IMPLEMENTASI MQTT BROKER DENGAN KEMAMPUAN *AUTO SCALING* PADA *INTERNET OF THINGS*

SKRIPSI

Untuk memenuhi sebagian persyaratan
memperoleh gelar Sarjana Komputer

Disusun oleh:
Kevin Jonathan Harnanta
NIM: 165150201111165

PROGRAM STUDI TEKNIK INFORMATIKA
JURUSAN TEKNIK INFORMATIKA
FAKULTAS ILMU KOMPUTER
UNIVERSITAS BRAWIJAYA
MALANG
2020

Repository Universitas Brawijaya
REPOSITORY.UB.AC.ID
UNIVERSITAS BRAWIJAYA

Repository Universitas Brawijaya

PENGESAHAN

IMPLEMENTASI MQTT BROKER DENGAN KEMAMPUAN *AUTO SCALING* PADA *INTERNET OF THINGS*

SKRIPSI

Diajukan untuk memenuhi sebagian persyaratan
memperoleh gelar Sarjana Komputer

Disusun Oleh :
Kevin Jonathan Harnanta
NIM: 165150201111165

Skripsi ini telah diuji dan dinyatakan lulus pada
26 Juni 2020

Telah diperiksa dan disetujui oleh:

Dosen Pembimbing I	Dosen Pembimbing II
Adhitya Bhawiyuga, S.Kom., M.Sc.	Achmad Basuki, S.T., M.MG., Ph.D.
NIP: 19890720 201803 1 002	NIP: 19741118 200312 1 002

Mengetahui
Ketua Muda Jurusan Teknik Informatika

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

REPOSITORY.UB.AC.ID
UNIVERSITAS BRAWIJAYA

Repository Universitas Brawijaya

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, 12 Juni 2020

Kevin Jonatnan Harnanta

NIM. 165150201111165

PRAKATA

Puji Syukur penulis ucapkan, karena berkat rahmat dari Tuhan Yang Mana Esa, penulis dapat menyelesaikan laporan skripsi berjudul "Implementasi MQTT Broker Dengan Kemampuan *Auto Scaling* Pada *Internet of Things*".

Dalam penyelesaian laporan penelitian ini, penulis mendapat banyak dukungan, bantuan dan bimbingan dari berbagai pihak. Maka dari itu, penulis menyampaikan rasa terima kasih yang sebesar-besarnya kepada:

1. Bapak dan ibu peneliti yang telah mendukung selama proses pengerjaan skripsi ini berlangsung.
2. Bapak Adhitya Bhawiyuga, S.Kom., M.Sc. dan Achmad Basuki, S.T, M.Mg, Ph.D selaku dosen pembimbing skripsi penulis yang selama ini telah memberikan fasilitas, banyak dukungan serta arahan kepada penulis untuk dapat menyelesaikan penelitian ini.
3. Bapak Tri Astoto Kurniawan, S.T., M.T., Ph.D. selaku selaku ketua Jurusan Teknik informatika.
4. Bapak Agus Wahyu Widodo, S.T, M.Cs, M.Kom. selaku ketua Program Studi Teknik Informatika.
5. Teman-teman praktisi *development operations* dari PT. Global Digital Niaga yang senantiasa membantu dan memberikan dukungan dalam pengerjaan skripsi ini.
6. Teman-teman kelompok komputasi berbasis jaringan Teknik Informatika yang saling memberikan dukungan selama proses pengerjaan skripsi.

Penulis menyadari sepenuhnya bahwa laporan penelitian ini jauh dari sempurna dikarenakan adanya keterbatasan wawasan dan kemampuan sehingga kritik dan saran yang bersifat membangun sangat diharapkan. Di samping itu, penulis berharap semoga laporan ini dapat bermanfaat bagi penulis dan pembaca.

Malang, 8 Juli 2020

Penulis
kevinchou@student.ub.ac.id

Repository Universitas Brawijaya
REPOSITORY.UB.AC.ID
UNIVERSITAS BRAWIJAYA

ABSTRAK

Kevin Jonathan Harnanta, Implementasi MQTT Broker Dengan Kemampuan *Auto Scaling* Pada *internet of Things*

Pembimbing: Adhitya Bhawiyuga, S.Kom., M.Sc. dan Achmad Basuki, S.T, M.MG, Ph.D

Internet of Things merupakan suatu topik pembahasan yang berkembang sangat pesat saat ini. Hal ini juga berdampak terhadap pertumbuhan perangkat IoT yang semakin hari semakin banyak. Salah satu protokol yang biasa digunakan dalam komunikasi antar perangkat IoT adalah MQTT. Dalam komunikasi perangkat-perangkat tersebut saling terhubung dengan perantara *MQTT broker*. Namun seiring dengan pertumbuhan perangkat IoT yang melakukan permintaan koneksi ke MQTT broker, MQTT broker menjadi titik poin kegagalan tunggal. Oleh karena itu dibutuhkan MQTT broker yang mampu memiliki kemampuan *auto scaling* sehingga mampu beradaptasi terhadap pertumbuhan permintaan koneksi dari perangkat-perangkat IoT. Pada implementasinya MQTT broker diletakkan di dalam kontainer dan dilakukan orkestrasi dengan kubernetes sehingga memiliki kemampuan *auto scaling*. Pada penelitian ini dilakukan perbandingan antara MQTT broker dengan kemampuan *auto scaling* dan tanpa kemampuan *auto scaling*. Pengukuran jumlah maksimal klien yang dapat terkoneksi dan jumlah koneksi terputus dilakukan pada saat pengujian dijalankan untuk mendapatkan hasil perbandingan. Hasilnya, MQTT broker dengan kemampuan *auto scaling* dapat menerima permintaan koneksi dari klien sebanyak 21463, sedangkan MQTT broker tanpa kemampuan *auto scaling* hanya dapat menampung 17030 koneksi. Hal ini menunjukkan MQTT broker dengan kemampuan *auto scaling* memberikan performa 26.03% lebih baik daripada MQTT broker tanpa kemampuan *auto scaling*. Sedangkan pada pengukuran jumlah koneksi yang terputus, MQTT broker dengan kemampuan *auto scaling* mengalami koneksi terputus sebanyak 9213 yang berarti sebanyak 42.92% dari jumlah koneksi maksimum koneksinya terputus. Pada MQTT broker tanpa mekanisme auto scaling terdapat 16577 koneksi yang terputus yang berarti sebanyak 97.34% dari jumlah koneksi maksimal yang dapat ditampung koneksinya terputus. Hal ini menunjukkan bahwa MQTT broker dengan kemampuan *auto scaling* mampu memberikan 54.42% koneksi terputus lebih sedikit daripada MQTT broker tanpa kemampuan *auto scaling*.

Kata Kunci: *internet of Things, MQTT, Container, Kubernetes, Auto Scaling*

Repository Universitas Brawijaya

ABSTRACT

Kevin Jonathan Harnanta, Implementasi MQTT Broker Dengan Kemampuan *Auto Scaling* Pada *Internet Of Things*

Supervisors: Adhitya Bhawiyuga, S.Kom., M.Sc. and Achmad Basuki. S.T, M.MG, Ph.D

Internet of Things is a topic of discussion that is growing rapidly now. One protocol commonly used in communication between IoT devices is MQTT. In communication, these devices are interconnected with brokers MQTT. But along with the growth of IoT devices that make connection requests to MQTT brokers, MQTT brokers have decreased resources, and become a single point of failure. Therefore we need MQTT brokers who are able to have auto scaling capabilities so they can adapt to the growing demand for connections from IoT devices. In its implementation, MQTT broker is placed in a container and orchestrated with kubernetes so that it has the ability to auto scaling. In testing, a comparison is made between MQTT brokers with auto scaling capability and without auto scaling capability. Testing is done by measuring the maximum number of clients that can be connected, and the number of connections lost at the time the test is running. As a result, MQTT brokers with auto scaling capabilities can receive connection requests from clients totaling 21463, while MQTT brokers without auto scaling capabilities can only accommodate 17030 connections. This shows that MQTT broker with auto scaling capability gives 26.03% better performance than MQTT broker without auto scaling capability. Whereas in the measurement of the number of disconnected connections, MQTT brokers with auto scaling capabilities experienced 9213 disconnected connections, which means as much as 42.92% of the maximum number of connections lost. In MQTT broker without auto scaling mechanism, there are 16577 connections that are lost, which means 97.34% of the maximum number of connections that can be accommodated. This shows that MQTT broker with auto scaling capability is able to provide 54.42% fewer disconnected connections than MQTT broker without auto scaling capability.

Keywords: Internet of Things, MQTT, Container, Kubernetes, Auto Scaling

DAFTAR ISI

Repository Universitas Brawijaya
REPOSITORY.UB.AC.ID
UNIVERSITAS BRAWIJAYA

Repository Universitas Brawijaya
REPOSITORY.UB.AC.ID
UNIVERSITAS BRAWIJAYA

DAFTAR TABEL

Repository Universitas Brawijaya
REPOSITORY.UB.AC.ID
UNIVERSITAS BRAWIJAYA

DAFTAR GAMBAR

Repository Universitas Brawijaya
UNIVERSITAS BRAWIJAYA
REPOSITORY.UB.AC.ID

DAFTAR GRAFIK

Repository Universitas Brawijaya
REPOSITORY.UB.AC.ID
UNIVERSITAS BRAWIJAYA

BAB 1 PENDAHULUAN

1.1 Latar Belakang

Internet of Things (IoT) merupakan suatu topik pembahasan yang berkembang sangat pesat saat ini. Hal ini dapat dilihat pada data yang didapat, jumlah dari perangkat IoT terus meningkat, diprediksikan pada tahun 2020 akan mencapai 50 billion perangkat (Burhan, Rehman, Khan, & Kim, 2018). Aspek komunikasi pada IoT menjadi salah satu dari enam aspek umum yang penting terhadap peningkatan jumlah perangkat IoT (Al-Fuqaha et al., 2015), karena aspek ini berfungsi untuk menentukan layanan komunikasi antar perangkat IoT.

Dalam aspek komunikasi salah satu protokol yang digunakan pada perangkat IoT adalah MQTT. Dalam komunikasinya, protokol MQTT memakai model komunikasi *publish subscribe.* Dalam model komunikasi *publish subscribe* terdapat tiga komponen utama yaitu *publisher, subscriber,* dan *message broker. Publisher* merupakan perangkat IoT yang secara berkala menerbitkan data sensor. *Subscriber* merupakan perangkat atau aplikasi yang berlangganan data yang diterbitkan oleh publisher. *Message broker* merupakan suatu entitas yang bertugas untuk menampung data yang diterbitkan oleh *publisher* dan mengirimkannya ke *subscriber* sesuai topik yang diinginkan (Simpson & Us, 2019).

Seiring bertambahnya perangkat IoT di internet, semakin banyak permintaan koneksi yang terjadi ke *message broker* baik itu dari *publisher* maupun *subscriber*. Hal tersebut menyebabkan sumber daya dari *message broker* berkurang dan menjadi titik pusat kegagalan tunggal. Pada saat sumber daya message broker sudah habis, message broker tidak mampu menangani permintaan koneksi lagi dan terjadi kegagalan. Hal ini menyebabkan, pesan dari *publisher* akan hilang, dan *subscriber* harus melakukan *subscribe* ulang ketika *message broker* sudah kembali hidup. Oleh karena itu diperlukan kemampuan skalabilitas pada *message broker*. Skalabilitas merupakan kemampuan suatu objek untuk beradaptasi dengan perubahan di lingkungan (Gupta et al., 2017). Dengan adanya kemampuan skalabilitas pada *message broker,* permintaan koneksi yang bertambah setiap saat mampu ditangani.

Pada penelitian terkait kemampuan skalabilitas, dilakukan proses klasterisasi MQTT broker. Pada penelitian tersebut lalu lintas koneksi yang masuk akan dilakukan *load balancing* keenam MQTT broker yang berada pada raspberry pi (Jutadhamakorn et al., 2018). Sayangnya dalam implementasi tersebut alokasi sumber daya bersifat statis, sehingga ketika tidak ada tugas, sumber daya yang terpakai akan tetap, yang mengakibatkan sumber daya terbuang secara percuma. Untuk itu diperlukan suatu mekanisme lain yang disebut *auto scaling.* Mekanisme ini akan memastikan bahwa suatu aplikasi dapat melakukan penskalaan secara otomatis dengan tetap menggunakan sumber daya secara efisien, dan tetap menjaga biaya operasional tetap rendah(Roy, Dubey, & Gokhale, 2011). Pada penelitian terkait *auto scaling* yang didapat, diterapkan auto scaling pada aplikasi berbasis web. Pada penelitian tersebut auto scaling dilakukan pada tingkat *mesin virtual* (Fernandez, Pierre, & Kielmann, 2014). auto

scaling yang dilakukan pada level mesin virtual memiliki kekurangan yaitu dalam hal kecepatan dan efisiensi sumber daya (Muzzammel, 2019).

1.2 Identifikasi Masalah

Penelitian ini dimulai dan dilatar belakangi oleh pertumbuhan perangkat IoT yang sangat pesat. Dalam komunikasinya antar perangkat IoT dalam bertukar pesan dibutuhkan sebuah perangkat lunak *message broker*. *Message* broker sering kali langsung menangani banyak *publisher* atau *subscriber* yang terus bertambah, dan menjadi titik pusat kegagalan tunggal (The HiveMQ Team, 2019). Pada saat sumber daya pada *message broker* sudah habis, *message broker* tidak mampu menangani permintaan koneksi lagi. Ketika sudah terjadinya kegagalan dalam *message broker,* pesan dari *publisher* akan hilang, dan *subscriber* harus melakukan *subscribe* ulang ketika *message broker* saat sudah kembali hidup. Oleh karena itu untuk mengatasi hal tersebut *message broker* perlu memiliki kemampuan skalabilitas.

Pada penelitian terkait dilakukan proses klasterisasi MQTT broker. Sayangnya dalam implementasi tersebut alokasi sumber daya bersifat statis, sehingga ketika tidak ada lalu lintas koneksi yang masuk sumber daya yang terpakai akan tetap, yang mengakibatkan sumber daya terbuang secara percuma (Jutadhamakorn et al., 2018). Dengan demikian diperlukan MQTT broker yang memiliki kemampuan *auto scaling* yang dapat memastikan bahwa MQTT *broker* dapat melakukan penskalaan otomatis, dengan tetap menjaga penggunaan sumber daya yang ada. Selain itu untuk menangani pertumbuhan permintaan koneksi baik itu dari *publisher* atau *subscriber* yang begitu cepat, MQTT broker perlu melakukan proses penskalaannya dengan cepat. Pada penskalaan level mesin virtual, penskalaan terjadi secara lambat (Muzzammel, 2019). Salah satu cara untuk menangani lambatnya proses penskalaan pada level mesin virtual adalah dengan menerapkan penggunaan kontainer sebagai pengganti mesin virtual. Kontainer merupakan standar perangkat lunak yang mampu mengemas kode perangkat lunak dan semua yang dibutuhkan sehingga aplikasi dapat berjalan. Penggunaan kontainer mampu memangkas waktu boot mesin virtual karena kontainer berbagi kernel dengan sistem operasi mesin sehingga tidak memerlukan sistem operasi untuk setiap aplikasi (Docker, 2015).

Penelitian ini bertujuan untuk mengimplementasikan kemampuan *auto scaling* pada MQTT broker sehingga mampu beradaptasi terhadap pertumbuhan permintaan koneksi dari perangkat-perangkat IoT yang semakin hari semakin banyak. MQTT broker tersebut dimasukkan ke dalam kontainer dan dilakukan orkestrasi melalui layanan *cloud*. Layanan *cloud* pada penelitian ini digunakan untuk memberikan layanan perangkat komputasi dan penyimpanan yang seolah-olah tidak terbatas (Botta, De Donato, Persico, & Pescape, 2014). Selain itu, layanan *cloud* yang digunakan pada penelitian ini mampu membuat MQTT broker menjadi memiliki kemampuan *auto scaling*. Dalam melakukan *auto scaling* diperlukan parameter agar *auto scaling* dapat terjadi. Terdapat tiga parameter *auto scaling* yang ditetapkan pada penelitian ini yaitu: penggunaan CPU, penggunaan memori, dan jumlah klien yang terkoneksi. Pada penerapan

auto scaling selain membutuhkan parameter, juga dibutuhkan suatu angka ambang batas agar *auto scaling* dapat terjadi (Kubernetes, 2018). Seiring terjadinya peningkatan permintaan koneksi, penggunaan sumber daya pada MQTT broker akan meningkat. Pada saat rata-rata penggunaan sumber daya MQTT broker melebihi angka ambang batas yang telah ditetapkan, maka akan dilakukan penambahan kontainer MQTT broker. Begitu juga ketika jumlah koneksi perangkat-perangkat yang terhubung tersebut berkurang dan rata-rata penggunaan berada di bawah ambang batas yang telah ditetapkan, akan dilakukan pengurangan kontainer MQTT broker.

1.3 Rumusan Masalah

Pertanyaan penelitian yang dibahas dalam bagian ini adalah sebagai berikut:

1. Bagaimana arsitektur dan implementasi MQTT broker dengan kemampuan *auto scaling* menggunakan Layanan Cloud?
2. Bagaimana mekanisme kerja *auto scaling* pada MQTT broker ?
3. Bagaimana kinerja MQTT broker dengan kemampuan *auto scaling* ditinjau dari parameter ukur jumlah maksimal klien terkoneksi, dan jumlah koneksi yang terputus ?

1.4 Tujuan

Tujuan dari penelitian ini adalah sebagai berikut:

1. Mengimplementasikan mekanisme *auto scaling* pada broker MQTT di *cloud*.
2. Menguji kemampuan *auto scaling* MQTT broker dalam menghadapi pertumbuhan permintaan koneksi klien.

1.5 Manfaat

Manfaat yang diharapkan dari penelitian ini adalah:

1. Dengan penelitian ini diharapkan dapat memberikan rancangan arsitektur MQTT Broker yang memiliki skalabilitas dan mampu mengatasi pertumbuhan.
2. Pengembang lain mampu mengetahui kelebihan dari *auto scaling* pada MQTT broker.
3. Dengan adanya pengembangan sistem ini, diharapkan penulis dapat menerapkan ilmu yang didapat dari program pendidikan strata satu pada Informatika Universitas Brawijaya dan mendapatkan wawasan mengenai ilmu pengetahuan dan konsep pemahaman dari penelitian.

1.6 Batasan Masalah

Agar pembahasan tidak mengarah ke hal yang lebih luas, penulis menekankan batasan – batasan masalah dalam penelitian sebagai berikut:

1. Arsitektur yang diimplementasikan disini hanya difokuskan pada layanan MQTT broker.

2. Jenis MQTT broker yang dipakai dalam penelitian ini hanya Mosquitto.
3. Platform orkestrasi yang digunakan pada penelitian ini hanya menggunakan kubernetes.
4. Parameter yang dipakai sebagai bahan dari terjadinya *auto scaling* sebanyak tiga parameter.

1.7 Sistematika Pembahasan

Untuk memudahkan pembaca dalam memahami penelitian ini, penulis membuat sistematika penulisan yang berisikan rangkuman setiap bab yang terdapat pada penelitian sebagai berikut :

BAB I PENDAHULUAN

Bab ini menjelaskan mengenai latar belakang tentang kelemahan kelemahan yang ada dari penelitian sebelumnya. Selain itu pada bab ini juga dijelaskan metode yang ditawarkan untuk mengatasi permasalahan tersebut.

BAB II LANDASAN PUSTAKA

Bab ini menjelaskan mengenai penelitian terkait yang sudah dilakukan atau penelitian yang menjadi landasan teori penelitian ini. Penelitian – penelitian terkait yang digunakan dipilih berdasarkan objek atau metode yang digunakan penelitian tersebut yang memiliki hubungan terhadap penelitian mengenai arsitektur MQTT broker, Kubernetes, dan horizontal pod autoscaler.

BAB III METODOLOGI PENELITIAN

Pada bab ini akan dijelaskan terkait dengan kerangka penelitian, perancangan sistem, dan metode evaluasi. Kerangka penelitian tersebut berisi penjelasan ulang terkait latar belakang masalah, selain itu juga dijelaskan apa yang menjadi solusi untuk mengatasi permasalahan tersebut.

Subbab perancangan sistem membahas terkait sistem yang akan dirancang dalam penelitian ini. Perancangan implementasi yang berada dalam penelitian ini terkait mekanisme MQTT broker pada IoT. Berisi mulai dari topologinya dan mekanisme auto scalingnya.

BAB IV IMPLEMENTASI

Pada bab ini dibahas tentang langkah-langkah implementasi dari sistem yang sudah dirancang pada bab metodologi penelitian. Implementasi disini yang dibahas terkait implementasi mekanisme auto scaling pada MQTT broker, yang dalam hal ini berisi pemasangan dan konfigurasi auto scaling, serta pemasangan *load balancer*.

BAB V PENGUJIAN DAN PEMBAHASAN

Pada bab ini dibahas tentang hasil dari pengujian yang telah dirancang pada metode evaluasi di bab metodologi penelitian.

Repository Universitas Brawijaya

BAB VI KESIMPULAN

Bab ini memuat kesimpulan yang menjawab rumusan masalah berdasarkan hasil penelitian secara keseluruhan dan juga memuat saran – saran untuk mekanisme auto scaling pada MQTT broker.

BAB 2 LANDASAN KEPUSTAKAAN

Pada penelitian ini kajian pustaka yang dilakukan mengacu pada penelitian-penelitian sebelumnya. Penelitian tersebut juga akan digunakan untuk menjadi landasan bagi penelitian ini. Tidak lupa juga peneliti menggali informasi yang berkaitan dengan penelitian ini baik secara *offline* maupun *online*. Semua informasi yang didapat dalam proses studi literatur menjadi bahan pertimbangan, baik itu kelebihan dan kekurangannya.

Referensi pertama, yang menjadi landasan penelitian ini adalah penelitian yang berjudul "A Scalable and Low-Cost MQTT broker Clustering System" . Pada penelitian tersebut dijelaskan terdapat MQTT broker 6 node yang dilakukan load balancing menggunakan Nginx. Selain itu pada penelitian tersebut MQTT broker diletakkan pada raspberry pi yang memiliki sumberdaya terbatas. Sayangnya dalam implementasi tersebut alokasi sumber daya bersifat statis, sehingga ketika tidak ada lalu lintas koneksi yang masuk sumber daya yang terpakai akan tetap, yang mengakibatkan sumber daya, dan biaya operasional terbuang secara percuma. Perbedaan penelitian ini dengan penelitian tersebut yaitu pada penelitian ini akan diimplementasikan mekanisme auto scaling untuk menjaga sumber daya agar tidak terbuang secara sia-sia.

Referensi yang kedua, yang menjadi landasan penelitian ini adalah penelitian terkait auto scaling yang dilakukan oleh Hector Fernandez dan teman-temannya. Pada penelitian itu menerapkan auto scaling pada aplikasi berbasis web, namun dalam penelitian tersebut *auto scaling* dilakukan pada tingkat mesin virtual (Fernandez et al., 2014). *Auto scaling* yang dilakukan pada tingkat memiliki kekurangan yaitu dalam hal kecepatan dan efisiensi resource. Hal tersebut juga diakui oleh Muhammad Abdullah dan teman-temannya dalam penelitian yang berjudul "Container vs Virtual Machine Auto-scaling Multi-tier Applications Under Dynamically Increasing Workloads". Penelitian ini menggunakan auto scaling pada level kontainer sehingga proses penskalaan terjadi lebih cepat dibandingkan pada mesin virtual (Muzzammel, 2019).

Referensi yang ketiga yang menjadi landasan penelitian ini terkait auto scaling pada broker MQTT. Penelitian tersebut memanfaatkan kubernetes sebagai platform untuk melakukan penyebaran MQTT broker yang dalam penelitian tersebut merupakan HiveMQ (Baier, 2020). Hanya saja pada penelitian tersebut mekanisme auto scaling dibuat sendiri menggunakan metode *long short term memory network* oleh peneliti sehingga pada perhitungan auto scalingnya lebih mengarah pada mekanisme prediksi terkait kapan suatu MQTT broker dilakukan penskalaan, sehingga proses penskalaan dapat tertunda karena melakukan pekerjaan lain selain seperti mengecek setiap status MQTT broker. Berbeda dengan penelitian ini, pada penelitian ini auto scaling akan terjadi secara langsung ketika terjadi suatu peningkatan pada sumber daya pada MQTT broker.

Tabel 2.1 Kajian Pustaka

No.	Nama Peneliti, Tahun, Judul	Persamaan	Perbedaan	
			Penelitian Terdahulu	Rencana Penelitian
1.	Pongnapat Jutadhamakorn, Tinnapat Pillavas, Vasaka Visoottiviseth, Ryousei Takano, Jason Haga, Dylan Kobayashi, 2017, *"A Scalable and Low-Cost MQTT Broker Clustering System"*	Penggunaan mqtt broker, dan kontainerisasi	MQTT broker diletakkan pada raspberry pi, tidak ada mekanisme auto scaling	MQTT broker diletakkan pada Cloud, terdapat mekanisme auto scaling
2.	Hector Fernandez, Guillaume Pierre, Thilo Kielmann, 2014, *"auto scaling Web Applications in Heterogeneous Cloud Infrastructures"*	Penggunaan auto scaling	Auto scaling dilakukan pada tingkat mesin virtual	Auto scaling dilakukan pada tingkat kontainer
3.	Simon Baier, 2020 *"Towards Zero Maintenance Operation of an MQTT Broker with a Kubernetes Operator"*	implementasi auto scaling pada MQTT broker, Penggunaan prometheus sebagai sumber parameter *auto scaling*.	Melakukan proses *auto scaling* menggunakan metode *long short term memory network*.	Melakukan proses *auto scaling* berdasarkan sumber daya MQTT broker dan jumlah klien yang terkoneksi.

2.1 Internet of Things (IoT)

Internet of Things atau yang selanjutnya disebut sebagai IoT merupakan suatu paradigma dimana dalam mengumpulkan dan mengirimkan data dalam dilakukan oleh komputer dan tanpa bantuan manusia. Data yang dikumpulkan dapat berasal dari mana saja, kemudian datanya digunakan untuk mengetahui suatu informasi yang ingin dicari dengan tujuan membantu manusia (Ashton Kevin, 2012). Pada hakikatnya IoT memiliki karakteristik yaitu tersusun dari perangkat-perangkat kecil yang bekerja dengan cara mengambil data dari lingkungan dan saling bertukar data dengan perangkat lain agar data tersebut dapat diolah menjadi informasi yang diinginkan.

Gambar 2.1 Elemen Internet of Things

Selain IoT memiliki karakteristik tertentu, IoT juga memiliki aspek-aspek umum penyusun agar sebuah sistem IoT terbentuk secara utuh, yaitu *identification*, *sensing*, *communication*, *computation*, *service* dan *semantic*. Berikut penjelasan untuk masing-masing aspek:

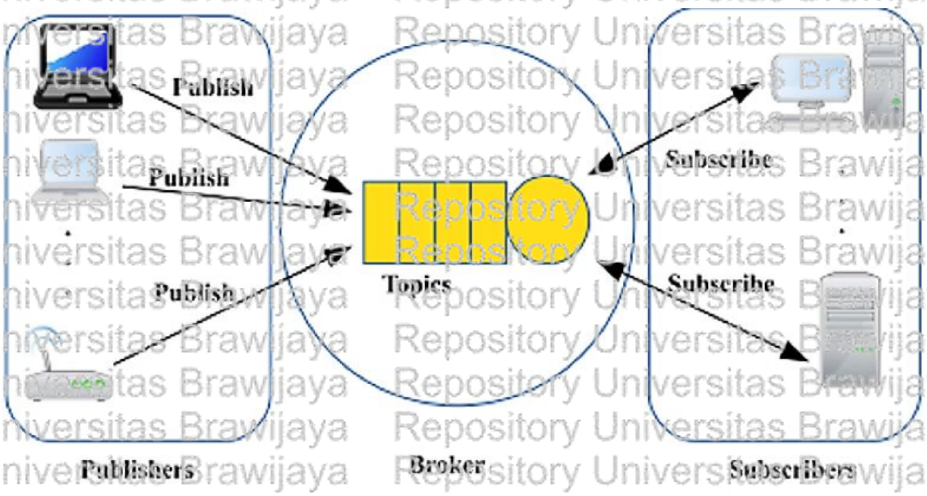
1. *Identification*, aspek ini berfungsi untuk mengenali perangkat IoT lain. Setiap perangkat IoT wajib memiliki ID agar perangkat tersebut dapat diidentifikasi oleh perangkat lain.
2. *Sensing*, aspek ini berfungsi untuk membaca kondisi di sekitar yang akan berbeda tergantung dimana suatu perangkat diletakkan.
3. *Communication*, aspek ini berfungsi untuk menentukan jenis layanan komunikasi yang dipakai antar perangkat IoT
4. *Computation*, aspek ini berfungsi untuk mengolah data yang telah didapatkan dari aspek sensing
5. *Services*, aspek ini berfungsi untuk menentukan layanan apa saja yang diperlukan pada suatu sistem IoT
6. *Semantics*, aspek ini berfungsi untuk mengekstrak pengetahuan dari data yang dikumpulkan oleh sistem IoT

2.2 Message Queuing Telemetry Transport (MQTT)

MQTT merupakan salah satu jenis protokol yang biasa digunakan pada IoT dengan model komunikasi publish subscribe, dan memiliki fitur ringan yang cocok untuk lingkungan terbatas (Akbar, Amron, Mulya, & Hanifah, 2018). Dalam model komunikasi *publish subscribe* terdapat tiga komponen utama yaitu *publisher*, *subscriber*, dan *message broker*. Dalam model komunikasi *publish subscribe* terdapat tiga komponen utama yaitu *publisher*, *subscriber*, dan *message broker*. *Publisher* merupakan perangkat IoT yang secara berkala

menerbitkan data sensor. *Subscriber* merupakan perangkat atau aplikasi yang berlangganan data yang diterbitkan oleh publisher. *Message broker* merupakan suatu entitas yang bertugas untuk menampung data yang diterbitkan oleh *publisher* dan mengirimkannya ke *subscriber* (Simpson & Us, 2019).

Untuk memastikan bahwa data yang diterbitkan oleh *publisher* benar-benar data yang diinginkan oleh *subscriber*, pada MQTT terdapat istilah topik. Topik merupakan entitas yang digunakan pada *broker* untuk menyaring data dari *publisher* sehingga dapat tepat dikirimkan ke *subscriber* yang menginginkannya. Klien MQTT merupakan pengguna dari sistem, yang dalam hal ini merupakan *publisher* dan *subscriber*. MQTT memiliki tiga level jaminan agar data dapat tersampaikan dari *publisher ke* subscriber. QoS level 0 merupakan jaminan penyampaian dengan kondisi pesan terkirim paling banyak satu kali. QoS level 1 merupakan jaminan penyampaian pesan dengan kondisi pesan terkirim setidaknya satu kali. QoS level 2 merupakan jaminan penyampaian pesan dengan kondisi pesan terkirim tepat satu kali.



Gambar 2.2 Arsitektur MQTT

2.3 Layanan Cloud

Layanan cloud merupakan model layanan yang memberikan akses jaringan dari mana saja, selain itu juga memungkinkan pengguna untuk mampu melakukan konfigurasi terhadap perangkat (misalnya: jaringan, server, penyimpanan, aplikasi) sesuai permintaan (Rao, Saluia, Sharma, Mittal, & Sharma, 2012).

Untuk memungkinkan konsumen memilih layanan yang sesuai dengan mereka, layanan dalam komputasi *Cloud* disediakan pada tiga tingkatan yang berbeda, yaitu yang pertama model *Software as a Service* (SaaS), di mana perangkat lunak dikirimkan melalui Internet kepada pengguna (misalnya: Google Drive) model yang kedua adalah *Platform as a Service* (PaaS), yang menawarkan tingkat lingkungan terintegrasi yang lebih tinggi yang dapat membangun, menguji, dan menggunakan perangkat lunak tertentu (misalnya: Heroku). Model yang terakhir adalah *Infrastruktur as a Service* (IaaS), infrastruktur seperti

penyimpanan, perangkat keras, dan server dikirimkan sebagai layanan (mis. Layanan Web Amazon). Pada penelitian ini, layanan yang akan dipakai adalah level layanan IaaS, berupa layanan kubernetes

2.4 Docker Container

Dilansir pada situs AWS, Docker merupakan sebuah platform perangkat lunak yang memungkinkan para developer / pengembang, membuat, menguji, dan menyebarkan aplikasi dengan cepat. Docker membantu mengemas kode program, pustaka, dalam suatu wadah yang disebut kontainer (AWS, 2018). Pada situs resmi docker juga menjelaskan bahwasannya Docker merupakan platform untuk *developer* dan *admin* untuk mengembangkan, mengirim, dan menjalankan aplikasi (Docker, 2015).

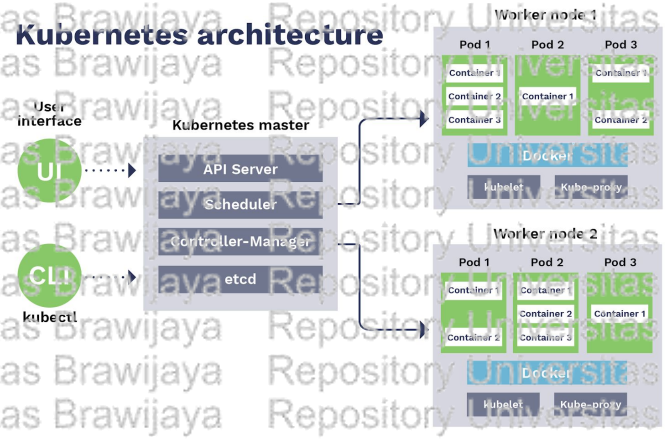
Dalam komunikasinya dengan sistem operasi, docker menggunakan *interprocessing communication* berbasis *unix socket*. Docker hanya dapat berjalan pada sistem operasi Linux dikarenakan *library* dari kontainer hanya tersedia di Linux. Namun dalam implementasinya pihak pengembang docker menyediakan docker untuk windows dan mac. Pada kenyataannya, docker akan membuat mesin virtual kecil tersembunyi yang menggunakan *host OS Linux* agar docker dapat berjalan.

Kontainer docker menjalankan suatu entitas yang bernama *docker images*. Docker images merupakan suatu *template* yang berisi dasar-dasar dari sistem operasi, atau dapat menampung aplikasi yang ingin dijalankan dalam kontainer tersebut. Dalam pembuatan *images*, docker akan membaca instruksi yang berada pada file dengan nama Dockerfile (Bernstein, 2014). Instruksi yang berada pada Dockerfile ini merupakan langkah-langkah yang dilakukan untuk membuat *images* baru seperti menambahkan layanan MQTT broker dan mengijinkan permintaan koneksi masuk pada port tertentu.

Gambar 2.3 Logo Docker

2.5 Kubernetes

Dilansir dari situs resmi Kubernetes, kubernetes merupakan sistem open-source untuk mengotomasi penyebaran, penskalaan, dan pengelolaan kontainer (Kubernetes, 2014). Kubernetes memiliki banyak sekali komponen seperti pada gambar 2.4.



Gambar 2.4 Komponen Kubernetes

Gambar 2.4 menunjukkan komponen kubernetes secara keseluruhan. Pada gambar tersebut terlihat ada dua jenis *node, kubernetes* master, dan worker node. Kubernetes master menyediakan komponen *control plane* yang mengatur bagaimana pengguna berinteraksi dengan kubernetes. Selain itu komponen master berperan dalam proses pengambilan secara global pada cluster (contohnya, mekanisme penjadwalan kontainer), serta berperan dalam proses deteksi serta pemberian respon terhadap peristiwa yang berlangsung di dalam kluster (contohnya, penjadwalan kontainer MQTT broker baru apabila jumlah kontainer yang ada pada replication controller tidak terpenuhi). Pada kubernetes master juga terdapat *api server*. Komponen ini dapat melakukan pengambilan, pembuatan, memperbarui, dan menghapus sumber daya utama melalui protokol HTTP standar dengan metode (POST, PUT, PATCH, DELETE, GET). Biasanya pengguna akan menggunakan bantuan perangkat lunak *kubectl* dalam berinteraksi dengan master kubernetes. Secara tidak langsung perangkat lunak tersebut berinteraksi langsung terhadap api server kubernetes dalam melakukan pengambilan, pembuatan, memperbarui, dan menghapus sumber daya.

Worker node merupakan tempat dimana menjalankan aplikasi yang di *deploy* oleh pengguna. Pada worker node terdapat suatu komponen bernama pod. Pod adalah sekelompok satu atau lebih kontainer (seperti kontainer Docker), dengan penyimpanan atau jaringan bersama, dan digunakan secara spesifik untuk menjalankan kontainer. Konten Pod selalu ditempatkan bersama dan dijadwalkan bersama, dan dijalankan dalam konteks bersama. Pod memodelkan satu atau lebih kontainer aplikasi yang tergabung erat - dalam dunia pra-wadah, dieksekusi pada mesin fisik atau virtual yang sama akan berarti dieksekusi pada host logis yang sama (Kubernetes, 2017). Pod dalam kubernetes dijadwalkan di dalam worker node. Node dalam sebuah kubernetes adalah mesin seperti mesin virtual, server fisik, dll.

Selain itu pada worker node terdapat komponen yang bernama kubelet. Kubelet adalah "*node agent*" utama yang berjalan pada setiap node. Kubelet bertugas mendaftarkan node dengan apiserver pada kubernetes .Kubelet bekerja dalam bentuk PodSpec. PodSpec adalah objek YAML atau JSON yang

menjelaskan pod. Kubelet mengambil satu set PodSpec yang disediakan melalui berbagai mekanisme (terutama melalui *api server*) dan memastikan bahwa wadah yang dijelaskan dalam PodSpec tersebut berjalan dan sehat. Kubelet tidak mengelola kontainer yang tidak dibuat oleh Kubernetes(Kubernetes, 2014).

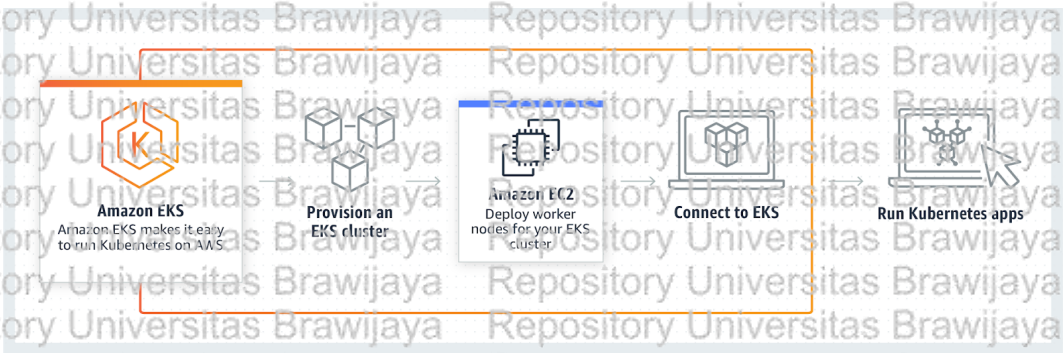
Dalam menjaga keberlangsungan pod dibutuhkan komponen bernama *Replication Controller*. Komponen ini dalam kubernetes juga memiliki tanggung jawab untuk menjaga jumlah pod agar jumlahnya sesuai dengan kebutuhan setiap objek *controller* replikasi yang ada di sistem. Jika terjadi suatu kegagalan dalam pod, contoh: jumlah pod MQTT broker berkurang. Dalam hal ini replication controller bertugas untuk membuat pod baru sesuai jumlah yang telah ditetapkan di file konfigurasi (Kubernetes, 2014).

Deployment pada kubernetes merupakan suatu jenis sumber daya yang mampu membantu penggunanya dalam mendefinisikan suatu kondisi terkait penyebaran aplikasinya di kubernetes. Seperti berapa jumlah pod yang diinginkan, *image* yang dipakai kontainer, berapa banyak kontainer yang digunakan dalam suatu pod, dan lain sebagainya. Deployment sendiri nantinya akan meminta bantuan replica set untuk menjaga agar pod tetap berjalan sesuai dengan yang didefinisikan di file konfigurasi (Kubernetes, 2014).

Dalam membuat *deployment*, klien perlu untuk terhubung dengan suatu komponen kubernetes yaitu master. Komponen master menyediakan *control plane* bagi cluster kubernetes. Komponen ini berperan dalam proses pengambilan secara global pada cluster (contohnya, mekanisme schedule), serta berperan dalam proses deteksi serta pemberian respons terhadap *events* yang berlangsung di dalam kluster (contohnya, penjadwalan pod MQTT broker baru apabila jumlah replika yang ada pada replication controller tidak terpenuhi).

Banyak sekali layanan *cloud* yang menyediakan layanan kubernetes. Salah satunya ialah layanan yang diberikan oleh pihak Amazon, yaitu Amazon Elastic Kubernetes (Amazon EKS). Amazon EKS memudahkan untuk menyebarkan, mengelola, dan skala aplikasi kemas menggunakan Kubernetes di AWS. Amazon EKS menjalankan infrastruktur manajemen Kubernetes untuk Anda di beberapa zona ketersediaan AWS untuk menghilangkan satu titik kegagalan.

Amazon EKS adalah distribusi Kubernetes bersertifikat sehingga dapat menggunakan alat yang ada dari mitra dan komunitas Kubernetes. Aplikasi yang berjalan pada lingkungan standar Kubernetes sepenuhnya kompatibel dan dapat dengan mudah dimigrasikan ke Amazon EKS. Amazon EKS umumnya tersedia untuk semua pelanggan AWS (AWS, 2019).



Gambar 2.5 Diagram alur Amazon Elastic Kubernetes Service

Salah satu fitur penting dari kubernetes yang dipakai pada penelitian ini adalah terkait mekanisme auto scaling pada level Pod. Secara singkat horizontal pod autoscaler merupakan fitur dari kubernetes yang berguna untuk melakukan auto scaling pada level pod dengan parameter-parameter yang telah ditetapkan. Untuk lebih jelasnya dijelaskan pada subbab 2.5 .

Selain itu masih banyak fitur-fitur lain kubernetes seperti Self-healing,DNS management, Load Balancing, Rolling update/ Rollback application, dan resource monitoring and logging. Oleh karena banyaknya fitur diatas kubernetes menjadi platform orkestrasi kontainer yang paling banyak digunakan di dunia (Kubernetes, 2014).

2.6 Skalabilitas

Berdasarkan jurnal yang didapatkan oleh peneliti, skalabilitas merupakan kemampuan perangkat untuk beradaptasi dengan perubahan di lingkungan dan memenuhi perubahan kebutuhan yang akan terjadi. Skalabilitas memiliki peran penting dalam membantu bekerja dan memanfaatkan sumber daya dengan baik (Gupta et al., 2017). Pada penelitian ini kemampuan skalabilitas digunakan untuk menangani pertumbuhan permintaan koneksi oleh perangkat-perangkat IoT.

Skalabilitas terbagi menjadi dua macam skalabilitas vertikal dan skalabilitas horizontal. Skalabilitas vertikal merupakan jenis skalabilitas dengan cara menambahkan sumber daya sebuah mesin, misalnya menambahkan jumlah processor, memory, kapasitas penyimpanan. Skalabilitas horizontal merupakan jenis skalabilitas dengan menambahkan jumlah mesin. Walaupun jumlah mesin lebih dari satu namun mesin-mesin tersebut bekerja secara bersamaan dalam menangani permintaan koneksi dari klien.

Setiap jenis skalabilitas memiliki kelebihan dan kekurangan. Skalabilitas vertikal memiliki kelebihan yaitu mengkonsumsi lebih sedikit daya jika dibandingkan dengan menjalankan beberapa mesin, selain itu mampu mengurangi upaya administratif yang dibutuhkan karena hanya menangani dan mengelola hanya satu mesin (Gupta et al., 2017). Kekurangan dari skalabilitas jenis ini yaitu jika terjadi kegagalan pada mesin tersebut, mesin tersebut tidak dapat lagi menangani permintaan klien karena jumlah mesin hanya satu.

Jenis skalabilitas horizontal memiliki kelebihan yaitu ketika suatu mesin mengalami kegagalan masih ada mesin lain yang masih dapat melayani permintaan klien. Kekurangan pada skalabilitas horizontal yaitu terletak pada upaya administratif dalam mengelola lebih dari satu mesin, dan konsistensi data antara mesin. Pada penelitian ini akan menggunakan jenis skalabilitas horizontal karena MQTT broker sering kali menjadi titik pusat kegagalan tunggal sehingga dibutunkan mesin-mesin tambahan agar MQTT broker tetap dapat melayani permintaan koneksi klien. Selain itu untuk menjaga agar biaya operasional tetap rendah dibutuhkan kemampuan *auto scaling* yang akan dibahas pada sub bab dibawah ini.

2.6.1 Auto Scaling

Dalam konteks skalabilitas, terdapat mekanisme yang disebut auto scaling. Mekanisme ini akan memastikan bahwa suatu aplikasi dapat melakukan penskalaan secara otomatis dengan tetap menggunakan sumber daya secara efisien, dan tetap menjaga biaya operasional tetap rendah(Roy et al., 2011). Pada penerapan *auto scaling* memerlukan dua komponen penting agar *auto scaling* dapat berfungsi dengan semestinya, yaitu parameter, dan angka ambang batas. Parameter merupakan suatu entitas sumber daya yang ingin dijadikan acuan perhitungan, sedangkan ambang batas merupakan angka yang menentukan batas penggunaan sumber daya suatu MQTT broker (Kubernetes, 2018).

2.7 Horizontal Pod Autoscaler

Pada penelitian terkait, dijelaskan bahwa *Horizontal Pod Autoscaler* merupakan suatu komponen dari kubernetes yang bertugas untuk melakukan proses auto scaling pada suatu pod, dengan memanfaatkan parameter penggunaan cpu (Minh Cao, 2019). Horizontal Pod Autoscaler diimplementasikan sebagai *controller*. *Controller horizontal pod autoscaler* secara berkala menyesuaikan jumlah replika pod dalam *controller deployment* agar sesuai dengan pemanfaatan rata-rata CPU atau parameter lain yang diamati dengan target yang ditentukan oleh pengguna.

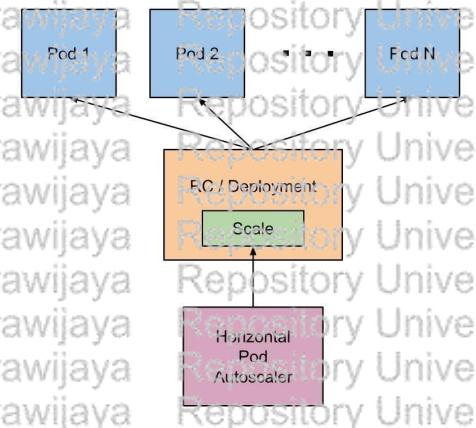
Selain penggunaan cpu pada setiap pod, *Horizontal Pod Autoscaler* juga dapat memakai parameter kustom yang dapat didapatkan dari berbagai macam sumber agar mendukung parameter lain selain penggunaan CPU. Untuk itu pada penelitian ini digunakan metrics server dan prometheus sebagai sumber metrik/parameter untuk penskalaan. Penggunaan metrics server, dan prometheus tersendiri akan dibahas pada sub bab selanjutnya.

Horizontal Pod Autoscaler memiliki algoritma untuk menghitung berapa jumlah pod yang dibuat ketika penggunaan sumber daya sudah melewati angka ambang batas yang telah ditetapkan. Berikut algoritmanya:

$$Jumlah\ Pod\ Baru = ceil[\ Jumlah\ Pod\ Sekarang\ * \frac{Nilai\ Penggunaan\ Sumber\ Daya\ Sekarang}{(Nilai\ Ambang\ Batas\ Sumber\ Daya)}\ ]$$

sebagai contoh ketika jumlah pod sekarang 1 pod, dengan penggunaan cpu 200 mili core, dan nilai ambang batas sumber daya 100 mili core, berarti 1 *

200/100 = 2. Maka jumlah pod yang akan tersedia untuk melayani permintaan klien adalah dua pod. Sedangkan ketika penggunaan cpu hanya 80 mill core hasil yang didapat adalah 0.8. Pada saat hasil yang didapat mendekati satu, maka tidak akan terjadi proses penambahan pod.



Gambar 2.6 Diagram Visualisasi Horizontal Pod Autoscaler

2.8 Metrics Server

Metrik Server merupakan aggregator seluruh data penggunaan sumber daya pada Kubernetes. Metrik Server mengumpulkan metrik sumber daya dari kubelet dan mengirimkan datanya ke apiserver Kubernetes melalui Metrik API untuk selanjutnya digunakan oleh Horizontal Pod Autoscaler (Kubernetes, 2019). Parameter yang didukung oleh Metrics Server adalah penggunaan CPU dan penggunaan memori setiap pod. Pada sumber yang didapat menyatakan walaupun dapat digunakan untuk melihat penggunaan cpu dan memori, metrics server tidak disarankan untuk digunakan sebagai perangkat untuk pemantauan suatu pod.

2.9 Prometheus

Prometheus adalah suatu perangkat lunak yang digunakan sebagai sistem pemantauan dan peringatan sistem yang awalnya dibuat di SoundCloud. (Prometheus, 2014). Prometheus akan mengumpulkan metrik dari target yang telah ditetapkan pada interval yang diberikan, mengevaluasi aturan ekspresi, menampilkan hasilnya, dan dapat memicu peringatan jika beberapa kondisi diamati benar. Target dalam prometheus merupakan sistem yang ingin dipantau, misalnya *web server, database server.*

Pada penelitian ini prometheus berfungsi untuk memantau data-data mengenai jumlah klien yang terkoneksi pada MQTT broker. Pada dasarnya Prometheus tidak dapat mengambil langsung data dari target yang diawasi. Diperlukan suatu perangkat lunak pihak ketiga yang disebut exporter. Perangkat lunak ini yang akan mengambil data ke sistem yang diawasi, dan akan membuat akses poin berbasis HTTP. Akses poin inilah yang biasa Prometheus gunakan untuk pengambilan data dari target.

Pada dasarnya prometheus sendiri tidak mampu mengirimkan data yang telah disimpan untuk dikirimkan ke kubernetes api server sebagai parameter dilakukannya *auto scaling*. Oleh sebab itu dibutuhkan perangkat lunak pihak ketiga yang mampu menangani kelemahan tersebut yaitu Prometheus Adapter.

Prometheus adapter merupakan perangkat lunak bebas yang bertugas untuk mengambil nilai dari Prometheus yang nantinya dapat dipakai sebagai sumber parameter penskalaan pada penelitian ini. Nilai-nilai tersebut diberikan ke apiserver kubernetes dapat dilihat melalui api yang disediakan oleh kubernetes dengan *path* "/apis/custom.metrics.k8s.io/" (Helm, 2019b). Tujuan penggunaan prometheus adapter pada penelitian ini agar mampu melengkapi kekurangan dari metrics server bawaan kubernetes yang hanya menyediakan metrik/parameter cpu dan memori saja.

2.10 Mosquitto

Mosquitto adalah broker pesan sumber terbuka (berlisensi EPL / EDL) besutan dari perusahaan Eclipse. MQTT broker ini mengimplementasikan protokol MQTT versi 5.0, 3.1.1 dan 3.1. Mosquitto ringan dan cocok untuk digunakan pada semua perangkat mulai dari komputer papan tunggal berdaya rendah hingga server penuh (Eclipse, 2018).

Dalam kaitannya dengan penelitian ini, Mosquitto digunakan sebagai *message broker* yang berguna untuk meneruskan pesan dari *publisher* ke subscriber. Dalam implementasinya Mosquitto ini dikemas menjadi *images*, agar nanti dapat dijalankan di kubernetes. Untuk membuat suatu kesatuan Mosquitto yang secara otomatis dapat di*deploy* langsung sekaligus beserta dengan *load balancer*nya, pada penelitian ini digunakan helm chart untuk mewujudkan itu.

Gambar 2.7 Logo Mosquitto

2.11 Helm Chart

Pada penjelasan yang didapatkan dari situs resmi Helm Chart , Helm adalah sebuah tools yang membantu para pengembang untuk mengelola aplikasi yang akan ditanamkan dan dijalankan di dalam pod kubernetes. Chart adalah kumpulan file yang menggambarkan sekumpulan sumber daya Kubernetes terkait (Helm, 2019a). Helm Chart dapat digunakan untuk membantu melakukan *deployment* sesuatu yang sederhana, seperti *memcached*, atau sesuatu yang kompleks, seperti instalasi wordpress secara lengkap mulai dari *web server, database* secara cepat.

Dapat diibaratkan, helm merupakan *package manager* seperti apt ,dnf , dan pacman pada distribusi linux. Chart merupakan file seperti *package pada linux* berekstensi deb, rpm pada distribusi linux. Chart dibuat sebagai file yang diletakkan di direktori tertentu, kemudian dapat dikemas ke dalam arsip yang memiliki versi untuk digunakan (Cloud Native Computing Foundation, 2019). Isi dari file tersebut merupakan *template* file konfigurasi terkait melakukan proses *deployment* di Kubernetes.

Pada penelitian ini akan dibuat helm-chart untuk MQTT broker, yang nantinya berguna bagi peneliti untuk mempermudah proses deployment MQTT broker di Amazon Elastic Kubernetes Service yang berada di cloud. Didalamnya terdapat beberapa file konfigurasi yang akan membuat proses deployment menjadi sangat dinamis, antara lain: pembuatan *load balancer*, sumberdaya hpa, dan sekaligus deployment MQTT broker. Selain itu helm-chart disini juga digunakan dalam deployment perangkat pihak ketiga lain seperti prometheus dan prometheus-adapter.

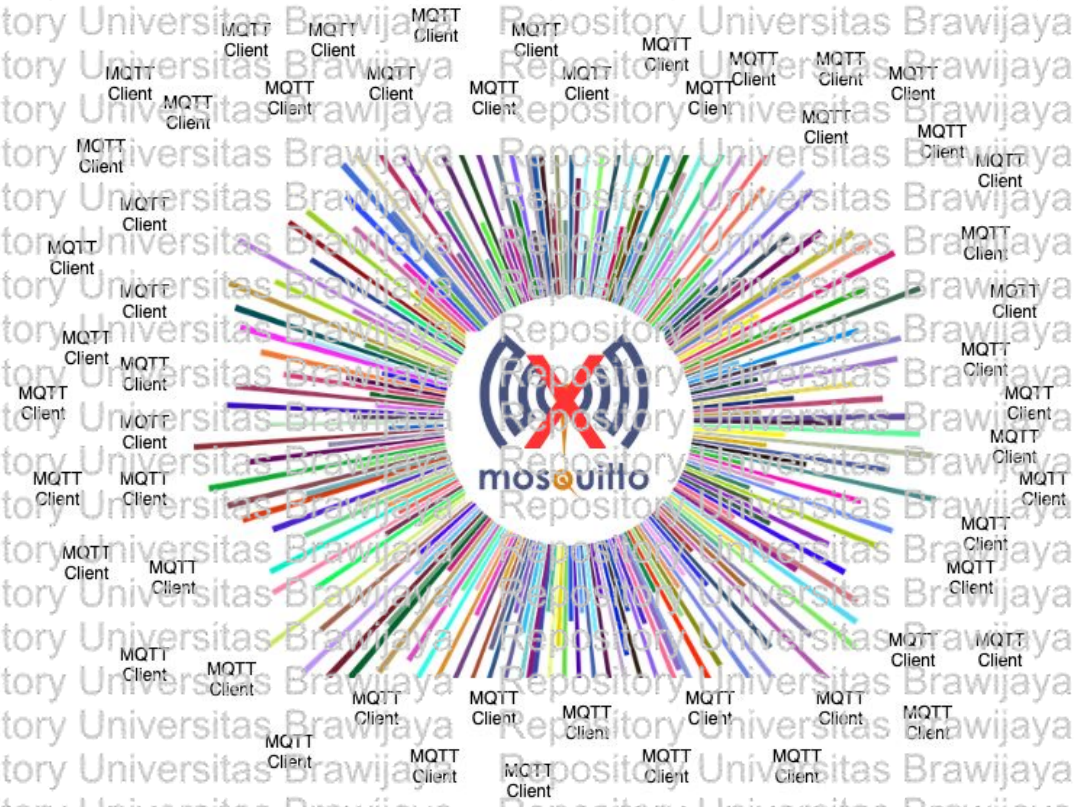
Gambar 2.8 Helm Chart

BAB 3 METODOLOGI PENELITIAN

Pada bab ini menjelaskan mengenai kerangka penelitian, perancangan sistem, dan metode evaluasi. Kerangka penelitian akan menjelaskan tujuan dari penelitian ini agar dapat tercapai. Subbab mengenai perancangan sistem, yang mampu menggambarkan suatu sistem secara utuh pada penelitian ini. Metode evaluasi yang nantinya menjadi dasar dari bab pengujian sistem yang dibuat.

3.1 Kerangka Penelitian

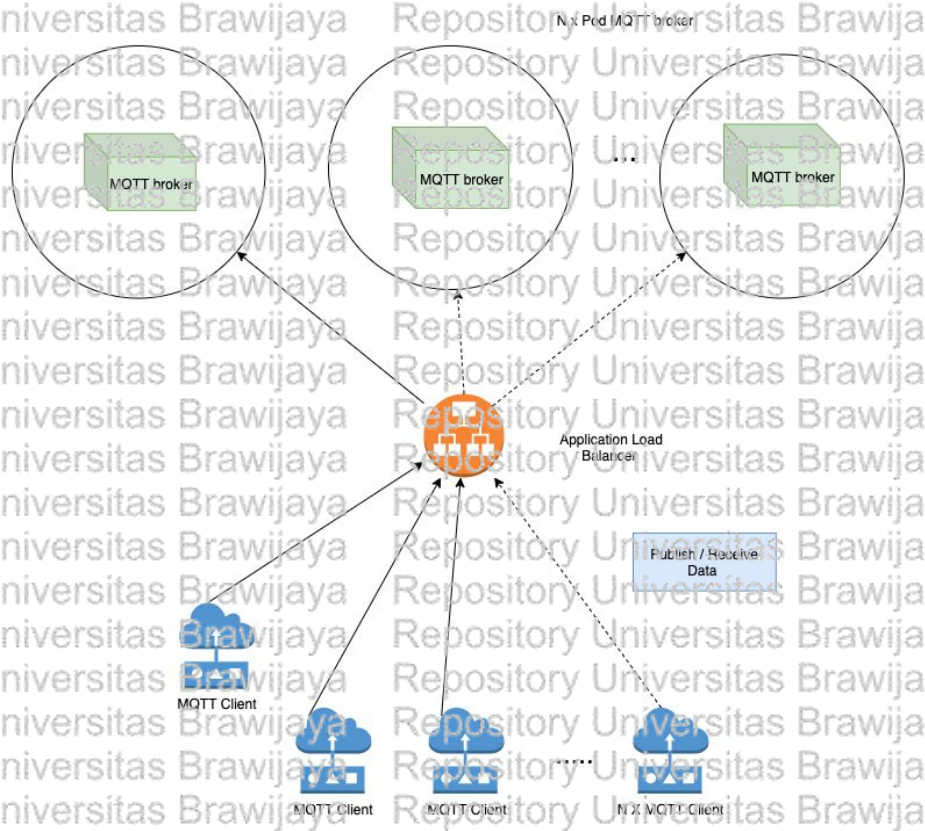
Penelitian ini dimulai dan dilatar belakangi oleh pertumbuhan perangkat IoT yang sangat pesat. Salah satu protokol yang sering ditemui dan digunakan dalam komunikasi antar perangkat IoT adalah MQTT. Dalam bertukar pesan antar perangkat IoT dengan protokol MQTT dibutuhkan sebuah perangkat lunak *MQTT broker*. Seiring bertambahnya perangkat IoT di internet, semakin banyak permintaan koneksi yang terjadi ke *MQTT broker* baik itu dari *publisher* maupun *subscriber*. Hal tersebut menyebabkan sumber daya dari *message broker* berkurang dan menjadi titik pusat kegagalan tunggal.



Gambar 3.1 Ilustrasi Permasalahan

Penelitian ini bertujuan untuk mengatasi pertumbuhan permintaan koneksi yang terjadi pada MQTT broker sehingga MQTT broker tidak mengalami kegagalan dengan cara mengimplementasikan kemampuan *auto scaling*. MQTT broker diimplementasikan menggunakan kontainer agar proses penskalaan

dengan cepat (Muzzammel, 2019). Selain itu untuk mengatasi terbatasnya sumberdaya yang biasa terjadi pada konteks IoT, dalam penelitian ini diintegrasikan IoT dan cloud.



Gambar 3.2 Tujuan Penelitian

3.2 Perancangan Sistem

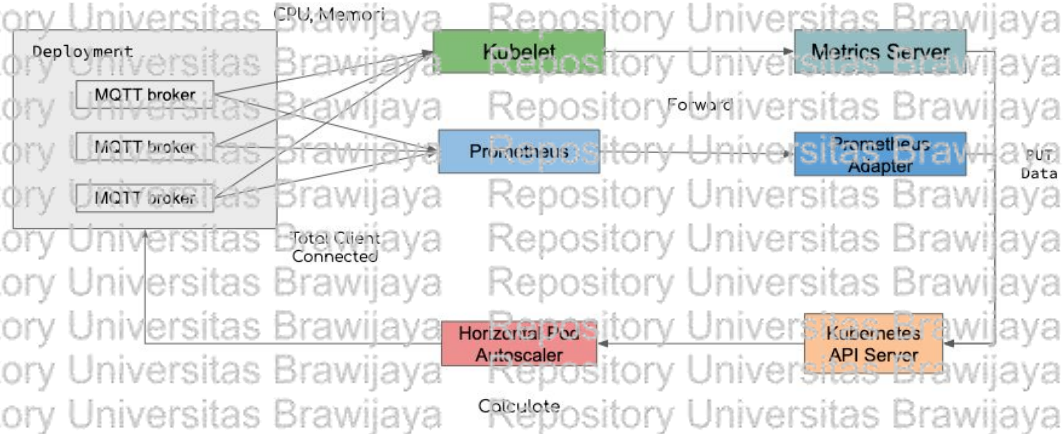
Perancangan sistem dilakukan untuk memperjelas bagaimana suatu sistem yang dibangun mampu memiliki kemampuan *auto scaling*. Selain itu perancangan arsitektur juga berfungsi agar sistem dapat diimplementasikan secara benar dan tepat. Pada penelitian ini akan hanya berfokus pada bagaimana auto scaling MQTT broker pada Internet of Things di cloud.

Salah satu pertimbangan mengapa diperlukannya *platform cloud* pada penelitian ini karena *cloud* memiliki tingkat komputasi, dan penyimpanan yang seolah-olah tidak terbatas yang mampu mengatasi kekurangan IoT. MQTT broker ini di *deploy* ke dalam layanan *cloud*. *Auto scaling* dilakukan pada tingkat kontainer atau dalam kubernetes disebut Pod. Dari rancangan yang telah dideskripsikan tersebut, peneliti merancang arsitektur seperti yang digambarkan pada gambar 3.2.

Pada gambar 3.2 terlihat sebanyak *N x MQTT client* akan melakukan *publish* atau *subscribe* ke MQTT broker melalui *load balancer* yang disediakan oleh layanan *cloud*. Permintaan yang disampaikan oleh MQTT client tadi akan diteruskan ke Pod-Pod yang secara bergantian. Begitupun sebaliknya nanti hasil

dari permintaan yang sudah diproses dari Pod-Pod akan dikembalikan ke MQTT client yang memberikan permintaan.

Pada penelitian ini *auto scaling* dilakukan pada level pod. Diperlukan dua komponen penting agar MQTT broker memiliki kemampuan *auto scaling,* yaitu parameter, dan angka ambang batas. Parameter merupakan suatu entitas sumber daya yang ingin dijadikan acuan perhitungan, sedangkan ambang batas merupakan angka yang menentukan batas penggunaan sumber daya suatu MQTT broker (Kubernetes, 2018). Pada penelitian ini akan digunakan tiga buah parameter sebagai acuan terjadinya *auto scaling*, yaitu penggunaan cpu, penggunaan ram, dan jumlah klien MQTT yang terhubung. Pemilihan penggunaan cpu dan memori sebagai parameter *autoscaling* dikarenakan latar belakang dari penelitian ini yaitu habisnya sumber daya dari MQTT broker. Sedangkan parameter jumlah klien yang terkoneksi dipilih karena jumlah klien yang meminta koneksi mampu memberikan pengurangan sumber daya. Untuk memberikan gambaran lebih mengenai sistem ini diberikan topologi gambar sebagai berikut:



Gambar 3.3 Diagram Alur Data Nilai Parameter Untuk *Auto Scaling*

Diagram alur diatas merupakan diagram alur pengambilan nilai parameter untuk auto scaling. Berikut alurnya:

1. Pod MQTT broker melalui kubelet akan memberikan data mengenai penggunaan cpu, dan memori ke metrics server
2. Pod MQTT broker akan memberikan data mengenai jumlah klien yang terkoneksi ke Prometheus.
3. Prometheus akan memberikan data tersebut ke Prometheus adapter untuk dijadikan nilai sebagai parameter auto scaling
4. Metrics Server akan mengambil dan mengumpulkan nilai cpu dan memory dari kubelet

5. Data yang berada pada prometheus adapter dan metrics server diteruskan ke Kubernetes API Server.
6. Horizontal Pod Autoscaler akan menggunakan data yang sudah berada pada Kubernetes API server dan akan melakukan perhitungan menggunakan algoritma yang ada. Setelah itu Horizontal Pod Autoscaler akan menghubungi bagian deployment untuk kemudian melakukan penskalaan sesuai hasil dari perhitungan yang telah dilakukan.

3.3 Metode Evaluasi

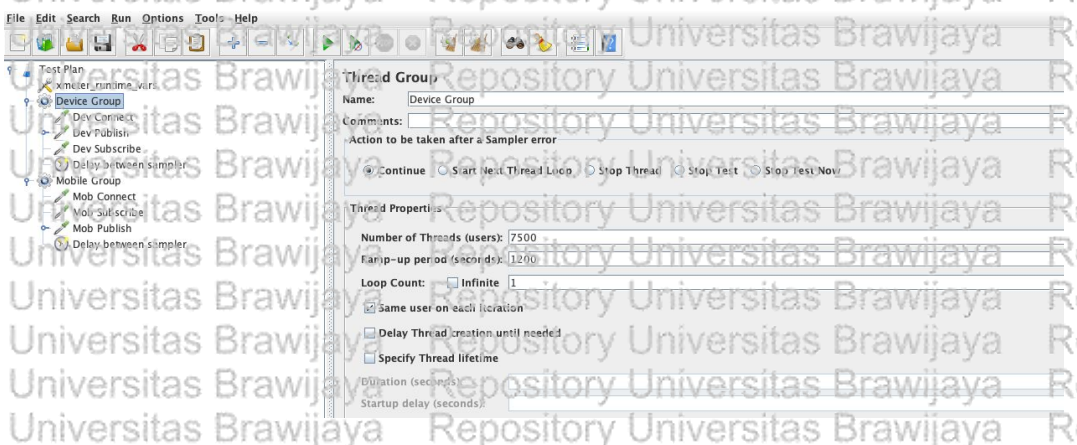
Tahap pengujian dilakukan setelah tahap implementasi selesai dilakukan. Pengujian sistem bertujuan untuk mengetahui apakah arsitektur yang ditawarkan mampu meningkatkan tingkat. Dalam proses uji sistem, memungkinkan ditemukannya kesalahan yang terdapat pada hasil implementasi. Sebagai bahan uji, digunakan jmeter untuk melakukan pengukuran terhadap sistem yang telah dibuat pada penelitian ini.

3.3.1 Skenario Uji

Dalam penelitian ini, peneliti membandingkan MQTT broker yang memiliki mekanisme *auto scaling* dan tidak memiliki mekanisme *auto scaling*. Pada penelitian ini kedua MQTT broker yang memiliki atau tidak memiliki kemampuan *autoscaling* akan berjalan pada pod. Hal ini dikarenakan pada penelitian terkait sudah membuktikan bahwa koneksi terputus lebih banyak pada level virtual mesin dikarenakan lamanya proses boot dari mesin virtual, untuk itu disini hanya membandingkan jika kedua MQTT broker tersebut berada dalam kontainer (Muzammel, 2019). Masing masing dari MQTT broker tersebut memiliki spesifikasi yang sama yaitu 0.1 core cpu dan 100 mebibyte memory. Setiap data yang dikirimkan dan diambil oleh klien MQTT menggunakan QoS level 0. Penggunaan QoS level 0 ini dipilih karena jenis MQTT broker yang dipilih tidak memiliki mekanisme replikasi pesan ke MQTT broker sesamanya. Peneliti mengamati jumlah klien yang terkoneksi ke pod MQTT broker untuk mengidentifikasi jumlah klien maksimal yang dapat terkoneksi pada MQTT broker tanpa *auto scaling*. Hasil dari pengamatan tersebut disimpan dan dijadikan sebagai parameter dilakukannya *auto scaling*. Dalam proses implementasi auto scaling diperlukan suatu angka ambang batas untuk menentukan kapan auto scaling itu dapat terjadi (Kubernetes, 2018). Pada penelitian ini ditetapkan 80% dari sumberdaya yang ada sebagai angka ambang batas terjadinya *auto scaling*.

Pada penelitian ini ditetapkan dua tolok ukur yang menjadi acuan pengaruh kemampuan *auto scaling* yaitu banyaknya klien yang dapat terkoneksi, dan banyaknya koneksi error yang dihasilkan. Untuk menguji macam tolok ukur dari penggunaan sumberdaya MQTT broker pada pengujian ini digunakan bantuan perangkat lunak jmeter untuk melakukan pengujian. Skenario skrip dari jmeter pada pengujian ini menyimulasikan bagaimana suatu pasangan perangkat IoT dan ponsel saling bertukar pesan kedua arah. Pada skrip ini akan digunakan 2 Thread grup. Thread grup yang pertama menunjukkan perangkat IoT, dan yang

kedua merupakan perangkat ponsel. Jumlah dari user untuk setiap thread group ialah 7500, dengan ramp up time 1200 detik (20 menit). Maka untuk setiap detik user yang akan terkoneksi ke MQTT broker ialah 6 user untuk 1 thread grup, yang berarti 12 klien pada 1 skrip jmeter yang dijalankan dalam satu mesin. Pesan yang dikirimkan dari perangkat IoT berupa kalimat "*Message from Device ${__threadNum}.*" dan dari perangkat ponsel berupa kalimat "*Mobile ${__threadNum} send this message*", dengan isi dari variabel threadNum menyimulasikan identitas perangkat IoT dan perangkat Ponsel. Misalnya thread ke 1 merupakan perangkat IoT dengan id 1, dan perangkat mobile dengan id 1. Hanya perangkat IoT dengan id 1, dan perangkat ponsel dengan id 1 yang boleh saling bertukar pesan. Untuk memastikan bahwa terjadi auto scaling, peneliti menambah 2 mesin lagi (total 3 mesin) yang menjalankan skrip jmeter tersebut.



Gambar 3.4 Skrip Jmeter

Skrip ini nantinya akan dijalankan pada mesin virtual sebanyak tiga dengan spesifikasi mesin sebagai berikut:

- 2 vCPU
- 4 GiB memory
- Ubuntu 18.04.3
- Jmeter 5.2.1
- Zone: us-east-2b

Berikut ini merupakan gambaran umum terkait cara kerja skrip jmeter yang diimplementasikan:

Gambar 3.5 Visualisasi Skrip Jmeter

Pada gambar 3.7 dapat dilihat terdapat 1 set perangkat IoT dan Mobile device. Setiap perangkat akan melakukan publish dengan topik N, dimana N merupakan index thread di jmeter. Dengan asumsi tersebut akan terjadi hubungan 1:1 antara IoT device dan mobile secara bidirection. Seperti yang dijelaskan sebelumnya, *user thread* yang diperuntukan pada penelitian ini adalah 7500 thread untuk setiap node jmeter, sehingga akan terbentuk 7500 set IoT Device dan Mobile Phone.

BAB 4 IMPLEMENTASI

4.1 Lingkungan Implementasi

Dalam mengimplementasikan suatu sistem dibutuhkan suatu lingkungan yang mendukung agar sistem tersebut dapat dibuat. Lingkungan implementasi pada subbab ini terbagi menjadi dua macam, yaitu lingkungan perangkat keras dan lingkungan perangkat lunak yang dijelaskan pada dua sub bab dibawah ini.

4.1.1 Lingkungan Perangkat Lunak

Lingkungan perangkat lunak disini menjelaskan perangkat lunak yang dipakai pada penelitian ini. Penjelasan lengkap terkait lingkungan perangkat lunak dapat dilihat pada tabel 4.1.

Tabel 4.1 Lingkungan Perangkat Lunak

No	Perangkat Lunak	Deskripsi
1	AWS EKS	Layanan kubernetes yang disediakan oleh AWS yang digunakan sebagai tempat berjalannya MQTT broker di cloud
2	Metrics Server	Layanan yang memberikan informasi tentang penggunaan CPU dan memory dari setiap pod
3	Mosquitto	Message broker yang menggunakan protokol MQTT dalam penggunaannya.
4	Mosquitto Exporter	Merupakan perangkat lunak pihak ketiga yang akan mengambil informasi sistem dari Mosquitto yang diberikan ke Prometheus
5	Prometheus	Merupakan perangkat lunak pihak ketiga sebagai tempat penyimpanan informasi yang diberikan oleh exporter
6	Prometheus Adapter	Merupakan layanan pihak ketiga yang digunakan untuk mengambil informasi dari prometheus, dan informasi tersebut nantinya dapat digunakan sebagai parameter auto scaling.
7	AWS Load Balancer	Merupakan layanan yang disediakan AWS untuk menyeimbangkan resource pada Pod.
8	AWS cluster auto scaling group	Merupakan layanan yang disediakan oleh AWS untuk mengelompokkan AWS EC2 instance menjadi suatu kelompok yang nantinya dapat ditambah maupun dikurangi sesuai permintaan.

9	Helm Chart	Sebagai package manager dari kubernetes yang berguna membantu proses deployment layanan pihak ketiga ke AWS EKS
10	Docker	Merupakan perangkat lunak yang digunakan untuk membungkus MQTT broker (Mosquitto dan exporternya) menjadi image agar nantinya dapat dijalankan pada AWS EKS
11	Kubectl	Merupakan perangkat lunak pihak ketiga berbasis command-line yang membantu peneliti agar dapat terhubung ke AWS EKS.

4.1.2 Lingkungan Perangkat Keras

Lingkungan perangkat keras merupakan perangkat keras yang dibutuhkan agar penelitian ini dapat berjalan. Penjelasan lengkap terkait lingkungan perangkat keras yang digunakan pada peneliitian ini dapat diiihat pada tabel 4.2.

Tabel 4.2 Lingkungan Perangkat Keras

No	Perangkat	Deskripsi
1	Virtual Server	Tempat menjalankan Pod MQTT broker pada AWS EKS, disini menggunakan tipe t2.small dengan spesifikasi 1vcpu dan 2gb memory, dan tiga buah virtual server dengan 2 vcpu dan 4gb memory untuk melakukan pengujian.

4.2 Implementasi Pembuatan Docker Image

Pembuatan Docker Image ini dilakukan agar MQTT broker dapat berjalan di AWS EKS. Pembuatan docker image ini terdiri dari dua macam, docker image untuk MQTT brokernya, dan docker image untuk MQTT exporternya. Kedua hal tersebut akan dijelaskan pada sub bab dibawah ini.

4.2.1 Pembuatan Docker Image MQTT broker

Pada pembuatan Docker image ini terdiri dari beberapa tahapan, yaitu mendownload distribusi Linux yang mampu menjalankan MQTT broker ini. Setelah itu diperlukan instalasi paket MQTT broker yang pada penelitian ini adalah Mosquitto, dan beberapa dependency untuk exporter. Langkah berikutnya adalah melakukan kompilasi program golang yang berada di link: https://github.com/sapcc/mosquitto-exporter. Program tersebut melakukan beberapa tugas. Tugas yang dilakukan antara lain melakukan subscribe ke mosquitto untuk mendapatkan sistem informasi yang diberikan dari mosquitto, jumlah klien yang terkoneksi, dan lain-lain. Setelah mendapatkan nilai dari mosquitto, program tersebut akan membuat web-server yang nanti nilai-nilainya akan diberikan ke prometheus.

Setelah itu dilakukan pemindahan kode yang sudah dikompilasi menjadi bentuk biner kedalam *docker images*. Langkah selanjutnya adalah memasang

beberapa paket agar kode biner tersebut dapat berjalan. Selanjutnya ialah membuka port 1883 yang merupakan port standar untuk terhubung ke protokol MQTT, dan port 9234 agar prometheus mampu mengambil nilai yang diberikan oleh exporter ini. Setelah beberapa tahapan tersebut dilakukan, diperlukan perintah untuk menjalankan MQTT exporter tersebut. Setelah beberapa tahapan tersebut dilakukan, diperlukan perintah untuk menjalankan MQTT broker tersebut. Dari yang dijelaskan diatas dapat diterjemahkan menjadi kode seperti pada tabel 4.3.

Tabel 4.3 Potongan Kode Program Pembuatan Docker Images

```
1  FROM alpine:3.10
2  LABEL maintainer="kevinchou" appname="Mosquitto Message
3  Broker"
4  RUN apk add --no-cache libc6-compat libstdc++ mosquitto
5  mosquitto-clients
6  COPY mosquitto-exporter /mosquitto-exporter
7  COPY entrypoint.sh /entrypoint.sh
8  EXPOSE 1883 9234
9  ENTRYPOINT [ "/entrypoint.sh" ]
```

Untuk menjalankan mosquitto dan *exporter*nya secara sekaligus akan dibuat suatu file bernama *entrypoint.sh* yang berisi proses tersebut. Sehingga setiap kali container dijalankan akan menjalankan kode tersebut. Isi dari file tersebut dapat dilihat pada tabel 4.4.

Tabel 4.4 Isi dari file *entrypoint.sh*

```
1  #!/bin/ash
2  /usr/sbin/mosquitto &
3  /mosquitto-exporter
```

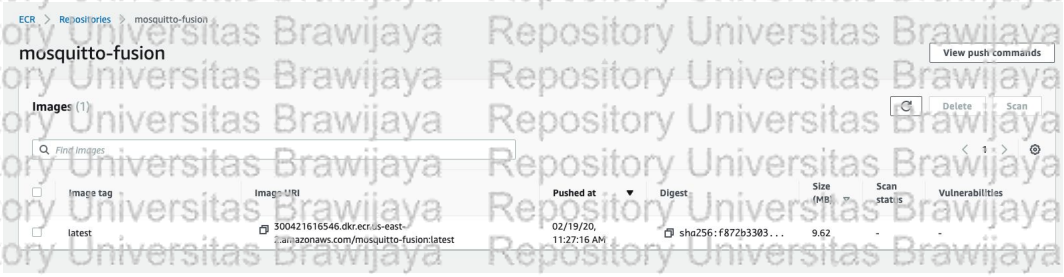
4.2.2 Implementasi Upload Docker Images ke AWS

Pada implementasi pembuatan docker images sebelumnya, MQTT broker dan exporter hanya bisa berjalan pada lokal komputer saja, oleh karena itu diperlukan *repository online* untuk menampung *image* yang telah dibuat. Pada penelitian ini image yang telah dibuat diletakkan pada AWS. Setelah image tersebut terbuat, baru dapat kita upload docker images yang telah dibuat ke *cloud* agar images tersebut dapat dipakai di AWS EKS. Isi dari perintah yang dipakai untuk melakukan upload *docker images* ke cloud dapat dilihat pada tabel 4.5.

Tabel 4.5 Kode Perintah yang Digunakan Untuk Upload Docker Images

```
1  docker build -t mosquitto-fusion .
2  docker tag mosquitto-fusion:latest
3  300421616546.dkr.ecr.us-east-2.amazonaws.com/mosquitto-fusi
   on:latest
4  docker push
5  300421616546.dkr.ecr.us-east-2.amazonaws.com/mosquitto-fusi
   on:latest
6
```

Berikut merupakan tampilan di cloud setelah image yang sudah dibuat berhasil terupload:



Gambar 4.1 Tampilan Image Mosquitto MQTT broker di AWS

4.3 Implementasi Deployment Prometheus

Prometheus pada peneiitian ini berfungsi sebagai metrics collector yang menyimpan nilai dari MQTT exporter. Pada implementasi ini prometheus akan di *deploy* menggunakan helm chart. Perintah yang digunakan untuk melakukan proses *deployment* ke *cloud* sebagai berikut:

```
$ helm install -name prometheus stable/prometheus -namespace monitoring
```

Setelah perintah diatas dilakukan dapat akan terdapat pod Prometheus di AWS EKS, berikut tampilan terkait pod Prometheus yang berhasil di deploy:

Gambar 4.2 Pod Prometheus

Untuk alasan keamanan prometheus tidak diberi service load Balancer sehingga dashboard ui prometheus tidak dapat diakses melewati internet, peneliti menggunakan bantuan port-forwarding dari perintah kubectl untuk mengakses dashboard prometheus. Berikut perintahnya :

```
kubectl --namespace monitoring port-forward svc/prometheus-server 9090:80
```

Perintah tersebut membantu melakukan port-forwarding yang menghubungkan prometheus yang berada pada AWS ke local komputer. Prometheus tersebut berada pada port 80, untuk di forward ke port lokal

komputer yaitu 9090. Setelah perintah tersebut dilakukan prometheus dapat diakses dari local komputer.

Gambar 4.3 Dashboard Prometheus

4.4 Implementasi Deployment Prometheus Adapter

Prometheus adapter merupakan perangkat lunak yang berguna untuk mengambil nilai metrics dari prometheus, agar dapat digunakan sebagai parameter dalam *auto scaling*. Pada implementasi ini digunakan helm-chart untuk membantu deployment ke AWS EKS. Berikut perintahnya:

```
$ helm install -f values.yaml --name prometheus-adapter
--namespace monitoring
```

Pada perintah tersebut terdapat suatu file bernama values.yaml. File tersebut berisi url dari prometheus. Pada mekanismenya prometheus adapter perlu untuk mengetahui dimana lokasi prometheus-server itu berada, sekaligus mendeklarasikan metrics apa yang akan diambil dari prometheus agar dapat dijadikan sebagai parameter *auto scaling*. Pada penelitian ini selain penggunaan CPU dan memori, juga memakai parameter lain yaitu jumlah klien yang terkoneksi pada MQTT broker. Pada file inilah hal tersebut didefinisikan. Definisi lengkap dari isi values.yml dapat dilihat pada tabel

```
1  prometheus:
     url: http://prometheus-server.monitoring.svc.cluster.local
2    port: 80
3  rules:
     default: false
4    custom:
5    - seriesQuery: 'broker_clients_connected'
       resources:
6        overrides:
7          kubernetes_namespace:
             resource: namespace
```

```
      kubernetes_pod_name:
        resource: pod
    name:
      matches: 'broker_clients_connected'
      as: ""
    metricsQuery: "<<.Series>>{<<.LabelMatchers>>}"
```

Tabel 4.6 Isi dari file values.yml pada Prometheus Adapter

Setelah prometehus adapter berhasil dideploy, metrics jumlah klien yang terkoneksi dapat dijadikan sebagai parameter untuk auto scaling. Berikut gambarnya:

Gambar 4.4 Deskripsi Parameter Auto Scaling

Pada gambar diatas terdapat suatu deployment dalam hal ini peneliti sudah melakukan deployment mosquitto di cloud. Dapat dilihat disana terdapat parameter "broker_clients_connected" berjumlah 1 dimana angka satu tersebut merupakan mosquitto-exporter yang bertugas untuk mengekspor data.

4.5 Implementasi Deployment Metrics Server

Metrics server pada penelitian ini berfungsi untuk menyediakan nilai mengenai penggunaan cpu dan memory yang nantinya dapat digunakan sebagai parameter dalam auto scaling. Pada implementasi ini perlu mengunduh terlebih dahulu file-file konfigurasi yang disediakan langsung secara official oleh kubernetes di github nya. Setelah itu memakai bantuan perintah kubectl, metrics server dapat di deploy dengan benar.

```
$ git clone https://github.com/kubernetes-sigs/metrics-server.git
$ kubectl create -f metrics-server/deploy/1.8+/
```

Setelah melakukan deployment ini proses auto scaling dapat menggunakan parameter cpu dan memory suatu pod seperti yang terlihat pada gambar 4.4.

4.6 Implementasi Grafana Dashboard Monitoring System

Grafana dashboard dalam penelitian ini berguna untuk mempermudah peneliti dalam proses pengujian. Grafana mengambil data dari prometheus setelah itu dibuat divisualisasikan didashboard grafananya. Berikut adalah perintah yang digunakan untuk melakukan instalasi grafana di cloud.

```
$ helm install --namespace monitoring stable/grafana
```

Untuk mengakses grafana dashboard peneliti melakukan port-forwarding yang menghubungkan grafana yang berada pada AWS EKS ke local komputer. Grafana tersebut berada pada port 80, berikut perintah yang digunakan:

```
$ kubectl --namespace monitoring port-forward svc/grafana 3000:80
```

Setelah perintah tersebut dijalankan dashboard grafana dapat diakses pada lokal komputer. Berikut gambarnya

Gambar 4.5 Tampilan Awal Dashboard Grafana

Setelah itu peneliti meminta bantuan template dashboard pada grafana dengan id dashboard 315. Beberapa kustomisasi ditambahkan seperti penambahan variabel deployment, sehingga nantinya peneliti mampu melakukan proses monitoring terhadap deployment MQTT broker yang dalam hal ini merupakan mosquitto. Berikut merupakan Dashboard yang sudah di kustomisasi untuk keperluan penelitian ini:

Gambar 4.6 Grafana Dashboard Yang Sudah Di Kostumisasi

Tujuan kostumisasi pada dashboard ini untuk melakukan penambahan monitoring terhadap jumlah klien yang terkoneksi, sehingga pada rentang waktu tertentu dapat dengan mudah menghitung berapa jumlah maksimal klien yang terkoneksi, dan berapa rata-ratanya.

4.7 Implementasi *Deployment* MQTT Broker di AWS EKS

Setelah docker image tadi dibuat, diperlukan untuk membuat file konfigurasi kubernetes untuk melakukan deployment kedalam kubernetes cluster yang berada di AWS. Namun jika membuat file konfigurasi tanpa

melakukan templating peneliti tidak dapat merubah-merubah isi dari file konfigurasi secara dinamis. Oleh karena itu dalam proses deploy MQTT broker di cloud, peneliti memutuskan untuk membuat paket sendiri untuk penelitian ini dengan bantuan Helm Chart. Paket tersebut berisi konfigurasi *horizontal pod autoscaler*, service loadBalancer, dan deployment MQTT broker itu sendiri. Potongan kode file deployment.yaml dapat dilihat pada tabel 4.7 .

Tabel 4.7 Isi dari file deployment.yaml

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: {{ include "mosquitto.fullname" . }}
5    labels:
6  {{ include "mosquitto.labels" . | indent 4 }}
7  spec:
8    replicas: {{ .Values.replicaCount }}
9    selector:
10     matchLabels:
11       app.kubernetes.io/name: {{ include "mosquitto.name" .
12 }}
13       app.kubernetes.io/instance: {{ .Release.Name }}
14   template:
15     metadata:
16       labels:
17         app.kubernetes.io/name: {{ include "mosquitto.name" .
18 }}
19         app.kubernetes.io/instance: {{ .Release.Name }}
20       annotations:
21         prometheus.io/scrape: {{ .Values.prometheus.scrape |
22 quote }}
23         prometheus.io/port: {{ .Values.exporter.port | quote
24 }}
25     spec:
26       containers:
27         - name: {{ .Chart.Name }}
28           image: "{{ .Values.mosquitto.image }}"
29           imagePullPolicy: {{ .Values.image.pullPolicy }}
30           ports:
31             - name: mqtt-port
                containerPort: {{ .Values.mosquitto.port }}
```

```
32              protocol: TCP
33            - name: exporter-port
34              containerPort: {{ .Values.exporter.port }}
35              protocol: TCP
36          resources:
37            {{- toYaml .Values.resources | nindent 12 }}
38          readinessProbe:
39            tcpSocket:
40              port: 1883
41            initialDelaySeconds: 5
42            periodSeconds: 10
43          livenessProbe:
44            tcpSocket:
45              port: 1883
46            initialDelaySeconds: 15
47            periodSeconds: 20
```

Pada file deployment.yaml ini menjelaskan bagaimana suatu MQTT broker di deploy di cloud dan dikonfigurasi. Didalamnya menjelaskan jumlah pod yang dibuat, isi dari pod yang dalam kasus penelitian ini terdapat satu buah container untuk setiap satu pod. Setiap pod tersebut wajib ditentukan berapa jumlah cpu dan memory yang direquest agar *horizontal pod autoscaler* dapat berjalan. Selain itu terdapat juga label-label yang digunakan agar controller lain dapat dengan mudah menemukan deployment MQTT broker yang dimaksud.

Terdapat juga readiness probe yang merupakan fitur dari kubernetes untuk melakukan health check kapan suatu pod siap menerima permintaan diawal waktu pertama kali pod hidup. Selain itu ada juga livenessProbe yang merupakan suatu fitur dari kubernetes dimana fitur ini mampu melakukan pengecekan apabila suatu proses terjadi kegagalan, dan tidak bisa dilakukan recovery terkecuali pod tersebut di restart. Untuk setiap health-check yang pada file konfigurasi ini menggunakan perintah tcpSocket ke port 1883 dimana port tersebut merupakan port MQTT broker.

Kubelet akan melakukan permintaan koneksi socket ke port 1883. Untuk readiness probe, jika koneksi yang dilakukan oleh kubelet itu berhasil maka MQTT broker maka state pada pod kubernetes akan berubah menjadi ready begitu juga sebaliknya, ketika ada kegagalan dalam koneksi state dari kubernetes pod akan menjadi not ready. Untuk liveness probe jika koneksi yang dilakukan oleh kubelet itu berhasil maka MQTT broker dianggap sehat jika tidak maka pod dianggap sedang dalam masalah dan perlu dilakukan restart pod.

Dalam readiness probe dan liveness probe terdapat konfigurasi waktu yaitu initialDelaySecond, dan periodSeconds. initialDelaySecond berarti berapa lama waktu yang dibutuhkan sebelum melakukan health check setelah pod dimulai.

periodSeconds merupakan berapa waktu sekali melakukan health check tersebut. Potongan kode file hpa.yaml dapat dilihat pada tabel 4.8.

Tabel 4.8 Isi dari file hpa.yaml

```
{{- if .Values.hpa.enabled }}
apiVersion: auto scaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: {{ include "mosquitto.fullname" . }}
  labels:
{{ include "mosquitto.labels" . | indent 4 }}
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: {{ include "mosquitto.fullname" . }}
  minReplicas: {{ .Values.replica.min }}
  maxReplicas: {{ .Values.replica.max }}
  metrics:
{{- with .Values.hpa.cpuPercentage }}
  - type: Resource
    resource:
      name: cpu
      targetAverageUtilization: {{ . }}
{{- end }}
{{- with .Values.hpa.memoryPercentage }}
  - type: Resource
    resource:
      name: memory
      targetAverageValue: {{ . | quote }}
{{- end }}
{{- with .Values.hpa.brokerClientsConnected }}
  - type: Pods
    pods:
      metricName: broker_clients_connected
      targetAverageValue: {{ . }}
{{- end }}
{{- end }}
```

Pada file hpa.yaml ini didefinisikan terkait konfigurasi auto scaling pada MQTT broker. Di file ini dapat didefinisikan jumlah pod minimal, maximal dalam melayani permintaan klien. Nilai dari berapa jumlah pod minimal,dan maksimalnya dapat diterapkan di file values.yaml.

Selain itu juga didefinisikan parameter apa saja yang digunakan agar auto scaling dapat terjadi. Pada file hpa.yaml ini terdapat tiga buah parameter, yaitu rata-rata penggunaan cpu, memori dan jumlah klien yang terkoneksi dalam semua pod. Nilai dari berapa rata-rata penggunaan cpu, memori dan jumlah klien dapat diterapkan di file values.yaml.

Pada dasarnya klien MQTT dari luar *kubernetes* tidak akan dapat melakukan permintaan koneksi ke MQTT broker tanpa adanya sumber daya *service* yang menerima permintaan koneksinya. Untuk itu perlu dibuat *service* dengan jenis *Load Balancer*, agar mampu mendistribusikan permintaan koneksi dari klien ke pod MQTT broker. Isi dari file service.yaml dapat dilihat pada tabel 4.9 .

Tabel 4.9 Isi dari file service.yaml

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: {{ include "mosquitto.fullname" . }}
5    labels:
6  {{ include "mosquitto.labels" . | indent 4 }}
7  spec:
8    type: {{ .Values.service.type }}
9    ports:
10     - port: {{ .Values.service.port }}
11       targetPort: 1883
12       protocol: TCP
13   selector:
14     app.kubernetes.io/name: {{ include "mosquitto.name" . }}
15     app.kubernetes.io/instance: {{ .Release.Name }}
```

Pada file service.yaml ini bertujuan agar klien dapat terkoneksi ke MQTT broker. Disini didefinisikan berapa port MQTT broker dan jenis protokolnya. Dalam hal ini jenis protokol pada MQTT adalah TCP. Port dalam file service.yaml ini memiliki makna port dari sisi klien, dan dapat ditetapkan di file values.yaml secara dinamis. Target port memiliki makna berapa port MQTT yang berada pada pod yang dalam hal ini adalah 1883.

Semua konfigurasi file yang telah dibuat untuk melakukan *deployment* MQTT broker hanya sebatas *template*, yang berarti file konfigurasi tersebut harus diisi dengan nilai yang sesuai. Oleh sebab itu dibutuhkan satu file

tambahan untuk mengisi *template* konfigurasi yang bernama values.yaml. Nama dari file ini bersifat statis tidak dapat diubah karena file ini nanti akan digunakan sebagai nilai default pada waktu pemaketan dengan *helm-chart*. Isi dari values.yaml dapat dilihat pada tabel 4.10.

Tabel 4.10 Isi dari file values.yml

```
# Default values for mosquitto.
# This is a YAML-formatted file.
# Declare variables to be passed into your templates.

replicaCount: 1

image:
  pullPolicy: IfNotPresent
mosquitto:
  image:
300421616546.dkr.ecr.us-east-2.amazonaws.com/mosquitto-fusion
  port: 1883

prometheus:
  scrape: true
exporter:
  port: 9234
service:
  type: LoadBalancer
  port: 1883
  annotations: {}
replica:
  min: 1
  max: 10
resources:
  requests:
    cpu: 100m
    memory: 100Mi
  limits:
    cpu: 100m
    memory: 100Mi

hpa:
  enabled: true
  cpuPercentage: 80
  memoryAverage: 80Mi
  brokerClientsConnected: 8000
```

Pada isi dari file values.yaml diatas menunjukkan bahwa secara default peneliti dapat melakukan deployment sebuah MQTT broker di cloud, dengan spesifikasi cpu 0.1 dan memory hanya 100 Mebibyte. Dengan bantuan AWS

peneliti dapat dengan mudah meminta *load balancer* yang memiliki IP public untuk melakukan *load balancer* ke setiap pod MQTT broker. Setiap Pod MQTT broker yang berada di cloud akan diberi label sehingga dengan membaca label tersebut *load balancer* mampu meneruskan traffic ke setiap pod MQTT broker.

Selain itu pada file values.yaml ini didefinisikan juga terkait anotasi prometheus, agar prometheus dapat mengambil data di exporter yang dalam hal ini berada pada port 9234. Dengan anotasi port saja belum cukup untuk memberi tahu prometheus mengambil data. Oleh sebab itu perlu juga diberikan definisi anotasi boolean, dalam hal ini kata kunci true, berarti akan memberi tahu prometheus untuk mengambil data.

Disini juga didefinisikan jumlah pod minimal dan maksimal MQTT broker. Jumlah minimal pod untuk melayani klien sebanyak satu pod. Jumlah maksimal untuk menangani pertumbuhan permintaan klien sebanyak 10 pod. Sehingga nantinya akan ada alokasi minimal 0.1 core untuk satu pod, dan maksimal 1 core untuk 10 pod.

Setiap pod berisi satu buah kontainer MQTT broker beserta exporternya. Kontainer tersebut memakai image yang sudah diupload sebelumnya ke AWS ECR. Dengan adanya file values.yaml deployment dari suatu MQTT broker dapat dieksekusi secara dinamis, berdasarkan isi dari file values.yaml.

Setelah penjelasan terkait isi dari konfigurasi kubernetes, selanjutnya adalah melakukan packaging keseluruhan file.yaml tadi menjadi suatu kesatuan berbentuk paket.

```
$ helm package mosquitto/
```

Untuk menampung paket tadi digunakan repository git yang berada di github sehingga dapat diakses dari manapun. Setelah repository tersebut dibuat dilakukan perintah dibawah ini untuk melakukan proses pencatatan paket-paket yang tersedia di repository github.

```
$ helm repo index githubRepo/
$ git push origin gh-pages
```

Setelah itu daftar paket yang berada pada repository dapat dilihat seperti gambar dibawah ini

Repository Universitas Brawijaya

```
apiVersion: v1
entries:
  mosquitto:
  - apiVersion: v1
    appVersion: "3.1"
    created: "2020-02-19T11:28:56.806558+07:00"
    description: A Helm chart for Kubernetes
    digest: 6aaa0e8327d9e79f4e9857089a36e66307e72d2d2da39eea1070005befd68dbe
    name: mosquitto
    urls:
    - mosquitto-1.0.1.tgz
    version: 1.0.1
  - apiVersion: v1
    appVersion: "3.1"
    created: "2020-02-19T11:28:56.806095+07:00"
    description: A Helm chart for Kubernetes
    digest: f7c76700b1286414861de453133b434ffe12429932a36664a19ec204b12fad66
    name: mosquitto
    urls:
    - mosquitto-1.0.0.tgz
    version: 1.0.0
  - apiVersion: v1
    appVersion: "1.9"
    created: "2020-02-19T11:28:56.805127+07:00"
    description: A Helm chart for Kubernetes
    digest: f82854a65757670b5ff98f134e10a350b80dda9fc4a36c487a230d605d984085
    name: mosquitto
    urls:
    - mosquitto-0.1.9.tgz
    version: 0.1.9
```

Gambar 4.7 Daftar Paket Helm Chart di Repository

Dari gambar 4.7 diatas dapat dilihat terdapat berbagai versi yang telah dibuat dalam pengembangan paket MQTT broker ini khususnya Mosquitto. Dalam repository ini peneliti dapat melakukan deployment spesifik ke versi tertentu. Akan tetapi dalam proses pengembangannya versi terakhirlah yang paling cocok dalam penelitian ini.

Untuk melakukan proses deployment ke aws-eks cukup menggunakan perintah dibawah ini. Dalam perintah dibawah ini akan didefiniskan nama deploymentnya, lalu repository mana yang digunakan untuk melakukan deployment dalam hal ini digunakan repository peneliti tempat menampung paket mosquitto seperti yang sudah dijelaskan sebelumnya.

```
$ helm install --name <namadeployment> githubRepo/mosquitto
```

Setelah perintah diatas dilakukan MQTT broker sudah dapat digunakan untuk melayani klien. Berikut keluaran dari perintah diatas

Gambar 4.8 Proses *Deploy MQTT broker menggunakan helm-chart*

BAB 5 PENGUJIAN DAN PEMBAHASAN

Setelah melakukan implementasi, akan dilakukan pengujian pada sistem yang dikembangkan. Tujuan pengujian adalah untuk menguji sistem yang dibuat sesuai dengan kebutuhan atau tidak. Pengujian yang dilakukan adalah pengujian fungsionalitas dan pengujian kinerja sistem terkait dampak dari mekanisme *auto scaling* yang diterapkan pada MQTT Broker dalam hal ini adalah Mosquitto.

5.1 Pengukuran Metrik

Dalam pengujian yang dilakukan terdapat dua buah metrik yang diukur dan diawasi pengaruhnya terhadap MQTT broker Mosquitto. Metrik yang pertama dilakukan pengukuran ialah jumlah klien yang terputus koneksinya. Jumlah klien ini mampu merepresentasikan berapa banyak *error* yang terjadi pada saat suatu skenario pengujian dieksekusi.

Metrik yang kedua adalah jumlah maksimal klien yang dapat terkoneksi kedalam suatu MQTT broker Mosquitto dengan skenario pengujian yang telah ditetapkan. Hal ini akan merujuk pada apakah ada peningkatan jumlah klien yang terkoneksi dengan diterapkannya mekanisme *auto scaling* pada MQTT broker. Selain itu, metrik ini mampu memperlihatkan bagaimana penggunaan sumber daya pada MQTT broker seperti penggunaan cpu dan memori.

5.2 Skenario Uji

Seperti yang dijelaskan pada perancangan pengujian, pengujian ini dilakukan dengan mensimulasikan hubungan bidirection antara perangkat IoT dan perangkat *mobile*. Setiap perangkat akan melakukan *publish* dan melakukan *subscribe* ke tepat satu perangkat lainnya. Dalam pengujian ini dibuat 7500 pasang perangkat IoT dan perangkat *mobile* yang dijalankan pada 3 mesin berbeda. Pada penelitian ini ditetapkan jenis QoS pengujiannya adalah QoS 0. Pada Mosquitto peneliti tidak melakukan perubahan konfigurasi apapun, dan setiap konfigurasi yang digunakan pada pengujian ini sama persis tidak ada perbedaan.

Terdapat dua jenis variasi pengujian disini yaitu pengujian MQTT broker dengan mekanisme *auto scaling* dan tanpa mekanisme *auto scaling*. Pada skenario ini akan diamati perbedaan diantara keduanya baik itu peningkatan dari jumlah klien, dan penurunan klien yang terputus. Skenario ini dilakukan tiga kali dan dihitung rata-ratanya untuk setiap variasi pengujian.

5.3 Lingkungan Pengujian

Seperti yang dibahas pada perancangan pengujian, pengujian dilakukan menggunakan jmeter dan dieksekusi dengan spesifikasi mesin sebagai berikut.

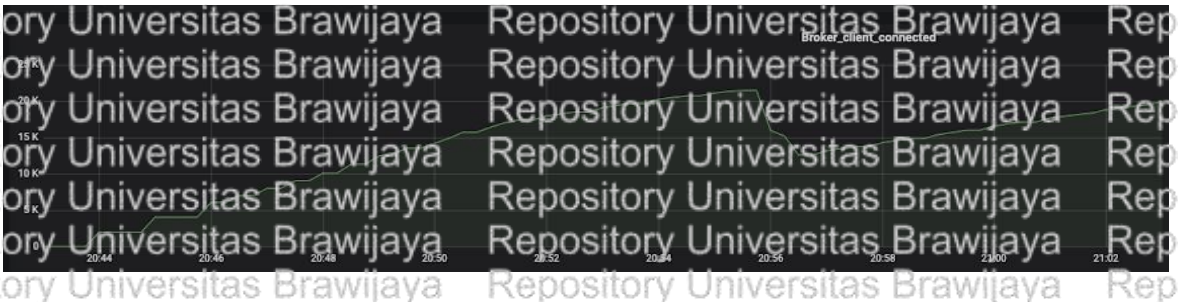
- 2 vCPU
- 4 GiB memory
- Ubuntu 18.04.3

- Jmeter 5.2.1
- Zone: us-east-2b

Setiap satu MQTT broker yang berhasil dibuat akan memiliki kapasitas sumber daya yaitu 0.1 core cpu dan 100 Mebibyte memori.

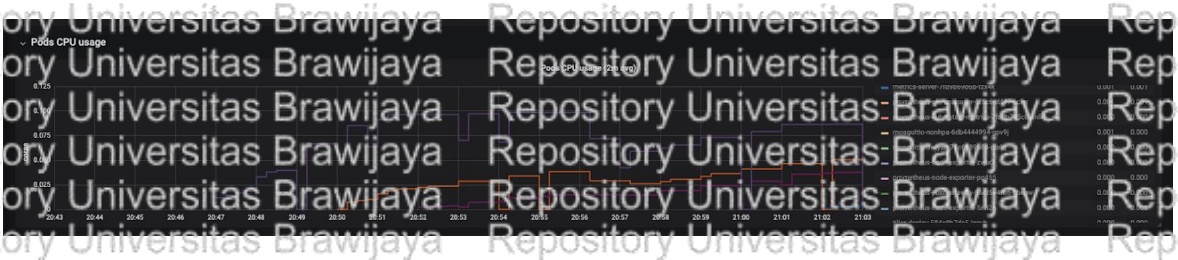
5.4 Pengumpulan Hasil

Pada pengujian ini peneliti menggunakan bantuan dari dashboard grafana untuk mengumpulkan hasil dari pengujian yang dilakukan, berikut dashboard grafana dashboard yang digunakan:



Gambar 5.1 Potongan Gambar Dashboard Grafana dengan panel Jumlah Klien yang Terkoneksi

Dari gambar 5.1 dapat dilihat jumlah maksimal klien yang terkoneksi ke MQTT broker, dan berapa klien yang terputus koneksinya pada saat skenario pengujian dieksekusi. Selain itu pada dashboard grafana juga dapat dilihat penggunaan cpu dan memori dari masing-masing Pod MQTT broker Mosquitto, baik itu yang memiliki mekanisme auto scaling dan tanpa auto scaling. Berikut gambarnya



Gambar 5.2 Potongan Gambar Dashboard Grafana dengan Panel Penggunaan CPU setiap Pod

Gambar 5.3 Potongan Gambar Dashboard Grafana dengan Panel Penggunaan Memori setiap Pod

Setelah dilakukan implementasi terkait mekanisme *auto scaling* MQTT broker, diperlukan validasi untuk memastikan bahwa auto scaling dapat terjadi. Pada file hpa.yaml di bagian implementasi terlihat *auto scaling* pada MQTT broker ini menggunakan tiga buah parameter yaitu penggunaan cpu, memori, dan jumlah klien yang terkoneksi. Berikut merupakan *screenshot log* ketika mekanisme *auto scaling* terjadi:

Gambar 5.4 Proses *scale up* MQTT broker

Gambar 5.5 Proses *scale down* MQTT broker

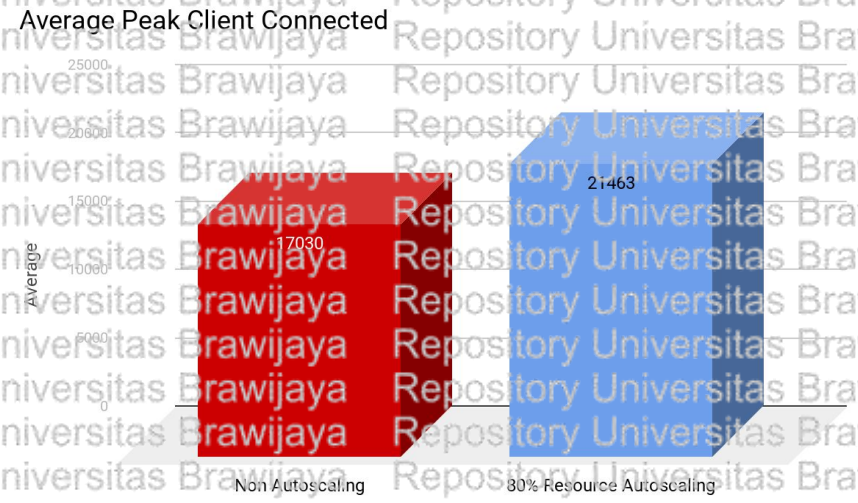
Pada gambar 5.4 merupakan proses bertambahnya Pod MQTT broker. Jumlah pod maksimal yang terbentuk pada satu skenario pengujian adalah tiga buah pod. Pada gambar 5.5 merupakan proses berkurangnya jumlah Pod MQTT broker. Jumlah pod ketika semua penggunaan sumber daya berada dibawah angka ambang batas akan berkurang menjadi satu. Dari kedua gambar diatas

dapat disimpulkan bahwa mekanisme auto scaling pada MQTT broker dapat dilakukan.

Pada gambar 5.1 sampai 5.3 dapat dilihat ketika terjadi peningkatan permintaan koneksi melalui *load balancer* yang disediakan oleh AWS, penggunaan sumber daya meningkat, dan dilakukan perhitungan rata-rata oleh kubernetes. Pada saat penggunaan sumber daya melebihi angka ambang batas yang telah ditetapkan, maka akan dilakukan penambahan pod MQTT broker dan permintaan koneksi akan didistribusikan juga ke pod yang baru sehingga peningkatan permintaan koneksi dari MQTT broker yang lama menjadi terbagi ke kontainer MQTT broker yang baru. Begitu juga ketika jumlah perangkat-perangkat yang terhubung tersebut berkurang, kubernetes menghitung rata-rata pada setiap parameter dan melakukan pengurangan kontainer MQTT broker.

5.5 Hasil Pengukuran

Pada hasil pengujian yang dilaksanakan dalam dua puluh menit, didapatkan hasil jumlah klien maksimal yang berhasil diraih oleh MQTT broker tanpa auto scaling, dan auto scaling sebagai berikut:

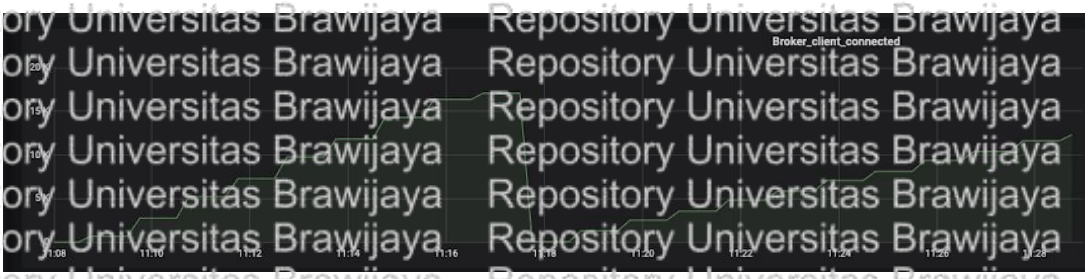


Grafik 5.1 Rata-rata jumlah klien yang dapat dilayani

Pada grafik 5.1 diatas terlihat rata-rata jumlah maksimal dari MQTT broker tanpa auto scaling, dan MQTT broker dengan mekanisme auto scaling yang didapatkan dari tiga kali pengujian. Dengan asumsi bahwa tanpa auto scaling merupakan data awal yang dimiliki, maka setelah dihitung peningkatan yang diberikan oleh MQTT broker dengan kemampuan *auto scaling* sebanyak 26.03%.

Selanjutnya ialah terkait berapa banyak koneksi yang terputus pada saat pengujian ini berlangsung. Setelah dilakukan tiga kali pengujian dengan membandingkan jumlah koneksi klien yang terputus, MQTT broker tanpa

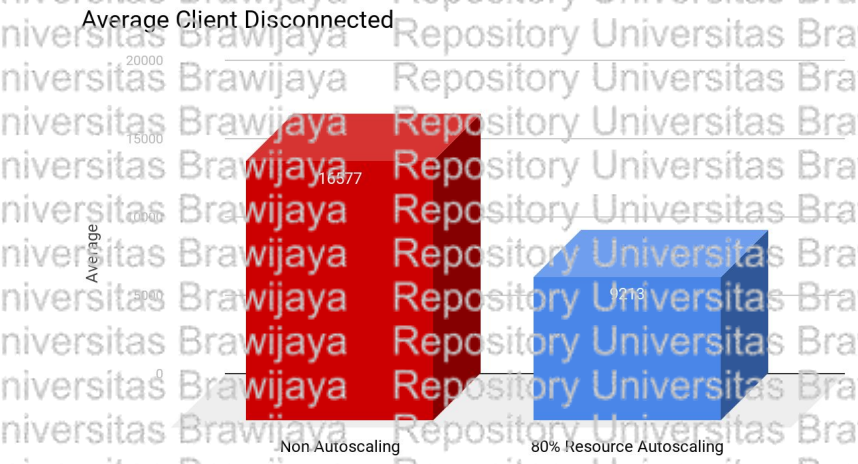
mekanisme *auto scaling* memiliki jumlah lebih banyak daripada MQTT broker dengan kemampuan *auto scaling*.



Gambar 5.6 Pergerakan jumlah koneksi klien yang terhubung pada MQTT broker tanpa kemampuan *auto scaling*

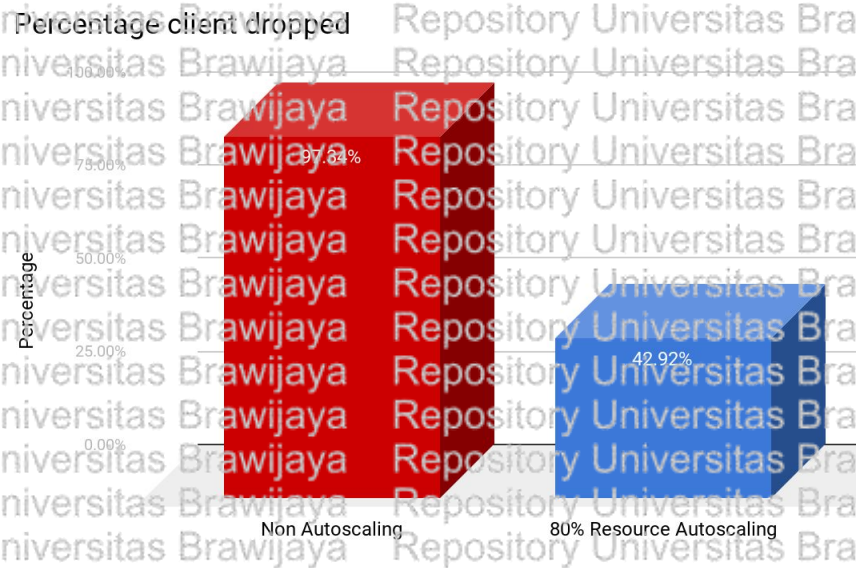
Gambar 5.7 Pergerakan jumlah koneksi klien yang terhubung pada MQTT broker dengan kemampuan *auto scaling*

Gambar 5.4 dan 5.5 menunjukan bagaimana terjadinya penurunan jumlah koneksi yang terhubung ke pod MQTT broker selama masa pengujian. Pada MQTT broker tanpa kemampuan *auto scaling*, jumlah koneksi yang terhubung menurun drastis, sedangkan MQTT broker dengan kemampuan *auto scaling* jumlah koneksi yang terhubung menurun secara perlahan dan tidak sebanyak MQTT broker tanpa kemampuan *auto scaling*. MQTT broker dengan kemampuan *auto* scaling masih menyebabkan koneksi yang terputus. Hal ini dikarenakan ketika dilakukan penambahan pod MQTT broker, Pod yang lama masih berada pada ambang batas penggunaan sumber daya seperti yang dapat dilihat pada gambar 5.2 dan 5.3. Dari hasil pengujian yang dilakukan tiga kali, dihitung rata-ratanya dan didapatkan hasil seperti grafik 5.2 .



Grafik 5.2 Rata-rata jumlah koneksi klien terputus

Setelah dilakukan rata-rata dari tiga kali pengujian yang dilakukan pada grafik di atas dapat dilihat MQTT broker yang memiliki kemampuan auto scaling memiliki jumlah koneksi klien yang terputus lebih rendah. Pada MQTT broker tanpa kemampuan *auto scaling,* sebanyak 16577 koneksi terputus dari jumlah rata-rata maksimal klien yang terhubung yaitu 17030. Pada MQTT broker dengan kemampuan *auto scaling* didapatkan sebanyak 9213 koneksi klien terputus dari jumlah rata-rata koneksi maksimal yang dapat terhubung 21463. Untuk memberikan perbandingan berapa persentase koneksi terputus dari MQTT broker tanpa kemampuan *auto scaling,* dan MQTT broker dengan kemampuan *autoscaling* di buatlah grafik 5.4 dibawah ini.



Grafik 5.3 Persentase koneksi klien yang terputus setelah di rata-rata

Pada grafik diatas dapat dilihat jumlah klien yang terputus menurun signifikan setelah diterapkan mekanisme auto scaling. Dari awal koneksi klien yang terputus sebanyak 97.34% menjadi 42.92% untuk MQTT broker dengan

mekanisme *auto scaling*. Hal ini membuat MQTT broker dengan mekanisme auto scaling lebih baik 54.42% dari MQTT broker tanpa mekanisme *auto scaling*.

BAB 6 PENUTUP

6.1 Kesimpulan

Berdasarkan perancangan sistem, implementasi dan pengujian yang telah dilakukan, dapat ditarik kesimpulan yang menjawab rumusan masalah sebagai berikut:

1. Arsitektur MQTT broker diimplementasikan dengan menerapkan kontainerisasi dan dilakukan orkestrasi menggunakan Kubernetes yang berada pada *Amazon Web Service*. Dengan bantuan platform orkestrasi kubernetes yang berada pada layanan cloud Amazon Web Service, MQTT broker dapat memiliki kemampuan *auto scaling*. Pada MQTT broker diambil nilai penggunaan CPU dan memori oleh agent kubelet, sedangkan nilai jumlah klien yang terkoneksi diambil oleh prometheus. Setelah itu agar nilai dapat dikirim ke *api-server* kubernetes dibutuhkan bantuan perangkat lunak pihak ketiga yaitu prometheus adapter, dan metrics server. Setelah nilai tadi ditampung, *controller Horizontal Pod Autoscaler* akan menghitung berapa jumlah MQTT broker yang harus disediakan dan meminta *controller deployment* untuk menambahkan atau mengurangi jumlah MQTT broker.
2. Berdasarkan hasil yang didapat pada pengujian didapati bahwa ketika terjadi peningkatan permintaan koneksi melalui *load balancer* yang disediakan oleh AWS, penggunaan sumber daya akan meningkat, dan dilakukan perhitungan rata-rata oleh kubernetes. Pada saat penggunaan sumber daya melebihi angka ambang batas yang telah ditetapkan, maka akan dilakukan penambahan kontainer MQTT broker dan permintaan koneksi akan didistribusikan juga ke kontainer yang baru sehingga peningkatan permintaan koneksi dari MQTT broker yang lama menjadi terbagi ke kontainer MQTT broker yang baru. Begitu juga ketika jumlah perangkat-perangkat yang ternubung tersebut berkurang, kubernetes menghitung rata-rata pada setiap parameter dan melakukan pengurangan kontainer MQTT broker.
3. Pada pengukuran kinerja MQTT broker ditinjau dari parameter ukur jumlah klien terkoneksi, dan jumlah koneksi yang terputus akibat sumber daya yang terus berkurang menunjukkan bahwa mekanisme *auto scaling* pada MQTT broker mampu memberikan peningkatan dari sisi jumlah koneksi yang dapat diterima dan terputusnya koneksi akibat sumber daya yang terus berkurang.
    a. Pada MQTT broker dengan mekanisme auto scaling jumlah maksimum koneksi yang dapat ditampung adalah 21463 koneksi sedangkan pada MQTT broker tanpa mekanisme auto scaling terdapat 17030 koneksi yang dapat ditampung. Hal ini membuktikan MQTT broker dengan mekanisme auto scaling memberikan peningkatan sebesar 26.03%
    b. Pada MQTT broker dengan kemampuan *auto scaling* didapati bahwa jumlah klien yang terputus koneksinya pada saat pengujian

dilakukan adalah 9213 koneksi yang berarti sebanyak 42.92% dari jumlah koneksi maksimum koneksinya terputus. Pada MQTT broker tanpa mekanisme auto scaling terdapat 16577 koneksi yang terputus yang berarti sebanyak 97.34% dari jumlah koneksi maksimal yang dapat ditampung koneksinya terputus. Hal ini membuktikan dari sisi jumlah klien yang terputus MQTT broker dengan mekanisme auto scaling memberikan peningkatan 54.42% lebih baik daripada mekanisme MQTT broker tanpa kemampuan autoscaling.

6.2 Saran

Berdasarkan hasil dari pengujian yang telah dilakukan dan kesimpulan yang didapat, berikut saran yang dapat dilakukan pada penelitian berikutnya:

1. Pada penelitian ini hanya tiga buah parameter yang diukur untuk dijadikan sebagai parameter terjadinya *auto scaling*. Masih banyak parameter yang selain yang berada pada penelitian ini misalnya jumlah pesan setiap detik, banyaknya isi dari pesan yang dikirim.
2. Pada penelitian ini setiap parameter yang ditetapkan berbasis pada sumber daya MQTT broker dan tidak memperhatikan aspek data yang berlalu lintas pada MQTT broker sehingga level QoS yang ditetapkan pada penelitian ini menggunakan level 0. Perlu penelitian lebih lanjut terkait aspek data yang berlalu lintas pada MQTT broker.
3. Pada penelitian ini platform untuk melakukan *auto scaling* merupakan kubernetes. Perhitungan dan algoritma yang digunakan untuk menentukan jumlah pod menggunakan algoritma yang tersedia di kubernetes. Perlu dilakukan penelitian lebih lanjut mengenai algoritma yang khusus diperuntukkan MQTT broker untuk memperkecil jumlah koneksi yang terputus ketika dilakukannya *auto scaling*.

DAFTAR REFERENSI

Akbar, S. R., Amron, K., Mulya, H., & Hanifah, S. (2018). Message queue telemetry transport protocols implementation for wireless sensor networks communication - A performance review. *Proceedings - 2017 International Conference on Sustainable Information Engineering and Technology, SIET 2017*, *2018-January*, 107–112. https://doi.org/10.1109/SIET.2017.8304118

AWS. (2018). Apa itu Docker? | AWS. Retrieved April 20, 2019, from https://aws.amazon.com/id/docker/

AWS. (2019). Amazon EKS - Managed Kubernetes Service. Retrieved August 6, 2019, from https://aws.amazon.com/eks/

Bernstein, D. (2014). Containers and cloud: From LXC to docker to kubernetes. *IEEE Cloud Computing*, *1*(3), 81–84. https://doi.org/10.1109/MCC.2014.51

Baier, S. (2020). *Towards Zero Maintenance Operation of an MQTT Broker with a Kubernetes Operator*.

Botta, A., De Donato, W., Persico, V., & Pescape, A. (2014). On the integration of cloud computing and internet of things. *Proceedings - 2014 International Conference on Future Internet of Things and Cloud, FiCloud 2014*, 23–30. https://doi.org/10.1109/FiCloud.2014.14

Burhan, M., Rehman, R. A., Khan, B., & Kim, B. S. (2018). IoT elements, layered architectures and security issues: A comprehensive survey. *Sensors (Switzerland)*, *18*(9). https://doi.org/10.3390/s18092796

Cao, M. (2019). *Implementation and Performance Analysis of Replicated Load-balanced Services Pattern in Distributed Systems*. (April).

Cloud Native Computing Foundation. (2019). Helm Docs | Helm. Retrieved July 22, 2019, from https://helm.sh/

Docker. (2015). About Docker Engine | Docker Documentation. Retrieved December 5, 2019, from https://docs.docker.com/engine/

Dua, R., Raja, A. R., & Kakadia, D. (2014). Virtualization vs containerization to support PaaS. *Proceedings - 2014 IEEE International Conference on Cloud Engineering, IC2E 2014*, 610–614. https://doi.org/10.1109/IC2E.2014.41

Eclipse. (2018). Eclipse Mosquitto. Retrieved August 21, 2019, from https://mosquitto.org/

Fernandez, H., Pierre, G., & Kielmann, T. (2014). auto scaling web applications in heterogeneous cloud infrastructures. *Proceedings - 2014 IEEE International Conference on Cloud Engineering, IC2E 2014*, 195–204. https://doi.org/10.1109/IC2E.2014.25

Grgić, K., Špeh, I., & Hedi, I. (2016). A web-based IoT solution for monitoring data using MQTT protocol. *Proceedings of 2016 International Conference on Smart Systems and Technologies, SST 2016*, 249–253. https://doi.org/10.1109/SST.2016.7765668

Gupta, A., Christie, R., & Manjula, R. (2017). Scalability in Internet of Things: Features, Techniques and Research Challenges. *International Journal of Computational Intelligence Research*, *13*(7), 1617–1627. Retrieved from http://www.ripublication.com

Helm. (2019a). Helm | Docs. Retrieved May 13, 2020, from https://v2.helm.sh/docs/

Helm. (2019b). prometheus-adapter v0.6.0 for Kubernetes | Helm Hub | Monocular. Retrieved May 13, 2020, from https://hub.helm.sh/charts/stable/prometheus-adapter

Hou, L., Zhao, S., Xiong, X., Zheng, K., PChatzimisios, Eriklis, ... Xiang, W. (2016). Internet ofThings Cloud: Architecture and Implementation. *Time*, *5*(5), 13783–13783.

Jutadhamakorn, P., Pillavas, T., Visoottiviseth, V., Takano, R., Haga, J., & Kobayashi, D. (2018). A scalable and low-cost MQTT broker clustering system. *Proceeding of 2017 2nd International Conference on Information Technology, INCIT 2017*, *2018-Janua*, 1–5. https://doi.org/10.1109/INCIT.2017.8257870

Kubernetes. (2014). Production-Grade Container Orchestration - Kubernetes. Retrieved December 5, 2019, from https://kubernetes.io/

Kubernetes. (2017). Pods - Kubernetes. Retrieved September 24, 2019, from https://kubernetes.io/docs/concepts/workloads/pods/pod/

Kubernetes. (2018). Cluster Management - Kubernetes. Retrieved November 18, 2019, from https://kubernetes.io/docs/tasks/administer-cluster/cluster-management/

Muzzammel, R. (2019). *Containers vs Virtual Machines for Auto-scaling Multi-tier Applications Under Dynamically increasing Workloads* (Vol. 932). https://doi.org/10.1007/978-981-13-6052-7

Prometheus. (2014). Overview | Prometheus. Retrieved August 6, 2019, from https://prometheus.io/docs/introduction/overview/

Rao, B. B. P., Saluia, P., Sharma, N., Mittal, A., & Sharma, S. V. (2012). Cloud computing for Internet of Things & sensing based applications. Proceedings of the International Conference on Sensing Technology, ICST, (April 2015), 374–380. https://doi.org/10.1109/ICSensT.2012.6461705

Roy, N., Dubey, A., & Gokhale, A. (2011). Efficient auto scaling in the cloud using predictive models for workload forecasting. *Proceedings - 2011 IEEE 4th International Conference on Cloud Computing, CLOUD 2011*, 500–507. https://doi.org/10.1109/CLOUD.2011.42

Sakurama, K., Verriest, E. I., & Egerstedt, M. (2018). Scalable Stability and Time-Scale Separation of Networked, Cascaded Systems. *IEEE Transactions on Control of Network Systems*, *5*(1), 321–332. https://doi.org/10.1109/TCNS.2016.2609146

Simpson, J. T., & Us, T. N. (2019). *(12) United States Patent*. *2*(12).

The HiveMQ Team. (2019). Client, Broker / Server and Connection Establishment - MQTT Essentials: Part 3. Retrieved December 1, 2019, from https://www.hivemq.com/blog/mqtt-essentials-part-3-client-broker-connection-establishment/

Repository Universitas Brawijaya
REPOSITORY.UB.AC.ID
UNIVERSITAS BRAWIJAYA