IMPLEMENTASI *CLUSTER MESSAGE BROKER* SEBAGAI SOLUSI SKALABILITAS *MIDDLEWARE* BERBASIS ARSITEKTUR *PUBLISH-SUBSCRIBE* PADA *INTERNET of THINGS* (IOT)

SKRIPSI

Untuk memenuhi sebagian persyaratan
memperoleh gelar Sarjana Komputer

Disusun oleh:
Jessy Ratna Wulandari
NIM: 145150200111071

PROGRAM STUDI TEKNIK INFORMATIKA
JURUSAN TEKNIK INFORMATIKA
FAKULTAS ILMU KOMPUTER
UNIVERSITAS BRAWIJAYA
MALANG
2018

PENGESAHAN

IMPLEMENTASI *CLUSTER MESSAGE BROKER* SEBAGAI SOLUSI SKALABILITAS *MIDDLEWARE* BERBASIS ARSITEKTUR *PUBLISH-SUBSCRIBE* PADA *INTERNET of THINGS* (IoT)

SKRIPSI

Diajukan untuk memenuhi sebagian persyaratan
memperoleh gelar Sarjana Komputer

Disusun Oleh :
Jessy Ratna Wulandari
NIM: 145150200111071

Skripsi ini telah diuji dan dinyatakan lulus pada
3 Agustus 2018
Telah diperiksa dan disetujui oleh:

Dosen Pembimbing I	Dosen Pembimbing II
Eko Sakti P, S.Kom, M.Kom	Ir. Heru Nurwasito, M.Kom
NIK: 201102 860805 1 001	NIP: 19650402 199002 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 di dalam 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, 3 Agustus 2018

Jessy Ratna Wulandari

NIM: 145150200111071

KATA PENGANTAR

Puji syukur penulis panjatkan atas kehadirat Allah SWT, karena berkat rahmat dan hidayah-Nya, penulis dapat menyelesaikan penyusunan skripsi dengan judul "Implementasi *Cluster Message Broker* Sebagai Solusi Skalabilitas *Middleware* Berbasis Arsitektur *Publish-Subscribe* Pada *Internet of things* (IoT)" dengan baik dan lancar.

Dalam penyusunan skripsi ini, penulis banyak mendapatkan bantuan, bimbingan serta dukungan baik materil maupun secara moril dari berbagai pihak. Oleh karena itu dalam kesempatan ini penulis menyampaikan ucapan terima kasih kepada:

1. Orang tua penulis, Bapak Supriyanto dan Ibu Kusmiyati serta keluarga besar yang selalu memberikan dukungan dan doa untuk kelancaran kuliah penulis.
2. Bapak Eko Sakti P, S.Kom., M.Kom dan bapak Ir. Heru Nurwasito, M.Kom., selaku dosen pembimbing I dan II yang selalu membimbing, membantu serta meluangkan waktu dan pikiran untuk penulis sehingga 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. Sahabat penulis Dini Indah N.R.P.R yang selalu memberikan doa dan dukungan serta teman dalam keadaan suka dan duka untuk penulis.
8. Teman spesial Eko Aditya Ramadianto yang selalu menemani dan memberikan dukungan dalam suka dan duka untuk penulis selama penyusunan skripsi.
9. Maxi Luckies G, Tuti Wardani H, Binariyanto Aji, Hilman Nihri, dan Ahmad Naufal serta teman-teman seperjuangan dalam keminatan jaringan yang selalu memberikan bantuan ilmu dan semangat serta menjadi motivasi dalam penyusunan skripsi.
10. Keluarga Eksekutif Mahasiswa Informatika periode 2014/2015, 2015/2016, dan 2016/2017 yang telah berbagi ilmu dalam berorganisasi di lingkungan perkuliahan.

11. Teman-teman kelas Tif-E, Teknik Informatika Angkatan 2014, Gazebo Army, PKL OTW Jogja yang telah berbagi canda tawa, semangat, dan ilmu dari awal perkuliahan.
12. Dan Terima kasih kepada semua pihak yang telah memberikan bantuan dan dukungannya 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 yang membacanya.

Malang, 3 Agustus 2018

Penulis

jessyratnawulandari@gmail.com

ABSTRAK

Jessy Ratna Wulandari, Implementasi *Cluster Message Broker* Sebagai Solusi Skalabilitas *Middleware* Berbasis Arsitektur *Publish-Subscribe* Pada *Internet of things (*IoT)

Dosen Pembimbing: Eko Sakti Pramukantoro, S.Kom., M.Kom dan Ir. Heru Nurwasito, M.Kom

Skalabilitas merupakan salah satu tantangan yang ada pada *middleware* IoT. Meningkatnya jumlah *publisher* dan *subscriber* maka akan berpengaruh terhadap kinerja *middleware*, oleh karena itu secara tidak langsung IoT *middleware* harus mampu mengatasi tantangan skalabilitas. Salah satu solusi dalam mengatasi tantangan skalabilitas yaitu *Cluster message Broker*. Tujuan dari penelitian ini adalah membangun IoT *middleware* yang *scalable* dengan menerapkan metode *cluster* pada *message Broker middleware*. *Cluster* akan dibangun pada *message Broker* yaitu Redis. Beberapa *redis* akan dikonfigurasikan ke dalam mode *Cluster* sehingga mampu saling berbagi data. Selanjutnya *Cluster* yang telah di bangun akan di integrasikan dengan *middleware*. Pengujian skalabilitas dilakukan dengan melakukan simulasi perbesaran jumlah beban *publisher* dan *subscriber* sebesar 100, 500, 1000 dan 1500. Dari pengujian skalabilitas *middleware* yang dilakukan terhadap penelitian sebelumnya dengan penilitian saat ini didapatkan nilai rata-rata *time publish* setiap variasi pada MQTT lebih cepat dibandingkan CoAP dalam pengiriman pesan. Pada *concurrent publish* penelitian sebelumnya, CoAP mampu mencapai tingkat skalabilitas 71 pesan per detik sedangkan MQTT mencapai 62 pesan per detik. Sedangkan hasil penilitian saat ini CoAP hanya mampu mencapai 43 pesan per detik dan MQTT mencapai 54 pesan per detik, hal tersebut disebabkan oleh proses replikasi yang terjadi diantara *node master* dan *slave.* Sedangkan pada hasil penelitian untuk *Websocket middleware* sebelumnya mencapai *35* pesan per detik*,* sedangkan *untuk Websocket* pada *middleware* dengan *cluster* mencapai 31 pesan per detik*.* Dapat disimpulkan bahwa *middleware* dengan *cluster message broker* pada penelitian ini belum mampu meningkatkan skalabilitas pada *middleware*.

Kata kunci: IoT *middleware, message Broker*, *Cluster*, skalabilitas

ABSTRACT

Jessy Ratna Wulandari, ***Implementation of Cluster Message Broker as A Solution For Middleware Scalability Based On Publish-Subscribe Architecture In Internet of Things (IoT).***

Dosen Pembimbing: Eko Sakti Pramukantoro, S.Kom., M.Kom. dan Ir. Heru Nurwasito, M.Kom.

Scalability is one of the challenges in IoT middleware. The increasing amount of publisher and subscriber affect the middleware performance. Which is why the IoT middleware should be able to face that challenge. The solution for this scalability challenge is Cluster. The goal in this research is develop scalable IoT middleware with cluster in message broker middleware. The Cluster will be applied on Redis message Broker. Some of redis nodes will be configured as Cluster mode which allows sharing data between them. The Cluster will be developed in Redis as message broker. A few Redis node will be configured in cluster mode so that the data can be shared. After that, the developed cluster will be integrated with middleware. The scalability testing was done by running a simulation for 100, 500, 1000 and 1500 publisher. The middleware scalability testing from previous research with this research shows that average time publish for every variation for MQTT is faster than CoAP in sending the message. The concurrent publish in previous research, CoAP was able to reach scalability 71 message per second, for MQTT reach 62 message per second. Meanwhile, these results show 43 message per second for CoAP and 54 message per second for MQTT, this result happens because the replication process that occurs between the master and slave node. The concurrent publish in previous research shows that Websocket was able to reach scalability 35 message per second, meanwhile these results show 31 message per second. It can be concluded that the middleware with cluster message broker on this research has not been able to increase the scalability of middleware.

Keywords: Middleware IoT, message Broker, Cluster, scalability.

DAFTAR ISI

DAFTAR TABEL

DAFTAR GAMBAR

BAB 1 PENDAHULUAN

1.1 Latar Belakang

Pada *Internet of things* (IoT), *middleware* memiliki peran yang sangat penting. *Middleware* memiliki peran penting karena bertanggung jawab untuk mengintegrasikan data dari berbagai perangkat, mengijinkan perangkat untuk berkomunikasi, dan membuat keputusan berdasarkan data yang dikumpulkan (da Cruz *et al.*, 2018). IoT dihadapkan pada beberapa permasalahan, salah satunya adalah *Interoperability*. *Interoperability* dapat diatasi dengan adanya *middleware* pada IoT. *Middleware* merupakan perangkat lunak di antara aplikasi, sistem operasi dan komunikasi jaringan yang memfasilitasi dan mengkoordinasikan beberapa aspek pemrosesan (Razzaque, Milojevic-jevric *and* Palade, 2016)

Pada penelitian sebelumnya Anwari berhasil mengatasi permasalahan salah satu jenis *Interoperability* dengan membuat *middleware* yang menyediakan *Gateway* multi-protokol untuk CoAP, MQTT, dan *Websocket*. *Gateway* yang dibuat untuk masing-masing protokol dihubungkan oleh sebuah *broker* yaitu Redis. *Middleware* tersebut dibangun dengan menggunakan pendekatan *event-driven* yang menerapkan pola *publish-subscribe*. Dalam implementasinya *middleware* tersebut diterapkan pada *Raspberry pi*, selain itu terdapat *publisher* yang mengirimkan topik *home/kitchen* menggunakan protokol CoAP dan topik *home/garage* menggunakan protokol MQTT, dan *subscriber* yang akan melakukan *subscribe* topik yang dikirimkan oleh *publisher* (Anwari, Pramukantoro *and* Hanafi, 2017).

Selanjutnya penelitian tersebut dilanjutkan oleh Rozi dengan melakukan pengujian performansi dan skalabilitas untuk mengetahui kinerja *middleware*. Hasil pengujian menyatakan bahwa performansi *Middleware* dalam pengiriman data yang dikirim oleh *node* MCU CoAP lebih baik daripada *node* MCU MQTT, terbukti dari 4 skenario yang digunakan menunjukkan rata-rata *delay node* MCU CoAP lebih kecil daripada *node* MCU MQTT. Hasil penelitian skalabilitas menunjukkan kemampuan *Middleware* dalam menangani *publish* atau *subscribe* dengan jumlah klien 100 hingga 1000 dalam satu detik bergerak naik seiring bertambahnya jumlah klien, serta jumlah klien bertambah *concurrent Middleware* juga bertambah (Rozi, Pramukantoro and Amron, 2017). Berdasarkan penelitian yang telah dilakukan, "Hasil *delay* yang tinggi dipengaruhi oleh arsitektur *Middleware* yang digunakan dan menggunakan Redis yang difungsikan sebagai *Broker*. Kinerja redis bergantung pada *space* yang tersedia untuk redis tersebut, kemudian dikarenakan terdapat banyak proses pengiriman data yang ditujukan ke Redis saat *memory* sedang menangani banyak proses sehingga ruang yang dialokasikan untuk Redis sedikit yang menyebabkan waktu tunggu yang dibutuhkan salah satu proses pengiriman data sampai menerima *ack* terjadi cukup lama" (Rozi, Pramukantoro and Amron, 2017). Oleh karena itu

akan terjadi permasalahan di mana klien yang mampu ditampung oleh sejumlah *memory* yang tersedia menjadi terbatas dan semakin banyak permintaan data juga akan berpengaruh terhadap kinerja *Middleware* yang tidak efisien.

Dalam mengatasi permasalahan tersebut dibutuhkan sebuah solusi yang mampu mendukung skalabilitas. Beberapa penelitian telah dilakukan untuk mengatasi masalah skalabilitas dalam berbagai penerapan. Son mengatasi masalah skalabilitas pada *Smart Home* dengan membangun sebuah *Middleware* terdistribusi yang dapat membuat perangkat cerdas bisa mempelajari perilaku pengguna dan bertindak secara otomatis dengan berkerja sama dengan perangkat lain melalui *Gateway* (Son *et al.*, 2015). Kemudian penelitian yang dilakukan oleh Jutadhamakorn membangun sebuah MQTT *Broker* yang bersifat *scalable* dengan metode *clustering* dalam menangani sejumlah besar perangkat IoT. Pengujian yang dilakukan untuk mengukur *throughput*, *elapsed time*, dan CPU *usage*. Hasil pengujian menunjukkan peningkatan *throughput* sebesar 8 kali dibandingkan dengan menggunakan *Broker* tunggal dan juga mampu mengurangi penggunaan CPU menjadi 1/6 dibandingkan dengan *Broker* tunggal (Jutadhamakorn *et al.*, 2017).

Berdasarkan permasalahan yang ada, redis dapat dikatakan sebagai objek dalam penelitian selanjutnya. Li mengusulkan teknologi pemrograman *cluster Web server* berbasis Redis untuk menjaga konsistensi *session cluster Web server* dan meningkatkan kemampuan ekspansi. Redis *cluster* digunakan untuk menyimpan *session*, sehingga *Web server* dapat mengaksesnya melalui jaringan. Hasilnya Redis *cluster* menyediakan kecepatan *read-write* data dan *availability* yang tinggi (Li, Jiang *and* Shi, 2017). Sedangkan penelitian oleh Thomas mengusulkan metode berbasis *cluster* yang memiliki performa tinggi untuk menyimpan, memproses, dan mengambil data geospasial dinamis menggunakan Redis dalam mengatasi masalah *tracking* lokasi kendaraan di lingkungan *smart city*. Hasilnya metode ini mampu menyimpan data GPS dalam jumlah besar dan lebih cepat dalam pencarian lokasi kendaraan (Thomas, Alexander *and* Pm, 2017).

Berdasarkan latar belakang tersebut, maka dalam penelitian ini diusulkan sebuah *cluster* yang dibangun menggunakan *message Broker* Redis pada *middleware*. Selanjutnya *cluster* akan diimplementasikan pada lingkungan IoT yang terdiri dari *publisher*, *middleware* dan *subscriber*. Untuk mengintegrasikan *middleware* dengan *cluster* maka ditambahkan Ioredis yang mampu mendukung *cluster*. Kemudian dilakukan pengujian dengan parameter *time publish*, *time subscribe*, *concurrent publish*, dan *concurrent subscribe* dengan jumlah variasi *publisher dan subscriber* yaitu 100, 500, 1000, dan 1500. Penelitian ini diharapkan akan mampu mengatasi permasalahan skalabilitas pada *middleware* IoT dalam menangani sejumlah beban yang diberikan.

1.2 Rumusan Masalah

Berdasarkan uraian latar belakang di atas, maka dapat dirumuskan permasalahan pada penelitian ini yaitu sebagai berikut:

1. Bagaimana menerapkan *cluster* pada *message Broker* ke dalam *Middleware*?
2. Bagaimana kinerja *Middleware* yang menerapkan *Cluster Message Broker* dalam menjawab permasalahan skalabilitas?

1.3 Tujuan

Tujuan dari penelitian ini yaitu sebagai berikut:

1. Menerapkan *Cluster* pada *message Broker Middleware*.
2. Mengetahui tingkat kinerja *Middleware* yang menerapkan *Cluster Message Broker* dalam menjawab permasalahan skalabilitas.

1.4 Manfaat

Penelitian ini diharapkan dapat memberikan beberapa manfaat diantaranya:

1. Penelitian ini dapat dijadikan media dalam mengimplementasikan teori yang telah dipelajari khususnya mengenai *cluster message Broker* dan dapat dijadikan alat dalam memperkaya ilmu pengetahuan.
2. Penelitian ini dapat dijadikan dasar untuk melakukan penelitian lebih lanjut dengan topik yang berkaitan.

1.5 Batasan Masalah

Batasan dari penelitian ini adalah:

1. Lingkungan IoT adalah pada penelitian sebelumnya yang mencakup tiga *node* yaitu antara *node* sensor sebagai *publisher*, *node middleware*, dan *node subscriber*.
2. Sistem menggunakan tiga mesin *Raspberry pi*.
3. Sistem menggunakan Redis yang digunakan sebagai *message Broker* dan basis data.
4. Sistem menggunakan sensor DHT11 dan DHT22.
5. Topo*log*i *cluster* yang digunakan yaitu *Mesh*, di mana satu *node* terhubung dengan *node* yang lainnya menggunakan TCP *connection*.

1.6 Sistematika Pembahasan

Sistematika pembahasan penelitian ini disusun sebagai berikut:

BAB I PENDAHULUAN

Bab ini berisi tentang penjelasan latar belakang, rumusan masalah, tujuan, manfaat, batasan penelitian, serta sistematika dari penelitian yang dilakukan.

BAB II LANDASAN KEPUSTAKAAN

Bab ini berisi dasar teori yang diambil dari jurnal, *web* resmi yang berkaitan dengan isi penelitian, dan kutipan buku serta berisi penjelasan pada penelitian sebelumnya yang digunakan sebagai panduan dalam proses penelitian.

BAB III METODOLOGI

Bab ini berisi tentang penjelasan langkah-langkah yang dilakukan dalam melaksanakan penelitian.

BAB IV ANALISIS KEBUTUHAN

Bab ini berisi penjelasan proses analisis kebutuhan sistem yang diperlukan dalam proses implementasi, baik dari kebutuhan fungsional dan non-fungsional.

BAB V PERANCANGAN DAN IMPLEMENTASI SISTEM

Bab ini berisi proses perancangan dan implementasi sistem *Middleware* yang menggunakan *cluster* pada *message Broker* yang dilakukan dalam proses penelitian.

BAB VI PENGUJIAN DAN ANALISIS

Bab ini menunjukkan hasil pengujian yang dilakukan dan menjelasan analisis dari hasi pengujian.

BAB VII PENUTUP

Bab ini berisi beberapa kesimpulan yang diperoleh dari perancangan, implementasi, dan pengujian sistem serta saran untuk pengembangan penelitian selanjutnya.

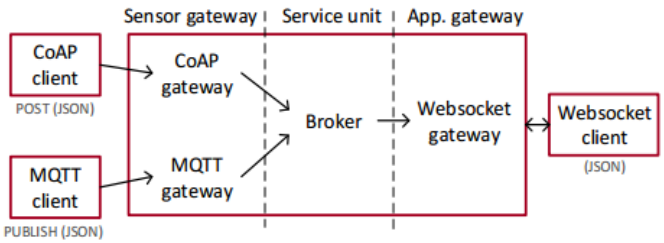
BAB 2 LANDASAN KEPUSTAKAAN

Pada penelitian ini, landasan kepustakaan sebagai referensi yang berasal dari beberapa literatur, jurnal, dan *web* resmi terkait konsep *middleware* pada *Internet of things*, protokol komunikasi pada *Internet of things*, Skalabilitas dan Redis *Cluster*. Jurnal dan literatur tersebut meliputi konsep *middleware* pada *Internet of things* yang ada pada penelitian sebelumnya, protokol CoAP, MQTT, dan *Websocket*, konsep pendekatan skalabilitas dan metodo*log*i pengujian dari segi skalabilitas, arsitektur *publish-subscribe*, dan Redis *cluster*.

2.1 Kajian Pustaka

Penelitian yang dilakukan oleh Anwari berhasil membangun sebuah *Middleware* melalui pendekatan *event-driven* dalam mendukung masalah *syntactic Interoperability* berbagai macam perangkat atau sensor dengan membuat *Gateway* multi-protokol. *Middleware* menggunakan Redis sebagai *message broker* yang juga bertindak sebagai media penyimpanan data dalam *memory*. Berdasarkan hasil percobaan, komunikasi antara CoAP, MQTT, dan *Websocket* dapat dicapai dengan membuat *Gateway* untuk setiap protokol, yang kemudian dihubungkan dengan *Broker* (Anwari, Pramukantoro and Hanafi, 2017).

Pada gambar 2.1 merupakan arsitektur *middleware* yang dibangun oleh Anwari yang menunjukkan tiga bagian dari *middleware* yaitu sensor *Gateway*, *service* unit, dan *application Gateway*. Sensor *Gateway* menyediakan *interface* bagi sensor untuk mengirimkan data dari sensor ke *middleware* melalui protokol CoAP dan MQTT. *Service* unit sebagai *service delivery* yang menyediakan *interface* bagi sensor dan *application Gateway* untuk mengakses *Broker*. *Application Gateway* menyediakan *interface* bagi aplikasi untuk membaca data yang dikirimkan sensor ke *middleware* melalui protokol *Websocket*. Dibawah ini merupakan gambar arsitektur *middleware* oleh (Anwari, Pramukantoro and Hanafi, 2017).



Gambar 2.1 Arsitektur *Middleware*

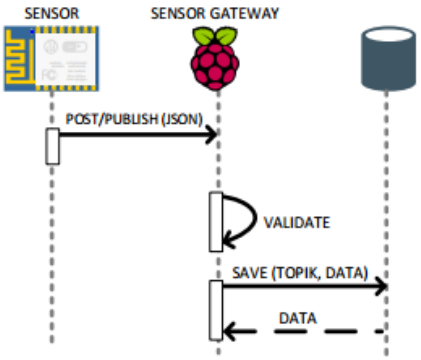
Sumber: (Anwari, Pramukantoro and Hanafi, 2017)

a. Sensor *Gateway*

Terdapat dua jenis *Gateway* pada *middleware* yaitu CoAP dan MQTT. CoAP *Gateway* menyediakan *interface* bagi sensor dalam mengirim data dengan

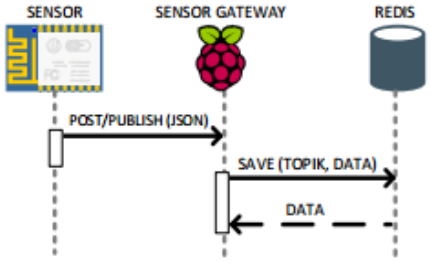
melalukan *POST request* ke *middleware*. Sedangkan MQTT *Gateway* menyediakan *interface* bagi sensor dalam mengirim data ke *middleware*. *Middleware* menerapkan pola *publish-subscribe* sehingga data yang dikirimkan memiliki satu topik. Dalam mengatasi permasalahan CoAP yang tidak mengenal istilah topik, nantinya topik akan ditempatkan sebagai bagian dari URL pada CoAP *resource* dengan *semantic* `/r/topik`.

Interaksi antara sensor dengan CoAP *Gateway* tidak jauh berbeda dengan MQTT *Gateway*. Ketika data dikirimkan dari sensor maka CoAP *Gateway* akan melalukan validasi URL untuk mencocokan dengan skema yang sudah dibuat, sedangkan pada MQTT *Gateway* tidak ada proses validasi URL karena MQTT sudah menerapkan pola *publish-subscribe*. Selanjutnya data yang telah valid akan disimpan ke dalam Redis. Untuk lebih jelasnya mengenai interaksi sensor dengan CoAP atau MQTT *Gateway* dapat dilihat pada Gambar 2.2 dan Gambar 2.3.



Gambar 2.2 *Sequence* diagram CoAP *Gateway*

Sumber: (Anwari, Pramukantoro and Hanafi, 2017)



Gambar 2.3 *Sequence* diagram MQTT *Gateway*

Sumber: (Anwari, Pramukantoro and Hanafi, 2017)

b. *Service* Unit

Service unit menyediakan *interface* untuk sensor dan *application Gateway* serta berinteraksi dengan Redis yang bertindak sebagai *broker middleware*. Berikut ini merupakan *interface* yang disediakan oleh *service unit*:

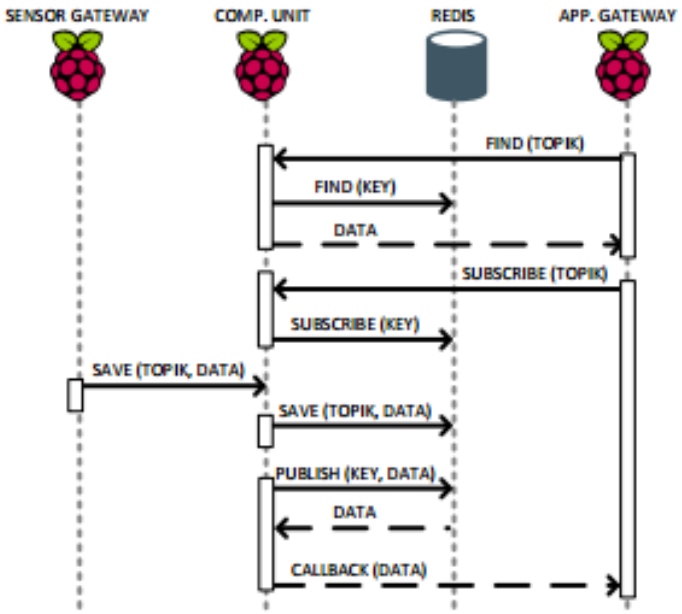
Tabel 2.1 API pada *Service* Unit

API	Parameter	Keterangan
`Find(topik)`	`topik` : nama topik	Mencari data pada Redis untuk topik tertentu
`Subscribe(topik)`	`topik` : nama topik	*Subscribe* ke topik tertentu
`Save(topik, data)`	`topik` : nama topik `data` : data yang dikirimkan	Menyimpan data untuk topik tertentu

Sumber: (Anwari, Pramukantoro and Hanafi, 2017)

Interaksi yang terjadi antara *service unit* dengan komponen lainnya dapat dilihat pada *sequence* diagram Gambar 2.4. Proses yang terjadi pada gambar 2.4 adalah ketika *sensor Gateway* mengirim data untuk sebuah topik, maka *service unit* akan menyimpan data tersebut di Redis sekaligus melakukan *publish* data. Ketika *application Gateway* telah melakukan *subscribe* untuk sebuah topik, maka *service unit* akan mencari data tersebut di Redis berdasarkan topik yang diminta. Apabila data tersebut tersedia selanjutnya *service unit* akan mengirimkan data tersebut ke aplikasi. Untuk selanjutnya ketika *sensor Gateway* mengirimkan data, maka *service unit* akan mengirimkan data tersebut ke aplikasi yang sudah *subscribe* untuk topik yang sama secara *real-time*.

Service unit juga menyediakan fungsi data *management*, di mana fungsi tersebut menyediakan abstratksi tipe data yang disimpan di Redis dengan melakukan konversi bentuk data yang dikirimkan menjadi pasangan `[key]:value` dengan skema `[topik:nama_topik]:data`.



Gambar 2.4 *Sequence* diagram *service* unit

Sumber: (Anwari, Pramukantoro and Hanafi, 2017)

c. *Application Gateway*

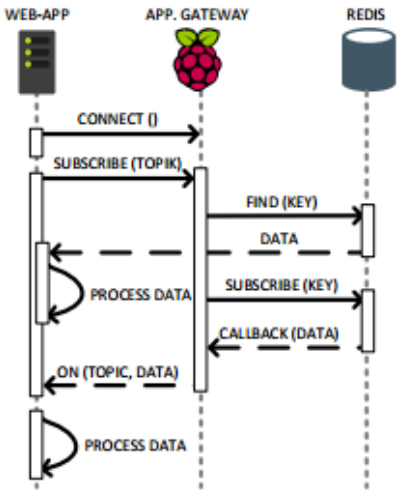
Interface yang disediakan oleh *application Gateway* yakni *interface* untuk connect ke *middleware* dan untuk *subscribe* sebuah topik.

Tabel 2.3 API pada *Application Gateway*

API	Parameter	Keterangan
`Connect()`		Terhubung dengan application *gateway* menggunakan Websocket
`Subscribe(topik)`	Topik : nama topik	*Subscribe* ke satu topik tertentu

Sumber: (Anwari, Pramukantoro and Hanafi, 2017)

Interaksi antara *application Gateway* dengan *service* unit dapat dilihat pada *Sequence* diagram berikut:



Gambar 2.5 *Sequence* diagram *application Gateway*

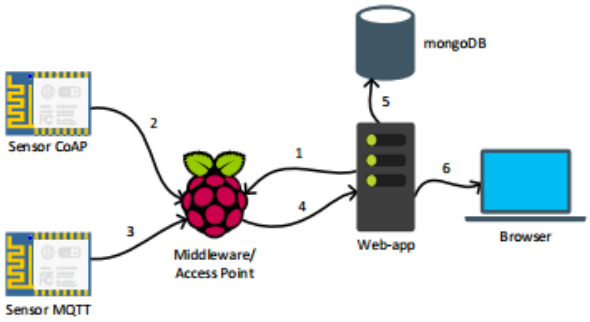
Sumber: (Anwari, Pramukantoro and Hanafi, 2017)

Proses yang terjadi pada gambar 2.5 adalah ketika aplikasi akan meminta data, pertama aplikasi harus terhubung ke *middleware* melalui *application Gateway* menggunakan *interface* `connect`. Kemudian ketika aplikasi melakukan *subscribe* sebuah topik, *application Gateway* akan melalukan *subscribe* sekaligus mencari data berdasarkan topik yang diminta. Apabila data tersedia pada Redis maka data tersebut akan dikirmkan ke aplikasi, selanjutnya setiap kali sensor mengirim data untuk topik yang sama maka aplikasi akan menerima data secara *real-time*.

d. Alur komunikasi sistem

Seperti yang sudah dijelaskan sebelumnya, penelitian oleh (Anwari, 2017) memiliki lingkungan sistem yang terdiri dari dua sensor dengan protokol CoAP dan MQTT sebagai *publisher*, *Middleware*, dan *web-app* sebagai *subscriber*.

Komponen-komponen tersebut memiliki alur komunikasi dari sensor ke *middleware* (*publish*) dan dari *middleware* ke aplikasi *web* (*subscribe*). Pada gambar 2.6 dijelaskan pertama aplikasi akan melakukan *subscribe* topik *home/kitchen* dan *home/garage* ke *middleware*. Selanjutnya sensor CoAP akan mengirimkan data suhu dan kelembapan dengan topik *home*/*kitchen* dan sensor MQTT akan mengirimkan data dengan topik *home*/*garage* ke *middleware*. Selanjutnya aplikasi menerima data dengan topik-topik yang dikirimkan oleh sensor. Selanjutnya aplikasi menyimpan data pada basis data MongoDB dan aplikasi bisa diakses menggunakan browser di laptop. Berikut ini merupakan gambar alur komunikasi sistem:

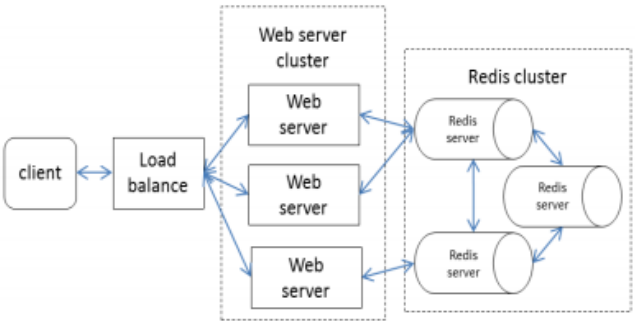


Gambar 2.6 Alur sistem

Sumber: (Anwari, Pramukantoro *and* Hanafi, 2017)

Selanjutnya Rozi melakukan penelitian lebih lanjut terhadap penelitian (Anwari, 2017) dengan melakukan pengujian performansi dan skalabilitas untuk mengetahui kinerja *Middleware* tersebut. Hasil pengujian skalabilitas menunjukkan kemampuan *Middleware* untuk menangani *publish* atau *subscribe* dalam satu detik dengan jumlah klien 100 hingga 1000 bergerak naik seiring bertambahnya jumlah klien, serta jumlah klien bertambah *concurrent Middleware.* Pada pengujian performansi untuk mengetahui kinerja *Middleware* jika data yang dikirim dari sensor diperbanyak. Hasil pengujian performansi menyatakan bahwa performansi *Middleware* untuk pengiriman data yang dikirim oleh *node* MCU CoAP lebih baik daripada *node* MCU MQTT (Rozi, Pramukantoro and Amron, 2017).

Li mengusulkan tekno*log*i pemrograman *cluster Web server* berbasis Redis untuk menjaga konsistensi *session cluster Web server* dan meningkatkan kemampuan ekspansi. Redis *cluster* digunakan untuk menyimpan *session*, sehingga *Web server* dapat mengaksesnya melalui jaringan. Hasilnya Redis *cluster* menyediakan kecepatan *read-write* data dan *availability* yang tinggi. *Web server* menjadi *stateless* dengan menyimpan *session* di Redis, yang membuatnya lebih mudah untuk menyesuaikan jumlah *Web server* dalam menyediakan pelayanan yang lebih baik bagi pengguna (Li, Jiang *and* Shi, 2017). Pada gambar 2.1 merupakan gambar system framework pada penelitian oleh (Li, Jiang and Shi, 2017):



Gambar 2.7 System Framework

Sumber ((Li, Jiang and Shi, 2017)

Pramukantoro melakukan pengujian *real-time*, performansi dan skalabilitas pada sebuah *middleware* yang mendukung permasalahan *syntactical Interoperability*. Hasil pengujian digambarkan dalam bentuk grafik sebaran data yang menunjukkan rata-rata penggunaan *CPU* dan memori dibawah 13%, waktu pengiriman dibawah 1 detik, skalabilitas pada CoAP mencapai 90 pesan/detik, MQTT 41 pesan/detik, dan *Websocket* mencapai 50 pesan/detik (Pramukantoro, Yahya *and* Bakhtiar, 2017).

Pada penelitian ini, kajian pustaka diambil dari beberapa penelitian sebelumnya. Berikut ini merupakan tabel beberapa penelitian sebelumnya yang menjadi referensi penelitian ini:

Tabel 2.1 Kajian pustaka

No	Judul Penelitian	Kesamaan	Perbedaan	
			Terdahulu	Sekarang
1	Pengembangan IoT *Middleware* Berbasis Event-Based dengan protokol komunikasi CoAP, MQTT, dan *Websocket* (Anwari, 2017)	Menggunakan protokol CoAP, MQTT, dan *Websocket*.	Menggunakan satu Redis yang digunakan sebagai *Broker* pada *Middleware*	Menggunakan tiga Redis sebagai *Broker* yang dibangun dengan metode *Cluster*
2	Analisa Performansi dan Skalabilitas pada Event-Based IoT *Middleware* (Rozi, 2017)	- Menggunakan *package* async untuk melakukan pengujian skalabilitas.	- Menggunakan grafik balok dalam menggambarkan hasil pengujian skalabilitas	- Menggunakan grafik *whisker* dalam menggambarkan hasil pengujian skalabilitas

3	Performance evaluation *of* IoT *middleware for syntactical Interoperability* (Pramukantoro, Yahya *and* Bakhtiar, 2017)	- Melakukan pengujian skalabilitas pada *middleware* - Menggunakan grafik *whisker* dalam menggambarkan sebaran data hasil pengujian skalabilitas	- Jumlah *publisher* dan *subscriber* yang digunakan dalam pengujian skalabilitas 100, 500, 1000	- Jumlah *publisher* dan *subscriber* yang digunakan dalam pengujian skalabilitas 100, 500, 1000, 1500
4	Redis-based *Web server Cluster Session* Maintaining Techno*log*y (Pramukantoro, Yahya *and* Bakhtiar, 2017)	Menggunakan Redis untuk membangun *cluster*	- Mengimplementasikan Redis *Cluster* untuk menyimpan *session* pada *cluster web* server	- Mengimplementasikan Redis *Cluster* sebagai broker pada *middleware*

2.2 Dasar Teori

Dasar teori membahas berbagai teori dasar untuk menunjang proses penilitian.

2.2.1 *Constrained Application Protocol* CoAP

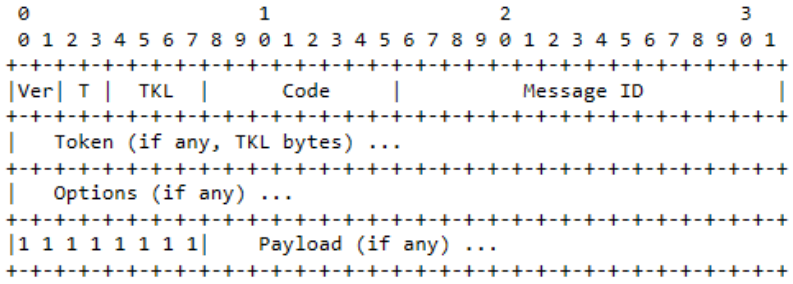
CoAP merupakan *web* transfer protokol khusus untuk digunakan pada *node* dan jaringan yang terbatas. CoAP menyediakan model *interaksi request/response* antar aplikasi, mendukung fitur *Discovery of Service* dan *resource*, serta beberapa konsep *Web* seperti URI dan tipe media Internet. Salah satu tujuan dari CoAP adalah merancang protokol *web* secara generik pada lingkungan yang terbatas, khususnya mempertimbangkan energi, *building automation* dan aplikasi M2M (IETF, The *Constrained Application Protocol* (CoAP), 2014).

Selain yang disebutkan diatas, berikut ini merupakan kelebihan yang dimiliki protokol CoAP:

1. Protokol *web* sesuai dengan persyaratan M2M dalam batasan lingkungan.
2. UDP [RFC0758] yang mendukung *reliability unicast* dan *multicast*.
3. Pertukaran pesan secara *asyncronus*.
4. *Overhead* yang rendah dan *parsing complexity*.
5. Mendukung URI dan *Content-type, proxy,* dan *caching* sederhana.

2.2.2.1 *Messaging* Model

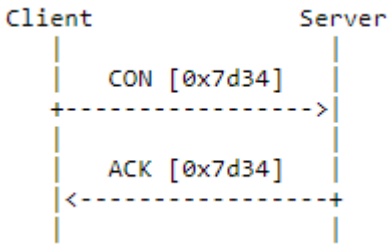
CoAP merupakan satu protokol dengan *request/respond* sebagai fitur pada CoAP *header*. Pada dasarnya CoAP dibuat berdasarkan pertukaran pesan antar *endpoint* menggunakan UDP, tetapi CoAP juga bisa digunakan pada protokol trans*port* lainnya seperti DTLS, SMS, TCP atau SCTP. Setiap pesan dalam CoAP mengandung *Message* ID yang digunakan untuk mendeteksi duplikasi jika menggunakan pilihan *reliability*.



Gambar 2.8 *Message* model *of* CoAP

Sumber (IETF, 2014)

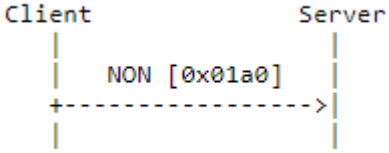
Realibility terdapat pada pesan CON (*Confirmable*). Apabila pengirim tidak mendapatkan pesan *ack* dalam waktu tertentu, maka pesan CON akan dikirimkan kembali. Namun jika penerima tidak bisa memproses pesan yang diterima maka ia akan mengirim pesan RST (*Reset*).



Gambar 2.9 Pesan *Reliable* CON

Sumber (IETF, 2014)

Pesan yang tidak memerlukan *reliability* terdapat pada pesan NON (*Non-Confirmable*). Pesan ini menandakan bahwa penerima tidak perlu mengirimkan pesan ACK, namun masih memungkinkan untuk mengirim pesan RST.



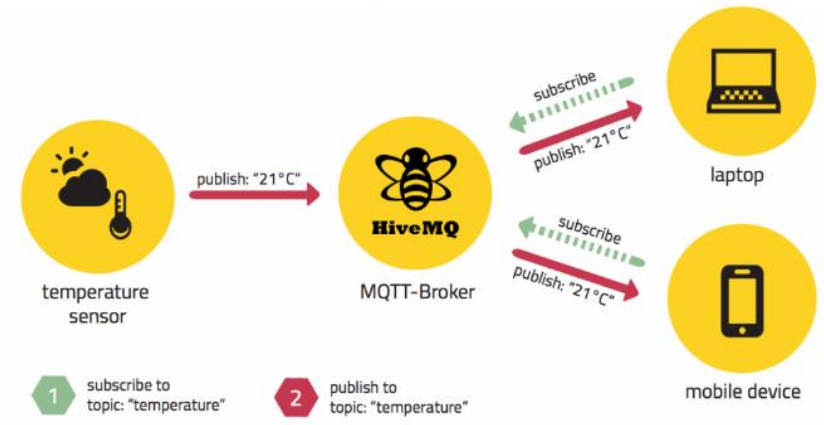
Gambar 2.10 Pesan *Unreliable* NON

Sumber (IETF, 2014)

2.2.2 MQTT

MQTT merupakan protokol komunikasi M2M atau IoT. MQTT menggunakan model *publish-subscribe* dalam komunikasinya, sehingga memberikan fleksibilitas dan kemudahan dalam implementasinya. MQTT dibangun di atas protokol TCP dan dapat mengirimkan pesan dengan tiga level *Quality of Service* (QoS). Protokol MQTT bisa mengirimkan data apapun seperti data teks, binary, XML atau JSON.

MQTT menggunakan model *publish-subscribe* yang membutuhkan dua komponen utama, yaitu MQTT klien dan MQTT *Broker*. MQTT klien terdiri dari klien yang menjadi pengirim pesan (*publisher*) dan klien yang menjadi penerima pesan (*subscriber*). MQTT *Broker* berperan dalam menghubungkan dan menangani pesan *publish-subscribe*. Dengan skema *publish-subscribe* ini membuat *publisher* dan *subscriber* tidak saling mengetahui keberadaan satu sama lain dikarenakan ada *Broker* yang berada di antara mereka, hal tersebut dinamankan *Space decoupling*. Selain itu apabila terjadi peristiwa di mana klien tiba-tiba terputus setelah melakukan *subscribe* ke *Broker* kemudian beberapa saat kemudian terhubung kembali maka klien akan tetap menerima data yang tertunda, hal tersebut dinamankan mode *offline*.



Gambar 2.11 *Publish-subscribe* pada MQTT

Sumber (HiveMQ, 2015)

Pada Gambar 2.11 merupakan Gambar skema *publish-subscribe* pada MQTT. Pertama *publisher* melakukan *subscribe* ke *Broker* untuk topik tertentu. Kemudian satu atau lebih *subscriber* melakukan *subscribe* ke *Broker* untuk topik yang sudah ada. misalnya *publisher* dengan topik A mengirim data ke *Broker*, maka selanjutnya *Broker* akan mengirimkan data tersebut ke semua klien yang telah *subscribe* untuk topik A.

2.2.3.1 *Quality of Service*

MQTT memiliki 3 level QoS di mana pesan-pesan yang di *publish* pasti memiliki salah satu dari ketiga level tersebut. Level-level tersebut memberikan

garansi konsistensi (*reliability*) dari pengiriman pesan-pesan. Berikut ini QoS yang ada pada MQTT:

a. Level 0

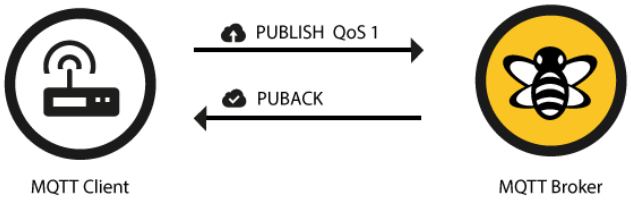
Pada level 0 pesan akan dikirimkan sekali. Pesan yang dikirim tergantung dari *reliability* TCP atau kualitas pada jaringan. QoS level 0 tidak akan mengirimkan pesan kembali apabila terjadi kegagalan dalam pengiriman pesan.

Gambar 2.12 MQTT QoS level 0

Sumber (HiveMQ, 2015)

b. Level 1

Pada level 1 pesan akan dikirimkan paling tidak sekali. Namun pesan bisa saja dikirimkan lebih dari sekali jika *subscriber* tidak mengirimkan pesan ACK.

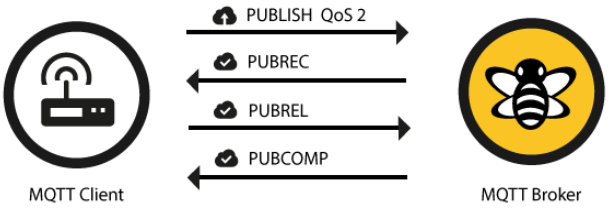


Gambar 2.13 MQTT QoS level 1

Sumber (HiveMQ, 2015)

c. Level 2

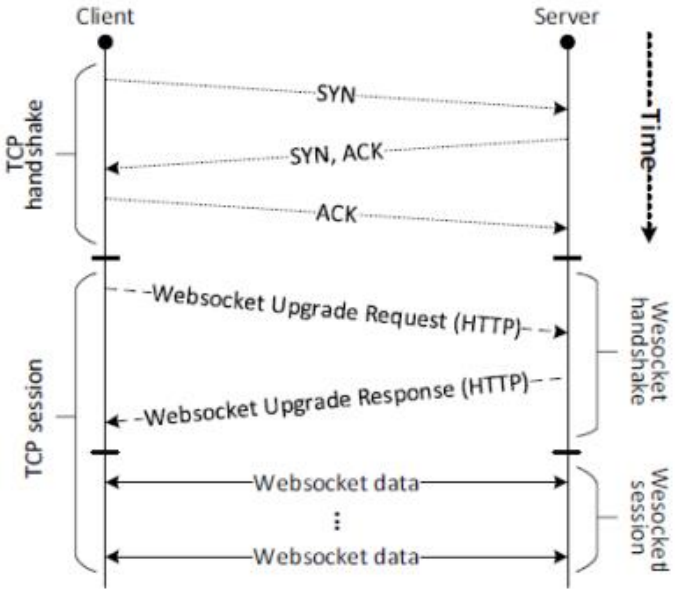
Pada level 2 pesan pasti akan diterima satu kali. MQTT dengan level ini akan memastikan bahwa pesan tersampaikan dan tidak akan terjadi duplikasi. Namun yang perlu diperhatikan adalah semakin tinggi level QoS yang digunakan maka akan semakin tinggi juga *overhead* yang terjadi.



Gambar 2.14 MQTT QoS level 2
Sumber (HiveMQ, 2015)

2.2.3 *Websocket*

Websocket di rancang untuk bekerja dengan baik dengan infrastruktur *Web* yang sudah ada. *Websocket* merupakan protokol pada layer aplikasi yang dibangun di atas koneksi TCP. Perpindahan protokol dari HTTP ke *Websocket* disebut *Websocket* handshake. *Websocket handshake* masih menggunakan standar HTTP 1.1. Gambar 2.15 dibawah ini menjelaskan *Sequence* diagram dari protokol *Websocket* serta perbandingannya degan protokol TCP. Dikarenakan *Websocket* dibangun menggunakan TCP, maka membutuhkan *handshake* terlebih dahulu untuk membuat koneksi, perbedaan dengan TCP diperlukan 3-way-*handshake* sedangkan pada *Websocket* klien akan mengirimkan *upgrade request* dan *server* membalas dengan *upgrade respond*. Dari situlah terbentuk koneksi antara *server* yang mendukung komunikasi *asynchronous* dan *full-duplex*.



Gambar 2.15 Proses komunikasi *Websocket*

Sumber (Skvorc, Horvat, & Srbljic, 2014)

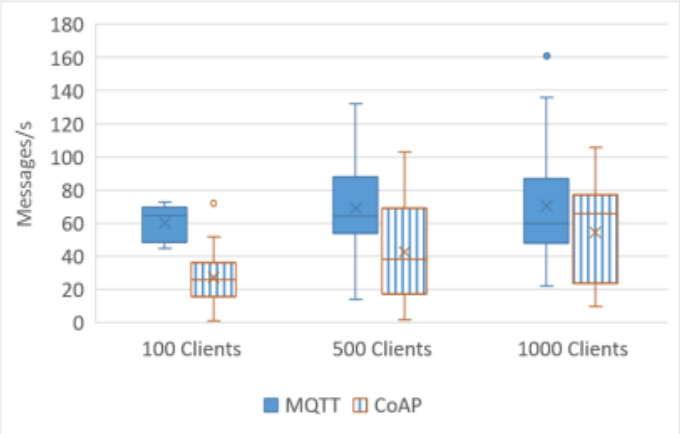
2.2.4 Skalabilitas

Sistem IoT menghubungkan sejumlah besar sensor, aktuator, dan perangkat lainnya yang menyediakan berbagai informasi dalam jumlah besar melalui internet. Hal ini menjadi tantangan bagi perancangan dan pertumbuhan sistem untuk memenuhi kebutuhan skalabilitas dan kemampuan beradaptasi terhadap perubahan lingkungan dan kebutuhan masyarakat. Pentingnya skalabilitas adalah membantu sistem bekerja dengan baik tanpa adanya *delay* dan konsumsi sumber daya yang tidak semestinya dan memanfaatkan sumber daya yang tersedia dengan baik. Dalam sistem yang *scalable*, jika kebutuhan memori sistem meningkat karena ada peningkatan jumlah data maka tidak akan terjadi ke tingkat yang tidak dapat diatasi.

Menurut Gupta skalabilitas terdapat dua jenis: Skalabilitas *Vertikal* dan Skalabilitas *Horizontal*. Skalabilitas *Vertikal* berarti meningkatkan kapasitas perangkat keras atau perangkat lunak yang ada dengan menambahkan sumber daya, contohnya menambah daya pemrosesan ke *server* supaya lebih cepat, *main memory*, *storage*, dan *network interfaces*. Keuntungan utama skalabilitas vertikal adalah konsumsi daya yang lebih kecil jika dibandingkan dengan menjalankan beberapa *server*, mengurangi upaya administratif, mengurangi biaya perangkat lunak, dan penerapannya lebih mudah. Sedangkan kekurangannya adalah risiko kegagalan perangkat keras yang lebih besar. Sedangkan skalabilitas *horizontal* berarti kemampuan untuk meningkatkan kapasitas dengan cara menghubungkan beberapa entitas perangkat keras atau perangkat lunak sehingga dapat berkerja sebagai satu unit. *Cluster* merupakan istilah yang tidak asing dalam pemrosesan skalabilitas horizontal (Gupta, Christie *and* Manjula, 2017).

Penelitian Chapuis melakukan pendekatan skalabilitas *horizontal* dalam membangun arsitektur *Middleware* terdistribusi untuk *location-based* menggunakan pola *publish-subscribe*. Dalam mengukur skalabilitasnya digunakan data *generator* yang dapat digunakan untuk melalukan uji beban secara *real time*. Hasilnya *Middleware* ini mampu memproses hingga 80.000 perbaruan lokasi per detik yang menghasilkan 25.000 kecocokan per detik (Chapuis *and* Garbinato, 2017).

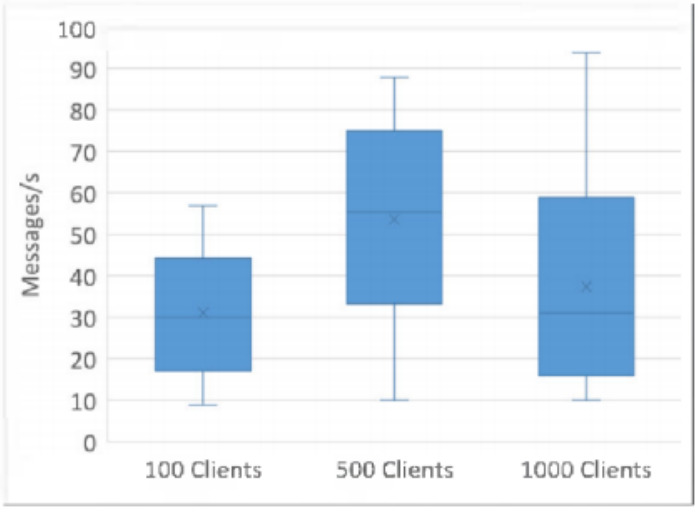
Penelitian oleh Pramukantoro melakukan pengujian *real-time*, performansi dan skalabilitas pada sebuah *middleware* yang mendukung permasalahan *syntactical Interoperability*. Pengujian skalabilitas memiliki parameter *concurrent publish* dan *concurrent subscriber*. Hasil pengujian digambarkan dalam bentuk grafik sebaran data yang menunjukkan rata-rata penggunaan *CPU* dan memori dibawah 13%, waktu pengiriman dibawah 1 detik, skalabilitas pada CoAP mencapai 90 pesan/detik, MQTT 41 pesan/detik, dan *Websocket* mencapai 50 pesan/detik (Pramukantoro, Yahya *and* Bakhtiar, 2017).



Gambar 2.16 *Concurrent Publish*

Sumber (Pramukantoro, Yahya *and* Bakhtiar, 2017)

Gambar 2.16 menunjukkan perbandingan kedua protokol CoAP dan MQTT dalam menangani sejumlah *publish* dari 100 klien hingga 1000 klien. Dapat dilihat pada gambar 2.16 menunjukkan CoAP mampu mencapai 93 pesan/detik. Pesan yang dikirim CoAP dapat meningkat sejalan dengan peningkatan jumlah pengirim. MQTT mencapai kemampuan maksimum saat jumlah klien 500, dengan maksimum 78 pesan/detik.



Gambar 2.17 *Concurrent subscriber*

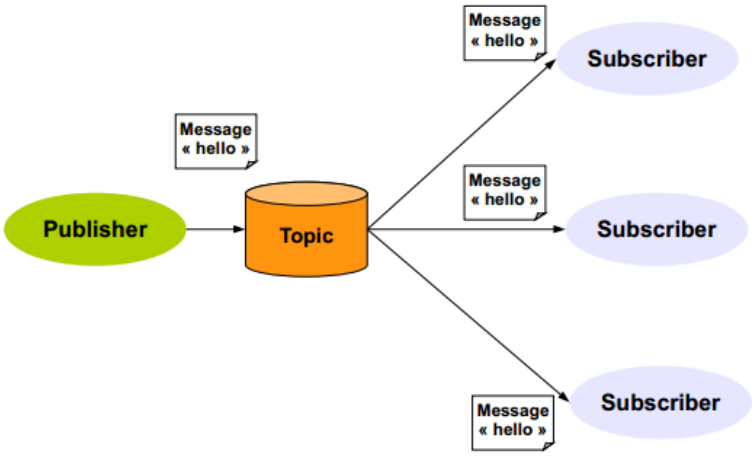
Sumber (Pramukantoro, Yahya *and* Bakhtiar, 2017)

Gambar 2.17 menunjukkan hasil pengujian skalabilitas pada sisi *subscriber*. Hasil tersebut menggambarkan kemampuan *middleware* dalam menangani sejumlah *subscriber*. Jumlah nilai *concurrent* meningkat seiring dengan jumlah *subscriber*. Pada 500+ *subscriber*, pengiriman data stabil diantara nilai 40-50 pesan/detik.

2.2.5 Arsitektur *Publish-Subscribe*

Pada arsitektur *publish-subscribe*, sebuah pesan di *publish* oleh *publisher* sekali pada topik tertentu, dan setiap *subscriber* yang terdaftar pada topik tertentu akan menerima salinan pesan tersebut (Scalagent, 2014). Pada arsitektur *publish-subscribe*, pesan yang dikirim oleh *publisher* dibagi menjadi beberapa topik yang kemudian disimpan oleh perantara atau *Broker. Subscriber* hanya menerima pesan sesuai dengan topik yang diminatinya. Dapat dikatakan *publisher* tidak perlu mengetahui siapa penerima pesan (*subscriber*) tersebut dari pesan yang dikirimnya.

Publisher dan *subscriber* tidak perlu ada pada waktu yang bersamaan saat proses pengiriman dan penerimaan pesan. Gambar 2.18 menunjukkan interaksi klien dengan pola *publish-subscribe*. Setiap klien akan berinteraksi dengan *Broker*, baik *publisher* maupun *subscriber*. *Broker* akan mengirimkan pesan yang di-*publish* ke *subscriber* yang telah berlangganan. Berikut ini merupakan gambar interaksi arsitektur *publish-subscribe*:



Gambar 2.18 Pola Interaksi *Publish-Subscribe*

Sumber: (Scalagent, 2014)

2.2.6 Redis

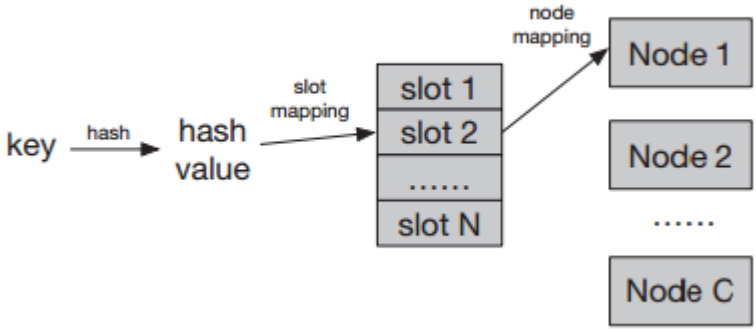
Redis merupakan *open source* (BSD license) yang dapat digunakan sebagai basis data, *cache*, atau *Broker*. Redis termasuk *in-memory* penyimpanan struktur data yang mendukung struktur data *string, hash, list, sets, bitmap* dll. Redis memiliki beberapa fitur seperti *built-in replication*, *Lua scripting*, *LRU eviction*, *transaction* dan berbagai *kever on-disk persistent*. Redis juga mendukung tingkat *Availability* yang tinggi dengan adanya Redis Sentinel dan Redis *Cluster* yang mendukung partisi otomatis. Redis dalam pola penyimpanan datanya menggunakan pasangan *key-value*, selain itu Redis juga mendukung pola *publish-subscribe*.

Penelitian ini menggunakan Redis sebagai *Broker*. Data yang ada pada Redis akan disimpan pada *memory* secara *persistent* sehingga jumlah data tidak dapat melebihi ukuran memory. Berikut ini merupakan penjelasan dari *web*site resmi Redis mengenai seberapa besar data yang dapat disimpan pada Redis:

- Redis yang tidak memiliki data membutuhkan ~ 1MB *memory*
- 1 juta *key* berbentuk pasangan *String key-value* membutuhkan ~ 100MB *memory*
- 1 juta *key* berbentuk *Hash* value atau object membutuhkan ~200MB memory.

Redis termasuk ke dalam sistem basis data *key-value* yang menggunakan strategi *key-slot-node* dalam melakukan pemetaan data. Gambar 2.18 menunjukkan keys dipetakan ke setiap *node*. Di dalam sebuah tingkatan *hash*, terdapat sebuah key, yang mana Redis akan menghitung nilai *hash* nya. Pada Redis, semua nilai *hash* akan diberikan ke setiap *slot* dan secara keseluruhan terdapat 16382 *slot*. Nilai *hash* dapat digunakan untuk mencocokan *slot* dengan *key* yang sesuai. Di Redis, semua *slot* dapat ditetapkan secara dinamis ke *node*

yang berbeda. Dalam hal ini, jika ada *node* dengan beban kerja yang berat, maka dapat dipindahkan sebagian dari *slot*nya ke *node* lain atau bahkan *node* baru untuk meningkatkan kinerja pada keseluruhan sistem. Dengan demikian pada langkah ketiga, kita dapat menemukan *node* yang sesuai dari *key* yang diberikan berdasarkan status pemetaan *slot-node* (Chen, 2016).



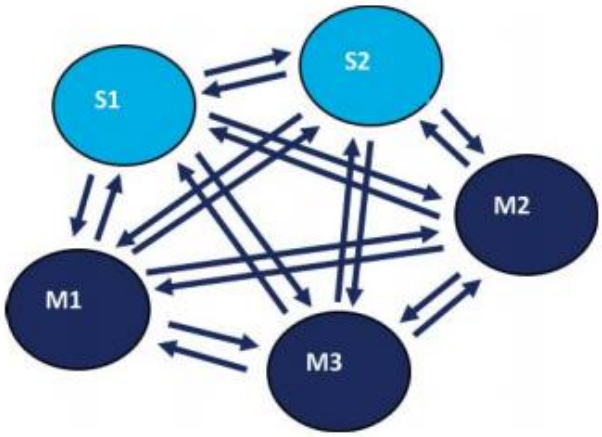
Gambar 2.19 Pemetaan *key to node* pada redis

Sumber (Chen *et al.*, 2016)

2.2.7 *Cluster Message Broker*

Penelitian ini akan menerapkan *cluster* pada Redis yang bertindak sebagai *message Broker*. *Cluster* dapat dideskripsikan sebagai penggunaan dua atau lebih sistem yang digabungkan. *Cluster* terdiri dari *server master* dan *server slave*. Redis *cluster* menyediakan cara untuk

menjalankan instalasi Redis di mana data secara otomatis di bagi ke beberapa *redis*. Redis *cluster* juga menyediakan kemampuan untuk melanjutkan operasi ketika subset dari *node* mengalami kegagalan/tidak dapat berkomunikasi dengan *cluster* lainnya (Redis, 2009).



Gambar 2.20 *Desentralized design for* redis

Sumber (Chen *et al.*, 2016)

Pada gambar 2.20 semua *node* dalam sistem terhubung satu sama lain dan mengetahui keadaan saat ini dari sistem. Setiap status di sistem berubah, info baru tersebut akan disebarkan ke setiap *node*. Jika ada *node master* yang tidak berfungsi, maka *node slave* akan menggantikan *node master* tersebut. Dengan desain seperti ini, sistem akan memiliki tingkat *reliability* yang tinggi dan klien dapat terhubung ke salah satu *node master* untuk melakukan penyimpanan data. Ketika *node master* menerima permintaan nilai *key* dari klien, maka *node master* tersebut akan memeriksa pemetaan key-*slot*-*node* untuk mencocokan nilai *key* yang ada pada *node master* tersebut. Jika cocok, maka *node master* akan memberikan nilai *key* tersebut ke klien. Jika tidak cocok, maka aka nada pesan *MOVED* yang berarti klien harus menghubungi *node* lain untuk meminta nilai *key* yang di maksud (Chen *et al.*, 2016).

BAB 3 METODOLOGI

Pada bab ini memberikan penjelasan mengenai metodologi atau langkah-langkah yang digunakan dalam penelitian. Penelitian ini merupakan pengembangan dari penelitian sebelumnya dan bersifat implementatif pengembangan. Ide pada penelitian ini adalah menerapkan *cluster message Broker* ke dalam *middleware* IoT sebagai solusi skalabilitas. Adapun langkah-langkah dalam pembuatan sistem yaitu studi literatur, analisis kebutuhan, perancangan sistem, implementasi, pengujian dan pembahasan hasil pegujian serta pengambilan kesimpulan.

Gambar 3.1 Diagram Alir Metodologi Penelitian

3.1 Studi Literatur

Studi literatur merupakan tahapan pencarian literatur dan melakukan penyusunan teori dasar dan referensi yang akan menunjang penelitian ini. Studi literatur pada penelitian ini dilakukan dengan pemahaman terhadap tinjauan pustaka dan dasar teori yang dapat diperoleh dari buku, jurnal, *web*site resmi

dan penelitian sebelumnya. Studi literatur dalam peneltian ini meliputi *middleware* pada IoT pada penelitian sebelumnya, protokol CoAP, MQTT, dan *Websocket*, skalabilitas dan metodo*log*i pengujian, Redis *Cluster*, dan arsitektur *publish subscribe*.

3.2 Analisis Kebutuhan

Analisis kebutuhan ditujukan untuk mendeskripsikan mengenai kebutuhan-kebutuhan yang diperlukan untuk menerapkan metode *cluster message Broker* sebagai solusi skalabilitas di *middleware* IoT. Analisa kebutuhan didapatkan dari studi literatur yang ada, termasuk dari penelitian sebelumnya. Hasil dari analisis kebutuhan ini nantinya akan digunakan pada tahap perancangan sistem agar menjadi lebih sistematis. Pada penelitian ini dibutuhkan lingkungan IoT yang terdiri dari *publisher*, *middleware* pada penelitian sebelumnya, dan *subscriber*. *Publisher* terdiri dari sensor suhu dan kelembapan yang menggunakan module dht11/22 dan menggunakan protokol komunikasi CoAP dan MQTT. *Middleware* pada penelitian sebelumnya beserta perangkat lunak lain seperti *framework Node* .js untuk menjalankan program java*script* pada *middleware* akan diletakkan pada *Raspberry pi*. Untuk membangun *cluster* diperlukan 3 *Raspberry pi* di mana masing-masing akan diimplementasikan redis yang berperan sebagai *master* dan *slave*. Salah satu dari ketiga *Raspberry pi* tersebut akan diletakkan *middleware* di dalamnya sehingga sistem memiliki satu *middleware* dengan beberapa Redis. *Subscriber* diletakkan pada sebuah *Raspberry pi* dengan menggunakan protokol komunikasi *Websocket* untuk *subscribe* data dari *middleware*.

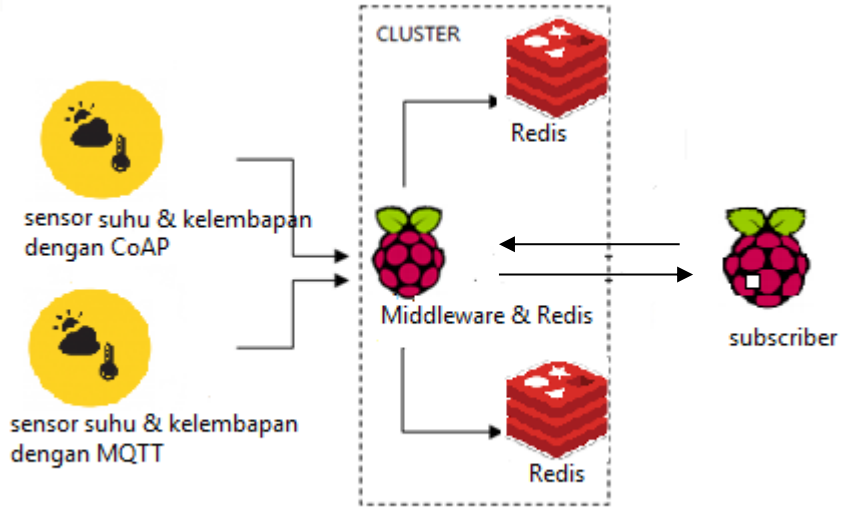
3.3 Perancangan Sistem

Perancangan sistem dilakukan setelah melewati tahap analisis kebutuhan sistem yang harus terpenuhi terlebih dahulu. Tujuan perancangan sistem agar tahapan implementasi berjalan secara sistematis dan terstruktur. Penelitian ini mengembangkan *middleware* IoT pada penelitian sebelumnya dengan ditambahkan metode *Cluster* pada *message Broker* yang digunakan yaitu Redis untuk mengatasi permasalahan skalabilitas *middleware*. Alur sistem menggunakan penelitian sebelumnya, dimulai dari *publisher* mengirimkan data suhu dan kelembapan ke *middleware* kemudian *middleware* mengirimkan data ke *subscriber* sesuai dengan topik yang sudah di *subscribe*.

Penelitian ini menambahkan metode *cluster message Broker* pada *middleware* sehingga *middleware* akan memiliki beberapa *redis* yang menunjang kinerja *middleware*. *Cluster* dibangun dari redis yang berfungsi sebagai *message Broker* dan basis data. *Cluster* terdiri dari 6 *redis*, yang mana terdapat 3 redis sebagai *master* dan 3 redis sebagai *slave*. Redis tersebut nantinya akan diletakkan pada *Raspberry pi*. Satu *Raspberry pi* akan memiliki 2 *redis*, yaitu *master* dan *slave*. Semua *redis* akan saling terhubung untuk berbagi data.

Pada gambar 3.2 menunjukkan perancangan sistem *publisher*, *cluster message Broker* pada *middleware*, dan *subscriber*. Terdapat dua sensor yang

berkomunikasi melalui protokol CoAP dan MQTT. Sensor akan menggunakan *Raspberry pi* dalam pengambilan dan pengiriman data dari sensor ke *middleware*. *Middleware* termasuk Redis akan dijalankan pada *Raspberry pi*. Selanjutnya data yang dikirimkan oleh sensor akan disimpan ke dalam *redis Cluster* dalam bentuk topik-topik secara *persistent*, dan kemudian *Middleware* akan mengirimkan data suhu dan kelembapan tersebut kepada *subscriber* setelah *subscriber* men-*subscribe* data atau topik tersebut. Program *subscriber* akan dijalankan pada *Raspberry pi*. *Subscriber* juga menggunakan protokol komunikasi *Websocket* untuk berkomunikasi dengan *middleware*.



Gambar 3.2 Perancangan sistem dengan metode *cluster*

3.4 Implementasi Sistem

Pada tahap implementasi akan mengacu pada hasil perancangan yang telah dilakukan. Pada penelitian ini akan dibuat sistem dengan komponen yang menggunakan penelitian sebelumnya, yaitu *middleware* dengan ditambahkan *cluster* redis. Langkah pertama yaitu yang dilakukan yaitu membuat beberapa *node instance* Redis berada pada mode *Cluster*. *Middleware* dan Redis tersebut akan dijalankan pada *Raspberry pi*. Kemudian membuat *Cluster* pada Redis sehingga Redis mampu berbagi data dalam bentuk topik/*key*. Selanjutnya mengintegrasikan konfigurasi *Cluster* yang sudah dibuat dengan *middleware* sebelumnya agar dapat berjalan sesuai dengan alur sistem. Langkah kedua yaitu membuat program sensor suhu dan kelembapan, di mana sensor tersebut menggunakan modul dht11/22 dan menerapkan protokol komunikasi CoAP dan MQTT. Langkah ketiga yaitu membuat program *subscriber* yang akan di gunakan untuk men-*subscribe* dan menerima data dari *middleware*. *Subscriber* tersebut menggunakan protokol komunikasi *Websocket* serta dijalankan pada *Raspberry pi*. Langkah selanjutnya menghubungkan sensor, *middleware* dan *subscriber* sehingga data yang dikirim oleh sensor dapat di terima oleh aplikasi/*subscriber*.

3.5 Pengujian dan Pembahasan Hasil Pengujian

Tahapan pengujian dan pembahasan hasil pengujian akan dilakukan untuk mengetahui apakah penerapan metode *cluster message Broker* sudah sesuai dan untuk mengetahui sejauh mana kinerja *middleware* berbasis *cluster* dalam mengatasi skalabilitas. Pengujian terdiri dari pengujian fungsional dan pengujian skalabilitas. Pengujian fungsional dilakukan berdasarkan kebutuhan fungsional yang telah ditentukan. Pengujian skalabilitas dilakukan untuk mengetahui kapasitas sistem dalam menangani berbagai proses yang diterima ketika terjadi perubahan jumlah beban yang menjadi lebih besar dari sebelumnya. Untuk mengukur tingkat kinerja *middleware* dari segi skalabilitas maka dilakukan skenario perbesaran jumlah beban *publisher* serta *subscriber* sebanyak 100, 500, 1000, dan 1500. Dalam simulasi tersebut juga dilakukan pengukuran terhadap nilai *concurrent* dan *time* dalam melakukan *publish* atau *subscribe*. Selanjutnya hasil pengujian akan di analisis untuk mengetahui tingkat kinerja *middleware* yang sudah diterapkan *cluster message broker* dalam meningkatkan skalabilitas melalui parameter *concurrent* dan *time publish* atau *subscribe*.

3.6 Kesimpulan

Pengambilan kesimpulan dilakukan setelah semua tahapan sudah berjalan yang dimulai dari tahap perancangan, implementasi, dan pengujian dan analisis. Kesimpulan dibuat untuk memberikan jawaban terhadap rumusan masalah tentang bagaimana mengimplementasikan suatu solusi dalam mengatasi masalah skalabilitas pada *middleware* IoT yaitu *cluster message Broker*. Selain itu kesimpulan juga akan akan memberikan jawaban terkait kinerja *middleware* yang telah menerapkan *cluster* dalam meningkatkan skalabilitas. Pada tahap ini juga terdapat saran untuk pengembangan sistem yang lebih sempurna baik dari kekurangan sistem ini maupun ide tambahan untuk mengembangkan sistem ini pada penelitian selanjutnya.

BAB 4 ANALISIS KEBUTUHAN

4.1 Deskripsi Umum Sistem

Penelitian ini merupakan pengembangan dari *Middleware* yang telah dibuat pada penelitian sebelumnya oleh (Anwari, Pramukantoro and Hanafi, 2017). *Middleware* yang telah dibangun pada penelitian sebelumnya bertujuan untuk memberikan pelayanan komunikasi multi protokol. Tujuan utama dari penelitian ini adalah mengatasi masalah skalabilitas saat dilakukan perbesaran jumlah klien yang melakukan *publish* atau *subscribe* ke *Middleware* tersebut. Penelitian ini akan berfokus pada bagian Redis yang bertindak sebagai *Broker*.

Sistem ini terdiri dari dua perangkat sensor yang akan mengirimkan data ke *Middleware* melalui *protocol* CoAP dan MQTT. Data yang diterima oleh *middleware* akan di bagi secara acak ke beberapa Redis yang berjalan pada mode *cluster* untuk disimpan dalam bentuk topik/*key*. Selanjutnya data yang dikirimkan oleh sensor akan dikirimkan ke aplikasi sesuai dengan topik yang telah di *subcribe* oleh aplikasi.

4.2 Lingkungan IoT

Lingkungan IoT menggunakan penelitian sebelumnya, mulai dari *publisher*, *middleware*, dan *subscriber*. *Node* sensor berfungsi sebagai *publisher*. *Node* sensor akan mengirimkan data suhu dan kelembapan menggunakan protokol CoAP dan MQTT ke *middleware* dalam format JSON. Data dengan topik `topic:home/kitchen` yang dikirim dengan menggunakan CoAP, sedangkan topik `topic:home/garage` akan dikirim dengan menggunakan MQTT. Pada *middleware* akan menggunakan Redis sebagai *Broker*. *Middleware* dan Redis dijalankan pada *Raspberry pi*. Pada sisi *subscriber* akan melakukan *subscribe* topik `topic:home/kitchen` dan `topic:home/garage` ke *middleware* sehingga setiap kali sensor mengirimkan data maka *subscriber* akan menerima data terebut secara *real-time*. Data yang akan diterima oleh *subscriber* dalam bentuk JSON.

4.3 Kebutuhan Sistem

Kebutuhan sistem bertujuan untuk mengetahui kebutuhan yang diperlukan dalam membangun sistem. Kebutuhan sistem terdiri dari kebutuhan fungsional dan kebutuhan non-fungsional.

4.3.1 Kebutuhan Fungsional

Kebutuhan fungsional adalah kebutuhan-kebutuhan yang harus dipenuhi dan proses-proses yang dapat dilakukan oleh sistem. Kebutuhan fungsional sistem dalam penelitian ini dijelaskan pada Tabel 4.1 berikut ini :

Tabel 4.1 Kebutuhan fungsional

No	Kebutuhan Fungsional
1	*Middleware* dengan mode *Cluster* mampu menerima data suhu dan kelembapan melalui *protocol* CoAP
2	*Middleware* dengan mode *Cluster* mampu menerima data suhu dan kelembapan melalui *protocol* MQTT
3	*Middleware* dengan mode *Cluster* mampu menerima data suhu dan kelembapan melalui protokol CoAP dan MQTT
4	*Cluster* mampu membagi data yang diterima *Middleware* dalam bentuk topik ke *redis* yang tergabung di dalam *Cluster*
5	*Middleware* mampu mengirimkan data suhu dan kelembapan dari sensor dengan protokol CoAP ke aplikasi *web* melalui protokol *Websocket*
6	*Middleware* mampu mengirimkan data suhu dan kelembapan dari sensor dengan protokol MQTT ke aplikasi *web* melalui protokol *Websocket*
7	*Middleware* mampu mengirimkan data suhu dan kelembapan dari sensor dengan protokol CoAP dan MQTT ke aplikasi *web* melalui protokol *Websocket*

4.3.2 Kebutuhan Non-Fungsional

Kebutuhan non-fungsional adalah kebutuhan yang berfokus pada perilaku dan batasan fungsi pada sistem. Kebutuhan non-fungsional juga berkaitan dengan kinerja dan kualitas dari sistem dari sisi skalabilitas. Skalabilitas dideskripsikan dengan bagaimana kapasitas sistem dalam menangani berbagai proses yang diterima ketika terjadi perubahan sistem yang menjadi lebih besar dari sebelumnya. Skalabilitas didefiniskan dengan banyaknya koneksi yang mampu ditangani oleh sistem. Skalabilitas Kebutuhan non-fungsional pada penelitian ini dijelaskan pada Gambar 4.3 berikut ini:

Tabel 4.2 Kebutuhan non-fungsional

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

4.3.3 Kebutuhan Perangkat Keras

Kebutuhan perangkat keras yang dibutuhkan dalam penelitian ini yaitu:

Tabel 4.3 Kebutuhan perangkat keras

Perangkat	Keterangan
Raspberry pi	*Raspberry pi* digunakan sebagai perangkat untuk menjalankan *Middleware,* redis dan juga sebagai *Access point*.
USB Adapter TL-WN725N-TP-Link	Perangkat ini digunakan sebagai pendukung *Raspberry pi* agar dapat berfungsi sebagai *Access point*.
MicroSd card 8GB	Perangkat ini digunakan sebagai tempat *Middleware* dan perangkat lunak lainnya.
DHT11/22	Modul sensor untuk membaca suhu dan kelembapan.

4.3.4 Kebutuhan Perangkat Lunak

Kebutuhan perangkat lunak yang dibutuhkan dalam penelitian ini yaitu:

Tabel 4.4 Kebutuhan perangkat lunak

Perangkat	Keterangan
Raspbian jessie	Sistem operasi untuk *Raspberry pi*
Redis	Media penyimpanan data dalam memori yang juga bertindak sebagai *Broker*
Node .js	Framework yang digunakan untuk menjalankan program *javascript* dalam *Middleware*

BAB 5 PERANCANGAN DAN IMPLEMENTASI

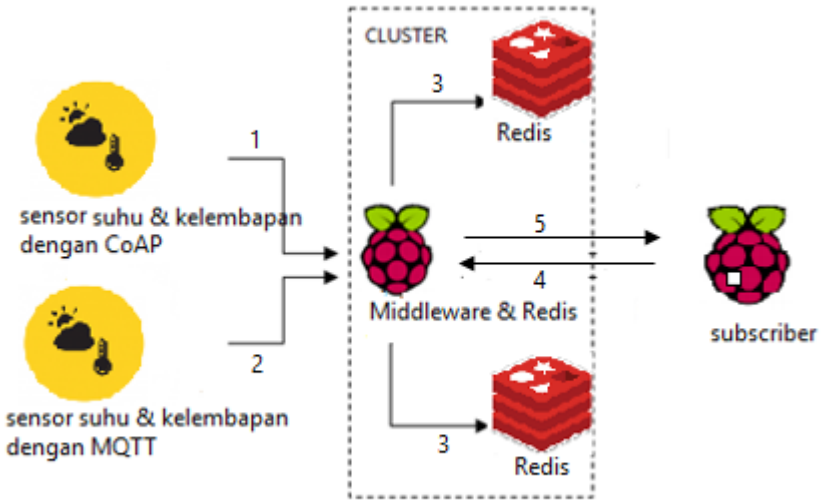
Bab ini akan menjelaskan proses perancangan dan implementasi sistem *cluster message Broker* dalam mengatasi skalabilitas pada *Internet of things* (IoT) menggunakan arsitektur *publish-subscribe*.

5.1 Perancangan Sistem

Perancangan sistem akan menjelaskan perancangan untuk sistem yang akan dibuat agar proses implementasi lebih sistematis dan terarah.

5.1.1 Perancangan Lingkungan Sistem

Dalam sistem ini terdapat tiga komponen yang saling berinteraksi yaitu sensor suhu dan kelembapan sebagai *publisher*, *Middleware,* dan *Raspberry pi* yang akan berperan sebagai *subscriber*. *Middleware* memiliki beberapa *redis* yang berjalan pada mode *cluster*. Redis difungsikan sebagai *Broker* dan basis data. Interaksi yang terjadi berupa pengiriman data dari sensor ke *Middleware* dan dari *Middleware* ke *subscriber*. Perancangan lingkungan sistem menggunakan *Middleware* yang dikembangkan dengan *cluster message Broker* dapat dilihat pada Gambar 4.1.



Gambar 5.1 Perancangan Lingkungan Sistem

Berikut ini penjelasan Gambar 5.1:

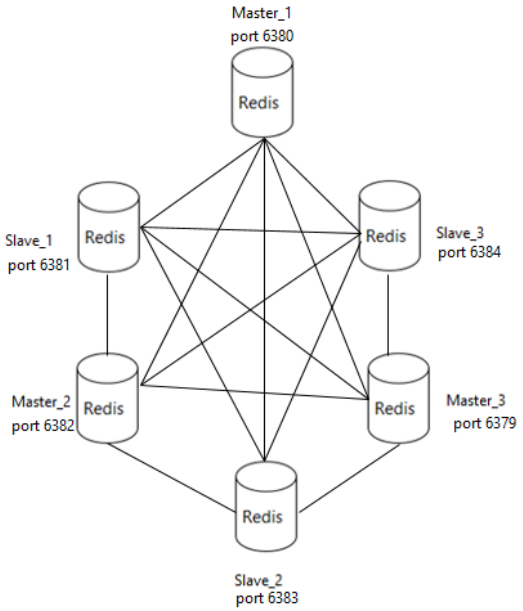
1. Sensor dht 11/22 mengirimkan data suhu dan kelembapan melalui protokol CoAP
2. Sensor dht 11/22 mengirimkan data suhu dan kelembapan melalui protokol MQTT

3. *Cluster* menyimpan data suhu dan kelembapan ke *redis* tempat di mana suatu topik/*key* di simpan.
4. *Subscriber* men*subscribe* sebuah topik tertentu ke *middleware*.
5. *Middleware* mengirimkan data suhu dan kelembapan untuk topik yang telah di *subscribe* oleh *subscriber*.

Pada sistem ini terdapat dua sensor yang akan mengirimkan data suhu dan kelembapan menggunakan protokol CoAP dan MQTT. Sensor CoAP akan mengirimkan pesan POST *request* ke *middleware*, sedangkan sensor MQTT akan mengirimkan data suhu dan kelembapan dengan topik tertentu ke *middleware* dengan mengirimkan pesan *publish* ke *middleware*. Setiap kali sensor mengirimkan data suhu dan kelembapan ke *middleware* maka *middleware* akan mengirimkan data tersebut ke *subscriber* yang telah melakukan *subscribe* untuk topik tertentu. Data tersebut akan dikirimkan ke *subscriber* secara *real-time* menggunakan protokol Websocket.

5.1.2 Perancangan *Cluster*

Pada penelitian ini *cluster message Broker* dibuat dengan topologi *mesh* di mana setiap *node* terhubung dengan menggunakan TCP *connection*. *Cluster* ini di bangun dengan menggunakan enam Redis yang ditanam pada tiga *Raspberry pi*. *Redis* tersebut dikonfigurasi agar berjalan pada mode *Cluster*. Untuk setiap Redis akan di konfigurasi sebagai *master* atau *slave*. Untuk *slave* hanya diberikan ijin untuk *Read*, yang artinya *redis slave* tersebut tidak bisa mengubah data yang disimpan pada *redis* master. *Cluster* akan berfungsi dalam pembagian data yang akan disimpan dalam bentuk topik/key. Berikut ini merupakan perancangan yang digunakan pada *cluster*:



Gambar 5.2 Perancangan *cluster*

Pada gambar 5.2 terdapat enam Redis yang saling terhubung. Redis yang saling terhubung tersebut dapat saling *Cluster* terdiri dari Redis yang memiliki *port* 6379, 6380, 6381, 6382, 6383, dan 6384. *Redis* dengan *port* 6379 dan 6384 terletak pada *Raspberry pi* dengan IP 192.168.42.28. *Redis* dengan *port* 6380 dan 6381 terletak pada *Raspberry pi* dengan IP 192.168.42.100. *Redis* dengan *port* 6382 dan 6383 terletak pada *Raspberry pi* dengan IP 192.168.42.41.

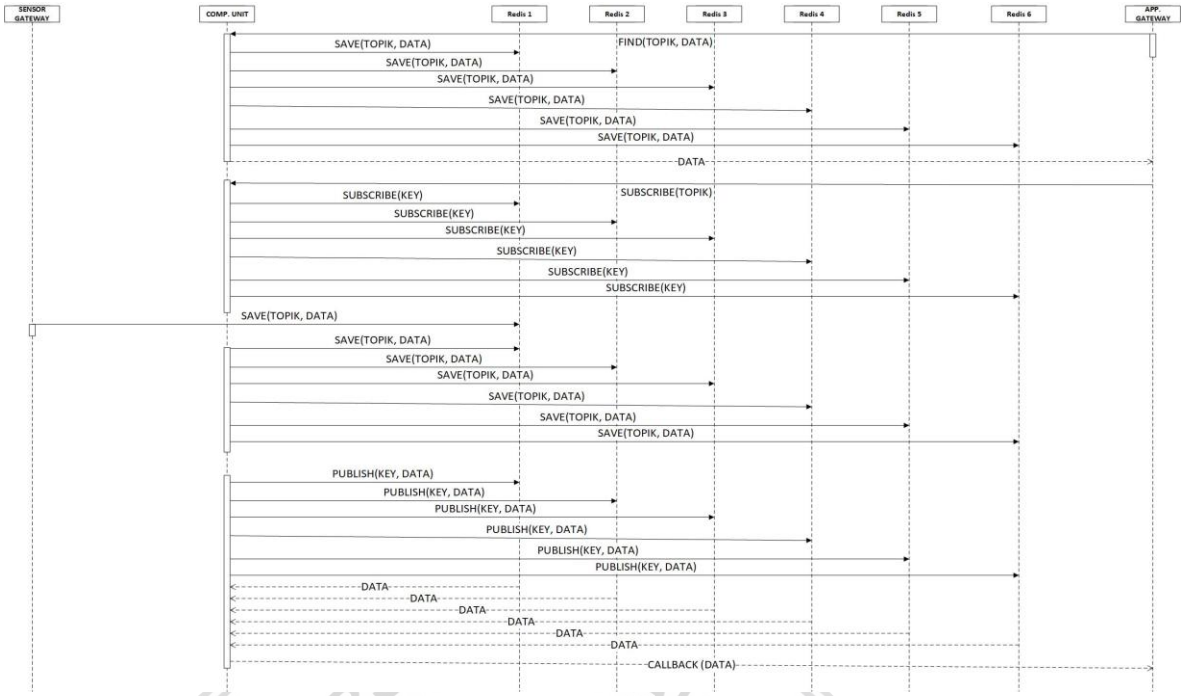
5.1.3 Perancangan *Middleware*

Perancangan alur pengiriman data dibuat pada bagian *service* unit *middleware*, di mana *service* unit merupakan komponen *middleware* yang berhubungan langsung dengan Redis. Berikut ini daftar *interface* yang disediakan oleh *service* unit:

Tabel 5.1 API *service* unit *middleware*

API	Parameter	Keterangan
Save (topik, data)	`Topik` :nama topik, data : paket yang dikirim	Menyimpan data yang dikirim dari *publisher* dengan topik tertentu
Subscriber(topik)	`Topik`: nama topik yang di *subscribe*	Melakukan *subscribe* untuk topik tertentu
Find(topik)	`Topik`: nama topik	Melakukan pencarian data pada redis dengan topik tertentu

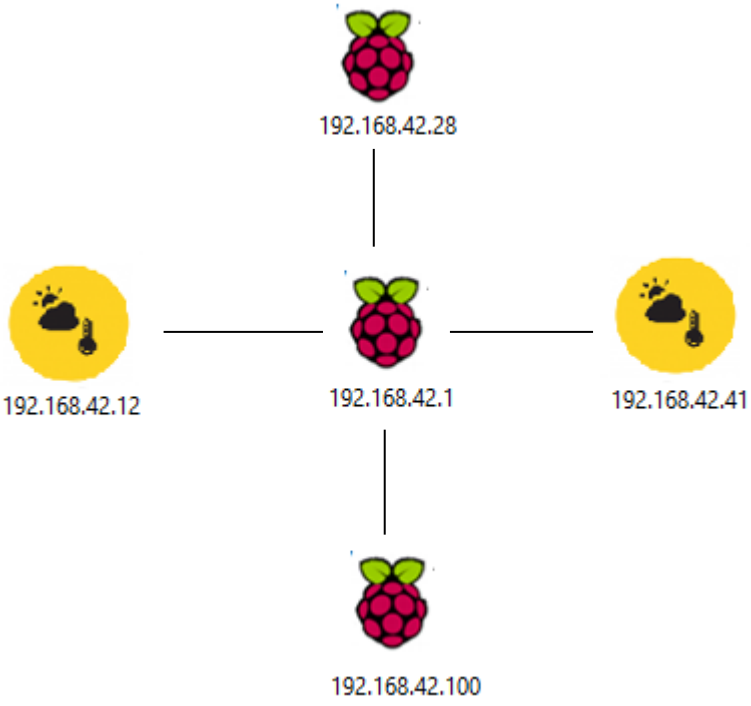
Pada penelitian ini *service* unit memberikan fungsi yang sama pada *service* unit di *middleware* penelitian sebelumnya. Namun pada saat *middleware* akan melakukan *save* data maka *middleware* akan menyimpan data tersebut pada salah satu *redis* yang ditentukan oleh *cluster*. Begitupula saat *middleware* akan melakukan *find* data untuk *subscriber* maka *middleware* akan mencari data yang diminta pada Redis untuk topik tertentu, *middleware* akan mencari pada setiap *redis* yang ada untuk menemukan topik yang dimaksut. Berikut ini merupakan *Sequence* diagram *service* unit:



Gambar 5.3 *Sequence* diagram *service* unit

5.1.4 Perancangan Pengalamatan dan Topologi Jaringan

Pada sistem ini menggunakan topologi star untuk mencerminkan bagaimana sensor terhubung dan mengirimkan data ke *Middleware* sekaligus bagaimana *Middleware* dapat mengirimkan data ke aplikasi. Rancangan pengalamatan dan topologi jaringan dapat dilihat pada gambar 5.4 di bawah ini:



Gambar 5.4 Perancangan pengalamatan dan topologi jaringan

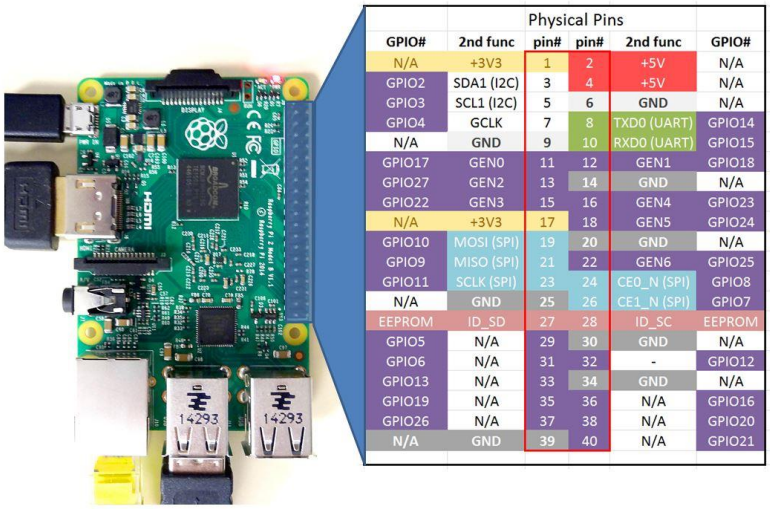
Pada gambar 5.4 terdapat Raspberry pi yang memiliki IP 192.168.42.1 yang berfungsi sebagai *Access Point*, selain itu juga terdapat Raspberry Pi dengan IP 192.168.42.100 sebagai *middleware*, IP 192.168.42.28 sebagai tempat implementasi redis, IP 192.168.42.41 dan IP 192.168.42.12 sebagai sensor yang mengirimkan data melalui CoAP dan MQTT dengan memanfaatkan Raspberr Pi untuk mengolah data. Pada *Raspberry pi* terdapat Perangkat USB Adapter TL-WN722N-TP-Link digunakan sebagai *wireless adapter*, selain itu *memory card* microSD 8GB digunakan sebagai tempat sistem operasi Raspibian Jessie dan semua perangkat lunak yang dibutuhkan.

Pada bagian pengiriman data, sensor CoAP mengirimkan data dengan topik `topic:home/kitchen` dengan mengirimkan POST *request* ke alamat `coap://192.168.42.41:5683/r/home/kitchen`. Sedangkan pada sensor MQTT mengirimkan data dengan topik `topic:home/garage` dengan melakukan *publish* ke alamat *Middleware* yakni `mqtt://192.168.42.12:1883`. Pada bagian *subscriber* menerima data dari *Middleware* dengan alamat `ws://192.168.42.100:3000`.

5.1.5 Perancangan Sensor

5.1.5.1 Perancangan Perangkat Sensor

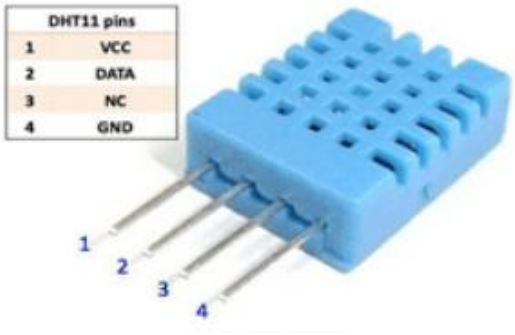
Pada penelitian ini terdapat *node* sensor yang terdiri dari *raspberry pi* dan sensor DHT. *Raspberry pi* pada sensor akan berfungsi sebagai perangkat yang berfungsi untuk mengolah data sensor dan mengirimkannya ke *middleware*. Sensor DHT11/22 adalah sensor yang bisa mendeteksi suhu dan kelembapan. Berikut ini merupakan perangkat *Raspberry pi* 2 dan susunan *layout* pinnya:



		Physical Pins			
GPIO#	2nd func	pin#	pin#	2nd func	GPIO#
N/A	+3V3	1	2	+5V	N/A
GPIO2	SDA1 (I2C)	3	4	+5V	N/A
GPIO3	SCL1 (I2C)	5	6	GND	N/A
GPIO4	GCLK	7	8	TXD0 (UART)	GPIO14
N/A	GND	9	10	RXD0 (UART)	GPIO15
GPIO17	GEN0	11	12	GEN1	GPIO18
GPIO27	GEN2	13	14	GND	N/A
GPIO22	GEN3	15	16	GEN4	GPIO23
N/A	+3V3	17	18	GEN5	GPIO24
GPIO10	MOSI (SPI)	19	20	GND	N/A
GPIO9	MISO (SPI)	21	22	GEN6	GPIO25
GPIO11	SCLK (SPI)	23	24	CE0_N (SPI)	GPIO8
N/A	GND	25	26	CE1_N (SPI)	GPIO7
EEPROM	ID_SD	27	28	ID_SC	EEPROM
GPIO5	N/A	29	30	GND	N/A
GPIO6	N/A	31	32	-	GPIO12
GPIO13	N/A	33	34	GND	N/A
GPIO19	N/A	35	36	N/A	GPIO16
GPIO26	N/A	37	38	N/A	GPIO20
N/A	GND	39	40	N/A	GPIO21

Gambar 5.5 Susunan *Layout* Pin *Raspberry pi* 2

Untuk mendapatkan nilai suhu dan kelembapan maka digunakan sensor DHT11/22. Sensor ini terdiri dari VCC untuk *power*, data untuk mengirim data, dan pin *ground*.



Gambar 5.6 *Layout* pin sensor DHT11

Rancangan instalasi pin *raspberry pi* dan DHT11 dapat dilihat pada tabel dibawah ini:

Tabel 5.2 Instalasi *Raspberry pi* dan DHT11

Type	***Raspberry pi***	**DHT 11**
Data	5	1
Vcc	2	2
Ground	6	3

5.1.5.2 Perancangan *payload*

Payload berisi informasi data yang akan dikirimkan dari sensor ke *middleware*. Struktur data yang dirancang berfungsi untuk menyamakan format *payload* data yang dikirimkan. Data dikirimkan menggunakan protokol CoAP dan MQTT. Berikut ini merupakan perancangan struktur data *payload* yang digunakan:

Tabel 5.3 Perancangan struktur data

```
Var payload = {
protocol: string, //nama protokol yang digunakan
timestamp: string, //time server
topic: string, //nama topik yang digunakan
sensor: {
        tipe: string, //tipe sensor
        index: string, //index sensor
        ip: string, //ip yang digunakan untuk sensor
        module: string, //modul yang digunakan dht11/22
        }
        humadity: {
```

```
        value: number, // nilai humadity
        unit: string, //satuan humadity
        }
        temperature: {
        value: number, //nilai temperature
        unit: string, //satuan temperature
      }
}
```

5.1.6 Perancangan Pengujian

Pada penelitian ini perancangan pengujian dibutuhkan untuk mengetahui batasan dan skenario yang digunakan dalam pengujian.

5.1.6.1 Pengujian Fungsional

Pengujian fungsional dilakukan untuk mengetahui apakah sistem yang dikembangkan sudah memenuhi kebutuhan fungsional yang sudah ditentukan. Untuk melakukan pengujian dibutuhkan skenario pengujian yang merepresentasikan beberapa kondisi yang kemungkinan terjadi. Berikut ini skenario pengujian fungsional:

Tabel 5.4 Skenario Pengujian Fungsional

Kode	Fungsi	Skenario
PF_001	*Middleware* mampu menerima data melalui protokol CoAP.	• *Node* sensor A berjalan mengirimkan data suhu dan kelembapan ke *middleware* melalui protokol CoAP. • Menjalankan *middleware* dan menjalankan *pm2 logs.* • Melalukan pengecekan apakah data telah diterima oleh *middleware* dan terekam pada program monitoring *pm2 logs.*
PF_002	*Middleware* mampu menerima data melalui protokol MQTT.	• *Node* sensor A berjalan mengirimkan data suhu dan kelembapan ke *middleware* melalui protokol MQTT. • Menjalankan *middleware* dan menjalankan *pm2 logs.* • Melalukan pengecekan apakah data telah diterima oleh *middleware* dan terekam

		pada program monitoring *pm2 logs.*
PF_003	*Middleware* mampu menerima data melalui protokol CoAP dan MQTT.	• *Node* sensor A berjalan mengirimkan data suhu dan kelembapan melalui protokol CoAP ke *middleware* dan node sensor B berjalan mengirimkan data suhu dan kelembapan melalui protokol MQTT ke *middleware.* • Menjalankan *middleware* dan menjalankan *pm2 logs.* • Melalukan pengecekan apakah data telah diterima oleh *middleware* dan terekam pada program monitoring *pm2 logs.*
PF_004	*Cluster* mampu membagi data yang diterima oleh *middleware.*	• *Node* sensor A mengirimkan data dalam bentuk topik *home/kitchen* dan *node* sensor B mengirimkan data dalam bentuk topik *home/garage.* • Menjalankan *middleware* dan menjalankan pm2 *logs.* • Melakukan pengecekan topik/*key* yang disimpan pada *redis*.
PF_005	*Middleware* mampu mengirimkan data dengan protokol CoAP ke *subscriber* melalui protokol *Websocket.*	• *Node* sensor B men-*subscribe* topik *home/kitchen* ke *middleware.* • *Node* sensor A mengirimkan data dengan topik *home/kitchen* ke *middleware* melalui protokol CoAP. • Menjalankan *middleware* dan menjalankan pm2 *logs.*
PF_006	*Middleware* mampu mengirimkan data dengan protokol MQTT ke *subscriber* melalui protokol *Websocket.*	• *Node* sensor B men*subscribe* topik *home/garage* ke *middleware.* • *Node* sensor A mengirimkan data dengan topik *home/garage* ke *middleware*

		melalui protokol MQTT. • Menjalankan *middleware* dan menjalankan pm2 *logs.*
PF_007	*Redis* mampu saling terhubung satu sama lain.	• Mengirimkan data suhu dan kelembapan ke *middleware* dengan topik yang berbeda-beda. • Menjalankan pm2 logs untuk melihat topik-topik yang masuk ke *middleware*. • Melakukan pengecekan status pada salah satu *redis*. • Melakukan akses data dengan topik/*key* yang disimpan pada *redis* lain menggunakan command MGET.

5.1.6.2 Pengujian Skalabilitas

Pengujian skalabilitas merupakan pengujian yang dilakukan untuk mengetahui seberapa besar kapasitas sistem mampu menangani berbagai proses ketika dilakukan perubahan jumlah klien pada *publisher* dan *subscriber*. Pengujian skalabilitas dilakukan dengan menggunakan *package async* dari *npm*. *Package async* akan menjalankan program secara *asynchronous*. Pada program ini akan diatur berapa banyak program yang ingin dijalankan secara *asynchronous*. Untuk memperoleh data *concurrent* dalam satu detik, peneliti menentukan jumlah *concurrent publisher* dan *subscriber* dengan variasi 100, 500, 1000, 1500. Dalam simulasi tersebut juga akan dilakukan pengukuran terhadap *concurrent publish*, *concurrent subscribe*, *time publish*, dan *time subscribe*.

Tabel 5.5 Skenario pengujian skalabilitas

PNF_001	Mengetahui berapa lama waktu yang dibutuhkan *middleware* untuk menangani sejumlah *publisher* melalui protokol CoAP dan MQTT.	Untuk pengujian dengan parameter *time publish* dengan protokol CoAP dan MQTT: a. Menjalankan pm2 *logs* pada *middleware* untuk memonitoring pesan yang masuk ke *middleware*. b. Menjalankan *filter* paket pada *middleware* menggunakan tcpdump. c. Menjalankan program pengiriman data yang berisi *package async* dengan variasi jumlah pengirim 100,

		500, 1000, dan 1500 melalui CoAP dan MQTT secara bergantian (tidak bersamaan). d. Melakukan analisa paket hasil tcpdump menggunakan *wireshark* untuk menentukan nilai *time publish* berdasarkan QoS CoAP dan MQTT yang digunakan.
PNF_002	Mengetahui berapa banyak jumlah pesan yang mampu ditangani oleh *middleware* dalam satu detik.	Untuk pengujian dengan parameter *concurrent publish* dengan protokol CoAP dan MQTT: a. Menjalankan pm2 logs pada *middleware* untuk memonitoring pesan yang masuk ke *middleware*. b. Menjalankan filter paket pada *middleware* menggunakan *tcpdump.* c. Menjalankan program pengiriman data melalui CoAP dan MQTT secara bersamaan yang dengan variasi jumlah pengirim 100, 500, 1000, 1500. d. Melakukan analisa paket hasil *tcpdump* menggunakan wireshark untuk menentukan nilai *concurrent publish* berdasarkan QoS CoAP dan MQTT yang digunakan.
PNF_003	Mengetahui berapa lama waktu yang dibutuhkan *middleware* untuk menangani sejumlah *subscriber* melalui protokol Websocket.	Untuk pengujian dengan parameter *time subscribe* dengan protokol Websocket: a. Menjalankan program pengiriman data dari sensor. b. Menjalankan pm2 *logs* pada *middleware.* c. Menjalankan *filter* paket pada *middleware* menggunakan *tcpdump.* d. Menjalankan program

		subscribe yang berisi *package* async dengan variasi jumlah 100, 500, 1000, dan 1500. e. Melakukan analisa paket hasil tcpdump menggunakan *wireshark* untuk menentukan nilai *time subscribe* dan *concurrent subscribe*.
PNF_004	Mengetahui berapa banyak jumlah pesan yang mampu ditangani oleh *middleware* dalam satu detik.	Untuk pengujian dengan *concurrent subscribe* dengan protokol Websocket: a. Menjalankan pm2 logs pada *middleware* untuk memonitoring pesan yang masuk ke *middleware.* b. Menjalankan program sensor untuk melakukan pengiriman data melalui CoAP dan MQTT. c. Menjalankan tcpdump untuk melakukan filter paket di *middleware.* d. Menjalankan program *subscriber* dengan variasi jumlah 100, 500, 1000, dan 1500. e. Melakukan analisa paket hasil *tcpdump* menggunakan wireshark untuk menentukan nilai *concurrent subscribe.*

5.2 Implementasi Sistem

5.2.1 Instalasi dan Konfigurasi Redis

Pada penelitian ini akan dilakukan instalasi Redis pada mesin Raspberry pi. Penelitian ini akan melakukan instalasi enam Redis pada tiga mesin *Raspberry pi*. Setelah dilakukan instalasi maka selanjutnya dilakukan konfigurasi pada *file* konfigurasi redis. Konfigurasi tersebut berkaitan dengan menjalankan redis pada mode *cluster*. Enam Redis tersebut akan dibedakan dengan *port* dan alamat IP. Langkah-langkah yang dilakukan dalam melakukan instalasi dan konfigurasi redis adalah sebagai beriktut:

Tabel 5.6 Perintah instalasi Redis

```
$ wget http://download.Redis.io/releases/Redis-stable.tar.gz
```

```
$ tar xzf Redis-stable.tar.gz
$ cd Redis-stable
$ make && make test
$ cd src
$ sudo cp redis-server /usr/local/bin
$ sudo cp redis-cli /usr/local/bin
```

Setelah melakukan instalasi, selanjutnya membuat direktori untuk menyimpan *file* konfigurasi Redis dan data Redis:

Tabel 5.7 Perintah Membuat Direktori Penyimpanan *File* Konfigurasi dan Data

```
$ sudo mkdir /etc/redis


$ sudo mkdir /var/redis
```

Selanjutnya menyalin *script* init yang akan ditemukan di distribusi Redis dibawah direktori utils ke /etc/init.d.

Tabel 5.8 Perintah Menyalin *Script* init

```
$ sudo cp utils/redis_init_script /etc/init.d/redis_6379
```

Selanjutnya menyalin *file* templete konfigurasi ke /etc/redis/6379.conf. kemudian membuat direktori yang akan bekerja sebagai data dan direktori kerja untuk *instance* Redis.

Tabel 5.9 Perintah Menyalin *File* Konfigurasi dan Pembuatan Direktori Kerja.

```
$ sudo cp redis.conf /etc/redis/6379.conf
$ sudo mkdir /var/redis/6379
```

Selanjutnya melakukan konfigurasi *file* redis `6379.conf`. *Port* disesuaikan untuk pembeda dengan *instance* Redis lain, *daemonize* membuat Redis dapat berjalan sebagai *Daemon* sehingga otomatis berjalan kembali setelah *Raspberry pi* dinyalakan.

Tabel 5.10 Konfigurasi Redis

```
// port disesuaikan
port 6379
daemonize yes
pidfile /var/run/redis_6379.pid
loglevel notice
dir /var/redis/6379
logfile /var/log/redis_6379.log
```

Selanjutnya tambahkan *script* init Redis baru ke semua runlevel default, kemudian jalankan *service* Redis.

Tabel 5.11 Perintah Menambahkan *Script* init dan Menjalankan Redis

```
Sudo update-rc.d redis_6379 defaults

Sudo service redis_6379 start
```

Setelah semua selesai, terakhir melakukan pengujian pada Redis untuk memastika Redis dapat berjalan dengan benar. Perintah yang dijalankan yaitu *redis-cli ping*, apabila Redis memberikan respon PONG artinya Redis sudah dapat digunakan.

Gambar 5.7 Redis pada *Raspberry pi*

5.2.2 Instalasi *Node* .js

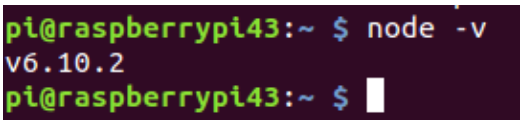
Middleware ini dikembangkan menggunakan framework *Node* .js, oleh karena itu diperlukan instalasi *Node* .js. *Node* .js akan di install pada mesin yang sama dengan Redis yaitu pada mesin *Raspberry pi* dengan OS Raspibian Jessie. Dalam penelitian ini menggunakan *Node* .js versi 6.10.2, perintah yang dibutuhkan yaitu:

Tabel 5.12 Perintah Instalasi *Node* .js

```
$ curl -sL https://deb.node source.com/setup_6.x | sudo -E
bash -

$ sudo apt-get install -y node js
```

Setelah itu untuk memastikan *Node* .js sudah terinstall dengan benar, kita dapat menjalankan perintah *node* -v untuk melihat versi *Node* .js yang terinstall.



Gambar 5.8 Pengecekan Versi *Node* .js

5.2.3 Implementasi *Cluster*

Pada penelitian ini mengimplementasikan *Cluster* pada *message Broker*, di mana *message Broker* yang digunakan pada *middleware* adalah Redis. *Cluster* menyediakan cara untuk menjalankan Redis di mana data akan secara otomatis

dibagi melalui multiple Redis. *Cluster* dibangun dengan enam Redis yang terinstall di tiga mesin *Raspberry pi*.

Cluster terdiri dari *node* dengan *port* 6379, 6380, 6381, 6382,6383, dan 6384. *Redis* master_3 dengan *port* 6379 dan 6384 terletak pada *Raspberry pi* dengan IP 192.168.42.28. *Redis* master_1 dengan *port* 6380 dan 6381 terletak pada *Raspberry pi* dengan IP 192.168.42.100. *Redis* master_2 dengan *port* 6382 dan 6383 terletak pada *Raspberry pi* dengan IP 192.168.42.41. Masing-masing *redis* tersebut memiliki *file* konfigurasi sendiri.

File konfigurasi terdapat pada `/etc/redis`. Ada beberapa konfigurasi yang perlu dilakukan pada *file* Redis masing-masing *node* supaya *Cluster* dapat berjalan dengan benar. Berikut ini beberapa hal yang ada pada *file* konfigurasi *Cluster* Redis:

Tabel 5.13 *File* Konfigurasi *Cluster*

```
port 6379
cluster-enabled yes
cluster-config-file node -6379.conf
cluster-node -timeout 5000
appendonly yes
```

Setelah melakukan konfigurasi *Cluster* tersebut, selanjutnya melakukan pengecekan status apakah *redis* sudah berjalan pada mode *Cluster* dengan benar. Perintah yang digunakan `sudo service redis_6379` status dan melakukan pengecekan proses Redis dengan perintah `ps aux | grep redis`.

```
pi@raspberrypi43:~ $ sudo service redis_6379 status
● redis_6379.service - LSB: start and stop redis_6379
   Loaded: loaded (/etc/init.d/redis_6379)
   Active: active (running) since Thu 2018-05-10 10:19:39 WIB; 13h ago
  Process: 421 ExecStart=/etc/init.d/redis_6379 start (code=exited, status=0/SUC
CESS)
   CGroup: /system.slice/redis_6379.service
           └─435 /usr/local/bin/redis-server 0.0.0.0:6379 [cluster]

May 10 10:19:38 raspberrypi43 redis_6379[421]: Starting Redis server...
May 10 10:19:38 raspberrypi43 redis_6379[421]: 425:C 10 May 10:19:38.707 # o...o
May 10 10:19:38 raspberrypi43 redis_6379[421]: 425:C 10 May 10:19:38.721 # R...d
May 10 10:19:38 raspberrypi43 redis_6379[421]: 425:C 10 May 10:19:38.721 # C...d
May 10 10:19:39 raspberrypi43 systemd[1]: Started LSB: start and stop redis_....
Hint: Some lines were ellipsized, use -l to show in full.
```

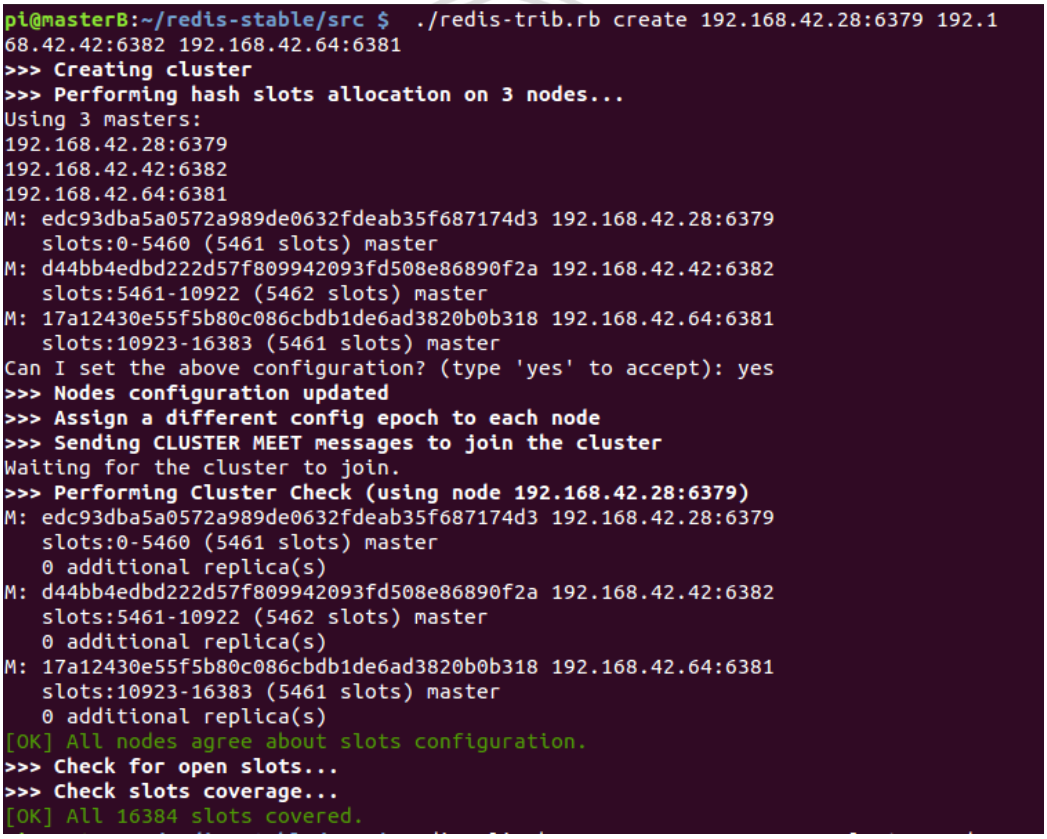
Gambar 5.9 Pengecekan Status Redis

Selanjutnya membuat *Cluster* pada Redis menggunakan `redis-trib`. Redis-trib merupakan program Ruby yang mengeksekusi perintah khusus pada *instance* dalam membuat *Cluster* baru, memeriksa atau me-*reshard Cluster* yang

ada dan sebagainya. Sebelum bisa menjalankan `redis-trib`, harus melakukan instalasi *redis gem*.

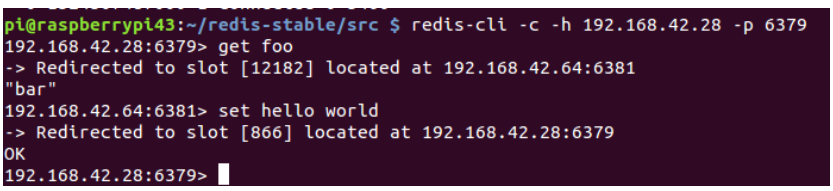
Gambar 5.10 Instalasi Redis Gem

Setelah selesai melakukan instalasi Redis Gem, selanjutnya membuat *Cluster* dengan perintah ./`redis-trib.rb create`. perintah tersebut digunakan untuk membuat *Cluster* pada beberapa *redis*. Dibawah ini merupakan perintah `redis-trib` untuk IP 192.168.42.28 dengan *port* 6379, IP 192.168.42.42 dengan *port* 6382, dan IP 192.168.42.100 dengan *port* 6381. Selain itu `redis-trib` juga akan mengkonfigurasi *key slot* secara otomatis untuk ketiga *redis* tersebut. Ketiga *redis* tersebut akan memiliki peran sebagai *master* pada *Cluster*.



Gambar 5.11 Implementasi *Cluster* Menggunakan Redis-trib

Selanjutnya melakukan pengecekan apakah *Cluster* sudah berjalan dengan benar. Pengecekan dapat dilakukan dengan *redis-cli* dan menjalankan perintah SET dan GET.



Gambar 5.12 SET dan GET *Cluster* Redis

Selanjutnya untuk mendukung penanganan *node* failure di Redis, setiap *master* akan diberikan satu *slave*. *Slave* tersebut akan berguna jika *node master* mengalami kegagalan. Pada penelitian ini *redis* yang berperan sebagai *slave* yang dibuat memiliki *port* 6380, 6383, dan 6384.

Redis slave akan ditambahkan sebagai anggota *Cluster* dengan perintah ./redis-trib add-*node* -*slave* -master-id [id-*node* -master] IP-*slave*:*port-slave* IP-master:*port*-master.

```
pi@pi2A:~/redis-stable/src $ ./redis-trib.rb add-node --slave --master-id 345fde
53b644bc71e4da239edb0270c2fdfddefd 192.168.42.42:6383 192.168.42.42:6382
>>> Adding node 192.168.42.42:6383 to cluster 192.168.42.42:6382
>>> Performing Cluster Check (using node 192.168.42.42:6382)
M: 345fde53b644bc71e4da239edb0270c2fdfddefd 192.168.42.42:6382
   slots:5461-10922 (5462 slots) master
   0 additional replica(s)
M: 17a12430e55f5b80c086cbdb1de6ad3820b0b318 192.168.42.100:6381
   slots:10923-16383 (5461 slots) master
   0 additional replica(s)
M: f9d83b1f35deab790f8800a60560b1a95c1cfb70 192.168.42.28:6379
   slots:0-5460 (5461 slots) master
   0 additional replica(s)
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
>>> Send CLUSTER MEET to node 192.168.42.42:6383 to make it join the cluster.
Waiting for the cluster to join.
>>> Configure node as replica of 192.168.42.42:6382.
[OK] New node added correctly.
pi@pi2A:~/redis-stable/src $
```

Gambar 5.13 Add-*node Slave Cluster*

5.2.4 Implementasi *Interface Raspberry pi*

Pada penelitian ini implementasi topologi jaringan akan membahas konfigurasi yang dibutuhkan oleh *Raspberry pi* dalam membangun sistem *Cluster* untuk pengujian *middleware*. Konfigurasi yang dilakukan yaitu mengatur alamat IP pada *interface* wlan0 menjadi *static*. *Raspberry pi* yang digunakan terhubung ke wpa-ssid The*Middleware*. Berikut ini konfigurasi *interface* wlan0 pada *file* /etc/network/*interfaces*

Tabel 5.14 Implementasi *Interface Raspberry pi*

```
auto wlan0
iface wlan0 inet static
     wpa-ssid TheMiddleware
     wpa-psk lumbalumba
     address 192.168.42.100 //IP address disesuaikan dengan
masing-masing raspberry
     netmask 255.255.255.0
     broadcast 192.168.42.255
```

5.2.5 Impelementasi *Middleware*

Pada penelitian ini membangun *cluster* pada *message Broker* yang ada pada penelitian sebelumnya yaitu Redis, sehingga setelah Redis sudah dalam keadaan *Cluster* maka perlu diintegrasikan dengan *middleware* sebelumnya. Pada sub bab 2.1 sudah dijelaskan bahwa Redis merupakan bagian dari *service unit middleware*. Redis berfungsi sebagai *Broker* sekaligus melakukan penyimpanan data secara *persistent*.

Middleware terletak pada *Raspberry pi* dengan IP 192.168.42.100. Integrasi *middleware* dengan *cluster* redis dilakukan dengan menambahkan *library Ioredis*. Hal tersebut dikarenakan *library ioredis* sudah mendukung *Cluster*. Selain itu juga dilakukan inisialisasi semua *port* dan IP host redis yang ada pada daftar *cluster*. Hal tersebut dilakukan agar *middleware* mengetahui redis mana saja yang berada pada daftar *cluster*. *Port* dan IP host redis terdiri dari *node* dengan IP 192.168.42.100 dengan *port* redis 6380 dan 6381, *node* dengan IP 192.168.42.41 dengan *port* 6382 dan 6383, dan *node* dengan IP 192.168.42.28 dengan *port* 6379 dan 6384. Berikut ini merupakan perubahan yang dilakukan pada program server.js:

Tabel 5.15 *Pseudocode* program server.js

```
  SET ioredis = require ioredis
  SET app.ascoltatore = new ascoltatori.RedisAscoltatore(
    {port: port, host: 'IP Address', redis: ioredis}, {port:
port, host: 'IP Address', redis: ioredis}, {port: port, host:
'IP Address', redis: ioredis}
   )
    return app.ascoltatore
  }
  module.exports.setup = setup = (opts) => {
    let args
    if (opts == null) {
      opts = {}
    }
    args = [opts.port, opts.host]
    SET app.redis.klien = new ioredis.Cluster([{port: port,
host: 'IP Address'}, {port: port, host: 'IP Address'}, {port:
port, host: 'IP Address'}])
    return setupAscoltatore(opts)
  }
```

5.2.6 Implementasi Sensor

Implementasi sensor akan menjelaskan proses pembuatan perangkat lunak sensor. Program sensor menggunakan *node* .js yang dijalankan pada *Raspberry pi*. Perbedaan dari pengiriman data menggunakan protokol CoAP dan MQTT adalah ketika menggunakan CoAP data yang telah dibaca oleh sensor akan langsung dikirim ke *middleware*, sedangkan ketika menggunakan MQTT sensor harus terhubung dahulu ke *middleware* sebelum mengirimkan data yang telah berhasil terbaca oleh sensor.

Tabel 5.16 merupakan *Pseudocode* program sensor yang mengirimkan data suhu dan kelembapan melalui protokol CoAP ke *middleware* dengan IP 192.168.42.100 dan *port* 5683 untuk CoAP. Pada proses ini data akan langsung dikirimkan ke *middleware* tanpa sensor harus terhubung ke *middleware* terlebih dahulu.

Tabel 5.16 *Pseudocode* program sensor dengan protokol CoAP

```
SET sensorlib = require node -dht-sensor
SET temp = read temperature from sensor
SET hum = read humidity from sensor
DEFINE coapp ()
   SET coap = require coap
   SET cron = require node -cron
   SET payload = payload data from sensor
   DO send payload in JSON with method POST
   DO print Date and "terkirim"
END
```

Pada tabel 5.17 merupakan *Pseudocode* program sensor yang mengirimkan data suhu dan kelembapan melalui protokol MQTT ke *middleware* dengan IP 192.168.42.100 dan *port* 1883 untuk MQTT. Proses ini akan mengirimkan data sensor setelah sensor melakukan koneksi dan terhubung ke MQTT *Gateway* pada *middleware*. Pada program ini juga di tentukan menggunakan QoS 1 pada MQTT.

Tabel 5.17 Kode program sensor dengan protokol MQTT

```
SET async = require async
SET MQTT = require mqtt
SET payload = payload data and read humidity and temperature
from sensor
DO async (1, next)
   DO print Date in String
   DO connect to MQTT Gateway using ip and port
```

```
   DO publish (topik, payload in JSON, use qos 1)
END
DO next (err)
   DO print "terkirim"
END
```

5.2.7 Implementasi *Subscriber*

Program *subscriber* dibuat dengan menggunakan bahasa javascript. *Subscriber* dijalankan pada *Raspberry pi*. Program *subscriber* juga mengimplementasikan modul async yang digunakan pada pengujian *concurrent subscriber*. Program *subscriber* akan melakukan koneksi ke *web*scoket *Gateway* pada *middleware* dengan IP 192.168.42.100 dan *port* 3000. Berikut ini kode program *subscriber*:

Tabel 5.18 *Pseudocode* program *subscriber* dengan protokol *Websocket*

```
SET async = require async
DO async (1, next)
    DO connect to Websocket Gateway using ip and port
    On socket ('/r/topik', function (data)
      DO print ('oke'+ data)
    END
    DO socket.emit ('subscribe', topik, function ()
    END
END
DO next (err)
   DO print "bisa"
END
```

BAB 6 PENGUJIAN DAN PEMBAHASAN HASIL PENGUJIAN

Setelah proses implementasi, langkah selanjutnya yaitu melakukan pengujian pada *middleware*. Pengujian ini dilakukan untuk mengetahui apakah implementasi yang dilakukan sudah sesuai dan mengetahui bagaimana kinerja *middleware* dari segi skalabilitas dalam menangani jumlah beban yang diberikan. proses pengujian terbagi menjadi yaitu pengujian fungsional dan pengujian skalabilitas.

6.1 Pengujian Fungsionalitas

Pengujian fungsional dilakukan untuk melihat kesesuaian fungsi-fungsi hasil implementasi dengan perancangan yang sudah dibuat. Pengujian fungsional juga menunjukkan bagaimana integrasi antara *middleware* dengan *cluster* dan sensor serta *subscriber*. Dari hasil pengujian fungsional dapat di tentukan apakah fungsi yang ada telah berjalan dengan benar. Pengujian dilakukan berdasarkan prosedur pengujian yang telah ditentukan. Pada pengujian fungsional akan dilakukan pengiriman data suhu dan kelembapan dari sensor dengan IP 192.168.42.41 untuk CoAP dan sensor dengan IP 192.168.42.12 untuk mengirim data suhu dan kelembapan melalui MQTT ke *middleware* dengan IP 192.168.42.100. Sensor tersebut akan mengirimkan data dengan topik tertentu sesuai dengan prosedur pengujian. Program sensor yang digunakan dijelaskan pada tabel 5.16 dan 5.17, sedangkan untuk program *middleware* yang digunakan sudah dijelaskan pada tabel 5.15. Untuk pengujian yang berkaitan dengan protokol *Websocket* atau *subscriber* program yang digunakan sudah dijelaskan pada tabel 5.18.

6.1.1 *Middleware* mampu menerima data melalui protokol CoAP

Tabel 6.1 *Middleware* mampu menerima data melalui protokol CoAP

Kode	PF_001
Nama Kasus Uji	*Middleware* mampu menerima data melalui protokol CoAP
Tujuan Pengujian	Mengetahui Apakah *Middleware* yang telah mengimplementasikan *cluster message Broker* mampu menerima data suhu dan kelembapan melalui protokol CoAP
Prosedur Pengujian	1. *Node* sensor A berjalan mengirimkan data suhu dan kelembapan ke *middleware* melalui protokol CoAP 2. Menjalankan *middleware* dan menjalankan *pm2 logs*

	3. Melalukan pengecekan apakah data telah diterima oleh *middleware* dan terekam pada program monitoring *pm2 logs*
Hasil yang diharapkan	Data yang dikirimkan oleh *node* sensor dapat diterima oleh *middleware*
Hasil pengujian	Data yang dikirimkan oleh *node* sensor dapat diterima oleh *middleware*
Keterangan	Valid

Pada Gambar 6.1 menunjukkan tampilan pm2 *logs* pada *middleware*. Hasil pengujian menunjukkan bahwa *middleware* dengan mode *cluster* mampu menerima data suhu dan kelembapan melalui *protocol* CoAP untuk topik *home/kitchen* dari klien *publisher* dengan IP 192.168.42.41. Berdasarkan hasil pengujian PF_001 disimpulkan bahwa integarasi *middleware* yang mengimplementasikan *cluster message Broker* dengan *node* sensor berhasil dilakukan.

```
0|qoap     | 2018-7-15 16:33:54 COAP - Incoming POST request from 192.168.42.41 for home/kitchen
0|qoap     | 2018-7-15 16:34:06 COAP - Incoming POST request from 192.168.42.41 for home/kitchen
0|qoap     | 2018-7-15 16:34:09 COAP - Incoming POST request from 192.168.42.41 for home/kitchen
0|qoap     | 2018-7-15 16:34:11 COAP - Incoming POST request from 192.168.42.41 for home/kitchen
```

Gambar 6.1 Hasil pengujian skenario PF_001

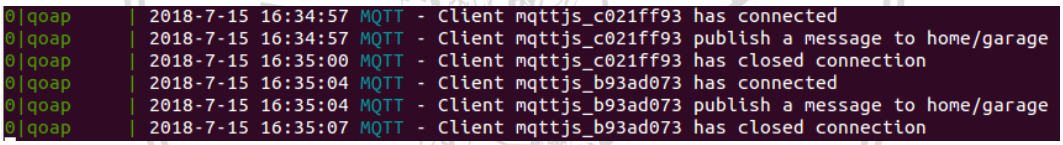
6.1.2 *Middleware* mampu menerima data melalui protokol MQTT

Tabel 6.2 *Middleware* mampu menerima melalui protokol MQTT

Kode	PF_002
Nama Kasus Uji	*Middleware* mampu menerima data melalui protokol MQTT
Tujuan Pengujian	Mengetahui apakah *Middleware* yang telah mengimplementasikan metode *cluster message Broker* mampu menerima data suhu dan kelembapan melalui protokol MQTT
Prosedur Pengujian	1. *Node* sensor A berjalan mengirimkan data suhu dan kelembapan ke *middleware* melalui protokol MQTT 2. Menjalankan *middleware* dan menjalankan *pm2 logs* 3. Melalukan pengecekan apakah

	data telah diterima oleh *middleware* dan terekam pada program monitoring *pm2 logs*
Hasil yang diharapkan	Data yang dikirimkan oleh *node* sensor dapat diterima oleh *middleware*
Hasil pengujian	Data yang dikirimkan oleh *node* sensor dapat diterima oleh *middleware*
Keterangan	Valid

Pada Gambar 6.2 menunjukkan tampilan pm2 *logs* pada *middleware*. Hasil pengujian menunjukkan bahwa *middleware* dengan mode *cluster* mampu menerima data suhu dan kelembapan melalui *protocol* MQTT untuk topik *home/garage* dari klien *publisher* dengan IP 192.168.42.41. Berdasarkan hasil pengujian PF_002 disimpulkan bahwa integarasi *middleware* yang mengimplementasikan *cluster message Broker* dengan *node* sensor berhasil dilakukan.



Gambar 6.2 Hasil pengujian skenario PF_002

6.1.3 *Middleware* mampu menerima data melalui protokol CoAP dan MQTT

Pengujian dengan kode PF_001 dilakukan dengan menggunakan program sensor yang sudah dijelaskan pada *Pseudocode* di tabel 5.16 dan 5.17.

Tabel 6.3 *Middleware* mampu menerima data melalui protokol CoAP dan MQTT

Kode	PF_003
Nama Kasus Uji	*Middleware* mampu menerima data melalui protokol CoAP dan MQTT
Tujuan Pengujian	Mengetahui apakah *Middleware* yang telah mengimplementasikan *cluster message Broker* mampu menerima data suhu dan kelembapan melalui protokol CoAP dan MQTT
Prosedur Pengujian	1. *Node* sensor A berjalan mengirimkan data suhu dan

	kelembapan ke *middleware* melalui protokol CoAP dan *Node* sensor B berjalan mengirimkan data suhu dan kelembapan ke *middleware* melalui protokol MQTT 2. Menjalankan *middleware* dan menjalankan *pm2 logs* 3. Melalukan pengecekan apakah data telah diterima oleh *middleware* dan terekam pada program monitoring *pm2 logs*
Hasil yang diharapkan	Data yang dikirimkan oleh *node* sensor dapat diterima oleh *middleware*
Hasil pengujian	Data yang dikirimkan oleh *node* sensor dapat diterima oleh *middleware*
Keterangan	Valid

Pada Gambar 6.3 menunjukkan tampilan pm2 *logs* pada *middleware*. Hasil pengujian menunjukkan bahwa *middleware* dengan mode *cluster* mampu menerima data suhu dan kelembapan melalui *protocol* CoAP untuk topik *home/kitchen* dari klien *publisher* dengan IP 192.168.42.41 dan melalui protokol MQTT untuk topik *home/garage* dari klien *publisher* mqtjs_a2830b21. Berdasarkan hasil pengujian PF_003 disimpulkan bahwa integarasi *middleware* yang mengimplementasikan *cluster message Broker* dengan *node* sensor berhasil dilakukan.

```
0|qoap     | 2018-7-15 16:36:02 COAP - Incoming POST request from 192.168.42.41 for home/kitchen
0|qoap     | 2018-7-15 16:36:02 MQTT - Client mqttjs_a2830b21 has connected
0|qoap     | 2018-7-15 16:36:02 MQTT - Client mqttjs_a2830b21 publish a message to home/garage
```

Gambar 6.3 Hasil pengujian skenario PF_003

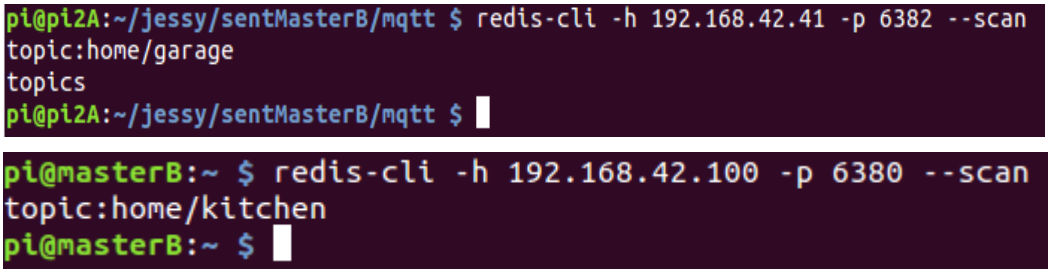
6.1.4 *Cluster* mampu membagi data yang diterima oleh *Middleware*

Tabel 6.4 *Cluster* mampu membagi data yang diterima oleh *Middleware*

No Uji	PF_004
Tujuan Pengujian	*Cluster* mampu membagi data yang diterima *middleware* ke beberapa *redis* dalam bentuk topik/*key*
Prosedur Pengujian	1. *Node* sensor A mengirimkan data dalam bentuk topik

	home/*kitchen* dan *node* sensor B mengirimkan data dalam bentuk topik *home*/*garage* 2. Menjalankan *middleware* dan menjalankan pm2 *logs* 3. Melakukan pengecekan isi topik/*key* pada *redis*
Hasil yang diharapkan	Data yang dikirimkan dalam bentuk topik/*key* yang bermacam-macam mampu di bagi dan disimpan ke beberapa *redis*
Hasil pengujian	Data yang dikirimkan dalam bentuk topik/*key* yang bermacam-macam mampu di bagi dan disimpan ke beberapa *redis*
Keterangan	Valid

Pada Gambar 6.4 merupakan tampilan yang menunjukkan hasil perintah *redis-cli -h [IP] -p [port] –scan*. Perintah tersebut berfungsi untuk melihat topik apa saja yang ada pada suatu *redis*. Hasil pengujian menunjukkan pada *redis* dengan IP 192.168.42.41 dan *port* 6382 menyimpan topik *home*/*garage*, sedangkan *redis* dengan IP 192.168.42.100 dan *port* 6380 menyimpan topik *home*/*kitchen* yang berarti bahwa *middleware* dengan mode *cluster* mampu membagi data yang diterima ke dalam beberapa redis yang terdapat pada *cluster*.



Gambar 6.4 Hasil Pengujian skenario PF_004

6.1.5 *Middleware* mampu mengirimkan data dengan protokol CoAP ke *subscriber* melalui protokol *Websocket*

Tabel 6.5 *Middleware* mampu mengirimkan data dengan protokol CoAP ke *subscriber* melalui protokol *Websocket*

Kode Uji	PF_005
Tujuan Pengujian	*Middleware* yang menerima data dari *publisher* melalui protokol CoAP mampu mengirimkan data tersebut ke *subscriber* melalui protokol *Websocket*
Prosedur pengujian	1. *Node* sensor B men*subscribe* topik *home*/*kitchen* ke *middleware* 2. *Node* sensor A mengirimkan data dengan topik *home*/*kitchen* ke *middleware* melalui protokol CoAP 3. Menjalankan *middleware* dan menjalankan pm2 *logs*
Hasil yang diharapkan	*Subscriber* menerima data dengan topik *home*/*kitchen* dari *middleware*
Hasil pengujian	*Subscriber* menerima data dengan topik *home*/*kitchen* dari *middleware*
Keterangan	Valid

Pada Gambar 6.5 menunjukkan sebuah tampilan *payload* data yang diterima oleh *subscriber*. *Payload* data yang berisi data suhu yang bernilai 30^{o}C dan kelembapan yang bernilai 20% tersebut dikirimkan oleh *publisher* dengan IP 192.168.42.41 melalui protokol CoAP dengan topik *home/kitchen* menggunakan module dht11. Berdasarkan hasil pengujian PF_005 disimpulkan bahwa integrasi *middleware* yang mengimplementasikan *cluster message Broker* dengan *subscriber* berhasil dilakukan.

```
pi@pi2A:~/jessy/sentMasterB/subscribe $ sudo node concurrentWeb.js
oke  Load packet:0{"protocol":"coap","timestamp":"2018-07-15T09:39:02.922Z","top
ic":"home/kitchen","sensor":{"tipe":"esp8266","index":"cocoap","ip":"192.168.42.
41","module":"dht11"},"humidity":{"value":"20","unit":"%"},"temperature":{"value
":"30","unit":"celcius"}}
```

Gambar 6.5 Hasil Pengujian skenario PF_005

6.1.6 *Middleware* mampu mengirimkan data dengan protokol MQTT ke *subscriber* melalui protokol *Websocket*

Tabel 6.6 *Middleware* mampu mengirimkan data dengan protokol MQTT ke *subscriber* melalui protokol *Websocket*

Kode Uji	PF_006
Tujuan Pengujian	*Middleware* yang menerima data dari *publisher* melalui protokol MQTT mampu mengirimkan data tersebut ke *subscriber* melalui protokol *Websocket*
Prosedur pengujian	1. *Node* sensor B men*subscribe* topik *home*/*garage* ke *middleware* 2. *Node* sensor A mengirimkan data dengan topik *home*/*garage* ke *middleware* melalui protokol MQTT 3. Menjalankan *middleware* dan menjalankan pm2 *logs*
Hasil yang diharapkan	*Subscriber* menerima data dengan topik *home*/*garage* dari *middleware*
Hasil pengujian	*Subscriber* menerima data dengan topik *home*/*garage* dari *middleware*
Keterangan	Valid

Pada Gambar 6.6 menunjukkan sebuah tampilan *payload* data yang diterima oleh *subscriber*. *Payload* data yang berisi data suhu yang bernilai 30^{o}C dan kelembapan yang bernilai 20% tersebut dikirimkan oleh *publisher* dengan IP 192.168.42.41 melalui protokol MQTT dengan topik *home/garage*. Berdasarkan hasil pengujian PF_006 disimpulkan bahwa integrasi *middleware* yang mengimplementasikan *cluster message Broker* dengan *subscriber* berhasil dilakukan.

```
pi@pi2A:~/jessy/sentMasterB/subscribe $ sudo node concurrentWeb.js
oke  Load packet:0{"protocol":"mqtt","timestamp":"Sun Jul 15 2018 16:39:03 GMT+0
700 (WIB)","topic":"home/garage","sensor":{"tipe":"esp8266","index":"mqtt","ip":
"192.168.42.41","module":"dht11"},"humidity":{"value":"20","unit":"%"},"temperat
ure":{"value":"30","unit":"celcius"}}
```

Gambar 6.6 Hasil Pengujian skenario PF_006

6.1.7 *Redis* dapat saling terhubung satu sama lain

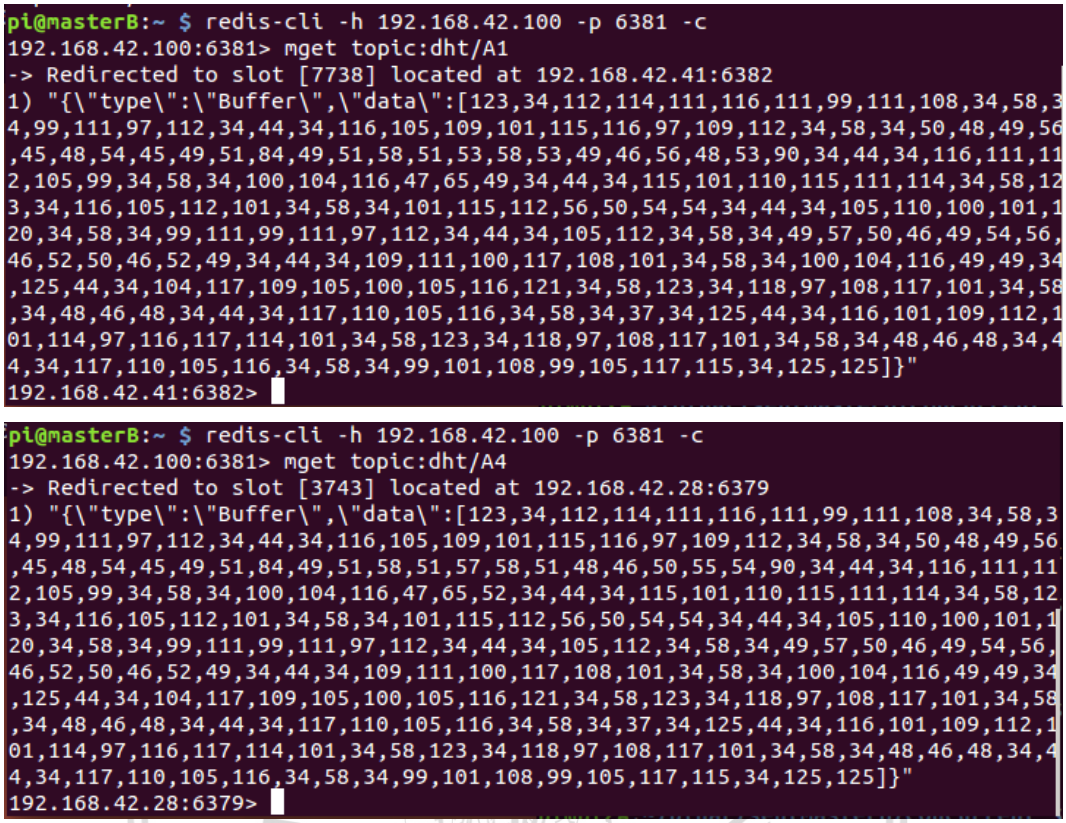
Tabel 6.7 *Redis* dapat saling terhubung satu sama lain

Kode Uji	PF_007
Tujuan Pengujian	Mengetahui apakah semua *redis* sudah dapat saling berkomunikasi ditandai dengan bisa saling mengakses data pada topik/*key* di *redis* lain
Prosedur pengujian	1. Mengirimkan data suhu dan kelembapan ke *middleware* dengan topik yang berbeda-beda 2. Melakukan pengecekan status pada salah satu *redis* 3. Melakukan akses data dengan topik/*key* yang disimpan pada *redis* lain
Hasil yang diharapkan	*Redis* dapat saling berkomunikasi dan mengakses data
Hasil pengujian	*Redis* dapat saing berkomunikasi dan mengakses data
Keterangan	Valid

Pada Gambar 6.7 menunjukkan topik dht/A1 ada di *redis* dengan *port* 6382 dan IP 192.168.42.41 sedangkan topik dht/A4 ada di *redis* dengan *port* 6379 dan IP 192.168.42.79. Kemudian *redis* dengan *port* 6381 dengan IP 192.168.42.100 mencoba mengakses topik-topik tersebut dengan melakukan perintah MGET [nama_topik]. Hasilnya dapat dilihat pada gambar bahwa *redis* dengan *port* 6380 dapat mengakses data di *redis* dengan *port* 6382 dan 6379. Hal tersebut ditunjukkan dengan keterangan "Redirected" yang berarti redis *port* 6380 dialihkan ke *port* 6382 dengan lokasi data untuk topik dht/A1 di *slot* 7738, dan redis *port* 6380 juga dialihkan ke *port* 6379 dengan lokasi data untuk topik dht/A2 berada di *slot* 3743. Data yang ditampilkan berupa Buffer.

Berdasarkan hasil pengujian PF_007 maka disimpulkan *cluster message Broker* yang dibangun pada *middleware* mampu saling terhubung satu sama lain sehingga antar *node* mampu saling mengakses data.

```
pi@pi2A:~/binar/sentMasterB/Concurrent $ redis-cli -h 192.168.42.41 -p 6382 --sc
an
topic:dht/A1
pi@raspberrypi43:~ $ redis-cli -h 192.168.42.28 -p 6379 --scan
topic:dht/A4
pi@raspberrypi43:~ $
```



Gambar 6.7 Hasil Pengujian skenario PF_007

6.2 Pengujian Non-Fungsional

Pengujian non-fungsional dilakukan untuk mengetahui batasan kemampuan sistem. Pengujian non-fungsional pada penelitian ini menguji kebutuhan non-fungsional skalabilitas.

6.2.1 Pengujian Skalabilitas

Pengujian skalabilitas bertujuan untuk mengetahui kemampuan *middleware* dalam menangani berbagai proses yang diterima ketika terjadi perubahan jumlah *publisher* dan *subscriber* yang menjadi lebih besar dari sebelumnya. Pengujian skalabilitas dilakukan dengan melakukan skenario perbesaran jumlah klien *publisher* atau *subscriber*. Pada simulasi yang dilakukan akan dihitung nilai *concurrent publish*, *concurrent subscribe*, *time publish*, dan *time subscribe*. Pada penelitian ini dalam menghitung nilai *concurrent* dan *time* digunakan *tool tcpdump* untuk melakukan *filter* paket dan *tool wireshark* untuk menganalisa hasil *capture* paket.

Nilai *time* merupakan waktu yang dibutuhkan untuk menyelesaikan *publish* dan *subscribe* dalam sejumlah *publisher* dan *subscriber* yang telah ditentukan. Untuk protokol CoAP digunakan QoS CON (*Confirmable*) karena lebih *reliability* di mana penerima akan mengirimkan *ack* apabila pesan telah sampai. Nilai *time* untuk protokol CoAP di hitung dari nilai waktu *ack* terakhir dikurangi nilai waktu CON pertama. Untuk protokol MQTT digunakan QoS level 1 karena

reliability di mana pesan akan dikirimkan setidaknya satu kali (dapat terjadi duplikasi) hingga penerima mengirimkan ACK. Nilai *time* untuk protokol MQTT di hitung dari nilai waktu *publish ack* terakhir di kurangi nilai waktu *publish message* pertama. Untuk protokol *Websocket* nilai *concurrent* dihitung dari data mulai dikirim hingga selesai melalui protokol *Websocket*.

Nilai *concurrent* merupakan banyaknya jumlah pesan yang mampu ditangani oleh *middleware* dalam satu detik, baik yang menggunakan protokol CoAP, MQTT atau *Websocket*. Nilai *concurrent* diperoleh dari banyaknya jumlah klien *publisher* atau *subscriber* dibagi waktu yang dibutuhkan untuk *publish* atau *subscribe*. Jadi sebelum mendapatkan nilai *concurrent* maka harus mencari nilai *time* terlebih dahulu.

Pengujian skalabilitas dilakukan terhadap penelitian sebelumnya yang tanpa Cluster dan penelitian saat ini yang sudah diterapkan Cluster untuk mengetahui perbandingan skalabilitas dari keduanya. Pengujian pada penelitian sebelumnya dilakukan pengiriman data sensor dari IP 192.168.42.41 ke *middleware* dengan IP 192.168.42.72. Untuk pengujian penelitian saat ini dilakukan pengiriman dari sensor IP 192.168.42.41 ke *middleware* dengan IP 192.168.42.100. Pengujian skalabilitas dibagi menjadi dua skenario. Pertama untuk *publisher* ke *middleware* dan kedua untuk *middleware* ke *subscriber*.

Tabel 6.8 Pengujian *time publish*

Kode Uji	PNF_001
Tujuan Pengujian	Mengetahui berapa lama waktu yang dibutuhkan untuk melakukan *publish* melalui protokol CoAP dan MQTT
Prosedur pengujian	Untuk pengujian dengan parameter *time publish* dengan protokol CoAP dan MQTT: a. Menjalankan pm2 logs pada *middleware* untuk memonitoring pesan yang masuk ke *middleware* b. Menjalankan filter paket pada *middleware* menggunakan tcpdump c. Menjalankan program pengiriman data melalui CoAP dan MQTT secara bersamaan yang dengan variasi jumlah pengirim 100, 500, 1000, 1500 d. Melakukan analisa paket hasil tcpdump menggunakan wireshark untuk menentukan nilai *concurrent publish*

Tabel 6.8 Pengujian *time publish* (Lanjutan)

Kode Uji	PNF_001
	berdasarkan QoS CoAP dan MQTT yang digunakan.
Hasil yang diharapkan	Mengetahui kemampuan *middleware* dalam menangani sejumlah klien saat *publish*
Hasil pengujian	Dijelaskan pada poin 6.2.2

Tabel 6.9 Pengujian *concurrent publish*

Kode Uji	PNF_002
Tujuan Pengujian	Mengetahui berapa banyak jumlah pesan yang mampu ditangani oleh *middleware* dalam satu detik ketika proses *publish*.
Prosedur pengujian	Untuk pengujian dengan parameter *concurrent publish* dengan protokol *CoAP dan MQTT:* a. Menjalankan pm2 logs pada *middleware* untuk memonitoring pesan yang masuk ke *middleware* b. Menjalankan filter paket pada *middleware* menggunakan tcpdump c. Menjalankan program pengiriman data melalui CoAP dan MQTT secara bersamaan yang dengan variasi jumlah pengirim 100, 500, 1000, 1500 d. Melakukan analisa paket hasil tcpdump menggunakan wireshark untuk menentukan nilai *concurrent publish* berdasarkan QoS CoAP dan MQTT yang digunakan
Hasil yang diharapkan	Mengetahui berapa pesan per detik yang mampu di tangani oleh *middleware s*aat *publish*
Hasil pengujian	Dijelaskan pada poin 6.2.2

Tabel 6.10 Pengujian *time subscribe*

Kode Uji	PNF_003
Tujuan Pengujian	Mengetahui berapa lama waktu yang dibutuhkan untuk melakukan *subscribe* melalui protokol Websocket
Prosedur pengujian	Untuk pengujian dengan parameter *time publish* dengan protokol CoAP dan MQTT: a. Menjalankan program pengiriman data dari sensor b. Menjalankan pm2 *logs* pada *middleware* c. Menjalankan *filter* paket pada *middleware* menggunakan tcpdump d. Menjalankan program *subscribe* yang berisi *package* async dengan variasi jumlah 100, 500, 1000, dan 1500. e. Melakukan analisa paket hasil tcpdump menggunakan *wireshark* untuk menentukan nilai *time subscribe* dan *concurrent subscribe*
Hasil yang diharapkan	Mengetahui kemampuan *middleware* dalam menangani sejumlah klien saat *subscribe*
Hasil pengujian	Dijelaskan pada poin 6.2.2

Tabel 6.11 Pengujian *concurrent subscribe*

Kode Uji	PNF_004
Tujuan Pengujian	Mengetahui berapa banyak jumlah *subscribe* yang mampu ditangani oleh *middleware* dalam satu detik
Prosedur pengujian	Untuk pengujian dengan parameter *concurrent subscribe* dengan protokol Websocket*:* a. Menjalankan pm2 logs pada *middleware* untuk memonitoring pesan yang masuk ke *middleware*

Tabel 6.11 Pengujian *concurrent subscribe* (Lanjutan)

Kode Uji	PNF_004
	b. Menjalankan program sensor untuk melakukan pengiriman data melalui CoAP dan MQTT c. Menjalankan tcpdump untuk melakukan filter paket di *middleware* d. Menjalankan program *subscriber* dengan variasi jumlah 100, 500, 1000, dan 1500 e. Melakukan analisa paket hasil tcpdump menggunakan wireshark untuk menentukan nilai *concurrent subscribe*
Hasil yang diharapkan	Mengetahui berapa pesan per detik yang mampu di tangani oleh *middleware* saat *publish*
Hasil pengujian	Dijelaskan pada poin 6.2.2

6.2.2 Hasil Pengujian Skalabilitas

6.2.2.2 Penelitian Sebelumnya dengan Non-Cluster

Pengujian dilakukan berdasarkan scenario PNF_001 dan PNF_002. Pada tahap ini, terdapat empat jenis parameter yaitu *time publish*, *time subscribe*, *concurrent publish dan concurrent subscribe.*

a. Pengujian *time publish*

Pengujian dilakukan berdasarkan skenario PNF_001. Pada tahap ini akan dilakukan penghitungan *time publish* saat proses *publish* ke *middleware*. Hasil pengujian yang dilakukan digambarkan dalam bentuk tabel dibawah ini:

Tabel 6.12 Hasil Pengujian *time publish* CoAP

Jumlah *Publisher*	**Percobaan**				
	1 *Time* (s)	**2 *Time* (s)**	**3 *Time* (s)**	**4 *Time* (s)**	**5 *Time* (s)**
100	**2.84**	**2.85**	**2.81**	**2.71**	**2.82**
500	**19.53**	**13.44**	**19.65**	**13.48**	**19.17**
1000	**40.63**	**32.03**	**20.17**	**33.13**	**38.73**
1500	**42.60**	**42.98**	**43.19**	**43.36**	**43.47**

Pada tabel 6.12 diperoleh rata-rata untuk jumlah *publisher* dengan variasi 100 yaitu 2.82, untuk variasi 500 yaitu 19.17, untuk variasi 1000 yaitu 33.13 dan untuk variasi 1500 yaitu 43.19.

Tabel 6.13 Hasil Pengujian *time publish* MQTT

Jumlah *Publisher*	Percobaan				
	1 *Time* (s)	2 *Time* (s)	3 *Time* (s)	4 *Time* (s)	5 *Time* (s)
100	2.03	2.75	2.43	2.05	2.69
500	11.46	11.69	11.80	12.01	9.24
1000	17.66	19.66	16.22	19.58	19.43
1500	24.69	20.60	25.58	23.56	24.36

Pada tabel 6.13 diperoleh rata-rata untuk jumlah *publisher* dengan variasi 100 yaitu 2.43, untuk variasi 500 yaitu 11.69, untuk variasi 1000 yaitu 19.43 dan untuk variasi 1500 yaitu 24.36.

b. Pengujian *concurrent publish*

Pengujian dilakukan berdasarkan skenario PNF_002. Pada pengujian ini dilakukan penghitungan nilai *concurrent subscribe*. Hasil dari pengujian digambarkan dalam bentuk tabel tiap jumlah variasi *publisher* seperti dibawah ini:

Tabel 6.14 Hasil Pengujian *concurrent publish* CoAP dan MQTT

Jumlah *Publisher*	Percobaan									
	1 *concurrent* (*message*/s)		2 *concurrent* (*message*/s)		3 *concurrent* (*message*/s)		4 *concurrent* (*message*/s)		5 *concurrent* (*message*/s)	
	CoAP	MQTT	CoAP	MQTT	CoAP	MQTT	CoAP	MQTT	CoAp	MQTT
100	34	37	36	40	35	41	36	42	40	48
500	25	43	34	41	13	21	31	45	38	46
1000	59	40	59	37	64	44	49	36	71	42
1500	35	34	34	62	47	35	38	38	58	27

Pada tabel 6.14 *middleware* yang menangani pesan dengan protokol CoAP mampu mencapai tingkat skalabilitas hingga 71 pesan/detik. Sedangkan untuk protokol MQTT mencapai 62 pesan/detik.

c. Pengujian *time subscribe*

Pengujian dilakukan berdasarkan scenario PNF_003. Pada pengujian ini akan dilakukan penghitungan terhadap nilai *time subscribe* saat proses *subscriber* yang terjadi. Hasil dari penelitian digambarkan dalam bentuk tabel seperti dibawah ini:

Tabel 6.15 Pengujian *time subscribe*

Jumlah *Subscriber*	100	500	1000	1500
Time	0,54	1,67	69,41	67,20

Tabel 6.15 juga menunjukkan bahwa nilai *time* akan bertambah meningkat seiring dengan jumlah klien *subscriber* yang meningkat.

d. Pengujian *concurrent subscribe*

Pengujian dilakukan berdasarkan scenario PNF_004. Pada pengujian ini akan dilakukan penghitungan terhadap nilai *concurrent subscribe* saat proses *subscriber* yang terjadi. Hasil dari penelitian digambarkan dalam bentuk tabel seperti dibawah ini:

Tabel 6.16 Pengujian *concurrent subscribe*

Jumlah *Subscriber*	100	500	1000	1500
Concurrent (pesan/detik)	18	22	27	35

Pada tabel 6.16 kemampuan *middleware* dalam menangani *subscribe* mencapai tingkat skalabilitas hingga 35 pesan per detik pada jumlah klien 1500.

6.2.2.3 Penelitian saat ini dengan Cluster Message Broker

Pengujian dilakukan berdasarkan scenario PNF_001, PNF_002, PNF_003, dan PNF_004. Pada tahap ini, terdapat empat jenis parameter yaitu *time publish*, *time subscribe, concurrent publish dan concurrent subscribe.*

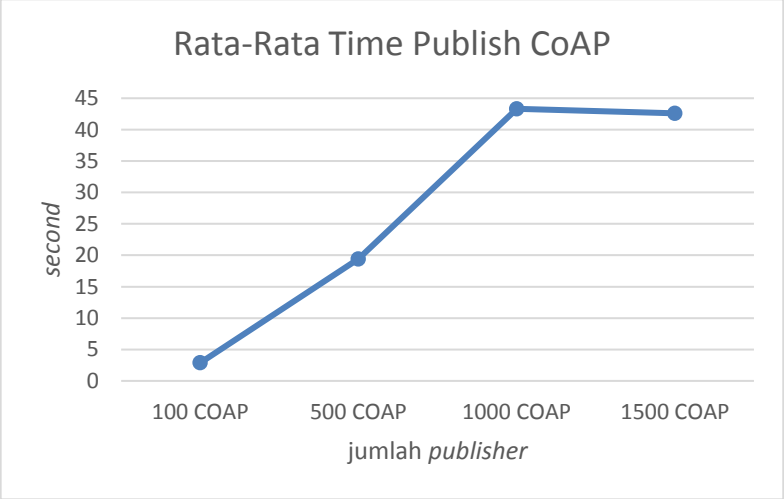
a. Pengujian pertama yakni *time publish*

Pengujian dilakukan berdasarkan skenario PNF_001. Pada tahap ini akan dilakukan penghitungan *time publish* saat proses *publish* ke *middleware*. Hasil pengujian yang dilakukan digambarkan dalam bentuk tabel dibawah ini:

Tabel 6.17 Hasil Pengujian *time publish* CoAP

Jumlah *Publisher*	Percobaan				
	1 *Time* (s)	2 *Time* (s)	3 *Time* (s)	4 *Time* (s)	5 *Time* (s)
100	2.86	2.85	2.89	2.92	2.88
500	14.69	39.93	19.40	20.80	17.60
1000	42.32	42.19	42.81	44.07	43.81
1500	43.80	42.61	39.88	43.80	42.42

Pada tabel 6.17 diperoleh rata-rata untuk jumlah *publisher* dengan variasi 100 yaitu 2.88, untuk variasi 500 yaitu 19.4, untuk variasi 1000 yaitu 43.32 dan untuk variasi 1500 yaitu 42.61. Nilai rata-rata tersebut digambarkan dalam bentuk grafik line seperti dibawah ini:



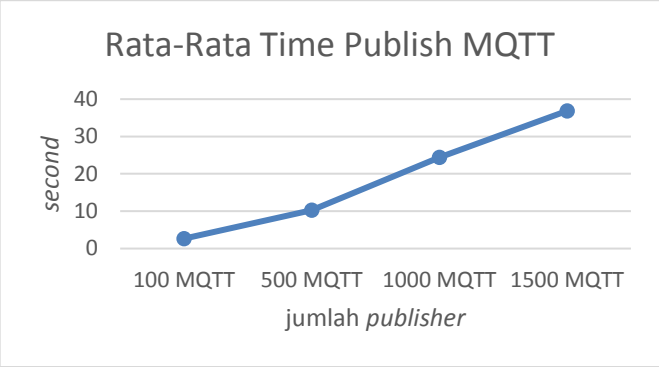
Gambar 6.8 Rata-rata *time publish* Cluster

Selanjutnya berikut ini merupakan hasil pengujian *time publish* dengan MQTT.

Tabel 6.18 Hasil pengujian *time publish* MQTT

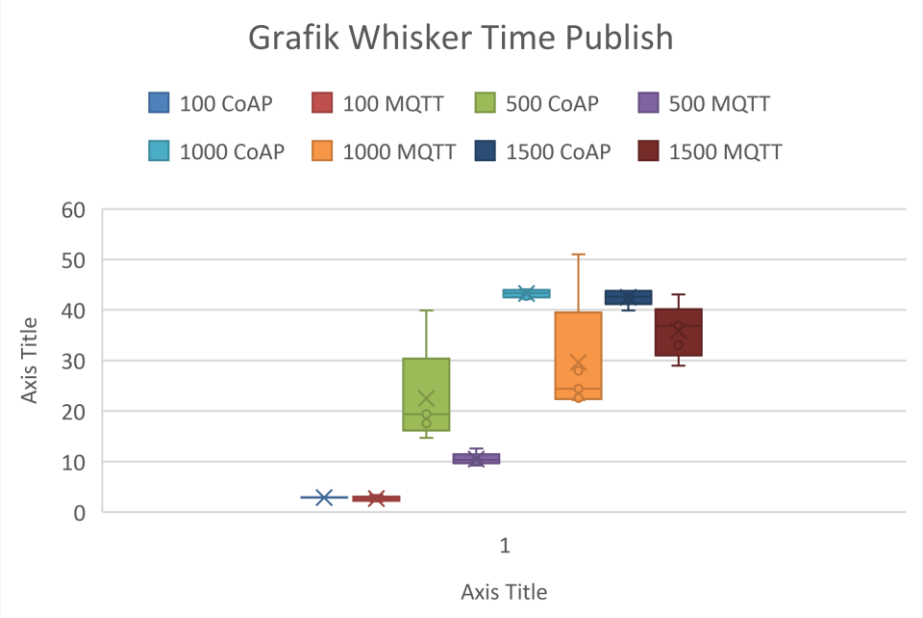
Jumlah *Publisher*	Percobaan				
	1 *Time* (s)	2 *Time* (s)	3 *Time* (s)	4 *Time* (s)	5 *Time* (s)
100	2.12	2.34	2.66	3.39	2.70
500	10.31	10.33	9.24	12.61	10.05
1000	27.96	24.43	22.17	51.04	22.57
1500	37.26	33.10	36.85	28.97	43.09

Pada tabel 6.18 diperoleh rata-rata untuk jumlah *publisher* dengan variasi 100 yaitu 2.66, untuk variasi 500 yaitu 10.31, untuk variasi 1000 yaitu 24.43 dan untuk variasi 1500 yaitu 36.85. Nilai rata-rata tersebut digambarkan dalam bentuk grafik seperti dibawah ini:



Gambar 6.9 Rata-rata *time publish* MQTT

Berdasarkan Tabel 6.17 dan Tabel 6.18 tersebut dibuat grafik *whisker* yang menggambarkan sebaran data pada parameter *time publish* dengan menggunakan protokol CoAP dan MQTT dengan berbagai variasi jumlah pengirim. Berikut ini grafik *whisker time publish* CoAP dan MQTT:



Gambar 6.10 Grafik *whisker time publish* CoAP dan MQTT

b. Pengujian *concurrent publish*

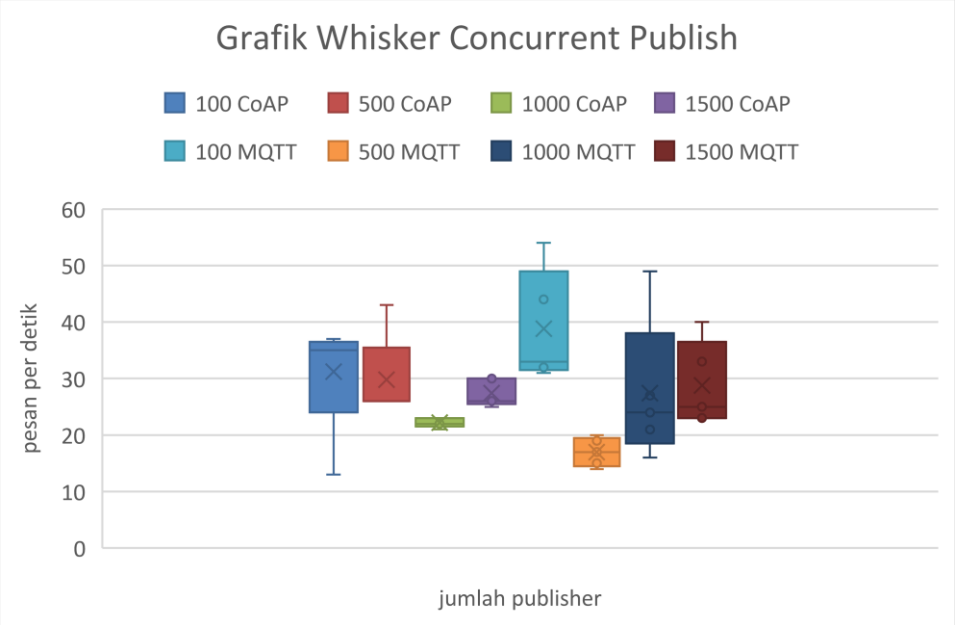
Tabel 6.19 Hasil pengujian *concurrent publish*

Jumlah *Publisher*	Percobaan				
	1	2	3	4	5
	concurrent (pesan/detik)	*concurrent* (pesan/detik)	*concurrent* (pesan/detik)	*concurrent* (pesan/detik)	*concurrent* (pesan/detik)

Tabel 6.19 Hasil pengujian *concurrent publish* (Lanjutan)

Jumlah *Publisher*	Percobaan									
	1		2		3		4		5	
	concurrent (pesan/detik)		*concurrent* (pesan/detik)		*concurrent* (pesan/detik)		*concurrent* (pesan/detik)		*concurrent* (pesan/detik)	
	CoAP	MQTT	CoAP	MQTT	CoAP	MQTT	CoAP	MQTT	CoAp	MQTT
100	35	44	37	54	36	33	13	31	35	32
500	26	20	28	15	26	19	43	17	26	14
1000	23	24	21	49	22	16	23	27	22	21
1500	26	33	30	23	26	40	26	23	25	25

Pada tabel 6.19 menunjukkan protocol CoAP mencapai tingkat skalabilitas hingga 43 pesan per detik, sedangkan MQTT mencapai tingkat skalabilitas hingga 54 pesan per detik. Gambar 6.11 merupakan grafik whisker yang menggambarkan sebaran data dari hasil perngujian:



Gambar 6.11 Grafik whisker *concurrent publish* CoAP dan MQTT

c. Pengujian *time subscribe*

Tabel 6.20 Hasil pengujian *time subscribe*

Jumlah *Subscriber*	100	500	1000	1500
Time	2.78	9.26	53.62	67.12

Tabel 6.20 menunjukkan waktu yang dibutuhkan untuk *subscribe* meningkat seiring bertambahnya jumlah klien *subscriber*.

d. Pengujian *concurrent subscribe*

Tabel 6.21 Hasil pengujian *concurrent subscribe*

Jumlah *Subscriber*	100	500	1000	1500
Concurrent (pesan/detik)	24	17	28	31

Tabel 6.21 menunjukkan websocket mampu mencapai tingkat skalabilitas hingga 31 pesan per detik dengan jumlah *subscriber* 1500 klien.

6.2.2.4 Perbandingan Hasil Pengujian penelitian sebelumnya dengan penelitian saat ini

1. Time Publish CoAP dan MQTT

Pada penelitian sebelumnya diperoleh rata-rata untuk jumlah *publisher* dengan variasi 100 yaitu 2.82, untuk variasi 500 yaitu 19.17, untuk variasi 1000 yaitu 33.13 dan untuk variasi 1500 yaitu 43.19. Sedangkan untuk MQTTdiperoleh rata-rata untuk jumlah *publisher* dengan variasi 100 yaitu 2.43, untuk variasi 500 yaitu 11.69, untuk variasi 1000 yaitu 19.43 dan untuk variasi 1500 yaitu 24.36. Waktu yang diperlukan MQTT lebih cepat dalam pengiriman pesan dibandingkan CoAP pada setiap jumlah variasinya.

Pada penelitian saat ini diperoleh rata-rata untuk jumlah *publisher* dengan variasi 100 yaitu 2.88, untuk variasi 500 yaitu 19.4, untuk variasi 1000 yaitu 43.32 dan untuk variasi 1500 yaitu 42.61. Sedangkan untuk MQTT diperoleh rata-rata untuk jumlah *publisher* dengan variasi 100 yaitu 2.66, untuk variasi 500 yaitu 10.31, untuk variasi 1000 yaitu 24.43 dan untuk variasi 1500 yaitu 36.85. Waktu yang diperlukan MQTT lebih cepat dalam pengiriman pesan dibandingkan CoAP pada setiap jumlah variasinya.

2. *Concurrent Publish* CoAP dan MQTT

Pada penelitian sebelumnya CoAP mampu menangani proses sampai 1500 *publisher* dengan maksimum 71 pesan/detik, sedangkan MQTT dengan maksimum 62 pesan/detik. Pada penelitian saat ini CoAP mampu menangani proses sampai 1500 *publisher* dengan maksimum 43 pesan/detik, sedangkan MQTT dengan maksimum 54 pesan/detik.

3. *Time Subscribe Websocket*

Pada penelitian sebelumnya waktu yang dibutuhkan untuk *subscribe* dengan beberapa variasi klien meningkat seiring dengan meningkatnya jumlah klien. Begitu juga pada penelitian ini, data pada tabel menunjukkan waktu yang dibutuhkan meningkat saat jumlah klien meningkat, hal ini dikarenakan waktu yang diperlukan untuk memproses lebih lama.

4. *Concurrent Subscribe Websocket*

Pada penelitian sebelumnya *Websocket* mampu mencapai tingkat skalabilitas hingga 35 pesan/detik ketika jumlah *subscriber* mencapai 1500 *subscriber*. Pada penelitian ini mengalami pencapaian hingga 31 pesan per detik saat klien *subscriber* 1500.

BAB 7 PENUTUP

7.1 Kesimpulan

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

1. Penerapan metode *cluster message Broker* menggunakan Redis dapat diimplementasikan pada *middleware* IoT. Dalam mengimplementasikan *cluster* tersebut digunakan beberapa *redis*. *Redis* yang digunakan pada penelitian ini yaitu 6 *redis*. *Redis* tersebut dikonfigurasi agar berjalan pada mode *cluster*. *Cluster* dibuat dengan topologi *mesh* agar setiap *node* bisa saling berkomunikasi dan mengakses data. Setelah *cluster* sudah terbentuk maka dilakukan integrasi dengan *middleware* pada penelitian sebelumnya dengan cara menambahkan ioredis dan mendefinisikan *redis* yang ada pada *cluster* di dalam *middleware*.
2. Pada penelitian ini akan dijelaskan kinerja *middleware* yang belum dan sudah menerapkan *cluster message Broker* dalam menjawab permasalahan skalabilitas *middleware* pada poin-poin dibawah ini:
    a. Hasil pengujian *time publish* pada kedua *Middleware*, menunjukkan prokol MQTT lebih cepat dalam pengiriman pesan daripada protokol CoAP yang ditunjukkan dari nilai rata-rata *time publish*.
    b. Hasil pengujian *concurrent publish* pada kedua *middleware*, menunjukkan tingkat skalabilitas *middleware* dengan *cluster* lebih rendah dibandingkan *middleware* tanpa *cluster*. Hal itu ditunjukkan dengan nilai CoAP pada *middleware cluster* mencapai 43 pesan per detik dan MQTT 54 pesan per detik, sedangkan *middleware* tanpa *cluster* protokol CoAP mencapai 71 pesan per detik dan MQTT mecapai 62 pesan per detik. Penurunan tingkat skalabilitas dikarenakan terjadinya proses replikasi pada *node master* redis ke *node slave* redis. Selain itu juga space yang dialokasikan untuk redis akan dibagi antara master dan slave yang ada.
    c. *Time* subcribe dengan *Websocket* pada *middleware* dengan *cluster* dan *middleware* tanpa *cluster* mengalami peningkatan dengan waktu yang terus bertambah seiring dengan bertambahnya jumlah klien *subscriber*.
    d. *Concurrent subscribe* pada *middleware* tanpa cluster mencapai tingkat skalabilitas hingga 35 pesan/detik. Sedangkan *middleware* dengan cluster mencapai tingkat skalabilitas hingga 31 pesan/detik.

7.2 Saran

Berdasarkan kesimpulan penelitian, maka peneliti memberikan beberapa saran untuk penelitian berikutnya sebagai berikut:

1. Digunakannya arsitektur *cluster* di mana *middleware* yang ada di semua *node* sehingga pengiriman data dari *publisher* tidak hanya bertumpu di satu titik *middleware*.
2. Digunakannya parameter lain seperti *n-size* dalam menguji skalabilitas untuk melihat kemampuan *middleware* dalam menampung sejumlah data.
3. Penempatan *node master* dan *slave* pada *Raspberry* yang berbeda, sehingga pembagian *memory* pada *Raspberry* lebih optimal.

DAFTAR PUSTAKA

Scalagent. 2014. JoramMQ, a distributed MQTT *Broker for* the *Internet of things*. Dipetik 4 September 2017, dari <http://www.scalagent.com/IMG/pdf/JoramMQ_MQTT_white_paper-v1-2.pdf>

Anwari, H. 2017. "*Pengembangan IoT Middleware Berbasis Event-Based dengan Protokol Komunikasi CoAP, MQTT, dan Websocket*". JPTIIK.

Redis, 2009. Redis *Cluster* Tutorial & Redis *Cluster* Specification. Dipetik 27 Agustus 2017, dari <https://Redis.io> [Diakses pada 27 agustus 2017].

IETF, I. E. (2016). The Constrained *Application* Protokol (CoAP) RFC7252. Internet Engineering Task Force.

HiveMQ. (2015). MQTT Essential Part 2: *Publish* & *Subscribe*. Dipetik 15 Juni 2018, dari HiveMQ: http://www.hivemq.com/b*log*/mqtt-essential-part2-*publish-subscribe*.

Skvorc, D., Horvat, M., & Srbljic, S. (2014). Performance Evaluation *of Websocket* Protokol *for* Implementation *of Full-duplex Web* Streams. Zagreb: University *of* Zagreb/School *of* Electrical Engineering *and* Computing.

Chapuis, B. and Garbinato, B. (2017) 'Scaling and Load Testing Location-Based *Publish* and *Subscribe*', *Proceedings - International Conference on Distributed Computing Systems*, pp. 2543–2546. doi: 10.1109/ICDCS.2017.234.

Chen, S. *et al.* (2016) 'Towards scalable and reliable in-memory storage system: A case study with Redis', *Proceedings - 15th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, 10th IEEE International Conference on Big Data Science and Engineering and 14th IEEE International Symposium on Parallel and Distributed Processing with Applications, IEEE TrustCom/BigDataSE/ISPA 2016*, pp. 1660–1667. doi: 10.1109/TrustCom.2016.0255.

da Cruz, M. A. A. *et al.* (2018) 'A Reference Model for Internet of Things *Middleware*', *IEEE Internet of Things Journal*, 4662(c), pp. 1–1. doi: 10.1109/JIOT.2018.2796561.

Gupta, A., Christie, R. and Manjula, P. R. (2017) 'Scalability in Internet of Things : Features , Techniques and Research Challenges', *International Journal of Computational Intelligence Research*, 13(7), pp. 1617–1627.

Jutadhamakorn, P. *et al.* (2017) 'A Scalable and Low-Cost MQTT Broker Clustering System'.

Li, S., Jiang, H. and Shi, M. (2017) 'Redis-based Web Server Cluster Session Maintaining Technology', *2017 13th International Conference on Natural*

Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD). IEEE, pp. 3065–3069.

Pramukantoro, E. S., Yahya, W. and Bakhtiar, F. A. (2017) 'Perform evaluation of IoT *middleware* for syntactical Interoperability', *International Conference on Advanced Computer Science and Information Systems (ICACSIS)*, 00. doi: 10.1109/ICACSIS.2017.8355008.

Razzaque, M. A., Milojevic-jevric, M. and Palade, A. (2016) '*Middleware* for Internet of Things : a Survey', *Ieee Internet of Things Journal*, 3(1), pp. 70–95. doi: 10.1109/JIOT.2015.2498900.

Rozi, M. F., Pramukantoro, E. S. and Amron, K. (2017) 'Analisis Performansi dan Skalabilitas pada Event-Based IoT *Middleware*', 1(7), pp. 593–601.

Son, H. *et al.* (2015) 'A Distributed *Middleware* for a Smart Home with Autonomous Appliances', *Proceedings - International Computer Software and Applications Conference*, 2, pp. 23–32. doi: 10.1109/COMPSAC.2015.101.

Thomas, G., Alexander, G. and Pm, S. (2017) 'Design of High PerformanceClusterbased Map for Vehicle Tracking of public transport vehicles in Smart City', *Ieee*, pp. 2–6.