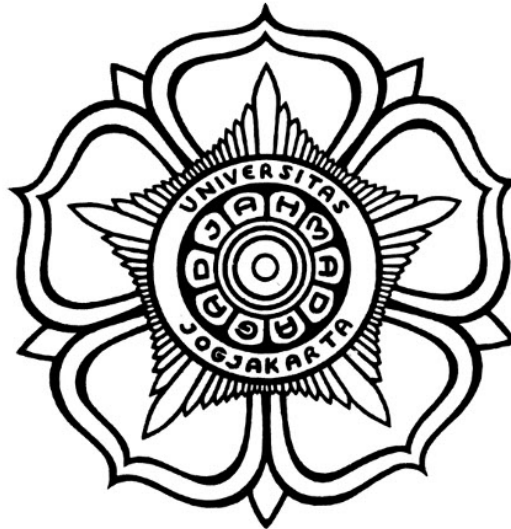


**ARSITEKTUR LAYANAN WEB YANG RESPONSIF DAN
AKUNTABEL BERBASIS *STATE MACHINE REPLICATION*
DENGAN JAMINAN *BYZANTINE FAULT TOLERANCE***

SKRIPSI



***THE SUSTAINABLE DEVELOPMENT GOALS
Industry, Innovation and Infrastructure
Peace, Justice, and Strong Institutions***

Disusun oleh:

**AHMAD ZAKI AKMAL
21/480179/TK/52981**

**PROGRAM SARJANA PROGRAM STUDI TEKNOLOGI
INFORMASI
DEPARTEMEN TEKNIK ELEKTRO DAN TEKNOLOGI INFORMASI
FAKULTAS TEKNIK UNIVERSITAS GADJAH MADA
YOGYAKARTA
2025**

HALAMAN PENGESAHAN

ARSITEKTUR LAYANAN WEB YANG RESPONSIF DAN AKUNTABEL BERBASIS *STATE MACHINE REPLICATION* DENGAN JAMINAN *BYZANTINE FAULT TOLERANCE*

SKRIPSI

Diajukan Sebagai Salah Satu Syarat untuk Memperoleh
Gelar Sarjana Teknik
pada Departemen Teknik Elektro dan Teknologi Informasi
Fakultas Teknik
Universitas Gadjah Mada

Disusun oleh:

AHMAD ZAKI AKMAL
21/480179/TK/52981

Telah disetujui dan disahkan

Pada tanggal

Dosen Pembimbing I

Dosen Pembimbing II

Dr. Ir. Guntur Dharma Putra, S.T., M.Sc.
NIP 111199104201802102

Azkario Rizky Pratama, ST, M.Eng., Ph.D.
NIP 199102182024061001

PERNYATAAN BEBAS PLAGIASI

Saya yang bertanda tangan di bawah ini :

Nama : Ahmad Zaki Akmal
NIM : 21/480179/TK/52981
Tahun terdaftar : 2021
Program : Sarjana
Program Studi : Teknologi Informasi
Fakultas : Teknik Universitas Gadjah Mada

Menyatakan bahwa dalam dokumen ilmiah Skripsi ini tidak terdapat bagian dari karya ilmiah lain yang telah diajukan untuk memperoleh gelar akademik di suatu lembaga Pendidikan Tinggi, dan juga tidak terdapat karya atau pendapat yang pernah ditulis atau diterbitkan oleh orang/lembaga lain, kecuali yang secara tertulis disitasi dalam dokumen ini dan disebutkan sumbernya secara lengkap dalam daftar pustaka.

Dengan demikian saya menyatakan bahwa dokumen ilmiah ini bebas dari unsur-unsur plagiasi dan apabila dokumen ilmiah Skripsi ini di kemudian hari terbukti merupakan plagiasi dari hasil karya penulis lain dan/atau dengan sengaja mengajukan karya atau pendapat yang merupakan hasil karya penulis lain, maka penulis bersedia menerima sanksi akademik dan/atau sanksi hukum yang berlaku.

Yogyakarta, 24-06-2025




Ahmad Zaki Akmal
21/480179/TK/52981

HALAMAN PERSEMBAHAN

Tugas akhir ini kupersembahkan kepada Allah SWT. yang telah senantiasa membimbing dan memberi kemudahan pada setiap langkah hamba-Nya. Kupersembahkan pula kepada kedua orang tuaku, keluarga besar, dan teman-teman, serta untuk bangsa, negara, dan agamaku.

* * *

"So be patient with the decree of your Lord, and do not be like the companion of the fish who cried out in sorrow.

Had it not been for a mercy from his Lord that reached him, he would surely have been cast upon the barren shore, while he was blameworthy.

But then his Lord chose him, and made him among the righteous."

Quran Surah Al-Qalam (68:48-50)

KATA PENGANTAR

Puji syukur ke hadirat Allah SWT atas limpahan rahmat, karunia, serta petunjuk-Nya sehingga tugas akhir berupa penyusunan skripsi ini telah terselesaikan dengan baik. Dalam hal penyusunan tugas akhir ini penulis telah banyak mendapatkan arahan, bantuan, serta dukungan dari berbagai pihak. Oleh karena itu pada kesempatan ini penulis mengucapkan terima kasih kepada:

1. Bapak Prof. Ir. Hanung Adi Nugroho, S.T., M.E., Ph.D., IPM. selaku Ketua Departemen Teknik Elektro dan Teknologi Informasi Fakultas Teknik Universitas Gadjah Mada.
2. Bapak Ir. Lesnanto Multa Putranto, S.T., M.Eng., Ph.D., IPM. selaku Sekretaris Departemen Teknik Elektro dan Teknologi Informasi Fakultas Teknik Universitas Gadjah Mada.
3. Bapak Dr. Ir. Guntur Dharma Putra, S.T., M.Sc. selaku Dosen Pembimbing Skripsi I yang selalu memberikan arahan, masukan dan saran, serta dukungan luar biasa selama proses tahapan penyusunan tugas akhir ini.
4. Bapak Azkario Rizky Pratama, S.T., M.Eng., Ph.D. selaku Dosen Pembimbing Skripsi II yang telah memberikan masukan serta dukungan dalam penyusunan tugas akhir ini.
5. Bapak Prof. Ir. Selo, S.T., M.T., M.Sc, Ph.D selaku Dosen Pembimbing Akademik yang telah memberikan bantuan bagi penulis dalam kebutuhan akademik perkuliahan.
6. Ibu penulis, Ariyani Setyoningrum, atas kasih sayang yang tak terhingga, yang senantiasa menjadi cahaya di setiap langkah kehidupan penulis.
7. Ibu penulis, Ariyani Setyoningrum, atas doa yang tak pernah putus, yang menjadi kekuatan saat penulis lemah dan lelah.
8. Ibu penulis, Ariyani Setyoningrum, atas pengorbanan dan kesabaran yang tak terhitung, yang menjadi fondasi dari setiap pencapaian penulis.
9. Ayah penulis, Syahriza Luthfi, yang telah bekerja tanpa lelah, menjadi panutan, dan memberi dukungan penuh sepanjang hidup penulis.
10. Kakak penulis, Ahmad Alif Naufal, dan adik penulis, Ahmad Nazhif Zulfiqar, yang dengan caranya masing-masing telah menghadirkan dukungan, keceriaan, dan ketenangan dalam hari-hari penulis selama ini.
11. Prof. Dr. Ir. Sri Suning Kusumawardani, S.T., M.T. beserta keluarga, yang telah banyak memberikan nasihat dan ilmu selama penulis berkuliah di Departemen Teknik Elektro dan Teknologi Informasi Fakultas Teknik Universitas Gadjah Mada.
12. Aufa, Arif, Aria, Ariq, Akhdan, Azfar, Brop, Budi, Difta, Ditya, Diamond, Evan, Giga, Nando, Nathan, Rizqi, Rayhan, dan banyak teman lainnya yang tidak bisa dituliskan semuanya sebagai *support system* dalam menjalani kehidupan perkuliahan di Departemen Teknik Elektro dan Teknologi Informasi Universitas Gadjah Mada.
13. Teman-teman KKN Lembaran Bayan 2024 yang telah mewarnai kehidupan penulis

dan menjadi teman bagi penulis untuk berkeluh kesah.

14. Seluruh pihak yang tidak dapat disebutkan satu per satu yang telah membantu, baik secara langsung maupun tidak langsung dalam proses penulisan skripsi ini.

Penulis menyadari bahwa penelitian ini masih jauh dari sempurna, untuk itu semua jenis saran, kritik, dan masukan yang bersifat membangun sangat diharapkan. Akhir kata penulis berharap semoga skripsi ini dapat memberikan manfaat bagi kita semua, amin.

DAFTAR ISI

HALAMAN PENGESAHAN	ii
PERNYATAAN BEBAS PLAGIASI	iii
HALAMAN PERSEMBAHAN	iv
KATA PENGANTAR	vi
DAFTAR ISI	vii
DAFTAR TABEL	x
DAFTAR GAMBAR	xii
DAFTAR SINGKATAN.....	xiii
INTISARI.....	xiv
ABSTRACT	xv
BAB 1 Pendahuluan	1
1.1 Latar Belakang	1
1.2 Rumusan Masalah	4
1.3 Tujuan Penelitian	5
1.4 Batasan Penelitian	7
1.5 Manfaat Penelitian	8
1.6 Sistematika Penulisan.....	9
BAB 2 Tinjauan Pustaka dan Dasar Teori	10
2.1 Tinjauan Pustaka	10
2.2 Dasar Teori	15
2.2.1 Sistem Terdistribusi	15
2.2.2 <i>Fault, Error, dan Failure</i>	17
2.2.3 Jenis <i>Fault</i> dalam Sistem Terdistribusi	18
2.2.3.1 <i>Crash Faults</i>	18
2.2.3.2 <i>Byzantine Faults</i>	18
2.2.4 <i>Byzantine Fault Tolerance</i>	19
2.2.5 Arsitektur <i>Web Services</i>	20
2.2.5.1 Arsitektur Monolitik.....	20
2.2.5.2 Arsitektur <i>Client-Server</i>	22
2.2.5.3 Arsitektur <i>Microservices</i>	23
2.2.6 <i>Blockchain</i> dan Algoritma Konsensus	24
2.2.6.1 Teknologi <i>Blockchain</i>	24
2.2.6.2 Algoritma dan Mekanisme Konsensus <i>Blockchain</i>	26
2.2.7 Solusi <i>Scalability</i> Blockchain	31
2.2.7.1 <i>Blockchain Sharding</i>	31
2.2.7.2 <i>Rollups</i>	33

2.2.7.3	<i>Sidechains</i>	35
2.2.8	Analisis Perbandingan Metode	36
2.2.8.1	Perbandingan Platform <i>Blockchain</i> dan Konsensus	36
2.2.8.2	Perbandingan Solusi <i>Scalability</i> Sistem	38
2.2.8.3	Perbandingan Bahasa Pemrograman <i>Web Service</i>	40
BAB 3	Metode Penelitian.....	42
3.1	Alat dan Bahan Tugas akhir	42
3.1.1	Alat Tugas akhir.....	42
3.1.2	Bahan Tugas akhir	43
3.2	Metode Penelitian.....	43
3.2.1	Metode Pengembangan	44
3.2.2	Metode Pengujian dan Evaluasi	45
3.3	Alur Tugas Akhir	45
3.3.1	Fase Persiapan.....	46
3.3.1.1	Studi Literatur.....	46
3.3.1.2	Penentuan Metode Penelitian	47
3.3.1.3	Perancangan Arsitektur Sistem	47
3.3.2	Fase Pengembangan	51
3.3.2.1	Pengembangan <i>Web Service</i>	51
3.3.2.2	Integrasi dengan <i>Database</i>	52
3.3.2.3	Implementasi Konsensus dan ABCI	53
3.3.2.4	Pengembangan <i>Service Registry</i> dan <i>Service Handler</i> ...	54
3.3.2.5	Konfigurasi Docker Compose	55
3.3.3	Fase Pengujian	55
3.3.3.1	Pengujian Fungsionalitas API.....	55
3.3.3.2	Pengukuran Metrik Evaluasi	55
BAB 4	Hasil dan Pembahasan.....	57
4.1	Implementasi dan <i>Proof-of-Concept</i> Sistem	57
4.1.1	Konfigurasi dan Pengaturan Sistem	57
4.1.2	Infrastruktur CometBFT	60
4.1.3	Implementasi Logika Konsensus Melalui ABCI	63
4.1.4	Skenario <i>Proof-of-Concept</i>	66
4.1.5	<i>Deployment</i> dan Kontainerisasi	67
4.2	Evaluasi dan Analisis Performa	68
4.3	Perbandingan dengan Pendekatan <i>State-of-the-Art</i>	75
4.4	Keterbatasan Sistem	76
BAB 5	Kesimpulan dan Saran.....	77
5.1	Kesimpulan.....	77
5.2	Saran.....	77

DAFTAR PUSTAKA.....	79
---------------------	----

DAFTAR TABEL

Tabel 2.1	Perbandingan Sistem BFT Terkait dengan Penelitian Ini.....	12
Tabel 2.2	Perbandingan platform <i>blockchain</i> dan konsensus.....	38
Tabel 2.3	Perbandingan solusi <i>scalability</i> sistem.	39
Tabel 2.4	Perbandingan bahasa pemrograman <i>web service</i>	41

DAFTAR KODE

4.1	Operasi database dengan <i>repository pattern</i>	57
4.2	Implementasi <i>endpoint route</i> , <i>service handler</i> , dan <i>service registry</i>	58
4.3	Inisialisasi <i>service registry</i>	59
4.4	Inisialisasi <i>web server</i>	60
4.5	Inisialisasi aplikasi <i>blockchain</i> CometBFT.	62
4.6	Inisialisasi <i>node</i> CometBFT.	62
4.7	Implementasi <i>method</i> CheckTx yang meng- <i>override interface</i> CometBFT.	63
4.8	Implementasi <i>method</i> ProcessProposal yang meng- <i>override interface</i> CometBFT.	64
4.9	Implementasi <i>method</i> FinalizeBlock yang meng- <i>override interface</i> CometBFT.	65
4.10	Implementasi benchmark workflow untuk pengukuran latensi.	69
4.11	Contoh data latensi yang disimpan dalam format CSV	70
L.1	Kode lengkap <i>method</i> ProcessProposal.	L-83
L.2	Kode lengkap <i>method</i> FinalizeBlock.	L-84

DAFTAR GAMBAR

Gambar 2.1	Hubungan kausal antara <i>Fault</i> , <i>Error</i> , dan <i>Failure</i>	18
Gambar 2.2	Arsitektur monolitik	20
Gambar 2.3	Arsitektur <i>client-server</i>	22
Gambar 2.4	Arsitektur <i>microservices</i> yang tersusun atas 4 <i>service</i>	23
Gambar 2.5	Struktur data <i>blockchain</i>	25
Gambar 2.6	Protokol tiga fase PBFT	29
Gambar 2.7	Ilustrasi <i>blockchain sharding</i>	32
Gambar 2.8	Tipe-tipe <i>sharding</i>	33
Gambar 2.9	Interaksi antara <i>Layer 1</i> dan <i>Layer 2</i> dalam arsitektur <i>Rollup</i>	34
Gambar 3.10	Alur tugas akhir yang dibagi menjadi tiga tahapan utama, Persiapan, Pengembangan, dan Pengujian.	46
Gambar 3.11	Arsitektur dua <i>layer</i> untuk memisahkan operasi interaktif dan operasi dengan keamanan BFT.	47
Gambar 3.12	Diagram <i>sequence</i> yang menunjukkan alur sistem, mulai dari interaksi responsif dengan L2 hingga komitmen ke L1.	51
Gambar 3.13	<i>Entity-relation Diagram</i> yang didesain untuk skenario <i>supply chain shipping</i>	52
Gambar 4.14	Interaksi aplikasi-konsensus dan pemanggilan <i>Method ABCI</i>	61
Gambar 4.15	Implementasi L1 dalam kontainer Docker dengan 4 <i>node</i>	67
Gambar 4.16	Implementasi L2 dalam kontainer Docker dengan 1 <i>node</i>	68
Gambar 4.17	Proses pengumpulan data menggunakan <i>script Go</i>	70
Gambar 4.18	Perbandingan performa antara <i>Layer 1</i> dengan <i>Layer 2</i> pada konfigurasi yang berbeda.	71
Gambar 4.19	Perbandingan performa tiap <i>endpoint Layer 1</i> dengan <i>Layer 2</i> pada konfigurasi yang berbeda.	72
Gambar 4.20	Distribusi latensi tiap <i>endpoint</i> untuk dua konfigurasi <i>Layer 2</i> berbeda.....	73
Gambar 4.21	Perbandingan latensi antara arsitektur 1 <i>layer</i> (DeWS) dengan arsitektur 2 <i>layer</i>	75
Gambar L.1	<i>QR Code Repository</i> GitHub	L-83

DAFTAR SINGKATAN

BFT	=	<i>Byzantine Fault-Tolerant</i> (adjektiva); <i>Byzantine Fault Tolerance</i> (nomina)
CFT	=	<i>Crash Fault-Tolerant</i> (adjektiva); <i>Crash Fault Tolerance</i> (nomina)
SMR	=	<i>State Machine Replication</i>
ABCI	=	<i>Application-BlockChain Interface</i>
ZKP	=	<i>Zero-Proof Knowledge</i>
L1	=	Layer 1
L2	=	Layer 2
PoW	=	<i>Proof of Work</i>
PoS	=	<i>Proof of Stake</i>
DPoS	=	<i>Delegated Proof of Stake</i>

INTISARI

Layanan web dengan *Byzantine fault tolerance* (BFT) menjamin konsistensi dan ketahanan terhadap manipulasi, tetapi umumnya mengalami latensi tinggi akibat proses konsensus antar *node*. Semakin banyak *node*, semakin lambat respons sistem, sehingga BFT sulit diterapkan pada aplikasi *real-time*. Untuk mengatasi hal tersebut, penelitian ini merancang arsitektur layanan web dua *layer* berbasis *State Machine Replication* (SMR) yang menjembatani *trade-off* antara ketanggapan dan jaminan BFT. *Layer 2* (L2) menyediakan *buffer* transaksi yang responsif melalui simulasi konsensus, sedangkan *Layer 1* (L1) melakukan komitmen akhir dengan konsensus BFT terhadap *state* yang direplikasi.

Untuk mengevaluasi arsitektur yang diusulkan, penelitian ini menggunakan alur kerja *supply chain* sebagai skenario *proof-of-concept*. Skenario *supply chain* membutuhkan kombinasi respons yang cepat dan integritas data yang tinggi, seperti pada proses pemindaian, validasi, inspeksi, dan pengiriman. Arsitektur dua *layer* yang dirancang memungkinkan langkah-langkah tersebut dijalankan secara interaktif di L2, lalu difinalisasi ke L1 melalui konsensus BFT untuk menjamin keutuhan dan auditabilitas. Dengan sesi sebagai satuan logis, sistem menjaga urutan proses, sinkronisasi *state*, dan verifikasi hasil tanpa mengganggu latensi di setiap langkah.

Sebuah prototipe dikembangkan menggunakan CometBFT untuk mengimplementasikan SMR, dengan rancangan API yang terstruktur, pelacakan sesi, serta mekanisme replikasi transaksi deterministik untuk menjaga konsistensi status dan auditabilitas. Pengujian pada berbagai konfigurasi menunjukkan bahwa operasi pada L2 memiliki latensi hingga empat kali lebih rendah dibandingkan dengan proses konsensus di L1, sambil tetap mempertahankan waktu respons di bawah satu detik bahkan dalam klaster BFT dengan 16 node. Arsitektur ini relevan untuk sistem seperti *supply chain management*, platform kolaboratif berbasis web, dan aplikasi pemerintahan digital (*e-government*) yang membutuhkan interaksi instan serta jaminan keabsahan melalui konsensus terdistribusi yang akuntabel.

Kata Kunci: layanan web, *Byzantine fault tolerance*, sesi, interaktivitas, ketanggapan, arsitektur berlapis, *state machine replication*

ABSTRACT

Byzantine fault-tolerant (BFT) web services ensure consistency and resilience against manipulation, but typically suffer from high latency due to the consensus process among distributed nodes. As the number of nodes increases, system responsiveness decreases, limiting BFT applicability in real-time applications. To address this, this study proposes a two-layer web service architecture based on State Machine Replication (SMR) to balance the trade-off between responsiveness and BFT guarantees. Layer 2 (L2) acts as a responsive transaction buffer through consensus simulation, while Layer 1 (L1) finalizes transaction batches using BFT consensus over replicated state.

To evaluate the proposed architecture, a supply chain workflow is used as a proof-of-concept scenario. This domain demands both fast response and strong data integrity across processes such as scanning, validation, inspection, and delivery. The dual-layer architecture enables these steps to run interactively in L2 and be finalized in L1 for auditability and data consistency. Session-based grouping preserves logical ordering, state synchronization, and verifiable execution without the latency of consensus after every step.

A prototype was developed using CometBFT to implement the SMR engine, supported by structured API design, session tracking, and deterministic transaction replay for state consistency and auditability. Benchmarking across multiple node configurations demonstrates that L2 operations achieve up to 4 times lower latency compared to L1 consensus, maintaining sub-second responsiveness even under 16-node BFT clusters. This architecture is suitable for supply chain systems, collaborative web platforms, and e-government applications where immediate interaction must coexist with verifiable distributed agreement.

Keywords : *web services, Byzantine fault tolerance, sessioning, interactivity, responsiveness, layered architecture, state machine replication*

BAB 1

PENDAHULUAN

1.1 Latar Belakang

Interaksi dunia digital kini sedang berkembang menuju *metaverse*, yang mengubah pengalaman virtual di internet dari bentuk yang terisolasi menjadi ruang 3D yang saling terhubung, memungkinkan pengguna untuk bersosialisasi, bekerja, dan melakukan transaksi jual-beli [1, 2]. Visi dari *metaverse* ini sejalan dengan prinsip-prinsip Web3, di mana teknologi *blockchain* memungkinkan kepemilikan yang terdesentralisasi, aset digital yang dapat ditransfer, dan transaksi *trustless* [3]. Berbeda dengan platform web tradisional, di mana terdapat badan sentral yang mengontrol data pengguna dan kepemilikan aset digital, *metaverse* dan Web3 memungkinkan pengguna untuk memegang kepemilikan aset digitalnya sendiri. Dengan berkembangnya ekonomi digital tersebut, pengguna dapat melakukan penukaran aset digital dengan harga dan nilai di dunia nyata [4], *web services* sebagai teknologi yang mendasarinya juga perlu berkembang dari arsitekturnya yang masih tersentralisasi. *Open metaverse* adalah *metaverse* yang terdesentralisasi dan dapat diakses siapa saja, namun karena sifatnya yang terbuka, pihak yang tidak bertanggung jawab juga dapat bergabung dan merusak sistem. Oleh karena itu, *open metaverse* sangat diuntungkan oleh properti *Byzantine fault tolerance* [5, 6] yang mampu memastikan validitas konsensus dan transaksi tanpa bergantung pada kepercayaan suatu pihak perantara [5]. Kemampuan tersebut sangat penting dalam melakukan tukar-menukar aset digital dalam sistem tanpa adanya otoritas sentral.

Strategi untuk meningkatkan keandalan sistem terdistribusi telah berkembang untuk dapat menangani skenario *fault* yang semakin kompleks. Sistem terdistribusi, yang terdiri dari banyak *node* yang terhubung melalui jaringan dan bekerja bersama sebagai satu kesatuan, menghadapi tantangan unik dalam hal *fault tolerance* karena komponennya dapat mengalami kegagalan secara independen dan parsial. Berdasarkan keparahan dampak dan kompleksitasnya, *fault* dalam sistem terdistribusi dibagi menjadi dua kategori utama: *crash fault* dan *Byzantine fault* [7].

Crash fault terjadi ketika sebuah *node* dalam sistem terdistribusi berhenti berfungsi sepenuhnya. *Node* tersebut mengalami *crash* dan menghentikan semua operasi. *Node* yang mengalami *crash* tidak lagi merespons permintaan, tidak mengirim pesan ke *node* lain, dan tidak memproses pesan masuk. Akan tetapi, sebelum mengalami *crash*, *node* berperilaku dengan benar sesuai protokol yang ditentukan pada sistem. Untuk menangani *crash fault* yang relatif sederhana, berbagai strategi seperti replikasi, *check-pointing*, *retrying*, *monitoring*, hingga algoritma *machine learning* telah

dikembangkan dalam konteks sistem terdistribusi [8].

Di sisi lain, *Byzantine fault*, yang dinamai berdasarkan makalah berjudul "*The Byzantine Generals Problems*" karya Lamport *et al.*, merepresentasikan tipe kegagalan yang jauh lebih kompleks dan parah dampaknya dalam sistem terdistribusi. *Byzantine fault* terjadi ketika *node* berperilaku tidak terprediksi atau bahkan berbahaya [9]. *Node* yang mengalami *Byzantine fault* dapat mengirim informasi yang kontradiktif ke berbagai bagian sistem, dengan kemungkinan sengaja berusaha untuk merusak sistem. *Byzantine node* juga bisa menampakkannya dirinya sebagai *node* yang berfungsi secara normal walaupun sebenarnya melakukan sesuatu yang tidak benar. Menangani jenis kegagalan ini dalam sistem terdistribusi merupakan tantangan yang jauh lebih besar karena komponen yang rusak dapat menunjukkan perilaku yang beragam.

Sistem *Crash Fault-tolerant* (CFT) dalam lingkungan terdistribusi dirancang untuk terus berfungsi dengan benar meskipun beberapa komponennya mengalami kegagalan karena *crash*. Sistem CFT dapat menjaga konsistensi ketika *node* hanya mengalami kegagalan *crash* sederhana [8]. Di sisi lain, sistem *Byzantine Fault-tolerant* (BFT) harus dapat beroperasi dengan benar dalam lingkungan terdistribusi bahkan ketika sepertiga dari *node* dalam sistem bertindak jahat atau tidak dapat diprediksi [9, 10].

Tantangan ini menjadi semakin signifikan dalam sistem terdistribusi berskala besar seperti jaringan *blockchain* dan aplikasi *metaverse*, di mana jumlah *node* yang terlibat bisa sangat banyak dan mencakup batas-batas kepercayaan lintas entitas ataupun organisasi. Kemampuan untuk menolerir *Byzantine fault* ini sangat penting untuk aplikasi seperti *open metaverse* dan Web3, di mana aset digital merepresentasikan nilai ekonomi yang nyata sehingga menjadikannya target menarik untuk serangan siber yang bertujuan menyebabkan *Byzantine fault*. Sebagai sistem terdistribusi yang beroperasi di antara berbagai entitas tanpa adanya kepercayaan satu sama lain, *metaverse* memerlukan mekanisme *fault tolerance* yang dapat menjamin integritas dan konsistensi meskipun menghadapi potensi perilaku *malicious* dari beberapa partisipan.

Dalam literatur, *web service* dengan *Byzantine fault tolerance* telah dikembangkan secara ekstensif selama dua dekade terakhir. Pendekatan awal seperti Thema [11] dan BFT-WS [12] mengimplementasikan properti *Byzantine fault tolerance* sambil mempertahankan kompatibilitas dengan protokol web standar seperti SOAP dan WSDL. Akan tetapi, pendekatan ini masih mengandalkan komponen *middleware* terpusat yang dapat menjadi *bottleneck* atau titik kelemahan sistem. Meskipun pendekatan ini memberikan ketahanan terhadap *Byzantine fault*, mereka tidak sepenuhnya memenuhi visi sistem yang benar-benar terdistribusi dan terdesentralisasi.

Desain yang lebih baru seperti WebBFT [13] mengembangkan landasan ini dengan membuat layanan BFT dapat diakses oleh klien berbasis browser melalui

layanan BFT-DNS. WebBFT bergeser dari model interaksi *request-compute-response* konvensional menuju model *publish-subscribe*, mendukung fitur kolaboratif *real-time* dan kompatibel dengan teknologi web modern seperti *WebSocket*. Pendekatan ini memberikan perkembangan yang signifikan dalam mengintegrasikan *Byzantine fault tolerance* dengan aplikasi web. Namun, WebBFT masih menghadapi beberapa tantangan teknis yang signifikan. Sistem ini memiliki *overhead* performa yang cukup tinggi dari kriptografi di sisi browser, penanganan JSON, dan rentan terhadap serangan *denial-of-service* yang disebabkan oleh klien berbahaya yang memicu perubahan pemimpin konsensus berulang-ulang. Masalah ini membatasi skalabilitas dan ketahanan sistem dalam lingkungan yang berisiko.

Kesamaan utama di antara penerapan-penerapan awal ini adalah bahwa mereka memperoleh *Byzantine fault tolerance* dengan mencapai konsensus pada hasil komputasi dari satu *node* server web atau melalui penggunaan komponen sentral, tidak dengan melakukan replikasi komputasi yang lengkap di seluruh *node*. Pendekatan ini menyebabkan keterbatasan dalam kemampuan sistem untuk menangani kasus *Byzantine fault* yang lebih kompleks.

Kemajuan signifikan dicapai dengan dikembangkannya DeWS (*Decentralized and Byzantine Fault-Tolerant Web Services*) [14]. Berbeda dengan pendekatan sebelumnya, Ramachandran *et al.* berpendapat bahwa mereplikasi komputasi dengan lengkap di setiap *node* diperlukan untuk memastikan keamanan terhadap adanya *Byzantine fault*. Prinsip ini menjadi fondasi desain mereka untuk DeWS, yang menciptakan model interaksi *web service* yang benar-benar baru. DeWS mengimplementasikan model *request-compute-consensus-log-response*, berbeda dari model konvensional *request-compute-response* yang digunakan oleh *web service* tradisional. Dalam model DeWS, semua operasi menjalani replikasi dan validasi konsensus sebelum respons dikembalikan ke klien. Pendekatan ini memastikan bahwa setiap respons disetujui oleh kuorum *node* dalam jaringan, memberikan jaminan integritas dan auditabilitas yang kuat untuk setiap transaksi.

Namun, pendekatan DeWS menghadapi tantangan signifikan dalam hal kinerja dan pengalaman pengguna. Proses konsensus BFT yang diimplementasikan oleh DeWS mengakibatkan latensi yang signifikan antara tindakan pengguna dan respons sistem. Pengujian menunjukkan bahwa untuk konfigurasi dengan 15 *node*, DeWS membutuhkan waktu sekitar 935ms untuk memproses sebuah *request*, bahkan dalam lingkungan kontainer Docker yang ideal, dibandingkan dengan penerapan di dunia nyata [14]. Latensi yang mencapai hampir satu detik menciptakan hambatan substansial untuk aplikasi yang membutuhkan interaktivitas tinggi, seperti lingkungan *metaverse* di mana pengguna mengharapkan *feedback* yang hampir instan terhadap aksi mereka. Dalam konteks aplikasi interaktif, *delay* bahkan hanya beberapa ratus milidetik dapat

secara signifikan menurunkan pengalaman pengguna dan membuat aplikasi terasa tidak responsif.

DeWS menghadapi *trade-off* mendasar antara dua aspek penting dalam pengembangan sistem terdistribusi modern: *Byzantine fault tolerance* dan ketanggapan sistem. Di satu sisi, sistem memerlukan jaminan keamanan yang kuat, khususnya dalam konteks aplikasi yang bersifat kritis seperti menangani aset digital yang bernilai. Di sisi lain, pengguna mengharapkan aplikasi yang responsif dan memberikan *feedback* instan. Tantangan ini sangat terasa dalam konteks aplikasi seperti *open metaverse*, *blockchain*, dan sistem manajemen *supply chain*. Dalam *metaverse*, pengguna melakukan interaksi *multi-step* yang kompleks, seperti membuat atau melakukan aksi jual-beli aset digital, dengan harapan adanya *feedback* yang instan. Tanpa ketanggapan sistem yang memadai, pengalaman pengguna menjadi terganggu. Namun di sisi lain, transaksi digital juga melibatkan aset bernilai yang memerlukan jaminan *Byzantine fault tolerance*.

Dalam sistem manajemen *supply chain*, operator membutuhkan *feedback* cepat saat memindai dan memproses paket, tetapi sistem juga memerlukan catatan yang *immutable* untuk menjaga integritas proses secara keseluruhan. Saat volume proses sedang tinggi, operator tidak dapat menunggu lebih dari satu detik setiap kali mereka menyelesaikan langkah dalam alur kerja *multi-step* mereka. Kompromi antara *Byzantine fault tolerance* dan ketanggapan sistem ini merupakan tantangan fundamental yang belum diselesaikan dengan baik oleh solusi yang ada. Sistem seperti DeWS memprioritaskan keamanan dengan mengorbankan ketanggapan, sementara aplikasi web tradisional memprioritaskan ketanggapan tetapi kurang dalam hal *fault tolerance* dan integritas yang kuat.

Tantangan ini semakin diperburuk oleh fakta bahwa pendekatan desain tradisional untuk *Byzantine fault tolerance* yang secara alami memerlukan langkah-langkah konsensus terdistribusi sehingga menambah latensi. Untuk mencapai konsensus di antara banyak *node*, sistem perlu menunggu respons dari kuorum *node*, yang mengakibatkan *delay*, terutama dalam jaringan dengan *node* yang tersebar secara geografis. Oleh karena itu, diperlukan pendekatan baru yang dapat menyelesaikan *trade-off* antara *Byzantine fault tolerance* dan ketanggapan sistem. Dalam konteks ini, sistem yang responsif berarti mampu memberikan umpan balik kepada pengguna dengan latensi rendah, sedangkan sistem yang interaktif memungkinkan pengguna untuk melakukan aksi secara berkelanjutan dalam banyak langkah tanpa hambatan yang mengganggu alur kerja.

1.2 Rumusan Masalah

Berdasarkan latar belakang yang telah diuraikan, penelitian ini berfokus pada permasalahan mendasar dalam pengembangan *web service* dengan properti BFT untuk

aplikasi yang responsif dan interaktif. Rumusan masalah dalam penelitian ini adalah sebagai berikut:

1. Sistem terkini untuk *web services* dengan konsensus BFT secara umum masih memiliki keterbatasan latensi yang signifikan, seperti pada DeWS yang mencatat latensi hampir 1 detik pada konfigurasi 15 *node*. Penundaan ini mencegah sistem konsensus BFT untuk dapat diterapkan pada domain yang memerlukan ketangguhan tinggi.
2. Seiring dengan peningkatan skala sistem (*scale up*), latensi sistem cenderung meningkat secara signifikan akibat adanya *overhead* koordinasi antar-*node*. Masalah ini mengindikasikan keterbatasan skalabilitas pada arsitektur sistem, yang menjadi hambatan serius dalam penerapan BFT pada aplikasi berskala besar dan dinamis. Ketika latensi bertambah seiring bertambahnya jumlah *node*, performa sistem menjadi tidak stabil dan tidak dapat memenuhi kebutuhan aplikasi *real-time* yang menuntut kecepatan dan throughput tinggi, seperti dalam konteks supply chain, sistem pembayaran digital, atau kolaborasi multi-pihak dalam metaverse.
3. Banyak arsitektur BFT yang ada tidak menjamin bahwa setiap *node* melakukan komputasi secara independen dan deterministik sebelum mencapai konsensus. Sistem yang hanya melakukan konsensus atas hasil akhir dari satu server atau bergantung pada *proof* berbasis kriptografi seperti *Zero-Knowledge Proof* (ZKP) memiliki kelemahan dalam hal transparansi. Selain itu, pendekatan tersebut berisiko menimbulkan *single point of failure* dalam proses komputasi, sehingga mengurangi jaminan keamanan yang seharusnya diberikan oleh sistem BFT.

1.3 Tujuan Penelitian

Berdasarkan rumusan masalah yang telah diidentifikasi, tujuan dari penelitian ini adalah sebagai berikut:

1. Mengurangi latensi sistem BFT melalui pendekatan arsitektural yang memungkinkan interaksi responsif tanpa mengurangi keamanan dan integritas sistem, sehingga memungkinkan penerapannya pada aplikasi yang memerlukan waktu respons cepat seperti pada *supply chain* atau *metaverse*.
2. Meningkatkan skalabilitas sistem dengan mengusulkan mekanisme yang dapat membatasi dampak peningkatan jumlah *node* terhadap latensi, melalui pendekatan seperti *session management*, *transaction buffering*, pemisahan tanggung jawab, dan validasi *local-first* sebelum konsensus global, sehingga sistem tetap praktis untuk aplikasi *real-time* pada skala besar.
3. Merancang dan mengimplementasikan arsitektur *web service* berbasis SMR yang

menjamin setiap *node* melakukan komputasi secara independen dan deterministik sebelum proses konsensus, sehingga memberikan transparansi, *verifiability*, dan jaminan *Byzantine fault tolerance* yang kuat tanpa bergantung pada *single point of computation* atau *proof* kriptografis eksternal seperti ZKP.

1.4 Batasan Penelitian

Untuk menjaga fokus dan kelayakan penelitian ini, berikut adalah batasan-batasan yang diterapkan:

1. Objek penelitian: Studi ini berfokus pada arsitektur *web service* dengan *Byzantine fault tolerance* untuk aplikasi yang memerlukan ketangguhan tinggi.
2. Metode penelitian: Pengembangan *proof-of-concept* dan evaluasi performa melalui simulasi dan pengujian terhadap skenario *supply chain* sebagai representasi *multi-step workflow*.
3. Waktu dan tempat penelitian: Penelitian dilakukan dari Januari hingga Mei 2025 di Yogyakarta.
4. Lingkup evaluasi: Penelitian ini terutama mengevaluasi aspek latensi dan ketangguhan sistem. Penelitian ini berfokus pada aspek keamanan tingkat konsensus untuk menangani *Byzantine fault*, sedangkan masalah keamanan di luar perilaku *Byzantine* seperti autentikasi dan integritas transaksi berada di luar lingkup penelitian. Selain itu, penelitian ini berfokus pada deteksi perilaku *Byzantine* dalam sistem konsensus, tetapi tidak mengembangkan strategi respons spesifik terhadap *Byzantine fault* yang terdeteksi, karena tindakan yang tepat dapat bervariasi tergantung pada domain aplikasi dan kebijakan keamanan yang diterapkan.
5. Skalabilitas sistem: Pengujian dibatasi pada konfigurasi dengan 4, 7, 10, 13, dan 16 *node* yang masing-masing mewakili toleransi terhadap 1, 2, 3, 4, dan 5 *Byzantine fault* berdasarkan formula $3f + 1$ untuk evaluasi sistem BFT.
6. Implementasi konsensus: Penelitian menggunakan algoritma konsensus berbasis Tendermint BFT yang diterapkan dalam CometBFT dan tidak membandingkan dengan algoritma konsensus alternatif.
7. Lingkungan pengujian: Pengujian dilakukan dalam jaringan *container* Docker yang dikontrol, bukan dalam penyebaran jaringan geografis yang sebenarnya.
8. Aplikasi *use case*: Implementasi *proof-of-concept* dibatasi pada skenario *supply chain* yang disederhanakan dan tidak mencakup semua kompleksitas yang mungkin ada dalam aplikasi interaktif yang sebenarnya.

Batasan-batasan ini membantu penelitian tetap terfokus pada tujuan utama yaitu mengatasi *trade-off* antara ketangguhan dan *Byzantine fault tolerance* dalam arsitektur *web services*.

1.5 Manfaat Penelitian

Penelitian ini diharapkan memberikan manfaat sebagai berikut:

1. Manfaat Akademis

- (a) Memberikan kontribusi teoretis dalam pengembangan arsitektur sistem terdistribusi yang mengatasi *trade-off* antara *Byzantine fault tolerance* dan ketanggapan sistem.
- (b) Memperluas pemahaman tentang pendekatan arsitektur untuk mengatasi keterbatasan latensi dalam sistem konsensus terdistribusi tanpa mengorbankan jaminan keamanan BFT.
- (c) Menjadi dasar bagi penelitian lanjutan dalam sistem terdistribusi yang menuntut keamanan dan ketanggapan tinggi

2. Manfaat Praktis

- (a) Memungkinkan pengembangan aplikasi interaktif yang dapat memberikan pengalaman pengguna responsif tanpa mengorbankan jaminan keamanan *Byzantine fault tolerance*.
- (b) Menyediakan solusi arsitektur yang dapat diterapkan pada domain aplikasi yang memiliki kebutuhan serupa terhadap keamanan dan ketanggapan tinggi.
- (c) Mengurangi hambatan adopsi teknologi konsensus BFT berbasis SMR dalam aplikasi yang memerlukan interaktivitas tinggi.

3. Manfaat Industri

- (a) Memberikan kerangka kerja arsitektur yang dapat membantu industri mengembangkan aplikasi terdistribusi dengan pengalaman pengguna yang lebih baik tanpa mengorbankan keamanan sistem.
- (b) Menciptakan dasar untuk pengembangan standar arsitektur *web service* yang mempertimbangkan kebutuhan interaktivitas dan keamanan secara bersamaan.
- (c) Mendorong inovasi dalam bidang aplikasi terdistribusi dengan memperkenalkan pendekatan baru yang mengoptimalkan keseimbangan antara keamanan dan kinerja sistem.

Secara keseluruhan, penelitian ini memiliki potensi untuk memperluas penerapan teknologi konsensus dan *Byzantine fault tolerance* ke dalam domain yang sebelumnya dianggap tidak praktis karena kendala latensi, sehingga membuka kemungkinan baru untuk aplikasi terdistribusi yang aman dan responsif.

1.6 Sistematika Penulisan

Tugas akhir ini dituliskan dalam lima bab dengan sistematika tiap babnya sebagai berikut:

1. Bab 1 membahas pendahuluan yang mencakup latar belakang, perumusan masalah, batasan masalah, serta tujuan dan manfaat penelitian yang diharapkan dapat menghasilkan rancangan arsitektur *web service* BFT yang menjaga keamanan tanpa mengorbankan ketanggapan sistem.
2. Bab 2 berisi tinjauan literatur untuk memberi konteks dan memposisikan penelitian ini di antara penelitian-penelitian terkait, serta menjelaskan dasar-dasar teori yang mendasari penelitian.
3. Bab 3 menjelaskan metode penelitian yang digunakan, termasuk alat dan bahan penelitian, serta alur kerja dan rincian setiap tahapannya.
4. Bab 4 memaparkan hasil perancangan arsitektur, implementasi, serta evaluasi dari sistem yang dikembangkan.
5. Bab 5 menyimpulkan hasil-hasil penelitian terkait arsitektur *web service* BFT dan memberikan saran untuk pengembangan atau penelitian lebih lanjut.

BAB 2

TINJAUAN PUSTAKA DAN DASAR TEORI

2.1 Tinjauan Pustaka

Pengembangan arsitektur *web service* dengan *Byzantine fault tolerance* (BFT) merupakan bidang penelitian yang telah berkembang selama dua dekade terakhir. Untuk memposisikan kontribusi penelitian ini, bagian ini mengulas beberapa karya penelitian sebelumnya yang berkaitan dengan BFT *web services*, arsitektur untuk *web services* dan sistem terdistribusi, serta metode-metode untuk mengatasi tantangan latensi dalam sistem yang menggunakan konsensus.

Salah satu tonggak awal dalam pengembangan sistem BFT adalah protokol PBFT (*Practical Byzantine Fault Tolerance*) yang diperkenalkan oleh Castro dan Liskov [10]. PBFT menawarkan pendekatan deterministik untuk mencapai konsensus di lingkungan yang memungkinkan perilaku *Byzantine*, dan menjadi acuan utama bagi berbagai penelitian lanjutan di bidang BFT. Meskipun protokol ini memberikan jaminan finalitas yang kuat, kompleksitas komunikasi kuadratik dan pola komunikasi *all-to-all* menjadikannya kurang efisien untuk sistem berskala besar atau interaktif. Keterbatasan ini mendorong lahirnya berbagai protokol dan sistem BFT baru seperti Zyzzyva, Thema, dan DeWS yang mencoba mengurangi latensi dan meningkatkan performa tanpa mengorbankan keamanan.

Salah satu penelitian awal mengenai *web service* dengan BFT dimulai dengan karya Merideth *et al.* yang mengembangkan sistem Thema [11]. Thema merupakan salah satu upaya pertama untuk mengintegrasikan BFT dengan *web service* sambil mempertahankan kompatibilitas dengan protokol SOAP dan WSDL. Namun, Thema masih bergantung pada komponen *middleware* terpusat yang dapat menjadi titik kegagalan tunggal (*single point of failure*). Pendekatan ini tidak sepenuhnya terdistribusi karena replikasi tidak dilakukan pada semua komputasi, melainkan hanya pada hasil akhir komputasi dari salah satu *web service*. Hal ini membatasi kemampuan sistem untuk mendeteksi dan menoleransi perilaku yang menyimpang pada tahap komputasi.

Zhao [12] mengembangkan BFT-WS yang meningkatkan desain Thema dengan arsitektur yang lebih fleksibel untuk *web service*. BFT-WS menawarkan kerangka kerja berbasis SOAP yang mendukung BFT tanpa mengubah standar *web service* yang ada. Sistem ini menggunakan voting dari replika untuk memverifikasi respons, tetapi, sama halnya dengan Thema, masih mengandalkan komponen *middleware* sentral. Keterbatasan ini menyebabkan sistem rentan terhadap *bottleneck* dan menjadi titik kegagalan tunggal, yang mengurangi keandalan sistem secara keseluruhan.

Penelitian oleh Pallemulle *et al.* [15] memperkenalkan Perpetual, sebuah protokol BFT untuk replikasi layanan web dalam arsitektur *multi-tier* dan *service-oriented*. Berbeda dari pendekatan terdahulu seperti Thema dan BFT-WS, Perpetual mendukung interaksi antara layanan yang direplikasi dengan derajat replikasi yang berbeda, serta menyediakan isolasi kesalahan yang ketat untuk menjamin *safety* dan *liveness*. Sistem ini memungkinkan eksekusi *thread* jangka panjang dan pemrosesan asinkron, menjadikannya lebih sesuai untuk arsitektur modern berbasis orkestrasi dan SOA. Dengan dukungan terhadap Apache Axis2, Perpetual-WS memperluas fungsionalitas ini ke dalam lingkungan *web service* nyata, serta menunjukkan skalabilitas yang baik. Pendekatan yang modular dan penggunaan *message authentication code* (MAC) menjadikan sistem ini ringan secara kriptografis dan dapat disesuaikan untuk berbagai protokol komunikasi [15].

Salah satu kemajuan penting dalam protokol BFT adalah Zyzyva, yang diperkenalkan oleh Kotla *et al.* [16]. Zyzyva menggunakan pendekatan *speculative* untuk mengurangi *overhead* komunikasi dalam SMR yang toleran terhadap *Byzantine fault*. Alih-alih menjalankan protokol *three-phase commit* seperti pada PBFT, Zyzyva memungkinkan replika untuk langsung mengeksekusi permintaan klien setelah menerima urutan yang diusulkan oleh *primary*, dan segera mengirimkan respons ke klien. Klien kemudian bertanggung jawab untuk membandingkan tanggapan dari replika. Jika klien menerima $3f + 1$ tanggapan identik, maka ia langsung menganggap permintaan selesai. Namun, jika hanya menerima antara $2f + 1$ hingga $3f$ tanggapan identik, klien harus mengirimkan *commit certificate* kembali ke replika untuk mengamankan urutan permintaan tersebut [16].

Dengan memisahkan deteksi ketidaksesuaian dan memindahkan tanggung jawab stabilitas hasil ke sisi klien, Zyzyva mengoptimalkan latensi dalam kasus normal (tanpa adanya kesalahan) hingga hanya satu ronde komunikasi. Strategi ini memungkinkan *throughput* tinggi lebih dari 86.000 permintaan per detik. Namun demikian, protokol ini dapat kembali ke mode dua fase saat terjadi kegagalan atau perbedaan tanggapan antar replika [16]. Zyzyva menandai pergeseran penting dari pendekatan pesimistis (seperti PBFT) ke optimistik, menjadikannya basis yang kuat untuk pengembangan sistem BFT interaktif dan berlatensi rendah seperti yang kemudian diadaptasi oleh Matsumoto *et al.* dalam sistem *peer-to-peer* hibrida.

Penelitian oleh Matsumoto dan Kobayashi [17] mengadaptasi protokol BFT spekulatif Zyzyva untuk sistem *peer-to-peer* (P2P) melalui pendekatan hibrida yang memperkenalkan *monitor node*. Arsitektur ini meningkatkan efisiensi dan deteksi kesalahan dalam lingkungan P2P yang terdistribusi dengan tetap mengandalkan pendekatan spekulatif khas Zyzyva. Protokol yang mereka usulkan terdiri dari dua fase, *speculative execution* dan *commit phase*, yang memungkinkan sistem mengidentifikasi

dan menangani *node Byzantine* secara efisien. Prinsip ini diterapkan dalam konteks permainan daring, namun juga relevan bagi pengembangan sistem BFT interaktif yang mengutamakan latensi rendah.

Berger dan Reiser [13] memperkenalkan WebBFT, sebuah sistem yang memperluas fungsionalitas BFT ke aplikasi web berbasis browser melalui layanan BFT-DNS untuk penemuan replika. Sistem ini bergeser dari model interaksi *request-compute-response* menuju model *publish-subscribe*, mendukung fitur kolaboratif *real-time* dan kompatibel dengan teknologi web modern seperti *WebSocket*. Namun, WebBFT menghadapi tantangan performa yang signifikan karena *overhead* pemrosesan kriptografi di sisi browser dan penanganan JSON, serta rentan terhadap serangan *denial-of-service* yang disebabkan oleh *malicious client* yang memicu perubahan pemimpin konsensus berulang-ulang.

Kemajuan signifikan dalam *web service* BFT dibuat oleh Ramachandran *et al.* pada 2023 dengan pengembangan DeWS (*Decentralized and Byzantine Fault-Tolerant Web Services*) [14]. Berbeda dengan pendekatan sebelumnya, DeWS memperkenalkan model interaksi *request-compute-consensus-log-response* yang memastikan bahwa semua transaksi divalidasi oleh jaringan *node* terdistribusi sepenuhnya melalui replikasi. DeWS menggunakan dasar *State Machine Replication* yang mengharuskan replikasi komputasi secara lengkap di setiap *node*, memberikan jaminan keamanan yang lebih kuat dalam menghadapi *Byzantine fault*. Sistem ini juga mendukung arsitektur *multi-stakeholder* dengan memberikan setiap *stakeholder* domain terpisah, memungkinkan kolaborasi antar organisasi yang berbeda dalam lingkungan BFT. Namun, pendekatan ini menimbulkan tantangan latensi sebesar 935ms pada sistem dengan 15 *node* dalam jaringan *container* Docker yang ideal [14]. Peningkatan latensi tersebut menciptakan sebuah *trade-off* antara jaminan keamanan BFT dan kebutuhan akan ketangguhan sistem yang tinggi untuk aplikasi interaktif.

Tabel 2.1. Perbandingan Sistem BFT Terkait dengan Penelitian Ini

Sistem / Algoritma	Kontribusi Utama	Kelebihan	Kekurangan
PBFT, 2002 (Castro <i>et al.</i> [10])	Protokol BFT deterministik tiga fase.	Finalitas kuat, ketahanan terhadap hingga f node <i>Byzantine</i> .	Komunikasi <i>all-to-all</i> , latensi tinggi untuk sistem besar.

Lanjutan Tabel 2.1

Sistem / Algoritma	Kontribusi Utama	Kelebihan	Kekurangan
Thema, 2005 (Merideth <i>et al.</i> [11])	Integrasi awal BFT dengan <i>web service</i> berbasis SOAP.	Kompatibel dengan WSDL dan SOAP.	Replikasi terbatas pada hasil akhir, tidak sepenuhnya terdistribusi.
BFT-WS, 2007 (Zhao [12])	Framework SOAP dengan voting hasil dari replika.	Tidak memodifikasi standar <i>web service</i> .	Masih bergantung pada <i>middleware</i> terpusat, <i>bottleneck</i> .
Perpetual-WS, 2008 (Pallemulle <i>et al.</i> [15])	BFT untuk arsitektur multi-tier dan SOA.	Mendukung <i>long-running thread</i> dan pemrosesan asinkron.	Kompleksitas integrasi dan dependensi pada Axis2.
Zyzyva, 2008 (Kotla <i>et al.</i> [16])	Protokol BFT spekulatif dengan <i>client-driven commit</i> .	Latensi rendah dalam kondisi normal, <i>throughput</i> tinggi.	Bergantung pada klien, kompleksitas pemulihan tinggi saat gagal.
Zyzyva-P2P, 2010 (Matsumoto <i>et al.</i> [17])	Adaptasi Zyzyva untuk sistem P2P hibrida.	Menyediakan deteksi kegagalan melalui <i>monitor node</i> .	Fokus spesifik pada permainan daring P2P, tidak umum untuk aplikasi lainnya.
WebBFT, 2018 (Berger & Reiser [13])	Aplikasi BFT pada browser dan WebSocket.	Kompatibel dengan interaksi <i>real-time</i> , mendukung <i>publish-subscribe</i> .	Kinerja kriptografi berat di sisi klien, rentan terhadap serangan <i>DoS</i> .
DeWS, 2023 (Ramachandran <i>et al.</i> [14])	Arsitektur terdistribusi penuh untuk BFT <i>web service</i> .	Replikasi penuh SMR, mendukung <i>multi-stakeholder</i> .	Latensi tinggi (935ms) untuk jaringan besar, tidak cocok untuk sistem interaktif murni.

Lanjutan Tabel 2.1

Sistem / Algoritma	Kontribusi Utama	Kelebihan	Kekurangan
Sistem Usulan, 2025 (Penelitian ini)	Arsitektur BFT untuk <i>web service</i> interaktif dengan pemisahan jalur interaksi cepat dan konsensus terdistribusi.	Responsivitas tinggi untuk pengguna, tanpa mengorbankan validitas konsensus BFT.	Kompleksitas tambahan dalam sinkronisasi sesi dan penanganan eksekusi ganda.

Tabel 2.1 merangkum perbandingan antara berbagai penelitian dan sistem terdahulu yang telah dijelaskan sebelumnya dalam subbagian ini. Setiap sistem dinilai berdasarkan kontribusi utamanya terhadap penerapan *Byzantine fault tolerance* (BFT), serta kelebihan dan kekurangannya dalam konteks implementasi untuk *web service* atau sistem interaktif. Tabel tersebut juga mencantumkan solusi yang diajukan dalam penelitian ini, yang berfokus pada pemisahan jalur umpan balik interaktif dan validasi konsensus untuk mengatasi *trade-off* antara responsivitas dan integritas BFT.

Penelitian tentang teknologi *Layer 2* dalam *blockchain* juga relevan dengan arsitektur sistem BFT. Mandal *et al.* [18] dan Thibault *et al.* [19] mengeksplorasi solusi skalabilitas *Layer 2* seperti *rollups* dan *sidechains* yang memungkinkan *throughput* lebih tinggi dan latensi lebih rendah sambil mempertahankan keamanan dengan cara melakukan komitmen *state* ke *blockchain* pada *Layer 1* secara berkala. Teknik-teknik tersebut menunjukkan bagaimana arsitektur *layered* dalam *blockchain* dapat memisahkan performa dan keamanan, prinsip yang juga diterapkan dalam penelitian ini.

Manajemen sesi (*session management*) dan pengelompokan sekumpulan operasi terkait menjadi unit logis, telah dieksplorasi dalam sistem terdistribusi terutama untuk autentikasi pengguna, *batching*, dan pelacakan *state* [20]. Namun, potensinya untuk mengoptimalkan operasi konsensus dalam *web service* BFT sebagian besar belum dieksplorasi. Penelitian ini mengisi kesenjangan tersebut dengan memanfaatkan konsep sesi untuk membantu sinkronisasi *state* sistem yang memungkinkan pemisahan antara operasi interaktif dan konsensus BFT.

Meskipun karya-karya sebelumnya telah mengembangkan penerapan BFT dalam *web service*, mereka secara kolektif belum berhasil mengatasi *trade-off* mendasar antara ketanggapan yang interaktif dan jaminan BFT yang kuat. Tidak adanya solusi yang mempertahankan baik latensi rendah untuk interaktivitas maupun jaminan integritas yang kuat dari BFT merupakan kesenjangan signifikan dalam literatur. Penelitian

ini secara langsung mengatasi kesenjangan tersebut dengan memperkenalkan arsitektur yang memisahkan tanggung jawab antara *feedback* yang cepat dan validasi konsensus, memungkinkan pengalaman pengguna yang responsif tanpa mengorbankan jaminan integritas dari konsensus *Byzantine fault tolerance*.

2.2 Dasar Teori

2.2.1 Sistem Terdistribusi

Sistem terdistribusi adalah kumpulan komponen komputasi independen yang berkomunikasi melalui jaringan dan bekerja sama untuk mencapai tujuan bersama [21]. Komponen-komponen ini beroperasi secara *concurrent* dan berkoordinasi hanya melalui pertukaran pesan tanpa adanya memori atau *clock* global yang digunakan bersama. Karakteristik sistem terdistribusi sangat penting karena menentukan bagaimana sistem merespons kegagalan, pertumbuhan skala, dan dinamika lingkungan operasional.

Menurut Thampi [21], sistem terdistribusi yang dirancang dengan baik sebaiknya memiliki karakteristik berikut:

- ***Fault-tolerant***: Kemampuan sistem untuk pulih dari kegagalan pada sebagian komponen tanpa mengganggu sistem secara keseluruhan. Sistem yang toleran terhadap *fault* mampu menghindari kondisi *single point of failure*, dan tetap memberikan layanan meskipun terjadi *crash*, kehilangan *message*, atau perilaku anomali. Dalam praktiknya, toleransi ini sering dicapai melalui replikasi data dan redundansi komputasi, serta deteksi dan isolasi *fault*.
- ***Scalability***: Kemampuan sistem untuk menangani pertumbuhan jumlah pengguna, volume data, atau jumlah *node* dalam sistem tanpa penurunan performa yang signifikan. Sistem terdistribusi yang *scalable* dapat beroperasi secara efisien baik dalam konfigurasi kecil maupun besar. Tantangan umum dalam skalabilitas mencakup *load balancing*, pembatasan *bandwidth*, dan *overhead* koordinasi antar *node*.
- ***Predictable Performance***: Kemampuan untuk memberikan respons dalam waktu yang dapat diprediksi dan konsisten terhadap beban kerja yang berubah-ubah. Hal ini penting dalam aplikasi interaktif atau *real-time*. Performa yang tidak dapat diprediksi dapat mengganggu pengalaman pengguna atau menimbulkan konsekuensi pada sistem yang krusial seperti pada sistem kendali industri atau layanan keuangan.
- ***Openness***: Sistem terdistribusi yang terbuka memungkinkan integrasi dengan komponen eksternal melalui protokol dan antarmuka standar.

Openness mendukung interoperabilitas dan pengembangan sistem yang modular. Misalnya, RESTful API dan gRPC sering digunakan dalam konteks sistem *microservices* yang terbuka.

- **Security:** Keamanan dalam sistem terdistribusi mencakup autentikasi, otorisasi, kerahasiaan data, integritas pesan, dan ketersediaan layanan. Karena data dan proses tersebar di banyak *node*, sistem harus tahan terhadap berbagai ancaman seperti *man-in-the-middle*, *replay attack*, atau *Byzantine faults*, terutama dalam lingkungan terbuka seperti Internet.
- **Transparency:** Pengguna sistem tidak perlu mengetahui kompleksitas teknis di balik distribusi komponen. Transparansi mencakup akses (akses ke sumber daya tanpa mengetahui lokasi fisiknya), lokasi, replikasi, kegagalan, dan performa sistem. Transparansi juga menjadi fondasi desain untuk memberikan ilusi sistem yang konsisten dan koheren, meskipun dijalankan di banyak tempat berbeda.

Selain karakteristik-karakteristik utama dari sistem terdistribusi, terdapat beberapa tantangan penting yang harus dihadapi. Salah satunya adalah koordinasi tanpa jam global, di mana ketiadaan referensi waktu yang akurat menyulitkan sinkronisasi dan pengurutan kejadian. Untuk mengatasi masalah ini, Lamport memperkenalkan konsep logical clock sebagai mekanisme bantu dalam pengurutan peristiwa di sistem yang tidak sinkron [22]. Tantangan lainnya adalah menjaga konsistensi data, khususnya dalam sistem yang mengandalkan replikasi. Ketika terjadi kegagalan jaringan atau *partition*, menjaga agar data di antara berbagai *node* tetap seragam menjadi hal yang sulit. Hal ini berhubungan erat dengan teorema CAP, yang menyatakan bahwa sebuah sistem hanya dapat menjamin dua dari tiga properti secara bersamaan, yaitu konsistensi (*consistency*), ketersediaan (*availability*), dan toleransi terhadap partisi jaringan (*partition tolerance*) [23]. Oleh karena itu, desain arsitektur sistem harus memperhatikan prioritas di antara ketiga aspek tersebut sesuai kebutuhan aplikasinya. Selain itu, Peter Deutsch merumuskan delapan asumsi keliru dalam merancang sistem terdistribusi, yang dikenal sebagai *Fallacies of Distributed Computing*. Asumsi-asumsi ini termasuk keyakinan bahwa latensi jaringan adalah nol, bandwidth yang tidak terbatas, jaringan selalu aman, dan transportasi data tanpa biaya [21]. Kesalahan dalam mengantisipasi asumsi-asumsi tersebut sering kali menyebabkan masalah serius dalam implementasi dan performa sistem. Memahami tantangan-tantangan tersebut sangat penting dalam merancang arsitektur web service BFT, yang menuntut replikasi deterministik, interaktivitas tinggi, dan jaminan integritas data dalam lingkungan yang rentan terhadap kegagalan.

2.2.2 *Fault, Error, dan Failure*

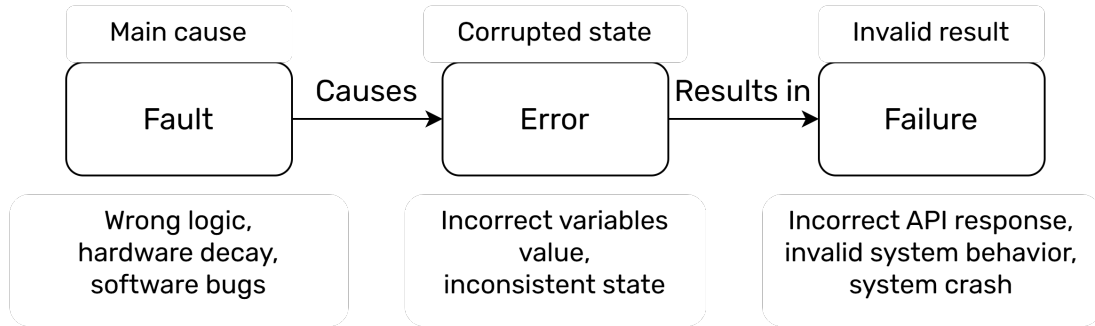
Dalam konteks keandalan sistem terdistribusi, penting untuk membedakan antara tiga konsep utama yang membentuk rantai kejadian dalam kegagalan sebuah sistem: *fault*, *error*, dan *failure* [24].

Fault adalah kondisi abnormal pada sistem yang berpotensi mengakibatkan *error*. *Fault* merupakan penyebab awal dari masalah sistem yang dapat muncul dalam berbagai bentuk. Cacat pada perangkat keras seperti degradasi komponen sering menjadi sumber *fault*. Selain itu, *bug* pada perangkat lunak berupa kesalahan logika program juga dapat menyebabkan sistem berperilaku tidak sesuai dengan harapan. Kesalahan operasional seperti konfigurasi yang tidak tepat dan kesalahan desain berupa asumsi yang keliru tentang lingkungan operasional sistem juga merupakan bentuk *fault* yang umum terjadi. Karakteristik penting dari *fault* adalah sifatnya yang sering kali tersembunyi dan dapat tetap tidak terdeteksi hingga kondisi pemicu tertentu terpenuhi.

Error merupakan manifestasi dari *fault* yang menyebabkan penyimpangan dari keadaan sistem yang diharapkan. *Error* terjadi sebagai konsekuensi ketika sebuah *fault* teraktivasi atau terpanggil saat eksekusi kode, interaksi antar komponen, atau perubahan dalam lingkungan operasi. Beberapa contoh *error* yang umum ditemui meliputi nilai yang tidak valid dalam variabel program, status jaringan yang tidak konsisten, serta perbedaan *state* yang tidak seharusnya terjadi antara replika dalam sistem terdistribusi. *Error* menjadi penghubung antara *fault* yang mendasari dengan *failure* yang akhirnya dapat teramati.

Failure merupakan kondisi akhir yang terjadi ketika sistem tidak mampu memberikan layanan sesuai dengan spesifikasi yang ditentukan karena adanya *error*. *Failure* merupakan manifestasi eksternal dari *error* yang dapat diamati oleh pengguna atau sistem lain yang berinteraksi dengannya. Kegagalan dapat muncul dalam berbagai bentuk seperti respons yang salah dari API, pelanggaran batas waktu respons yang telah ditentukan (*timeout*), aplikasi yang mengalami *crash*, atau perilaku sistem yang tidak sesuai dengan dokumentasi dan spesifikasi. *Failure* menjadi indikator permasalahan yang terlihat oleh *end user* dari sistem.

Hubungan antara ketiga konsep ini membentuk rantai kausal seperti ditunjukkan pada Gambar 2.1. *Fault* menyebabkan *error* yang dapat mengakibatkan *failure*. *Failure* pada satu komponen dapat menjadi *fault* bagi komponen lain, menciptakan efek domino dalam sistem yang kompleks seperti sistem terdistribusi jika tidak dirancang dengan baik untuk mengisolasi dan menolerir *fault* [24].



Gambar 2.1. Hubungan kausal antara *Fault*, *Error*, dan *Failure*.

2.2.3 Jenis *Fault* dalam Sistem Terdistribusi

Sistem terdistribusi harus mampu mengatasi berbagai jenis *fault* yang dapat terjadi dalam komponen-komponennya. Dua kategori utama *fault* yang dibahas dalam literatur adalah *crash faults* dan *Byzantine faults* [8].

2.2.3.1 *Crash Faults*

Crash faults terjadi ketika komponen sistem berhenti berfungsi sepenuhnya tanpa memberikan respons atau melakukan komputasi lebih lanjut. Dalam kasus ini, komponen tersebut berhenti beroperasi secara total, tidak merespons pesan yang masuk, tidak mengirimkan pesan keluar, dan keadaannya tidak berubah setelah mengalami *crash*. Perilaku komponen yang mengalami *crash fault* relatif mudah diprediksi dan dikelola karena komponen tersebut hanya memiliki satu mode kegagalan, yaitu berhenti berfungsi sepenuhnya.

Sistem yang mampu beroperasi dengan benar meskipun beberapa komponennya mengalami *crash* disebut *Crash Fault-Tolerant* (CFT). Algoritma konsensus seperti Paxos [25, 26] dan Raft [27] dirancang khusus untuk menangani jenis *fault* ini [27]. Secara matematis, sistem CFT dapat menoleransi sebanyak f *fault* dengan kebutuhan minimal $2f + 1$ *node* total dalam sistem, di mana f merepresentasikan jumlah maksimum *node* yang dapat gagal secara bersamaan tanpa mengganggu operasi sistem secara keseluruhan.

2.2.3.2 *Byzantine Faults*

Byzantine faults, yang diperkenalkan oleh Lamport, Shostak, dan Pease dalam makalah klasik mereka yang berjudul "*The Byzantine Generals Problem*" [9], merepresentasikan model *fault* yang jauh lebih kompleks dan berbahaya dibandingkan dengan *crash faults*. *Fault* jenis ini mencakup segala bentuk perilaku yang tidak terduga atau berbahaya yang dapat mengganggu sistem terdistribusi.

Karakteristik *Byzantine fault* meliputi *node* yang mengirimkan pesan yang tidak konsisten ke berbagai *node* penerima, *node* yang secara sengaja memanipulasi sistem dengan menunda, memodifikasi, memalsukan pesan, atau *node* yang berkolusi untuk menyabotase sistem secara terkoordinasi. Selain itu, *Byzantine faults* juga mencakup perilaku non-deterministik atau perilaku yang tidak dapat diprediksi (*arbitrary*), serta pengiriman respons yang secara teknis valid tetapi dirancang untuk memboroskan sumber daya sistem.

Byzantine fault dapat terjadi karena berbagai alasan, mulai dari serangan berbahaya oleh aktor yang berniat jahat (*malicious actor*), komponen perangkat keras rusak yang menghasilkan output yang tidak terprediksi, *bug* pada perangkat lunak yang menyebabkan perilaku menyimpang, hingga *race condition* dan masalah pengaturan waktu dalam sistem asinkronus yang dapat memicu perilaku yang tidak konsisten.

2.2.4 *Byzantine Fault Tolerance*

Byzantine Fault Tolerance (BFT) adalah kemampuan sistem terdistribusi untuk mencapai konsensus dan beroperasi dengan benar meskipun beberapa komponen mengalami *Byzantine fault* [8, 10]. Konsep ini sangat penting dalam aplikasi yang membutuhkan tingkat keamanan dan keandalan tinggi, seperti sistem keuangan, infrastruktur kritis, *open metaverse*, dan aplikasi *blockchain*.

Secara teoretis, sistem BFT memiliki batasan fundamental yang dikenal sebagai BFT *Threshold*, yang menyatakan bahwa untuk menoleransi f *node* yang bersifat *Byzantine*, sistem memerlukan minimal $3f + 1$ total *node* [9]. Batasan ini telah terbukti secara matematis sebagai batas terendah yang mungkin dalam sistem asinkron yang tidak memiliki asumsi kriptografi tambahan.

Konsekuensi dari batasan ini adalah bahwa sistem BFT hanya dapat menolerir kurang dari sepertiga *node* yang berperilaku *Byzantine*. Jika proporsi *node Byzantine* melebihi sepertiga dari total *node*, tidak ada protokol BFT yang dapat menjamin kebenaran dan *liveness* secara bersamaan.

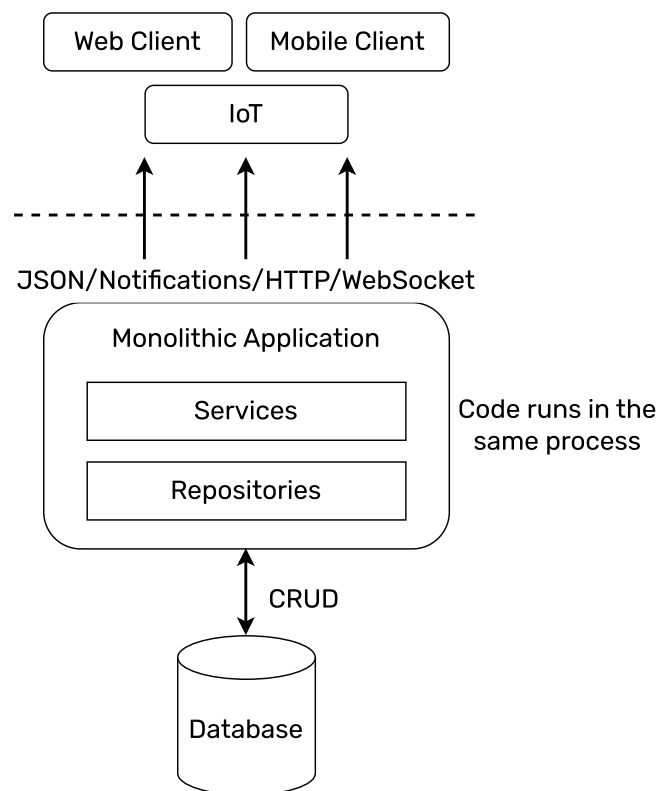
Selain jumlah *node*, sistem BFT juga memerlukan:

- Mekanisme autentikasi pesan yang kuat, biasanya menggunakan kriptografi *public key*.
- Jaringan komunikasi yang dapat menyampaikan pesan, meskipun mungkin terdapat *delay*.
- Determinisme dalam logika aplikasi untuk memastikan *node* yang jujur mencapai hasil akhir yang sama.

2.2.5 Arsitektur Web Services

Web services adalah teknologi yang memungkinkan aplikasi untuk berkomunikasi melalui jaringan dan memberikan layanan ke sistem lain menggunakan protokol standar seperti HTTP, terlepas dari platform atau bahasa pemrograman yang digunakan [28]. Arsitektur *web services* telah berkembang seiring waktu untuk memenuhi kebutuhan yang berbeda, khususnya dalam aspek ketahanan terhadap *fault* dan performa.

2.2.5.1 Arsitektur Monolitik



Gambar 2.2. Arsitektur monolitik. Diadaptasi dari karya Coelho *et al.* [29]

Arsitektur monolitik merupakan pendekatan tradisional dalam pengembangan *web services* di mana seluruh aplikasi dibangun sebagai satu kesatuan terintegrasi. Dalam arsitektur ini, semua fungsi aplikasi, termasuk antarmuka pengguna, logika bisnis, dan akses data berada dalam satu codebase yang dijalankan sebagai proses tunggal. *Web services* monolitik biasanya diimplementasikan sebagai layanan tunggal yang merespons berbagai jenis permintaan melalui API tunggal [29].

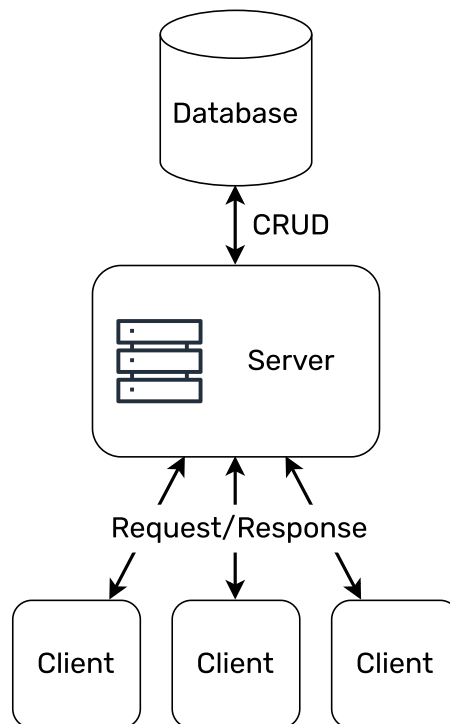
Gambar 2.2 menunjukkan arsitektur monolitik secara umum, di mana berbagai klien seperti *web client*, *mobile client*, dan perangkat IoT berinteraksi langsung dengan satu aplikasi terintegrasi yang mencakup layanan, repositori, serta koneksi ke basis data. Semua komponen tersebut dijalankan dalam satu proses yang sama.

Keunggulan utama arsitektur monolitik adalah kesederhanaan dalam pengembangan dan deployment. Pengembang dapat dengan mudah memahami keseluruhan aplikasi, *debugging* menjadi lebih mudah karena alur kode dapat dilacak dalam satu *codebase*, dan proses *deployment* hanya melibatkan satu aplikasi. Selain itu, performa untuk operasi internal cenderung lebih baik karena tidak ada *overhead* komunikasi jaringan antar komponen.

Namun, arsitektur monolitik memiliki keterbatasan signifikan ketika aplikasi berkembang lebih kompleks. Skalabilitas horizontal menjadi tantangan besar karena seluruh aplikasi harus di-*scale* sebagai satu unit, bahkan ketika hanya satu komponen yang membutuhkan sumber daya lebih. Fleksibilitas teknologi juga terbatas karena seluruh aplikasi harus menggunakan *stack* teknologi yang sama. Faktor paling penting adalah ketahanannya terhadap *fault* yang relatif rendah, karena kegagalan pada satu bagian aplikasi dapat mempengaruhi keseluruhan sistem [29].

Dari perspektif *Byzantine fault tolerance*, arsitektur monolitik sangat rentan karena memiliki titik kegagalan tunggal. Komponen yang mengalami *Byzantine fault* dapat membahayakan integritas dan ketersediaan seluruh aplikasi. Meskipun replikasi dapat diterapkan untuk mengurangi risiko ini, kompleksitas dalam mengelola replika monolitik lebih tinggi dibandingkan dengan arsitektur terdistribusi modern seperti *microservices* [29].

2.2.5.2 Arsitektur *Client-Server*



Gambar 2.3. Arsitektur *client-server*. Dibuat ulang berdasarkan karya Thampi *et al.* [21].

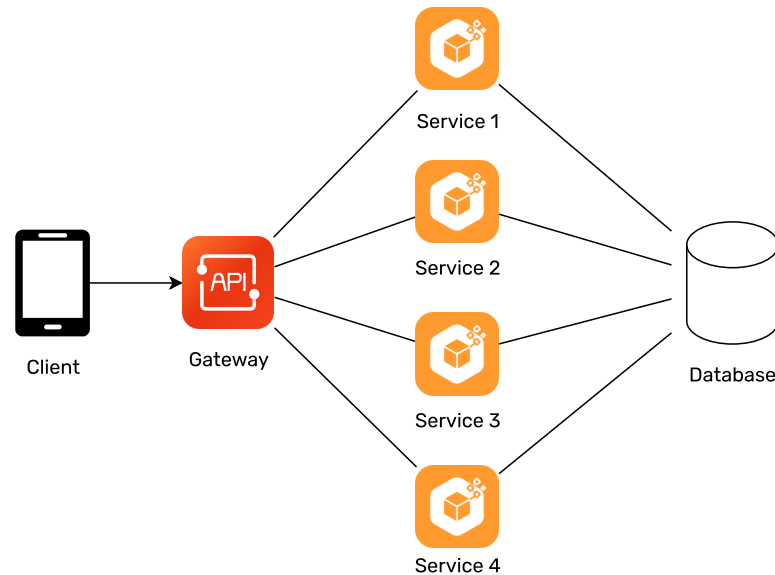
Arsitektur *client-server* adalah model dasar dalam pengembangan sistem terdistribusi dan *web services*, yang menggambarkan hubungan struktural antara penyedia layanan (server) dan konsumen layanan (klien) [30]. Gambar 2.3 memperlihatkan ilustrasi arsitektur ini, di mana sejumlah klien mengirimkan permintaan ke satu entitas server pusat yang kemudian memproses permintaan tersebut dan merespons kembali, biasanya dengan proses mengakses *database* di belakangnya.

Pola komunikasi yang umum digunakan dalam arsitektur ini adalah *request-compute-response* [14]. Dalam pola ini, klien mengirimkan permintaan ke server, server melakukan komputasi yang diperlukan dan pengambilan data, lalu mengembalikan respons ke klien [21]. Pola ini bersifat sinkron, di mana klien harus menunggu respons sebelum melanjutkan eksekusi, dan cenderung *stateless* karena setiap permintaan dianggap independen [30].

Keunggulan utama dari model ini terletak pada kesederhanaan implementasi, skalabilitas yang lebih baik dari arsitektur monolitik, serta kompatibilitasnya yang tinggi dengan infrastruktur web modern. Namun, model ini bergantung pada asumsi bahwa server akan selalu *available* dan bertindak jujur, yang menjadi kelemahan dalam konteks sistem yang memerlukan ketahanan dan jaminan terhadap gangguan atau *Byzantine fault*. Ketika server mengalami kegagalan atau berperilaku *Byzantine*, seluruh sistem klien

dapat terdampak, menjadikan model ini kurang cocok untuk aplikasi yang menuntut ketahanan tinggi.

2.2.5.3 Arsitektur *Microservices*



Gambar 2.4. Arsitektur *microservices* yang tersusun atas 4 *service*.

Arsitektur *microservices* merupakan pendekatan modern dalam pengembangan aplikasi yang memecah sistem menjadi serangkaian layanan kecil, independen, dan terdesentralisasi. Masing-masing layanan, atau *microservice*, memiliki tanggung jawab tunggal (*single responsibility*) dan dapat dikembangkan, diuji, di-deploy, serta di-scale secara independen [31]. Gambar 2.4 menunjukkan bagaimana layanan-layanan tersebut diakses melalui API gateway dan masing-masing dapat memiliki *database* atau sumber daya baik itu global atau terpisah.

Komunikasi antar-layanan biasanya dilakukan melalui protokol ringan seperti HTTP REST, gRPC, atau messaging queue seperti Kafka atau RabbitMQ. Setiap layanan dapat pula menggunakan bahasa pemrograman, *library*, atau sistem penyimpanan yang berbeda sesuai kebutuhan fungsionalnya. Fleksibilitas ini memberikan keunggulan dalam pengembangan lintas tim dan pemeliharaan jangka panjang.

Keunggulan utama dari arsitektur *microservices* terletak pada tiga aspek penting. Pertama adalah skalabilitas, di mana setiap layanan dapat di-scale secara terpisah sesuai dengan kebutuhan beban kerjanya masing-masing, tanpa harus ikut meng-scale seluruh sistem. Kedua adalah ketahanan, karena kegagalan pada satu layanan tidak langsung memengaruhi layanan lain, sistem menjadi lebih toleran terhadap kegagalan lokal. Ketiga adalah kemudahan dan fleksibilitas dalam pengembangan, di mana tim *developer* dapat bekerja secara mandiri pada layanan tertentu, sehingga iterasi dan pengiriman fitur dapat

dilakukan dengan lebih cepat.

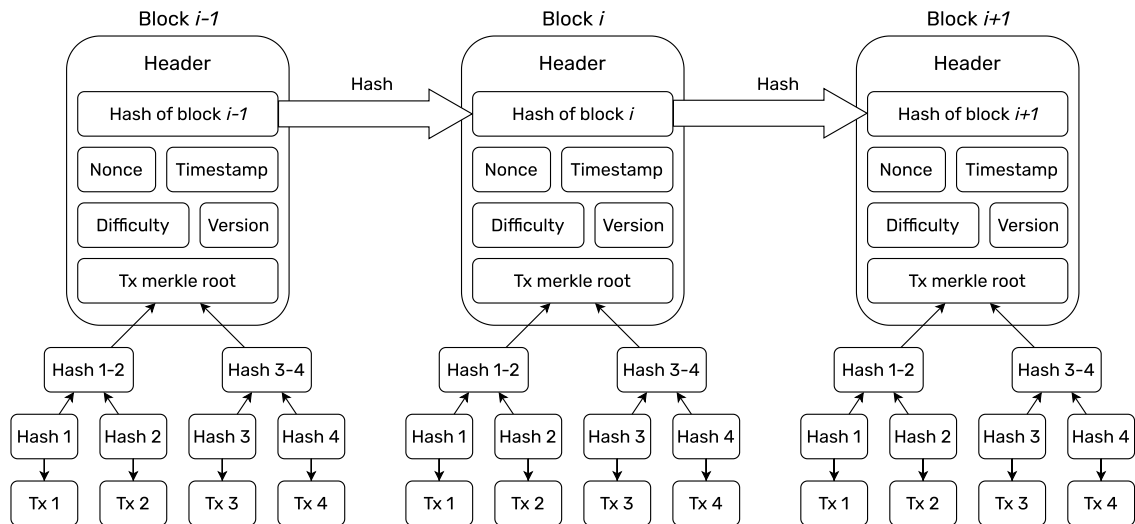
Walaupun demikian, pendekatan *microservices* ini juga menimbulkan sejumlah tantangan baru. Salah satunya adalah kompleksitas koordinasi. Dibandingkan dengan arsitektur monolitik, koordinasi antar-layanan menjadi jauh lebih kompleks dan membutuhkan pendekatan orkestrasi yang lebih matang. Tantangan lainnya berkaitan dengan konsistensi data, karena data tersebar di berbagai layanan, menjaga konsistensi global menjadi lebih sulit, terlebih dalam sistem yang membutuhkan transaksi lintas layanan. Terakhir, pengujian dan observabilitas juga menjadi lebih sulit, karena pengujian menyeluruh serta pelacakan lintas layanan memerlukan alat dan strategi tambahan akibat meningkatnya kompleksitas sistem.

Dalam konteks ketahanan terhadap *Byzantine fault*, arsitektur *microservices* tidak serta merta memberikan properti dan ketahanan BFT. Namun, karena sifat terdesentralisasi dan terbatasnya efek kerusakan akibat kegagalan suatu *microservice*, pendekatan ini lebih mudah dikombinasikan dengan sistem konsensus SMR atau model layanan berbasis *Byzantine Fault Tolerance* dibandingkan arsitektur monolitik. Oleh karena itu, banyak sistem modern yang membutuhkan ketahanan tingkat tinggi dan fleksibilitas operasional lebih memilih pendekatan *microservices* sebagai fondasi arsitektur mereka.

2.2.6 Blockchain dan Algoritma Konsensus

2.2.6.1 Teknologi Blockchain

Blockchain adalah teknologi fundamental dalam sistem terdistribusi modern yang pada awalnya dirancang untuk mendukung mata uang kripto (*cryptocurrency*) seperti Bitcoin, namun kini telah berkembang menjadi fondasi arsitektur untuk berbagai sistem yang membutuhkan integritas, transparansi, dan kepercayaan dalam sistem terdistribusi. Secara konseptual, *blockchain* adalah struktur data berbentuk rantai terhubung dari blok-blok data, di mana setiap blok menyimpan sekelompok transaksi dan informasi tambahan serta mengandung *hash* kriptografis dari blok sebelumnya. Terhubungnya blok dengan relasi kriptografis tersebut menciptakan struktur rantai yang sangat sulit dimodifikasi, karena perubahan pada satu blok akan memengaruhi semua blok setelahnya. Karakteristik tersebut disebut sebagai *immutability* yang merupakan salah satu karakteristik utama dari *blockchain*.



Gambar 2.5. Struktur data *blockchain*. Diadaptasi dari karya Sadawi *et al.* [32].

Gambar 2.5 menunjukkan visualisasi struktur dasar *blockchain*. Setiap blok terdiri atas dua bagian utama: *header* dan *Merkle tree*. *Header* memuat metadata seperti hash blok sebelumnya, *timestamp*, tingkat kesulitan (*difficulty*), versi protokol, dan nilai *nonce* yang relevan dalam proses *mining*. Sementara itu, *Merkle root* merupakan hasil *hashing* rekursif dari semua transaksi dalam blok. Dengan menggunakan struktur *Merkle tree*, *blockchain* mampu melakukan verifikasi transaksi secara efisien tanpa harus mengakses seluruh isi blok, yang sangat penting dalam konteks *light client* dan auditabilitas.

Dari sudut pandang arsitektural, *blockchain* tidak hanya struktur data, tetapi juga sistem jaringan terdistribusi berbasis *peer-to-peer* (P2P). Dalam jaringan ini, setiap node menyimpan salinan lengkap dari *ledger*, menerima transaksi baru, dan mengambil bagian dalam proses validasi dan konsensus. Ketika sebuah transaksi baru diajukan, transaksi tersebut disebar ke seluruh jaringan melalui mekanisme *broadcast* seperti algoritma gosip atau semacamnya, diverifikasi berdasarkan aturan protokol, dan kemudian dikumpulkan oleh *node* tertentu untuk dimasukkan ke dalam blok baru. Penambahan blok ini bukan hanya proses lokal, tetapi melibatkan keputusan kolektif seluruh jaringan melalui konsensus.

Efisiensi dan kecepatan dalam *blockchain* sangat bergantung pada jenis dan konfigurasi jaringannya. *Blockchain* publik seperti Bitcoin dan Ethereum menawarkan transparansi, tetapi mengorbankan throughput dan latensi demi desentralisasi. Sebaliknya, *blockchain private* dan *permissioned* seperti Hyperledger Fabric atau Quorum dirancang untuk skenario bisnis, dengan fokus pada efisiensi, kendali akses, dan performa tinggi. *Blockchain* jenis ini memungkinkan partisipasi entitas yang dikenali dan dipercaya secara parsial, sehingga mekanisme konsensus dapat lebih ringan dan cepat.

Dari sisi keamanan, *blockchain* menawarkan kombinasi kuat antara kriptografi, replikasi data, dan protokol konsensus. Integritas data dijaga melalui penggunaan fungsi *hash* kriptografis yang bersifat satu arah dan sangat sulit untuk dipalsukan. Autentikasi identitas umumnya berbasis kriptografi *public key*. Selain itu, properti *immutability* membuat manipulasi data hampir tidak mungkin tanpa kontrol terhadap sebagian besar (mayoritas) *node* dalam jaringan. Meskipun demikian, teknologi ini tidak lepas dari tantangan. Masalah utama seperti skalabilitas, konsumsi energi (terutama dalam PoW), latensi, *throughput*, dan kesulitan dalam integrasi dengan sistem eksternal masih menjadi subjek penelitian yang aktif.

Salah satu perbedaan utama antara *blockchain* dan sistem *database* terdistribusi tradisional adalah pada model kepercayaannya (*trust model*). Sistem *database* klasik diasumsikan berada dalam kontrol badan administratif yang terpercaya dan berfokus pada konsistensi serta efisiensi transaksi. Sebaliknya, *blockchain* dirancang untuk lingkungan tanpa kepercayaan penuh antar entitas yang berpartisipasi. Oleh karena itu, *blockchain* mengutamakan transparansi, redundansi, dan verifikasi independen atas setiap transaksi, meskipun itu berarti mengorbankan performa.

Dalam konteks penelitian ini, *blockchain* memiliki peran yang sangat relevan sebagai basis bagi arsitektur *web service* terdesentralisasi yang tahan terhadap *Byzantine fault*. Implementasi seperti DeWS [14] menunjukkan bagaimana *blockchain* tidak hanya berfungsi sebagai *ledger* untuk pencatatan data, tetapi juga sebagai mekanisme koordinasi antar *node* dalam mencapai kesepakatan atas hasil eksekusi semua server. DeWS mengintegrasikan *blockchain* dalam skema *request-compute-consensus-log-response*, di mana setiap *request* dari klien tidak langsung diberikan respons, tetapi direplikasi dan diverifikasi secara deterministik oleh seluruh *node* sebelum disimpan dalam *immutable ledger*. Pendekatan ini membuka jalan untuk arsitektur *web services* yang tidak hanya *scalable* dan modular, tetapi juga mendukung proses audit, akuntabel, dan aman terhadap ancaman pihak *malicious* yang berpotensi menguasai sebagian besar sistem.

Dengan demikian, pemahaman mendalam mengenai prinsip dan arsitektur *blockchain* menjadi landasan penting dalam merancang sistem terdistribusi yang tidak hanya fungsional, tetapi juga tahan terhadap *fault*, transparan dalam operasionalnya, dan dapat dipercaya dalam jangka panjang khususnya dalam lingkungan *multi-stakeholder* yang tidak saling mempercayai.

2.2.6.2 Algoritma dan Mekanisme Konsensus *Blockchain*

Komponen kunci dari setiap sistem *blockchain* adalah algoritma dan mekanisme konsensus yang digunakan untuk menyetujui keadaan *ledger* saat ini. Algoritma konsensus yang mendasari *blockchain* adalah protokol yang memungkinkan sekelompok

entitas terdistribusi untuk mencapai kesepakatan tentang nilai atau *state* bersama, meskipun dalam lingkungan di mana kepercayaan tidak dapat dibangun antar entitasnya. Berbagai pendekatan telah dikembangkan, masing-masing dengan *trade-off* antara keamanan, skalabilitas, dan desentralisasi [33]:

Proof of Work (PoW) adalah algoritma konsensus berbasis kompetisi komputasi, di mana *node* bersaing untuk menemukan nilai *nonce* yang ketika dikombinasikan dengan data blok menghasilkan *hash* di bawah ambang batas (*threshold*) tertentu (*target*). Ambang batas ini dikontrol oleh tingkat kesulitan (*difficulty*) yang disesuaikan secara berkala agar waktu antara blok tetap konstan, misalnya 10 menit dalam Bitcoin. Fungsi hash yang digunakan, seperti SHA-256, bersifat deterministik namun tidak dapat diprediksi, sehingga satu-satunya cara untuk menemukan *nonce* yang valid adalah dengan mencoba semua kemungkinan secara *brute-force* yang boros energi.

Keamanan dalam PoW bergantung pada asumsi bahwa mayoritas kekuatan komputasi (*hash rate*) dalam jaringan dikendalikan oleh entitas yang jujur. Jika entitas jahat menguasai lebih dari 50% dari total *hash rate*, mereka dapat melakukan serangan seperti *double spending* atau melakukan reorganisasi *blockchain* yang juga dikenal sebagai 51% *attack*. Selain itu, karena *miner* perlu menyiarkan blok ke *node* lain dalam jaringan, begitu mereka menyelesaikan persoalan kriptografinya, penundaan propagasi blok dapat menyebabkan munculnya *stale blocks*, yaitu blok valid yang tidak masuk ke rantai utama. Terutama dalam jaringan dengan latensi tinggi.

PoW juga bersifat probabilistik. Meskipun blok yang valid ditemukan dalam rata-rata interval yang tetap, waktu sebenarnya yang diperlukan untuk menemukan blok bersifat acak. Karena itu, finalitas dalam sistem PoW tidak bersifat instan, melainkan probabilistik. Semakin banyak konfirmasi blok yang diterima, semakin tinggi probabilitas bahwa transaksi tersebut terkonfirmasi sebagai final dan tidak dibatalkan. Dalam Bitcoin, standarnya adalah menunggu enam blok sebagai konfirmasi agar transaksi dianggap final. Meskipun menawarkan keamanan dan desentralisasi yang tinggi, PoW dikritik karena konsumsi energi yang sangat besar dan tingkat *throughput* yang rendah, hanya sekitar 7 transaksi per detik (TPS) dalam jaringan Bitcoin [34, 35]. Dibandingkan dengan *throughput* sistem non-*blockchain* seperti Visa yang dapat mencapai 65000 TPS, *throughput* Bitcoin sangat rendah.

Proof of Stake (PoS) adalah algoritma konsensus yang memilih *validator* untuk membuat dan mengusulkan blok berdasarkan jumlah aset digital yang mereka pertaruhkan (*staking*) dalam sistem [35]. Semakin besar jumlah token yang dipertaruhkan, semakin besar peluang validator tersebut untuk dipilih. Pendekatan ini menghilangkan kebutuhan komputasi intensif seperti pada *Proof-of-Work*, sehingga jauh lebih efisien dalam penggunaan energi dan mampu menghasilkan *throughput* yang lebih tinggi.

Dalam mekanisme PoS, keamanan sistem dijaga melalui insentif ekonomi. Validator yang mencoba bertindak curang, misalnya dengan menyetujui dua blok berbeda pada saat yang sama, dapat dikenakan hukuman berupa pengurangan token yang mereka pertaruhkan. Sistem ini juga biasanya menetapkan periode penguncian token untuk mencegah validator menarik kembali taruhannya setelah melakukan pelanggaran.

Namun, PoS juga menghadapi tantangan tertentu. Salah satunya adalah masalah *nothing-at-stake*, yaitu ketika validator, terutama dalam desain PoS awal yang belum memiliki mekanisme penalti, memiliki insentif untuk mendukung beberapa blok secara bersamaan karena tidak ada risiko langsung terhadap aset yang mereka miliki. Untuk mengatasi hal ini, sistem PoS modern biasanya dilengkapi dengan mekanisme penalti dan deteksi perilaku yang tidak memenuhi protokol. Selain itu, beberapa sistem juga menghadapi risiko serangan jangka panjang dari pihak yang pernah menjadi validator di masa lalu, sehingga memerlukan teknik seperti *checkpointing* untuk menjaga riwayat *blockchain*.

Ethereum adalah salah satu platform terbesar yang telah beralih dari PoW ke PoS. Perubahan ini dilakukan melalui *update* besar yang dikenal sebagai *The Merge* pada September 2022 [36]. Sejak saat itu, Ethereum menggunakan validator sebagai pengganti *miner* dan secara signifikan mengurangi konsumsinya sambil mempertahankan tingkat keamanan dan desentralisasi yang tinggi.

Delegated Proof of Stake (DPoS) merupakan varian dari PoS yang menggabungkan elemen partisipasi komunitas dan efisiensi delegatif. Dalam mekanisme ini, pemilik token tidak berpartisipasi langsung sebagai *validator*, melainkan memilih sejumlah kecil perwakilan melalui proses voting berbasis token untuk menjalankan tugas validasi blok [35]. Model ini memungkinkan waktu blok yang lebih cepat dan peningkatan *throughput*, namun mengorbankan sebagian aspek desentralisasi karena hanya ada sedikit *validator* aktif yang beroperasi dalam satu waktu.

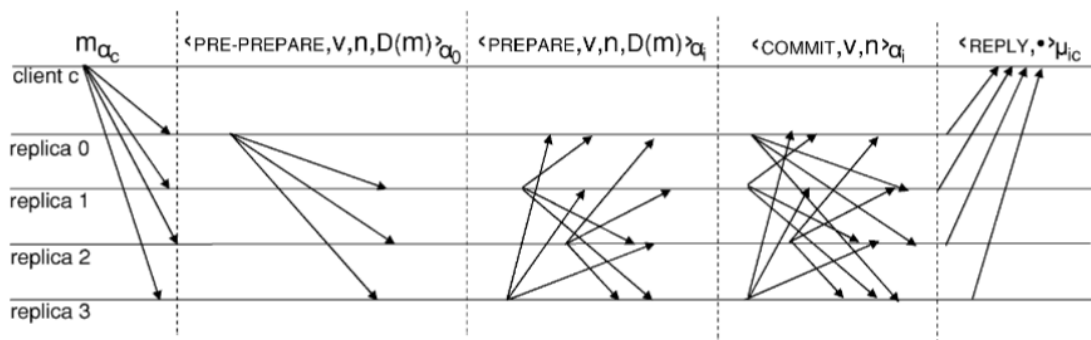
Keamanan dan keandalan dalam DPoS sangat bergantung pada akuntabilitas publik karena validator yang tidak aktif atau bertindak tidak jujur dapat segera diganti oleh pemilihnya. Sistem ini digunakan dalam beberapa *blockchain* seperti EOS dan TRON, yang memprioritaskan kecepatan dan efisiensi untuk aplikasi terdesentralisasi skala besar [37,38].

Paxos [25, 26] dan **Raft** [27] adalah algoritma konsensus yang dirancang untuk sistem dengan model *Crash Fault Tolerance* (CFT), di mana *node* hanya dianggap gagal jika berhenti bekerja, bukan jika bertindak secara jahat atau arbitrer seperti dalam model *Byzantine*. Kedua algoritma ini menggunakan pendekatan berbasis pemimpin (*leader-based*) untuk menyederhanakan proses pemilihan nilai yang disepakati dalam replikasi status sistem.

Paxos dikenal dengan pendekatannya yang formal dan terbukti secara matematis, namun sulit dipahami dan diimplementasikan. Raft dikembangkan sebagai alternatif yang lebih mudah dipahami dan diimplementasikan secara praktis, dengan membagi proses konsensus menjadi fase pemilihan pemimpin, replikasi log, dan komitmen. Karena kompleksitas komunikasi yang lebih rendah dibandingkan protokol BFT, algoritma ini banyak digunakan dalam sistem terdistribusi yang dikendalikan secara administratif seperti database replikasi dan penyimpanan konsensus terpusat.

Practical Byzantine Fault Tolerance (PBFT) menggunakan pendekatan tiga fase untuk mencapai konsensus di antara *node* yang mungkin berperilaku *Byzantine*, tiga fase tersebut adalah:

1. **Pre-prepare:** *Leader* membuat blok transaksi dan mengirimkannya ke semua replika lain.
2. **Prepare:** Replika memverifikasi proposal dari *leader* dan menyebarkan konfirmasi persiapan mereka ke semua replika.
3. **Commit:** Setelah menerima $2f$ pesan *prepare*, replika menyebarkan pesan *commit* dan menunggu $2f + 1$ pesan *commit* sebelum menerapkan transaksi ke *state machine* mereka.



Gambar 2.6. Protokol tiga fase PBFT untuk mencapai konsensus dalam sistem BFT (Castro *et al.* [10]).

Gambar 2.6 menunjukkan alur komunikasi lengkap dalam protokol PBFT yang melibatkan satu klien dan empat replika dalam sistem BFT. Proses dimulai ketika klien mengirimkan permintaan operasi (m_c) ke semua replika dalam sistem. Replika 0 bertindak sebagai *primary node* yang akan memimpin proses konsensus untuk permintaan tersebut.

Dalam fase *pre-prepare*, *primary* (replika 0) mengirimkan pesan $\langle \text{PRE-PREPARE}, v, n, D(m) \rangle$ ke semua *backup replicas* (replika 1, 2, dan 3). Pesan ini berisi nomor *view* (v), nomor urut (n), dan *digest* dari permintaan klien ($D(m)$). Setiap *backup replica* memverifikasi validitas pesan berdasarkan tanda tangan digital, nomor urut yang konsisten, dan kesesuaian dengan *view* saat ini.

Fase *prepare* ditunjukkan dengan pola komunikasi *all-to-all* di mana setiap replika (termasuk *primary*) mengirimkan pesan $\langle \text{PREPARE}, v, n, D(m) \rangle$ ke semua replika lain. Dalam fase ini, replika mengkonfirmasi bahwa mereka telah menerima dan memvalidasi proposal dari *primary*. Setiap replika akan menunggu hingga menerima setidaknya $2f$ pesan *prepare* yang valid dari replika lain sebelum melanjutkan ke fase berikutnya.

Fase *commit* juga menggunakan pola komunikasi *all-to-all*, di mana setiap replika mengirimkan pesan $\langle \text{COMMIT}, v, n \rangle$ ke semua replika lain. Replika akan menunggu hingga mengumpulkan $2f + 1$ pesan *commit* (termasuk voting miliknya sendiri) sebelum menjalankan operasi yang diminta. Setelah eksekusi selesai, setiap replika mengirimkan balasan $\langle \text{REPLY}, * \rangle$ langsung ke klien, yang berisi hasil operasi.

Klien akan menunggu hingga menerima $f + 1$ balasan yang identik dari replika yang berbeda sebelum menerima hasil sebagai valid. Mekanisme ini memastikan bahwa klien menerima hasil yang benar meskipun hingga f replika berperilaku *Byzantine*. Kompleksitas komunikasi protokol PBFT adalah $O(n^2)$ karena setiap fase *prepare* dan *commit* memerlukan pertukaran pesan antara semua pasangan replika.

PBFT lebih efisien daripada algoritma konsensus blockchain seperti PoW tetapi skalabilitasnya terbatas oleh kompleksitas komunikasi yang meningkat secara kuadratik dengan jumlah *node* [10, 35]. Setiap fase memerlukan persetujuan dari mayoritas *node* untuk menjamin integritas transaksi, dengan setidaknya $2f + 1$ *node* yang harus setuju dalam sistem dengan total $3f + 1$ *node*. Kompleksitas komunikasi ini menjadi bottleneck utama dalam implementasi PBFT untuk sistem dengan jumlah *node* yang besar, meskipun protokol ini menyediakan finalitas deterministik dan latensi yang dapat diprediksi dibandingkan dengan algoritma konsensus probabilistik.

Tendermint adalah algoritma konsensus *Byzantine fault-tolerant* yang menyediakan finalitas deterministik tanpa memerlukan kerja komputasi intensif seperti pada *Proof-of-Work* [39, 40]. Protokol ini mengadopsi model konsensus dua putaran yang terinspirasi dari PBFT, namun disederhanakan menjadi tiga tahap pesan utama: *Proposal*, *Pre-Vote*, dan *Pre-Commit*. Setiap *round* dalam Tendermint memiliki satu *proposer* yang bertugas mengusulkan blok, yang kemudian divalidasi oleh *validator* lain melalui proses voting.

Konsensus dicapai apabila lebih dari dua pertiga dari total *voting power* mengirimkan pesan *Pre-Commit* untuk proposal yang sama dalam suatu *round*. Jika kuorum tidak tercapai, protokol secara otomatis melanjutkan ke *round* berikutnya dengan *proposer* yang berbeda. Proses ini memungkinkan toleransi terhadap *crash fault* maupun *Byzantine fault* dalam batas $f < \frac{n}{3}$, di mana n adalah jumlah total *validator*.

Tendermint Core, sebagai implementasi referensi dari protokol ini, telah

berkembang menjadi CometBFT yang mempertahankan prinsip desain aslinya sambil meningkatkan stabilitas dan modularitas untuk adopsi sistem di dunia nyata. Karena protokol ini memberikan latensi rendah, finalitas instan, serta keamanan deterministik terhadap *Byzantine fault*. Tendermint dan turunannya, yang digunakan dalam Tendermint Core dan CometBFT, sering diadopsi dalam jaringan *blockchain* sebagai bagian dari ekosistemnya, salah satunya pada *blockchain* Cosmos [41].

Salah satu keunggulan arsitektural Tendermint dalam Tendermint Core dan CometBFT adalah pemisahan antara *consensus engine* dan logika aplikasi melalui *Application Blockchain Interface* (ABCI). Hal ini memungkinkan pengembang untuk membangun aplikasi deterministik dalam berbagai bahasa pemrograman yang dapat dijalankan secara konsisten di atas jaringan replikasi BFT.

Selain itu, Tendermint mengandalkan protokol penyebaran pesan berbasis *gossip*, di mana pesan konsensus seperti *Pre-Vote* dan *Pre-Commit* disebarkan secara efisien antar *validator* melalui propagasi bertingkat, bukan pertukaran langsung antar setiap pasangan *node*. Pendekatan ini menghasilkan kompleksitas komunikasi yang lebih rendah secara praktis, yakni mendekati $O(n \log n)$, dibandingkan PBFT yang memerlukan $O(n^2)$ pertukaran pesan pada setiap tahap konsensus. Dengan demikian, Tendermint memiliki skalabilitas yang lebih baik untuk jaringan dengan jumlah *validator* yang besar, sekaligus mempertahankan finalitas deterministik dan toleransi terhadap $f < \frac{n}{3}$ *Byzantine validator*. Karakteristik ini menjadikannya pilihan unggulan untuk implementasi sistem berbasis *blockchain* yang memerlukan efisiensi komunikasi dan kepastian eksekusi dalam ekosistem terdistribusi.

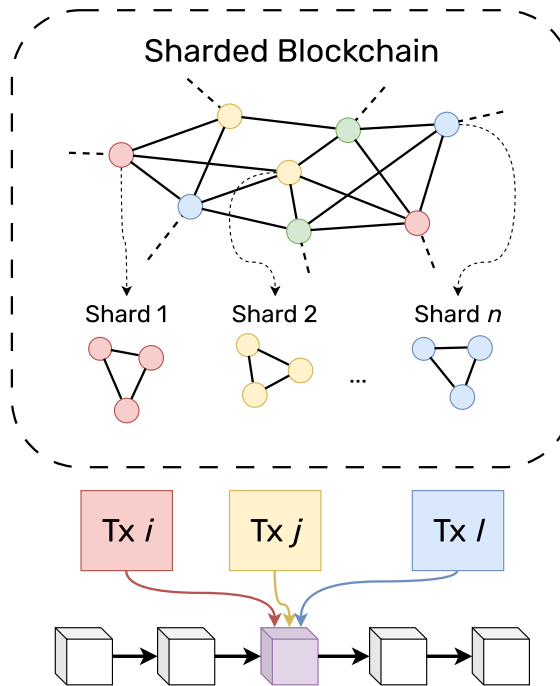
2.2.7 Solusi Scalability Blockchain

Masalah skalabilitas dalam sistem blockchain dan konsensus terdistribusi telah mendorong pengembangan berbagai solusi arsitektural. Solusi-solusi ini bertujuan untuk meningkatkan *throughput* dan mengurangi latensi sambil mempertahankan properti keamanan yang diperlukan.

2.2.7.1 Blockchain Sharding

Blockchain sharding adalah teknik penskalaan horizontal yang membagi jaringan *blockchain* menjadi beberapa bagian yang disebut *shard*, di mana masing-masing *shard* dapat memproses transaksi dan menyimpan data secara paralel. Pendekatan ini bertujuan untuk mengatasi batasan skalabilitas pada arsitektur *blockchain* monolitik, di mana seluruh *node* harus memproses semua transaksi secara global. Dengan membagi beban kerja ke dalam beberapa *shard*, *throughput* jaringan secara teoretis dapat meningkat secara linier seiring dengan jumlah *shard* yang tersedia [42].

Gambar 2.7 menunjukkan arsitektur umum dari sistem *blockchain* yang

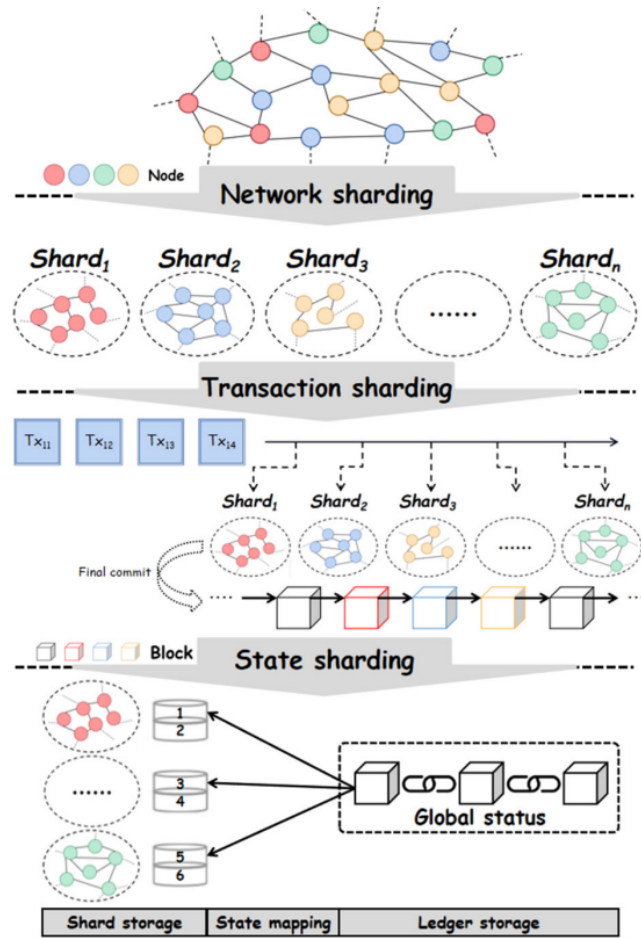


Gambar 2.7. Ilustrasi *blockchain sharding*. Diadaptasi karya dari Liu *et al.* [42]

menggunakan *sharding*. Dalam model ini, setiap *shard* terdiri dari subset *validator* yang hanya bertanggung jawab atas transaksi dan status dalam *shard* tersebut. Meskipun *shard* berjalan secara independen, diperlukan mekanisme khusus untuk menangani transaksi lintas *shard* dan memastikan konsistensi global antar status *shard* yang berbeda.

Terdapat beberapa pendekatan dalam menerapkan *sharding* pada *blockchain*, sebagaimana diilustrasikan dalam Gambar 2.8. Pertama, *network sharding* membagi jaringan node menjadi kelompok-kelompok yang bertanggung jawab terhadap subset *shard* tertentu, mengurangi beban komunikasi antar seluruh node. Kedua, *transaction sharding* mendistribusikan transaksi ke dalam *shard* yang berbeda, sehingga setiap transaksi hanya diproses oleh validator dalam satu *shard*. Ketiga, *state sharding* membagi data status (misalnya saldo akun atau *smart contract*) ke berbagai *shard*, sehingga setiap validator hanya menyimpan sebagian kecil dari keseluruhan status jaringan.

Masing-masing pendekatan memiliki kompleksitas tersendiri, terutama dalam hal komunikasi antar *shard* (*cross-shard communication*), validasi lintas data, dan pemeliharaan konsistensi global. Misalnya, transaksi lintas *shard* memerlukan protokol khusus seperti penggunaan *receipt*, *relay*, atau protokol sinkronisasi berbasis waktu komit global. Selain itu, perlu ada mekanisme penempatan validator secara acak ke dalam *shard* untuk mencegah konsentrasi kekuasaan atau serangan koordinatif pada *shard* tertentu, yang biasanya diimplementasi menggunakan *verified random functions* (VRF) atau pengambilan sampel terdesentralisasi.



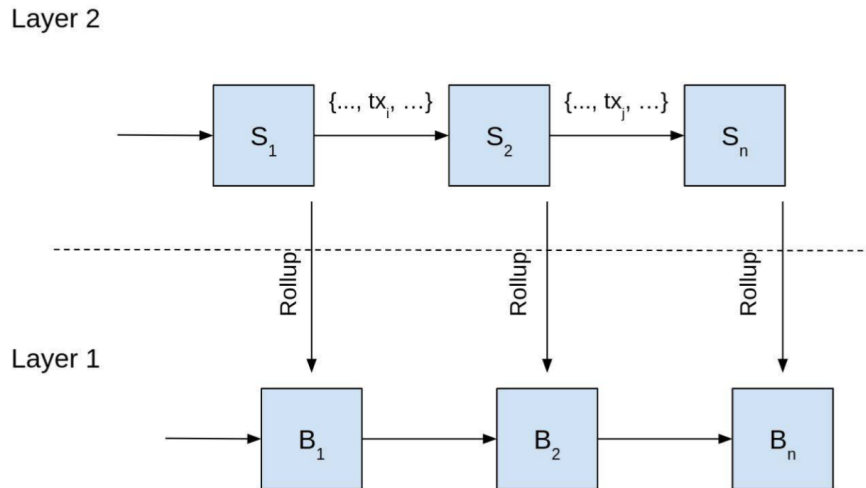
Gambar 2.8. Tipe-tipe *sharding* (Liu *et al.* [42]).

Salah satu contoh implementasi nyata dari *sharding* adalah pada Ethereum 2.0 yang menggabungkan *state sharding* dan *transaction sharding*, yang direncanakan akan bekerja dengan sistem *rollup* untuk mencapai efisiensi skalabilitas yang lebih tinggi. Meskipun *sharding* menawarkan potensi peningkatan *throughput* yang signifikan, desain dan implementasinya masih dalam penelitian dan harus memperhitungkan aspek keamanan, keterbukaan, dan interoperabilitas untuk memastikan bahwa sistem tetap tahan terhadap *Byzantine fault* dan kegagalan sinkronisasi.

2.2.7.2 Rollups

Rollups adalah solusi *scaling* yang memindahkan eksekusi transaksi ke *layer* terpisah (*Layer 2*), sambil tetap menyimpan data transaksi atau bukti eksekusi pada *blockchain* utama (*Layer 1*). Dengan demikian, sistem dapat meningkatkan *throughput* dan mengurangi biaya komputasi sambil tetap mewarisi tingkat keamanan dan desentralisasi dari *Layer 1*. Gambar 2.9 mengilustrasikan hubungan antara *Layer 2* sebagai tempat eksekusi dan *Layer 1* sebagai tempat komitmen dan verifikasi hasil. Setiap kali *state* pada L2 berpindah, yaitu dari S_1 ke S_2 , dan seterusnya hingga S_n , perpindahan tersebut mencerminkan hasil dari eksekusi satu batch transaksi yang

membentuk urutan perubahan keadaan sistem secara deterministik. Setelah setiap transisi state selesai, hasil eksekusi seperti root state atau bukti eksekusi dalam bentuk *fraud proof* atau *zero-knowledge proof* akan dikirim ke *Layer 1* dalam bentuk *Rollup* dan disimpan dalam blok yang sesuai (B_1, B_2, \dots, B_n). Dengan demikian, *Layer 2* bertindak sebagai mesin eksekusi yang efisien, sedangkan *Layer 1* menjaga integritas data dan menyediakan fondasi keamanan melalui pencatatan hasil state transition dari *Layer 2*.



Gambar 2.9. Interaksi antara *Layer 1* dan *Layer 2* dalam arsitektur *Rollup*, (Thibault *et al.* [19]). *Layer 2* mengeksekusi transaksi dan mengirim bukti kriptografis ke *Layer 1*.

Terdapat dua jenis utama *rollup* yang banyak digunakan saat ini: *Optimistic Rollups* dan *ZK-Rollups*. *Optimistic Rollups* mengasumsikan bahwa semua transaksi valid secara *default*, dan hanya akan dilakukan pemeriksaan jika ada pihak yang mengajukan *fraud proof* dalam periode tertentu. Pendekatan ini relatif lebih sederhana secara komputasi, namun memperkenalkan latensi dalam finalitas karena menunggu masa tantangan (*challenge period*). Sementara itu, *ZK-Rollups* menggunakan *Zero-Knowledge Proofs* untuk membuktikan bahwa transaksi dalam satu *batch* valid sebelum dicatat ke *Layer 1*. Pendekatan ini memberikan finalitas yang lebih cepat dan keamanan kriptografis yang lebih kuat, tetapi lebih kompleks secara teknis dan memerlukan proses pembuktian (*proof generation*) yang berat.

Keunggulan utama dari *rollup* terletak pada efisiensinya, di mana transaksi dapat diproses secara massal di luar *Layer 1* dan kemudian digabungkan ke dalam satu transaksi komitmen yang mewakili ribuan transaksi individual. Hal ini tidak hanya menurunkan biaya transaksi, tetapi juga mengurangi beban pemrosesan dan penyimpanan di *blockchain* utama. Karena data dan bukti tetap dicatat di *Layer 1*, *rollup* tetap mendapatkan jaminan keamanan dari *layer* dasar tanpa memerlukan validator tambahan atau konsensus tersendiri.

Namun, *rollup* juga menghadapi tantangan teknis, terutama dalam hal ketersediaan data (*data availability*) dan interoperabilitas antar *rollup*. Untuk memastikan bahwa transaksi dapat diverifikasi oleh *node* independen, data transaksi yang diproses di *Layer 2* tetap harus tersedia dan dapat diakses dari *Layer 1*. Selain itu, ekosistem *rollup* yang terfragmentasi dapat menyebabkan hambatan dalam transfer aset dan kontrak antar *layer*, sehingga diperlukan pengembangan solusi lintas *rollup* seperti *inter-rollup bridges*.

Rollup saat ini menjadi pendekatan paling populer dalam ekosistem Ethereum, dengan berbagai implementasi seperti Optimism, Arbitrum, dan zkSync yang menunjukkan keberhasilan dalam menurunkan biaya transaksi dan meningkatkan kapasitas sistem. Pendekatan ini juga kompatibel dengan model keamanan yang sudah mapan. Oleh karena itu, *rollup* banyak dipertimbangkan dalam pengembangan arsitektur *Layer 2* yang tahan terhadap *Byzantine fault*.

2.2.7.3 Sidechains

Sidechains adalah *blockchain* terpisah yang berjalan paralel dengan *blockchain* utama dan dihubungkan melalui mekanisme *bridge*. Tidak seperti *rollups*, yang tetap bergantung pada keamanan *Layer 1*, *sidechain* menjalankan mekanisme konsensus dan eksekusi mereka sendiri secara independen. Hal ini memungkinkan *sidechain* untuk menggunakan parameter jaringan yang berbeda, seperti waktu blok yang lebih cepat atau model biaya transaksi yang lebih ringan, tergantung pada kebutuhan aplikasi tertentu [19].

Dalam praktiknya, interaksi antara *blockchain* utama dan *sidechain* dilakukan melalui sistem penguncian dan pencerminan aset. Misalnya, pengguna dapat mengunci token mereka di *smart contract* pada *Layer 1*, dan menerima representasi token yang ekuivalen di *sidechain*. Proses ini biasanya diatur melalui *bridge* yang dapat berupa sistem terdesentralisasi atau bahkan layanan tersentralisasi tergantung pada desain protokolnya.

Keunggulan utama dari *sidechain* adalah fleksibilitas dan skalabilitas yang dapat disesuaikan. Karena *sidechain* memiliki aturan konsensus sendiri, pengembang dapat mengoptimalkan *throughput*, biaya transaksi, atau bahkan menerapkan eksperimen fitur tanpa membebani jaringan *blockchain* utama. Selain itu, pendekatan ini memungkinkan pembuatan rantai aplikasi khusus (*application-specific chains*) yang dapat dioptimalkan untuk *use-case* tertentu.

Namun, karena *sidechain* tidak mewarisi keamanan dari *blockchain* utama, maka sistem ini memperkenalkan asumsi kepercayaan tambahan terhadap pihak-pihak yang mengoperasikan konsensus dan *bridge*. Jika validator atau mekanisme konsensus pada *sidechain* dikendalikan oleh aktor jahat atau bertindak secara kolusif, maka aset pengguna

yang dipindahkan dari *Layer 1* ke *sidechain* melalui *bridge* bisa dicuri atau dimanipulasi. Risiko ini menjadi lebih signifikan karena dalam beberapa tahun terakhir, banyak serangan terhadap infrastruktur *blockchain* justru terjadi akibat kerentanan pada *bridge* dan bukan pada protokol konsensusnya.

Contoh implementasi *sidechain* yang populer adalah Polygon PoS, yang beroperasi sebagai *sidechain* dari Ethereum dengan konsensus berbasis *Proof of Stake*, dan Gnosis Chain (sebelumnya xDai), yang dirancang untuk pembayaran *low-cost* dan cepat. Keduanya menawarkan kompatibilitas dengan Ethereum Virtual Machine (EVM) dan sering digunakan untuk mengurangi biaya transaksi yang dibuat oleh *end-user* dalam aplikasi.

2.2.8 Analisis Perbandingan Metode

Bagian ini menjelaskan terkait perbandingan antara metode atau teknologi yang digunakan dalam pengembangan sistem *web service Byzantine fault-tolerant* yang responsif dan interaktif. Berikut adalah analisis terkait perbandingan metode yang digunakan dalam penelitian tugas akhir ini.

2.2.8.1 Perbandingan Platform *Blockchain* dan Konsensus

Pengembangan sistem *web service* yang responsif dan memberikan jaminan *Byzantine fault tolerance* memerlukan pemilihan platform *blockchain* dan algoritma konsensus yang tidak hanya aman, tetapi juga mendukung fleksibilitas arsitektur dan efisiensi operasional. Beberapa platform telah dipertimbangkan berdasarkan kriteria seperti dukungan terhadap BFT, kustomisasi, kematangan ekosistem, serta kompatibilitas dengan kebutuhan sistem *state machine replication* (SMR) yang digunakan dalam penelitian ini.

Hyperledger Fabric merupakan platform *permissioned blockchain* yang dikembangkan oleh Linux Foundation untuk kebutuhan enterprise dengan modularitas tinggi. Meskipun menawarkan fleksibilitas dalam konfigurasi komponen dan identitas peserta, Fabric tidak dirancang untuk menoleransi *Byzantine fault* secara eksplisit dan lebih menekankan pada kontrol administratif di lingkungan tertutup sambil menggunakan algoritma CFT seperti Raft [43].

Ethereum adalah platform *blockchain* publik yang bersifat *permissionless* dan dirancang untuk mendukung *smart contract* serta aplikasi terdesentralisasi (*dApps*) dalam ekosistem terbuka [44]. Platform ini memiliki komunitas yang luas, populer, dan berbasis algoritma *Proof-of-Stake* (PoS). Ekosistemnya sangat aktif, dengan dokumentasi dan komunitas pengembang yang luas. Namun, konsensus PoS Ethereum tidak menyediakan replikasi deterministik seperti pada sistem SMR, dan tingkat kustomisasinya terbatas pada pengembangan *smart contract*.

Tendermint Core adalah platform konsensus berbasis algoritma BFT dan SMR yang menyediakan finalitas deterministik dan latensi rendah [45]. Platform ini memisahkan logika aplikasi dari *consensus engine* melalui antarmuka *Application Blockchain Interface* (ABCI), memungkinkan pengembang untuk membangun aplikasi deterministik di atas jaringan replikasi BFT. Pengembangan resmi Tendermint Core telah dihentikan dan proyek ini kemudian dilanjutkan oleh *Informal Systems* melalui *public fork* yang menjadi dasar bagi CometBFT sebagai penerus resminya [46].

CometBFT merupakan penerus dan pengganti dari Tendermint Core, yang sejak awal tahun 2023 telah dihentikan pengembangannya. CometBFT kini dilanjutkan sebagai proyek utama dalam ekosistem *blockchain* Cosmos dan telah menjadi *consensus engine* bawaan dalam Cosmos SDK [46–48]. CometBFT mempertahankan sifat replikasi deterministik, dukungan penuh terhadap konsensus BFT, serta pemisahan logika aplikasi melalui *Application Blockchain Interface* (ABCI). Platform ini menyediakan kemampuan kustomisasi tinggi dan dokumentasi yang semakin berkembang, menjadikannya pilihan yang relevan untuk membangun sistem *web service* yang membutuhkan performa dan ketahanan tinggi.

Tabel 2.2. Perbandingan platform *blockchain* dan konsensus.

Parameter	Hyperledger Fabric	Ethereum	Tendermint Core	CometBFT
Kustomisasi	Sangat tinggi, dapat disesuaikan dengan kebutuhan spesifik	Rendah, terbatas pada <i>smart contracts</i>	Tinggi, konsensus dapat dikustomisasi melalui <i>interface</i>	Tinggi, warisan dari Tendermint Core selaku penerusnya
Sumber Pembelajaran	Terbatas, dokumentasi kurang lengkap	Melimpah, komunitas besar dan aktif	Cukup tersedia namun sudah <i>deprecated</i>	Tersedia dan berkembang
Dukungan BFT	Mengutamakan CFT dalam lingkungan terkontrol	Menggunakan <i>Proof-of-Stake</i> , bukan BFT murni	Mendukung konsensus BFT penuh berbasis algoritma Tendermint	Mendukung konsensus BFT penuh berbasis algoritma Tendermint
Status Pengembangan	Aktif dikembangkan	Aktif dikembangkan	<i>Deprecated</i> , tidak lagi didukung	Aktif dikembangkan
Komunitas	Terbatas	Sangat besar dan aktif	Terbatas karena <i>deprecated</i>	Berkembang dalam ekosistem <i>blockchain</i> Cosmos

Berdasarkan perbandingan pada Tabel 2.2, **CometBFT** dipilih sebagai platform konsensus karena aktif dikembangkan dan menyediakan dukungan penuh terhadap konsensus BFT deterministik dan SMR. Tingkat kustomisasi yang tinggi serta pemisahan logika aplikasi dan konsensus melalui ABCI menjadikannya sangat sesuai untuk kebutuhan sistem *web service* interaktif berbasis SMR. Selain itu, CometBFT adalah penerus dari Tendermint Core yang digunakan pada sistem *state-of-the-art* seperti DeWS [14], sehingga memungkinkan perbandingan yang relevan dalam konteks penelitian ini.

2.2.8.2 Perbandingan Solusi *Scalability* Sistem

Untuk mengatasi tantangan skalabilitas sistem *web service* BFT, beberapa pendekatan arsitektur telah dikaji dan dibandingkan berdasarkan tingkat

kompatibilitasnya dengan model *State Machine Replication* (SMR). Tiga pendekatan utama yang dievaluasi adalah *blockchain sharding*, *rollups*, dan *sidechains*. Ketiga pendekatan tersebut telah dibahas secara rinci pada Subbagian 2.2.7, dan perbandingan berikut menyoroti implikasi arsitekturalnya terhadap integrasi dengan CometBFT serta kesesuaiannya dalam konteks sistem BFT yang responsif.

Tabel 2.3. Perbandingan solusi *scalability* sistem.

Parameter	Blockchain Sharding	Rollups	Sidechains
Kompleksitas Implementasi	Sangat tinggi	Tinggi	Sedang
Kompatibilitas dengan BFT dan SMR	Rendah, perlu banyak penyesuaian terhadap proses konsensus	Tidak kompatibel dengan SMR	Sedang
Dukungan CometBFT	Tidak didukung	Tidak kompatibel	Tidak didukung secara langsung, tapi dapat dikustom secara manual
Keamanan	Berkurang dengan fragmentasi	Tinggi dengan ZKP	Bergantung pada <i>bridge security</i>
Throughput	Tinggi	Sangat tinggi	Tinggi
Arsitektur	<i>Horizontal partitioning</i>	<i>Layered execution</i>	<i>Parallel chains</i>
Mekanisme <i>Batching</i>	Tidak ada	<i>Bundling</i> transaksi	<i>Per-chain batching</i>
<i>Maturity</i>	Masih dalam penelitian	<i>Mature</i>	<i>Mature</i>

Berdasarkan perbandingan pada Tabel 2.3, tidak ada solusi skalabilitas yang secara langsung dirancang untuk sistem BFT berbasis *State Machine Replication* (SMR). Namun, di antara ketiganya, *sidechains* menunjukkan pendekatan yang paling kompatibel. *Blockchain sharding* memiliki kompleksitas sangat tinggi dan tidak didukung secara langsung oleh CometBFT. Sementara itu, *rollups* tidak selaras dengan prinsip replikasi deterministik karena bergantung pada verifikasi eksternal seperti *Zero-Knowledge Proof* (ZKP). Untuk mengakomodasi *rollups*, CometBFT perlu digantikan oleh hasil *fork*-nya yang memang dirancang untuk mendukung mekanisme

rollup, yaitu Rollkit [49], yang tidak berbasis SMR. Dalam konteks ini, *sidechains* menawarkan titik tengah antara fleksibilitas dan integrasi potensial dengan BFT, meskipun tetap memerlukan penyesuaian arsitektural.

Oleh karena itu, penelitian ini mengembangkan solusi yang mengadaptasi elemen-elemen dari *sidechains* dan *rollups* dalam bentuk arsitektur *multi-layer* untuk sistem BFT. Seperti *sidechains*, pendekatan ini memanfaatkan pemrosesan cepat pada *layer* terpisah dan mekanisme *batching* untuk mengurangi *overhead* konsensus pada *layer* utama. Di saat yang sama, inspirasi dari *rollups* digunakan untuk efisiensi kompresi hasil eksekusi. Namun, pendekatan *multi-layer* ini menghadirkan tantangan dalam sinkronisasi *state* antarlayer serta mempertahankan properti ACID (*Atomicity*, *Consistency*, *Isolation*, *Durability*) dalam konteks terdistribusi. Untuk mengatasi tantangan tersebut, solusi ini juga memperkenalkan mekanisme pengelolaan sesi (*session management*) yang mengelompokkan operasi terkait dan memprosesnya secara atomik, memastikan konsistensi *state* antarlayer, menjaga jaminan keamanan BFT, dan tetap memberikan ketanggapan yang dibutuhkan dalam proses *web service* yang bersifat krusial.

2.2.8.3 Perbandingan Bahasa Pemrograman Web Service

Pemilihan bahasa pemrograman dalam pengembangan sistem *web service* BFT mempertimbangkan beberapa aspek penting, antara lain performa, dukungan terhadap *concurrency*, integrasi dengan platform CometBFT, serta ekosistem dan tingkat familiaritas di kalangan pengembang. Dua bahasa yang dievaluasi dalam konteks ini adalah JavaScript dan Go.

Go atau **Golang** merupakan bahasa pemrograman yang dirancang oleh Google dengan fokus pada efisiensi sistem, kesederhanaan sintaks, dan mendukung *concurrency* yang tinggi. Melalui penggunaan *goroutines* dan *channel*, Go memungkinkan pengembangan sistem paralel secara ringan dan efisien tanpa kompleksitas manajemen *thread* secara manual. Go juga memiliki integrasi langsung dengan CometBFT dan banyak digunakan dalam pengembangan sistem terdistribusi dan layanan backend performa tinggi. Karakteristik ini menjadikan Go cocok untuk membangun sistem replikasi deterministik dengan tuntutan *throughput* tinggi.

JavaScript, di sisi lain, adalah bahasa pemrograman yang telah lama digunakan dalam pengembangan antarmuka pengguna dan *web service* berbasis REST. Bahasa ini memiliki ekosistem pustaka dan komunitas yang sangat besar serta kemudahan adopsi oleh pengembang. Namun, model *event-driven* dan eksekusi *single-threaded*-nya menyebabkan keterbatasan dalam pemrosesan paralel dan performa pada skala sistem replikasi. JavaScript juga memerlukan *wrapper* atau *binding* khusus untuk dapat berinteraksi dengan CometBFT, sehingga menambah kompleksitas dalam integrasi.

Tabel 2.4. Perbandingan bahasa pemrograman *web service*.

Parameter	JavaScript	Go
Performa	Rendah untuk aplikasi <i>high-throughput</i>	Tinggi, cocok untuk aplikasi <i>concurrent</i>
Familiaritas	Tinggi, sudah familiar	Lebih rendah, perlu pembelajaran
Integrasi dengan CometBFT	Memerlukan <i>wrapper</i> atau <i>binding</i>	<i>Native support</i> , antarmuka langsung
Ekosistem	Besar, terutama untuk <i>web development</i>	Berkembang untuk sistem terdistribusi
<i>Concurrency</i>	Terbatas pada <i>single-thread</i> dengan <i>event loop</i>	Sangat baik dengan menggunakan <i>goroutines</i>

Berdasarkan perbandingan pada Tabel 2.4, **Go** dipilih sebagai bahasa pemrograman utama karena menyediakan performa yang tinggi, dukungan *native* terhadap CometBFT melalui antarmuka langsung, serta model *concurrency* yang efisien menggunakan *goroutines*. Meskipun JavaScript lebih familiar dan memiliki ekosistem besar dalam pengembangan *web*, keterbatasannya dalam performa dan integrasi dengan sistem replikasi BFT menjadikannya kurang sesuai untuk kebutuhan sistem *mission-critical* yang bersifat *high-throughput* dan toleran terhadap *fault*.

BAB 3

METODE PENELITIAN

Bab ini menjelaskan metode atau cara yang digunakan dalam penelitian ini untuk mencapai tujuan dalam mengembangkan sistem *web service* BFT dengan arsitektur dua *layer* yang memisahkan operasi interaktif dari finalisasi konsensus untuk mengeliminasi permasalahan terkait ketanggapan sistem yang mendukung skenario dunia nyata dalam aplikasi terdesentralisasi yang memerlukan *Byzantine fault tolerance* dan interaksi *real-time*.

3.1 Alat dan Bahan Tugas akhir

Peralatan dan bahan yang digunakan dalam penelitian tugas akhir ini dijabarkan pada Subbagian 3.1.1 mengenai Alat Tugas Akhir serta Subbagian 3.1.2 yang membahas Bahan Tugas Akhir.

3.1.1 Alat Tugas akhir

Sebagai bentuk pemenuhan kebutuhan dalam pengembangan tugas akhir ini, diperlukan sejumlah alat yang terdiri dari perangkat keras dan perangkat lunak berikut:

1. Laptop dengan spesifikasi sistem operasi Windows 11, *processor* 13th Gen Intel(R) Core(TM) i7-13650HX @ 2.60 GHz, memori 16GB DDR4, sistem 64-bit operating system dengan *x64-based processor*, dan penyimpanan SSD 512GB.
2. CometBFT sebagai *consensus engine* untuk implementasi *Byzantine fault tolerance* menggunakan algoritma Tendermint dengan *Application BlockChain Interface* (ABCI) yang dapat dikustomisasi.
3. Visual Studio Code versi 1.100.2 sebagai *code editor* utama untuk pengembangan sistem *web service*.
4. Go *compiler* dan *runtime environment* untuk pengembangan aplikasi yang kompatibel dengan CometBFT *native codebase*.
5. Docker sebagai *containerization platform* untuk deployment dan testing sistem BFT dalam lingkungan terisolasi yang konsisten.
6. Postman sebagai *API testing tool* untuk pengujian dan validasi fungsionalitas *web service* dalam sistem BFT.
7. Windows Subsystem for Linux (WSL) untuk lingkungan pengembangan yang mendukung *tools* berbasis Unix.
8. Zotero sebagai *reference management software* untuk pengelolaan literatur dan

dokumentasi penelitian.

9. Draw.io untuk pengembangan diagram arsitektur sistem dan *Unified Modeling Language* (UML) dari aplikasi *web service* BFT secara keseluruhan.

3.1.2 Bahan Tugas akhir

Bahan yang digunakan dalam penelitian tugas akhir ini adalah sebagai berikut:

1. Dokumen. Dokumen yang digunakan merupakan dokumen digital, mencakup penelitian sebelumnya, proyek aplikasi yang telah dibuat, serta dokumen lain yang berkaitan dengan topik-topik berikut.
 - Sistem *web service* dengan *Byzantine fault tolerance* dan tantangan ketanggapan dalam aplikasi interaktif.
 - Arsitektur berlapis (*layered*) untuk memisahkan operasi interaktif dari finalisasi konsensus dalam sistem terdistribusi.
 - Implementasi CometBFT dan algoritma Tendermint untuk *consensus engine* dalam aplikasi *web service*.
 - Teknologi *blockchain* dan implementasinya dalam sistem terdistribusi untuk aplikasi terdesentralisasi.
 - Solusi *scalability* dan *throughput* pada teknologi *blockchain*.
 - Algoritma konsensus dalam sistem terdistribusi, khususnya mekanisme *Byzantine fault tolerance*.
 - Arsitektur dan protokol sistem terdistribusi untuk aplikasi yang memerlukan konsistensi dan ketersediaan tinggi.
 - Aplikasi *metaverse* dan kebutuhan ketanggapan *real-time* dalam lingkungan virtual.
 - Prinsip Web3 dan pengembangan aplikasi terdesentralisasi dengan arsitektur *peer-to-peer*.
 - Metode evaluasi performa dan latensi dalam sistem konsensus terdistribusi.
2. Data. Data yang dimanfaatkan dalam penelitian tugas akhir ini terdiri dari informasi serta gambar atau diagram yang relevan dengan kebutuhan pengembangan aplikasi tugas akhir ini.

3.2 Metode Penelitian

Penelitian tugas akhir ini berupa pengembangan sistem *web service* dengan *Byzantine fault tolerance* (BFT) yang menggunakan arsitektur dua *layer* dan *sessioning*

untuk mengatasi *trade-off* fundamental antara keamanan dan ketangguhan dalam aplikasi interaktif. Pengembangan sistem ini dibagi menjadi dua komponen utama: Layer 1 (L1) yang mengimplementasikan konsensus BFT penuh untuk jaminan keamanan, dan Layer 2 (L2) yang berfungsi sebagai *session-aware transaction buffer* untuk memberikan *feedback* langsung kepada pengguna. Sistem dikembangkan menggunakan CometBFT sebagai *consensus engine* dengan Go sebagai bahasa pemrograman utama, dan dievaluasi melalui implementasi *proof-of-concept* sistem manajemen *supply chain* yang memerlukan ketangguhan tinggi dan jaminan keamanan yang kuat dalam *workflow multi-step*.

3.2.1 Metode Pengembangan

Berdasarkan analisis perbandingan metode yang telah dilakukan pada Bagian 2.2.8, pengembangan sistem *web service* BFT menggunakan arsitektur dua *layer* dengan pendekatan yang memisahkan operasi interaktif dan finalisasi konsensus. Pemilihan CometBFT sebagai *consensus engine* dan bahasa pemrograman Go didasarkan pada keunggulan performa, integrasi *native*, dan kompatibilitas optimal untuk sistem BFT yang memerlukan *throughput* tinggi.

Layer 1 mengimplementasi konsensus BFT penuh menggunakan CometBFT dengan algoritma Tendermint, memerlukan minimum 4 *node* untuk menolerir 1 *node* Byzantine sesuai formula $n = 3f + 1$. Layer ini menyediakan *immutable blockchain ledger* dan jaminan keamanan untuk transaksi final. Layer 2 dikembangkan sebagai *transaction buffer* yang melakukan simulasi konsensus dengan data *state* terkini, memungkinkan *feedback* langsung dalam waktu singkat sambil mempertahankan aturan aplikasi yang konsisten.

Implementasi teknis menggunakan CometBFT sebagai *consensus engine* dengan *Application BlockChain Interface* (ABCI) yang dapat dikustomisasi sesuai kebutuhan. Komponen sistem dikembangkan dalam bahasa pemrograman Go untuk kompatibilitas optimal dengan CometBFT *native codebase*, meliputi *web server*, *service registry* untuk mapping API *routes* ke *service handlers*, dan *database* Postgresql untuk penyimpanan *session data*. Sistem mengimplementasikan *session-based transaction buffering* yang mengelompokkan operasi terkait menjadi unit logis, di mana setiap sesi mempertahankan *state* konsisten dalam *workflow* sebelum dilakukan *commit* atomik ke L1.

Proof-of-concept dikembangkan menggunakan skenario sistem manajemen *supply chain* yang merepresentasikan *use case* ideal untuk kebutuhan ketangguhan tinggi dan jaminan keamanan yang kuat. *Workflow* terdiri dari lima tahap berurutan: inisiasi *session*, pemindaian paket, validasi *digital signature* dari pemasok, inspeksi *quality control*, dan proses pelabelan paket. Implementasi ini mendemonstrasikan bagaimana arsitektur dua *layer* dapat memberikan pengalaman pengguna yang responsif untuk

operasi interaktif sambil menjamin *tamper-proof audit trail* melalui BFT consensus dan *blockchain*.

3.2.2 Metode Pengujian dan Evaluasi

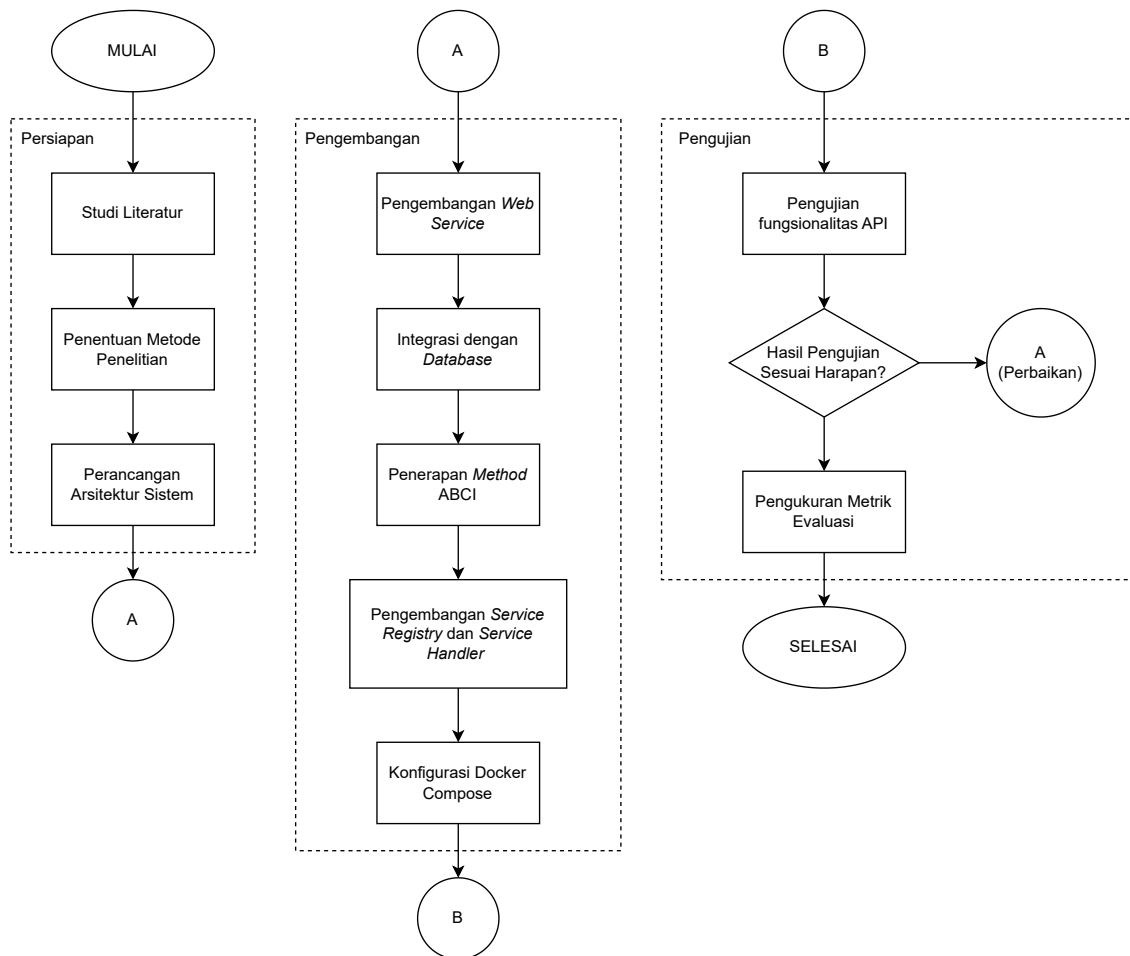
Evaluasi sistem dilakukan melalui *comprehensive performance benchmarking* yang berfokus pada metrik latensi sebagai indikator utama peningkatan ketanggapan. Latensi didefinisikan sebagai waktu dari inisiasi *request* hingga diterimanya *response*, didefinisikan secara formal sebagai $t_{res} - t_{req}$. Pengujian dilakukan secara sistematis dalam variasi konfigurasi L1 dengan *nodes* berjumlah 4, 7, 10, 13, dan 16. Jumlah *node* yang dipilih merepresentasikan BFT *thresholds* dari 1 hingga 5. Sementara itu, untuk konfigurasi *nodes* pada L2 dipilih 1 dan 2. Setiap konfigurasi diuji dengan 100 iterasi untuk memastikan signifikansi statistik.

Benchmarking diimplementasikan dengan aplikasi *standalone* menggunakan Go yang merekam metrik performa dalam format CSV, kemudian diproses menggunakan Python untuk visualisasi. Pengujian meliputi detail performa *endpoint* untuk setiap tahap dalam *workflow*, waktu keseluruhan operasi *workflow*, dan perbandingan antara latensi operasi di L2 dengan latensi operasi *commit* di L1. Evaluasi juga membandingkan hasil dengan arsitektur *web service* BFT *state-of-the-art*, yaitu DeWS [14], untuk menunjukkan peningkatan ketanggapan sambil mempertahankan integritas transaksi.

Deployment sistem menggunakan konfigurasi kontainer Docker untuk *environment* testing yang konsisten dan *reproducible*. Evaluasi dilakukan dalam lingkungan terkontrol untuk meminimalkan faktor eksternal yang dapat mempengaruhi hasil pengukuran latensi. Hasil evaluasi dianalisis untuk membuktikan bahwa arsitektur dua *layer* dapat secara efektif menyelesaikan *trade-off* fundamental antara properti BFT dan ketanggapan, membuka kemungkinan aplikasi BFT *web services* dalam domain yang sebelumnya tidak praktis karena keterbatasan dalam latensi.

3.3 Alur Tugas Akhir

Alur penelitian tugas akhir tentang pengembangan arsitektur dua *layer* untuk sistem BFT yang responsif dan interaktif ini mengikuti pendekatan sistematis yang terbagi menjadi tiga fase utama sebagaimana ditunjukkan dalam Gambar 3.10. Fase persiapan mencakup studi literatur dan penentuan metode penelitian untuk membangun fondasi teoretis dalam mengembangkan arsitektur *session-based consensus decoupling*. Fase pengembangan berfokus pada implementasi sistem BFT dua *layer*, mengintegrasikan komponen *web service* dengan sistem *database*, dan implementasi konsensus menggunakan ABCI untuk kustomisasinya. Terakhir, fase pengujian melibatkan evaluasi komprehensif terhadap fungsionalitas API dan metrik kinerja untuk memvalidasi efektivitas solusi yang diusulkan dalam mengatasi tantangan latensi sambil



Gambar 3.10. Alur tugas akhir yang dibagi menjadi tiga tahapan utama, Persiapan, Pengembangan, dan Pengujian.

tetap mempertahankan jaminan BFT. Dalam bagian ini, akan dijelaskan tentang setiap fase dan setiap proses yang ada dalam tiap fasenya.

3.3.1 Fase Persiapan

Fase persiapan merupakan fase paling awal dalam penelitian ini yang bertujuan untuk membangun landasan teoretis dan metodologis yang kuat. Fase ini terdiri dari dua tahapan utama yaitu studi literatur untuk memahami *state-of-the-art* dalam sistem BFT dan *web services*, serta penentuan metode penelitian yang akan digunakan.

3.3.1.1 Studi Literatur

Tahapan studi literatur dilakukan untuk mengidentifikasi permasalahan fundamental dalam sistem BFT tradisional, khususnya *trade-off* antara keamanan dan ketanggapan dalam domain *web services* yang memerlukan interaktivitas. Studi ini mencakup analisis mendalam terhadap penelitian terkait seperti DeWS [14] dan pendekatan *layer 2* pada teknologi *blockchain* untuk memahami keterbatasan sistem

yang ada dan peluang pengembangan arsitektur baru yang dapat mengatasi tantangan latensi tanpa mengorbankan jaminan BFT.

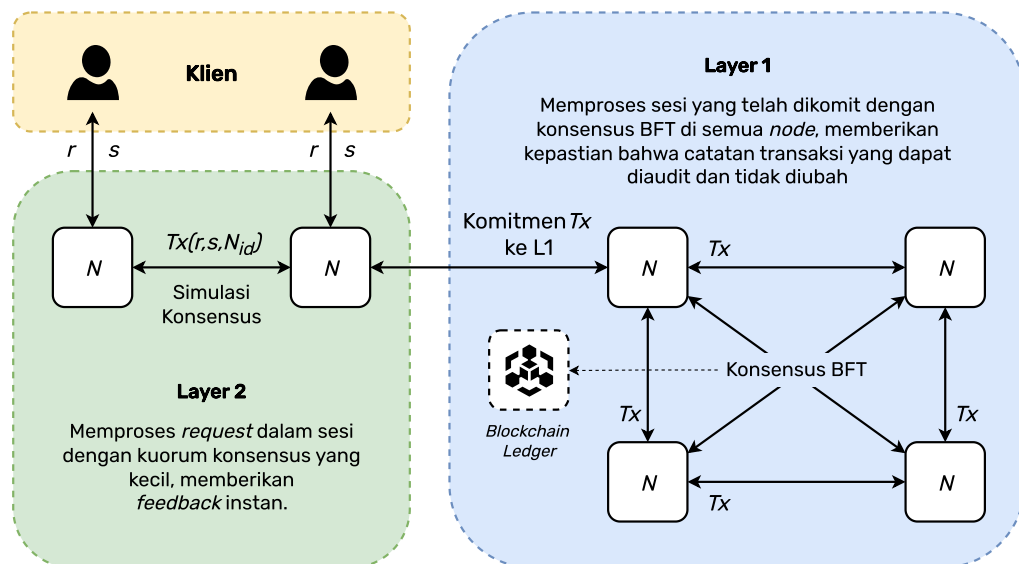
3.3.1.2 Penentuan Metode Penelitian

Berdasarkan hasil studi literatur, tahapan ini melakukan pemilihan dan penetapan metodologi pengembangan yang sesuai dengan karakteristik sistem BFT dua *layer*. Seperti dijelaskan pada Subbagian 2.2.8, Penentuan metode mencakup pemilihan teknologi implementasi seperti CometBFT sebagai *consensus engine*, bahasa pemrograman Go untuk pengembangan *web service*, serta kerangka kerja evaluasi yang akan digunakan untuk mengukur performa sistem dalam hal latensi.

3.3.1.3 Perancangan Arsitektur Sistem

Setelah menentukan metode yang dipilih, tahapan perancangan arsitektur sistem dilakukan untuk merancang desain teknis. Perancangan arsitektur mencakup definisi komponen dan proses dalam sistem, interaksi antar *layer*, serta alur kerja keseluruhan sistem yang akan diimplementasikan.

Berdasarkan studi literatur dan perbandingan metode yang telah dipaparkan di Bab-Bab sebelumnya, dibentuk arsitektur yang memisahkan proses interaktif dan jaminan keamanan BFT dengan cara membagi sistem menjadi 2 *layer* seperti yang ditunjukkan pada Gambar 3.11.



Gambar 3.11. Arsitektur dua *layer* untuk memisahkan operasi interaktif dan operasi dengan keamanan BFT.

Untuk merealisasikan arsitektur ini, sistem dibangun dari sejumlah komponen inti yang tersebar di kedua *layer* dan bekerja secara terkoordinasi. Komponen-komponen ini dirancang untuk menjalankan tanggung jawab fungsional seperti pemrosesan *request*,

pemetaan rute API, validasi transaksi, dan pelaksanaan konsensus. Penjelasan berikut menguraikan fungsi dan peran masing-masing komponen dalam mendukung operasional sistem dua *layer* ini.

- **Node (N):** komponen fundamental dari jaringan terdistribusi. Dalam sistem ini, sebuah *node* terdiri dari *web server*, *consensus engine*, dan *database*. *Node-node* ini saling terhubung membentuk sebuah jaringan pada setiap *layer*.
- **Service Handler:** Fungsi yang memproses tipe HTTP *request* spesifik berdasarkan *method* dan *path*-nya. Secara formal, untuk request r , *service handler* merupakan fungsi $H : R \rightarrow S$ yang memetakan dari domain *request* R ke domain *response* S .
- **Service Registry:** Komponen manajemen yang bertugas mempertahankan pemetaan antara *API routes* (kombinasi HTTP *method* dan *path*) dengan *service handler* yang sesuai. Pemetaan ini dapat direpresentasikan sebagai $R = \{(m_i, p_i) \mapsto H_i\}$, dengan fungsi pencarian $L(m, p) \rightarrow H$ untuk menentukan *handler* berdasarkan metode dan path tertentu.
- **Transaction Originator:** *Node* yang pertama kali menerima *request* dari klien. *Transaction originator* memproses *request* menggunakan *service handler* yang sesuai dan membuat transaksi awal dengan identifier N_{id} . *Node* ini bertanggung jawab untuk mem-broadcast transaksi ke *node* lain dalam jaringan sehingga konsensus bisa dilakukan.
- **Transaction (Tx):** Sebuah struktur data yang direpresentasikan sebagai $Tx = (r, s, N_{id})$, mencakup *request* r , *response* s yang dihasilkan oleh *service handler*, serta identifier N_{id} dari *node* asal transaksi tersebut. Transaksi merupakan unit fundamental dalam konsensus yang memungkinkan *node-node* untuk memverifikasi apakah suatu *request* menghasilkan *response* yang diharapkan sesuai dengan aturan bisnis yang diprogram dalam *service handler*. Suatu transaksi dianggap valid ketika memenuhi dua kondisi: (1) *response* dihasilkan dengan menerapkan *service handler* yang benar terhadap request, dan (2) mayoritas *node* dalam jaringan telah memverifikasi secara independen bahwa penerapan *service handler* yang sama terhadap *request* menghasilkan *response* yang identik. Proses verifikasi ini memastikan bahwa *node Byzantine* tidak dapat memanipulasi hasil transaksi, sehingga transaksi yang tidak memenuhi kriteria tersebut akan ditandai sebagai invalid.
- **BFT Consensus:** Sebuah proses untuk mencapai kesepakatan atas *state* atau suatu keputusan dalam sistem terdistribusi yang mampu menoleransi

Byzantine faults. Konsensus BFT membutuhkan minimal 4 *node* dan dinyatakan mencapai *agreement* apabila setidaknya $\lceil \frac{2n+1}{3} \rceil$ *node* memvalidasi transaksi, sehingga memastikan keabsahan transaksi bahkan jika hingga $\lfloor \frac{n-1}{3} \rfloor$ *node* bersifat *Byzantine*. Setelah consensus tercapai, hasil transaksi dikomit ke *blockchain ledger* yang bersifat *immutable*, memberikan jaminan *finality* yang kuat dan memungkinkan audit yang dapat diverifikasi untuk semua operasi yang telah disetujui oleh kuorum.

- **Simulation Consensus:** Sebuah proses persetujuan ringan pada *Layer 2* yang memprioritaskan ketanggapannya dibandingkan dengan keamanan dan jaminan BFT. Mekanisme ini meniru logika validasi dari konsensus BFT utama, namun beroperasi dengan persyaratan partisipasi yang lebih longgar untuk menghasilkan respons interaktif dengan latensi rendah. Konsep ini merupakan inovasi utama dalam penelitian ini, karena memungkinkan pemisahan antara operasi interaktif dan proses finalisasi konsensus. Dengan demikian, sistem dapat memberikan *feedback* instan kepada pengguna sambil tetap mempertahankan integritas melalui komitmen ke *Layer 1*.

Arsitektur dua *layer* ini dirancang untuk mengatasi keterbatasan latensi yang ditemukan dalam sistem DeWS yang memerlukan waktu sekitar 935ms untuk setiap operasi pada konfigurasi 15 *node* [14]. Pemisahan ini didasarkan pada prinsip bahwa tidak semua operasi memerlukan tingkat keamanan BFT yang sama, namun semua operasi memerlukan ketanggapannya yang tinggi untuk pengalaman pengguna yang baik. *Layer 1* (L1) sebagai *layer* fondasi berfungsi sebagai penyedia keamanan BFT dengan mengimplementasikan protokol konsensus BFT penuh. Setiap *node* di L1 mempertahankan *service registry* yang identik dan hanya melakukan konsensus ketika menerima *request* komit dari *node* L2. Setiap validator secara independen memverifikasi validitas transaksi dengan menerapkan *service handler* yang sama seperti *transaction originator* dan membandingkan respons yang dihasilkan.

Layer 2 (L2) sebagai *Interactive Transaction Layer* merupakan inovasi inti dari arsitektur ini yang berfungsi sebagai *transaction buffer*. L2 melakukan simulasi konsensus dan *batching* operasi untuk komitmen yang efisien ke L1. Setiap *node* di L2 menggunakan *service registry* dan *service handler* yang sama baik di komponen *web server* maupun konsensus. Berbeda dengan L1, layer ini tidak diwajibkan memenuhi persyaratan jumlah *node* BFT yang ketat karena mewarisi properti BFT dari jaringan L1 yang mendasarinya. Untuk menjaga konsistensi *state* antara kedua *layer*, sistem mengimplementasikan mekanisme *session-based transaction buffering*. Konsep sesi dalam konteks ini adalah pengelompokan operasi yang saling terkait ke dalam unit logis yang diproses melalui simulasi responsif L2 sebelum dikomit sebagai *batch* per sesinya ke konsensus L1 yang lebih aman namun lambat. Hal ini juga berfungsi untuk mencegah

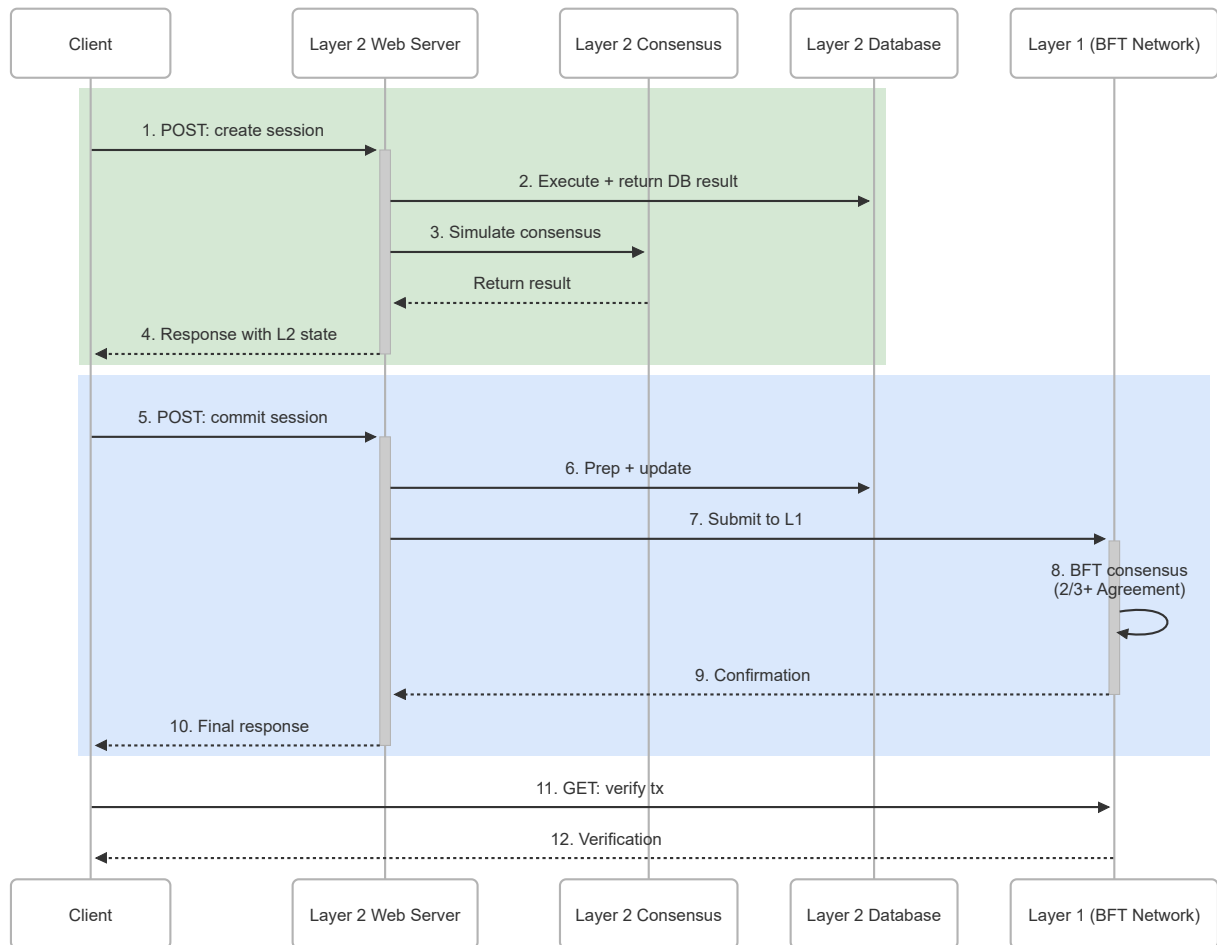
terjadinya konflik *state* ketika terdapat pemrosesan yang terjadi dalam waktu bersamaan.

Mekanisme dalam arsitektur ini bekerja dalam tiga fase utama. Pertama, fase inisiasi dimana ketika klien mengirimkan *request*, *transaction originator* di L2 memproses *request* menggunakan *service registry* untuk mengidentifikasi *service handler* yang sesuai. Handler memproses *request* secara lokal dan menyimpan sesi dalam database *node*. Kedua, fase simulasi dimana setelah pemrosesan, *service handler* menghasilkan respons, yang akan dikombinasikan dengan *request* dan identifier *node* untuk membentuk transaksi. *Transaction originator* meneruskan transaksi ini ke *layer* konsensus L2 yang melakukan *simulation consensus* dengan mereplikasi operasi di *node* L2 lainnya. Jika hanya ada satu *node* L2, maka *node* akan melakukan validasi sendiri dengan *state* lokalnya. Ketiga, fase komitmen dimana klien telah menyelesaikan serangkaian operasi dalam sesi yang sama, seluruh sesi akan dikomit sebagai unit atomik ke L1. Proses ini memastikan bahwa meskipun operasi individual memberikan respons instan melalui L2, integritas akhir tetap dijamin melalui konsensus BFT di L1.

Untuk mencegah percabangan (*forking*) *state* antara kedua *layer*, sistem mengimplementasikan protokol sinkronisasi yang memastikan L2 selalu memiliki pandangan *state* yang konsisten dengan L1. Mekanisme *state propagation* memastikan setiap kali terjadi komitmen di L1, perubahan *state* dipropagasi ke semua *node* L2 untuk memastikan simulasi konsensus di L2 beroperasi dengan *state* yang akurat dan terkini.

Arsitektur dua *layer* ini secara khusus mengatasi keterbatasan yang diidentifikasi dalam sistem DeWS melalui peningkatan ketanggapan dengan memisahkan operasi interaktif ke L2 sehingga sistem dapat memberikan respons dengan cepat untuk operasi yang menghadap pengguna secara langsung. Jaminan keamanan tetap terjaga karena meskipun L2 memberikan respons cepat, keamanan BFT tetap terjamin melalui komitmen akhir ke L1. Efisiensi pemanfaatan resource dicapai melalui *batching* operasi dalam sesi yang mengurangi overhead konsensus BFT dengan mengurangi frekuensi operasi L1 yang mahal. Arsitektur ini memungkinkan pemisahan proses yang mengutamakan ketanggapan dari keamanan sehingga sistem dapat di-*scale* secara independen untuk setiap aspek sesuai kebutuhan aplikasi.

Alur kerja sistem dirancang untuk memisahkan operasi interaktif dari proses finalisasi konsensus, sebagaimana diilustrasikan dalam diagram *sequence* pada Gambar 3.12. Layer 2 berfungsi sebagai *transaction buffer* yang memberikan respons interaktif kepada klien melalui *Simulation Consensus*, sementara Layer 1 beroperasi sebagai fondasi BFT yang mengimplementasikan BFT *Consensus* secara penuh untuk memberikan jaminan integritas dan *immutability* terhadap catatan transaksi. Desain ini memungkinkan sistem memberikan pengalaman pengguna yang responsif sambil tetap mempertahankan properti keamanan yang diperlukan dalam lingkungan *Byzantine*.



Gambar 3.12. Diagram *sequence* yang menunjukkan alur sistem, mulai dari interaksi responsif dengan L2 hingga komitmen ke L1.

3.3.2 Fase Pengembangan

Fase pengembangan merupakan tahapan inti penelitian yang mengimplementasikan arsitektur dua *layer* untuk sistem BFT responsif. Fase ini terdiri dari lima tahapan yang saling berkaitan untuk membangun sistem lengkap yang dapat memberikan pengalaman interaktif kepada pengguna sambil mempertahankan jaminan keamanan BFT.

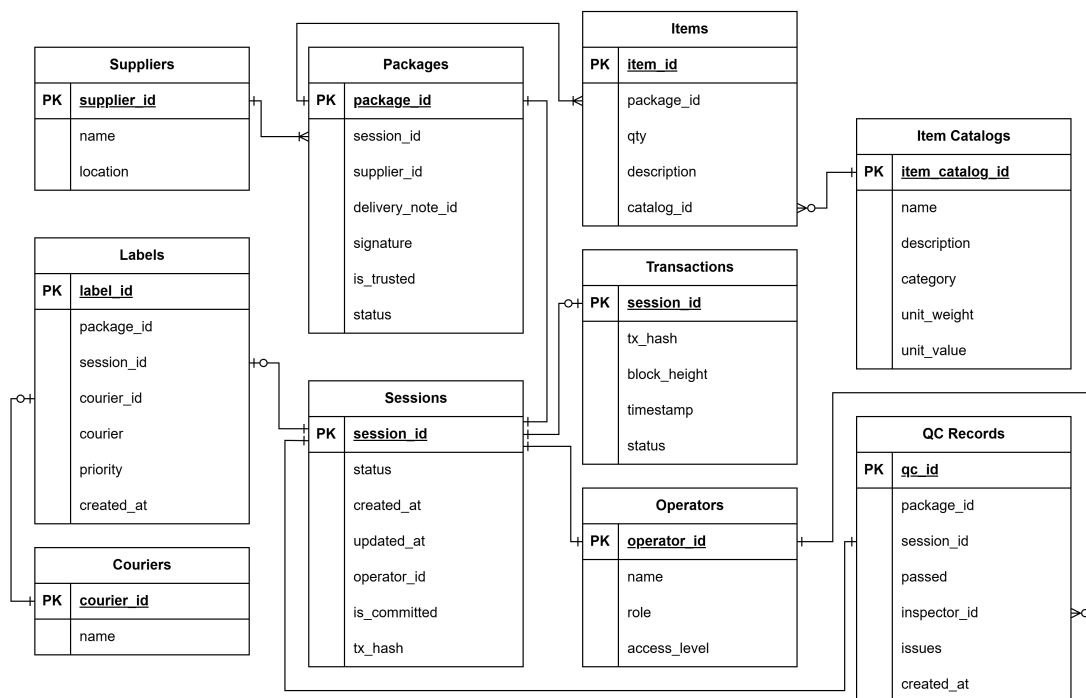
3.3.2.1 Pengembangan *Web Service*

Tahapan ini fokus pada implementasi komponen *web service* yang menyediakan *interface* HTTP untuk interaksi dengan aplikasi klien. Pengembangan dilakukan menggunakan bahasa pemrograman Go dengan memanfaatkan *standard library* `net/http` untuk implementasi HTTP server dan `encoding/json` untuk *serialization* data. Pengembangan mencakup pembuatan program utama dengan mekanisme *routing* menggunakan `http.ServeMux` untuk menangani berbagai

endpoint API, implementasi struktur *repository pattern* untuk abstraksi akses data, serta penggunaan *context package* untuk manajemen *request lifecycle* dan *timeout handling*. Konfigurasi awal melibatkan integrasi dengan CometBFT melalui *method ABCI* yang disediakan, menggunakan `github.com/cometbft/cometbft` *package* untuk komunikasi dengan consensus engine. *Web service* ini dirancang untuk dapat beroperasi pada kedua *layer* sistem dengan logika bisnis yang konsisten.

3.3.2.2 Integrasi dengan Database

Integrasi *database* dilakukan untuk menyediakan penyimpanan persisten bagi data aplikasi dan *session state* dalam sistem *web service* BFT dua layer. Tahapan ini meliputi desain skema *database* yang mendukung operasi transaksional dalam skenario *supply chain shipping* menggunakan PostgreSQL sebagai RDBMS utama, implementasi *connection* dan *object-relational mapping* (ORM) dengan *package* `gorm.io/gorm` untuk abstraksi operasi *database*, serta konfigurasi *connection* menggunakan *driver* `gorm.io/driver/postgres`.



Gambar 3.13. *Entity-relation Diagram* yang didesain untuk skenario *supply chain shipping*.

Seperti yang ditunjukkan pada Gambar 3.13, skema *database* dirancang dengan struktur relasional yang mendukung *workflow supply chain* sederhana. Entitas utama sistem meliputi *Sessions* sebagai satuan kerja atomik yang mengelompokkan operasi terkait dalam satu sesi pemrosesan, *Packages* untuk menyimpan informasi paket yang diproses dalam setiap sesi, dan *Operators* yang mencatat informasi pengguna yang

menjalankan operasi sistem. Entitas *Suppliers* dan *Couriers* menyimpan informasi pihak eksternal yang terlibat dalam *supply chain*, sementara *Items* dan *Item Catalogs* mengelola detail isi konten paket dengan referensinya ke katalog. Entitas *QC Records* mencatat hasil pemeriksaan kualitas untuk setiap paket, *Labels* menyimpan informasi label pengiriman, dan *Transactions* merekam transaksi yang telah di-*commit* ke *Layer 1* BFT dengan informasi *block height* dan *transaction hash*.

Pengembangan data *access layer* mencakup definisi *model structs* dengan GORM untuk *mapping* ke tabel *database*, implementasi *repository pattern* dengan `gorm.DB` untuk enkapsulasi logika *database*, dan penggunaan GORM *transaction support* melalui `db.Tx()` untuk memastikan setiap operasi *session* memenuhi properti ACID. *Database layer* dirancang untuk dapat bekerja dengan kedua *layer* sistem (*Layer 1* untuk BFT *consensus* dan *Layer 2* untuk *simulation consensus*) secara efisien, dengan implementasi *migration system* menggunakan `gorm.AutoMigrate()` untuk *schema versioning* dan konfigurasi `gorm.Session` untuk optimasi performa *query* pada operasi di kedua *layer*. Relasi antar entitas dalam ERD mendukung integritas referensial yang ketat, dengan *foreign key constraints* yang memastikan konsistensi data selama operasi *session-based* dan proses *commit* ke blockchain.

3.3.2.3 Implementasi Konsensus dan ABCI

Application Blockchain Interface (ABCI) diimplementasikan sebagai jembatan penghubung antara aplikasi dan *consensus engine* CometBFT. Tahapan ini mencakup pengembangan *method* ABCI menggunakan *package* `github.com/cometbft/cometbft/abci/types` dengan implementasi *interface* *Application*. *Method* utama yang diimplementasikan meliputi *CheckTx* untuk validasi transaksi yang memverifikasi format dan logika bisnis sebelum transaksi masuk ke *mempool*, *ProcessProposal* untuk replikasi proses yang terdapat pada transaksi di mana setiap *node* mengeksekusi transaksi secara independen dan membandingkan hasilnya untuk mendeteksi perilaku *Byzantine*, dan *FinalizeBlock* untuk finalisasi *state* yang mengkomit perubahan ke penyimpanan persisten. Detail implementasi *method-method* tersebut akan dijelaskan lebih lanjut di bab selanjutnya.

Algoritma yang diimplementasikan mengikuti *Unified Consensus Process* sebagaimana ditunjukkan dalam Algoritma 1, di mana *CheckTransaction* melakukan pengecekan format transaksi dan eksekusi terhadap kondisi *state* terbaru untuk *Layer 1*, sementara untuk *Layer 2* dilakukan pengecekan terhadap *session state* dan logika bisnis. *Method* *ProcessProposal* mengimplementasikan algoritma di mana validator pada *Layer 1* mengeksekusi transaksi secara independen dan membandingkan hasil, sedangkan *Layer 2* melakukan simulasi validasi dengan mengeksekusi operasi terhadap *state* lokal. Implementasi ABCI memungkinkan kustomisasi proses konsensus

Algorithm 1 Unified Consensus Process (Both L1 and L2)

```
1: procedure CHECKTRANSACTION(transaction)
2:   Layer 1: Verify transaction is well-formed and properly signed
3:   Layer 2: Check transaction against session state and business rules
4:   Decrypt and parse transaction content
5:   if transaction format is invalid then
6:     return FALSE ▷ Malformed transaction
7:   end if
8:   if transaction violates state constraints then
9:     return FALSE ▷ Transaction is invalid
10:  end if
11:  Execute transaction against current state
12:  return TRUE ▷ Transaction is valid
13: end procedure
14: procedure PROCESSPROPOSAL(proposedBlock)
15:   Layer 1: Validators independently execute transactions and compare results
16:   Layer 2: Simulates validation by re-executing operations against local state
17:   if results from execution differ from proposed results then
18:     return REJECT ▷ Byzantine behavior detected
19:   else
20:     return ACCEPT ▷ Proposed transactions is valid
21:   end if
22: end procedure
23: procedure FINALIZEBLOCK(acceptedBlock)
24:   Layer 1: Update blockchain state with transaction results
25:   Layer 2: Update session state with L1 consensus results
26:   return Block results with cryptographic proof
27: end procedure
28: procedure COMMIT
29:   Layer 1: Persist block and state changes to blockchain
30:   Layer 2: Mark session as committed with blockchain reference
31:   return Consensus success acknowledgment
32: end procedure
```

dan memastikan bahwa logika aplikasi dapat berinteraksi dengan mekanisme konsensus secara *seamless*.

3.3.2.4 Pengembangan *Service Registry* dan *Service Handler*

Service Registry berfungsi sebagai pusat manajemen routing yang memetakan *endpoint* API ke fungsi *handler* yang sesuai. Pengembangan *Service Handler* meliputi implementasi logika bisnis untuk setiap operasi aplikasi, penanganan *error*, dan logika validasi. *Handler* function dirancang untuk bersifat deterministik, memastikan bahwa input yang sama akan selalu menghasilkan output yang identik di semua *node* dalam jaringan. Kedua komponen ini dirancang untuk memastikan konsistensi pemrosesan antara sisi konsensus dan sisi *web service* dalam arsitektur sistem.

Implementasi *service registry* dilakukan secara terpisah untuk L1 dan L2 sesuai dengan peran dan tanggung jawab masing-masing *layer*. Pada L2, *service registry* menangani operasi bisnis yang detail dan kompleks untuk memberikan respons interaktif kepada pengguna melalui simulasi konsensus. Sementara itu, L1 menggunakan *service registry* yang difokuskan untuk menerima dan memproses *commit* dari L2, dengan melakukan validasi menggunakan logika bisnis yang sama untuk memastikan konsistensi *state* dan integritas transaksi melalui mekanisme konsensus BFT.

3.3.2.5 Konfigurasi Docker Compose

Tahapan terakhir dari fase pengembangan adalah konfigurasi infrastruktur *deployment* menggunakan Docker Compose. Konfigurasi ini mencakup pengaturan *container* untuk aplikasi, database, dan *node-node* pada sistem, pengaturan jaringan antar *container*, dan konfigurasi *environment variables*. Docker Compose memungkinkan *deployment* sistem yang konsisten dan mudah direplikasi untuk berbagai skenario pengujian.

3.3.3 Fase Pengujian

Fase pengujian merupakan tahapan validasi untuk memastikan bahwa sistem yang dikembangkan dapat memenuhi tujuan penelitian dalam mengatasi tantangan latensi sistem BFT. Fase ini terdiri dari dua tahapan utama yang mengevaluasi aspek fungsional dan performa sistem.

3.3.3.1 Pengujian Fungsionalitas API

Pengujian fungsionalitas dilakukan menggunakan metode *black-box testing* untuk memverifikasi bahwa semua *endpoint* API berfungsi sesuai dengan spesifikasi yang dirancang tanpa perlu mengetahui implementasi internal sistem. Pengujian mencakup validasi format *response*, penanganan *error*, dan konsistensi perilaku antara *Layer* 1 dan *Layer* 2. Metodologi pengujian dikembangkan menggunakan Postman untuk menguji berbagai skenario, termasuk pengujian *session management*, validasi proses transaksi, dan verifikasi integritas data antarlayer.

3.3.3.2 Pengukuran Metrik Evaluasi

Metodologi pengukuran metrik evaluasi dirancang untuk fokus pada aspek performa sistem, khususnya dalam hal latensi operasi. Metrik yang akan diukur meliputi *response time* untuk operasi *Layer* 2, latensi *commit* ke *Layer* 1, dan latensi keseluruhan *workflow*. Pengujian direncanakan pada berbagai konfigurasi *node* menggunakan metodologi yang telah ditetapkan pada Subbab 3.2.2, dengan variasi jumlah *node* L1 (4, 7, 10, 13, 16) dan L2 (1 dan 2) untuk menganalisis skalabilitas sistem.

Kerangka kerja *benchmarking* diimplementasikan sebagai aplikasi terpisah menggunakan bahasa pemrograman Go yang merekam metrik performa dalam format CSV. Data yang dikumpulkan kemudian akan diproses menggunakan Python untuk menghasilkan visualisasi yang menyoroti karakteristik performa sistem pada berbagai konfigurasi. Hasil pengukuran akan dibandingkan dengan arsitektur satu *layer* yang digunakan oleh DeWS [14] untuk mendemonstrasikan efektivitas pendekatan dua *layer* dalam mengurangi latensi dalam sistem BFT.

BAB 4

HASIL DAN PEMBAHASAN

Bab ini menyajikan hasil penelitian yang menjawab tujuan penelitian yaitu merancang dan mengimplementasikan arsitektur *web service* dua *layer* dengan *Byzantine fault tolerance* yang dapat mengatasi *trade-off* fundamental antara ketanggapan dan jaminan keamanan BFT.

4.1 Implementasi dan *Proof-of-Concept* Sistem

Subbagian ini membahas detail implementasi teknis dari arsitektur dua *layer* yang diajukan sebagai solusi *trade-off* antara ketanggapan dan keamanan BFT. Implementasi ini menggunakan CometBFT sebagai *consensus engine* dengan skenario *supply chain* sebagai *proof-of-concept* untuk mendemonstrasikan efektivitas pemisahan konsensus simulasi (*Layer 2*) dan konsensus BFT penuh (*Layer 1*).

4.1.1 Konfigurasi dan Pengaturan Sistem

Implementasi sistem dimulai dengan pengaturan komponen-komponen dasar yang mendukung operasional arsitektur dua *layer*. Komponen utama meliputi *repository* sebagai abstraksi untuk pengaksesan data, *service registry* untuk mengelola *endpoint* HTTP, dan inisialisasi konfigurasi sistem.

Repository pattern diimplementasikan untuk mengelola operasi database dan komunikasi dengan *node Layer 1*. Struktur ini menyediakan abstraksi yang memisahkan logika bisnis dari detail implementasi penyimpanan data, seperti yang ditunjukkan pada Kode 4.1. Implementasi ini mencakup pengelolaan sesi dengan mekanisme transaksi database yang aman, termasuk melakukan *rollback* jika terjadi kesalahan, memastikan setiap transaksi ke database memenuhi prinsip ACID.

```
1 type Repository struct {  
2     db          *gorm.DB  
3     // ... other attributes  
4 }  
5  
6 func (r *Repository) CreateTestPackage(  
7     requestID string ,  
8 ) (string , *RepositoryError) {  
9     supplierID := "SUP-001"  
10    pkgID := fmt.Sprintf("PKG-%s" , requestID[:8])  
11  
12    tx := r.db.Begin()
```



```

13 pkg := models.Package{
14     ID:          pkgID,
15     // ... other attributes
16 }
17 if err := tx.Create(&pkg).Error; err != nil {
18     tx.Rollback()
19     // ... error handling
20 }
21 if err := tx.Create(&item).Error; err != nil {
22     tx.Rollback()
23     // ... error handling
24 }
25 if err := tx.Commit().Error; err != nil {
26     // ... error handling
27 }
28 return pkg.ID, nil
29 }
30
31 // ... other Repository methods to interact with the database

```

Kode 4.1. Operasi database dengan *repository pattern*.

Service Registry berfungsi sebagai komponen sentral yang mengelola pemetaan antara *endpoint* HTTP dan *handler* yang sesuai. Setiap *handler* didaftarkan dengan informasi metode HTTP, pola *path*, dan *method repository* yang dijalankan sebagaimana terlihat pada Kode 4.2. Perbedaan ini krusial untuk arsitektur dua *layer* karena menentukan apakah operasi akan diproses di *Layer 2* dengan simulasi konsensus atau langsung diteruskan ke *Layer 1* menggunakan konsensus BFT.

```

1 type ServiceRegistry struct {
2     handlers      map[RouteKey] ServiceHandler
3     exactRoutes    map[RouteKey] bool
4     mu             sync.RWMutex
5     repository     *repository.Repository
6     logger         cmtlog.Logger
7     isByzantine    bool
8 }
9
10 func (sr *ServiceRegistry) RegisterDefaultServices() {
11     // Endpoints
12     // Test Create Package Endpoint
13     sr.RegisterHandler(
14         "POST",

```

```

15     "/session/test-package",
16     true,
17     sr.CreateTestPackage,
18 )
19     // ... the rest of the routes
20 }

```

Kode 4.2. Implementasi *endpoint route*, *service handler*, dan *service registry*.

Inisialisasi *service registry* dilakukan dengan mengonfigurasi dependensi yang diperlukan dan meregistrasi *service handler* default yang mendukung skenario *supply chain* yang telah dipilih (Kode 4.3). Proses ini memastikan bahwa semua *endpoint* yang diperlukan untuk *workflow* telah terdaftar dan siap digunakan.

```

1 func NewServiceRegistry(
2     repository *repository.Repository,
3     logger cmtlog.Logger,
4     isByzantine bool,
5 ) *ServiceRegistry {
6     return &ServiceRegistry{
7         handlers:    make(map[RouteKey]ServiceHandler),
8         exactRoutes: make(map[RouteKey]bool),
9         repository:  repository,
10        logger:      logger,
11        isByzantine: isByzantine,
12    }
13 }
14
15 // Initialize Service Registry
16 serviceRegistry := service_registry.NewServiceRegistry(
17     repository,
18     logger,
19     false,
20 )
21 serviceRegistry.RegisterDefaultServices()

```

Kode 4.3. Inisialisasi *service registry*.

Setelah *service registry* didefinisikan, *web server* diinisialisasi untuk menyediakan antarmuka HTTP yang memungkinkan klien berinteraksi dengan sistem. Server ini terintegrasi dengan *service registry* untuk mengarahkan *request* dari klien ke *handler* dan fungsi *repository* yang sesuai untuk mengakses data yang diperlukan. Integrasi ini memungkinkan pemrosesan permintaan yang efisien baik untuk operasi *Layer 2* maupun komitmen ke *Layer 1*, seperti yang ditunjukkan pada Kode 4.4.

```

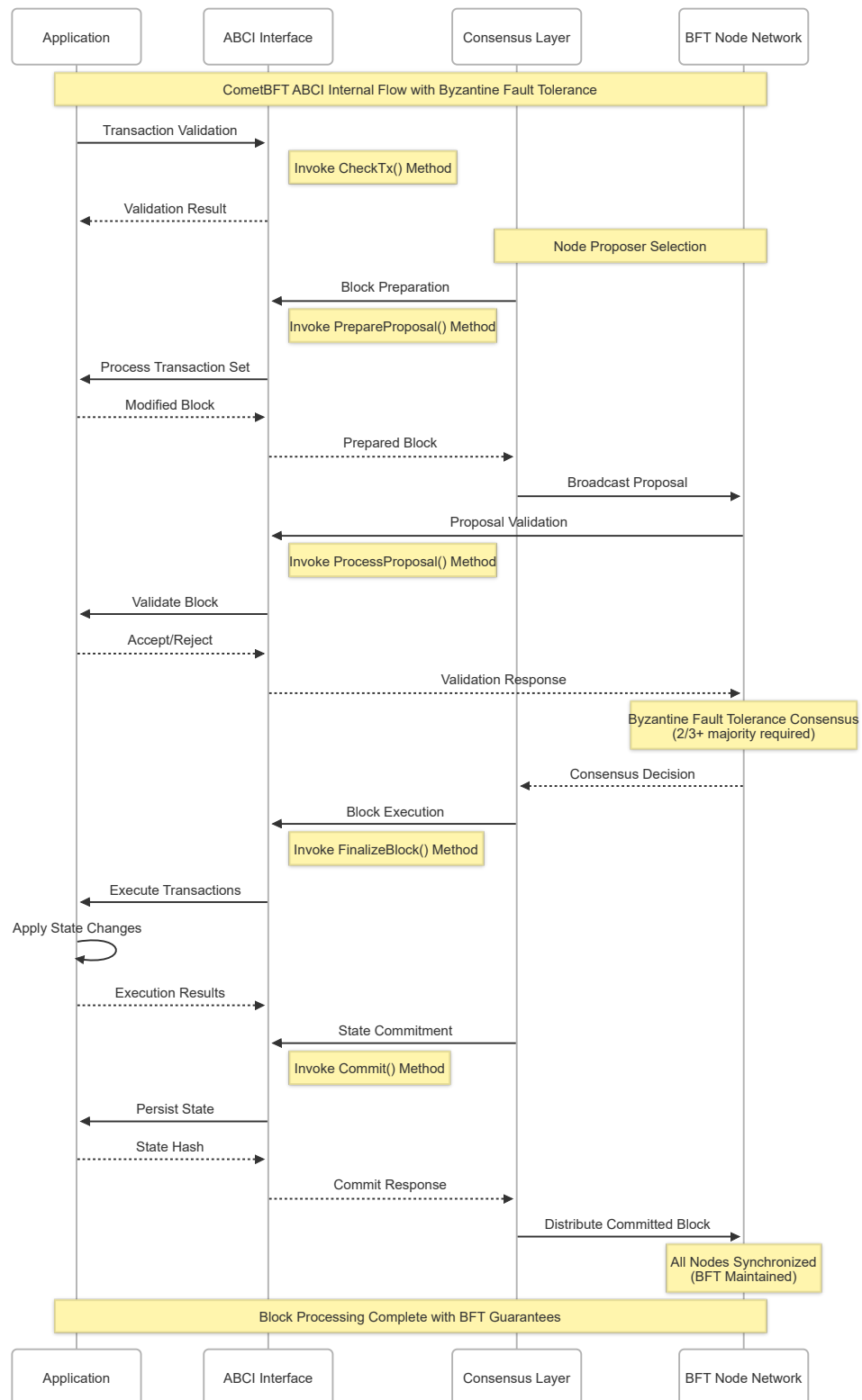
1 // Start Web Server
2 webserver, err := server.NewWebServer(
3     app,
4     httpPort,
5     logger,
6     node,
7     serviceRegistry,
8     repository,
9 )
10 if err != nil {
11     log.Fatalf("Creating web server: %v", err)
12 }
13 err = webserver.Start()
14 if err != nil {
15     log.Fatalf("Starting HTTP server: %v", err)
16 }

```

Kode 4.4. Inisialisasi *web server*.

4.1.2 Infrastruktur CometBFT

Infrastruktur CometBFT menyediakan fondasi konsensus, komunikasi *peer-to-peer*, dan *blockchain* untuk kedua *layer* dalam arsitektur ini. Pengaturan komponen ini mencakup inisialisasi aplikasi ABCI (*Application BlockChain Interface*), konfigurasi *node* CometBFT, dan pengaturan *web service* untuk menangani *request* HTTP.



Gambar 4.14. Interaksi aplikasi-konsensus dan pemanggilan *Method* ABCI.

Gambar 4.14 menunjukkan alur internal CometBFT yang menggambarkan interaksi antara komponen-komponen utama sistem. Diagram ini memperlihatkan bagaimana aplikasi berinteraksi dengan *consensus layer* melalui ABCI Interface

untuk mencapai *Byzantine fault tolerance*. Alur ini dimulai dari validasi transaksi, persiapan blok, fase konsensus BFT dengan jaringan *node*, hingga eksekusi blok dan komitmen state. Setiap tahap melibatkan pemanggilan *method-method* ABCI yang spesifik, seperti `CheckTx()`, `PrepareProposal()`, `ProcessProposal()`, `FinalizeBlock()`, dan `Commit()`, yang dapat dikustomisasi dan berguna memastikan sinkronisasi antar *node* dalam jaringan BFT.

Aplikasi ABCI diinisialisasi dengan konfigurasi yang menentukan parameter operasional sistem, termasuk identitas *node*, jumlah suara yang diperlukan untuk konsensus, dan pengaturan *logging* transaksi. Konfigurasi ini memungkinkan sistem untuk beroperasi dalam mode yang sesuai dengan kebutuhan arsitektur dua *layer*, seperti yang ditunjukkan pada Kode 4.5.

```

1 // Create ABCI Application\
2 appConfig := &app.AppConfig{
3     NodeID:      filepath.Base(homeDir),
4     LogAllTxns:   true,
5 }
6 logger := cmtlog.NewTMLogger(cmtlog.NewSyncWriter(os.Stdout))
7 app := app.NewABCIAApplication(
8     db,
9     serviceRegistry,
10    appConfig,
11    logger,
12    repository,
13 )

```

Kode 4.5. Inisialisasi aplikasi *blockchain* CometBFT.

Inisialisasi *node* CometBFT merupakan langkah krusial yang menghubungkan aplikasi dengan jaringan konsensus. Proses ini melibatkan konfigurasi validator, kunci kriptografis *node*, dan penyedia database. *Node* juga dikonfigurasi untuk berkomunikasi dengan *node* *Layer* 1 lainnya melalui *RPC client*, memungkinkan *Layer* 2 untuk mengirimkan transaksi yang perlu dikonsensus ke *Layer* 1, sebagaimana terlihat pada Kode 4.6.

```

1 // Initialize CometBFT node
2 node, err := nm.NewNode(
3     context.Background(),
4     nodeKey,
5     // ... other params
6 )
7 if err != nil {
8     log.Fatalf("Creating node: %v", err)
9 }

```

```

9   }
10  // Pass Node ID to app
11  app.SetNodeID(string(node.NodeInfo().ID()))
12  // Instantiate rpc client from node
13  rpcClient := cmtrpc.New(node)
14  // Start CometBFT node
15  node.Start()

```

Kode 4.6. Inisialisasi *node* CometBFT.

4.1.3 Implementasi Logika Konsensus Melalui ABCI

Implementasi konsensus BFT dilakukan dengan meng-*override* tiga *method* utama dari ABCI CometBFT yang sebelumnya telah dijelaskan pada Subbagian 4.1.2, guna merealisasikan logika konsensus sesuai dengan arsitektur sistem, skenario *supply chain*, dan Algoritma 1 yang telah dirancang. Proses ini memungkinkan sistem untuk memvalidasi transaksi, memproses proposal blok, dan menyelesaikan komitmen blok secara konsisten dengan jaminan toleransi terhadap *Byzantine fault*. Dengan demikian, sistem dapat memastikan bahwa setiap langkah eksekusi berjalan sesuai spesifikasi, sekaligus menjaga keandalan dan integritas layanan dalam konteks aplikasi terdistribusi dan multi-pihak.

CheckTx berfungsi sebagai gerbang validasi awal untuk setiap transaksi yang masuk ke dalam sistem. Metode ini memverifikasi format transaksi dan memastikan bahwa data dapat di-*unmarshal* dengan benar sebelum transaksi diproses lebih lanjut dalam *pipeline* konsensus, seperti yang ditunjukkan pada Kode 4.7.

```

1 func (app *Application) CheckTx(
2     _ context.Context,
3     check *abcitypes.CheckTxRequest,
4 ) (*abcitypes.CheckTxResponse, error) {
5     txBytes := check.Tx
6
7     var tx srvreg.Transaction
8     err := json.Unmarshal(txBytes, &tx)
9     if err != nil {
10        return &abcitypes.CheckTxResponse{
11            Code: 1,
12        },
13        fmt.Errorf(
14            "fail to parse tx on CheckTx: %s",
15            err.Error(),
16        )

```

```

17     }
18
19     return &abctypes.CheckTxResponse{
20         Code: 0,
21     }, nil
22 }

```

Kode 4.7. Implementasi *method* CheckTx yang meng-*override interface* CometBFT.

ProcessProposal merupakan inti dari mekanisme *Byzantine fault tolerance* dalam arsitektur ini. Metode ini memvalidasi setiap proposal blok dengan mereplikasi eksekusi transaksi dan membandingkan hasilnya untuk mendeteksi perilaku *Byzantine*. Proses validasi mencakup pemeriksaan originator transaksi dan perbandingan respons untuk memastikan konsistensi di seluruh *node*, sebagaimana terlihat pada potongan Kode 4.8, yang versi lengkapnya dapat dilihat pada Kode L.1.

```

1  for _, txBytes := range proposal.Txs {
2      var tx *srvreg.Transaction
3      json.Unmarshal(txBytes, &tx)
4      isTxOriginator := app.nodeID == tx.OriginNodeID
5      if !isTxOriginator {
6          // node is not the originator,
7          // replicate the request and compare the response
8          handler, isHandlerFound := app.serviceRegistry.
9              GetHandlerForPath(
10                 tx.Request.Method,
11                 tx.Request.Path,
12             )
13         if isHandlerFound {
14             response, err := handler(&tx.Request)
15             if err != nil {
16                 // vote invalid, error occurred
17             }
18             if !compareResponses(response, &tx.Response) {
19                 // vote invalid, response is not equal
20                 // Byzantine behavior detected
21             } else {
22                 // vote invalid, handler not found
23                 // Byzantine behavior detected
24             }
25         }
26         // vote valid
27     return &abctypes.ProcessProposalResponse{

```

28

```
Status: abcitypes.PROCESS_PROPOSAL_STATUS_ACCEPT}, nil
```

Kode 4.8. Implementasi *method* ProcessProposal yang meng-*override* interface CometBFT.

FinalizeBlock menyelesaikan proses konsensus dengan mengeksekusi semua transaksi yang telah divalidasi dan menyimpannya ke dalam *state* aplikasi. Metode ini mengelola transaksi database, membuat *application hash*, dan menyimpan informasi blok untuk mempertahankan konsistensi *state* di seluruh *node*, seperti yang ditunjukkan pada *snippet* Kode 4.9, dengan versi lengkap pada Kode L.2.

```
1 // ... pre-processing and validation
2 for i, txBytes := range req.Txs {
3     var tx srvreg.Transaction
4     txID := generateTxID(
5         tx.Request.RequestID,
6         tx.OriginNodeID,
7     )
8     // accept all tx that made it through to this method
9     status := "accepted"
10    txResults[i] = app.storeTransaction(
11        // id, tx, etc.
12    )
13 }
14 blockHeight := req.Height
15 appHash := calculateAppHash(txResults)
16 // store block info and app hash
17 err := app.onGoingBlock.
18     Set(
19         []byte("last_block_height"),
20         int64ToBytes(blockHeight),
21     )
22 // ... error handling
23 err = app.onGoingBlock.
24     Set(
25         []byte("last_block_app_hash"),
26         appHash,
27     )
28 // ... error handling
29 return &abcitypes.FinalizeBlockResponse{
30     TxResults: txResults,
31     AppHash:   appHash,
```


Kode 4.9. Implementasi *method* FinalizeBlock yang meng-*override interface* CometBFT.

4.1.4 Skenario *Proof-of-Concept*

Skenario manajemen *supply chain* dipilih sebagai *proof-of-concept* yang ideal untuk mendemonstrasikan kemampuan dalam menangani proses *multi-step* yang kompleks sambil mempertahankan ketangguhan dan *Byzantine fault tolerance*. Skenario ini merepresentasikan proses bisnis dunia nyata di mana operator memerlukan *feedback* langsung selama operasi, namun catatan akhir harus bersifat *immutable* dan dapat diaudit lintas batas organisasi.

Implementasi *workflow* dalam skenario *use case* ini terdiri dari lima operasi berurutan yang mendemonstrasikan *batching* transaksi berbasis sesi:

1. **Inisiasi Sesi** (POST /session/start)
Membuat sesi pemrosesan baru untuk penanganan paket
2. **Pemindaian Paket** (GET /session/:id/scan/:packageID)
Mengidentifikasi dan mengambil informasi paket
3. **Validasi Tanda Tangan Digital** (POST /session/:id/validate)
Pemeriksaan asal paket menggunakan tanda tangan digital dari *supplier*
4. **Pemeriksaan Kualitas (*Quality Check*)** (POST /session/:id/qc)
Melakukan penilaian kondisi dan kualitas paket
5. **Pembuatan Label Pengiriman** (POST /session/:id/label)
Menugaskan kurir dan menghasilkan informasi pengiriman kepada penerima

Setiap langkah beroperasi dalam konsensus simulasi *Layer 2*, memberikan *feedback* langsung kepada operator sambil mempertahankan konsistensi *state*. Sesi lengkap kemudian di-*commit* ke *Layer 1* sebagai unit atomik melalui *endpoint commit* (POST /commit/:id). Selain itu, ditambahkan juga *endpoint* pembuatan paket (POST /session/test-package) yang digunakan untuk membuat paket baru yang unik sehingga memudahkan perulangan ketika proses evaluasi.

Pendekatan berbasis sesi ini memungkinkan *batching* operasi terkait, secara signifikan mengurangi frekuensi operasi konsensus BFT yang mahal sekaligus memudahkan sinkronisasi dan mencegah konflik *state* dalam sistem. Arsitektur ini memungkinkan jaminan keamanan BFT yang diperlukan untuk catatan akhir yang bersifat *tamper-proof* dan mendukung proses audit. Di sisi lain, para operator tetap mendapatkan ketangguhan yang diperlukan untuk produktivitas tinggi ketika memproses

paket.

Implementasi ini menunjukkan bagaimana arsitektur dua *layer* dapat diterapkan pada domain aplikasi lain yang memerlukan optimasi serupa antara ketangguhan dan jaminan keamanan, seperti layanan keuangan, *e-government*, sistem manajemen identitas, atau platform perdagangan aset digital dan *metaverse*.

4.1.5 Deployment dan Kontainerisasi

Validasi arsitektur BFT dua *layer* ini dilakukan melalui *deployment* dalam lingkungan kontainer menggunakan Docker. Pendekatan ini memungkinkan pengujian sistem dalam kondisi yang mendekati *production* sambil memberikan isolasi dan konsistensi lingkungan yang diperlukan untuk evaluasi performa.

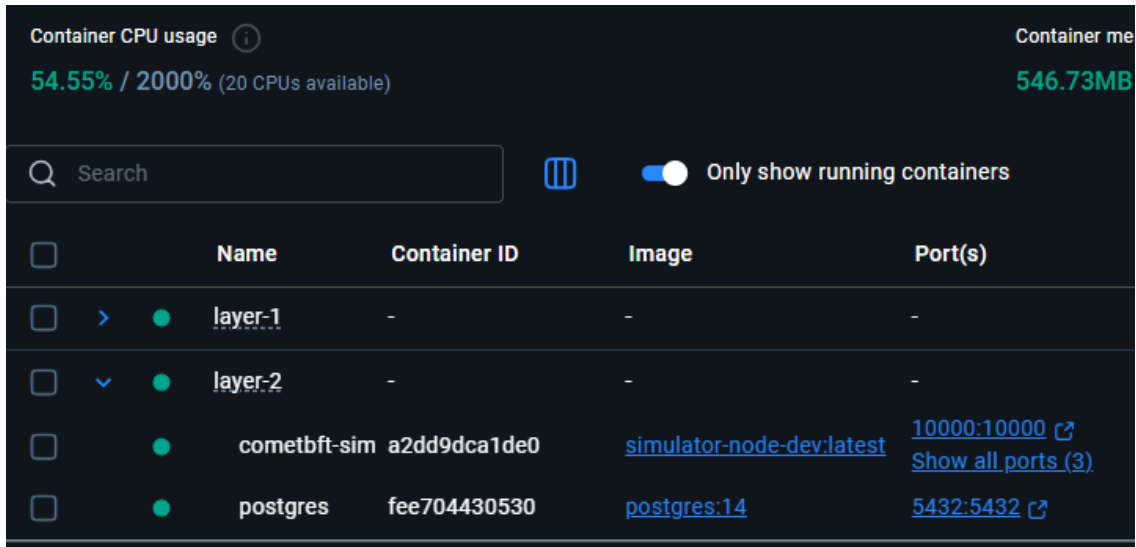
Layer 1 di-*deploy* sebagai jaringan BFT dengan konfigurasi minimum 4 *node* untuk memenuhi persyaratan BFT ($3f + 1$ dimana $f = 1$). Setiap *node Layer 1* berjalan dalam kontainer Docker terpisah dengan konfigurasi yang identik dan berpasangan dengan satu kontainer *database* Postgresql, memastikan replikasi komputasi penuh seperti yang ditunjukkan pada Gambar 4.15. Konfigurasi ini memvalidasi kemampuan sistem untuk mencapai konsensus BFT dalam lingkungan terdistribusi.

<input type="checkbox"/>	Name	Container ID	Image	Port(s)
<input type="checkbox"/>	layer-1	-	-	-
<input type="checkbox"/>	node1	19a6b529961a	layer1-node-dev:latest	5001:5001 ↗ Show all ports (3)
<input type="checkbox"/>	node2	145ac8ae154e	layer1-node-dev:latest	5002:5002 ↗ Show all ports (3)
<input type="checkbox"/>	node3	e1c5a83d7d81	layer1-node-dev:latest	5003:5003 ↗ Show all ports (3)
<input type="checkbox"/>	node0	f2a83e850fa0	layer1-node-dev:latest	5000:5000 ↗ Show all ports (3)
<input type="checkbox"/>	postgres0	ba1e89d1a6a8	postgres:14	5440:5432 ↗
<input type="checkbox"/>	postgres3	27f17cbaa8f4	postgres:14	5443:5432 ↗
<input type="checkbox"/>	postgres2	f47ebb8f9b18	postgres:14	5442:5432 ↗
<input type="checkbox"/>	postgres1	d986f91a72ce	postgres:14	5441:5432 ↗

Gambar 4.15. Implementasi L1 dalam kontainer Docker dengan 4 *node*.

Layer 2 diimplementasikan dengan konfigurasi *node* dengan jumlah satu atau dua karena berfokus pada ketangguhan dan simulasi konsensus. Meskipun menggunakan *node* di bawah batas minimum BFT, *Layer 2* tetap menjalankan logika validasi yang sama dengan *Layer 1* untuk memastikan konsistensi aturan bisnis, seperti yang terlihat pada

Gambar 4.16. Pendekatan ini mendemonstrasikan bagaimana *Layer 2* dapat memberikan *feedback* langsung sambil mempertahankan kompatibilitas dengan *Layer 1*.



Container CPU usage 54.55% / 2000% (20 CPUs available) Container memory 546.73MB

Search Only show running containers

	Name	Container ID	Image	Port(s)
<input type="checkbox"/>	layer-1	-	-	-
<input type="checkbox"/>	layer-2	-	-	-
<input type="checkbox"/>	cometbft-sim	a2dd9dca1de0	simulator-node-dev:latest	10000:10000 ↗ Show all ports (3)
<input type="checkbox"/>	postgres	fee704430530	postgres:14	5432:5432 ↗

Gambar 4.16. Implementasi L2 dalam kontainer Docker dengan 1 *node*.

Konfigurasi kontainerisasi ini memungkinkan pengujian skenario *use case* secara *end-to-end*, mulai dari operasi interaktif di *Layer 2* hingga finalisasi konsensus di *Layer 1*. *Setup* ini juga menyediakan fondasi untuk evaluasi performa yang akan dibahas pada bagian selanjutnya, memungkinkan pengukuran latensi dan *throughput* dalam kondisi yang terkontrol namun menyerupai kondisi aplikasi nyata.

4.2 Evaluasi dan Analisis Performa

Evaluasi performa sistem dilakukan untuk mengukur efektivitas arsitektur dua layer dalam mencapai ketanggapan tinggi sambil mempertahankan jaminan BFT. Metodologi evaluasi menggunakan implementasi *proof-of-concept* dengan skenario pendataan data paket dalam *supply chain management* yang mensimulasikan *workflow multi-step* dengan berbagai konfigurasi *node*. Skenario ini dipilih karena merepresentasikan kebutuhan ideal dimana ketanggapan tinggi dan jaminan keamanan BFT yang kuat sama-sama diperlukan.

Workflow yang diimplementasikan terdiri dari lima tahap berurutan: (1) inisiasi sesi untuk memulai pemrosesan paket, (2) pemindaian paket untuk identifikasi dan mengambil data barang dalam paket, (3) validasi *digital signature* paket untuk autentikasi sumber paket, (4) inspeksi *quality control*, dan (5) generasi label pengiriman dan penugasan kurir. Setelah selesai, seluruh sesi dapat di-*commit* sebagai unit atomik ke L1, mendemonstrasikan bagaimana arsitektur dua *layer* ini dapat secara efektif menyeimbangkan pengalaman pengguna yang *responsive* dengan jaminan keamanan BFT dalam proses bisnis *multi-step*.

Kerangka kerja *benchmarking* untuk evaluasi diimplementasikan sebagai aplikasi Go terpisah yang merekam performance metrics dalam format CSV, potongan kode untuk aplikasi *benchmark* ditunjukkan pada Kode 4.10. Contoh data CSV hasil rekaman latensi untuk tahap pertama ditunjukkan pada Kode 4.11. Data kemudian diproses menggunakan Python dengan library `matplotlib` untuk menghasilkan visualisasi yang menunjukkan karakteristik performa sistem dengan konfigurasi yang berbeda-beda.

```

1  var results []RequestResult
2  totalStart := time.Now()
3  // 1. Create Test Package
4  start := time.Now()
5  resp, err := requestClient.POST("/session/test-package",
6                                     nil, opts)
7  elapsed := time.Since(start)
8  var testPackageResponse testPackageResponse
9  client.UnmarshalBody(resp, &testPackageResponse)
10 packageId := testPackageResponse.Body.PackageID
11 fmt.Printf("PackageID : %s [Delay: %v]\n",
12            packageId, elapsed)
13 results = append(results, RequestResult{
14     Name:      "Create Package",
15     Method:    "POST",
16     Endpoint:  "/session/test-package",
17     Layer:     "L2",
18     Latency:   elapsed,
19 })
20 // ... the rest of the workflow steps
21 // (Scan Package, Validate Package, Quality Check, etc.)
22 return results
23 }
```

Kode 4.10. Implementasi benchmark workflow untuk pengukuran latensi.

Konfigurasi yang diuji meliputi kombinasi L1 (BFT *layer*) dengan 4, 7, 10, 13, dan 16 *node*, serta L2 (*interactive layer*) dengan 1 dan 2 *node*. Jumlah *node* spesifik di L1 dipilih untuk merepresentasikan peningkatan threshold *Byzantine fault tolerance* dari 1 hingga 5, dihitung sesuai formula standar $n = 3f + 1$, dimana n adalah total jumlah *node* dan f adalah jumlah maksimum *Byzantine faults* yang dapat ditoleransi. Sementara itu, jumlah *node* di L2 dipilih dengan angka terkecil untuk mencapai latensi minimum. Setiap konfigurasi diuji sebanyak 100 kali perulangan *workflow* untuk memastikan hasilnya signifikan secara statistik dan dapat dipercaya. Untuk menyingkat, digunakan notasi $L1-L2$ untuk menunjukkan jumlah *node* pada masing-masing layer, misalnya 4-1 berarti

4 *node* L1 dan 1 *node* L2.

```

obscurian@LAPTOP-LEUJCDS:~/go/src/github.com/ahmadzakiakmal/thesis/src$ ./benchmark/benchmark
-n 5 -l1 4 -l2 1

[Iteration 1/5]
PackageID : PKG-a6eec04a [Delay: 118.051871ms]
SessionID : SESSION-c79834873d8847a752ca33328c7baa80 [Delay: 43.312311ms]
Package scan success [Delay: 35.182873ms]
Package validation success [Delay: 35.773827ms]
QC request successful [Delay: 38.259739ms]
Package labelling successful [Delay: 41.411761ms]
Session SESSION-c79834873d8847a752ca33328c7baa80, committed successfully to L1, block height 3
[Delay: 389.203705ms]

Total workflow execution time: 1.304119832s

[Iteration 2/5]
PackageID : PKG-c0277775 [Delay: 25.144672ms]
SessionID : SESSION-d86cdb642f8500412658501aa27819b4 [Delay: 29.429235ms]
Package scan success [Delay: 57.338661ms]
Package validation success [Delay: 31.850846ms]
QC request successful [Delay: 29.955809ms]
Package labelling successful [Delay: 23.77911ms]
Session SESSION-d86cdb642f8500412658501aa27819b4, committed successfully to L1, block height 6
[Delay: 208.908302ms]

Total workflow execution time: 1.008660805s

[Iteration 3/5]
PackageID : PKG-c7ba23d3 [Delay: 37.335591ms]

```

Gambar 4.17. Proses pengumpulan data menggunakan *script* Go.

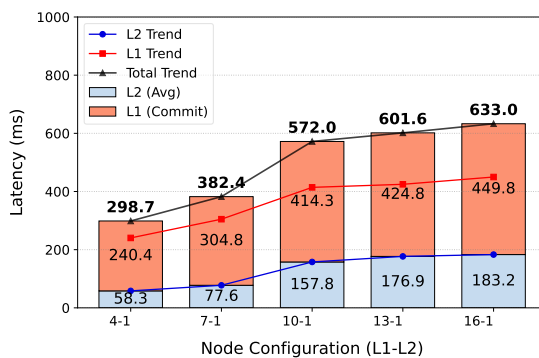
1	Iteration	Step	Method	Endpoint	Layer	Latency_ms	BlockHeight
2	1	Create Package	POST	/session/test-package	L2	40	0
3	1	Start Session	POST	/session/start	L2	31	0
4	1	Scan Package	GET	session/:id/scan/:packageId	L2	22	0
5	1	Validate Package	POST	session/:id/validate	L2	29	0
6	1	Quality Check	POST	session/:id/qc	L2	29	0
7	1	Label Package	POST	session/:id/label	L2	53	0
8	1	Commit Session	POST	commit/:id	L1+L2	211	36
9	1	Complete Workflow	WORKFLOW	complete-workflow	TOTAL	1022	0
10	2	Create Package	POST	/session/test-package	L2	34	0
11	2	Start Session	POST	/session/start	L2	27	0
12	2	Scan Package	GET	session/:id/scan/:packageId	L2	47	0
13	2	Validate Package	POST	session/:id/validate	L2	25	0
14	2	Quality Check	POST	session/:id/qc	L2	26	0
15	2	Label Package	POST	session/:id/label	L2	33	0
16	2	Commit Session	POST	commit/:id	L1+L2	286	39
17	2	Complete Workflow	WORKFLOW	complete-workflow	TOTAL	1082	0

Kode 4.11. Contoh data latensi yang disimpan dalam format CSV

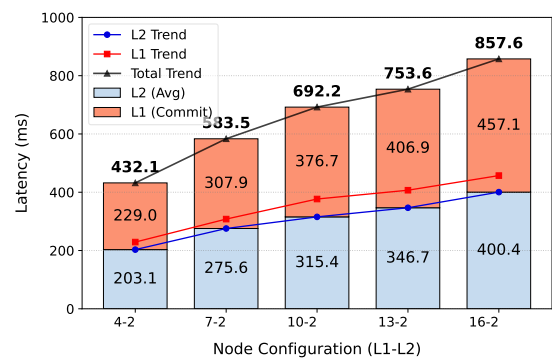
Hasil evaluasi menunjukkan pola performa yang cukup jelas dalam berbagai konfigurasi BFT cluster. Untuk konfigurasi dengan satu L2 *node*, operasi *Layer 2* (*Create*

Package, *Start Session*, *Scan Package*, *Validate Package*, *Quality Check*, dan *Label Package*) mempertahankan latensi yang relatif rendah bahkan seiring bertambahnya ukuran BFT cluster, seperti ditunjukkan pada Gambar 4.18a. Pada konfigurasi 4-1, operasi L2 rata-rata hanya 58.3ms, sementara operasi *Commit* yang memerlukan konsensus mencapai 240.4ms, menghasilkan latensi total *workflow* selama 298.7ms.

Ketika kluster BFT L1 diperbesar hingga 16 *node*, rata-rata latensi operasi L2 meningkat menjadi 183.2ms, sementara latensi operasi *Commit* naik menjadi 449.8ms, menghasilkan latensi *workflow* dengan total 633ms. Hal ini mendemonstrasikan bahwa meskipun cluster BFT yang lebih besar mengakibatkan peningkatan latensi, operasi interaktif tetap responsif, dengan operasi L2 secara konsisten selesai dalam waktu kurang dari 200ms.



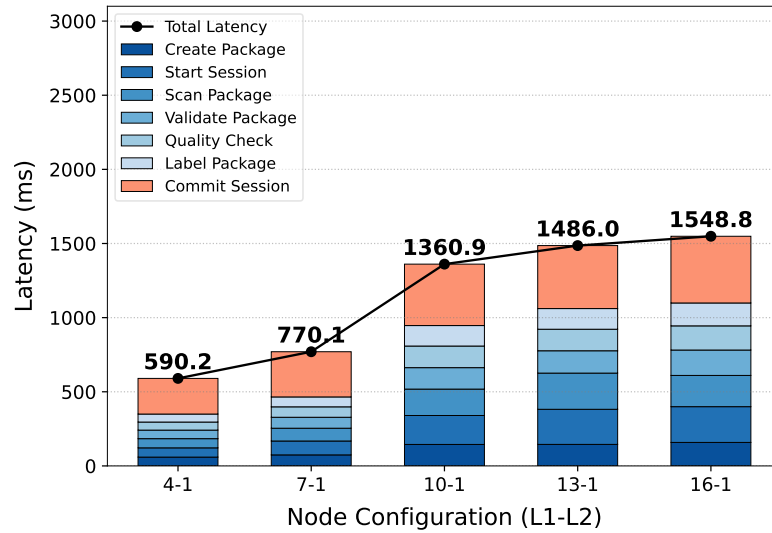
(a) Konfigurasi L2=1



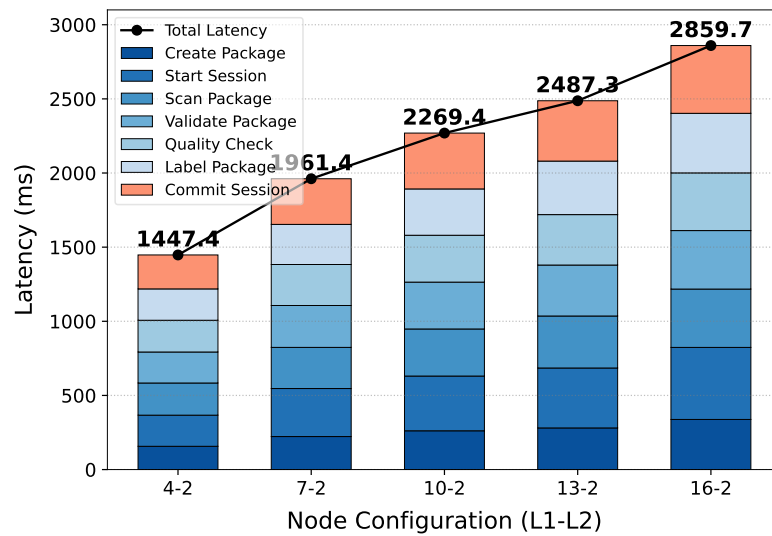
(b) Konfigurasi L2=2

Gambar 4.18. Perbandingan performa antara *Layer 1* dengan *Layer 2* pada konfigurasi yang berbeda.

Untuk merepresentasikan simulasi konsensus yang lebih realistis, dilakukan pengujian konfigurasi dengan dua *Layer 2 node*. Gambar 4.18b menunjukkan bahwa menambahkan *node* L2 meningkatkan *overhead* untuk koordinasi tambahan namun menyediakan lingkungan simulasi yang lebih akurat. Dengan konfigurasi 4-2, operasi L2 menghabiskan waktu rata-rata 203.1ms, sementara operasi *Commit* mencapai 229.0ms, menghasilkan total latensi *workflow* 432.1ms.



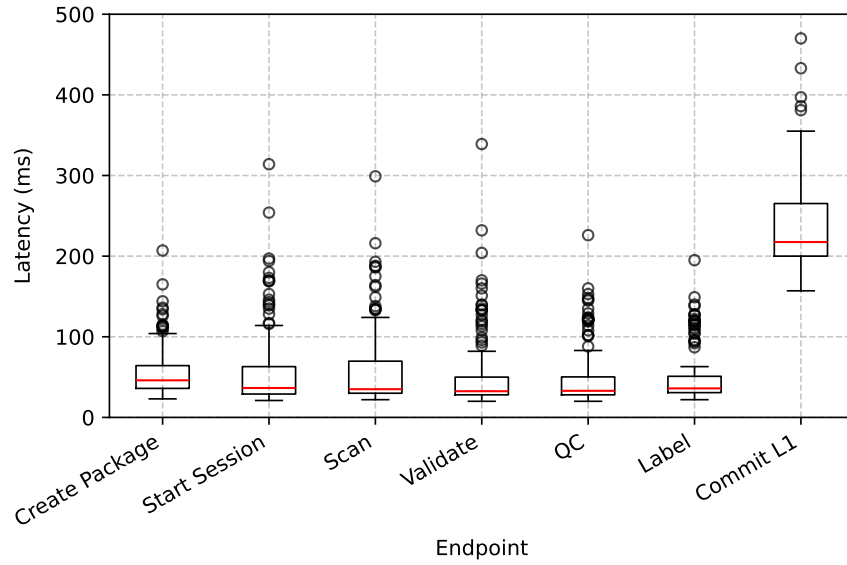
(a) Konfigurasi L2=1



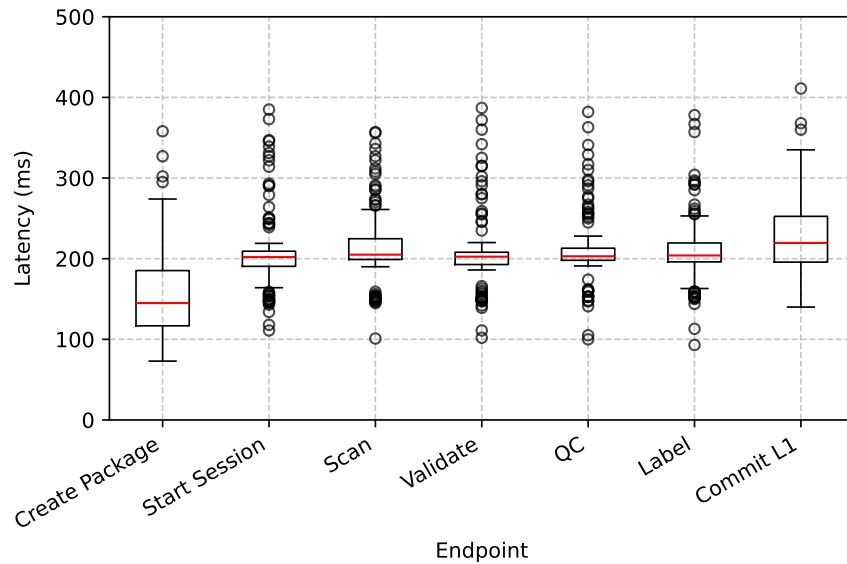
(b) Konfigurasi L2=2

Gambar 4.19. Perbandingan performa tiap *endpoint* Layer 1 dengan Layer 2 pada konfigurasi yang berbeda.

Ketika dilakukan *scaling* hingga 16 L1 *node* (konfigurasi 16-2), rata-rata latensi operasi L2 meningkat menjadi 400.4ms, dengan operasi *Commit* memerlukan 457.1ms, sehingga total latensi *workflow* menjadi 857.6ms. Latensi ini jauh lebih tinggi daripada konfigurasi L2 dengan satu *node*. Hal yang perlu digarisbawahi adalah bahwa latensi operasinya mendekati operasi *Commit* pada L1, sehingga keunggulan yang seharusnya diberikan oleh arsitektur dua layer menjadi tidak terasa dalam konfigurasi ini.



(a) Konfigurasi 4-1



(b) Konfigurasi 4-2

Gambar 4.20. Distribusi latensi tiap *endpoint* untuk dua konfigurasi *Layer 2* berbeda.

Analisis evaluasi dengan berbagai konfigurasi *node* menunjukkan bahwa konfigurasi L2=1 memberikan keseimbangan yang optimal antara performa dan *fault tolerance* untuk sebagian besar aplikasi. Konfigurasi dengan 10 hingga 16 *node* L1 mampu meningkatkan toleransi terhadap *Byzantine fault* dan L2 mampu mempertahankan latensi alur kerja yang tetap responsif untuk aplikasi seperti *supply chain* dan sejenisnya. Perbandingan antara operasi L1 dan L2 pada konfigurasi dengan satu *node* L2 menunjukkan penurunan latensi yang signifikan, di mana operasi L2 sekitar 2,5 hingga 4,1 kali lebih cepat dibandingkan proses *commit* pada L1. Rincian performa untuk setiap *endpoint* ditampilkan pada Gambar 4.19, yang menunjukkan konsistensi

performa operasi interaktif pada *Layer 2* di berbagai tahapan alur kerja.

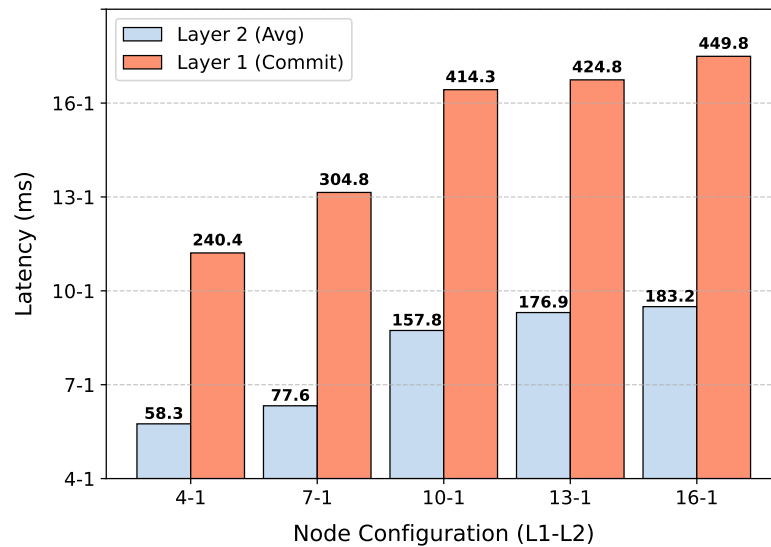
Gambar 4.20 memberikan pandangan yang lebih detail terhadap distribusi latensi pada tiap tahapan *workflow*. Terlihat bahwa pada konfigurasi $L2=1$, operasi interaktif di $L2$ seperti *Scan*, *Validate*, dan *Label* menunjukkan median latensi yang rendah dan distribusi yang sempit. Sebaliknya, konfigurasi $L2=2$ memperlihatkan peningkatan median yang jelas dan persebaran nilai latensi yang lebih lebar, dengan sejumlah besar *outlier*. Pola ini mengindikasikan bahwa penambahan *node* pada *Layer 2* mengakibatkan *overhead* koordinasi tambahan, meskipun tanpa protokol konsensus penuh. Dengan demikian, konfigurasi $L2$ multi-*node* dapat mengurangi efisiensi dan determinisme latensi, terutama dalam skenario dengan interaksi tinggi.

Konfigurasi $L2$ *node* ganda, meskipun secara teoretis diharapkan menghasilkan simulasi yang lebih akurat, juga menyebabkan latensi yang secara signifikan lebih tinggi tanpa diiringi peningkatan ketanggapan operasi $L2$ secara proporsional. Temuan ini mengindikasikan bahwa pada aplikasi yang mengutamakan performa konfigurasi $L2$ tunggal atau operasi langsung pada $L1$ lebih disarankan dibandingkan konfigurasi $L2$ multi-*node*, karena kompleksitas tambahan tidak memberikan manfaat berarti dalam skenario yang diuji. Secara teoretis, konfigurasi $L2$ ganda mungkin berguna bila rasio jumlah *node* antara $L1$ dan $L2$ sangat tinggi, misalnya 100 banding 2. Namun, situasi semacam itu belum dieksplorasi dalam penelitian ini.

Kemampuan $L2$ untuk mencapai waktu respons di bawah 200ms pada konfigurasi optimal, bahkan dalam kluster BFT dengan 16 *node*, menjadi bukti bahwa ketanggapan tinggi dapat dicapai tanpa mengorbankan jaminan BFT. Dengan melakukan *buffer* transaksi dan menunda proses konsensus hingga sesi selesai, sistem mampu memenuhi kebutuhan ketanggapan aplikasi modern sekaligus mempertahankan aspek keamanan dari BFT.

Trade-off antara keamanan dan ketanggapan berhasil diselesaikan melalui pemisahan proses yang sensitif terhadap latensi dari mekanisme konsensus BFT. Arsitektur ini membuka peluang baru bagi penerapan sistem BFT di domain yang sebelumnya dianggap tidak praktis karena kendala latensi, khususnya dalam alur kerja bertahap seperti manajemen *supply chain*. Hasilnya menunjukkan bahwa pendekatan dua *layer* mampu menggabungkan ketanggapan instan dari konsensus simulasi dengan jaminan keamanan dari konsensus BFT, menciptakan sistem yang interaktif dan andal.

4.3 Perbandingan dengan Pendekatan *State-of-the-Art*



Gambar 4.21. Perbandingan latensi antara arsitektur 1 *layer* (DeWS) dengan arsitektur 2 *layer*.

Perbandingan kinerja antara arsitektur 2 *layer* yang diusulkan dengan pendekatan *state-of-the-art* DeWS menunjukkan peningkatan signifikan dalam ketangguhan sistem. Seperti yang terlihat pada Gambar 4.21, *Layer 2* (simulasi) mencapai latensi rata-rata yang jauh lebih rendah dibandingkan *Layer 1* (commit) di semua konfigurasi *node*.

Keunggulan arsitektur yang diusulkan menjadi lebih menonjol ketika diaplikasikan pada *workflow* dengan beberapa langkah di dalamnya. Dalam skenario penggunaan *supply chain* yang terdiri dari 6 tahap seperti yang diimplementasikan dalam penelitian ini, arsitektur 2 *layer* akan menghasilkan total latensi sekitar 590.2ms untuk konfigurasi 4-1 ($58.3\text{ms} \times 6$ langkah interaktif + 240.4ms untuk commit akhir). Sebaliknya, pendekatan 1 *layer* seperti DeWS akan memerlukan waktu sekitar 1442.4ms ($240.4\text{ms} \times 6$ langkah) untuk menyelesaikan *workflow* yang sama. Peningkatan performa ini mencapai faktor 2.4x lebih cepat untuk satu *workflow* penuh, yang sangat krusial untuk aplikasi interaktif seperti *metaverse* atau sistem manajemen *supply chain real-time*.

Solusi ini terbukti menjadi pendekatan skalabilitas yang tepat untuk sistem terdistribusi yang spesifik menggunakan konsensus BFT dan *state machine replication*. Berbeda dengan solusi-solusi skalabilitas blockchain umum seperti *sidechain*, *rollups*, atau *sharding* yang sulit atau tidak dapat diimplementasikan pada sistem BFT *web services*, arsitektur 2 *layer* yang diusulkan memberikan solusi khusus untuk *trade-off* ketangguhan-keamanan dalam konteks aplikasi interaktif yang memerlukan jaminan BFT.

4.4 Keterbatasan Sistem

Meskipun arsitektur dua *layer* yang diusulkan mampu menyelesaikan *trade-off* antara kecepatan respons dengan jaminan keamanan BFT, solusi ini tetap memiliki sejumlah keterbatasan teknis yang penting untuk dipahami dalam konteks implementasi nyata. Keterbatasan ini berkaitan dengan aspek keamanan, efisiensi arsitektur, skalabilitas teknis, dan kesiapan produksi dalam skenario operasional yang kompleks.

1. *Layer 2* secara eksplisit mengorbankan *fault tolerance* untuk mencapai latensi rendah, sehingga tidak memiliki jaminan BFT. Dengan hanya 1–2 *node*, L2 rentan terhadap manipulasi data atau kegagalan validasi apabila salah satu *node* berperilaku *Byzantine* atau gagal. Meskipun mitigasi dapat dilakukan melalui pemisahan jalur komitmen ke *Layer 1*, arsitektur tetap membuka ruang risiko pada operasi interaktif.
2. Sistem saat ini belum dilengkapi dengan kontrol konkurensi yang kuat seperti *distributed locking* atau *optimistic concurrency control* untuk manajemen sesi multi-*node*. Hal ini dapat menimbulkan inkonsistensi data dalam skenario sesi paralel yang berjalan di lingkungan produksi terdistribusi.
3. Meskipun *Layer 2* hanya melakukan simulasi, implementasinya tetap menggunakan *consensus engine* CometBFT dengan konsensus BFT penuh. Ini menimbulkan overhead yang tidak sesuai terhadap tujuan L2 sebagai lapisan interaktif ringan. Solusi ke depannya perlu mengeksplorasi protokol konsensus ringan atau bahkan non-BFT untuk L2 yang kompatibel dengan logika BFT di L1.
4. Arsitektur diuji dalam konteks jaringan lokal dengan jumlah *node* terbatas. Dalam penyebaran global yang melibatkan latensi jaringan nyata dan jumlah *node* yang jauh lebih besar, performa sinkronisasi, propagasi status, dan *throughput* masih belum divalidasi secara menyeluruh.
5. Meskipun arsitektur mendukung proses audit melalui komitmen transaksi ke *ledger*, sistem belum mengintegrasikan fitur *production-ready* seperti *tracing*, audit *log* untuk tiap sesi, atau *monitoring* konsensus. Hal ini membatasi kemampuan audit dan *troubleshooting* pada skala operasional.
6. Sistem mengasumsikan bahwa logika validasi pada L2 identik dengan L1. Jika terjadi divergensi akibat perbedaan kode, versi, atau konfigurasi, hasil simulasi di L2 tidak lagi mencerminkan hasil yang valid dari konsensus yang akan dijalankan oleh L1, sehingga membuka potensi inkonsistensi.

Keterbatasan-keterbatasan teknis ini memberikan arah bagi pengembangan lanjutan, terutama untuk meningkatkan ketahanan sistem, mengurangi overhead, dan memastikan bahwa arsitektur dapat dioperasikan di lingkungan produksi skala besar.

BAB 5

KESIMPULAN DAN SARAN

5.1 Kesimpulan

Berdasarkan tujuan yang telah ditetapkan dan hasil penelitian yang telah didapatkan, dapat disimpulkan bahwa:

1. Penelitian ini berhasil mengurangi latensi sistem BFT melalui arsitektur dua *layer* yang memisahkan interaksi (L2) dari finalisasi konsensus (L1). Latensi interaksi di L2 tercatat 58,3 hingga 183,2 ms, atau sekitar 2,4 hingga 4 kali lebih cepat dibanding latensi komitmen L1 yang mencapai 240,4 hingga 449,8 ms. Arsitektur ini cocok untuk aplikasi dengan kebutuhan waktu respons rendah seperti *supply chain*.
2. Pendekatan dua *layer* yang diusulkan juga terbukti meningkatkan skalabilitas sistem. Evaluasi pada berbagai konfigurasi *node* (4, 7, 10, 13, dan 16) menunjukkan bahwa performa interaksi tetap konsisten seiring peningkatan skala sistem. Hal ini dicapai melalui strategi validasi *local-first*, batching transaksi menggunakan sesi, serta pemisahan tanggung jawab antara proses interaktif dan proses konsensus.
3. Arsitektur yang dikembangkan menerapkan model *State Machine Replication* dengan replikasi komputasi secara independen di setiap *node* sebelum proses konsensus BFT. Tidak seperti sebagian besar arsitektur BFT lainnya yang hanya mencapai konsensus atas hasil akhir dari satu *node*, sistem ini menjamin bahwa setiap *node* mengeksekusi *request* secara deterministik, memperkuat keandalan sistem tanpa bergantung pada mekanisme kriptografi eksternal seperti ZKP.

5.2 Saran

Berdasarkan hasil penelitian dan keterbatasan yang ditemukan, saran untuk penelitian selanjutnya adalah sebagai berikut:

1. Mengembangkan algoritma validasi yang lebih ringan dan efisien untuk menggantikan protokol BFT penuh di L2. Mekanisme ini harus tetap mampu mereplikasi logika bisnis secara deterministik dan menjaga kesesuaian hasil dengan L1, tanpa menimbulkan overhead komunikasi dan konsensus yang berat. Contohnya dapat mencakup pendekatan seperti konsensus lokal non-*Byzantine* dengan *fallback* ke L1 untuk finalitas.
2. Mengembangkan mekanisme *session management* yang lebih cocok untuk menangani *workflow* yang lebih kompleks, dependensi antar-operasi yang rumit,

atau *concurrency* yang lebih tinggi.

3. Melakukan pengujian pada skala yang lebih besar dengan ratusan *node* dan dalam lingkungan jaringan geografis yang sesungguhnya untuk memvalidasi skalabilitas solusi.
4. Mengembangkan model atau algoritma untuk menentukan konfigurasi optimal antara jumlah *node* L1 dan L2 berdasarkan karakteristik *workload* dan kebutuhan aplikasi spesifik.

DAFTAR PUSTAKA

- [1] Stylianos Mystakidis. Metaverse. *Encyclopedia*, 2(1):486–497, 2022.
- [2] Jahid Hasan Rony, Razib Hayat Khan, Jonayet Miah, and MM Mahbubul Syeed. E-Commerce Application in Metaverse: Requirements, Integration, Economics and Future Trends. In *2024 IEEE CONECCT*, pages 1–6, July 2024.
- [3] Mohsen Hatami, Qian Qu, Yu Chen, Hisham Kholidy, Erik Blasch, and Erika Ardiles-Cruz. A Survey of the Real-Time Metaverse: Challenges and Opportunities. *Future Internet*, 16(10):379, October 2024. Number: 10 Publisher: Multidisciplinary Digital Publishing Institute.
- [4] Sean Yang and Max Li. Web3.0 Data Infrastructure: Challenges and Opportunities. *IEEE Network*, 37(1):4–5, January 2023.
- [5] Thippa Reddy Gadekallu, Thien Huynh-The, Weizheng Wang, Gokul Yenduri, Pasika Ranaweera, Quoc-Viet Pham, Daniel Benevides da Costa, and Madhusanka Liyanage. Blockchain for the Metaverse: A Review, March 2022. arXiv:2203.09738 [cs].
- [6] Anand Singh Rajawat, S.B. Goyal, Aarti Goyal, Kavita Rajawat, Maria Simona Raboaca, Chaman Verma, and Traian Candin Mihaltan. Enhancing Security and Scalability of Metaverse with Blockchain-based Consensus Mechanisms. In *2023 15th Int. Conf. Electron., Comput. Artif. Intell. (ECAI)*, pages 01–06, June 2023.
- [7] A. U. Rehman, Rui L. Aguiar, and João Paulo Barraca. Fault-Tolerance in the Scope of Cloud Computing. *IEEE Access*, 10:63422–63441, 2022. Conference Name: IEEE Access.
- [8] Sucharitha Isukapalli and Satish Narayana Srirama. A systematic survey on fault-tolerant solutions for distributed data analytics: Taxonomy, comparison, and future directions. *Computer Science Review*, 53:100660, August 2024.
- [9] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.
- [10] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.
- [11] M.G. Merideth, Arun Iyengar, T. Mikalsen, S. Tai, I. Rouvellou, and P. Narasimhan. Thema: Byzantine-fault-tolerant middleware for Web-service applications. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 131–140, October 2005. ISSN: 1060-9857.
- [12] Wenbing Zhao. BFT-WS: A Byzantine Fault Tolerance Framework for Web Services. In *2007 Eleventh International IEEE EDOC Conference Workshop*, pages 89–96, October 2007.

- [13] Christian Berger and Hans P. Reiser. Webbf: Byzantine fault tolerance for resilient interactive web applications. In Silvia Bonomi and Etienne Rivière, editors, *Distributed Applications and Interoperable Systems*, pages 1–17, Cham, 2018. Springer International Publishing.
- [14] Gowri Sankar Ramachandran, Thi Thuy Linh Tran, and Raja Jurdak. DeWS: Decentralized and Byzantine Fault-tolerant Web Services. In *2023 IEEE ICBC*, pages 1–9, Dubai, United Arab Emirates, May 2023. IEEE.
- [15] Sajeeva L. Pallemulle, Haraldur D. Thorvaldsson, and Kenneth J. Goldman. Byzantine Fault-Tolerant Web Services for n-Tier and Service Oriented Architectures. In *2008 The 28th International Conference on Distributed Computing Systems*, pages 260–268, June 2008. ISSN: 1063-6927.
- [16] Ramakrishna Kotla, Allen Clement, Edmund Wong, Lorenzo Alvisi, and Mike Dahlin. Zyzzyva: Speculative Byzantine Fault Tolerance – Communications of the ACM, November 2008.
- [17] Yusuke Matsumoto and Hiromi Kobayashi. A Speculative Byzantine Algorithm for P2P System. In *2010 IEEE 16th Pacific Rim International Symposium on Dependable Computing*, pages 231–232, December 2010.
- [18] Mahen Mandal, Mohd Sameen Chishti, and Amit Banerjee. Investigating Layer-2 Scalability Solutions for Blockchain Applications. In *2023 IEEE HPCC/DSS/SmartCity/DependSys*, pages 710–717, December 2023.
- [19] Louis Tremblay Thibault, Tom Sarry, and Abdelhakim Senhaji Hafid. Blockchain Scaling Using Rollups: A Comprehensive Survey. *IEEE Access*, 10:93039–93054, 2022.
- [20] Stefano Calzavara, Hugo Jonker, Benjamin Krumnow, and Alvisè Rabitti. Measuring Web Session Security at Scale. *Computers & Security*, 111:102472, December 2021.
- [21] Sabu M. Thampi. Introduction to Distributed Systems, November 2009. arXiv:0911.4395 [cs].
- [22] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.
- [23] Seth Gilbert and Nancy Lynch. Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News*, 33(2):51–59, June 2002.
- [24] A. Avizienis and J.-C. Laprie. Dependable computing: From concepts to design diversity. *Proceedings of the IEEE*, 74(5):629–638, May 1986.
- [25] Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.
- [26] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column)* 32, 4 (Whole Number 121, December 2001), pages 51–58, December 2001.

- [27] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. pages 305–319, 2014.
- [28] Ahmet Vedat Tokmak, Akhan Akbulut, and Cagatay Catal. Web service discovery: Rationale, challenges, and solution directions. *Computer Standards & Interfaces*, 88:103794, March 2024.
- [29] Nuno Mateus-Coelho, Manuela Cruz-Cunha, and Luis Gonzaga Ferreira. Security in Microservices Architectures. *Procedia Computer Science*, 181:1225–1236, January 2021.
- [30] Sawsan Ali Hamid, Rana Alaudeen Abdulrahman, and Dr.Ruaa Ali Khamees. What is Client-Server System: Architecture, Issues and Challenge of Client -Server System (Review). February 2020. Publisher: Zenodo.
- [31] Johannes Thönes. Microservices. *IEEE Software*, 32(1):116–116, January 2015.
- [32] Alia Al Sadawi, Batool Madani, Sara Saboor, Malick Ndiaye, and Ghassan Abu-Lebdeh. A comprehensive hierarchical blockchain system for carbon emission trading utilizing blockchain of things and smart contract. *Technological Forecasting and Social Change*, 173:121124, December 2021.
- [33] Linhui Li, Peichang Shi, Xiang Fu, Peng Chen, Tao Zhong, and Jinzhu Kong. Three-Dimensional Tradeoffs for Consensus Algorithms: A Review. *IEEE Transactions on Network and Service Management*, 19(2):1216–1228, June 2022.
- [34] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. October 2008.
- [35] Shelke Kavita and Shinde S.K. A Comprehensive Survey of Consensus Protocols, Challenges, and Attacks of Blockchain Network. In *2024 Fourth International Conference on Advances in Electrical, Computing, Communication and Sustainable Technologies (ICAECT)*, pages 1–6, January 2024.
- [36] Ethereum Foundation. The merge. <https://ethereum.org/en/roadmap/merge/>, n.d. Accessed: 2025-05-20.
- [37] EOS Network Foundation. Consensus. <https://docs.eosnetwork.com/docs/latest/core-concepts/blockchain-basics/consensus>, 2025. Accessed: 2025-05-31.
- [38] TRON Network. Getting started. <https://developers.tron.network/docs/getting-start>, 2025. Accessed: 2025-05-31.
- [39] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus, November 2019. arXiv:1807.04938 [cs].
- [40] Ethan Buchman, Rachid Guerraoui, Jovan Komatovic, Zarko Milosevic, Dragos-Adrian Seredinschi, and Josef Widder. Revisiting Tendermint: Design Tradeoffs, Accountability, and Practical Use. In *2022 52nd Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. – Suppl. Vol. (DSN-S)*, pages 11–14, June 2022.
- [41] Interchain Foundation. The interchain ecosystem. <https://tutorials.cosmos.network/academy/1-what-is-cosmos/2-cosmos-ecosystem.html>, 2023. Accessed: 2025-06-04.

- [42] Xinmeng Liu, Haomeng Xie, Zheng Yan, and Xueqin Liang. A survey on blockchain sharding. *ISA Transactions*, 141:30–43, October 2023.
- [43] Hyperledger Fabric Contributors. Introduction — hyperledger fabric documentation, 2025. Accessed: 2025-06-05.
- [44] Ethereum Foundation. What is ethereum?, 2025. Accessed: 2025-06-05.
- [45] Tendermint Contributors. Tendermint core documentation. <https://docs.tendermint.com/master/>, 2024. Accessed: 2025-06-05.
- [46] CometBFT Team. Cometbft documentation, 2023.
- [47] Cosmos Network. Cosmos sdk documentation, 2024.
- [48] Cosmos SDK Contributors. Issue #14915: CometBFT as Default in Cosmos SDK. <https://github.com/cosmos/cosmos-sdk/issues/14915>, 2023. Accessed: 2025-06-05.
- [49] Rollkit Contributors. About rollkit. <https://rollkit.dev/learn/about>, 2024. Accessed: 2025-06-08.

LAMPIRAN

Link Repository GitHub:

<https://github.com/dteti-sys-rsch/fast-interactive-bft-web-services>



Gambar L.1. *QR Code Repository GitHub*

```
1 func (app *Application) ProcessProposal(  
2     _ context.Context ,  
3     proposal *abcitypes.ProcessProposalRequest ,  
4 ) (*abcitypes.ProcessProposalResponse , error) {  
5     fmt.Println("[PROCESSPROPOSAL]:")  
6     for _, txBytes := range proposal.Txs {  
7         var tx *srvreg.Transaction  
8         json.Unmarshal(txBytes , &tx)  
9  
10        isTxOriginator := app.nodeID == tx.OriginNodeID  
11        if !isTxOriginator {  
12            // node is not the originator ,  
13            // replicate the request and compare the response  
14            handler , isHandlerFound := app.serviceRegistry .  
15                GetHandlerForPath(  
16                    tx.Request.Method ,  
17                    tx.Request.Path ,  
18                )  
19            if isHandlerFound {  
20                response , err := handler(&tx.Request)
```

```

21     if err != nil {
22         fmt.Println("Voted invalid", err)
23         return &abcitypes.ProcessProposalResponse{
24             Status: abcitypes.PROCESS_PROPOSAL_STATUS_REJECT,
25         }, err
26     }
27     if !compareResponses(response, &tx.Response) {
28         fmt.Println("Voted invalid")
29         fmt.Println("Different responses")
30         fmt.Println("byzantine behavior detected")
31         return &abcitypes.ProcessProposalResponse{
32             Status: abcitypes.
33                 PROCESS_PROPOSAL_STATUS_REJECT,
34         },
35         fmt.Errorf("response is different")
36     }
37 } else {
38     fmt.Println("Voted invalid")
39     fmt.Println("Handler not found")
40     fmt.Println("Byzantine behavior detected")
41     return &abcitypes.ProcessProposalResponse{
42         Status: abcitypes.PROCESS_PROPOSAL_STATUS_REJECT,
43     },
44     fmt.Errorf("handler not found")
45 }
46 }
47 }
48 fmt.Println("Voted valid")
49 return &abcitypes.ProcessProposalResponse{ Status: abcitypes.
50     PROCESS_PROPOSAL_STATUS_ACCEPT,
51 }, nil
52 }

```

Kode L.1. Kode lengkap *method* ProcessProposal.

```

1 func (app *Application) FinalizeBlock(
2     _ context.Context,
3     req *abcitypes.FinalizeBlockRequest,
4 ) (*abcitypes.FinalizeBlockResponse, error) {
5     var txResults = make([]*abcitypes.ExecTxResult, len(req.Txs))
6
7     app.mu.Lock()

```

```

8  defer app.mu.Unlock()
9
10 app.onGoingBlock = app.badgerDB.NewTransaction(true)
11
12 for i, txBytes := range req.Txs {
13     var tx srvreg.Transaction
14
15     if err := json.Unmarshal(txBytes, &tx); err != nil {
16         txResults[i] = &abcitypes.ExecTxResult{
17             Code: 1,
18             Log:  "Invalid transaction format",
19         }
20         continue
21     }
22
23     txID := generateTxID(
24         tx.Request.RequestID,
25         tx.OriginNodeID,
26     )
27     // accept all tx that made it through to this method
28     status := "accepted"
29     txResults[i] = app.storeTransaction(
30         txID,
31         &tx,
32         status,
33         txBytes,
34     )
35 }
36
37 // store the last block info
38 blockHeight := req.Height
39
40 // calculate application hash
41 appHash := calculateAppHash(txResults)
42
43 // store block info
44 err := app.onGoingBlock.
45     Set(
46         []byte("last_block_height"),
47         int64ToBytes(blockHeight),
48     )

```

```

49  if err != nil {
50      log.Printf(
51          "Error storing block height: %v",
52          err,
53      )
54  }
55
56  err = app.onGoingBlock.
57      Set(
58          []byte("last_block_app_hash"),
59          appHash,
60      )
61  if err != nil {
62      log.Printf(
63          "Error storing app hash: %v",
64          err,
65      )
66  }
67
68  return &abcitypes.FinalizeBlockResponse{
69      TxResults: txResults,
70      AppHash:   appHash,
71  }, err
72 }

```

Kode L.2. Kode lengkap *method* FinalizeBlock.