IoT Power Consumption & DTLS Modeling

- A seventh semester project -

Project Report
DAT7-02

Aalborg University
Computer Science

Computer Science
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
IoT Power Consumption & DTLS Modeling

Theme:
Computer Science, Experiments

Project Period:
Fall Semester 2023

Project Group:
CS-23-DAT-7-02

Participant(s):
Christoffer Brejnholm Koch
Katja Lindell Thesbjerg
Lise Bech Gehlert
Malthe Peter Højen Jørgensen
Signe Kirstine Rusbjerg
Tobias Møller

Supervisor(s):
Rene Rydhof Hansen
Tobias W. Bøgedal

Page Numbers: 89

Date of Completion:
December 20, 2023

Abstract:

This report aims to optimize power consumption and lower response times when using the DTLS communication protocol between a client and a server, where the client is placed behind a router that implements NAT. It does this by modeling the DTLS 1.3 protocol using UPPAAL and UPPAAL SMC. Multiple cases are tested to find potential benefits, especially when it comes to power consumption. The results generally seem promising and within the cases, each of them implements near instant response time.

Contents

Preface

This report was written by the computer science group "CS-23-DAT-7-02" in the period from the 1st of September to the 21st of December at Aalborg University.

The project is focused on internet communication protocols, with a deeper focus into battery-driven IoT devices and their power consumption.

The group would like to thank the supervisor René Rydhof Hansen and the co-supervisor Tobias Worm Bøgedal for their assistance and guidance throughout the project.

Aalborg University, December 20, 2023

Lise Bech Gehlert
<lgehle20@student.aau.dk>

Christoffer Brejnholm Koch
<ckoch20@student.aau.dk>

Signe Kirstine Rusbjerg
<srusb20@student.aau.dk>

Katja Lindell Thesbjerg
<kthesb20@student.aau.dk>

Malthe Peter Højen Jørgensen
<mphj20@student.aau.dk>

Tobias Møller
<tmalle20@student.aau.dk>

Chapter 1

Introduction

Power consumption is especially important when considering low-powered IoT devices. This can oftentimes be achieved, by sacrificing response times. Therefore, this project explores ways to optimize power consumption and lower response times when using the DTLS communication protocol between a client and a server. Specifically, when the client is placed behind a router that implements NAT.

The efficacy of this new approach is compared to existing methods. The power consumption metric for each of these are compared in simulations, using UPPAAL SMC. this allows for a theoretical comparison between existing methods as well as a way to validate and verify the model with the newly implemented changes.

This project is structured into the multiple phases seen below:

- Information gathering on communication protocols.
- Analysing network-traffic generated from existing code implementations.
- Simulation of solutions in UPPAAL.
- Comparison of results and conclusion.

Chapter 2

Problem Analysis

The following chapter will provide knowledge on selected network protocols and technologies that all relate to the project. Section 2.1 will briefly explain network protocols and what they are used for. Subsection 2.2.1 is about the UDP protocol, and subsection 2.3.2 is about the DTLS protocol and how it is built on top of UDP. Section 2.4 is about Network Address Translation, how it is used, and what problems it incurs. Section 2.5 is about IoT devices their states, and their estimated power usage. Section 2.6 is about sockets used in DTLS alongside some early implementations of DTLS in WolfSSL and OpenSSL. Finally, section 2.7 concludes upon the entire problem analysis and leads into the specified problem statement.

2.1 Network Protocol Stack

The internet has grown to be a huge part of our lives, facilitating many everyday needs. From an outside user, the internet may seem as a magic black box, where you just do some things and it works. This magic black box, however, is the combined work of many computer scientists and engineers over the years.

The internet is essentially just many computers, all interconnected with each other. This is done by a physical medium linking two or more computers to each other. Sitting from home, the first link is often one's home router or modem. Next, the router may be linked to a variety of other routers, which in turn are linked to other routers. In this manner, a lot of devices are interconnected with each other through some physical medium [8].

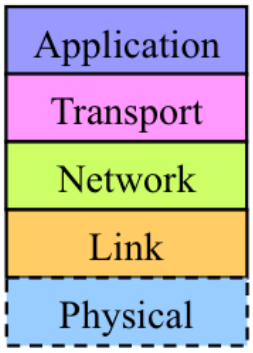

Figure 2.1: 5-layered protocol stack [6].

This however still leaves a lot of problems, such as exactly how to communicate over the Internet. How do we find our way through the network? How do I know if my message has been received? Is it safe? Questions like these can often be answered with "Depends on what you want?", as there exists a large variety of protocols designed to solve these exact problems, depending on one's desired use.

These protocols are often implemented to fit a layered architecture, such as the protocol stack. The protocol stack divides the work and responsibility amongst different abstraction levels, with each layer only being interested in what the other layers have archived, and not how they archived it. This reduces complexity and is in general a very useful architecture, but it also has another very desired trait. Since each layer is not concerned with the process of another layer and only its result, layer X can easily use a different protocol to get the job done. Layer X will carry out its protocols in isolation and deliver the needed messages to the layers above and below.

A typical 5-layered protocol stack can be seen in figure 2.1, which uses the 5 layers; Application, Transport, Network, Link, and Physical (Specified in section 2.1.1 throughout 2.1.5).

More technically, the layers are wrapped in a header with information for the receiving device's corresponding layer. So the network layer will wrap what is received by the application layer in a header with information for the end systems network layer. This is often referred to as enveloping. Imagine each layer putting the envelope received from the previous layer in another envelope with relevant header information. This way, intermediate routers can unwrap only the layers needed for further transmission, and may also "rewrap" some layers during transmission. The following sections will further explore the responsibility of each layer in 5-layer stack (see figure 2.1) [8].

2.1.1 Application Layer

The application layer is responsible for the communication between the applications running on the two end systems. Here, the payload is often referred to as the message. This layer should only be unpacked by the end-point and rarely has anything to do with the traversal through the network [8].

2.1.2 Transport layer

The transport layer is responsible for collecting application data from the application layer and transporting them between the two endpoints. These endpoints are the sockets of the two end-systems. The transport layer's main purpose is to divide the messages from the application layer into packets suitable for the network layer, and on the receiving end collecting the packages and sending the full message to the application layer. Common protocols for this layer are TCP and UDP, which are two well-known protocols for data transmission. This layer is often only unpacked at the two end systems [8].

2.1.3 Network Layer

The network layer is the first to be unpacked at the intermediate links. The network layer's responsibility is to transport the segments from the transport layer, through the internet to the desired destination. A very common protocol here is the IP protocol, which transports the segments through the internet with the help of IP addresses. This layer is unpacked at the intermediate links of the network, to allow other links to forward the segments to the next link towards the end system [8].

2.1.4 Link Layer

The link layer is responsible for transferring communication between one link and its immediate neighbor. The protocols used on this layer may be different among all the intermediate devices in a connection, and the header on this layer may be changed by the intermediate devices [8].

2.1.5 Physical Layer

The physical layer is responsible for breaking the individual packages into bits, on the form needed for the physical transmission. This can be light pulses (fiber-optics), radio waves (wireless communication), or electric pulses (wired communication). Like in the link layer, the protocols used in this layer differ throughout a connection [8].

2.1.6 Missing Security Layer

The previous description of the 5 different layers in a 5-layer protocol stack will ensure, depending on chosen protocols, a transferal of application data between two end systems over the internet. Different protocols can make this connection reliable, stateless or connection-based, but the stack does not intuitively support secure communications.

One further abstraction is needed to provide a secure connection between two users, since the model just described is not currently able to support one of the most common security protocols, that being TLS. TLS is a protocol that is mostly suitable for the TCP protocol, and will setup a secure connection between the two end systems before the transmission of application data begins. This puts the protocol somewhere around the application and transport layer. The protocol unfortunately cannot be placed nicely into the stack, since it can be argued to appear both within and after the application layer, or within the transport layer. Later sections will go into more detail concerning the TLS protocol, but it is worth noting that, just because it does not fit nicely into the model, it is still a very useful tool that still cooperates nicely with the protocol stack.

The following sections will go into more detail on a selection of protocols worth noting for this project.

2.2 Transport Layer Protocols

Two common protocols which are used in the transport layer are UDP and TCP. Both have different use cases, with a generalization being UDP is rather light, whereas TCP has more useful properties. The following section will discuss the two protocols, with a summation and further comparison at the end.

2.2.1 UDP

The User Datagram Protocol (UDP), is a protocol used for transmitting a user's application data over a network [22]. The protocol is part of the transport layer, and operates upon the internet protocol, referred to as a protocol suite UDP/IP [19]. UDP is a connection-less protocol, meaning no connection (both parties agreeing on their communication and how to do it) needs to be established between the two end systems before transmission. This can cause a system to send UDP datagrams to another system, not knowing if the system is online or exists at all. UDP is also not a reliable connection and package drops are not handled. This all contributes to a fast transmission of data with, in some cases, no significant negative side effects.

The UDP protocol "lives" in a socket on the port that the application sends its messages to. The protocol then splits the application data into datagrams that are individually sent to the network layer. The UDP protocol does ensure checksum within each datagram, to detect errors in the data received. This is handled by the protocol on the receiving end system, before a message is relayed to the receiving end system's application layer. Packages with a bad checksum are dropped [18].

UDP datagram header

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port															
4	32	Length																Checksum															

Figure 2.2: The structure of a UDP header [25].

UDP, like many other protocols, forwards its messages to the receiving protocol via the headers. The header is wrapped around the UDP datagrams, and are sent with the message [23]. The UDP header, seen in figure 2.2, is quite simple and has a fixed size of 8-bytes. This contains all necessary information, whereas the remaining space consists of UDP data. Each UDP port number field consists of 16 bits, which is used to separate the user requests. The header includes four different parts. Firstly, a source port specifying the port from which the data came from. Secondly, a destination port specifying the port where the packet should be delivered. Thirdly, the length of the UDP data and lastly a checksum to check for errors, which is not mandatory [19].

2.2.2 TCP

The Transmission Connection Protocol (TCP), is a heavily used transport protocol due to its sought out properties. TCP divides application data into segments, and in contrast to UDP, TCP ensures the ordering of messages and a reliable transmission, with its connection-state approach. TCP establishes a connection with the end system before transmitting data, thereby ensuring the segments are sent to an existing live server, before transmission start. This also enables the sending end system to retrieve information on whether the connection failed or succeeded.

As mentioned earlier, TCP also ensures reliable and in-order message transmissions. This means that a segment is resent if it were to be dropped, and that out-of-order messages are reordered, before forwarded to the application layer. In-order transmission is ensured by assigning each segment in the stream to a sequence number, which allows the end system to order the segments in the same sequence that they were sent in. To ensure reliable transmission different approaches have been implemented throughout the years, to optimize performance, but still the essence is the same. The receiving end system will send an ACK (Acknowledged message) when a segment is received, and a NAK (Negative Acknowledgement) when a message is missing from the sequence.

TCP segment header

Offsets		0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
0	0	Source port																Destination port															
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved 0 0 0 0				CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size															
16	128	Checksum																Urgent pointer (if URG set)															
20	160	Options (if *data offset* > 5. Padded at the end with "0" bits if necessary.)																															
⋮	⋮																																
56	448																																

Figure 2.3: The structure of a TCP header [24].

TCP, like UDP, also operates in the socket of the end systems. TCP wraps each segment in a header, with quite a bit more information than the UDP header (see figure 2.3). UDP has a header of a collected size of 64-bits, with the TCP header being at a minimum 160-bits having fields like sequence number and ACK number

using a combined 64-bits, along with other fields. TCP also uses checksum for error detection.

2.2.3 Pros and Cons

UDP and TCP both have positive and negative aspects. UDP is cheap, but does not provide a lot of sought out properties like reliable and ordered messages. TCP however, does ensure these properties, but with the cost of a 3 times as large header. This is the reason that UDP and TCP are used in different environments.

2.3 Security Protocols

The following section will discuss the DTLS protocol in detail, and how it enables TCP-like properties on the UDP protocol. The section will have a short introduction to the TLS security protocol, as to assist in the explanation of DTLS.

2.3.1 TLS

TLS is a security protocol ensuring secure and encrypted communication over the internet. Proper secure encryption between two parts is not a trivial task. One has to assume that everything sent over the internet unencrypted is compromised.

Secure communication can roughly be divided into two parts; secure key exchange and authentication. Making sure data is encrypted during transmission is not worth much, if one cannot make sure the other end is who they say they are. The TLS protocol ensures secure key-exchange with the use of the Diffie-Hellman key exchange algorithm, and secure authentication with the use of some digital signature algorithm.

TLS will establish a secure connection via TCP by safely exchanging keys and authenticate the other end system, before any application data is transmitted. This is also referred to as a TLS handshake, and forces the first x transmissions to not include application data while the secure connection is being established. Different methods exist to lower the value of x, with some methods managing to lower x to 0 resulting in a 0 round trip time (RTT) handshake (a faster connection allows the

client to send data without awaiting the server to respond).

The rough makeups of TLS is; establish a secure connection with the help of a TLS handshake, exchanging keys and authentication is a secure manner. The next section will discuss DTLS which is a protocol trying to establish a connection with the same properties of TLS, but on the UDP protocol.

2.3.2 DTLS

In order to ensure a secure and reliable connection when using the UDP protocol, the datagram transport layer security protocol (DTLS), is used. This protocol is built upon the transport layer security (TLS, section 2.3.1) protocol to establish a secure, authenticated and confidential communication channel between a client and a server. The two security protocols are primarily used for different Transport protocols, with TLS needing a reliable channel for transportation, and as such typically uses TCP, while DTLS uses datagrams for communication, and as such it uses the UDP protocol described above (Section 2.2.1).

Some key issues with implementing a TLS like protocol on UDP is the unreliable nature of UDP. TLS expects a reliable transmission, meaning handshake messages cannot be lost. TLS will break if any handshake message is lost, so DTLS must be implemented to handle lost handshake messages. This is done by having a timer on both server and client side, re-transmitting a handshake messages if no response is given before the timer expires. This should work for both scenarios of lost client handshake messages and a lost server handshake messages. In a scenario where the client sends ClientHello, it will expect to see a HelloVerifyRequest. If the timer expires, the client knows that either the ClientHello or the HelloVerifyRequest was lost, and the client will resend the ClientHello. Either the server will never know a message was lost, or it will know to resend the HelloVerifyRequest.

For packages outside the handshake, TLS breaks on package loss, since it does not allow for independent decryption of packages. If package N does not arrive, the integrity check of package N+1 will not be correct.

To support the exchange of data, a secure connection between two parties should be established, through a handshake, initiated by the client sending a ClientHello message. This process can be seen in 2.4. The server will respond to the ClientHello message with a HelloRetryRequest [12], which has the same format as the ServerHello request [11]. The HelloRetryReqeust has a cookie identifier sent

with it, which needs to be included in the next ClientHello message in order for the server to verify the client based on the cookie.

As a default, cookie exchange should be performed at the beginning of every new handshake, because this exchange is used to protect against different denial-of-service attacks. There are two types of attacks which is of great concern. The first attack is where an attacker consumes excessive amounts of resources on a server by making excessive amounts of handshake initiation requests. The second attack is where the attacker floods a targeted client, by using the server as an amplifier to send connection initiation messages to the victim. In relation to the second attack type, there are a few exceptions in which the cookie exchange may be chosen not to be performed. One exception is when operating in an environment where amplification is not a problem. Another situation where cookies may be excluded is during session resumption, which can be done though PSK. If the servers process 0-RTT requests and sends 0.5-RTT responses, where the size of the outgoing message greatly exceeds the size of the messages received, these servers are in a risk of being used in an amplification attack [12].

After the cookie exchange is done, the server can proceed the handshake by negotiating the handshake parameters, including the first encrypted message, as well as optional parameters [12]. These include asking for the client's certificate from which the endpoints can be set, a signature which corresponds to the public key, an authentication giving key confirmation, and different parameters about the session that is negotiated with the client [11]. When the client receives the message, it ensures that the server also uses DTLS version 1.3. This is done to protect against downgrade attacks [12].

To finish the handshake, the client sends necessary information back to the server. This includes information about how to generate key material, the connection version, a client ID, which must match the one sent in the ClientHello message, and a cookie value that is null [11]. If any of these requirements are not met, the handshake is aborted. If all requirements are met, the server sends an acknowledgment, and optionally application data and a secure session between the client and server is established for the application data to be transmitted between them [12].

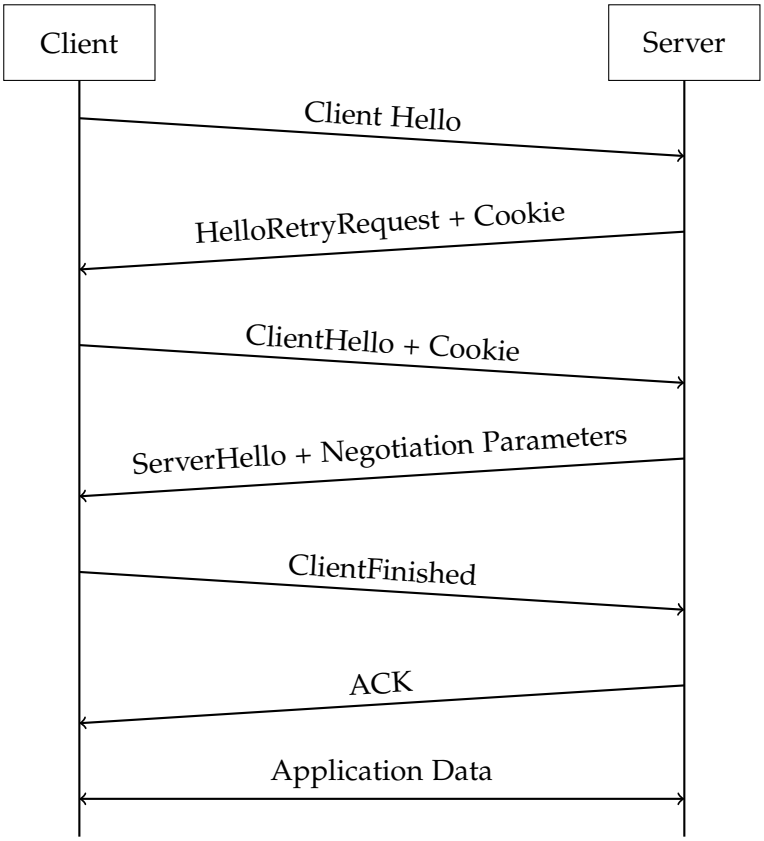

Figure 2.4: Example of a full DTLS 1.3 handshake.

2.3.3 Session Resumption

A feature in DTLS, called “Session resumption”, allows the server and client to resume their session in an already existing DTLS session. This can be done instead of completing the whole handshake process again. Resuming the session reduces overhead, as the communication skips the certificate authentication process. An example of the use of this feature is when multiple connections are established between the same client and server. In DTLS versions before version 1.3, the functionalities used for session resumption were provided by either the "Session ID’s" approach or the "Session tickets" approach. These have been obsoleted in version 1.3 by "resumption" with Pre-Shared Key (PSK), which is depicted in 2.5. The approach with PSK in version 1.3 functions by establishing a new communication using an already established key from a previous connection. A PSK can only be created after a handshake has been completed once before. To establish a PSK connection, the server requests a PSK identity. This PSK identity is derived from the unique key in the initial handshake. If the client sends a PSK identity, a PSK connection will be established on the existing session. The new connection is therefore cryptographically tied to the previous connection between the client and server, however, only if the server accepts the PSK. When the server accepts the PSK with resumption, the cryptographic state is bootstrapped with the derived key from the initial handshake, instead of completing a new full certificate-based authentication handshake.

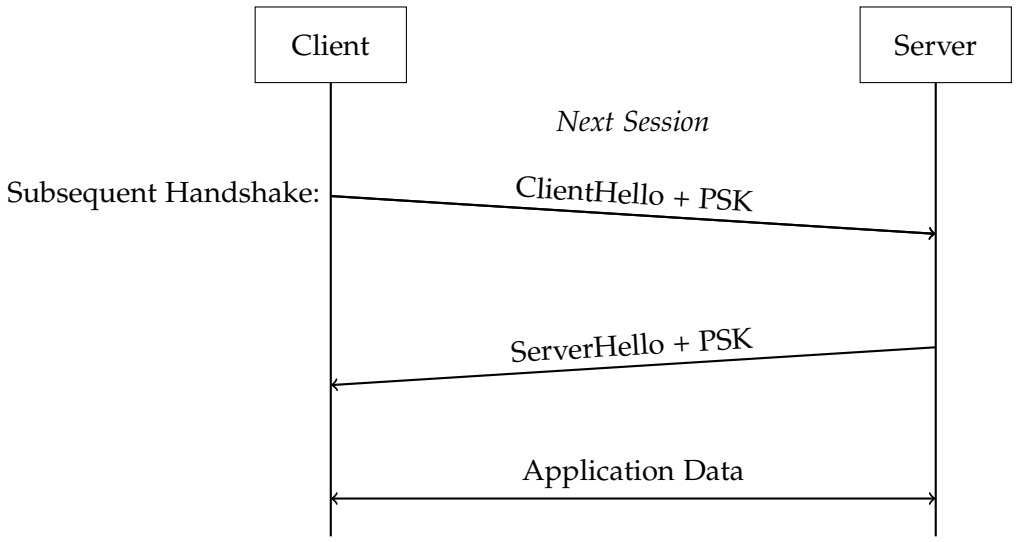

Figure 2.5: Example of session resumption without cookie exchange.

2.4 Network Address Translation

The following section will explain the concept of Network Address Translation (NAT); what it is, and what it is used for.

Network Address Translation (NAT) is a technique used in routers to allow multiple devices on a local network to share a single public IP address for accessing resources on the internet. NAT plays a part in managing and converging IP addresses. NAT can be used in a local network, to map multiple private addresses to public IP addresses, before the request is transferred over the internet [10]. Within a network, each device is given a unique private IP address to use when communicating internally. These private IP addresses can dynamically be assigned by the order that they connected to the network, and every subsequent connection is incremented by one, but they can also be statically assigned [5]. Private IP addresses are only used to communicate internally in a network, as multiple networks can use the same private IP addresses. Therefore, a public IP address is needed to communicate externally. The public address is commonly a single IP address that the network's router controls [10]. An example of how NAT translates private into public addresses can be seen in figure 2.6.

Also, it is not possible to directly access a device on a private network. To access a device on a private network, one has to use their public IP, which will route the message to the private networks router. For the router to further this message to the right device on the network, it has to have an entry in the NAT table specifying where the message is suppose to go within the network. This can either be a dynamic entry or a static one.

The dynamic NAT entries are removed from the router's NAT table when the host hits a specified timeout for communication. When this occurs, the address will be reused later by another host, by assigning its value to the pool of addresses [3].

2.5 Internet of Things

Internet of Things (IoT) is a network of appliances, devices and other items containing embedded software, sensors and hardware allowing for wireless connections. This allows for remote collection and sharing of data. When thinking about IoT devices, they are usually in the context of smart home devices such as lights, cam-

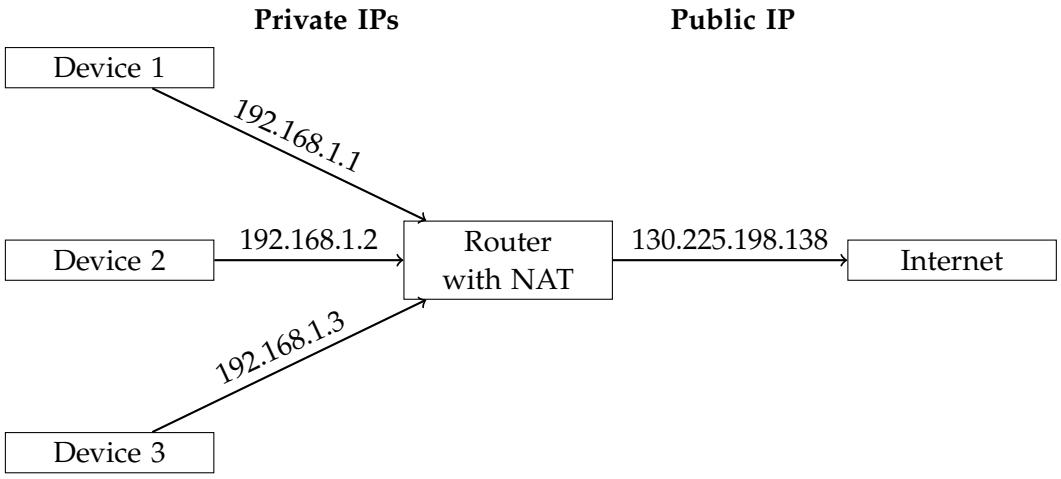

Figure 2.6: How NAT works.

eras or smart locks. Such devices can usually be controlled from an app on your phone. For this section, IoT will be tough of as a wireless connected device that can be remotely interacted with [21].

IoT is, by the nature of being small, usually constrained in a few areas like computing power, power supply (battery or wired) or range. IoT devices themselves generally only has a few responsibilities, like a temperature sensor reporting what it measures, a light turning itself on at specific times or conditions, or maybe a camera alerting of movement when set to be on alert. Data collected and sent by IoT devices can be quite sensitive and thus require a certain level of security. This can be problematic, as the lack of resources on some devices can restrict the type of encryption algorithms a device can achieve. Since asymmetric encryption algorithms are more resource intensive, only symmetric algorithms might be available on some devices. This introduces a new problem of how to safely get a shared key established for symmetric encryption. Access to the physical device might be needed to manually share a key for symmetric encryption.

If an IoT device is connected to a power source by wire, then there is usually not a concern of how much power the device draws. If an IoT device is instead battery powered, then there is an incentive to reduce power consumption since you would have to swap out or charge its batteries. This is inconvenient and may be unfeasible if you have a lot of such devices, or maybe they are in far away places. Instead, improving the efficiencies is a possible option like having specific efficient hardware, improving the efficiency of its main function, or reducing the amount of wireless communication a device requires.

2.5.1 IoT Power Consumption

In terms of power consumption, it would be nice to get an idea of which part of an operating IoT device requires what amount of power. This is not an easy task as there are many different IoT devices which all operate differently. In [15] they measured a narrow band IoT (NB-IoT) device, which is a low power wide area cellular technology for low-capability and low-cost IoT device. The device was measured to analyze the power consumption of performing different operations in isolation. These operations consisted of three symmetric cryptography algorithms (EEA/EIA 1, 2, and 3), sending/receiving data and more. They also measured how many watts it costs for the device to be in three different working modes (Working, light sleep, and deep sleep). Working, being actively running, light sleep is where the device only runs processes that enable receiving of outside signals or wake-up timers. In deep sleep, the device is unreachable from the outside and only a clock is running which is used to wake up the device after some time.

The results they get in [15] for the EEA/EIA 2 operations are shown in table 2.1. These operations cost are around the same as sending and receiving data, as shown in table 2.2.

One could assume power consumption does not change drastically between different IoT systems, when it comes to key generation, encryption and decryption. The numbers in 2.1 are likely very suitable to use when speaking of IoT devices in general, and can estimate power consumption of protocols using keys to encrypt and decrypt messages.

Operation	Power Consumption (J)
EEA/EIA 2 Encryption (200 bytes)	$\approx 0.05 * 10^{-3}$
EEA/EIA 2 Decryption (200 bytes)	$\approx 0.2 * 10^{-3}$

Table 2.1: Estimate readings of power consumption of EEA/EIA 2 a NB-IoT [15].

Transferring and receiving data, on the other hand, may change depending on the IoT system used. The information gathered from the paper [15] will be used, since a possible change should not change the comparative analysis by much, due to the power consumption changing by a constant factor across all measurements.

Lastly, looking at the power consumption they measured for the different working modes, which are shown in table 2.3. Here one can see that deep sleep is way more power efficient, which makes sense since the NB-IoT device is not doing much. Light sleep is only an order of magnitude less power-consuming than

Operation	Power Consumption (J)
Transferring (200 bytes)	$\approx 10^{-3}$
Receiving (200 bytes)	$\approx 10^{-3}$

Table 2.2: Estimate readings of power consumption for transferring and receiving data for a NB-IoT [15].

Work Mode	Power Consumption (Watt)
Working	$\approx 10^{0}$
Light Sleep	$\approx 10^{-1}$
Deep Sleep	$\approx 10^{-4}$

Table 2.3: Estimate readings of power consumption of the three different working modes for a NB-IoT [15].

working. So it is always better to be in deep sleep when possible.

The numbers provided by the paper [15] provide a good ratio to use between different modes. Since the paper analyses NB-IoT systems, the exact values may not be completely accurate, but using the ratio between the modes could prove rather useful. States of a device may not change a lot depending on the network they operate in, as long as they operate in the same manner. The paper [15] mentions eDRX as their light sleep method, which is a mode allowing the device to wake up in given intervals to listen for an incoming transmission before going back to sleep. The paper uses devices with a 4-5 second interval, so if similar methods are used in other IoT systems, the numbers provided should be assumed a good approximation.

2.6 Analysing DTLS on an active implementations

The earlier section 2.5.1, explained how power consumption in NB-IoT devices can be measured by how many bytes they send and receive. This following section is the result of analyzing the DTLS protocol on different real implementations from providers like wolfSSL and openSSL. The section will contain information about specific amounts of bytes needed to perform individual steps of a DTLS handshake. This leads to an analysis and the rise of a problem that may occur when trying to communicate safely over the internet.

This section will introduce the methods used during the analysis, as well as the final results and thoughts used to later formulate the problem statement.

2.6.1 Sockets

A Socket is an endpoint of a two-way communication. Sockets facilitate a bidirectional first in, first out communication over a network. To establish a connection, one socket is created at each end, each having an IP address and port associated with them. There are two types of sockets [14]:

- **Datagram socket**: This type has a connectionless point for receiving and sending packets. This means that the sender will not get any acknowledgement of whether the message was received. It is just a place where messages can be sent to, delivered and received [14].
- **Stream socket**: Can provide an inter-process communications socket or a network socket. Stream sockets provide a connection-oriented flow of data, meaning that a connection must be agreed upon before data can flow between the sockets [14].

The initial idea for a solution, was to modify/extend the DTLS protocol. This should be done in a way where the server can keep the connection alive, without using any client resources. But to do this, an initial implementation of DTLS is needed as a starting point. The group decided to look for implementations in the C language since it is very performant.

2.6.2 WolfSSL

WolfSSL is a lightweight TLS library that also supports DTLS targeted towards IoT, embedded and RTOS environments [26]. WolfSSL was the first supporting library the group tried to use for an initial DTLS implementation. Getting and building WolfSSL for Windows was a hard task that the group was not able to get running. There were plenty of examples of code for WolfSSL, but only for Linux/MacOS. Getting equivalent libraries for Windows that were used in the examples were tough and even when using them, getting WolfSSL to build still did not work. Therefore, the group opted to use WSL. It took about a week to get a connection between WolfSSL client and server up and running on the same device. However,

this setup quickly led to some major issues. Connections between multiple devices proved to be impossible when the server was connected to the university network, as the group had no access to port-forwarding. To remedy this, the group started testing the connections when hosting the server on their home networks. While testing the connections, both TLS and DTLS were used, but in each of the test cases DTLS failed while TLS passed. The group later found out that Windows does not support port proxying for UDP connections like it does with TCP. The group had been using this to forward traffic from Windows into WSL. This meant that the WolfSSL server could not be run in WSL, so either a MacOS or Linux machine was needed to host it. After troubleshooting this issue, a connection was finally established using WolfSSL where a Macbook ran the server and the client used WSL.

2.6.3 OpenSSL

OpenSSL can be used similarly to WolfSSL, as it is an open-source implementation used for cryptography and secure communications [20]. The benefit of OpenSSL compared to WolfSSL is the amount of optional arguments that can be enabled to show detailed information on each message sent back and forth in a communication. This allowed the group to get a greater technical understanding of the DTLS connection and the extra parts like session resumption, pre-shared keys and cookies. There were however some problems, as the group tried to exclude the cookie exchange when it came to session resumption. None of the configurations of the server functioned as intended, and each of them kept the cookie exchange. To try and find a solution for this issue, multiple versions of DTLS were tested, as well as trying multiple DTLS versions with WolfSSL to no avail. So while it is technically possible, the implementation of DTLS used for the project does not allow the exclusion of cookie exchange. OpenSSL also did not have a DTLS 1.3 implementation, so when it came time to test 1.3 specific features, the group had to return to wolfSSL.

2.6.4 Wireshark

After a non-satisfactory outcome from the experiments regarding cookies, the group decided to stop testing the cookie exchange. Instead, session resumption was tested. While OpenSSL had a relatively satisfactory amount of data, the group wanted a few more statistics. To easily access these, the group decided to use the network protocol analyzing tool "Wireshark". Wireshark allows a user to monitor

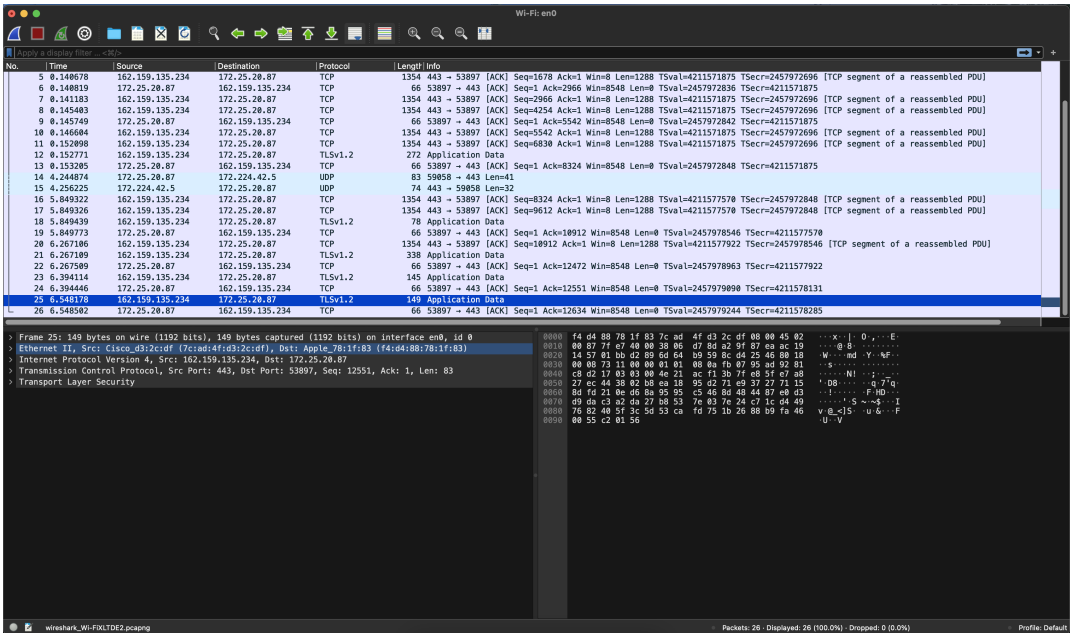

Figure 2.7: Example image of Wireshark.

all communication that occurs to and from the network. An example of Wireshark running can be seen in figure 2.7.

Within this example, one can see the three main windows of Wireshark. The upper window shows all communication scanned by the program. This includes a timestamp, source/destination IP and the protocol used. It also showcases the length of each transmission in bytes and an info panel, but said info panel has varying amounts of detail from the implementation monitored, and thus cannot be used for a lot.

The bottom left window in figure 2.7 will showcase almost all data when a connection is selected in the top window. This data is split into folders as to keep it readable and organized. Lastly, the bottom right shows the hex data that was transmitted during the connection. One can hover their mouse over this hex data to see the individual segments, and if a segment is clicked, the respective section is opened in the bottom left data viewer.

Wireshark aided in understanding some of the weirder behaviors, as each part of each connection step can be seen individually. The group's understanding of session resumption and PSK's benefited greatly from using Wireshark, as the group were unable to find any good metrics for DTLS 1.3. (OpenSSL would show the data, but only works for DTLS 1.2 in the group's experience)

Lastly it allowed the group to test an idea for keeping the connection alive by simply letting the server ping the client with "phantom messages". By doing this, the communication can easily be continued by the client without the need for a new handshake.

2.6.5 Test Results

After a long battle with Windows, OpenSSL not having a DTLS 1.3 implementation, and Wireshark not recognizing and identifying DTLS 1.3 protocol packages, the code implementation of the protocol leads to some important results in the form of numbers and knowledge. These proved to be very valuable later on in the project for changing and setting up the initial DTLS protocol in UPPAAL, but also for calculating power consumption of the handshake with knowledge of package sizes for each step in the protocol.

OpenSSL, DTLS 1.2

As mentioned earlier, openSSL has a user-friendly implementation of both a server and client, ready to call with any needed parameters. These implementations would also provide information about the connection and the packages sent, including information about total bytes sent and received during a handshake. These tests for DTLS 1.2 lead to the results seen in table 2.4. An interesting pattern can be seen when looking through the data sent by the client and server, which shows that when the amount of data sent by the server decreases, the data sent by the client increases.

Type of connection	Server	Client
Default	2253	563
PSK	1890	927
PSK and Session resumption	229	1075

Table 2.4: Bytes written by server and client during a single handshake using the DTLS 1.2 protocol implemented by openSSL.

This happens due to the responsibility of using PSK and session resumption mainly residing with the client, meaning that the client has to inform the server about its PSK identity, and/or its session ID. This, combined with the fact that cookie exchange forces the client to send its ClientHello messages twice, means

that all additional information added in the ClientHello, will also be sent twice. This will thus increases the total amount of bytes sent by the client.

This discovery was useful information, but the group needed to find similar results for DTLS 1.3, and since openSSL was limited to DTLS 1.2, the group moved over to wolfSSL. WolfSSL did not have a test server and client implemented, forcing the group to write their own server and client implementation. Implementing the base of an advanced security protocol proved rather easy, but changing and interacting with the protocol to provide further information was quite a bit harder.

WolfSSL, DTLS 1.3

The group used Wireshark to analyze the packages sent between the server and the client. Wireshark gave even better information than the openSSL server/client implementation did, and gave the results seen in table 2.5. Cookie exchange was omitted in this analysis, since the group believed cookie exchange would be omitted in the favor of PSK. The DTLS documentation describes how cookie exchange may be omitted when PSK or session resumption is used [12]. Leading the group to ignore cookie exchange in favor of a lower power-consumption at this time.

Info	Size (bytes)	Endsystem (Sending)
ClientHello	606	C
ServerHello	186	S
EncryptionExtension	78	S
Certificate	1345	S
Certificate Verify	336	S
Finished	108	S
Finished	108	C
ACK	82	S
Total	714	C
	2147	S

Table 2.5: Transmission between server/client in a system using default DTLS 1.3, with cookie exchange omitted.

From table 2.5, it can be seen that sending the certificate is by far the largest task that has to be sent by the server. In addition, it is also an encrypted action, and as such a large amount of bytes has to be encrypted by the server, and then decrypted by the client. Something else worth noting is that the ClientHello includes a long list of cipher suits. Depending on use case, the client can be programmed to omit

sending a long list of available ciphersuites, and just include a few that are relevant for its use case.

Cookies were omitted during the experiments. Cookie exchange also includes a HelloVerifyRequest with a cookie with the size of 60 bytes, and resending of ClientHello with the cookie attached.

Info	Size (bytes)	Endsystem (Sending)
ClientHello	668	C
ServerHello	192	S
EncryptionExtension	78	S
Finished	108	S
Finished	108	C
ACK	82	S
Total	776	C
	460	S

Table 2.6: Transmission between server/client in a system using PSK/Session resumption in DTLS 1.3, with cookie exchange omitted.

Session resumption and PSK are combined into one in DTLS 1.3. When using PSK and Session resumption, a small amount of information is added to the ClientHello and ServerHello, in exchange for not sending the certificate. Since the size of the certificate accounts for a large amount of the bytes encrypted, decrypted and sent during the handshake, skipping this part saves a lot of time and computational power.

Heartbeat and routers

On a final note of the information gathering using a real implementation of the DTLS 1.3 protocol, the group tried simulating a heartbeat on different networks. Firstly, a package containing a singular letter of 4 bytes, would amount to a final package size of 66 bytes. The size of the package stem from the letter being encrypted as well as headers. In an unreliable UDP connection, the client would intuitively also need some sort of acknowledgement of the server receiving the heartbeat, meaning an ACK message of similar size would have to be sent by the server. With the total bytes used in a PSK/Session resumption handshake and the bytes used to send a heartbeat, mathematically the client would be able to send $776/56 = 13.86$ heartbeat per handshake, and the server would be able to send $460/56 = 8.21$ heartbeat per handshake. These numbers were very disappointing,

leading the group to seek alternative methods.

For this project, the heartbeat has one primary purpose, as well as a large inconvenience. The primary purpose is to keep the server's entry in the NAT table of the client's router alive, with the inconvenience being that the UDP connection is unreliable, causing an ACK message to intuitively be needed.

The primary purpose of the heartbeat is to not drop the server's entry in the NAT table of the router, meaning sending data through the specific DTLS connection may not be needed. The group honed in on the primary purpose, figuring that the DTLS protocol maybe was not needed to keep the server in the NAT table. Experiments were made with a client/server setup, where a UDP connection between the two endpoints were established using a DTLS handshake. After the connection was set up, the client and server exchanged a few encrypted messages, before the client entered a simulated sleep state. The server would continue to send the client heartbeats over the UDP connection, without encrypting the message with the keys acquired from the DTLS protocol. Sending one byte in this manner would create a package with a total size of $8 + 1 = 9$ bytes, for a normal UDP header and 1 byte of application data, and would allow the server to stay active in the NAT table. This method would allow for $460/9 = 51.11$ heartbeats to be sent by the server per PSK handshake.

Focusing again on the inconvenience mentioned before, where UDP is an unreliable connection, this could result in a heartbeat never reaching the client's router. In this case, the server is dropped from the NAT table, and the server is none the wiser. Heartbeats will continuously be sent by the server, with the router on the client side discarding the messages upon being received. The server will only realize the connection is down upon requesting data from the client, and expecting a response. Investigating the ACK problem was put on the back burner, while more about how the router operates was investigated.

Earlier in section 2.4, the workings of the NAT table were described. As mentioned earlier, the server had to be hosted from within a group member's network to allow port forwarding, but the client could be on any network. Therefore, on group member had a server running from their home network, while another member had the client running from both the university's network and their home network.

At home, with no one else using the internet, keeping one entry in the NAT table proved trivial. The entry would only be dropped on max timeout, which was the max at 5 minutes in this case. Test were made where the NAT entry timed-out,

and the server were not able to contact the client. Upon the client contacting the server through the connection with the timed out NAT, the entry was reentered into the NAT table, and the connection was again live and well.

On the university's network, the server would lose its entry in the NAT table a lot faster. With a busy router, old entries will be removed in favor of newer connections, allowing more people to operate on the internet. This meant that heartbeats needed to be sent more often and with an unknown optimal interval to ensure that the connection was kept alive. Usually, the client was able to revive a connection, but in this case it was not always true. When a router receives a connection from within the network, it assigns the device a port used for the rest of the traversal. When working with a busy router, the device may be assigned a different port when it revives the connection, leading to the server not recognizing the device, which is identified by its IP-address and port number. This is a separate problem, discussed later in section 2.7.1, which the group did not choose to focus on.

2.7 Concluding on the Problem Analysis

This section will present two challenges arising from low-powered IoT devices deployed behind a router that implements NAT.

2.7.1 NAT Rebinding

In order for a DTLS peer to choose a security context of an incoming DTLS record, the peer must be able to identify the origin of the incoming record. This is typically done using a 5-tuple, consisting of source IP address, source port, transport protocol, destination IP address, and destination port. However, IoT devices can be prone to NAT rebinding. This can for example happen because of extended sleep periods to save power or unstable network environments. When a NAT rebinding occurs, the public address and port of a device may shift during a DTLS session. This means that the identifying 5-tuple used to identify the appropriate security context, has changed, and as such can no longer be found [13]. A scenario where the client attempts to transmit data to the server, after a NAT rebinding occurs, can be seen in figure 2.8 below.

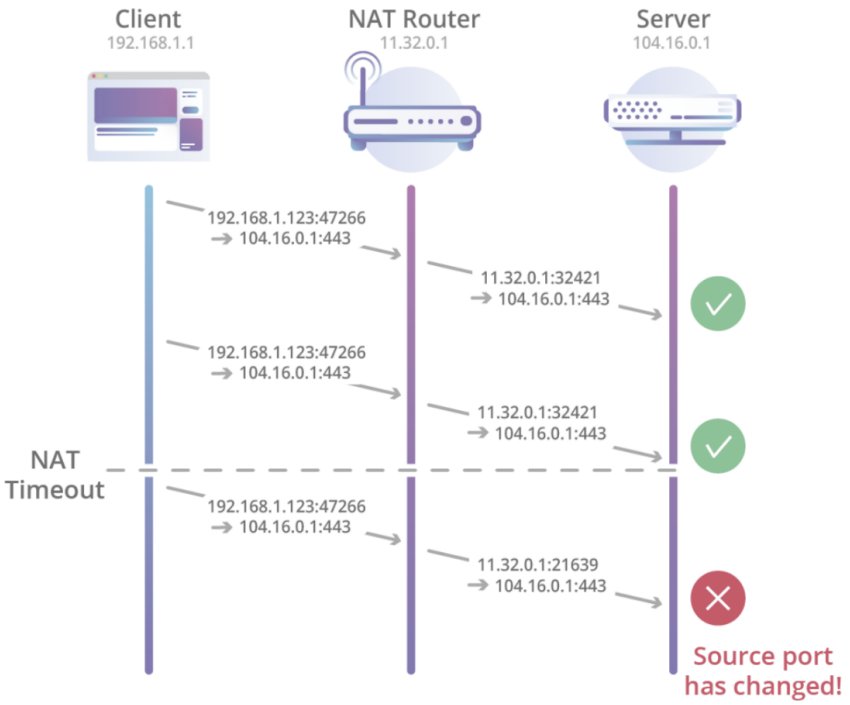

Figure 2.8: A client attempts to send data to the server before and after a NAT rebind occurs [7].

The consequences of such a rebind are costly in terms of power for a low-powered IoT device, as this causes a prolonged timeout, where the client attempts to connect to the server, before giving up and re-running the DTLS handshake.

A solution to this problem has been introduced alongside DTLS 1.3. The solution introduces a Connection ID carried in the record layer header of a DTLS datagram, which is then used by the receiver to identify the appropriate security context. However, this solution introduces security vulnerabilities, because it gives bad actors a new possibilities to perform D(DoS) attacks [13].

2.7.2 On-Demand Connectivity

From the results of section 2.6.5, it can be seen that keeping a connection between two end systems open over a longer period can be quite hard. In an IoT setup, contacting the client may be hard or sometimes impossible. After the client contacts the server, an entry in the NAT table is made, which at this point is the server's only point of entry for communicating with the client.

As was seen, when this entry is dropped the server is at the mercy of the client to reestablish the connection, since only the client must ping the server in order to get a new entry in the NAT table. It is also worth noting that NAT rebinding as mentioned in section 2.7.1, could become an issue here as well, but will be disregarded in this minor discussion.

The client could periodically send data or ping the server, but this would not allow for on-demand connectivity, which in some cases is a much required trait. For low powered IoT devices, constantly pinging the server to keep a connection open is also not a viable option due to battery power constraints.

Heartbeat may be a viable option for this problem, but that comes with it's own set of problems as explained in detail in section 2.6.5. One must find a way to keep the servers entry in the NAT table, allowing the server to make on demand request to the client, but this should also not be more expensive than other means. Other methods likely involves a new handshake, or requires the client to use power constantly by updating the server with the data collected, which as stated is seemingly not a viable option.

Chapter 3

Problem Statement

Within the problem analysis, the general functionality of DTLS and by extension UDP and NAT has been explained. Some problems regarding power consumption and constant connectivity in IoT devices, as well as NAT rebinding and on-demand request has been discovered and elaborated upon. These problems exist within the same realm, with one most likely having a solution the other benefits from. Although only one of the problems will be in focus, with the other assumed not a problem, in the context of this project. This has led to a problem that the group would like to further research and potentially solve. This problem is stated as follows:

"How can a DTLS connection between a client and a server be kept alive in the NAT-table of the client's router, allowing the server to send requests to a client, while reducing client power consumption and lowering response time?"

Chapter 4

Proposed Solution

This chapter will briefly explain the main proposed solution which the group explored and tested within the rest of the project.

The main proposed solution is based upon the idea of heartbeat packets being sent from the server to the client in order to keep the session between them open. The idea behind the solution was to modify the heartbeat setup, as the usual implementation requires the client to send an acknowledgment back to the server, whenever the server sends a heartbeat package to the client [9]. In the solution however, the heartbeat connection simply skips the acknowledgment from the client, to reduce energy consumption for the client. This allows the client to stay mostly idle, while still keeping the communication autonomous. This autonomy is achieved by having the client try to reestablish communication, if it has not received a heartbeat packet in a long enough period to remove the server from the client's router's NAT table. The server will then continue to send heartbeat packets on this new communication, and as such it is back to the status quo. Theoretically, this setup would give large improvements to communications with frequent data transfer, while having diminishing returns at larger data transfer intervals. While this solution is theoretically great for the client, it might severely increase the power usage on the server side. This is is expected, as the server is required to send the heartbeat packets at the designated frequencies to keep the communication alive.

An important note on the proposed solution, is the major pre-requisite for the success of the solution. The group is expecting to use a setup that allows the client to be in a sleep-like state, while being ready at any time to receive and send packages. A state like this is realistically available, as some existing set-ups already

allow this [2]. With mobile communication methods like DRX and eDRX existing for this purpose. Having a device sleep for short periods of time, before opening up its router functions to listening for request, then going back to sleep, is not a unimaginable concept. On this notion, the group abstracts away from specifying how this is achieved, and assumes it is possible.

Chapter 5

UPPAAL

This chapter aims to introduce UPPAAL and explain a little about how it works. Section 5.1 explains how to model and verify using UPPAAL. Afterwards subsection 5.2 focuses is on the extension UPPAAL SMC, what is different, and how it can be used to get statistical information about a model.

5.1 UPPAAL Core

Making a practical implementation change to the DTLS protocol is beyond the scope of the project. In any case, it is more fitting to analyze the impact of proposed changes to a protocol, before actually implementing these. As such, UPPAAL will be used to model the DTLS protocol and possible changes to the protocol, as well as theoretically analyzing the impact of these changes on the protocol.

UPPAAL is a toolbox developed for model validation and verifications of systems that can be modeled as networks of timed automata. UPPAAL was chosen due to its ability to model the system and validate as well as verifying the system's correctness, safeness, reachability, and fault tolerance.

Definition 1 (Timed Automata) *From [1], a Timed Automaton is a tuple* $\langle L, l_0, C, A, E, I \rangle$ *where*

- L *is a set of locations,*
- $l_0 \in L$ *is the initial location,*

- *C is the set of clocks,*
- *A is the set of actions, co-actions and the internal τ-action,*
- *$E \subseteq L \times A \times B(C) \times 2^C \times L$ is the set of edges with their action, guard and a set of clocks that are to be reset, and*
- *$I : L \to B(C)$ assigns invariant to locations.*

UPPAAL is based on the theory of timed automata, which by definition is a finite state machine extended with clock variables. The formal definition for this can be found in definition 1, referenced from citation [1]. Throughout this section, an example of a device requiring a double press to turn on is used and a user whom can do only `press!` actions, which is depicted in 5.1 and 5.2, which are inspired from an example from [1]. In order to model the system, multiple templates are used and defined so they can run in parallel, and work together. In the example 5.1, there are three locations with the names `off`, `await` and `on` . In addition to the standard location which allows time to pass, they can also be marked as urgent, which ensures that time cannot pass while the system is in a state that contains this location. Another marking is committed. Committed has the same property as urgent, but also is a bit more restrictive because the system can only take transitions from the committed location. In the example, only standard locations are used. Locations can have invariants ensuring that the ability to enter a location and stay in it, is limited to whenever a predicate holds true. In example 5.1 there is an invariant on the `await` location saying that the clock x must be below or equal 5.

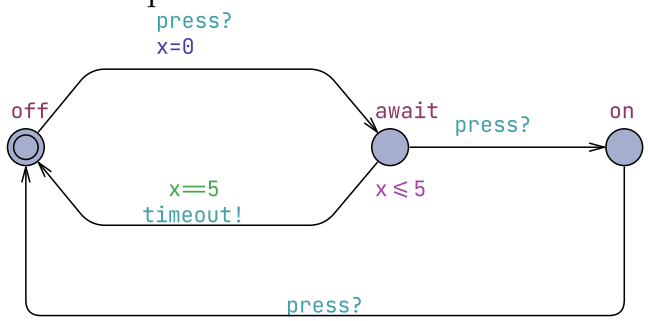

Figure 5.1: UPPAAL model of a device requiring two presses to turn on.

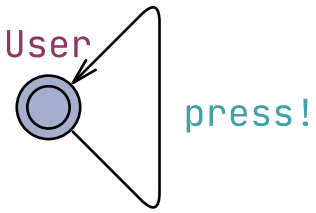

Figure 5.2: UPPAAL model of a user that can `press!`.

To connect the locations, edges are placed between them. These edges are used to specify how and when the system can move from one location to another. When traversing an edge, a transition is made. This can be seen in the example as moving from location `off` to location `await` or from location `on` to location `off`.

In relation to the locations, there are clock variables, which are considered in both time constraints through guards and invariants, but also when resetting the system. Firstly, looking at the guards. These can be seen in figure 5.1, for example where the edge which goes from the `await` location to the `off` location, has the guard `x==5`. This expression will be evaluated as a Boolean value, and if the clock x is equal to 5 then it will be able to do the transition. If the expression evaluates to false, then the only action which can be taken is synchronizing on the `press?` edge, and the system moves to location `on`. On edges, we can also have updates, which is where we can update variables and run code. In the example the edge which contains a synchronization on press, which goes from location `off` to location `await` has an update on x, which assigns it the value 0. This resets x making it ready to be used to measure time in the awaiting location. This update happens when that edge is traversed.

When two timed automata need to communicate, they can synchronize on a channel though the labels on the edges. In the example, this is done on the channel `press`, where the user does a press action using an exclamation mark after the channel name. The device uses a question mark on the `press` channel, which means it should wait on the request from the user. From this, the user and the device will be able to synchronize.

Another type of channel is a broadcast channel. Here, an action on a channel can synchronize with any number of receivers waiting for the action on the channel, as opposed to only one on a standard channel. Output actions on broadcast channels can always be executed in opposition to a standard channel, where a synchronization action can only be done if someone is ready to synchronize. The example showcases such a broadcast channel, i.e. the `timeout` channel. The device model in figure 5.1 can therefore take the `timeout!` output action, without synchronizing with anyone. If this were to be a standard channel, it would not be

possible to model the desired behavior in this manner. The type of channel has no visual indication in UPPAAL.

5.1.1 Standard Query Language

When the system has been modeled, the model needs to be validated. It is possible to verify on individual states through state formulas, or on the paths which can be further described in relation to reachability, safety and liveliness. When checking state formulas, it is possible to check whether the system is in a specified state, but also if there are states, which lead to deadlocks. This is done through the keyword deadlock, but it needs to be included into a reachability or invariant path formulas.

The reachability formula is used to check whether a given state formula possibly is satisfied using any reachable state [1]. This possibly property is modeled as `E<> p`. If the formula is satisfied, it will be evaluated to true, meaning that there exists a sequence of transitions, which reaches a state that satisfies `p` [16].

In relation to the safety properties, these will be states that ensures something good will happen [1]. Here there are also two properties, where the first is modeled `A[] p`, also called the "invariantly" property, and will evaluate to true if all the reachable states satisfy `p`. Then there is `E[] p`, also called the "potentially always" property, which says that there exists an infinite branch where the property `P` is true in every state on the branch.

The last path checking category is the liveliness property, which is used to check that the specified formula will eventually happen. This property is modeled by `A<> p`, also called the "eventually" property, which states that all possible branches eventually reaches a state where the property `p` holds.

Lastly, there is the leads to property, which is denoted $p --> q$, and states that whenever `p` holds, eventually `q` will also hold. A combination of the above properties can be used to verify the model, and ensure that it works as intended [16].

5.2 UPPAAL SMC

An extension to core UPPAAL, is Statistical Model Checking (SMC). The extension offers several techniques to monitor various simulations of the model with respect to some properties. The statistical results can be used to estimate the overall correctness of the model. The extension is used in cases where the verification of the model is beyond the scope of the symbolic model checker, like when the system model enacts stochastic behavior. Herein lies the key difference between classical UPPAAL and UPPAAL SMC, since it is possible to specify probability distribution, to control the model's timed behavior in SMC. Likewise, in SMC, one can compute the statistical estimation of probability, compare probability with value, and lastly compare two or more probabilities without computing them. Additionally, UPPAAL is restricted to only using broadcast channels.

UPPAAL SMC is capable of estimating the probability or testing whether the probability is greater than or equal to some value. The extension also offers the functionality of visualizing the statistical results in the form of the probability distribution over the number of runs stated. With the visualization of expressions, it is possible to observe the different behavior of the model [4].

Below in figure 5.3, there are given 3 stochastic timed automata from the UPPAAL SMC tutorial [4]. subFigure A2 and A3 implement different features of UPPAAL SMC. In subfigure A2, a branch point with weighted edges can be seen. The idea is that the branch point incorporates a probabilistic behavior, presenting a discrete probabilistic choice in the model. This means that there is a 1 in 6 chance that a transition on the upper edge is taken, and a 5 in 6 chance that a transition on the lower edge is taken.

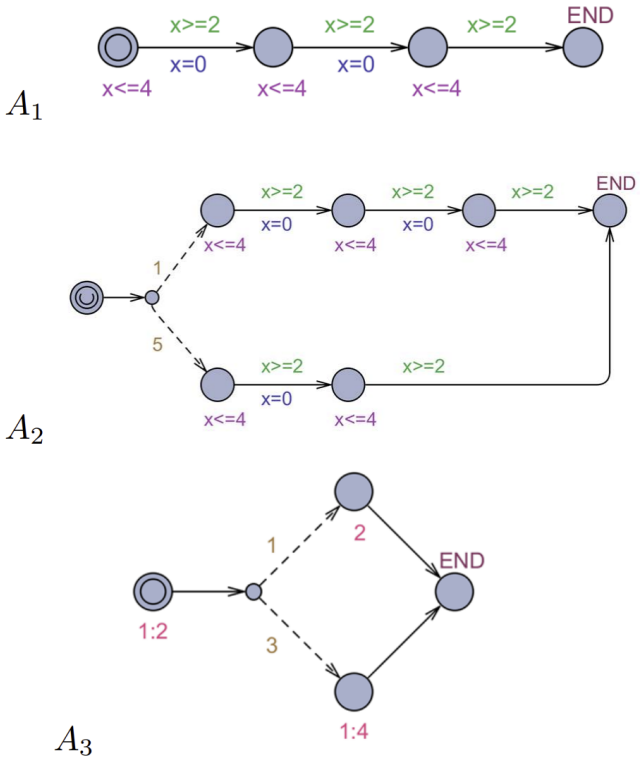

Figure 5.3: Three examples of stochastic timed automata [4].

In subfigure A3 in figure 5.3 above, the locations have also been assigned a weight. This weight is the rate of exponential, which specifies the rate of exponential probability distribution. The weight expression, as seen in subfigure A3 in figure 5.3, can either be an integer or a ratio, such as $1:4$, where the weight is then determined as $1/4$. The meaning of the rate of exponential is to apply a probability to a location, which conveys how likely the model is to leave the location, given some time t. The equation looks like so:

$$1 - e^{-\lambda t}$$

Where e is Euler's number, λ is the rate of exponential and t is a time. That is, the lower the user specified rate, the higher the chance of staying longer in a location, before leaving it. Note that only when a location does not possess an invariant, will the delays be chosen based on the exponential distributions derived from the user-supplied rates, and instead it will be chosen uniformly.

The figure 5.4 below is from the UPPAAL SMC tutorial [4]. It showcases the

distribution of the reachability time to the END-location and how it varies depending on the type of implementation. When looking at subfigure A1 in figure 5.4, the overall reachability time follows a normal distribution, since the delays are chosen uniformly inside the [2,4] interval, between the 3 transitions in subfigure A1 in figure 5.3.

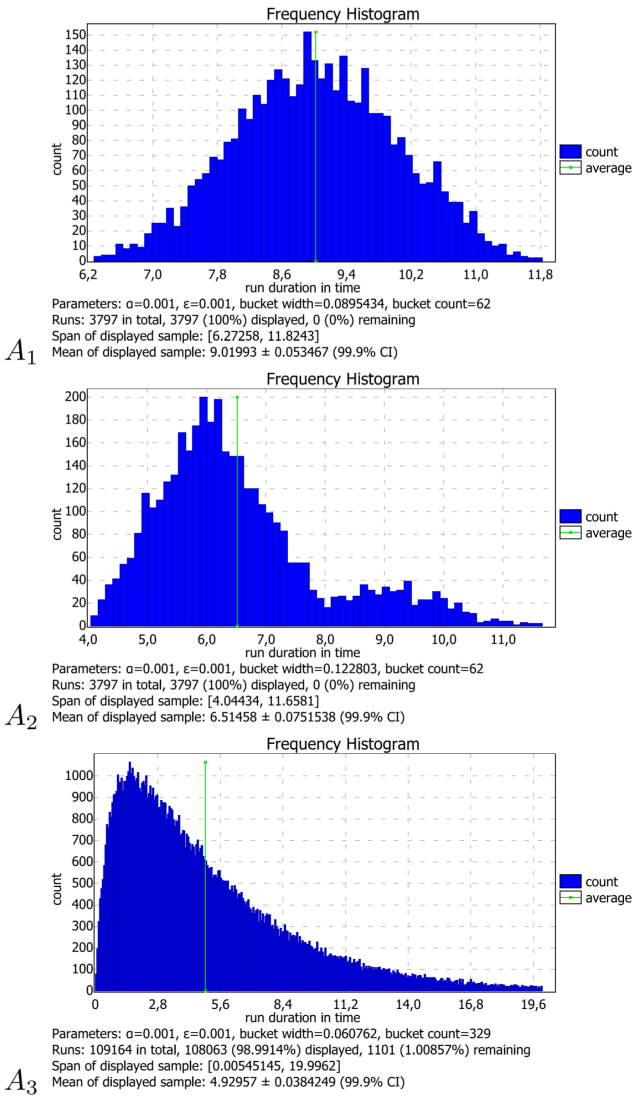

Figure 5.4: Distribution of the overall reachability time to the END-location and how it varies depending on the type of implementation [4].

This changes when looking at subfigure A2 in the figure 5.4. This is because the model introduces a discrete probabilistic choice. When looking at subfigure A2

in figure 5.3, it can be seen that there is a 5 in 6 chance, that the lower edge will be chosen, whose path has a decreased reachability time, compared to the upper path. This is reflected in subfigure A2 in the figure 5.4, where it can be seen that the majority of simulations have a decreased reachability time, due to this discrete probabilistic choice.

As stated earlier, the rate of exponential, increases the likelihood of leaving a location over time. As such, it makes sense that subfigure A3 in the figure 5.4 above, follows this trend, where an exponentially decreasing number of simulations reach the END-location the more time passed.

5.2.1 SMC Query Language

UPPAAL SMC provides new queries related to the stochastic interpretation of timed automata, which can visualize the values of expressions along simulations. This allows the user to explore interesting model properties, visualized in graphically presented statistics in the model-checker.

An example of the syntax for a simulate query can look as follows [17]:

$$simulate\ [\leq\ bound; N]\ E_1, \ldots, E_k$$

- Here N is a natural number, that indicates how many simulations will be done,
- *bound* is the amount of time steps for each simulation, and lastly
- $E1, \ldots, Ek$ is the state-based expression that is visualized.

When using the SMC extension, different statistical algorithms can be used to find different probabilities and testing hypotheses. Firstly, there is probability estimation, which is used to find the number of runs needed to be inside a confidence interval. Secondly, there is hypothesis testing, where it checks whether the expression is greater or equal to some probability. Lastly, there is probability comparison, which checks whether a probability for one expression is greater than another expression.

Chapter 6

Implementation

This chapter will present the implemented DTLS 1.3 UPPAAL model, as well as four different implementation cases that will be simulated, analyzed and compared.

In order to model the protocol, the group followed the DTLS 1.3 documentation. The documentation works as a template, for how DTLS can be implemented, and as such there is an abundance of possible configurations. Therefore, a simple implementation following the DTLS guidelines was chosen for the project.

Within the project, some restrictions were made, to scope the project and simplify the problems at hand. In the DTLS connection and the model in UPPAAL, the following restrictions are made. The IP is consistent, meaning if a connection is removed from the NAT table, due to a possible handover, when re-connected the client resumes with the same IP and port. Another restriction made is that cookies in a handshake process are always included when session resumption is used after a connection is established. The reasoning for always including cookies is described in 2.3.2. Lastly, the project abstracts away from fluctuations in energy consumption caused by the transmission of application data, as this does not affect the problem at hand. Sending and receiving application data should be equal among all cases, and is therefore abstracted away from.

6.1 Handshake Model

The following section will describe the UPPAAL model of the DTLS 1.3 handshake protocol. The section divides the handshake model into three parts in total. The model presented has been simplified in a way that strips away redundant logic that is unnecessary for the understanding of the DTLS 1.3 handshake. A full view of both the server and client simplified handshakes can be seen in appendix A.

The figures presented in this section are parts of the server UPPAAL template, whereas the client template can be found in appendix A. The reason for only presenting the server side here, is that the client and server templates are very similar, and the points of deviation are more interesting from the server's perspective.

6.1.1 Hello & Cookie

The initial part of the DTLS handshake can be seen in figure 6.1. The general channel naming convention in the model is such that a channel is prefixed with either an "S" or a "C", indicating whether it is the server or the client performing an action in the protocol. As an example, `Chello?` means that the server is waiting to receive a client hello from the client, i.e. synchronizing on the `Chello` channel.

As stated in section 2.3.2, cookies are always enabled when performing a new handshake. This is modelled by going to the `HelloRetryRequest` location, when the server receives the first client hello, from the client. When the server arrives in this location, the server will then send back a cookie, and update the `sentCookies` local variable, to remember that a cookie has been sent. After receiving the cookie, the client can then send back the cookie, by synchronizing on the `Ccookie` channel. In order to receive a cookie, the server must first have sent a cookie, so a guard `sentCookies` on the `Ccookie?` channel is added, to check whether the server has indeed sent a cookie. If true, the server and client can synchronize and move to the next part of the handshake.

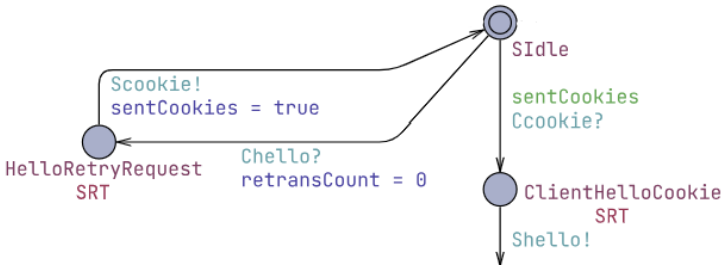

Figure 6.1: The initial part of the DTLS handshake server model.

In total, this initial part of the model captures the three first flights of the DTLS handshake. That is, the ClientHello, HelloRetryRequest+cookie and ClientHello+cookie, which can be seen in figure 2.4 in the DTLS section.

6.1.2 Cryptographic Section

The next part of the DTLS handshake, is the cryptographic part, which can be seen in figure 6.2. It continues from the initial part, after the server has received a cookie from the client. In this section, the cryptographic properties of the handshake are established. That is, whether the handshake is going to perform a certificate-based authentication or make use of pre-shared keys (PSK).

This part of the handshake corresponds to the ServerHello in figure 2.4 in the DTLS section. As such, everything being sent by the server here, actually consists of just a single flight. This has been implemented in the model by inserting committed states after the first `Shello!` action. This implements that no delay action can be performed from this state, and that the next action must be performed from this state. This nicely simulates the fact that all these actions are in reality just one transmission, while still complying with the order of operations stated by the DTLS 1.3 documentation [12].

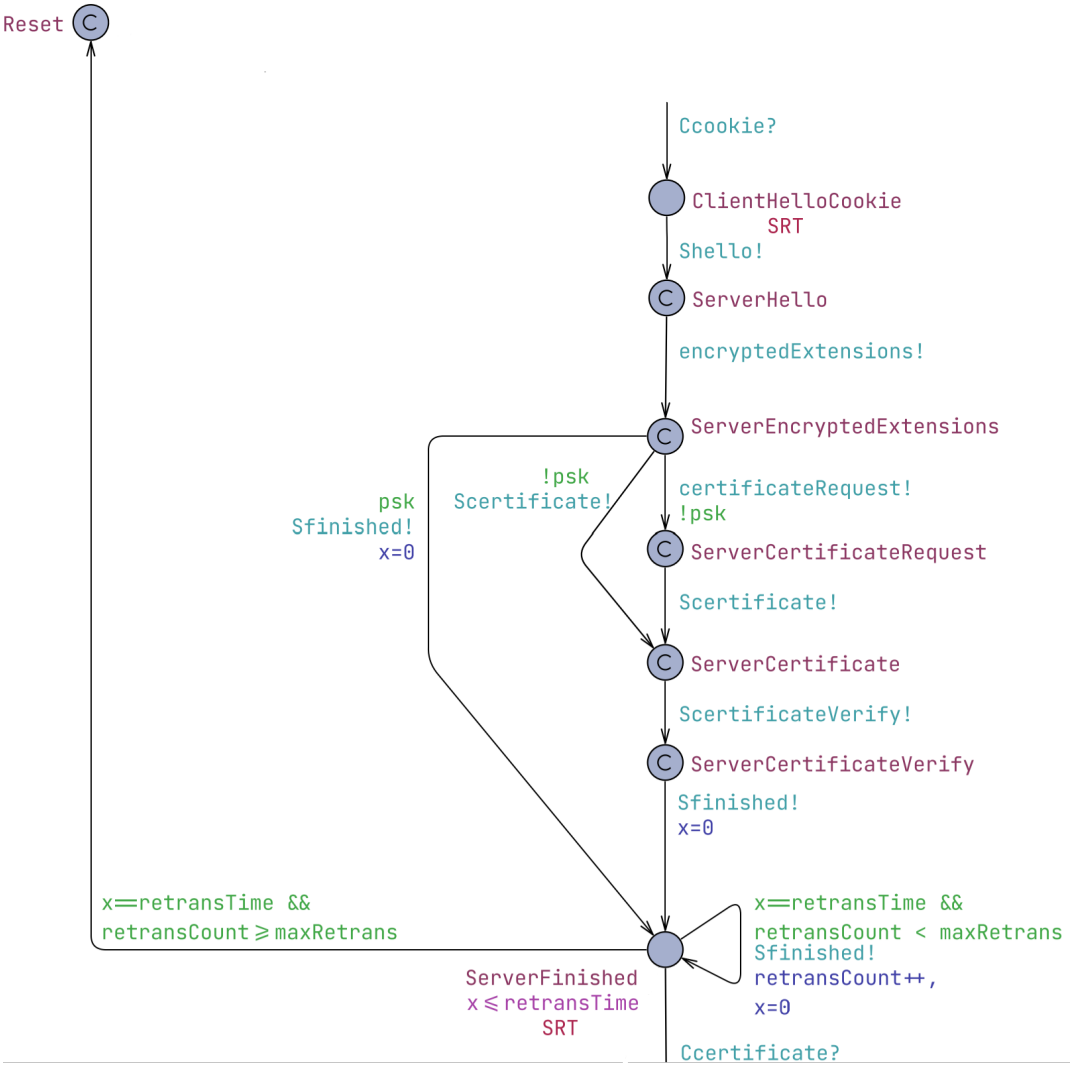

Figure 6.2: The cryptographic part of the DTLS handshake server model.

As stated, the handshake can either perform PSK or certificate-based authentication. This is reflected in the `psk` flag. This flag is set to true, whenever a full handshake has been completed, as can be seen in figure 6.3 in the section below. An assumption made here is that the server will always perform PSK, if such an option is available, but this can vary in practice. If PSK is not available, the server can first choose to make a certificate request with `certificateRequest!`, requesting the client to also do certificate-based authentication. If this is not preferable, the server will send its certificate using `Scertificate!`, along with a certificate verify using `ScertificateVerify!`, and lastly the finished message with `Sfinished!`.

When the server has sent the entire ServerHello flight, it reaches the `ServerFinished` location. In this location, the server waits for a response from the client, as seen in figure 6.3 in the section below. To reflect reality, a retransmission functionality is implemented, since there is a chance that either a transmission from the client never reaches the server or the client could have crashed. This retransmission functionality is implemented both in the client model and the server model, following a flight. However, an exception to this functionality is after the server "HelloRetryRequest" flight, as stated by the DTLS 1.3 documentation [12]. This is why the `SIdle` location in figure 6.1 in the above section, does not have this retransmission functionality. The functionality works by declaring a local clock, which when reaching the `retransTime` variable value, will perform a retransmission and reset the clock, as seen in the self-loop on the `ServerFinished` location. A retransmission is simulated by simply performing the `Sfinished!` action again. If the `retransCount` variable value reaches the `maxRetrans` variable value, the handshake will take an action to the `Reset` location, which simply resets the handshake. This is ensured by the invariant `x <= retransTime` in the `ServerFinished` location.

6.1.3 End of Handshake

The final part of the handshake model seen corresponds to the ClientFinished flight, as well as the server acknowledgement in figure 2.4 in the DTLS section. Figure 6.3 shows that the client must go one of two ways. The client has received a `certificateRequest!` from the server, as described in the previous section, and therefore must adhere to this request, by sending a certificate and a certificate verify, alongside the client finished message. If the client has not received a "CertificateRequest" from the server, the client will send its "ClientFinished".

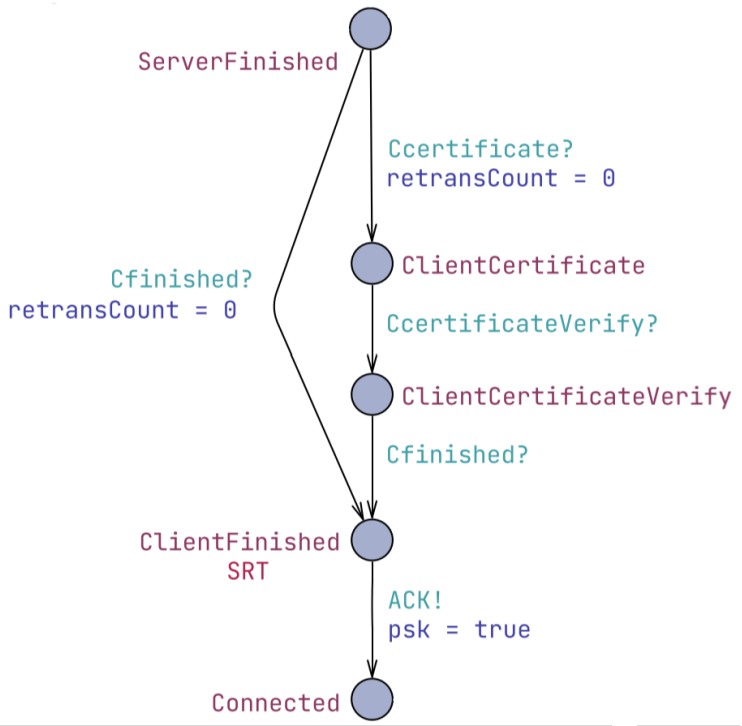

Figure 6.3: The last part of the DTLS handshake server model.

When the server receives the `ClientFinished?`, it will send back an acknowledgment with `ACK!`, set the `psk` flag to true, and move to the `Connected` location, indicating that the handshake has completed and that the parties have established a secure DTLS connection, where from application data can be sent.

6.1.4 Handshake Validation

The handshake process modeled in UPPAAL was validated by model-checking crucial parts of the handshake to validate that the modeled handshake behaves as the DTLS handshake protocol, as described in section 2.3.2.

The queries presented in this section utilizes the "possibly" property and the "invariantly" property, as described in 5.1.1. The "leads-to" and "eventually" properties are not used, since the handshake theoretically always can time out after sending a flight, and as such, it is never a guarantee that a handshake is completed eventually. In the below tables, the different queries are presented, with a

description of the query and whether each query is expected to fail or pass the given query. In all the below tables with queries, the actual result is the same as the expected value stated in each query.

Query	Description	Expect
`E<> (Server.ServerHello and !Server.sentCookies)`	This query checks that there exists a state where the server can proceed in the handshake process without having received a cookie from the client.	Fail
`A[] !Server.sentCookies imply !Server.ServerHello`	This query checks that for all states, the server cannot send a server hello to the client if the server has not yet sent a cookie to the client.	Pass
`A[] Server.ServerHello imply Server.sentCokkies`	This query checks that for all states, if the server has sent a server hello to the client, the server has also sent a cookie to the client	Pass

Table 6.1: Validation of Cookies in Handshake.

Table 6.1 above, shows the collection of queries that validate that the cookie exchange functionality of the DTLS handshake is working correctly and behaves as intended. Essentially, it verifies that no handshake can take place if the server has not received a cookie from the client, as stated in section 2.3.2.

Query	Description	Expect
`E<> (PSK and Server.ServerCertificate)`	This query verifies that there exists a state where the server has sent a certificate to the client and the PSK guard is set to true.	Fail
`A[] psk imply !Server.ServerCertificate`	This query verifies that for all states, if the PSK flag is set to true, then the server process cannot be in a location where it has sent a certificate to the client.	Pass
`A[] Server.ServerCertificate imply !psk`	This query verifies that for all states, if the server has sent a certificate to the client, then the PSK guard is set to false	Pass
`A[] Client.Connected imply psk`	This query verifies that for all states, if the client is in the connected location, then the PSK guard is set to true. Note: This is a programmatic check to validate the setting of PSK flag is correctly modeled.	Pass

Table 6.2: Validation of PSK in Handshake.

Table 6.2 contains the queries which are used to validate that the pre-shared key functionality of the DTLS handshake is correctly modeled and is behaving as intended. As described in section 2.3.2, a handshake can only perform either certificate-based authentication or pre-shared keys. These queries validate that the model can only do one or the other, and that after the first full certificate-based handshake, the PSK flag is set to true. This allows the model to use session resumption for future handshakes.

Query	Description	Expect
`A[] Client.ClientCertificate imply Client.isCertificateRequsted`	This query verifies that for all states, if the client sends a certificate to the server, then the server must have requested it, which is indicated in the isCertificateRequested flag.	Pass
`A[] !Client.isCertificateRequested imply !Client.ClientCertificate`	This query verifies that for all states if the server has not requested a certificate from the client, then the client has not sent a certificate.	Pass
`A[] (Server.ServerCertificateRequest and Client.ServerCertificateRequest) imply (Client.isCertificateRequested)`	This query verifies that for all states, if both the server and client are in ServerCertificateRequest location, then the isCertificateRequested is set to be true. Note: This is a programmatic check to validate the setting of isCertificateRequested flag is correctly modeled.	Pass

Table 6.3: Validation of Client certificate request in Handshake.

In table 6.3, the queries shown is for checking whether the client only sends its certificate when it has been specifically requested to do so by the server.

Query	Description	Expect
`A[] Client.Reset imply Client.retransCount>=Client.maxRetrans`	This query verifies that for all states, if the client is in the Reset location, it could only have happened if the maximum allowed amount of retransmissions has been met.	Pass
`A[] Server.Reset imply Server.retransCount>=Client.maxRetrans`	This query verifies that for all states, if the server is in the Reset location, it could only have happened if the maximum allowed amount of retransmissions has been met.	Pass
`A[] Server.retransCount > 0 imply Server.flightSent`	This query verifies that for all states, if the server is retransmitting, it can only happen in states where the server has sent a handshake flight.	Pass
`A[] Client.retransCount > 0 imply Client.flightSent`	This query verifies that for all states, if the client is retransmitting, it can only happen in states where the client has sent a handshake flight.	Pass

Table 6.4: Validation of retransmission in handshake.

In table 6.4 are the queries conducted to validate that the retransmission functionality is behaving as intended. Here, it should be noted that the "flightSent" variable is a flag that is set to be true after a process has completed the sending of an entire handshake flight.

Query	Description	Expect
`E<> Server.Connected and Client.Connected`	This query verifies that there exists a case where the server and the client are in the connected location at the same time.	Pass
`E<> Server.Connected`	This query verifies that there exists a case where the server is connected.	Pass
`E<> Client.Connected`	This query verifies that there exists a case where the client is connected.	Pass
`A[] !deadlock`	This query verifies that for all possible states, no state is a deadlock.	Pass

Table 6.5: Validation of general queries model.

Lastly, the general queries that validate that the client and server can connect to each other, and that the model does not contain any deadlocks, can be seen in 6.5.

6.2 Router Model

Before delving further into how the client and server are modeled post-handshake, it is necessary to discuss the router model. The router models the client's router, and more importantly, models how the client will not synchronize with the server, if the server has been dropped from the router's NAT table. In figure 6.4 a simplified version of the router model can be seen. In the full router model, there exists an edge for each output channel of the client and for each output channel of the server. In the simplified model, the `allClientActions` edge is a compression of all the client output channel edges. The same is the case for the `allServerActions`. Refer to appendix C.1 to see the full model.

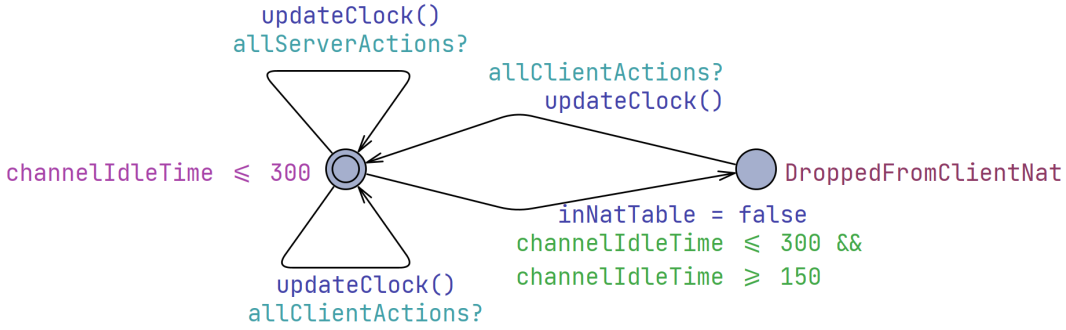

Figure 6.4: The simplified router model.

The router only has two locations, one being its initial location and the other the `DroppedFromClientNat` location. The initial location will synchronize with all channels, updating and resetting the clock each time. This simulates a router receiving and forwarding a message, thereby resetting the NAT table's timeout timer of the connection between the given server and client. The only edge, not synchronizing on any channels, is the edge leading into `DroppedFromClientNat`. This transition is taken when the model is forced out of the initial state because of the invariant `channelIdleTime <= 300` in combination with the guard `channelIdleTime <= 300 && channelIdleTime >= 150`. This simulates a router that drops the server/client connection entry in the NAT table between 150 and 300 time units (seconds). This guard can be changed to simulate other routers, with `channelIdleTime == 300`

used for most of the experiments, to simulate a 5-minute timeout.

When this transition is taken, the variable `inNatTable` is set to false, simulating that the client will not be able to receive any messages sent by the server. At this time, the router will only synchronize with the client's output channels, where a synchronization will transition the model back into the initial location.

This model can thus simulate a client's router, by only allowing the client to synchronize with the server if `inNatTable` is set to true.

6.3 Communication Model

The communication model will be described from case 2's perspective, since this is the base of the problem solution, and other cases are generally used to compare, or further develop on case 2.

After a handshake is completed, communication between the server and the client can occur. For readability, the server will transition to the `Connected` location, where it will synchronize with the ConnectionServer model. It is important to specify that this does not represent another server nor another server IP on the same machine, but is only for readability.

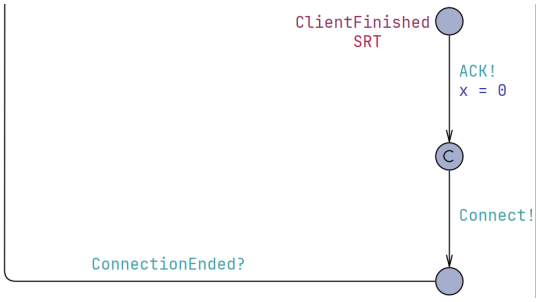

Figure 6.5: The part of the server model, synchronizing with the ConnectionServer model 6.9.

Immediately after the handshake is finished, and the server sends its ACK, the server will take the edge synchronizing with the ConnectionServer on `Connect` , as seen in figure 6.5. This allows the ConnectionServer model to model the open connection between the server and the client. The ConnectionServer is now at the `Connected` location, and will do all major actions from here. It is also from this location, that the ConnectionServer will synchronize with the server on

`ConnectionEnded`, if the client attempts to close the connection. This will allow the server model to go back to its initial state, ready for a new connection. See `ConnectionEnded` in figure 6.6 and 6.5.

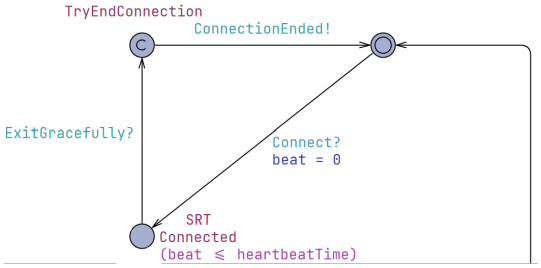

Figure 6.6: The connect and disconnect part of the ConnectionServer model.

When the server and client are connected, the heartbeat protocol is initiated. When seen from the `Connected` location of the ConnectionServer in figure 6.9, two transitions can be taken. One modeling the heartbeat and one modeling a user's request for data.

6.3.1 Heartbeat Protocol

Focusing on the heartbeat seen in figure 6.7, the invariant `beat <= heartbeatTime` forces the model out of the `Connected` location after some time. The edge has a guard `beat==heartbeatTime` only allowing the model to take the transition when the expression is true.

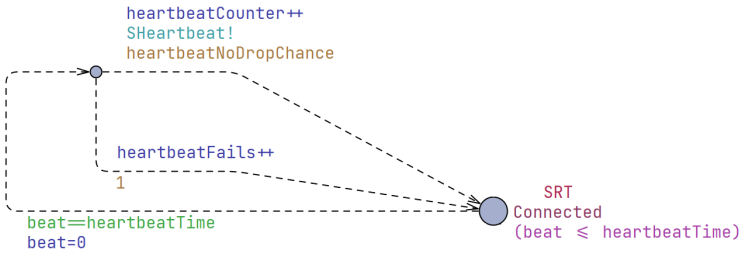

Figure 6.7: The heartbeat part of the ConnectionServer model.

In other words, this simulates a heartbeat once every `heartbeatTime`, which is a given variable, and the model will then based on `heartbeatNoDropChance` either synchronize with the client on `SHeartbeat`, or it will simulate a dropped package and not synchronize with the client.

At this time, it makes sense to talk about the client. The client can be seen in figure 6.8, and it can be hard to make sense of what is going on, but in short, it only simulates 3 different actions. Starting with the edge that synchronizes on `SHeartbeat`. This is the same channel as was talked about on the server side in figure 6.7. When the server sends the client a heartbeat, it will attempt to synchronize on `SHeartbeat`. The client may only synchronize with the server if the variable `inNatTable` is set to true. See section 6.2 for a detailed explanation of the router, and how the NAT table is modeled. If the client is able to synchronize with the server on `SHeartbeat`, it will reset its clock and go back into the `Connected` location.

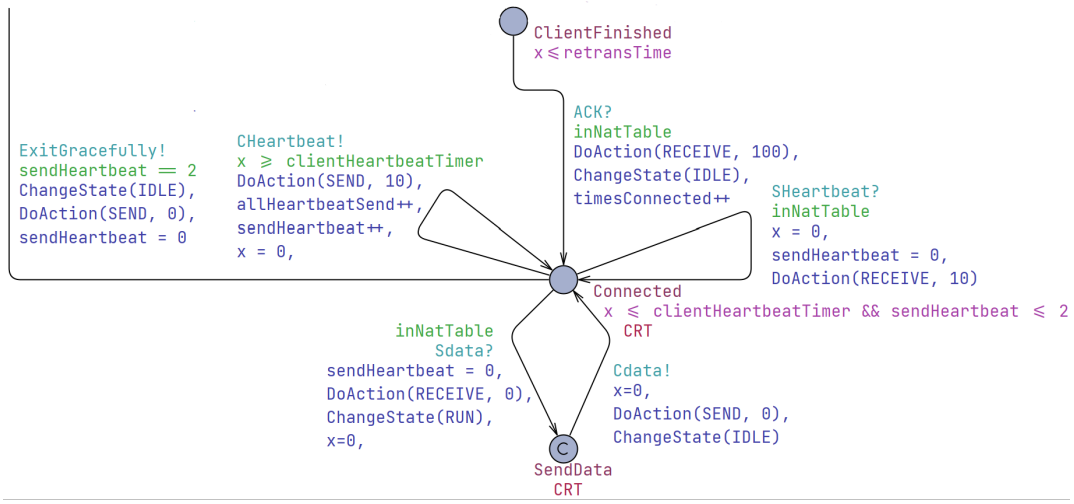

Figure 6.8: The client models connected part.

The `Connected` location has the invariant `x <= clientHeartbeatTimer && sendHeartbeat <= 2`. Focusing on the first part of the logical statement, if the clock `x` were to exceed the value of the variable `clientHeartbeatTimer`, it must have left the location. The only transition available when $x == clientHeartbeatTimer$, is the one synchronizing on `CHeartbeat`. This transition will simulate the client not receiving any heartbeats for some time, and then sending one to the server itself. This will revive the connection entry in the NAT table (see section 6.2), and allow the client to again receive heartbeats and requests from the server.

Another note on this edge is that it will increment the `sendHeartbeat` variable. This leads us to the other part of the invariant. The client cannot be in `Connected` location if `sendHeartbeat` were to exceed 2, thereby forcing the client to take the edge synchronizing on `ExitGracefully` before `sendHeartbeat` increments to

3. This will terminate the connection with the server and return the client to the initial location.

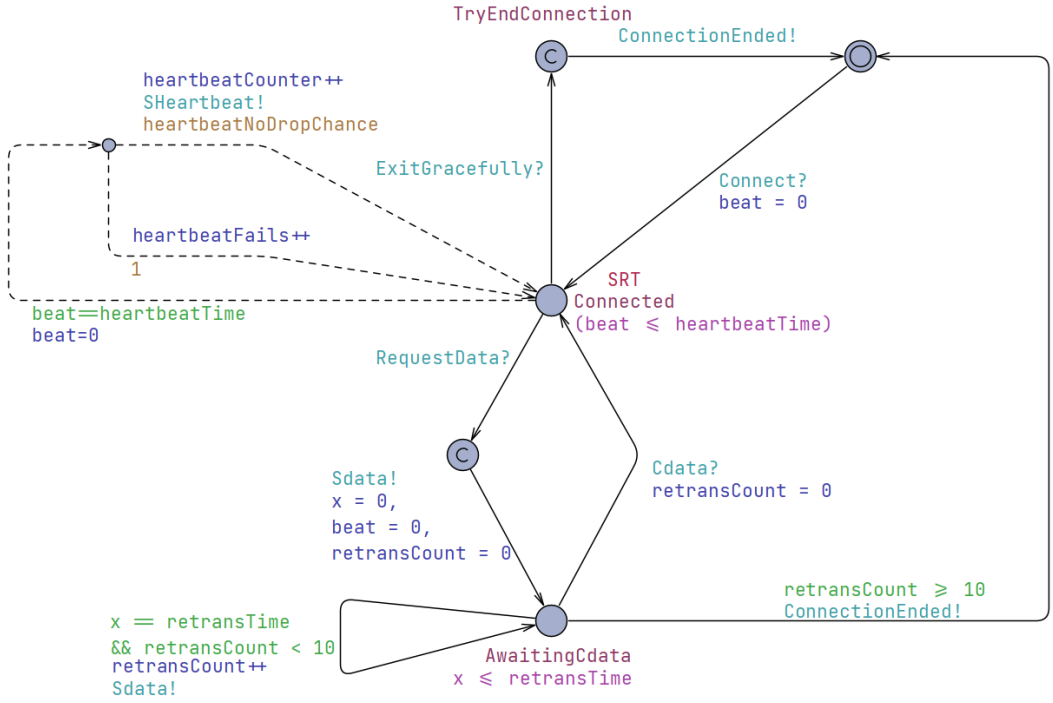

Figure 6.9: The ConnectionServer model. This model shows the behavior of the server, when the handshake is done and a connection has been established.

This fully covers the heartbeat protocol implemented in the models. The server will send a heartbeat to the client every given time interval. The client will receive the heartbeat and reset it's clock, going back to light sleep until the next heartbeat is expected. In the case the server drops the heartbeat package and the client never receives it, it will not reset its timer, and the client will stay waiting for the heartbeat. If the server has been dropped from the NAT table (see section 6.2), after dropping the heartbeat package, the next heartbeat will also not go through. In this case after waiting for some time, the client will send a heartbeat to the server, in an attempt to revive the connection, and reenter the server into the NAT table of the router. If this succeeds, the client will soon after receive a heartbeat from the server, and will go back to light sleep until the next heartbeat is expected.

If the heartbeat sent by the client does not resolve in a received heartbeat from the server soon after, the client will send another heartbeat. The client will send at max two heartbeats without receiving one back, before terminating the connection, attempting to gracefully exit.

6.3.2 User Request

Returning to the ConnectionServer, beside from the heartbeat protocol, user requests for data is also implemented in the model. In figure 6.10, the request part of the ConnectionServer model can be seen. A user can request data from the client through the ConnectionServer. The user model is very small, and is located in appendix C.4. The ConnectionServer will synchronize with the user on `RequestData`, and will then immediately attempt to synchronize with the client on `Sdata`. The ConnectionServer will enter the `AwaitingCdata` location, and will periodically retry to synchronize with the client on `Sdata`, simulating requesting the client for data.

The ConnectionServer will only attempt to request data from the client a set amount of times, and if this is exceeded, it will instead end the connection. If the request is answered, the ConnectionServer will re-enter the `Connected` location by synchronizing on `Cdata`.

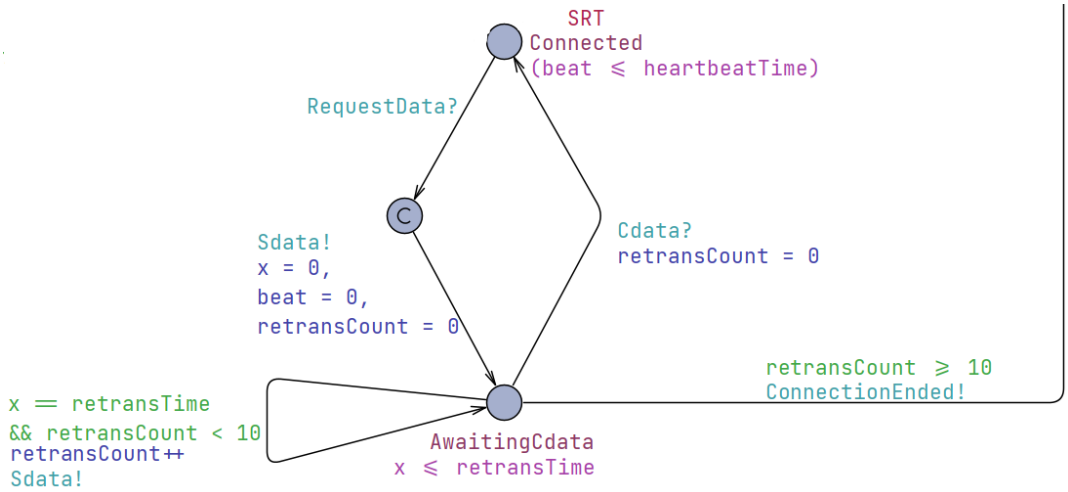

Figure 6.10: User requests modeled in ConnectionServer.

On the client side, as seen in figure 6.11, things look similar. The client will synchronize on `Sdata`, if `inNatTable` is set to true, and will then immediately simulate sending data back to the server, by synchronizing on `Cdata`.

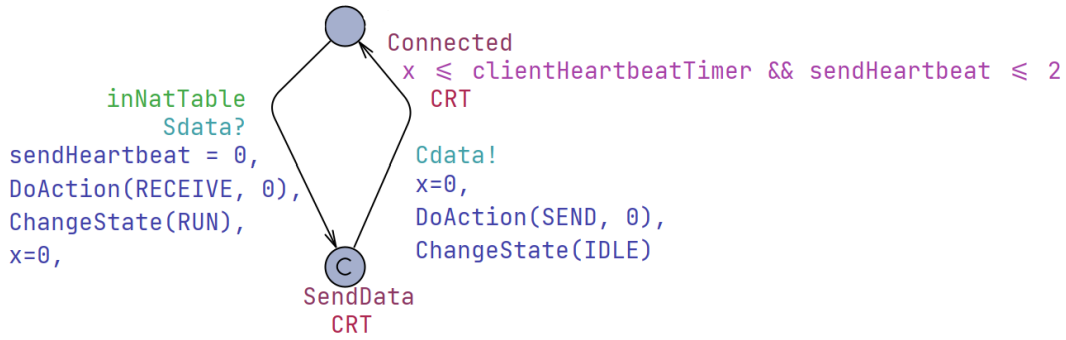

Figure 6.11: User requests modeled in client.

The only reason data may not be sent to the ConnectionServer when requested, is if the connection has timed out and the server has been dropped from the routers NAT table. In this case, the server will resend the data request by trying to synchronize on `Sdata`, until it has reached a set amount of retries, or the client sends a heartbeat that reenters the server in the NAT table, allowing the data request to go through, resulting in a data response from the client. Depending on the value of the variables, the server should never exit this way if the client is still connected. The server will then only exit here, if the client has terminated the connection without the server knowing. The client will then not be in a location where it is able to synchronize on `Sdata`.

Later, section 6.5.7 will go into detail about how often the model is in a state where the client has disconnected, but the server still believes the connection is up.

6.3.3 Communication Validation

Validating the communication part of the model is a rather interesting case. The communication part does not operate under specific rules that cannot be deduced from the model itself. Expected behavior vary with different variable values, and are very few in between. This part of the model can better be measured in case of performance, with questions like "How well" or "How often". To see more on the performance of the model, refer to the later section 6.5.7, where questions on connectivity and client/server synchronization are answered.

6.4 Power Consumption Modeling

To model the power consumption, two new things needed modeling. Which working state the Client is in (RUN, IDLE or SLEEP), and the cost of sending and receiving messages of different sizes. To start, let's look at the working state modeling.

6.4.1 Working State

An array containing all the estimated power consumption for different modes, is declared in the global declarations of the UPPAAL model. Unfortunately, UPPAAL does not have support for enums yet, so indexing for each mode is also separately declared.

Also, there is kept track of which working state the client is in, and the total power that has been consumed for the working state, actions and sum of the two totals. This can be seen in listing 6.1. The values for the power consumption of the three states seen in line 5, are based on the values from table 2.3 for the three states with `SLEEP` being deep sleep, `IDLE` being light sleep, and `RUN` being working. These numbers were scaled up by a factor of 100 simply so the numbers were not as small.

To use these working states to calculate power consumption, a new template was created in UPPAAL called `PowerMeter` (shown in figure 6.12). Every 1 time unit it calculates the `totalPowerConsumption` by calling the function as shown in line 12-15 of listing 6.1 on the update of the edge.

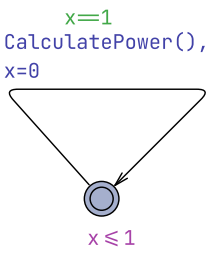

Figure 6.12: `PowerMeter` template used to calculate the power consumption every 1 time unit.

```
1   //States
2   const int SLEEP = 0;
3   const int IDLE = 1;
4   const int RUN = 2;
5   double statePower[3] = {0.001, 1, 100};
6
7   int currentState = IDLE;
8   double totalStatePowerConsumption = 0.0;
9   double totalActionPowerConsumption = 0.0;
10  double totalPowerConsumption = 0.0;
11
12  void CalculatePower(){
13      totalStatePowerConsumption = totalStatePowerConsumption +
    statePower[currentState;
14      totalPowerConsumption = totalStatePowerConsumption +
    totalActionPowerConsumption;
15  }
```

Listing 6.1: The code for working states.

Finally, the Client needs to be updated to change states at differing points. To do this, it simply calls the function *ChangeState* with the state it wishes to change to. An example of this can be seen in figure 6.13.

6.4.2 Actions

Here there is a similar setup as before, with two variables for the different actions and an array to index into for the power consumption of each value. For the power consumption values for sending and receiving are calculated from the following tables 2.1 and 2.2 from the problem analysis. For sending and receiving, the value was calculated as such to be per byte and multiplying by the same factor 100 as for the working states:

$$SendCost = \left(\frac{0.05 * 10^{-3}}{200} + 10^{-3}\right) * 100 = 2.5 * 10^{-5}$$

$$ReceiveCost = \left(\frac{0.2 * 10^{-3}}{200} + 10^{-3}\right) * 100 = 10^{-4}$$

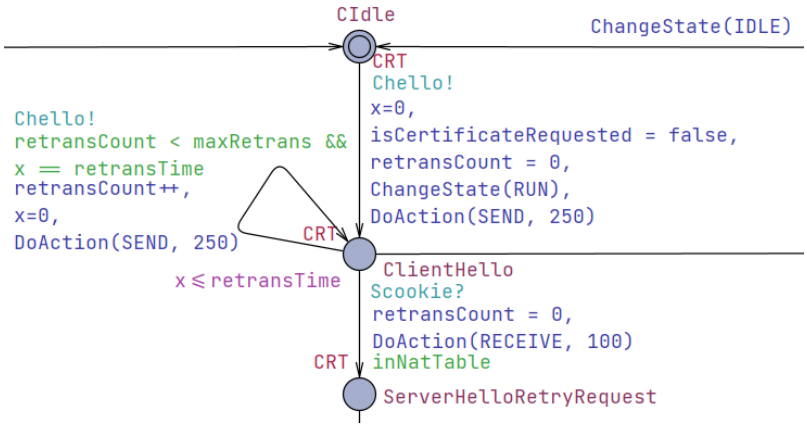

Figure 6.13: Client changing states and doing actions with calls to *ChangeState* and *DoAction*.

```
1  //Actions
2  const int SEND = 0;
3  const int RECEIVE = 1;
4  double actionPower[2] = {0.000525, 0.0006};
5
6  void DoAction(int action, int bytes){
7      totalActionPowerConsumption = totalActionPowerConsumption +
   actionPower[action]*bytes;
8  }
```

Listing 6.2: The code for actions.

To calculate the cost of an action, the function *DoAction(**int** action, **int** bytes)* is called. The function is shown in listing 6.2. Then on every edge corresponding to sending or receiving data *DoAction* is added as an update and the bytes for each action are based on the values from the tables 2.5 and 2.6. Examples of it on the model can be seen in figure 6.13.

6.4.3 Issues with Units

After having implemented the power consumption for both working states and actions, a problem had emerged. Having the values for the working states power consumption within the model would heavily skew the power consumption of the

system towards making actions have no real impact on the system. This is due to the fact that having the working states power in watts (because of the source it is from) and the `PowerMeter` that is executing every one time unit in UPPAAL now means that a UPPAAL time unit is a second. Because of this the DTLS handshake is now not at all accurate as it can take many time units to complete, but in actuality a DTLS handshake does not take more than a second on a bad day. This would lead to power consumption being completely dominated by the amount of time spent in different states.

Having to find and implement accurate timing for the entire UPPAAL model would prove too time-consuming for the group at this moment. Therefore in the following section 6.5, only the `totalActionPowerConsumption` is looked as for the results.

6.5 UPPAAL Cases

After implementing a model of DTLS in UPPAAL, the group wanted to see the power-based benefits of using improvements like heartbeat and pre-shared keys. To do this, the group created 4 cases of varying levels of complexity and permutations. This section will go through each case one by one to explain what was done, and why each case was interesting to try. Lastly, the section will compare relevant results to see improvements when it comes to power consumption. Some elements are relevant to each case, and as such these will be listed below:

- Each case is simulated over a time span equivalent to a day
- Each case assumes a NAT timeout of 300 seconds (the default for most systems)
- The client (IoT device) will establish a connection if not already established, and update the server at every variable interval (30 minutes, 1 hour and 2 hours)
- Application data sent between server and client is lossless, and only heartbeat packets has a rate of loss

6.5.1 Base Case

The first of the four cases is very simple, as this is a simple base case without any kind of optimizations. This case will receive a data request from the server, before proceeding to establish a DTLS connection. Every time they establish this communication, it is a full hand-shake. Whenever the client is finished sending data, it goes into a LIGHT SLEEP state, listening for the next request. This case has three permutations, one for each interval.

This case assumes some effort has been put in, to allow the server to reach the client on-demand. Either by setup of some form of hub or port-forwarding on the router is needed for a real implementation of this case.

Case	Interval	PSK	Heartbeat	Heartbeat package loss	Heartbeat frequency	Average power use
Base case	1800	No	No	N/A	N/A	97.321
Base case	3600	No	No	N/A	N/A	47.689
Base case	7200	No	No	N/A	N/A	22.806

Table 6.6: The variables set for case 1 and the resulting average power of running 500 runs.

6.5.2 Case 2

This case builds upon the base case by adding a heartbeat. In this scenario, the server tries to keep the connection to the client open for the entire span of the day. These heartbeat packets are generally cheaper than reestablishing a new connection every time, especially in the case of 30 minutes or even less between connections. It is worth noting that the workload is mainly put on the server-side, in an effort to limit power consumption of the client, see chapter 4. Also, this will most likely add a larger workload to the server, than is removed from the client.

The open channel between client and server also allows the server to request data at any point, without relying on other methods, as mentioned in the previous case (Base case). This is a large case permutation-wise, as it has three intervals, multiplied by three heartbeat package loss rates, multiplied by three heartbeat frequencies. As such, this case ends with 27 permutations.

To avoid too many variables in the results, a graph showcasing the effectiveness of heartbeat rate and package loss was created.

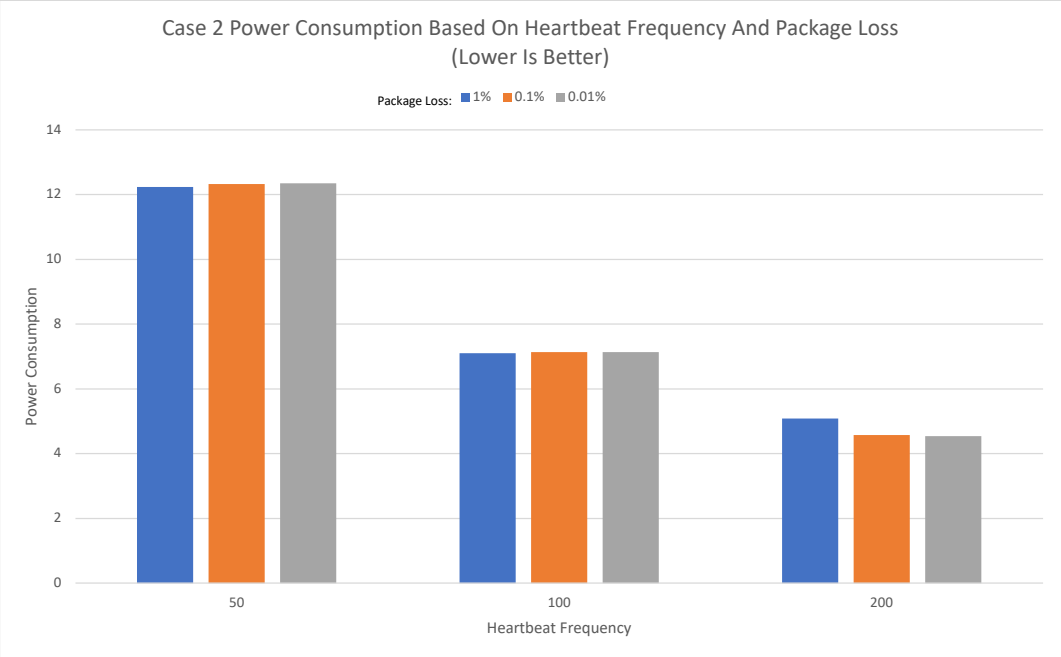

Figure 6.14: A graph showcasing the tie between power and the variables heartbeat frequency and package loss.

Within figure 6.14 one can clearly see large improvements at the higher heartbeat frequencies. Another noteworthy aspect of this figure is the fact that the 1% package loss rate does result in power consumption increase when looking at the 200-second heartbeat frequency. This makes sense, as at this frequency, if a heartbeat from the server had already been dropped, the server would be dropped from the client's NAT table before the next heartbeat is sent. These results show that the higher heartbeat frequencies are not worth it when it comes to client power consumption, and as such, only the results of 200 second heartbeat frequencies are compared to the other cases,

Case	Interval	PSK	Heartbeat	Heartbeat package loss	Heartbeat frequency	Average power use
Case 2	1800	No	Yes	[1%,0.1%,0.01%]	200	4.815
Case 2	3600	No	Yes	[1%,0.1%,0.01%]	200	4.705
Case 2	7200	No	Yes	[1%,0.1%,0.01%]	200	4.673

Table 6.7: The variables set for case 2 and the resulting average power of running 500 runs (this average takes data from all package loss rates).

6.5.3 Case 3

Case 3 deviates from the heartbeat functionality of case 2, and simply adds pre-shared keys and session resumption on top of the base case. This should in theory improve the power consumption, as only the first handshake of the simulation is a full one, while all subsequent connections use session resumption. Similarly to the base case, this case also only has three permutations stemming from the three communication intervals.

Case	Interval	PSK	Heartbeat	Heartbeat package loss	Heartbeat frequency	Average power use
Case 3	1800	Yes	No	N/A	N/A	32.358
Case 3	3600	Yes	No	N/A	N/A	16.244
Case 3	7200	Yes	No	N/A	N/A	8.783

Table 6.8: The variables set for case 3 and the resulting average power of running 500 runs.

6.5.4 Case 4

This case is the culmination of the two previous cases, and could realistically be more accurate to a real life scenario. In this case, the assumption that a user is more active during the day than at night is used to carefully select periods for heartbeat and periods with session resumption. Heartbeats will be used to maintain the connection through a 12-hour period, with handshakes being used for the next 12 hours, when a user is less likely to request data.

Case 4 has the same amount of permutations as case 2, and as such it is also a large case at 27 permutations. These 27 permutations are limited down to 3 by averaging over each interval.

Case	Interval	PSK	Heartbeat	Heartbeat package loss	Heartbeat frequency	Average power use
Case 4	1800	No	Yes	[1%,0.1%,0.01%]	[50,100,200]	18.592
Case 4	3600	No	Yes	[1%,0.1%,0.01%]	[50,100,200]	11.677
Case 4	7200	No	Yes	[1%,0.1%,0.01%]	[50,100,200]	8.339

Table 6.9: The variables set for case 4 and the resulting average power of running 500 runs (this average takes data from all package loss rates and all heartbeat frequencies).

6.5.5 Results

Figure 6.15 showcases the average power consumption of each case for the tested time intervals.

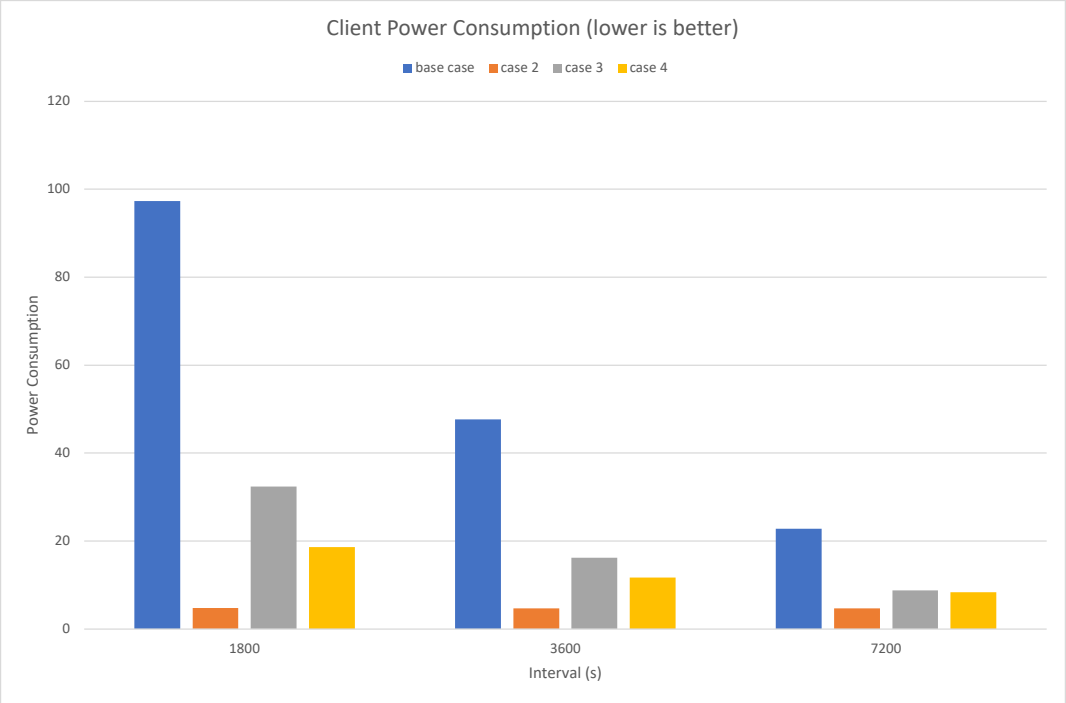

Figure 6.15: Graph showcasing power consumption of each case at the best average result for each interval.

In the graph, it is clear to see that the base case has an extremely high power consumption, to a point where it is at least double the second-highest value, with the 30-minute interval getting close to triple the second highest. This is expected, as a handshake is quite expensive in power, and doing it 48 times every day is very inefficient. The results for the 1 and 2 hour intervals also behave as expected, as 1 hour has roughly half the power consumption of 30 minutes, and 2 hours has roughly half the power consumption of 1 hour.

Case 2 has the best results at every interval, and even at each interval it is roughly the same. This makes sense, since case 2 does not really care for the interval in which the data is sent. Case 2 will always have the connection open, and no extra action, outside sending the data, has to be taken when a request is received. Sending a heartbeat every 200 seconds can be a problem, however, as only a single heartbeat will be sent within each NAT timeout range. As such,

the client has to send something back to reopen the connection if even a single heartbeat drops.

Case 3 has rather large improvements on each of the base case intervals. At each interval, case 3 has roughly a third of the power consumption of its base case counterpart. However, this case is worse than both case 2 and 4 at each interval. This can again be due to short intervals, as it was expected to potentially outdo these cases with lengthier intervals.

Case 4 manages to get relatively close to case 2's performance at the 2-hour interval, but the power consumption is still roughly double. While this case is not the best for client power consumption, it might generally be the most efficient setup if the entire efficiency was in consideration.

Looking at the results, there is a clear winner if the sole focus is to optimize client power consumption, as case 2 is always beneficial compared to the other cases. However, it is worth exploring whether case 4 will perform better than case 2 when given more optimal intervals, with few request during simulated nighttime. It is also interesting to see how low the interval should become before case 3 will outdo either case 2 or case 4.

6.5.6 General Modifications

The results of these tests, sparked a few questions that would be valuable to investigate.

Firstly, the group found an interesting modification for case 4. Case 4 has the same interval of communication in the "off-hours" as cases 1 and 3. This is not a realistic rate of communication, and generally the group would expect 1, maybe 2 communications in this time period. To test this, a so-called "case 5" was created. Within this case, there is 9 requests in the day hours, but only a single request at night. This was extremely beneficial for the power consumption, and managed to outperform case 4 in every test.

Case	PSK	Heartbeat	Heartbeat package loss	Heartbeat frequency	Average power use
Case 5	Yes	Yes	[1%,0.1%,0.01%]	50	7.641
Case 5	Yes	Yes	[1%,0.1%,0.01%]	100	5.116
Case 5	Yes	Yes	[1%,0.1%,0.01%]	200	3.865

Table 6.10: Case 5 results.

It was also worth to look into when case 3 would outdo case 2 or case 4, with knowledge that case 3 benefits greatly from a low amount of data request. Best scenario for case 3 would be no request at all, with case 3 then using ≈ 0 power, from the sources taken into account in these experiments. In this scenario, case 3 will always outperform case 2 and case 4. More realistically case 3 will receive some amount of data request, which gave the results seen in 6.11. Looking at the data, there has to be sent between 2 and 4 data requests in average through a day, for case 4 to outperform case 3.

Case	Amount of requests	Heartbeat	Average power use
Case 3	4	No	4.641
Case 3	2	No	2.719
Case 3	1	No	2.034

Table 6.11: Case 3 on infrequent data requests.

6.5.7 Further Testing

Further tests on the performance of the model were made, with the focus not surrounding power consumption. Power consumption by itself does not tell much about how the system operates, or which action caused the numbers to go up, but is great as a combined measure of evaluation. Information about amount of connections made through a day, number of heartbeats sent by the client, and other areas like how often the client/server is in idle while the counterpart believes a connection is still established. These tests are made exclusively on case 2 and case 4, since these tests surround heartbeat.

All queries simulated a full day of 86400 seconds, with estimation running for 2000 runs.

First, the query below finds the max value of `allHeartbeatSent` of the client, which is a variable that keeps track of the amount of heartbeats sent by the client through all connections made through the day.

```
E[<=86400; 2000] (max: Client0.allHeartbeatSent)
```

Focusing on case 2, two variables play a factor in the amount of heartbeats sent by the client. Interval will on average not change the outcome of the query, but heartbeat package loss as well as heartbeat frequency will.

Starting with frequency; all combinations of frequencies below 200 and heartbeat package loss, will result in the number of heartbeats sent by the client being < 0.05. In the cases with frequency set to 50 the amount of heartbeats sent by the client approximated 0. These numbers arise from the fact that a NAT timeout occur after 300 seconds, meaning that a frequency below 150 will allow for two heartbeats to be sent before timeout. In the case with a frequency of 100, two subsequent heartbeats has to be dropped, before the server is removed from the routers NAT table. With a frequency of 50 this is even less plausible, since 5 subsequent packages must be dropped, and with the highest drop rate being 1%, there is a 0.00000001% chance that the client is ever required to send a heartbeat, which over 2000 runs ≈ 0. Using a probability query similar result can be seen.

```
Pr[<=86400] (<> Router.DroppedFromClientNat)
```

All permutations with frequencies of 50 approximate zero. Frequencies of 100 will in the worst case (1% package loss) have a probability of 5% to drop the server from the NAT table, with lower package loss percents also approximating zero.

Focusing on the frequencies above 150, where one loss of a heartbeat package will result in the server being removed from the NAT table, the following results were acquired.

Heartbeat package loss	Heartbeat frequency	Number of heartbeats sent by client	Probability of being dropped by NAT table
1%	200	4.154 ± 0.1	$\approx 95\%$
0.1%	200	0.4225 ± 0.3	$\approx 35\%$
0.01%	200	0.0425 ± 0.009	$\approx 3\%$

Table 6.12: Heartbeats sent by client and probability of being dropped by the NAT table for all permutations of package loss with a frequency equal to 200.

Here a mathematical pattern is quite apparent, with a change in package loss and an equal change in the number of heartbeats sent can be seen. Also, the probability of ever being dropped from the NAT table drastically decreases with package loss rate. Another pattern can be seen from the number of connections made through the day.

```
E [<=86400; 2000] (max: Client0.timesConnected)
```

All current permutations of case 2 approximate zero, with only a frequency of 200 and drop rate of 1% having a different result of ≈ 1.5. From this it can be seen that through a day, on average 1.5 handshakes are made, and the rest of the power consumption stem from heartbeats.

Following previous tests, it is interesting to see if or how often the models are out of sync, in terms of one being connected while the other is disconnected. It can be seen from the table 6.13, that runs over 86400 time units, it is almost a guarantee that the `ConnectionServer` will be in idle mode, if the client is idle. It can also be seen that there is $\approx 10\%$ chance that the `client` will be considered connected while the `server` and `ConnectionServer` is disconnected. This shows that the server may decide to drop the connection, without informing the client, but not the other way around.

The last two queries in table 6.13 explores the desynchronization between data request and the sending of data. Again a similar pattern can be observed, where the client will with almost certainty never attempt to send data, without it being requested, or after the `ConnectionServer` timed out, and is no longer waiting for the data. However, the other way around, the server will quite often end in a scenario where it has requested data, but the client is unaware, with $\approx 16\%$ for this to happen at least once over a simulated day.

Query	Description	Expect
`Pr [<=86400] (<> Server.SIdle && ConnectionServer.CSIdle && Client0.Connected)`	The query returns the probability of there existing a state, where both the server and the connection server is Idle, while the client is still connected.	≈ 0.1
`Pr [<=86400] (<> Client0.CIdle && ConnectionServer.Connected)`	This query returns the probability that a state exists, where the client is idle, where the connection server is still connected.	< 0.05
`Pr [<=86400] (<> ConnectionServer.AwaitingCdata && Client0.Connected)`	This query will specify the likelihood of the connection server to be awaiting data from the client, without the client being aware.	≈ 0.16
`Pr [<=86400] (<> !ConnectionServer.AwaitingCdata && Client0.SendingData)`	The query will specify the likelihood of a client attempting to send data, when the server did not request it.	< 0.05

Table 6.13: Verification of general queries model.

6.5.8 Varying NAT Timeout

Earlier testing has led to interest in adjusting the logic of the router. Having varying NAT timeouts, similar to a more busy router, could yield some interesting results.

150-300 Timeout

With a timeout between 150 and 300, meaning timeouts cannot happen before time 150 and must at the latest happen at time 300, the permutations of case 2 are looked at once more. Again, interval will not be included, since this has minimal effect on the results of the queries. Combining all queries from the previous section, focusing on a set frequency of 50, some very interesting results can be seen in table 6.14.

At a frequency of 50 the client on average sends an amount of heartbeats approximating zero, but the probability of being dropped by the NAT table is between 46% and 39%. This showcase the limitations and abilities of UPPAAL

Heartbeat package loss	Heartbeat frequency	Number of heartbeats sent by client	Probability of being dropped by NAT table
1%	50	0.0005 ± 0.0009	$\approx 46\%$
0.1%	50	≈ 0	$\approx 40\%$
0.01%	50	≈ 0	$\approx 39\%$

Table 6.14: Heartbeats sent by client and probability of being dropped by the NAT table for all permutations of package loss with a frequency equal to 50 and a NAT timeout of 150-300.

and SMC queries, since in theory there is a rather high change to being dropped from the NAT table, since only two packages in theory has to be dropped during $\frac{86400}{50} = 1728$ subsequent packages sent. This is quite plausible, but in reality these circumstances rarely align, since the server has to be dropped from the NAT table at exactly time 150 and this also has to be when two subsequent packages are both being dropped.

For frequencies of 100 we have quite an increase in heartbeats, but still at a very acceptable degree. In table 6.15 a similar story unfolds, with a heartbeat count of above one, but still very low at ≈ 3 for the worst case. A frequency of 100 falls under the category that averages 1-2 two heartbeats before a timeout.

Heartbeat package loss	Heartbeat frequency	Number of heartbeats sent by client	Probability of being dropped by NAT table
1%	100	3.11 ± 0.08	$\approx 96\%$
0.1%	100	1.05 ± 0.01	$\approx 60\%$
0.01%	100	0.04	$\approx 45\%$

Table 6.15: Heartbeats sent by client and probability of being dropped by the NAT table for all permutations of package loss with a frequency equal to 100 and a NAT timeout of 150-300.

Finally with a frequency of 200, we can see the limitations but also the potential of the model. With a timeout between 150 and 300, on average 1/3 of the time, a heartbeat will not be sent in time leaving the client with a lot of responsibility to keep the connection open.

An interesting point to make is the amount of heartbeats sent by the client, does not match 1/3 of the heartbeats sent by the server over a day. The server with a frequency of 200 needs to send $86400/200 = 432$ heartbeats a day, and will on average not send $432/3 = 144$ in time. This would make one believe that the heartbeats sent by the client should be around that number or above, but in fact the client does not send a heartbeat for all missed heartbeats sent by the server.

The client will send a heartbeat, after not receiving a heartbeat for 700 seconds. Therefore, when the amount of heartbeats dropped or missed increases, the client will send fewer heartbeats than the server has missed, due to waiting 1.5 heartbeat cycles before reestablishing the connection.

HB package loss	HB frequency	#HB sent by client	Probability of being dropped by NAT table	#Times connection was fully lost
1%	200	74.207 ± 0.149	$\approx 100\%$	11.68
0.1%	200	73.56 ± 0.15	$\approx 100\%$	11.62
0.01%	200	73.336 ± 0.15	$\approx 100\%$	11.58

Table 6.16: Heartbeats sent by client and probability of being dropped by the NAT table for all permutations of package loss with a frequency equal to 200 and a NAT timeout of 150-300.

An addition to this table is the "Number of times the connection was fully lost". This number represents the amount of times through a day the connection has to be fully reestablished, meaning that a connection could not be successfully reestablished by the client, and a full handshake had to be executed.

These numbers can be quite concerning, meaning a lot of handshakes has to be made through a day to keep the connection open.

In earlier tests with frequency set to 100, the amount of times the connection was fully lost and had to reestablished was < 0.5, showing a clear difference between frequencies larger and smaller than the minimum timeout.

Conclusion

In conclusion, changing the router to simulate a more realistic situation with varying timeouts, allows for more potential of the algorithm to be seen. Heartbeat frequencies larger than the lower end of the timeout scale, will limit the model's capabilities by a lot, but will never break the system, showing some resilience. The model will keep functioning, but the power consumption will increase as a result. For scenarios where the heartbeat frequency is still below the minimum timeout, the great potential for the model is still apparent and performance is still good.

Chapter 7

Discussion

This chapter will reflect upon the entirety of the project. It will go in chronological order, while weighing in on what went well and what could have been better.

This semester had some very open topics to choose a problem from. While this was fun, as the group was not limited by the initial topics, it also led to the first hurdle of the project. The group chose a general topic quite early on in the span of the project, but a more precise problem was harder to find. A few different directions were explored, some examples being a focus on hardware keys, asymmetric keys and backwards/forwards security. The initial protocol focused upon was an older version of TLS, but as it turns out, a newer version of TLS had already implemented the group's initial ideas. While these topics did not bear fruit, the group did attain a large amount of knowledge when it came to communication protocols in general, and this resulted in the project progressing relatively fast when a problem was chosen.

The group's supervisor had a friend/colleague who had a problem closely related to what the group wanted to do, and while the project is not exactly based on this problem, it did give the group a good base of what was possible, while affirming that this subject was an actual problem that could be worked on. The addition of this real world case also motivated and inspired the group to find a solution for the problem.

After finally choosing the problem statement (chapter 3), there was a period of smooth sailing for the project, as the group now had a direction, and there was no shortage of research to be done. However, fully understanding a security pro-

tocol by its implementation guide proved quite difficult [12]. This is the reason a subgroup of two people, started implementing an actual instance of the DTLS protocol. This was estimated to take a week or two in the span of the project, as it was the simple original implementation without any changes.The group did quickly realize that this was a larger endeavor than initially expected, and more group members were allocated to this implementation. However, even with this increased manpower, the implementation still took a little over a month. This was clearly not a viable rate of progression for the project, as a deadline was approaching rather quickly.

However when the implementation was up and running, two group members were able to start testing how the protocol acted in different environments, section 2.6.5. This greatly benefited the understanding of the protocol, and was in a stark contrast to the guide like documentation that was mainly used up until this point. In parallel, the rest of the group members started on implementing the DTLS handshake in UPPAAL, which was supposed to hold the changes to the heartbeat protocol as well as being a simulation and testing ground for the project.

As expected, the initial setup in UPPAAL was quite tough, and it did take some time. It was however a little harder than expected, as the group was unable to find any existing examples or documentation when it came to modeling DTLS. With the nature of the guide like documentation, finding the correct model for the project also took some time. The model used for DTLS handshake in the test cases, section 6.5, went through quite a few versions before the group settled with the one used. Another hurdle was also right around the corner when it came to these cases and power consumption. The group did not find an actual ratio when it came to power consumption until rather late in the project. This meant that these cases were tested with limited amount of time to analyze and reflect upon them.

Fortunately some positive aspects can be mentioned here. For the final iterations of the base DTLS model, the subgroup that had been working with an implementation of DTLS joined the UPPAAL subgroup, for a model review. This led to some positive changes, since the tests on the implementation had given a great inside into how the model behaved in different situations.

The last giant hurdle that the group had to overcome, was the fact that the tests and results were inaccurate. The group realized this less than a week before the project deadline, and thus underwent a large amount of pressure of fixing them, before handing in the project. The issue stemmed from the group gathering incorrect data for the type of encryptions and IoT devices, while also forgetting an important step when it came to the power calculation. Possible changes in data,

tests and results this late in the project was definitely not optimal, but fortunately did not lead to any major changes in the ratios between the cases.

Using estimated power consumption from the wrong encryption algorithms, did not prove to have much change when it came to the comparative analysis. This stems from all cases using the power consumption from encrypting and decrypting with a factor of bytes handled. A change from one type of encryption to another just meant the power consumption would change with a high factor, but evenly across the board. The only concern was with the different states a model could be in, since these numbers were correct from the beginning, and therefore were not overestimated with a factor of 10000 like the encryption and decryption costs were. Unfortunately there was another error with timing in the UPPAAL model during the handshake not being realistic. This meant different states could not be used directly on the model due to its power being measured in watts instead of joules like encryption and decryption was. See section 2.5.1, for a full description of the different costs and the full reasoning behind excluding the power consumption from different sates. Overall the period was stressing, but lead to more accurate, albeit very similar results.

Turning the focus towards the results of the tests (section 6.5.5, figure 6.15), some rather interesting trends and conclusion can be made. The different cases tested were created from a perspective, of what easily and intuitively could and maybe would be implemented for an IoT client/server architecture. However, these cases do not take into account, the inaccessibility of a private network. The client/server architecture relies on the server having some form of port-forwarding or other method providing similar attributes, for the client to be able to access the server. One cannot simply send a request to a device located on a private network, without extra measures taken by the user. Case 2, while not mentioned a lot, is the only case that does not suffer from this problem. In case 2, the client will always try to establish a connection to the server, when it realizes that there is not one currently. Case 2 then is the only case out of the 4 described, that does not rely on any other methods to ensure the server can get a hold of the client when a data request is issued. Therefore, it is very worthwhile remembering this key property when looking at the results. Other methods like case 3 and case 4, might outperform case 2 in some scenarios, but they will also need to rely on some other methods to ensure the server can get a hold of the client at any given time, ensuring on-demand data request from the user perspective.

Following the last thought and delving further into the result of the test cases, a clear battle between case 3 and case 4 could be seen. There is the argument that case 3 is best given its best case scenario, since case 3 technically can have a

zero cost from the parameters measured, if it never gets a data request. This is continuously true for one, two and maybe three data requests through an entire day, with case 3 still consuming less power than case 4. This however does not hold true for 4 or more data requests throughout a day. As a matter of fact, an increase in data requests, does not affect the power consumption of case 4 on the parameters measured, if it is within the time period of being continuously available. For case 2 we also do not have to stay within a certain window, and an increase in data requests will not affect the power usages by a noticeable amount on the parameters measured. On the other hand, an increase in data requests will linearly increase power consumption.

Time was a general constraint throughout the entire project as a whole, but generally a satisfactory result was achieved, albeit with less detail than originally imagined.

Chapter 8

Conclusion

In conclusion, the group has reached a result when it comes to the original problem statement. However, this conclusion is still quite open, as the optimal setup for a given scenario can vary wildly. For a case where client power consumption is of utmost importance, a simple heartbeat setup is the most beneficial, but it might lead to higher power consumption for the server-side, see section 6.5.2.

A final conclusion on the best method can, like in most cases, not be made. Given a system that is rarely used, keeping a connection alive like this would never be an optimal solution, as to with high frequency data request, a method utilizing heartbeat will be superior in more than one aspect. Case 2 utilizes heartbeat to keep a connection open, and will on the client side self revive, when a connection is considered lost (explained in sections 6.3, 6.5.2). The client will only be forced to send a heartbeat to the server, given something went wrong. This solves the issue with dropped packages over a UDP connection, without having the client do a relatively expensive action of sending an ACK to the server for each heartbeat (section 2.2). The self revive from the client side, also allows for on-demand data request from the server, since the client and server are always in communication. The server can request data from the client, without the client having its router port-forwarded or something similar, to achieve similar results on a private network. Also, since it is the client's responsibility to reestablish the connection if it were lost, the client and server will always be in communication given they have internet access. This directly answers the first part of the problem statements; *"How can a DTLS connection between a client and a server be kept alive in the NAT-table of the client's router, allowing the server to send requests to a client..."*. (Chapter 3).

Case 4 saw a lot of potential, but will not be able to provide these desirable properties when it comes to always on-demand connectivity. Case 2 performed very well in the tests, in the result section 6.5.5, along with providing more desired properties, and will be a strong recommendation for systems needing on-demand access on somewhat busy low powered devices, while wanting to save on power consumption.

The results of the tests in section 6.5.5, shows lower power consumption on the client side in case 2 compared to the two comparable cases; case 1 and case 3. With case 1 and case 3 representing a basic implementation, and case 2 being an implementation proven to use less power on the parameters measured (figure 6.15), case 2 also answers the later part of the problem statements; *"...,while reducing client power consumption and lowering response time?"*. (Chapter 3).

All in all, the group has found some seemingly promising data from the simulations, as the modified heartbeat model had quite impressive results in the simulated environment. While the results seem promising, there is no "one" perfect solution for this subject, and at the end of the day it will strongly vary from usecase to usecase.

8.1 Further Work

Due to time constraints, the different solutions were only constructed by modeling them in UPPAAL where different cases were made. This was done to allow the group to conduct an analysis of the different proposed cases and find the statistically best solution. However, the different cases were not implemented with actual code. If more time was available, further development of the project would be to implement these solutions instead of only having modeled simulations of them. Lastly, fewer restrictions on the protocol would allow it to more closely mirror realistic use cases.

Case 4

Expanding on case 4, this could especially prove useful for specific scenarios. As seen in the results, case 4 outperforms case 2 in raw power consumption, in scenarios favored by case 4 (section 6.5.6). If the protocol were to be implemented in a system with specifications aligning with case 4's uses, case 4 could outperform case

2, while still keeping case 2's desired properties of not needing port-forwarding.

Case 4 implemented in a system with time periods of expected request and a need for live data, with later time periods being fine with outdated information, could prove very efficient. Further testing on the link between heartbeat over time and amount of session resumption would be needed to more accurately decide between case 2 and case 4.

Bibliography

[1] Gerd Behrmann, Alexandre David, and Kim G. Larsen. *A Tutorial on Uppaal*. https://homes.cs.aau.dk/~adavid/RTSS05/UPPAAL-tutorial.pdf. (Accessed on 12/04/2023). 2005.

[2] Bowen Cai, Yu Chen, and Izzat Darwazeh. *Analyzing Energy Efficiency for IoT Devices with DRX Capability and Poisson Arrivals*. 2019. DOI: 10.1109/ICT.2019.8798791.

[3] *Configure Network Address Translation*. (Accessed on 10/30/2023). 2022. URL: https://www.cisco.com/c/en/us/support/docs/ip/network-address-translation-nat/13772-12.html.

[4] Alexandre David et al. *Uppaal SMC Tutorial*. https://uppaal.org/documentation/. (Accessed on 12/11/2023). 2018.

[5] Tim Fisher. *Private IP Addresses: Everything You Need to Know*. (Accessed on 10/25/2023). 2023. URL: https://www.lifewire.com/what-is-a-private-ip-address-2625970.

[6] *Internet protocol stack :: Computer Systems with Project Operating 2019*. https://www.it.uu.se/education/course/homepage/dsp/vt19/modules/module-1/tcpip-protocol-stack/. (Accessed on 24/10/2023).

[7] Puneet Kumar. *QUIC - A Quick Study*. (Accessed on 12/14/2023). URL: https://arxiv.org/pdf/2010.03059.pdf.

[8] James Kurose and Keith Ross. *Computer networking: A top-down approach, global edition*. en. 8th ed. London, England: Pearson Education, 2021.

[9] Robin Seggelmann Michael Williams Michael Tüxen. *RFC 6520 - Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension*. Feb. 2012. URL: https://datatracker.ietf.org/doc/rfc6520/.

[10] *Network Address Translation Definition | How NAT Works | Computer Networks | CompTIA*. (Accessed on 10/25/2023). URL: https://www.comptia.org/content/guides/what-is-network-address-translation.

[11] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. (Accessed on 10/18/2023). 2018. DOI: 10.17487/RFC8446. URL: https://www.rfc-editor.org/info/rfc8446.

[12] Eric Rescorla, Hannes Tschofenig, and Nagendra Modadugu. *The Datagram Transport Layer Security (DTLS) Protocol Version 1.3*. RFC 9147. Apr. 2022. DOI: 10.17487/RFC9147. URL: https://www.rfc-editor.org/info/rfc9147.

[13] *Return Routability Check for DTLS 1.2 and DTLS 1.3*. (Accessed on 12/14/2023). URL: https://datatracker.ietf.org/doc/html/draft-ietf-tls-dtls-rrc-09.

[14] *Socket in Computer Network - GeeksforGeeks*. https://www.geeksforgeeks.org/socket-in-computer-network/. (Accessed on 11/13/2023).

[15] Galini Tsoukaneri, Franscisco Garcia, and Mahesh K. Marina. "Narrowband IoT Device Energy Consumption Characterization and Optimizations". English. In: *International Conference on Embedded Wireless Systems and Networks (EWSN) 2020*. International Conference on Embedded Wireless Systems and Networks 2020, EWSN 2020 ; Conference date: 17-02-2020 Through 19-02-2020. Junction Publishing, Feb. 2020, pp. 1–12. ISBN: 978-0-9949886-4-5. URL: https://ewsn2020.conf.citi-lab.fr/.

[16] *UPPAAL Documentation*. https://docs.uppaal.org/. (Accessed on 04/12/2023).

[17] *UPPAAL Documentation: statistical$_q$ueries*. https://docs.uppaal.org/language-reference/query-syntax/statistical_queries/. (Accessed on 04/12/2023).

[18] *User Datagram Protocol - IBM Documentation*. https://www.ibm.com/docs/en/aix/7.1?topic=protocols-user-datagram-protocol. (Accessed on 10/19/2023).

[19] *User Datagram Protocol (UDP) - GeeksforGeeks*. https://www.geeksforgeeks.org/user-datagram-protocol-udp/. (Accessed on 10/19/2023).

[20] *Welcome to OpenSSL*. (Accessed on 11/08/2023). URL: https://www.openssl.org/.

[21] *What is the internet of things? | IBM*. https://www.ibm.com/topics/internet-of-things. (Accessed on 11/15/2023).

[22] *What is the User Datagram Protocol (UDP)? | Cloudflare*. https://www.cloudflare.com/learning/ddos/glossary/user-datagram-protocol-udp/. (Accessed on 10/19/2023).

[23] *What is UDP | From Header Structure to Packets Used in DDoS Attacks | Imperva*. https://www.imperva.com/learn/ddos/udp-user-datagram-protocol/. (Accessed on 10/19/2023).

[24] Wikipedia contributors. *Transmission Control Protocol — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Transmission_Control_Protocol&oldid=1189995605. [Online; accessed 19-December-2023]. 2023.

[25] Wikipedia contributors. *User Datagram Protocol — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=User_Datagram_Protocol&oldid=1189006171. [Online; accessed 19-December-2023]. 2023.

[26] *wolfSSL – Embedded SSL/TLS Library*. https://www.wolfssl.com/. (Accessed on 11/08/2023).

Appendix A

UPPAAL Handshake Template

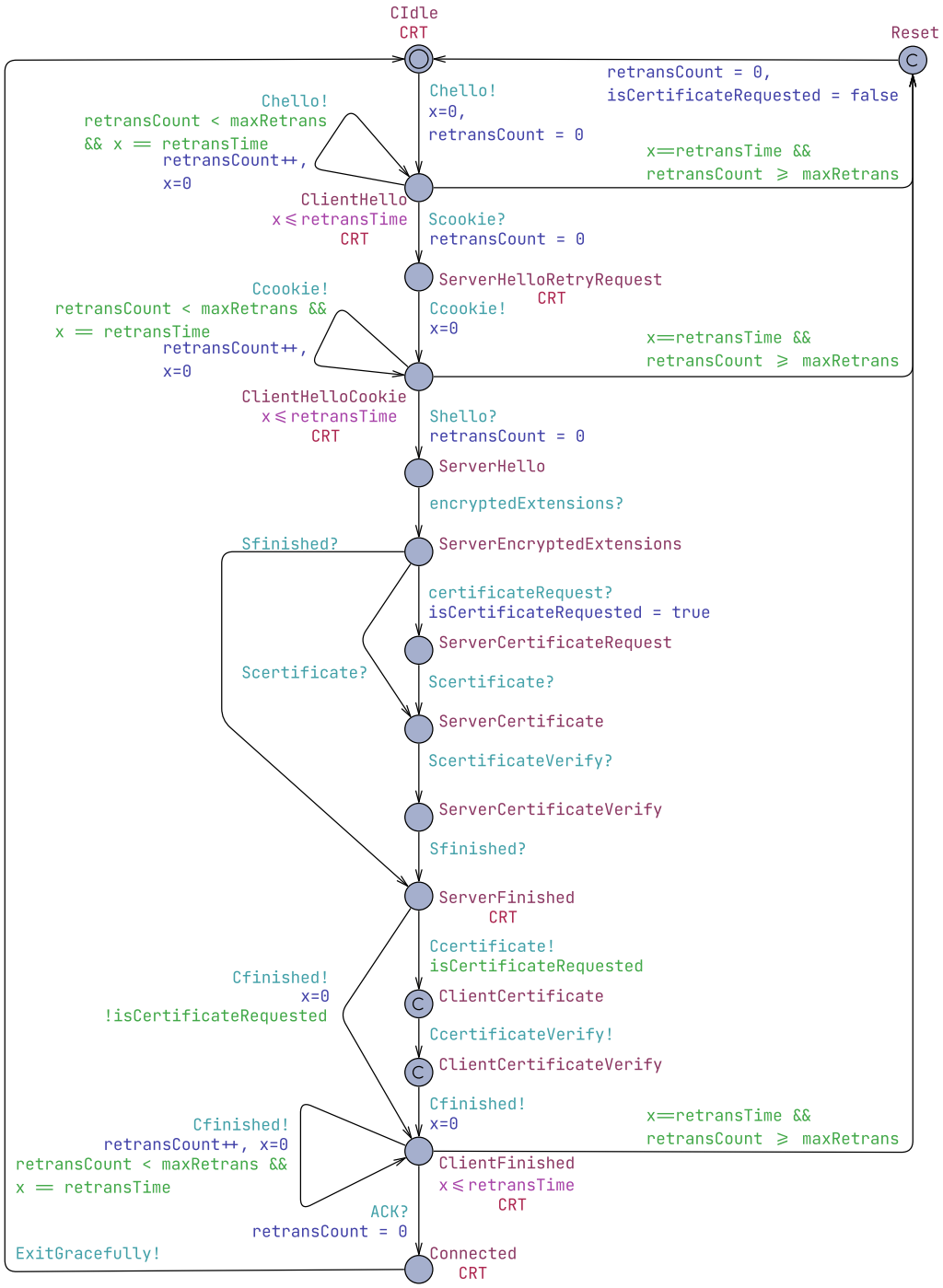

Figure A.1: The DTLS 1.3 handshake UPPAAL client model.

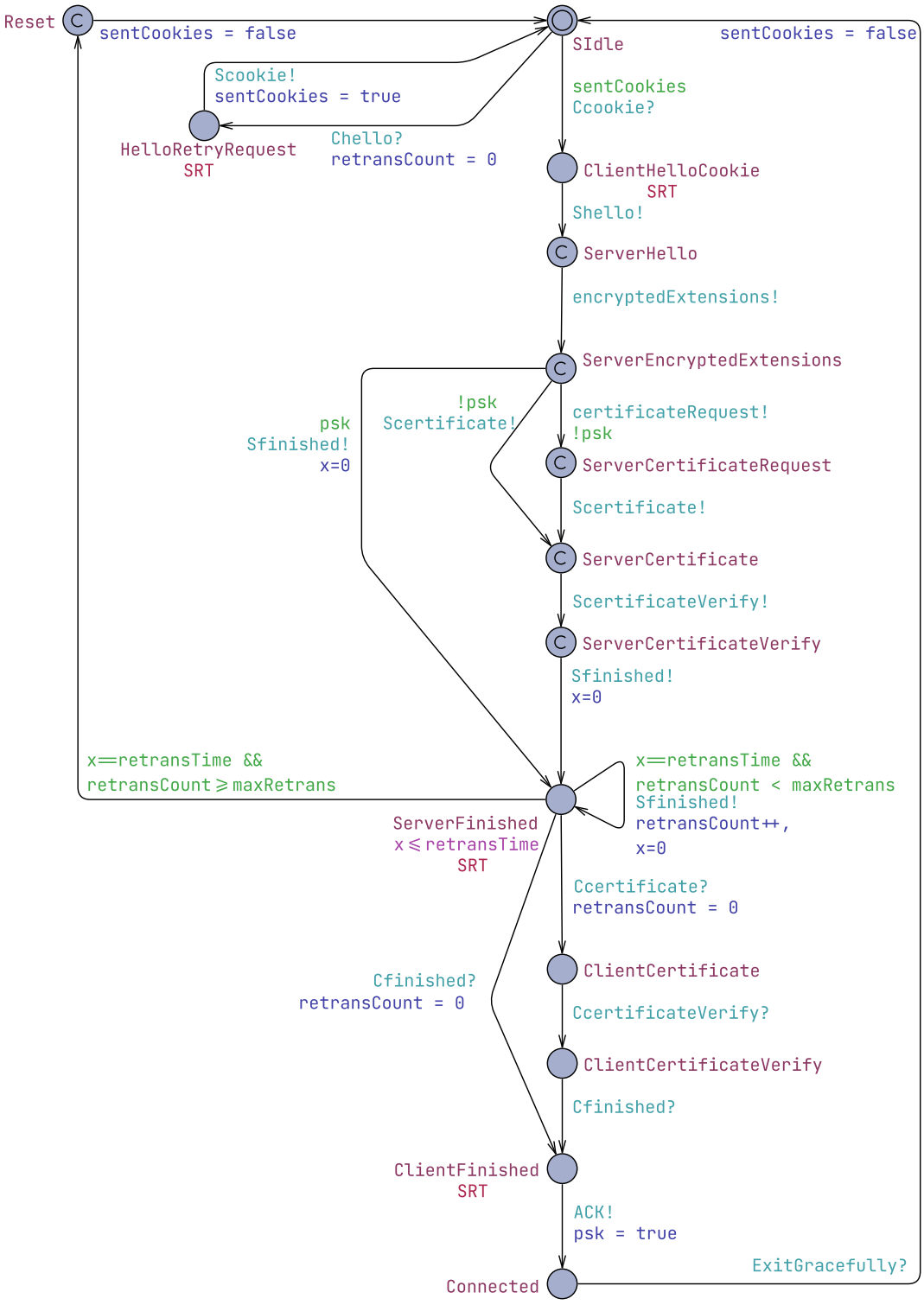

Figure A.2: The DTLS 1.3 handshake UPPAAL server model.

Appendix B

Full Case Data

Case	Interval	PSK	Heartbeat	Heartbeat package loss	Heartbeat frequency	Average power use
Base case	1800	No	No	N/A	N/A	97.321 ± 0.2307
Base case	3600	No	No	N/A	N/A	47.689 ± 0.1492
Base case	7200	No	No	N/A	N/A	22.806 ± 0.1089
Case 2	1800	No	Yes	1%	50	12.164 ± 0.0346
Case 2	3600	No	Yes	1%	50	12.251 ± 0.0345
Case 2	7200	No	Yes	1%	50	12.285 ± 0.0339
Case 2	1800	No	Yes	0.1%	50	12.263 ± 0.0335
Case 2	3600	No	Yes	0.1%	50	12.328 ± 0.0334
Case 2	7200	No	Yes	0.1%	50	12.369 ± 0.0335
Case 2	1800	No	Yes	0.01%	50	12.276 ± 0.0334
Case 2	3600	No	Yes	0.01%	50	12.351 ± 0.0334
Case 2	7200	No	Yes	0.01%	50	12.393 ± 0.0337
Case 2	1800	No	Yes	1%	100	7.034 ± 0.0351
Case 2	3600	No	Yes	1%	100	7.111 ± 0.0336
Case 2	7200	No	Yes	1%	100	7.150 ± 0.0334
Case 2	1800	No	Yes	0.1%	100	7.077 ± 0.0340
Case 2	3600	No	Yes	0.1%	100	7.137 ± 0.0334
Case 2	7200	No	Yes	0.1%	100	7.174 ± 0.0334
Case 2	1800	No	Yes	0.01%	100	7.064 ± 0.0335
Case 2	3600	No	Yes	0.01%	100	7.164 ± 0.0336
Case 2	7200	No	Yes	0.01%	100	7.179 ± 0.0336
Case 2	1800	No	Yes	1%	200	5.389 ± 0.1305
Case 2	3600	No	Yes	1%	200	5.025 ± 0.0928
Case 2	7200	No	Yes	1%	200	4.822 ± 0.0750
Case 2	1800	No	Yes	0.1%	200	4.573 ± 0.0483
Case 2	3600	No	Yes	0.1%	200	4.538 ± 0.0392
Case 2	7200	No	Yes	0.1%	200	4.607 ± 0.0362
Case 2	1800	No	Yes	0.01%	200	4.483 ± 0.0365
Case 2	3600	No	Yes	0.01%	200	4.553 ± 0.0363
Case 2	7200	No	Yes	0.01%	200	4.591 ± 0.0334
Case 3	1800	Yes	No	N/A	N/A	32.358 ± 0.0347
Case 3	3600	Yes	No	N/A	N/A	16.244 ± 0.0346
Case 3	7200	Yes	No	N/A	N/A	8.783 ± 0.0337

Table B.1: The variables set for base case, case 2 and case 3 and the resulting average power of running 500 runs.

Case	Interval	PSK	Heartbeat	Heartbeat package loss	Heartbeat frequency	Average power use
Case 4	1800	Yes	Yes	1%	50	22.369 ± 0.0341
Case 4	3600	Yes	Yes	1%	50	13.745 ± 0.0337
Case 4	7200	Yes	Yes	1%	50	10.422 ± 0.0339
Case 4	1800	Yes	Yes	0.1%	50	20.454 ± 0.0341
Case 4	3600	Yes	Yes	0.1%	50	13.760 ± 0.0343
Case 4	7200	Yes	Yes	0.1%	50	10.443 ± 0.0336
Case 4	1800	Yes	Yes	0.01%	50	20.452 ± 0.0342
Case 4	3600	Yes	Yes	0.01%	50	13.786 ± 0.0337
Case 4	7200	Yes	Yes	0.01%	50	10.454 ± 0.0339
Case 4	1800	Yes	Yes	1%	100	17.867 ± 0.0338
Case 4	3600	Yes	Yes	1%	100	11.245 ± 0.0342
Case 4	7200	Yes	Yes	1%	100	7.906 ± 0.0340
Case 4	1800	Yes	Yes	0.1%	100	17.945 ± 0.0338
Case 4	3600	Yes	Yes	0.1%	100	11.238 ± 0.0341
Case 4	7200	Yes	Yes	0.1%	100	7.908 ± 0.0339
Case 4	1800	Yes	Yes	0.01%	100	17.926 ± 0.0345
Case 4	3600	Yes	Yes	0.01%	100	11.263 ± 0.0344
Case 4	7200	Yes	Yes	0.01%	100	7.938 ± 0.034
Case 4	1800	Yes	Yes	1%	200	16.975 ± 0.1429
Case 4	3600	Yes	Yes	1%	200	10.093 ± 0.0413
Case 4	7200	Yes	Yes	1%	200	6.653 ± 0.0368
Case 4	1800	Yes	Yes	0.1%	200	16.698 ± 0.0351
Case 4	3600	Yes	Yes	0.1%	200	9.953 ± 0.0346
Case 4	7200	Yes	Yes	0.1%	200	6.684 ± 0.0343
Case 4	1800	Yes	Yes	0.01%	200	16.641 ± 0.0342
Case 4	3600	Yes	Yes	0.01%	200	10.010 ± 0.0336
Case 4	7200	Yes	Yes	0.01%	200	6.644 ± 0.0339

Table B.2: The variables set for case 4 and the resulting average power of running 500 runs.

Appendix C

UPPAAL Communication Templates

C.1 Router model

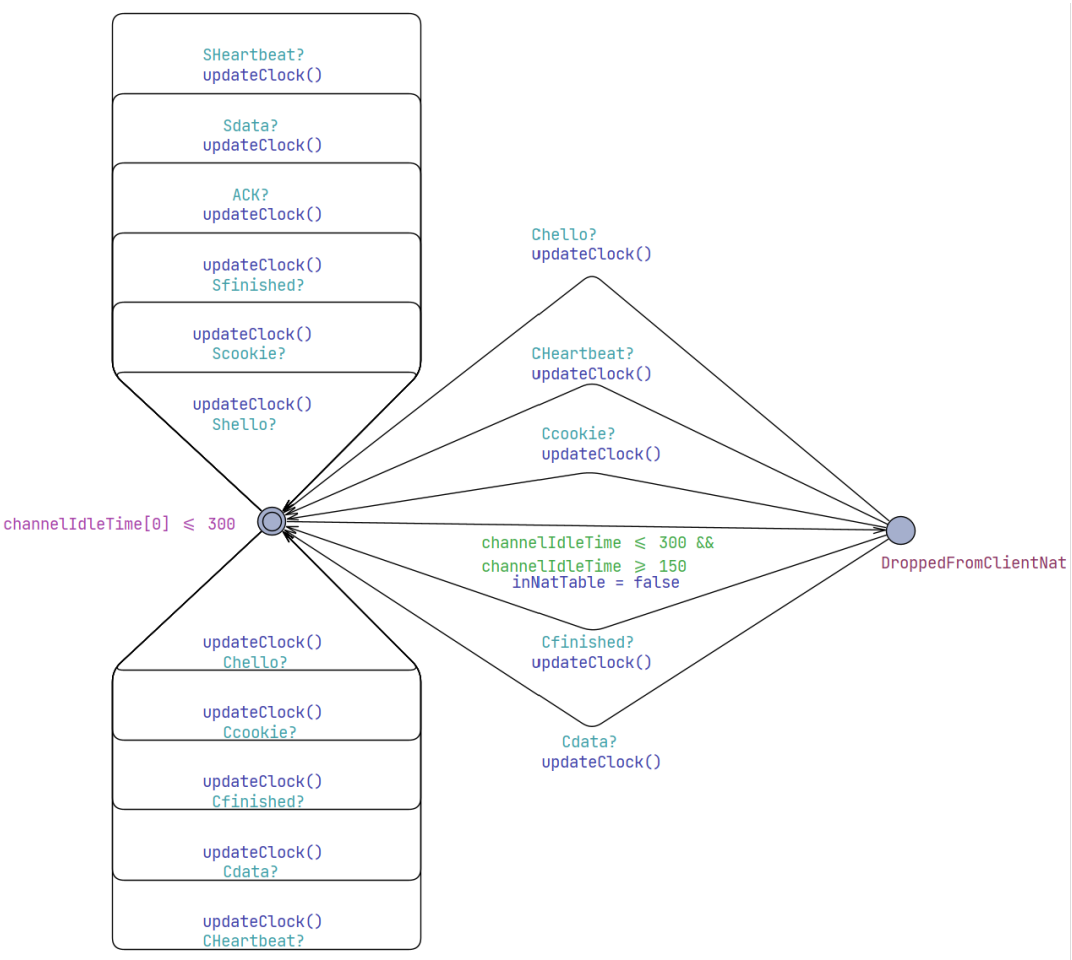

Figure C.1: The full router model.

C.2 Client Model

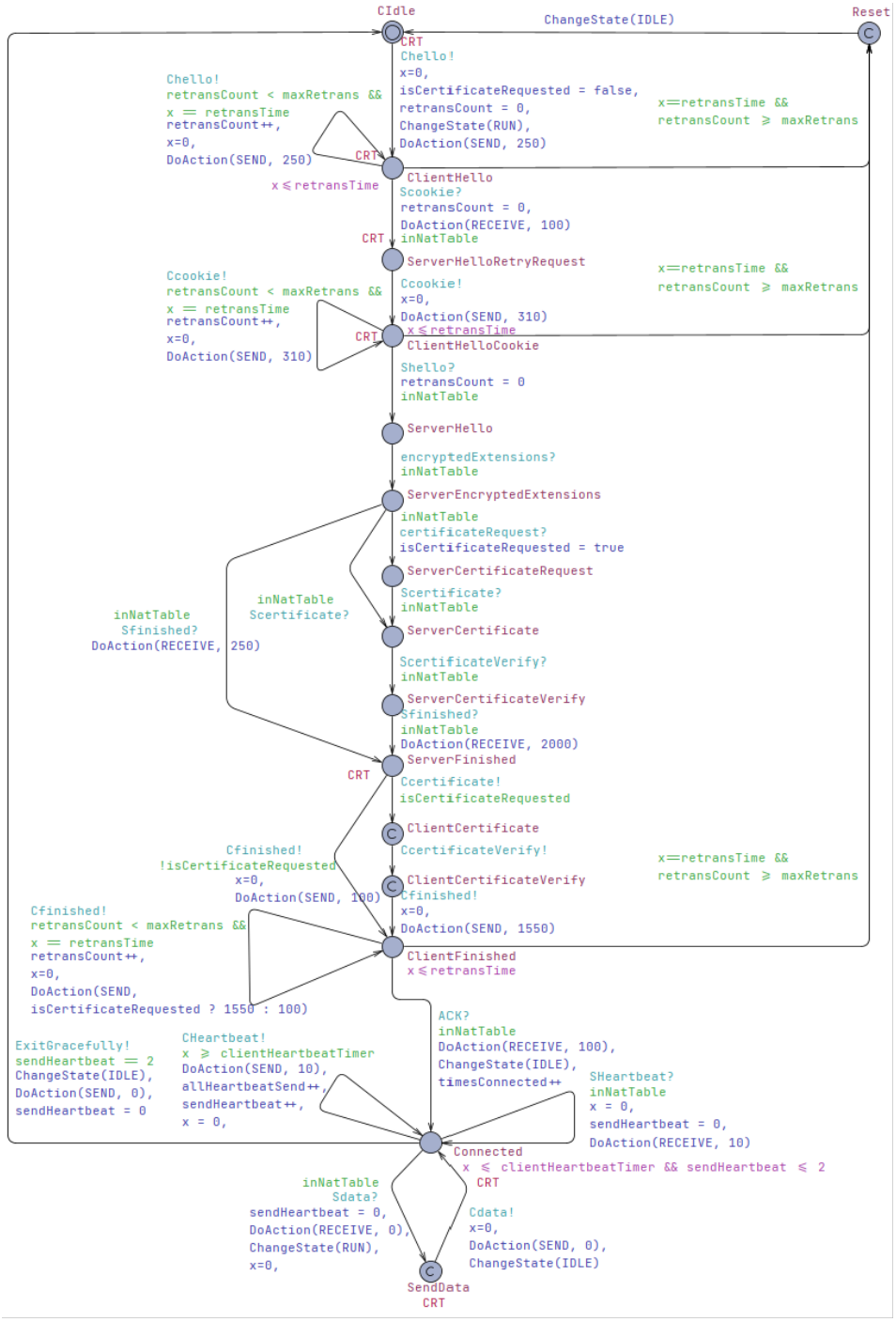

Figure C.2: The full case 2 client model.

C.3 ConnectionServer Model

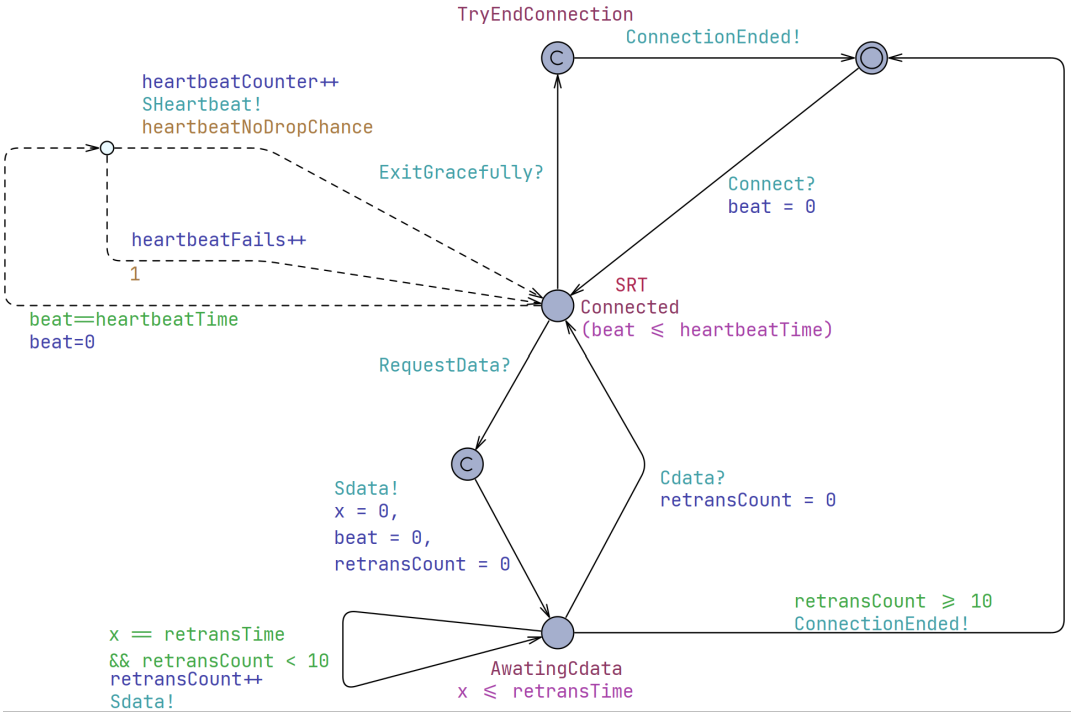

Figure C.3: The full case 2 ConnectionServer model.

C.4 User Model

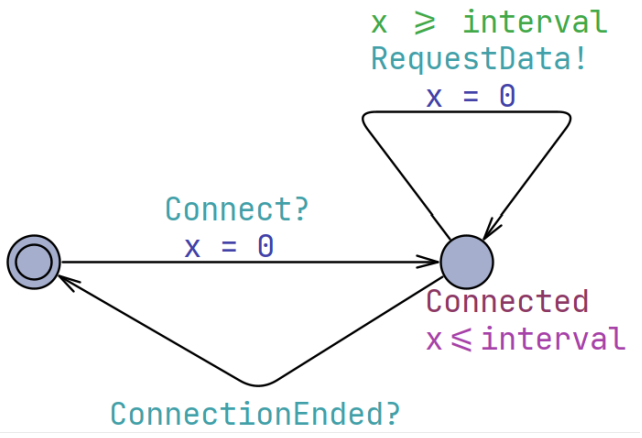

Figure C.4: The full User model.