BAB III

ANALISIS DAN PERANCANGAN SISTEM

Bab ini akan membahas berbagai aspek mengenai rancangan sistem, dimulai dari penjelasan secara umum berupa spesifikasi dan kemampuan dari sistem yang diimplementasikan, skema arsitektur yang digunakan beserta rancangan umum dari sistem, serta rancangan dari modul atau bagian dari sistem tersebut. Rancangan yang dibahas di sini hanya berupa rancangan umum yang belum terikat dengan perangkat ataupun teknologi tertentu supaya rancangan ini dapat dijadikan acuan untuk implementasi atau perbaikan pada penelitian-penelitian ke depannya. Pemilihan teknologi secara khusus dari rancangan yang dibahas pada bagian ini akan dirinci kembali di Bab IV dalam bentuk implementasi dari sistem yang digunakan untuk penelitian ini.

3.1. Fitur dan Spesifikasi Penelitian

Fitur dan spesifikasi dari sistem yang diimplementasikan dalam penelitian ini antara lain:

1) Sistem ini akan memungkinkan *IoT Engineer* untuk melakukan *firmware update* kepada beberapa perangkat sekaligus dengan jenis sama dengan cara *over-the-air*. Perangkat dengan jenis yang sama yang dimaksud adalah perangkat yang memiliki jenis *microchip* dan *development board* yang sama.
2) Sistem ini akan mendukung *development board* berupa *NodeMCU DevKit* berbasis *ESP8266* serta *Raspberry Pi Single-Board Computer* terutama *Raspberry Pi 3 Model B+*.
3) Pemilihan perangkat tujuan akan dilakukan menggunakan suatu *identifier* yang tersimpan dan dapat diakses oleh *server*.

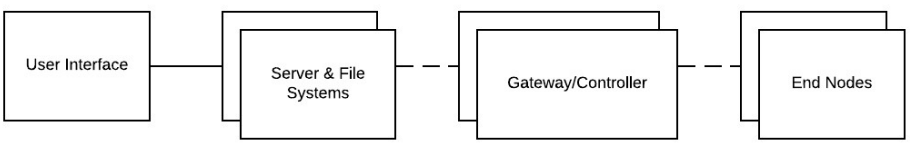
4) Ukuran dari *updated firmware/code* yang dapat dikirimkan serta kesuksesan dari proses *firmware update* bergantung pada *free space memory* dari perangkat tujuan.
5) Protokol, teknologi, dan perangkat yang digunakan dalam sistem ini perlu memiliki kemampuan untuk melakukan *multicast* untuk mengirimkan data maupun pesan.

3.2. Rancangan Sistem

Rancangan sistem akan terdiri dari berbagai komponen yang saling terpisah (*decoupled*) satu sama lainnya dengan antar muka yang didefinisikan dengan kriteria yang umum untuk mendukung modularitas dan skalabilitas. Komponen-komponen tersebut antara lain *user interface, server* dan *file system*, *gateway node*, dan *end node*.

Arsitektur dari sistem akan menggunakan arsitektur *hierarchical* sesuai dengan pertimbangan yang sudah disebutkan pada Bagian 2.2. Rancangan sistem dengan arsitektur ini dibuat untuk tetap membuka kemungkinan untuk implementasi yang fleksibel namun tetap dapat berjalan dengan baik.

Rancangan sistem secara keseluruhan dan hubungan antar komponen digambarkan pada Gambar 3.1. Kotak yang bertumpuk melambangkan bahwa komponen tersebut dapat dirancang sedemikian rupa untuk bisa mendukung proses *scale out.* Koneksi antara komponen *server* dan *file system*, *gateway node*, dan *end node* (ditandai dengan garis putus-putus) melambangkan koneksi yang mendukung komunikasi *multicast*.



Gambar 3.1 *High Level Architecture Diagram*

3.3. Rancangan Modul

Berdasarkan *high level architecture diagram* yang ditunjukkan Gambar 3.1, komponen-komponen dari arsitektur tersebut akan dirinci dari segi kriteria perangkat keras dan kondisi umum yang dipertimbangkan untuk pemilihan perangkat. Untuk penjelasan detail dari segi perangkat lunak maupun pemilihan perangkat keras akan dibahas di Bab IV.

3.3.1. User Interface

Komponen ini digambarkan sebagai suatu antar muka yang dapat digunakan oleh pengguna sistem ini, misalnya *IoT Developer* atau *Engineer* yang ingin melakukan *firmware update* ke perangkat-perangkatnya secara *over-the-air*. Pengguna dapat memberikan masukan berupa kode yang ingin didistribusikan, serta identitas dari perangkat atau kumpulan perangkat (*cluster*) yang ingin diperbarui kodenya. Sebaiknya pengaturan soal pengelompokkan perangkat juga dapat diatur oleh pengguna melalui komponen ini.

Cara pengguna memberikan masukan berupa kode dapat dilakukan dengan berbagai cara, selama masukan tersebut nantinya tersimpan dalam *file system* yang dapat diakses oleh *server*, Antar muka yang nyaman dan mudah digunakan dapat menjadi nilai tambah, tetapi tidak menjadi keharusan. Beberapa contoh implementasi yang dapat diwujudkan untuk komponen ini antara lain dalam bentuk *CLI Application, GUI Application, Web Application*, atau dengan mengintegrasikan *version control system* seperti Git atau Mercurial untuk melakukan pembaruan secara berkala dengan bantuan *shell script* atau *batch file*.

3.3.2. Server dan File System

Komponen *server* dan *file system* di sini menjalankan dua peran yang berbeda. *File system* di sini hanya berupa sebuah istilah yang mewakili

media penyimpanan data, sehingga istilah *database* juga dapat digunakan secara bergantian. *Server* berperan dalam menangani masukan-masukan dari pengguna dan berinteraksi dengan *file system* untuk menyimpan masukan pengguna, menyimpan informasi-informasi yang penting untuk jalannya sistem seperti identitas dari pengguna (bila ada), identitas dari perangkat, dan pengelompokkan perangkat-perangkat (*clustering*) atau informasi soal komponen *gateway/controller*. Selain itu, *server* juga membaca data yang tersimpan berupa *firmware/code update* untuk selanjutnya didistribusikan ke *gateway/controller*.

Komponen ini dapat diimplementasikan secara tunggal, atau secara terdistribusi. Implementasi secara distribusi tentunya memerlukan perancangan yang lebih kompleks untuk memastikan komunikasi antar komponen dapat memberikan informasi yang konsisten dan terbaru, tetapi hal tersebut tidak dibahas lebih dalam di penelitian ini. Antar muka yang menghadap ke komponen *user interface* hendaknya memiliki implementasi yang sesuai dengan komponen tersebut untuk dapat berkomunikasi dengan pengguna. Antar muka yang menghadap ke komponen *gateway/controller* sebaiknya dilakukan melalui jalur komunikasi yang dapat mendukung persyaratan dari *use case* yang ingin diimplementasikan, dan tentunya harus mampu melakukan komunikasi secara *multicast*.

3.3.3. Gateway/Controller

Komponen *gateway/controller* adalah komponen yang menjadi perantara dari komunikasi antar *server* dengan *end nodes*. Untuk skala kecil, komponen ini bersifat opsional, akan tetapi untuk *deployment* dalam skala yang besar, keberadaan komponen ini akan membantu meringankan beban dari komponen *server*, dan tergantung dari desain dan rancangan sistem spesifik, dapat mempercepat proses distribusi data maupun komunikasi ke *end nodes*.

Mempertimbangkan kemampuan komunikasi dari *end nodes*, komponen ini dapat ditempatkan (secara fisik) di lokasi yang dekat dengan *end nodes* untuk mendukung komunikasi jarak dekat, atau dapat ditempatkan di dekat *server* bila *end nodes* memungkinkan komunikasi jarak jauh. Komponen ini juga dapat menjembatani perangkat-perangkat yang berkomunikasi dengan cara yang berbeda, akan tetapi hal tersebut bersifat opsional dan tergantung pada *use case*. Tentunya, komponen ini harus dapat mendukung protokol komunikasi yang dijalankan baik dari sisi *server* maupun sisi *end nodes*, dan harus dapat mendukung komunikasi *multicast* dari dan ke kedua sisi.

3.3.4. End Nodes

Komponen *end nodes* merupakan komponen yang menjadi target dari *over-the-air update* yang akan dilakukan. Tentunya komponen ini dapat berjumlah lebih dari satu, dan setiap *end nodes* harus memiliki suatu antar muka yang dapat digunakan untuk berkomunikasi. *End nodes* di sini tidak membatasi perangkat untuk memenuhi semua persyaratan dalam satu perangkat, sebuah *end node* juga dapat terdiri dari beberapa perangkat keras khusus yang masing-masing memiliki fungsi spesifik, akan tetapi keseluruhan dari perangkat-perangkat tersebut akan dianggap sebagai satu entitas.

Sebuah *end node* yang ingin dikelola *firmware*-nya melalui *over-the-air* update harus memenuhi beberapa syarat sebagai berikut:

1) *End node* harus memiliki *handler* yang dapat berkomunikasi dengan komponen *gateway/controller* serta dapat mengeksekusi langkah-langkah yang diperlukan untuk melakukan *firmware update*.
2) *Handler* dari *end node* tersebut harus dapat mengakses (membaca dan menulis) program atau *flash memory* sebagai tempat *application code* disimpan.

Poin pertama akan memerlukan implementasi yang berbeda-beda tergantung pada jenis perangkat dan cara kerja dari perangkat tersebut, akan tetapi *handler*-nya dapat dirancang untuk memberikan gambaran umum mengenai langkah-langkah apa yang perlu dilakukan untuk mengeksekusi *firmware update*. Kemampuan komunikasi dengan *gateway/controller* tentunya memerlukan antar muka yang memiliki kemampuan komunikasi *multicast*. Secara umum, cara perangkat mengeksekusi *firmware update* dibedakan menjadi 2, yaitu *hardware bootloader* dan *software bootloader.*

A. *Hardware Bootloader*

Hardware bootloader yang dimaksud di sini adalah suatu perangkat keras tambahan yang berperan sebagai *bootloader* bagi suatu *end node*. *Hardware bootloader* umumnya berinteraksi secara fisik melalui *hardware pins* untuk mengelola alur kerja perangkat dalam menjalankan *code logic* ataupun *firmware update*, misalnya *flash write, reboot*, dan lain-lain.

B. *Software Bootloader*

Software bootloader yang dimaksud di sini merupakan suatu perangkat lunak berupa *code logic* yang memiliki peran yang sama seperti *hardware bootloader*, yaitu mengelola alur dari perangkat dalam menjalankan *code logic* ataupun *firmware update*. Yang membedakan adalah "lokasi fisik" dari kedua *bootloader*. *Hardware bootloader* umumnya berupa *code logic* yang terletak pada perangkat khusus yang terpisah dari perangkat yang menjalankan fungsionalitas dari aplikasi (misal sensor atau aktuator), sedangkan *software bootloader* umumnya adalah berupa *code logic* khusus yang terletak pada *program* atau *flash memory* yang sama dengan *application code logic*. Dengan kata lain, *software bootloader* dapat dianggap sebagai sebuah *virtual hardware bootloader*.

Dari definisi di atas, pendekatan untuk implementasi *software bootloader* akan menjadi lebih fleksibel dikarenakan konsep yang sama dapat dengan mudah dimigrasikan seandainya *end node* menggunakan *hardware bootloader*, yang berbeda hanyalah lokasi fisiknya. Pendekatan ini akan memberikan gambaran untuk implementasi dari *handler* yang dimaksud pada poin pertama, sehingga nantinya *handler* akan berinteraksi dengan perangkat melalui *internal API* dan melakukan proses-proses seperti menerima *firmware/code update*, menyimpan *update* tersebut ke dalam *flash memory* (caranya akan menyesuaikan dengan masing-masing perangkat), kemudian melakukan *reboot* atau proses lainnya untuk menjalankan kode yang sudah diperbarui tersebut.

Mengkaji lebih dalam mengenai *software bootloader* (dan terkait pula dengan poin kedua dari kriteria *end node*), langkah yang diambil dalam menjalankan *firmware update* untuk *end node* akan berbeda bergantung pada apakah *end node* tersebut bersifat *OS-based* atau *bare metal*. *End node* yang bersifat *OS-based* (contohnya *Raspberry Pi* pada umumnya) akan menjalankan *application code logic* sebagai sebuah proses di atas sebuah sistem operasi atau sejenisnya, dan umumnya akan memiliki akses yang lebih mudah untuk mengelola jalannya proses tersebut. *End node* yang bersifat *bare metal* di sini merupakan *end node* yang tidak memiliki sistem operasi yang mengelola jalannya proses, sehingga dapat dikatakan ketika perangkat menyala, *bootloader* akan langsung menginstruksikan untuk menjalankan *application code logic* di atas *bare metal*.

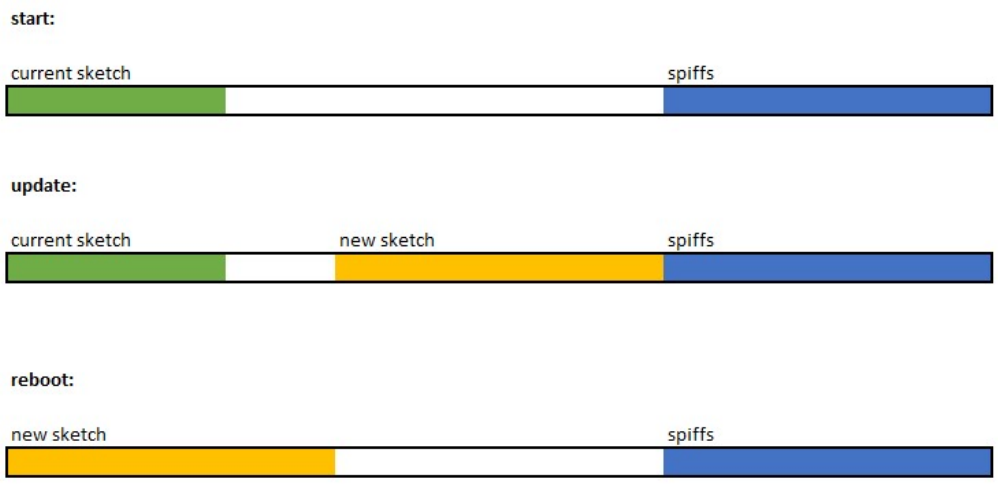
Pendekatan yang akan dilakukan dalam penelitian ini akan lebih memfokuskan ke *end node* yang bersifat *bare metal*, dikarenakan keterbatasan yang membuat rancangannya cenderung lebih rumit, dan rancangan untuk *end node bare metal* akan dapat dijalankan dengan mudah di *end node OS-based*. Pendekatan ini akan membuat deskripsi dan arah implementasi menjadi lebih umum dan fleksibel.

Dalam poin kedua dari kriteria *end node*, *handler* harus dapat membaca dan menulis *program* atau *flash memory* tempat *application code* disimpan. Kriteria tersebut juga akan berpengaruh pada cara *handler* menangani *firmware update*. Penanganan *firmware update* dengan memperhatikan *reliability* dan kapasitas *flash memory* secara umum terbagi menjadi 2, yaitu *atomic* dan *non-atomic firmware update*:

A. *Atomic Firmware Update*

Atomic firmware update merujuk pada proses *firmware update* yang *reliable* namun membutuhkan *flash memory* yang cukup besar. Karakteristik tersebut dicapai dengan menyimpan sementara *firmware update* di *free flash memory* hingga *update* diterima secara utuh dan benar. Setelah *update* diterima dan diverifikasi kebenarannya, baru dilakukan proses *booting* menggunakan *updated code* yang baru dan menghapus *code* yang lama. Gambar 3.2 menunjukkan susunan dari *flash memory* selama proses *firmware update* untuk *microchip ESP8266*.

Prosedur ini akan menjamin keberlangsungan dari perangkat seandainya terjadi masalah saat proses penerimaan *firmware update*, karena masih ada *code logic* yang lama untuk menjadi *backup*. Akan tetapi, prosedur ini memerlukan *flash memory* dengan ukuran kurang lebih 2 kali dari ukuran rata-rata *application code*, belum termasuk dengan pembagian *flash memory* untuk keperluan-keperluan lainnya. Hal ini membuat *firmware update* dengan cara ini memerlukan *flash memory* dengan ukuran lebih dari biasanya, tetapi dengan *trade-off reliability* yang tinggi.



Gambar 3.2 *Flash memory layout* selama proses *atomic firmware update*

B. *Non-atomic Firmware Update*

Berlawanan dengan cara yang *atomic*, cara *non-atomic* memerlukan ukuran *flash memory* yang tidak terlalu dibatasi kriteria di atas, tetapi dengan *trade-off reliability* yang berpotensi menyebabkan perangkat tidak dapat berjalan dengan baik (dan harus diperbarui *firmware*-nya secara fisik). Akan tetapi, bila berhasil dijalankan, prosedur *firmware update* ini akan memberikan keleluasaan lebih bagi pengguna dari segi ukuran kode yang dapat dikirimkan.

Dengan mempertimbangkan *use case* penggunaan *IoT System* di luar ruangan dan tempat terpencil (yang mempersulit pendekatan *non-atomic* ketika terjadi kegagalan), penelitian ini akan mengambil pendekatan *atomic firmware update*. Pendekatan ini diambil dengan pertimbangan lainnya pula yakni optimisasi-optimisasi yang dapat dilakukan di tahap-tahap atau komponen lainnya untuk menyesuaikan ukuran *firmware update* supaya dapat dilakukan pada pendekatan *atomic firmware update* dengan lancar.

3.4. Cara Kerja Sistem

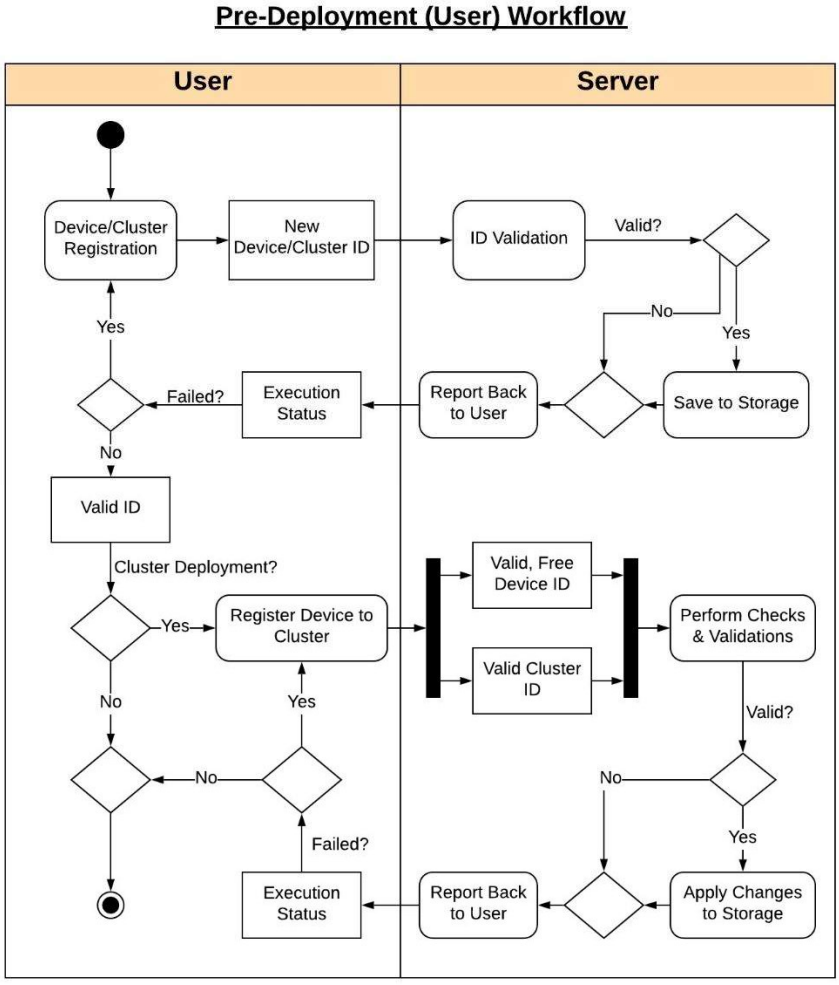
Cara kerja dari sistem yang dirancang pada bagian ini akan dijelaskan terkait dengan interaksi yang dilakukan oleh pengguna, serta proses-proses yang dikerjakan komponen-komponen yang sudah dijelaskan sebelumnya. Kasus interaksi pengguna akan dibagi menjadi 3, yakni *pre-deployment*, *deployment*, dan *post-deployment*. Alur dan proses kerja akan digambarkan menggunakan *activity diagram* dengan *swimlanes* yang memberikan pemahaman mengenai proses apa yang dilakukan oleh komponen apa dalam sistem. Untuk penyederhanaan dan pemahaman, istilah "inisialisasi" yang akan banyak digunakan dalam bagian ini mengacu pada kondisi ketika perangkat akan menjalankan *code logic*, tidak harus untuk yang pertama kalinya.

3.4.1. Pre-Deployment

Tahap *pre-deployment* akan mencakup kurun waktu dimana pengguna masih berinteraksi dengan komponen *server* melalui *user interface* untuk melakukan hal-hal seperti proses pembuatan ID baru (ID *cluster* maupun *device*). ID yang dibuat ini merupakan *logical ID* yang ditentukan oleh pengguna, dan tidak terikat dengan faktor fisik apapun dari *end node* yang ingin di-*deploy*. Alasan penggunaan *logical ID* ini adalah untuk memberikan kemudahan bagi pengguna untuk mengelompokkan perangkat-perangkat, serta menyediakan fleksibilitas dengan tidak terikat pada faktor fisik yakni perangkat *end node*. Tentunya, dapat diberikan batasan-batasan dari sisi *server* maupun *user interface* untuk membatasi masukan ID dari pengguna.

Dalam tahap ini, langkah-langkah yang terjadi akan bergantung pada keinginan pengguna untuk melakukan *deployment* dari *end device,* apakah ingin di-*deploy* secara *standalone* tanpa *cluster management* atau ingin di-*deploy* sebagai bagian dari suatu *cluster*. Gambar 3.3 menggambarkan alur yang lebih lengkap, sedangkan alur secara garis besar untuk tahap ini adalah sebagai berikut:

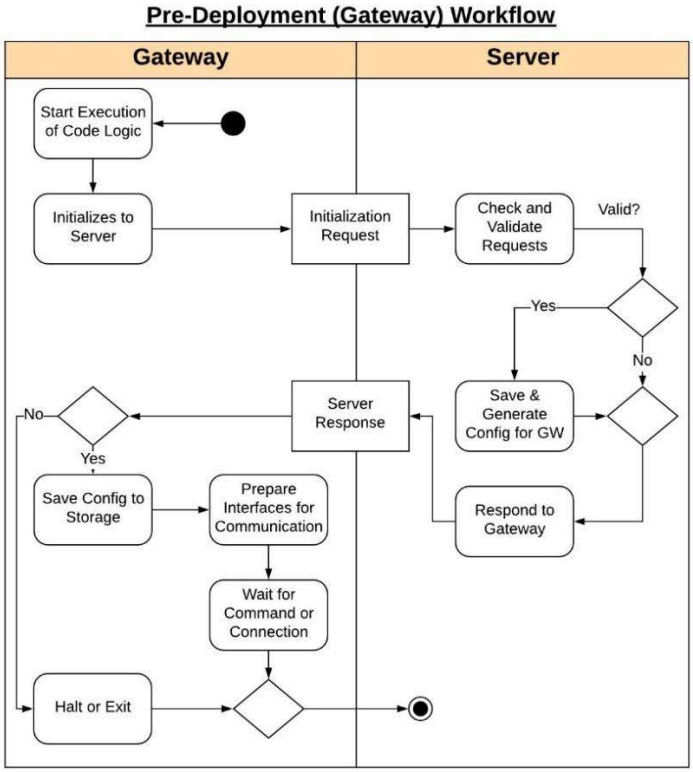
1) Pengguna membuat ID baru untuk perangkat dengan mengirimkan masukkan ke komponen *server*.
2) Komponen *server* melakukan validasi dan pemeriksaan terhadap masukan ID dari pengguna.
3) Komponen *server* menginformasikan pengguna mengenai status dan validitas ID. Bila valid, maka ID tersebut akan disimpan oleh komponen *server* dan dapat digunakan oleh pengguna di *end device* yang ingin di-*deploy*.
4) Pengguna dapat kembali ke langkah 1 untuk membuat ID baru, atau melakukan *register* untuk mendaftarkan *membership* dari ID *device* ke ID *cluster* yang sudah dibuat sebelumnya.



Gambar 3.3 *Activity Diagram* alur kerja *Pre-Deployment Phase (User)*

Selain interaksi dengan pengguna, tahap *pre-deployment* juga melingkupi proses inisialisasi yang dijalankan oleh *gateway/controller*. Proses inisialisasi ini penting untuk *gateway* agar dapat berkomunikasi dengan *server*, serta untuk mendapatkan informasi-informasi yang akan diberikan kepada *end device* dalam tahap *deployment*. Gambar 3.4 mendeksripsikan alur ini secara lebih mendetail, sedangkan secara garis besar alur untuk tahap ini adalah sebagai berikut:

1) *Gateway* menjalankan *code logic* dan melakukan inisialisasi ke *server*.
2) *Server* menerima inisialisasi tersebut dan menyimpan informasi tersebut, kemudian merespon kepada *gateway* dengan konfigurasi.
3) *Gateway* menerima respon dari *server* dan menyimpan *informasi* tersebut sebagai konfigurasi.
4) *Gateway* menyiapkan antar muka untuk berkomunikasi lebih lanjut dengan *server*, serta untuk berkomunikasi dengan *end device*.



Gambar 3.4 *Activity Diagram* alur kerja *Pre-Deployment Phase (Gateway)*

Beberapa poin tambahan yang perlu disinggung adalah cara *server* untuk melakukan validasi dan pemeriksaan akan bergantung pada implementasi yang dilakukan untuk menyesuaikan dengan kebutuhan. Selain itu, perilaku dari setiap komponen dalam menangani kegagalan atau *error* juga akan bergantung pada implementasi. Beberapa penanganan yang dapat dilakukan antara lain menghentikan eksekusi program (*halt*), melanjutkan fungsi utama program seperti biasa (*continue*), atau mengulangi proses yang gagal (*retry*) tersebut untuk *reliability* tambahan.

3.4.2. Deployment

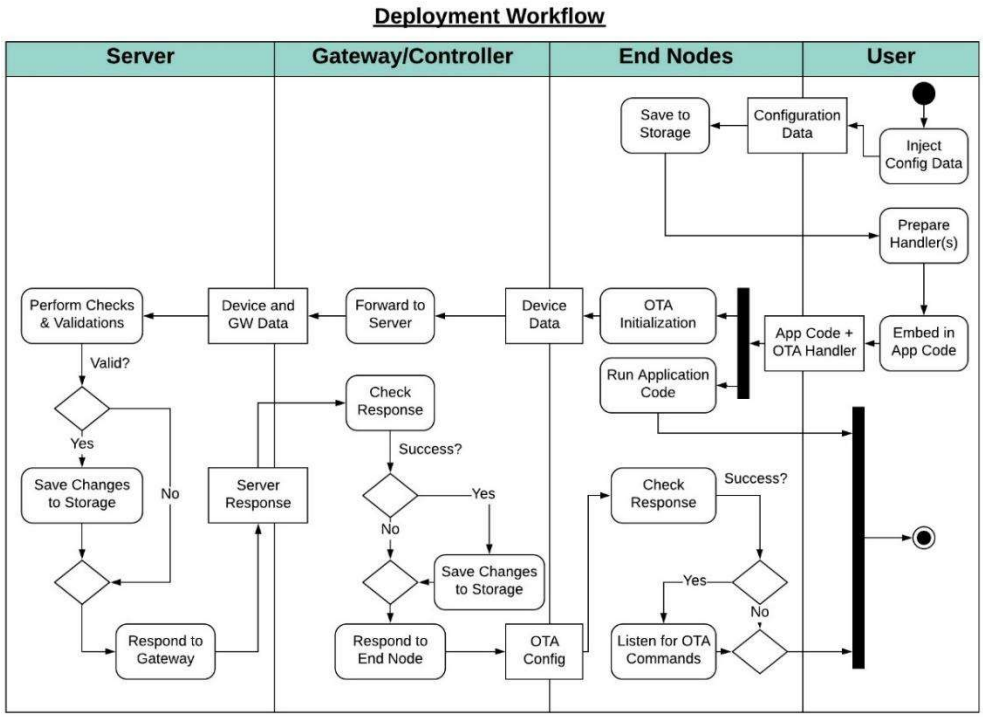
Tahap *deployment* mencakup kegiatan-kegiatan mulai dari interaksi langsung antara pengguna dan *end node* untuk persiapan awal, *end node* di-*flash* dengan *over-the-air update handler code* dan *application code*, serta hingga *end node* berinteraksi dengan *gateway* (yang kemudian berinteraksi dengan *server*) untuk melakukan inisialisasi awal agar *end node* dapat dikelola menggunakan *over-the-air update* melalui sistem. Meski tidak dijelaskan dalam langkah-langkah, ada baiknya sebelum proses *deployment* (terutama ketika pengguna mempersiapkan *end node*) tetap dilakukan proses *testing* untuk memastikan *end node* berhasil berjalan dengan baik dan dapat melakukan inisialisasi.

Dalam tahap *deployment*, alurnya secara mendetail akan dipengaruhi oleh implementasi yang dilakukan dan protokol yang digunakan, sehingga alur yang dijelaskan pada Gambar 3.5 hanya akan berupa gambaran umum dari proses yang terjadi dalam komponen-komponen yang terlibat. Langkah-langkah yang terjadi adalah:

1) Pengguna menyimpan informasi konfigurasi awal yang dibutuhkan *end node* untuk bisa melakukan inisialisasi, misalnya ID *device* dan *cluster*.
2) Pengguna menyimpan (*flash* atau *file transfer*) *over-the-air handler code* beserta dengan *application code* ke *end node*.

3) *Over-the-air handler* melakukan inisialisasi ke *gateway/controller.*
4) *Gateway/controller* yang menerima inisialisasi *end node* tersebut dan meneruskannya ke *server*.
5) *Server* memproses inisialisasi *end node* dari *gateway* dan merespon setelah melakukan validasi dan pemeriksaan.
6) *Gateway/controller* menerima respon dari *server* dan bila sukses, memberikan informasi yang diperlukan *end node* untuk dapat menerima *over-the-air update.*
7) Berdasarkan respon dari *gateway/controller*, *end node* akan mempersiapkan antar muka untuk menerima perintah *over-the-air update* bila inisialisasi berhasil.

Cara setiap komponen untuk melakukan pemeriksaan dan validasi, serta menangani *error* akan bergantung pada implementasi yang dilakukan. Beberapa penanganan sudah dijelaskan di akhir Bagian 3.4.1, dan Gambar 3.5 menunjukkan penanganan *error* berupa *continue* di bagian *end nodes*.

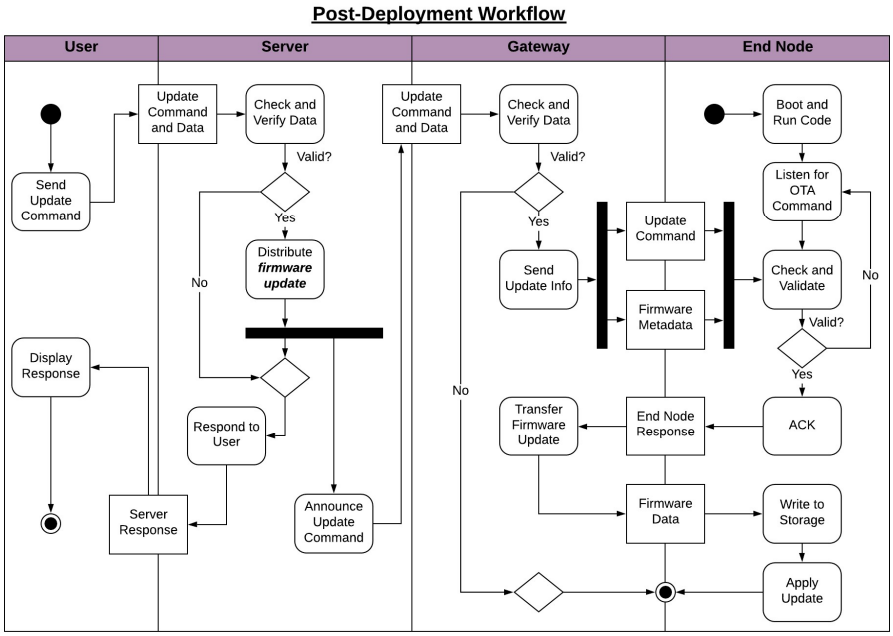


Gambar 3.5 *Activity Diagram* alur kerja *Deployment Phase*

3.4.3. Post-Deployment

Tahap *post-deployment* mencakup *over-the-air update* yang dilakukan oleh pengguna melalui interaksi dengan *server*. Tahap ini menganggap semua *code logic* yang berjalan di setiap komponen, termasuk *code logic* yang akan dikirim melalui *over-the-air update* berjalan tanpa *error*, dan tidak mencakup proses *software* atau *code development*. Gambar 3.6 menggambarkan lebih detail mengenai proses yang terjadi dalam tahap ini. Secara umum, langkah-langkah yang diambil adalah:

1) Pengguna mengirimkan perintah untuk melakukan *firmware update* ke *server* beserta dengan data dan informasi yang dibutuhkan.
2) *Server* menerima perintah tersebut, melakukan validasi, kemudian mendistribusikan *firmware* ke *gateway* yang memerlukan.
3) Server meminta gateway untuk meneruskan firmware ke end node.
4) *Gateway* yang menerima perintah dan *firmware update* tersebut mengirimkan perintah ke *end node*.
5) *End node* menerima perintah dan *firmware*, lalu melakukan *update*.



Gambar 3.6 *Activity Diagram* alur kerja *Post-Deployment Phase*