Group 22
DD1367 Software Engineering in Project Form

Fine-grained Time Synchronization for Embedded Industrial Devices

24th of May 2024

Company: ABB
Company contact person: Ognjen Dobrijevic
KTH mentor: Roberto Guanciale
Group: 22
Group members: Filip Dimitrijevic (filipdi@kth.se), Vilhelm Prytz (vprytz@kth.se), Marcus Carlbom (mcarlbo@kth.se), Oscar Jonsson (oscjonss@kth.se), Erik Smit (esmit@kth.se), Hugo Larsson Wilhelmsson (hugolw@kth.se), Albin Wallenius Woxnerud (alww@kth.se), Jack Gugolz (jgugolz@kth.se)
Number of pages: 45

Table of Contents

Planning & Organizing

We have decided to divide the team into different focus areas. Since our project team and work will align with the principles of agile and scrum methodologies, the titles and responsibilities of each team member are well-defined. Some titles are not traditional scrum roles, but we see the need for them in this project. All team members will contribute to the development process, however, some members will have specific responsibility for an area.

The Team

Filip Dimitrijevic - Scrum Master
Jack Gugolz - Product Owner
Vilhelm Prytz - Technical Lead (Architect)
Albin Wallenius Woxnerud - Software Lead
Erik Smit - Report Lead
Hugo Larsson Wilhelmsson - Quality Assurance Lead
Oscar Jonsson - Development Process
Marcus Carlbom - Development Process

Scrum master is responsible for leading and guiding the entire development team. They act as a facilitator and coach, rather than a traditional manager or team leader. Overseeing the project, ensuring the team works efficiently and meets goals and deadlines. Scrum master also has comprehensive communication with all team members as well as our contact person from ABB and mentor from KTH.

Product owner is key in defining the product's features and deciding on release dates and content. In our project, release dates are corresponding deadline dates with regard to the course. They continuously interact with stakeholders and the team to clarify requirements and accept or reject work results. The Product owner also prioritizes the work in the backlog to guide the team on what to work on next.

Technical lead is not a traditional scrum role, but we see the need for it. Technical lead handles the technical aspects of the project and development, such as general technical architecture. Planning and monitoring technical solutions, making sure technical goals are met, by using the right technologies and tools.

Software lead focuses on leading and managing software development within our project. Responsible for defining software architecture, and development strategy, and ensuring the project follows satisfying software practices.

Development supporters are members of the development team and have a supportive role in the project process. Contribute to various tasks such as testing, quality assurance, documentation, or technical tasks needed to support the development of the project.

Quality assurance lead is a critical technical risk identifier that ensures that our client's design adheres to the formal specifications of the NTP protocol, as well as verifies NTP security threats. It's essential to verify that the client effectively communicates with public NTP servers. That is the specific responsibility of the quality assurance lead.

Report lead is not a traditional scrum role, but we see a need for it since report documentation is a comprehensive part of the project. Report leads are mainly focusing on deadlines regarding the report and that they are met. They oversee the content of the report in various stages and guide the rest of the team in report documentation.

Project Summary

The goal of the project is to develop and test a prototype implementation of an NTP client that is tailored to a selected embedded-device platform, has a small memory footprint, and can achieve the needed clock accuracy. We are also supposed to measure the clients' performance and resource consumption.

Need for Time Synchronization

There is a substantial need for time synchronization. This is because time synchronization (time-sync, for short) is a vital part of many industrial applications [1], which require local clocks of networked industrial devices not to differ from each other by more than a few milliseconds or even microseconds. It is commonly the responsibility of communication protocols such as Network Time Protocol (NTP) or Precision Time Protocol (PTP) to minimize time differences among distributed clocks. Several time-sync use cases rely on fine-grained data timestamping with clock accuracies of less than 10 ms.

The code for the project will be written in the C language to ensure compatibility with any hardware capable of running FreeRTOS. Specifically, we will use a MPS2 Cortex-M3 AN385 platform emulated using QEMU.

Proposed Methodology

Our ambition is to develop an NTP client on FreeRTOS (Free Real-Time Operating System) and it will be designed to run FreeRTOS operating system on any hardware capable of running FreeRTOS. FreeRTOS is a real-time operating system (RTOS), which differs from a time-sharing operating system (such as Unix and Windows). An RTOS is event-driven which in essence means that the OS handles tasks based on its priorities, while a time-sharing OS handles tasks based on interrupts and scheduling [2].

We will run demonstrations by implementing FreeRTOS on an emulator. This emulator will be set up by using QEMU, which is a generic and open-source machine emulator, and

visualizer. QEMU supports various architectures, making it a powerful tool for cross-platform development and testing (in our case, an ARM platform is emulated). Therefore the emulator will be compatible with FreeRTOS. A FreeRTOS task will then be constructed to handle the NTP client. This task will be responsible for sending requests to NTP servers, receiving responses, and updating the local clock. This task will also be scheduled to run at regular intervals, which addresses the need for NTP synchronization. In accordance with the project description, the network and servers are already set up and thus we will only need to develop the client. The emulator will however need to be configured to work with the existing network.

FreeRTOS has an SNTP library, but it is specifically for core SNTP (Simple Time Network Protocol) [3]. As we are specifically targeting a time accuracy of less than 10 milliseconds in local area networks, and SNTP has a targeted time accuracy of more than 10 milliseconds, we can not use this FreeRTOS library.

Furthermore, we will also implement a timekeeping mechanism. Since it is not guaranteed that the board and chip that we implement our NTP client on has a system clock or an RTC (Real-Time Clock), we intend to use tick counts to implement a timekeeping mechanism (the system ticks 1000 times per second and using this we can calculate the difference between saved timestamps). In a more advanced system, using RTC would be more suitable. Another issue is to construct the actual communication, meaning in what way or form we send our data between the devices. One way to approach this issue is to use UDP (User Datagram Protocol) [4], which is a standardized protocol that is used to quickly transfer data in packets between two or more devices. NTP, in its original design, employs UDP as the underlying transport protocol. While UDP lacks certain reliability and security features found in other protocols these limitations can be offset by using the NTS (Network Time Security) protocol [5]. However, when considering the specific requirements and scope of this project UDP is adequate, and consequently, NTS will not be implemented in this context as the simplicity and efficiency of UDP is well aligned with our objectives. To be able to meet the clock accuracy of 10 ms on local area networks, we also need to implement a clock synchronization algorithm. This algorithm includes three sub-algorithms which are the selection, cluster, and combine algorithms. A well-implemented algorithm will result in accurate and reliable clock adjustment.

We propose to establish a network setup using both TCP and UDP protocols, utilizing QEMU to emulate the MPS2 Cortex-M3 AN385 platform within a Docker-based VM. Initially focusing on a TCP Echo Client setup to validate network communications, we will progress to configuring the network to support UDP. This setup will facilitate the functioning of our FreeRTOS tasks and enable the accurate operation of our NTP client within the FreeRTOS environment.

In order to effectively manage the workload, we have defined two primary approaches. The first track will concentrate on the skeleton code (RFC5905), and the second will focus on the

primary implementation in FreeRTOS. The RFC5905 track will be dedicated to running and compiling the skeleton code within our architectural framework, which incorporates Docker and Alpine Linux. Meanwhile, the FreeRTOS track will aim to integrate all data types, structs, and foundational functions from the RFC 5905 skeleton. This strategy allows us to leverage the functional implementations from RFC 5905 and effectively integrate them into the FreeRTOS environment.

Project Plan

I. 24 January - Project Report 1: Everyone should have an understanding of the project goals, and we should have organized and planned the project.
II. 29 March - Project Report 2: Focus on analysis, design, and system development.
III. 9 April - Half-time Presentation: Present the requirement analysis, solution design, system development, and change management.
IV. 30 April - Project requirements should be met.
V. 24 May - Project Report 3: Final changes should have been made.
VI. 31 May - Final Presentation: End of the project.

These deadlines correspond to specific assignments in the course DD1367. We must meet these deadlines as they are integral parts of our course assessment. Each deadline is designed to align with the course's structured progression, ensuring that we stay on track with both our project objectives and requirements.

Development Process Planning

1. Methodology - Agile Scrum
    - We aim to work with agile and scrum methodologies as earlier explained in the section *The Team*. Agile scrum is a methodology that is primarily used in software development since it provides flexibility and adaptability. It involves breaking down the project into small sprints. Our sprints can be divided with regard to corresponding deadlines for the report documentation. The sprints last for one to two weeks, and during that time the team will work agile with the product, in order to meet requirements and deliverables. The specific sprints are decided during our weekly meeting and are connected to the action points that are decided during the meeting. Key features of agile include regular adaptation to changing requirements, which gives us flexibility in the way we will develop and work with the project.
2. Task Allocation
    - Assigning specific tasks to team members based on the designated focus areas. The tasks will mainly be tracked via GitHub issues, especially regarding software development. Report leads will guide the team regarding report documentation and assign corresponding tasks. Specific roles in the team can be reviewed in section *The Team.*

- Regular team meetings for progress updates and coordination. The team will schedule internal meetings once per week. During these meetings, the team will also work together on the project and allocate new tasks if needed. Also, the team will have scheduled meetings once per week with a company contact person. Then there are meetings with the KTH contact person as well, some of these meetings are scheduled. Additional meetings with the KTH contact person can be scheduled if needed.

3. Milestone Planning
    - Establishing key milestones such as completion of initial design, prototype development, testing phases, and final implementation.
    - Setting tentative deadlines for each milestone to ensure timely progress.
4. Risk Management
    - An incorrect implementation of the NTP protocol could result in potential security threats, such as servers without authority can manipulate the time synchronization, and/or potential communication issues with public NTP servers.
        i. The NTP protocol already incorporates robust security measures such as Client-Server Authentication, Access Control Lists (ACLs), and Symmetric Key Authentication [6]. These features effectively mitigate security risks, provided that the NTP protocol is correctly implemented, a responsibility entrusted to our Quality Assurance lead for thorough verification.
        ii. Similarly, the Quality Assurance lead will diligently ensure that our design aligns with the protocol's specifications, reducing the risk of communication problems with public NTP servers. Continuous monitoring and adherence to protocol specifications will safeguard against potential issues in this regard.
    - Resource perspective can be a risk if we do not have enough time or people to meet the requirements.
        i. By defining clear team roles and responsibilities we minimize this risk.
        ii. Another solution would be to adapt the list of requirements by having more and less prioritized features.
    - If the algorithm is dependent only on one NTP server, we can not achieve time synchronization if something happens to the NTP server as we can not reach any NTP server at that point.
        i. A solution would be implementing multiple connections to the NTP server to introduce redundancy which would ensure access.
        ii. Another solution is to use anycast NTP servers for higher redundancy. Anycast is a network methodology where a single IP is shared across multiple devices (in our case, multiple NTP servers).
    - Network problems may cause the time synchronization process to fail.

    i. Implementing network monitoring and using redundant NTP servers allows us to understand when network problems arise and have possible solutions if they happen.
    ii. Ability to rely on the system clock for shorter periods to manage the temporary loss of network connectivity.
  - By working agile, we will continuously adapt and find solutions to new problems that arise during the project.

5. Quality Assurance and Testing
  - Implementing quality control measures throughout the development process.
  - Automated processes for testing and code linting can be managed through GitHub Actions.
    i. Using ClangFormat for linting C source code [7].
    ii. Configure GitHub actions to try compilation upon every commit push.

6. Resource Management
  - Allocating hardware and software resources effectively.
  - The source code for software development will be stored and version-controlled in a GitHub repository using Git.

Requirements

For clarity, we break the requirements down into two parts, one for general requirements, *Project/Business requirements*, and one for more specified requirements, *Specified/Functional requirements*.

REQUIREMENTS TRACEABILITY MATRIX								
Project Name: Fine-grained Time Synchronisation for embedded Industrial Devices								
Business Requirement Document BRD		Functional Specification Document FRD			Test Case Document			
Business Requirement ID#	Business Requirement / Business Use Case	Functional Requirement ID#	Functional Requirement / Use Case	Priority	Test Case ID#	Test Case Description	Execution Status	Defect ID
BR-NTP-DEV	NTP-Client	FR-REC-SEND-PACKETS	Recieve/Send NTP packets	High	TC#001	Verify if the NTP-client can recieve and send NTP packets	Pass	
		FR-ALGORITHMS	Implementation of clock mitigation algorithms	High	TC#002	Verify if the NTP-client correctly implement select, cluser and combine algorithms	Fail	DEF#001
		FR-CORRUPT-PACKETS	Handle corrupt packets	Medium	TC#003	Verify if the NTP-client can handle corrupt packets	Pass	
BR-COMP-EFF	Efficiency & Compatibility with embedded devices	FR-MEM-EFF	Memory efficient	Medium	TC#004	Verify if the NTP-client takes up less than 1 MB of space	Pass	
		FR-ASYNC-TASKS	Asynchronous task execution	High	TC#005	Verify if the device can perform other tasks while waiting for response.	Pass	
		FR-TIME-SYNC	Synchronise time	High	TC#006	Verify if the NTP time drift is less than 1 s.	Pass	
		FR-TIME-SYNC-PRECISION	High time synchronisation precision	High	TC#007	Verify if the NTP time drift is less than 10 ms	Fail	DEF#002
		FR-COMP-FREERTOS	Compatable with FreeRTOS	High	TC#008	Verify that the NTP-client runs on FreeRTOS	Pass	
		FR-TIMEKEEP	TImekeeping	High	TC#009	Verify if the device keeps track of its current time.	Pass	
BR-USABILITY	Maintanable and Readable code	FR-QA-TESTING	Code linting/formatting	Low	TC#010 TC#011	Verify if the code is formatted and linted correctly.	Pass	

Figure 1: Traceability matrix of the requirements.

Project/Business requirements

- **BR-NTP-DEV:** Build and develop the NTP client from the NTPv4 specification that supports retrieving time from NTP servers (of clock strata of Stratum 1 and Stratum 2 servers, see definition of these terms under Clock strata). Our client implementation should be based on the skeleton implementation from the NTPv4 specification (see RFC 5905).
- **BR-COMP-EFF:** Ensure that the NTP client is compatible with embedded Industrial devices (emulated environment MPS2 Cortex-M3 AN385 platform) that run on FreeRTOS and that it is time and memory-efficient.
- **BR-USABILITY:** Ensure that the code is readable and maintainable for potential users.

Specified/Functional requirements

- **FR-REC-SEND-PACKETS:** The system should be able to send and receive packets from more than three stratum 1/ stratum 2 servers.
- **FR-CORRUPT-PACKETS:** The system should be able to handle corrupt and broken messages.
- **FR-ALGORITHMS:** The NTP client should be able to use the clock mitigation algorithms from the skeleton implementation, such as mitigation of errors, selection algorithms, cluster algorithms, and combine algorithms. The client should leverage the algorithms to determine the best time from multiple NTP servers.
- **FR-COMP-FREERTOS:** Verify that the client runs on FreeRTOS.
- **FR-TIMEKEEP**: Implement a timekeeping mechanism.
- **FR-ASYNC-TASKS**: The system should be able to perform tasks during the time that it is waiting for a response from the NTP server.
- **FR-MEM-EFF**: The NTP client must be finely tuned for efficient memory utilization, with a strict constraint that limits its memory usage to a maximum of 1 megabyte (1 MByte) of total memory usage.
- **FR-TIME-SYNC**: The client must synchronize time with NTP servers, precision should be less than 1 s over local area networks.
- **FR-TIME-SYNC-PRECISION:** The client must achieve a high degree of time synchronization precision that guarantees a precision of less than 10 ms over local area networks.
- **FR-QA-TESTING**: Implementing quality control measures throughout the development process. Use GitHub Actions to handle automatic testing and code quality assurance.

Deliverables

The following four deliverables will be provided upon the project's completion:

I. Documented Source Code: The source code will be comprehensively documented and made available through a GitHub repository.
II. Test Reports: Detailed reports outlining the outcomes and findings from various testing phases.
III. Measurement Results Analysis: An analysis of the measurement results, providing insights into the performance and efficiency of the implemented solution.
IV. A final project report: A report including the requirements, if they are met, and how we solved them. The report will also include planning and organizing, analysis and design, system development, and system testing.

Analysis & Design

This is the proposed project design based on the analysis of the project.

1. Architecture Design

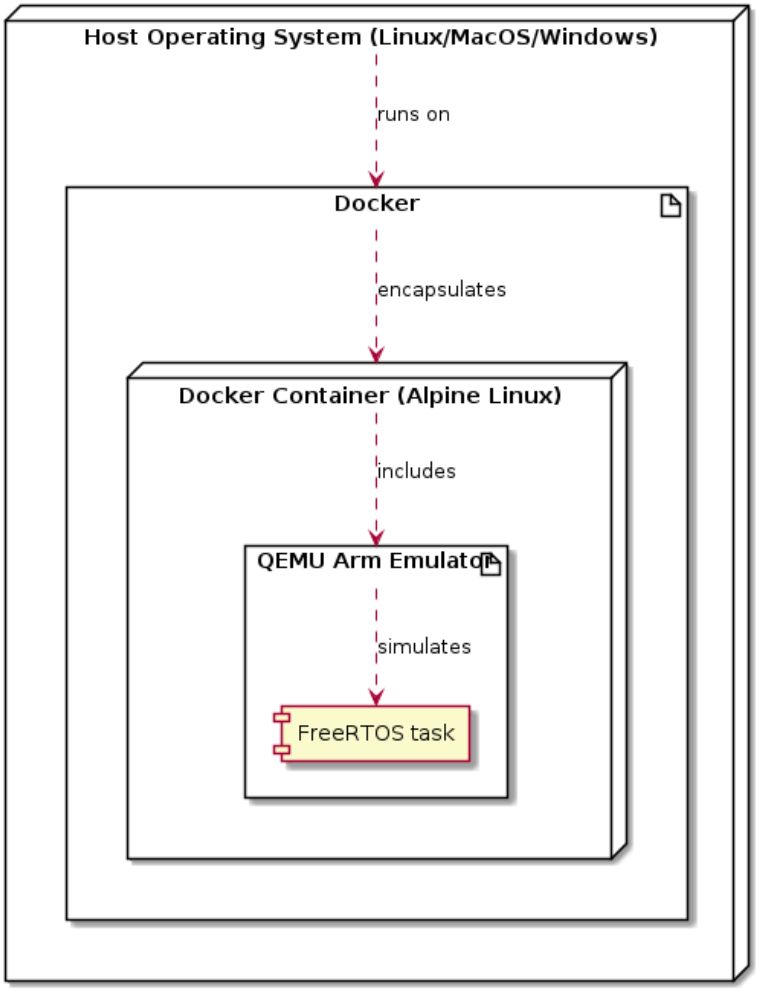

Figure 2: General technical overview of local development setup, UML deployment diagram.

The following is an explanation of the general technical overview of the local development setup, also seen in Figure 2.

I. Host Operating System Layer

The foundational layer of the architecture is the host operating system which can be any operating system with Docker support such as Windows, macOS, and Linux. This offers greater flexibility and allows the architecture to be implemented on a wide range of hardware and software environments independent of the host. The primary function of the host OS in this context is to provide a stable platform for running Docker which ensures that the development environment is consistent.

II. Docker Layer

Docker serves as the second layer of the architecture and is a powerful tool that enables the use of containers to encapsulate the following layers which provides easy isolation and resource management. It is configured to run Secure Shell Daemon, sshd on port 22, with port forwarding to port 2022 on the host OS (using Docker Compose). This setup provides access to the container environment. Docker is not a full virtual machine virtualization, hence the containers running inside Docker share the kernel from the host operating system.

III. Operating System within Docker Container

The third layer of the architecture, within the Docker container, is a set of libraries and tools derived from Alpine Linux which was chosen due to its lightweight and efficient nature. This leverages the kernel of the host OS while providing the Alpine Linux development environment. It is important to note that this setup is not fixed to Alpine Linux as any system that supports the GNU Arm Embedded Toolchain and QEMU can be substituted. This layer ensures that the necessary tools for development are available in a minimal and efficient environment, enhancing the overall performance and reducing the overhead on the host system. Although this layer does not directly execute the FreeRTOS task, it provides the tools that enable it to compile, run, and test the task in an emulated ARM environment that is similar to the target embedded system. It is inside the Docker container that the code is compiled (using Make and a Makefile). Within the repository with all the source code, the FreeRTOS source code is also made available using Git submodules. When the code is then compiled, it is compiled using the kernel from the FreeRTOS source code as well as all the FreeRTOS libraries required by our project.

IV. QEMU Arm Emulator Layer

The fourth layer of the architecture involves the use of QEMU within the Docker container. QEMU is an open-source emulator that simulates different hardware platforms. The QEMU Arm Emulator stays within the Docker container and simulates

the ARM processor architecture. This emulation is important to run ARM-compatible code within the development environment. By implementing this specific layer, there is no need for physical ARM-based hardware in the development process.

V. FreeRTOS on Simulated Hardware

The final and most crucial layer of the architecture is the FreeRTOS operating system running the QEMU-simulated hardware. This is where the actual code development, testing, and execution take place with a focus on the implementation of the NTP client for time synchronization.

2. Network Configuration

Docker provides built-in support for networking. Within the container, a TUN/TAP bridge is set up, allowing the QEMU virtual machine inside the Docker container to have network connectivity. This setup enables FreeRTOS to communicate with NTP servers over the network. By default, Docker employs Network Address Translation (NAT), which means that devices external to the host operating system will only recognize the IP address of the host OS. They will not see the container's IP address, even though the container operates within the host OS.

TCP and UDP Echo Client Demo for MPS2 Cortex-M3 AN385 emulated using QEMU

This section outlines the setup phase of networking configuration, separate from the development of the NTP client. This setup is necessary for demoing the NTP client.

The following was implemented in the setup phase of networking configuration. The FreeRTOS+TCP example demonstrates a TCP Echo Client that sends echo requests to an Echo Server and then receives the echo reply. The Echo Client is written in FreeRTOS and runs on the MPS2 Cortex-M3 AN385 platform emulated using QEMU. The setup and the specific demo require:

- Echo Client - The demo in the repository.
- Echo Server - An external echo server (such as Netcat).

Echo Client runs in the Virtual Machine (VM) and Echo Server runs on the host machine. The initial setup involves installing virtual machine software, in our case. It was set up the same way as our architecture design, within Docker and with QEMU.

For the network to function, the VM's network settings are configured to ensure the Echo Client can communicate with the Echo Server. This involves setting up network interfaces, and IP addresses, and ensuring proper routing within the VM. The Echo Server is launched

on the host, listening on a specified port, ready to receive and respond to echo requests sent by the client.

The demonstration is run by building the FreeRTOS TCP demo within QEMU and initiating a session where the Echo Client interacts with the Echo Server, testing the network communication capabilities of the setup. This setup serves as a practical example of configuring and testing networked applications in a simulated environment. It was essential to update the virtual machine's network settings to later support UDP, necessary for the proper functioning of FreeRTOS tasks.

After successfully implementing the TCP Echo Client as described, we moved forward with implementing a UDP client using a similar setup to leverage the established network configuration. Similar to the TCP example, this UDP implementation was designed to operate within the same FreeRTOS+TCP framework on the MPS2 Cortex-M3 AN385 platform emulated via QEMU.

The UDP client was set up to communicate with an external UDP server (we used this for testing), using the existing virtual machine and network setup. The process involved adjusting the networking configuration to handle UDP, in order to ensure that the virtual machine's network settings were updated to support UDP protocols, which is needed for FreeRTOS tasks.

3. Software Development

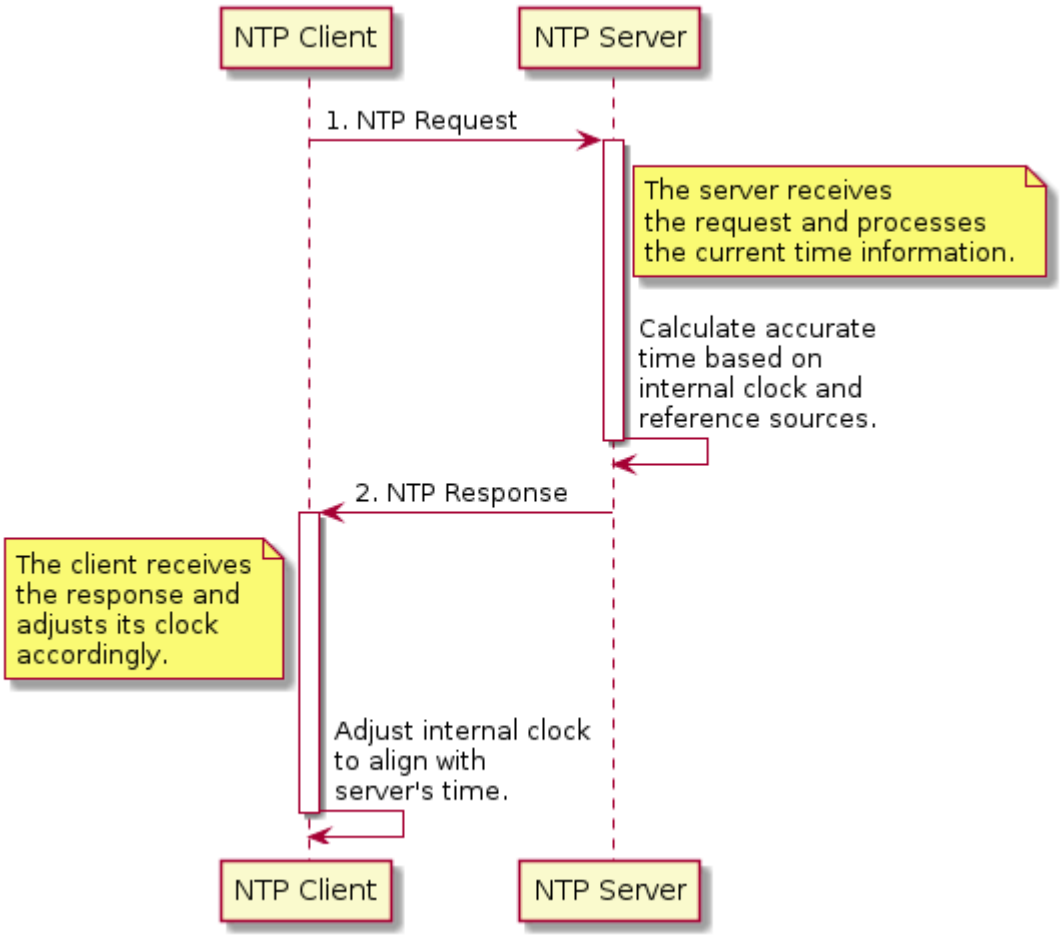

Figure 3: UML diagram shows the NTP time synchronization sequence between a client and a server.

Programming in the C language involves several steps. Implementing NTP client logic includes sending requests, receiving responses, and updating the clock. Figure 3 illustrates the synchronization sequence. Additionally, designing the timekeeping mechanism involves syncing the system clock with the time received from the NTP client.

Understanding the Network Time Protocol: An Overview

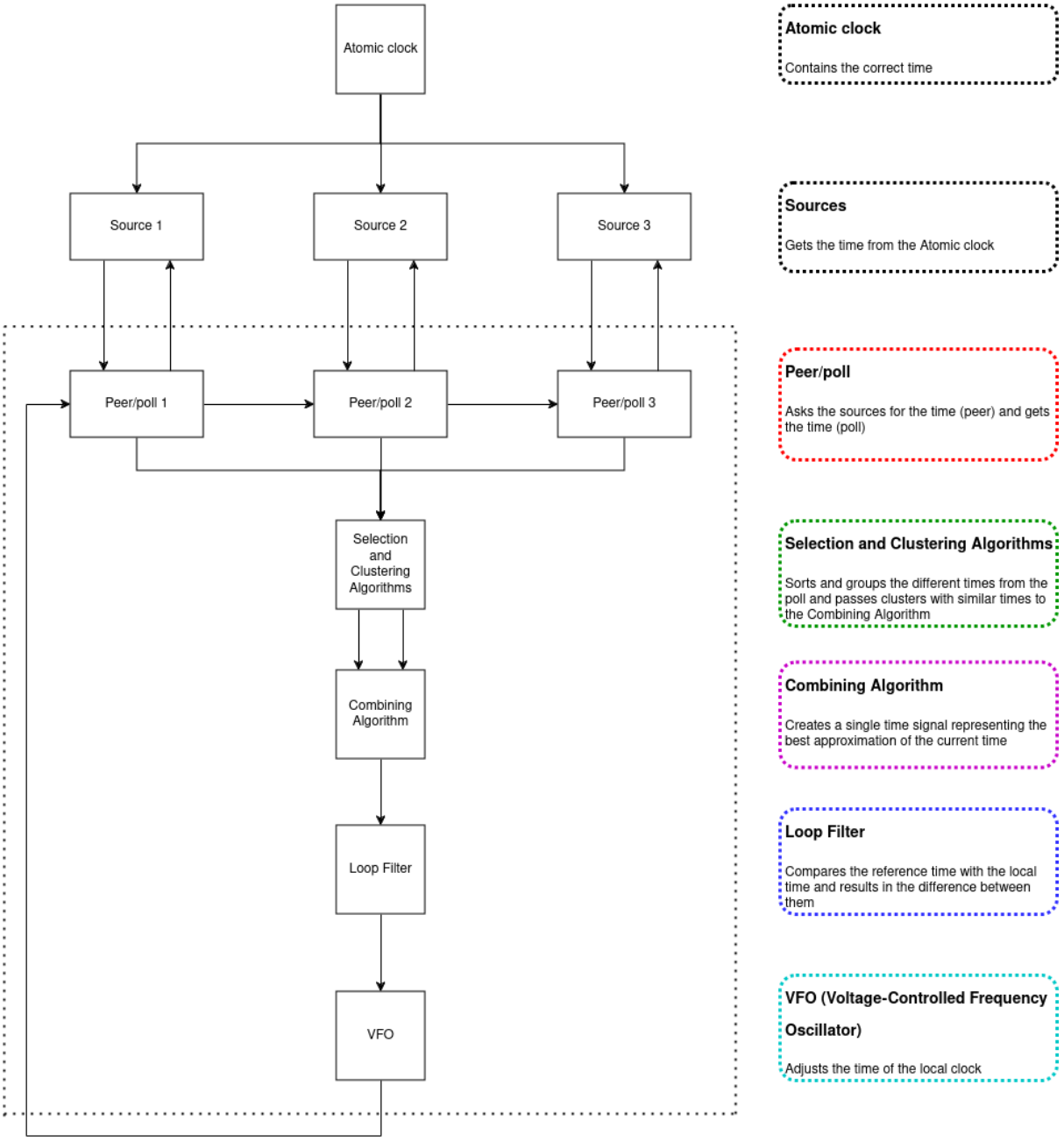

Figure 4: Data flow in a functional NTP system

Figure 4 describes the general data flow in an NTP system. The areas inside the dotted lines are the parts that have to deal with the project. Below is an explanation of each component in the data flow, shown in Figure 4. The following explanation is based on RFC 5905 [6].

Atomic Clock and Sources

The Atomic Clock is the clock with the exact or rather most accurate time compared to the other time sources. Other time sources include a hierarchy of NTP servers. These NTP servers are already implemented alongside a shared network through which they can

communicate with each other. This entails that our project can directly utilize this network of time sources/servers without any additional work.

Peer/Poll

The peer and poll stage is the communicative stage between servers and peers when sending/receiving NTP packets. This stage is how a device handles polling a server, polling a higher-ranking peer in the system, comparing time with equal-ranked (same stratum) NTP servers in the system, and sending its own time to lower-ranking peers in the system.

Selection, Cluster, and Combining Algorithm

There are three algorithms used in the NTP system. The first one is the selection algorithm. This takes the time from different NTP servers, compares them and if some time is off, that time packet from the server is discarded. The selection algorithm results in the times left.

The clustering algorithm takes the selected times passed from the selection algorithm, then sorts and groups them so similar times are in the same cluster. If the cluster algorithm for example gets the times 1, 5, 2, 2, 9, 6, 9, 8, 5 (not real time, just to demonstrate) the cluster could look like 1, 2, 2, and 5, 5, 6 and 8, 9, 9 (simplified example).

The clusters from the clustering algorithm are then passed to the combining algorithm. This algorithm creates the best approximation of the times from the clusters and results in the final time which the local clock is changed to. The combining algorithm takes the number of times in each cluster, and how much the times between the clusters differ in consideration when creating the final time approximation.

Loop Filter

The loop filter is responsible for continuously filtering and adjusting the system clock's frequency. Its primary functions include estimating the clock's drift rate and calculating the phase offset between the local and reference clocks. Employing a Proportional-Integral (PI) control algorithm, it adjusts the clock frequency based on the phase offset and integrates past offsets for smoother adjustments. Operating within a feedback loop, it optimizes parameters like proportional and integral gains to balance response speed and stability. Ultimately, the loop filter aims to ensure both stability and accuracy in clock synchronization by minimizing phase offset and compensating for drift over time.

VFO

The VFO is the part of the code that adjusts the built-in clock to match the previously calculated best-fit time and frequency. This process modifies the time and frequency of the built-in clock to minimize deviation from the rest of the servers. This part of the code does not perform any calculations as it only applies the previously determined adjustments. Additionally, it updates the values used in the client’s peer/poll stage for communication with other devices.

Clock Strata

Clock strata or the clock stratum layers define the hierarchy of time sources used within the NTP system. The stratum level specifies the distance from the reference clock, typically an atomic clock which is considered Stratum 0 since it has the highest accuracy. The timekeeping devices of Stratum 0 are not directly connected to a network but are instead used via Stratum 1. Stratum 1 consists of servers often referred to as primary time servers which are responsible for broadcasting the time measurements obtained from the reference clocks. To reduce the load on these Stratum 1 servers a layer of balancing Stratum 2 servers which synchronize their time with one or more Stratum 1 servers are used. In widespread networks Stratum 3 servers which synchronize their time with Stratum 2 servers further reduce load. These stratums can keep going but the upper limit for stratum is 15.

The NTP algorithm constructs a Bellman-Ford shortest-path spanning tree [8]. This approach is utilized to minimize the accumulated round-trip delay to the stratum 1 servers.

4. Hardware Considerations

For testing, development, and the final product, we are utilizing an MPS2 Cortex-M3 AN385 platform emulated with QEMU.

5. Documentation

Our documentation is divided into two parts: GitHub repository and project reporting.

I. GitHub

On GitHub, there are markdown documents that provide guidance for the necessary setup. This guide assists in setting up and using the FreeRTOS development environment on the emulated ARM platform.

GitHub Issues: We have specific GitHub issues for each individual task, clearly defining the requirements. This allows us to track progress, manage contributions, and ensure that the requirements are met.

GitHub Kanban Board: Our project management process is visualized through a GitHub Kanban board. It categorizes tasks into ‘To Do’, ‘In Progress’, and ‘Done’ columns. The board provides our real-time overview of the project status and allows the team to monitor the flow of tasks.

II. Project Report

In the project report, we divide the focus areas of the project according to the section Project Plan. At the end of the project, the project report will cover all the necessary

topics including Planning & Organization, Analysis & Design, and System Development.

System Development

Verification of Requirements

We intend to verify that requirements are met by testing our NTP client.

- **BR-NTP-DEV:** Verify that FR-REC-SEND-PACKETS, FR-ALGORITHMS, and FR-CORRUPT-PACKETS requirements are met.
- **BR-COMP-EFF:** Verify that FR-MEM-EFF, FR-ASYNC-TASKS, FR-TIME-SYNC, FR-TIME-SYNC-PRECISION, FR-COMP-FREERTOS, FR-TIMEKEEP, requirements are met.
- **BR-USABILITY**: Verify that FR-QA-TESTING requirement is met.
- **FR-REC-SEND-PACKETS:** To confirm that the system can send and receive packets from multiple servers. We intend to use publicly available NTP servers in Sweden, such as the ones provided by Netnod [9]. If the client is able to retrieve the correct time from more than three Stratum 1/Stratum 2 servers, we consider that the NTP client works as expected.
- **FR-CORRUPT-PACKETS:** We test that our client can handle corrupt NTP messages by testing to send invalid, broken, or corrupted packets to the NTP client. We will only test with NTP messages where the relevant parts of the NTP message are corrupted.
- **FR-ALGORITHMS:** We intend to test the algorithms by passing the data retrieved from the NTP servers to the algorithms, and verifying that the output is reasonable and that it fulfills the FR-TIME-SYNC-PRECISION requirement.
- **FR-COMP-FREERTOS:** To verify that the client runs on FreeRTOS we want to check that the code compiles and runs on our described environment.
- **FR-TIMEKEEP**: Confirm that we are able to get and store timestamps using `xTaskGetTickCount`. This requires implementing a function that stores the current timestamp and tick count as well as a function that retrieves the stored timestamp and calculates the time that has passed since the stored timestamp in ticks. By corresponding each timestamp with the associated tick count, we are able to implement a timekeeping mechanism.
- **FR-ASYNC-TASKS**: Verify that the system is not stalled when waiting for a response, preferably by executing other tasks on the same FreeRTOS system. We intend to collaborate with our contact person at ABB to get example tasks that can be tested with our NTP client, or if not available, demo/dummy tasks provided by the authors of the FreeRTOS project.

- **FR-MEM-EFF**: We intend to use the built-in memory monitoring features in FreeRTOS, such as run time stats [10].
- **FR-TIME-SYNC**: The precision will be initially measured against publicly available NTP servers. Naturally, the precision against public NTP servers will be worse than over local area networks, which we will take into account and calculate an estimated precision if the NTP server were to be on the same network. Ideally, we would like to test against an NTP server on LAN and our ambition is to do so if provided by our contact person at ABB.
- **FR-TIME-SYNC-PRECISION:** Similar to FR-TIME-SYNC, the precision will be initially measured against publicly available NTP servers. Ideally, the synchronization should be measured over LAN. Want to achieve a precision of less than 10 ms.
- **FR-QA-TESTING**: GitHub actions will be used for code linting. To verify documentation clarity, we will ask third-party persons to test our NTP client with the provided instructions and observe if anything is ambiguous or unclear in the documentation.

Data Structures

Due to the complexity of the relationships/associations between the data structures, we have decided to limit this report to only showcasing the main functionality/use case of the data structures and their content. For more information about the associations, one can read the RFC 5905 [6]. All the data structures we utilize are a part of the skeleton implementation from the RFC 5905 [6] on which we've based our report on and not part of our software design.

Received Packet Structure

The data structure in Table 1 represents an NTP packet that has been received.

Data Type	Field Name	Description
`uint32_t`	`srcaddr`	Source (remote) address
`uint32_t`	`dstaddr`	Destination (local) address
`char`	`version`	Version number
`char`	`leap`	Leap indicator
`char`	`mode`	Mode
`char`	`stratum`	Stratum
`char`	`poll`	Poll interval

`s_char`	`precision`	Precision
`tdist`	`rootdelay`	Root delay
`tdist`	`rootdisp`	Root dispersion
`tdist`	`refid`	Reference ID
`tstamp`	`reftime`	Reference time
`tstamp`	`org`	Origin timestamp
`tstamp`	`rec`	Receive timestamp
`tstamp`	`xmt`	Transmit timestamp
`int`	`keyid`	Key ID
`digest`	`mac`	Message digest
`tstamp`	`dst`	Destination timestamp

Table 1: `ntp_r` data structure.

Transmit Packet Structure

The data structure in Table 2 is used for packets that are being transmitted.

Data Type	Field Name	Description
`uint32_t`	`srcaddr`	Source (remote) address
`uint32_t`	`dstaddr`	Destination (local) address
`char`	`version`	Version number
`char`	`leap`	Leap indicator
`char`	`mode`	Mode
`char`	`stratum`	Stratum
`char`	`poll`	Poll interval
`s_char`	`precision`	Precision
`tdist`	`rootdelay`	Root delay
`tdist`	`rootdisp`	Root dispersion

`tdist`	`refid`	Reference ID
`tstamp`	`reftime`	Reference time
`tstamp`	`org`	Origin timestamp
`tstamp`	`rec`	Receive timestamp
`tstamp`	`xmt`	Transmit timestamp
`int`	`keyid`	Key ID
`digest`	`dgst`	Message digest

Table 2: `ntp_x` data structure.

Filter Stage Structure

The data structure in Table 3 is used to represent a stage in the NTP clock filter algorithm. It contains the details necessary to process and filter time values.

Data Type	Field Name	Description
`tstamp`	`t`	Update time
`double`	`offset`	Clock offset
`double`	`delay`	Roundtrip delay
`double`	`disp`	Dispersion

Table 3: `ntp_f` data structure.

Association Structure

The data structure in Table 4 is shared between the peer process and poll process, representing an NTP association. An NTP association is the relationship established between an NTP client and an NTP server.

Data Type	Field Name	Description
`uint32_t`	`srcaddr`	Source (remote) address
`uint32_t`	`dstaddr`	Destination (local) address
`char`	`version`	Version number
`char`	`hmode`	Host mode

`int`	`keyid`	Key identifier
`int`	`flags`	Option flags
`char`	`leap`	Leap indicator
`char`	`pmode`	Peer mode
`char`	`stratum`	Stratum
`char`	`ppoll`	Peer poll interval
`double`	`rootdelay`	Root delay
`double`	`rootdisp`	Root dispersion
`tdist`	`refid`	Reference ID
`tstamp`	`reftime`	Reference time
`tstamp`	`org`	Originate timestamp
`tstamp`	`rec`	Receive timestamp
`tstamp`	`xmt`	Transmit timestamp
`tstamp`	`t`	Update time
`struct ntp_f[]`	`f`	Clock filter
`double`	`offset`	Peer offset
`double`	`delay`	Peer delay
`double`	`disp`	Peer dispersion
`double`	`jitter`	RMS jitter
`char`	`hpoll`	Host poll interval
`int`	`burst`	Burst counter
`int`	`reach`	Reach register
`int`	`ttl`	TTL (manycast)
`int`	`unreach`	Unreach counter
`int`	`outdate`	Last poll time

`int`	`nextdate`	Next poll time

Table 4: `ntp_p` data structure.

Chime List Structure

The data structure in Table 5 is used in the chime list.

Data Type	Field Name	Description
`struct ntp_p*`	`p`	Pointer to peer
`int`	`type`	Type
`double`	`edge`	Edge

Table 5: `ntp_m` data structure.

Survivor List Structure

The data structure in Table 6 is used within the clustering algorithm.

Data Type	Field Name	Description
`struct ntp_p*`	`p`	Pointer to peer
`double`	`metric`	Metric

Table 6: `ntp_v` data structure.

Local Clock Structure

The data structure in Table 7 holds all the necessary information for tracking, updating, and adjusting the local time.

Data Type	Field Name	Description
`tstamp`	`localTime`	Local based on ticks time
`tstamp`	`t`	Update time
`int`	`state`	Current state
`double`	`offset`	Current offset
`double`	`last`	Previous offset

`int`	`count`	Jiggle counter
`double`	`freq`	Frequency
`double`	`jitter`	RMS jitter
`double`	`wander`	RMS wander
`TickType_t`	`lastTimeStampTick`	FreeRTOS tick

Table 7: `ntp_c` data structure.

Association Data Structures

The data structure in Table 8 is used to find associations between peers and addresses.

Data Type	Field Name	Description
`ntp_p**`	`peers`	Pointer to peers
`int`	`size`	Size of the table

Table 8: `Assoc_table` data structure.

System Structure

The data structure in `ntp_s` is used to represent the status of the system. It contains various fields that store important metrics and references required for system operation.

Data Type	Field Name	Description
`tstamp`	`t`	Update time
`char`	`leap`	Leap indicator
`char`	`stratum`	Stratum
`char`	`poll`	Poll interval
`char`	`precision`	Precision
`double`	`rootdelay`	Root delay
`double`	`rootdisp`	Root dispersion
`char`	`refid`	Reference ID

`tstamp`	`reftime`	Reference time
`struct m`	`m[NMAX]`	Chime list
`struct v`	`v[NMAX]`	Survivor list
`struct p`	`*p`	Association ID
`double`	`offset`	Combined offset
`double`	`jitter`	Combined jitter
`int`	`flags`	Option flags
`int`	`n`	Number of survivors

Table 9: `ntp_s` data structure.

Implementation of the Network Time Protocol

Below is a detailed description of how we implement the NTP system in C, using the design and algorithms as described under 3. Software Development. The system operates within a FreeRTOS task. Our implementation is based on the skeleton code provided in RFC 5905 [6].

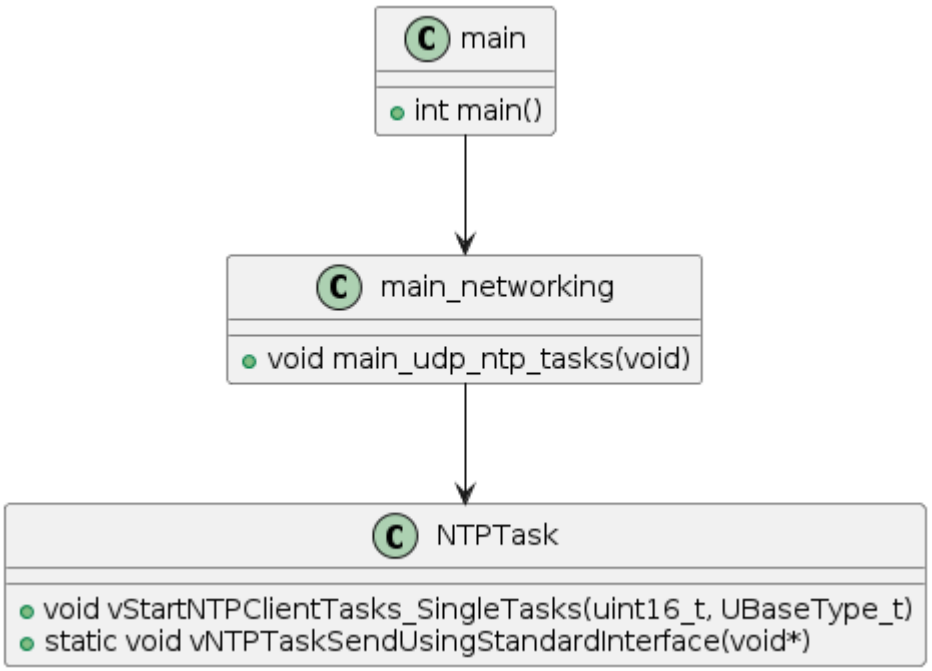

Figure 5: UML diagram of classes connected to figure 6 (UML diagram based on [11]).

Client Initialization

The client implementation starts in the file `main.c` with the main method, see Figure 5. This method is executed by the FreeRTOS kernel and acts as an entry point to our code (see Figure 7 - activity 1). The main method in `main.c` calls `main_udp_ntp_tasks()` which is located in `main_networking.c`. Inside `main_networking.c`, the FreeRTOS scheduler is set up and configured. `main_udp_ntp_tasks()` also handles initializing the network (see Figure 7 - activity 2). After the scheduler is initialized and the network is set up, the scheduler calls `vStartNTPClientTasks_SingleTasks()` which in turn starts our task and therefore starts the NTP client (see Figure 7 - activity 3).

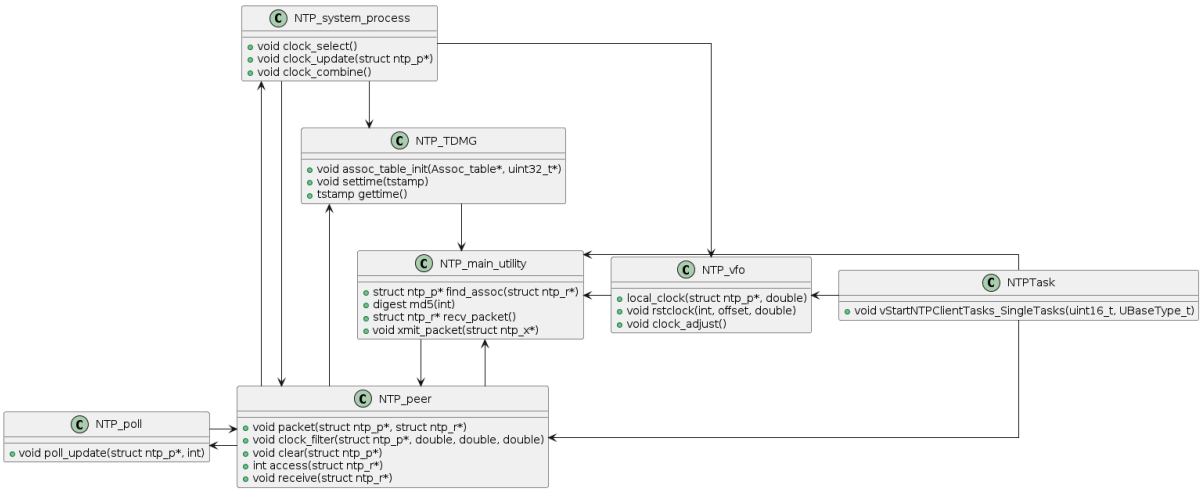

Figure 6: UML diagram of the different classes in the NTP system (UML diagram based on [11]).

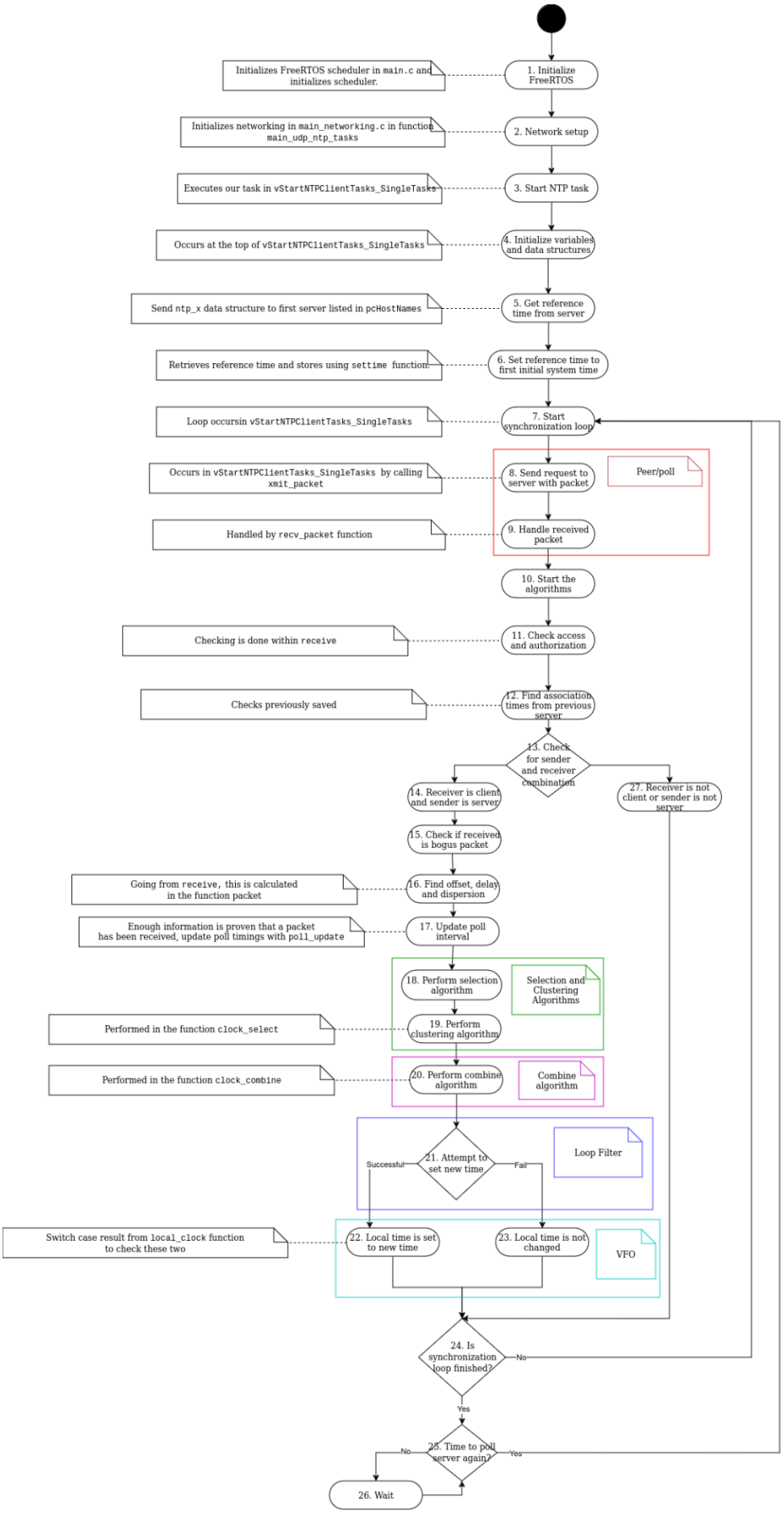

Figure 7: UML Activity Diagram for the NTP system

During the client execution, the local time is tracked by initially setting a base time and updating it based on the tick count. The tick count is the number of ticks that have passed since the task started and this count is retrieved by calling a FreeRTOS built-in function. The tick length is configurable and set to 1000 ticks per second, as per the FreeRTOS standard.

Variable Initialization

The first action of the NTP task is to initialize variables (see Figure 7 - activity 4). This involves initializing multiple variables in the structures `ntp_s` (system structure) and `ntp_c` (local clock structure), (see Table 9, Table 7). Afterward, the frequency setting is applied to the system using `rstclock()`. A char pointer array named `pcHostNames` (see Data Structures) is initialized and set to the host names of the NTP servers we would like to query (hard-coded hostnames).

A variable named `assoc_table` of type `Assoc_table` (see Table 8) is initialized using the function `assoc_table_init()`. It contains pointers to `ntp_p` correlating to each server and the number of servers.

Time Setting and Synchronization

Initially, we set the time of the built-in clock to the reference time from the first NTP server in our list of servers (see Figure 7 - activity 5, 6). To get the time from this first server, we call `prep_xmit()` to prepare a struct `ntp_x`, the transmit packet structure (refer to Table 2), which is then sent to the server with `xmit_packet()`. The response is received using `recv_packet()`, which returns a new `ntp_r`, the received packet structure (refer to Table 1), which contains all the information received from the NTP server. To set the time of our built-in clock, we call the function `settime()` with the received timestamp which updates the local time to the time received from the server (conversely, there is a function called `gettime()` that retrieves the saved timestamp and calculates a new timestamp based on the number of ticks that has passed). It also sets the client reference time to this received timestamp. Now the client has a correct reference and local time and the client is ready for synchronization.

Synchronization Loop

The synchronization loop (see Figure 7 - activity 7) starts by going through the same procedures as described in Time Setting and Synchronization. That is, call `prep_xmit()`, `xmit_packet()`, and `recv_packet()` (see Figure 7 - activity 8-10). Then the `receive()` function is called using the `ntp_r` struct. After `receive()` is finished, the server is considered as either synchronized or unsynchronized, regardless the synchronization loop continues until all servers in `pcHostNames` (see Data Structures) have been synchronized or unsynchronized.

Start Algorithms → Selection

Once the synchronization loop is initiated, we start utilizing the algorithms, where the first step is to eliminate bogus and unauthorized packets by checking the legitimacy of a packet using the `access()` function (see Figure 7 - activity 11). This function verifies the packet's version and checks for the MAC address, as well as computing and storing a message digest of the `keyid` using the `md5()` function.

Subsequently, we retrieve association times from previous server interactions (see Figure 7 - activity 12). This previous data helps to validate the current packet. The association data is stored in structure `ntp_p` (refer to Table 4) and `assoc_table` (refer to Table 8). The function `find_assoc()` in the `NTP_main_utility` class (Figure 6) is responsible for retrieving data. The next step is to check for the sender and receiver combination (see Figure 7 - activity 13). This step ensures that the received packet comes from a valid and expected NTP server. This is handled in the `NTP_peer` class (Figure 6), specifically in the function `receive()`. The function verifies that the sender is a server and the receiver is a client. If this combination is incorrect, the packet is rejected.

Tied to the previous step, the next part validates that the sender is a known NTP server and the receiver is an NTP client (see Figure 7 - activity 14). The `receive()` function performs this check as part of its packet processing routine. After this, the system checks if the received packet is bogus (see Figure 7 - activity 15). This involves verifying the packet format and content in structure `ntp_r` (refer to Table 1) and `ntp_x` (refer to Table 2). The `receive()` function includes checks for packet format, version, and MAC length. Packets that fail these checks are rejected.

Once a packet passes checks, the next step involves calling `packet()` which calculates the offset, delay, and dispersion (see Figure 7 - activity 16). These calculated values are critical for adjusting the local clock. The offset is the difference between the server's time and the client's time and the delay represents the round-trip time for the packet (see Appendix 1). These calculations are initially performed in the `packet()` function. This process checks for potential errors, such as popcorn spikes (see Appendix 1), and discards those packets in addition to finding the best sample based on delay. It also updates the poll interval (see Figure 7 - activity 17) based on these calculations. This determines how frequently the client sends synchronization requests. If no errors are found then the `clock_filter()` function in the `NTP_peer` class is called to supplement and further refine the calculated values.
Subsequently, we apply the selection and clustering algorithms which are explained in the next section.

Selection and Clustering Algorithms

The selection and clustering algorithms are responsible for selecting the best timestamp received from the NTP servers defined in `pcHostNames` (see Data Structures) and filtering out any anomalies. These steps ensure that only reliable time samples are used to adjust the system clock. The `clock_filter()` function compares incoming packets with previously received packets from the same server, sorting them based on their delay to find the best sample. The filtered list of packets is then saved for the next cycle. This function also compares the offset against the jitter (refer to Appendix 1) to remove any spurious values before calling `clock_select()`.

The `clock_select()` function's role is to eliminate falsechimers (servers providing inaccurate time) from the server population, leaving only truechimers. It checks the offset and the root distance of each server. The algorithm iterates through the chime list, searching for the largest intersection between chimes to determine the limits of the intersection. The chime list is then culled so that only chimes with an offset between the endpoints are preserved. These are categorized into being survivors. To ensure a usable time source, a set of N survivors must still exist, which is currently set to 1 but is often configured to 4. The remaining survivors are used to calculate the system jitter. The survivors must now be reduced to a single best clock which is a truechime, which is done by calculating the selection jitter as the square root of the sum of squares. (see Figure 7 - activity 18). Each iteration refines the maximum and minimum jitter until the maximum jitter is less than the minimum or until too few survivors remain. The best clock is then selected, either a new survivor or by re-implementing the old best-fitting chimer.

Combine Algorithm

The `clock_combine()` function implements the clock combine algorithm (see Figure 7 - activity 20) in NTP which is called inside the function `clock_update()`. `clock_combine()` merges the offsets of selected time samples (survivors of the clustering algorithm) using a weighted average, weights are inversely proportional to the root distance, a measure of the reliability of each sample. It calculates the combined offset by summing the offsets of the survivors divided by their root distances, normalized by the total weight. Additionally, it computes the selection jitter, which represents the variability among the offsets, as the weighted root mean square (RMS) difference between the first survivor and the rest. The function `clock_combine()` saves offset and jitter to the system structure `ntp_s` to be used by `clock_update()`.

Loop Filter

Loop filter, activity 21 in Figure 7, is not a standalone function or specific part of the code but is implemented through a series of steps integrated within the NTP system. Data flow is previously explained in the section related to Figure 4, and incorporated in the activity

diagram, Figure 7. The loop filter smooths out time variations and ensures gradual adjustments to the system clock. It integrates functionality from several components in the NTP system.

The `clock_filter()` function in the `NTP_peer` class (Figure 6) filters incoming time samples, comparing them with previously received samples to find the best one and removing bogus values by comparing offsets against jitter (see Figure 7 - activity 15). Then the `clock_combine()` function in the `NTP_system_process` class merges offsets from the selected time samples. This function computes the combined offset and selection jitter, storing these values in the `ntp_s` structure (see Figure 7 - activity 20). The `clock_update()` function updates the system clock using the combined offsets and checks the local clock against the calculated offset (see Figure 7 - activity 21). It calls `local_clock()` which continues in the `NTP_vfo` class, it is explained in the following section.

VFO

The VFO (Voltage-Frequency Oscillator) (Figure 7 - activity 22, 23) is responsible for controlling the frequency adjustments of the system clock to match the reference clock, which in the case of the NTP client is the NTP server. This ensures clock accuracy by continuously correcting the rate at which the system clock runs.

It uses `local_clock()` which checks if the offset is within acceptable bounds, then it adjusts the clock frequency and phase accordingly. It also utilizes `rstclock()` to reset the state machine after changes to offset and clock frequency. Additionally, it uses `clock_adjust()` to apply the calculated frequency and phase adjustments to the system clock to achieve precise and reliable timekeeping. This ensures system clock stability and accuracy in relation to the reference clock.

Final Synchronization

After attempting to set the new time using `local_clock()`, the NTP system checks if the synchronization loop is finished (Figure 7 - activity 24). This involves verifying all servers in `pcHostNames` (see Data Structures) have been polled and their times are synchronized. If it has been processed the system proceeds to the next step, if not all servers have been processed the loop continues. Then the NTP system checks if it is time to poll the server again (Figure 7 - activity 25). This is determined based on the updated poll interval calculated in a previous step (see Figure 7 - activity 17). If it is time to poll the server, the system sends another synchronization request to the server, starting the synchronization loop again. If the synchronization loop is finished, or if it is not yet time to poll the server again, the system enters a wait state (Figure 7 - activity 26). During this wait state the system maintains its current synchronization status and periodically checks if it is time to poll the servers based on the configured poll interval.

Change Management

Lack of Acquisition of the Development Board

At the beginning of the project, we defined requirements with the client that we split into two scenarios. One scenario was called the *Development Board Scenario* which listed requirements that we would follow if we received a board from ABB before the 10th of April. The board would be used to get the program to work on actual hardware. Otherwise, we would continue with an emulator that emulated the board. In this scenario, we would follow the requirements listed under *Emulator Scenario.*

We did not receive a board from ABB before the 10th of April and therefore we continued with the requirements listed under *Emulator Scenario.* The requirements we listed under the Development Board Scenario, were the following

- **DB-DEADLINE:** Given the aspects of whether we receive the development board, shipping, hardware setup, and testing. If we get the development board on time, we can work on those specific requirements after finishing the emulator part. However, we need to set a clear deadline. If the board has not received according to the *Project Plan* we will focus only on emulator requirements.
- **DB-NTP-DEV:** Build and develop an NTP client compatible with the development board that supports retrieving time from NTP servers (of clock strata of Stratum 1 and Stratum 2 servers, see definition of these terms under Clock strata).

We also had a requirement for the deadline of the emulator, which was the following

- **EM-DEADLINE:** Establish a specific deadline for the emulator-based requirements, which can be seen in the *Project Plan*.

Instead, we adjusted and updated the requirements that are listed under *Project/Business requirements* in the Requirements section, which originally were the requirements listed under Emulator Scenario.

64-bit → 32-bit

Very late in the process, we realized that the skeleton NTP client skeleton implementation from RFC 5905, on which we based our project, was written and designed for 64-bit systems. This posed a challenge for us since FreeRTOS only supports 32-bit, and even more specifically the chip we used is 32-bit only.

This problem is relevant since the NTP skeleton implementation uses 64-bit timestamps, where the first 32-bits represent the number of seconds since the NTP epoch (1 of January

1900 and the second 32-bits represent the fractions of a second, where 2^32 represents one full second.

Adapting the skeleton implementation for a 32-bit system would involve writing software that allows us to perform 64-bit arithmetic on 32-bit systems, often using two separate 32-bit registers. This means that most of the algorithms already implemented in the skeleton implementation would have to be rewritten and adjusted to work on 32-bit systems, which would involve a lot of work.

The scope of this project was initially decided upon with the presumption that the algorithms for the NTP client had already been implemented in C so that we could base our project on that. In light of this challenge, we reassessed the project scope and decided that rewriting and reimplementing all of the algorithms for our 32-bit systems would be too much work for the scope of this project (it is over 2500 lines of code with complicated arithmetic operations and algorithms).

We communicated this challenge with our contact person at ABB (our main stakeholder) to discuss potential risks and prioritization of goals. During this meeting, we agreed that the best approach would be to annotate everywhere in the code where the algorithms would have to be rewritten to work with 32-bit so that our client could potentially continue our work. Our contact person also agreed that rewriting all of the algorithms would be outside the scope of the project, and enlightened that we still fulfill the main business requirements.

Not using a server from ABB

At the beginning of the project, we decided with our contact person at ABB that we would receive a physical NTP server from them (oscillator with NTP functionality). This server was supposed to be used by us to test that our program could ask for a time and receive the correct time. Due to complications at ABB, we did not receive a physical NTP server.

To adapt to this change, we created a local NTP server on a computer. We initialized a specific time on this server and let our program pull the time from this server instead. When our program received the same time as the one we initialized on the server we could confirm that our program could ask for the time, and receive the correct time.

After verifying our program with the local server, we extended our testing by pulling time data from the Netnod servers located in Sweden. Despite this, we continued to use our own server for additional testing. Our local server proved useful for verifying that our program met all requirements, such as handling incorrect NTP packets with unwanted data.

Due to the fact that we did not receive an NTP server from ABB, we were not able to measure the functional requirement FR-TIME-SYNC-PRECISION under the planned circumstances, which we took into account under the testing phase.

Verification & Validation

To check that our product is working as expected we thoroughly tested our implementation according to our requirements and proposed methodology of requirements testing. The tests were split into two different areas, verification and validating, as explained in the course lectures. This section explains how we tested our product, based on the tests described in *Verification and Requirements*, and the results of the testing process.

Verification Tests

The tests that we categorized as verification touch the more technical part of the project, including performance requirements and to see that the overall product is working.

FR-REC-SEND-PACKETS

TC#001

This test case aims to fulfill the requirement FR-REC-SEND-PACKETS.

With the FR-REC-SEND-PACKETS requirements, we wanted to confirm that we can retrieve the correct time from more than one NTP server. In our code, we implemented data structures for defining a list of target NTP servers where we declared the target hostnames

```
// Array of hostname strings
const char *pcHostNames[NMAX] = {"sth1.ntp.se", "sth2.ntp.se",
"svl1.ntp.se", "mmo1.ntp.se", "lul1.ntp.se"};
```

Figure 8: Code snippet of server names.

To test the basic functionality of the NTP client, we used publicly available NTP servers in Sweden hosted by Netnod. These server names can be seen in Figure 8.

Our client then sends a well formatted NTP packet and retrieves a packet from the specified NTP servers and the intention was to use the algorithms from the skeleton implementation to determine the best reference time from the listed NTP servers.

FR-ALGORITHMS

TC#002

This test case aims to fulfill the requirement FR-ALGORITHMS.

The clock mitigation algorithms from the skeleton implementation are meant to be used to determine the best time reference out of several NTP servers. We already fetch time from multiple NTP servers, so to fulfill this requirement we simply tested that we can pass the data

from each NTP server to the algorithm functions, and then check that the algorithms are able to determine the best reference time and calculate offset and delay.

DEF#001

Unfortunately, due to the aforementioned problem with 64-bit conversion on a 32-bit system (mentioned in Change management), it was not possible to use all the algorithms from the RFC 5905 skeleton implementation and therefore we did not fully fulfill the FR-ALGORITHMS requirement. Some of the algorithms, like calculating drift did however work.

FR-CORRUPT-PACKETS

TC#003

This test case aims to fulfill the requirement FR-CORRUPT-PACKETS.

We wanted to confirm that our NTP client is able to handle receiving NTP packets where the parts of the packet relevant to our project and scope are corrupt or broken.

This was fulfilled by implementing a small NTP server in Golang, that we ran separately. This NTP server was programmed to send bad or invalid fields, such as invalid stratum levels or invalid timestamps. We concluded that our NTP client is able to determine invalid fields such as invalid stratum levels. Outrageous timestamps were however not detected by our client, since there is no “invalid” timestamp; all 64-bit integers can be interpreted as valid timestamps but they can be wildly off since our client does not know the actual time it will interpret the timestamp based on its value (meaning it can be anything between year 1900 and 2036, defined as the NTP era, rollover for 2^32) [12]. NTP is supposed to handle this by comparing multiple NTP servers using the previously defined algorithms, but since TC#002 was not fulfilled, this is not handled.

FR-MEM-EFF:

TC#004

This test case aims to fulfill the requirement FR-MEM-EFF.

To test and verify the FR-MEM-EFF requirement, we initially intended to use the built-in run time stats feature in FreeRTOS. When trying to fulfill this requirement, we realized that the `vTaskGetRunTimeStats` function in FreeRTOS is only able to provide data regarding execution time and absolute time, and no data on memory usage. While further trying to measure the memory efficiency, we stumbled upon kernel function `vPortGetHeapStats()` but this, unfortunately, does not support the heap_3 memory allocation our project is based on (heap_3 mimics `malloc()` and `free()` from C, which is what we have based our project on after recommendation from our mentor at ABB). We also

asked on the FreeRTOS Forums if there is any way to measure memory usage within heap_3, we came to the conclusion that there is no built-in function to measure memory usage in FreeRTOS given heap_3.

To still fulfill this requirement, we used wrapper functions around all of our `malloc()` and `free()` function calls in order to measure stack memory usage. At peak, we allocated at most 676 bytes of memory which is 0.000644 megabytes and therefore well within our requirement of 1 Mb. This figure, however, does not include any overhead from FreeRTOS itself which is related to the task. According to the official documentation [13], the overhead per task and queue is approximately 76 bytes + 64 bytes = 140 bytes, which together with the previously measured usage comes up to a total of 816 bytes which equals 0.000778 megabytes. On top of this, we also have a statically allocated heap (configured using `#define configTOTAL_HEAP_SIZE`) that was set to 279,000 bytes by default, meaning 0.266 megabytes. Combined with our previous figure we end up with a total of 0.266778 megabytes which is well within our requirement. Note that the whole FreeRTOS kernel together with our task exceeds 1 megabyte of memory usage, but the requirement itself only concerns our task and not the kernel. We therefore fulfill this requirement.

FR-ASYNC-TASKS

TC#005

This test case aims to fulfill the requirement FR-ASYNC-TASKS.

The TC#005 test involves verifying that the system does not stall while waiting for a response, in order to execute other tasks on the same FreeRTOS system. However, it can only run one task on one core at a time because the FreeRTOS operating system is a single-core system [14]. To run tasks in parallel, more than one core is needed. Nevertheless, the tasks appear to be executed in parallel because they are rapidly switched in and out.

Operating systems like Linux apply kernels that enable seemingly simultaneous access for multiple users. Multiple users can execute multiple programs that appear to run concurrently. Each executing program is a task (or thread) controlled by the operating system. When an operating system can execute multiple tasks as explained, it is considered to be multitasking.

We test this by running the NTP client requesting time from NTP servers every 30 seconds. Besides that, we also run a UDP client task that sends and receives packets to and from a remote server. This task is set up to periodically send data and listen for server responses, providing a real-time network traffic simulation. The dual operation of the NTP client and the UDP client on the same FreeRTOS system allows us to verify the system's capability to handle multiple network tasks without stalling. This test aims to assess the multitasking efficiency of FreeRTOS, ensuring both time synchronization and network communication are managed effectively under concurrent operational conditions.

FR-TIME-SYNC:

TC#006

This test case aims to fulfill the requirement FR-TIME-SYNC.

To test and verify the FR-TIME-SYNC requirement, we performed a test against a publicly available Netnod server located in Stockholm, where we measured the latency between the client's time and the server's time. This latency should be no longer than 1 s on LAN, however we were not able to test the precision on LAN.

To make a good benchmark for precision on the current implementation, we tested the system over a period of time with a relatively high amount of readjustments during this time. We performed this test to evaluate how much the algorithms found it necessary to change the local time after the time started to settle with multiple iterations of received packets. We let the program run for an hour, seeing how the continuous local adjustment of time changed during this period. This shows how the time drift settles/stabilizes. We see this as a fitting test as smaller adjustments over time would indicate that the time drift has decreased as a sure value of offset and delay has been found and accounted for.

FR-TIME-SYNC-PRECISION

TC#007

Similar to #TC008, we let the program run for an hour, seeing how the continuous local adjustment of time changed during this period. After monitoring the drift for over this period of time we noticed that the time drift did not stabilize under 10 ms, which suggests we failed to meet this requirement.

DEF#002

Since we were not able to implement the algorithms to their full extent (described in DEF#001), we were not able to achieve high time precision, as the algorithms are essential for clock accuracy and precise time calculation.

FR-COMP-FREERTOS

TC#008

This test case aims to fulfill the requirement FR-COMP-FREERTOS.

To test and verify that our NTP client is compatible with FreeRTOS we ran and compiled our code with the FreeRTOS kernel. This proved to be no issue and our code compiled. This suggests that the code structure, syntax, and dependencies are compatible with the FreeRTOS environment.

FR-TIMEKEEP

TC#009

This test case aims to fulfill the requirement FR-TIMEKEEP.

This test involved using the `gettime()` and `settime()` functions to keep track of the current time. We set the time by using `settime()` and let the system sleep for an amount of time and then get the new time from `gettime()`. If this time has been updated correctly, by adding the difference in time to the time we set, it works. We let the system sleep for different amounts of times, ranging from one second to one minute with intervals of 0,5 seconds, to see that even the fractions were updated correctly. This indicates that we fulfill the requirement.

FR-QA-TESTING

TC#010

This test case aims to fulfill the requirement FR-QA-TESTING.

We did two tests connected to the FR-QA-TESTING requirement. The first test was through GitHub actions (a continuous integration service provided by GitHub) which is used to create an automated workflow. This workflow can be programmed to do specific tasks. In our case, we programmed the workflow to check the code when someone pushes to the main branch or merges into the main branch. When doing this, the workflow executes the clang-format CLI tool that checks that the code is correct according to the .clang-format file. The .clang-format file has been configured to check that the code is indented by four, has a column limit of 120, and checks if braces are starting at new lines. If the code is correctly formatted, according to these specifications, the workflow will indicate this by setting a green mark on the commit in the repository. If the code is not correctly formatted though, the workflow will indicate this by setting a red mark on the commit instead.

After adding the GitHub actions, we got a lot of red marks on our commits. To resolve this we created a script with the -i flag (in place edit flag) to handle formatting errors. This script finds all the .c and .h files and similarly to the GitHub actions workflow checks that they conform with the format but instead of reporting a failure, this script modifies the files directly so that the code matches the style. With this script, we can easily make sure our code is well-formatted before we push.

This GitHub actions testing resulted in the code being more readable and easier to use and modify by a third-party user. In other words, it resulted in better usability. It also makes sure our code is more conformant with industry best practices and that the code style is more consistent.

Validation (Usability Test)

The tests that we categorized as validation touches areas like usability and customer acceptance. These tests focus on whether the product can be used by the customer as well as whether the code is well-structured and understandable.

FR-QA-TESTING

TC#011

This test case aims to fulfill the requirement FR-QA-TESTING.

To test and validate our product's usability further we let a third-party user test the product. This person is another registered student in the course. We let the person use one of our GitHub accounts to get access to the repo. The person's main task was to follow our guidelines and to get the system working. The person thought that the code itself was well documented and understandable but had some issues getting the system working. The person therefore asked for a more detailed readme file that more clearly explains how to get Docker working, step by step. After updating the readme file with this, our test person could install the system and get it up and running. This suggests that users outside the project group can get the system up and running as well as get familiar with the code and modify it and that our documentation and getting started guide is clear.

We divided the testing into three different stages that our testing participant went through:

1. **Setup**: The participants were provided the code along with the README which contains setup instructions. No further help was offered to the user, only that the goal was to run the NTP client and that the result should be that the current time was displayed in a terminal. The participant recorded and noted any issues or unclear instructions they encountered, while one from our team also noted their behavior (with a focus on what part of the documentation they read and focused on and what parts they skipped over).
2. **Operational testing:** The participant then followed the instructions to get the NTP client up and running. Here, we monitored the participant and focused on what part of the code they looked at and if they understood the terminal output correctly.
3. **Feedback testing:** After completing the above tasks, the participant was given a chance to provide their own feedback on the documentation and the code, if there was anything unclear in the code, the documentation or how they verified that the client worked as expected.

References

[1] B. K, D. R, B. B. Sinha, and G. R, “Clock synchronization in industrial Internet of Things and potential works in precision time protocol: Review, challenges and future

directions," *International journal of cognitive computing in engineering*, vol. 4, pp. 205–219, 2023, doi: 10.1016/j.ijcce.2023.06.001.

[2] "Why RTOS and What is RTOS?," *FreeRTOS*, 2019. https://www.freertos.org/about-RTOS.html

[3] "coreSNTP," *FreeRTOS*. https://www.freertos.org/coresntp/index.html

[4] *IETF.org*, 2022. https://www.ietf.org/rfc/rfc768.txt

[5] D. F. Franke, D. Sibold, K. Teichel, M. Dansarie, and R. Sundblad, "Network Time Security for the Network Time Protocol," *IETF*, Sep. 01, 2020. https://datatracker.ietf.org/doc/html/rfc8915 (accessed Mar. 29, 2024).

[6] D. Mills, J. Martin, J. Burbank, W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification", *IETF*, June 2010. https://datatracker.ietf.org/doc/html/rfc5905 (accessed Mar. 29, 2024).

[7] "ClangFormat — Clang 12 documentation," *clang.llvm.org*. https://clang.llvm.org/docs/ClangFormat.html

[8] Stanford, "CS 161 Lecture 14 - Amortized Analysis," n.d. Accessed: Mar. 29, 2024. [Online]. Available: https://web.stanford.edu/class/archive/cs/cs161/cs161.1168/lecture14.pdf

[9] "Connect to NTP Servers", *NetNod*. https://www.netnod.se/ntp/connect-to-ntp-servers (accessed Mar. 29, 2024)

[10] "FreeRTOS Run Time Stats," *FreeRTOS*. https://www.freertos.org/rtos-run-time-stats.html (accessed Mar. 29, 2024).

[11] IBM, "Dependency relationships in C/C++ domain modeling class diagrams," Last Updated: 2021-03-02. [Online]. Available: https://www.ibm.com/docs/en/rational-soft-arch/9.7.0?topic=relationships-dependency

[12] "Time Synchronization and Rollovers" 2021, Accessed: May. 5. 2024. [Online]. Available: https://kb.meinbergglobal.com/kb/time_sync/time_synchronization_and_rollovers

[13] "Memory Usage, Boot Times, & Context Switch Times" RTOS Fundamentals, Amazon Web Services, Inc. Available: [https://www.freertos.org/FAQMem.html#RAMUse]. [Accessed: May. 12. 2024.]

[14] "Multitasking," RTOS Fundamentals, Amazon Web Services, Inc. Available: [https://www.freertos.org/implementation/a00004.html]. [Accessed: May. 12. 2024.]

[15] "Introduction — QEMU documentation," *www.qemu.org*. https://www.qemu.org/docs/master/system/introduction.html (accessed Mar. 29, 2024)

[16] D. Hemmendinger, "operating system | Definition, Examples, & Concepts," *Encyclopedia Britannica*, Feb. 18, 2022. https://www.britannica.com/technology/operating-system (accessed Mar. 29, 2024)

[17] "What is an ARM processor?," *www.redhat.com*, Jul. 21, 2022. https://www.redhat.com/en/topics/linux/what-is-arm-processor (accessed Mar. 29, 2024)

[18] GitHub, Inc., "Build software better, together," *GitHub*, 2019. https://github.com/about (access Mar. 29, 2024)

[19] Cisco, "What is a LAN? Local Area Network," *Cisco*, Oct. 2019. https://www.cisco.com/c/en/us/products/switches/what-is-a-lan-local-area-network.html (access Mar. 29, 2024)

[20] "Docker overview," *Docker Documentation*, Apr. 09, 2020. https://docs.docker.com/get-started/overview/ (access Mar. 29, 2024)

[21] "What is a container?," *Docker Documentation*, Mar. 28, 2024. https://docs.docker.com/guides/docker-concepts/the-basics/what-is-a-container/ (accessed Mar. 29, 2024).

[22] "C skeleton code," *IBM*. https://www.ibm.com/docs/en/integration-bus/10.0?topic=extensions-c-skeleton-code (accessed Mar. 29, 2024).

[23] "Massachusetts Institute of Technology Lecture 2 6.895: Advanced Distributed Algorithms Medium Access Control," 2006. Accessed: Mar. 29, 2024. [Online]. Available: http://courses.csail.mit.edu/6.885/spring06/notes/lect2.pdf

Appendix 1: Glossary and Definitions

Term	**Definition**
Emulator	An emulator is software that allows one computer system (the host) to imitate the functions of another computer system (the target). In the context of QEMU, it refers to the ability of QEMU to replicate the architecture and functions of different computing systems, enabling software designed for a specific platform to run on another, different platform. [15]
OS (Operating system)	An operating system is the primary software that manages computer hardware and software resources and provides common services for computer programs. Essentially, it acts as an intermediary between the user and the computer hardware, facilitating the execution of applications, management of files, and handling of peripheral devices. [16]
ARM platform	ARM platform refers to a family of processors and architectures well-known for their energy efficiency and reduced complexity as compared to the x86 architecture used by Intel and AMD. [17]
GitHub	A platform for version control and collaboration, using a software called Git. GitHub Issues allows to track tasks, ideas, enhancements, or bugs. GitHub Actions is a CI/CD platform that automates workflows. [18]
Local Area Networks	A Local Area Network (LAN) is a network that interconnects computers within a limited area and facilitates resource sharing and communication without the need for an internet connection. [19]
Docker	A platform used to develop and run applications. Docker packages software into containers (explained below). [20]
Containers	Lightweight executable software packages that include everything required to run a piece of software (like code, system tools, libraries, etc). [21]
Skeleton code	Refers to the basic framework of a function/program. It contains the structure and minimal implementation details. Often used as a starting point for further development. [22]
MAC	This means *Media Access Control*, is an address used to uniquely identify network interfaces at the data link layer of the network segment. It is usually used for managing network access and control. [23]
Truechimer	A timeserver that provides time that is believed to be good, meaning low jitter compared to UTC. [6]

Falseticker / Falsechimer	When statistical filtering deems a timeserver unreliable. It often reflects not on the server's inherent performance but rather on the unpredictable and uneven network delays encountered between the server and its clients. [6]
Jitter	The variability in time measurement differences between readings. It can be used to measure the stability of the time source by analyzing fluctuations over time. Low jitter means time measurements are more consistent. [6]
Round-trip delay	Round-trip delay in the context of NTP is the total time it takes for a packet to travel from the client to the server, and then back to the client. [6]
Root distance	Root distance measures the total error or uncertainty in the time provided by an NTP server. It includes factors like round-trip delay, dispersion, clock offset, root delay, and root dispersion. [6]
Popcorn spikes	These are sudden and large deviations in time offset in NTP which can disrupt the synchronization process. Within the protocol, there is an algorithm called “popcorn spike suppressor” which is used to mitigate the spikes. [6]