Analisis Perbandingan Kinerja *Container Network Interface Flannel* dan *Cilium* sebagai *Interface* Utama pada Multus CNI dalam Jaringan Klaster *Kubernetes*

Bayu Agung Prakoso[1], Unan Yusmaniar Oktiawati[1,*]
[1]Departemen Teknik Elektro dan Informatika, Sekolah Vokasi, Universitas Gadjah Mada;
bayu.a@mail.ugm.ac.id
*Korespondensi: unan_yusmaniar@ugm.ac.id;

Abstract – Container technology has become an alternative for virtualizing internet service infrastructure due to resource efficiency. An IT infrastructure can consist of numerous containers, with Kubernetes acting as Container Orchestration. The Container Network Interface (CNI) is used in Kubernetes service scenarios to manage networks, thus facilitating interconnection of services. However, issues such as limited network capability, lack of flexibility, and scalability, as well as security, arise in the use of CNI plugins. The solution to these problems is the Multus CNI, which allows multiple network interfaces on a single pod. This study evaluates the performance of Flannel and Cilium as CNI plugins in a Kubernetes Cluster environment involving Multus CNI. The metrics analyzed include latency, packet loss, throughput, and CPU usage. The research results will provide a better understanding of the trade-offs that must be made when choosing between Flannel and Cilium as CNI plugins in a Kubernetes Cluster environment.

Keywords – Kubernetes Cluster, CNI, Flannel, Cilium, Multus CNI

Intisari – *Container* menjadi alternatif virtualisasi infrastruktur layanan internet berkat efisiensi penggunaan sumber daya. Infrastruktur IT dapat terdiri dari beragam *container*, dengan *Kubernetes* berperan sebagai *Container Orchestration*. *Container* Network *Interface* (CNI) dipergunakan dalam skenario layanan pada *Kubernetes* untuk mengatur jaringan sehingga memudahkan terhubungnya layanan. Namun, masalah seperti kemampuan jaringan terbatas, kurangnya fleksibilitas, dan terbatasnya skalabilitas serta keamanan menjadi isu dalam penggunaan CNI *plugin*. Solusi atas persoalan tersebut adalah Multus CNI yang memungkinkan beragam antarmuka jaringan pada satu *pod*. Studi ini melakukan evaluasi kinerja antara *Flannel* dan *Cilium* sebagai *plugin* CNI dalam lingkungan *Kubernetes Cluster* dengan melibatkan Multus CNI. Metrik yang dianalisis mencakup *latency, packet loss, throughput,* dan CPU *usage*. Hasil penelitian akan menghasilkan pemahaman lebih baik mengenai kompromi yang harus dilakukan saat memilih antara *Flannel* dan *Cilium* sebagai *plugin* CNI dalam lingkungan *Kubernetes Cluster*.

Kata kunci – *Kubernetes Cluster*, CNI, *Flannel, Cilium,* Multus CNI

I. PENDAHULUAN

Dalam dunia yang terus bertransformasi menuju digitalisasi, kebutuhan akan layanan internet yang efisien menjadi prioritas. *Cloud computing* dan teknologi pendukungnya telah mendorong munculnya teknologi virtualisasi, di mana satu perangkat dapat menjalankan lebih dari satu sistem operasi secara bersamaan. Dari teknologi virtualisasi, kemudian berkembanglah teknologi *container* yang memungkinkan penggunaan *kernel* yang sama atau yang dikenal sebagai *shared kernel*, menambah efisiensi penggunaan sumber daya.

Dalam konteks layanan seperti *microservices*, konektivitas antar *container* menjadi vital, sehingga diperlukan aplikasi untuk mengatur dan menghubungkan berbagai *container* menjadi suatu *cluster* dengan *container orchestration*. Di antara berbagai jenis *orchestration* yang tersedia, *Kubernetes* muncul sebagai pilihan utama karena kemampuannya dalam mengotomatisasi penyebaran, skala, dan manajemen aplikasi *container* [1].

Kubernetes merupakan sistem orkestrasi kontainer *open-source* yang otomatis melakukan penyebaran, penskalaan, dan manajemen aplikasi yang dikontainerisasi [2]. Sistem ini menyediakan cara untuk mengorganisir dan mengelola beban kerja yang dikontainerisasi dan layanan di dalam klaster mesin [2]. *Kubernetes* telah digunakan dalam berbagai bidang, seperti penyebaran aplikasi yang dikontainerisasi pada penyedia *cloud* [3], implementasi sistem manajemen perangkat lunak non-kontainer [4], pengamanan klaster *Kubernetes* [5], mempercepat proses migrasi antar *cloud* [6], serta mengevaluasi performa dan skalabilitas dalam penyebaran kontainer [7].

Namun, dalam konfigurasi jaringan *container*, *Kubernetes* bergantung pada *plugin Container* Network *Interface* (CNI). Dua implementasi CNI yang populer adalah *Cilium*, yang telah melihat pertumbuhan besar dalam kontribusi dan adopsi [8], dan *Flannel*, implementasi CNI pertama untuk *Kubernetes* yang masih menjadi pilihan yang baik untuk penggunaan awal [9]. Untuk menambah fleksibilitas dan memungkinkan penggunaan berbagai *plugin* CNI dalam satu *cluster Kubernetes*, Multus CNI menjadi solusi.

Tujuan dari penelitian ini adalah untuk membandingkan kinerja dari *Flannel* dan *Cilium* ketika digunakan di atas Multus CNI, termasuk evaluasi *latency*, *packet loss*, *throughput*, dan CPU *usage*. Kontribusi utama dari penelitian ini adalah memberikan wawasan tentang bagaimana performa kedua *plugin* CNI dalam kondisi yang sama, yang dapat membantu pengguna *Kubernetes* dalam memilih strategi

jaringan yang paling sesuai dengan kebutuhan mereka. Penelitian ini menjadi penting karena, meski *Flannel* dan *Cilium* telah banyak digunakan dalam lingkungan produksi, penelitian yang membandingkan kinerja keduanya ketika digunakan di atas Multus CNI masih terbatas.

II. DASAR TEORI

Dalam evolusi teknologi informasi dan komunikasi, teknologi kontainerisasi telah menjadi titik balik penting dalam proses pengembangan dan penyebaran aplikasi. Kemampuan kontainerisasi untuk mengemas aplikasi dan dependensinya dalam format yang dapat ditransfer antar lingkungan berbeda menunjukkan efisiensi yang signifikan dalam meningkatkan proses *provisioning*, *delivering*, dan *maintaining* infrastruktur. *Kubernetes*, menjadi pilar dalam orkestrasi kontainer, memegang peran kunci dalam menerapkan keuntungan ini pada skala produksi yang besar. Salah satu elemen vital dalam struktur *Kubernetes* adalah *Container* Network *Interface* (CNI), yang merupakan spesifikasi dan serangkaian perangkat lunak yang merancang bagaimana kontainer berkomunikasi dengan jaringan. Terdapat penelitian yang telah melibatkan analisis perbandingan kinerja dan aplikasi antara *Flannel*, *Cilium*, dan Multus CNI, yang relevan dan mendukung penelitian ini dalam memperluas pemahaman tentang cara kerja dan kinerja mereka dalam lingkungan *Kubernetes*.

Penelitian yang berjudul "*Assessing Container* Network *Interface Plugins*: *Functionality*, *Performance*, *and Scalability*" membahas dua metode utama yang digunakan untuk pengiriman paket yaitu eBPF dan solusi berbasis *bridge*. Dalam penelitian itu eBPF digunakan untuk pengiriman paket antar-*host* dan memiliki *overhead* yang lebih tinggi dibandingkan dengan pengiriman paket intra-*host* karena titik tambahan di *host* untuk mendukung mode *overlay* dan proses *tunneling* VxLAN. Sebaliknya, *Flannel*, *Weave*, dan *Kube-router* menggunakan solusi berbasis *bridge* untuk pengiriman paket intra-*host*, di mana paket melewati Linux *bridge* dan mengeksekusi panggilan fungsi yang terkait dengan *bridge* [10]. Penelitian tersebut juga melakukan evaluasi kinerja berbagai *plugin* CNI, termasuk *Flannel*, *Calico*, *Weave*, *Cilium*, dan Kube-router. Hasil evaluasi menunjukkan bahwa *Flannel* dan *Weave* memiliki *overhead* terendah untuk komunikasi intra-*host*, sementara *Calico* dan *Cilium* memiliki *overhead* tertinggi [10]. Untuk komunikasi antar-*host*, *Calico* dan *Cilium* memiliki *overhead* terendah, sementara *Flannel* dan *Weave* memiliki *overhead* tertinggi [10]. Kesimpulannya, penelitian tersebut menyatakan bahwa pilihan *plugin* CNI tergantung pada kasus penggunaan dan kebutuhan khusus, dan tidak ada solusi yang cocok untuk semua skenario [10].

Dalam penelitian yang berjudul "*Performance Studies of Kubernetes Network Solutions*", [11] melakukan penilaian terhadap empat *plugin* CNI yang direkomendasikan oleh CNCF dalam lingkungan *datacentre* fisik. Penilaian ini bertujuan untuk mengevaluasi kinerja mereka dalam hal latensi dan rata-rata *throughput* TCP untuk berbagai ukuran *Maximum Transmission* Unit (MTU), jumlah berbeda dari antarmuka jaringan yang digabungkan, dan kondisi *offloading* segmen antarmuka yang berbeda. Selama pengujian, kinerja empat *plugin* CNI (*Flannel*, *Calico*, Kube-router, dan *Weave*) diukur dalam hal latensi dan rata-rata *throughput* TCP [11]. Dari segi kinerja, *Flannel* menunjukkan kinerja terbaik di antara *plugin overlay* yang diuji saat menggunakan MSS (*Maximum Segment Size*) yang kecil. Di sisi lain, *Calico* dan Kube-router, yang merupakan *plugin* non-*overlay*, menunjukkan *throughput* yang hampir sama dengan *throughput* baremetal [11]. Namun, penggunaan *Calico* dalam mode VXLAN memberikan hasil yang kurang memuaskan saat digunakan dengan ukuran MSS yang kecil [11]. Untuk *plugin Weave*, meskipun menggunakan teknologi *overlay* yang sama, performanya kurang dibandingkan *Flannel* [11]. Kesimpulannya, penelitian tersebut menyarankan bahwa kinerja *plugin* CNI sangat bergantung pada berbagai faktor seperti ukuran MTU, jumlah antarmuka jaringan yang digabungkan, dan kondisi *offloading* segmen antarmuka [11]. Pengguna disarankan untuk memilih *plugin* CNI berdasarkan kasus penggunaan dan persyaratan spesifik mereka [11].

Dalam penelitian mereka yang berjudul "*Network Policies in Kubernetes*: *Performance Evaluation and Security Analysis*", [12] mengevaluasi performa dan keamanan dari kebijakan jaringan dalam *Kubernetes*. Penelitian ini secara khusus menyoroti *overhead* performa dari solusi berbasis eBPF seperti *Calico* dan *Cilium*. Pengukuran *throughput* dan latensi *end-to-end* dilakukan menggunakan TCP *stream* mode dan *request-response* (RR) mode dari netperf, dengan tujuan untuk mendapatkan hasil pengukuran yang akurat [12]. Menurut hasil penelitian ini, kebijakan jaringan adalah solusi keamanan yang memiliki *overhead* rendah dan cocok untuk komunikasi *inter-container* yang membutuhkan latensi rendah. Penelitian ini juga menunjukkan bahwa *overhead* performa dari kebijakan jaringan hampir dapat diabaikan, dengan hanya sedikit variasi tergantung pada jumlah kebijakan dan berbagai resep kebijakan yang berbeda [12].

Penelitian yang berjudul "Multi-level Network *Software Defined Gateway Forwarding* System *Based on* Multus", mempresentasikan implementasi dan proposal sistem *gateway* penerusan yang ditentukan oleh perangkat lunak jaringan *multi*-level berdasarkan Multus. Metodologi yang diadopsi dalam penelitian ini mencakup penggunaan *Kubernetes* untuk manajemen terpusat dan kontrol *cluster* layanan, pemanggilan dinamis *plugin* CNI jenis berbeda untuk konfigurasi antarmuka, serta pendukung jaringan *multi*-level mode *kernel* dan mode pengguna [13]. Mereka melaksanakan tiga kelompok eksperimen komparatif untuk mengevaluasi sistem yang diusulkan. Hasilnya menunjukkan bahwa sistem yang diusulkan menunjukkan kinerja penerusan yang lebih tinggi dibandingkan perangkat *gateway* tradisional ketika dibatasi hingga 1GB memori yang tersedia [13]. Selain itu, penelitian ini menunjukkan bahwa sistem yang diusulkan memiliki skalabilitas yang baik, toleransi kesalahan, dan kemampuan penyeimbangan beban setelah pengenalan skema manajemen klaster *Kubernetes* [13].

Dalam penelitian yang berjudul "*Enhancement* in Multus CNI *for* DPDK *Applications* in *the Cloud Native Environment*", [14] mengusulkan penggunaan *plugin* Multus CNI dalam lingkungan *Kubernetes* untuk menyediakan berbagai antarmuka ke *pod*, dengan menggunakan konfigurasi jaringan dari *plugin* CNI sekunder. Penelitian ini memfokuskan pada implementasi kebijakan jaringan menggunakan *plugin* CNI sekunder untuk memproses paket data dalam aplikasi yang didukung DPDK yang berjalan di dalam kontainer [14]. Selain itu, penelitian ini juga mengusulkan integrasi CNI ruang pengguna dengan Multus CNI untuk memfasilitasi aplikasi yang didukung DPDK dengan menyediakan jaringan untuk pesawat kontrol. Hasil dari penelitian ini menunjukkan bahwa melalui penggunaan Multus CNI, *vhost-user* dapat digunakan oleh aplikasi DPDK untuk mengakses *stack* jaringan *kernel*. Pendekatan ini memberikan fleksibilitas dan skalabilitas pada penyebaran CNF dalam lingkup *cloud native* [14].

III. METODOLOGI

A. Alat dan Bahan

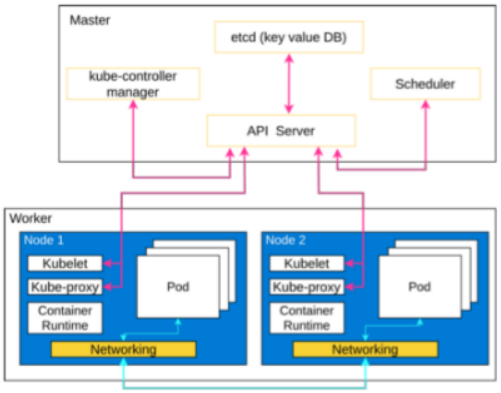
Dalam penelitian ini, digunakan beberapa perangkat keras dan perangkat lunak penting yang mendukung operasi dan hasil penelitian. Untuk perangkat keras, penelitian ini memanfaatkan klaster *Kubernetes* yang terdiri dari dua *node* pekerja dan satu *node* master. Spesifikasi untuk setiap *node* adalah 2 *vCPU*, 4 GB *memory*, dan *hard disk* 80 GB. Selain itu, penelitian ini juga menggunakan sebuah laptop sebagai titik akses dan kontrol utama, yang digunakan untuk mengakses layanan *cloud* dan menjalankan perangkat lunak yang dibutuhkan dalam penelitian. Spesifikasi laptop yang digunakan adalah sistem operasi Windows 10, CPU AMD *Ryzen* 5 5600H, 16 GB *memory*, dan *hard disk* 512 GB.

Untuk perangkat lunak, penelitian ini menggunakan beberapa perangkat berikut:

1. Ubuntu 20.04 x64: Sistem operasi berbasis Linux yang digunakan pada semua *node* dalam klaster.
2. *Kubernetes* versi 1.21: Platform open-*source* untuk orkestrasi kontainer yang digunakan untuk mengelola dan menjalankan aplikasi dalam kontainer.
3. Kubeadm: Alat yang membantu *mem-bootstrapping* klaster *Kubernetes*, memudahkan penyiapan dan konfigurasi komponen kunci *Kubernetes*.
4. *Python* versi 3.9.6: Bahasa pemrograman yang digunakan untuk menulis skrip otomatisasi pengaturan klaster dan menjalankan pengujian.
5. *Fabric*: Pustaka *Python* yang memudahkan otomatisasi perintah *shell* melalui SSH.
6. *Flannel*, *Cilium*, dan Multus CNI: *Plugin* CNI yang digunakan untuk memungkinkan komunikasi antar *pod* di klaster *Kubernetes*.
7. Iperf3: Alat untuk melakukan pengujian kinerja jaringan yang digunakan untuk mengevaluasi kinerja dari *plugin* CNI.

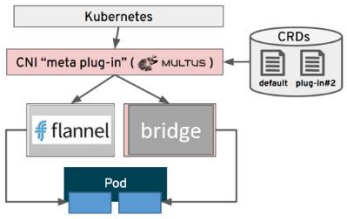
B. Perancangan Arsitektur

Penelitian ini merancang arsitektur dengan komponen utama *Kubernetes*: Master, *Worker*, dan *Networking* via CNI (*Container* Network *Interface*) seperti yang ditunjukkan oleh Gambar 1. Master, sebagai pengendali kluster, memiliki komponen ectd, *Scheduler*, *kube-controller-manager*, dan API Server. *Worker*, tempat menjalankan *pod*, terdiri dari *Kubelet*, *Container Runtime*, dan *Kube-proxy*.

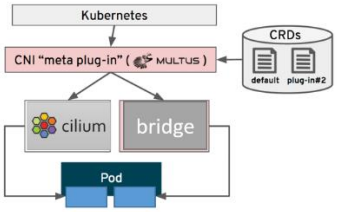


Gambar 1. Arsitektur *Kubernetes Cluster*

Model jaringan CNI diadopsi untuk *networking*, dengan menggunakan *Flannel*, *Cilium*, dan Multus CNI. *Flannel* dan *Cilium* digunakan secara bergantian di atas Multus CNI sesuai kebutuhan pengujian, sementara Multus CNI memungkinkan konfigurasi *multiple network interfaces* pada *pods* seperti yang ditunjukkan oleh Gambar 2 dan Gambar 3.

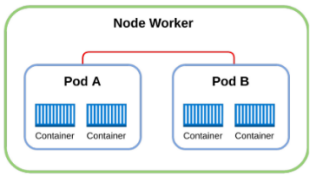


Gambar 2. Arsitektur *Flannel* di atas Multus CNI



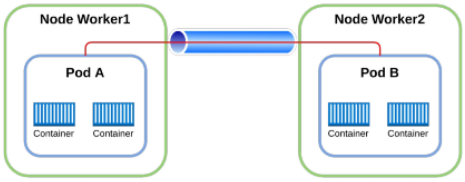
Gambar 3. Arsitektur *Cilium* di atas Multus CNI

Penelitian ini mengadopsi dua skenario pengujian: pengujian komunikasi antar *Pod* dalam satu *Node* seperti yang ditunjukkan oleh Gambar 4 dan antar *Pod* pada *Node* berbeda seperti yang ditunjukkan dalam Gambar 5. Skenario pertama relevan untuk kasus dengan ketergantungan sumber daya lokal atau spesifik, sementara skenario kedua membantu memaksimalkan ketersediaan dan ketahanan.



Gambar 4. Pengujian antar *Pod* dalam Satu *Node*

Tujuan pengujian ini adalah untuk mengevaluasi kinerja *Flannel* dan *Cilium* dalam penggunaan nyata, memberikan gambaran tentang performa mereka dalam berbagai situasi dan membantu dalam pemilihan solusi CNI terbaik untuk lingkungan produksi.



Gambar 5. Pengujian antar *Pod* berbeda *Node*

C. Pembuatan *Kubernetes Cluster*

Dalam penelitian ini, proses pembuatan kluster *Kubernetes* dilakukan dengan menggunakan *Python script* yang disokong oleh *library Fabric*. Berikut adalah langkah-langkahnya:

1. Pengunduhan Repositori: Kode sumber yang diperlukan dapat ditemukan di repositori *GitHub* pada *link* berikut: https://github.com/testdrivenio/kubernetes-fabric. Repositori ini dapat diunduh menggunakan perintah *clone Git*.
2. Penyiapan Lingkungan Virtual *Python*: Sebuah lingkungan virtual *Python* dibuat dan diaktifkan. Lingkungan ini menyediakan ruang kerja terisolasi untuk mengelola dependensi *Python*.
3. Pemasangan Dependensi: Semua dependensi yang diperlukan untuk menjalankan *script* dalam repositori diinstal menggunakan perintah *pip*.
4. Pendaftaran Akun Digital Ocean: Untuk menampung kluster, dibutuhkan server yang dalam konteks Digital Ocean disebut "*droplet*". Maka, perlu membuat akun di Digital Ocean.
5. Generasi *Token* Akses Digital Ocean: *Token* ini akan digunakan untuk otentikasi saat berinteraksi dengan API Digital Ocean.
6. Penambahan *Token* Akses ke Variabel Lingkungan: *Token* akses yang telah dibuat ditambahkan ke variabel lingkungan.
7. Penambahan Kunci SSH Publik ke Akun Digital Ocean: Kunci ini digunakan untuk otentikasi saat melakukan koneksi SSH ke *droplet*.
8. Pembuatan Kluster: Proses ini dimulai dengan menjalankan *script* create.sh. *Script* ini mengandung serangkaian instruksi untuk membuat kluster, termasuk pembuatan *droplets*, penentuan alamat IP, pemasangan *Docker*, pengaturan *Kubernetes*, dan lainnya.

Langkah-langkah selanjutnya adalah pemasangan dan konfigurasi Multus CNI dan *Flannel*.

D. Konfigurasi *Flannel* pada Multus CNI

Dalam tahap ini, *Flannel* dikonfigurasikan sebagai solusi CNI pada Multus untuk menciptakan lingkungan yang diuji. Hal ini dilakukan dengan menginstal *Flannel* ke dalam kluster *Kubernetes* dan kemudian membuat Network *Attachment Definition* yang merujuk ke *Flannel* sebagai CNI *plugin*. Proses konfigurasi ini melibatkan beberapa langkah teknis termasuk penyetelan jaringan virtual, penjadwalan *Pod*, dan penentuan peraturan lalu lintas jaringan.

E. Uji *Performance* dan Pengambilan Data pada *Flannel*

Setelah *Flannel* berhasil dikonfigurasikan pada Multus CNI, tahap pengujian *performance* dan pengambilan data dimulai. Pengujian dilakukan dengan skenario komunikasi antar *Pod* dalam satu *Node* dan antar *Node* yang berbeda. Pengukuran utama yang diambil adalah *latency*, *packet loss*, *throughput*, dan penggunaan CPU. Untuk menjamin validitas dan reliabilitas data, setiap pengujian dijalankan beberapa kali dan hasil rata-rata digunakan sebagai representasi kinerja.

F. Migrasi *Default* CNI dari *Flannel* ke *Cilium*

Tahap ini melibatkan migrasi CNI *default* dari *Flannel* ke *Cilium*. Langkah pertama adalah membuat *snapshot* cadangan untuk semua *node* sebelum migrasi, dilanjutkan dengan proses migrasi yang melibatkan penghapusan Network *Attachment Definition* yang ada untuk *Flannel*, penghapusan *Flannel* dari kluster, dan instalasi *Cilium*. Setelah migrasi selesai, semua *node* diaktifkan kembali dan status *node* diverifikasi.

G. Konfigurasi *Cilium* pada Multus CNI

Konfigurasi *Cilium* pada Multus CNI dilakukan setelah proses migrasi selesai. Hal ini melibatkan penyesuaian berbagai parameter dan konfigurasi untuk mengoptimalkan kinerja *Cilium*. Dalam tahap ini, Network *Attachment Definition* baru dibuat untuk *Cilium* dan penyesuaian konfigurasi Multus CNI dilakukan untuk merujuk ke *Cilium* sebagai *plugin* CNI *default*.

H. Uji *Performance* dan Pengambilan Data pada *Cilium*

Seperti pada tahap pengujian *Flannel*, tahap pengujian untuk *Cilium* melibatkan evaluasi kinerja dalam dua skenario yaitu antar *Pod* dalam satu *Node* dan antar *Node* yang berbeda. Pengukuran yang sama - *latency*, *packet loss*, *throughput*, dan penggunaan CPU - juga digunakan dalam tahap ini. Seperti sebelumnya, setiap pengujian dijalankan beberapa kali dan hasil rata-rata digunakan sebagai representasi kinerja.

I. Analisa Hasil

Tahap akhir dari metodologi penelitian ini adalah analisis hasil pengujian. Analisis ini difokuskan pada evaluasi kinerja *Flannel* dan *Cilium* dalam berbagai skenario pengujian berdasarkan hasil pengukuran *latency*, *packet loss*, *throughput*, dan penggunaan CPU. Selain itu, analisis juga melibatkan perbandingan hasil pengujian kedua solusi

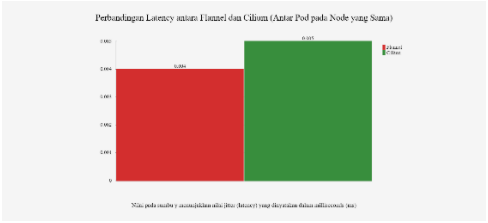
tersebut untuk memberikan gambaran yang jelas tentang kinerja dan efisiensi masing-masing dalam konteks penggunaan nyata. Dengan demikian, hasil analisis ini dapat digunakan untuk membuat keputusan yang lebih berinformasi tentang pemilihan solusi CNI terbaik untuk implementasi dalam lingkungan produksi.

IV. HASIL DAN PEMBAHASAN

Analisis berfokus pada penjelasan dan evaluasi dari hasil pengujian yang telah diimplementasikan. Fokus pengujian adalah tiga jenis *Container* Network *Interface* (CNI) yang berbeda, yakni *Flannel*, *Cilium*, dan Multus CNI. Dalam penelitian ini digunakan kombinasi *Flannel* dan Multus CNI, serta kombinasi *Cilium* dan Multus CNI secara bergantian untuk mengamati dampaknya terhadap efisiensi komunikasi antar *Pod* dalam kluster *Kubernetes*. Diadopsi dua skenario pengujian utama yaitu pengujian komunikasi antar *Pod* dalam satu *Node* yang sama dan pengujian komunikasi antar *Pod* yang berada pada *Node* yang berbeda. Penampilan hasil pengujian akan disajikan dalam format grafik untuk memfasilitasi proses analisis hasil yang lebih efisien.

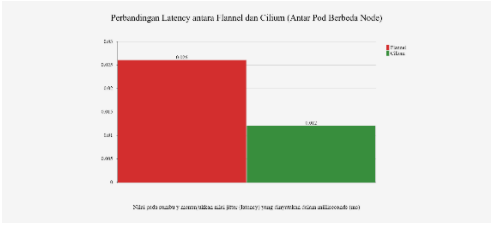
A. Perbandingan Kinerja dalam Aspek *Latency*

Dalam pengujian *performance* dalam aspek *latency*, dilakukan perbandingan antara *Flannel* dan *Cilium* dalam dua skenario yang berbeda: komunikasi antar *pod* dalam satu *node* yang sama dan komunikasi antar *pod* dalam *node* yang berbeda. *Performance latency* antar *pod* dalam satu *node* ditunjukkan pada Gambar 6, sedangkan *performance latency* antar *pod* berbeda *node* ditunjukkan pada Gambar 7.



Gambar 6. *Performance Latency* antar *Pod* dalam Satu *Node*

Berdasarkan hasil pengujian *latency* pada skenario antar *pod* dalam *node* yang sama, *Flannel* mencatat *jitter* sebesar 0.004 ms, sedangkan *Cilium* menunjukkan nilai sedikit lebih tinggi, yaitu 0.005 ms seperti yang ditunjukkan pada Gambar 6.

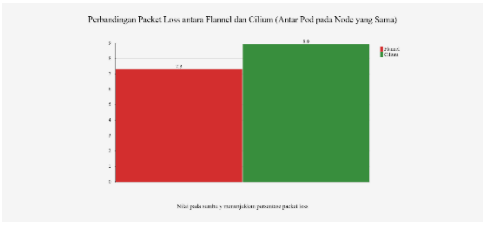


Gambar 7. *Performance Latency* antar *Pod* Berbeda *Node*

Pada skenario komunikasi antar *pod* di antara *node* yang berbeda, *Flannel* mencatat *jitter* sebesar 0.026 ms, sementara *Cilium* menunjukkan nilai yang jauh lebih rendah, yaitu 0.012 ms seperti yang ditunjukkan pada Gambar 7.

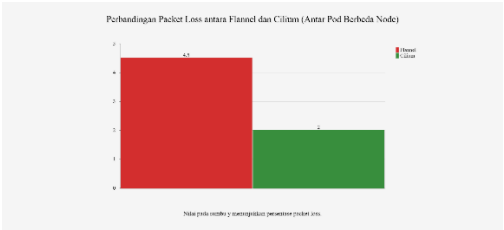
B. Perbandingan Kinerja dalam Aspek *Packet Loss*

Pada skenario antar *pod* dalam *node* yang sama, ditemukan bahwa *Flannel* mengalami *packet loss* sebesar 7,3%, sedangkan *Cilium* mengalami *packet loss* sebesar 8,9% seperti yang ditunjukkan pada Gambar 8.



Gambar 8. *Performance Latency* antar *Pod* dalam Satu *Node*

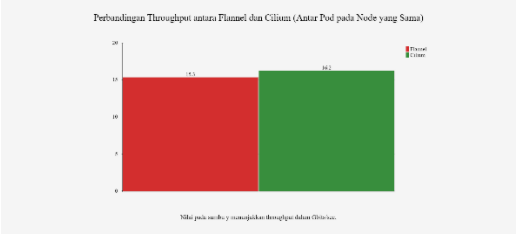
Sedangkan dalam skenario antar *pod* di antara *node* yang berbeda, *Flannel* diperoleh *packet loss* sebesar 4.5%, sedangkan pada *Cilium*, *packet loss* hanya sebesar 2% seperti yang ditunjukkan pada Gambar 9.



Gambar 9. *Performance Latency* antar *Pod* Berbeda *Node*

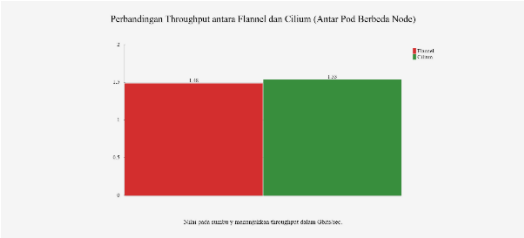
C. Perbandingan Kinerja dalam Aspek *Throughput*

Untuk skenario antar *pod* dalam *node* yang sama, *Flannel* menghasilkan *throughput* sebesar 15.3 Gbits/*sec*, sementara *Cilium* menghasilkan *throughput* yang lebih tinggi, yaitu sebesar 16.2 Gbits/*sec* seperti yang ditunjukkan pada Gambar 10.



Gambar 10. *Performance Latency* antar *Pod* dalam Satu *Node*

Sedangkan dalam skenario antar *pod* di antara *node* yang berbeda, *Flannel* mencapai *throughput* sekitar 1.48 Gbits/*sec*, sementara *Cilium* mencapai 1.53 Gbits/*sec* seperti yang ditunjukkan pada Gambar 11.

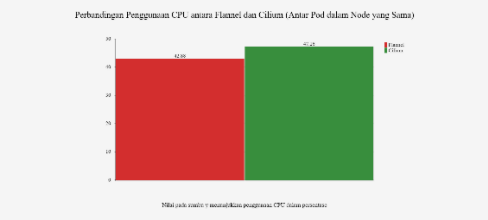


Gambar 11. *Performance Latency* antar *Pod* Berbeda *Node*

Mengenai peningkatan *throughput*, penting untuk mempertimbangkan peningkatan ini dalam konteks fungsi dan fitur tambahan yang disediakan oleh *Cilium*. Salah satu keunggulan penting *Cilium* dibanding *Flannel* adalah dukungan terhadap BPF (*Berkeley Packet* Filter), yang menyediakan kecepatan dan fleksibilitas yang lebih baik dalam pengolahan paket serta penerapan kebijakan jaringan.

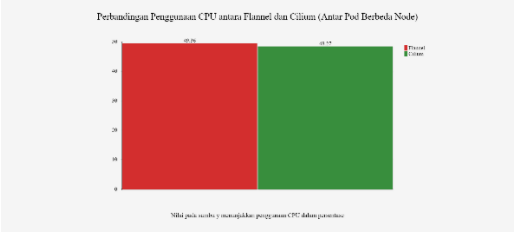
D. Perbandingan Kinerja dalam Aspek CPU *Usage*

Untuk skenario antar *pod* dalam *node* yang sama, *Flannel* mencapai penggunaan CPU sebesar 42.88%, sedangkan *Cilium* mencapai penggunaan CPU sebesar 47.16% seperti yang ditunjukkan pada Gambar 12.



Gambar 12. *Performance Latency* antar *Pod* dalam Satu *Node*

Sedangkan dalam skenario antar *pod* di antara *node* yang berbeda, *Flannel* mencapai penggunaan CPU sebesar 49.36%, sedangkan *Cilium* mencapai penggunaan CPU sebesar 48.27% seperti yang ditunjukkan pada Gambar 13.



Gambar 13. *Performance Latency* antar *Pod* Berbeda *Node*

Penggunaan CPU ini sebanding dengan efisiensi yang ditawarkan oleh setiap CNI. *Flannel*, sebagai solusi jaringan *overlay* yang sederhana, dapat mengonsumsi lebih banyak CPU karena *overhead* yang terkait dengan enkapsulasi paket. Di sisi lain, *Cilium*, yang menggunakan teknologi BPF (*Berkeley Packet* Filter) modern, mungkin lebih efisien dalam penggunaan CPU, yang dapat menjelaskan penggunaan CPU yang sedikit lebih rendah.

Namun, juga penting untuk diperhatikan bahwa hasil ini mungkin dipengaruhi oleh spesifikasi *node*, termasuk jumlah *vCPU*. Dalam penelitian ini, *node* yang digunakan memiliki dua *vCPU*, dan perbedaan dalam penggunaan CPU mungkin sebagian dihasilkan dari sejauh mana masing-masing CNI dapat memanfaatkan sumber daya tersebut.

V. SIMPULAN

Melalui pengujian dan analisis data yang telah dilakukan, beberapa penemuan penting dapat disimpulkan. Dalam konteks latensi, ketika komunikasi terjadi antar *pod* dalam satu *node*, *Flannel* menunjukkan kinerja yang sedikit lebih baik dengan latensi yang lebih rendah dibandingkan dengan *Cilium*. Namun, ketika skenarionya melibatkan komunikasi antar *node* yang berbeda, *Cilium* mengungguli *Flannel* dengan latensi yang lebih rendah. Perbedaan ini bisa memiliki dampak signifikan pada performa aplikasi, terutama bagi aplikasi yang sangat sensitif terhadap latensi.

Kemudian, dalam aspek *packet loss*, *Flannel* menunjukkan efisiensi yang lebih baik dalam komunikasi antar *pod* dalam satu *node*, dengan *packet loss* sebesar 7,3%, dibandingkan dengan *Cilium* yang memiliki *packet loss* sebesar 8,9%. Namun, dalam skenario komunikasi antar *node*, *Cilium* unggul dengan *packet loss* sebesar 2% dibandingkan dengan *Flannel* yang mencapai 4,5%. Kinerja kedua solusi CNI ini tampaknya dapat bervariasi tergantung pada skenario penggunaan.

Selanjutnya, dalam hal *throughput*, *Cilium* menunjukkan performa yang sedikit lebih baik dibandingkan dengan *Flannel* dalam kedua skenario tersebut. Keunggulan lain dari *Cilium* terletak pada dukungannya terhadap BPF, yang memberikan kecepatan dan fleksibilitas yang lebih tinggi dalam pengolahan paket dan implementasi kebijakan jaringan.

Akhirnya, dalam pengukuran penggunaan CPU, *Cilium* umumnya memerlukan sedikit lebih banyak sumber daya CPU dibandingkan dengan *Flannel*, baik dalam skenario komunikasi di dalam *node* maupun antar *node*. Namun, mempertimbangkan bahwa *Cilium* memanfaatkan BPF, ini bisa menandakan bahwa *Cilium* lebih efisien dalam penggunaan CPU, meski perbedaan ini tampaknya kecil.

REFERENSI

[1] Datadog, "9 insights on real world container use," November 2022. [Online].

[2] C. Carrión, "Kubernetes scheduling: Taxonomy, ongoing issues and challenges," *ACM Comput. Surv.*, vol. 55, p. 1–37, July 2022.

[3] N. M. P. G. Sindhu G, "Deploying a Kubernetes Cluster With Kubernetes Operation Kops On Aws Cloud Experiments And Lessons Learned," *IJEAT*, vol. IJEAT, p. 984–989, 2020.

[4] G. Shi, W. Cai, J. Zhang, C. Gao, S. Sun, Y. Zhang and Y. Jiang, "KubeCOM: an implementation of a non-containerized software management system based on Kubernetes," in *Third International Conference on Computer Science and Communication Technology (ICCSCT 2022)*, Beijing, 2022.

[5] A. Rahman, S. I. Shamim, D. B. Bose and R. Pandita, "Security Misconfigurations In Open Source Kubernetes Manifests: An Empirical Study," *ACM Trans. Softw. Eng. Methodol*, vol. 32, p. 1–36, 2023.

[6] F. R. Poetra, S. Prabowo, S. A. Karimah and R. D. Prayogo, "Performance analysis of video streaming service migration using container orchestration," *IOP Conf. Ser. Mater. Sci. Eng.*, vol. 830, p. 022100, April 2020.

[7] G. Avolio, M. Cadeddu and R. Hauser, "Evaluating Kubernetes as an orchestrator of the Event Filter computing farm of the Trigger and Data Acquisition system of the ATLAS experiment at the Large Hadron Collider," *EPJ Web Conf.*, vol. 214, p. 07024, 2019.

[8] C. N. C. Foundation, "Announcing the Cilium annual report," January 2023. [Online].

[9] M. Mackrory, "The ultimate guide to using calico, flannel, weave and cilium," June 2021. [Online].

[10] S. Qi, S. G. Kulkarni and K. K. Ramakrishnan, "Assessing container network interface plugins: Functionality, performance, and scalability," *IEEE Trans. Netw. Serv. Manag.*, vol. 18, p. 656–671, March 2021.

[11] N. Kapocius, "Performance studies of kubernetes network solutions," in *2020 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream)*, Vilnius, 2020.

[12] G. Budigiri, C. Baumann, J. T. Muhlberg, E. Truyen and W. Joosen, "Network policies in kubernetes: Performance evaluation and security analysis," in *2021 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit)*, Porto, 2021.

[13] Z. Wang, Y. Ji, W. Zheng and M. Li, "Multi-level Network Software Defined Gateway Forwarding System Based On Multus," in *Proceeding of 2021 International Conference on Wireless Communications, Networking and Applications*, 2022, p. 166–176.

[14] A. Ayub, M. Ishaq and M. Munir, "Enhancement in multus CNI for DPDK applications in the cloud native environment," in *2023 26th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, Paris, 2023.