IMPLEMENTASI MEKANISME *LOAD BALANCER* DAN *FAILOVER* PADA IOT *MIDDLEWARE* BERBASIS *PUBLISH-SUBSCRIBE*

SKRIPSI

Untuk memenuhi sebagian persyaratan
memperoleh gelar Sarjana Komputer

Disusun oleh:
Ahmad Naufal Romiz
NIM: 145150200111164

PROGRAM STUDI TEKNIK INFORMATIKA
JURUSAN TEKNIK INFORMATIKA
FAKULTAS ILMU KOMPUTER
UNIVERSITAS BRAWIJAYA
MALANG
2019

PENGESAHAN

IMPLEMENTASI MEKANISME *LOAD BALANCER* DAN *FAILOVER* PADA IOT *MIDDLEWARE* BERBASIS *PUBLISH-SUBSCRIBE*

SKRIPSI

Diajukan untuk memenuhi sebagian persyaratan
memperoleh gelar Sarjana Komputer

Disusun Oleh :
Ahmad Naufal Romiz
NIM: 145150200111164

Skripsi ini telah diuji dan dinyatakan lulus pada
28 Mei 2019
Telah diperiksa dan disetujui oleh:

Dosen Pembimbing I	Dosen Pembimbing II
Eko Sakti P, S.Kom., M.Kom	Widhi Yahya, S.Kom., M.Sc.
NIK: 201102 860805 1 001	NIK: 201607 891121 1 001

Mengetahui
Ketua Jurusan Teknik Informatika

Tri Astoto Kurniawan, S.T, M.T, Ph.D
NIP: 19710518 200312 1 001

PERNYATAAN ORISINALITAS

Saya menyatakan dengan sebenar-benarnya bahwa sepanjang pengetahuan saya, di dalam naskah skripsi ini tidak terdapat karya ilmiah yang pernah diajukan oleh orang lain untuk memperoleh gelar akademik di suatu perguruan tinggi, dan tidak terdapat karya atau pendapat yang pernah ditulis atau diterbitkan oleh orang lain, kecuali yang secara tertulis disitasi dalam naskah ini dan disebutkan dalam daftar pustaka.

Apabila ternyata didalam naskah skripsi ini dapat dibuktikan terdapat unsur-unsur plagiasi, saya bersedia skripsi ini digugurkan dan gelar akademik yang telah saya peroleh (sarjana) dibatalkan, serta diproses sesuai dengan peraturan perundang-undangan yang berlaku (UU No. 20 Tahun 2003, Pasal 25 ayat 2 dan Pasal 70).

Malang, 7 April 2019

Ahmad Naufal Romiz

NIM: 145150200111164

KATA PENGANTAR

Puji syukur penulis panjatkan kehadirat Allah *subhanahu wa ta'ala*, karena atas rahmat dan bimbingan-Nya, penulis dapat menyelesaikan penyusunan skripsi berjudul "Implementasi Mekanisme *Load balancer* dan *Failover* pada IoT *Middleware* Berbasis *Publish-Subscribe*" dengan sangat baik. Sholawat serta salam kepada Nabi Muhammad *shallallahu'alaihi wa sallam* sebagai junjungan terbaik umat manusia akan selalu penulis sampaikan pada setiap kesempatan.

Dalam penyusunan skripsi ini, banyak sekali bantuan yang penulis dapatkan dari berbagai pihak. Dukungan berupa materi dan moral penulis dapatkan sembari menyelesaikan penyusunan skripsi ini. Oleh karena itu, dalam kesempatan ini penulis sampaikan terimakasih kepada:

1. Kedua orang tua tercinta, yang mana tidak akan pernah terbalas budi baik dan jasa-jasa mereka bagi penulis.
2. Bapak Eko Sakti P, S.Kom., M.Kom dan Bapak Widhi Yahya, S.Kom., M.Sc selaku dosen pembimbing I dan II yang selalu membimbing, membantu, meluangkan waktu dan pikiran sehingga penulis dapat menyelesaikan skripsi ini.
3. Bapak Wayan Firdaus Mahmudy S.Si., M.T., Ph.D., selaku Dekan Fakultas Ilmu Komputer.
4. Bapak Tri Astoto Kurniawan, S.T., M.T., Ph.D., selaku Ketua Jurusan Teknik Informatika.
5. Bapak Agus Wahyu Widodo, S.T., M.Cs., selaku Ketua Program Studi Teknik Informatika.
6. Seluruh dosen dan civitas Program Studi Teknik Informatika, Universitas Brawijaya, atas dukungan dan kerjasamanya.
7. Ustadz Muhmmad Faqih dan ustadz Zain selaku guru pembimbing keislaman penulis, Catur Ryan Pamungkas, Pebrio Adi, Reksi Syahputera, dan Juahir selaku kawan seperjuangan penulis dalam berdakwah di organisasi.
8. Guru bela diri penulis shihan Iskamto yang telah memberikan gemblengan mental yang sangat berarti bagi penulis, dan juga kawan-kawan di perguruan karate Mushikawa dojo Al-Ghifari.
9. Teman-teman kelas L, Teknik Informatika Angkatan 2014 yang telah memberikan kenangan berharga selama penulis menempuh *study* di Fakultas Ilmu Komputer.
10. Terima kasih khusus kepada Jessy Ratna Wulandari, yang telah meluangkan waktunya selama berlangsungnya penyusunan skripsi oleh penulis.

11. Dan terimakasih kepada semua pihak yang telah memberikan bantuan dan dukungan untuk penulis.

Penulis mengakui bahwa dalam penyusunan skripsi ini masih terdapat banyak kesalahan dan kekurangan, sehingga penulis membutuhkan kritik dan saran yang membangun dari teman-teman. Akhir kata penulis berharap skripsi ini dapat memberikan manfaat bagi pembacanya.

Malang, 7 April 2019

Penulis

ahmadnromiz@gmail.com

ABSTRAK

Sebelumnya, IoT *middleware* dibangun untuk mengatasi masalah *syntactical interoperability* (silo). IoT *middleware* tersebut diimplementasikan pada Raspberry Pi. Selain itu, untuk membuat fungsi IoT *middleware* lebih *scalable* dikembangkan *cluster message edge storage* yang dapat menambah kapasitas *memory* pada *node* IoT *middleware* tersebut (mesin Raspberry Pi). Pada sistem tersebut keseluruhan proses pengiriman pesan terbatas pada satu *node* IoT *middleware* yang berfungsi sebagai *broker*. Hal itu mengakibatkan tidak seimbangnya *load* dengan menumpuk pada satu *node* IoT *middleware*. Permasalahan ketidakseimbangan *load*, dapat diatasi dengan menambahkan *load balancer* pada sistem, dengan *multiple* IoT *middleware* sebagai tujuan *traffic load balancing*. Algoritme *round robin* digunakan sebagai metode distribusi *traffic* oleh *load balancer*. *Load balancer* dikembangkan sebagai poin masuk tunggal pada sistem yang dibangun. Perangkat *load balancer* yang digunakan sebanyak dua buah. Keepalived juga diimplementasikan agar mekanisme *failover* pada *node load balancer* dapat dilakukan. Pengujian dilakukan untuk mengetahui waktu pemrosesan IoT *middleware* pada pesan *publish* dan *subscribe*. Selain, itu pengujian juga digunakan untuk mengetahui jumlah pesan per detik yang mampu ditangani IoT *middleware*. Dari hasil pengujian, rata-rata nilai *concurrent publish* CoAP adalah 62 pesan/detik pada sistem tanpa *load balancer* dan 63 pesan/detik pada sistem dengan *load balancer*. Nilai rata-rata *concurrent publish* MQTT adalah 41 pesan/detik pada sistem tanpa *load balancer* dan 73 pesan/detik pada sistem dengan *load balancer*. Nilai rata-rata *concurrent subscribe* adalah 37 pesan/detik pada sistem tanpa *load balancer* dan 68 pesan/detik pada sistem dengan *load balancer*.

Kata kunci: *load balancer*, *back end server*, *IoT middleware*, *failover*, *traffic*, *publish-subscribe*

ABSTRACT

Previously, IoT middleware was built to overcome the problem of syntactical interoperability (silo). The IoT middleware was implemented on the Raspberry Pi. In addition, to make IoT middleware functions more scalable, cluster message edge storage was developed which can increase memory capacity on the IoT middleware node (Raspberry Pi engine). In that system, the entire message delivery process was limited to one IoT middleware node which function is as a broker. That results in unbalanced loads by stacking on one IoT middleware node. The problem of unbalanced loads, can be overcome by adding a load balancer on the system, with multiple IoT middleware as the aim of traffic load balancing. Round robin algorithm is used in the research as a traffic distribution method by load balancer. Load balancers are developed as a single entry point on a system. Two devices are used as load balancer. Keepalived is also implemented so that a failover mechanism in the node load balancer can be occured. Testing was carried out to determine the time process done by IoT middleware on publish and subscribe messages. In addition, the testing was also used to determine the number of messages per second which IoT middleware can handle. From the testing result, the average concurrent publish value of CoAP is 62 messages/second on a system without a load balancer and 63 messages/second on a system with a load balancer. The concurrent publish average value of MQTT is 41 messages/second on systems without load balancers and 73 messages/second on systems with load balancers. The concurrent subscribe average value is 37 messages/second on the system without a load balancer and 68 messages/second on the system with a load balancer.

Keywords: load balancer, back end server, IoT middleware, failover, traffic, publish-subscribe

DAFTAR ISI

DAFTAR TABEL

DAFTAR GAMBAR

DAFTAR LAMPIRAN

BAB 1 PENDAHULUAN

1.1 Latar Belakang

IoT *Middleware* adalah perangkat lunak yang bertindak sebagai jembatan antarlingkungan IoT (*Internet of Things*). *Middleware* sebagai jembatan antarlingkungan IoT, menjadi salah satu solusi untuk mengatasi masalah *interoperability* (Razzaque et al., 2016). Seperti *middleware* pada penelitan Anwari, berfungsi sebagai jembatan antarprotokol CoAP, MQTT dan *Websocket* yang dapat mengatasi permasalahan *syntactical interoperability* (Anwari, Pramukantoro, & Hanafi, 2017). Pada penelitan tersebut memiliki satu IoT *middleware* dan penampungan *memory* yang terbatas, sehingga proses pemrosesan pesan *publish* dan *subscribe* terbatas satu perangkat.

Penelitian yang dilakukan Wulandari, Pramukantoro, & Nurwasito (2018) mengembangkan *cluster message edge storage* pada penelitian Anwari. *Cluster* tersebut menambah kapasitas penampungan *memory* pada penelitian Anwari, Pramukantoro, & Hanafi (2017). Wulandari menggunakan Redis untuk mengimplementasikan *cluster message edge storage* pada penelitiannya. Selain itu, pada *middleware* ditambahkan *Ioredis* untuk mengintegrasikan *cluster message edge storage* dengan *middleware*. Pola komunikasi sistem yang digunakan adalah *publish/subscribe*. Adanya *cluster* pada IoT *middleware* sebagai *edge storage* membuat beban komputasi IoT *middleware* menjadi lebih besar, karena harus mendistribusikan data ke *storage* dalam *cluster*. Dalam mengatasi permasalahan tersebut, dibutuhkan solusi untuk mengurangi beban pada IoT *middleware*. Penelitian yang dilakukan oleh Jutadhamakorn et al. (2017) berhasil mengembangkan *load balancing* untuk mendistribusikan *traffic* pada MQTT *client* ke dalam *cluster* MQTT *broker*. Pada penelitian tersebut dikembangkan sebuah *load balancer* sebagai poin masuk tunggal antara MQTT *client* dan MQTT *broker*. Sehingga *traffic* MQTT dari *client* mampu didistribusikan ke *cluster* MQTT *broker*.

Fokus pada penelitian ini adalah diusulkan sebuah mekanisme yang dapat mengurangi beban pada IoT *middleware*, dan mekanisme apabila terjadi kegagalan *node* pada *load balancer.* Peran *load balancer* sebagai poin masuk tunggal dari *publisher* ataupun *subscriber*, berakibat komunikasi pada sistem akan lumpuh jika *node* tersebut mengalami kegagalan. Maka dalam penelitian ini dibuat *load balancing* yang mampu mendistribusikan *traffic* ke beberapa *middleware* yang berfungsi sebagai *broker* dan mekanisme *failover* pada *node load balancer* tersebut. *Load balancer* dibangun pada sistem dengan komunikasi *publish/subscribe*. *Publisher* berkomunikasi dengan *middleware* menggunakan protokol MQTT dan CoAP. Sedangkan *subscriber* berkomunikasi dengan *middleware* menggunakan protokol MQTT. *Load balancer* memanfaatkan API (*application programming interface*) NGINX yang beroperasi di *layer* aplikasi. Hal

tersebut karena CoAP dan MQTT berjalan pada *layer* aplikasi. API NGINX membaca satu koneksi CoAP dan satu koneksi MQTT sebagai *state* yang berbeda. *Load balancer* membaca layanan pada *port* 5683 untuk protokol CoAP dan *port* 1883 untuk protokol MQTT. *Publisher* adalah *node sensor* yang mem-*publish* data dalam bentuk JSON. *Node sensor* CoAP mem-*publish* data ke topik *home/garage*. Adapun *node sensor* MQTT mem-*publish* data ke topik *home/kitchen*. *Subscriber* merupakan *node* yang menerima data dalam bentuk JSON.

Sistem yang dibangun terdiri atas beberapa perangkat. Dua buah perangkat *middleware* digunakan sebagai *back end server* dari *load balancer*. *Back end server* adalah *server* tujuan aliran data pada saat proses *load balancing* dilakukan. *Middleware* diimplementasikan pada perangkat Raspberry Pi. Algoritme distribusi yang digunakan adalah *round robin*. *Round robin* dipilih karena tidak perlu mengetahui *state* dari *back end server* terlebih dahulu sebelum mendistribusikan *traffic*. Selain itu, algoritme *round robin* merupakan algoritme *load balancing* yang paling sederhana (XU Zongyu & Wang Xingxuan, 2015). Oleh karena itu *round robin* cocok diterapkan untuk lingkungan yang homogen. *Load balancer* yang digunakan berjumlah dua buah. Setiap perangkat *load balancer* diimplementasikan perangkat lunak Keepalived. Perangkat lunak tersebut memungkinkan *failover* dilakukan pada *node load balancer*. *Failover* terjadi jika salah satu perangkat *load balancer* mengalami kegagalan proses, maka perangkat *load balancer* lain menggantikan tugas *load balancing* pada perangkat yang gagal. *Load balancer* diimplementasikan pada dua buah perangkat Raspberry Pi.

1.2 Rumusan Masalah

Berdasarkan latar belakang yang telah dijelaskan di atas, maka dapat diambil rumusan masalah sebagai berikut:

1. Bagaimana mengimplementasikan mekanisme *load balancer* dan *failover* pada IoT *middleware*?
2. Bagaimana kinerja IoT *middleware* pada sistem setelah diterapkan mekanisme *load balancer* dan *failover*?
3. Bagaimana perbandingan kinerja IoT *middleware* pada sistem sebelumnya dengan sistem sekarang?

1.3 Tujuan

Tujuan dari penelitian ini adalah sebagai berikut:

1. Mengembangkan sistem yang mampu mendistribusikan *traffic* pada *broker* IoT *middleware* dengan menggunakan *load balancer.*
2. Mengetahui kinerja IoT *middleware* pada sistem setelah diterapkan mekanisme *load balancer* dan *failover*.

1.4 Manfaat

Manfaat yang di harapkan dari penelitian ini adalah sebagai berikut:

1. Meningkatkan ilmu pengetahuan tentang *load balancing*, *failover,* klusterisasi *message broker* dan dapat mengembangkan sistem sebelumnya.
2. Penelitian ini dapat dijadikan dasar topik terkait untuk penelitian lebih lanjut.

1.5 Batasan Masalah

Batasan masalah dari penelitian ini adalah sebagai berikut:

1. Algoritme *load balancing* menggunakan *round robin*.
2. *Node* yang digunakan berjumlah enam buah.
3. Lingkungan penelitian terdiri atas lima jenis *node*, *node publisher*, *node middleware*, *node subscriber*, *node cluster* dan *node load balancer*.
4. Sistem menggunakan tiga mesin *Raspberry* Pi untuk *cluster message broker* dan dua mesin sebagai *load balancer*.
5. Sistem menggunakan Redis sebagai *message broker* dalam bentuk *cluster*.
6. *Load balancer* memanfaatkan API (*application programming interface*) NGINX untuk proses *load balancing*
7. Sistem menggunakan *sensor* DHT11.
8. Topologi kluster yang digunakan yaitu *Mesh*, di mana satu *node* terhubung dengan *node* yang lainnya menggunakan TCP *connection*.

1.6 Sistematika Pembahasan

Sistematika pembahasan dalam penulisan penelitian ini terdiri dari 7 bab yaitu sebagai berikut:

BAB 1 : PENDAHULUAN

Pada bab ini berisi latarbelakang, rumusan masalah, tujuan penelitian, manfaat penelitian, batasan masalah, dan sistematika pembahasan terkait penelitian ini.

BAB 2 : LANDASAN KEPUSTAKAAN

Pada bab ini berisi dasar teori yang diambil dari jurnal *web* resmi berkaitan dengan masalah yang diteliti, kutipan buku dan berisi penjelasan dari penelitian sebelumnya yang digunakan sebagai panduan dalam proses penelitian.

BAB 3 : METODOLOGI PENELITIAN

Pada bab ini berisi langkah dan metode yang digunakan untuk melakukan penelitian.

BAB 4 : PERANCANGAN

Pada Bab ini berisi penjelasan proses perancangan sistem yang diperlukan dalam proses implementasi, yang terdiri atas kebutuhan fungsional, perancangan lingkungan penelitian, alur sistem, desain/model sistem, data, pengujian, skenario pengujian, bagaimana metode analisis, rencana implementasi.

BAB 5 : IMPLEMENTASI

Pada bab ini berisi proses implementasi *load balancing* pada *cluster message broker* yang dilakukan dalam proses penelitian.

BAB 6 : PENGUJIAN DAN PEMBAHASAN HASIL PENGUJIAN

Pada bab ini berisi penjelasan pengujian dan hasil pengujian terhadap sistem yang di implementasikan.

BAB 7 : PENUTUP

Pada bab ini berisi kesimpulan yang didapat dari perancangan, implementasi, pengujian. Serta saran dari seluruh proses dipenelitian untuk pengembangan penelitian selanjutnya.

BAB 2 LANDASAN KEPUSTAKAAN

Bab landasan kepustakaan ini berisi mengenai uraian dan pembahasan tentang teori, konsep, dan juga metode, atau sistem dari literatur ilmiah, jurnal dan *web* resmi. Adapun materi yang hendak dijelasakn pada landasan kepustakaan adalah beberapa hal terkait *Internet of Things*, konsep *middleware* pada IoT, komunikasi protokol yang digunakan pada IoT, *load balancing*, skalabilitas dan kluster Redis.

2.1 Kajian Pustaka

Pada penelitian sebelumnya telah dikembangkan *middleware* dengan tujuan mengatasi masalah *syntactical interoperability*. *Middleware* tersebut berfungsi sebagai *gateway* multi-protokol, MQTT, CoAP dan *websocket*. Sistem yang dibangun menggunakan *middleware* tersebut berbasis *event-driven*. Terdapat *node sensor* sebagai *publisher* yang menggunakan protokol MQTT dan CoAP dalam komunikasinya dengan *middleware*. Sedangkan *node* yang berperan sebagai *subscriber*, menggunakan *websocket* dalam komunikasinya dengan *middleware*. *Message broker* yang digunakan pada penelitian tersebut adalah Redis (Anwari, Pramukantoro dan Hanafi, 2017).

Ada persamaan dan perbedaan, penelitian saat ini dengan penelitian sebelumnya. Persamaan penelitian saat ini dengan penelitian sebelumnya adalah menggunakan lingkungan sistem yang terdiri atas *node sensor*, *middleware*, dan *publisher*. *Middleware* sebagai *gateway* multi-protokol mempunyai tiga *service unit*. *Service unit* tersebut adalah `Find(topik)`, `Subscribe(topik)`, dan `Save(topik,data)`(Anwari, Pramukantoro, Hanafi, 2017). Fungsi `Find(topik)` digunakan untuk mencari data pada Redis untuk topik tertentu. Fungsi `Subscribe(topik)` digunakan untuk subscribe ke topik tertentu. Fungsi `Save(topik,data)` digunakan untuk menyimpan data dengan topik tertentu. Perbedaannya adalah pada penelitian ini terdapat *load balancer* dalam sistem dan *subscriber* pada penelitian ini menggunakan protokol MQTT bukan *websocket*.

Penelitian selanjutnya adalah penelitian yang dikembangkan Wulandari, Pramukantoro dan Nurwasito (2018). Wulandari mengembangkan Redis sebagai *message broker* pada penelitan sebelumnya ke bentuk *cluster*. Wulandari, Pramukantoro dan Nurwasito (2018) mengembangkan enam Redis (tiga *node master* dan tiga *node slave*) yang diimplementasikan ke tiga perangkat Raspberry Pi. *Ioredis* juga diterapkan untuk mengintegrasikan *cluster* dengan *middleware. Cluster* pada *message broker* bertugas mendistribusikan data atau topik ke dalam beberapa *node* Redis (Wulandari, Pramukantoro, Nurwasito, 2018).

Penelitian saat ini dengan penelitian yang dikembangkan oleh Wulandari, terdapat persamaan dan perbedaan. Persamaannya *cluster message edge storage* tetap digunakan agar data atau topik dapat didistribusikan secara acak ke dalam beberapa *node* Redis. *Ioredis* juga masih digunakan untuk menyambungkan *cluster* ke *middleware.* Perbedaannya terdapat *load balancer* yang digunakan sebagai poin masuk tunggal, data dari *publisher* dan *subscriber*. Pada penelitian saat ini

juga menggunakan dua buah *middleware* yang berperan sebagai tempat tujuan distribusi *traffic* oleh *load balancer*.

Penelitian yang dilakukan Jutadhamakorn et al., (2017) berhasil mengembangkan *load balancer* untuk mendistribusikan *traffic* dari *publisher* ke beberapa *broker*. *Broker* pada penelitian tersebut berperan sebagai tempat tujuan *traffic load balancing*. Protokol komunikasi yang digunakan adalah MQTT. Penelitian tersebut menggunakan perangkat lunak NGINX sebagai *load balancer*. *Broker* yang digunakan berbentuk *cluster* menggunakan bantuan *docker swarm*.

Persamaan penelitian saat ini dengan penelitian di atas adalah menggunakan perangkat lunak NGINX sebagai *load balancer*. Sedangkan, perbedaannya terletak pada protokol komunikasi yang digunakan dan *back end server*, tempat tujuan *traffic* didistribusikan. Pada penelitian kali ini, *client* menggunakan dua protokol komunikasi, yakni MQTT dan CoAP. Sedangkan *back end server* menggunakan dua buah *middleware* pada perangkat fisik Raspberry Pi.

Penelitian berikutnya adalah pengujian aspek non-fungsional *middleware* dilakukan oleh Pramukantoro, Yahya, dan Bakhtiar (2017). Penelitian tersebut menguji aspek performansi dan skalabilitas pada pengembangan *middleware* yang dilakukan oleh Anwari, Pramukantoro dan Hanafi (2017). Metode pengujian menggunakan *performance testing* dan *scalability testing*. *Performance Testing* menggunakan parameter performa CPU, dan *memory*. Jeda transmisi *middleware* pada komunikasi antar-*node* juga digunakan pada *performance testing*. Sedangkan untuk *scalability testing* menggunakan *package async* pada *npm*. Program *asynchronous* dijalankan untuk mengirimkan data dengan variasi jumlah *client* 100, 500 dan 1000.

Persamaan penelitian saat ini dengan penelitian di atas adalah menggunakan program *asynchronous* untuk menjalankan *scalability testing*. Perbedaannya adanya tambahan parameter yang ditampilkan. Parameter tersebut adalah *time publish* dan *time subscribe* serta variasi *client* ditambah menjadi 1500.

Tabel 2.1 Kajian Pustaka

No	Nama Penulis, Tahun, Judul	Persamaan	Perbedaan	
			Terdahulu	Sekarang
1	Husnul Anwari, 2017, "Pengembangan Iot *Middleware* Berbasis *Event-Based* dengan Protokol Komunikasi	Penggunaan *middleware* dengan *service unit* yang sama	Pengembangan *middleware* sebagai *gateway* multi-protokol, CoAP dan MQTT untuk *publisher* dan *websocket* untuk *subscriber*	Terdapat *load balancer* pada sistem, selain itu *subscriber* menggunakan protokol MQTT

Tabel 2.1 Kajian Pustaka (lanjutan)

	CoAP, MQTT dan *Websocket*"			
2	Jessy Ratna W., 2108, "Implementasi *Cluster Message Broker* Sebagai Solusi Skalabilitas *Middleware* Berbasis Arsitektur *Publish-Subscribe* pada *Internet of Things* (IoT)"	Penggunaan Redis sebagai *cluster message broker* pada tiga perangkat Raspberry Pi dengan enam *node* (tiga *node master* dan tiga *node slave*) Penggunaan *loredis* untuk mengintegrasikan *cluster* dengan *middleware*	*Cluster message broker* Redis digunakan untuk mendistribusikan data ke beberapa *node*	*Load balancer* digunakan sebagai poin masuk tunggal *publisher* dan *subscriber*, sebelum data masuk ke *cluster message broker* Dua buah *middleware* digunakan untuk tempat tujuan distribusi *traffic* oleh *load balancer*
3	Jutadhamakorn et al., 2017, "*A Scalable and Low-Cost MQTT Broker Clustering System*"	Penggunaan perangkat lunak NGINX sebagai *load balancer*	Menggunakan satu protokol komunikasi, yakni MQTT *Back end server* berbentuk *cluster* menggunakan bantuan *docker swarm*	Menggunakan dua protokol komunikasi, yakni MQTT dan CoAP *Back end server* menggunakan dua buah *middleware* pada perangkat Raspberry Pi
4	Eko S. P., 2017, "*Performance evaluation of* IoT *middleware for syntactical Interoperability*"	Penggunaan program *asynchronous* untuk menjalankan *scalability testing*	*Package async* menggunakan *npm* untuk skenario *scalability testing*, dan variasi jumlah	Menampilkan parameter *time publish* dan *time subscribe* serta variasi *client* menjadi 100,

Tabel 2.1 Kajian Pustaka (lanjutan)

			client yang digunakan adalah 100, 500 dan 1000 *client*	500, 1000, dan 1500

2.2 Dasar Teori

Dasar teori membahas teori dasar yang menunjang proses penelitian.

2.2.1 Internet of Things

Terdapat istilah yang banyak menjadi topik penelitian dan perbincangan pada bidang *metworking* adalah *Internet of Things*. IoT merupakan konsep yang mempunyai kemungkinan agar berbagai benda (*things*) terhubung ke jaringan (*internet*). Pada IoT juga membuat kemungkinan adanya *smart environtment* yang bekerja memanfaatkan berbagai teknologi yang tergabung di dalamnya. Perangkat fisik pada jaringan IoT, umumnya dipasang dengan seperti actuator, *sensor* nirkabel dan tag RFID. Selanjutnya perangkat tersebut terhubung ke internet membentuk jaringan IoT.

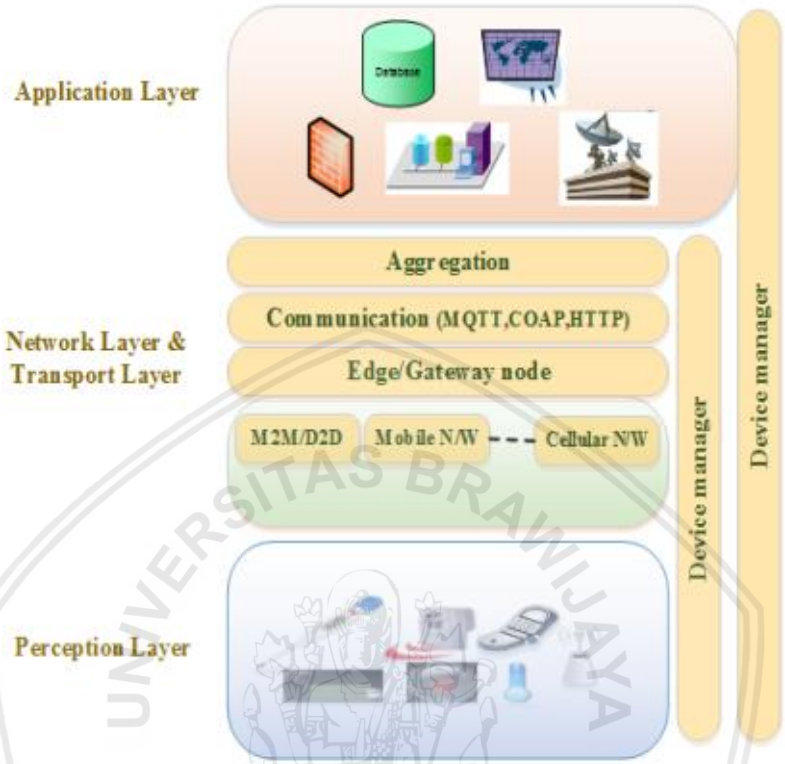
IoT berperan penting dalam menciptakan *smart environtment*. Perangkat IoT diterapkan berdasarkan konteks aplikasi mulai dari jaringan seluler, komunikasi Machine-to-Machine (M2M), kemudian dari *vehicular network* ke *wireless sensor networks* (WSN) dan juga dalam sistem terdistribusi. Hasil dari semua itu contohnya scenario *smart home*, *smart buildings*, smart *city*, dan *health automation* (Tayyaba et al., 2017).

2.2.1.1 Komponen Arsitektural IoT

IoT memiliki beberapa komponen yang menyusunnya. Ketika jaringan IoT membentuk sebuah jaringan, tentu saja komponen tersebut memiliki perbedaan dengan konsep jaringan konvensional. Sebagaimana yang dijelaskan Tayyaba et al. (2017) komponen-komponen arsitektural IoT adalah sebagai berikut:

- *Perception layer:* lapisan ini merupakan objek fisikal yang berupa *sensor*, aktuator, RFID, perangkat bergerak, *bluetooth*, dan sebagainya. Perangkat-perangkat bertugas untuk mendapatkan informasi dari tempatnya kemudian dikirimkan ke bagian *edge network* seperti *gateway* atau sink
- *Network Layer*: lapisan ini bertanggungjawab untuk mengirimkan data dari objek fisikal menuju *gateway*/*edge network* untuk pemrosesan lebih lanjut. Teknologi pengiriman yang berperan antara lain ZigBee, *bluetooth*, dan Wi-Fi
- *Application layer*: lapisan ini berurusan dengan layanan yang diinginkan pengguna dengan memanipulasi informasi yang dikumpulkan pada *perception layer* dan diproses pada sistem pengoperasian
- *Middleware layer*: *middleware* menyediakan layanan untuk menerjemahkan pesan dari suatu layanan informasi tanpa memperatikan

detail perangkat kerasnya. *Middleware layer* terasosiasi dengan manajemen layanan, pengalamatan, dan penamaan dari layanan yang diinginkan



Gambar 2.1 Arsitektur IoT

Sumber: Tayyaba et al. (2017)

2.2.2 MQTT (Message Queue Telemetry Transport)

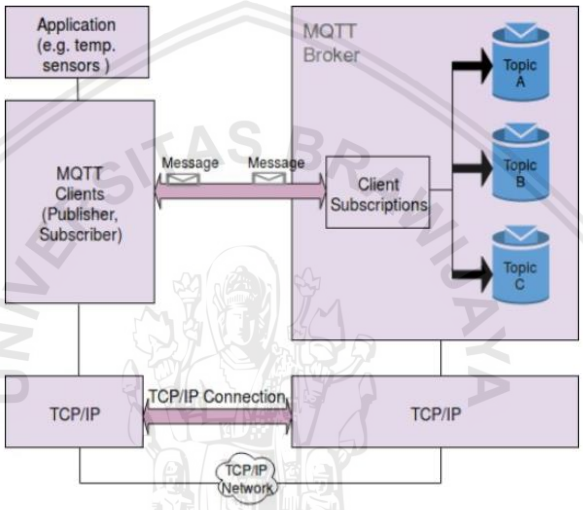
MQTT menyediakan layanan komunikasi berbentuk *publish/subscribe* antara perangkat (klien) dengan server. MQTT menyediakan protokol pengiriman pesan yang ringan antar aplikasi M2M/IoT (Antonic et al., 2015). Pada tahun 1999, MQTT diciptakan sebagai protokol dengan hak milik IBM. Akan tetapi, sekarang MQTT telah bersifat *open source* dan semenjak 2014 segala aktifitas standarisasi MQTT dilakukan oleh OASIS (*Advancing Open Standards for The Information Society*). MQTT juga menyediakan skalabilitas dan efisiensi *cost* untuk menghubungkan perangkat-perangkat melalui internet.

Pada protokol MQTT, mendefinisikan dua entitas, klien dan *broker*. Klien dapat men-*subscribe* topik yang diinginkan serta mem-*publish* konten yang dirasa perlu, sedangkan *broker* bertugas untuk pemrosesan terkait *publications* dan *subscriptions*. MQTT mempunyai cara komunikasi dengan implementasi yang sederhana pada sisi klien, sedangkan seluruh kompleksitas proses terjadi di *broker* (Antonic et al., 2015).

2.2.2.1 Arsitektur MQTT

Arsitektur MQTT terbagi atas dua bagian utama, *client* dan *broker*. Gambar 2.1 menerangkan dengan singkat komponen dari MQTT. Setiap komponen dijelaskan sebagai berikut (Soni dan Makwana, 2017):

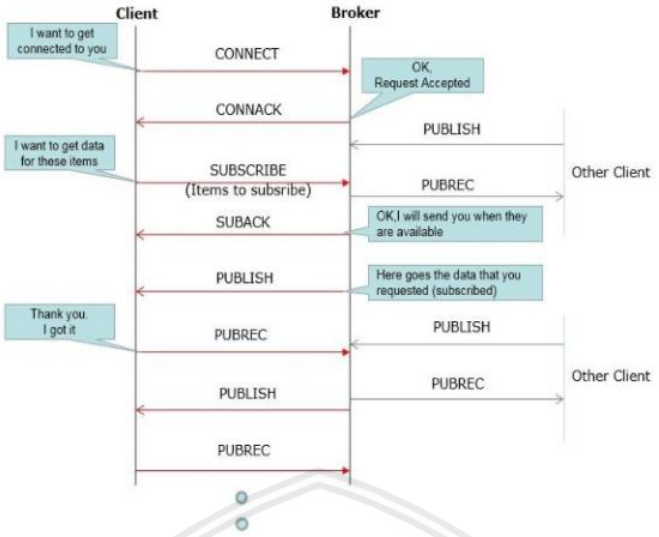
1. *Client*: *client* dapat berupa *subscriber* atau *publisher*, keduanya selalu menjaga koneksinya dengan *server* (*broker*). *Client* berperan sebagai berikut:
   - Mem-*publish* pesan yang diinginkan pengguna
   - *Subscribe* ke subjek yang diinginkan untuk mendapatkan pesan informasi
   - *Unsubscribe* untuk mengekstraksi subjek yang diikuti
   - Lepas dari *broker*



Gambar 2.2 Arsitektur MQTT

Sumber (Soni dan Makwana, 2017)

2. *Broker*: fungsi utama *broker* mengontrol distribusi informasi dan bertanggung jawab untuk menerima semua pesan dari *publisher*, menyaring pesan tersebut dan menentukan *client* mana yang tertarik dengan pesan tersebut, lalu menyebarkannya ke semua *subscriber*. Gambar 2.2 menerangkan bagaimana MQTT bekerja. *Broker* mampu melakukan hal berikut:
   - Menerima permintaan *client*
   - Menerima pesan yang di-*publish*
   - Memproses berbagai permintaan seperti *subscribe*, *unsubscribe* dari pengguna
   - Setelah menerima pesan dari *publisher*, kemudian mengirimkan pesan yang diinginkan tersebut ke pengguna terkait



Gambar 2.3 Cara Kerja MQTT

Sumber (Soni dan Makwana, 2017**)**

2.2.2.2 *Quality of Service*

MQTT memiliki tiga level *quality of service* (QoS). Setiap pesan yang di-*publish* memiliki salah satu diantara ketiga level tersebut. Hal ini menjamin *reliability* dari setiap pesan. Level QoS tersebut dijelaskan sebagai berikut:

a. Level 0: pada level 0 pesan dikirimkan sekali bergantung pada *reliability* TCP pada jaringan. QoS level 0 tidak akan mengirim pesan kembali apabila mengalami kegagalan pada pengiriman pesan.
b. Level 1: pada level 1 pesan setidaknya dikirimkan sekali. Akan tetapi, pesan bisa dikirimkan lebih dari sekali jika *subscriber* tidak mengirimkan pesan ACK.
c. Level 2: pada level 2 dipastikan pesan diterima satu kali. MQTT dengan level QoS 2 akan memastikan pesan akan tersampaikan, dan tidak akan terjadi duplikasi. Akan tetapi, perlu diperhatikan, semakin tinggi level QoS yang digunakan, semakin tinggi juga kemungkinan *overhead* terjadi.

2.2.3 CoAP (*Constrained Application Protocol*)

CoAP adalah protokol transfer *web* yang khusus pada kondisi *node* dan jaringan yang terbatas. Protokol CoAP menyediakan interaksi *request/respond* antara aplikasi *endpoint*, mendukung fitur *discovery of service* dan sumber daya, juga termasuk konsep kunci dari *web* seperi URI dan *internet media types*. Protokol CoAP didesain agar mudah berantarmuka dengan protokol HTTP. CoAP berintegrasi dengan *web* dengan menjembatani kebutuhan khusus akan *multicast*, *low-overhead*, dan *simplicity* untuk lingkungan *constrained* (IETF, The *Constrained Application Protocol*, 2014).

Tidak seperti HTTP, CoAP berurusan dengan pertukaran data asinkron melalui transport *datagram-oriented* seperti UDP. CoAP mendefinisikan empat tipe pesan, *confirmable*, *non-confirmable*, *acknowledgement*, dan *reset*. Dasar pertukaran dari keempat pesan tersebut seperti berbentuk orthogonal terhadap

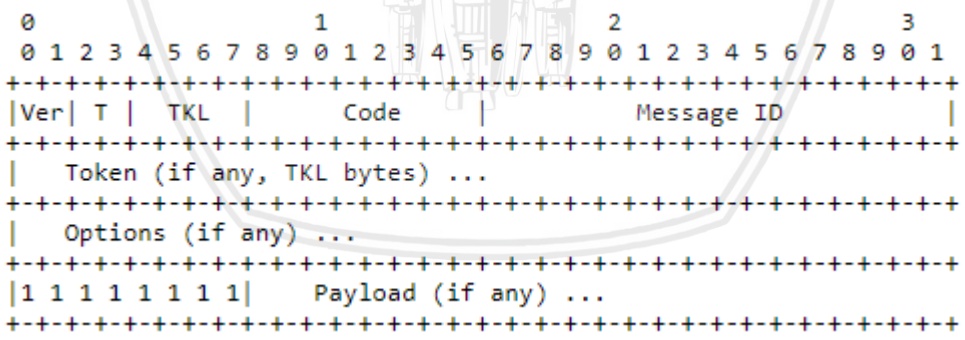
interaksi *request/response*. *Request* dapat dibawa melalui pesan-pesan *confirmable* dan *non-confirmable*. *Response* juga dapat dibawa melalui pesan-pesan tersebut seperti membawa pesan *acknowledgement*.

Berikut adalah kelebihan yang ada pada protokol CoAP (IETF, The *Constrained Application Protocol*, 2014):

1. Protokol *web* yang memenuhi kebutuhan M2M (*machine-to-machine*) dalam lingkungan terbatas.
2. UDP [RFC0758] yang mendukung reliabilitas *unicast* dan *multicast*.
3. Pertukaran pesan secara asinkron.
4. *Overhead* dan *parsing complexity* yang rendah.
5. Dukungan terhadap URI dan *content-type*.
6. Kemampuan *proxy* dan *caching* yang sederhana
7. *Stateless* HTTP *mapping*, memungkinan *proxies* dibangun dengan menyediakan akses ke sumberdaya CoAP melalui HTTP pada cara yang seragam.
8. Dukungan terhadap DTLS [RFC6347].

2.2.3.1 Model Pesan CoAP

Fitur pada *header* CoAP adalah *request/response*. CoAP pada dasarnya didesain berdasarkan pertukaran pesan antara *endpoint* melalui transport UDP, namun CoAP juga dapat digunakan pada protokol transport lainnya, seperti SMS, TCP, SCTP dan DTLS. Masing-masing pesan pada CoAP mengandung *message* ID yang berguna untuk deteksi duplikasi apabila menggunakan pilihan *reliability*. Gambar 2.3 adalah format pesan dari protokol CoAP.



Gambar 2.4 The CoAP *message format*

Sumber (IETF, 2014)

2.2.4 Skalabilitas

Perspektif IoT pada tahun-tahun terakhir menjadi sangat relevan, begitu juga jumlah *sensor*/aktuator yang terlibat di IoT. Sedangkan banyak penelitian menunjukkan jumlah perangkat IoT meningkat secara eksponensial pada masa yang akan datang (Bellavista dan Zanni, 2016). Oleh karena itu, skalabilitas menjadi salah satu topik IoT yang menjadi penelitian dan eksploitasi. Ketika menagani masalah skalabilitas, harus juga memperhatikan hal lainnya. Masalah seperti distribusi perangkat berdasarkan letak geografis pada area yang luas,

interoperabilitas, dan efisiensi perlu diperhatikan dalam membangun skalabilitas pada IoT.

Dalam sistem yang *scalable* sistem diharapkan mampu beradaptasi terhadap kebutuhan pengguna yang luas. Skalabilitas ini, berkaitan dengan tingkatan *delay* dan sumber daya pada suatu sistem. Ketika jumlah data dan memori meningkat, sistem harus mampu menangani hal tersebut dengan *delay* dan konsumsi sumber daya yang minimal. Oleh karena itu, sistem tersebut dapat dikatan *scalable*.

Skalabitas terbagi menjadi dua sudut pandang, skalabilitas yang bersifat horizontal dan vertikal. Skalabilitas vertikal adalah kemampuan *hardware* atau *software* untuk meningkatkan kapasitasnya dengan menambahkan sumber daya tambahan pada *hardware* atau *software* tersebut (Gupta, Christie dan Manjula, 2017). Contohnya seperti menambahkan daya *processor* ke *server* untuk menambah kecepatan memprosesnya. Lebih dari itu pada skalabilitas vertikal, penambahan *main memory*, *storage*, dan *network interface* ke dalam *node* tunggal adalah cara untuk memenuhi *request* per sistem. Adapun skalabilitas horizontal adalah kemampuan untuk meningkatkan kapasitas, dengan cara menyambungkan banyak entitas *hardware* atau *software* sehingga dapat bekerja bersama sebagai satu unit (Gupta, Christie dan Manjula, 2017). Skalabilitas horizontal dapat dicapai dengan menambahkan mesin ke dalam sekelompok sumber daya dan menambahkan *node* ke dalam sistem. Contohnya menambahkan komputer ke dalam aplikasi terdistribusi.

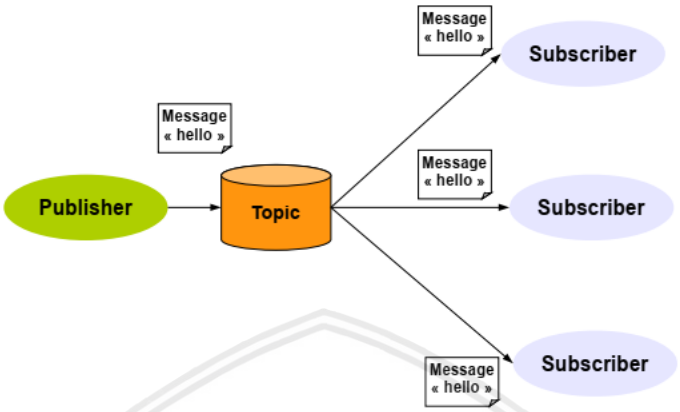
2.2.5 Arsitektur *publish-subscribe*

Pada intinya, arsitektur *publish-subscribe* merupakan arsitektur berdasarkan *subject of interest* (Scalagent, 2014). Pesan di *publish* sekali pada topik tertentu, dan setiap konsumen yang terdaftar atau men-*subscribe* topic tersebut akan menerima pesan dari *publisher*. Diantara *publisher* dan *subscriber* terdapat *broker* yang berfungsi sebagai *server*. *Broker* mensesuaikan setiap pesan yang di-*publish* ke *subscriptions*. Jika tidak ada pesan yang sesuai, pesan tersebut akan dibuang. Jika ada satu atau beberapa pesan sesuai, *broker* akan meneruskan pesan tersebut ke *subscriber*. Gambar 2.4 menerangkan tentang interaksi *publish-subscribe*.

Model *broker* memiliki beberapa kelebihan sebagai berikut (Scalagent, 2014):

- *Space decoupling*: aplikasi yang menggunakan pola interaksi *publish/subscribe* tidak perlu mengetahui lokasi aplikasi yang lain. Satu-satunya alamat yang harus diketahui oleh aplikasi adalah alamat jaringan *broker*. *Broker* kemudian merutekan pesan ke aplikasi yang benar berdasarkan data semantik.
- *Time decoupling*: *publisher* dan *subscriber* tidak perlu berada pada waktu yang sama. Aplikasi *publisher* dapat mengirimkan pesan *publish* kemudian mengakhirinya. Pesan tetap akan tersedia untuk *subscriber* beberapa waktu kemudian.

- *Reliability*: *broker* menjamin reliabilitas pesan antara *publisher* dan *subscriber*. Hal ini karena *broker* dapat menjamin aplikasi *publisher* dan *subscriber* saling berkirim pesan tanpa adanya *coupling*.



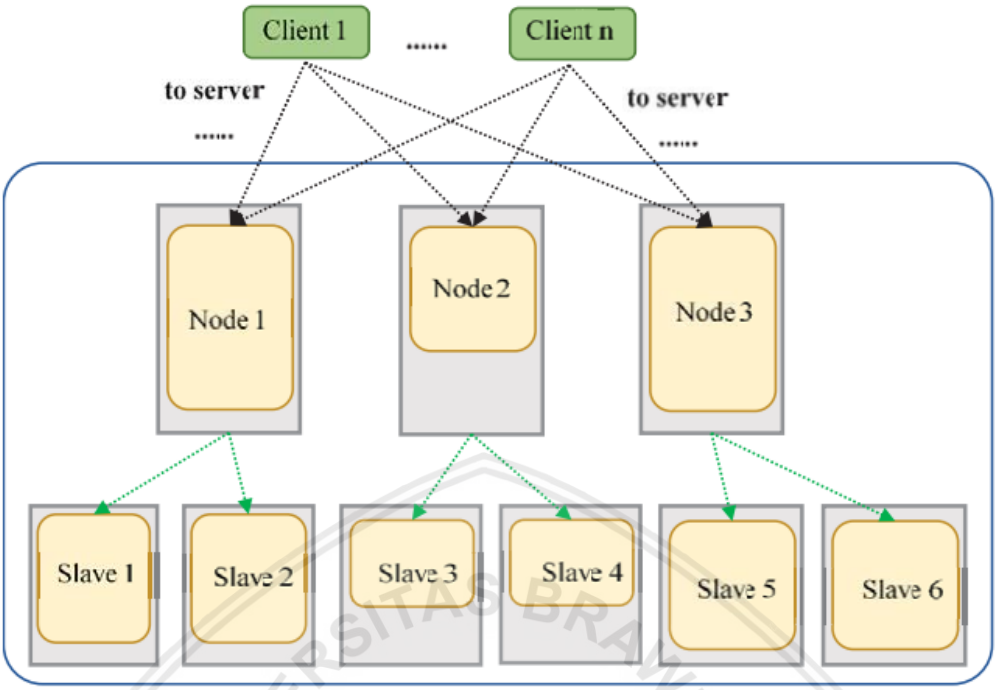
Gambar 2.5 Pola Interaksi *Publish/Subscribe*

Sumber: Scalagent, 2014

2.2.6 Redis dan klasterisasi

Redis merupakan *open source* (BSD *license*) yang dapat digunakan sebagai *database*, *cache* dan *message broker*. Redis adalah sistem penyimpanan *in-memory* dengan metode *key-value*. *Redis* mendukung struktur data seperti *string*, *hash*, dan *list*. Beberapa fitur yang dimiliki Redis adalah *built-in replication*, *Lua scripting*, LRU *eviction*, *transaction* dan berbagai level pada *disk persistent*. Selain itu dengan adanya Redis Sentinel, Redis juga mendukung availabilitas yang tinggi. Dan juga fitur Redis *cluster* yang mendukung partisi otomatis. Hal lain yang utama, bahwa Redis mendukung pola komunikasi *publish/subscribe*.

Dukungan klasterisasi yakni mampunya data dapat disimpan ke *node-node* yang terdistribusi. Hal ini bertujuan agar kapasitas penyimpanan pada suatu sistem bertambah. Adanya konsep skalabilitas horizontal, yakni menambahkan kemampuan dan kapasitas sistem dengan menambahkan berbagai mesin didukung oleh adanya fitur klasterisasi pada Redis. Pada Redis teknik klasterisasi terbagi menjadi *master node* dan *slave node*. *Master node* merupakan *node* utama pada Redis, sedangkan *slave node* adalah replikasi dari *master node* tersebut. Gambar 2.5 merupakan struktur layanan penyimpanan terdistribusi Redis menggunakan *master node* dan *slave node*.

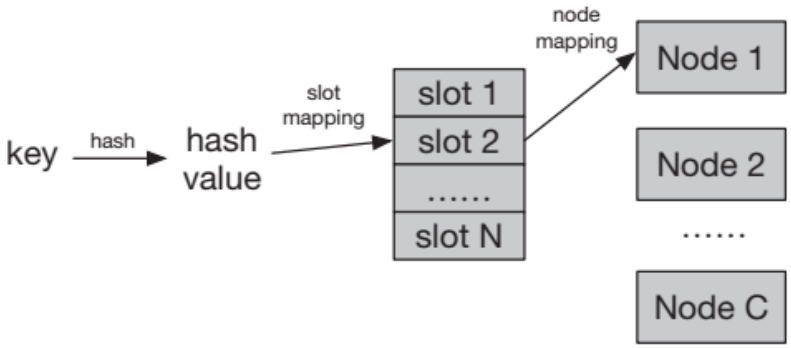


Gambar 2.6 Struktur Layanan Penyimpanan Terdistribusi pada Redis

Sumber: Chen et al., 2016

2.2.6.1 Desain Distribusi Data pada Redis

Redis adalah sistem basis data *key-value* dengan teknik penyimpanan *key-slot-node* dalam memetakan data. Gambar 2.6 menunjukkan bagaimana Redis memetakan *key* ke setiap *node*. Pada fase *hash*, Redis akan mengkalkulasi nilai *hash* pada *key* yang diberikan. Setiap nilai *hash* yang dihasilkan, ditetapkan ke setiap slot yang berjumlah 16382. Kemudian pada slot fase *mapping*, dapat ditemukan slot yang sesuai untuk *key* berdasarkan nilai *hash*-nya. Pada Redis setiap slot dapat ditetapkan secara dinamis ke *node* yang berbeda. Jika ada *node* yang memiliki *workload* terlalu berat, slot tersebut dapat dapat dipindahkan ke *node* yang lain, atau bahkan ke *node* baru agar kinerja sistem meningkat. Pada tahap ketiga dapat ditemukan *node* yang sesuai dengan *key*, berdasarkan status pemetaan *slot-node*.



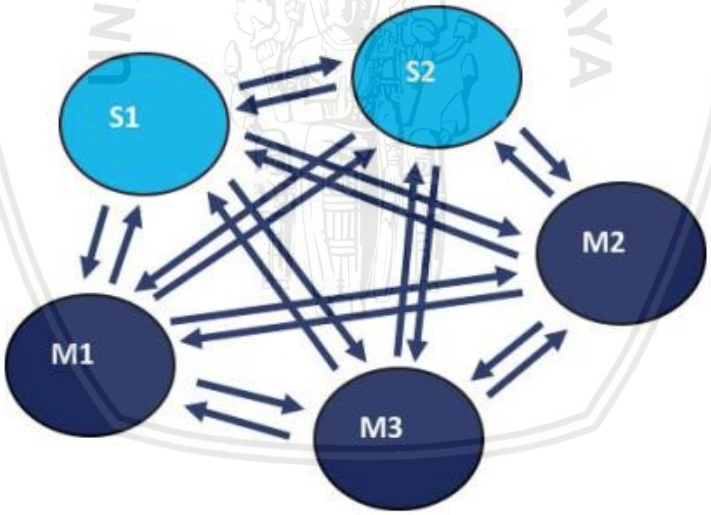
Gambar 2.7 Tiga Tahapan *Mapping Key* ke *Node*

Sumber: Chen et al., 2016

2.2.6.2 Desain Skalabilitas Sistem Redis

Redis pada salah satu fiturnya mampu mengatasi *single-node failure* menggunakan fitur *automatic failure*. Yakni, ketika *slave node* dipromosikan menjadi *master node*. *Slave node* yang lainnya akan di re-konfigurasi untuk menggunakan *master node* yang baru, kemudian aplikasi yang menggunakan *server* Redis diberikan informasi tentang alamat baru tersebut, untuk kemudian digunakan pada saat nanti terkoneksi (Redis, 2018).

Redis menggunakan desentralisasi *node* dan *gossip protocol* untuk menjaga informasi penting pada setiap *node*. Sebagaimana yang dijelaskan pada gambar 2.7, setiap *node* pada sistem terhubung penuh dan mengetahui status terkini dari sistem. Ketika status sistem berubah, informasi ini akan disebar ke setiap *node*. *Node* juga secara acak mengirimkan pesan PING ke *node* lainnya, dan berharap mendapatkan pesan balasan berupa pesan PONG. Pesan PONG adalah bukti bahwa kluster bekerja dengan benar. Jika ditemukan *node* yang tidak berkerja dengan benar, setiap *node* akan melakukan *voting* untuk menggunakan *slave node*, menggantikan *master node* yang *down*.



Gambar 2.8 Desain Desentralisasi *node* pada Redis dengan Gossip *protocol*

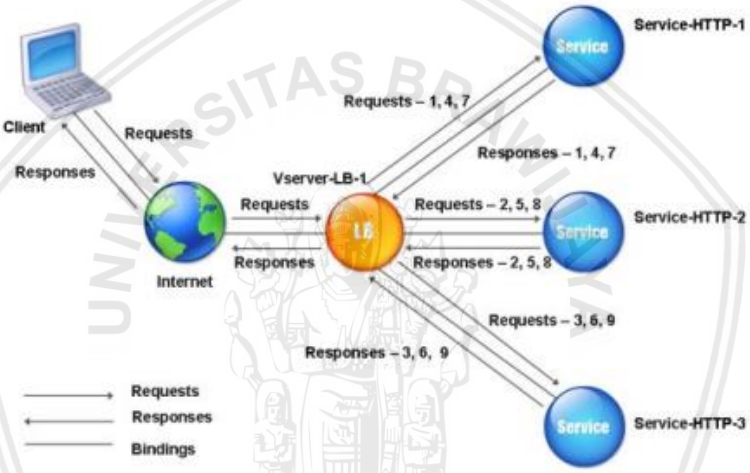
Sumber: Chen et al., 2016

2.2.7 *Load Balancing*

Load balancing adalah teknik untuk memanfaatkan sumber daya komputasi yang tersedia dengan lebih efektif, dengan cara membagi tugas berdasarkan strategi yang diinginkan (Balasubramanian et al., 2004). *Load balancing* biasa digunakan untuk mendistribusikan *traffic* jaringan ke *server*. Distribusi tersebut bisa menggunakan algoritme yang telah disesuaikan parameternya. Hal ini dikarenakan kebutuhan pada setiap *server* untuk setiap jenis *request* berbeda. Dengan digunakannya *load balancing*, *server* diharapkan dapat melayani *request* dengan *cost* yang efisien. Arsitektur *load balancing* berada

diantara *client* dan *server*. Ketika terjadi *request*, *load balancing* akan mengantarkan *request* tersebut ke *server* yang dianggap mempunyai kemampuan untuk menangani *request* tersebut. Gambar 2.5 menunjukkan alur *load balancing* tradisional bekerja.

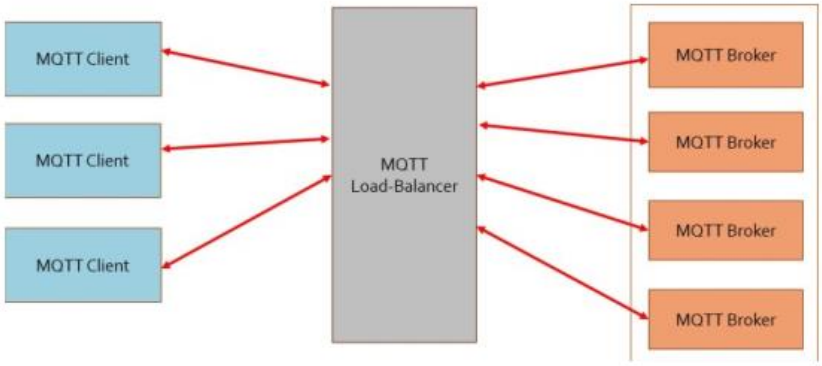
Teknik *load balancing* telah dipelajari secara ekstensif. Biasanya aplikasi dengan *load* reguler, dapat menerapkan *static load balancing*. Penerapan statik *load balancing* dengan cara memetakan data ke *processor* secara hati-hati. Banyak algoritme yang telah dikembangkan untuk tujuan partisi statik *computational mesh*. Metode ini memodelkan komputasi sebagai graf, dan menggunakan algoritme graf partisi untuk membagi graf diantara *processor* (Godha dan Prateek, 2014). Sedangkan aplikasi non-reguler menambahkan *work stealing* untuk melakukan *task scheduling*.



Gambar 2.9 *Load Balancing* Tradisional

Sumber: Godha dan Prateek, 2014

Pada lingkungan IoT, metode *load balancing* yang sesuai untuk mengurangi konsumsi energi di jaringan dapat memperpanjang usia lingkungan tersebut (Wajgi dan Thakur, 2012). Pada IoT metode *clustering* merupakan salah satu metode untuk meningkatkan usia lingkungan IoT. Metode *clustering* didapatkan dengan cara mengatur jaringan-jaringan IoT kedalam satu kelompok. Pada tiap kelompok setidaknya terdapat satu *head node.* Beragam kelebihan yang ditawarkan oleh metode *cluster*, seperti berkurang ukurang tabel *routing*, konservasi *bandwith* jaringan, memperpanjang usia jaringan, berkurangnya paket data yang redundan, dan konsumsi energi yang rendah (Wajgi dan Thakur, 2012). *Load balancing* merupakan komponen esensial dalam solusi manajemen pada IoT untuk operasi jaringan yang lancer. Gambar 2.6 menjelaskan contoh skema penggunaan *load balancing* pada lingkungan IoT menggunakan protokol MQTT.



Gambar 2.10 Skema MQTT *Load Balancing*

Sumber: Jutadhamakorn et al., 2017

Teknik *load balancing* pada lingkungan IoT membutuhkan algoritme yang sesuai. Pada lingkungan IoT perangkat yang digunakan rata-rata memiliki spesifikasi *resource constrain*. Oleh karena itu, perlu dipertimbangkan algoritme yang sesuai dengan kondisi perangkat IoT. Algoritme *round robin* adalah adalah algoritme yang sesuai dengan lingkungan IoT (XU & WANG, 2015). *Round robin* bekerja secara sirkular dengan metode FCFS (*fisrt-cum-first-serve*) (Mayanka Katyal & Atul Mishra, 2013). Cara kerja algoritme ini yang sederhana, akan sesuai jika digunakan pada perangkat yang *resource constrain*. *Round robin* dapat menjamin penyaluran *traffic* yang sangat cepat pada sejumlah besar *request per second*, karena tidak perlu menyimpan *state* dari setiap *backend server* (XU Zongyu & WANG Xingxuan, 2015).

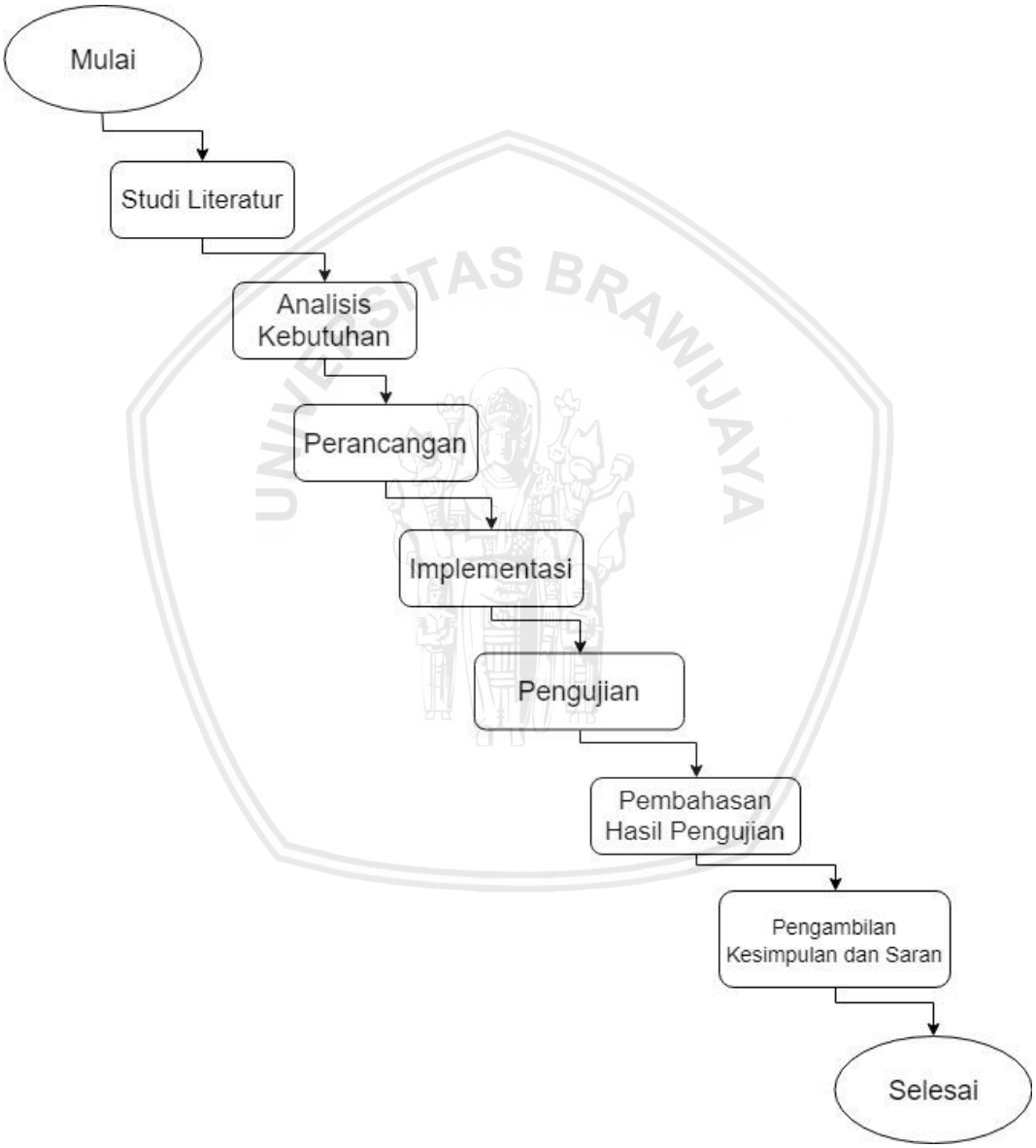
2.2.8 *Failover*

Setiap perangkat fisik yang bekerja pada sebuah *cluster* disebut *node* (Handoko dan Isa, 2018). *Node-node* yang bekerja dalam sebuah grup, bersama-sama membentuk sebuah *cluster*. Semua *node* di dalam *cluster* saling berkomunikasi secara konstan. Jika sebuah *node* menjadi *unavailable*, *node* yang lain secara otomatis menjalankan tugasnya dan memberikan layanan yang sama seperti *node* yang *unavailable*. Proses ketika sebuah *node* gagal dan *node* yang lain segera menggantikan peran *node* tersebut secara aktif, disebut *failover* (Handoko & Isa, 2018).

Teknik *failover* yang digunakan pada penelitian ini adalah VRRP (*Virtual Redundancy Router Protocol*). VRRP menetapkan *node* fisik terhubung ke virtual *router* secara dinamis (Ed., 2004). *Node* yang terhubung dengan alamat IP virtual VRRP dan mengirimkan paket melalui alamat tersebut disebut *node master*. VRRP menyediakan pilihan *default gateway* otomatis pada IP *subnetwork*. Pada sisi perangkat lunak, Keepalived telah menggunakan VRRP agar suatu perangkat mempunyai mekanisme *failover*. Keepalived banyak digunakan berdasarkan dokumentasi yang lengkap dan implementasi yang mudah (Handoko dan Isa, 2018).

BAB 3 METODOLOGI

Pada bab ini dijelaskan metode, teknik, atau langkah-langkah yang digunakan dalam penelitian implementasi *load balancing* sebagai salah satu solusi skalabilitas dalam lingkungan IoT. Tahapan-tahapan penelitian yang diusulkan yaitu studi literatur, analisis kebutuhan, perancangan sistem, implementasi, pengujian dan pembahasan pengujian, serta pengambilan kesimpulan dan saran. Gambar 3.1 menunjukkan tahapan-tahapan penelitian.



Gambar 3.1 Diagram Alur Metodologi Penelitian

3.1 Studi Literatur

Tahapan untuk pencarian dasar teori, dan berbagai literatur merupakan bagian dari metode pertama dalam penelitian ini. Studi literatur dibutuhkan untuk menunjang penulisan serta pengerjaan penelitian. Secara detail studi literatur terdiri atas referensi yang didapat pada jurnal ilmiah, buku, *website* resmi ataupun penelitian-penelitian yang terkait. Studi literatur pada penelitian ini antara lain tentang *internet of things*, MQTT, CoAP, arsitektur *publish/subscribe*, *load balancing*, *failover* dan Redis kluster.

3.2 Analisis Kebutuhan

Analisis dilakukan untuk mendapatkan kebutuhan-kebutuhan yang diperlukan dan mendukung seluruh tahapan dalam penelitian. Analisis kebutuhan bisa diperoleh dari proses pencarian studi literature. Kebutuhan pada penelitian ini meliputi kebutuhan fungsional dan non-fungsional yang digunakan untuk membangun sistem. Kebutuhan fungsional sistem, merupakan kebutuhan yang dibutuhkan agar sistem dapat berjalan. Sedangkan kebutuhan non-fungsional berfokus pada perilaku dan batasan fungsi dari sistem. Kebutuhan non-fungsional pada penelitian ini adalah skalabilitas.

3.3 Perancangan

Tahap perancangan sistem merupakan tahapan yang dilaksanakan setelah kebutuhan pada tahap analisis kebutuhan terpenuhi. Tahapan ini dimaksudkan agar mempermudah implementasi selanjutnya. Manfaat dari tahapan perancangan sistem, implementasi bisa berjalan dengan sistematis dan terstruktur.

Penelitian ini mengembangkan mekanisme *load balancing* untuk mendistribusikan beban ke dua *middleware*. Mekanisme *failover* juga dikembangkan agar dapat mengatasi kegagalan pada layanan *load balancing*. Agar hal tersebut terwujud diperlukan perancangan terhadap penelitian yang akan dibangun. Perancangan tersebut yaitu:

- Perancangan lingkungan sistem
- Perancangan alur komunikasi sistem.
- Perancangan pengalamatan dan topologi jaringan.
- Perancangan *load balancing*.
- Perancangan perilkau Keepalived pada *load balancer*.
- Perancangan data suhu dan kelembapan *node sensor* CoAP dan *node sensor* MQTT.
- Perancangan pengujian fungsional.
- Perancangan pengujian non-fungsional.

3.4 Implementasi

Di tahap ini akan dilakukan implementasi berdasarkan perancangan yang telah dilakukan. Langkah awal dalam proses implementasi pada penelitian ini adalah implementasi topologi dan pengalamatan jaringan pada sistem. Kemudian implementasi mekanisme *load balancing* dan *failover*. *Load balancer* tersebut kemudian mendistribusikan beban pada IoT *middleware* berdasarkan arsitektur yang sudah dibuat sebelumnya. Lalu implementasi *sensor* sebagai *publisher* dengan mengatur *payload*-nya.

3.5 Pengujian

Pengujian di lakukan untuk mengetahui pengaruh *load balancer* terhadap kinerja sistem. Tahap pengujian bertujuan untuk mengetahui kebutuhan fungsional yang telah tercapai oleh sistem yang dibangun. Jika terdapat kebutuhan fungsional yang belum tercapai maka dilakukan evaluasi terhadap perancangan dan implementasi sistem. Selain menguji kebutuhan fungsional yang telah dicapai sistem, pengujian terhadap aspek non-fungsional juga dilakukan, dalam hal ini adalah skalabilitas. Pengujian skalabilitas sistem bertujuan untuk mengetahui kapasitas sistem dalam menangani berbagai proses yang diterima ketika terdapat perubahan jumlah beban yang lebih besar dari sebelumnya

3.6 Pembahasan Hasil Pengujian

Hasil dari tahap pengujian kemudian dijelaskan dan diperbandingkan dengan hasil penelitian sebelumnya. Hal tersebut untuk mengetahui perbedaan kinerja sistem saat ini dengan kinerja sistem sebelumnya. Hasil pengujian kemudian digunakan untuk menunjang kesimpulan dan menjawab rumusan masalah. Hasil pengujian dapat berbentuk positif (mendukung hipotesis) atau berbentuk negatif (tidak mendukung hipotesis).

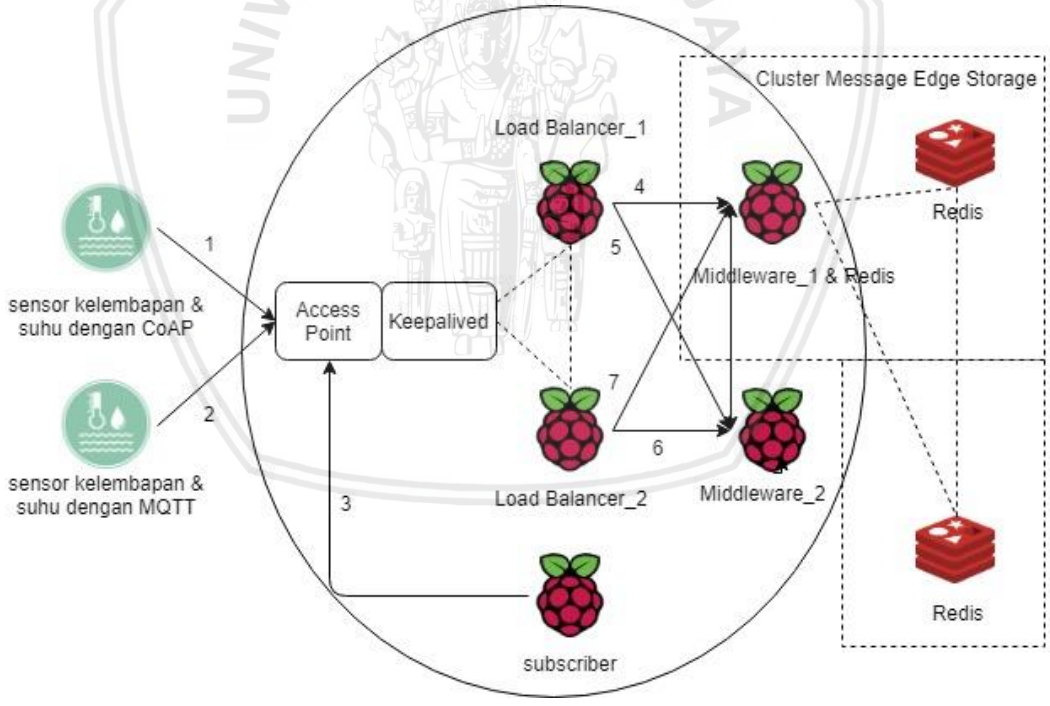
3.7 Pengambilan Kesimpulan dan Saran

Setelah semua tahap penelitian mulai dari studi literatur sampai dengan pengujian sistem selesai di lakukan, selanjutnya dapat di lakukan penarikan kesimpulan berdasarkan penelitian yang telah di lakukan. Kesimpulan diambil dari hasil pengujian dan analisis. Pengambilan saran menjadi tahap terakhir yang dilakukan untuk mengevaluasi adanya kesalahan penulisan dan kekurangan penelitian. Evaluasi dilakukan agar terjadi perbaikan dan penyempurnaan pada penelitian selanjutnya.

BAB 4 PERANCANGAN

4.1 Deskripsi Umum Sistem

Sistem yang dibangun terdiri atas dua perangkat *sensor*, yang mengirimkan data ke IoT *middleware* menggunakan protokol MQTT dan CoAP. Diantaranya dipasang *load balancer* yang akan mendistibusikan *traffic* ke dua IoT *middleware* yang bertindak sebagai *message broker*. Sistem akan menggunakan dua *load balancer. Load balancer* pertama bertindak sebagai *master node*, sedangkan *load balancer* kedua bertindak sebagai *back up node*. Ketentuan yang demikian, untuk mengantisipasi adanya *node failure* pada *node load* balancer. Pada setiap perangkat *load balancer* akan diimplementasikan perangkat lunak Keepalived. Implementasi perangkat lunak tersebut untuk menjamin selalu tersedianya layanan *load balancing* pada salah satu *node load* balancer, apabila terjadi kegagalan. Setelah itu, data yang diterima oleh IoT *middleware* akan didistribusikan secara acak ke dalam beberapa *cluster edge storage* menggunakan Redis, dan disimpan dalam bentuk topic/*key*. Selanjutnya *subscriber* akan menerima data yang telah dikirimkan oleh *node sensor* ke *cluster* oleh *message broker*. Gambar 4.1 menjelaskan deskripsi umum sistem.



Gambar 4.1 Deskripsi Umum Sistem

4.2 Analisis Kebutuhan

Pada bagian ini akan dijelaskan kebutuhan-kebutuhan pada sistem yang akan dibagun. Kebutuhan tersebut merupakan kebutuhan fungsional dan non-fungsional.

4.2.1 Kebutuhan Fungsional

Kebutuhan fungsional merupakan kebutuhan yang harus dipenuhi, agar sistem dapat berjalan dengan semestinya. Tabel 4.1 menjelaskan kebutuhan-kebutuhan fungsional yang harus dipenuhi oleh sistem.

Tabel 4.1 Kebutuhan Fungsional Sistem

No	Kebutuhan Fungsional
1	Perangkat lunak Keepalived dapat menentukan *node master* dan *node backup* pada *load balancer*
2	*Load balancer* mampu meneruskan *traffic* MQTT dari *sensor* ke IoT *middleware*
3	*Load balancer* mampu meneruskan *traffic* CoAP dari *sensor* ke IoT *middleware*
4	*Load balancer* mampu mendistribusikan *traffic* CoAP dan MQTT dari *sensor* ke IoT *middleware*
5	IoT *Middleware* dapat menerima data dari protokol MQTT
6	IoT *Middleware* dapat menerima data dari protokol CoAP
7	IoT *Middleware* dapat menerima data dari protokol CoAP dan MQTT
8	*Cluster* dapat mendistribusikan data dari IoT *middleware*
9	IoT *Middleware* dapat mengirimkan data dengan protokol CoAP dan MQTT ke *subscriber* menggunakan protokol MQTT.
10	Mengetahui semua *key* Redis dapat terhubung satu sama lain.

4.2.2 Kebutuhan Non-Fungsional

Kebutuhan non-fungsional adalah kebutuhan yang merupakan persyaratan tertentu agar dapat digunakan untuk menilai sistem. Hal yang menjadi fokus pada kebutuhan non-fungsional adalah batasan fungsi dan perilaku sistem. Skalabilitas adalah parameter dalam kebutuhan non-fungsional pada penelitian ini. Skalabilitas merujuk kepada kemampuan sistem dalam menangani berbagai proses yang diterima ketika terdapat perubahan sistem dengan jumlah yang besar dari kondisi sebelumnya. Ada empat parameter pengujian skalabilitas pada penelitian ini. Parameter tersebut adalah *time publish*, *concurrent publish*, *time subscribe*, dan *concurrent subscribe*. *Time publish* merupakan waktu yang dibutuhkan oleh IoT *middleware* untuk melakukan proses *publish* data dari *node sensor* ke *cluster message broker* . Sedangkan *time susbscibe* merupakan waktu yang dibutuhkan oleh IoT *middleware* untuk mencari dan mengirimkan data dari *cluster message broker* ke *client subscriber*. Adapun *concurrent* merupakan kemampuan IoT *middleware* dalam mengirimkan pesan per detik, baik dari sisi *publish* maupun *subscribe*. *Concurrent* didapat dari jumlah *client publisher*/*subscriber* dibagi jumlah *time publish* atau *time subscribe*. Sistem disebut lebih *scalable* apabila hasil pengujian menunjukkan *time publish* dan *time*

subscribe lebih rendah dari penelitian sebelumnya. Tabel 4.2 menjelaskan kebutuhan non-fungsional yang dibutuhkan pada penelitian ini.

Tabel 4.2 Kebutuhan non-fungsional sistem

No	Kebutuhan Non-Fungsional	Deskripsi
1	Skalabilitas	1. Dapat menangani proses *publish* dari sejumlah koneksi *publisher* yaitu 100, 500, 1000, 1500 koneksi 2. Dapat menangani proses *subscribe* dari koneksi *subscriber* dengan jumlah koneksi 100, 500, 1000, dan 1500

4.3 Lingkungan Penelitian

Lingkungan penelitian merupakan lingkungan IoT yang terdiri atas *publisher*, perangkat lunak Keepalived, *load balancer*, IoT *middleware*, *subscriber*, lingkungan perangkat lunak dan lingkungan perangkat keras. *Node sensor* adalah *client* yang bertindak sebagai *publisher*. Satu *sensor* akan mengirimkan data dengan protokol MQTT, dan *sensor* kedua akan mengirimkan data dengan protokol CoAP. Format data yang dikirimkan *publisher* berbentuk format data JSON. Setiap *publisher* akan mengirimkan data dengan topik yang ditentukan. *Publisher* dengan protokol MQTT akan mengirimkan data ke topik *home/kitchen*. *Publisher* dengan protokol CoAP, akan mengirimkan data ke topik *home/garage*. Sedangkan *subscriber* pada penelitian ini akan menggunakan MQTT sebagai protokol pengiriman data antara *middlware* dan *subscriber*. Perangkat lunak Keepalived yang akan digunakan adalah Keepalived. Dan *Load balancer* menggunakan NGINX sebagai perangkat lunak *load balancing*. *Message broker* pada IoT *middleware* menggunakan perangkat lunak Redis yang diatur dalam mode *cluster*. *Cluster* Redis sendiri akan berada pada mesin Raspberry Pi yang berbeda. Sebagai cara integrasi antara perangkat IoT *middleware* dan *cluster* yang terpisah, digunakan *ioredis*. Dengan adanya integrasi tersebut, maka *message broker* akan membentuk *cluster message broker*.

4.3.1 Lingkungan Perangkat Lunak

Lingkungan perangkat lunak yang akan digunakan pada penelitan ini adalah sebagai berikut:

Tabel 4.3 Lingkungan Perangkat Lunak

Perangkat	Keterangan
Redis	Perangkat lunak untuk menyimpankan data ke *memory*

Tabel 4.3 Lingkungan Perangkat Lunak (lanjutan)

Node.js	Sebuah *framework* yang digunakan untuk menciptakan *environment* pada IoT *middleware* menggunakan program *javascript*
Raspbian Stretch	Sistem operasi yang digunakan pada *Raspberry* Pi
NGINX	Perangkat lunak yang digunakan sebagai *load balancer*
Keepalived	Perangkat lunak yang digunakan untuk menjalankan fungsi *failover*

4.3.2 Lingkungan Perangkat Keras

Lingkungan perangkat keras yang akan digunakan pada penelitian ini adalah sebagai berikut:

Tabel 4.4 Lingkungan Perangkat Keras

Perangkat	Keterangan
Raspberry Pi	Media untuk implementasi IoT *middleware*, Redis, *load balancer* dan *access point*
USB Adapter EW-7722UTn EDIMAX	Perangkat pendukung *Raspberry* Pi agar berfungsi sebagai *access point*
MicroSD card 8GB	Perangkat sebagai media penyimpanan perangkat lunak pada *Raspberry* Pi
DHT11/22	DHT11/22 merupakan modul *sensor* untuk membaca suhu dan kelembapan

4.4 Perancangan Sistem

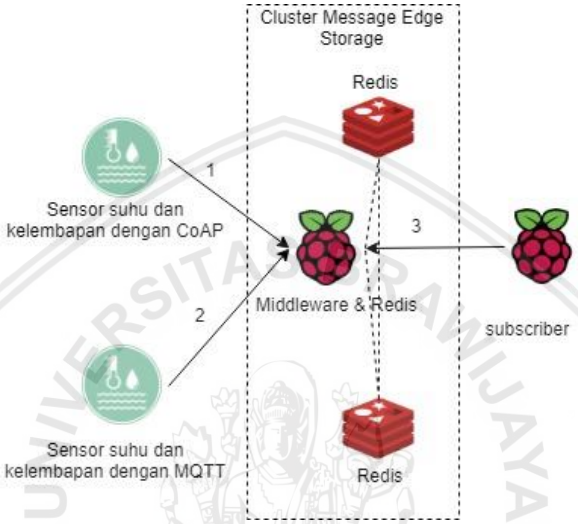
Perancangan sistem akan menjelaskan rancangan sistem yang akan dibuat. Tujuannya agar proses implementasi lebih sistematis dan terarah.

4.4.1 Perancangan Alur Komunikasi Sistem

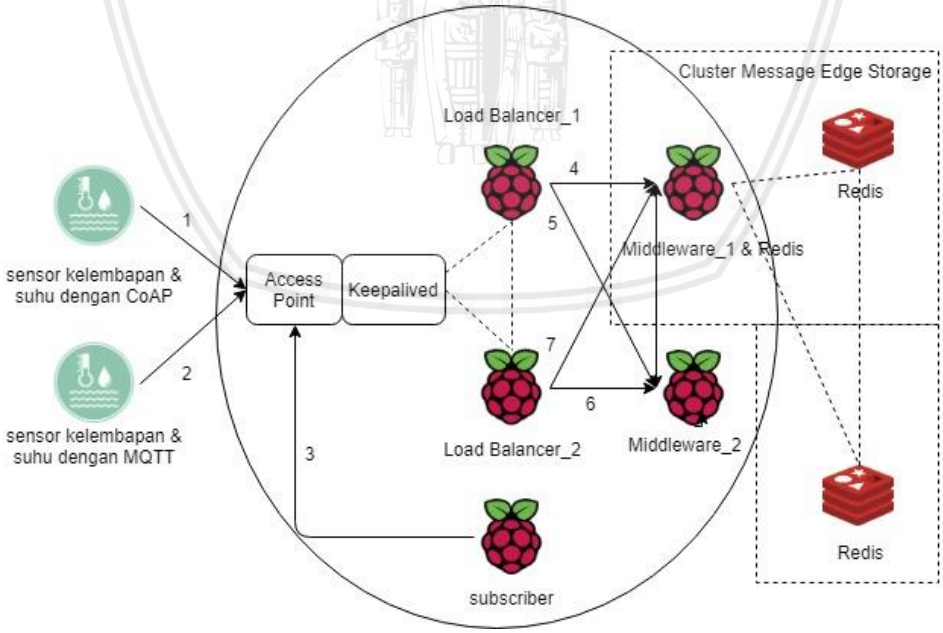
Pada penelitian ini sistem yang dibangun terdapat perangkat lunak untuk melakukan *failover*, *load balancer*, IoT *middleware*, *publisher* dan *subscriber*. Ke empat komponen tersebut saling berinteraksi membentuk pola komunikasi *publish/subscribe*. Penelitian sebelumnya, menunjukkan lingkungan alur komunikasi sistem menggunakan satu IoT *middleware*. Gambar 4.2 adalah topologi komunikasi sistem pada penelitian sebelumnya. Sedangkan gambar 4.3 adalah topologi komunikasi sistem yang akan digunakan pada penelitian ini.

Hal yang menjadi fokus pada penelitian ini adalah penambahan *load balancer* yang dapat melakukan *failover* untuk mendistribusikan *traffic* ke dua IoT *middleware*. Gambar 4.4 menjelaskan pola interaksi sub sistem antara *sensor* dengan *load balancer*. Gambar 4.5 menjelaskan pola interaksi antara *subscriber* dengan *load balancer*. Gambar 4.6 menunjukkan pola interaksi sub sistem antara

perangkat lunak Keepalived dengan *load balancer*. Gambar 4.7 menunjukkan pola interaksi sub sistem antara *load balancer* dengan IoT *middleware*. Gambar 4.8 menjelaskan alur komunikasi *sensor* CoAP ke IoT *middleware* melalui *load balancer*. Gambar 4.9 menjelaskan menjelaskan alur komunikasi *sensor* MQTT ke IoT *middleware* melalui *load balancer*. Gambar 4.10 menjelaskan alur komunikasi *subscriber* ke IoT *middleware* melalui *load balancer*. Gambar 4.11 menjelaskan alur komunikasi perangkat lunak Keepalived dengan *load balancer*. Gambar 4.12 menjelaskan komunikasi IoT *middleware* dengan *cluster*. Dan gambar 4.13 menjelaskan komunikasi IoT *middleware* dengan *subscriber*.



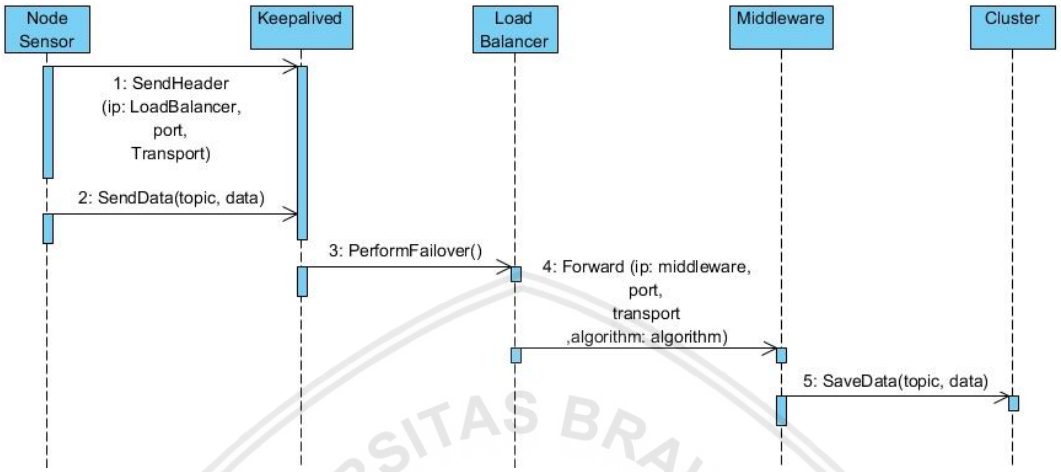
Gambar 4.2 Topologi Komunikasi Sistem Penelitian Sebelumnya



Gambar 4.3 Topologi Komunikasi Sistem Penelitian Sekarang

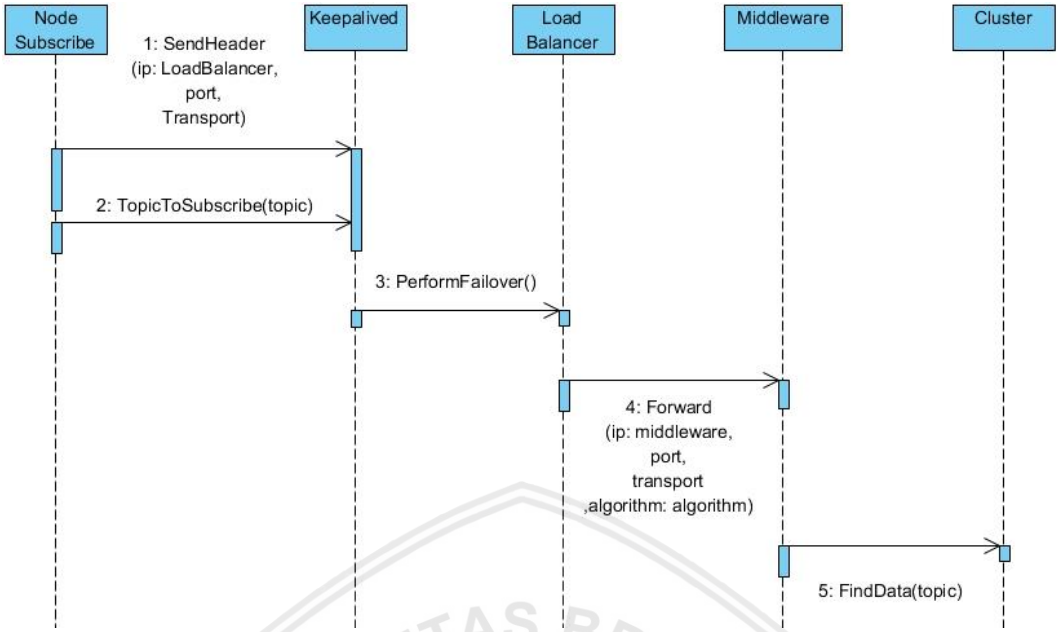
Setiap sub sistem memiliki alur komunikasinya masing-masing. Komunikasi sub sistem terbagi menjadi enam bagian. Alur komunikasi setiap bagiannya akan dijelaskan menggunakan diagram *use case*. Komunikasi tersebut, pertama adalah komunikasi *sensor* dengan *load balancer*. Kedua, komunikasi *subscriber* dengan *load balancer*. Ketiga, komunikasi perangkat lunak Keepalived dengan *load*

balancer. Keempat, komunikasi *load balancer* dengan IoT *middleware*. Alur komunikasi yang kelima adalah komunikasi antara IoT *middleware* dengan *cluster*. Alur komunikasi yang keenam adalah komunikasi antara IoT *middleware* dengan *subscriber*.



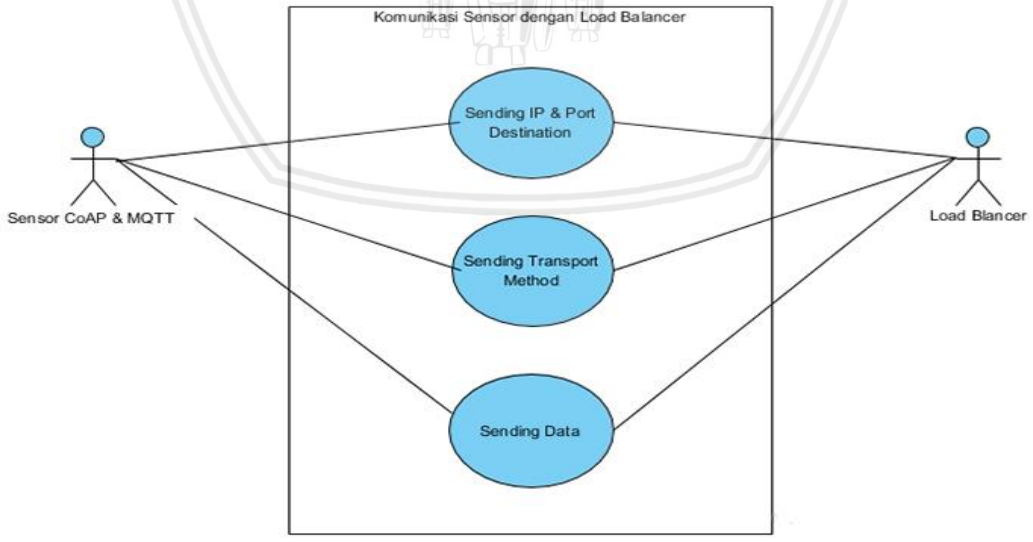
Gambar 4.4 Alur Komunikasi Sistem dari Sisi *Node Sensor*

Gambar 4.4 adalah alur komunikasi sistem secara umum dari sisi *node sensor*. Alur tersebut dimulai dari *node sensor* mengirimkan topik dan data ke IoT *middleware* hingga topik dan data tersebut disimpan ke *cluster*. Pertama *node sensor* mengirimkan *header* dan data ke IoT *middleware* melalui *load balancer*. Mekanisme *failover* dijalan apabila *node load balancer* mengalami kegagalan proses. *Load balancer* kemudian meneruskan *traffic* dari *node sensor* ke IoT *middleware*. Pesan berupa topik dan data dari *node sensor* kemudian disimpan oleh IoT *middleware* ke *cluster.*



Gambar 4.5 Alur Komunikasi Sistem dari Sisi *Node Subscribe*

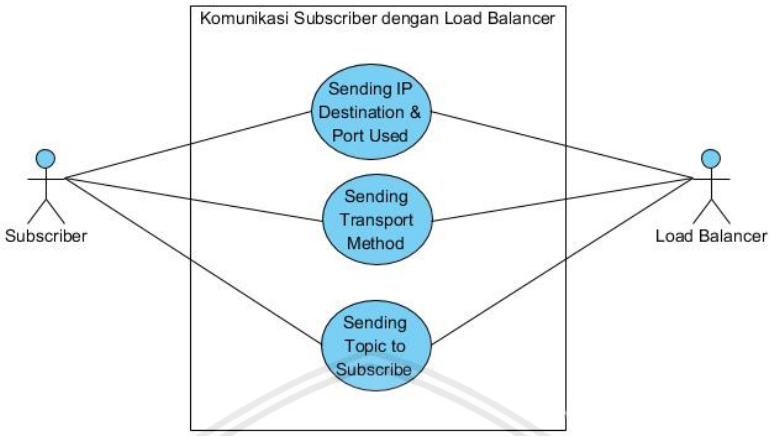
Gambar 4.5 adalah alur komunikasi sistem secara umum dari sisi *node subscribe*. Alur tersebut dimulai dari *node subscribe* mengirimkan *request* terkait topik yang diinginkan hingga IoT *middleware* menampilkan data dari topik yang di-*request*. Pertama *request* dikirimkan ke IoT *middleware* melalui *load balancer*. Kemudian mekanisme *failover* dijalan apabila *node load balancer* mengalami kegagalan proses. *Traffic* kemudian diteruskan oleh *load balancer* dari *node subscribe* ke IoT *middleware*. Kemudian dilakukan pencarian data oleh IoT *middleware* dari *cluster* sesuai topik yang di-*request.*



Gambar 4.6 Komunikasi *Sensor* dengan *Load Balancer*

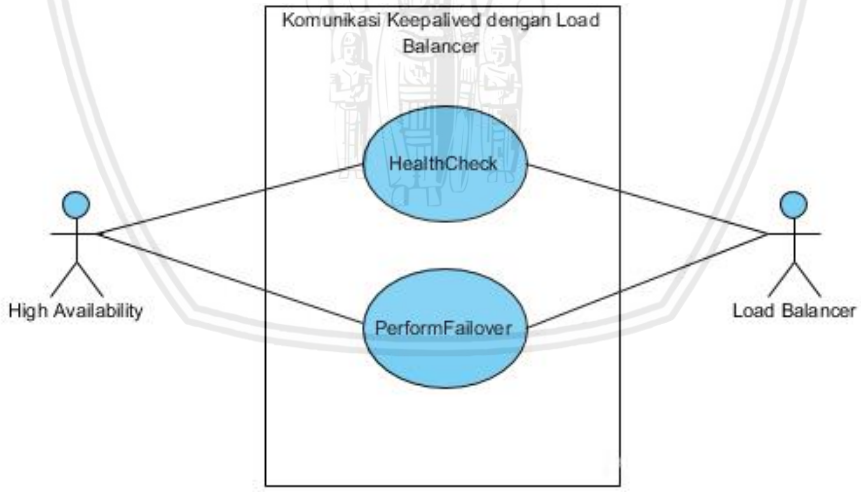
Pertama adalah komunikasi antara *sensor* dengan *load balancer*. Alur komunikasi tersebut dijelaskan pada gambar 4.4. Komunikasi yang ditunjukkan gambar 4.6 menggambarkan pengiriman data oleh *sensor* yang berupa *header* alamat IP tujuan dan *port* yang digunakan, serta metode *transport* yang dipakai. *Sensor* juga mengirimkan *payload* ke *load balancer* yang kemudian akan

diteruskan oleh *load balancer* ke IoT *middleware* yang dituju. *Payload* berisi data yang akan di-*publish* ke IoT *middleware*. Lebih jelasnya tentang *payload* akan diterangkan pada bab perancangan *payload*.



Gambar 4.7 Komunikasi *Subscriber* dengan *Load Balancer*

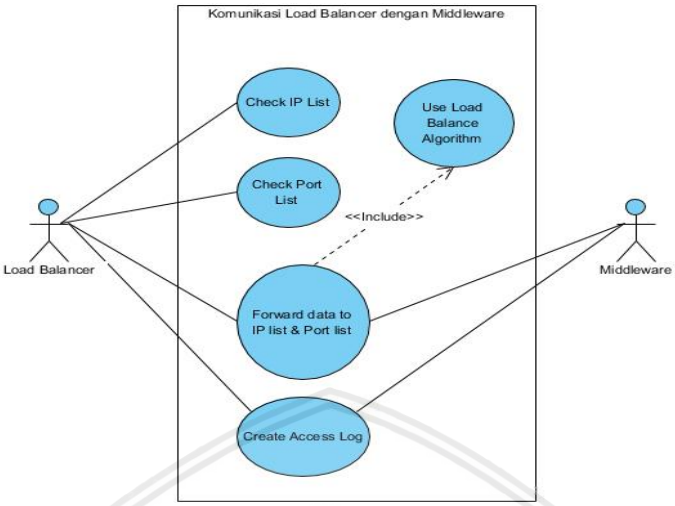
Kedua adalah komunikasi antara *subscriber* dengan *load balancer*. Alur komunikasi tersebut dijelaskan pada gambar 4.5. Komunikasi yang ditunjukkan gambar 4.7 menggambarkan pengiriman data oleh *subscriber* yang berupa *header* alamat IP yang dituju dan *port* yang digunakan, serta metode *transport* yang dipakai. *Subscriber* juga mengirimkan topik yang ingin di-*subscribe* ke *load balancer* yang kemudian akan diteruskan oleh *load balancer* ke IoT *middleware* yang dituju.



Gambar 4.8 Komunikasi *Software* Keepalived dengan *Load Balancer*

Ketiga adalah komunikasi perangkat lunak Keepalived dengan *load balancer*. Pola komunikasi tersebut dijelaskan pada gambar 4.6. Komunikasi pada gambar 4.8 menjelaskan proses perangkat lunak Keepalived dalam melakukan *failover*. Perangkat lunak Keepalived akan melakukan pemeriksaan kesehatan pada *load balancer* yang memiliki layanan *load balancing*. Satu *node load balancer* akan menjadi *node master* dan *node* lainnya akan menjadi *node backup*. Proses pemeriksaan kesehatan dilakukan pada kedua jenis *node* tersebut. Jika *node* yang menjadi *node master* mengalami kegagalan, perangkat lunak Keepalived akan memindahkan fungsi *node master* ke *node backup*. Tetapi, jika *node master* masih

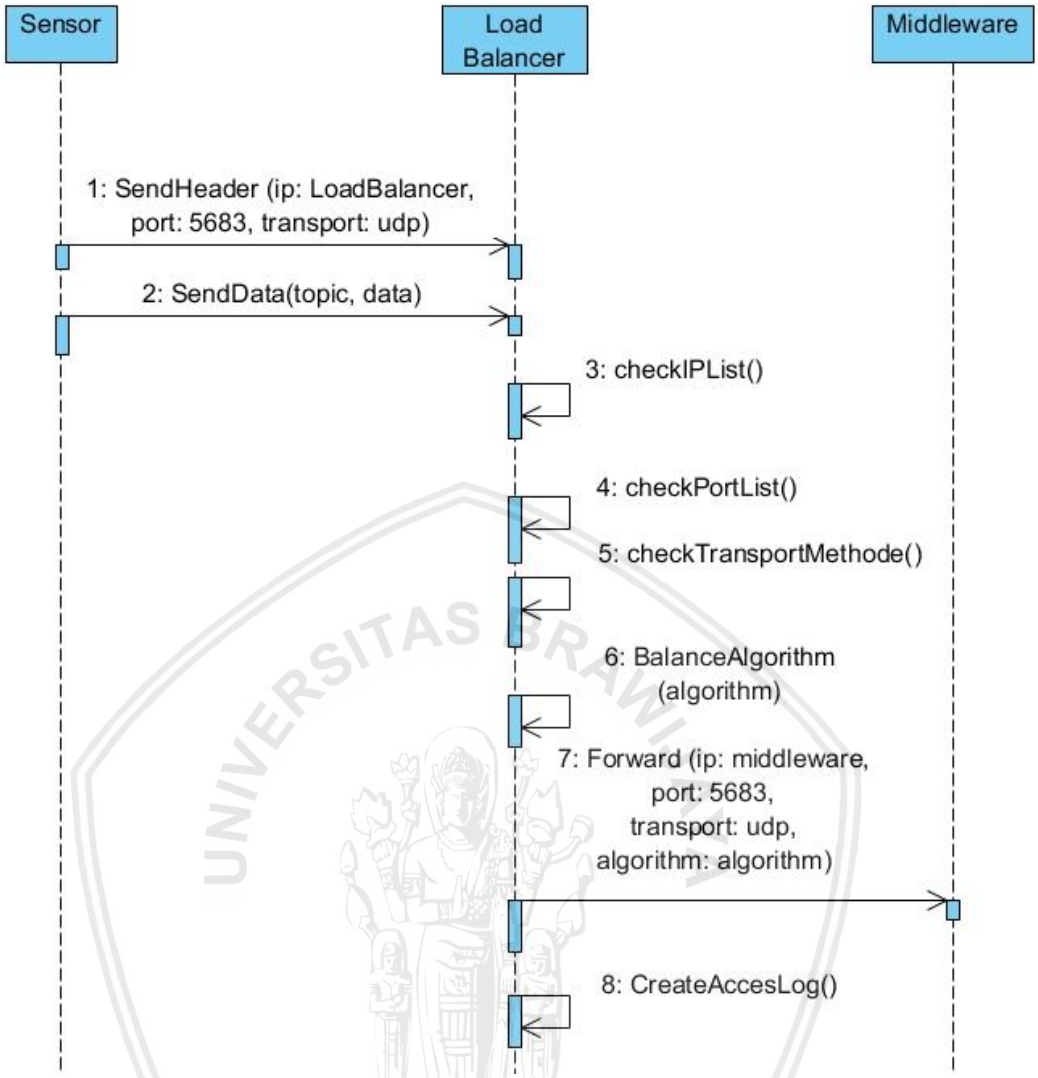
memiliki ketersediaan layanan, *node master* akan mengatasi semua proses yang terjadi pada *load balancer*.



Gambar 4.9 Komunikasi *Load Balancer* dengan IoT *Middleware*

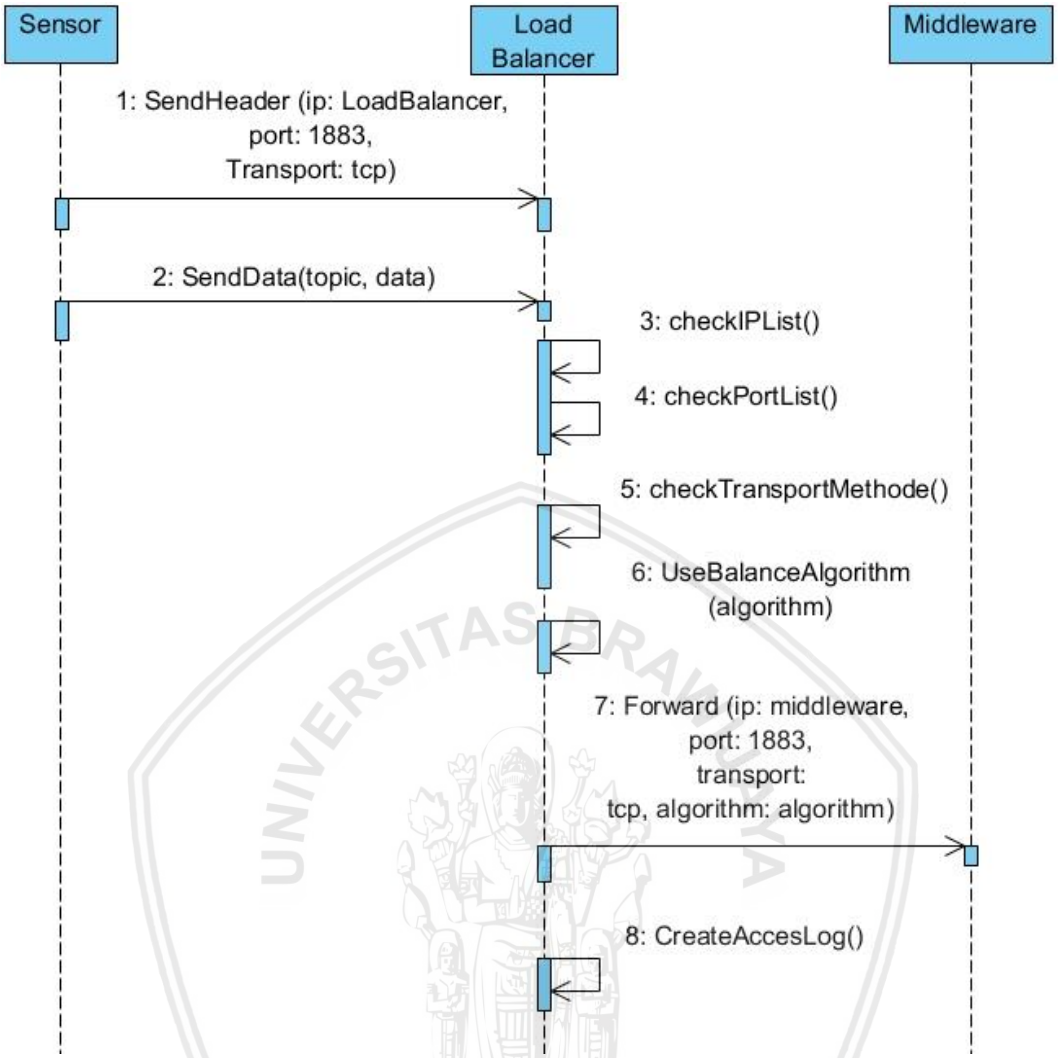
Keempat merupakan komunikasi sub sistem antara *load balancer* dengan IoT *middleware*. Gambar 4.9 menjelaskan pola komunikasi antara *load balancer* dengan IoT *middleware*. Setelah *load balancer* menerima data dari *sensor*, *load balancer* akan melakukan pemeriksaan alamat IP dan *port* yang ada di *file* konfigurasi. *Load balancer* juga melakukan pemeriksaan metode *transport* yang digunakan oleh *server* sesuai *file* konfigurasi pada *load balancer*. Kemudian, *load balancer* akan mengirimkan *traffic* ke daftar *server* (IoT *middleware*) sesuai dengan *port* dan metode *transport* yang digunakan. Lalu, *load balancer* akan membuat *file* catatan *log* akses terkait *traffic* yang diteruskan. Pengaturan daftar *server*, daftar *port* dan metode *transport* terdapat pada *file* konfigurasi di perangkat lunak *load balancer*.

Pada penelitian ini, berfokus pada implementasi *load balancer* yang dapat melakukan *failover*. Gambar 4.10 menunjukkan pola perilaku *load balancer* terhadap *sensor* yang mengirimkan data menggunakan protokol CoAP. Sedangkan gambar 4.11 menunjukkan pola perilaku *load balancer* terhadap *sensor* yang mengirimkan data menggunakan protokol. Gambar 4.12 menunjukkan pola perilaku perangkat lunak Keepalived dalam melakukan *failover*.



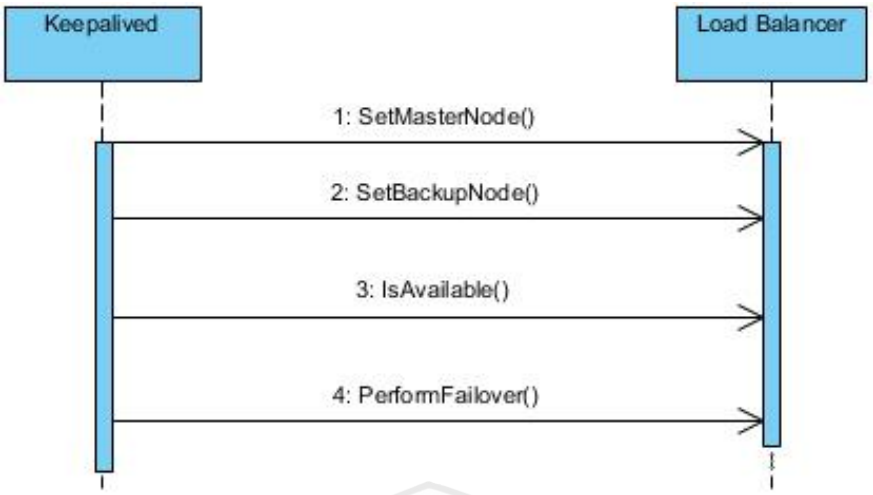
Gambar 4.10 Komunikasi *Sensor* CoAP ke *Middleware* melalui *Load Balancer*

Pada gambar 4.10 *sensor* mengirimkan data suhu dan kelembapan menggunakan protokol CoAP. *Sensor* akan mengirimkan *header* berupa alamat IP tujuan, *port* yang dijalankan dan metode *transport* yang digunakan. *Sensor* juga mengirimkan data *payload* ke *load balancer*. Pengiriman pesan dengan protokol CoAP menggunakan *port* 5683 sebagai *port* layanan, dan *transport* UDP sebagai metode *transport* pesannya. Kemudian *load balancer* akan melakukan pemeriksaan terhadap daftar *server* dan daftar *port* yang ada dalam *file* konfigurasi. *Load balancer* juga akan melakukan pemeriksaan metode *transport* yang digunakan oleh *sensor* untuk disesuaikan pada *file* konfigurasi. Setelah selesai melakukan pemeriksaan, *load balancer* akan menggunakan algoritme *load balance* yang telah ditentukan untuk meneruskan *traffic* ke *backend server* yang berupa IoT *middleware*. *Load balancer* juga membuat *file access log* untuk merekam *log traffic* yang masuk ke *load balancer*.



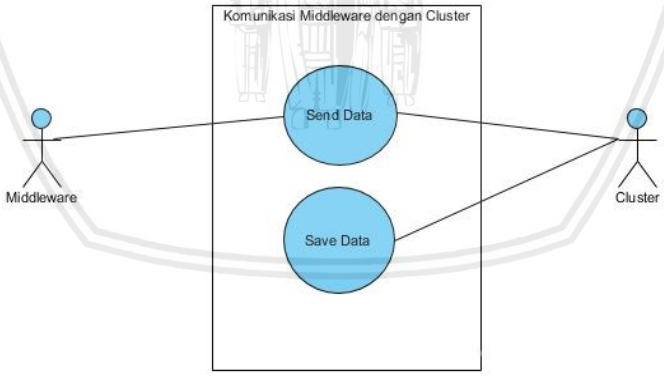
Gambar 4.11 Komunikasi *Sensor* MQTT ke *Middleware* melalui *Load Balancer*

Pada gambar 4.11 *sensor* mengirimkan data suhu dan kelembapan menggunakan protokol MQTT. *Sensor* akan mengirimkan *header* berupa alamat IP tujuan, *port* yang dijalankan dan metode *transport* yang digunakan. *Sensor* juga mengirimkan data *payload* ke *load balancer*. Pengiriman pesan dengan protokol MQTT menggunakan *port* 1883 sebagai *port* layanan, dan *transport* TCP sebagai metode *transport* pesannya. Kemudian *load balancer* akan melakukan pemeriksaan terhadap daftar *server* dan daftar *port* yang ada dalam *file* konfigurasi. *Load balancer* juga akan melakukan pemeriksaan metode *transport* yang digunakan oleh *sensor* untuk disesuaikan pada *file* konfigurasi. Setelah selesai melakukan pemeriksaan, *load balancer* akan menggunakan algoritme *load balance* yang telah ditentukan untuk meneruskan *traffic* ke *backend server* yang berupa IoT *middleware*. *Load balancer* juga membuat *file access log* untuk merekam *log traffic* yang masuk ke *load balancer*.



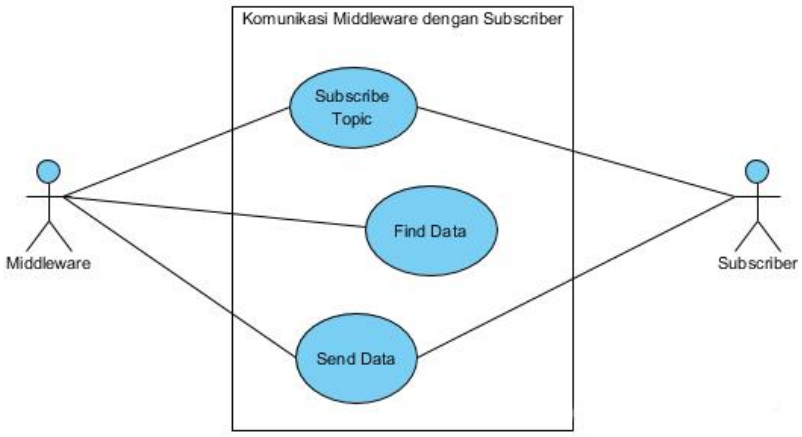
Gambar 4.12 Alur Komunikasi *Software* Keepalived dengan *Load Balancer*

Kelima adalah komunikasi sub sistem Keepalived dengan *load balancer*. Pada gambar 4.12 perangkat lunak Keepalived menentukan status perangkat *load balancer* menjadi *node master* atau *node backup*. Proses penentuan tersebut di dasarkan pada pengaturan yang diimplementasikan pada perangkat lunak Keepalived. Setelah itu perangkat lunak Keepalived akan melakukan pemeriksaan terhadap masing-masing *node* yang telah didaftarkan. Pemeriksaan tersebut dilakukan secara berkala. Jika *node master* masih memiliki ketersediaan layanan, semua proses *load balancing* akan diatasi oleh *node master*. Namun, apabila *node master* tidak memiliki ketersediaan layanan, perangkat lunak Keepalived akan melakukan proses *failover* ke *node backup* yang telah didaftarkan. Sehingga, seluruh proses *load balancing* akan berpindah dari *node master* ke *node backup*.



Gambar 4.13 Komunikasi IoT *Middleware* dengan *Cluster*

Komunikasi sub sistem keenam, adalah komunikasi antara IoT *middleware* dengan *cluster*. Gambar 4.13 menjelaskan pola komunikasi antara IoT *middleware* dengan *cluster*. Pada gambar tersebut data *payload* akan dikirimkan oleh IoT *middleware cluster*. Kemudian data tersebut akan disimpan oleh *cluster* ke *node* yang tersedia secara acak. Penggunaan perangkat lunak Redis, membantu mewujudkan penyimpanan data pada *node* yang tersedia secara acak.

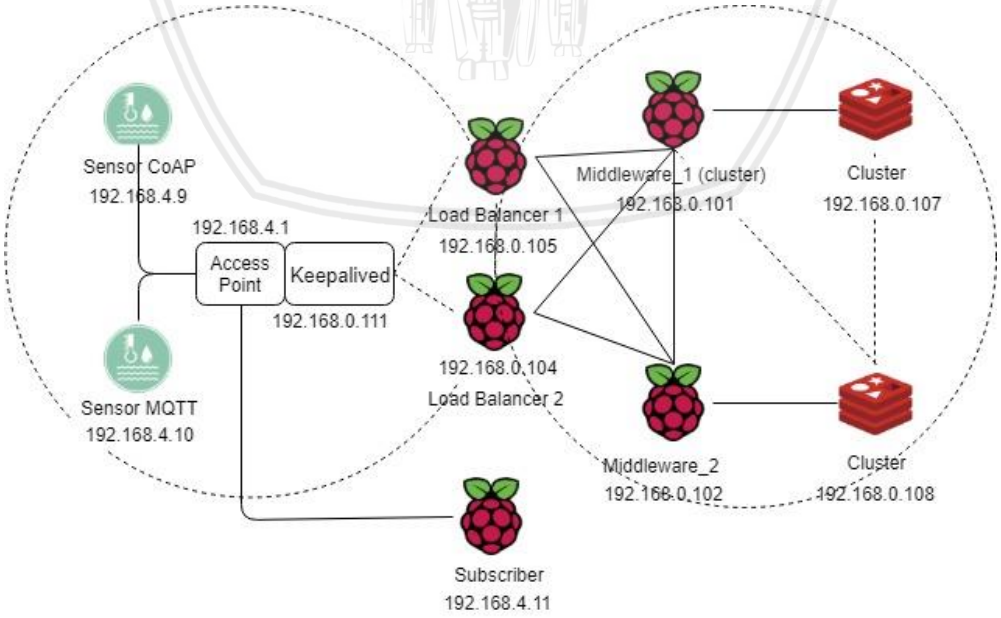


Gambar 4.14 Komunikasi IoT *Middleware* dengan *Subscriber*

Komunikasi sub sistem ketujuh, adalah komunikasi yang terjadi antara IoT *middleware* dengan *subscriber*. Gambar 4.14 menjelaskan pola komunikasi antara IoT *middleware* dengan *subscriber*. Pada gambar tersebut, *subscriber* mulai men-*subscribe* topik yang tersedia. Setelah itu *service unit* pada IoT *middleware* akan mencari data pada *cluster* berdasarkan topik yang di-*subscribe* oleh *subscriber*. Kemudian IoT *middleware* akan mengirimkan data tersebut ke *subscriber*.

4.4.2 Perancangan Pengalamatan dan Topologi Jaringan

Perancangan alamat dan topologi jaringan bertujuan agar sistem dapat berkomunikasi sesuai dengan yang diharapkan. Pada perancangan ini menggambarkan cara *sensor* dapat terhubung dan mengirimkan data ke IoT *middleware* melewati *load balancer*. Gambar 4.13 menjelaskan rancangan pengalamatan dan topologi jaringan pada penelitian ini.



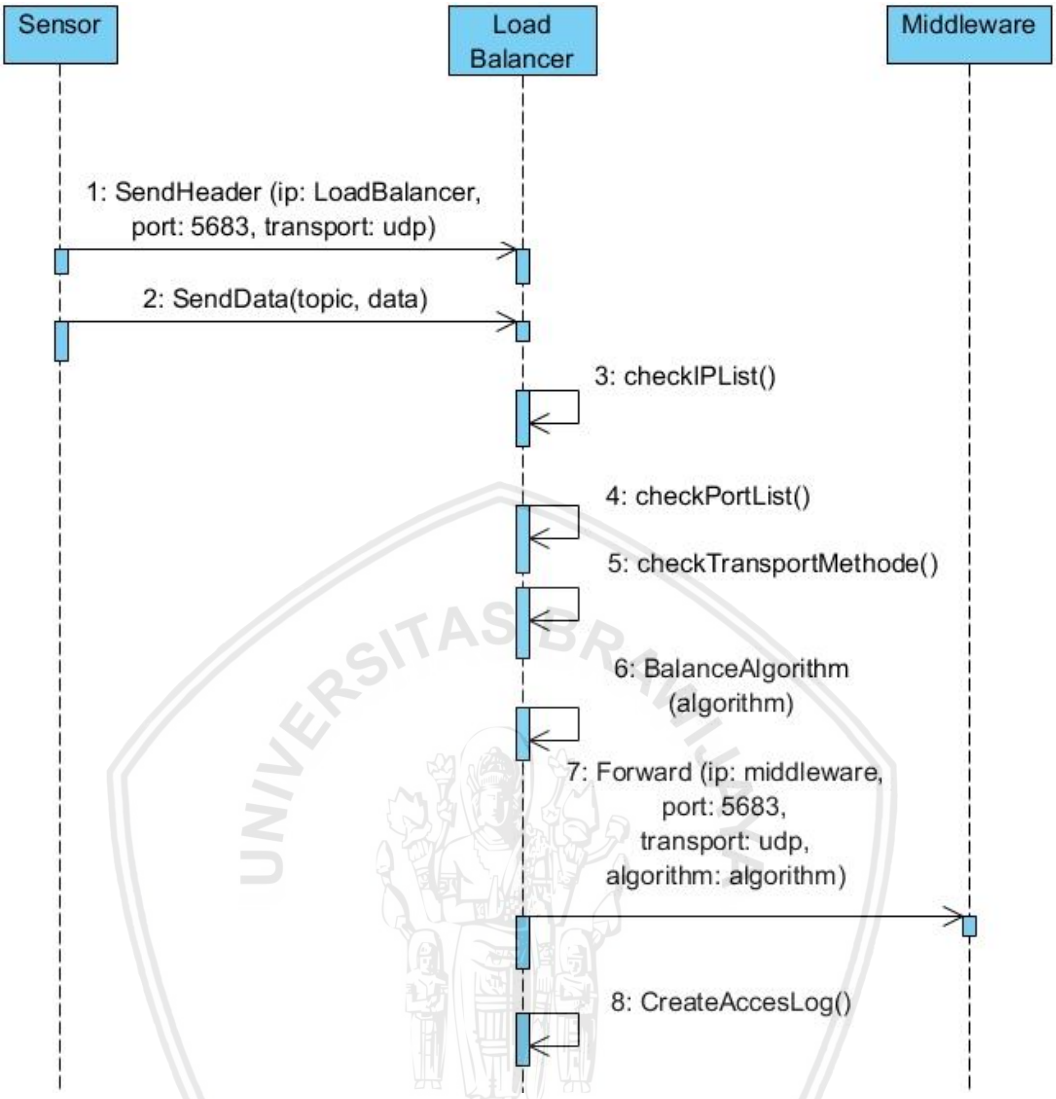
Gambar 4.15 Perancangan Pengalamatan dan Topologi Jaringan

Pada gambar 4.15 terdapat perangkat yang bertindak sebagai *load balancer* dengan alamat IP 192.168.0.104 dan 192.168.0.105 yang menerima data dari dua perangkat *sensor* suhu dan kelembapan. *Sensor* pertama dengan alamat

IP 192.168.4.9 mengirimkan data suhu dan kelembapan menggunakan protokol CoAP. Sedangkan *sensor* kedua dengan alamat IP 192.168.4.10 mengirimkan data suhu dan kelembapan menggunakan protokol MQTT. Alamat IP *virtual* perangkat lunak Keepalived diatur pada alamat 192.168.0.111. Sedangkan IoT *middleware* yang mempunyai *service unit* sebagai *broker*, masing-masing memiliki alamat IP, 192.168.0.101 dan 192.168.0.102. Tempat implementasi *cluster* Redis sebagai *edge storage* memiliki alamat IP 192.168.0.107, 198.168.0.101, dan 192.168.0.108. *Sensor* CoAP akan mengirimkan data dengan `topic:home/garage` menggunakan POST *request*. Sedangkan *sensor* MQTT mengirimkan data dengan `topic:home/kitchen`. Kedua *sensor* akan mengirimkan datanya ke alamat *virtual* perangkat lunak Keepalived. Untuk *sensor* CoAP mengirimkan data ke *uniform resource locator* (*uri*) `coap://192.168.0.111:5683/r/home/garage`. Sedangkan *sensor* MQTT mengirimkan data ke *uri* `mqtt://192.168.0.111:1883/home/kitchen`. *Subscriber* akan terhubung ke *access point* sebagaimana *publisher*. *Subscriber* akan memiliki alamat IP 192.168.4.11.

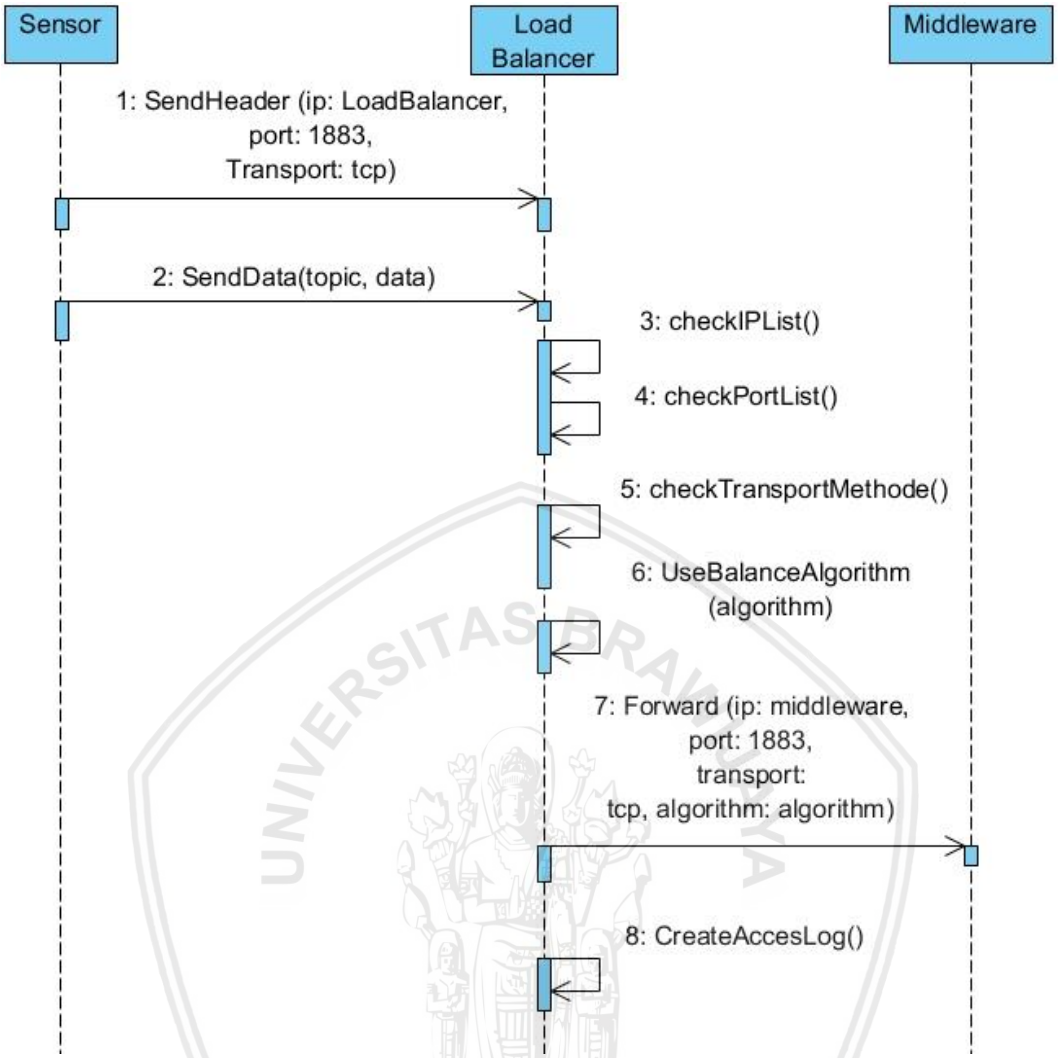
4.4.3 Perancangan *Load Balancing*

Load balancer memanfaatkan API perangkat lunak NGINX untuk melakukan *load balancing*. *Load balancer* nantinya akan meneruskan *traffic* dari *sensor* dan *subscriber* ke *server* tujuan sesuai dengan algoritme yang diterapkan. Untuk itu diperlukan alamat IP *server* tujuan, jenis protokol *transport* dan juga *port* yang digunakan. Setiap *server* mendengarkan *port* sesuai dengan layanan protokol yang digunakan. Protokol CoAP mendengarkan *service* pada *port* 5683, sedangkan MQTT mendengarkan *service* pada *port* 1883. Perancangan *load balancing* mengacu pada pola perilaku *load balancing* terhadap sistem yang dijelaskan pada gambar 4.16, gambar 4.17 dan gambar 4.18.



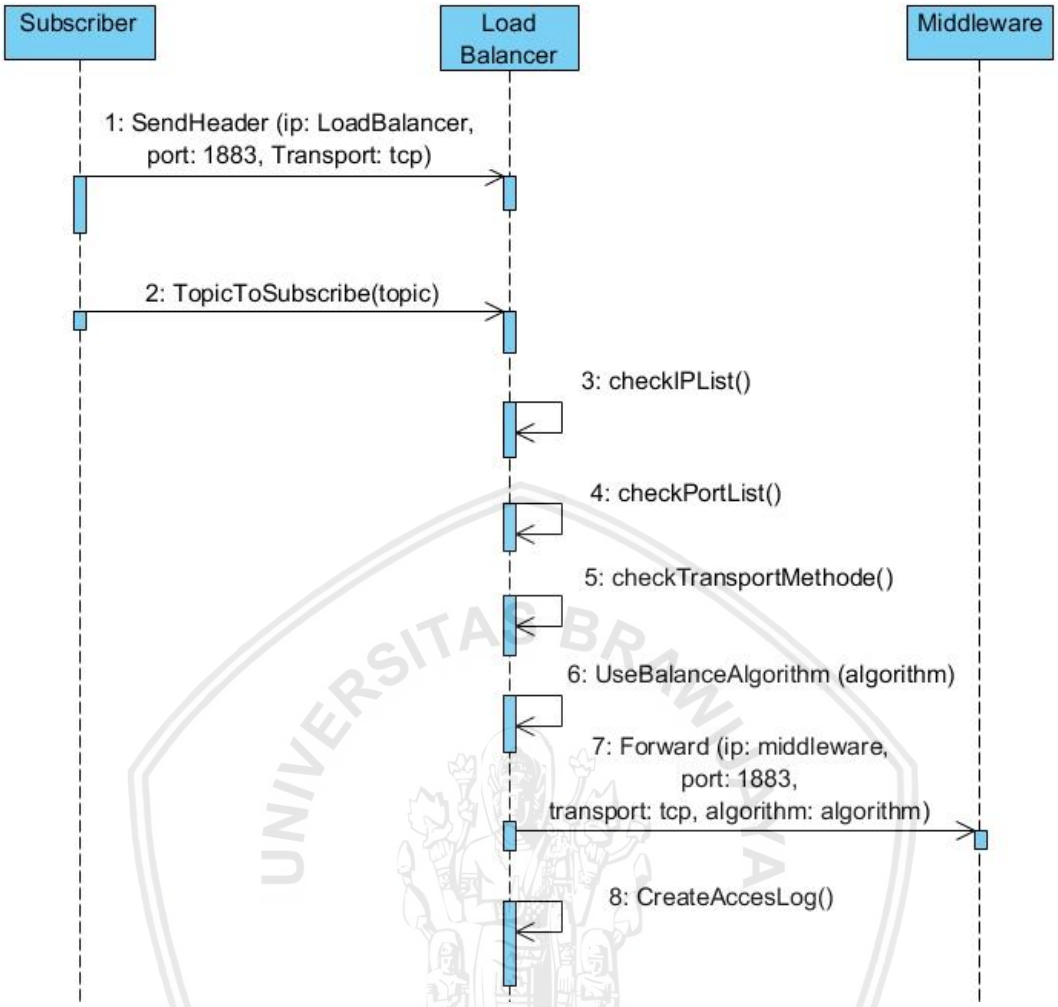
Gambar 4.16 Perilaku *Load Balancing* terhadap *Sensor* CoAP dalam Sistem

Pada gambar 4.16 ketika *sensor* CoAP mengirimkan data suhu dan kelembapan, *sensor* tersebut akan pertama-tama mengirimkan *header*-nya yang berisi *uri* yang telah ditranslasi oleh perangkat lunak Keepalived ke alamat IP *load balancer*. *Sensor* CoAP juga akan mengirimkan nomor *port* yang dipakai dan jenis metode *transport* aliran yang digunakan. Untuk *sensor* CoAP mengirimkan data ke *port* 5683 dan menggunakan jenis metode *transport* UDP. Sehingga pada *load balancer* perlu menambahkan deskripsi *port* dan *transport* yang didengar ke 5683 dan *transport* UDP, untuk bisa mendistribusikan *traffic* dari protokol CoAP. Selain itu *load balancer* perlu juga menambahkan deskripsi alamat IP *back end server*, sebagai tujuan kemana *traffic* tersebut dialihkan. Algoritme *round robin*, akan ditambahkan juga pada perancangan *load balancing*. Terakhir untuk merekam aktifitas *load balancing*, *load balancer* akan membuat *file log* yang berisi seluruh aktifitas *load balancing* protokol CoAP pada sistem.



Gambar 4.17 Perilaku *Load Balancing* terhadap *Sensor* MQTT dalam Sistem

Pada gambar 4.17 ketika *sensor* MQTT mengirimkan data suhu dan kelembapan, *sensor* tersebut akan pertama-tama mengirimkan *header*-nya yang berisi *uri* yang telah ditranslasi oleh perangkat lunak Keepalived ke alamat IP *load balancer*. *Sensor* MQTT juga akan mengirimkan nomor *port* yang digunakan dan jenis metode *transport* aliran yang dipakai. Untuk *sensor* MQTT mengirimkan data ke *port* 1883 dan menggunakan jenis metode *transport* TCP. Sehingga pada *load balancer* perlu menambahkan deskripsi *port* dan *transport* yang didengar ke *port* 1883 dan *transport* TCP, untuk bisa mendistribusikan *traffic* dari protokol MQTT. Selain itu *load balancer* perlu juga menambahkan deskripsi alamat IP *back end server*, sebagai tujuan kemana *traffic* tersebut dialihkan. Algoritme *round robin*, akan ditambahkan juga pada perancangan *load balancing*. Terakhir untuk merekam aktifitas *load balancing*, *load balancer* akan membuat *file log* yang berisi seluruh aktifitas *load balancing* protokol MQTT pada sistem.

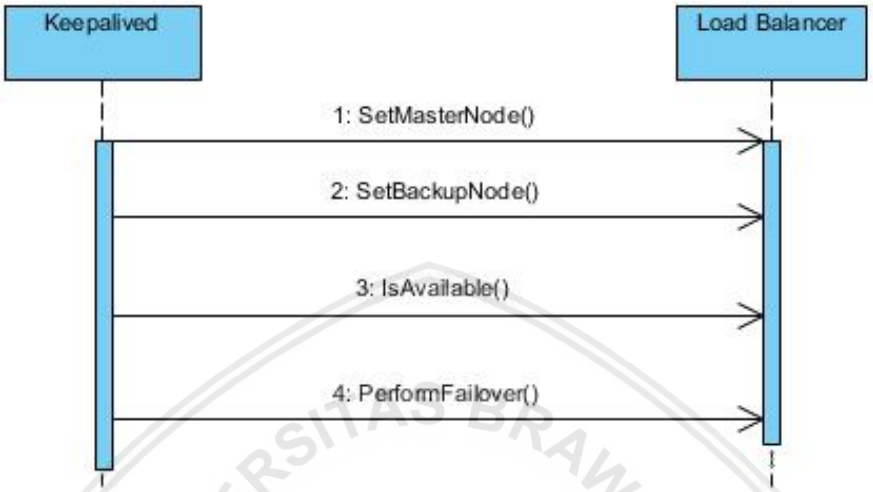


Gambar 4.18 Perilaku *Load Balancing* terhadap *Subscriber* dalam Sistem

Pada gambar 4.18 ketika *subscriber* mengirimkan topik yang ingin di-*subscribe* melalui *load balancer*, *subscriber* pertama-tama akan mengirimkan *header*-nya yang berisi *uri* yang telah ditranslasi oleh perangkat lunak Keepalived ke alamat IP *load balancer*. *Subscriber* juga akan mengirimkan nomor *port* yang digunakan dan jenis metode *transport* aliran yang dipakai. *Subscriber* mengirimkan topik melalui *port* 1883 dan menggunakan jenis metode *transport* TCP. Sehingga pada *load balancer* perlu menambahkan deskripsi *port* dan *transport* yang didengar ke *port* 1883 dan *transport* TCP, untuk bisa mendistribusikan *traffic* dari protokol MQTT *subscriber*. Selain itu *load balancer* perlu juga menambahkan deskripsi alamat IP *back end server*, sebagai tujuan kemana *traffic* tersebut dialihkan. Algoritme *round robin*, akan ditambahkan juga pada perancangan *load balancing*. Terakhir untuk merekam aktifitas *load balancing*, *load balancer* akan membuat *file log* yang berisi seluruh aktifitas *load balancing* protokol MQTT *subscriber* pada siste

4.4.4 Perancangan Perilaku Keepalived pada *Load Balancer*

Perangkat lunak Keepalived menggunakan VRRP untuk melakukan *failover*, jika terjadi kegagalan *node*. Perancangan perangkat lunak Keepalived mengacu pada pola perilaku yang ditunjukkan gambar 4.18.

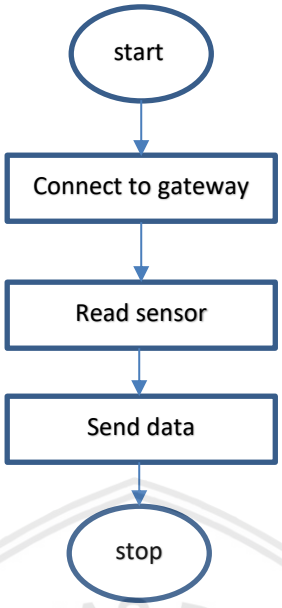


Gambar 4.19 Perilaku *Software* Keepalived terhadap *Load Balancer*

Keepalived akan menentukan perangkat *load balancer* yang bertindak sebagai *node master* dan *node backup*. Selanjutnya Keepalived akan melakukan pemeriksaan ketersediaan layanan pada perangkat *load balancer*. Pemeriksaan itu akan dilaksanakan secara berkala. Apabila *node master* mengalami kegagalan, Keepalived akan menjadikan *node back up* menjadi *node* yang melakukan proses *load balanc*ing, sampai *node master* tersedia kembali.

4.4.4 Perancangan *Payload Data Sensor*

Data *sensor* akan disusun berdasarkan *payload* yang akan dikirimkan dalam bentuk JSON. Perancangan struktur *payload* berisi rangkuman informasi guna menyamakan format data yang akan dikirimkan. Sedangkan, data sendiri akan dikirimkan menggunakan protokol CoAP dan MQTT. Perancangan *payload* dimulai dengan menghubungkan *sensor* dengan dengan *access point*. Setelah terhubung, program diminta untuk menentukan alamat *gateway* yang dituju. Program kemudian akan membaca data suhu dan kelembapan dari *sensor* dht11. Program akan mengirimkan data suhu dan kelembapan ke alamat IP tujuan, setelah berhasil membaca data dari *sensor*. Perancangan kode program *payload* mengikuti alur yang ditunjukkan gambar 4.19.



Gambar 4.20 Diagram Alir Kode Program *Payload*

Struktur data *payload* akan berbentuk seperti pada tabel 4.5.

Tabel 4.5 Struktur *Payload Data Sensor*

```
Algoritme 1: Struktur Payload
1   Var payload = {
2        protokol: String      //namaProtokol
3        timestamp: String     //waktuKirim
4        topic: String         //topikPublish
5        sensor: {
6            tipe: String      //tipeSensor
7            index: String     //indexSensor
8            ip: String        //alamatIpSource
9            module: String    //moduleSensor
10       }
11       humidity: {
12           value: number,    //nilaiHum
13           unit: String      //satuanHum
14       }
15       temperature: {
16           value: number,    //nilaiTemp
17           unit: String      //satuanTemp
18       }
19  }
```

4.4.5 Perancangan Pengujian

Perancangan pengujian adalah untuk mendefinisikan kebutuhan yang diperlukan pada penelitian. Kebutuhan tersebut adalah kebutuhan fungsional dan non-fungsional.

4.4.5.1 Perancangan Pengujian Fungsional

Perancangan pengujian fungsional bertujuan untuk mengetahui tingkat fungsionalitas sistem sudah berjalan dengan semestinya. Untuk melakukan pengujian fungsional dibutuhkan skenario yang mendefinisikan kemungkinan pengujian yang terjadi. Skenario yang akan dijalankan pada pengujian fungsional melibatkan beberapa aspek dalam sistem. Aspek tersebut adalah *node sensor*, Keepalived, *load balancer*, IoT *middleware* dan *subscriber*.

Skenario pertama tentang pengujian fungsional yakni peran *load balancer* pada *publisher. Pertama*-tama kedua *node sensor* harus terhubung ke dalam *access point* yang terdapat perangkat lunak Keepalived di dalamnya. Perangkat lunak Keepalived kemudian akan mentranslasi alamat virtualnya ke alamat *load balancer*. *Node sensor* A berjalan dan mengirimkan data kelembapan dan suhu ke IoT *middleware* melalui *load balancer* yang telah ditranslasi menggunakan protokol MQTT. *Node sensor* B berjalan dan mengirimkan data kelembapan dan suhu ke IoT *middleware* melalui *load balancer* yang telah ditranslasi menggunakan protokol CoAP. *Node sensor* A akan mem-*publish* data dengan topik *home/*kitchen, sedangkan *node sensor* B akan mem-*publish* data dengan topik *home/garage*. *Load balancer* meneruskan *traffic* tujuan *node sensor* ke IoT *middleware* sesuai dengan algoritme *round robin*. *Load balancer* akan merekam aktifitas *load balancing* pada *file log*-nya. Ketika pesan berhasil masuk ke IoT *middleware*, data akan terekam pada program *pm2 logs* di IoT *middleware*.

Skenario kedua berfokus pada peran *subscriber* dalam sistem. *Subscriber* terlebih dahulu harus terhubung dengan *access point* yang terdapat perangkat lunak Keepalived di dalamnya. *Subscriber* melakukan proses *subscriber* dengan topik yang diinginkan menggunakan protokol MQTT. *Subscriber* men-*subscribe* ke alamat *load balancer* yang telah ditranslasi oleh perangkat lunak Keepalived. *Load balancer* meneruskan *traffic* ke IoT *middleware* menggunakan algoritme *round robin*. *Load balancer* akan merekam aktifitas *load balancing* pada *file log*-nya. Apabila koneksi berhasil dibuat, data akan terekam pada program *pm2 logs* di IoT *middleware*. Tabel 4.6 menjelaskan justifikasi terkait pengujian yang akan dilakukan.

Tabel 4.6 Justifikasi Pengujian Fungsional

Kode	Fungsi	Justifikasi
PF_001	Perangkat lunak Keepalived dapat menentukan *node master* dan *node backup* pada *load balancer*	• Mengakses *file syslog*. • Memeriksa apakah Keepalived dapat menentukan *node master* dan *node backup*.
PF_002	*Load balancer* mampu meneruskan *traffic* MQTT dari *sensor* ke IoT *middleware*	• Menjalankan *access.log* pada *load balancer.* • Melakukan pemeriksaan apakah *traffic* dapat diteruskan ke IoT *middleware* oleh *load balancer* melalui *access.log* nya.
PF_003	*Load balancer* mampu meneruskan *traffic* CoAP	• Menjalankan *access.log* pada *load balancer.*

Tabel 4.6 Justifikasi Pengujian Fungsional (lanjutan)

	dari *sensor* ke IoT *middleware*	• Melakukan pemeriksaan apakah *traffic* dapat diteruskan ke IoT *middleware* oleh *load balancer* melalui *access.log* nya.
PF_004	*Load balancer* mampu mendistribusikan *traffic* CoAP dan MQTT dari *sensor* ke IoT *middleware*	• Men-*generate* sepuluh paket CoAP dan MQTT. • Menjalankan *access.log* pada *load balancer*. • Melakukan pemeriksaan apakah *traffic* dapat diteruskan ke IoT *middleware* oleh *load balancer* melalui *access.log* nya.
PF_005	IoT *middleware* dapat menerima data dari protokol MQTT	• Menjalankan IoT *middleware* dan menjalankan program *pm2 logs.* • Melakukan pemeriksaan data pada IoT *middleware* apakah telah diterima dan terekam pada program *monitoring pm2 logs.*
PF_006	IoT *middleware* dapat menerima data dari protokol CoAP	• Menjalankan IoT *middleware* dan menjalankan program *pm2 logs.* • Melakukan pemeriksaan data pada IoT *middleware* apakah telah diterima dan terekam pada program *monitoring pm2 logs.*
PF_007	IoT *middleware* dapat menerima data dari protokol CoAP dan MQTT	• Menjalankan IoT *middleware* dan menjalankan program *pm2 logs.* • Melakukan pemeriksaan data pada IoT *middleware* apakah telah diterima dan terekam pada program *monitoring pm2 logs.*
PF_008	*Cluster* dapat mendistribusikan data dari IoT *middleware*	• Melakukan pemeriksaan pada Redis tentang topik yang disimpan.
PF_009	IoT *middleware* dapat mengirimkan data dengan protokol CoAP dan MQTT ke *subscriber* menggunakan protokol MQTT.	• Menjalankan IoT *middleware* dan program *pm2 logs.* • Memeriksa apakah data dari *sensor node* A dan *sensor node* B berhasil diterima.

Tabel 4.6 Justifikasi Pengujian Fungsional (lanjutan)

PF_010	Mengetahui semua *key* Redis dapat terhubung satu sama lain.	• Memeriksa status pada salah satu Redis. • Mengakses data dengan topik yang disimpan di Redis lain.

4.4.5.2 Perancangan Pengujian Non-Fungsional

Perancangan pengujian non-fungsional pada penelitian ini adalah skalabilitas. Perancangan pengujian skalabilitas, bertujuan untuk mengukur kinerja *load balancer* pada IoT *middleware* dalam menangani berbagai proses pada kasus-kasus yang berbeda. Kasus-kasus tersebut berkaitan dengan berubahnya jumlah klien pada *publisher* dan *subscriber*. Parameter yang akan diuji adalah *time publish*, *time subscribe*, *concurrent publish*, dan *concurrent subscribe. Time publish* adalah waktu yang dibutuhkan sejumlah *publisher* untuk melakukan proses *publish* ke IoT *middleware*. *Time subscribe* adalah adalah waktu yang dibutuhkan sejumlah *subscriber* untuk melakukan proses *subscribe* ke IoT *middleware*. *Concurrnet publish* adalah banyak jumlah pesan yang mampu ditangani IoT *middleware* pada sejumlah *publisher*. *Concurrent subscribe* adalah banyak jumlah pesan yang mampu ditangani IoT *middleware* pada sejumlah *subscriber*. Variasi klien yang ditentukan untuk pengujian non-fungsional adalah 100, 500, 1000, 1500.

Skenario pengujian non-fungsional terbagi menjadi dua skenario. Skenario pertama merupakan skenario yang melibatkan peran *publisher* pada sistem. Skenario kedua merupakan skenario yang melibatkan peran *subscriber* pada sistem. Skenario pertama, pada sisi *publisher* men-*generate client* dan data yang akan di-*publish* sebesar 100, 500, 1000, dan 1500 baik untuk *publisher* MQTT atau CoAP. Untuk mencapai hal tersebut digunakan program yang berjalan secara *asynchronous*. *Client publisher* yang telah dibuat, mengirimkan koneksi dan datanya satu per satu untuk setiap protokol yang digunakan (berurut sesuai protokol). Selanjutnya mengambil data paket yang telah direkam pada IoT *middleware*. Skenario kedua, pada sisi *subscriber* men-*generate client subscribe* sebesar 100, 500, 1000, dan 1500 yang berkomunikasi menggunakan protokol MQTT.

Tabel 4.7 Justifikasi Pengujian Non-Fungsional

Kode	Fungsi	Justifikasi
PNF_001	Mengetahui berapa lama waktu yang dibutuhkan IoT *middleware* untuk	Pengujian dengan parameter *time publish*:

Tabel 4.7 Justifikasi Pengujian Non-Fungsional (lanjutan)

	menangani sejumlah *publisher* dengan protokol MQTT dan CoAP melalui *load balancer.*	• Menjalankan *pm2 logs* pada IoT *middleware* untuk memantau pesan pada IoT *middleware*. • Filter paket dijalankan pada IoT *middleware* menggunakan tcpdump. • Analisis paket hasil tangkapan dari tcpdump untuk menentukan nilai *time publish.*
PNF_002	Mengetahui berapa banyak jumlah pesan *publish* yang mampu ditangani IoT *middleware* dalam satu detik.	Pengujian dengan parameter *concurrent publish* : • pm2 *logs* dijalankan pada IoT *middleware* untuk memantau pesan pada IoT *middleware*. • Filter paket dijalankan pada IoT *middleware* menggunakan tcpdump. • Analisis paket hasil tangkapan dari tcpdump untuk menentukan nilai *time publish.*
PNF_003	Mengetahui berapa lama waktu yang dibutuhkan IoT *middleware* untuk menangani sejumlah *subscriber* dengan protokol MQTT melalui *load balancer.*	Pengujian dengan parameter *time subscribe*: • pm2 *logs* dijalankan pada IoT *middleware* untuk memantau pesan pada IoT *middleware*. • Filter paket dijalankan pada IoT *middleware* menggunakan tcpdump. • Analisis paket hasil tangkapan dari tcpdump untuk menentukan nilai *time subscribe.*
PNF_004	Mengetahui berapa banyak jumlah pesan yang mampu ditangani IoT *middleware* dalam satu detik.	Pengujian dengan parameter *concurrent subscribe* : • pm2 *logs* dijalankan pada IoT *middleware* untuk memantau pesan pada IoT *middleware*. • Filter paket dijalankan pada IoT *middleware* menggunakan tcpdump. • Analisis paket hasil tangkapan dari tcpdump untuk menentukan nilai *time publish.*

BAB 5 IMPLEMENTASI

Pada bab ini dijelaskan implementasi *load balancing* berdasarkan perancangan yang dibangun. Implementasi pada penelitian ini meliputi implementasi *load balancer*, implementasi Keepalived, implementasi topologi jaringan, implementasi *sensor* dan implementasi *subscriber*.

5.1 Implementasi *Load Balancer*

Pada bagian implementasi *load balancer* dijelaskan pemanfaatan API NGINX untuk proses *load balancing.*

5.1.1 Instalasi dan Konfigurasi NGINX

Pada penelitian ini akan dilakukan instalasi NGINX sebagai perangkat lunak *load balancing*. Instalasi akan dilakukan diatas sistem operasi Raspbian. Selanjutnya Perangkat lunak NGINX akan dikonfigurasi sesuai dengan perancangan pada bab 4 untuk memenuhi kebutuhan pada sistem.

Proses pemasangan perangkat lunak NGINX dengan menjalankan *script* pada repositori github. Cara mendapatkan *script* tersebut adalah dengan mengunduhnya menggunakan perintah `wget` ke *link* yang dituju. Setelah itu merubah *file permission* dari *script* tersebut. Agar NGINX mampu mendengarkan *traffic* MQTT dan CoAP, pada saat pemasangan konfigurasi NGINX menggunakan modul *stream*. Gambar 5.1 menjelaskan penambahan modul *stream* pada *file script*. Setelah selesai menambahkan modul *stream*, lalu menjalankan *script* yang dimaksud pada sistem operasi Debian. NGINX yang dipasang adalah NGINX versi *mainline*. Tabel 5.1 menunjukkan langkah keseluruhan mengakses *file script* pemasangan NGINX.

```
NGINX_MODULES="--without-http_ssi_module \
--without-http_scgi_module \
--without-http_uwsgi_module \
--without-http_geo_module \
--without-http_split_clients_module \
--without-http_memcached_module \
--without-http_empty_gif_module \
--without-http_browser_module \
--with-threads \
--with-stream \
--with-file-aio \
--with-http_ssl_module \
--with-http_v2_module \
--with-http_mp4_module \
--with-http_auth_request_module \
--with-http_slice_module \
--with-http_stub_status_module \
--with-http_realip_module"
```

Gambar 5.1 Modul *Stream* NGINX

Tabel 5.1 Pemasangan NGINX dengan *File Script*

No	Instalasi 1: Perintah Unduh Script Pemasangan NGINX
1	$ sudo wget https://raw.githubusercontent.com/Angristan/nginx-autoinstall/master/nginx-autoinstall.sh
2	
3	$ chmod +x nginx-autoinstall.sh
4	$ ./nginx-autoinstall.sh

Konfigurasi perangkat lunak NGINX dilakukan dengan mengubah *file* `nginx.conf` pada direktori `/etc/nginx/`. Langkah pertama adalah membuat direktori `stream_conf.d` di dalam direktori `/etc/nginx/`. Kemudian membuat *file* baru dengan nama `stream.conf`, berisi konfigurasi yang diperlukan untuk menjadikan NGINX sebagai perangkat lunak *load balancer* yang diinginkan. Secara *default* NGINX akan menggunakan algoritme *round robin* sebagai algoritme *load balancing*. Tabel 5.2 menjelaskan baris yang ditambahkan pada *file* `nginx.conf`. Sedangkan tabel 5.3 menjelaskan pengaturan pada *file* `stream.conf`.

Tabel 5.2 Kode Pengaturan pada *File* nginx.conf

No	Pengaturan 1: Konfigurasi pada File nginx.conf
1	stream {
2	include stream_conf.d/*.conf;
3	}

Penjelasan kode pada tabel 5.4 adalah sebagai berikut:

- Blok `stream {}`, memerintahkan NGINX untuk menangkap koneksi yang berupa TCP dan UDP.
- Direktif `include`, untuk memasukkan konfigurasi yang ada pada sebuah file ke pengaturan NGINX.

Tabel 5.3 Kode Pengaturan pada *File* stream.conf

No	Pengaturan 2: Konfigurasi pada File stream.conf
1	log_format protocol
2	'$proxy_protocol_addr $remote_addr [$time_local]'
3	'$status $bytes_sent $bytes_received $upstream_addr';
4	
5	upstream mqtt {
6	# blok untuk menentukan algoritme *load balancing*
7	server 192.168.0.101:1883; #middleware 1
8	server 192.168.0.102:1883; #middleware 2
9	}
10	
11	upstream coap {
12	# blok untuk menentukan algoritme *load balancing*
13	server 192.168.0.101:5683; #middleware 1
14	server 192.168.0.102:5683; #middleware 2
15	}
16	
17	server {
18	listen 1883;
19	proxy_pass mqtt;
20	proxy_connect_timeout 1s;
21	access_log /var/log/nginx/protocol_access.log protocol;

Tabel 5.3 Kode Pengaturan pada *File* stream.conf (lanjutan)

```
22  }
23
24  server {
25     listen 5683 udp;
26  proxy_pass coap;
27     proxy_connect_timeout 1s;
28     proxy_responses 1;
29     access_log /var/log/nginx/protocol_access.log protocol;
30  }
```

Penjelasan pada tabel 5.3 adalah sebagai berikut:

- Pengaturan format pada *log* yang akan diterima oleh NGINX ditunjukkan dengan memanggil direktif `log_format`.
- Blok `upstream` menjelaskan daftar dari alamat *backend server* beserta *port* yang digunakan untuk tujuan *load balancing*.
- Direktif `server`, menunjukkan alamat *server* beserta *port* yang digunakan baik protokol MQTT ataupun CoAP.
- Blok `server {}`, menjelaskan pengaturan yang akan diterapkan pada *server*.
- Direktif `listen`, menunjukkan *port* yang didengar layanannya oleh *server*.
- Keteranga `udp` untuk dapat mendengarkan paket *datagram*.
- Direktif `proxy_pass` untuk mendefinisikan koneksi yang akan diteruskan ke *proxy* tujuan.
- Direktif `proxy_connect_timeout`, mengatur *timeout* yang dibutuhkan untuk membangun koneksi dengan sejumlah *server* dalam grup.
- Direktif `proxy_responses`, menunjukkan jumlah *datagram* yang diterima sebagai respon ke klien *datagram*.
- Direktif `access_log` untuk menuliskan *log* pada file yang ditentukan.

5.2 Implementasi Keepalived pada *Load Balancer*

Perangkat lunak Keepaalived yang digunakan pada penelitian ini adalah versi 2.0.10.

5.2.1 Proses *Instalasi* Perangkat Lunak Keepalived

Proses awal *instalasi* Keepalived adalah melakukan pemasangan *dependencies* yang dibutuhkan terlebih dahulu. Kemudian mengunduh *file source* keepalived di alamat http://www.keepalived.org/download.html. Lalu mengekstraksi *file* kompresi yang telah diunduh tersebut. Lalu, konfigurasi *file* dengan *prefix* yang diinginkan sebelum melaksanakan *make install*. Perintah *prefix* berguna untuk menentukan letak *folder instalasi*. Sedangkan *make install* adalah perintah *instalasi* menggunakan informasi dari konfigurasi. Tabel 5.4 menunjukkan perintah untuk *instalasi dependencies* yang diperlukan.

Tabel 5.4 *Instalasi Dependencies* yang diperlukan

No	Instalasi 2: Instalasi Dependencies yang diperlukan
1 2	$ sudo apt-get install curl gcc libssl-dev libnl-3-dev libnl-genl- 3-dev libsnmp-dev

Tabel 5.5 menjelaskan cara untuk mengunduh *file* Keepaalived, konfigurasi beserta pemasangannya.

Tabel 5.5 Langkah *Instalasi* Keepalived

No	Instalasi 3: Langkah *Instalasi* Keepalived
1	`$ wget http://www.keepalived.org/software/keepalived-2.0.20.tar.gz`
2	`$ tar -zxf keepalived-2.0.20.tar.gz`
3	`$ cd keepalived-2.0.20`
4	`$ ./configure --prefix=/usr/local/keepalived`
5	`$ make`
6	`$ sudo make install`

5.2.2 Proses Pengaturan *File Script* Keepalived

Pada penelitian ini dibuat juga *script* untuk menjalankan perangkat lunak Keepalived pada *service unit* sistem operasi. Langkah awal adalah membuat *file* konfigurasi pada direktori `/etc/init`. Kemudian mendeskripsikan pada *file* konfigurasi tingkat layanan Keepalived pada saat berjalan dan berhenti. Layanan Keepalived ini akan aktif pada setiap kondisi normal, dan tidak aktif pada setiap kondisi yang lain. Setelah itu memastikan layanan Keepalived menyala kembali, setiap ada *event failure*. Pembuatan *script* Keepalived adalah dengan mengetikkan perintah `sudo nano /etc/init/keepalived.conf`. Tabel 5.6 menunjukkan baris pengaturan pada *file script* yang dibuat.

Tabel 5.6 Pengaturan *Script* Keepalived

No	Pengaturan 3: Pembuatan *Script* Keepalived
1	`description "load-balancing and high-availability service"`
2	
3	`start on runlevel [2345]`
4	`stop on runlevel [!2345]`
5	
6	`respawn`
7	
8	`exec /usr/local/keepalived/sbin/keepalived --dont-fork`

5.2.3 Proses Konfigurasi Perangkat Lunak Keepalived

Keepaalived membutuhkan pengaturan pada *file* perangkat lunaknya. Hal tersebut agar fungsi yang diinginkan dapat berjalan dengan benar. Untuk itu perlu dibuat *file* konfigurasi pada perangkat lunak Keepalived. *File* konfigurasi perangkat lunak berbeda dengan *file script* Keepadlived. *File* konfigurasi bertujuan untuk mengimplementasikan fungsi yang telah dijelaskan pada bab perancangan. Sedangkan *file script*, bertujuan agar Keepalived dapat berjalan pada *service unit* sistem operasi. Langkah awal adalah dengan membuat *file* konfigurasi perangkat lunak pada direktori `/etc/keepalived/`. Pembuatan *file* konfigurasi Keepalived menggunakan *text editor nano*. Terdapat dua pengaturan *file* konfigurasi Keepalived. Pengaturan pertama merupakan pengaturan pada mesin *load balancer master*. Pengaturan kedua merupakan pengaturan pada mesin *load balancer back up*. Tabel 5.7 menunjukkan pengaturan pada *file* konfigurasi *load balancer master*. Tabel 5.8 menunjukkan pengaturan pada *file* konfigurasi *load balancer back up*.

Tabel 5.7 Konfigurasi Keepalived pada *Load Balancer Master*

No	Pengaturan 4: Konfigurasi Keepalived pada *Load balancer Master*
1	global_defs {
2	# Keepalived process identifier
3	router_id nginx
4	}
5	# Script used to check if Nginx is running
6	vrrp_script check_nginx {
7	script "/bin/check_nginx.sh"
8	interval 2
9	weight 20
10	}
11	# Virtual interface
12	# The priority specifies the order in which the assigned interface
13	to take over in a failover
14	vrrp_instance VI_01 {
15	state MASTER
16	interface eth0
17	virtual_router_id 51
18	priority 123
19	# The virtual ip address shared between the two loadbalancers
20	virtual_ipaddress {
21	192.168.0.111
22	}
23	track_script {
24	check_nginx
25	}
26	authentication {
27	auth_type AH
28	auth_pass secret
29	}
30	}

Tabel 5.8 Konfigurasi Keepalived pada *Load Balancer Back Up*

No	Pengaturan 5: Konfigurasi Keepalived pada *Load balancer Back Up*
1	global_defs {
2	# Keepalived process identifier
3	router_id nginx
4	}
5	# Script used to check if Nginx is running
6	vrrp_script check_nginx {
7	script "/bin/check_nginx.sh"
9	interval 2
10	weight 20
11	}
12	# Virtual interface
13	# The priority specifies the order in which the assigned interface
14	to take over in a failover
15	vrrp_instance VI_01 {
16	state BACKUP
17	interface eth0
18	virtual_router_id 51
19	priority 120
20	# The virtual ip address shared between the two loadbalancers

Tabel 5.8 Konfigurasi Keepalived pada *Load Balancer Back Up* (lanjutan)

```
21    virtual_ipaddress {
22      192.168.0.111
23    }
24    track_script {
25    check_nginx
26    }
27    authentication {
28      auth_type AH
29      auth_pass secret
30    }
31  }
```

Penjelasan pengaturan penting pada tabel 5.8:

- `global_defs { }` menjelaskan proses yang ingin diidentifikasi pada sistem operasi.
- `vrrp_script [nama_script]` mendifinisikan *script* yang digunakan untuk memeriksa proses *load balancing*.
- `vrrp_instance VI_01 { }` mendefinisikan antarmuka *virtual* yang digunakan oleh Keepalived. Pada blok ini ditentukan status *load balancer* adalah *master* menggunakan *directive* `state MASTER`. Kemudian *master* harus memiliki prioritas lebih tinggi dari *back up*. Pada blok ini juga mendefinisikan alamat IP *virtual* yang digunakan, yakni 192.168.0.111.

Penjelasan pengaturan penting pada tabel 5.9:

- `global_defs { }` menjelaskan proses yang ingin diidentifikasi pada sistem operasi.
- `vrrp_script [nama_script]` mendifinisikan *script* yang digunakan untuk memeriksa proses *load balancing*.
- `vrrp_instance VI_01 { }` mendefinisikan antarmuka *virtual* yang digunakan oleh Keepalived. Pada blok ini ditentukan status *load balancer* adalah *master* menggunakan *directive* `state BACKUP`. Kemudian *back up* harus memiliki prioritas lebih rendah dari *master*. Pada blok ini juga mendefinisikan alamat IP *virtual* yang digunakan, yakni 192.168.0.111.

5.2.4 Proses Pembuatan *Script* Pengecekan Proses NGINX

Perangkat lunak Keepalived mendefinisikan parameter *failover*-nya pada proses NGINX yang berjalan. Jika pada *load balancer master* proses tersebut tersedia, Keepalived akan menetapkan semua proses *load balancing* pada *node master*. Akan tetapi, jika pada *load balancer master* proses tersebut tidak tersedia, Keepalived akan menetapkan semua proses *load balancing* pada *node back up*. Untuk itu diperlukan *script* yang melakukan pemeriksaan terhadap proses NGINX di setiap mesin *load balancer*. *Script* tersebut harus diletakkan pada direktori `/bin`. Tabel 5.9 menunjukkan *pseudococe* program pengecekan proses NGINX.

Tabel 5.9 *Script* Pemeriksaan Proses NGINX

No	Algoritme 1: *Pseudocode Script* Pengecekan Proses NGINX
1	#!/bin/sh
2	if pid = nginx_pid
3	exit 1
4	endif

5.3 Implementasi *Sensor*

Implementasi pada *sensor* menjelaskan penarapan perangkat lunak untuk pengiriman data suhu dan kelembapan. Penelitian ini menggunakan dua buah protokol pengiriman data, CoAP dan MQTT. Keduanya terdapat perbedaan dalam teknis pengiriman datanya. Pengiriman data menggunakan protokol CoAP, data pada *sensor* akan langsung dikirimkan ke IoT *middleware*. Sedangkan, pengiriman data menggunakan protokol MQTT, *node* harus terhubung terlebih dahulu dengan IoT *middleware* sebelum dapat mengirimkan data. Tabel 5.10 merupakan *pseudocode* program yang akan mengirimkan data suhu dan kelembapan menggunakan protokol CoAP ke IoT *middleware* melalui *load balancer*. Dan tabel 5.11 adalah *pseudocode* untuk pengiriman data menggunakan protokol MQTT.

Tabel 5.10 *Pseudocode* Program *Sensor* pada Protokol CoAP

No	Algoritme 2 : *Pseudocode* Program *Sensor* dengan Protokol CoAP
1	SET *sensor*lib = require *node* -dht-*sensor*
2	SET temp = read temperature from *sensor*
3	SET hum = read humidity from *sensor*
4	DEFINE coapp ()
5	SET coap = require coap
6	SET cron = require *node*-cron
7	SET payload = payload data from *sensor*
8	DO send payload in JSON with method POST
9	DO print Date and "terkirim"
10	END

Tabel 5.11 *Pseudocode* Program *Sensor* pada Protokol MQTT

No	Algoritme 3 : *Pseudocode* Program *Sensor* dengan Protokol MQTT
1	SET async = require async
2	SET MQTT = require mqtt
3	SET payload = payload data and read humidity and temperature
4	from *sensor*
5	DO async (1, next)
6	DO print Date in String
7	DO connect to MQTT Gateway using ip and port
8	DO publish (topik, payload in JSON, use qos 1)
9	END
10	DO next (err)
11	DO print "terkirim"
12	END

5.4 Implementasi *Subscriber*

Program *subscriber* dibuat dengan menggunakan bahasa Python. *Subscriber* dijalankan pada *Raspberry* Pi. Program *subscriber* juga mengimplementasikan modul *asynchronous* pada Python dengan memanfaatkan *thread*. Modul *asynchronous* tersebut digunakan pada pengujian *concurrent*

subscribe. Pada program *subscriber* terdapat fungsi `loop()` agar *callback* dapat diproses. Tabel 5.12 adalah *pseudocode* program yang diimplementasikan pada *subscriber*.

Tabel 5.12 *Pseudocode* Program *Subscriber*

No	Algoritem 4: *Pseudocode* Program *Subscriber*
1	import from paho.mqtt.client as mqtt
2	import threading
3	import time
4	
5	FUNCTION callback on_message:
6	time.sleep(1 second)
7	print("received message =",String(message.payload.decode("utf-8")))
8	ENDFUNCTION
9	
10	FUNCTION subscribe:
11	client <-- mqtt.Client
12	Set broker's address
13	client.on_message <-- on_message
14	client.connect(broker's address)
15	client.loop_start()
16	client.subscribe("name of topic")
17	time.sleep(1 second)
18	client.disconnect()
19	client.loop_stop
20	ENDFUNCTION
21	
22	FUNCTION main:
23	FOR i <-- to Desired range
24	t = threading.Thread(name <-- "Function's name", target <--
25	Function's name, args <-- (i,))
26	ENDFOR
27	ENDFUNCTION

5.4 Implementasi Topologi Jaringan

Implementasi topologi jaringan akan membahas pengaturan yang dibutuhkan pada setiap perangkat untuk saling berkomunikasi. Pengaturan ini adalah pengaturan pada *interface* di setiap perangkat dengan memberikannya alamat IP *static*. Perangkat yang akan dikonfigurasi meliputi perangkat *sensor*, *load balancer*, IoT *middleware*, dan *cluster*.

5.4.1 Konfigurasi pada *Load Balancer*

Konfigurasi pada *load balancer* adalah mengatur *interface eth0* agar mendapatkan IP *static*. Konfigurasi dapat dilakukan dengan menambahkan baris pada tabel 5.13 ke dalam *file* `/etc/dhcpcd.conf`. Alamat IP *static* yang digunakan pada perangkat *load balancer* adalah 192.168.0.104 pada *load balancer* pertama. Sedangkan *load balancer* kedua memiliki alamat IP 192.168.0.105. Pengaturan alamat IP pada *load balancer* kedua adalah dengan menambahkan baris yang tertera di tabel 5.14

Tabel 5.13 Konfigurasi Interface Load Balancer Pertama

No	Pengaturan 6: Konfigurasi *interface Load Balancer* Pertama
1	interface eth0
2	static ip_address=192.168.0.104/24
3	static routers=192.168.0.1
4	static domain_name_servers=192.168.0.1

Tabel 5.14. Konfigurasi Interface Load Balancer Kedua

No	Pengaturan 7: Konfigurasi *interface Load Balancer* Kedua
1	interface eth0
2	static ip_address=192.168.0.105/24
3	static routers=192.168.0.1
4	static domain_name_servers=192.168.0.1

5.4.2 Konfigurasi pada IoT *Middleware*

Lingkungan pada penelitian ini akan menggunakan dua perangkat IoT *middleware*. Masing-masing alamat IP nya adalah 192.168.0.101 dan 192.168.0.102. pengaturan pada *middleware* 1 dilakukan dengan menambahkan baris pada tabel 5.15 ke *file* `/etc/dhcpcd.conf` pada perangkat IoT *middleware* 1. Sedangkan pengaturan pada IoT *middleware* 2 dilakukan dengan menambahkan baris pada tabel 5.16 ke *file* `/etc/dhcpcd.conf` pada perangkat IoT *middleware* 2.

Tabel 5.15 Konfigurasi *Interface* IoT *Middleware* 1

No	Pengaturan 8: Konfigurasi interface IoT Middleware 1
1	interface eth0
2	static ip_address=192.168.0.101/24
3	static routers=192.168.0.1
4	static domain_name_servers=192.168.0.1

Tabel 5.16 Konfigurasi *Interface* IoT *Middleware* 2

No	Pengaturan 9: Konfigurasi interface IoT Middleware 2
1	interface eth0
2	static ip_address=192.168.0.102/24
3	static routers=192.168.0.1
4	static domain_name_servers=192.168.0.1

5.4.3 Konfigurasi pada *Cluster*

Lingkungan pada penelitian ini akan menggunakan dua perangkat *cluster*. Masing-masing alamat IP nya adalah 192.168.0.107 dan 192.168.0.108. pengaturan pada *cluster* 1 dilakukan dengan menambahkan baris pada tabel 5.17 ke *file* `/etc/dhcpcd.conf` pada perangkat *cluster* 1. Sedangkan pengaturan pada *cluster* 2 dilakukan dengan menambahkan baris pada tabel 5.18 ke *file* `/etc/dhcpcd.conf` pada perangkat *cluster* 2.

Tabel 5.17 Konfigurasi *Interface Cluster* 1

No	Pengaturan 10: Konfigurasi interface Cluster 1
1	interface eth0
2	static ip_address=192.168.0.107/24
3	static routers=192.168.0.1
4	static domain_name_servers=192.168.0.1

Tabel 5.18 Konfigurasi *Interface Cluster* 2

No	Pengaturan 11: Konfigurasi interface Cluster 2
1	interface eth0
2	static ip_address=192.168.0.108/24
3	static routers=192.168.0.1
4	static domain_name_servers=192.168.0.1

5.4.4 Konfigurasi pada *Publisher*

Klien yang bertindak sebagai *publisher* pada penelitian ini menggunakan dua protokol, MQTT dan CoAP. Setiap protokol akan diterapkan pada perangkat yang berbeda. Alamat IP *static* pada perangkat dengan protokol MQTT adalah 192.168.4.10, sedangkan alamat IP *static* pada perangkat yang menggunakan protokol CoAP adalah 192.168.4.9. pengaturan pada klien MQTT dilakukan dengan menambahkan baris pada tabel 5.19 ke *file* `/etc/dhcpcd.conf` pada perangkat klien MQTT. Sedangkan pengaturan pada klien CoAP dilakukan dengan menambahkan baris pada tabel 5.20 ke *file* `/etc/dhcpcd.conf` pada perangkat klien CoAP.

Tabel 5.19 Konfigurasi *Interface* Klien MQTT

No	Pengaturan 12: Konfigurasi interface Klien MQTT
1	interface wlan0
2	static ip_address=192.168.4.10/24
3	static routers=192.168.4.1
4	static domain_name_servers=192.168.4.1

Tabel 5.20 Konfigurasi *Interface* Klien CoAP

No	Pengaturan 13: Konfigurasi interface Klien CoAP
1	interface wlan0
2	static ip_address=192.168.4.9/24
3	static routers=192.168.4.1
4	static domain_name_servers=192.168.4.1

5.4.5 Konfigurasi pada *Subscriber*

Klien yang bertindak sebagai *subscriber* pada penelitian ini menggunakan protokol MQTT. Alamat IP *static* pada *subscriber* adalah 192.168.4.11. pengaturan pada *subscriber* dilakukan dengan menambahkan baris pada tabel 5.21 ke *file* `/etc/dhcpcd.conf` pada perangkat *subscriber*.

Tabel 5.21 Konfigurasi *Interface Subscriber*

No	Pengaturan 10: Konfigurasi interface *Subscriber*
1	interface wlan0
2	static ip_address=192.168.4.11/24
3	static routers=192.168.4.1
4	static domain_name_servers=192.168.4.1

BAB 6 PENGUJIAN DAN PEMBAHASAN HASIL PENGUJIAN

Setelah implementasi selesai dilakukan, langkah selanjutnya adalah melakukan pengujian pada IoT *middleware* dengan *load balancer*. Pengujian ini dilakukan untuk mengetahui setiap fungsi pada implementasi dapat berjalan dengan benar dan mengetahui kinerja sistem dari aspek skalabilitas dalam menagani sejumlah beban yang diberikan. Proses pengujian terbagi menjadi pengujian fungsional dan pengujian non-fungsional, yakni skalabilitas.

6.1 Pengujian Fungsional

Pengujian fungsional merupakan sekumpulan skema untuk melihat setiap fungsi yang dirancang pada perancangan dapat berjalan dengan benar. Kesesuain fungsi hasil implementasi harus sesuai dengan perancangan yang telah dilakukan. Pengujian fungsional juga akan menunjukkan integrasi antara sub-sistem. Integrasi yang dimaksud adalah integrasi antara *sensor*, perangkat lunak Keepalived, *load balancer*, IoT *middleware*, *cluster* dan *subscriber*. Skenario pengujian fungsional menggunakan skenario pada bab perancangan pengujian fungsional. Terdapat dua skenario yang akan dijalankan sebagaimana pada perancangan pengujian fungsional. Skenario pertama adalah skema pengujian fungsional bertema peran *load balancer* pada *publisher*. Skenario kedua adalah skema pengujian fungsional bertema peran *load balancer* pada *subscriber*. Hasil pengujian akan dijelaskan perbagian sesuai skema pada perancangan.

6.2.1 Perangkat Lunak Keepalived dapat menentukan *node master* dan *node backup* pada *load balancer*

A. Tujuan Pengujian

Mengetahui apakah perangkat lunak Keepalived yang diimplementasikan pada *load balancer* mampu menentukan *node master* dan *node backup*.

B. Prosedur pengujian

1. Mengakses *file syslog* pada sistem operasi perangkat *load balancer*.
2. Memeriksa apakah Keepalived dapat menentukan *node master* dan *node backup*.

C. Hasil yang diharapkan

Keepalived mampu menentukan *node master* dan *node back up* pada *load balancer*.

D. Hasil Pengujian

Keepalived mampu menentukan *node master* dan *node backup* pada *load balancer.* Hasil pengujian dijelaskan oleh gambar 6.1 dan gambar 6.2. Gambar 6.1 menunjukkan hasil pengujian pada *node master*. Gambar 6.2 menunjukkan hasil pengujian pada *node back up*.

```
Line 25970: Mar 29 09:43:19 TheLoadBalancer_1 Keepalived_vrrp[983]:
Assigned address 192.168.0.104 for interface eth0

Line 25976: Mar 29 09:43:19 TheLoadBalancer_1 Keepalived_vrrp[983]:
VRRP_Script(check_nginx) succeeded

Line 25977: Mar 29 09:43:19 TheLoadBalancer_1 Keepalived_vrrp[983]:
(VI_01) Changing effective priority from 103 to 123

Line 25978: Mar 29 09:43:22 TheLoadBalancer_1 Keepalived_vrrp[983]:
(VI_01) Receive advertisement timeout

Line 25979: Mar 29 09:43:22 TheLoadBalancer_1 Keepalived_vrrp[983]:
(VI_01) Entering MASTER STATE

Line 25980: Mar 29 09:43:22 TheLoadBalancer_1 Keepalived_vrrp[983]:
(VI_01) setting VIPs.

Line 25981: Mar 29 09:43:22 TheLoadBalancer_1 Keepalived_vrrp[983]:
Sending gratuitous ARP on eth0 for 192.168.0.111

Line 25982: Mar 29 09:43:22 TheLoadBalancer_1 Keepalived_vrrp[983]:
(VI_01) Sending/queueing gratuitous ARPs on eth0 for 192.168.0.111
```

Gambar 6.1 Hasil Pengujian Kode PF_001 pada *Load Balancer* Satu

Gambar 6.1 menunjukkan tampilan *file syslog* pada direktori `/var/log/`. Perangkat lunak Keepalived menetapkan alamat IP 192.168.0.104 pada *interface* `eth0` sebagai alamat virtual Keepalived pada 192.168.0.111. *Priority* pada *node Load balancer* satu dirubah menjadi bernilai 123 (lebih besar dari *priority load balancer* dua). Kemudian Keepalived menetapkan *node load balancer* satu sebagai *node master*. Hasil pengujian menunjukkan bahwa Keepalived mampu menentukan *load balancer* satu dengan alamat IP 192.168.0.104 sebagai *node master*.

```
Line 18160: Mar 29 09:43:28 TheLoadBalancer_2 Keepalived_vrrp[985]:
Assigned address 192.168.0.105 for interface eth0

Line 18166: Mar 29 09:43:28 TheLoadBalancer_2 Keepalived_vrrp[985]:
VRRP_Script(check_nginx) succeeded

Line 18167: Mar 29 09:43:28 TheLoadBalancer_2 Keepalived_vrrp[985]:
(VI_01) Changing effective priority from 101 to 121

Line 18163: Mar 29 09:43:28 TheLoadBalancer_2 Keepalived_vrrp[985]:
(VI_01) removing VIPs.

Line 18164: Mar 29 09:43:28 TheLoadBalancer_2 Keepalived_vrrp[985]:
(VI_01) Entering BACKUP STATE (init)
```

Gambar 6.2 Hasil Pengujian Kode PF_001 pada *Load Balancer* Dua

Gambar 6.2 menunjukkan tampilan *file syslog* pada direktori `/var/log/`. Perangkat lunak Keepalived menetapkan alamat IP 192.168.0.104 pada *interface* `eth0` sebagai alamat virtual Keepalived pada 192.168.0.111. *Priority* pada *node Load balancer* dua dirubah menjadi bernilai 121 (lebih kecil dari *priority load balancer* satu). Kemudian Keepalived menetapkan *node load balancer* dua sebagai *node back up*. Hasil pengujian menunjukkan bahwa Keepalived mampu

menentukan *load balancer* dua dengan alamat IP 192.168.0.105 sebagai *node back up*.

E. Keterangan

Berdasarkan hasil pengujian PF_001 disimpulkan bahwa integrasi perangkat lunak Keepalived pada *load balancer* berhasil dilakukan.

6.2.2 *Load balancer* mampu meneruskan *Traffic* MQTT dari *Sensor* ke IoT *Middleware*

A. Tujuan Pengujian

Mengetahui apakah *Load balancer* mampu meneruskan *traffic* MQTT dari *sensor* ke IoT *middleware*.

B. Prosedur pengujian

1. Menjalankan *access.log* pada *load balancer.*
2. Melakukan pemeriksaan apakah *traffic* dapat diteruskan ke IoT *middleware* oleh *load balancer* melalui *access.log* nya.

C. Hasil yang diharapkan

Load balancer mampu meneruskan *traffic* MQTT dari *sensor* ke IoT *middleware*.

D. Hasil Pengujian

Load balancer mampu meneruskan *traffic* MQTT dari *sensor* ke IoT *middleware*. Gambar 6.3 menunjukkan hasil pengujian fungsional dengan kode uji PF_002.

```
192.168.4.9 [29/Nov/2018:06:24:33 +0000] 200 8 192.168.0.101:1883
192.168.4.9 [29/Nov/2018:06:24:33 +0000] 200 8 192.168.0.102:1883
192.168.4.9 [29/Nov/2018:06:24:33 +0000] 200 8 192.168.0.101:1883
192.168.4.9 [29/Nov/2018:06:24:33 +0000] 200 8 192.168.0.102:1883
192.168.4.9 [29/Nov/2018:06:24:33 +0000] 200 8 192.168.0.101:1883
```

Gambar 6.3 Hasil Pengujian Kode PF_002

Pengujian fungsional dengan kode PF_002 dilakukan dengan mengakses *file* `protocol_access.log` pada direktori `/var/log/nginx/`. Pada gambar 6.3 menunjukkan *node sensor* MQTT (port 1883) mengirimkan data ke IoT *middleware* melalui *load balancer*. *Load balancer* kemudian menginterupsi *traffic* dan mendistribusikannya ke IoT *middleware* tujuan. Akses *log* dilakukan menggunakan perintah `tail –f /var/log/nginx/protocol_access.log` pada terminal. Hasil pengujian kode PF_002 menunjukkan *load balancer* mampu meneruskan *traffic* MQTT (*port* 1883) ke IoT *middleware* satu dengan alamat IP 192.168.0.101 dan IoT *middleware* dua dengan alamat IP 192.168.0.102.

E. Keterangan

Berdasarkan hasil pengujian PF_002 disimpulkan bahwa integrasi *sensor* MQTT dengan *load balancer* berhasil dilakukan.

6.2.3 *Load Balancer* mampu meneruskan *Traffic* CoAP dari *Sensor* ke IoT *Middleware*

A. Tujuan Pengujian

Mengetahui apakah *Load balancer* mampu meneruskan *traffic* CoAP dari *sensor* ke IoT *middleware*.

B. Prosedur pengujian

1. Menjalankan *access.log* pada *load balancer.*
2. Melakukan pemeriksaan apakah *traffic* dapat diteruskan ke IoT *middleware* oleh *load balancer* melalui *access.log* nya.

C. Hasil yang diharapkan

Load balancer mampu meneruskan *traffic* CoAP dari *sensor* ke IoT *middleware*.

D. Hasil Pengujian

Load balancer mampu meneruskan *traffic* CoAP dari *sensor* ke IoT *middleware*. Gambar 6.4 menunjukkan hasil pengujian fungsional dengan kode uji PF_003.

```
192.168.4.10 [29/Nov/2018:06:24:33 +0000] 200 28 192.168.0.101:5683
192.168.4.10 [29/Nov/2018:06:24:33 +0000] 200 28 192.168.0.102:5683
192.168.4.10 [29/Nov/2018:06:24:33 +0000] 200 28 192.168.0.101:5683
192.168.4.10 [29/Nov/2018:06:24:33 +0000] 200 28 192.168.0.102:5683
192.168.4.10 [29/Nov/2018:06:24:33 +0000] 200 28 192.168.0.101:5683
192.168.4.10 [29/Nov/2018:06:24:33 +0000] 200 28 192.168.0.102:5683
```

Gambar 6.4 Hasil Pengujian Kode PF_003

Pengujian fungsional dengan kode PF_003 dilakukan dengan mengakses *file* `protocol_access.log` pada direktori `/var/log/nginx/`. Pada gambar 6.3 menunjukkan *node sensor* CoAP (port 5683) mengirimkan data ke IoT *middleware* melalui *load balancer*. *Load balancer* kemudian menginterupsi *traffic* dan mendistribusikannya ke IoT *middleware* tujuan. Akses *log* dilakukan menggunakan perintah `tail -f /var/log/nginx/protocol_access.log` pada terminal. Hasil pengujian kode PF_003 menunjukkan *load balancer* mampu meneruskan *traffic* CoAP (*port* 5683) ke IoT *middleware* satu dengan alamat IP 192.168.0.101 dan IoT *middleware* dua dengan alamat IP 192.168.0.102.

E. Keterangan

Berdasarkan hasil pengujian PF_003 disimpulkan bahwa integrasi *sensor* CoAP dengan *load balancer* berhasil dilakukan.

6.2.4 *Load Balancer* mampu mendistribusikan *Traffic* CoAP dan MQTT dari *sensor* ke IoT *Middleware*

A. Tujuan Pengujian

Mengetahui apakah *Load balancer* mampu mendistribusikan *traffic* MQTT dan CoAP dari *sensor* ke IoT *middleware*.

B. Prosedur pengujian

1. Men-*generate* sepuluh paket CoAP dan MQTT.
2. Menjalankan *access.log* pada *load balancer.*
3. Melakukan pemeriksaan apakah *traffic* dapat diteruskan ke IoT *middleware* oleh *load balancer* melalui *access.log* nya.

C. Hasil yang diharapkan

Load balancer mampu mendistribusikan *traffic* CoAP dari *sensor* ke IoT *middleware*.

D. Hasil Pengujian

Load balancer mampu mendistribusikan *traffic* MQTT dan CoAP dari *sensor* ke IoT *middleware*. Gambar 6.5 menunjukkan hasil pengujian fungsional dengan kode uji PF_004.

```
192.168.4.9 [29/Nov/2018:06:24:33 +0000] 200 8 192.168.0.101:1883
192.168.4.9 [29/Nov/2018:06:24:33 +0000] 200 8 192.168.0.102:1883
192.168.4.9 [29/Nov/2018:06:24:33 +0000] 200 8 192.168.0.101:1883
192.168.4.9 [29/Nov/2018:06:24:33 +0000] 200 8 192.168.0.102:1883
192.168.4.9 [29/Nov/2018:06:24:33 +0000] 200 8 192.168.0.101:1883
192.168.4.9 [29/Nov/2018:06:24:33 +0000] 200 8 192.168.0.102:1883
192.168.4.9 [29/Nov/2018:06:24:33 +0000] 200 8 192.168.0.101:1883
192.168.4.9 [29/Nov/2018:06:24:33 +0000] 200 8 192.168.0.102:1883
192.168.4.9 [29/Nov/2018:06:24:33 +0000] 200 8 192.168.0.101:1883
192.168.4.9 [29/Nov/2018:06:24:33 +0000] 200 8 192.168.0.102:1883
192.168.4.10 [29/Nov/2018:06:24:33 +0000] 200 28 192.168.0.102:5683
192.168.4.10 [29/Nov/2018:06:24:33 +0000] 200 28 192.168.0.101:5683
192.168.4.10 [29/Nov/2018:06:24:33 +0000] 200 28 192.168.0.102:5683
192.168.4.10 [29/Nov/2018:06:24:33 +0000] 200 28 192.168.0.101:5683
192.168.4.10 [29/Nov/2018:06:24:33 +0000] 200 28 192.168.0.102:5683
192.168.4.10 [29/Nov/2018:06:24:33 +0000] 200 28 192.168.0.101:5683
192.168.4.10 [29/Nov/2018:06:24:33 +0000] 200 28 192.168.0.102:5683
192.168.4.10 [29/Nov/2018:06:24:33 +0000] 200 28 192.168.0.101:5683
192.168.4.10 [29/Nov/2018:06:24:33 +0000] 200 28 192.168.0.102:5683
192.168.4.10 [29/Nov/2018:06:24:33 +0000] 200 28 192.168.0.101:5683
```

Gambar 6.5 Hasil Pengujian Kode PF_004

Pengujian fungsional dengan kode PF_004 dilakukan dengan mengakses *file* `protocol_access.log` pada direktori `/var/log/nginx/`. Pada gambar 6.5 menunjukkan *node sensor* MQTT (*port* 1883) dan *node sensor* CoAP (*port* 5683) mengirimkan data ke IoT *middleware* melalui *load balancer*. *Load balancer* kemudian menginterupsi *traffic* dan mendistribusikannya ke IoT *middleware* tujuan. Hasil pengujian kode PF_004 menunjukkan *load balancer* mampu mendistribusikan sepuluh paket *traffic* MQTT dan CoAP (*port* 1883 dan *port* 5683) ke IoT *middleware* satu dengan alamat IP 192.168.0.101 dan IoT *middleware* dua dengan alamat IP 192.168.0.102.

E. Keterangan

Berdasarkan hasil pengujian PF_004 disimpulkan bahwa integrasi *sensor* MQTT dan *sensor* CoAP dengan *load balancer* berhasil dilakukan.

6.2.5 IoT *Middleware* dapat menerima Data dari Protokol MQTT

A. Tujuan Pengujian

IoT *Middleware* dapat menerima data dari protokol MQTT.

B. Prosedur pengujian

1. Menjalankan IoT *middleware* dan menjalankan program *pm2 logs.*
2. Melakukan pemeriksaan data pada *middleware* apakah data telah diterima dan terekam pada program *monitoring pm2 logs*.

C. Hasil yang diharapkan

IoT *Middleware* dapat menerima data dari protokol MQTT.

D. Hasil Pengujian

IoT *Middleware* dapat menerima data dari protokol MQTT. Gambar 6.6 menunjukkan hasil pengujian IoT *middleware* dengan kode uji PF_005.

```
0|qoap | 2018-12-7 15:04:53 MQTT - Client 192.168.4.10 has connected
0|qoap | 2018-12-7 15:04:53 MQTT - Client 192.168.4.10 publish a message to home/kitchen
0|qoap | 2018-12-7 15:04:53 MQTT - Client 192.168.4.10 has closed connection
```

Gambar 6.6 Hasil Pengujian Kode PF_005

Pengujian fungsional dengan kode PF_005 dilakukan dengan mengakses *logs* pada IoT *middleware* menggunakan perintah `pm2 logs`. Gambar 6.6 menunjukkan tampilan *log* pada IoT *middleware*. Hasil pengujian kode PF_005 menjelaskan IoT *middleware* mampu menerima data kelembapan dan suhu melalui protokol MQTT untuk topik *home/kitchen*.

F. Keterangan

Berdasarkan hasil pengujian PF_005, disimpulkan bahwa integrasi IoT *middleware*, *load balancer*, *dan sensor* MQTT berhasil dilakukan.

6.2.6 IoT *Middleware* dapat menerima data dari protokol CoAP

A. Tujuan Pengujian

IoT *Middleware* dapat menerima data dari protokol CoAP.

B. Prosedur pengujian

1. Menjalankan IoT *middleware* dan menjalankan program *pm2 logs.*
2. Melakukan pemeriksaan data pada *middleware* apakah data telah diterima dan terekam pada program *monitoring pm2 logs*.

C. Hasil yang diharapkan

IoT *Middleware* dapat menerima data dari protokol CoAP.

D. Hasil Pengujian

IoT *Middleware* dapat menerima data dari protokol CoAP. Gambar 6.7 menunjukkan hasil pengujian IoT *middleware* dengan kode uji PF_006.

```
0|qoap | 2018-12-5 22:49:20 COAP - Incoming POST request from 192.168.0.104 for home/garage
0|qoap | 2018-12-5 22:49:20 COAP - Incoming POST request from 192.168.0.104 for home/garage
0|qoap | 2018-12-5 22:49:20 COAP - Incoming POST request from 192.168.0.104 for home/garage
```

Gambar 6.7 Hasil Pengujian Kode PF_006

Pengujian fungsional dengan kode PF_006 dilakukan dengan mengakses *logs* pada IoT *middleware* menggunakan perintah `pm2 logs`. Gambar 6.7 menunjukkan tampilan *log* pada IoT *middleware*. Hasil pengujian kode PF_006 menjelaskan *middleware* mampu menerima data kelembapan dan suhu melalui protokol CoAP untuk topik *home/garage*.

E. Keterangan

Berdasarkan hasil pengujian PF_006, disimpulkan bahwa integrasi IoT middleware, *load balancer*, dan *sensor* CoAP berhasil dilakukan.

6.2.7 IoT *Middleware* dapat menerima data dari protokol CoAP dan MQTT

A. Tujuan Pengujian

IoT *Middleware* dapat menerima data dari protokol MQTT dan CoAP.

B. Prosedur pengujian

1. Menjalankan IoT *middleware* dan menjalankan program *pm2 logs.*
2. Melakukan pemeriksaan data pada IoT *middleware* apakah data telah diterima dan terekam pada program *monitoring pm2 logs*.

C. Hasil yang diharapkan

IoT *Middleware* dapat menerima data dari protokol MQTT dan CoAP.

D. Hasil Pengujian

IoT *Middleware* dapat menerima data dari protokol MQTT dan CoAP. Gambar 6.8 menunjukkan hasil pengujian *middleware* dengan kode uji PF_007.

```
0|qoap | 2018-12-7 18:07:15 MQTT - Client 192.168.4.10 has connected
0|qoap | 2018-12-7 18:07:15 MQTT - Client 192.168.4.10 publish a message to home/kitchen
0|qoap | 2018-12-7 18:07:16 MQTT - Client 192.168.4.10 has closed connection
0|qoap | 2018-12-7 18:07:21 COAP - Incoming POST request from 192.168.0.105 for home/garage
```

Gambar 6.8 Hasil Pengujian Kode PF_007

Pengujian fungsional dengan kode PF_007 dilakukan dengan mengakses *logs* pada IoT *middleware* menggunakan perintah `pm2 logs`. Gambar 6.8 menunjukkan tampilan *log* pada IoT *middleware*. Hasil pengujian kode PF_007 menjelaskan IoT *middleware* mampu menerima data kelembapan dan suhu melalui protokol MQTT untuk topik */home/kitchen* dan CoAP untuk topik *home/garage*.

E. Keterangan

Berdasarkan hasil pengujian PF_007, disimpulkan bahwa integrasi IoT middleware, *load balancer*, *sensor* MQTT, dan *sensor* CoAP berhasil dilakukan.

6.2.8 *Cluster* dapat mendistribusikan Data dari IoT *Middleware*

A. Tujuan Pengujian

Cluster dapat mendistribusikan data dari IoT *middleware*.

B. Prosedur pengujian

1. Melakukan pemeriksaan pada Redis tentang topik yang disimpan.

C. Hasil yang diharapkan

Cluster dapat mendistribusikan data dari IoT *middleware*.

D. Hasil Pengujian

Cluster dapat mendistribusikan data dari IoT *middleware*. Gambar 6.9 menunjukkan hasil pengujian *cluster* dengan kode uji PF_008.

```
pi@TheCluster_1:~ $ redis-cli -h 192.168.0.108 -p 6383 --scan
topic:home/kitchen
pi@TheCluster_1:~ $ redis-cli -h 192.168.0.107 -p 6381 --scan
topics
topic:home/garage
pi@TheCluster_1:~ $
```

Gambar 6.9 Hasil Pengujian Kode PF_008

Pengujian fungsional dengan kode PF_008 dilakukan dengan memeriksa topik yang tersimpan dalam *cluster* Redis. Gambar 6.9 menunjukkan tampilan dari perintah `redis-cli -h [IP] -p [port] --scan` pada terminal. Perintah tersebut berfungsi untuk melihat semua topik yang tersimpan dalam *cluster* Redis. Hasil pengujian kode PF_008 menjelaskan Redis dengan alamat IP 192.168.0.108 dan *port* 6383 menyimpan topik *home/kitchen*, sedangkan *cluster* Redis dengan alamat IP 192.168.0.107 dan *port* 6381 menyimpan topik *home/garage*. Berdasarkan hasil pengujian PF_008, disimpulkan bahwa *cluster* mampu mendistribusikan data ke beberapa Redis.

E. Keterangan

Berdasarkan hasil pengujian PF_008, disimpulkan bahwa *cluster* mampu mendistribusikan data ke dalam beberapa Redis.

6.2.9 IoT *Middleware* dapat mengirimkan data dengan protokol MQTT ke *Subscriber*

A. Tujuan Pengujian

IoT *Middleware* dapat mengirimkan data dengan protokol MQTT ke *subscriber*.

B. Prosedur pengujian

1. Menjalankan IoT *middleware* dan menjalankan program *pm2 logs.*
2. Melakukan pemeriksaan data pada IoT *middleware* apakah data telah diterima dan terekam pada program *monitoring pm2 logs*.

C. Hasil yang diharapkan

IoT *Middleware* dapat mengirimkan data dengan protokol MQTT ke *subscriber*.

D. Hasil Pengujian

IoT *Middleware* dapat mengirimkan data dengan protokol MQTT ke *subscriber*.. Gambar 6.9 menunjukkan hasil pengujian IoT *middleware* dengan kode uji PF_009.

```
0|qoap | 2019-2-3 14:18:49 MQTT - Client Subscriber has connected
0|qoap | 2019-2-3 14:18:49 MQTT - Client Subscriber subscribe to home/garage
0|qoap | 2019-2-3 14:18:50 MQTT - Client Subscriber has disconnected
0|qoap | 2019-2-3 14:18:50 MQTT - Client Subscriber has closed connection


0|qoap | 2019-2-3 14:18:51 MQTT - Client Subscriber has connected
0|qoap | 2019-2-3 14:18:51 MQTT - Client Subscriber subscribe to home/kitchen
0|qoap | 2019-2-3 14:18:52 MQTT - Client Subscriber has disconnected
0|qoap | 2019-2-3 14:18:52 MQTT - Client Subscriber has closed connection
```

Gambar 6.10 Hasil Pengujian Kode PF_009

Pengujian fungsional dengan kode PF_009 dilakukan dengan mengakses *logs* pada IoT *middleware* menggunakan perintah `pm2 logs`. Gambar 6.10 menunjukkan tampilan *log* pada IoT *middleware*. Hasil pengujian kode PF_009 menjelaskan IoT *middleware* mampu mengirimkan data kelembapan dan suhu melalui protokol MQTT untuk topik */home/kitchen* dan CoAP untuk topik *home/garage*.

E. Keterangan

Berdasarkan hasil pengujian PF_009, disimpulkan bahwa integrasi IoT *middleware*, *cluster* Redis dan *subscriber* berhasil dilakukan.

6.2.10 Mengetahui semua *key* Redis dapat terhubung satu sama lain

A. Tujuan Pengujian

Mengetahui semua *key* Redis dapat terhubung satu sama lain.

B. Prosedur pengujian

1. Memeriksa status pada salah satu Redis.
2. Mengakses data dengan topik yang disimpan di Redis lain.

C. Hasil yang diharapkan

Mengetahui semua *key* Redis dapat terhubung satu sama lain.

D. Hasil Pengujian

Mengetahui semua *key* Redis dapat terhubung satu sama lain. Gambar 6.11 menunjukkan hasil pengujian *key* Redis dengan kode uji PF_010.

```
pi@TheCluster_1:~ $ redis-cli -h 192.168.0.108 -p 6383 -c
192.168.0.108:6383> mget topic:home/garage
-> Redirected to slot [5711] located at 192.168.57.107:6381
1)
"{\"type\":\"Buffer\",\"data\":[123,34,112,114,111,116,111,99,111,108,3
4,58,32,34,99,111,97,112,34,44,32,34,116,101,109,112,101,114,97,116,117
,114,101,34,58,32,34,51,48,32,99,101,108,99,105,117,115,34,44,32,34,116
,105,109,101,115,116,97,109,112,34,58,32,49,53,53,48,51,49,55,56,48,56,
46,50,57,57,48,49,49,44,32,34,104,117,109,105,100,105,116,121,34,58,32,
34,50,49,32,37,34,44,32,34,116,111,112,105,99,34,58,32,34,104,111,109,1
01,47,103,97,114,97,103,101,34,44,32,34,115,101,110,115,111,114,34,58,3
2,123,34,105,110,100,101,120,34,58,32,34,99,111,109,113,116,116,34,44,3
2,34,116,105,112,101,34,58,32,34,101,115,112,56,50,54,54,34,44,32,34,10
9,111,100,117,108,101,34,58,32,34,100,104,116,49,49,34,44,32,34,105,112
,34,58,32,34,49,57,50,46,49,54,56,46,52,46,57,34,125,125]}"
192.168.57.107:6381>
pi@TheCluster_1:~ $ redis-cli -h 192.168.0.107 -p 6381 -c
192.168.0.107:6381> mget topic:home/kitchen
-> Redirected to slot [14886] located at 192.168.57.108:6383
1)
"{\"type\":\"Buffer\",\"data\":[123,34,112,114,111,116,111,99,111,108,3
4,58,32,34,99,111,97,112,34,44,32,34,116,101,109,112,101,114,97,116,117
,114,101,34,58,32,34,51,48,32,99,101,108,99,105,117,115,34,44,32,34,116
,105,109,101,115,116,97,109,112,34,58,32,49,53,52,57,49,57,52,50,53,48,
46,49,57,50,51,49,53,44,32,34,104,117,109,105,100,105,116,121,34,58,32,
34,50,49,32,37,34,44,32,34,116,111,112,105,99,34,58,32,34,104,111,109,1
01,47,103,97,114,97,103,101,34,44,32,34,115,101,110,115,111,114,34,58,3
2,123,34,105,110,100,101,120,34,58,32,34,99,111,109,113,116,116,34,44,3
2,34,116,105,112,101,34,58,32,34,101,115,112,56,50,54,54,34,44,32,34,10
9,111,100,117,108,101,34,58,32,34,100,104,116,49,49,34,44,32,34,105,112
,34,58,32,34,49,57,50,46,49,54,56,46,52,46,57,34,125,125]}"
192.168.57.108:6383>
```

Gambar 6.11 Hasil Pengujian Skenario PF_010

Gambar 6.11 menunjukkan tampilan hasil akses menggunakan perintah `redis-cli -h [IP] -p [port] -c` . Perintah `mget [key topik]` adalah untuk mengakses topik yang tersimpan pada Redis, dan menunjuk ke *node* yang menyimpan topik tersebut. Hal tersebut dapat dilihat ketika Redis yang terdapat pada *port* 6383 dan alamat IP 192.168.0.108 ingin mendapatkan topik *home/garage*. Redis tersebut memberitahukan bahwa topik *home/garage* berada pada *node* dengan *port* 6381 dan alamat IP 192.168.0.107 dan mengalihkan *pointer* ke tempat topik yang ingin didapatkan. Begitu juga ketika Redis dengan *port* 6379 dan alamat IP 192.168.0.101 ingin mendapatkan topik *home/kitchen*, maka diberitahukan bahwa topik yang diinginkan berada pada Redis dengan *port* 6383 dan alamat IP 192.168.0.108, dan mengalihkan *pointer* ke *node* tersebut.

E. Keterangan

Berdasarkan hasil pengujian PF_010, disimpulkan bahwa semua *key* Redis dapat terhubung satu sama lain.

6.2 Pengujian Non-Fungsional

Pengujian non-fungsional dilakukan agar batasan sistem dapat diketahui oleh peneiliti. Adapun pengujian non-fungsional pada penelitian ini adalah untuk menguji skalabilitas sistem.

6.2.1 Pengujian Skalabilitas

Pengujian skalabilitas bertujuan mengetahui kemampuan IoT *middleware* dalam menangani sejumlah proses yang diterima. Pengujian skalabilitas dilakukan untuk mengukur elastisitas sumber daya yang tersedia dengan kebutuhan yang diinginkan. IoT *Middleware* pada pengujian skalabilitas akan dikenakan proses *publish* dengan jumlah *publisher* yang bervariasi. Jumlah *publisher* akan bertambah seiring interval yang diberikan. Pada skenario pengujian yang akan dijalankan, terdapat beberapa parameter yang akan diuji. Parameter tersebut adalah *time publish* dan *concurrent publish*. Untuk menghitung *time publish* digunakan *tool tcpdump* ketika melakukan *packet sniffing* dan *tool wireshark* untuk melakukan analisis pada paket yang telah ditangkap. Skenario Pengujian skalabilitas terbagi menjadi dua skenario. Skenario pertama merupakan skenario yang melibatkan peran *load balancer* pada *publisher.* Skenario kedua merupakan skenario yang melibatkan peran *load balancer* pada *subscriber*.

Skenario pertama, pada sisi *publisher* men-*generate client* dan data yang akan di-*publish* sebesar 100, 500, 1000, dan 1500 baik untuk *publisher* MQTT atau CoAP. Untuk mencapai hal tersebut digunakan program yang berjalan secara *asynchronous*. *Client publisher* yang telah dibuat, mengirimkan koneksi dan datanya satu per satu untuk setiap protokol yang digunakan (berurut sesuai protokol). Selanjutnya mengambil data paket yang telah direkam pada IoT *middleware*.

Skenario kedua, pada sisi *subscriber* men-*generate client subscriber* dan data masing-masing sebesar 100, 500, 1000, dan 1500 yang berkomunikasi menggunakan protokol MQTT. Program yang digunakan, juga berjalan secara *asynchronous*. Pada skenario ini *subscriber* akan mengirimkan sinyal CON ke IoT *middleware*. Kemudian IoT *middleware* akan membalasnya dengan sinyal ACK. Setelah itu, *subscriber* akan mengirimkan *request* topik yang diinginkan, dan IoT *middleware* akan mengirimkan data dari topik yang di-*subscribe*. Topik yang akan di-*subscribe* pada skenario ini adalah */home/kitchen*.

Nilai *time* merupakan waktu yang diperlukan untuk menyelesaikan proses *publish* dan *subscribe* pada sejumlah *publisher* dan *subscriber*. Nilai *time* pada protokol CoAP dihitung ketika nilai QoS CON dikirimkan hingga penerima mengirimkan paket ACK. Sedangkan pada protokol MQTT menggunakan QoS level 1, dimana pesan akan dikirimkan setidaknya satu kali (kemungkinan terjadi duplikasi) hingga penerima mengirimkan ACK. Nilai *time* untuk protokol MQTT dihitung dari nilai waktu *publish* ACK terakhir dikurangi nilai waktu *publish message* pertama. Nilai *time* untuk *subscriber* dihitung ketika *client* mengirimkan pesan ACK ke IoT *middleware* hingga, IoT *middleware* mengirimkan data dari topik yang di-*subscribe*.

Nilai *concurrent* adalah banyaknya pesan yang mampu ditangani IoT *middleware* dalam satu detik. Nilai *concurrent* pada IoT *middleware* akan dihitung ketika telah mendapatkan nilai *time publish* dan *time subscribe* dari masing-masing protokol pengiriman data. Protokol pengiriman data yang digunakan adalah CoAP dan MQTT.

Pengujian skalabilitas akan membandingkan, nilai *time publish*, *time subscribe*, *concurrent publish*, dan *concurrent subscribe* pada penelitian sebelumnya. Pada penelitian sebelumnya tidak terdapat *load balancer* yang akan membagi *traffic* ke beberapa IoT *middleware*. Poin skenario pengujian adalah menjalankan *publisher* dan *subscriber* secara *asynchronous* dengan jumlah 100, 500, 1000, dan 1500 *client*.

6.2.1.1 Pengujian *Time Publish*

A. Tujuan Pengujian

Mengetahui berapa lama waktu yang dibutuhkan IoT *middleware* untuk menangani sejumlah *publisher* dengan protokol MQTT dan CoAP melalui *load balancer.*

B. Prosedur pengujian

1. Menjalankan *pm2 logs* pada IoT *middleware* untuk memantau pesan pada *middleware*.
2. Filter paket dijalankan pada IoT *middleware* menggunakan tcpdump.
3. Analisis paket hasil tangkapan dari tcpdump untuk menentukan nilai *time publish.*

C. Hasil yang diharapkan

Mengetahui berapa lama waktu yang dibutuhkan IoT *middleware* untuk menangani sejumlah *publisher* dengan protokol MQTT dan CoAP melalui *load balancer.*

D. Hasil Pengujian

Hasil Pengujian diijelaskan pada poin 6.2.2.

6.2.1.2 Pengujian *Concurrent Publish*

A. Tujuan Pengujian

Mengetahui berapa banyak jumlah pesan *publish* yang mampu ditangani IoT *middleware* dalam satu detik.

B. Prosedur pengujian

1. Menjalankan *pm2 logs* pada IoT *middleware* untuk memantau pesan pada IoT *middleware*.
2. Filter paket dijalankan pada IoT *middleware* menggunakan tcpdump.
3. Analisis paket hasil tangkapan dari tcpdump untuk menentukan nilai *concurrent publish.*

C. Hasil yang diharapkan

Mengetahui berapa banyak jumlah pesan *publish* yang mampu ditangani IoT *middleware* dalam satu detik.

D. Hasil Pengujian

Hasil Pengujian diijelaskan pada poin 6.2.2.

6.2.1.3 Pengujian *Time Subscribe*

A. Tujuan Pengujian

Mengetahui berapa lama waktu yang dibutuhkan IoT *middleware* untuk menangani sejumlah *subscriber* dengan protokol MQTT melalui *load balancer*.

B. Prosedur pengujian

1. Menjalankan *pm2 logs* pada IoT *middleware* untuk memantau pesan pada IoT *middleware*.
2. Filter paket dijalankan pada *middleware* menggunakan tcpdump.
3. Analisis paket hasil tangkapan dari tcpdump untuk menentukan nilai *time subscribe.*

C. Hasil yang diharapkan

Mengetahui berapa lama waktu yang dibutuhkan IoT *middleware* untuk menangani sejumlah *subscriber* dengan protokol MQTT melalui *load balancer.*

D. Hasil Pengujian

Hasil Pengujian diijelaskan pada poin 6.2.2.

6.2.1.4 Pengujian *Concurrent Subscribe*

A. Tujuan Pengujian

Mengetahui berapa banyak jumlah pesan *publish* yang mampu ditangani IoT *middleware* dalam satu detik.

B. Prosedur pengujian

1. Menjalankan *pm2 logs* pada *middleware* untuk memantau pesan pada IoT *middleware*.
2. Filter paket dijalankan pada IoT *middleware* menggunakan tcpdump.
3. Analisis paket hasil tangkapan dari tcpdump untuk menentukan nilai *time publish.*

C. Hasil yang diharapkan

Mengetahui berapa banyak jumlah pesan *publish* yang mampu ditangani IoT *middleware* dalam satu detik.

D. Hasil Pengujian

Hasil Pengujian diijelaskan pada poin 6.2.2.

6.2.2 Hasil Pengujian Skalabilitas

Pengujian skalabilitas dilakukan pada sistem yang tidak menggunakan *load balancer* dan yang menggunakan *load balancer*. Pengujian skalabilitas

menggunakan parameter *time publish*, *time subscribe*, *concurrent publish*, dan *concurrent subscribe*.

6.2.2.1 Penelitian Sebelumnya tanpa *Load Balancer*

Pengujian dilakukan mengacu pada kode uji PNF_001, PNF_002, PNF_003, dan PNF_004. Tahapan pengujian ini terdapat empat parameter yang diukur, yakni *time publish*, *concurrent publish*, *time subscribe* dan *concurrent subscribe*.

a. Pengujian *Time Publish*

Pengujian *time publish* dilakukan berdasarkan kode uji PNF_001. Tahapan ini akan dilakukan perhitungan *time publish* pada proses *publish* ke IoT *middleware* menggunakan protokol CoAP dan MQTT. Skenario pengujian yang digunakan dengan men-*generate* sejumlah klien yang ditentukan, pada sisi *publisher.* Pengujian dilakukan sebanyak lima iterasi. Tangkapan paket kemudian dianalisis menggunakan *tool wireshark*. Tabel 6.1 menjelaskan hasil pengujian *time publish* protokol CoAP tanpa *load balancer*. Tabel 6.2 menunjukkan hasil pengujian *time publish* protokol MQTT tanpa *load balancer*.

Tabel 6.1 Hasil Pengujian *Time Publish* CoAP

Jumlah *Publisher*	**Percobaan**				
	1 *Time* (s)	**2 *Time* (s)**	**3 *Time* (s)**	**4 *Time* (s)**	**5 *Time* (s)**
100	1.55	1.63	1.7	1.54	1.56
500	7.88	7.93	7.97	7.99	7.9
1000	16.06	16.12	17.84	16.03	15.79
1500	24.11	23.98	24.03	24.28	24.27

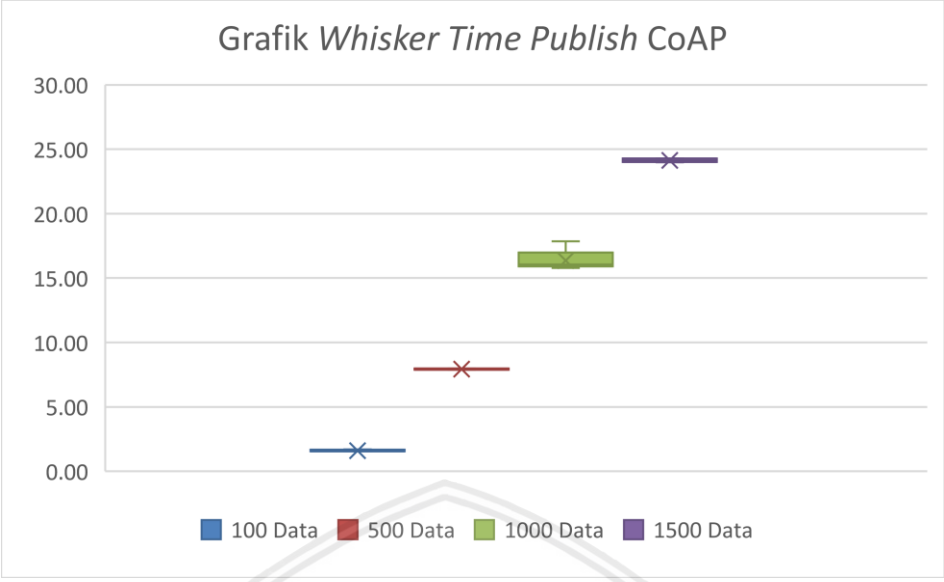
Hasil detail pengujian *time publish* CoAP terdapat pada lampiran A.

Tabel 6.2 Hasil Pengujian *Time Publish* MQTT

Jumlah *Publisher*	**Percobaan**				
	1 *Time* (s)	**2 *Time* (s)**	**3 *Time* (s)**	**4 *Time* (s)**	**5 *Time* (s)**
100	2.6	2.5	3.18	2.44	2.54
500	12.95	13.05	12.71	12.31	12.24
1000	24.61	24.61	22.92	24.73	23.16
1500	35.59	31.56	33.72	36.23	38.76

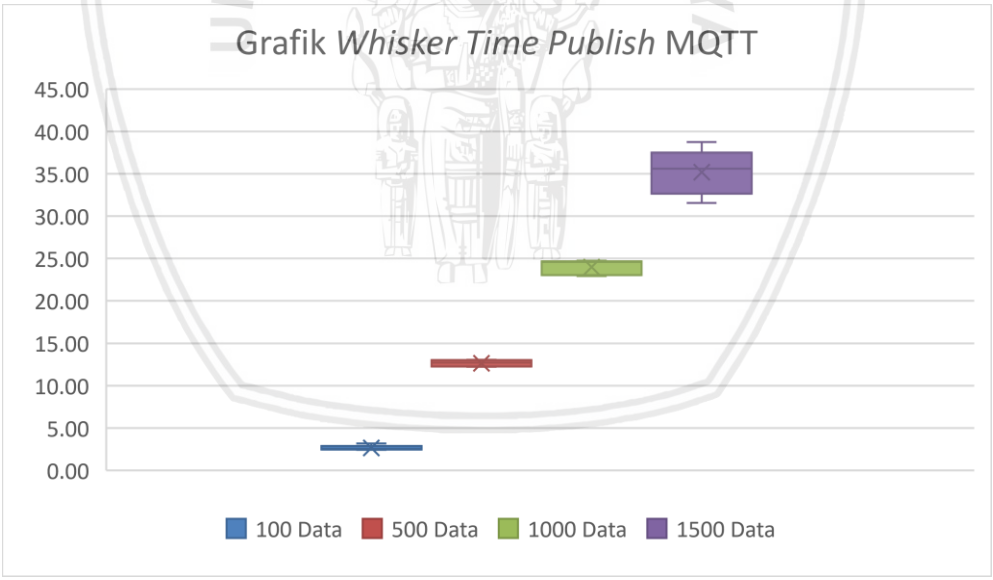
Hasil detail pengujian *time publish* MQTT terdapat pada lampiran B.

Berdasarkan tabel 6.1 dan tabel 6.2 diperoleh grafik diagram *whisker* seperti yang ditunjukkan oleh gambar 6.12 dan 6.13. Gambar 6.12 menunjukkan grafik *whisker time publish* protokol CoAP tanpa *load balancer*. Sedangkan gambar 6.13 menunjukkan grafik *whisker time publish* protokol MQTT tanpa *load balancer*.



Gambar 6.12 Grafik *Whisker Time Publish* Protokol CoAP

Grafik *whisker* pada gambar 6.12 menunjukkan distribusi data *time publish* dengan sedikit rentang variasi. Pada lima kali percobaan pengujian, dihasilkan area sebaran yang kecil. Hal tersebut menunjukkan pada lima kali percobaan pengujian hasil *time publish* yang diperoleh relatif stabil untuk protokol CoAP.



Gambar 6.13 Grafik *Whisker Time Publish* Protokol MQTT

Pada gambar 6.13 terdapat kenaikan jumlah variasi *time publish* MQTT yang signifikan untuk jumlah *client* 1000 dan 1500. Sedangkan untuk jumlah *client* 100 dan 500 area sebarannya relatif sama. Hal tersebut menunjukkan, pada titik tertentu, jumlah *client* yang banyak, jika menggunakan protokol MQTT memerlukan *time publish* yang relatif lama.

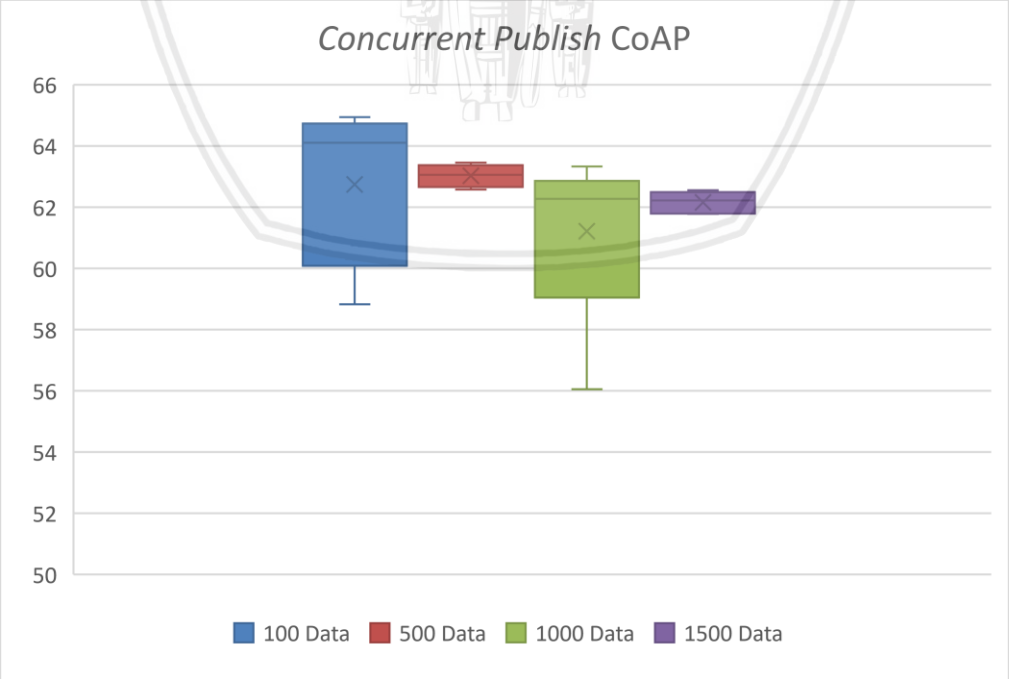
b. Pengujian *Concurrent Publish*

Pengujian *concurrent publish* dilakukan berdasarkan kode uji PNF_002. Perhitungan dilakukan untuk mengetahui pesan yang dapat di-*publish* per detik

pada setiap variasi *publisher* yang diberikan. Hasil *concurrent publish* diperoleh dari jumlah *client publisher* dibagi rata-rata *time publish* pada tabel 6.1 dan tabel 6.2 untuk setiap tingkatan *client*. Tabel 6.3 menunjukkan hasil pengujian pada parameter *concurrent publish*. Gambar 6.14 menunjukkan grafik *whisker concurrent publish* CoAP. Gambar 6.15 menunjukkan grafik *whisker concurrent publish* MQTT.

Tabel 6.3 Hasil Pengujian *Concurrent Publish* CoAP dan MQTT

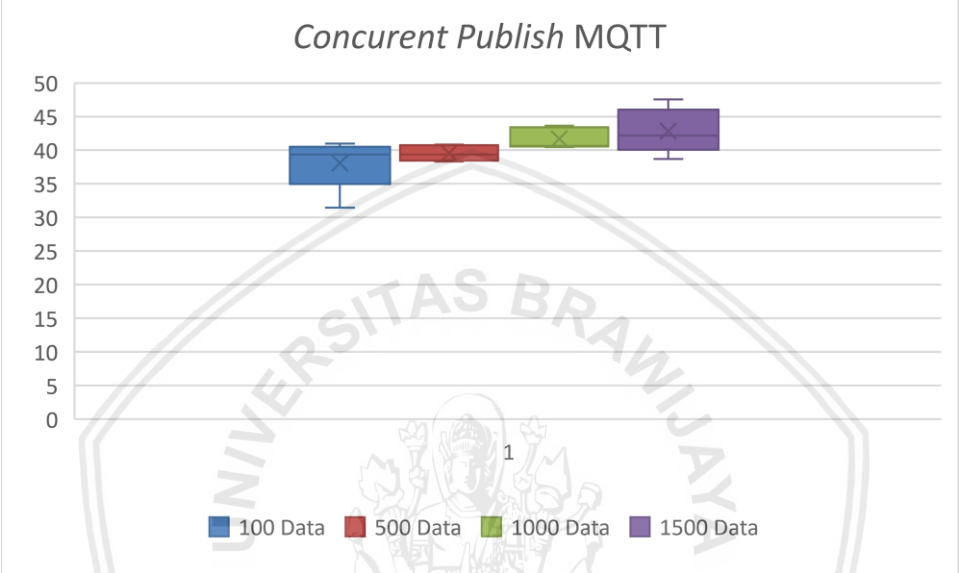
Jumlah Publisher	Percobaan									
	Concurrent (*message/s*)									
	1		2		3		4		5	
	CoAP	MQTT	CoAP	MQTT	CoAP	MQTT	CoAP	MQTT	CoAP	MQTT
100	65	38	61	40	59	31	65	41	64	39
500	63	39	63	38	63	39	63	41	63	41
1000	62	41	62	41	56	44	63	40	63	43
1500	62	42	63	48	62	44	62	41	62	39



Gambar 6.14 Grafik *Whisker Concurrent Publish* Protokol CoAP

Gambar 6.14 menunjukkan hasil *concurrent publish* menggunakan protokol CoAP. Hasil menunjukkan dari lima kali percobaan untuk variasi 100 *client*

diperoleh jumlah *concurrent* 65 pesan/detik, 61 pesan/detik, 59 pesan/detik, 65 pesan/detik dan 64 pesan/detik. Hasil *concurrent* untuk variasi 500 *client* adalah 63 pesan/detik. Perolehan *concurrent publish* untuk variasi 1000 *client* adalah 62 pesan/detik, 62 pesan/detik, 56 pesan/detik, percobaan ke-empat dank ke-lima adalah 63 pesan/detik. Sedangkan untuk variasi 1500 *client*, pada percobaan ke-dua diperoleh hasil 63 pesan/detik, pada sisa percobaan lainnya diperoleh hasil 62 pesan/detik. Jika dilihat, maka rentang data maksimal adalah 9 angka.



Gambar 6.15 Grafik *Whisker Concurrent Publish* Protokol MQTT

Gambar 6.15 menunjukkan kenaikan nilai sebaran *concurrent publish* MQTT pada setiap jumlah variasi *client*.

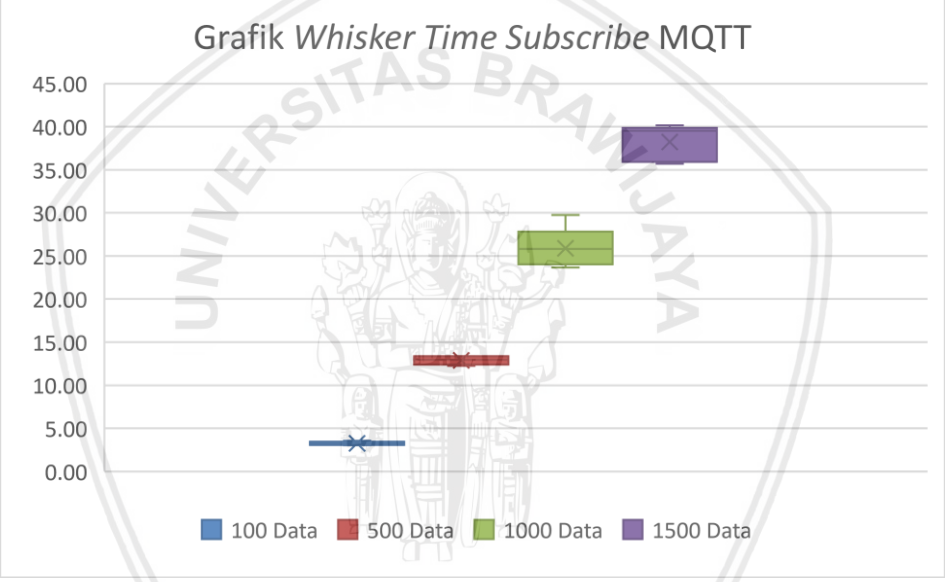
c. Pengujian *Time Subscribe*

Pengujian *time subscribe* dilakukan berdasarkan kode uji PNF_003. Tahapan ini akan dilakukan perhitungan *time subscribe* pada proses *subscribe* ke IoT *middleware* menggunakan protokol MQTT. Skenario pengujian yang digunakan dengan men-*generate* sejumlah klien yang ditentukan, pada sisi *subscriber.* Pengujian dilakukan sebanyak lima iterasi. Tangkapan paket kemudian dianalisis menggunakan *tool wireshark*. Tabel 6.3 menjelaskan hasil pengujian *time subscribe* protokol MQTT tanpa *load balancer*. Gambar 6.16 menunjukkan grafik *whisker* pengujian *time subscribe* tanpa *load balancer*.

Tabel 6.4 Hasil Pengujian *Time Subscribe*

Jumlah *Publisher*	Percobaan				
	1 *Time* (s)	2 *Time* (s)	3 *Time* (s)	4 *Time* (s)	5 *Time* (s)
100	3.04	3.29	3.13	3.2	3.57
500	13.33	13.38	12.97	12.26	12.51
1000	29.75	25.9	25.8	24.37	23.67
1500	39.54	36.13	40.14	35.71	39.54

Hasil detail pengujian *time subscribe* terdapat pada lampiran C.



Gambar 6.16 Grafik *Whisker Time Subscribe*

Berdasarkan gambar 6.16 jumlah sebaran *time subscribe* lebih bervariasi untuk jumlah *client* 1000 dan 1500. Hasil pada gambar tersebut serupa dengan hasil yang ditunjukkan oleh gambar 6.13. Berdasarkan hasil tersebut, pada jumlah *client* tertentu, jika menggunakan protokol MQTT memerlukan *time subscribe* yang relatif lebih lama.

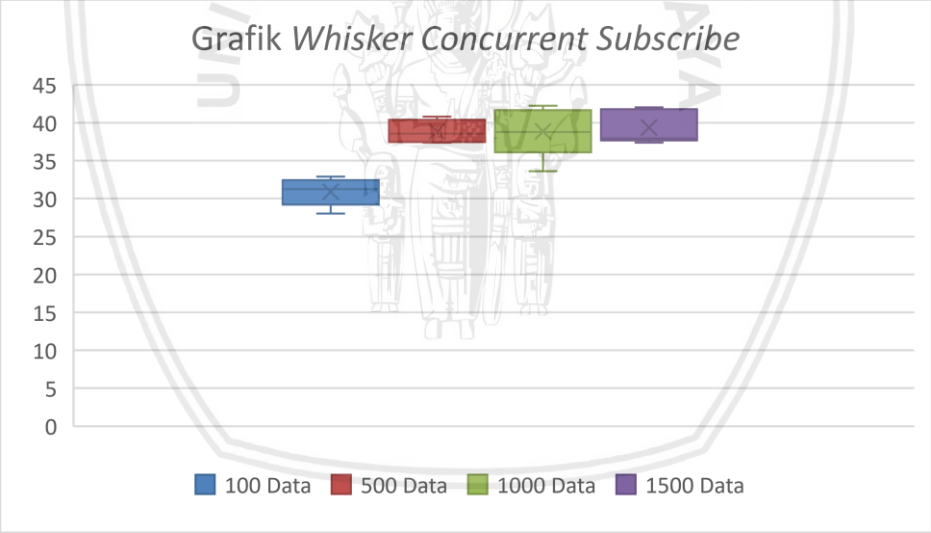
d. Pengujian *Concurrent Subscribe*

Pengujian *concurrent subscribe* dilakukan berdasarkan kode uji PNF_004. Perhitungan dilakukan untuk mengetahui pesan yang dapat di-*subscribe* per detik pada setiap variasi *subscriber* yang diberikan. Tabel 6.4 menunjukkan hasil pengujian pada parameter *concurrent subscribe*.

Tabel 6.5 Hasil Pengujian *Concurrent Subscribe*

Jumlah Subscriber	Percobaan				
	Concurrent (*message/s*)				
	1	1	1	1	1
100	33	30	32	31	28
500	38	37	39	41	40
1000	34	39	39	41	42
1500	38	42	37	42	38

Berdasarkan tabel 6.5, diperoleh hasil grafik *whisker concurrent subscribe* tanpa *load balancer* pada gambar 6.17.



Gambar 6.17 Grafik *Whisker Concurrent Subscribe*

Gambar 6.17 menunjukkan nilai *concurrent subscribe* terkecil berada pada nilai 28 pesan/detik. Sedangkan untuk nilai *concurrent subscribe* terbesar berada pada nilai 42 pesan/detik.

6.2.2.2 Penelitian Pada Sistem dengan *Load Balancer*

Skenario pengujian sama dengan skenario pengujian sistem tanpa *load balancer*. Pengujian dilakukan mengacu pada kode uji PNF_001, PNF_002, PNF_003, dan PNF_004. Parameter yang akan diuji sama dengan parameter pengujian pada tanpa, yakni *time publish*, *concurrent publish*, *time subscribe* dan *concurrent subscribe*.

a. Pengujian *Time Publish* dengan *Load Balancer*

Pengujian dilakukan berdasarkan skenario PNF_001. Pada tahap ini akan dilakukan perhitungan *time publish* saat proses *publish* ke IoT *middleware* melalui *load balancer*. *Load balancer* menggunakan algoritme *round robin* ketika melakukan ditribusi *traffic* ke IoT *middleware* satu dan IoT *middleware* dua. Oleh karena itu setiap IoT *middleware* akan menerima *traffic* sebesar *n*/2. Tabel 6.6 menunjukkan hasil *time publish* CoAP pada IoT *middleware* satu. Tabel 6.7 menunjukkan hasil *time publish* MQTT pada IoT *middleware* satu. Tabel 6.8 menunjukkan hasil *time publish* CoAP pada IoT *middleware* dua. Tabel 6.9 menunjukkan hasil *time publish* MQTT pada IoT *middleware* dua.

Tabel 6.6 Hasil Pengujian *Time Publish* CoAP pada IoT *Middleware* Satu

Jumlah *Publisher*	**Percobaan**				
	1 *Time* (s)	**2 *Time* (s)**	**3 *Time* (s)**	**4 *Time* (s)**	**5 *Time* (s)**
50	1.54	2.85	1.56	1.53	1.54
250	7.90	7.88	7.86	7.84	7.90
500	15.70	15.79	15.92	15.75	15.71
750	23.82	23.76	23.80	23.88	23.56

Penggunaan *load balancer* dengan algoritme *round robin* pada sistem memungkinkan distribusi *traffic* dengan jumlah *n*/2. Hasil tersebut dapat dilihat pada tabel 6.6. Pada skenario pengujian digunakan jumlah variasi *client* 100, 500, 1000 dan 1500. Dengan prinsip *round robing load balancing*, penanganan *request* dari *client* dipecah menjadi 50, 250, 500, dan 750. Tabel 6.6 adalah hasil *time publish* CoAP dengan prinsip tersebut pada IoT *middleware* satu. Sedangkan untuk tabel 6.7 adalah hasil *time publish* MQTT pada IoT *middleware* satu menggunakan prinsip *round robin load balancing*. Hasil detail perolehan *time publish* CoAP di IoT *middleware* satu pada sistem dengan *load balancer* terdapat pada lampiran A. sedangkan hasil detail perolehan *time publish* MQTT di IoT *middleware* satu pada sistem dengan *load balancer* terdapat pada lampiran B.

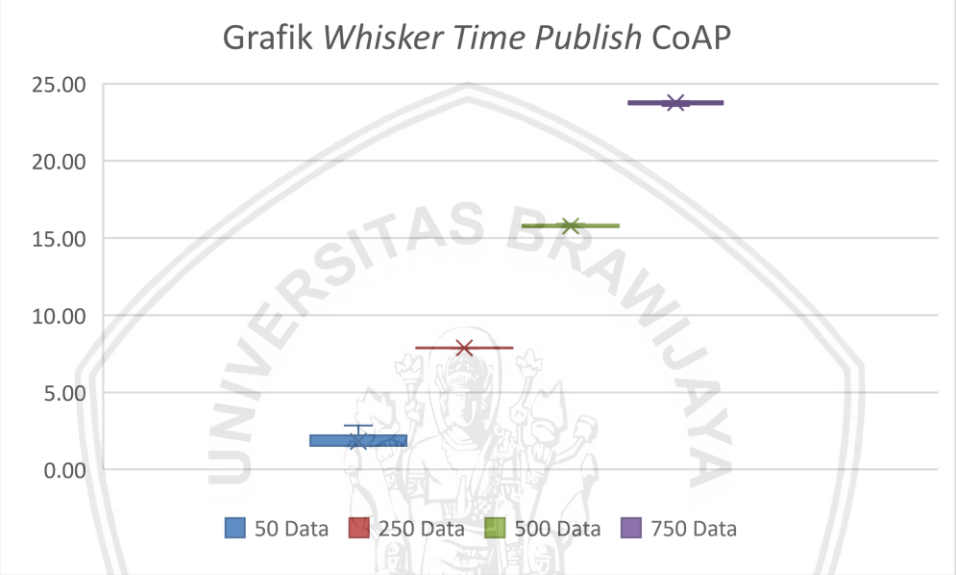
Tabel 6.7 Hasil Pengujian *Time Publish* MQTT pada IoT *Middleware* Satu

Jumlah *Publisher*	**Percobaan**				
	1 *Time* (s)	**2 *Time* (s)**	**3 *Time* (s)**	**4 *Time* (s)**	**5 *Time* (s)**
50	1.30	1.16	1.63	1.06	1.08
250	7.41	7.43	6.76	7.12	7.31

Tabel 6.7 Hasil Pengujian *Time Publish* MQTT pada IoT *Middleware* Satu (lanjutan)

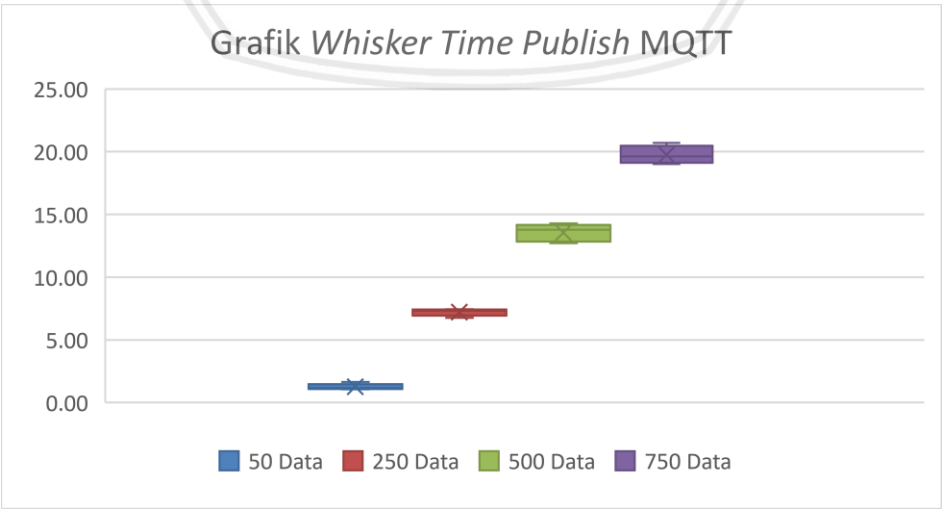
500	14.04	12.69	14.29	12.99	13.77
750	19.23	20.71	19.00	19.62	20.24

Berdasarkan tabel 6.6 dan tabel 6.7 akan diperoleh grafik *whisker time publish* untuk protokol CoAP dan MQTT. Gambar 6.18 menunjukkan grafik *whisker time publish* protokol CoAP pada IoT *middleware* satu. Gambar 6.19 menunjukkan grafik *whisker time publish* protokol MQTT pada IoT *middleware* satu.



Gambar 6.18 *Whisker Time Publish* CoAP dengan LB di IoT *Middleware* Satu

Gambar 6.18 menunjukkan *time publish* untuk jumlah *n*/2 *client* pada protokol CoAP relatif stabil. Hal tersebut ditunjukkan oleh area sebaran pada grafik *whisker* yang tidak terlalu luas.



Gambar 6.19 *Whisker Time Publish* MQTT dengan LB di IoT *Middleware* Satu

Gambar 6.19 menunjukkan *time publish* untuk protokol MQTT. Berdasarkan gambar diketahui luas area sebaran untuk jumlah *client* 50 dan 250

besarnya relatif sama. Sedangkan luas area yang relatif sama juga ditunjukkan untuk *client* 500 dan 750.

Tabel 6.8 Hasil Pengujian *Time Publish* CoAP pada IoT *Middleware* Dua

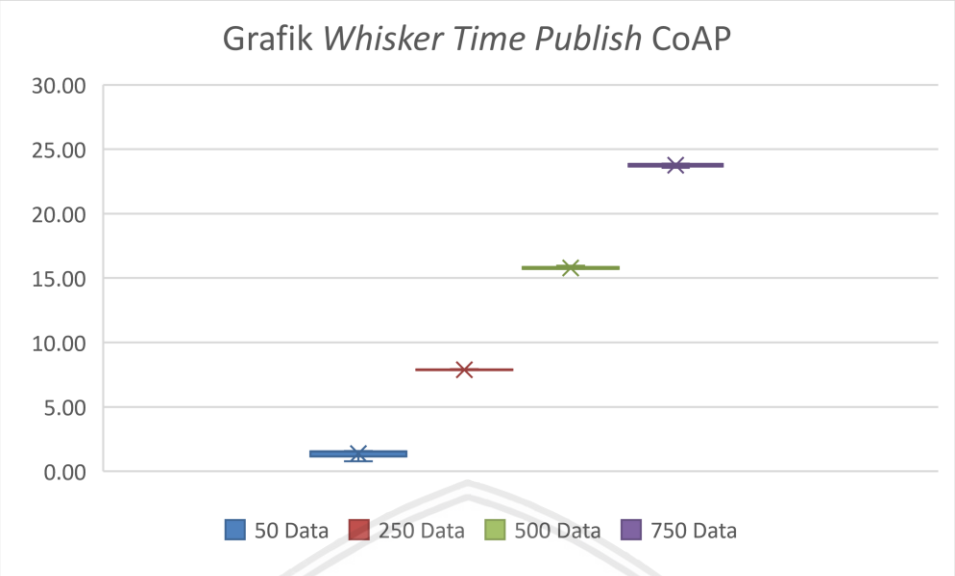
Jumlah *Publisher*	Percobaan				
	1 *Time* (s)	2 *Time* (s)	3 *Time* (s)	4 *Time* (s)	5 *Time* (s)
50	1.53	0.78	1.57	1.53	1.52
250	7.93	7.90	7.88	7.85	7.87
500	15.72	15.77	15.95	15.77	15.72
750	23.82	23.77	23.81	23.88	23.56

Tabel 6.8 menunjukkan hasil perolehan *time publish* CoAP pada IoT *middleware* dua menggunakan *tool tcpdump*. Angka 50, 250, 500 dan 750 adalah jumlah *client* yang masuk pada IoT *middleware* setelah melalui *load balancer*. Percobaan dilakukan sebanyak lima kali. Hasil detail perolehan *time publish* CoAP di IoT *middleware* dua pada sistem dengan *load balancer* terdapat pada lampiran A.

Tabel 6.9 Hasil Pengujian *Time Publish* MQTT pada IoT *Middleware* Dua

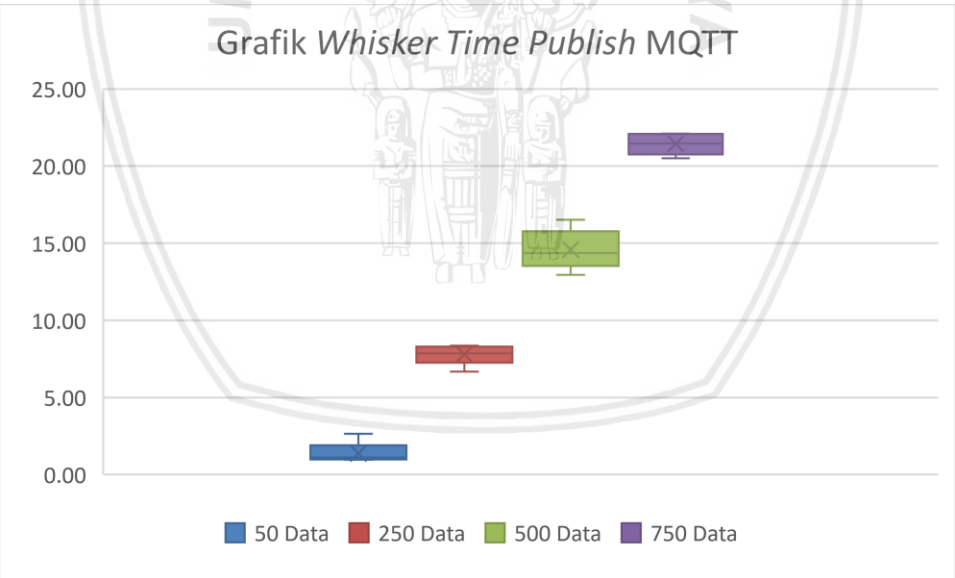
Jumlah *Publisher*	Percobaan				
	1 *Time* (s)	2 *Time* (s)	3 *Time* (s)	4 *Time* (s)	5 *Time* (s)
50	1.27	1.00	0.97	1.10	1.16
250	7.82	7.86	8.35	8.23	6.67
500	16.52	14.13	15.02	14.36	12.94
750	22.11	21.46	21.01	20.50	22.07

Tabel 6.9 menunjukkan hasil perolehan *time publish* MQTT pada IoT *middleware* dua menggunakan *tool tcpdump*. Angka 50, 250, 500 dan 750 adalah jumlah *client* yang masuk pada IoT *middleware* setelah melalui *load balancer*. Percobaan dilakukan sebanyak lima kali. Hasil detail perolehan *time publish* MQTT di IoT *middleware* dua pada sistem dengan *load balancer* terdapat pada lampiran B.



Gambar 6.20 *Whisker Time Publish* CoAP dengan LB di IoT *Middleware* Dua

Luas area pada grafik *whisker* gambar 6.20 menunjukkan perolehan *time publish* CoAP pada IoT *middleware* dua relatif stabil. Sedangkan distribusi nilai *time publish* MQTT pada IoT *middleware* dua pada jumlah *client* 500 menunjukkan sebaran data yang lebih variatif.



Gambar 6.21 *Whisker Time Publish* MQTT dengan LB di IoT *Middleware* Kedua

Penerapan *load balancer* pada sistem, memungkinkan pemrosesan *request* pada waktu yang hampir bersamaan. Hal tersebut dikarenakan sistem dengan IoT *middleware* tunggal harus memproses sejumlah *client* pada satu IoT *middleware*. Sedangkan, sistem dengan dua IoT *middleware* memproses sejumlah *client* yang sama dengan bantuan distribusi oleh *load balancer* pada kedua IoT *middleware* tersebut. Oleh karena itu, diperoleh hasil rata-rata *time publish* pada sistem yang menggunakan *load balancer*. Tabel 6.10 dan 6.11 menunjukkan rata-rata *time publish* dengan *load balancer* pada protokol CoAP dan MQTT.

Tabel 6.10 Rata-rata *Time Publish* CoAP dengan *Load Balancer*

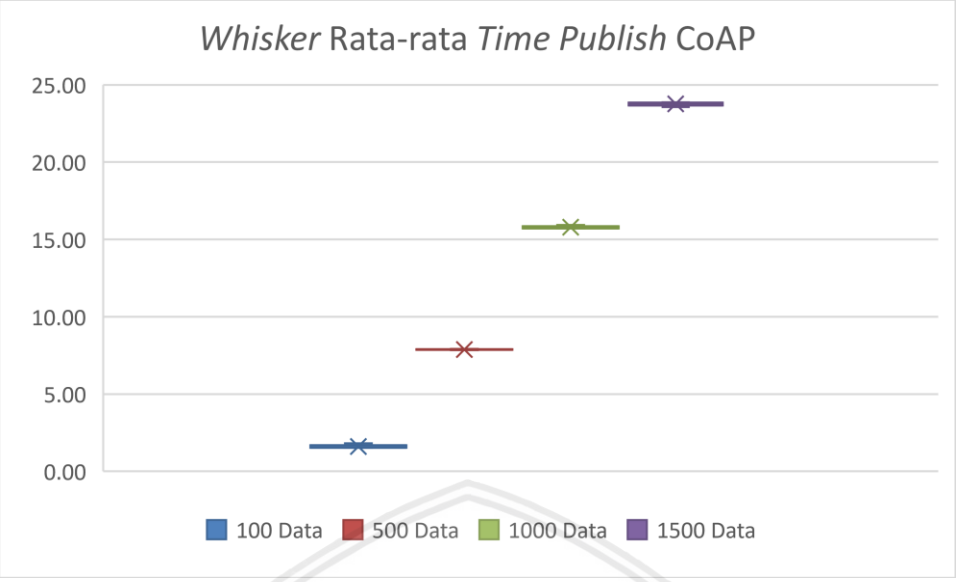
Jumlah *Publisher*	Percobaan				
	1 *Time* (s)	2 *Time* (s)	3 *Time* (s)	4 *Time* (s)	5 *Time* (s)
100	1.54	1.82	1.57	1.53	1.53
500	7.92	7.89	7.87	7.85	7.89
1000	15.71	15.78	15.94	15.76	15.72
1500	23.82	23.77	23.81	23.88	23.56

Dari tabel 6.8 dan 6.9 dicari nilai rata-rata untuk *time publish* CoAP dan MQTT. Nilai rata-rata didapatkan dengan menjumlahkan hasil *time publish* dari IoT *middleware* satu dan dua pada masing-masing variasi *client* di setiap percobaan dibagi dua. Tabel 6.10 dan 6.11 adalah perolehan rata-rata waktu *time publish* pada sistem dengan *load balancer* untuk protokol CoAP dan MQTT.

Tabel 6.11 Rata-rata *Time Publish* MQTT dengan *Load Balancer*

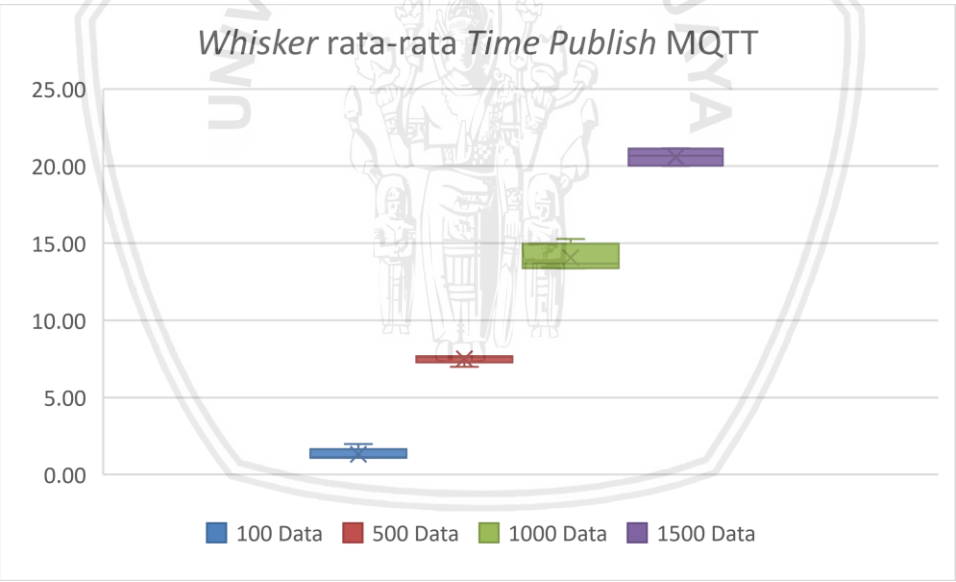
Jumlah *Publisher*	Percobaan				
	1 *Time* (s)	2 *Time* (s)	3 *Time* (s)	4 *Time* (s)	5 *Time* (s)
100	1.29	1.08	1.30	1.08	1.12
500	7.62	7.65	7.56	7.68	6.99
1000	15.28	13.41	14.66	13.68	13.36
1500	20.67	21.09	20.01	20.06	21.16

Berdasarkan tabel rata-rata *time publish* 6.10 dan 6.11, akan diperoleh grafik *whisker* pada gambar 6.22 dan 6.23. Gambar 6.22 menunjukkan grafik *whisker* rata-rata *time publish* pada protokol CoAP. Gambar 6.23 menunjukkan grafik *whisker* rata-rata *time publish* pada protokol MQTT.



Gambar 6.22 *Whisker* Rata-Rata *Time Publish* CoAP dengan *Load Balancer*

Grafik *whisker* pada gambar 6.22 menunjukkan nilai *time publish* CoAP yang cukup stabil. Hal tersebut berdasarkan luas area grafik yang diperoleh tidak begitu besar.



Gambar 6.23 *Whisker* Rata-Rata *Time Publish* MQTT dengan *Load Balancer*

Luas area untuk jumlah varian *client* 100, 500 dan 1500 pada gambar 6.23 tidak begitu besar. Hal itu menunjukkan nilai *time publish* yang stabil untuk varian *client* tersebut. Sedangkan luas area grafik untuk varian *client* 1000 lebih luas dari varian yang lain.

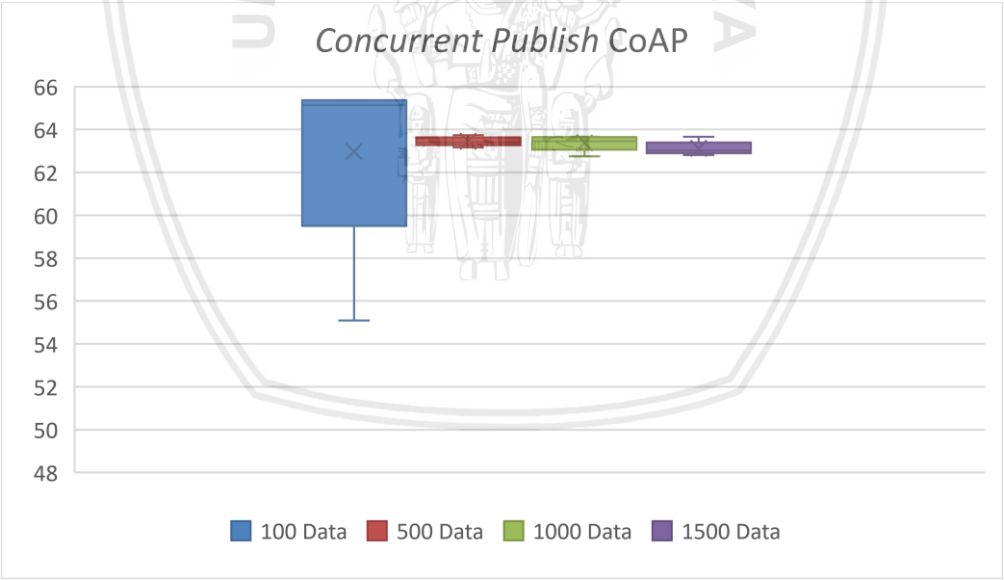
b. Pengujian *concurrent publish* dengan *load balancer*

Pengujian *concurrent publish* dilakukan berdasarkan kode uji PNF_002. Perhitungan dilakukan untuk mengetahui pesan yang dapat di-*publish* per detik pada setiap variasi *publisher* yang diberikan. Hasil *concurrent publish* dengan *load balancer* diperoleh dari jumlah *publisher* dibagi nilai *time publish* dengan *load balancer* pada tabel 6.10 dan 6.11 untuk setiap tingkatan jumlah *publisher*.

Tabel 6.12 Hasil Pengujian *Concurrent Publish* dengan *Load Balancer*

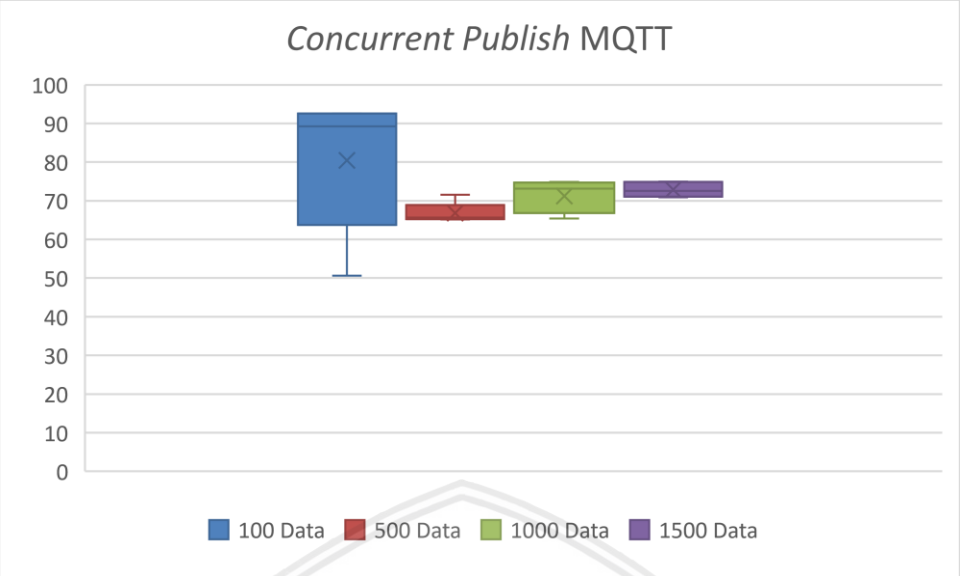
Jumlah Publisher	Percobaan									
	Concurrent (*message/s*)									
	1		2		3		4		5	
	CoAP	MQTT	CoAP	MQTT	CoAP	MQTT	CoAP	MQTT	CoAP	MQTT
100	65	78	55	93	64	77	65	93	65	89
500	63	66	63	65	64	66	64	65	63	72
1000	64	65	63	75	63	68	63	73	64	75
1500	63	73	63	71	63	75	63	75	64	71

Berdasarkan tabel 6.12, diperoleh hasil grafik *whisker concurrent publish* CoAP dengan *load balancer* pada gambar 6.23 dan *concurrent publish* MQTT pada gambar 6.24.



Gambar 6.23 *Whisker Concurrent Publish* CoAP dengan *Load Balancer*

Grafik *concurrent publish* protokol CoAP pada sistem dengan *load balancer* ditunjukkan oleh gambar 6.23. Dari gambar tersebut diperoleh nilai minimum *concurrent publish* CoAP adalah 55 pesan/detik. Sedangkan nilai maksimum *concurrent publish* CoAP adalah 65 pesan/detik.



Gambar 6.24 *Whisker Concurrent Publish* MQTT dengan *Load Balancer*

Gambar 6.24 menunjukkan nilai minimum *concurrent publish* MQTT pada sistem dengan *load balancer* adalah 78 pesan/detik. Sedangkan nilai maksimum untuk *concurrent publish* MQTT adalah 93 pesan/detik.

c. Pengujian *Time Subscribe* dengan *Load Balancer*

Pengujian *time subscribe* dengan *load balancer* dilakukan berdasarkan kode uji PNF_003. Tahapan ini akan dilakukan perhitungan *time subscribe* pada proses *subscribe* ke IoT *middleware* menggunakan protokol MQTT. Skenario pengujian yang digunakan dengan men-*generate* sejumlah klien yang ditentukan, pada sisi *subscriber.* Pengujian dilakukan sebanyak lima iterasi. Tangkapan paket dianalisis dengan *tool wireshark*. Tabel 6.12 menjelaskan hasil pengujian *time subscribe* pada IoT *middleware* satu. Tabel 6.13 menunjukkan hasil pengujian *time subscribe* pada IoT *middleware* kedua. Gambar 6.24 menunjukkan grafik *whisker* pengujian *time subscribe* dengan *load balancer* pada IoT *middleware* satu. Gambar 6.25 menunjukkan grafik *whisker* pengujian *time subscribe* dengan *load balancer* pada IoT *middleware* kedua.

Tabel 6.12 Hasil Pengujian *Time Subscribe* pada IoT *Middleware* Satu

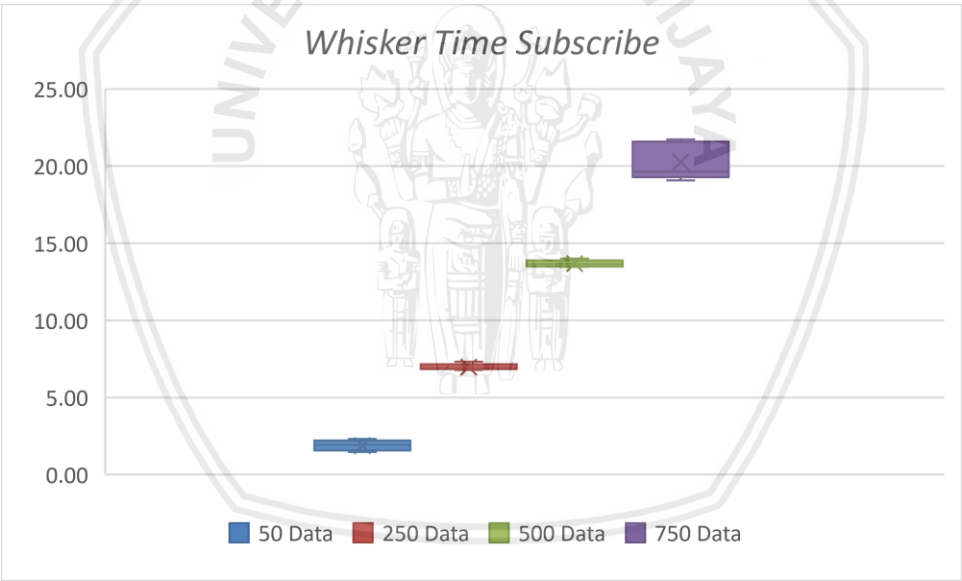
Jumlah *Publisher*	**Percobaan**				
	1 *Time* (s)	**2 *Time* (s)**	**3 *Time* (s)**	**4 *Time* (s)**	**5 *Time* (s)**
50	1.91	1.46	1.65	2.31	2.09
250	6.78	6.88	6.89	7.31	6.99
500	13.51	13.47	13.76	14.00	13.65
750	21.41	21.74	19.63	19.08	19.47

Load balancer mendistribusikan *traffic* ke IoT *middleware* satu dan dua menggunakan algoritem *round robin*. Hal tersebut menjadikan jumlah *client* pada

setiap IoT *middleware* adalah *n*/2. Tabel 6.12 menunjukkan perolehan *time subscribe* pada IoT *middleware* satu. Sedangkan tabel 6.13 menunjukkan perolehan *time subscribe* pada IoT *middleware* dua. Jumlah varian *client* yang masuk pada setiap IoT *middleware* adalah 50, 250, 500, dan 750. Hasil detail perolehan *time subscribe* pada IoT *middleware* di sistem dengan *load balancer* terdapat pada lampiran C.

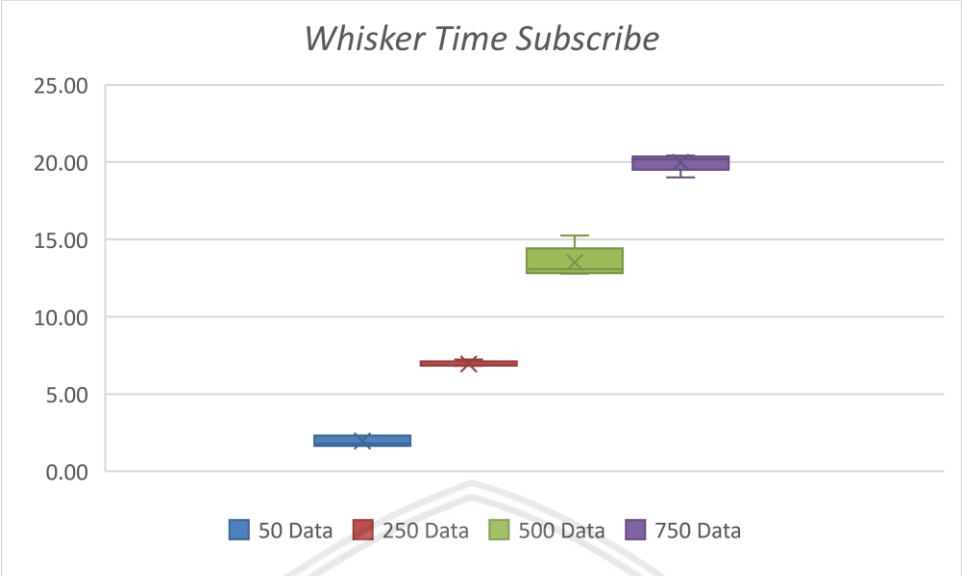
Tabel 6.13 Hasil Pengujian *Time Subscribe* pada IoT *Middleware* Kedua

Jumlah *Publisher*	Percobaan				
	1 *Time* (s)	2 *Time* (s)	3 *Time* (s)	4 *Time* (s)	5 *Time* (s)
50	1.80	1.64	1.67	2.31	2.32
250	6.82	6.87	6.85	7.23	6.99
500	12.78	12.87	13.61	13.06	15.24
750	19.00	20.02	20.18	20.42	20.28



Gambar 6.24 *Whisker Time Subscribe* dengan LB di IoT *Middleware* Satu

Gambar 6.24 dan 6.25 adalah grafik *whisker* yang didapatkan dari hasil pengujian *time subscribe* pada IoT *middleware* satu dan IoT *middleware* dua. Pada gambar 6.24 luas area grafik terlihat lebih besar pada varian *client* 750. Sedangkan pada gambar 6.25 luas area grafik terlihat lebih besar pada varian *client* 500.



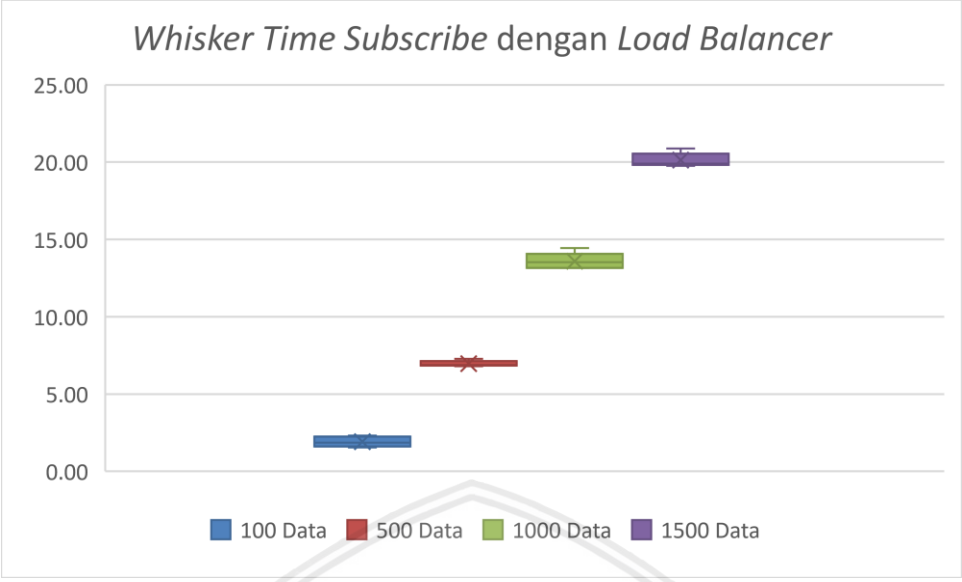
Gambar 6.25 *Whisker Time Subscribe* dengan LB di IoT *Middleware* Dua

Penerapan *load balancer* pada sistem, memungkinkan pemrosesan *request subscribe* pada waktu yang hampir bersamaan. Hal tersebut dikarenakan sistem dengan IoT *middleware* tunggal harus memproses sejumlah *client subscribe* pada satu IoT *middleware*. Sedangkan, sistem dengan dua IoT *middleware* memproses sejumlah *client subscribe* yang sama dengan bantuan distribusi oleh *load balancer* pada kedua IoT *middleware* tersebut. Oleh karena itu, akan diperoleh hasil rata-rata *time subscribe* pada sistem yang menggunakan *load balancer*. Tabel 6.14 menunjukkan rata-rata *time subscribe* dengan *load balancer*. Gambar 6.26 menunjukkan grafik *whisker* rata-rata *time subscribe* dengan *load balancer*.

Tabel 6.14 Rata-rata *Time Subscribe* dengan *Load Balancer*

Jumlah *Publisher*	**Percobaan**				
	1 *Time* (s)	**2 *Time* (s)**	**3 *Time* (s)**	**4 *Time* (s)**	**5 *Time* (s)**
100	1.86	1.55	1.66	2.31	2.21
500	6.80	6.88	6.87	7.27	6.99
1000	13.15	13.17	13.69	13.53	14.45
1500	20.21	20.88	19.91	19.75	19.88

Nilai rata-rata didapatkan dengan menjumlahkan hasil *time subscribe* dari IoT *middleware* satu dan dua pada masing-masing variasi *client* di setiap percobaan ,dibagi dua. Tabel 6.14 merupakan hasil rata-rata waktu *time subscribe* pada sistem dengan *load balancer*.



Gambar 6.26 Whisker Rata-Rata *Time Subscribe* dengan *Load Balancer*

Gambar 6.26 merupakan sebaran data rata-rata dari lima percobaan untuk melihat hasil *time subscribe*. Jika diperhatika luas area *whisker* pada gambar 6.26 lebih kecil daripada luas area *whisker* pada gambar 6.17.

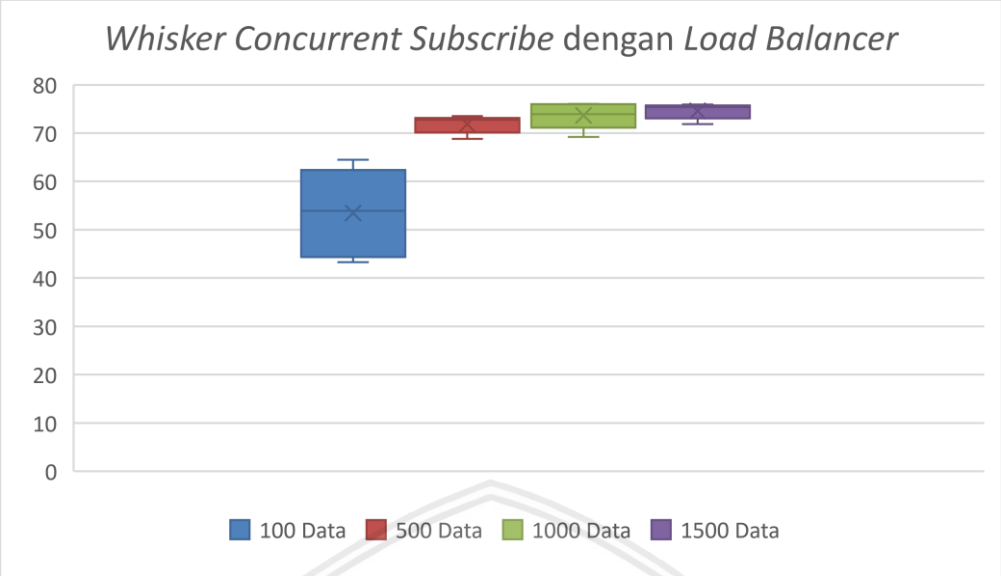
d. Pengujian *Concurrent Subscribe* dengan *Load Balancer*

Pengujian *concurrent subscribe* dilakukan berdasarkan kode uji PNF_004. Perhitungan *concurrent subscribe* dengan membagi variasi jumlah *subscriber* yang diberikan dengan rata-rata *time subcribe*. Tabel 6.15 menunjukkan hasil pengujian *concurrent subscribe* dengan *load balancer*. Gambar 6.27 menunjukkan grafik *whisker concurrent subscribe* dengan *load balancer*.

Tabel 6.15 Hasil *Concurrent Subscribe* dengan *Load Balancer*

Jumlah Subscriber	**Percobaan**				
	Concurrent (*message/s*)				
	1	**2**	**3**	**4**	**5**
100	54	65	60	43	45
500	74	73	73	69	72
1000	76	76	73	74	69
1500	74	72	75	76	75

Hasil *concurrent subscribe* pada tabel 6.15 menunjukkan nilai terendah adalah 43 pesan/detik. Sedangkan nilai tertinggi adalah 76 pesan/detik.



Gambar 6.27 *Whisker Concurrent Subscribe* dengan *Load Balancer*

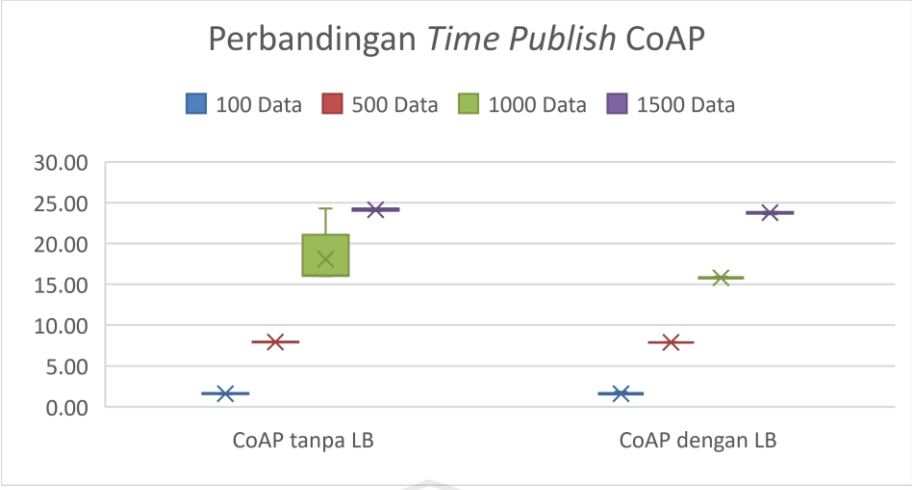
Grafik *whisker* pada gambar 6.27 menunjukkan perolehan nilai *concurrent subscribe* yang meningkat untuk setiap varian *client*. Nilai *concurrent* terendah berada pada nilai 43 pesan/detik. Sedangkan nilai tertinggi ada pada 76 pesan/detik.

6.2.3 Pembahasan Hasil Pengujian

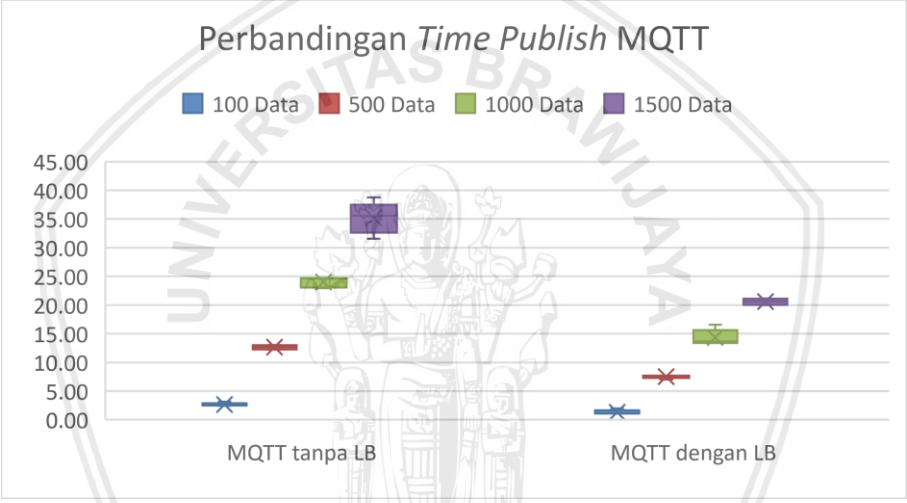
Pada bagian ini akan dilakukan pembahasan perbandingan hasil pengujian. Pembahasan bertujuan untuk mengetahui perbandingan penelitian sebelumnya dengan penelitan saat ini. Parameter analisis hasil pengujian adalah *time publish*, *concurrent publish*, *time subscribe* dan *concurrent subscribe*.

6.2.3.1 Pembahasan Hasil Pengujian *Time Publish*

Penelitian sebelumnya menggunakan IoT *middleware* tunggal pada sistem yang digunakan. Sedangkan sistem pada penelitian ini menggunakan dua buah IoT *middleware*. *Load balancer* digunakan sebagai poin masuk tunggal untuk *publisher* dan *subscriber*. Gambar 6.28 dan gambar 6.29 menunjukkan *whisker* perbandingan *time publish* CoAP dan MQTT dari penelitian sebelumnya. Perbandingan hasil *time publish* CoAP tidak terlalu signifikan pada sistem tanpa *load balancer* dengan sistem yang menggunakan *load balancer*. Hal tersebut dikarenakan protokol *transport* UDP yang digunakan oleh CoAP tidak menggunakan *handshake* seperti pada protokol *transport* TCP. Sehingga kinerja pada sistem tanpa *load balancer* dengan sistem yang menggunakan *load balancer* tidak dipengaruhi oleh *delay handshake*.



Gambar 6.28 Grafik Perbandingan *Time Publish* CoAP



Gambar 6.29 Grafik Perbandingan *Time Publish* MQTT

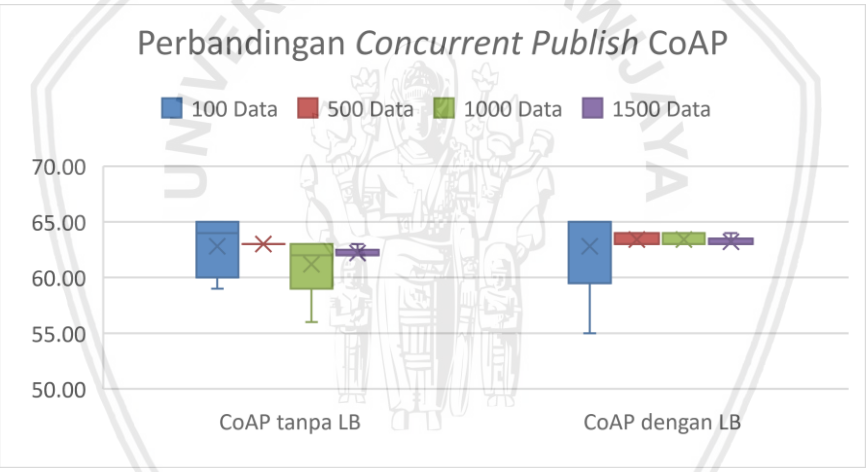
Grafik pada gambar 6.28 dan gambar 6.29 mengacu pada nilai yang terdapat pada tabel 6.1, 6.2, 6.9, dan 6.10. Tabel 6.1 adalah hasil pengujian *time publish* protokol CoAP tanpa *load balancer*. Tabel 6.2 adalah hasil pengujian *time publish* protokol MQTT tanpa *load balancer*. Tabel 6.9 adalah hasil pengujian *time publish* protokol CoAP dengan *load balancer*. Tabel 6.10 adalah hasil pengujian *time publish* protokol MQTT dengan *load balancer*. Berdasarkan gambar 6.28 dan gambar 6.29, sebaran hasil *time publish* pada sistem yang menggunakan *load balancer* menunjukkan hasil yang lebih rendah dari sistem yang tidak menggunakan *load balancer*. Hasil tersebut lebih terlihat perbedaannya pada varian *publisher* 1000 dan 1500.

Berdasarkan nilai *time publish* pada gambar 6.28 dan gambar 6.29, didapatkan rata-rata *time publish* untuk prokol CoAP dan MQTT pada setiap penelitian. Rata-rata *time publish* CoAP pada sistem yang tidak menggunakan *load balancer* adalah 1.6 detik untuk varian 100 *publisher*, 7.93 detik untuk varian 500 *publisher*, 16.36 detik untuk varian 1000 *publisher*, 24.13 detik untuk varian 1500 *publisher*. Rata-rata *time publish* CoAP pada sistem yang menggunakan *load balancer* adalah 1.6 detik untuk varian 100 *publisher*, 7.88 untuk varian 500

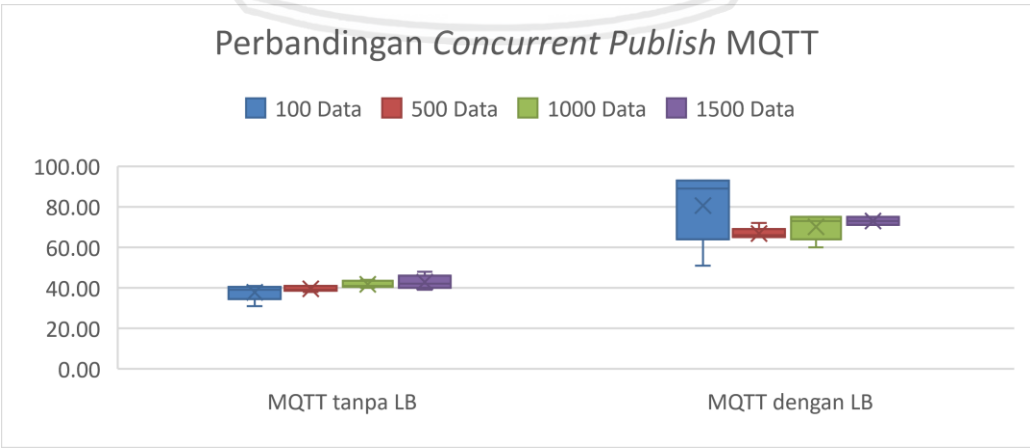
publisher, 15.78 untuk varian 1000 *publisher*, dan 23.77 detik untuk varian 1500 *publisher*. Rata-rata *time publish* MQTT pada sistem yang tidak menggunakan *load balancer* adalah 2.65 detik untuk varian 100 *publisher*, 12.65 detik untuk varian 500 *publisher*, 24.01 detik untuk varian 1000 *publisher*, 35.17 detik untuk varian 1500 *publisher*. Rata-rata *time publish* MQTT pada sistem yang menggunakan *load balancer* adalah 1.31 detik untuk varian 100 *publisher*, 7.5 detik untuk varian 500 *publisher*, 14.33 detik untuk varian 1000 *publisher*, dan 20.6 detik untuk varian 1500 *publisher*.

6.2.3.2 Pembahasan Hasil Pengujian *Concurrent Publish*

Terdapat perbedaan hasil pengujian saat ini dari penelitian sebelumnya terkait hasil *concurrent publish*. Gambar 6.30 dan gambar 6.31 menunjukkan grafik perbandingan *concurrent publish* CoAP dan MQTT dari penelitian sebelumnya. Pada protokol CoAP, sistem tanpa *load balancer* dan sistem dengan *load balancer* tidak dipengaruhi oleh *delay handshake*, seperti yang terjadi pada protokol MQTT. Hal tersebut mengakibatkan hasil *concurrent* CoAP pada sistem tanpa *load balancer* dengan sistem yang menggunakan *load balancer* tidak terlalu signifikan.



Gambar 6.30 Grafik Perbandingan *Concurrent Publish* CoAP



Gambar 6.31 Grafik Perbandingan *Concurrent Publish* MQTT

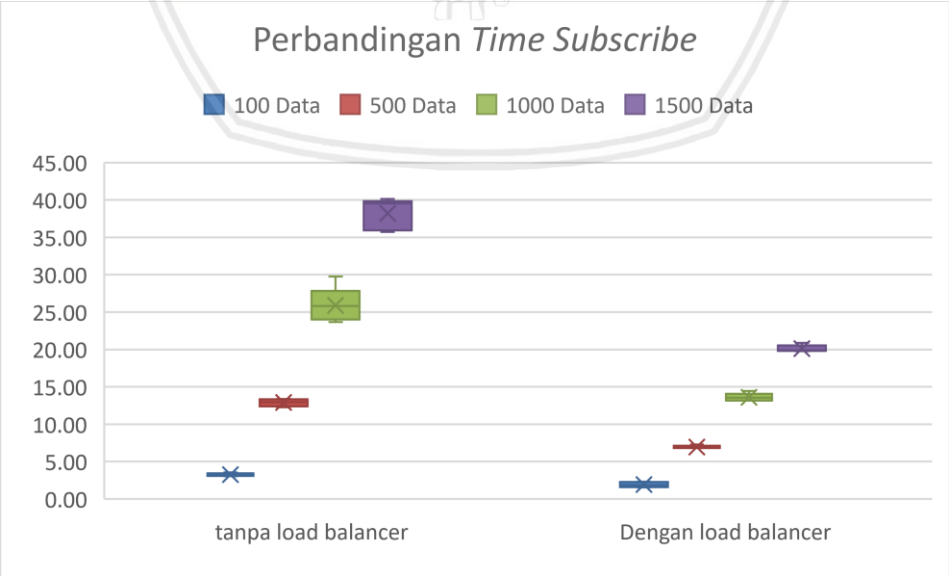
Grafik *whisker* pada gambar 6.30 dangambar 6.31 mengacu pada nilai yang terdapat pada tabel 6.3 dan 6.11. Tabel 6.3 adalah hasil pengujian *concurrent*

publish protokol CoAP dan MQTT tanpa *load balancer*. Tabel 6.11 adalah hasil pengujian *concurrent publish* protokol CoAP dan MQTT dengan *load balancer*. Berdasarkan gambar 6.30 dan gambar 6.31, sebaran hasil *concurrent publish* pada sistem yang menggunakan *load balancer* menunjukkan hasil yang relatif lebih tinggi dari sistem yang tidak menggunakan *load balancer*. Hasil yang signifikan terlihat pada *concurrent publish* MQTT dengan *load balancer*.

Berdasarkan nilai *concurrent publish* pada gambar 6.29, didapatkan rata-rata *concurrent publish* untuk prokol CoAP dan MQTT pada setiap penelitian. Rata-rata *concurrent publish* CoAP pada sistem yang tidak menggunakan *load balancer* adalah 63 pesan/detik untuk varian 100 *publisher*, 53 pesan/detik untuk varian 500 *publisher*, 61 pesan/detik untuk varian 1000 *publisher*, 62 pesan/detik untuk varian 1500 *publisher*. Rata-rata *concurrent publish* CoAP pada sistem yang menggunakan *load balancer* adalah 63 pesan/detik untuk seluruh varian *publisher*. Rata-rata *concurrent publish* MQTT pada sistem yang tidak menggunakan *load balancer* adalah 38 pesan/detik untuk varian 100 *publisher*, 40 pesan/detik untuk varian 500 *publisher*, 42 pesan/detik untuk varian 1000 *publisher*, 43 pesan/detik untuk varian 1500 *publisher*. Rata-rata *concurrent publish* MQTT pada sistem yang menggunakan *load balancer* adalah 80 pesan/detik untuk varian 100 *publisher*, 67 pesan/detik untuk varian 500 *publisher*, 70 pesan/detik untuk varian 1000 *publisher*, dan 73 pesan/detik untuk varian 1500 *publisher*.

6.2.3.3 Pembahasan Hasil Pengujian *Time Subscribe*

Hasil pengujian *time subscribe* pada tabel 6.4 akan dibandingkan dengan hasil pengujian *time subscribe* pada tabel 6.14. Tabel 6.4 adalah hasil pengujian *time subscribe* pada sistem yang tidak menggunakan *load balancer*. Tabel 6.14 adalah hasil pengujian *time subscribe* pada sistem yang menggunakan *load balancer*.



Gambar 6.32 Grafik Perbandingan *Time Subscribe*

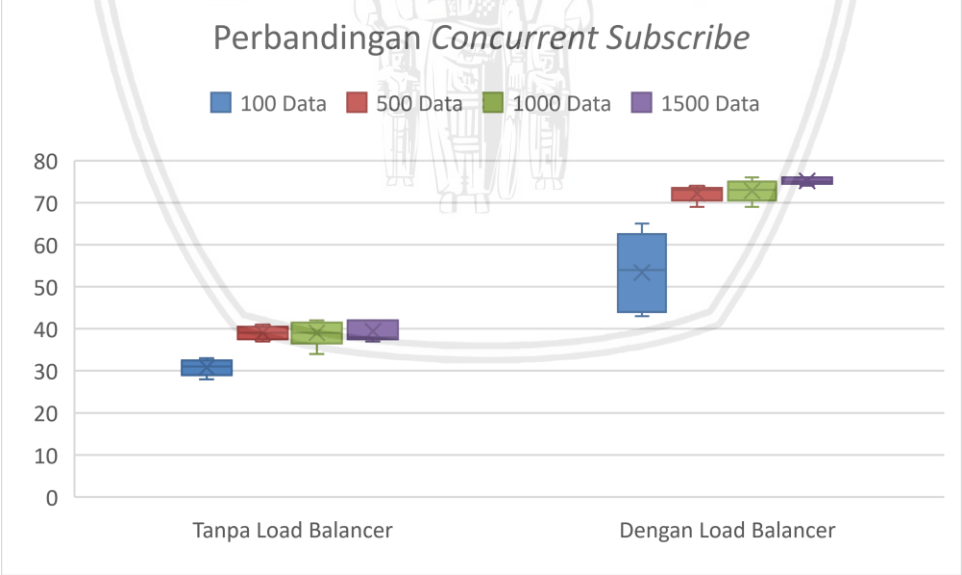
Gambar 6.32 menjelaskan grafik perbandingan *time subscribe* sistem saat ini dengan sistem sebelumnya. Berdasarkan gambar 6.32, sebaran hasil *time*

subscribe pada sistem yang menggunakan *load balancer* menunjukkan hasil yang relatif lebih rendah dari sistem yang tidak menggunakan *load balancer*. Sebaran hasil *time subscribe* pada sistem yang menggunakan *load balancer* menunjukkan selisih yang lebih kecil untuk setiap varian *client subscriber*. Selisih nilai tertinggi dan terendah pada varian *client* 100 di sistem dengan *load balancer* adalah 0.45 detik. Pada varian *client* 500 adalah 0.49 detik. Pada varian *client* 1000 adalah 1.30 detik. Dan pada varian *client* 1500 adalah 1.13 detik. Berdasarkan hasil tersebut diperoleh nilai kenaikan grafik pada setiap titik tidak begitu besar. Hal tersebut menandakan hasil *time subscribe* pada sistem dengan *load balancer* relatif stabil.

Rata-rata nilai *time subscribe* pada sistem yang tidak menggunakan *load balancer* adalah 3.25 detik untuk 100 *subscriber*, 12.89 detik untuk 500 *subscriber*, 25.90 detik untuk 1000 *subscriber*, 38.21 detik untuk 1500 *subscriber*. sedangkan Rata-rata nilai *time subscribe* pada sistem yang menggunakan *load balancer* adalah 2.7 detik untuk 100 *subscriber*, 10.36 detik untuk 500 *subscriber*, 19.24 detik untuk 1000 *subscriber*, 28.13 detik untuk 1500 *subscriber*.

6.2.3.4 Pembahasan Hasil Pengujian *Concurrent Subscribe*

Hasil pengujian *concurrent subscribe* pada tabel 6.5 akan dibandingkan dengan hasil pengujian *concurrent subscribe* pada tabel 6.15. Tabel 6.4 adalah hasil pengujian *concurrent subscribe* pada sistem yang tidak menggunakan *load balancer*. Tabel 6.15 adalah hasil pengujian *concurrent subscribe* pada sistem yang menggunakan *load balancer*.



Gambar 6.33 Grafik Perbandingan *Concurrent Subscribe*

Gambar 6.33 menjelaskan grafik perbandingan *concurrent subscribe* pada sistem tanpa *load balancer* dengan sistem yang menggunakan *load balancer*. Berdasarkan gambar 6.33, sebaran hasil *concurrent subscribe* pada sistem yang menggunakan *load balancer* menunjukkan hasil yang relatif lebih tinggi dari sistem yang tidak menggunakan *load balancer.* Pada proses *subscribe* terdapat dua proses pengiriman informasi. Pertama *broker* akan membaca data topik yang

dikirim oleh *subscriber*, kemudian *broker* mengirimkan data dengan topik yang di-*request subscriber*.

Pada grafik 6.33, varian *client* 100 untuk sistem dengan *load balancer* menunjukkan nilai yang lebih tidak stabil dari varian *client* yang lainnya. Selisih nilai *concurrent* tertinggi dan terendah pada varian *client* 100 tersebut adalah 22 pesan/detik. Hasil tersebut menunjukkan pada jumlah *client* tertentu (jumlah *client* sedikit) penggunaan *load balancer* menghasilkan nilai *concurrent* yang relatif tidak stabil. Sedangkan pada sistem yang tidak menggunakan *load balancer*, sebaran nilai *concurrent* relatif lebih stabil untuk setiap varian *client*. Akan tetapi, hal tersebut juga tidak menunjukkan perolehan *concurrent* pada sistem tanpa *load balancer* lebih banyak daripada sistem dengan *load balancer* dan dua IoT *middleware*. Nilai *concurrent* tertinggi pada sistem tanpa *load balancer* adalah 42 pesan/detik. Sedangkan nilai *concurrent* terendah pada sistem dengan *load balancer* adalah 43 pesan/detik.

Rata-rata nilai *concurrent subscribe* pada sistem yang tidak menggunakan *load balancer* adalah 31 pesan/detik untuk 100 *subscriber*, 39 pesan/detik untuk 500 *subscriber*, 1000 *subscriber*, dan 1500 *subscriber*. Rata-rata nilai *concurrent subscribe* pada sistem yang menggunakan *load balancer* adalah 37 pesan/detik untuk 100 *subscriber*, 48 pesan/detik untuk 500 *subscriber*, 52 pesan/detik untuk 1000 *subscriber*, 53 pesan/detik untuk 1500 *subscriber*.

BAB 7 PENUTUP

7.1 Kesimpulan

Berdasarkan perancangan, implementasi dan analisis pengujian yang telah dilakukan, dapat diambil beberapa kesimpulan sebagai berikut:

1. Penerapan mekanisme *load balancer* dan *failover* pada IoT *middleware* dapat diimplementasikan pada sistem IoT. *Load balancer* memanfaatkan API NGINX. Integrasi *failover* menggunakan bantuan perangkat lunak Keepalived. *Load balancer* yang diimplementasikan berjumlah dua buah. *Load balancer* pertama bertindak sebagai *node* master. *Load balancer* kedua bertindak sebagai *node back up*. *Load balancer* dikonfigurasi agar dapat menerima layanan dari protokol CoAP dan MQTT, baik untuk *publisher* dan *subscriber*. *Load balancer* mampu mendistribusikan *traffic* dari *publisher* dan *subscriber* ke dua perangkat IoT *middleware* yang berperan sebagai *back end server*. Perangkat lunak Keepalived dapat diimplementasikan untuk mengatasi proses *failover* pada dua buah perangkat *load balancer*. Algoritme yang digunakan untuk *load balancing* adalah *round robin*.

2. Pada penelitian ini, kinerja sistem dengan mekanisme *load balancer* dan *failover* dilihat dari kemampuan IoT *middleware* dalam memproses pesan *publish* dan *subscribe.* Kinerja tersebut dilihat dari hasil *time publish* dan *time subscribe* serta *concurrent publish* dan *concurrent subscribe*. Adapun hasil uji kinerja IoT *middleware* tersebut seperti berikut:

    a. Rata-rata nilai *time publish* CoAP pada sistem tanpa *load balancer* untuk varian *client* 100, 500, 1000 dan 1500 adalah 1.60 detik, 7.93 detik, 16.37 detik dan 24.13 detik. Sedangkan nilai *time publish* CoAP pada sistem dengan *load balancer* untuk varian *client* 100, 500, 1000 dan 1500 adalah 1.60 detik, 7.88 detik, 15.78 detik, dan 23.77 detik. Hasil tersebut dipengaruhi oleh jenis protokol *transport* yang digunakan dalam komunikasi.

    b. Rata-rata nilai *time publish* MQTT pada sistem tanpa *load balancer* untuk varian *client* 100, 500, 1000, dan 1500 adalah 2.65 detik, 12.65 detik, 24.01 detik dan 35.17 detik. Sedangkan nilai *time publish* MQTT pada sistem dengan *load balancer* untuk varian *client* 100, 500, 1000, dan 1500 adalah 1.17 detik, 7.50 detik, 14.38 detik, 20.60 detik. Hasil tersebut dipengaruhi oleh proses *handshake* yang digunakan oleh protokol MQTT.

    c. Rata-rata nilai *time subscribe* pada sistem tanpa *load balancer* untuk varian *client* 100, 500, 1000 dan 1500 adalah 3.04 detik, 3.29 detik, 3.13 detik, 3.20 detik dan 3.57 detik. Sedangkan nilai *time subscribe*

pada sistem dengan *load balancer* untuk varian *client* 100, 500, 1000 dan 1500 adalah 1.92 detik, 6.96 detik, 13.60 detik, dan 20.12 detik.

d. Rata-rata perolehan *concurrent publish* CoAP pada sistem tanpa *load balancer* untuk varian *client* 100, 500, 1000 dan 1500 adalah 63 pesan/detik, 63 pesan/detik, 61 pesan/detik, 62 pesan/detik. Sedangkan rata-rata perolehan *concurrent publish* CoAP pada sistem dengan *load balancer* untuk variasi *client* 100, 500, 1000 dan 1500 adalah 63 pesan/detik untuk setiap variasi *client*. Hal itu dipengaruhi oleh perolehan *time publish* CoAP pada kedua sistem.

e. Rata-rata nilai *concurrent publish* MQTT pada sistem tanpa *load balancer* untuk varian *client* 100, 500, 1000 dan 1500 adalah 38 pesan/detik, 40 pesan/detik, 42 pesan/detik, dan 43 pesan/detik. Sedangkan rata-rata nilai *concurrent publish* MQTT pada sistem dengan *load balancer* untuk varian *client* 100, 500, 1000 dan 1500 adalah 86 pesan/detik, 67 pesan/detik, 71 pesan/detik dan 73 pesan/detik. Hal itu dipengaruhi oleh perolehan *time publish* MQTT pada kedua sistem.

f. Rata-rata nilai *concurrent subscribe* pada sistem tanpa *load balancer* untuk varian *client* 100, 500, 1000 dan 1500 adalah 31 pesan/detik, 39 pesan/detik, 39 pesan/detik dan 39 pesan/detik. Sedangkan rata-rata nilai *concurrent subscribe* pada sistem dengan *load balancer* untuk varian *client* 100, 500, 1000 dan 1500 adalah 53 pesan/detik, 72 pesan/detik, 74 pesan/detik, 75 pesan/detik.

3. Perbandingan kinerja sistem tanpa mekanisme *load balancer* dan *failover* dengan sistem yang menggunakan *load balancer* dan *failover*, terlihat dari hasil *concurrent* yang didapatkan. Adapun hasil *concurrent* tersebut adalah sebagai berikut:
   a. Nilai *concurrent publish* CoAP pada sistem tanpa mekanisme *load balancer* dan *failover* adalah 60 pesan/detik. Sedangkan nilai *concurrent publish* CoAP pada sistem dengan mekanisme *load balancer* dan *failover* adalah 63 pesan/detik.
   b. Nilai *concurrent publish* MQTT pada sistem tanpa mekanisme *load balancer* dan *failover* adalah 41 pesan/detik. Sedangkan nilai *concurrent publish* MQTT pada sistem dengan mekanisme *load balancer* dan *failover* adalah 74 pesan/detik.
   c. Nilai *concurrent subscribe* pada sistem tanpa mekanisme *load balancer* dan *failover* adalah 37 pesan/detik. Sedangkan nilai *concurrent subscribe* pada sistem dengan mekanisme *load balancer* dan *failover* adalah 68 pesan/detik.

7.2 Saran

Berdasarkan kesimpulan yang telah dijelaskan, peneliti memberikan beberapa saran agar dapat menjadi bahan penelitian selanjutnya. Saran-saran tersebut antara lain:

1. Digunakannya parameter *n-size* pada pengujian skalabilitas untuk dapat melihat kemampuan IoT *middleware* dalam menampung sejumlah data.

2. Digunakan parameter *cpu usage* pada pengujian skalabilitas untuk dapat lebih presisi mengetahui kinerja *load balancer* pada sistem yang diteliti.

3. API NGINX membaca satu koneksi CoAP dan satu koneksi MQTT sebagai *state* yang berbeda, sehingga pada penelitian selanjutnya perlu API lain yang membaca koneksi dari protokol yang berbeda sebagai *state* yang sama.

DAFTAR PUSTAKA

Antonic, A. e. a., 2015. *Comparison of the CUPUS middleware and MQTT protocol for smart city services.* Graz, IEEE.

Anwari, H., 2017. Pengembangan IoT Middleware Berbasis Event-Based dengan. *JPTIIK.*

Anwari, Pramukantoro & Hanafi, 2017. Pengembangan Iot Middleware Berbasis Event-Based dengan Protokol Komunikasi CoAP, MQTT dan Websocket. *Jurnal Pengembangan Teknologi Informasi dan Ilmu Komputer,* 1(12), pp. 1560-1567.

Balasubramanian, e. a., 2004. *Evaluating the performance of middleware load balancing strategies.* Monterey, IEEE.

Bandyopadhyay, D. & Sen, J., 2011. Internet of Things: Applications and Challenges. *Wireless Personal Communication,* 58(1), pp. 46-69.

Bellavista, P. & Zanni, A., 2016. *Towards Better Scalability for IoT-Cloud Interactions via Combined Exploitation of MQTT and CoAP.* Bologna, IEEE.

Chen, e. a., 2017. *Towards Scalable and Reliable In-Memory Storage System: A Case Study with Redis.* Tianjin, IEEE.

dc-square, 2018. *MQTT.* [Online] Available at: https://www.hivemq.com/mqtt/ [Accessed 20 Agustus 2018].

Ed., R. H., 2004. *Virtual Router Redundancy Protocol (VRRP).* [Online] Available at: https://tools.ietf.org/html/rfc3768 [Accessed 28 Maret 2019].

Godha, R. & Prateek, S., 2014. Load Balancing in a Network. *International Journal of Scientific and Research Publications,* 4(10), pp. 1-3.

Gupta, A., Christie, R. & Manjula, R., 2017. Scalability in Internet of Things: Features,. *International Journal of Computational Intelligence Research,* Volume 13, pp. 1617-1627.

Handoko, H. & Isa, M. S., 2018. *High Availability Analysis With Database Cluster, Load Balancer And Virtual.* Nagoya, IEEE.

Jutadhamakorn, P. e. a., 2017. *A Scalable and Low-Cost MQTT Broker Clustering.* Nakhonpathom, IEEE.

keepalived, 2018. *Home.* [Online] Available at: www.keepalived.org [Accessed 18 December 2018].

Li, S., Jiang, H. & Shi, M., 2017. *Redis-based Web Server Cluster Session Maintaining.* Shanghai, IEEE.

Oxford, 2019. *Dictionary: definition.* [Online] Available at: https://en.oxforddictionaries.com/definition/middleware [Accessed 2 January 2019].

Perera, C., Jayaraman, P. P. & Christen, P., 2014. *MOSDEN: An Internet of Things Middleware for.* Hawaii, IEEE.

Razzaque, M. e. a., 2016. Middleware for Internet of Things: A Survey. *IEEE INTERNET OF THINGS JOURNAL,* Volume III, pp. 70-95.

Redis, 2018. *Redis Sentinel Documentation.* [Online] Available at: https://redis.io/topics/sentinel [Accessed 25 Agustus 2018].

Scalagent, 2014. *JoramMQ, a distributed MQTT Broker for the Internet of things.* [Online] Available at: http://www.scalagent.com/IMG/pdf/JoramMQ_MQTT_white_paper-v1-2.pdf [Accessed 25 Agustus 2018].

Shelby, Z., Hartke, K. & Borman, C., 2014. *The Constrained Application Protocol (CoAP).* [Online] Available at: https://www.rfc-editor.org/info/rfc7252 [Accessed 25 Agustus 2018].

Son, H. e. a., 2015. *A Distributed Middleware for a Smart Home with.* Taichung, IEEE.

Soni, D. & Makwana, A., 2017. *A SURVEY ON MQTT: A PROTOCOL OF INTERNET OF THINGS(IOT).* Chennai, Research Gate.

Tayyaba, S. K., Shah, M. A., Khan, O. A. & Ahmed, A. W., 2017. *Software Defined Network (SDN) Based Internet of Things (IoT): A Road Ahead.* Cambridge, United Kingdom, Association for Computing Machinery, pp. 1-8.

Wajgi, R. & Thakur, N. V., 2012. Load Balancing Algorithms in Wireless *Sensor*. *International Journal of Computer Networks and Wireless Communications (IJCNWC),* Volume 2, pp. 456-460.

Whitmore, A., Agarwal, A. & Xu, L. D., 2014. The Internet of Things—A survey of topics and trends. *Information System Frontier,* 17(2), pp. 261-274.

Wulandari, Pramukantoro & Nurwasito, 2018. Implementasi Cluster Message Broker Sebagai Solusi Skalabilitas Middleware Berbasis Arsitektur Publish-Subscribe pada Internet of Things (IoT). *Jurnal Pengembangan Teknologi Informasi dan Ilmu Komputer,* 2(12), pp. 6861-6867.

XU, Z. & WANG, X., 2015. *A Predictive Modified Round Robin Scheduling algorithm for web server clusters.* Hangzhou, Chinese Control Conference.