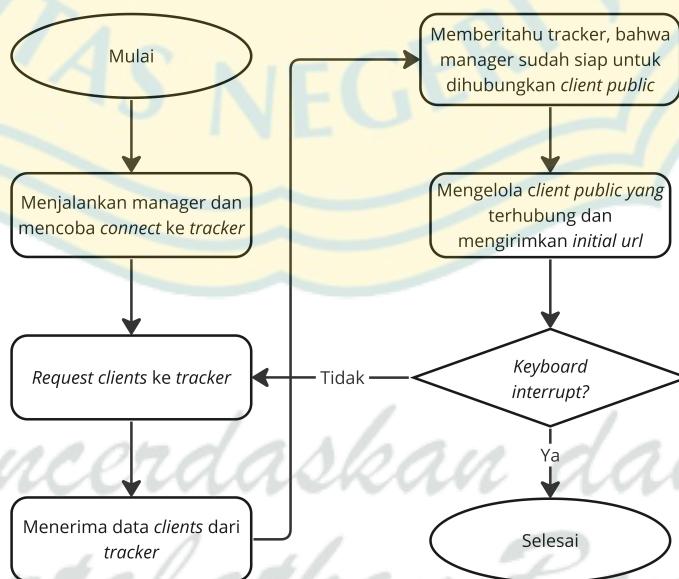
```

Gambar 4.7: Potongan kode *tracker* memberitahu klien *private* untuk *connect*

Setelah manager sudah siap untuk dihubungkan dengan klien *public*. Lalu klien *public* mengirimkan kembali info ke *tracker* bahwa sudah terhubung ke manager. Lalu manager mengirim info ke klien *private* untuk dapat terhubung ke klien *public*.

4.1.2 Manager

Manager akan memiliki *public IP address*, dikarenakan *manager* memiliki tugas untuk mengatur perangkat terhubung yang nantinya akan memiliki *public IP address*. *Manager* juga perlu untuk terhubung ke *tracker*. *Manager* mendapatkan klien yang terhubung dengan meminta data klien yang terhubung kepada *tracker*. Lalu, *tracker* akan mengirimkan data klien yang terhubung. *Manager* akan membuka koneksi untuk klien yang nantinya memiliki *public IP address* untuk terhubung dengannya.



Gambar 4.8: Flowchart manager

- Potongan kode untuk menerima kumpulan klien dari *tracker* dan mengirim kembali ke *tracker* bahwa *manager* sudah bersedia untuk dihubungkan dengan klien *public*

```

if response_data.get("msg") == "CLIENT_LIST":
    new_clients_info = response_data.get("data", [])
    for client_info in new_clients_info:
        address = tuple(client_info.get("address", ()))
        type = client_info.get("type", None)
        is_private = client_info.get("is_private", True)

        # Check if a client with the same address already exists
        existing_client = next((c for c in clients if c.address == address), None)

        if existing_client:
            # Update the existing client information if needed
            existing_client.type = type
            existing_client.is_private = is_private
        else:
            # Add a new client if it doesn't exist
            new_client = Client(address, type=type, is_private=is_private)
            clients.add(new_client)

        # Check if the new client has a public IP
        if type == 'client' and not is_private:
            print('public:', address)
            print('notify tracker')
            tcp_sock.send(json.dumps({"msg": "MANAGER_READY", "client_destination": address}).encode())

```

Gambar 4.9: Potongan kode *manager* yang menerima data klien dan menginformasikan *tracker*

- Potongan kode untuk membuka koneksi agar klien *public* dapat terhubung dan inisisasi *starting url*

```

load_dotenv()
start_urls = os.getenv("CRAWLER_START_URLS").split()
print(f"TCP Manager listening on {manager_sock.getsockname()[0]}:{manager_sock.getsockname()[1]}")
tcp_client_sock, tcp_client_address = manager_sock.accept()
print(f"Accepted connection from {tcp_client_address}")

tcp_client_sock.send(json.dumps({"msg": "CRAWLER_START_URLS", "start_urls": start_urls}).encode())
threading.Thread(target=handle_client_tcp, args=(tcp_client_sock,), daemon=True).start()

```

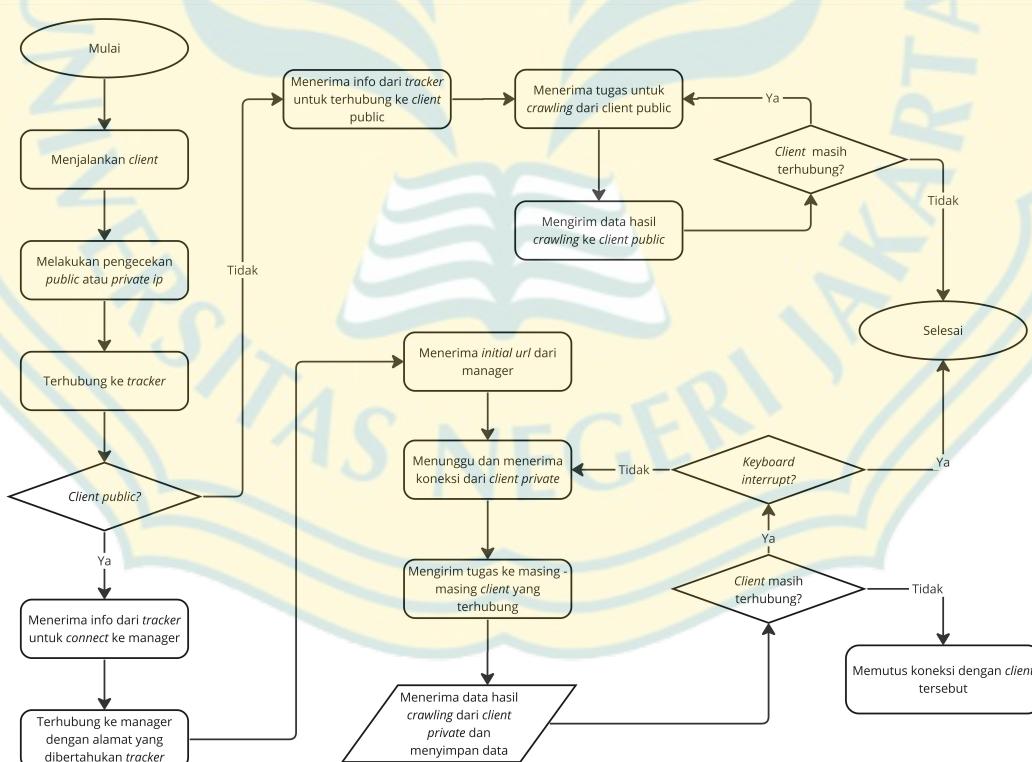
Gambar 4.10: Potongan kode *manager* untuk membuka koneksi agar klien *public* dapat terhubung

4.1.3 Klien

Klien dapat memiliki *public* dan *private IP address*. Dalam penelitian ini terdapat 1 klien dengan *public IP address* dan 2 klien dengan *private IP address*, yang nantinya klien dengan *private IP address* akan menjadi *worker* yang melakukan *crawling*. Setelah klien *public* terhubung dengan *tracker*. Dan *tracker* juga sudah mendeteksi adanya *manager* yang sudah terhubung. Maka, *tracker* akan mengirim informasi *manager* kepada klien tersebut. Agar klien tersebut dapat terhubung dengan *manager*. Untuk klien *private* nantinya akan terhubung dengan

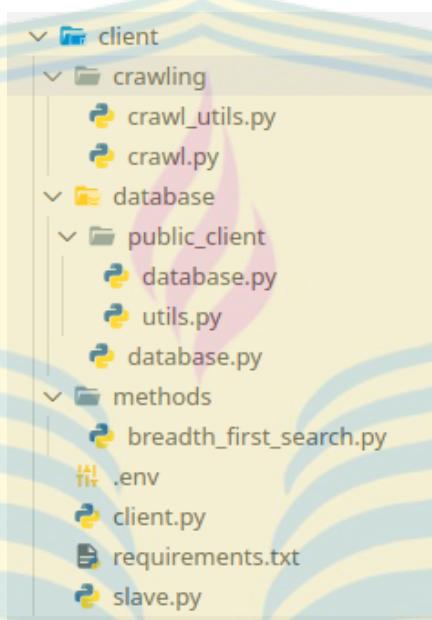
klien *public* setelah mendapatkan informasi dari *tracker* bahwa klien *public* sudah siap untuk dihubungkan. Klien *private* akan melakukan *crawling* yang mana pembagian url-nya akan diatur oleh klien *public*. Data hasil *crawling* akan dikembalikan ke klien *public* untuk disatukan.

Proses *crawling* harus dilakukan oleh klien *private* dikarenakan jika dilakukan oleh klien *public* akan berdampak pada *Domain Name System* (DNS) klien tersebut yang dapat diblokir. Karena situs web memiliki syarat dan kebijakan yang memiliki batasan terhadap *crawling* otomatis, dan dengan penggunaan klien *private* dapat meminimalkan risiko. Klien *private* juga membantu untuk menghindari batasan tingkat permintaan (*rate limiting*), karena kalau dengan klien *public* peraturannya akan lebih ketat karena terikat dari penyedia layanan internet atau server layanan.



Gambar 4.11: Flowchart client

- Struktur direktori dari kode klien



Gambar 4.12: Struktur direktori kode klien

Pada direktori "*client*" terdapat tiga buah folder penunjang. "*crawling*", folder ini berisikan fungsi utama untuk melakukan *crawling* serta utilitasnya. "*database*", folder ini berisikan fungsi untuk mengelola basis data, mulai dari menentukan table serta attributnya. Di folder tersebut juga terdapat "*public_client*" folder, yang berguna untuk klien *public* mendefinisikan basis datanya. "*methods*", pada folder ini berisikan metode yang digunakan untuk melakukan *scraping data* dari internet berdasar url yang ditentukan.

File utama yang akan dijalankan adalah file "*client.py*". Pada file tersebut berisikan proses dari klien penentuan *public* atau *private* klien berdasarkan perangkat yang menjalankan program, terhubung dengan *tracker*, dan melakukan komunikasi antar klien. Juga terdapat file "*.env*" sebagai penentuan ekosistem program, seperti berapa lamanya *worker* atau klien *private* akan bekerja. File "*requirements.txt*" berisikan modul - modul yang diperlukan untuk menjalankan program. File "*slave.py*" juga tidak kalah penting, karena didalamnya terdapat fungsi - fungsi untuk membantu pengiriman data ke klien *public*. Pada penelitian ini akan lebih mendalam membahas tentang proses distribusinya. Karena untuk metode *crawling* sebagian besar akan sama dengan penelitian terdahulu (Khatulistiwa 2023).

- Fungsi untuk mendapatkan IP *address* apakah *public* atau *private*

```

def get_ip_address():
    try:
        # Get the local machine's IP address
        my_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        my_socket.connect(("8.8.8.8", 80))
        ip_address = my_socket.getsockname()[0]
        my_socket.close()
        return ip_address
    except Exception as e:
        print("Error:", e)
        return None

def is_private_ip(ip_address):
    try:
        ip = ipaddress.ip_address(ip_address)

        # Check if it's the loopback address
        if ip == ipaddress.IPv4Address("127.0.0.1"):
            return True

        private_ip_ranges = [
            ipaddress.IPv4Network("10.0.0.0/8"),
            ipaddress.IPv4Network("172.16.0.0/12"),
            ipaddress.IPv4Network("192.168.0.0/16"),
        ]

        # Check if the IP address is within any of the private ranges
        for private_range in private_ip_ranges:
            if ip in private_range:
                return True
    return False

```

Gambar 4.13: Fungsi untuk mendapatkan IP *address* apakah *public* atau *private*

Pada fungsi ini dapat menentukan IP *address*-nya dengan cara mendapatkan *socket address* yang sudah terhubung dengan *socket* lain. Dan *address*-nya akan dibandingkan dengan IP *address* *private*. Kalau tidak sama, berarti *address* tersebut adalah *public*. Kalau sama, berarti *private*.

4.1.3.1 Klien Public

- Pendeklarasian kelas *Client*

```
class Client:
    def __init__(self, unique_identifier, socket):
        self.unique_identifier = unique_identifier
        self.socket = socket
        self.queue = []
        self.average_crawling_time = [] # contain list of average 10 url crawling time
        self.health = 100
        self.lock = threading.Lock()

    def append_queue(self, urls):
        for url in urls:
            with self.lock:
                self.queue.append(url)

    def remove_queue(self, urls):
        with self.lock:
            for url in urls:
                if url in self.queue:
                    self.queue.remove(url)
```

Gambar 4.14: Pendeklarasian kelas *Client*

- Pendeklarasian kelas *LoadBalancer*

```
class LoadBalancer:
    def __init__(self, queue = list):
        self.slave_connections = {}
        self.visited_urls = set()
        self.queue = queue
        self.lock = threading.Lock()
        self.db_public = DatabasePublic()
        self.db_utils = DatabaseUtils()
```

Gambar 4.15: Pendeklarasian kelas *LoadBalancer*

- Potongan kode *public* untuk merespons komunikasi dengan *tracker* untuk mendapatkan informasi ketika *manager ready* dan menginformasikan klien *private* untuk terhubung

```

while True:
    try:
        data = tcp_sock.recv(1024)

        if not data:
            break

        response_data = json.loads(data.decode('utf-8'))
        if response_data.get("msg") == "MANAGER_READY":
            global manager_address
            manager_info = response_data.get("data", [])[0]
            manager_address = tuple(manager_info.get("address", ()))
            print('manager address:', manager_address)
        elif response_data.get("msg") == "CLIENT_PUBLIC_IP_READY":
            global client_public_ready, client_public_address
            client_public_ready = True
            client_public_address = tuple(response_data.get("client_public_destination", ()))

```

Gambar 4.16: Potongan kode klien *private*

- Potongan kode klien *public* mencoba terhubung dengan *manager*

```

elif manager_address != None and not private_ip:
    tcp_client_sock.connect(manager_address)
    tcp_client_sock.send(b'Hi from client with public ip')

    data = tcp_client_sock.recv(1024)
    response_data = json.loads(data.decode('utf-8'))
    if response_data.get("msg") == "CRAWLER_START_URLS":
        start_urls = response_data.get("start_urls", [])[0].split()
        print('start_urls:', start_urls)

    tcp_sock.send(json.dumps({"msg": "CLIENT_CONNECTED_TO_THE_MANAGER"}).encode())
    break

```

Gambar 4.17: Potongan kode klien *public* mencoba terhubung dengan *manager*

Mencerdaskan dan
Memartabatkan Bangsa

- Potongan kode komunikasi antar klien serta mengirimkan url pertama untuk di-*crawling* kepada klien *private*

```
def handle_slave(self, slave_socket):
    data_buffer = b''
    max_chunk_size = 65536
    delimiter = b'\x00\x01\x02\x03\x04\x05'
    db_public_connection = self.db_public.connect()
    try:
        compressed_identifier = slave_socket.recv(max_chunk_size).split(delimiter, 1)[0]
        unique_identifier = zlib.decompress(compressed_identifier).decode()

        # Check if the client is a new slave or an existing one
        if unique_identifier in self.slave_connections:
            print(f"Client with identifier {unique_identifier} is an existing slave.")
        else:
            print(f"Client with identifier {unique_identifier} is a new slave.")
            slave = Client(unique_identifier, slave_socket)
            self.slave_connections[unique_identifier] = slave

        # Send an initial URL to the newly connected slave
        initial_url = self.pop_queue()
        self.visited_urls.add(initial_url)
        slave.queue.append(initial_url)
        print("init_url", initial_url)
        if initial_url is not None:
            # Serialize and compress the initial URL before sending
            initial_url_data = json.dumps(initial_url).encode()
            compressed_initial_url = zlib.compress(initial_url_data)
            slave_socket.send(compressed_initial_url + delimiter)
```

Gambar 4.18: Potongan kode komunikasi antar klien

Klien *public* akan mengatur klien *private* yang terhubung. Menyimpan *unique identifier*-nya, akan dipastikan bahwa tidak ada klien *private* dengan address yang sama terhubung. Dan tentunya akan menghapus klien *private* yang sudah tidak lagi terhubung lagi dengan klien *public*. Pencatatan jumlah url untuk setiap klien *private* juga dipertimbangkan.

Serta pengiriman data akan di-*encode* ke JSON dan di-*compress* agar data yang dikirimkan tidak begitu besar. Setiap data yang diterima oleh klien *public* pun juga akan di-*decompress*.

- Potongan kode klien *public* menerima data hasil *crawling* yang diberikan oleh klien *private* serta memasukkan informasi terkait ke dalam database

```

while True:
    chunk = slave_socket.recv(max_chunk_size)
    if not chunk:
        break

    data_buffer += chunk
    while delimiter in data_buffer:
        chunk, data_buffer = data_buffer.split(delimiter, 1)
        try:
            decompressed_data = zlib.decompress(chunk)
            complete_json = decompressed_data.decode()
            if str(complete_json) != str(unique_identifier):
                try:
                    json_data = json.loads(complete_json)
                    self.append_queue(json_data["outgoing_link"])
                    if "crawled_url" in json_data:
                        slave.remove_queue(json_data["crawled_url"])
                        average = round(mean(json_data["duration_crawled_url"]), 2)
                        with self.lock:
                            slave.average_crawling_time.append(average)
                            if len(slave.average_crawling_time) > 10:
                                slave.average_crawling_time.pop(0)
                    if "page_information_data" in json_data:
                        page_information_data = json_data["page_information_data"]
                        with self.lock:
                            page_id = self.db_utils.insert_page_information(
                                db_public_connection,
                                page_information_data["url"],
                                page_information_data["crawl_id"],
                                page_information_data["html5"],
                                page_information_data["title"],
                                page_information_data["description"],
                                page_information_data["keywords"],
                                page_information_data["content_text"],
                                page_information_data["hot_url"],
                                page_information_data["size_bytes"],
                                page_information_data["model_crawl"],
                                page_information_data["duration_crawl"]
                            )
                
```

Gambar 4.19: Potongan kode klien *public* menerima data hasil *crawling*

- Fungsi untuk pengecekan duplikasi url yang didapatkan dari hasil *crawling*

```

def append_queue(self, urls):
    with self.lock:
        for url in urls:
            if url not in self.visited_urls and url not in self.queue:
                self.queue.append(url)

```

Gambar 4.20: Fungsi untuk pengecekan duplikasi url yang didapatkan dari hasil *crawling*

- Potongan kode klien *public* mengirimkan url yang akan di-*crawling* ke klien *private* dengan mempertimbangkan kondisi dari klien tersebut berdasarkan kecepatan *crawling*. Klien yang lambat akan dikirim sedikit url untuk

mengurangi bebannya. Sedangkan, klien yang cepat akan mendapatkan jumlah tugas lebih banyak.

```

for unique_identifier, slave in active_slave_connections.items():
    if not self.is_socket_connected(slave.socket):
        with self.lock:
            del self.slave_connections[unique_identifier]
    else:
        print("len queue:", len(slave.queue))
        print("health:", slave.health)

        if slave.health == 100:
            if len(slave.queue) <= 100:
                # Get 30 URLs to send to the current slave
                urls_to_send = self.get_next_urls(30)

                if urls_to_send:
                    # Serialize the URLs and compress the data
                    data_to_send = json.dumps(urls_to_send).encode()
                    compressed_data = zlib.compress(data_to_send)

                    slave.socket.send(compressed_data + b'\x00\x01\x02\x03\x04\x05')

                    self.add_visited_urls(urls_to_send)
                    slave.append_queue(urls_to_send)
                    self.remove_queue(urls_to_send)

            else:
                if len(slave.queue) <= 70:
                    # Get 20 URLs to send to the current slave
                    urls_to_send = self.get_next_urls(20)

                    if urls_to_send:
                        # Serialize the URLs and compress the data
                        data_to_send = json.dumps(urls_to_send).encode()
                        compressed_data = zlib.compress(data_to_send)

                        slave.socket.send(compressed_data + b'\x00\x01\x02\x03\x04\x05')

                        self.add_visited_urls(urls_to_send)
                        slave.append_queue(urls_to_send)
                        self.remove_queue(urls_to_send)

```

Gambar 4.21: Potongan kode klien *public* mengirimkan url yang akan di-*crawling*

Setiap beberapa detik sekali klien *public* akan mengirimkan url yang akan di-*crawling* oleh klien *private*. Url yang dikirim sudah dipastikan tidak terduplikasi dan dilakukan *compress*. Penentuan pengiriman jumlah url juga ditentukan dari "kesehatan" dari klien *private*. Kalau kondisinya "sehat", maka akan mendapatkan 30 url setiap kiriman dan sebaliknya akan mendapatkan 20 url.

- Fungsi untuk pengecekan "kesehatan" dari setiap klien yang terhubung

```
def manage_slave_health(self):
    while True:
        try:
            for unique_identifier, slave in self.slave_connections.items():
                if len(slave.average_crawling_time) > 10:
                    if mean(slave.average_crawling_time) > 10:
                        slave.health = 50
                    else:
                        slave.health = 100
        except Exception as e:
            print(f"Error: {str(e)}")
        time.sleep(20)
```

Gambar 4.22: Fungsi untuk pengecekan "kesehatan" dari setiap klien yang terhubung

4.1.3.2 Klien *Private*

- Fungsi untuk klien *private* terhubung ke klien *public*

```
def connect_to_client_public(self, address):
    print('connecting to client public')
    self.slave_socket.connect(address)

    # Send the unique identifier
    unique_identifier = self.slave_socket
    compressed_identifier = zlib.compress(str(unique_identifier).encode())
    self.slave_socket.send(compressed_identifier + b'\x00\x01\x02\x03\x04\x05')
    return self.slave_socket
```

Gambar 4.23: Fungsi untuk klien *private* terhubung ke klien *public*

Ketika pertama kali klien *private* terhubung ke klien *public*. Maka akan mengirimkan *unique identifier* untuk membedakan client yang terhubung. Dan tidak lupa pula untuk di-compress.

- Fungsi untuk klien *private* menerima url yang akan di-*crawling* dari klien *public*

```

def receive_url_chunks():
    max_chunk_size = 65536
    delimiter = b'\x00\x01\x02\x03\x04\x05'
    data_buffer = b''

    while True:
        try:
            chunk = slave_socket.recv(max_chunk_size)
            if not chunk:
                break

            data_buffer += chunk

            while delimiter in data_buffer:
                chunk, data_buffer = data_buffer.split(delimiter, 1)
                try:
                    # Decompress the chunk before JSON decoding
                    decompressed_data = zlib.decompress(chunk)
                    urls = json.loads(decompressed_data.decode())
                    slave_data.append_url(urls)
                except zlib.error as e:
                    print(f"Error decompressing data: {e}")
                except json.JSONDecodeError as e:
                    print(f"Error decoding JSON: {e}")

                except Exception as e:
                    print(f"Error while receiving data: {e}")
                    break
        # print(len(slave_data.queue))
    
```

Gambar 4.24: Fungsi untuk klien *private* menerima url yang akan di-*crawling* dari klien *public*

*Mencerdaskan dan
Memartabatkan Bangsa*

- Potongan kode dari fungsi untuk klien *private* mengirim url hasil *crawling* ke klien *public*

```

if crawled_url is None:
    if page_information_data is None:
        serialize_queue = {
            "outgoing_link": outgoing_link,
        }
    else:
        serialize_queue = {
            "outgoing_link": outgoing_link,
            "page_information_data": page_information_data
        }
else:
    if page_information_data is None:
        serialize_queue = {
            "outgoing_link": outgoing_link,
            "crawled_url": crawled_url,
            "duration_crawled_url": duration_crawled_url
        }
    else:
        serialize_queue = {
            "outgoing_link": outgoing_link,
            "page_information_data": page_information_data,
            "crawled_url": crawled_url,
            "duration_crawled_url": duration_crawled_url
        }

# Serialize the JSON object
json_str = json.dumps(serialize_queue)

# Compress the serialized JSON data
compressed_data = zlib.compress(json_str.encode())

# Send the compressed data in smaller chunks
for i in range(0, len(compressed_data), max_chunk_size):
    chunk = compressed_data[i:i + max_chunk_size]
    if i + max_chunk_size >= len(compressed_data):
        # If it's the last chunk, add a newline delimiter
        self.slave_socket.send(chunk + delimiter)
    else:
        self.slave_socket.send(chunk)

```

Gambar 4.25: Potongan kode dari fungsi untuk klien *private* mengirim url hasil *crawling* ke klien *public*

- Potongan kode dari fungsi untuk klien *private* melakukan *scraping* halaman web

```
def scrape_page(self, url: str, slave_data) -> None:
    try:
        page_start_time = time.time()
        response = self.crawl_utils.get_page(url)
        if response and response.status_code == 200:
            db_connection = self.db.connect()
            # self.lock.acquire()
            with self.lock:
                now = datetime.now()
                print(url, "| BFS |", now.strftime("%d/%m/%Y %H:%M:%S"))
            # self.lock.release()
            soup = bs4.BeautifulSoup(response.text, "html.parser")
            title = soup.title.string
            article_html5 = soup.find("article")
            if article_html5 is None:
                # extract text content from html4
                html5 = []
                texts = soup.find("body").findAll(text=True)
                visible_texts = filter(self.tag_visible, texts)
                text = " ".join(t.strip() for t in visible_texts)
                text = text.lstrip().rstrip()
                text = text.split(",")
                clean_text = ""

```

Gambar 4.26: Potongan kode untuk klien *private* melakukan *scraping* halaman web

4.2 Pengujian

Pada penelitian ini akan terfokus untuk menguji proses komunikasi yang terjadi, konsistensi data, efisiensi sumber daya dan optimalisasi data . Dijalankan proses *crawling* selama satu jam. Pada *crawler* individual dan *crawler* terdistribusi. Pengujian dilakukan dengan *initial url*: "<https://www.detik.com/>".

Tabel 4.1: Hasil *crawling* selama satu jam

Jenis Crawler	Total Rows	Unique Rows
Crawler Individual	7069	7069
2 Crawler Terdistribusi	9175	9175

4.2.1 Proses Komunikasi

Berikut adalah proses komunikasi pada setiap perangkat:

- Proses komunikasi pada *tracker*

```
(env) root@vmi1165273:~/p2p/new_crawler# python tracker.py
Server listening on 0.0.0.0:55555
New connection from ('185.227.134.123', 35676)
Received from ('185.227.134.123', 35676): {'type': 'client', 'ip': '185.227.134.123', 'is_private': False}
New connection from ('103.166.156.127', 46120)
Received from ('103.166.156.127', 46120): {'type': 'manager', 'ip': '103.166.156.127', 'is_private': False}
clients: [{'address': ('185.227.134.123', 35676), 'type': 'client', 'is_private': False}, {'address': ('103.166.156.127', 46120), 'type': 'manager', 'is_private': False}]
manager ready? True
notify client
received data: {'msg': 'CLIENT_CONNECTED_TO_THE_MANAGER'}
clients: [{'address': ('185.227.134.123', 35676), 'type': 'client', 'is_private': False}, {'address': ('103.166.156.127', 46120), 'type': 'manager', 'is_private': False}]
New connection from ('180.244.164.42', 6403)
Received from ('180.244.164.42', 6403): {'type': 'client', 'ip': '192.168.1.4', 'is_private': True}
clients: [{'address': ('185.227.134.123', 35676), 'type': 'client', 'is_private': False}, {'address': ('180.244.164.42', 6403), 'type': 'client', 'is_private': True}, {'address': ('103.166.156.127', 46120), 'type': 'manager', 'is_private': False}]
```

Gambar 4.27: Proses komunikasi pada *tracker*

- Proses komunikasi pada *manager*

```
(env) [root@jft p2p]# python manager.py
My IP address: 103.166.156.127
Information sent to the server.
TCP Manager listening on 103.166.156.127:46120
public: ('185.227.134.123', 35676)
notify tracker
Received client list: [{'address': ['185.227.134.123', 35676], 'type': 'client', 'is_private': False}, {'address': ['103.166.156.127', 46120], 'type': 'manager', 'is_private': False}]
Clients: {<__main__.Client object at 0x7f68d64f68d0>, <__main__.Client object at 0x7f68d64f6860>}
Accepted connection from ('185.227.134.123', 37298)
Received data: Hi from client with public ip
```

Gambar 4.28: Proses komunikasi pada *manager*

- Proses komunikasi pada klien *public*

```
(env) root@vmi1165273:~/p2p/new_crawler/client# python client.py
My IP address: 185.227.134.123
Connected to the tracker
Private: False
manager address: ('103.166.156.127', 46120)
start_urls: ['https://www.detik.com/']
table page_information already exists
Client public listening for incoming connection..
0 Slave(s) Connected
0 Slave(s) Connected
0 Slave(s) Connected
0 Slave(s) Connected
Accepted connection from ('180.244.164.42', 7720)
Client with identifier <socket.socket fd=5, family=2, type=1, proto=0, laddr=('192.168.1.4', 50148), raddr=('185.227.134.123', 35676)> is a new slave.
init_url https://www.detik.com/
len queue: 1
health: 100
1 Slave(s) Connected
len queue: 1
health: 100
```

Gambar 4.29: Proses komunikasi pada klien *public*

- Proses komunikasi pada klien *private*

```
> python client.py
My IP address: 192.168.1.18
Connected to the tracker
Private: True
connecting to client public
table crawling already exists
table page_information already exists
table page_linking already exists
table page_images already exists
table page_tables already exists
table page_styles already exists
table page_scripts already exists
table page_list already exists
table page_forms already exists
table tfidf already exists
table tfidf_word already exists
table pagerank already exists
table pagerank_changes already exists
Starting the crawler from the start urls...
Running breadth first search crawler...
https://news.detik.com/berita/d-7136374/husen-s-20-tahun-bui | BFS | 12/01/2024 10:03:56
```

Gambar 4.30: Proses komunikasi pada klien *private*

4.2.2 Konsistensi Data

Berdasarkan Tabel 4.1, didapatkan hasil data url yang di-*crawling* itu unik dan tidak ada duplikasi. Berdasarkan hasil perhitungan sebagai berikut.

$$\text{Jumlah baris yang terduplicasi} = \text{Total Rows} - \text{Unique Rows} = 9175 - 9175 = 0$$

4.2.3 Efisiensi Sumber Daya dan Optimalisasi Data

Berdasarkan Tabel 4.1, didapatkan hasil peningkatan jumlah data dengan waktu yang relatif sama, yaitu satu jam. Pada *crawler* individual didapatkan jumlah baris data dalam tabel database sebanyak 7069 dan pada *crawler* terdistribusi didapatkan jumlah baris data dalam tabel database sebanyak 9175. Terdapat peningkatan jumlah data yang didapatkan oleh *crawler* terdistribusi. Penambahan total data yang terkumpul dapat dicapai beriringan dengan sumber daya yang digunakan secara efisien.

$$\text{Persentase peningkatan data} = \left(\frac{9175 - 7069}{7069} \right) \times 100 \approx 30.0\%$$

4.3 Analisis Hasil

Berdasarkan proses perbandingan *crawling* yang berjalan selama satu jam, analisis hasil pengujian adalah sebagai berikut:

1. Proses komunikasi antar perangkat berjalan dengan sistematis.
2. Data yang dihasilkan dari *crawler* terdistribusi lebih banyak daripada *crawler* individual. Didapatkan sekitar 30% lebih banyak.
3. Duplikasi data dapat terhindarkan saat mengelola lebih dari satu *crawler* dengan melakukan *crawling* berdasarkan url yang diterima dari pengelola (klien *public*) dan mengembalikan temuan url baru kepada pengelola.
4. Efisiensi sumber daya dan optimalisasi data dapat tercapai, hanya dengan melakukan *crawling* selama satu jam. Bisa lebih dari satu *crawler* yang berjalan di satu waktu yang bersamaan. Peningkatan tersebut juga dipengaruhi oleh latensi jaringan, *overhead* koordinasi, dan potensi *bottleneck*.
5. Proses penyatuan data menjadi *centralized* pada satu perangkat, yang mana merupakan penggabungan dari setiap *crawler* yang melakukan pekerjaan.

Hasil perbandingan *crawler* individual dengan *crawler* terdistribusi menunjukkan hasil yang baik. Peningkatan efisiensi dapat diraih dengan menerapkan metode terdistribusi. Ada hal yang harus dibayar dalam meningkatkan efisiensi, walaupun perbedaan yang terlihat hanya 30% tapi itu merupakan hal yang baik karena hasil akhirnya mengalami peningkatan.

*Mencerdaskan dan
Memartabatkan Bangsa*

BAB V

KESIMPULAN DAN SARAN

5.1 Kesimpulan

Berdasarkan hasil implementasi dan pengujian terhadap *crawler* terdistribusi. Maka diperoleh kesimpulan seperti berikut:

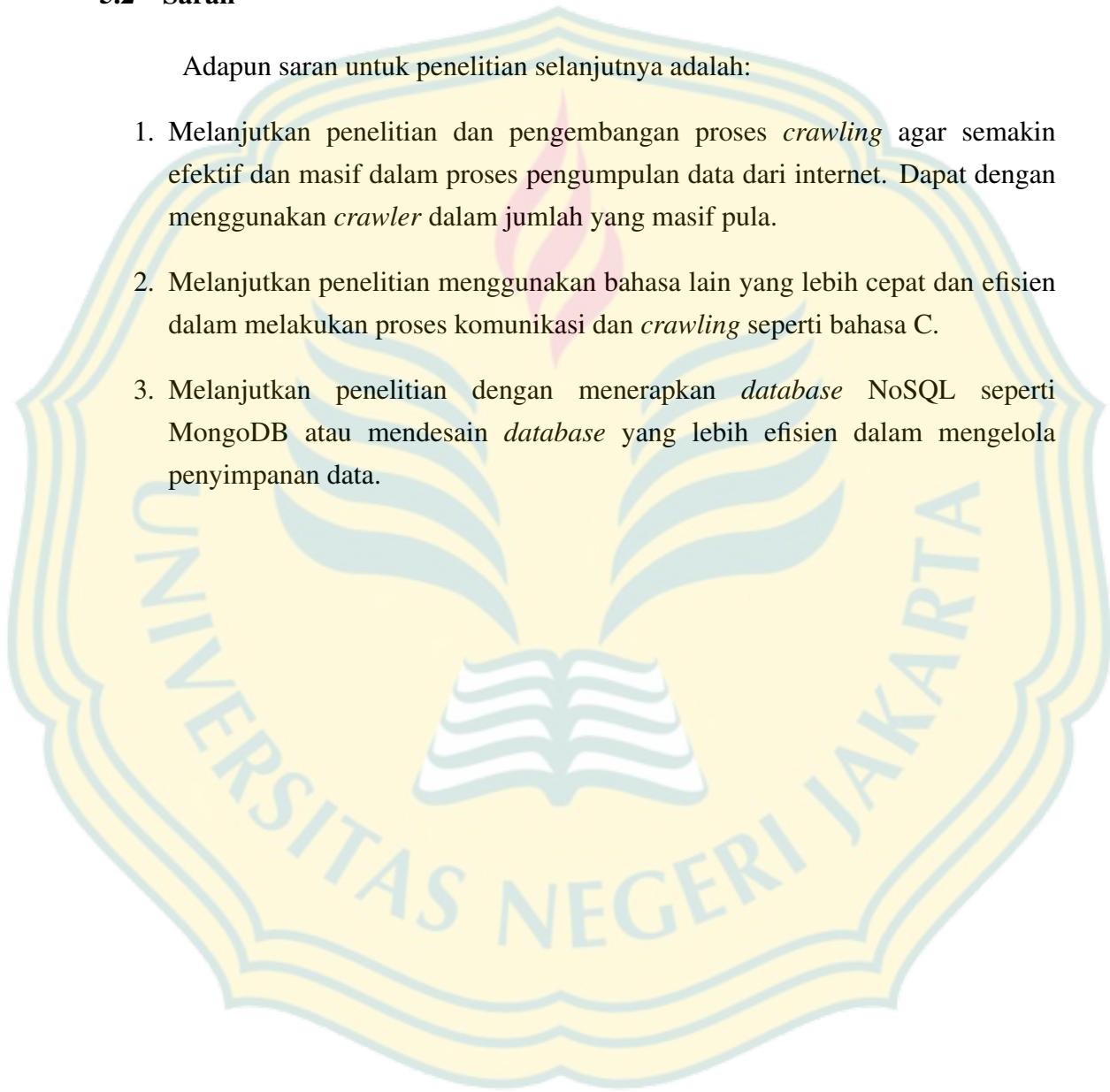
1. Pembuatan *crawler* terdistribusi dengan menggunakan mekanisme *socket programming* memerlukan pemahaman yang baik terkait jaringan komputer. Karena sering berfokus pada proses komunikasi antar perangkat.
2. Pembuatan *crawler* terdistribusi memerlukan setidaknya tiga perangkat dengan *public IP address* sebagai pondasi utama komunikasi dapat berjalan dengan sistematis.
3. Penyimpanan data pada *crawler* terdistribusi menggunakan sqlite3 sebagai *database*-nya. Dan datanya dapat ter-centralized pada pengelola *crawler* (klien *public*).
4. Pengembangan *crawler* terdistribusi menjadikan proses *crawling* yang dilakukan pada peta web dapat lebih cepat untuk mengumpulkan data dalam jumlah yang banyak.
5. Perbandingan *crawler* individual dengan *crawler* terdistribusi memiliki rasio pengumpulan data lebih besar sekitar 30% dengan menggunakan *crawler* terdistribusi.

*Mencerdaskan dan
Memartabatkan Bangsa*

5.2 Saran

Adapun saran untuk penelitian selanjutnya adalah:

1. Melanjutkan penelitian dan pengembangan proses *crawling* agar semakin efektif dan masif dalam proses pengumpulan data dari internet. Dapat dengan menggunakan *crawler* dalam jumlah yang masif pula.
2. Melanjutkan penelitian menggunakan bahasa lain yang lebih cepat dan efisien dalam melakukan proses komunikasi dan *crawling* seperti bahasa C.
3. Melanjutkan penelitian dengan menerapkan *database* NoSQL seperti MongoDB atau mendesain *database* yang lebih efisien dalam mengelola penyimpanan data.



*Mencerdaskan dan
Memartabatkan Bangsa*

DAFTAR PUSTAKA

- Asmara, A. (2024). "Perancangan User Interface Search Engine dan Admin Console Untuk Manajemen dan Visualisasi Data Hasil Pengindeksan". In.
- Babatunde, O. and O. Al-Debagy (2014). "A comparative review of internet protocol version 4 (ipv4) and internet protocol version 6 (ipv6)". In: arXiv preprint arXiv:1407.2717.
- Boldi, P. dkk., (2018). "BUBiNG: Massive crawling for the masses. ACM Transactions on the Web (TWEB)". In: pp. 1–26.
- Cambazoglu dkk., (2009). "Quantifying performance and quality gains in distributed web search engines". In: *International Journal of Information Technology & Decision Making*, pp. 411–418.
- Ford, B., P. Srisuresh, and D. Kegel (2005). "Peer-to-Peer Communication Across Network Address Translators". In: *USENIX Annual Technical Conference*, pp. 179–192.
- IBMCorporation (2021). *What is a socket?* URL: <https://www.ibm.com/docs/en/zos/2.1.0?topic=services-what-is-socket>.
- Khatulistiwa, L. (2023). "Perancangan Arsitektur Search Engine dengan Mengintegrasikan Web Crawler, Algoritma Page Ranking, dan Document Ranking". In.
- Orlando dkk., (2002). "Design of a parallel and distributed web search engine. In Parallel Computing: Advances and Current Issues". In: pp. 197–204.
- Pradana, F. H. (2023). "Perbandingan Implementasi Algoritma-Algoritma Pagerank pada Satu Mesin Komputer". In.
- Qiaoqiao, L. (2021). *What Is Network Address Translation (NAT)?* URL: <https://info.support.huawei.com/info-finder/encyclopedia/en/NAT.html>.
- Qoriiba, M. F. (2021). "Perancangan Crawler sebagai Pendukung pada Search Engine". In.
- Rosenberg, J. dkk., (2003). "STUN-simple traversal of user datagram protocol (UDP) through network address translators (NATs)". In: *rfc3489*.
- Shkpenyuk, V. and T. Suel (2002). "Design and implementation of a high-performance distributed web crawler". In: *Proceedings 18th International Conference on Data Engineering*, pp. 357–368.
- Tanenbaum, A. S., N. Feamster, and D. J. Wetherall (Sept. 2021). *Computer Network (Sixth Edition)*. Encyclopedia of Database Systems. Springer Verlag.

DAFTAR RIWAYAT HIDUP



MUHAMMAD RIDHO RIZQILLAH. Lahir di Jakarta, 24 Agustus 2001. Anak ketiga dari pasangan Bapak Sumarsono dan Ibu Siti Haeriyah. Saat ini penulis tinggal di Komplek Pertamina Tugu Jl. Permata III Blok G Nomor 11 RT 002/RW 016, Kelurahan Tugu Utara, Kecamatan Koja, Kota Jakarta Utara, Provinsi DKI Jakarta.

Riwayat Pendidikan: Penulis mengawali pendidikan di SD Negeri Tugu Utara 15 pada tahun 2007-2013. Setelah itu, penulis melanjutkan pendidikan ke SMP Negeri 30 Jakarta pada tahun 2013-2016. Kemudian melanjutkan pendidikan di SMA Negeri 52 Jakarta pada tahun 2016-2019. Pada tahun 2019, penulis melanjutkan studi ke Universitas Negeri Jakarta (UNJ) di program studi Ilmu Komputer.

Riwayat Organisasi: Selama di bangku perkuliahan, penulis pernah mengikuti organisasi Badan Eksekutif Mahasiswa selama dua periode yakni 2 semester sebagai Staff dan Kepala Departemen Advokasi. Lalu mengikuti organisasi pengabdian masyarakat Desa Binaan selama 2 semester sebagai Kepala Sub Divisi Kurikulum. Serta menjadi bagian dari Tim Pembela Mahasiswa di UNJ selama satu semester sebagai Staff Kajian dan Aksi Strategis. Tidak lupa pula, penulis juga mengikuti DEFAULT, yakni kelompok studi di program studi Ilmu Komputer yang bergerak di bidang pengembangan teknologi seperti pada bidang arsitektur, website, animasi, dan juga aplikasi. Dan menjabat selama 2 semester sebagai Ketua *Website Development Study Interest Group*.

*Mencerdaskan dan
Memartabatkan Bangsa*

METADATA

Judul : Improvisasi Crawling pada Peta Web Menggunakan Algoritma Terdistribusi dengan Model Koordinasi Berbasis Socket Programming

Nama : Muhammad Ridho Rizqillah

NIM : 1313619033

Pembimbing I : Muhammad Eka Suryana, M.Kom

Pembimbing II : Med Irzal, M.Kom

Keyword : 1. *search engine*
2. *distributed crawler*
3. *socket programming*
4. *crawling*



*Mencerdaskan dan
Memartabatkan Bangsa*