MSc Internship Report

3T B.V, Institutenweg 1, Enschede

FOTA Update Mechanism for Embedded Devices

Author:
Prawin Kumar Srinivasa Kumar
s2731991

Internship period:
05.02.2024 - 13.05.2024

Internship supervisors:
ir. Rudie Alderden
ir. Wout Laren

UT supervisor:
dr.ir.Kuan Chen

May 2024

UNIVERSITY OF TWENTE.

3T

Contents

1 Introduction

This internship explores the feasibility of performing a large-scale rollout of firmware updates to a fleet of devices. The existing firmware update mechanism is manually flashing the new firmware binary onto the device via USB through the Debugger or flash tool. This mechanism is quite a universal problem, especially in the realm of IoT, where many devices scattered across the globe have to undergo firmware updates with crucial bug fixes or patches for security. Although this is the most secure option available, it is pretty tedious when performing the same task on many devices. Therefore, the organization has reached a consensus on identifying other alternative solutions to help alleviate this pain point and save critical human resources through some form of automation. From the study, it was determined that there are quite a few solutions to address the issue. However, there is no one fit for all. Based on the requirements identified for the problem, it was clear that a combination of solutions must be applied to create a comprehensive solution that addresses this issue.

2 Problem Statement:

The problem statement provided by the organization:

1. Research standard solutions that are available to upgrade/manage the firmware on multiple devices as alternatives to the USB flash option.
2. Identify one to be used for this assignment. The assumption is that a server that works out of the box is available.
3. Implement a proof of concept client for a microcontroller that demonstrates the communication with the server. It should demonstrate the status update of the device (which firmware version is loaded, etc.) and download an image. It is not necessary to write the image to flash.
4. Optional tasks depending on the time remaining: program image to flash and boot it, encryption of the communication, authentication

3 System Design Requirements:

Requirements for the solution:

1. It should be possible to install it locally. It should not be a cloud-only solution.
2. Solution to be robust to update multiple devices
3. Have a containerized environment for the server for easy setup
4. It should be possible to implement the communication protocol on a microcontroller. Microcontrollers don't need to be supported out of the box, but it should be feasible to implement it on a microcontroller. The main concern is the amount of resources required. For example, it is not possible to first load the image completely in RAM and then write it to flash

Other Identified Requirements for the Server

1. Server has a UX to view information about the connected devices and to upload new firmware images
2. It should be possible to encrypt the communication with the server.
3. Authentication of both the server and the clients should be supported

4 Design Space Exploration

Careful consideration of various factors is crucial in designing a Firmware Over-The-Air (FOTA) update solution for an organization seeking to manage firmware updates across multiple devices. The scoring system outlined in the decision matrix table aims to evaluate different server alternatives based on essential criteria tailored to meet the organization's requirements. These factors serve as a tool in determining the most suitable solution for the company's FOTA application. The decision matrix assesses alternatives such as Hawkbit, SWUpdate, and Mender across crucial categories, including Business Cost, Central Management, Deployability, Development & Setup, and Usability. Each alternative is rated on a scale from one to five, with five indicating the highest level of suitability. Business Cost evaluates the financial implications of adopting each solution, considering licensing fees and operational expenses. Central Management assesses the solution's ability to centrally administer firmware updates across a fleet of devices, an essential requirement for efficient management. Deployability examines the solution's ease of deployment and integration within the company's infrastructure. Development & setup consider the complexity of configuring and implementing the solution, ensuring that it aligns with the organization's technical capabilities and resources. Usability focuses on the user experience aspects of the solution, including the availability of user-friendly interfaces for device management and firmware image uploads. This factor is beneficial for centrally managing fleet updates through a dashboard. Considering the organization's requirements, such as the need for a locally installable solution, robustness in updating multiple devices, containerized server environment for ease of setup, and compatibility with microcontroller communication protocols, the decision matrix provides a structured approach to evaluate and compare the server alternatives. From the decision matrix, the Hawkbit framework emerges as the clear winner. Further reasoning for the scoring is discussed in detail in the following sections.

Alternatives / Factors	**Hawkbit**	SWUpdate	Mender
Business Cost	5	5	1
Central Management	5	5	3
Deployability	5	3	5
Development & Setup	5	3	1
Usability	5	3	2

Table 1: Decision Matrix for Server

4.0.1 SWUpdate:

The SWUpdate framework [2] stands as a formidable Linux update agent meticulously engineered to ensure the secure and efficient updating of embedded Linux devices. Its core objective is to furnish users with a robust platform for executing updates seamlessly, locally or remotely, via physical connections or over-the-air (OTA) transmissions, with minimal integration overhead into embedded Linux projects utilizing Yocto, Buildroot, or Debian build systems. A pivotal feature of SWUpdate is its implementation of atomic updates, a mechanism designed to safeguard against device failure in the face of disruptions such as power outages or network failures. This framework ensures that updates are executed reliably without the risk of partial software installations inherent in conventional package-based approaches. SWUpdate further fortifies security measures by enforcing stringent package signing and verification protocols, thereby instilling trust in the integrity of the update process. Moreover, it allows users to enhance security by utilising RSA keys/certificates within their own Public Key Infrastructure (PKI) frameworks. Expanding upon its security architecture, SWUpdate incorporates a robust rollback mechanism seamlessly coordinated by the bootloader. This mechanism continuously monitors the behaviour of installed software, promptly reverting to previous firmware iterations should any anomalies be detected, thereby preserving operational stability. Furthermore, SWUpdate's extensibility is underscored by its support for Lua hooks, supplementing its versatile C framework and enabling updates for diverse software components across various device environments. SWUpdate supports custom configurations that could result in obtaining a minimal footprint and optimal performance for specific use cases. For instance, advanced techniques such as zero-copy loading, where images are directly loaded

into memory, ensure efficient memory utilization by avoiding the extra copy of the image on the ROM. Additionally, SWUpdate accommodates a broad spectrum of embedded storage formats, including NOR, NAND, eMMC, SPI Flash, and UBI volumes, ensuring compatibility with diverse hardware configurations. Notably, SWUpdate seamlessly integrates with fleet deployments through its compatibility with the Hawkbit server, offering a comprehensive solution for managing updates across distributed device networks. Crucially, SWUpdate remains committed to open-source principles, eliminating the need for a commercial license in real-time applications and ensuring accessibility and collaboration within the development community. The reliability and efficacy of the SWUpdate framework were validated through local testing, as detailed in [22], providing insights into its performance and functionality. The SWUpdate framework offers various features that are interesting for the FOTA mechanism, the majority of which are targeted for embedded Linux devices that are not in the scope of this assignment. Hence, this framework follows Hawkbit closely in the scoring system.

4.0.2 Mender:

The Mender framework [1] employs a central management system utilizing a simple server/client architecture to deliver secure and reliable software updates to many devices. This architecture facilitates seamless integration with the build system, enabling the generation of Mender artefacts directly from any local workstation with a Yocto project or from a Jenkins system. The generated artefacts are tailored to the target device and stored within the Mender Server. At the heart of the framework lies the Mender Server, orchestrating the deployment of artefacts to the respective Mender clients, whether through fleet rollouts or individual deployments. Mender clients are versatile, extending support to embedded Linux, various operating systems, and other microcontrollers. The framework offers two distinct update initiation modes: The managed mode, where the client requests the new firmware/update binary, and the standalone mode, allowing users to initiate updates, for instance, via USB. Security is paramount within the Mender framework, with provisions for extended security options, including secure gateways for devices operating on isolated networks. Furthermore, Mender supports artefacts in diverse formats and a wide range of devices and facilitates phased and dynamic rollouts based on groups, leveraging delta updates to minimize device downtime significantly. In the delta update mechanism, two copies of the firmware reside in memory, enabling the device to remain operational with the previous image while the new firmware is installed. Upon successful installation, the device undergoes a reset/reboot, ensuring the bootloader picks up the latest firmware. Thus, downtime is optimized for reboot duration, maintaining operational continuity during the update process. Despite its numerous advantages, including seamless integration with backend and cloud services, it's essential to note that Mender is not an open-source solution. Procurement of an enterprise plan is necessary based on specific requirements, as it involves management via a hosted Mender Server available on AWS in the US and EU regions. However, setting up example projects following the documented instructions and guidelines from [21] was intuitive and straightforward. The framework pricing and dependency on the cloud make this an unattractive alternative for this mechanism.

4.0.3 Hawkbit

The Eclipse hawkBit™ framework [5] represents a domain-independent backend solution tailored for facilitating software updates across a spectrum of devices, ranging from resource-constrained edge devices to more potent controllers and gateways integrated with IP-based networking infrastructure. The significance of software update capabilities in IoT cannot be overstated, as they are indispensable for fortifying IoT ecosystems against the evolving landscape of cybersecurity threats. Devices, once connected, are susceptible to a myriad of security vulnerabilities, making the absence of software/firmware updates a risky proposition. Failure to apply security patches/fundamental functional changes exposes devices to potential exploitation/buggy code being utilized on the field, endangering individual devices and entire IoT networks. The hawkBit framework addresses the ubiquitous requirement for software updates across diverse IoT scenarios. While software updates are necessary, the conventional approach of handling updates within individual IoT solutions or through standalone device management systems leads to redundant efforts and complex implementations. The hawkBit framework seeks to streamline and standardize the software update process, offering a robust and efficient solution tailored specifically for the IoT landscape. Central to its value proposition is its ability to ensure the safety and reliability of remote software update processes. By abstracting the complexities of software updates from individual IoT solutions, hawkBit minimizes duplication of effort and optimizes resource utilization. The framework provides a unified approach to software update management, facilitating seamless rollouts across diverse device architectures and connectivity protocols. It distinguishes itself from traditional device management systems by focusing on software updates as a standalone process, independent of specific application domains. While existing device management systems may offer rudimentary update functionalities, they often lack the scalability and flexibility required for large-scale IoT deployments. hawkBit, on the other hand, is purpose-built for efficiently managing software updates at the IoT scale, offering sophisticated features such as phased rollouts, error handling, progress monitoring and scheduled deployment. The framework's cloud-ready architecture ensures technical scalability, enabling the global management of millions of devices and terabytes of software updates. Functionally, hawkBit supports rollouts involving hundreds of thousands of individual devices, with robust reliability and fault tolerance mechanisms in place. Furthermore, hawkBit's integration flexibility allows seamless interoperability with existing device management systems and protocols, ensuring compatibility with diverse IoT ecosystems. Key Features and Capabilities hawkBit's comprehensive feature set encompasses every aspect of the software update lifecycle, from artefact creation to rollout management and monitoring, hence providing a sophisticated management interface that can be used to streamline the entire deployment process that can perform large-scale rollouts while also not compromising by integrating authorization primitives to enable secure communication and identity management, all of which are encapsulated in 1. Based on the points mentioned above, it is evident that this framework is the clear winner. However, some pitfalls were identified during the development and testing stage, which are discussed in section 10.

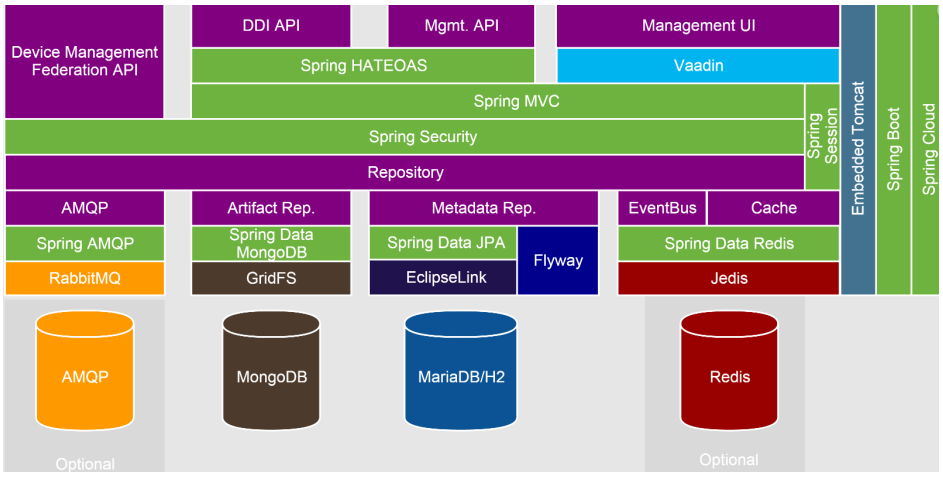


Figure 1: Architecture of hawkbit Framework [5]

4.1 Networking Libraries:

In this context, alternatives such as Mongoose, FreeRTOS-plus-TCP, and LwIP are evaluated across critical categories, including Business Cost, Porting Support, Deployability, Development & Setup, and Usability. Pertaining to the business cost for the organization, Mongoose receives a score of 1 in this category due to the library requiring a commercial license to be purchased to be used in commercial settings, While both FreeRTOS and LWIP have been awarded the highest score considering the open source license and the freedom to develop and deploy commercial applications using the same. Porting Support (H/W) assesses the level of support provided for hardware porting, with higher scores indicating better support. Mongoose and LwIP both receive high scores of 5 in this category, suggesting strong support for hardware porting since drivers and interface code for a wide range of hardware are readily available. Although the FreeRTOS-plus-TCP supports a wide range of microcontroller families, the network interface manager for the DUT was not readily available. In terms of deployability, Mongoose and LwIP both score high in this category, suggesting they are highly deployable solutions as specific examples for the integration of the DUT being made available. FreeRTOS-plus-TCP receives a slightly lower score, indicating potential challenges in deployment. Development & Setup assesses the complexity of configuring and implementing each networking library, with lower scores indicating simpler setup processes. While Mongoose and FreeRTOS-plus-TCP receive moderate scores in this category, LwIP scores the highest, suggesting it may offer the most straightforward setup process. Usability focuses on the user experience aspects of each networking library, including the availability of user-friendly interfaces and documentation. LwIP receives the highest score in this category as this is the most mature and predominantly used networking library by every device manufacturer, with extensive support for interfacing from both the community and the manufacturer forums, indicating superior usability compared to Mongoose and FreeRTOS-plus-TCP. Moreover, the FreeRTOS-plus-TCP library induces the dependency of the FreeRTOS kernel to be present as a pre-requisite for the latter.

Alternatives / Factors	Mongoose	FreeRTOS-plus-TCP	**LwIP**
Business Cost	1	5	5
Porting Support (H/W)	5	2	5
Deployability	5	3	4
Development & Setup	3	4	5
Usability	3	4	5

Table 2: Decision Matrix for Networking Library

4.1.1 Mongoose:

The Mongoose Networking library [23] presents a highly adaptable networking solution that seamlessly integrates into C/C++ projects with minimal effort. Covering a broad spectrum of communication protocols, including UDP, TCP, HTTP, MQTT, and WebSockets, this library is renowned for its flexibility and widespread acceptance, even being employed in missions beyond Earth's atmosphere. Compatibility extends across various architectures and device families, ensuring versatility and applicability across diverse hardware environments. One of the standout features of the Mongoose Networking library is its extensive extension capability. This empowers developers to leverage existing projects using FreeRTOS + lwIP stack or deploy it in bare-metal setups with its built-in network drivers and TCP/IP stack. This flexibility facilitates the incorporation of networking functionalities into various projects, ranging from embedded web servers to facilitating Over-The-Air (OTA) updates. A noteworthy aspect of the library is its comprehensive example and tutorial database, offering detailed insights into API usage across different hardware configurations and use-case scenarios. These resources aid in seamlessly integrating and utilising the library's features. However, it's important to acknowledge that while the library's feasibility has been tested on the Device Under Test (DUT), further development is required to fully support OTA updates for the DUT. This aspect of the project is currently undergoing continuous refinement and enhancement. It's also pertinent to mention that the adoption of this solution for commercial or industrial applications necessitates the procurement of a commercial license. This licensing requirement underscores the library's utilization in commercial applications to be used by the stakeholders, i.e. 3T.

4.1.2 FreeRTOS-plus-TCP:

FreeRTOS-Plus-TCP [9] stands as a versatile and robust open-source TCP/IP stack meticulously crafted for integration with FreeRTOS, offering a seamless network communication solution for MCU-based applications. The library prioritizes scalability and modularity, catering to a spectrum of hardware constraints, from low-power microcontrollers to high-performance microprocessors. This scalability extends to its feature set and RAM footprint, allowing developers to tailor the stack to their project requirements. FreeRTOS-Plus-TCP adheres to the widely adopted Berkeley sockets API, ensuring compatibility with existing network applications and easing the porting process. With support for IPv4 and IPv6, the stack facilitates interoperability across diverse network environments, accommodating legacy systems while embracing emerging networking standards. A distinguishing feature of FreeRTOS-Plus-TCP is its support for multiple network interfaces and endpoints, enabling concurrent utilization of Ethernet, Wi-Fi, and other interfaces within a single application. Furthermore, the library is equipped with full reentrancy and thread safety, fostering robustness in multi-threaded environments and allowing seamless integration into complex applications with multiple concurrent tasks. Essential networking protocols such as ARP, DHCP, DNS, and ICMP are included, along with optional services like static and dynamic IP address assignment, enriching the stack's capabilities and versatility. The library also offers an optional callback interface, providing greater flexibility in handling network events and customizing behaviour to suit specific application needs. The stack's optimized code size balances functionality and memory usage, ensuring suitability for resource-constrained devices without compromising performance. This library has been subjected to extensive testing to ensure its deployment in critical applications. However, this is a relatively new networking stack; while the community forum is quite helpful, it may not be possible to identify solutions for problems associated with porting for newer hardware. Furthermore, as discussed previously, this stack also induces the inherent dependency of utilising the FreeRTOS kernel. While this is not inherently an issue, if there is the decision to utilize a different RTOS like Zephyr OS this might become an issue.

4.1.3 LwIP

This library [19] is designed to focus on minimizing resource consumption while delivering a comprehensive TCP stack. Tailored for embedded systems with limited resources, it operates efficiently with as little as 10+ KB of available RAM and 40+ KB of ROM. The library boasts many supported protocols, including IP, UDP, TCP, ARP, PPPoS, DHCP, and DNS, making it a versatile choice for diverse networking tasks. It is equipped with specialized APIs for optimized performance and embraces the renowned Berkley Sockets (BSD) model for seamless integration. With additional features like IP forwarding, TCP congestion control, and robust error handling mechanisms, this mature library, with over two decades of industry presence, has garnered widespread adoption among leading chip manufacturers, offering comprehensive support and integration documentation for their hardware platforms.

Network Interface Management:
This library segment is pivotal for configuring the network interface, facilitating the connection and disconnection of the device from the network via device drivers for physical network hardware. In the client application, Ethernet input is utilized within a threaded environment. Configuration adjustments in the library enable routing all IP and ARP packets to the Ethernet input.

Address Resolution Protocol (ARP):
ARP is a link layer protocol that translates hardware addresses into network layer addresses. Its primary application is bridging Ethernet networks to the internet by facilitating translation between MAC and IP addresses. Furthermore, in this project, direct invocation of this protocol is unnecessary as the library handles all requisite calls in the background. However, configuration settings in the library's configuration file enable queue setup.

Dynamic Host Configuration Protocol (DHCP):
The DHCP protocol introduces an additional layer of flexibility by managing IP negotiation with the router within the local network. This dynamic allocation of IP addresses comes with a trade-off of increased resource utilization in the library, as DHCP operates over UDP, which also requires configuration

for the client application. LWIP offers an advantage in the networking stack by autonomously handling sudden changes in the physical network, thereby informing dependent protocols for appropriate action. PHY auto-negotiation is employed, where interrupts are passed to the tcpip thread, enabling necessary hardware register adjustments based on the current link status in this assignment.

Internet Protocol Version 4 (IPv4):

IPv4, the backbone of internet communication, employs a 32-bit address format represented in dot notation. It faces architectural differences between host and network, leading to varying byte orders: big-endian for network and little-endian for ARM processor-based hosts. The library efficiently manages these variations through tailored conversion APIs. In this project, IP assignment is dynamically obtained from the router via DHCP negotiation, ensuring seamless integration within the local network.

Internet Protocol Version 6 (IPv6):

IPv6,[3] an evolution over IPv4, introduces a 128-bit address space, effectively mitigating the addressing limitations encountered in IPv4. Its extensive address range, approximately $3.4 * 10^{38}$ devices, caters to the burgeoning demands of modern networking. Moreover, IPv6 fosters reliability and faster speeds through innovations like jumbograms, capable of handling packets of up to 2^{32} bytes. Despite its technological advancements, IPv6 adoption remains limited, primarily due to compatibility concerns with IPv4. One notable feature of IPv6 is its streamlined header architecture, which optimizes routing decisions, promoting faster data transmission. Additionally, IPv6 embraces stateless address configuration, eliminating the need for a DHCP server in the network. Hosts within the same network segment communicate using link-local addresses, facilitating seamless intercommunication. However, IPv6's extension capabilities are somewhat constrained by packet size limitations, unlike IPv4, which accommodates a maximum header size of 40 bytes. Despite these challenges, this protocol is promising for future networking endeavours, offering enhanced reliability, scalability, and security features. In this project, IPv4 is preferred over IPv6 due to its widespread adoption and compatibility with existing infrastructure.

Transport Layer Protocol:

At the core of network communication lies the transport layer protocol, which facilitates end-to-end connectivity and reliable data transmission. In the client application, TCP Streaming sockets are leveraged. This connection-oriented protocol is renowned for its robustness in ensuring ordered and error-free packet delivery. TCP's transmission reliability stems from its meticulous connection establishment process, typically initiated through a three-way handshake between the client and server. This handshake involves the exchange of control messages, including sequence numbers and checksums, to validate packet integrity and establish a synchronized transmission window. By closely monitoring acknowledgements and sequence numbers, TCP can detect and mitigate packet loss, ensuring data consistency between sender and receiver. One noteworthy aspect of TCP is its efficient handling of out-of-order packets. TCP promptly identifies and reorders packets through sequence and acknowledgement numbers to maintain transmission integrity. Additionally, TCP employs congestion control mechanisms to regulate data flow, preventing congestion and ensuring fair bandwidth allocation.

Configuring LWIP:

The LWIP library offers extensive configurability through its configuration file, Lwipopts.h, enabling precise tailoring to suit specific application requirements. This customization ensures that only necessary code is built, optimizing storage utilization on hardware with limited storage capacity. By default, the library includes essential components such as ARP, IP, TCP, UDP, RAW IP, and Stats. Notably, TCP constitutes a significant portion, representing over 60% of the library's footprint.

Memory Management in LWIP:

- Memory Pools: LWIP employs memory pools to expedite and efficiently manage memory allocation, affording flexibility in handling various memory requirements. A static chunk in the data section is allocated for LWIP, subdivided into distinct pools, each catering to specific purposes. Each pool is associated with a dedicated structure, determining the maximum number of concurrent connections it can accommodate. However, increasing this value escalates memory consumption. In scenarios like OTA updates, where the client communicates with a single server, deterministic behavior is ensured, mitigating fragmentation concerns and minimizing memory wastage.
- C Standard Library: While offering flexibility and compatibility with standard C functions used across the project, the C Standard Library introduces certain drawbacks. Dynamic memory allocation via malloc() may encounter failures due to inadequate contiguous memory availability, stemming from continuous allocation and deallocation operations. Additionally, managing numerous small allocations entails overhead and introduces non-deterministic behavior, as allocation times remain unpredictable.
- Custom LWIP Heap: Leveraging a dedicated heap memory reserved for dynamic memory allocation within the LWIP section of the application yields a flexible memory management solution. This approach, coupled with memory pools, delivers comprehensive memory management capabilities tailored to the application's needs.

Configuring the Code Size:

The library provides support for both bare-metal and RTOS-based stacks. While the application utilizes FreeRTOS, the networking stack operates on a purely bare-metal level. This decision was made to streamline the application, allowing the lwIP stack to run its tasks independently and free from concerns about context switches and preemptions. Additionally, the library offers separate locking APIs to ensure synchronization within critical tasks. Code size comparisons were conducted with debugging options enabled and disabled, including the omission of assertions. Furthermore, essential statistics were gathered from the library, which includes built-in support for identifying missing packets.

Configuring for Throughput:

Several optimizations were implemented to enhance throughput. These include disabling debugging options and augmenting the MEMP_NUM_* parameter to leverage maximum memory capacity, thereby preventing pool depletion during high-load scenarios. Moreover, checksum generation was disabled to alleviate hardware strain and boost packet transmission rates. Utilizing custom memory pools instead of a heap, although it may result in some memory wastage, significantly accelerates processing. Using raw APIs for the TCP protocol offers superior throughput compared to sequential or socket APIs, enabling zero-copy sends and receives, thereby enhancing performance. Most data is consolidated into a single tcp_write() call, as TCP can aggregate data from multiple calls into one packet. However, this aggregation may lead to packet fragmentation, impacting performance. Passing data without the TCP_WRITE_FLAG_COPY flag avoids unnecessary memory operations, reducing processing time. Disabling Nagle's algorithm also ensures immediate data transmission, circumventing delays caused by waiting for more data to form larger packets. Furthermore, maximizing data transmission before awaiting an ACK, instead of sending small chunks and waiting, can further optimize throughput, ensuring efficient data flow without unnecessary delays.

Configuring for Multithreading Environment:

LWIP's architecture centres around a critical player in multithreading: the tcpip_thread. Crafted precisely for single-threaded environments, this thread orchestrates the library's functions. However, delving into the intricacies of multithreading reveals a landscape rife with challenges. Raw API functions, fundamental to low-level network interactions, require careful handling in multithreaded setups. Their execution should be confined to the core thread to prevent concurrency conflicts, except for those involved in memory management. Communication between application threads, facilitated by socket APIs and the primary thread, is a conduit for seamless data exchange, ensuring coherence amid concurrency. Memory management, a cornerstone of multithreaded systems, assumes a critical role safeguarded by mechanisms like

SYS_LIGHTWEIGHT_PROT. Sharing TCP/UDP control blocks across threads warrants caution, demanding meticulous application of locking protocols to prevent race conditions. Internally, within the intricate pathways of tcpip.c, the library adeptly manages incoming packets and timer events. A prudent delay in packet processing, orchestrated by the tcpip_input() function, allows for the orderly queuing of packets, maintaining a delicate balance between thread dynamics and processing efficiency.

Tuning TCP:

Tuning the TCP is crucial as this protocol provides the majority of the communication, and the most straightforward parameters to perform the same include Maximum Segment Size (TCP_MSS) and Window Size (TCP_WND). TCP_MSS determines the optimal payload size per packet, essential for maximizing throughput potential. Adjusting this parameter unlocks the gateway to enhanced data transfer rates. However, the bottleneck of transmission efficacy lies in TCP_WND, the window size, which ensures the fluidity of data transmission. Maintaining a window size at least double the TCP_MSS mitigates congestion-induced latency, facilitating the uninterrupted data flow across the network. The Nagle algorithm was specifically designed to reduce network overhead. However, its interaction with delayed ACKs can hinder transmission speed, emphasizing the importance of aligning TCP window size with TCP_MSS to mitigate latency-induced throughput degradation.

4.2 Encryption Libraries

In evaluating the decision matrix 3 for selecting an encryption library for the client application facilitating Firmware Over-The-Air (FOTA) updates, the suitability of wolfSSL and mbedTLS were assessed based on several key factors. The scoring reflects the project's specific needs and priorities. Both wolfSSL and mbedTLS received identical scores of 5 for Business Cost. This suggests that both options offer similar cost-effectiveness, making them viable choices for integration into the project without significant budgetary constraints. While mbedTLS scored higher with a 5 for Porting Support (H/W), indicating robust support for hardware platforms, wolfSSL received a score of 3 in this category due to its lack of integration with device SDK projects. This implies that mbedTLS may offer a smoother integration process, particularly for hardware configurations, compared to wolfSSL, which may require additional effort for porting. Both libraries scored 5 for Deployability. Setup for wolfSSL requires additional steps to incorporate for the DUT. Therefore, it has a score of 3 on Development & Setop, while mbedTLS has extensive integration support into SDK projects. Regarding Usability, mbedTLS received a higher score of 5 compared to wolfSSL's score of 4. This scoring suggests that mbedTLS may offer slightly better ease of use and developer experience, although both libraries provide user-friendly and comprehensive documentation. Considering these factors, while wolfSSL and mbedTLS offer compelling features and capabilities, there can be only one winner. Given mbedTLS's slightly higher scores in Setup, Support & Usability, it emerges as a strong contender for integration into the client application.

Alternatives / Factors	wolfSSL	**mbedTLS**
Business Cost	5	5
Porting Support (H/W)	3	5
Deployability	5	5
Development & Setup	3	5
Usability	4	5

Table 3: Decision Matrix for Encryption Libraries

Figure 2: TCP Flow of the Communication between Server and Client

4.2.1 wolfSSL:

wolfSSL [25] is an embedded SSL/TLS library designed for use in resource-constrained environments, including embedded systems and real-time operating systems (RTOS). It is known for its small size, high performance, and extensive feature set, making it a popular choice for developers seeking secure communication solutions for their applications. One of the key highlights of this is its compact size, which ranges from 20 to 100 kB, depending on the specific build options and target environment. This small footprint is crucial for embedded devices with limited memory resources, allowing developers to integrate SSL/TLS encryption capabilities without significantly impacting overall system performance. In addition to its small size, wolfSSL offers excellent performance, often outperforming larger libraries such as OpenSSL. This performance advantage is attributed to its optimized implementation and support for new progressive ciphers. User benchmarking and feedback consistently report better performance with wolfSSL than other libraries. Moreover, it is highly portable and supports a wide range of platforms and operating environments, including Win32/64, Linux, macOS, FreeBSD, Android, iOS, and various embedded platforms such as ARM, Intel, NXP/Freescale, and STM32. Its portability ensures that developers easily use this library across diverse hardware and software ecosystems. Another notable feature of wolfSSL is its support for industry-standard protocols up to TLS 1.3 and DTLS 1.3, providing developers with the latest security enhancements and cryptographic algorithms. This support includes full client and server capabilities and features such as OCSP, CRL support, and OpenSSL compatibility layer for seamless integration with existing applications. Furthermore, this library offers comprehensive hardware encryption and acceleration support on various platforms, enabling developers to leverage hardware cryptographic capabilities for improved performance and security. Additionally, the library provides commercial support packages, including long-term support and maintenance, to meet the specific needs of enterprise customers. Despite its many advantages, it is essential to consider some potential limitations of wolfSSL. For example, while its small size and high performance make it suitable for many applications, developers may encounter challenges when implementing complex cryptographic operations or integrating wolfSSL into highly specialized environments. As with any cryptographic library, proper configuration and integration are essential to ensure optimal security and performance, which requires support from other community developers.

4.2.2 mbedTLS:

The mbedtls library serves as an open-source toolkit renowned for its robust implementation of cryptographic algorithms, X.509 certificates, and support for Secure Shell Layer (SSL) and Transport Layer Security (TLS) protocols [20]. Within the project scope, version 3.5 of this library seamlessly integrates into the SDK for the board package. Its support for the Platform Security Architecture (PSA), tailor-made for Arm processors, encompasses a comprehensive set of threat models, security analyses, and specifications alongside an open-source implementation. Insights into its integration are gleaned from documentation, with a decision made not to adopt the PSA framework. Security is paramount in the application, where the server is initially configured to utilize HTTPS, ensuring secure communication between server and client. The setup entails establishing a TLS channel with the server, a process detailed in the Flow Graph illustrated in Figure 2. This intricate handshake involves a series of 15 messages between server and client, including the exchange of server and client certificates. In the testing environment, the server mandates client-side authentication via a specific target token, further bolstering security measures. For local hosting on port 8443, a custom self-signed certificate is generated using mkcert's keytool. This process yields two key files: a .pem file containing the public key and metadata, shared with the client to validate the server's authenticity, and a -key.pem file housing the private key, safeguarded and accessible only to the server for decrypting incoming data. Certificate validation operations are orchestrated within mbedtls's mbedtls_x509_crt_parse and mbedtls_pk_parse_key functions, accommodating both PEM and DER certificate formats. In the local testing scenario with a self-signed certificate, server validation logic is disabled to facilitate secure communication establishment, given the locally controlled server. Enabling this logic with a self-signed certificate would trigger a failure in the certificate chain validation step, necessitating the disablement of auth_mode for testing purposes. Cipher suite negotiation plays a pivotal role in this handshake. In this assignment, the TLS-ECDHE-RSA-WITH-AES-256-GCM-SHA384 was used. TLS, a cornerstone cryptographic protocol, ensures the privacy, integrity, and authenticity of data exchanged between client and server. The chosen suite indicates key exchange via the Rivest-Shamir-Adleman (RSA) algorithm with Elliptic Curve Diffie

Hellman Ephemeral (ECDHE), combining the strengths of RSA's asymmetric encryption and ECDHE's compact key sizes, ideal for resource-constrained environments. AES-256-GCM is the chosen symmetric encryption algorithm. It operates with a fixed 256-bit key derived from the shared secret established during key exchange. This algorithm ensures robust data encryption, complemented by Secure Hash Algorithm (SHA) 384 for Message Authentication Code (MAC) generation. The MAC, appended to the encrypted data, safeguards against tampering during transmission, crucial for maintaining data integrity. While the MCU's compute resources comfortably accommodate AES encryption, scalability considerations for more constrained devices would necessitate different design choices. Notably, compatibility issues plague IoT deployments, highlighting the importance of aligning client-server security standards. In this project, the server's configuration was updated to default to TLS v1.2 and support TLS v1.3, bolstering security against potential vulnerabilities exploited in previous attacks.

Tuning mbedTLS:
Mbed TLS offers a sophisticated system for fine-tuning to accommodate diverse project requirements, allowing developers to optimize performance while minimizing unnecessary overhead. Encryption within this relies on a robust RSA 2048-bit key, securely stored in PEM certificate format internally. The encryption process hinges on the size of the RSA key, with symmetric and asymmetric techniques employed synergistically. Specifically, an X-bit key string is generated and subsequently integrated with AES_CBC for data encryption. Notably, the key string undergoes RSA encryption before transmission, significantly extending the maximum data encryption capacity achievable solely through RSA. Encountering numerous TLS handshake failures during project execution highlighted the necessity of activating debugging options. Enabling debug mode, set at level 3 within the configuration file, emerged as a critical measure to pinpoint the root causes of failures swiftly. Given the substantial size of the library, debugging challenges underscored the importance of this step in ensuring efficient issue resolution. Furthermore, leveraging mbedtls_strerror() proved invaluable in extracting failure context, particularly beneficial in scenarios with limited debugging capabilities. The entropy collection module within the library is integral as it is the central hub for random number generation. The entropy pool and random generator operate within the same thread context in single-threaded applications. However, architects must weigh the merits of centralizing entropy resources in multi-threaded environments versus adopting thread-specific solutions. The entropy collector aggregates entropy from diverse sources, amalgamating them into its internal state to ensure a robust entropy pool. Concurrently, random generators like the CTR-DRBG module optimize entropy utilization, generating random numbers without depleting the original entropy. In threaded environments, a central entropy collector is initialized once and emerges as the preferred approach. This strategic choice prevents entropy contention among sibling collectors, enhancing overall entropy efficiency. Two distinct options exist for employing random generators in threaded environments. The first option entails deploying separate CTR-DRBG or HMAC-DRBG random generators for each thread, coupled with a thread-specific custom personalization string. This approach minimizes the correlation between random generators across threads, bolstering their independence. Conversely, a central CTR-DRBG or HMAC-DRBG context, initialized in the main thread and shared across threads, ensures thread safety through context-level locks. Further optimization measures include leveraging custom functionality tailored for the MCU, leveraging FreeRTOS, and defining the MBEDTLS_MCUX_FREERTOS_THREADING_ALT macro. These modifications, integrated into the configuration file, optimize hardware resource utilization in multi-threaded environments, enhancing overall system efficiency and performance. In addition to fine-tuning, optimization efforts have been bolstered by implementing alternative cryptographic modules to maximise hardware acceleration capabilities. This strategic enhancement ensures that modules supporting alternative implementations harness hardware acceleration features, optimizing hardware resource utilization. Facilitating this process is the macro MBEDTLS_ENTROPY_HARDWARE_ALT, which triggers the invocation of hardware entropy implementation through hardware_poll() within the library. This, in turn, activates the True Random Number Generator (TRNG) driver module (fsl_trng), augmenting cryptographic operations with hardware-based entropy sources. Furthermore, meticulous attention has been devoted to thread safety features within encryption algorithms, memory management, and context sharing. Although implicit context sharing is preferable, scenarios may necessitate explicit implementation. Similarly, akin to the lwIP (lightweight IP) library, this library has been meticulously configured to leverage a customized memory pool/buffer to enhance operational efficiency. Notably, reducing the debug context to 0 yields a substantial reduction in memory footprint, further optimizing performance. Benchmark results about the employed

Symbol	Size(KB)
mbedtls_ssl_handshake_client_step	8.15
mbedtls_ssl_handshake_server_step	6.02
mbedtls_ssl_read_record	5.21
mbedtls_internal_sha512_process	3.07
mbedtls_high_level_strerr	2.58
Total Size of the mbedtls Library	**114.56**

Table 4: mbedTLS library code size

cipher suite underscore impressive cipher and hash speeds, with AES-128-CBC and SHA384-HMAC clocking in at 320KB/s and 440KB/s, respectively. However, performance metrics for SSL/TLS communication via TCP stream averaged around 7 KB/s, indicating potential avenues for refinement. Addressing scenarios where specific features are unsupported, optimization measures have been tailored to curtail binary footprint and ensure optimal utilization of flash/ROM. This optimization includes resizing Maximum Precision Integers (MPI) to 1024 bytes and adjusting the sliding window size of MPIs. Additional optimizations involve disabling large, unused elliptic curve algorithms and reducing the maximum ECP bits, albeit with minimal memory impact. Disabling fixed-point optimizations for elliptic curves, while impacting performance, reduces memory consumption. Moreover, strategic memory management practices have been adopted, leveraging a 16KB frame buffer for incoming and outgoing data frames. Further optimization opportunities exist on the server side, allowing users to configure maximum frame size via MBEDTLS_SSL_MAX_CONTENT_LEN, thereby minimizing buffer requirements. Leveraging the macro MBEDTLS_AES_ROM_TABLES enables the storage of AES implementation tables in ROM, effectively reducing RAM utilization. Furthermore, careful management of X509 certificates during handshakes is paramount for memory optimization. It involves refraining from storing duplicate copies of certificates in RAM and promptly removing certificates post-SSL handshake. The memory abstraction layer, defined via MBEDTLS_PLATFORM_MEMORY, integrates custom memory allocation and deallocation implementations via FreeRTOS APIs, ensuring efficient resource management. Notably, the macro MBEDTLS_ERR_PLATFORM_FEATURE_UNSUPPORTED proves instrumental in error determination during cipher suite negotiations, identifying incorrect implementations or disabled cipher suites. In the assignment context, an alternative implementation of SHA-384 has been leveraged to maximize hardware utilization for secure communication. Comprehensive details regarding the library's occupied size are available in 4, providing valuable insights into memory utilization and optimization potential.

4.3 Parsing Libraries

4.3.1 coreHTTP

The coreHTTP library [6], designed as an HTTP C client implementation tailored for Microcontroller Units (MCUs) and smaller Microprocessing Units (MPUs), provides a lightweight solution for HTTP/1.1 communication. Running atop the TCP/IP stack, this library offers a fully synchronous API that optimizes concurrency management while minimizing memory footprint. Operating on fixed buffers, developers retain control over memory allocation strategies, enhancing flexibility and resource utilization. With intuitive APIs for header creation, request transmission, and response parsing, coreHTTP simplifies HTTP message handling, enabling swift integration into MCU applications. Additionally, the library seamlessly integrates with secure transport interfaces, including Transport Layer Security (TLS), leveraging the features of the mbedtls library for enhanced security. Notably, for the ARM Cortex M series, the library achieves an impressive code size of 23.9KB with default O1 optimization, further reduced to 20.7KB when optimized for size. llhttp, a low-level parser embedded within coreHTTP, plays a pivotal role in this optimization, proficiently handling both HTTP/1.1 and HTTP/2 requests and responses. Given the challenge of processing HTTP messages received in chunks, llhttp ensures efficient message handling, contributing significantly to the library's compact size and efficient operation.

4.3.2 coreJSON

The coreJSON library [7] serves as a robust JSON response parser, crucial for interpreting data exchanged with popular servers employing Representational State Transfer (REST) APIs. Implemented entirely in C, this parser ensures compliance with the ECMA-404 JSON standard, facilitating seamless integration with diverse server architectures. Notably, coreJSON supports efficient key lookups, enabling rapid extraction of pertinent data from complex JSON structures. Internally, the library employs a stack mechanism to navigate nested JSON objects, ensuring accurate parsing and interpretation. The stack, dynamically managed during function calls, provides essential context for handling hierarchical JSON data. Developers can customize the stack size using the JSON_MAX_DEPTH macro, offering flexibility to optimize memory usage based on application requirements. In terms of resource efficiency, the library achieves impressive performance metrics, with a code size of 2.9KB for the Cortex M series under default O1 optimization, further reduced to 2.4KB with size optimization. This compact footprint, coupled with its robust functionality, positions coreJSON as a vital component for MCU applications requiring seamless JSON parsing capabilities.

The subsequent phase of the application focuses on establishing the network transport interface, a pivotal component for facilitating communication over the network. This process entails encapsulating the TCP sockets responsible for transmitting and receiving data within a network context structure. This implementation is conveniently provided as part of the coreHTTP library, offering seamless integration with the application's networking layer. One notable advantage of leveraging the core libraries from FreeRTOS is their independence from the FreeRTOS kernel. This unique characteristic enables their utilization in diverse project environments, whether single-threaded bare-metal configurations or those employing alternative real-time operating systems such as Zephyr. This versatility underscores the adaptability and interoperability of the coreHTTP library within varied embedded systems architectures.

4.4 FreeRTOS Kernel

The decision to utilize the FreeRTOS kernel [8] in the application was driven by its straightforward usability and scalability, especially when compared to bare-metal programming. Additionally, FreeRTOS offers seamless support for configuring various libraries, making it highly adaptable across devices with minimal porting overhead. Despite its stability, the project continues to undergo continuous development, ensuring it remains robust and adaptable to evolving requirements. Moreover, FreeRTOS boasts minimal memory and processing overhead, with typical kernel binary images averaging around 10 KB. Crucially, the project is open-source, eliminating the need for a commercial license, which aligns well with the requirements of stakeholders, including 3T, who already leverage FreeRTOS extensively within their existing applications. For the FreeRTOS application, a kernel port with Memory Protection Unit (MPU) support for the Cortex M7 arm cores was chosen. This feature enhances the application's robustness and security by enabling tasks to run in either privileged or unprivileged mode, thus restricting access to critical resources such as RAM, executable code, peripherals, and memory beyond a task's stack limit. By preventing code execution from RAM, the application becomes less vulnerable to malicious code injection attacks, including buffer overflow and Return Oriented Programming (ROP). This level of isolation is particularly advantageous in the context of Firmware Over-the-Air (FOTA) updates, where frequent flash writes/erases occur. Utilizing MPU-based isolation mitigates the risk of faults in one task, compromising the integrity of the OTA process. Furthermore, leveraging MPU support allows for the designation of specific memory regions as non-executable, ensuring that only required memory regions are accessible to necessary tasks, thereby enhancing security. Additionally, prioritizing the OTA task within the system ensures that firmware updates receive the necessary attention and resources. Another benefit of FreeRTOS is its backward compatibility, extending support to Cortex M3 and M4-F cores. Regarding task management, FreeRTOS offers robust mechanisms such as task delay, queue management, and synchronization primitives like mutex, notifications, and semaphores. These features simplify task coordination and ensure data security within the application. By utilizing task delay, essential operations, such as obtaining an IP address via DHCP, ensure adequate time is facilitated for performing Ethernet initialization. Moreover, FreeRTOS's queue management solution simplifies data passing, enhancing flexibility and efficiency, especially when dealing with large messages. Binary semaphores have been employed for task synchronization between the OTA task and other tasks and are responsible for security and networking functions, ensuring smooth operation. Dynamic memory management, facilitated by FreeRTOS's heap management scheme, allows for efficient allocation of resources. By extending the heap

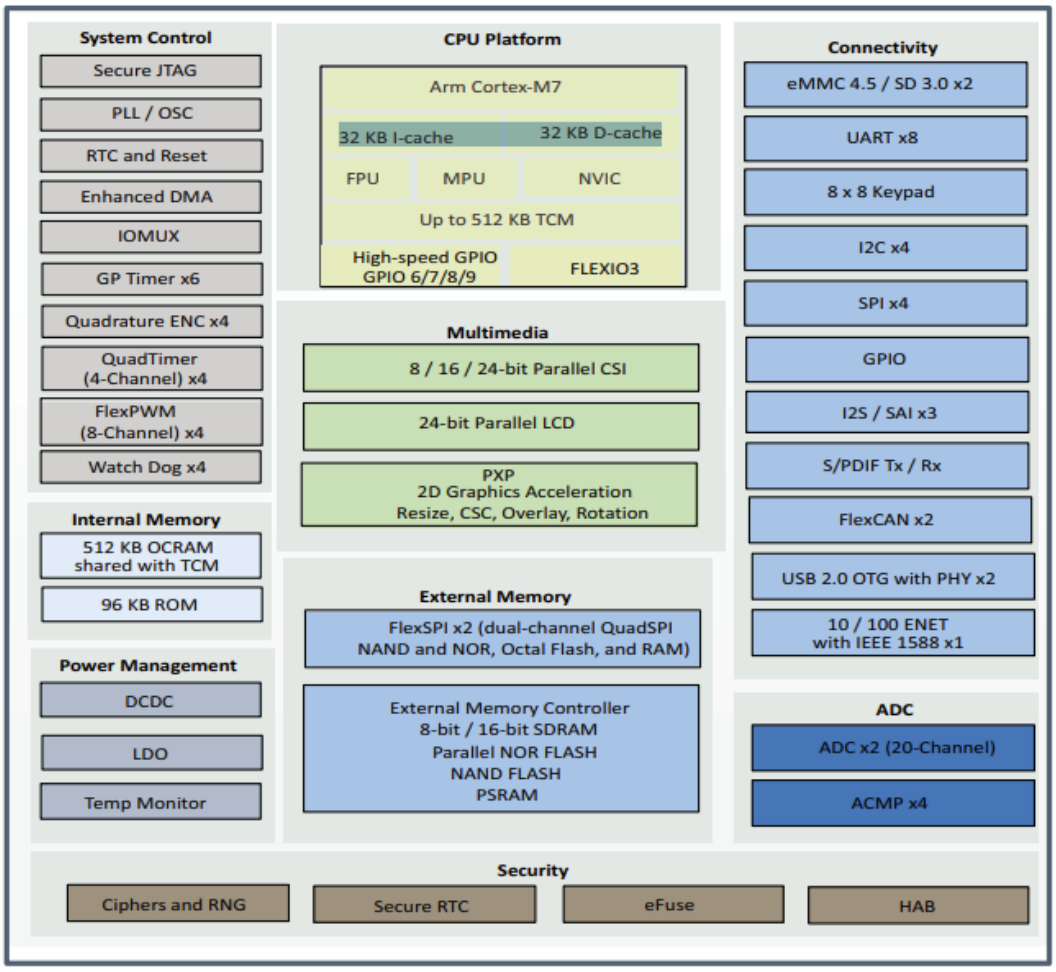


Figure 3: System Block Diagram of the iMXRT1060 [14]

size to accommodate larger responses and chunk allocations for firmware packets, scalability and flexibility are ensured. The application utilizes heap_4, the most widely used heap allocation strategy, for its efficiency and flexibility. Thread-local storage pointers and stack overflow detection mechanisms have been implemented to enhance data security and mitigate stack overflow risks. The application's runtime statistics were captured by using the General Purpose Timer (GPT) to track the runtime of the threads utilized by the kernel. This is further discussed in section 9.2.

5 Device Under Test:

The iMXRT1060 MCU [14] 3, leveraging the Arm Cortex-M7 Core Platform, boasts a rich array of features ideal for diverse applications. At its core lies a single Arm Cortex-M7 Core, fortified with a 32 KB L1 Instruction Cache and a 32 KB L1 Data Cache, enhancing computational efficiency. Its full-featured Floating Point Unit (FPU) adheres to the VFPv5 architecture, ensuring robust numerical processing capabilities while supporting the Armv7-M Thumb instruction set for streamlined code execution. Noteworthy is its integrated Memory Protection Unit (MPU), offering up to 16 individual protection regions, bolstering system security. Tightly coupled GPIOs operating at the core frequency enhance real-time responsiveness. Additionally, with up to 512 KB of Tightly Coupled Memory (TCM), comprising both instruction and data TCM, the iMXRT1060 facilitates high-speed access to critical code and data. Operating at a frequency of 528 MHz, the Cortex M7 CoreSight™ components integrated within the core streamline debugging pro-

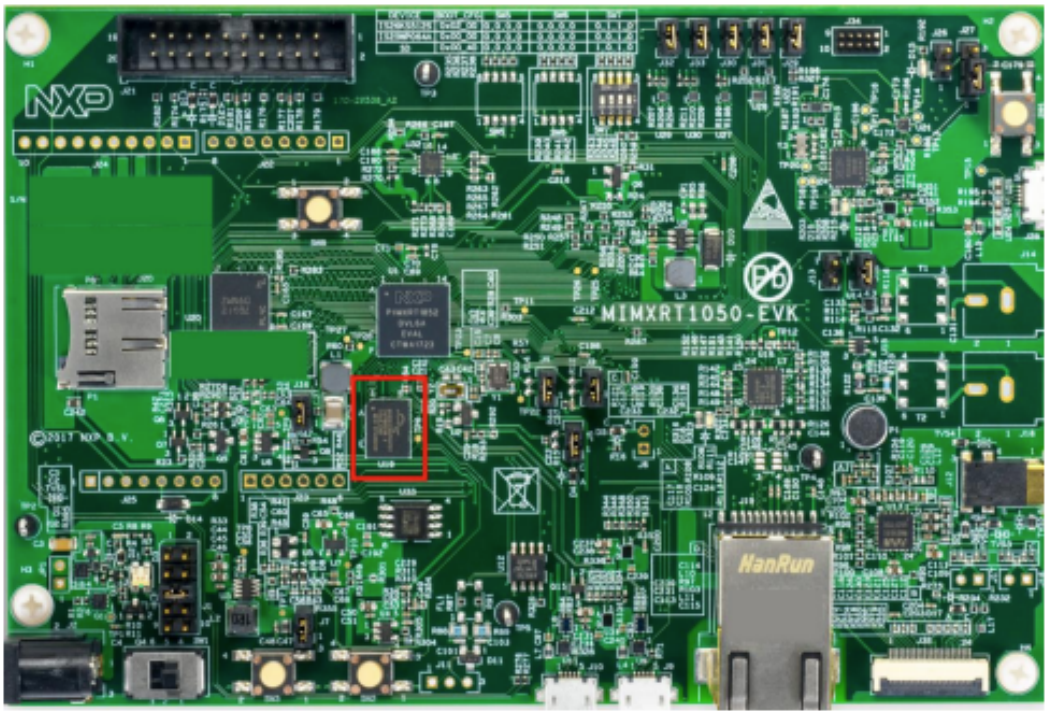


Figure 4: NXP iMXRT1060 EVKB [12]

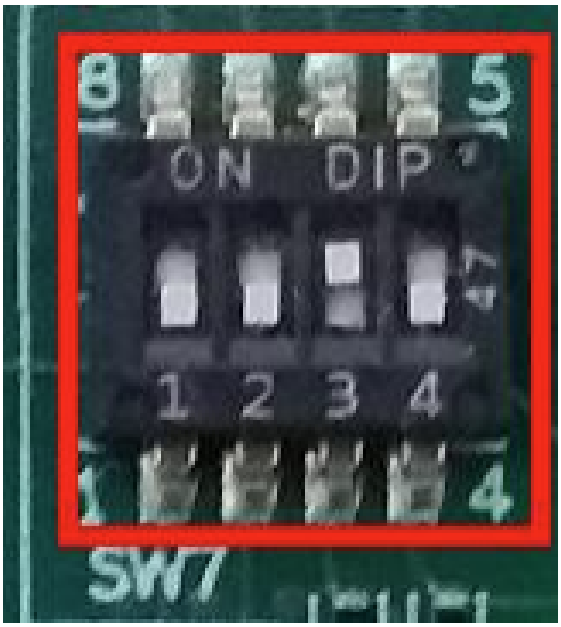


Figure 5: Boot Switch Configuration [12]

cesses, ensuring efficient software development and debugging cycles. On the system-on-chip (SoC) level, the i.MX RT1060 is equipped with essential memory components, including a 128 KB Boot ROM and 1 MB on-chip RAM, ensuring seamless boot-up and ample working memory. External memory interfaces cater to a range of memory types, from SDRAM to various flash memory configurations, offering versatility in memory expansion options. The MCU further supports an array of timers and PWMs, facilitating precise timing control in applications. Its extensive range of interfaces caters to diverse connectivity needs, including USB 2.0 OTG controllers, Ultra Secure Digital Host Controller (uSDHC) interfaces, Ethernet controllers, UART modules, I2C modules, SPI modules, FlexCAN modules, and FlexIO modules, enabling seamless integration with external devices. Advanced power management units and controllers optimize power consumption, ensuring energy efficiency in operation. The Arm CoreSight debug and trace architecture facilitates system-level debugging, streamlining development workflows. Analog interfaces, such as ADCs and ACMPs, enrich the MCU's sensing capabilities, enabling data acquisition in various applications. Security is a paramount concern addressed by hardware-based features, including High Assurance Boot (HAB), Data Co-Processor (DCP), Bus Encryption Engine (BEE), True Random Number Generation (TRNG), Secure Non-Volatile Storage (SNVS), and Secure JTAG Controller (SJC), fortifying the MCU against potential threats and ensuring data integrity and confidentiality. The complete system architecture of the board under test is best depicted in 3 From the figure 4, the highlighted section represent the Hyper Flash from the board that has to be disabled to enable boot from QSPI Flash [12]. The boot switches have to be set to the configuration pointed out by 5 for the same.

6 FOTA Design processes:

The FOTA process described herein draws insights from [4] and is specifically tailored for NXP's RT series, incorporating the Secondary Bootloader (SBL) and Secure Firmware (SFW). This intricate process leverages a combination of hardware and software components to facilitate seamless firmware updates.

The core of the FOTA process lies in the Secure Firmware (SFW), responsible for orchestrating the procurement and installation of new firmware images. These images are sourced from three available options: Cloud, U-Boot, or the SD card. Background FreeRTOS tasks are instrumental in this process, as they handle the acquisition of new images and write them onto the flash storage of the device. Once the installation is complete, the device is rebooted, triggering the SBL to verify the integrity of the new image before finalizing the update.

Central to the FOTA process are two distinct modes of operation supported by the RT1060: Swap mode and Remap Mode. Regardless of the mode, the flash memory is logically partitioned into three sections: Bootloader, Firmware 1, and Firmware 2, facilitating seamless firmware reversion if necessary. The SBL, acting as the arbiter of firmware exchange, employs several states during the FOTA process:

- Test: Temporary Exchange
- Perm: Permanent Exchange
- Revert: Rollback Exchange:
- None: No Exchange

6.1 Swap mechanism

In the Swap mechanism, the FlexSPI, a versatile SPI module, facilitates communication with the flash module. Configuring the fuse switches on the board to boot from Flash initiates the ROM code on power-on. This code is responsible for hardware initialization and bootloader loading. Transitioning into eXecute In Place (XIP) mode allows the ROM to directly execute the bootloader code from non-volatile memory, optimizing RAM usage and potentially expediting execution.

Firmware images adhere to specific formats outlined in Figure 8b. To integrate a new firmware image, a script is required to add the necessary header and signature. In this mode, Slot1 serves as the jump address for the SBL, while Slot2 is designated for downloading the new firmware. Upon reset initiated by the SFW, the corresponding image trailer values are updated to notify the SBL of the new image. Subsequent integrity

checks performed by the SBL culminate in the swapping of images across slots, ensuring seamless firmware transition.

However, in iMXRT chips, hardware verification is linked to the link address. Consequently, the SBL swaps the images in both slots before verifying the signature, reverting to the swap if validation fails. This robust verification mechanism guarantees the proper execution of new firmware images while aligning with the ROM's hardware verification protocols.

Additionally, to streamline bootloader functionality, the execution address of the new firmware remains constant and predetermined. This enables the bootloader to relocate the latest firmware to the address specified by the link file and execute it seamlessly. This approach ensures the preservation of the old firmware, facilitating reversion if necessary. Notably, this mode imposes a size constraint on firmware, necessitating that it does not exceed the size of a slot or the size of two sectors, one reserved for Image Trailer and the other serving as a temporary buffer for sector-wise swap operations across slots.

6.2 Flash Remapping

The MXRT1060 introduces a powerful feature known as Flash Remapping [13], enabling uninterrupted data streaming from flash memory via the FlexSPI. While not universally supported, this technique leverages the Advanced High-Performance Bus (AHB) of the ARM architecture to facilitate seamless data transmission.

This feature is particularly significant for FOTA operations, as it circumvents the need for image exchange scenarios by allowing the Secure Firmware (SFW) to execute using the same linker file across different physical addresses. Doing so significantly reduces the number of flash erase/write operations, thereby enhancing device longevity and shortening update times.

Operationally, Flash Remapping adopts a format similar to the Swap process, albeit with modifications. Each slot no longer includes the Image Trailer section, as the Secondary Bootloader (SBL) dynamically manages the execution address. The last 32 bytes of each slot contain essential flags for the update process, facilitating seamless firmware transition.

Central to this mechanism is the SBL's ability to utilize a fixed logical jump address aligned with the start address of the image in Slot1. By adjusting the Remap area and offset values, the SBL ensures that the AHB accesses the same position in Slot 2. This dynamic address mapping allows the SBL to execute firmware updates from various physical addresses, a departure from the static nature of the Swap mechanism.

Upon reset, the SBL reads the Remap flags to ascertain the location of the current firmware and check for updates. In the event of a failed signature verification for the updated firmware, the SBL initiates a board restart, reverts the image, and completes the rollback process. This validation process ensures the reliability and robustness of the firmware, maintaining the system's integrity.

A straightforward implementation of this mechanism is exemplified through the provided SDK's example application, which utilizes the NOR Flash connected via FlexSPI. This practical demonstration underscores the versatility and efficacy of this feature in facilitating seamless FOTA operations. (Refer to Figure 6 for visualization.). Hence, this mechanism is used in the context of this application. However, the application can be configured such that the user is able to switch to a swap mechanism to perform the upgrade on devices where this functionality is not supported.

6.3 Secondary Bootloader Design:

As an integral component of the system architecture, the Secondary Bootloader (SBL) assumes a pivotal role, serving as a preferable alternative to a ROM Bootloader. Its primary functions encompass flash management, communication, and security protocols tailored to meet diverse operational requirements. The bootloader operates flexibly in Execute in Place (XIP) and non-XIP modes, adapting seamlessly to the system's needs. However, it is important to note that this versatility comes with the caveat of consuming critical memory space in resource-constrained devices.

6.3.1 Flash Operation:

The iMXRT1060 leverages the FlexSPI interface to interface with Quad SPI (QSPI). Upon configuring the hardware to boot from QSPI, the ROM initializes the FlexSPI and QSPI configuration during the boot phase, streamlining the boot process. By default, flash programming operations are disabled, necessitating

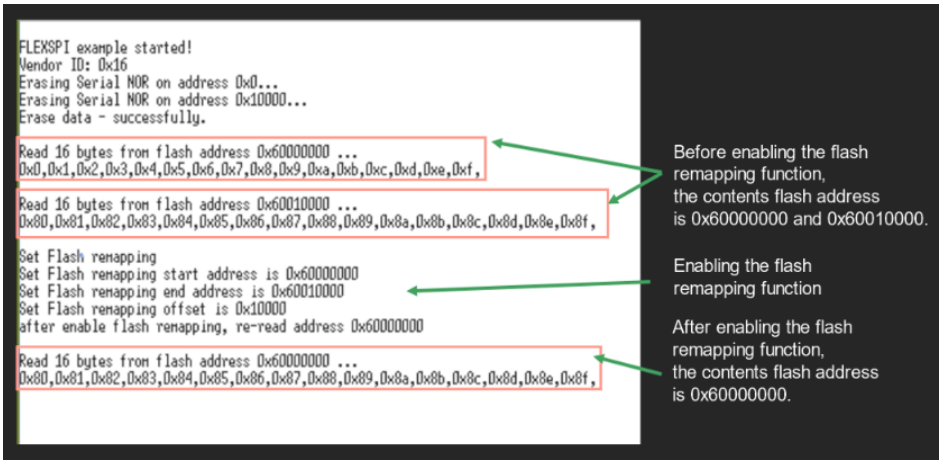


Figure 6: Flash Remapping performed for the iMXRT1060 [13]

the configuration of the Look-Up Table (LUT) to support additional commands. As documented in [17], certain precautions are taken to ensure stable operation:

- Interrupts and prefetch buffers are disabled before any FlexSPI operation. To enable FlexSPI software reset, the MCR0 register's SWRESET bit was set.
- Code for FlexSPI operations is allocated to internal RAM.
- FlexSPI software reset is executed after successful flash program execution. APIs provided in the "flexspi_nor_flash_ops.c" from the board's SDK serve as the basis for implementation, drawing insights from reference examples.

6.3.2 Communication:

The current iteration of the bootloader facilitates Over-the-Air (OTA) updates, wherein the Secure Firmware (SFW) loads the new firmware image onto an empty slot and utilizes either Flash remapping or the Swap mechanism to execute the update. Essential data such as status, addresses, and header information are stored in partitioned slots within the flash. Temporary storage on the stack ensures that the bootloader can take appropriate actions as required by the application.

6.3.3 Security Management

The SBL plays a crucial role in facilitating updates of both raw and encrypted images. The current configuration expects an encrypted RSA-2048 image to be downloaded from the server. Leveraging the trusted Bus Encryption Engine (BEE) region on the board, the SBL verifies the image's validity after a specified timeout period. In the event of encryption, the SBL utilizes the EKIB and EPRDB to configure BEE region one and reserve region 0 for bootloader encryption. Image encryption employs SW_GP2 as a key and utilizes BEE region 1 for decryption. Furthermore, BEE region 0 is reserved for the SBL, allowing OTPMK to serve as the bootloader key. Figure 7 depicts the address space configuration. A build script integrated with imgtool automatically generates encrypted images, fetching the raw firmware image from the build server and dispatching it to the local update server seamlessly.

Based on documentation cited in [24], the RT1060 features a Memory Management Unit (MMU) with a minimum page limit of 4KB, resulting in the alignment of functional address spaces accordingly. Each slot is allocated 1MB of space. It's crucial to note that the application program's initial address (Interrupt Vector Table - IVT) doesn't align with the start address of slot one due to the 32-byte header reserved for OTA process information. Determining the IVT's start address for the application program is essential. The RT1060 with the CM7 processor [15] supports 256 interrupt sources, yet only 96 interrupt vectors are available for this project, leading to an IVT size of 384 bytes with a memory-aligned value of 0x200. Consequently, the

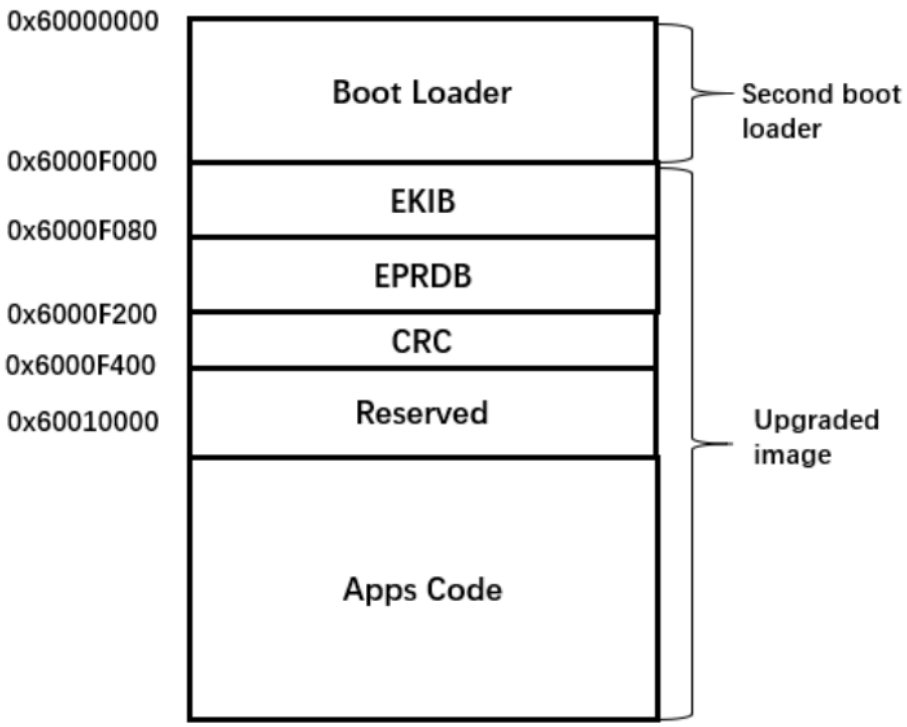
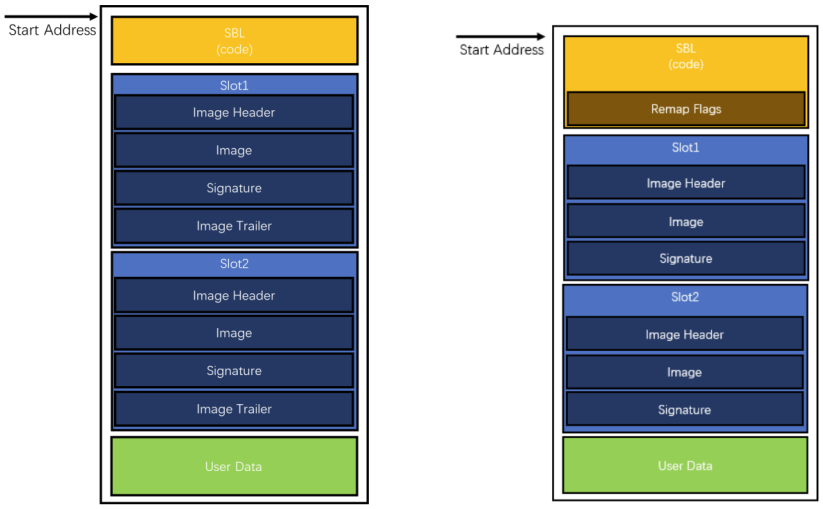


Figure 7: Address Space Configuration for the Secure Image [4]

application code begins with an offset of 0x200. Table 5 provides a comprehensive breakdown of the address space allocation. Secure Boot Feature: The SecureBoot feature, an integral part of the board, ensures secure booting based on the ROM bootloader and encrypted booting facilitated by the hardware engine, enabling image signing, encryption, or both. The default method entails generating an RSA hash of the image and signing the hash, preventing unauthorized code execution on the board. Upon reset, the ROM initiates its execution, examining the first user executable image in flash memory for code authenticity. If deemed authentic, control transfer occurs, establishing a chain of trust from ROM to Secondary Bootloader (SBL) to Secure Firmware (SFW) application. High Assurance Boot (HAB): HAB plays a pivotal role in detecting and preventing unauthorized software during the boot sequence, employing asymmetric cryptography for image signing via the CST tool within BootUtility software. The CST tool generates binary files comprising command sequences and signatures based on input CSF files. Deployment scenarios necessitate generating private and public keys to encrypt the image digest, with the public key's hash burned into the iMXRT chips' eFuses, ensuring immutable verification. The private key creates the digest (signature), while the corresponding public key validates firmware authenticity. Encrypted Execute in Place (XIP) Boot based on BEE: This mechanism allows code execution directly from flash memory, negating the need for copying into RAM. Currently, the BootROM supports two encrypted regions (one for SBL and another for SFW), which are protected by AES keys. Proper BEE configuration, detailed in the Protection Region Descriptor Block (PRDB), ensures encryption integrity. The PRDB is encrypted using AES-CBC-128 mode, with its key and initialization vector stored in the Key Info Block (KIB). Additionally, the KIB is encrypted as Encrypted KIB (EKIB) using the AES key from eFUSE. In this Proof of Concept (POC), the AES key in the KIB decrypts the image, while the Key Encryption Key (KEK) for the KIB is SW_GP2. The BootUtility tool encrypts the dispatched firmware image on the local server, which is then retrieved, decrypted, authenticated, and executed on the local client running on the board.

Area	Function	Add _Start	Add_End
SBL	SBL functions	0x60000000	0x6003EFFF
OTA Flag Data	User Data	0x6003F000	0x6003FFDF
32 Bytes	OTA Flag	0x6003FFE0	0x6003FFFF
Slot 1	Image Header	0x60040000	0x600401FF
	Application Image	0x60040200	0x6004043FF
Slot 2	Image Header	0x60240000	0x602401FF
	Application Image	0x60240200	0x602403FF

Table 5: Complete Address Space Allocation for the RT1060



(a) Swap Format (b) Flash Remap Format

Figure 8: Swap Format and Flash Remap Format [4]

6.4 Ethernet Configuration

The RT1060 series incorporates a single ethernet module enabling device connection to fast ethernet at rates of 10/100 Mbps, supporting two fast ethernet interfaces compatible with Media Independent Interfaces (MII) and Reduced Media Independent Interfaces (RMII) for MAC and PHY communication. This media independence ensures versatility, allowing the same MAC hardware to adapt to different media forms. Additionally, all devices support the Serial Management Interface (SMI), with MDIO and MDC signals utilized for Read/Write control and status register manipulation of the PHY to configure each PHY pre-operation. Pin and clock configurations, crucial for proper functionality, have been outlined in [16] and must be adhered to for the application's success. The Software Development Kit (SDK) provided for the board furnishes the necessary ethernet driver interface (fsl_enet) and physical interface modules (fsl_phyksz8081) to seamlessly interact with their corresponding hardware components.

7 System Integration Test:

Rigid testing was conducted on both the server and client components individually to assess the feasibility and functionality of the entire system.

7.1 Server System Test:

Initially, basic functionality tests were conducted using curl commands to simulate client-server interactions. However, extensive testing was conducted using a Python script to simulate real-world scenarios where multiple clients concurrently connect to the server. This script created mock devices that initiated connections, performed request sequences, and simulated connections and disconnections.

7.2 Client System Test:

The client application underwent testing using a mock server application, which served files at specific URLs. The client established connections, requested necessary files, and implemented flash remapping techniques to boot with the new application.

7.3 Final Test:

Testing with the Hawkbit server posed additional challenges, necessitating several adjustments compared to testing with the mock server. Configuration via TLS was required, and despite efforts to bypass certificate validation errors, direct curl commands yielded inappropriate responses. Debugging became complex when server responses did not align with expected results. Moreover, issues with the coreHTTP library were identified during attempts to stop HTTP/JSON responses from the server. A community post provided insights [10] which identified the problem in the llhttp response parser that made it difficult to track chunked responses and provided a solution to halt parsing in this specific scenario. Although an alternative Lua hook [18] was available, leveraging the JSON parser proved more effective for capturing and processing JSON data in subsequent requests. Subsequent tests involved deploying different binaries to ensure consistent flash remapping across various application functionalities and peripherals. System configurations on the server side were adjusted to enforce TLS-based client authorization using security tokens generated during client-server interactions, as detailed in [11]. This token, stored on the client, was utilized in subsequent request headers. While various deployment configurations exist, the specific test configuration involved mapping a single distribution to a hardware-specific software module for forced updates. Detailed results of these tests are presented in the subsequent section, highlighting the system's performance and reliability under varying conditions.

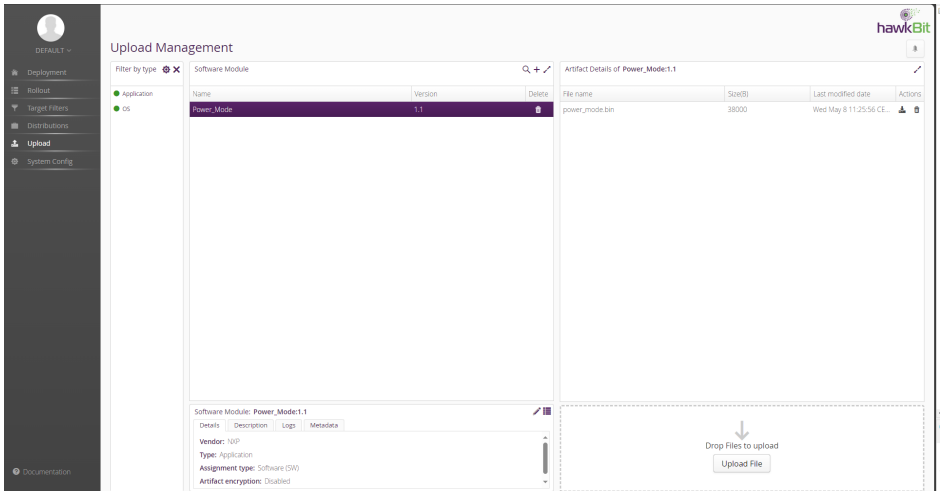


Figure 9: Upload Management for Server

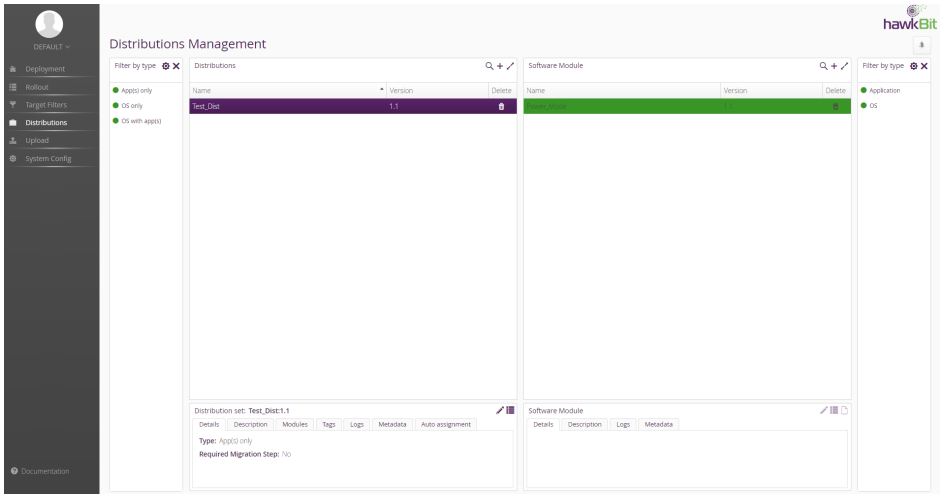


Figure 10: Distribution Management for Server

8 Results:

The detailed steps for using the Hawkbit server for deployment are as follows:

1. The binary to be used for deployment is uploaded through the Upload Management tab of the UI by creating a software module of the required configuration. In this assignment, the power mode demo application was identified to provide the best representation as it makes use of peripherals and the LPU UART driver for obtaining output via the serial console 9

2. The corresponding distribution for the software module is created such that the software module is now tagged/assigned to the specific distribution. In this case, a new distribution named Test DS has been created, assigned the required parameters, and now tagged to the created software module in the previous step. 10

3. A custom filter for the new distribution can now be created based on which the connected device with the matching criterion is provided with the deployment base upon request to be used for downloading the corresponding artefact. In this case, the client device has been configured to connect with the name 'iMXRT1060'; therefore, the filter is designed to identify the corresponding device and auto-assigned to serve the distribution 'Test DS' to the matching device.

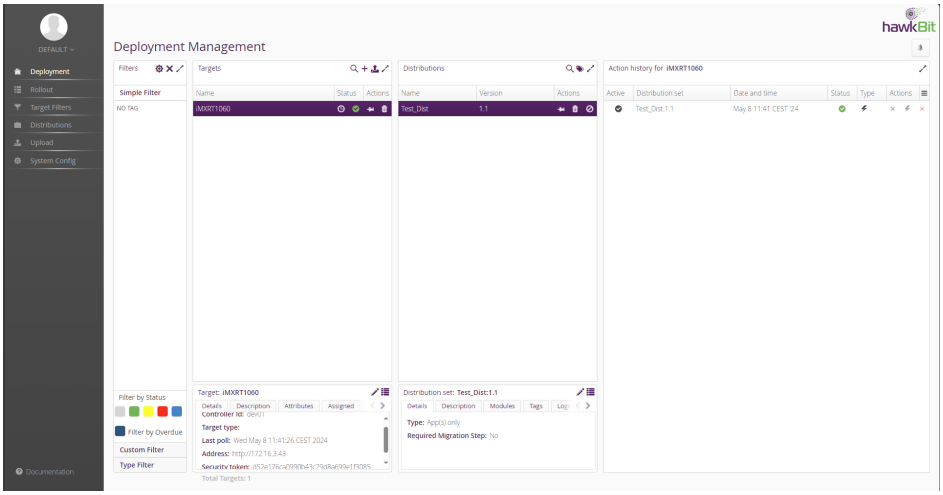


Figure 11: Deployment Management for Server

4. The deployment management contains the list of connected devices and the corresponding actions. As can be seen from the action history depicted in 13 the client device has successfully performed an update action that can be confirmed by the green check on the status obtained after the client's feedback for a successful download action.

5. On the client side, the device, after downloading the required artefact, undergoes a reboot, as can be identified from the serial console

6. However, the new candidate application, i.e. the power mode application downloaded on Slot 2 of the Flash memory located at address 0x60240400, is now executed at address 0x60040400 using the same linker file identified below.

9 Analysis:

The secure communication between the server and the client, detailed in section 4.2.1, underwent a comprehensive analysis to evaluate its performance and efficiency.

9.1 mbedtls Library Optimization:

The mbedtls library emerged as a major contributor to the application's size, underscoring its significance in optimization efforts. Optimizing the mbedtls library for debug mode yielded results summarized in Table 4. Subsequent release builds, with varying compile-time optimizations, were conducted, and their outcomes are presented in Table 8. Notably, optimization efforts demonstrated significant improvements, particularly with the -Os optimization, which resulted in a nearly 20% reduction in application size compared to the original build. While the -O3 optimization produced the largest binary size among all optimizations, this outcome aligns with expectations, as -O3 is designed to prioritize execution speed, often leading to increased code size due to function inlining and loop unrolling.

9.2 Memory Utilization and Task Analysis:

Heap memory utilization, as illustrated in Figure 14, was approximately 41%, indicating efficient memory management within the system. Analysis of FreeRTOS task lists revealed predominant periods of idle thread execution of around 99% with separate threads allocated for lwIP library operations as the tcpip thread and the receive run on different threads. A possible explanation for this separation could be that one thread is used to maintain the connection sockets. At the same time, the other anticipates the connected

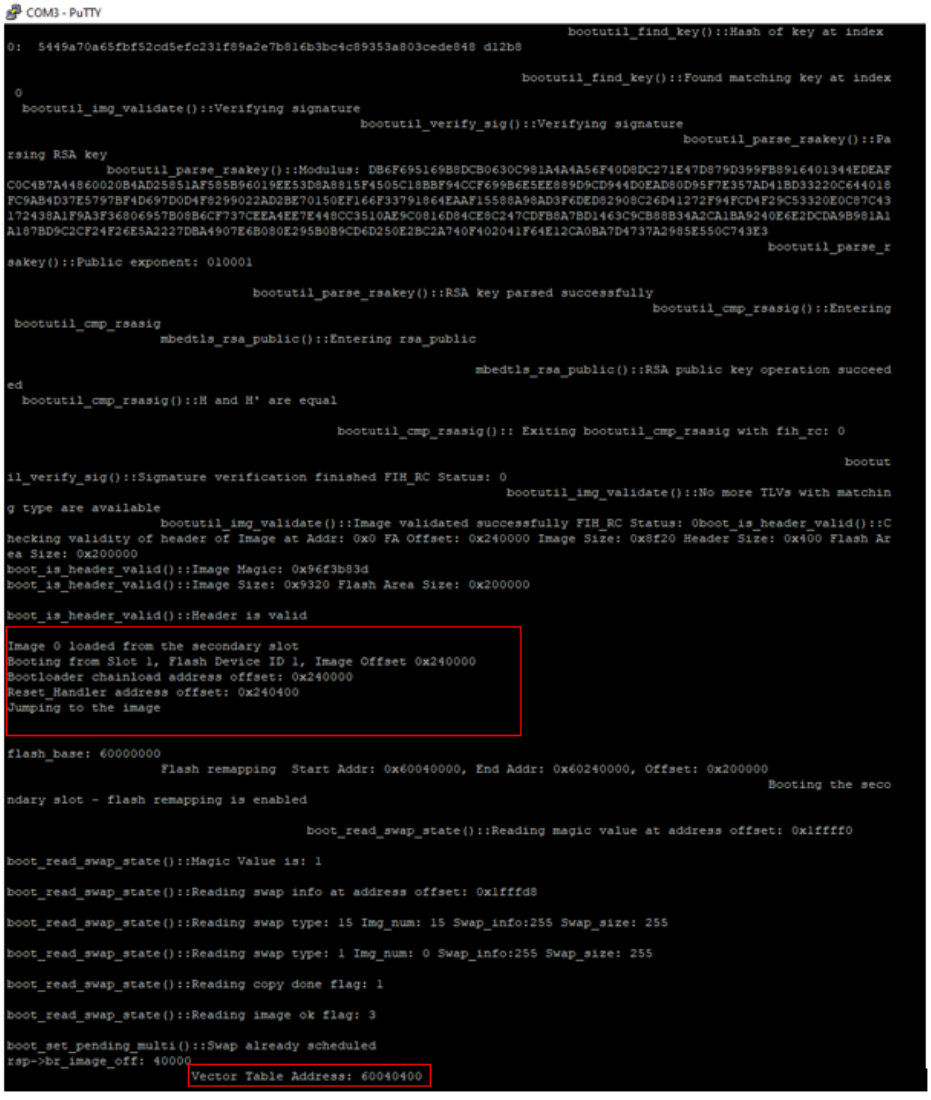


Figure 12: Client Side- Flash remapping (Jumping to the Address space 0x6004040)

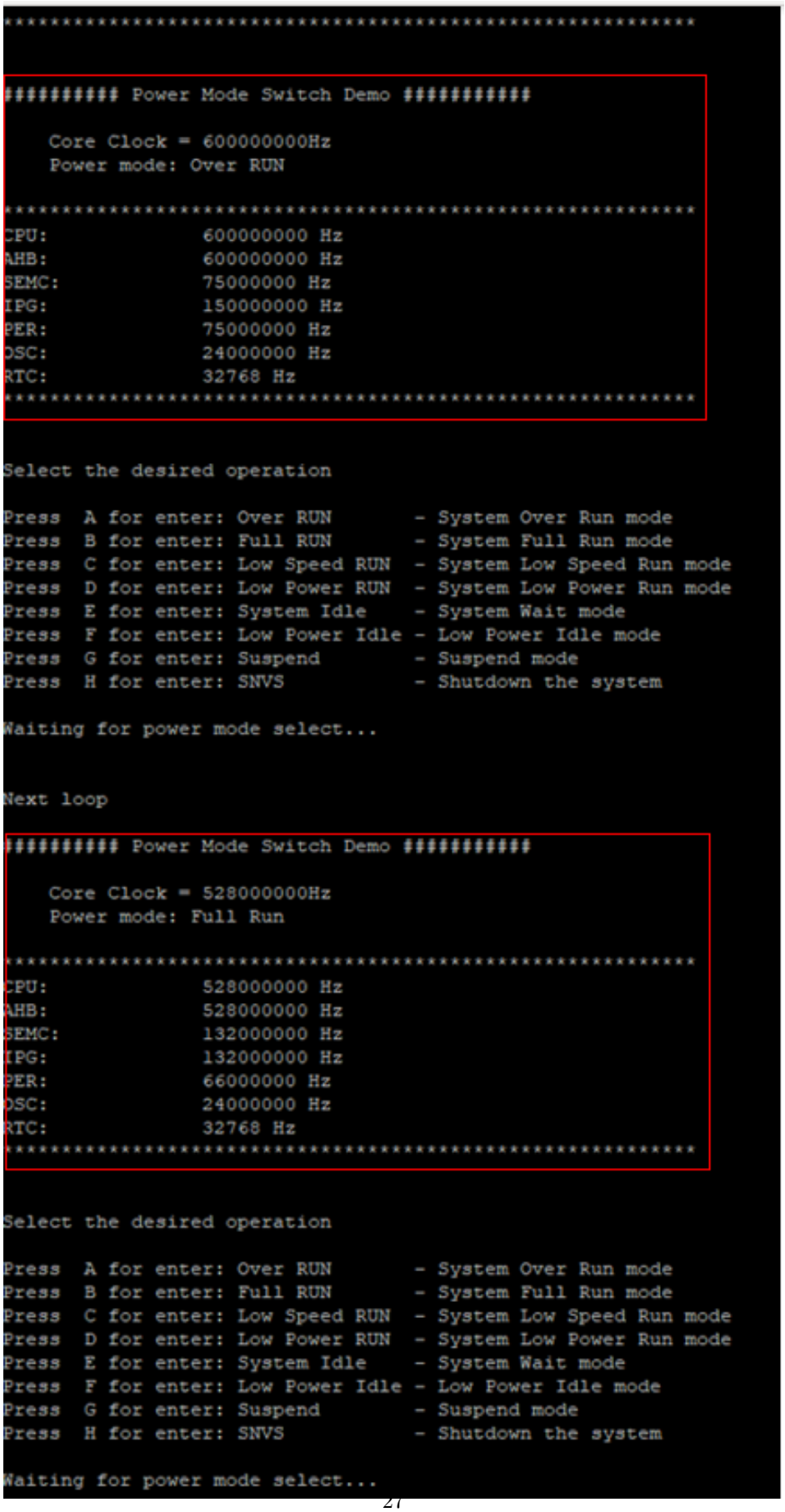


Figure 13: Power Mode application working on the primary client application address space

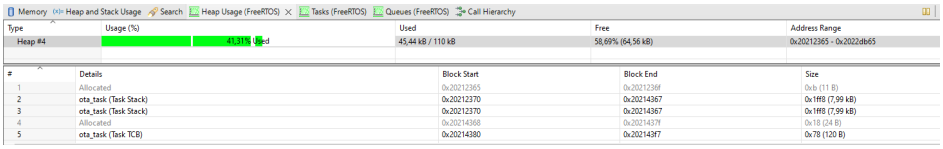


Figure 14: Heap Memory Utilized for the Application

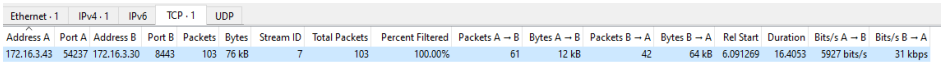


Figure 15: Server-Client Conversation Statistics

server's response, which can proceed with appropriate logical steps to be performed. Notably, the OTA task relinquished control to a higher-priority TCP thread upon initiating server-client communication.

9.3 lwIP Statistics and Packet Behavior:

Enabling lwIP statistics revealed no significant issues with packet behaviour, as evidenced by Tables 6 and 7. Conversation statistics, depicted in Figure 15, highlighted the exchange of approximately 70KB of packets between the server and client throughout the connection lifecycle. This exchange included TLS handshake messages and the sequence of request and response exchanges for downloading a 38KB artefact. The remaining data accounted for the overhead associated with TLS and TCP protocols. Moreover, the behaviour observed in the tables is consistent with actual behaviour, where the ethernet link is used to transmit a few packets of information to the server. At the same time, the client receives quite a lot of information from the server. Furthermore, some packets are dropped, probably resulting from protocol errors, as they might not be of the expected format. This is not a cause for concern as this lies within an acceptable range. However, this might be worth looking into during periods of heavy congestion.

Factors	**LINK**	**ETHARP**	**IP**	**UDP**	**TCP**
xmit	38	2	36	2	34
recv	307	143	161	118	27
fw	0	0	0	0	0
drop	0	3	16	0	0
lenerr	0	0	0	0	0
chkerr	0	0	0	0	0
rterr	0	0	0	0	0
proterr	0	3	0	0	0
opterr	0	0	0	0	0
err	0	0	0	0	0
cachehit	0	33	0	2	27

Table 6: LWIP Protocol Stats

System Factors	avail	used	max	err
MEM_HEAP	22528	60	1688	0
MEM UDP_PCB	6	1	1	0
MEM TCP_PCB	10	1	1	0
MEM TCP_PCB_LISTEN	6	0	0	0
MEM TCP_SEG	22	0	2	0
MEM REASSDATA	2	0	0	0
MEM FRAG_PBUF	15	0	0	0
MEM NETBUF	2	0	0	0
MEM NETCONN	4	1	1	0
MEM TCPIP_MSG_API	8	0	0	0
MEM TCPIP_MSG_INPKT	8	0	6	0
MEM SYS_TIMEOUT	10	7	7	0
MEM PBUF_REF/ROM	15	0	0	0
MEM PBUF_POOL	5	0	0	0
SYS_SEM	-	1	1	0
SYS_MUTEX	-	2	2	0
SYS_MBOX	-	2	2	0

Table 7: LWIP System Stats

Compile Time Optimization	Binary Size(KB)	mbedTLS Size(KB)
-O1	347.26	97.87
-O2	344.15	95.62
-O3	376.9	110.53
-Os	310.84	86.27
-Od	351.86	114.56

Table 8: Compile Time Optimizations for the client application

10 Discussion & Conclusion:

Throughout the meticulously planned execution of this assignment, several challenges surfaced. Initially, the endeavour encountered hurdles as fundamental as establishing the server, requiring a nuanced understanding of the Spring Boot application and debugging skills within the Maven build system. Additionally, at the project's inception in February, the latest branch lacked a user interface for debugging the mocked connection of device-side APIs, necessitating an older version utilizing a deprecated iteration of Vaadin, the current UI library for the application dashboard.

Considerable time was devoted to configuring the server to support HTTPS (HTTP + TLS) to ensure secure communication between server and client. Furthermore, the client utilized the coreHTTP library to send requests and parse responses, albeit with the drawback of being unable to support chunked transfer responses from the Hawkbit server. Consequently, additional configuration of the Hawkbit server was imperative to accommodate content-length-based requests, as discussed in section 7.3. This adjustment was necessitated by the inability to allocate a contiguous memory block for the requested artefact, requiring multiple requests of varying sizes to be made to the server, with the client subsequently writing them onto flash memory.

Moreover, the client application encountered challenges booting with the new image after a flash reboot, necessitating significant time investment in debugging and liaising with the NXP hardware support team to rectify the issue. Notably, a drawback of the current client application state lies in the lack of logic to persist the new bootable image in memory. Although the new image is recognized and booted upon reboot, its identification as faulty triggers a flash erase. Due to time constraints and the availability of requisite logic from stakeholders, the prioritization of the logic to make the new bootable image persistent was deferred.

Furthermore, assigning static IP addresses to server and client emerges as a crucial measure to ensure

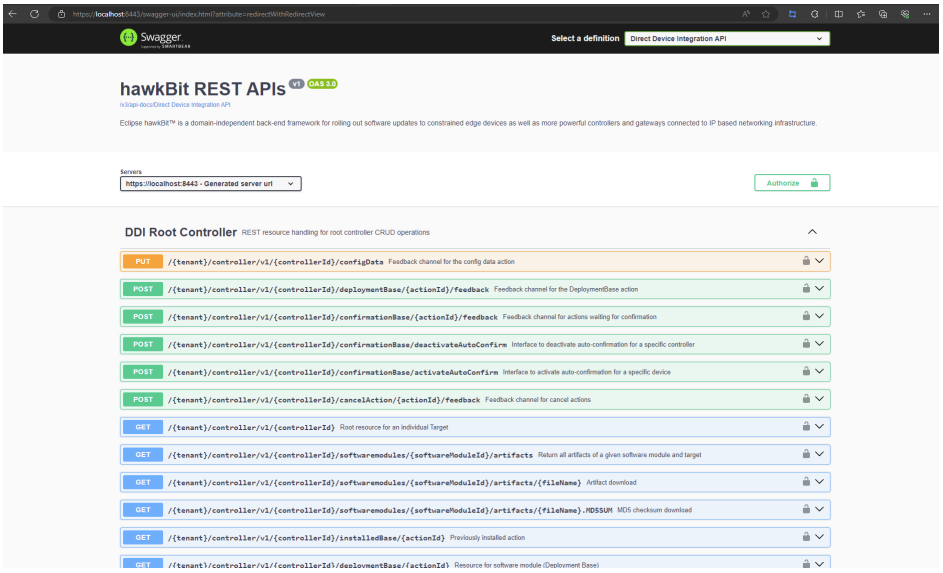


Figure 16: Hawkbit Version 0.5 UI

communication exclusively between traceable hosts, particularly pertinent for fleet updates, as stipulated by stakeholders for future rollouts. This implementation is deemed feasible and advantageous, given its ease of implementation and inherent security benefits. Furthermore, implementing static IP addresses for both the server and client will ensure consistent communication between traceable hosts, a pivotal requirement articulated by stakeholders for future rollouts. This approach streamlines communication and reduces the application's footprint by disabling unnecessary DHCP, DNS, and UDP protocols. The client undergoes DHCP negotiation, needlessly inflating the final image size and incurring overhead with each downtime occurrence; also, as evident from the table, the mbedtls library contributes to the most size for the final binary. It might be worth investigating a similar port of wolfSSL to compare the two alternatives and identify the most size-efficient implementation that can be integrated into the client application.

To achieve an optimal solution, it is imperative to adopt a differential update approach, retrieving only the necessary patches or updates instead of the entire file from the client side. By integrating delta updates, akin to those executed by SWUpdate, leveraging the ZChunk project, the application can efficiently download and apply differences between new and previous versions, conserving memory and time resources.

Additionally, the server's utilization of Vaadin 8, discontinued as of February 2022, necessitates an overhaul in UI development. The current development effort by the Eclipse community for Spring Boot 3.2.2 application with Vaadin 24 results in a simplistic UI, primarily beneficial for newcomers or testers, albeit devoid of the central management dashboard. Existing frameworks utilizing Hawkbit operate in daemon mode or feature a separate front end for the server dashboard.

Lastly, concerning the server's infrastructure, while the H2 database sufficed for assessment purposes, it proved inadequate for production scenarios due to data loss upon server session termination. Thus, deploying an SQL server with a RabbitMQ instance for message passing becomes imperative to ensure persistent deployment details are locally accessible on disk. Although setting up the corresponding Docker container for version 0.4 initially posed challenges, the appropriate configuration of the Dockerfile ultimately facilitated a functional setup. If the application were to be deployed on high-performance microcontrollers, such as the iMXRT1170, featuring dual cores (M4 and M7), the FreeRTOS application could be optimized. A separate task could be allocated for authorization and identity management using mbedtls and mapped to an individual core to enhance security. Additionally, a distinct task within the main application could utilize the other available core via core affinity configuration, enabling symmetric or asymmetric multiprocessing as per design requirements.

The application is configured to execute a predetermined set of requests to the server sequentially, assuming the server is operational and holds a valid update file for the target device. However, in realistic scenarios, this assumption is often improbable. To address this, low-power support features can be enabled

through the FreeRTOS kernel, ensuring the client polls the server at configured intervals. As firmware binaries are infrequently sent to clients, especially during deployment, the IdleTask hook function part of the FreeRTOS kernel can be reconfigured to enter a deep power-saving state until an interrupt occurs or a task must transition into a Ready state.

Furthermore, due to time constraints, comprehensive testing of the application has not been conducted, and only system tests have been performed. During client application design, unpredictable server responses were observed under high sequential request loads, requiring temporary request retries up to a maximum limit. However, this approach is not scalable and necessitates better handling in the client application.

Moreover, while some code size and efficiency optimisations have been implemented based on corresponding documentation, integrating the Google Test framework for code coverage and memory profiling would be highly beneficial. This framework aids in identifying potential dead code compiled due to incorrect or unnecessary configurations. Nonetheless, porting the project into C++ and further abstracting hardware-specific calls and multiple stub implementations within the assignment's timelines proved unfeasible.

References

[1] Northern.tech AS. *Mender Documentation*. 2023. URL: `https://docs.mender.io/` (visited on 02/12/2024).

[2] Stefano babic. *Embedded Software Update Documentation*. 2023. URL: `https://sbabic.github.io/swupdate/` (visited on 02/10/2024).

[3] Chiradeep BasuMallick. *What Is IPv6*. 2022. URL: `https://www.spiceworks.com/tech/networking/articles/what-is-ipv6/` (visited on 04/14/2024).

[4] *FOTA Design for SBL and SFW*. Tech. rep. AN13460. NXP Semiconductors, 2021. URL: `https://www.nxp.com/docs/en/application-note/AN13460.pdf`.

[5] Eclipse Foundation. *Hawkbit Documentation*. 2019. URL: `https://eclipse.dev/hawkbit/whatishawkbit/` (visited on 02/07/2024).

[6] FreeRTOS. *coreHTTP Documentation*. URL: `https://www.freertos.org/http/index.html` (visited on 04/15/2024).

[7] FreeRTOS. *coreJSON Documentation*. URL: `https://www.freertos.org/json/index.html` (visited on 04/15/2024).

[8] FreeRTOS. *FreeRTOS kernel Documentation*. 2017. URL: `https://www.freertos.org/RTOS.html` (visited on 03/15/2024).

[9] FreeRTOS. *FreeRTOS-plus-TCP Tutorial*. 2023. URL: `https://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_TCP/TCP_Networking_Tutorial.html` (visited on 02/20/2024).

[10] *FreeRTOS-coreHTTP support for chunked transfer encoding*. 2024. URL: `https://forums.freertos.org/t/how-to-enable-transfer-encoding-chunked-respose/18909/3` (visited on 04/20/2024).

[11] *Hawkbit Tutorial EclipseCon Europe 2019*. 2019. URL: `https://stefbehl.github.io/hawkbit-101/#/` (visited on 03/15/2024).

[12] *How to Enable Boot from QSPI Flash*. Tech. rep. AN12108. NXP Semiconductors, 2019. URL: `https://www.nxp.com/docs/en/application-note/AN12108.pdf`.

[13] *How to Use Flash Remapping Function*. Tech. rep. AN12255. NXP Semiconductors, 2018. URL: `https://www.nxp.com/docs/en/application-note/AN12255.pdf`.

[14] *i.MX RT1060 Crossover Processors for Industrial Products - Datasheet*. Tech. rep. MXRT1060IEC. NXP Semiconductors, 2022.

[15] *i.MX RT1060X Processor Reference Manual*. Tech. rep. IMXRT1060XRM. NXP Semiconductors, 2022.

[16] *i.MX RT1xxx – Ethernet Capabilities and PHY Connection*. Tech. rep. AN14251. NXP Semiconductors, 2024. URL: `https://www.nxp.com/docs/en/application-note/AN14251.pdf`.

[17] *Implement Second Bootloader on i.MX RT10xx Series*. Tech. rep. AN12604. NXP Semiconductors, 2023. URL: `https://www.nxp.com/docs/en/application-note/AN12604.pdf`.

[18] *LuaHTTP parser for coreHTTP*. 2024. URL: https://github.com/alis-is/lua-corehttp/tree/3c41f0b73cf2727b3bea384806f45697697675cd (visited on 04/20/2024).

[19] *lwIP Wiki*. URL: https://lwip.fandom.com/wiki/LwIP_Wiki (visited on 03/05/2024).

[20] *mbedTLS Documentation*. URL: https://mbed-tls.readthedocs.io/en/latest/kb/how-to/mbedtls-tutorial/ (visited on 03/05/2024).

[21] mirzak. *Updating Embedded Linux Devices: Mender*. 2018. URL: https://mkrak.org/2018/02/09/updating-embedded-linux-devices-mender/ (visited on 02/14/2024).

[22] mirzak. *Updating Embedded Linux Devices: SWUpdate*. 2018. URL: https://mkrak.org/2018/01/26/updating-embedded-linux-devices-part2/ (visited on 02/07/2024).

[23] *Mongoose Documentation*. 2024. URL: https://mongoose.ws/documentation/ (visited on 02/09/2024).

[24] *OTA Storage Structure on RT1010*. Tech. rep. AN13505. NXP Semiconductors, 2022. URL: https://www.nxp.com/docs/en/application-note/AN13505.pdf.

[25] *wolfSSL*. 2024. URL: https://www.wolfssl.com/products/wolfssl/ (visited on 03/15/2024).

A Code Snippets:

The server-side modification to add content length information from the server side to support the coreHTTP configuration included the addition of the Content length filter class as depicted by 17 into the RESTful class of the server by adding the code as performed in 19.

```
import com.fasterxml.jackson.databind.ObjectMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.web.util.ContentCachingResponseWrapper;

import javax.servlet.*;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

...
@Component
public class ContentLengthFilter implements Filter {

    @Autowired
    private ObjectMapper objectMapper;

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
            throws IOException, ServletException {
        ContentCachingResponseWrapper responseWrapper = new ContentCachingResponseWrapper((HttpServletResponse) response);
        chain.doFilter(request, responseWrapper);
        responseWrapper.setHeader(name:"Content-Length", String.valueOf(responseWrapper.getContentAsByteArray().length));
        responseWrapper.copyBodyToResponse();
    }
}
```

Figure 17: Content Length Filter Class

A sample code for the client-side application to download the artefact from the server is depicted in 20 and 21. GitHub URL contains the complete client-side application with the link to the configured setup to replicate the test environment. The program and configuration files are too large to be included in the report.

```
# User Security
spring.security.user.name=admin
spring.security.user.password={noop}admin
spring.main.allow-bean-definition-overriding=true
# Http Encoding
server.servlet.encoding.charset=UTF-8
server.servlet.encoding.enabled=true
server.servlet.encoding.force=true
# DDI authentication configuration
hawkbit.server.ddi.security.authentication.anonymous.enabled=false
hawkbit.server.ddi.security.authentication.targettoken.enabled=false
hawkbit.server.ddi.security.authentication.gatewaytoken.enabled=false
# Optional events
hawkbit.server.repository.publish-target-poll-event=false
# Need to run as root to use port 443
server.hostname=172.16.3.26
server.port=8443
# Overriding some of hawkbit-artifactdl-defaults.properties is required
hawkbit.artifact.url.protocols.download-http.protocol=https
hawkbit.artifact.url.protocols.download-http.port=8443
server.security.require-ssl=true
server.use-forward-headers=true
server.forward-headers-strategy=FRAMEWORK

server.ssl.key-store=localhost.jks
server.ssl.key-store-password=hawkbit
server.ssl.key-password=hawkbit
server.ssl.key-store-password=hawkbit
## Configuration for DMF/RabbitMQ integration
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
spring.rabbitmq.virtual-host=/
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
server.ssl.protocol=TLS
server.ssl.enabled-protocols=TLSv1.2,TLSv1.3
# New Users can be added via :
#hawkbit.server.im.users[0].username=hawkbit #hawkbit.server.im.users[0].password={noop}isAwesome!
#hawkbit.server.im.users[0].firstname=Eclipse #hawkbit.server.im.users[0].lastname=HawkBit
#hawkbit.server.im.users[0].permissions=ALL
springdoc.show-oauth2-endpoints=true
springdoc.api-docs.version=openapi_3_0
logging.level.org.springframework.web=DEBUG
springdoc.show-login-endpoint=true
springdoc.packages-to-scan=org.eclipse.hawkbit.mgmt,org.eclipse.hawkbit.ddi
springdoc.swagger-ui.oauth2RedirectUrl=/login/oauth2/code/suite
springdoc.paths-to-exclude=/system/**
logging.level.org.eclipse.hawkbit.ddi=DEBUG
```

Figure 18: Hawkbit Configuration

```
/*
 * The Content-Length filter is used to set the Content-Length header in the HTTP Response
 */
@Bean
public FilterRegistrationBean<ContentLengthFilter> contentLengthFilter() {
    FilterRegistrationBean<ContentLengthFilter> registrationBean = new FilterRegistrationBean<>();
    registrationBean.setFilter(new ContentLengthFilter());
    registrationBean.addUrlPatterns("/*"); // Apply filter to all URLs
    return registrationBean;
}
```

Figure 19: Content Length Filter Added to the Rest Configuration Class

```c
int https_client_ota_download(const char *host, const char *fPath, uint32_t dstAddrPhy, size_t dstSize, size_t *fSize)
{
    NetworkContext_t coreHttp_NetCtx = {&tlsDataParams.ssl};
    TransportInterface_t ti = {.recv = coreHttp_recv, .send = coreHttp_send, .pNetworkContext = &coreHttp_NetCtx};
    struct OtaHttpConf httpConf = {.ti = &ti, .dataBuf = https_buf, .dataBufSize = sizeof(https_buf), .hostName = host};
    int ret;
    unsigned char md_net[32], md_flash[32];
    mbedtls_sha256_context sha256_ctx;
    mbedtls_sha256_init(&sha256_ctx);
    mbedtls_sha256_starts_ret(&sha256_ctx, 0);
    uint32_t addr_phy   = dstAddrPhy;
    uint32_t file_size  = 0;
    size_t storage_size = dstSize;
    PRINTF("\nGetting size of requested file '%s'\n", fPath);
    file_size = OtaHttp_GetFileSize();
    if (ret != HTTPSuccess)
    {
        PRINTF("Failed to get file size, ret=%d\n", ret);
        return EXIT_FAILURE;
    }

    PRINTF("Determined file size is %u bytes\n", file_size);
    if (file_size > storage_size)
    {
        PRINTF("Requested file TOO BIG! file_size=%d storage_size=%d\n", file_size, storage_size);
        return EXIT_FAILURE;
    }
    PRINTF("Starting download of %u bytes with block size of %u bytes\n", file_size, OTA_HTTP_BLOCK_SIZE);
    /* Read the file in chunks from the downloaded file */
    uint32_t bytes_recvd = 0;
    while (bytes_recvd < file_size)
    {
        size_t flash_offset = addr_phy + bytes_recvd;
        status_t status;
        size_t remains      = file_size - bytes_recvd;
        size_t chunk        = (remains < OTA_HTTP_BLOCK_SIZE) ? remains : OTA_HTTP_BLOCK_SIZE;
        int cnt;
        /* Read the chunk from the downloaded file */
        cnt = OtaHttp_DownloadArtifact(&httpConf, bytes_recvd, chunk, https_buf);

        PUTCHAR('.');
        if (cnt < 0)
        {
            PRINTF("\nHTTPS CLIENT::https_client_ota_download()::HTTP File Request at offset %u failed with %d", bytes_recvd, cnt);
            break;
        }
        if (cnt != chunk)
        {
            PRINTF("UNEXPECTED size read at offset %u: %u read, %u expected.\n", bytes_recvd, cnt, chunk);
            break;
        }

        mbedtls_sha256_update(&sha256_ctx, https_buf, cnt);
        /* Flash erase is done on the fly with download since erasing large portion
        * of flash while executing from it would cause other system tasks to starve
        * (e.g. TCP connection)
            status = mflash_drv_sector_erase(flash_offset);
            if (status != kStatus_Success)
            {
                PRINTF("FAILED to erase sector at offset %u\n", addr_phy);
                break;
            }
        }
        PRINTF("\nHTTPS CLIENT::https_client_ota_download()::Data before writing to flash:");
        for (size_t i = 0; i < cnt; i++) {
            PRINTF("%02x", https_buf[i]);
        }
        PRINTF("\n");
        flash_program(flash_offset, https_buf, cnt);

        bytes_recvd += cnt;
    }
    PRINTF("\nDownload loop completed with size %u, expected %u\n\n", bytes_recvd, file_size);
    if (bytes_recvd != file_size)
    {
        PRINTF("FAILED to download requested file.\n");
        return EXIT_FAILURE;
    }
    // Message Digest check
    mbedtls_sha256_finish(&sha256_ctx, md_net);
    flash_sha256(addr_phy, file_size, md_flash);
    if (memcmp(md_net, md_flash, sizeof(md_flash)) != 0)
    {
        PRINTF("FAILED to compare MD.\n");
        return EXIT_FAILURE;
    }
    if (fSize)
    {
        *fSize = file_size;
    }
    return SUCCESS;
}
```

Figure 20: Client Side Download Artefact

```c
int OtaHttp_DownloadArtifact(const struct OtaHttpConf* cfg, uint32_t offset, uint32_t size, void* data)
{
    HTTPRequestInfo_t request;
    HTTPRequestHeaders_t requestHeaders;
    HTTPResponse_t response;
    int ret;
    char* authHeader;
    char rangeHeader[50];

    if (!(controllerId && securityToken && cfg && deploymentBaseUrl && downloadArtifactUrl))
    {
        return -HTTPInvalidParameter;
    }

    if (size == 0)
        return 0;

    if (cfg->dataBufSize <= size)
    {
        return -HTTPInsufficientMemory;
    }

    memset(&request, 0, sizeof(request));
    memset(&requestHeaders, 0, sizeof(requestHeaders));
    memset(&response, 0, sizeof(response));

    request.pPath = downloadArtifactUrl;
    request.pathLen = strlen(request.pPath);
    request.pHost = hostname;
    request.hostLen = strlen(request.pHost);
    request.pMethod = HTTP_METHOD_GET;
    request.methodLen = sizeof(HTTP_METHOD_GET) - 1;
    request.reqFlags = HTTP_REQUEST_KEEP_ALIVE_FLAG;

    requestHeaders.pBuffer = cfg->dataBuf;
    requestHeaders.bufferLen = cfg->dataBufSize;
    response.pBuffer = cfg->dataBuf;
    response.bufferLen = cfg->dataBufSize;

    ret = HTTPClient_InitializeRequestHeaders(&requestHeaders, &request);
    PRINTF("\nInitializing request headers... Statuss = %d", ret);
    if (ret != HTTPSuccess)
    {
        return ret;
    }

    authHeader = malloc(strlen("TargetToken ") + strlen(securityToken) + 1);
    if (authHeader == NULL)
    {
        return -HTTPInvalidResponse;
    }

    sprintf(authHeader, "TargetToken %s", securityToken);
    ret = HTTPClient_AddHeader(&requestHeaders, "Authorization", strlen("Authorization"), authHeader, strlen(authHeader));
    free(authHeader);
    if (ret != HTTPSuccess)
    {
        return -HTTPInvalidResponse;
    }

    sprintf(rangeHeader, "bytes=%u-%u", offset, offset + size - 1);
    ret = HTTPClient_AddHeader(&requestHeaders, "Range", strlen("Range"), rangeHeader, strlen(rangeHeader));
    if (ret != HTTPSuccess)
    {
        return -HTTPInvalidResponse;
    }

    ret = HTTPClient_Send(cfg->ti, &requestHeaders, NULL, 0, &response, HTTP_REQUEST_TIMEOUT_MS);
    if (ret != HTTPSuccess)
    {
        PRINTF("\nHTTPClient_Send failed with status = %d", ret);
        return -HTTPInvalidResponse;
    }

    if (response.contentLength > size)
    {
        PRINTF("\nInvalid content length: %d", response.contentLength);
        return -HTTPInvalidResponse;
    }
    else
    {
        memcpy(data, response.pBody, response.contentLength);
        return response.contentLength;
    }
}
```

Figure 21: HTTP Client Code