Analisis Dampak Interaksi Heuristik Greedy terhadap Kinerja dan Skalabilitas Sistem Microservices

Subtitle as needed ***(paper subtitle)***

Albertus Christian Poandy - 13523077
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: achrisp05@gmail.com , 13523077@std.stei.itb.ac.id

***Abstract*—This electronic document is a "live" template and already defines the components of your paper [title, text, heads, etc.] in its style sheet. **CRITICAL: Do Not Use Symbols, Special Characters, or Math in Paper Title or Abstract.* (*Abstract*)**

Keywords—component; formatting; style; styling; insert (key words)

I. Latar Belakang

Perkembangan teknologi digital yang pesat telah mendorong transformasi fundamental dalam desain perancangan dan Pembangunan perangkat lunak. Aplikasi modern, mulai dari platform *e-commerce* hingga layanan *streaming media*, menuntut tingkat ketersediaan, skalabilitas, dan kecepatan pengembangan yang belum pernah terjadi sebelumnya. Untuk menjawab tantangan ini, industri perangkat lunak secara bertahap beralih dari arsitektur *monolith*, yaitu arsitektur dengan seluruh aplikasi dibangun sebagai satu unit tunggal, menuju arsitektur *microservices*. Arsitektur *microservices* menawarkan pendekatan modular dengan memecah apliikasi menjadi kumpulan layanan kecil yang independen, yang memungkinkan pengembangan yang lebih lincah, tingkat skalabilitas yang tinggi, dan fleksibilitas teknologi yang lebih besar.

Meskipun memberikan banyak keuntungan, sifat arsitektur *microservices* yang terdistribusi juga memperkenalkan serangkaian tantangan teknis yang kompleks. Dua tantangan utama yang sering kali menonjol adalah menjaga kualitas kinerja (*performance*) dan skalabilitas (*scalability*) sistem. *Performance*, yang sering diukur melalui latensi respons, menjadi krusial untuk pengalaman pengguna yang baik. Sementara itu, skalabilitas, yaitu kemampuan sistem untuk menangani peningkatan beban kerja tanpa adanya penurunan kinerja, adalah kunci untuk pertumbuhan bisnis. Mengelola kedua aspek ini dalam ekosistem yang terdiri dari puluhan bahkan ratusan layanan yang saling berkomunikasi adalah sebuah persoalan yang signifikan dalam proses rekayasa perangkat lunak.

Untuk mengatasi tantangan kinerja dan skalabilitas tersebut, berbagai strategi dan algoritma diterapkan pada komponen-komponen kunci yang berada dalam sistem. Dalam konteks arsitektur *microservices*, dua mekanisme yang memainkan peran sentral adalah *load balancing* dan *caching*. *Load balancing* bertanggung jawab untuk mendistribusikan lalu lintas permintaan secara efisien ke beberapa instansi layanan, sehingga mencegah adanya suatu instansi yang kelebihan beban, serta menjamin ketersediaan sistem. Di sisi lain, *caching* digunakan untuk menyimpan data yang sering diakses pada lokasi penyimpanan berkecepatan tinggi, dengan tujuan mengurangi waktu akses data secara drastis dan meringankan beban pada komponen *backend* seperti database, sehingga meningkatkan kinerja sistem secara keseluruhan.

Dalam implementasi praktisnya, baik *load balancing* maupun *caching* seringkali mengandalkan heuristik *greedy* untuk pengambilan keputusan yang cepat. Heuristik *greedy* sering kali dipilih karena kecepatannya dalam eksekusi, yang sangat penting dalam sistem yang membutuhkan tingkat latensi rendah, yaitu melalui pengambilan keputusan-keputusan yang optimal secara lokal dan dibuat dengan informasi yang terbatas pada saat itu.

Namun, optimasi lokal ini dapat menimbulkan konsekuensi ketika berinteraksi dalam sebuah sistem yang utuh. Oleh karena itu, penting untuk menganalisis bagaimana heuristik greedy diterapkan dalam konteks sistem *microservices*, serta mengevaluasi keuntungan dan keterbatasannya dalam kebutuhan sistem yang bersifat dinamis dan kompleks. Melalui makalh ini, penulis bertujuan untuk menganalisis berbagai skenario penggunaan heuristik *greedy* dalam sistem *microservices* dan mengevaluasi pengaruhnya terhadap kinerja serta kemampuan sistem untuk menangani skala besar.

II. Dasar Teori

A. Arsitektur Microservices

Arsitektur *microservices* adalah sebuah gaya arsitektur perangkat lunak yang menstrukturkan sebuah aplikasi sebagai kumpulan layanan kecil (*services*) yang terhubung secara longgar (*loosely coupled*). Setiap layanan dalam arsitektur ini dirancang untuk menjalankan satu fungsi bisnis yang spesifik, memiliki basis kodenya sendiri, mengelola datanya sendiri, dan dapat dikembangkan serta di-*deploy* secara independen dari layanan lainnya [1]. Pendekatan ini kontras dengan arsitektur

monolit tradisional, yaitu arsitektur yang seluruh aplikasinya dibangun sebagai satu unit tunggal.

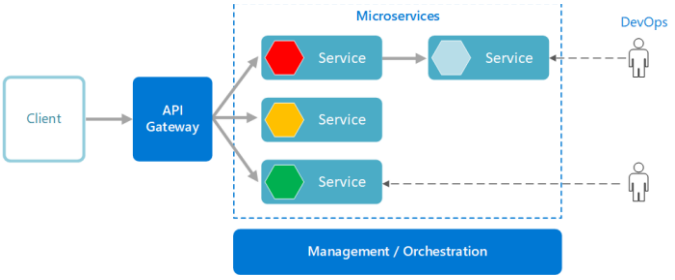

Fig. 1. Model Arsitektur Microservices (Sumber: https://learn.microsoft.com/id-id/azure/architecture/guide/architecture-styles/microservices)

Terdapat beberapa keuntungan utama dari arsitektur *microservices* [2]. Pertama, arsitektur ini mendukung heterogenitas teknologi, yaitu setiap layanan dalam sistem dapat menggunakan teknologi yang berbeda sesuai kebutuhan dan tujuan kinerjanya . Kedua, microservices memberikan ketahanan terhadap kegagalan (resilience), yaitu apabila satu layanan mengalami kegagalan, hal tersebut tidak secara langsung mempengaruhi keseluruhan sistem. Ketiga, proses penskalaan lebih fleksibel dibandingkan aplikasi monolitik, karena hanya layanan yang membutuhkan peningkatan kapasitas yang akan diskalakan, bukan seluruh sistem. Pendekatan ini dapat mengurangi penggunaan sumber daya perangkat keras secara keseluruhan. Keunggulan keempat adalah kemudahan dalam proses deployment, karena setiap layanan dapat di-*deploy* secara independen tanpa mengganggu performa layanan lainnya.

Komponen kunci dalam arsitektur *microservices* yang relevan dalam makalah ini meliputi:

- *API Gateway*: Komponen yang bertindak sebagai pintu masuk tunggal (*singgle entry point*) untuk seluruh permintaan dari klien. API Gateway meneruskan permintaan ke layanan internal yang sesuai dan seringkali juga bertanggung jawab untuk tugas-tugas seperti autentikasi, *rate limiting*, dan *load balancing*.
- *Service Instance*: Sebuah proses yang berjalan dari sebuah *microservice*. Untuk mencapai skalabilitas dan ketersediaan tinggi, sebuah layanan umumnya dijalankan dalam beberapa instansi.
- *Service Discovery*: Sebuah mekanisme yang berfungsi sebagai tempat informasi dinamis terkait lokasi layanan-layanan lainnya. Ketika sebuah layanan perlu berkomunikasi dengan layanan lain, ia akan menanyakannya pada *service discovery* untuk mendapatkan alamat jaringan dari instansi layanan yang tersedia.

Meskipun menawarkan keuntungan yang besar dalam hal skalabilitas dan kelincahan, sifat tersitribusi dari arsitektur *microservices* juga menimbulkan tantangan, terutama dalam hal latensi jaringan dan kompleksitas operasional, yang mendorong perlunya mekanisme optimasi seperti *caching* dan *load balancing*.

B. *Algoritma Greedy*

Algoritma *greedy* adalah salah satu metode perancangan algoritma yang paling populer dan sederhana untuk memecahkan persoalan optimasi, yaitu persoalan yang bertujuan mencari solusi optimal [3]. Sesuai dengan namanya, algoritma ini memiliki sifat "rakus" atau "tamak". Prinsip utama yang mendasarinya dapat diringkas sebagai "take what you can get now!", yang berarti algoritma ini membuat pilihan terbaik yang bisa diperoleh pada saat itu juga, tanpa memerhatikan konsekuensi dari pilihan tersebut di masa depan [3].

Pendekatan *greedy* membangun solusi secara bertahap, *step-by-step*. Pada setiap langkah, algoritma akan membuat keputusan yang dianggap sebagai optimum lokal (*local optimum*) dengan harapan bahwa rangkaian keputusan optimum lokal ini pada akhirnya akan mengarah pada solusi optimum global (*global optimum*). Sebuah keputusan yang telah diambil pada suatu langkah tidak dapat diubah kembali pada langkah-langkah berikutnya.

Elemen-elemen dari algoritma *greedy* meliputi: himpunan kandidat (C) yang berisi kandidat yang akan dipilih pada setiap langkah, himpunan solusi (S) yang berisi kandidat yang sudah dipilih, fungsi solusi yang menentukan apakah himpunan kandidat yang dipilih sudah memberikan solusi, fungsi seleksi (*selection function*) yang memilih kandidat berdasarkan strategi *greedy* tertentu, fungsi kelayakan (*feasible*) yang memeriksa apakah kandidat yang dipilih dapat dimasukkan ke dalam himpunan solusi, serta fungsi objektif yang memaksimumkan atau meminimumkan [3].

Untuk mengilustrasikan cara kerja dan keterbatasan dari algoritma *greedy*, dapat digunakan contoh klasik Persoalan Penukaran Uang (*Coin Exchange Problem*). Persoalan ini bertujuan untuk menukar sejumlah uang A dengan jumlah koin seminimal mungkin dari himpunan koin yang tersedia. Strategi *greedy* yang intuitif dan sering dipilih adalah: "pada setiap langkah, pilih koin dengan nilai terbesar dari himpunan koin yang tersisa yang tidak menyebabkan nilai total melebihi A".

Sebagai contoh, jika tersedia koin $\{1, 5, 10, 25\}$ dan uang yang ditukar adalah $A = 32$, strategi *greedy* akan menghasilkan solusi optimal: $32 = 25 + 5 + 1 + 1$ (total 4 koin digunakan). Namun, algoritma *greedy* tidak selalu menjamin solusi optimal. Hal ini dapat ditunjukkan melalui sebuah *counterexample* pada persoalan serupa. Jika himpunan koin yang tersedia adalah $\{1, 3, 4\}$ dan uang yang ditukar adalah $A = 6$, strategi *greedy* akan memilih koin $6 = 4 + 1 + 1$ (total 3 koin digunakan). Padahal, solusi optima yang sebenarnya adalah $6 = 3 + 3$ (total 2 koin digunakan). Kegagalan ini terjadi karena algoritma *greedy* tidak beroperasi secara menyeluruh terhadap semua kemungkinan solusi dan terpaku pada optimum lokal. Oleh karena itu, untuk persoalan-persoalan yang algoritma *greedy*-nya tidak terbukti optimal, solusi yang dihasilkannya disebut sebagai solusi sub-optimal atau *approximation*.

C. *Heuristik Greedy pada Mekanisme Caching*

Kinerja sebuah layanan dalam arsitektur *microservices* sangat dipengaruhi oleh latensi akses data. Panggilan ke database atau layanan eksternal lainnya adalah operasi yang mahal dan lambat. Untuk menyelesaikan permasalahan ini, digunakan mekanisme *caching*. *Caching* adalah teknik menyimpan salinan data yang sering diakses di lokasi penyimpanan sementara yang berkecepatan tinggi (misalnya, di dalam memori server aplikasi itu sendiri). Ketika permintaan untuk data tersebut datang lagi, layanan dapat mengambilnya langsung dari *cache* (*cache hit*), yang jauh lebih cepat daripada mengambilnya dari sumber asli (*cache miss*).

Secara intuitif, agar tercapai kecepatan akses data yang tinggi di *cache*, ukuran data yang disimpan tidak boleh terlalu besar. Maka, tantangan fundamental dari *caching* adalah ukurannya yang terbatas. Ketika *cache* penuh dan ada data baru yang perlu disimpan, sistem harus membuat keputusan tentang data mana yang akan "dibuang" atau dikeluarkan (*eviction*). Proses pengeluaran ini diatur oleh *eviction policy* (kebijakan eviksi). Kebijakan-kebijakan ini seringkalli menggunakan heuristik *greedy*. Heuristik-heuristik yang sering digunakan adalah *Least Recently Used* (LRU), *Least Frequently Used* (LFU), dan *First-In First-Out* (FIFO).

Kebijakan dengan heuristik *Least Frequently Used* (LFU) adalah kebijakan yang mengeluarkan item yang paling lama tidak diakses. Heuristik ini sangat efektif untuk beban kerja yang menunjukkan prinsip lokalitas temporal (*temporal locality*), yaitu data yang baru saja diakses kemungkinan besar akan segera diakses lagi. Implementasinya juga relatif efisien. Namun, LRU akan berkinerja sangat buruk ketika menghadapi pola akses yang siklik atau pemindaian besar (*large scan*). Bayangkan sebuah proses yang membaca seluruh entri dari tabel besar di database yang ukurannya relatif jauh lebih besar daripada ukuran *cache*. Setiap data hanya akan diakses sekali, sehingga proses ini pada dasarnya hanya akan "membersihkan" isi seluruh *cache*, membuang semua data "panas" yang berguna yang sebelumnya ada di sana, dan menggantinya dengan data yang berpotensi untuk tidak akan digunakan lagi. Keputusan *greedy* untuk membuang data yang "lama" menjadi tidak efektif dalam kasus ini.

Kebijakan dengan heuristik *Least Frequently Used* (LFU) adalah kebijakan yang mengeluarkan item dengan frekuensi akses paling rendah berdasarkan sebuah *counter* (penghitung) sederhana. Secara intuitif, LFU terasa lebih baik daripada LRU karena ia mempertahankan item yang populer (sering diakses) meskipun sudah lama tidak diakses. Hal ini efektif untuk data referensi yang sering digunakan. Namun, LFU juga memiliki masalah, yaitu item yang dulunya sangat populer tetapi tidak lagi relevan sekarang bisa tetap berada di *cache* untuk waktu yang sangat lama karena memiliki frekuensi akses yang tinggi. Fenomena ini disebut *cache pollution*. Akibatnya, item baru yang berpotensi memiliki frekuensi akses tinggi tidak bisa masuk ke dalam *cache*. Selain itu, implementasi LFU yang efisien juga lebih kompleks daripada LRU.

Kebijakan dengan heuristik *First-In First-Out* (FIFO) adalah kebijakan yang mengeluarkan item yang pertama kali masuk, seperti sebuah antrean (*queue*) biasa. Implementasinya sangatlah sederhana dan paling cepat, hampir tidak memerlukan *overhead*. Ini adalah kebijakan yang paling "naif", ia sama sekali tidak mempertimbangkan pola akses. Sebuah item yang sangat populer dan sering diakses bisa dengan mudah dikeluarkan hanya karena ia adalah salah satu item pertama yang dimuat ke dalam *cache*.

Dalam arsitektur *microservices*,setiap instansi layanan sering kali memiliki *cache* pribadinya sendiri, sehingga kebijakan-kebijakan *greedy* ini beroperasi secara terisolasi. Efektivitas *cache* pada sebuah server sangat bergantung pada jenis permintaan yang ia terima, yang pengaturannya merupakan tugas dari mekanisme *load balancing* yang akan dibahas pada sub-bab berikutnya.

D. *Heuristik Greedy pada Mekanisme Load Balancing*

Dalam arsitektur *microservices*, skalabilitas pada dimensi horizontal, yaitu kemampuan untuk enambah atau mengurangi jumlah instansi layanan sesuai dengan beban kerja, adalah sebuah aspek kunci. Untuk dapat memanfaatkan beberapa instansi layanan secara efektif, diperlukan sebuah mekanisme yang mengatur beban kerja yang akan dilimpahkan pada tiap layanan, yaitu mekanisme *load balancing*. *Load balancer* bertindak sebagai pengontrol yang mendistribusikan permintaan masuk ke salah satu dari banyak server yang tersedia di *service pool*. Tujuannya adalah untuk memastikan tidak ada satu server pun yang kelebihan beban, meningkatkan ketersediaan sistem, dan mengoptimalkan waktu respons.

Karena kebutuhan untuk membuat keputusan perutean dalam hitungan milidetik, banyak algoritma *load balancing* mengadopsi pendekatan heuristik *greedy*. Pendekatan ini membuat keputusan yang optimal secara lokal berdasarkan metrik yang tersedia pada saat itu juga. Beberapa strategi heuristik *greedy* yang umum digunakan adalah *Least Connection*, *Geographic Routing*, dan *Least Response Time* [4].

Least Connections adalah strategi *greedy* dengan heuristik yang mengarahkan permintaan baru ke server yang saat ini memiliki jumlah koneksi aktif paling sedikit. Implementasinya sederhana dan sangat efektif dalam mendistribusikan lalu lintas secara merata (dalam skenario di mana semua permintaan dianggap setara dan semua server identik). Strategi ini mencegah suatu server menjadi *bottleneck*. Namun, kelemahan utama dari strategi ini adalah "buta" terhadap konteks permintaan. Strategi ini tidak peduli data apa yang diminta. Contoh situasinya adalah: bayangkan sebuah permintaan untuk data *user-a* tiba. *Load balancer* mengirimnya ke Server 1 yang menyebabkan *cache miss*. Sesaat kemudian, terdapat lagi permintaan untuk data *user-a* yang sama. Jika Server 2 kebetulan memiliki koneksi lebih sedikit, *load-balancer* dengan strategi ini akan mengirim permintaan ke Server 2, yang lagi-lagi menyebabkan *cache miss*, padahal jika dikirim ke Server 1 akan menjadi lebih optimal karena *cache hit*. Keputusan *greedy* untuk menyeimbangkan koneksi justru mengorbankan efisiensi *cache* secara global.

Strategi *Geographic Routing*, yang sering digunakan oleh *Content Delivery Network* (CDN), mengarahkan permintaan pengguna ke pusat data atau server yang secara geografis paling dekat dengan lokasi mereka. Strategi ini sangat efektif dalam mengurangi latensi jaringan, karena memperpendek jarak fisik yang harus ditempuh data. Ini memberikan pengalaman

pengguna yang jauh lebih cepat untuk aplikasi skala global. Namun, kelemah utamanya adalah strategi ini buta terhadap beban aktual server. Contohnya, bayangkan terjadi sebuah *event* yang viral di Asia Tenggara. *Geographic routing* secara *greedy* akan mengirim semua pengguna dari wilayah tersebut ke pusat data di Singapura. Akibatnya, server di Singapura bisa kelebihan beban dan gaagal, padahal pusat data di Australia yang sedikit lebih jauh mungkin sedang tidak sibuk dan dapat memberikan layanan yang lebih stabil. Keputusan *greedy* untuk memilih yang terdekat justru dapat menurunkan kualitas layanan secara keseluruhan.

Strategi *Least Response Time* adalah strategi yang secara dinamis mengukur waktu respons dari setipa server dan mengarahkan permintaan baru ke server yang saat ini merespons paling cepat. Stratagi ini sangat adaptif, yaitu jika suatu server melambat karena tugas berat, strategi ini akan secara otomatis menghindarinya. Namun, strategi ini rentan terhadap efek "Thundering Herd" atau osilasi. Contohnya: bayangkan Server 1, karena memiliki *cache hit rate* yang tinggi, menjadi sangat cepat. *Load balancer* secara greedy akan mengalihkan semua lalu lintas ke Server 1. Akibatnya, Server 1 menjadi kelebihan beban dan melambat. Kemudian, Server 2 (yang tadinya diam) menjadi yang "tercepat", dan *load balancer* pun tiba-tiba mengalihkan semua lalu lintas ke Server 2. Perilaku ini dapat menciptakan ketidakstabilan beban kerja.

Pemilihan strategi *load balancing* yang tepat memiliki dampak signifikan terhadap keseluruhan sistem. Seperti yang telah diilustrasikan, keputusan *greedy* yang diambil oleh *load balancer* dapat secara langsung memengaruhi efektivitas mekanisme lain seperti *caching*. Interaksi antara pilihan *greedy* pada *load balancing* dan pada kebijakan eviksi *cache* inilah yang akan menjadi pusat analisis eksperimental pada makalah ini.

III. METODOLOGI EKSPERIMEN

Bab ini akan menguraikan secara rinci rancangan dan prosedur eksperimen sederhana yang dilakukan untuk menganalisis dampak interaksi antara berbagai heuristik *load balancing* dengan mekanisme *caching*. Eksperimen ini dilakukan melalui sebuah lingkungan simulasi terprogram yang dirancang khusus untuk memodelkan faktor-faktor kunci dalam sistem terdistribusi, seperti latensi I/O, beban server, dan latensi jaringan geografis.

A. Desain Sistem Simulasi

Lingkungan simulasi dirancang untuk meniru arsitektur *microservices* yang umum dan terdiri dari komponen-komponen berikut.

- API Gateway (Load Balancer): Berfungsi sebagai titik masuk tunggal yang menerima semua permintaan dan mendistribusikannya ke *pool* layanan. Logika *load balancing* yang diterapkan dapat diganti sesuai dengan skenario pengujian yang berjalan.
- Service Pool: Sebuah kelompok yang terdiri dari tiga instansi server identik yang ditempatkan pada lokasi geografis virtual yang berbeda. Lokasi geografis yang digunakan akan direpresentasikan dengan sebuah angka 0—100.
- Server Instance: Setiap instansi server dimodelkan dengan beberapa atribut kunci untuk menciptakan simulasi yang dekat dengan realita, yaitu terkait *cache pribadi* dan model waktu respons permintaan. Setiap server akan memiliki *cache* pribadi dengan kapasitas 10 item. Kebijakan eviksi (*eviction policy*) yang digunakan dapat dikonfigurasikan per eksperimen. Model waktu respons untuk setiap permintaan dihitung berdasarkan tiga faktor utama:
  1. Waktu proses dasar, yaitu waktu yang dibutuhkan untuk operasi I/O, yaitu ditetapkan sebesar 5 ms jika terjadi *cache hit* dan 100 ms jika terjadi *cache miss.*
  2. Penalti beban server, yaitu penalti yang digunakan untuk mensimulasikan kejenuhan server, yang diimplementasikan dengan cara mengalikan waktu proses dasar dengan sebuah faktor beban, yang dihitung sebagai $1 + (jumlah\ koneksi\ aktif * 0.05)$. Ini berarti setiap koneksi aktif akan memperlambat server sebesar 5%.
  3. Latensi geografis, yaitu latensi jaringan yang didapatkan berdasarkan jarak virtual antara klien dan server, yang didapatkan melalui perhitungan $|lokasi\ server - lokasi\ klien| * 0.01$ ms.

B. Skenario Beban Kerja

Untuk menguji bagaimana strategi yang berbeda berkinerja di bawah kondisi yang beragam, tiga jenis skenario beban kerja (*workload*) yang ekstrem dirancang.. Setiap skenario terdiri dari total 1000 permintaan. Kasus-kasus yang diuji antara lain:

- Fokus Item Panas: Skenario ini mensimulasikan beban kerja dengan afinitas data yang tinggi. Sebanyak 80% dari total permintaan akan mengakses hanya 20% dari variasi data yang ada (disebut "item panas"). Skenario ini dirancang untuk menguji efektivitas mekanisme *caching*.
- Fokus lokasi: Skenario ini mensimulasikan lonjakan lalu lintas dari satu wilayah geografis tertentu. Sebanyak 80% dari total permintaan akan berasal dari satu rentang lokasi klien yang sempit. Skenario ini dirancang untuk menguji hasil dari strategi yang berbasis geografi.
- Terdistribusi Merata: Skenario ini merepresentasikan kondisi "kekacauan" yaitu tidak ada pola yang jelas. Baik item yang diminta maupun lokasi klien bersifat acak dan terdistribusi merata. Skenario ini dirancang untuk menguji kinerja dan keandalan setiap strategi.

C. Prosedur dan Strategi Pengujian

Eksperimen dilakukan secara sistematis dengan menjalankan seluruh kemungkinan kombinasi dari skenario beban kerja, strategi *load balancing*, dan kebijakan *cache*. Strategi *load balancing* yang diuji meliputi:

- Sticky Session: Strategi berbasis *hash* yang memastikan afinitas *cache* maksimal.

- Round Robin: Strategi statis yang menyebar beban secara bergiliran.
- Least Connections: Strategi *greedy* yang memilih server berdasarkan beban koneksi saat ini.
- Least Response Time: Strategi *greedy* yang memilih server berdasarkan waktu respons tercepat.
- Geographic: Strategi *greedy* yang memilih server berdasarkan kedekatan geografis.

Kebijakan *cache* yang diuji meliputi:

- Least Recently Used (LRU): Kebijakan yang mengeluarkan item yang paling lama tidak diakses.
- Least Frequenty Used (LFU): Kebijakan yang mengeluarkan item dengan frekuensi akses paling rendah.
- First-In First-Out (FIFO): Kebijakan ini mengeluarkan item yang pertama kali masuk, dengan urutan seperti antrean (*queue*) biasa.

D. *Metrik Pengukuran*

Untuk mengevaluasi hasil dari setiap kombinasi eksperimen secara kuantitatif, dua metrik utama diukur dan dibandingkan:

- Rata-rata waktu respons (ms): Metrik kinerja utama yang merepresentasikan latensi rata-rata yang dialami oleh klien. Nilai yang lebih rendah menunjukkan kinerja yang lebih baik.
- Cache Hit Ratio (%): Metrik efisiensi sistem yang mengukur permintaan yang berhasil dilayani dari *cache*. Nilai yang lebih tinggi menunjukkan pemanfaatan *cache* yang lebih efektif.

Hasil dari metrik-metrik ini kemudian akan disajikan dalam bentuk tabel dan grafik untuk dianalisis lebih lanjut pada bab berikutnya.

IV. HASIL DAN ANALISIS

Bab ini akan menyajikan dan menganalisis data kuantitatif yang diperoleh dari lingkungan simulasi yang telah dijelaskan pada Bab III. Tujuan dari analisis ini adalah untuk mengevaluasi kinerja berbagai strategi *load balancing* dan kebijakan *cache* di bawah skenario beban kerja yang berbeda, serta untuk mengidentifikasi dampak dari interaksi antar komponen tersebut.

A. *Hasil*

Eksperimen dilakukan dengan menjalankan seluruh kombinasi dari 3 skenario beban kerja, 5 strategi *load balancing*, dan 3 kebijakan *cache*. Data yang dikumpulkan untuk metrik Rata-rata Waktu Respons dan Cache Hit Ratio disajikan pada Table 1. di bawah ini. Visualisasi data dari tabel ini juga disajikan pada Fig 2. untuk mempermudah interpretasi.

TABLE I. HASIL EKSPERIMEN SIMULASI

Kasus Workload	Strategi LB	Kebijakan Cache					
		LRU		*LFU*		*FIFO*	
		T (ms)	*%*	*T (ms)*	*%*	*T (ms)*	*%*
Fokus Item Panas	***Sticky Session***	755	36,2	711	43,4	771	33,4
	LC	902	12,8	882	15,6	894	12,1
	LRT	880	12,7	857	15,8	870	13,1
	Geographic	668	11,8	640	15,5	665	10,9
	Round Robin	915	13,9	908	14,2	904	12,5
Fokus Lokasi	***Sticky Session***	841	12,2	845	11,8	840	13,4
	LC	889	3,8	850	4,1	871	4,2
	LRT	808	3,8	815	4,3	810	3,5
	Geographic	734	3,5	736	4,5	745	3,4
	Round Robin	889	3,4	888	3,9	881	4
Terdistribusi Merata	***Sticky Session***	898	11,3	921	11,2	928	11
	LC	981	2,9	938	2,8	937	3,6
	LRT	937	2,4	940	2,8	922	3,9
	Geographic	713	3,7	718	3,2	710	4,6
	Round Robin	947	3,8	940	4,1	967	2,7

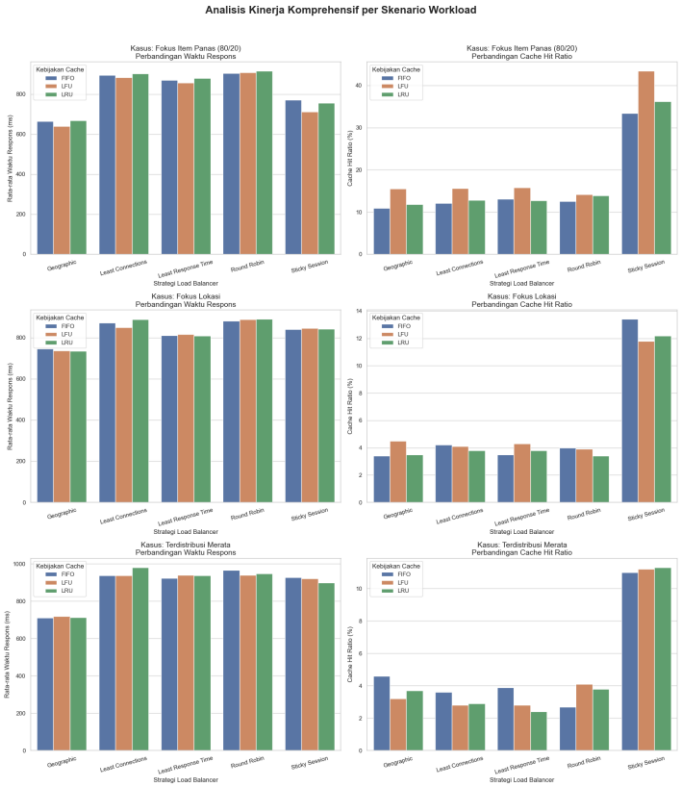

Fig. 2. Grafik Perbandingan Kinerja Komprehensif per Skenario Workload

B. Analisis Hasil Berdasarkan Skenario

1) Skenario Fokus Item Panas

Pada skenario ini, terungkap sebuah *trade-off* yang sangat menarik. Jika dilihat dari metrik *Cache Hit Ratio*, strategi Sticky Session adalah pemenang absolult dengan pencapaian hingga 43,4% (dengan LFU), jauh meninggalkan strategi lain yang hanya berkisar di 10-15%. Ini membuktikan Sticky Session berhasil melakukan tugas utamanya, yaitu memaksimalkan afinitas *cache*. Namun, jika dilihat dari metrik waktu respons, strategi Geographic justru menjadi yang terbaik dengan waktu ~640 ms, mengungguli Sticky Session (~711 ms). Hasil ini menunjukkan bahwa bahkan dengan keuntungan *cache hit* yang masif, Sticky Session tidak serta-merta menjadi yang tercepat. Ini mengimplikasikan bahwa kerugian akibat penalti beban server (karena memusatkan permintaan) dan potensi latensi geografis (jika *hash*mengarahkan ke server yang lebih jauh) cukup signifikan untuk menutupi sebagian besar keuntungan dari *cache hit*. Sebaliknya, Geographic unggul karena secara konsisten meminimalkan salah satu faktor penalti, yaitu latensi jaringan. Kemudian, kebijakan LFU secara jelas memberikan kinerja terbaik, karena kemampuannya untuk mengidentifikasi dan mempertahankan item-item "panas" yang sering diakses dalam skenario ini, manghasilkan *cache hit ratio* tertinggi. Maka, untuk beban kerja dengan afinitas data tinggi, memaksimalkan *cache* adalah krusial, tetapi tidak menjamin kinerja terbaik jika tidak diimbangi dengan manajemen beban dan latensi jaringan.

2) Skenario Fokus Lokasi

Pada skenario ini, strategi Geographic kini menjadi yang terbaik, dengan waktu respons terendah (T = 734 ms dengan LRU). Dalam skenario ini, karena item yang diminta bersifat acak, efektivitas *cache* menjadi minimal untuk semua strategi. Akibatnya, faktor dominan yang memengaruhi kinerja bergeser ke latensi jaringan. Geographic unggul karena ia secara konsisten meminimalkan penalti latensi geografis dengan mengarahkan pengguna ke server terdekat. Karena *cache hit ratio* secara umum sangat rendah (sekitar 3-4%), perbedaan kinerja antara LRU, LFU, dan FIFO menjadi tidak signifikan. Tidak ada cukup "data panas" untuk dipertahankan, sehingga "kecerdasan" dari LFU atau LRU tidak memberikan manfaat yang berarti. Sticky Session masih menunjukkan *cache hit* tertinggi, tetapi angka ini tidak cukup tinggi untuk mengkompensasi kerugian akibat perutean yang mungkin tidak optimal secara geografis. Maka, ketika afinitas data rendah tetapi afinitas lokasi tinggi, meminimalkan latensi jaringan melalui strategi *Geographic* menjadi strategi yang terbaik. Kebijakan *cache* memiliki dampak yang tidak terlalu signifikan dalam skenario ini.

3) Skenario Terdistribusi Merata

Mirip dengan fokus lokasi, strategi Geographic kembali menunjukkan waktu respons terbaik (T = 710 ms dengan FIFO), sementara sisanya memiliki performa yang mirip-mirip. Skenario ini merepresentasikan kondisi acak, yaitu tidak ada pola yang bisa dieksploitasi. Dalam kondisi ini, manfaat dari *cache* juga sangat minim. Oleh karena itu, faktor pembeda utama sekali algi adalah efisiensi dasar dari algoritma perutean. Geographic menunjukkan performa paling baik karena secara rata-rata ia akan selalu memilih jalur dengan latensi jaringan terendah bagi pengguna. Sama seperti fokus lokasi, *cache hit ratio* sangat rendah di semua kombinasi dan perbedaan antara LRU, LFU, dan FIFO sangat minimal dan tidak menunjukkan pola yang konsisten. Ini menegaskan bahwa dalam beban kerja yang benar-benar acak, kebijakan eviksi *cache* yang kompleks tidak memberikan keuntungan dibandingkan kebijakan yang sederhana sekalipun. Maka, di hadapan beban kerja yang acak, heuristik *greedy* untuk memilih jalur terpendek secara geografis tetap terbukti paling efektif.

C. Pembahasan

Analisis dari ketiga skenario menyimpulkan sebuah konsep fundamental dalam desain sistem terdistribusi, yaitu kinerja optimal adalah hasil dari keseimbangan antara tiga faktor yang saling bersaing: afinitas *cache*, distribusi beban, dan latensi jaringan. Tidak ada strategi tunggal yang dapat mengoptimalkan ketiganya secara bersamaan.

Ketika afinitas data tinggi, manfaat dari *cache* sangat signifikan. Namun, strategi yang hanya fokus pada hal ini dapat dikalahkan oleh strategi yang menyeimbangkan antara latensi jaringan dan beban, menunjukkan betapa pentingnya mempertimbangkan semua faktor. Ketika afinitas data rendah, manfaat *cache* menjadi minimal, dan perbandingannya beralih antara distribusi beban dan latensi jaringan. Dalam simulasi ini, dengan penalti beban yang tidak terlalu besar, meminimalkan latensi jaringan secara geografis terbukti lebih menguntungkan daripada sekadar menyebar beban. Kemudian, strategi dinamis seperti Least Connections dan Least Response Time tidak selalu menjadi pemenang. Mereka hanya unggul ketika beban server menjadi satu-satunya masalah paling dominan, sebuah kondisi yang tidak secara eksplisit menjadi faktor paling menentukan dalam ketika skenario yang diuji.

Lebih lanjut, penting untuk ditekankan bahwa hasil dari simulasi ini sangat dipengaruhi oleh parameter-parameter lingkungan yang telah ditetapkan, seperti waktu jika *cache miss* dan *cache hit*, bobot untuk penalti beban berlebih, faktor latensi geografis, cara perhitungan total latensi, jumlah server yang digunakan, jumlah item yang tersedia, jumlah request, dan lain-lain. Faktor-faktor ini merepresentasikan "biaya" dari setiap jenis penalti dalam sistem sesungguhnya. Perubahan pada rasio biaya ini akan secara langsung mengubah strategi mana yang terbukti paling optimal. Sebagai contoh, jika simulasi dijalankan pada sistem dengan penalti beban yang jauh lebih tinggi, yaitu merepresentasikan sistem yang sangat sensitif terhadap beban CPU, strategi yang unggul dalam mendistribusikan beban seperti Least Connection mungkin akan mengungguli strategi lainnya. Sebaliknya, jika sistem terhubung ke database yang lambat sehingga meningkatkan waktu penalti *cache miss*, keuntungan dari *cache hit* menjadi sangat krusial, sehingga Sticky Session memiliki potensi untuk menjadi pemenang yang lebih unggul. Parameter-parameter lingkungan yang dipilih pada eksperimen ini adalah nilai yang sebisa mungkin mereplikasi karakteristik dan rasio kinerja yang secara umum ditemui pada sistem aktual. Misalnya, rasio antara waktu *cache hit* (5 ms) dan waktu *cache miss* (100 ms) mencerminkan fakta fundamental bahwa akses memori (RAM) secara inheren *orders of magnitude* lebih cepat daripada akses I/O ke database melalui jaringan. Dengan mendasarkan simulasi pada parameter-parameter yang masuk akal dan representatif ini, diharapkan bahwa *trade-off* kompleks yang terungkap bukanlah hanya hasil artifisial

simulasi semata, melainkan juga cerminan dari situasi yang sesungguhnya terjadi pada sistem produksi aktual.

Secara keseluruhan, hasil simulasi ini dengan kuat mendukung hipotesis bahwa interaksi antara heuristik *greedy* bersifat kompleks. Keputusan desain yang efektif tidak dapat dibuat dengan hanya mengoptimalkan satu metrik secara terisolasi. Sebaliknya, seorang desainer sistem harus memahami karakteristik beban kerja yang diharapkan dan memilih strategi paling efektif dalam memitigasi faktor penalti yang paling dominan untuk skenario yang berkaitan. Kesimpulan yang dapat ditarik tidak hanya terbatas pada pemilihan strategi berdasarkan pola beban kerja, tetapi juga pada pentingnya melakukan *profiling* terhadap lingkungan sistem itu sendiri. Seorang desainer harus terlebih dahulu mengidentifikasi di mana letak *bottleneck* utama sistemnya sebelum dapat secara efektif menentukan strategi yang paling tepat untuk diterapkan.

V. KESIMPULAN

Setelah melakukan analisis terhadap dampak interaksi antara berbagai heuristik *greedy* yang diterapkan pada mekanisme *load balancing* dan *caching* dalam sebuah arsitektur *microservices* simulasi, yaitu melalui serangkaian eksperimen yang sistematis dengan berbagai skenario beban kerja, berhasil ditunjukkan bahwa pemilihan strategi optimasi tidak dapat dilakukan secara terisolasi, melainkan harus mempertimbangkan *trade-off* multidimensi antara afinitas *cache*, distribusi beban, dan latensi jaringan.

Kesimpulan utama dari eksperimen ini adalah bahwa tidak ada satupun strategi heuristik yang superior secara universal, kinerja optimal sangat bergantung pada karakteristik beban kerja yang dihadapi. Pada skenario fokus item panas, strategi Sticky Session terbukti paling unggul dalam memaksimalkan *cache hit ratio*, tetapi keuntungan ini tidak serta-merta menjadikannya yang tercepat karena dikalahkan oleh strategi *Geographic routing* yang lebih efektif dalam meminimalkan latensi jaringan. Sebaliknya, pada skenario fokus lokasi dan permintaan terdistribusi merata, yaitu skenario yang manfaat *cache* menjadi minimal, faktor latensi jaringan menjadi penentu utama, sehingga strategi *Geographic* menjadi pemenang yang jelas.

Secara keseluruhan, kontribusi utama dari eksperimen ini adalah pembuktian kuantitatif bahwa interaksi antara heuristik *greedy* pada sistem *microservices* bersifat kompleks dan seringkali kontra-intuitif. Keputusan desain yang efektif menuntut pemahaman mendalam dan tidak dapat dibuat dengan hanya mengoptimalkan satu metrik secara terisolasi. Sebagai saran untuk eksperimen selanjutnya, arah yang dapat dieksplorasi adalah melakukan analisis sensitivitas parameter yang lebih luas.

UCAPAN TERIMA KASIH

Terima kasih kepada Tuhan Yang Maha Esa karena berkat rahmat dan karunia-Nya, makalah yang berjudul "Analisis Dampak Interaksi Heuristik Greedy terhadap Kinerja dan Skalabilitas Sistem Microservices" dapat diselesaikan dengan lancar tanpa adanya hambatan. Terima kasih juga kepada dosen-dosen pengampu mata kuliah IF2211 Strategi Algoritma, terutama kepada Bapak Rinaldi Munir karena telah membagikan ilmu yang mendukung pembuatan makalah ini.

REFERENCES

[1] J. Lewis and M. Fowler, "Microservices: a definition of this new architectural term," 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html. [Diakses: Jun. 23, 2025].

[2] R. Munir, "Algoritma Greedy (Bagian 1)," Materi Kuliah IF2211 Strategi Algoritma, Program Studi Teknik Informatika, Institut Teknologi Bandung, 2024. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/04-Algoritma-Greedy-(2025)-Bag1.pdf. [Diakses: Jun. 23, 2025]

[3] O. Al-Debagy and P. Martinek, "A Comparative Review of Microservices and Monolithic Architectures," in 2018 18th IEEE International Symposium on Computational Intelligence and Informatics (CINTI), Nov. 2018, pp. 149-154.

[4] NGINX, Inc., "Load Balancing Methods," NGINX Administration Guide, 2024. [Online]. Available: https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/. [Diakses: Jun. 23, 2025]

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 1 Juni 2025

Nama dan NIM