Hak cipta dan penggunaan kembali:

Lisensi ini mengizinkan setiap orang untuk menggubah, memperbaiki, dan membuat ciptaan turunan bukan untuk kepentingan komersial, selama anda mencantumkan nama penulis dan melisensikan ciptaan turunan dengan syarat yang serupa dengan ciptaan asli.

Copyright and reuse:

This license lets you remix, tweak, and build upon work non-commercially, as long as you credit the origin creator and license it on your new creations under the identical terms.

Perpustakaan Univeristas Multimedia Nusantara

BAB II

TINJAUAN PUSTAKA

Bagian ini akan mencantumkan beberapa referensi penelitian yang menjadi dasar dan pertimbangan untuk berbagai aspek dalam penelitian ini. Inti dari setiap penelitian akan dijelaskan terlebih dahulu, kemudian disimpulkan pada akhir bagian ini. Selain membahas referensi penelitian, perangkat lunak, perangkat keras, serta protokol yang digunakan juga akan dibahas untuk menjadi pengantar ke bagian selanjutnya.

2.1. Penelitian Terkait

Terdapat beberapa penelitian yang menjadi dasar atas pemilihan arsitektur, rancangan sistem, serta pemilihan protokol yang akan digunakan dalam penelitian ini. Penelitian-penelitian yang menjadi referensi tersebut antara lain:

2.1.1. Firmware Update Over The Air (FOTA) for Automotive Industry [6]

Penelitian dengan judul "*Firmware Update Over The Air (FOTA) for Automotive Industry*" ini dilakukan oleh Moshe Shavit, Andy Gryc, dan Radovan Miucic. Penelitian ini membahas rancangan umum dan implementasi dari *firmware update over-the-air* dan membandingkannya dengan solusi dengan menggunakan media kabel dalam konteks *embedded systems* pada kendaraan. Dari penelitian ini, dapat dilihat proses-proses yang terjadi dalam sebuah *firmware update* serta *use case* perbandingan dari *firmware update* yang dilakukan menggunakan media kabel serta yang dilakukan *over-the-air*. Penelitian ini menerangkan beberapa poin yang cukup penting antara lain:

1) Meskipun lebih sulit dan tidak efisien, *firmware update* melalui media kabel memiliki keuntungan yang dapat dipertimbangkan untuk *use case* tertentu.
2) Melihat *trend* dari kebutuhan industri sekarang, kemudahan serta efisiensi yang ditawarkan oleh *over-the-air update* membuatnya menjadi pilihan yang lebih baik dibandingkan dengan menggunakan kabel.
3) Tantangan yang dihadapi oleh *over-the-air updates* umumnya sangat terkait dengan masalah-masalah dalam bidang *wireless communications*, seperti efisiensi daya, *bandwidth* yang terbatas, *reliability*, *tradeoff*, dan lain-lain.

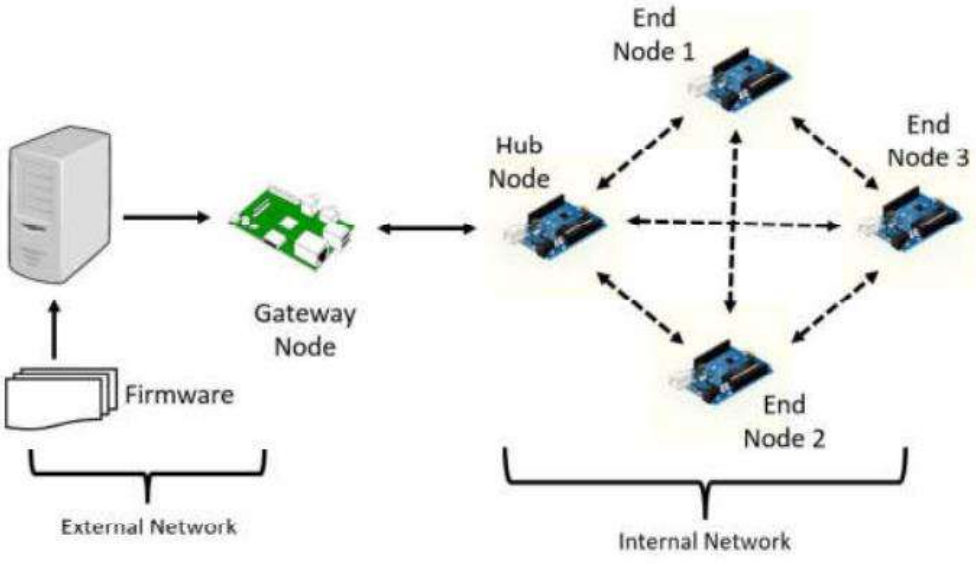
2.1.2. Wireless Multicasting for Remote Software Upload in Vehicles With Realistic Vehicle Movements [7]

Penelitian dengan judul "*Wireless Multicasting for Remote Software Upload in Vehicles With Realistic Vehicle Movements*" ini dilakukan oleh Radovan Miucic dan Syed Masud Mahmud. Penelitian ini dapat dibilang sejalan dan mendukung dengan penelitian yang ditunjukkan pada Bagian 2.1.1, akan tetapi penelitian ini lebih menekankan pada aspek cara komunikasi yang dilakukan melalui media *wireless*. Seperti yang sudah disebutkan sebelumnya, masalah yang dihadapi dalam melakukan *over-the-air update* sangat terkait dengan masalah-masalah yang dihadapi dalam *wireless communication*, salah satunya adalah masalah keterbatasan *bandwidth*. Penelitian ini merancang dan membuktikan bahwa komunikasi secara *multicast* jauh lebih efisien dan dapat menyelesaikan permasalahan keterbatasan *bandwidth* dalam *over-the-air update* yang perlu mengirimkan data berupa *updated code* yang ukurannya jauh lebih besar dibandingkan data-data yang umumnya dibaca dari sensor. Simulasi yang dilakukan dalam penelitian ini mengambil *use case* berupa *firmware update* untuk *embedded systems* di transportasi dengan berbagai *constraint* yang dihadapi dalam *wireless communications*. Poin yang dapat difokuskan dari hasil

penelitian ini adalah bahwa dalam *wireless communication* terutama untuk *over-the-air update*, metode *multicast* jauh lebih *cost-efficient* dibandingkan dengan cara *unicast*.

2.1.3. Internet of Things: Over-The-Air (OTA) Firmware Update in LightweightMesh Network Protocol for Smart Urban Development [1]

Penelitian dengan judul "*Internet of Things: Over-the-Air (OTA) Firmware Update in LightweightMesh Network Protocol for Smart Urban Development*" ini dilakukan oleh Hans Chandra dan Erwin Anggadjaja. Penelitian ini merupakan contoh implementasi dari arsitektur *mesh peer-to-peer* untuk *over-the-air firmware update* dengan menggunakan protokol *LightweightMesh Network*, yang merupakan protokol *mesh network proprietary* milik Atmel. Di dalam penelitian ini juga dijelaskan pemisahan *layer* seperti *external network* dan *internal network*, serta mempertegas poin-poin mengenai kelemahan dari *over-the-air update* yang terkait dengan masalah dalam *wireless communication*.



Gambar 2.1 Arsitektur *Mesh* dalam *OTA Update* untuk *IoT Systems*

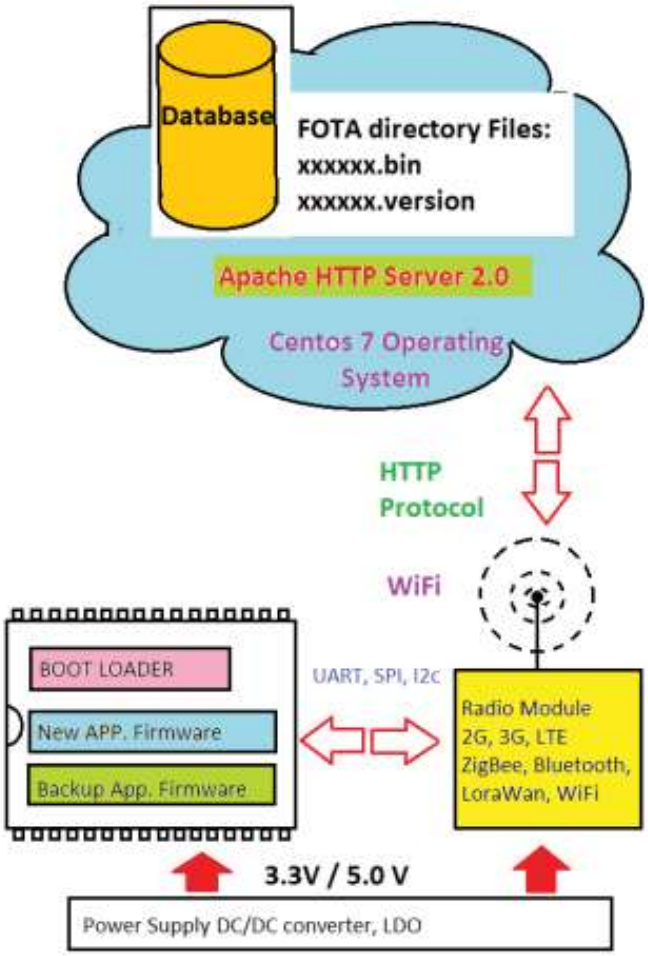
Gambar 2.1 mendeskripsikan arsitektur yang digunakan dalam penelitian ini. Penelitian ini menggunakan arsitektur *semi-hierarchical* sampai dengan tingkat *gateway node*, dan setelahnya hubungan antar e*nd node* menggunakan *mesh peer-to-peer*. Poin-poin yang dapat dicermati dari penelitian ini antara lain:

1) Arsitektur *mesh peer-to-peer* hanya diberlakukan di *end node*, tetap diperlukan sebuah *gateway node* yang berperan sebagai *controller* yang menjembatani antara *server* dan *end nodes*.
2) Komunikasi *peer-to-peer* sangat mungkin diimplementasikan untuk *over-the-air firmware update*, akan tetapi diperlukan cara handshaking yang lebih *reliable* untuk mencegah *key collision*.

2.1.4. Research Firmware Update Over the Air from the Cloud [3]

Penelitian dengan judul "*Research Firmware Update Over the Air from the Cloud*" ini dilakukan oleh Neven Nikolov dan lebih menekankan pada implementasi spesifik *over-the-air update* untuk ESP8266. Penelitian ini dapat dijadikan referensi dari segi implementasi, akan tetapi implementasi dalam penelitian ini masih berupa *over-the-air update* dengan arsitektur *client-server* dan *periodical update*. Hasil implementasi tersebut dapat dibilang cukup *scalable* hingga tingkat tertentu, dan cukup fleksibel karena mengimplementasikan protokol yang umum digunakan yakni HTTP.

Gambar 2.2 menggambarkan arsitektur yang cukup mendetail yang diimplementasikan dalam penelitian ini. Sekali lagi dapat dilihat penggunaan arsitektur *hierarchical* berbasis *client-server* dalam komunikasi antara *end node* melalui *controller* dan *server*. Gambar berikut juga menjelaskan lebih jauh mengenai interaksi antara *controller* yang digambarkan dalam persegi berwarna kuning dan *end node* yang menjadi target dari *firmware update*, terutama interaksinya dengan *flash memory* dan *bootloader* dari *end node*.



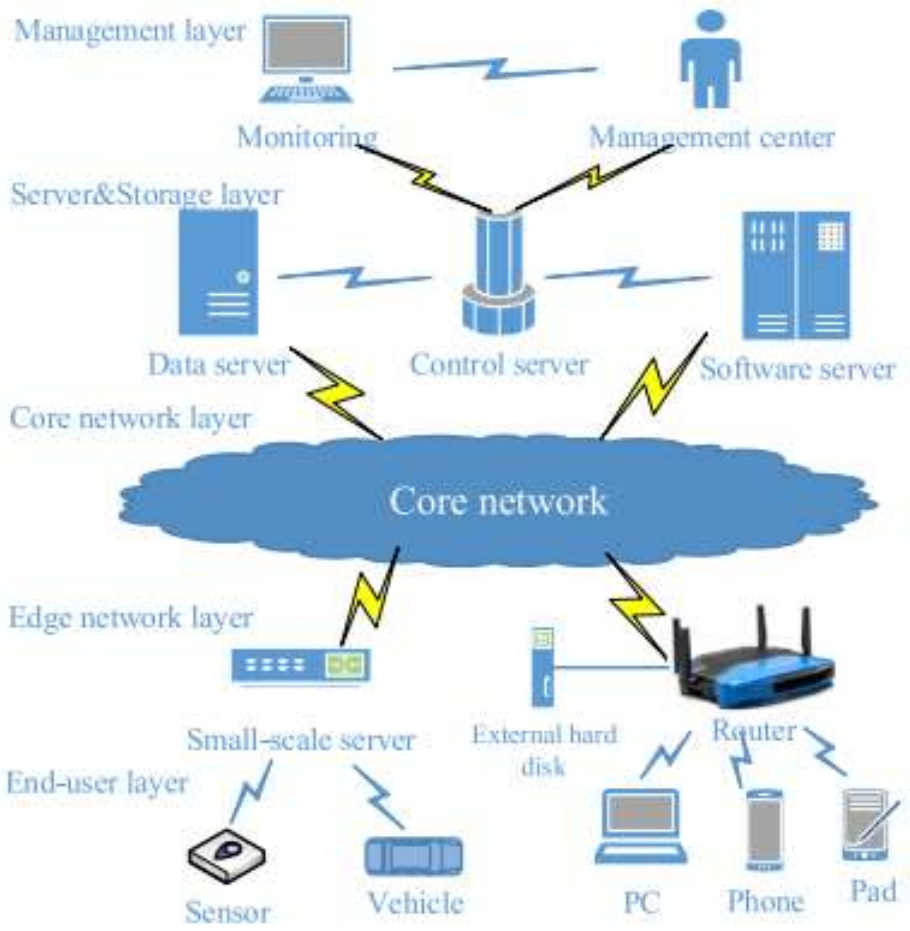
Gambar 2.2 Arsitektur *OTA Update IoT System* yang diimplementasikan di *cloud*

2.1.5. A Scalable and Manageable IoT Architecture Based on Transparent Computing [2]

Penelitian dengan judul "*A Scalable and Manageable IoT Architecture Based on Transparent Computing*" ini dilakukan oleh Hui Guo, Ju Ren, Deyu Zhang, Yaoxue Zhang, dan Junying Hu. Hasil penelitian ini merepresentasikan salah satu contoh arsitektur *IoT System* yang *scalable* dan modular. Hal tersebut dilakukan melalui *transparent computing* yang pada dasarnya melakukan pemisahan antara perangkat lunak dan keras, serta memisahkan komponen-komponen dalam suatu *IoT System* berdasarkan peran dan fungsinya. Penelitian ini menyinggung aspek eksekusi dan *data transfer* antar *IoT node* dengan mengusung konsep

transparent computing, tetapi dapat dilihat bahwa *IoT node* yang dijadikan target eksekusi dan data transfer di sini masih berupa perangkat keras yang kemampuan komputasinya cukup tinggi seperti PDA, telepon genggam, serta laptop. Dijelaskan bahwa arsitektur yang diajukan dalam penelitian ini masih menghadapi tantangan berupa banyaknya variasi dari *IoT Nodes* yang memiliki perangkat keras dan lunak yang berbeda-beda, serta permasalahan dari segi jaringan *wireless communications*, yakni mengenai ketidakstabilan dan *bandwidth* yang kecil dari media komunikasi nirkabel. Ide berupa arsitektur dari *IoT System* yang *scalable* dapat diadaptasi untuk digunakan peneliti dalam membantuk sebuah *IoT Support System*, akan tetapi untuk mengatasi tantangan keberagaman *IoT Node* dan tantangan dari jaringan nirkabel masih perlu penyempurnaan.

Gambar 2.3 mendeskripsikan arsitektur yang digunakan dalam penelitian ini. Penggunaan konsep *transparent computing* dalam arsitektur ini memungkinkan pemisahan antara *hardware* dan *software* sehingga komponen-komponen dalam arsitektur ini yang memiliki fungsi berbeda dapat dipisahkan sehingga menjadi lebih *scalable*, tetapi tetap mudah untuk di-*manage*. Hal tersebut menggambarkan salah satu keuntungan dari arsitektur *hierarchical* seperti yang digunakan juga pada penelitian-penelitian sebelumnya.



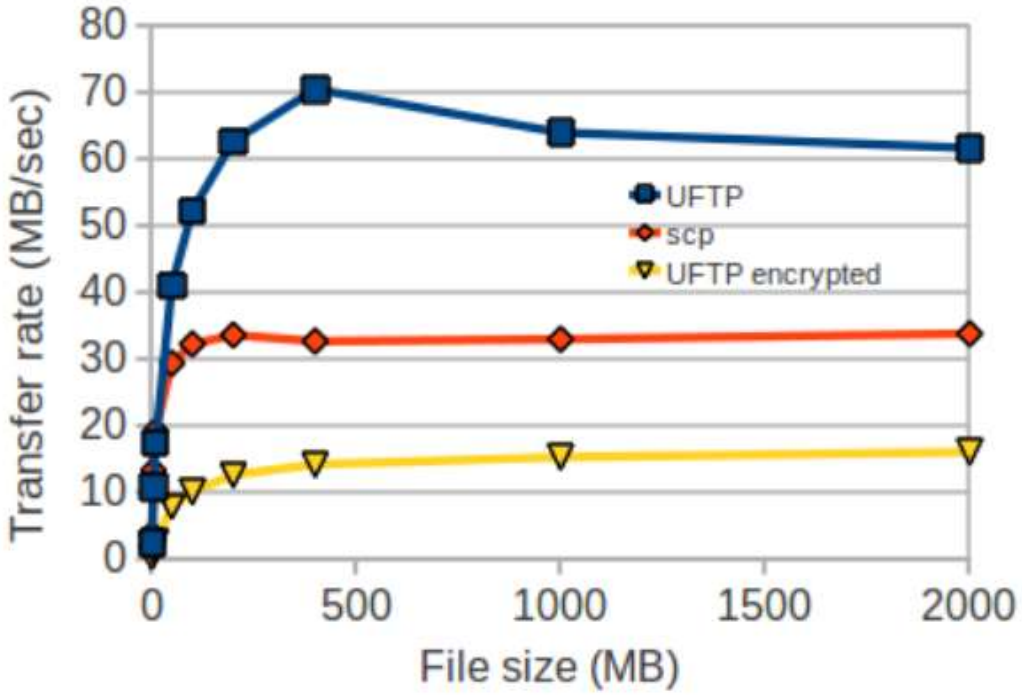
Gambar 2.3 Arsitektur *IoT System* yang *scalable* berbasis *transparent computing*

2.1.6. UFTP: High-Performance Data Transfer for UNICORE [4]

Penelitian dengan judul "*UFTP: High-Performance Data Transfer for UNICORE*" ini dilakukan oleh Bernd Schuller dan Tim Pohlmann. Penelitian ini membahas mengenai kebutuhan akan *high-performance data transfer protocol* untuk digunakan dalam sebuah *distributed computing middleware* yang sangat mementingkan distribusi data dengan cepat dan efisien. Dalam penelitian ini, dikenalkan sebuah protokol *file transfer* berbasis UDP yang dapat mengirimkan data dengan *bandwidth efficient* tetapi tetap memiliki *transfer rate* yang tinggi, serta memiliki kemampuan untuk melakukan *parallel distribution of data* ke beberapa target sekaligus. Hasil dari penelitian ini membandingkan performa dari pendistribusian data

menggunakan UFTP dan dibandingkan dengan protokol BFT yang digunakan *middleware* secara default dan cara distribusi yang umum digunakan seperti *scp*. Dari hasil penelitian tersebut, terbukti UFTP dapat memberikan performa yang lebih baik dari segi *transfer rate* dan tetap memberikan *reliability* yang baik untuk *use case distributed systems* sekalipun.

Gambar 2.4 menunjukkan perbandingan performa antara UFTP biasa, *UFTP encrypted*, serta *scp* dalam melakukan pengiriman data. Dapat dilihat bahwa semakin besar ukuran file yang ingin dikirimkan, semakin tampak perbedaan performa antara aplikasi-aplikasi ini. Pengukuran ini dilakukan pada platform *grid computing UNICORE*.

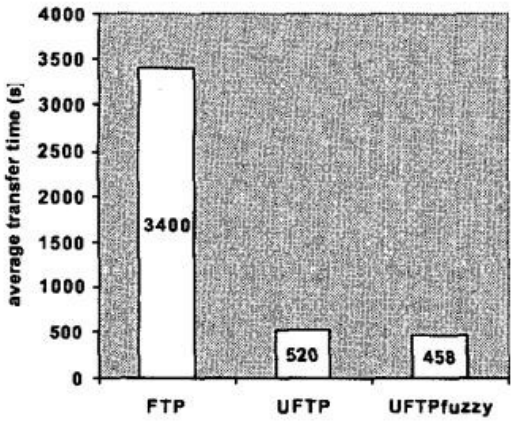


Gambar 2.4 Perbandingan *transfer rate* UFTP dan *scp* dalam *file transfer*

2.1.7. A UDP-Based File Transfer Protocol with Flow Control Using Fuzzy Logic Approach [5]

Penelitian dengan judul "*A UDP-Based File Transfer Protocol with Flow Control Using Fuzzy Logic Approach*" ini dilakukan oleh Jingsong Zhang dan Robert D. McLeod. Dalam penelitian ini, dibahas secara singkat mengenai perbandingan antara protokol TCP dan UDP dalam *use case file transfer*. Seperti yang dijelaskan di [4], implementasi dari protokol TCP untuk *file transfer* terbukti kurang efisien dari segi kecepatan dan penggunaan *bandwidth* dikarenakan sifat protokol TCP yang sangat *connection-oriented*. Penelitian ini mencoba mengembangkan salah satu protokol *file transfer* berbasis UDP yakni UFTP dengan menambahkan sebuah *fuzzy controller* untuk mengatur *transfer delay* antar paket dalam protokol UDP ketika dilakukan *file transfer*. Penelitian ini juga tetap melakukan perbandingan dengan protokol FTP yang berbasis TCP.

Gambar 2.5 kembali menunjukkan perbandingan antara waktu yang diperlukan untuk melakukan *point-to-point file transfer* melalui internet pada lokasi geografis yang cukup jauh dengan membandingkan FTP, UFTP, serta UFTP yang diberikan tambahan *fuzzy flow controller*. Tampak bahwa kedua jenis UFTP, yang berbasis UDP, memiliki rata-rata waktu transfer yang jauh lebih cepat dibandingkan FTP yang berbasis TCP. Pengukuran ini dilakukan dengan file berukuran sekitar 30 *MegaByte*.



Gambar 2.5 Perbandingan *transfer time* dari FTP, UFTP standard, & UFTP *fuzzy*

2.1.8. An Efficient Scheme for Applying Software Updates in Pervasive Computing Applications [8]

Penelitian dengan judul "*An Efficient Scheme for Applying Software Updates in Pervasive Computing Applications*" ini dilakukan oleh Kostas Kolomvatsos. Penelitian ini membahas lebih dalam mengenai sistem IoT pada umumnya dan kelemahan-kelemahan yang dihadapi oleh "*legacy systems*" yang pada umumnya bersifat terpusat dan menggunakan metode *broadcast* untuk bertukar data. Fokus dari penelitian ini lebih ke merancang model yang dapat mendeteksi waktu yang tepat, dengan mempertimbangkan berbagai parameter, untuk melakukan *update* secara terdistribusi dan terdesentralisasi. Akan tetapi, beberapa poin mengenai kelemahan dari sistem yang terpusat dapat dijadikan acuan dalam merancang arsitektur dalam penelitian penulis:

1) Sistem tersentralisasi dengan metode *broadcast* dapat memiliki dampak buruk pada *bandwidth* jaringan dalam berbagai aspek (*power usage*, *collision rate*, *message overhead*, dan lain-lain).
2) Keberagaman *IoT Nodes* sangat berpengaruh terhadap ditentukannya suatu protokol yang dapat diaplikasikan ke semua atau sebagian besar jenis *IoT nodes*.
3) Permasalahan skalabilitas berdasarkan jumlah *IoT Node* yang terhubung ke sistem pusat akan sangat menambah *load* atau kompleksitas dari aplikasi.

2.1.9. Design and Implementation of a Platform for Hyperconnected Cyber Physical Systems [9]

Penelitian dengan judul "*Design and Implementation of a Platform for Hyperconnected Cyber Physical Systems*" ini dilakukan oleh Ian Thomas, Shinji Kikuchi, Emmanuel Baccelli, Kaspar Joerg Doerr, dan Andreas Morgenstern. Penelitian ini dapat dijadikan contoh dari

implementasi sistem *over-the-air update* untuk IoT Systems berbasis *cloud*. Dalam penelitian ini, *updates* dikirimkan dengan protokol CoAP yang dapat dikatakan sebagai "HTTP untuk perangkat yang sangat terbatas". CoAP di penelitian ini digunakan untuk mengirimkan *string* yang nantinya akan dibaca oleh *interpreter* yang berjalan di *end node* dan dijalankan sebagai kode, dan *code update* dilakukan hanya dengan mengirimkan baris-baris yang ingin dijalankan. Akan tetapi, pendekatan ini berarti implementasi seperti yang ditunjukkan penelitian ini hanya akan mendukung *end nodes* yang dapat menjalankan *interpreter*. Terlebih lagi, dengan menjalankan *interpreter* pada *end node*, *resource* dan *memory* akan lebih terbatasi lagi. Beberapa poin yang dapat dititikberatkan dalam penelitian ini antara lain:

1) Metode pengiriman *updated code* berupa *string* untuk selanjutnya dijalankan (*streaming*) dapat menjadi metode *update* untuk dipertimbangkan, tetapi memerlukan *end node* yang menjalankan *interpreter*.
2) Permasalahan banyaknya variasi *IoT Nodes* dalam penelitian ini "diselesaikan" dengan melakukan implementasi pada *IoT Platform* yang *portable* (dapat digunakan di banyak *IoT Node*) yakni RIOT.

2.1.10. A Comparative Evaluation of AMQP and MQTT Protocols over Unstable and Mobile Networks [10]

Penelitian dengan judul "*A Comparative Evaluation of AMQP and MQTT Protocols over Unstable and Mobile Networks*" ini dilakukan oleh Jorge E. Luzuriaga, Miguel Perez, Pablo Boronat, Juan Carlos Cano, Carlos Calafate dan Pietro Manzoni untuk membandingkan performa dari protokol *messaging* AMQP dan MQTT dalam jaringan-jaringan yang bersifat *mobile* dan tidak stabil. Kedua protokol tersebut merupakan protokol yang paling sering digunakan baik dalam dunia industri maupun akademik. Penelitian ini membandingkan aspek perilaku dari kedua protokol dalam menangani

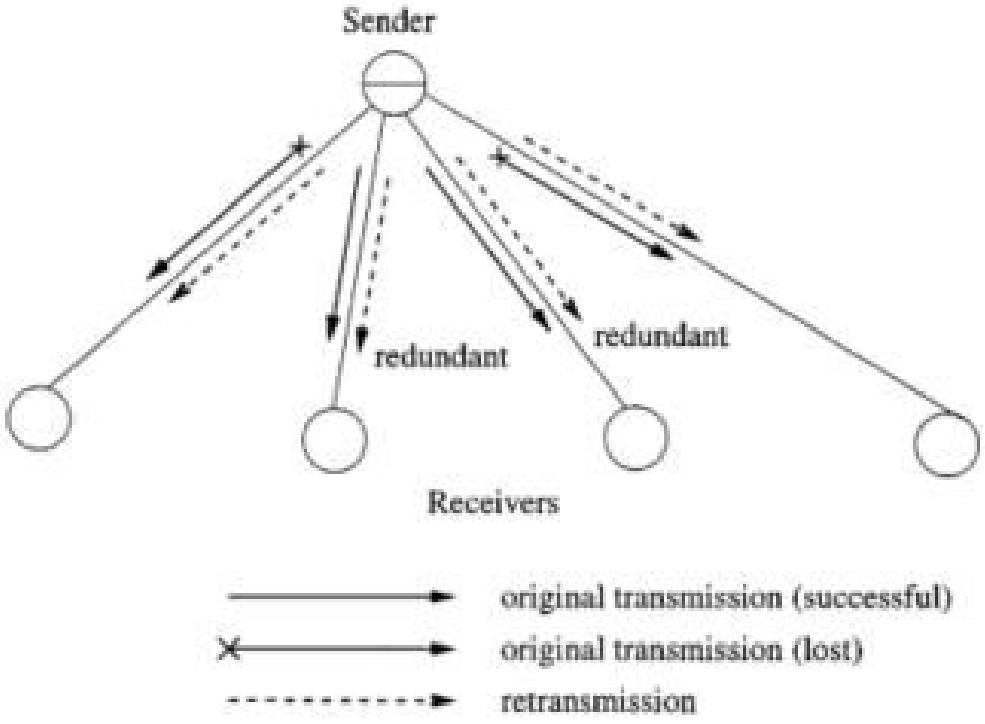
masalah *handoff* yang sering terjadi dalam *mobile network*, serta *jitter* yang dihasilkan dan kemampuan serta batas atas dari kedua protokol dalam kondisi jaringan yang memiliki *load* tinggi. Dari penelitian ini, diambil kesimpulan bahwa dari segi reliabilitas, skalabilitas, dan fitur, AMQP lebih unggul dibandingkan MQTT. Akan tetapi protokol AMQP membutuhkan jaringan yang lebih stabil dan aman, sedangkan MQTT mampu mendukung perangkat-perangkat dengan kemampuan komputasi yang kecil serta dapat bekerja dengan lebih baik dalam kondisi jaringan yang lebih terbatas.

2.1.11. Scalable Reliable Multicast Using Multiple Multicast Channels [11]

Penelitian dengan judul "*Scalable Reliable Multicast Using Multiple Multicast Channels*" ini dilakukan oleh Sneha Kumar Kasera, Gísli Hjálmtýsson, Donald F. Towsley, dan James F. Kurose untuk bereksperimen pada penggunaan beberapa *multicast channel*, terutama *IP Multicast* dalam beberapa konfigurasi berbeda dalam mengirimkan kembali *lost packet* yang ditujukan pada beberapa *receivers*. Pendekatan pada penelitian ini dilakukan dengan beberapa cara, antara lain dengan mengasumsikan jumlah *multicast channel* yang tersedia sebagai jumlah yang terbatas dan tidak berhingga (*infinite*). Selain itu, pengujian juga dilakukan dengan beberapa konfigurasi, antara lain konfigurasi yang melibatkan perangkat infrastruktur tambahan seperti *router* dan konfigurasi *local network*. Hasilnya, tampak bahwa dari sisi *receiver*, jumlah *retransmission packet* yang diterima menjadi lebih minim, sehingga *resource* di sisi *receiver* dapat digunakan dengan lebih efisien. Hal tersebut dapat terjadi karena bila *sender* mendeteksi adanya paket-paket yang gagal diterima oleh *receiver*, maka pengiriman kembali paket-paket tersebut akan dilakukan pada *temporary multicast channel* supaya hanya *receiver-receiver* yang membutuhkannya yang akan melakukan *listen* pada *temporary multicast channel* tersebut. Setelah proses *retransmit* selesai, *multicast channel* tersebut kemudian di-*free* untuk digunakan kembali

nantinya. Gambar 2.6 menggambarkan masalah yang ingin diselesaikan pada penelitian ini, dimana *receiver* yang tidak membutuhkan *retransmission data* tetap menerima data itu dan dianggap sebagai *wasted receiver resource*. Beberapa poin penting yang dapat dikaji dari penelitian ini adalah:

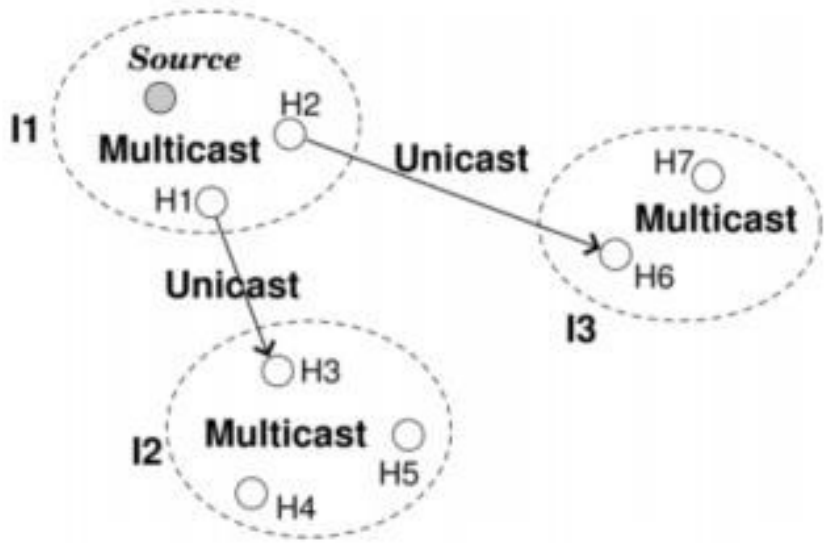
1) Penggunaan beberapa *multicast channel* untuk *data retransmission* berhasil mengurangi *wasted receiver resource* dalam konteks *network resource*.
2) Untuk kasus *one-to-many communication*, maka beberapa *multicast channel* dalam jumlah yang kecil sudah dapat memberikan peningkatan efisiensi yang baik bagi performa keseluruhan sistem dalam menangani *data retransmission*.
3) Untuk kasus *many-to-many communication*, diperlukan jumlah *multicast channel* yang lebih banyak dibandingkan kasus *one-to-many* untuk dapat memberikan peningkatan efisiensi dan performa sistem yang baik.
4) Penggunaan komponen infrastruktur tambahan seperti dengan melibatkan *router* dapat mengurangi penggunaan *bandwidth* dalam jaringan serta semakin mengurangi penggunaan *resource* bagi perangkat *receiver*.



Gambar 2.6 Redundansi *retransmission data* yang diterima *receiver* dalam *multicast data transmission* pada umumnya.

2.1.12. Island Multicast: Combining IP Multicast with Overlay Data Distribution [12]

Penelitian dengan judul "*Island Multicast: Combining IP Multicast with Overlay Data Distribution*" ini dilakukan oleh Xing Jin, Kan-Leung Cheng, dan S.-H. Gary Chan ini berusaha memberikan solusi pada permasalahan yang dihadapi sistem yang mengimplementasikan *multicast*, yakni ruang lingkup infrastruktur yang mendukung *multicast* yang cenderung terisolasi satu sama lainnya. Istilah yang digunakan penelitian ini untuk merujuk pada cara menghubungkan jaringan-jaringan kecil yang mendukung adalah *island multicast*. Dengan menggunakan beberapa konsep *distributed systems*, peneliti-peneliti ini merancang protokol *island multicast* yang menggunakan komunikasi *unicast* untuk menghubungkan *multicast islands* yang saling terisolasi.

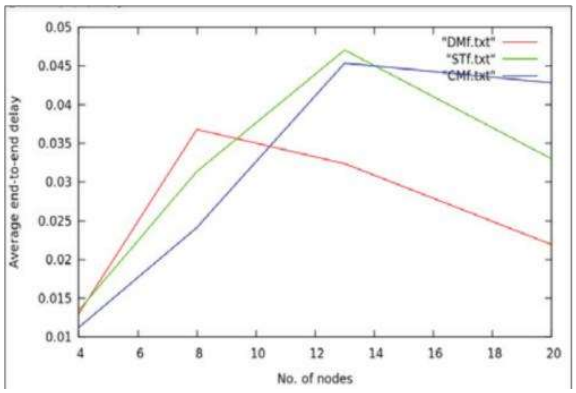


Gambar 2.7 *Island Multicast* mehubungkan jaringan *multicast* yang terisolasi

Terdapat 2 rancangan protokol yang menjadi *output* dari penelitian ini, dan masing-masing protokol memfokuskan pada *use case* yang berbeda. Protokol CIM (*Centralized Island Multicast*) lebih cocok digunakan pada komunikasi *many-to-many* dengan keperluan *bandwidth* yang tinggi dan memerlukan *efisiensi* pengiriman yang baik. Protokol DIM (*Distributed Island Multicast*) cocok digunakan untuk *use case* yang memerlukan *scalability* dan dengan jumlah *receiver* yang besar. Penelitian ini memberikan gambaran akan masalah yang dihadapi jaringan *multicast* pada umumnya, serta menawarkan solusi-solusi yang dapat diimplementasikan untuk mengatasi keterbatasn jaringan *multicast* tersebut. Gambar 2.7 menggambarkan masalah *multicast islands* dan solusi yang ditawarkan untuk mengatasi masalah tersebut dengan menggunakan komunikasi *unicast* untuk menjembatani *multicast islands* tersebut.

2.1.13. Comparative Study of Multicasting Protocols Based on Average End-to-End Delay [13]

Penelitian dengan judul "*Comparative Study of Multicasting Protocols Based on Average End-to-End Delay*" ini dilakukan oleh Anuja Golechha, Snehal Karanje, dan Jibi Abraham. Penelitian ini lebih memfokuskan pada karakteristik dari beberapa *multicast routing protocol* dan membandingkan performa mereka dalam beberapa konfigurasi jaringan. Protokol-protokol *multicast routing* yang dibandingkan antara lain *Centralized Multicast (CM)*, *Protocol Independent Multicast – Dense Mode (PIM-DM)*, serta *Protocol Independent Multicast – Sparsse Mode (PIM-SM)*. Penelitian ini juga cukup informatif dalam menjelaskan *multicast routing protocol* tersebut dan cara kerjanya, beserta dengan *use case* yang mendukung. Pengujian dilakukan dengan menjalankan simulasi NS2 dan mengukur *end-to-end delay* dalam beberapa konfigurasi jaringan. Dari hasil pengujian, diambil kesimpulan bahwa perbedaan *end-to-end delay* antar protokol tidak memiliki perbedaan yang terlalu signifikan, tetapi untuk jumlah *receiver* kurang dari 10, protokol *CM* memberikan *end-to-end delay* paling rendah sedangkan untuk jumlah *receiver* lebih dari 10, performa *end-to-end delay* dari *PIM-DM* merupakan yang paling baik diantara ketiga protokol tersebut. Gambar 2.8 menggambarkan perbandingan antara *end-to-end delay* dari ketiga protokol dan hubungannya dengan jumlah *receiver*.



Gambar 2.8 Rata-rata *end-to-end delay* untuk beberapa *multicast routing protocol* dan hubungannya dengan jumlah *end nodes*

2.2. Summary

Berdasarkan studi pustaka yang telah dilakukan terhadap referensi-referensi penelitian yang sudah disebutkan, penelitian ini dijalankan dengan beberapa aspek pertimbangan sebagai berikut:

1) Dari segi arsitektur, yang paling umum digunakan dalam *IoT System* adalah arsitektur *hierarchical* yang terdiri dari *server*, *gateway/controller*, serta *end nodes*. Arsitektur jenis ini digunakan karena modularitas yang didukung arsitektur tersebut membuka kemungkinan implementasi yang fleksibel dan beragam.
2) Kendala utama yang dihadapi *IoT System* yang melibatkan jaringan nirkabel adalah berupa efisiensi daya pada perangkat, bandwidth yang terbatas, serta *reliability* dari kanal komunikasi nirkabel pada umumnya. Kendala ini sering ditemui pada sistem yang mengimplementasikan protokol berbasis TCP.
3) Untuk mengatasi kendala dari jaringan nirkabel tersebut, digunakan protokol-protokol yang mendukung penggunaan *bandwidth* yang kecil namun tetap mengimplementasikan suatu mekanisme untuk mendukung *reliability* dari data-data yang yang dikirimkan untuk memastikan pesan atau data sampai di tujuan.
4) Metode komunikasi secara *multicast* dapat menjadi metode yang *bandwidth friendly* dan efisien dari segi *resource* milik *sender* untuk mengirimkan data atau informasi dalam kasus *one-to-many* atau *many-to-many communications*, tetapi terkadang terbatas pada ruang lingkup dan dukungan dari infrastruktur yang memadai serta perlu implementasi tambahan untuk mewujudkan *multicast communication* yang tetap *reliable* dan cepat.

2.3. Perangkat Keras

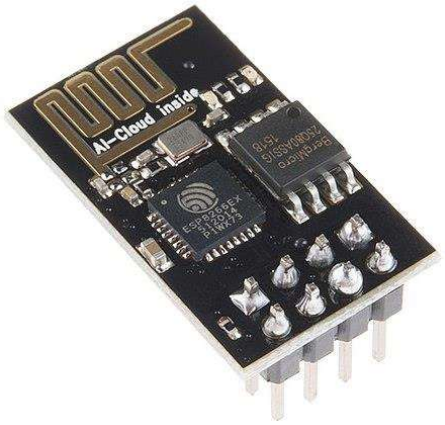
Perangkat-perangkat keras yang dideskripsikan pada bagian ini merupakan perangkat keras yang terlibat dalam penelitian ini, terutama pada implementasi sistem untuk penelitian. Deskripsi pada bagian ini berperan untuk memberikan gambaran sebelum memasuki Bab III yang menjelaskan rancangan sistem serta keterlibatan perangkat-perangkat keras berikut dalam sistem untuk penelitian ini.

2.3.1. Raspberry Pi

Raspberry Pi merupakan jenis *single-board computer* yang dikembangkan oleh *Raspberry Pi Foundation* dengan berbagai fitur yang lebih diarahkan untuk penggunaan *software development environment* atau *robotic system development*. Biaya yang murah, perangkat keras yang memadai, serta ukuran yang kecil membuat *Raspberry Pi* banyak digunakan dalam *IoT System*, terutama seri *Raspberry Pi Zero* yang memiliki ukuran paling kecil diantara semua jenis *Raspberry Pi* (seukuran kartu kredit).

Perlu diingat bahwa hanya *Raspberry Pi* model tertentu yang memiliki *onboard WiFi* (*Pi 3, Zero W, Pi 4*) dan model-model tersebut semuanya memiliki *general purpose input output pin* (GPIO) dengan susunan dan jumlah yang sama sebanyak 40 pin. Untuk penelitian ini, model *Raspberry Pi* yang akan digunakan adalah *Raspberry Pi 3 Model B+* seperti yang ditunjukkan pada Gambar 2.9.

Gambar 2.9 *Raspberry Pi 3 Model B+*



Gambar 2.10 Modul ESP-01 yang menggunakan *chip* ESP8266

2.3.2. ESP8266

ESP8266 merupakan sebuah 32-bit *WiFi microcontroller chip* generasi pertama yang dilengkapi dengan *Full TCP/IP Stack* serta kemampuan dari *microcontroller* pada umumnya dalam bentuk yang kecil serta harga yang murah. Istilah ESP8266 merujuk pada *microcontroller chip* yang digunakan, sedangkan *IoT Engineer* dan *Developer* umumnya menggunakan modul yang menggunakan *chip* ESP8266 (ESP-12E, ESP-12, ESP-01, dan lain-lain). Modul-modul varian terbaru menyediakan *flash memory* berukuran 4 MiB, sedangkan modul varian lama (ESP-12 ke bawah) memiliki *flash memory* berukuran 1 MiB. Gambar 2.10 menunjukkan modul ESP-01 yang merupakan modul generasi pertama buatan Ai-Thinker untuk kawasan barat yang menggunakan *chip* ESP8266.

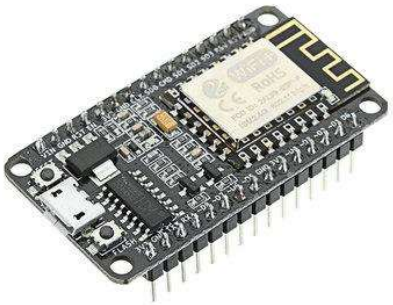
Modul-modul yang menggunakan *chip* ESP8266 seperti yang sudah disebutkan di atas umumnya berfokus pada ukuran kecil dan optimisasi penggunaan daya untuk penggunaan pada tingkat *production*, sedangkan untuk *development* dibutuhkan berbagai bagian tambahan seperti sumber daya eksternal 3.3V serta *USB-to-UART Bridge* untuk mengembangkan aplikasi pada modul-modul tersebut. Oleh karena itu, berbagai jenis *development board* mulai muncul dan menjadi popular dikarenakan sudah mengintegrasikan sumber daya dan *USB-to-UART Bridge* dengan

MicroUSB Connector untuk mempermudah penggunaan. Salah satu *development board* berbasis ESP8266 yang paling popular adalah *NodeMCU DevKit.*

2.3.3. NodeMCU

NodeMCU merupakan istilah yang umumnya merujuk kepada *NodeMCU DevKit*, sebuah *IoT Platform* berbasis modul ESP-12E yang mengintegrasikan *USB-to-UART Bridge*, 3,3V *voltage regulator* yang dapat diakses melalui *MicroUSB port* serta akses GPIO yang mudah dalam bentuk yang tetap kecil. Akan tetapi, *NodeMCU* sebenarnya lebih merujuk kepada *firmware* yang digunakan untuk melakukan *open-source board design prototyping*, dan *NodeMCU DevKit* yang ditunjukkan pada Gambar 2.11 merupakan salah satu hasil rancangan dari *firmware* tersebut.

Umumnya, *NodeMCU DevKit* menggunakan *chip* ESP-12E yang memiliki *flash memory* 4 MiB. Selain itu, akses-akses pin untuk *chip* tersebut juga dimudahkan dengan *header pin*. *Firmware* pada *IoT Platform* tersebut menggunakan bahasa *scripting* Lua, akan tetapi dengan Arduino IDE dan fitur *Boards Manager*, pengembangan aplikasi untuk *NodeMCU DevKit* dapat dilakukan melalui Arduino IDE dengan mudah.



Gambar 2.11 *NodeMCU DevKit*

2.4. Perangkat Lunak

Selama perencanaan dan implementasi sisten untuk penelitian ini, dikembangkan beberapa aplikasi untuk perangkat-perangkat yang terlibat. Aplikasi-aplikasi tersebut dikembangkan dengan mempertimbangkan perangkat yang digunakan, *cross-platform compatibility*, *maintaibility*, serta fleksibilitas dari segi implementasi. Implementasi dari perangkat lunak akan dibahas pada Bab IV, sedangkan bagian ini akan membahas bahasa dan konsep yang digunakan dalam mengembangkan perangkat lunak untuk sistem pada penelitian ini.

2.4.1. Python

Python merupakan salah satu *high-level interpreted language* yang dapat digunakan untuk menyelesaikan berbagai masalah mulai dari administrasi sistem, keperluan penelitian matematis, pemrosesan grafis, pengembangan *web*, dan lain sebagainya. Bahasa ini berfokus pada pengembangan kode yang bisa dibaca dengan *syntax* yang terdiri dari bahasa-bahasa yang mudah dipahami, serta kumpulan *syntactic sugar* yang sangat memudahkan dan mempersingkat kode dengan tetap mempertahankan *code readability*. Bahasa ini juga mendukung paradigma pemrograman yang bermacam-macam. Semua fitur tersebut bertujuan memberikan *developer* pengalaman yang lebih baik dalam mengembangkan perangkat lunak.

Pengembangan perangkat lunak menggunakan Python juga dimudahkan dengan ketersediaan *library* dengan berbagai fungsi tambahan. Menjaga kebaruan dari *library-library* tersebut juga dimudahkan dengan *package manager* dan kemampuan otomasi yang disediakan *framework* tertentu. Kemudian apabila pengembangan kode sudah selesai dan program ingin dikemas dalam bentuk *executable* tertentu, beberapa *library* juga disediakan untuk melakukan kompilasi program Python ke bentuk-bentuk *executable* yang dapat dijalankan di berbagai sistem operasi.

Python sendiri berjalan di atas sebuah *interpreter*, dan saat ini sudah tersedia berbagai variasi *interpreter* yang mendukung berbagai teknologi dan berbagai sistem operasi. Hal tersebut memungkinkan program yang dikembangkan dengan Python untuk dijalankan di hampir semua jenis perangkat, bahkan di perangkat yang terbatas seperti *Raspberry Pi* yang memang mendukung Python, atau di *NodeMCU* melalui *MicroPython interpreter*.

2.4.2. C++

C++ merupakan bahasa pemrograman yang berawal sebagai ekstensi dari bahasa pemrograman C. Bahasa C++ kini telah berkembang menjadi bahasa yang umum untuk digunakan dalam kasus yang membutuhkan program dengan performa tinggi dan *resource efficient*, antara lain *system programming* dan penggunaan untuk *embedded systems* pada umumnya. C++ juga banyak digunakan untuk program-program aplikasi *desktop*, *server*, bahkan permainan video menggunakan bahasa C++. Dukungan untuk fitur-fitur yang cukup *high-level* maupun *low-level* membuat bahasa ini cukup fleksibel untuk diimplementasikan.

Selain itu, pemrograman melalui Arduino IDE atau pemrograman *development board* untuk *IoT* pada umumnya juga menggunakan bahasa C++ sebagai bahasa pemrogramannya. Program untuk *embedded* ataupun *IoT Systems* yang sudah dikembangkan nantinya akan dikompilasi menjadi *binary code* yang akan dibaca oleh *microchip* pada *development board* untuk dijalankan fungsionalitasnya. Meskipun beberapa *development board* sudah mendukung dijalankannya sistem operasi ataupun *interpreter*, menjalankan *binary code* yang sudah dikompilasi sebelumnya memastikan program dapat berjalan dengan performa yang baik dan efisien dari segi waktu maupun perangkat keras.

2.4.3. Network Socket

Network Socket merupakan istilah yang merepresentasikan sebuah *logical network endpoint* dari suatu sistem atau perangkat yang terhubung dalam jaringan. *Network Socket* ini umumnya digunakan sebagai pintu keluar ataupun pintu masuk yang dapat diakses oleh suatu aplikasi untuk menerima ataupun mengirimkan data ke aplikasi lain yang berada dalam suatu jaringan.

Socket di sini hanyalah komponen dasar dalam komunikasi antar jaringan. Protokol yang digunakan, *identifier* yang terkait dengan *socket* tersebut, serta jenis dari *socket* tersebut umumnya didefinisikan oleh aplikasi yang membuat dan menggunakan *socket* tersebut. Fungsi dan peran dari *socket* tersebut akan berubah tergantung parameter dan konfigurasi yang diberikan oleh aplikasi. Suatu *socket* dapat digunakan hanya sebagai *input* atau *output* saja, dapat menggunakan alamat IPv4 atau IPv6, mengimplementasikan *transport layer protocol TCP* atau *UDP*, melakukan komunikasi secara *multicast*, dan lain sebagainya.

2.4.4. Multicast

Multicast merupakan cara komunikasi dalam suatu jaringan yang memungkinkan data atau informasi untuk dikirim ke beberapa perangkat tertentu sekaligus. *Multicast* tidak sama dengan *broadcast*, karena *multicast* hanya mengirimkan ke perangkat-perangkat tertentu yang jumlahnya dapat lebih dari satu, sedangkan *broadcast* mengirimkan data atau informasi ke seluruh perangkat dalam jaringan yang bersangkutan.

Cara komunikasi dengan *multicast* dibedakan menjadi *application layer multicasting* dan *network assisted multicast*. *Application layer multicast* umumnya berupa *unicast* yang dilakukan berulang-ulang untuk meniru pola komunikasi *multicast*, sedangkan *multicast* pada tingkat jaringan merupakan cara komunikasi yang lebih efisien. *Network assisted multicast* memastikan bahwa data atau informasi yang dikirim hanya akan

digandakan dan disebarkan kepada perangkat jaringan lainnya melalui jalur-jalur yang sudah diperhitungkan sebelumnya untuk memastikan informasi dikirim ke semua perangkat tujuan dengan lebih optimal.

2.5. Protokol

Dalam penelitian ini, akan dilibatkan beberapa protokol jaringan yang dipilih berdasarkan pertimbangan-pertimbangan yang digarisbawahi di Bagian 2.1. Penggunaan protokol dalam sistem pada penelitian ini akan dijelaskan pada Bab III, dan implementasinya akan dijelaskan pada Bab IV.

2.5.1. UDP

UDP merupakan suatu protokol jaringan yang bekerja pada *transport layer* dari *7 OSI Layers*. Protokol UDP menggunakan model *connectionless communication* dan memiliki karakteristik yang berkebalikan dengan TCP, yakni *unordered*, *unreliable*, serta *lightweight*. Dikarenakan karakteristik-karakteristik tersebut, UDP umumnya digunakan untuk *use case* yang memfokuskan pada aspek *real-time* dan dapat memaklumi adanya data yang tidak sampai ke tujuan, ataupun sampai dalam urutan yang tidak seharusnya. Contoh penggunaan protokol UDP adalah untuk *video/audio streaming* dan *voice call*.

Sifat lainnya yang membedakan UDP dan TCP adalah dukungan untuk mengirimkan data secara *multicast*. Akan tetapi, perlu diingat bahwa UDP sendiri tidak memastikan paket data yang dikirimkan sampai sesuai urutan, atau bahkan memastikan bahwa paket tersebut sampai ke tujuan. Untuk dapat menggunakan UDP dan tetap memastikan *reliability* yang baik, diperlukan implementasi tambahan yang biasanya dilakukan di level aplikasi.

2.5.2. UFTP

UFTP merupakan sebuah *file transfer protocol* dengan kemampuan *multicast* untuk bisa mengirimkan data dengan lebih efisien dan *reliable*. Hal tersebut dicapai dengan menggunakan protokol jaringan UDP untuk pengiriman data dibandingkan penggunaan TCP untuk pengiriman data seperti yang dilakukan di FTP standar. Dengan menggunakan UDP dan sedikit modifikasi dari segi *controller*, UFTP memungkinkan pengiriman ke beberapa *client* sekaligus dengan cara yang *bandwidth efficient* dan dapat dilakukan di dalam jaringan dengan kondisi yang kurang baik. Meski berbasis UDP, protokol ini tetap memiliki mekanisme berupa NACK untuk memungkinkan *client* meminta *server* untuk mengirimkan kembali potongan data yang gagal terkirim. Selain itu juga ada pilihan untuk menambahkan aspek keamanan dalam distribusi datanya.

Sebuah proses *file transfer* dengan UFTP terbagi menjadi 3 langkah:

A. Announce/Register

Langkah ini mempersiapkan sesi *multicast file transfer* dengan *client* dengan cara mengirimkan pesan *ANNOUNCE* ke *public multicast address* yang dapat mencapai *client* yang dituju. Dalam pesan *ANNOUNCE* tersebut juga dikirimkan berbagai informasi, salah satunya *private multicast address* yang akan digunakan untuk fase selanjutnya. Dalam tahap ini, *client-client* yang dituju oleh *server* atau *server-server* yang dapat didengar dan diterima oleh *client* dapat disesuaikan dengan suatu *file* yang berisikan daftar *client/server* yang dituju.

Client yang menerima pesan *ANNOUNCE* tersebut (bila sesuai dengan daftar *client* tujuan dari sisi *server*, serta *allowed server* dari sisi *client*) akan membalas dengan pesan *REGISTER* untuk memberitahukan kepada *server* bahwa *client* tersebut telah mendapatkan informasi yang sesuai dan akan berpartisipasi dalam *multicast file transfer session* tersebut. *Server* kemudian akan membalas dengan pesan berbeda tergantung apakah fitur keamanan dinyalakan atau tidak. Bila tidak dinyalakan, *server* hanya

akan mengirimkan konfirmasi atas pesan *REGISTER* dari *client*. Bila iya, *server* akan mengirimkan *encryption keys* yang kemudian akan diterima oleh *client* dan digunakan untuk fase selanjutnya.

B. File Transfer

Langkah ini akan dilakukan ketika langkah *Announce/Register* dianggap sudah selesai dan jumlah *client* yang berpartisipasi dalam *session* berjumlah lebih dari 0. Langkah ini akan dibagi menjadi 2 tahapan yaitu *File Info* dan *Data Transfer* untuk setiap *file* yang dikirimkan. Tahapan *File Info* akan mengirimkan informasi atau *metadata* dari *file* yang akan dikirimkan, serta bagaimana *file* tersebut akan dipecah-pecah dalam pengiriman. Tahapan *Data Transfer* akan mengirimkan *file* yang sudah dipecah-pecah tersebut ke *client* dengan menomori paket. Hal tersebut dilakukan karena protokol UDP pada dasarnya tidak mendukung *packet reordering* sehingga penomoran paket-paket tersebut akan membantu sisi *client* untuk menyusun kembali paket *file* tersebut.

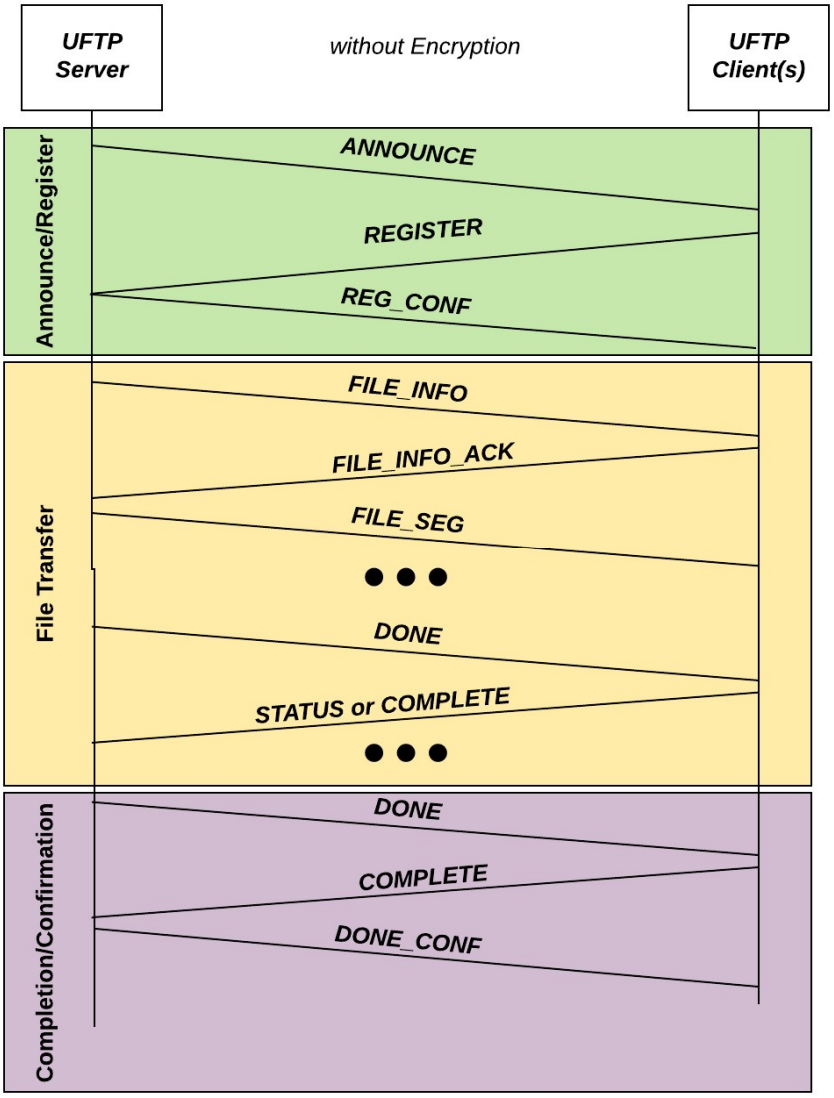
Bila *client* mendeteksi adanya paket dari *file* tertentu yang tidak terkirim, atau ketika *server* sudah mengumumkan bahwa pengiriman *file* telah selesai, *client* akan mengirimkan *negative ACK* (NACK) ke *server* untuk menginformasikan potongan *file* mana saja yang gagal diterima oleh *client*, bila ada. *Server* yang menerima NACK tersebut (dari satu atau beberapa *client*) akan mengirimkan kembali paket-paket yang di NACK oleh *client*, kemudian melanjutkan pengiriman paket selanjutnya, bila ada. Ketika *client* sudah menerima keseluruhan paket untuk sebuah *file*, sebuah pesan konfirmasi akan dikirimkan ke *server*. *Server* akan menunggu hingga waktu tertentu, atau hingga semua *client* yang berpartisipasi dalam *session* tersebut sudah melakukan konfirmasi penerimaan keseluruhan *file*. *Client* yang tidak membalas pesan konfirmasi akan dianggap terputus dari jaringan.

C. Completion/Confirmation

Langkah ini hanya berupa langkah untuk menyelesaikan *multicast file transfer session*. Langkah ini dimulai dengan pesan *end of session* dari *server* ke semua *client* yang terlibat dalam *session* tersebut, kemudian setiap *client* membalas pesan *end of session* tersebut dengan pemberitahuan bahwa *file* sudah diterima dengan baik. *Server* kemudian mengkonfirmasi pemberitahuan tersebut dan *client* dapat meninggalkan *session*.

Selama menjalankan langkah-langkah tersebut, bila terjadi kesalahan maka baik *client* maupun *server* dapat mengirimkan pesan *ABORT* untuk membatalkan proses *file transfer*. Pesan ini (bila dikirimkan dari *server*) dapat dikirimkan ke satu *client* saja untuk membatalkan proses hanya pada *client* tersebut, atau dikirimkan ke semua *client* yang sedang terlibat dalam *file transfer session*.

Gambar 2.12 merepresentasikan alur pengiriman paket untuk *normal use case* antara UFTP *server* dan UFTP *client(s)* tanpa mengaktifkan enkripsi. Tanda *ellipsis* menunjukkan bahwa pesan dapat dilakukan berulang kali, seperti pada *file transfer phase* yang akan mengirimkan paket hingga jumlah tertentu, kemudian dilanjutkan dengan pesan *DONE*. Pesan *STATUS* ataupun *COMPLETE* juga dapat berjumlah lebih dari satu bila *file transfer* dilakukan untuk beberapa *file*.



Gambar 2.12 Skema Mekanisme Kerja Protokol UFTP

2.5.3. HTTP

HTTP atau *Hypertext Transfer Protocol* merupakan protokol jaringan yang bekerja pada *application layer* dalam *7 OSI Layers* yang umumnya digunakan untuk menjembatani komunikasi *client* dan *server* dalam bertukar data berupa *hypertext*. Rancangan dari HTTP mengasumsikan implementasi dari *transport layer protocol* yang *reliable*, sehingga umumnya digunakan TCP sebagai protokol pada *transport*

layernya. Akan tetapi hal tersebut tidak menutup kemungkinan digunakannya UDP sebagai *transport layer* dari HTTP.

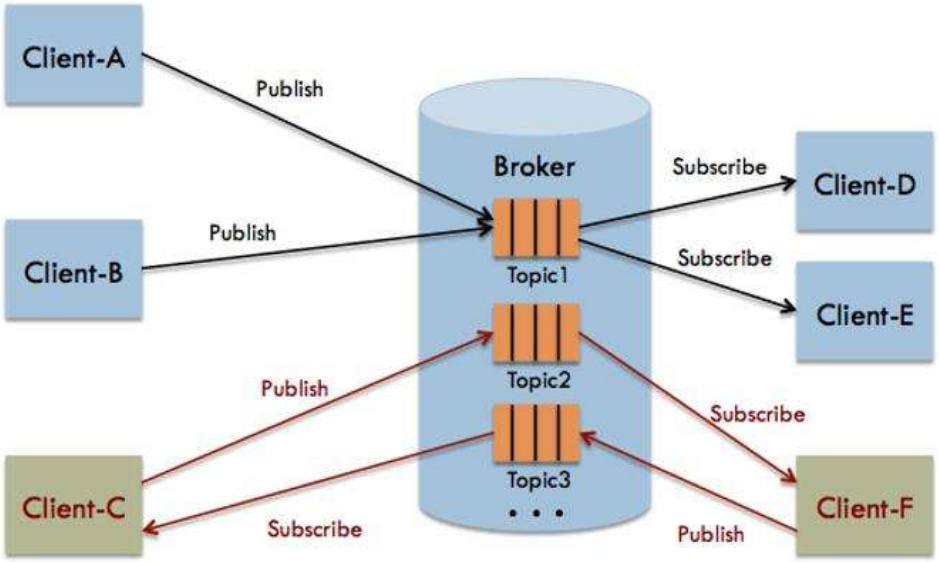
HTTP umumnya digunakan dalam susunan arsitektur *client-server* sebagai protokol komunikasi berbasis *request-response*. Baik *client* maupun *server* mengenali suatu data atau *resource* yang tersimpan melalui *uniform resource identifier* atau URI. *Client* dapat mengakses *resource* tersebut dengan mengirimkan *HTTP Request* dengan *method* tertentu untuk memberitahukan kepada *server resource* apa yang ingin dibaca atau dimanipulasi. Pihak *server* akan melakukan aksi sesuai dengan *HTTP Request Method* yang dikirimkan *client* ke *endpoint* bila aksi tersebut dapat dilakukan oleh *server*.

2.5.4. MQTT

Message Queuing Telemetry Transport atau dikenal dengan istilah MQTT merupakan suatu protokol *messaging* berbasis TCP/IP yang menggunakan konsep *client-server publish-subscribe* yang ringan dan sederhana sehingga cocok digunakan dalam *use case* seperti *machine-to-machine communication* atau *IoT System* yang membutuhkan protokol komunikasi yang ringan serta *bandwidth efficient*. MQTT juga memiliki banyak kapabilitas yang menjadi alasan utama tingginya penggunaan MQTT dalam *IoT System,* antara lain kemampuan untuk mengirimkan pesan *one-to-many* dengan ringan dan tingkat *reliability* yang dapat diatur serta fleksibilitas dalam hal jenis pesan yang dikirimkan. Dari segi performa dan fitur, terdapat protokol *messaging* lain yang memiliki performa dan fitur yang lebih lengkap yakni AMQP. Hal itu membuat AMQP cocok untuk diimplementasikan bila dibutuhkan protokol *messaging* yang kaya akan fitur serta digunakan untuk aplikasi yang kompleks. Akan tetapi, untuk kondisi jaringan yang kurang stabil dan *bandwidth* terbatas, protokol MQTT lebih cocok untuk digunakan dibandingkan dengan AMQP.

MQTT bekerja dengan konsep *publish-subscribe* pada *topic*. Artinya, perangkat-perangkat yang menggunakan MQTT dan ingin

berkomunikasi dengan perangkat-perangkat lainnya perlu melakukan *listen* atau *subscribe* pada URL tertentu yang dikenal dengan istilah *topic*. *Topic* melambangkan suatu URL yang berperan dalam mengelompokkan pesan-pesan yang dikomunikasikan antar perangkat. Perangkat-perangkat yang melakukan *subscribe* pada *topic* akan menerima pesan-pesan yang dikirimkan pada *topic* tersebut. Perangkat juga dapat melakukan *publish* pada *topic* untuk mengirim pesan-pesan untuk perangkat yang sedang *subscribe* pada *topic* tersebut. Semua perilaku *subscribe*, *publish*, *topic handling*, dan fitur-fitur lainnya seperti *last will* dan *QoS* akan ditangani oleh suatu *broker*, yang di sini berperan sebagai *server* dalam komunikasi melalui MQTT. Gambar 2.13 mendeskripsikan penjelasan pada paragraf berikut dalam bentuk skema sederhana.



Gambar 2.13 Skema Mekanisme Kerja Protokol MQTT