BAB 2
TINJAUAN PUSTAKA

2.1 Penelitian Terdahulu

Penelitian sebelumnya telah menunjukkan bahwa pengembangan arsitektur *server* asinkron memberikan keunggulan signifikan dalam menangani konkurensi tinggi dibandingkan pendekatan sinkron konvensional. (Zhang et al., 2020) dalam penelitian "The Impact of Event Processing Flow on Asynchronous *Server* Efficiency" mengungkapkan bahwa *server* asinkron dapat mengalami penurunan kinerja signifikan akibat dua faktor: model one-event-one-handler yang menyebabkan context switch berlebihan dan masalah write-spin pada kondisi respons berukuran besar. Penelitian tersebut menunjukkan bahwa solusi hybrid yang menggabungkan keunggulan berbagai arsitektur asinkron mampu mencapai peningkatan throughput 10-90% dibandingkan *server* lainnya pada berbagai kondisi beban kerja dan jaringan.

Dalam konteks pengembangan web server, (Winardi et al., 2023) melakukan analisis perbandingan kinerja *web server* berbasis PHP dan Python Twisted. Hasil penelitian menunjukkan bahwa pemilihan teknologi yang tepat sangat penting untuk memastikan sistem dapat berfungsi dengan baik dalam kondisi nyata. Python Twisted dengan arsitektur event-driven menunjukkan kemampuan menangani banyak koneksi simultan dengan penggunaan thread yang minimal.

Di bidang IoT dan protokol komunikasi, (Saha et al., 2024) dalam "Performance evaluation framework of MQTT client libraries for IoT applications in manufacturing" menyajikan framework evaluasi kinerja untuk library client MQTT pada aplikasi IoT di industri manufaktur. Penelitian tersebut menekankan pentingnya latensi rendah, pertukaran data aman, dan kompatibilitas dengan perangkat terbatas sumber daya.

(Eka Putra et al., 2024) dalam *"Analisis Komparasi Protokol Websocket dan MQTT Dalam Proses Push Notification"* melakukan evaluasi eksperimental terhadap protokol WebSocket dan MQTT dalam konteks push notification. Hasil penelitian menunjukkan bahwa WebSocket lebih efisien dalam skenario aplikasi *real-time* dengan waktu delay lebih rendah (rata-rata 0.700s dibandingkan MQTT

1.008s) dan penggunaan CPU lebih rendah (6.602% vs 8.056% pada 1 client), sementara MQTT unggul dalam pertukaran pesan terdistribusi pada lingkungan IoT.

Penelitian oleh (Hicham G. Elmongui, 2014) yang dikutip dalam Gračanin et al. (2017) tentang "Performance evaluation of MQTT and CoAP via a common middleware" menyediakan metodologi untuk evaluasi kinerja protokol IoT menggunakan *middleware* bersama, yang menjadi dasar untuk pengujian komparatif dalam penelitian ini.

Berdasarkan penelitian-penelitian tersebut, terdapat kesenjangan penelitian yang belum terisi, yaitu implementasi spesifik mekanisme deferToThread pada Twisted Python untuk menangani operasi basis data *blocking* dalam konteks *dashboard* IoT *real-time*, serta evaluasi komprehensif terhadap dampaknya pada berbagai metrik kinerja (latensi, *Transaction per second*, *throughput*, CPU, dan RAM) dengan variasi beban data dan tingkat konkurensi.

Tabel 2. 1 Perbandingan penelitian

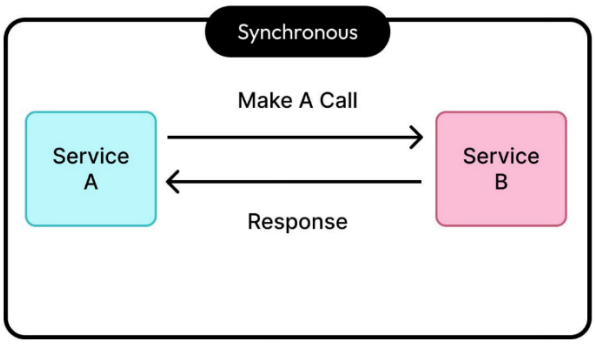
Nama	Judul	Komponen	Hasil
Maulana et al.	Smart Dashboard Design and Water Sensor Integration Architecture by Applying Internet of Things (IoT) Technology Using Data Analysis and Prediction Methods	• Antarmuka pengguna • IoT • *dashboard*	*Dashboard* pintar memberikan antarmuka intuitif, mempermudah pengguna tanpa latar belakang teknis untuk mengakses data.
Winardi et al.	Analisis Perbandingan Kinerja Web *Server* Berbasis PHP dan Web *Server* Berbasis Python Twisted	• *Web server* • bahasa pemrograman • *dashboard*	Pemilihan bahasa pemrograman memengaruhi kecepatan dan responsivitas *dashboard*, berdampak pada kinerja sistem.

Nama	Judul	Komponen	Hasil
Arif N. & Surantha	IoT Cloud Platform Based on Asynchronous Processing for Reliable Multi-user Health Monitoring	• *Cloud* • pemrosesan asinkro • multi-pengguna	Teknologi asinkron meningkatkan efisiensi pengelolaan data dan mendukung pengambilan keputusan lebih cepat dan akurat.
Zikri Zahron	IMPLEMENTASI JARINGAN ASINKRON PADA DASHBOARD IOT *REAL-TIME* BERBASIS TWISTED PYTHON	• *Web Server* • *Dashboard* • *Pemrosesan Asinkron* • *Multi Pengguna*	Dashboard dengan *backend* Asinkron

2.2 Landasan Teori

2.2.1 Synchronous

Pemrograman sinkron adalah pola eksekusi di mana instruksi atau operasi dijalankan secara berurutan dan saling menunggu; suatu operasi tidak dapat dimulai sampai operasi sebelumnya selesai. Dalam konteks komunikasi antar-layanan atau operasi I/O, model sinkron bersifat *blocking*, sehingga pengirim permintaan menunggu hingga penerima memproses dan mengembalikan respons sebelum melanjutkan eksekusi selanjutnya. Karakteristik ini mempermudah penalaran terhadap alur kontrol dan konsistensi keadaan aplikasi karena urutan eksekusi deterministik, namun dapat menurunkan responsivitas dan pemanfaatan sumber daya ketika operasi berlatensi tinggi karena waktu tunggu yang tidak dapat digunakan untuk pekerjaan lain. Secara praktis, pola sinkron sering digunakan pada alur yang memerlukan hasil segera, transaksi yang menuntut urutan ketat, atau lingkungan di mana latensi layanan relatif rendah dan determinisme lebih diutamakan..

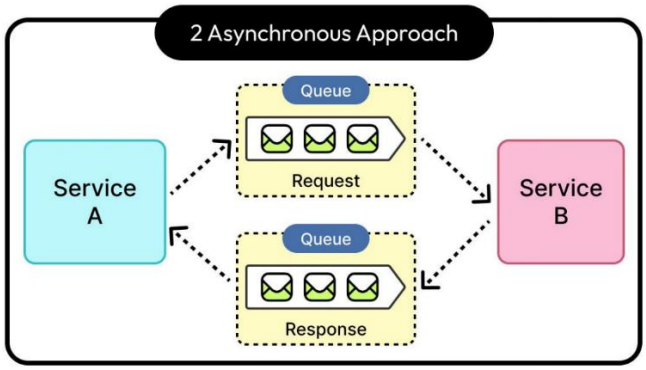


Gambar 2. 1 Synchronous

Gambar 2.1 memperlihatkan aliran komunikasi sinkron antara Service A dan Service B Service A mengirimkan request dan menunggu hingga Service B menyelesaikan pemrosesan dan mengembalikan response sebelum melanjutkan eksekusi. Skema ini menonjolkan sifat *blocking* dari eksekusi sinkron selama periode tunggu, *thread* atau proses pemanggil terblokir dan tidak dapat melakukan tugas lain. di sisi lain, pendekatan sinkron memudahkan pengurutan operasi dan konsepsi logika yang sederhana. Oleh karena itu, sinkron umumnya cocok untuk skenario kebutuhan transaksi segera, atau ketika kesederhanaan dan determinisme lebih bernilai daripada throughput maksimum.

2.2.2 Asynchronous

Pemrograman asinkron merupakan paradigma eksekusi yang memungkinkan unit pekerjaan dijalankan secara independen tanpa memblokir alur utama program. Pendekatan ini terutama diterapkan pada operasi I/O, komunikasi jaringan, dan kueri basis data yang berlatensi tinggi, sehingga penggunaan mekanisme non-blocking seperti callback, promise, atau event loop menjadi krusial untuk menjaga responsivitas sistem. Dengan memisahkan pengiriman permintaan dari pemrosesan hasil, sistem asinkron meningkatkan throughput dan skalabilitas dengan cara memanfaatkan penjadwalan tugas dan antrean pesan untuk mengelola kerja paralel. Selain mempercepat waktu tanggap untuk pengguna akhir, desain asinkron juga mendorong arsitektur terdesentralisasi yang lebih toleran terhadap kegagalan komponen tunggal, karena interaksi antar-layanan dapat diatur melalui kontrak pesan yang loose-coupled.

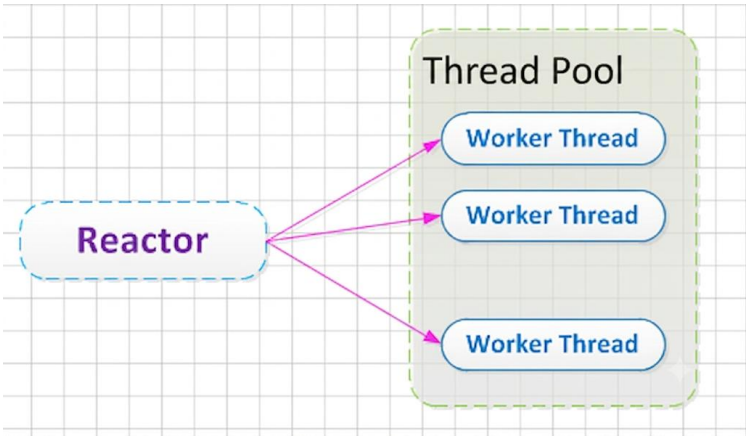


Gambar 2. 2 Asynchronous

Gambar 2.2 memperlihatkan representasi konseptual aliran komunikasi asinkron antara Service A dan Service B, di mana request ditempatkan pada antrean yang diterima oleh Service B dan response dikembalikan melalui antrean terpisah ke Service A. Skema ini memperlihatkan sifat non-blocking, pengirim tidak harus menunggu penyelesaian pemrosesan, melainkan melanjutkan eksekusi sambil menunggu event selesai.

2.2.3 Twisted Framework

Twisted berfokus pada model pemrograman berbasis *event*, dimana alur eksekusi program ditentukan oleh peristiwa yang terjadi, seperti penerimaan data dari jaringan atau interaksi pengguna. Twisted menggunakan konsep *"Deferred"* untuk menangani operasi yang memerlukan waktu, memungkinkan pengembang untuk menulis kode yang lebih bersih dan lebih mudah dibaca tanpa harus menggunakan *callback* yang rumit. Dengan arsitektur yang modular, Twisted mendukung berbagai protokol jaringan, termasuk HTTP, SMTP, dan FTP, serta menyediakan sistem untuk membangun *server* dan klien yang dapat berkomunikasi secara efisien. Keunggulan utama dari Twisted adalah kemampuannya untuk menangani ribuan koneksi secara bersamaan dengan penggunaan sumber daya yang minimal, menjadikannya pilihan yang ideal untuk aplikasi yang memerlukan skalabilitas tinggi.

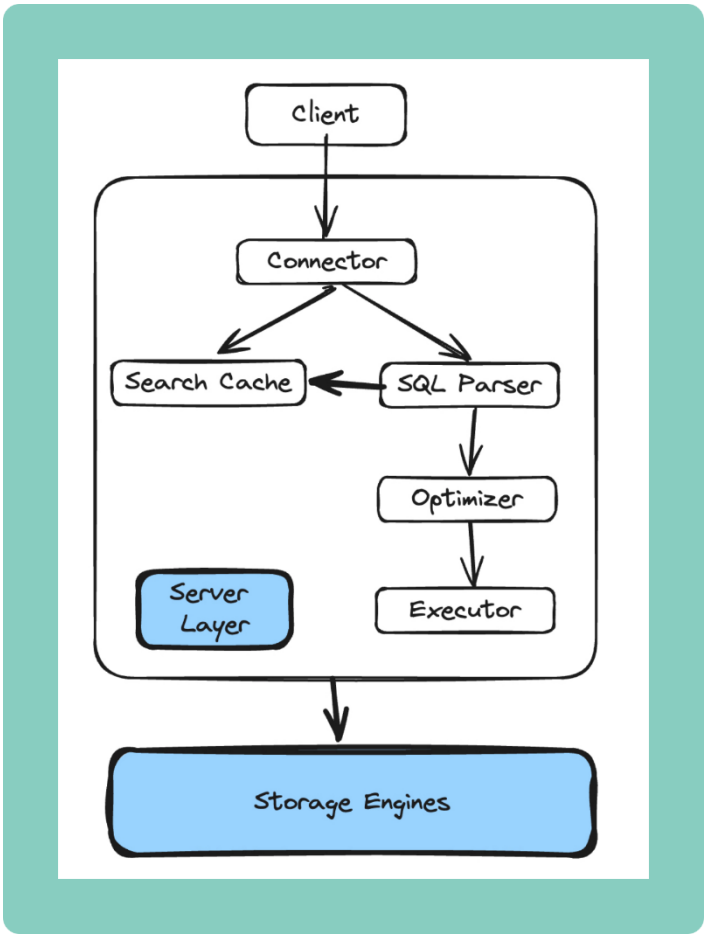


Gambar 2. 3 Arsitektur Twisted Framework

Gambar 2.3 menunjukkan arsitektur Twisted Framework. Di pusat terdapat komponen Reactor yang menerima dan mendeteksi peristiwa I/O lalu mengarahkan penanganan event tersebut. Untuk tugas-tugas yang bersifat blocking atau intensif komputasi, Reactor mendelegasikan pekerjaan ke kumpulan worker thread yang berada di dalam Thread Pool, sehingga loop event utama tidak terblokir. Hubungan antara Reactor dan Worker Thread menggambarkan alur delegasi tugas dan callback ke Reactor. Skema ini menekankan sifat event-driven (non-blocking) untuk I/O jaringan sekaligus menyediakan jalur terkontrol untuk mengeksekusi operasi blocking, sehingga aplikasi memperoleh antara latensi rendah pada operasi I/O dan kemampuan menangani pekerjaan berat secara terisolasi.

2.2.4 Database

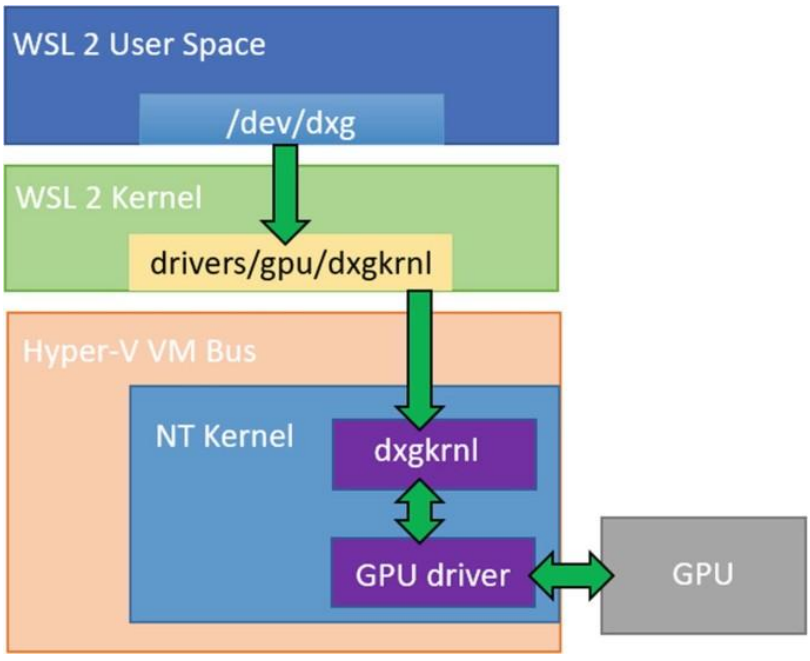
Database adalah sistem manajemen basis data relasional yang dirancang untuk menyimpan, mengelola, dan mengambil data secara efisien. Data disimpan dalam struktur tabel yang saling terhubung melalui kunci primer dan kunci asing, memungkinkan integrasi data yang kuat dan akses yang cepat. Dengan dukungan untuk *Structured Query Language* (SQL), pengguna dapat melakukan operasi kompleks pada data dengan mudah. *Database* juga menawarkan fitur seperti replikasi, keamanan yang ditingkatkan, dan kemampuan untuk menangani transaksi, yang memastikan integritas data. Sebagai solusi yang handal dan fleksibel, *database* menjadi komponen penting dalam berbagai aplikasi *web* dan layanan *online*.



Gambar 2. 4 Arsitektur Database

2.2.5 Windows Subsystem For Linux (WSL)

Windows Subsystem for Linux (WSL) adalah fitur pada sistem operasi Windows yang memungkinkan pengguna untuk menjalankan lingkungan GNU/Linux termasuk sebagian besar perangkat lunak berbasis *command-line*, utilitas, dan aplikasi secara langsung di Windows, tanpa modifikasi dan tanpa membebani sistem dengan perangkat lunak mesin virtual tradisional atau konfigurasi *dual-boot*, WSL adalah *virtual machine* yang berjalan di atas *platform hypervisor* bawaan Windows. Pembaruan arsitektur ini memberikan kernel Linux seutuhnya, sehingga menghasilkan kompatibilitas *system call* yang penuh dan kinerja operasi *file system* yang jauh lebih cepat dibandingkan metode translasi perintah pada versi sebelumnya. Dalam konteks pengembangan sistem dan pengelolaan *server*, WSL memberikan efisiensi tinggi karena memungkinkan eksekusi *script* atau perangkat berbasis Linux (seperti *broker* pesan, basis data, dan *web server*) secara lokal dengan latensi jaringan internal yang minim terhadap sistem *host,* pada gambar 2.5 dapat dilihat cara kerja WSL pada Windows.



Gambar 2. 5 Arsitektur Windows Subsystem For Linux

Pada Gambar 2.5 dapat dilihat cara kerja WSL pada Windows, khususnya dalam mengelola komunikasi lintas sistem untuk mengakses perangkat keras grafis (GPU). Diagram tersebut mengilustrasikan alur perintah dari dalam lingkungan Linux menuju perangkat keras host Windows secara bertahap:

1. WSL 2 User Space: Linux memulai permintaan dengan mengakses *device node* /dev/dxg yang berada di level pengguna.
2. WSL 2 Kernel: Instruksi diteruskan ke kernel Linux milik WSL 2 untuk diproses oleh modul driver internal drivers/gpu/dxgkrnl.
3. Hyper-V VM Bus: Kernel Linux kemudian mengirimkan instruksi tersebut menyeberangi jalur komunikasi virtualisasi Hyper-V untuk mencapai sistem *host* Windows.
4. NT Kernel: Di dalam kernel host Windows, instruksi diterima oleh komponen subsistem DirectX Graphics Kernel (dxgkrnl) yang selanjutnya berinteraksi dengan *GPU driver* bawaan sistem operasi.
5. *GPU Hardware*: Pada tahap akhir, *GPU driver* berkomunikasi secara langsung dengan perangkat keras GPU fisik.

Arsitektur ini menunjukkan bagaimana WSL 2 tidak hanya berfungsi untuk kebutuhan komputasi prosesor standar, tetapi juga dirancang untuk melakukan *passthrough* instruksi grafis secara efisien. Hal ini memungkinkan aplikasi Linux untuk sepenuhnya memanfaatkan akselerasi GPU perangkat keras host dengan bantuan jembatan komunikasi di dalam Hyper-V.