Jurnal Pengembangan Sains dan Teknologi
Volume 1, Nomor 2, Juli 2025, p. 101-115
ISSN : 3090-2762 (*Print*) | 3089-9516 (*Online*)
DOI : 10.63866/jpst.v1i2.79

Implementasi *Algoritma Raft* untuk Menjamin *High Availability* pada *Docker Swarm Clustering*

Khaeruddin[1*], Zumhur Alamin[2], Arisandi[3], Mohammad Nur Cholis[1]

[1]Program Studi Teknik Elektro, Universitas Insan Budi Utomo, Malang, Indonesia
[2]Program Studi Ilmu Komputer, Universitas Muhammadiyah Bima, Indonesia
[3]Program Studi Teknik Mesin, Universitas Insan Budi Utomo, Malang, Indonesia

Email Koresponden: lamone41@gmail.com
(* : corresponding author)

Abstrak − Penelitian ini bertujuan untuk mengevaluasi efektivitas implementasi Docker Swarm dalam membangun sistem orkestrasi kontainer dengan pendekatan *High Availability* (HA). Fokus utama penelitian adalah menganalisis mekanisme failover, konsistensi status klaster, dan stabilitas komunikasi antar node menggunakan algoritma konsensus Raft. Metode yang digunakan bersifat eksperimental dengan simulasi skenario kegagalan pada node worker dalam lingkungan virtual yang terdiri dari tiga node: satu manager (master-swarm) dan dua worker (swarm1 dan swarm2). Pengujian diawali dengan validasi konektivitas dan kestabilan jaringan antar node. Hasil menunjukkan komunikasi berjalan efisien, dengan trafik meningkat saat terjadi deployment layanan. Uji HA dilakukan dengan mensimulasikan kegagalan pada node swarm2. Docker Swarm berhasil melakukan penjadwalan ulang layanan secara otomatis ke node lain yang aktif dalam waktu singkat, menunjukkan efektivitas algoritma Raft dalam menjaga ketersediaan layanan dan konsistensi status klaster. Namun, ditemukan keterbatasan sumber daya pasca-failover yang menyebabkan beberapa tugas mengalami status *rejected*, terutama ketika beban kerja harus dialihkan ke dua node yang tersisa. Penelitian ini juga melibatkan pemantauan real-time menggunakan ntopng dan pengelolaan kontainer melalui Portainer, yang terbukti membantu dalam visualisasi status klaster dan identifikasi potensi hambatan. Kesimpulannya, Docker Swarm mampu menjamin HA dalam skala kecil-menengah, namun implementasi optimal memerlukan perencanaan kapasitas sumber daya yang matang agar sistem tetap stabil meskipun terjadi kegagalan node.

Kata Kunci: Docker Swarm, High Availability, Raft, Failover, Portainer

Raft Algorithm Implementation to Ensure High Availability in Docker Swarm Clustering

Abstract − This study aims to evaluate the effectiveness of Docker Swarm implementation in building a container orchestration system with a High Availability (HA) approach. The main focus of the study is to analyze the failover mechanism, cluster status consistency, and communication stability between nodes using the Raft consensus algorithm. The method used is experimental with simulation of failure scenarios on worker nodes in a virtual environment consisting of three nodes: one manager (master-swarm) and two workers (swarm1 and swarm2). Testing begins with validation of connectivity and network stability between nodes. The results show that communication runs efficiently, with traffic increasing when service deployment occurs. HA testing is carried out by simulating failures on the swarm2 node. Docker Swarm successfully reschedules services automatically to other active nodes in a short time, demonstrating the effectiveness of the Raft algorithm in maintaining service availability and cluster status consistency. However, post-failover resource limitations were found which caused several tasks to experience rejected status, especially when the workload had to be shifted to the two remaining

nodes. This study also involves real-time monitoring using ntopng and container management through Portainer, which has proven to be helpful in visualizing cluster status and identifying potential bottlenecks. In conclusion, Docker Swarm is able to guarantee HA on a small-medium scale, but optimal implementation requires careful resource capacity planning to keep the system stable even if a node fails.

Keywords: Docker Swarm, High Availability, Raft, Failover, Portainer

Received	Revised	Published
27-05-2025	19-06-2025	27-06-2025

1. PENDAHULUAN

Dalam era komputasi awan dan pengembangan perangkat lunak modern, kebutuhan terhadap sistem yang andal, skalabel, dan selalu tersedia (*high availability*) menjadi sangat penting. Organisasi dan perusahaan semakin mengandalkan infrastruktur kontainer seperti *Docker* untuk melakukan *deployment* aplikasi secara efisien. Dalam konteks ini, virtualisasi telah menjadi fondasi penting bagi banyak organisasi untuk membangun lingkungan sistem yang efisien, fleksibel, dan mudah disesuaikan. Salah satu inovasi paling signifikan dalam bidang virtualisasi adalah teknologi kontainer, dengan *Docker* sebagai salah satu platform yang paling banyak digunakan.

Docker adalah platform *open-source* yang memungkinkan developer dan administrator sistem untuk membuat, mengemas, dan menjalankan aplikasi dalam *container* yang ringan dan portabel. Teknologi ini menawarkan keunggulan dalam hal konsistensi antara pengembangan dan produksi, efisiensi pemanfaatan sumber daya, serta kemudahan dalam melakukan *deployment* dan scaling aplikasi [1], [8]. *Docker* memungkinkan sebuah aplikasi dan seluruh dependensinya dibungkus ke dalam satu unit kontainer yang dapat dijalankan secara konsisten di berbagai platform, baik lokal maupun cloud.

Salah satu fitur unggulan *Docker* adalah *Docker Swarm*, yang memungkinkan pengelolaan banyak kontainer secara terdistribusi melalui model klasterisasi. Dengan menggunakan *Swarm Mode*, beberapa *host Docker* dapat dikumpulkan menjadi satu entitas logis yang disebut cluster, dan manajemen kontainer dilakukan secara terpusat. Fitur ini mendukung replikasi layanan, distribusi beban kerja (*load balancing*), serta mekanisme *failover* yang otomatis, menjadikannya cocok untuk implementasi sistem yang memerlukan tingkat *high availability* [6], [13].

Namun, dalam sistem terdistribusi seperti *Docker Swarm*, tantangan terbesar adalah menjamin konsistensi dan ketersediaan layanan ketika terjadi kegagalan pada node tertentu. Untuk mengatasi tantangan ini, *Docker Swarm* mengadopsi algoritma konsensus yang dikenal sebagai Raft. Raft adalah algoritma yang dirancang untuk memfasilitasi replikasi log secara aman dan memastikan bahwa hanya satu node dalam cluster yang dapat bertindak sebagai leader pada suatu waktu. Hal ini menghindari konflik dalam pengambilan keputusan serta menjamin *data consistency* dan *fault tolerance* [5], [10], [14].

Raft dikembangkan oleh Diego Ongaro dan John Ousterhout sebagai alternatif yang lebih mudah dipahami dari algoritma Paxos. Keunggulan Raft terletak pada struktur modularnya, yang membagi masalah konsensus menjadi tiga bagian: pemilihan *master slave*, replikasi log, dan keamanan. Dragoni et al. [10] menyatakan bahwa pendekatan formal terhadap Raft memungkinkan analisis dan verifikasi terhadap keandalannya dalam sistem nyata, menjadikannya algoritma pilihan dalam berbagai implementasi sistem terdistribusi modern, termasuk dalam *Docker Swarm*.

Sementara itu, penelitian sebelumnya telah menunjukkan potensi besar teknologi *Docker* dalam berbagai skenario penerapan. Iqbal et al. [1] mengkaji integrasi *Docker* dengan *OpenStack* untuk menciptakan sistem cloud yang skalabel dan efisien. Di sisi lain, Haikal et al. [2] membuktikan bahwa penggunaan *Docker* dalam konteks *CDN (Content Delivery Network)* dapat meningkatkan performa distribusi konten dengan integrasi ke layanan *Cloudflare*. Penelitian tentang sistem database yang bersifat dinamis dan toleran terhadap kegagalan oleh Afirda et all [3], serta studi penerapan teknik enkripsi dalam sistem mail server oleh Faruq [4], menegaskan pentingnya high availability dan keamanan dalam infrastruktur digital.

Lebih lanjut, studi oleh Bender et al. [9] mengenai penggunaan *Kubernetes* dan *Docker* untuk workflow ilmiah berbasis cloud menyoroti pentingnya orkestrasi kontainer dan ketersediaan layanan berkelanjutan pada skala data yang lebih kompleks. Hal ini didukung pula oleh riset Abu-Libdeh et al. [11] yang mempromosikan pendekatan diversifikasi penyimpanan cloud guna mengurangi risiko kehilangan data dan downtime.

Untuk mempermudah proses pengelolaan Docker Swarm Clustering, banyak administrator mengandalkan alat bantu seperti *Portainer*. Portainer merupakan dashboard berbasis web yang menyediakan antarmuka visual untuk mengelola *container, service, stack, dan volume* secara intuitif. Dengan *Portainer*, pengguna dapat melakukan manajemen *deployment*, melakukan *scale up/down* layanan, memantau performa *container*, serta mengatur *user access* secara lebih efisien [7], [15]. Di sisi lain, untuk keperluan pemantauan lalu lintas jaringan secara real-time dalam lingkungan klaster, *ntopng* menjadi solusi yang banyak digunakan. ntopng mampu menganalisis trafik berdasarkan protokol, host, dan throughput jaringan secara detail sehingga membantu administrator mengidentifikasi bottleneck dan pola penggunaan jaringan [16].

Studi oleh Pavlenko & Ivanov [15] menegaskan bahwa penggunaan antarmuka grafis seperti Portainer tidak hanya meningkatkan efisiensi dalam manajemen klaster, tetapi juga menurunkan potensi kesalahan konfigurasi. Sementara itu, penelitian oleh Geetha & Dinesh [16] menunjukkan bahwa integrasi alat monitoring jaringan seperti ntop dalam sistem kontainerisasi mendukung deteksi dini terhadap anomali dan serangan siber. Penelitian oleh Singh & Juneja [17] membandingkan performa Docker Swarm dengan Kubernetes dan menyimpulkan bahwa Swarm menawarkan keunggulan dalam setup awal dan kecepatan failover. Selain itu, riset Basyoni & El-Shishtawy [18] menunjukkan bahwa performa klaster Swarm dapat dianalisis menggunakan parameter sistem seperti CPU usage, RAM, dan throughput, untuk mendukung proses tuning dan perbaikan performa layanan.

Berdasarkan paparan di atas, terlihat jelas bahwa tantangan dan kebutuhan akan sistem yang dapat menjamin *high availability, fault tolerance, dan scalability* semakin meningkat. Oleh karena itu, penelitian ini bertujuan untuk mengkaji secara mendalam implementasi algoritma Raft dalam menjamin *high availability* pada *Docker Swarm Cluster*. Fokus penelitian mencakup pengamatan terhadap proses pemilihan leader, sinkronisasi log antar node, serta mekanisme failover yang terjadi dalam lingkungan klaster dengan menggunakan *Portainer* sebagai antarmuka manajemen.

Penelitian ini bertujuan untuk mengkaji secara spesifik bagaimana algoritma Raft diimplementasikan dalam *Docker Swarm* dan bagaimana mekanisme ini mampu menjamin *high availability* dalam klaster kontainer. Penelitian juga akan menyimulasikan skenario kegagalan node dan menganalisis bagaimana Swarm melakukan proses pemulihan melalui pemilihan pemimpin baru, serta sejauh mana algoritma Raft berperan dalam menjaga konsistensi status klaster. Dengan pendekatan ini, diharapkan dapat memberikan kontribusi nyata terhadap pengembangan sistem distribusi kontainer yang tangguh dan andal.

2. METODE PENELITIAN

Penelitian ini menggunakan pendekatan eksperimental dan simulatif untuk mengamati secara langsung bagaimana algoritma Raft yang diimplementasikan dalam *Docker Swarm* dapat menjamin *high availability* pada klaster kontainer. Metode ini dipilih karena sesuai untuk menguji performa dan respons sistem terhadap skenario kegagalan yang disengaja. Jenis penelitian ini adalah kuantitatif eksperimental, dengan pendekatan studi kasus implementasi dan pengujian langsung pada lingkungan virtual. Tujuan utamanya adalah untuk mengevaluasi bagaimana mekanisme konsensus Raft pada *Docker Swarm* bekerja dalam menghadapi *node failure* dan memastikan ketersediaan layanan tetap terjaga. Untuk manajemen Docker Swarm mengggunakan portainer, untuk monitoring menggunakan ntop.

2.1 Topologi

Topologi Docker Swarm Cluster, yang merupakan arsitektur dasar dari Docker Swarm, sebuah alat orkestrasi container buatan Docker Inc. yang digunakan untuk mengelola dan mengatur deployment container dalam skala besar. Pada topologi berikut memperlihatkan terdapat 3 server (*Swarm Manager Node, Swarm Worker Node A, Swarm Worker Node B*, dan 1 client.

Tabel 1. IP Address Server Docker Swarm

No	Nama Perangkat	Aplikasi	IP Address
1.	Swarm Manager Node	Docker, Portainer, NtopNG	10.1.1.20/24
2.	Swarm Worker Node A	Docker	10.1.1.21/24
3.	Swarm Worker Node B	Docker	10.1.1.22/24
4.	Client	Browser	10.1.1.23/24

2.1.1. Swarm Manager Node

Node ini berperan sebagai master untuk mengontrol dari seluruh node. Manager Node menangani berbagai tugas penting:

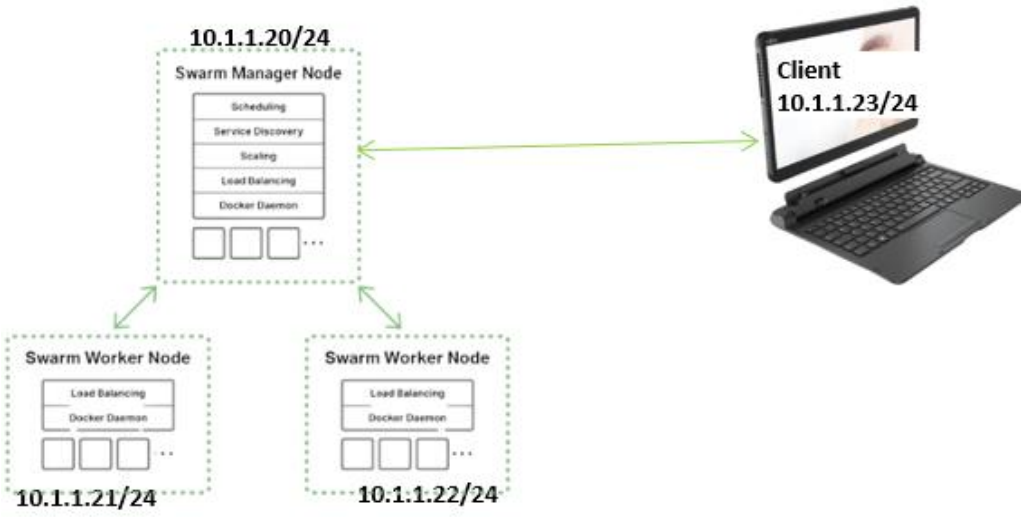
a. **Scheduling:** Menentukan kontainer mana yang dijalankan di node mana berdasarkan ketersediaan sumber daya.
b. **Service Discovery:** Mengatur agar layanan-layanan bisa saling menemukan satu sama lain di dalam klaster.
c. **Scaling:** Mengatur skala layanan, misalnya menjalankan 3 replika kontainer dari suatu aplikasi.
d. **Load Balancing:** Mendistribusikan traffic antar node atau antar replika kontainer.
e. **Docker Daemon:** Proses utama Docker yang berjalan di semua node.

2.1.2. Swarm Worker Node

Node ini bertugas menjalankan container yang dijadwalkan oleh Manager Node. Worker Node memiliki:

a. Salah satu fungsi dari server *Swarm Manager Node* dengan IP address 10.1.1.20/24 pada Gambar 1, yaitu sebagai *Load Balancing* yang mengatur penerimaan dan meneruskan permintaan client 10.1.1.23/24 ke kontainer yang relevan. Apakah permintaan diteruskan ke *Swarm Worker Node* 10.1.1.21/24 atau ke *Swarm Worker Node* 10.1.1.22/24.
b. *Swarm Worker Node* 10.1.1.21/24 dan *Swarm Worker Node* 10.1.1.22/24 pada Gambar 1 terdapat *Docker Daemon,* komponen ini berfungsi untuk untuk menjalankan kontainer dan melakukan operasi Docker lainnya yang ada pada kedua node tersebut.

c. Pada topologi Gambar 1, *Swarm Worker Node* tidak memiliki akses untuk melakukan pengontrolan, kedua node ini hanya bisa melakukan apa yang diperintahkan *Swarm Manager Node* denga IP address 10.1.1.20/24.



Gambar 1. Topologi penelitian Docker Swarm

2.1.2. Client

Digunakan untuk meremote server node *Docker Swarm,* serta memonitoring server manager node dan juga server worker node.

2.2 Konfigurasi

2.2.1. Sistem Operasi

Penelitian ini menggunakan Ubuntu Server 22.04 LTS sebagai sistem operasi dasar untuk seluruh node pada penelitian ini.

2.2.2. Docker

Docker yang digunakan adalah *Docker Engine 24.x,* yakni versi stabil terbaru pada saat penelitian berlangsung. Pemilihan versi ini bertujuan untuk mendapatkan fitur-fitur terbaru serta perbaikan keamanan dan performa. Versi ini juga mendukung penuh konfigurasi *Docker Swarm Mode*, termasuk orkestrasi layanan dan *load balancing*. Proses update paket yang dibutuhkan serta instalasi aplikasi *Docker* dengan mengetikan command :

```
apt update
apt install -y ca-certificates curl gnupg lsb-release
mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
gpg --dearmor -o /etc/apt/keyrings/docker.gpg
apt update
apt install -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
usermod -aG docker $USER
docker –version
systemctl status docker
```

2.2.3. Konfigurasi Docker Swarm

Langkah awal untuk membentuk klaster adalah dengan menginisialisasi node manager, setelah inisialisasi, Docker akan memberikan token khusus yang dapat digunakan oleh node lain untuk bergabung ke dalam klaster. Berikut merupakan konfigurasinya.

docker swarm init --advertise-addr 10.1.1.20
docker swarm init: Mengaktifkan Swarm Mode dan mengonversi node menjadi manager.
--advertise-addr: Menentukan IP publik/privat yang digunakan oleh node lain untuk berkomunikasi dengan manager.

Pada node manager, dibuat layanan web berbasis NGINX dalam Docker Swarm Clustering, dengan dua replika aktif untuk memastikan load balancing dan high availability.

```
root@master-swarm:~# docker service create --name TA-SWARM --network cbncloud --replicas 2 -p
 80:80 nginx:latest
xfno365y48eniridvcbn2wa88
overall progress: 0 out of 2 tasks
1/2: assigned
2/2: assigned
```

Gambar 2. Load Balancing dan Replication Docker Swarm

Pada Gambar 2 menampilkan proses implementasi layanan web server berbasis NGINX dengan menggunakan Docker Swarm Cluertering dengan command docker service create. Command ini digunakan untuk mendefinisikan sebuah layanan TA-SWARM dengandua container replikasi aktif *(--replicas 2)* yang dijalanakan jaringan overlay *cbncloud.* Setiap replikasi menjalankan image *nginx:latest* dan memetakan port 80 container ke port 80 host *(-p 80:80).* Status "*assigned:"* menunjukan bahwa scheduler Docker Swarm telah berhasil mendistribusikan kedua replikasi ke node dalam cluster. Konfigurasi ini digunakan untuk mendukung proses *load balancing* juga meningkatkan *high availability* dari layanan yang berjalan.

2.2.4. Konfigurasi Portainer

Portainer merupakan aplikasi GUI berbasis web untuk mengelola container Docker dan Kubernetes, untuk menginstall portainer dilakukan setelah kita menginstall docker, versi yang digunakan merupakan versi community.

apt install docker.io -y
docker pull portainer/portainer-ce

Konfigurasi portainer agar berjalan secara permanen serta untuk mengakses Docker socket buat containernya dengan menambhakan konfigurasi berikut.

docker volume create portainer_data
*docker run -d *
*-p 9000:9000 *
*-p 9443:9443 *
*--name portainer *
*--restart=always *
*-v /var/run/docker.sock:/var/run/docker.sock *
*-v portainer_data:/data *
portainer/portainer-ce

```
root@master-swarm:~# docker run --name portainer -h portainer -d -p 8000:8000 -p 127.0.0.1:90
00:9000 -v /var/run/docker.sock:/var/run/docker.sock -v portainer_data:/data portainer/portai
ner-ce
3a8ffe42f18b308ffd8c58b720c20d1c87f0e36dfef006950ace6acd226f3316
root@master-swarm:~#
```

Gambar 3. Menjalankan Portainer

Perintah yang terlihat pada Gambar 3, digunakan untuk men-deploy Portainer secara lokal, memberikan akses ke Docker engine agar dapat memantau dan mengelola container melalui antarmuka grafis, serta mengatur agar hanya dapat diakses dari localhost pada port 9000 demi alasan keamanan.Perintah ini digunakan untuk men-deploy Portainer secara lokal, memberikan akses ke Docker engine agar dapat memantau dan mengelola container melalui antarmuka grafis, serta mengatur agar hanya dapat diakses dari localhost pada port 9000 demi alasan keamanan.

2.2.5. Konfigurasi Ntopng

Untuk menginstal ntopng, langkah pertama yang harus dilakukan adalah mengunduh paket dari repository resmi ntop. Proses ini diawali dengan menambahkan kunci GPG publik untuk memastikan integritas dan keamanan paket, diikuti dengan penambahan repository resmi ntopng ke dalam sistem operasi berbasis Debian/Ubuntu. Gambar 4 memperlihatkan tahapan penambahan repository tersebut.

Penambahan kunci GPG dilakukan menggunakan perintah wget dan apt-key, yang bertujuan untuk memastikan bahwa paket yang diunduh berasal dari sumber yang tepercaya. Selanjutnya, repository ditambahkan ke dalam direktori konfigurasi sumber paket, yaitu sources.list.d. Setelah repository berhasil ditambahkan, sistem perlu diperbarui menggunakan perintah apt-get update atau apt update, kemudian dilanjutkan dengan proses instalasi aplikasi menggunakan apt-get install atau apt install.

Langkah-langkah ini penting untuk memastikan bahwa versi aplikasi ntopng yang diinstal merupakan versi terbaru dan stabil, sebagaimana yang direkomendasikan pada situs resmi ntopng.

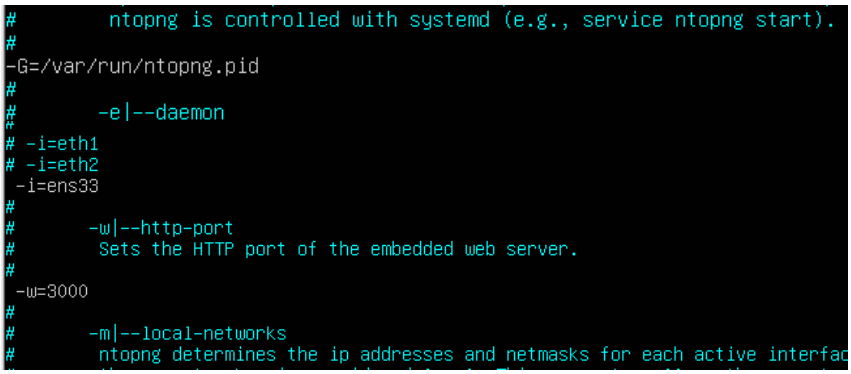
Gambar 4. Installasi ntopng Bersumber dari Repository

Setelah instalasi berhasil, tahap berikutnya adalah melakukan konfigurasi pada file utama ntopng.conf yang terletak di direktori /etc/ntopng/. Gambar 5 menunjukkan tampilan file konfigurasi tersebut. Beberapa parameter penting yang perlu ditambahkan dalam file ini meliputi pengaturan interface jaringan, HTTP port, serta port untuk akses web.

Konfigurasi ini bertujuan untuk menentukan parameter dasar operasi ntopng, antara lain:

- Mode daemon (-d)
- Interface jaringan yang akan dimonitor (-i=eth1, -i=ens33)
- Port HTTP untuk akses antarmuka web (-w=3000)
- Jaringan lokal yang dipantau (--local-networks)

Dengan konfigurasi ini, ntopng dapat menjalankan antarmuka pemantauan berbasis web yang dapat diakses melalui browser. Pengguna dapat melihat informasi lalu lintas jaringan secara real-time berdasarkan interface yang telah ditentukan.

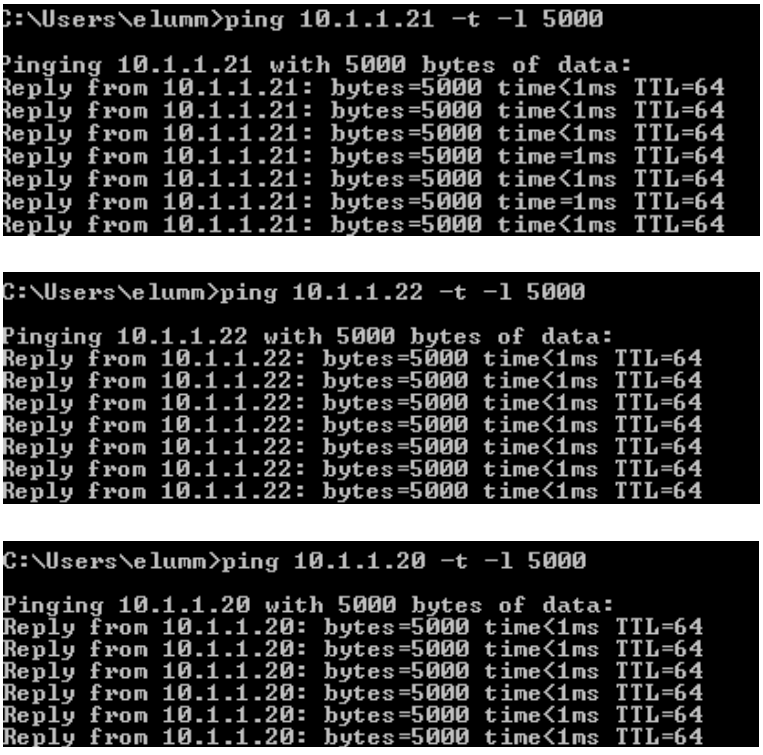


Gambar 5. File Konfigurasi ntopng

2.3 Pengujian

Pengujian *Docker Swarm* penting untuk mengevaluasi stabilitas, *high availability (HA),* dan kemampuan *load balancing* dalam lingkungan terdistribusi. Berikut ini skenario pengujian *Docker Swarm*.

2.3.1 Menguji koneksi antara manager, worker node



Gambar 6. Pengujian Koneksi Docker Swarm

Koneksi antara host dan IP 10.1.1.22 menunjukkan kualitas yang sangat baik dan respons yang cepat. Hal ini terlihat dari tidak adanya packet loss selama pengujian, serta waktu respons yang stabil di bawah 1 milidetik. Selain itu, host 10.1.1.22 mampu merespons secara konsisten terhadap pengiriman paket data berukuran besar sebesar 5000 byte, yang menandakan kestabilan jaringan dan kapasitas penanganan data yang optimal.

Tabel 2. Pengujian koneksi antar server

No	Status Ping	Los	Waktu Respon	Keterangan
1	OK	0%	0 – 2ms	10.1.1.20/24
2	OK	0%	0 – 1ms	10.1.1.21/24
3	OK	0%	0ms	10.1.1.22/24
4	OK	0%	0ms	10.1.1.23/24

2.3.2 Menguji High Availability (HA)

```
root@master-swarm:~# docker service ls
ID             NAME       MODE         REPLICAS   IMAGE          PORTS
xfno365y48en   TA-SWARM   replicated   2/2        nginx:latest   *:80->80/tcp
root@master-swarm:~#
```

Gambar 7. Docker service

Pada Gambar 7 menunjukkan hasil perintah docker service ls Service bernama TA-SWARM dijalankan dalam mode Replicated, yang berarti jumlah instance (replika) ditentukan secara manual, bukan otomatis seperti pada mode global. Dalam konfigurasi ini, terdapat 2 container aktif dari 2 replika yang diminta, menandakan bahwa semua replika berjalan dengan baik. Image yang digunakan adalah nginx:latest, sehingga setiap container menjalankan web server NGINX. Service ini juga dikonfigurasi dengan port mapping pada port 80, sehingga dapat diakses dari luar melalui HTTP standar. Hal ini memungkinkan layanan web dapat diakses melalui protokol HTTP dari luar klaster. Konfigurasi ini mengindikasikan bahwa mekanisme *high availability* dan *load balancing* di Docker Swarm telah berhasil diuji, karena seluruh replika berjalan dengan status aktif tanpa kesalahan.

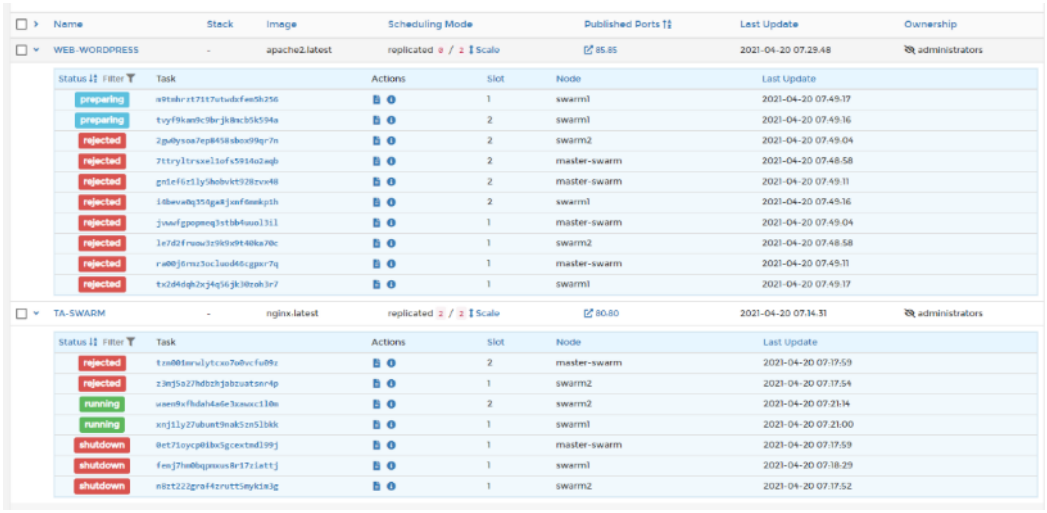
```
root@master-swarm:~# docker node ls
ID                            HOSTNAME       STATUS    AVAILABILITY   MANAGER STATUS   ENGINE
 VERSION
ljd5fln1zj0myvgzv530gw84y *   master-swarm   Ready     Active         Leader           20.10.
3
xsvmnq54aonka0akoi47yp4mm     swarm1         Ready     Active                          20.10.
3
kq1bg73f5v23oirxhecvh13p1     swarm2         Ready     Active                          20.10.
3
```

Gambar 8. Docker Node

Berdasarkan Gambar 8, perintah docker node ls, terlihat bahwa cluster Docker Swarm terdiri dari tiga node: master-swarm, swarm1, dan swarm2. Ketiga node berada dalam status Ready dan Active, yang menunjukkan bahwa semuanya siap menerima tugas. Node master-swarm berperan sebagai Leader, yaitu node manager utama dalam cluster. Sementara itu, swarm1 dan swarm2 bertindak sebagai worker node yang juga aktif. Konfigurasi ini mendukung skenario High Availability (HA) karena jika node leader gagal, salah satu node lainnya dapat diangkat menjadi leader baru, memastikan kelangsungan operasional layanan tanpa gangguan.

2.3.3 Menguji algoritma raft pada docker swarm

Algoritma Raft dalam Docker Swarm berfungsi untuk menjaga konsistensi status cluster dan melakukan replikasi log antar node manager. Dalam pengujian ini, hanya terdapat satu node manager (*master-swarm*), sehingga Raft tidak sepenuhnya menunjukkan mekanisme failover atau pemilihan *master-slave* (*leader election*). Namun, kondisi ini tetap dapat dijadikan dasar untuk menilai stabilitas dan ketergantungan sistem terhadap satu node manager.

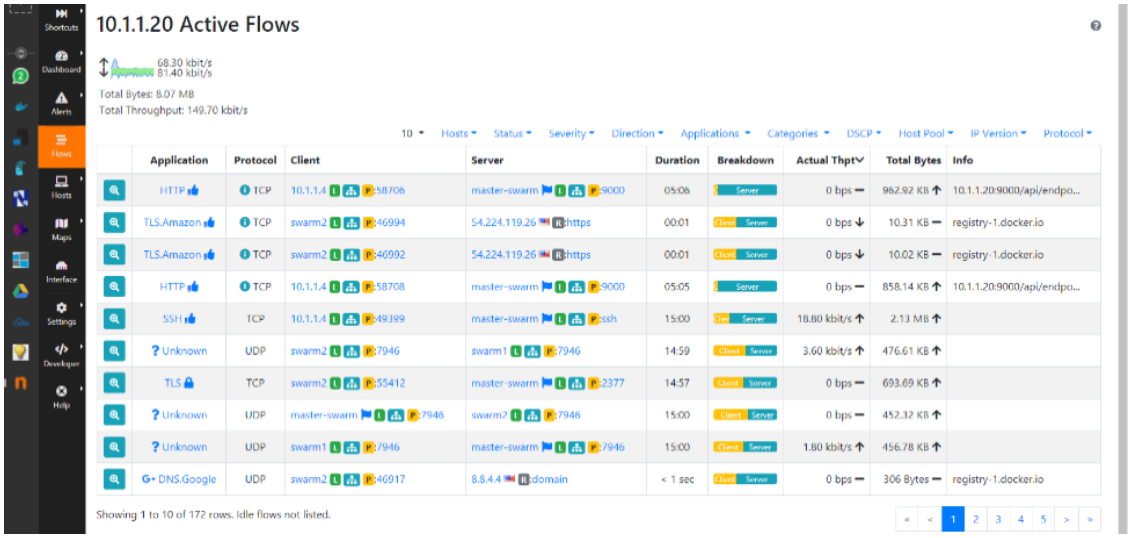


Gambar 9. Algoritma raft pada docker swarm

Algoritma Raft yang digunakan oleh Docker Swarm untuk mencapai keandalan dan konsistensi dalam kluster (swarm) yang terdistribusi. Raft memastikan hanya satu node *master-swarm* (leader) yang bertanggung jawab untuk membuat keputusan seperti men-deploy atau menghentikan layanan. Node slave *swarm1* (follower) menerima log perintah dari *master-swarm* dan mereplikasinya.

2.3.4 Menguji sistem monitoring NtopNG

Ntopng merupakan aplikasi yang cukup ringan namun efektif untuk memonitor sistem Docker Swarm, khususnya dalam mengawasi kestabilan komunikasi antar node, aktivitas layanan kontainer, serta kestabilan jaringan yang berdampak langsung pada algoritma Raft dan distribusi layanan (HA). Monitoring ini sangat penting untuk memastikan konsistensi, keandalan, dan pemulihan otomatis klaster Swarm.



Gambar 10. Penggunaan Ntopng untuk Monitoring Docker-Swarm

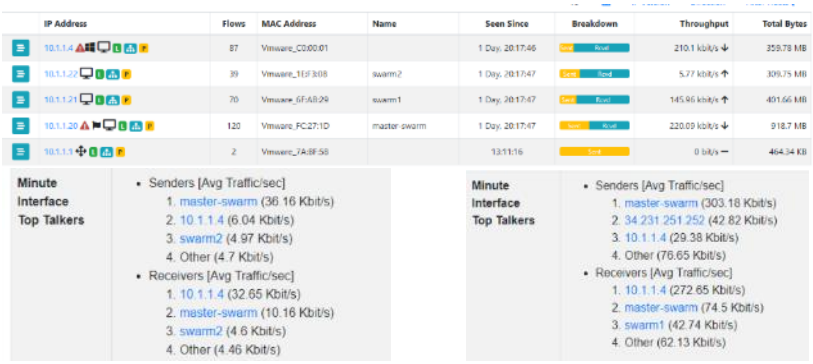
Untuk memverifikasi bahwa node dalam Docker Swarm dapat saling berkomunikasi dengan benar, dan memastikan layanan aktif serta port penting berjalan normal. Fokus utama ada pada komunikasi antar-node dan koneksi terhadap layanan utama (seperti port API Docker, Raft, dan registry Docker). Pada Gambar 10 memperlihatkan monitoring server master-swarm, swarm1, dan swarm1.

3. HASIL DAN PEMBAHASAN

Pada bagian ini disajikan hasil pengujian dan analisis terhadap penerapan Docker Swarm dalam membangun sistem orkestrasi kontainer dengan pendekatan high availability (HA). Pengujian dilakukan dalam beberapa skenario untuk mengevaluasi ketahanan dan konsistensi sistem dalam kondisi operasional yang melibatkan beberapa node, baik sebagai manager maupun worker.

3.1 Menguji Koneksi antar Node (Swarm-master, swarm1, & swarm2)

Konektivitas antar node merupakan prasyarat utama bagi stabilitas sistem Swarm. Oleh karena itu, dilakukan pengujian komunikasi dan kestabilan jaringan antar node menggunakan alat pemantauan jaringan seperti **ntopng** dan pengamatan port aktif, guna memastikan proses pertukaran data berjalan tanpa hambatan.



Gambar 11. Master Swarm Sebelum Proses Replikasi

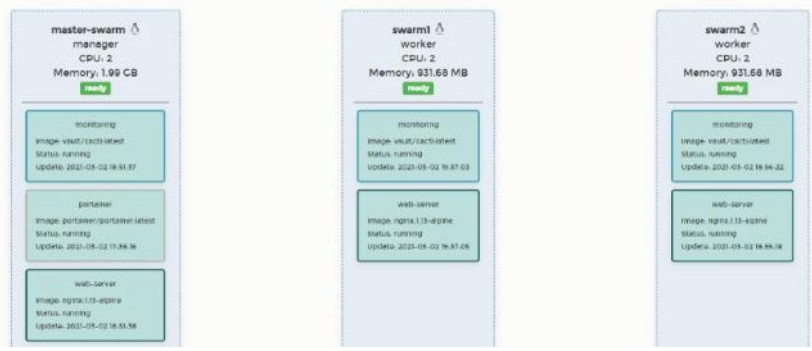
Pada Gambar 11 Docker Swarm ini menunjukkan dinamika trafik jaringan yang bervariasi. Pada kondisi normal, komunikasi internal antar node seperti master-swarm, swarm1, dan swarm2, serta interaksi dengan entitas internal lain seperti 10.1.1.4, berlangsung pada tingkat rendah. Namun, terdapat lonjakan trafik signifikan di mana master-swarm menjadi pengirim data dominan dan 10.1.1.4 sebagai penerima utama, disertai munculnya IP publik, mengindikasikan adanya akses eksternal intensif dan aktivitas internal klaster yang lebih sibuk, seperti deployment atau pemantauan layanan. Fluktuasi ini memperlihatkan kemampuan klaster dalam menangani beban kerja dan interaksi jaringan yang dinamis.

3.2 Menguji High Availability (HA)

Untuk mengevaluasi sejauh mana kemampuan Docker Swarm Clustering mempertahankan layanan ketika terjadi kegagalan (failover) pada node manager utama. Pengujian ini bertujuan untuk mengukur efektivitas replikasi, distribusi service, serta pemilihan ulang pemimpin (leader election) secara otomatis.

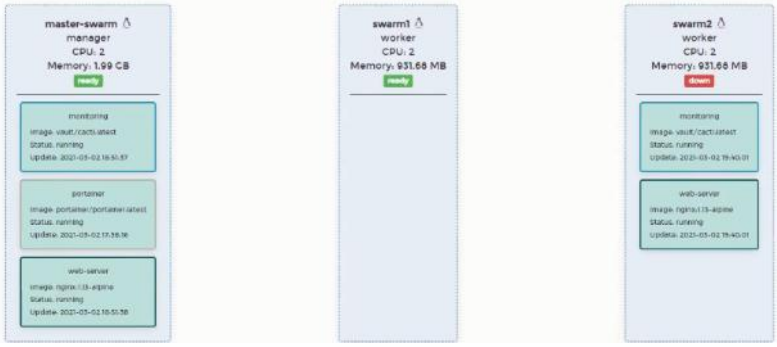
Gambar 12. Master swarm sebelum proses replikasi

Pada Gambar 12 menunjukkan implementasi dasar Docker Swarm Clustering dengan satu node manajer (master-swarm) dan dua node worker (swarm1 dan swarm2).



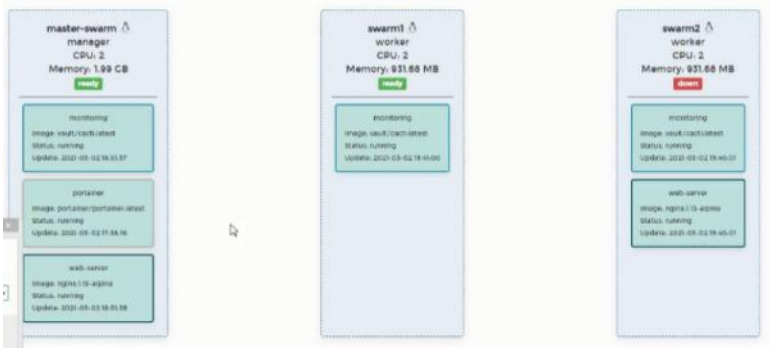
Gambar 13. Master swarm setelah proses replikasi

Gambar 13 menunjukkan konfigurasi Docker Swarm Clustering dengan satu node manajer dan dua node worker, serta bagaimana layanan-layanan telah direplikasi dan didistribusikan di antara node-node tersebut.Gambar ini menunjukkan konfigurasi Docker Swarm Clustering dengan satu node manajer dan dua node worker, serta bagaimana layanan-layanan telah direplikasi dan didistribusikan di antara node-node tersebut. Pada Gambar 12 dengan jelas menunjukkan kemampuan Docker Swarm dalam melakukan replikasi layanan dan distribusi beban kerja (load balancing) secara otomatis. Dengan memiliki beberapa replika dari layanan monitoring dan web-server yang tersebar di semua node (manajer dan worker)



Gambar 14. Proses fail over docker swarm

Pada Gambar 14 dengan jelas menunjukkan kemampuan Docker Swarm dalam melakukan replikasi layanan dan distribusi beban kerja (load balancing) secara otomatis. Dengan memiliki beberapa replika dari layanan monitoring dan web-server yang tersebar di semua node (manajer dan worker)



Gambar 15. Proses fail over docker swarm masih satu aplikasi

Dari Gambar 15, layanan web-server juga memiliki 3 replika. Dengan swarm2 yang down, replika web-server yang ada di sana tidak aktif. Kita melihat satu replika web-server berjalan di master-swarm. Docker Swarm belum berhasil menjadwalkan ulang replika web-server dari swarm2 ke swarm1 karena alasan tertentu (misalnya, swarm1 tidak memiliki cukup sumber daya, atau ada kendala penjadwalan lainnya). Atau, total replika untuk web-server sengaja dikurangi menjadi 2, dan Swarm hanya menjaga 2 replika yang berjalan (satu di master-swarm dan satu di swarm1). Replika yang diinginkan adalah 3, maka Swarm akan terus berusaha menempatkan replika yang hilang dari swarm2 ke swarm1 atau master-swarm. Jika swarm2 down, maka proses re-scheduling belum sepenuhnya selesai atau belum tampil di UI.

Gambar 16. Proses fail over docker swarm berhasil

Pada Gambar 16 replika web-server telah berhasil dijadwalkan ulang dan sekarang berjalan di swarm1. Perhatikan waktu Update pada layanan web-server di swarm1 (2021-05-02 19:41:31) yang sangat dekat dengan waktu Update layanan monitoring di swarm1 (2021-05-02 19:41:00), dan juga dekat dengan waktu down-nya swarm2 (19:40:01). Ini adalah indikasi kuat bahwa Docker Swarm mendeteksi kegagalan di swarm2 dan segera menjadwalkan ulang task yang hilang ke node swarm1 yang masih ready.

3.3 Menguji Algoritma Raft pada Docker Swarm

Sebagai algoritma konsensus yang mendasari kontrol klaster pada Swarm, Raft diuji untuk mengamati proses sinkronisasi log, pemilihan leader, dan replikasi perintah antar manager node. Pengujian mencakup skenario seperti penambahan node, pemutusan node, serta dampaknya terhadap konsistensi state dan pemrosesan layanan.

1. **Awal (Klaster Not Fail & Full Replication):**
   - Gambar 13 menunjukkan Docker Swarm Clustering yang sehat (master-swarm, swarm1, swarm2 semuanya ready).
   - Layanan monitoring dan web-server berhasil direplikasi di semua tiga node (masing-masing 3 replika).
   - Trafik jaringan mungkin relatif stabil atau rendah pada awalnya.
2. **Skenario Fail Node:**
   - Gambar 14 menunjukkan swarm2 dalam kondisi fail/down, ini untuk menunjukan proses simulasi fail node.
   - Sebagai respons terhadap kegagalan ini, algoritma raft pada proses clustering Docker Swarm secara otomatis akan mengulangi lagi proses replikasinya sampai menghasilkan replikasi yang sukses, swarm2 ke node lain yang ready (master-swarm atau swarm1).
   - Gambar 15 mengkonfirmasi keberhasilan *re-scheduling* ini, di mana layanan web-server yang sebelumnya di swarm2 kini terlihat berjalan di swarm1. Layanan monitoring juga sudah memiliki replika di swarm1.

3. Dampak rescheduling

- Gambar 16, dengan banyaknya tugas rejected untuk "WEB-WORDPRESS" dan beberapa "TA-SWARM", mengindikasikan bahwa swarm node mengalami proses *failover*.
- Ketika swarm2 mati, beban kerja dari 1 CPU dan 931.68 MB memori yang semula ditangani oleh swarm2 harus didistribusikan ke master-swarm dan swarm1. Ini secara efektif mengurangi total kapasitas komputasi klaster yang tersedia untuk menjalankan tugas.
- Node master-swarm memiliki 1.99 GB memori dan swarm1 memiliki 931.68 MB. Jika layanan WEB-WORDPRESS dan TA-SWARM yang baru mencoba dijadwalkan (atau replika yang hilang mencoba *reschedule*) membutuhkan lebih banyak sumber daya daripada yang tersedia di master-swarm dan swarm1 (terutama setelah proses replika dari swarm2), maka tugas-tugas tersebut akan rejected.
- Lonjakan trafik yang terlihat di Gambar koneksi (panel kanan) mungkin terjadi pada saat swarm2 mengalami *down* atau setelahnya, karena *client* masih mencoba mengakses layanan, dan trafik yang biasanya didistribusikan ke swarm2 kini harus ditangani oleh master-swarm dan swarm1. Ini bisa meningkatkan tekanan pada sumber daya node yang tersisa, yang semakin memperparah masalah penjadwalan.

4. KESIMPULAN

Penelitian ini menganalisis efektivitas implementasi Docker Swarm dalam membangun sistem orkestrasi kontainer dengan pendekatan *High Availability* (HA). Fokus utamanya adalah mengevaluasi ketahanan dan konsistensi sistem terhadap berbagai skenario operasional yang melibatkan node manajer dan worker. Pengujian dimulai dengan validasi konektivitas antar node klaster: master-swarm, swarm1, dan swarm2. Hasil pemantauan menunjukkan komunikasi antar node berjalan efisien, dengan lonjakan trafik saat terjadi deployment layanan atau akses eksternal. Hal ini menunjukkan klaster mampu merespons beban kerja dan dinamika jaringan secara adaptif.

Uji utama dilakukan terhadap skenario HA, yakni saat node swarm2 sengaja dibuat gagal (*down*). Sebelumnya, layanan monitoring dan web-server tersebar merata di semua node. Ketika swarm2 gagal, Docker Swarm menunjukkan kemampuan *failover* otomatis dengan cepat menjadwalkan ulang kontainer layanan ke node lain seperti swarm1. Proses ini terjadi dalam waktu singkat, menunjukkan keandalan mekanisme konsensus Raft dalam menjaga konsistensi klaster. Namun, penelitian juga mengungkap keterbatasan penting. Setelah swarm2 gagal, terjadi penurunan kapasitas komputasi yang signifikan. Banyak tugas, seperti WEB-WORDPRESS dan TA-SWARM, berstatus *rejected*, menandakan kekurangan sumber daya (CPU atau memori) pada node yang tersisa untuk menampung beban tambahan. Lonjakan trafik setelah kegagalan turut memperbesar tekanan pada sistem.

Docker Swarm terbukti kapabel dalam mendukung sistem HA dengan *failover* otomatis. Namun, untuk menjaga ketersediaan layanan secara optimal, diperlukan perencanaan kapasitas yang matang. Klaster harus memiliki sumber daya cadangan yang cukup agar tetap stabil saat terjadi kegagalan node. Dengan perencanaan yang tepat, Docker Swarm dapat diandalkan untuk orkestrasi kontainer di lingkungan produksi.

5. DAFTAR PUSTAKA

[1] Iqbal Abadilah Umar, Nurhadi, Lailis Syafaah, K. (2021). Analisis efektifitas implementasi sistem aplikasi docker terintegrasi openstack. *JIRE (Jurnal Informatika & Rekayasa Elektronika)* (4), 60–67. https://doi.org/10.36595/jire.v4i1.334

[2] Tuara, H. A., Maridyah, N., & Khaerudin, K. (2021). Implementasi CDN (Content Delivery Network) Menggunakan Cloudflare terintegrasi Dengan Docker Container. *Journal of Mechatronic and Electrical Engineering*. (1), 42–51.

[3] Afirda Desember Riawati, M Irfan, Khaeruddin, A. F. (2022). HIGH AVAILABILITY DYNAMIC SHARDING DATABASE SERVER DENGAN DATABASE SERVER DENGAN METODE FAIL OVER DAN CLUSTERING. *Jurnal Manajemen Informatika Nformatika & Sistem Informasi (MISI)*, *5*(1), 1–10. https://doi.org/10.36595/misi.v5i1.416

[4} Faruq, A., Khaerudin, K., & Lestandy, M. (2020). SISTEM KEAMANAN MULTI MAIL SERVER DENGAN TEKNIK ENKRIPSI MULTI MAIL SERVER SECURITY SYSTEM USING OPENPGP ENCRYPTION. *JTIIK (Jurnal Teknologi Informasi dan Ilmu Komputer) 7*(3), 493–500. https://doi.org/10.25126/jtiik.202071869
[5] Ongaro, D., & Ousterhout, J. (2014). In search of an understandable consensus algorithm. *Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC 2014*, 305–319.
[6] Ileana, M., & Marian, C. V. (2024). *Using Docker Swarm to Improve Performance in Distributed Web Systems*. *March 2025*. https://doi.org/10.1109/DAS61944.2024.10541234
[7] Panthofer, M. (2019). *Mastering Docker Enterprise* (1st ed.). Packt Publishing Ltd.
[8] Turnbull, J. (2015). *The Docker Book: Containerization is the New Virtualization*, 1st ed. San Francisco, CA: James Turnbull Publishing.
[9] Orzechowski, M., B, B. B., & Słota, R. G. (2020). *Reproducibility of Computational Experiments on Kubernetes-Managed Container Clouds with HyperFlow*. *1*, 220–233. https://doi.org/10.1007/978-3-030-50371-0
[10] Dragoni, N., Giallorenzo, S., Lluch-lafuente, A., & Mazzara, M. (2017). *Microservices: yesterday, today, and tomorrow*. *Present and Ulterior Software Engineering*, vol. 2, pp. 195–216, 2017.
[11] Abu-Libdeh, H., Princehouse, L., & Weatherspoon, H. (2010). *RACS: A Case for Cloud Storage Diversity*. 229–240. https://doi.org/10.1145/1807128.1807165
[12] Kapinos-Gorczyca, A., Gorczyca, P., Kapinos, M., & Hese, R. T. (2008). Impossibility of Distributed Consensus with One Faulty Process. *Postepy Psychiatrii i Neurologii*, *17*(2), 171–173.
[13] Orzechowski, M., B, B. B., & Słota, R. G. (2020). *Reproducibility of Computational Experiments on Kubernetes-Managed Container Clouds with HyperFlow*. *1*, 220–233. https://doi.org/10.1007/978-3-030-50371-0
[14] Raft Consensus Algorithm. (2021) The Raft Consensus Project Website. [Online]. Available: https://raft.github.io/
[15] Kithulwatta, W. M. C. J. T., Jayasena, K. P. N., Kumara, B. T. G. S., & Rathnayaka, R. M. K. T. (2022). Docker Containerized Infrastructure Orchestration with Portainer Container-native Approach. *2022 3rd International Conference for Emerging Technology (INCET)*, 1–6. https://doi.org/10.1109/INCET54531.2022.9825257
[16] Deri, L., & Cardigliano, A. (2022). Using CyberScore for Network Traffic Monitoring. *2022 IEEE International Conference on Cyber Security and Resilience (CSR)*, 56–61. https://doi.org/10.1109/CSR54599.2022.9850289
[17] Yekollu, R. K., Haldikar, S. V, Ghuge, T. B., Abdul Kader, O. F. M., & Biradar, S. S. (2024). Resource Management and Scalability in Container Orchestration Platforms: A Comparative Study. *2024 IEEE 16th International Conference on Computational Intelligence and Communication Networks (CICN)*, 1146–1151. https://doi.org/10.1109/CICN63059.2024.10847490
[18] Todorov, M. H. (2024). Performance Analysis of Docker Swarm on Raspberry Pi Clusters. *2024 32nd National Conference with International Participation (TELECOM)*, 1–4. https://doi.org/10.1109/TELECOM63374.2024.10812238