FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Synchronization of hierarchical data
Student:	Bc. Richard Molnár
Supervisor:	RNDr. Josef Pelikán
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2020/21

Instructions

Design and implement network layers for data synchronization of complex hierarchical data. 3D data for AR/VR will be considered as a typical use case, concrete data formats and conditions will be defined in cooperation with Pocket Virtuality company.

Tasks:
1. Analyze user requirements and create a detail specification.
2. Define a low-level layer/library for packet transport.
3. Define and implement a higher (logical) layer for data transfer and synchronization. Synchronization of a hierarchical 3D data structure (FMcore) should be considered as a primary task of the project.
4. Consider a reliable transfer of small data packets (events).
5. Consider an optional transport of multimedia data with different requirements (non-reliable, time-critical data delivery).
6. Consider data security (encryption) options.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 5, 2020

Master’s thesis

Synchronization of Hierarchical Data

Bc. Richard Molnár

Department of Software Engineering
Supervisor: RNDr. Josef Pelikán

August 3, 2020

Acknowledgements

Throughout the writing of my master's thesis, I have received a great deal of support and assistance.

I would like to thank my supervisor, RNDr. Josef Pelikán, and everybody from Pocket Virtuality for their help and wonderful collaboration. I am very grateful for the opportunity I was given to work on such an amazing project.

I would also like to thank my family, friends, and everybody else who helped me during this time, especially Betka.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on August 3, 2020 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2020 Richard Molnár. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Molnár, Richard. *Synchronization of Hierarchical Data.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

Synchronizace a přenos komplexních dat po síti může být náročným problémem. Společnost Pocket Virtuality potřebovala řešení pro svou platformu pro vzdálenou přítomnost, která by poskytovala synchronizační služby mezi zařízeními v této platformě. Tato zařízení musí mít možnost rychle a snadno vyměňovat velká a komplexní hierarchická data ve formě grafů scén obsahujících 3D modely, textury a další binární data a synchronizovat kopie těchto struktur na jiných zařízeních.

Tato práce se zabývá analýzou stávajícího prostředí platformy Fata Morgana a jejích požadavků. Tato analýza je pak základem pro seznam funkčních a nefunkčních požadavků na síťové řešení.

Knihovna FMLink je výsledkem fáze návrhu a implementace této práce. Jedná se o síťovou knihovnu navrženou pro více platforem, vysoce efektivní a odolnou vůči zhoršeným síťovým podmínkám. Pro splnění bezpečnostních požadavků poskytuje FMLink podporu kryptografického protokolu TLS.

Tato knihovna je testována proti mnoha možným degradacím sítě a testována v různých emulovaných síťových podmínkách. Testy prokázaly, že knihovna může být nasazena a použita mnoha aplikacemi v platormě Fata Morgana.

Klíčová slova počítačové sítě, síťová synchronizace, TCP, UDP, Internet, TLS

Abstract

Synchronization and transfer of complex data over the network can be a challenging problem. The company Pocket Virtuality needed a solution for its remote-presence platform, which would provide synchronization services between the devices in this platform. These devices need to quickly and easily exchange large and complex hierarchical data in the form of the scene graphs containing 3D models, textures, and other binary data and synchronize copies of these structures on other devices.

This thesis revolves around the analysis of the existing environment of the Fata Morgana platform and its requirements. This analysis is then the basis for the list of functional and non-functional requirements of the networking solution.

The FMLink library is the result of the design and implementation phase of this thesis. It is a networking library designed to be multi-platform, highly efficient, and resistant to degraded network conditions. To satisfy the security requirements, FMLink provides support for the TLS cryptographic protocol.

This library is tested against multiple possible network degradations and tested in various emulated network conditions. The tests proved the library can be deployed and used by many Fata Morgana applications.

Keywords computer networks, network synchronization, TCP, UDP, Internet, TLS

Contents

Introduction **1**

1 Analysis **3**
1.1 Fata Morgana . . . 3
1.2 Basic architecture of Fata Morgana . . . 3
1.3 Synchronization Requirements . . . 4
1.3.1 Data transfer . . . 4
1.3.2 Data synchronization . . . 4
1.4 FMCore . . . 4
1.4.1 Node . . . 4
1.4.1.1 Node addresses . . . 5
1.4.2 Project . . . 5
1.4.3 Messenger . . . 5
1.4.4 Execution engine . . . 6
1.5 Environment . . . 6
1.6 Supported Devices . . . 6
1.6.1 Microsoft Hololens . . . 6
1.6.2 Desktops and Servers . . . 7
1.7 Software platforms . . . 9
1.7.1 Universal Windows Platform . . . 9
1.7.2 .NET . . . 9
1.7.2.1 .NET Framework vs .NET Core . . . 9
1.8 Legacy solutions . . . 9

2 Requirements **11**
2.1 Functional Requirements . . . 11
2.1.1 Create connections between Fata Morgana devices . . . 11
2.1.2 Sending of messages . . . 11
2.1.3 Project Synchronization . . . 12

2.2 Non-Functional Requirements 12
2.2.1 Compatibility 12
2.2.2 Security 13
2.2.3 Compliance 13
2.2.4 Reliability 13
2.2.5 Efficiency 13

3 Design **15**
3.1 StartSync and Notify 15
3.2 Node Shadows 15
3.3 Finding tree changes 16
3.3.1 Diff Utils 16
3.4 Serialization 17
3.4.1 FMCore BinaryArchive 17
3.4.2 FMPatchatableArchive 18
3.5 Choosing an internet layer 18
3.6 Messages 19
3.6.1 IMessage 19
3.6.1.1 Transport Message 20
3.6.2 Transport Chunks 20
3.7 Sending Messages 21
3.7.1 IFMLinkSocket 21
3.7.2 IFMLinkClient 22
3.8 Channels 23
3.9 FMLinkQueue 24
3.10 FMLinkSession 25

4 Implementation **27**
4.1 Platforms 27
4.1.1 Memory managemenent 27
4.1.2 Interfaces 28
4.2 Usage example 29
4.3 FMPatchableArchive 30
4.4 Messages 31
4.4.1 Synchronization Messages 31
4.4.2 TreeDiffMessage 31
4.4.3 Channel Messages 31
4.4.4 Data Messages 32
4.5 Synchronization 32
4.5.1 Start/Stop Sync 32
4.5.2 Notify 32
4.5.3 Detecting change 32
4.6 Interaction with Project 33
4.6.1 Multiple variants of API 33

4.7 Implementing the IFMLinkSocket and choosing network layer . 34
4.7.1 UDP 34
4.7.2 TCP 35
4.8 Establishing connections 35
4.9 Security 36

5 Testing **39**
5.1 Wireshark 39
5.1.1 Wireshark Generic Dissector 40
5.2 Creating various test conditions 41
5.2.1 Test environment 41
5.2.2 Testing the environment 43
5.2.2.1 Bandwidth test 43
5.2.2.2 Latency test 43
5.2.3 Emulating network conditions 43
5.2.4 Toxiproxy 44
5.2.5 Clumsy 45
5.2.6 TMnetsim 45
5.2.7 NetBalancer 46
5.3 FMLinkTester 46
5.3.1 Ping-Pong Test Scenario 46
5.3.1.1 Test results without network modifications . . 47
5.3.2 Testing delayed connection 47
5.3.3 Testing a connection with jitter 48
5.3.4 Testing connection with packet loss 49
5.3.5 Testing tree exchange 50
5.3.5.1 Testing maximum possible network utilization 51
5.3.5.2 Tree synchronization over lossy network 52

Conclusion **55**

Bibliography **57**

List of Figures

1.1 FMCore::Node class diagram 5
1.2 Example of network conditions. 7
1.3 Drawing of Microsoft Hololens 1 (from patent[1]) 8
1.4 Intel Compute Stick 8

3.1 Internet protocol suite model[2] 19
3.2 Transport Chunks and Transport Messages 20
3.3 FMLinkClient connection state machine 23
3.4 Simplified internal structure of FMLink 25

4.1 Example usage of FMLink for .NET 29
4.2 Example usage of FMLink for UWP 30

5.1 Part of .fdesc FMLink format 40
5.2 Wireshark Interface with FMLink plugin 41
5.3 Scheme of the test network 42
5.4 Output of the bandwidth test 43
5.5 Network latency histogram 44
5.6 Ping-Pong protocol 47
5.7 Distribution of round-trip times over an emulated network with delay and jitter 48
5.8 Histograms of cycle times in a network with packet loss 49
5.9 Utilization of the network 51

List of Tables

5.1 Test Machine Configurations 42
5.2 Test result of tree synchronization with different packet loss scenarios 52

Introduction

The purpose of this thesis is to analyze the problem of data synchronization in a very complex environment. Companies currently use many different solutions to transfer data over the network. These solutions can range from very simple to highly complex enterprise systems.

This thesis will not try to do develop an universal solution for every use case in every environment, instead, the goal is to develop a highly integrated solution for one specific software platform – Fata Morgana.

Fata Morgana is a platform for remote presence in virtual and augmented reality with real-time capabilities. The platform is already in development for several years now, but until now, it lacked an appropriate way of synchronizing data over the network between the devices in the Fata Morgana platform. These devices need an easy way of transporting large amounts of data, very efficiently, and in potentially highly varying network conditions.

This thesis was written for the purpose of analyzing, designing„ and implementing an universal and comprehensive networking solution for the needs of the Fata Morgana platform. The target is to develop a FMLink library, which will be able to support all the requirements of the Fata Morgana platform.

This networking solution needs to be designed to be high performance, efficient, and easily expandable in the future. It will need to be tested against network complications it might encounter in the environments where it will be deployed. Applications in the Fata Morgana ecosystem can be deployed on many different operating systems, device types, and CPU architectures and FMLink will have to consider this in its design.

Chapter 1

Analysis

1.1 Fata Morgana

Fata Morgana is the main product of the company Pocket Virtuality, based in Prague. To cite the company website:

> *"Fata Morgana is a remote presence platform. The system scans the environment and produces its virtual representation in real-time. This virtual venue can be joined and shared by other users from remote locations. The system facilitates collaboration of connected users in remote locations and in the real environment."[3]*

Fata Morgana(FM) is a very complex product and the full description of it is beyond the scope of this thesis, so I will focus only on the parts directly affecting FMLink.

The Fata Morgana platform consists of many physical devices, each of them having their own roles. These devices can run on different CPU architectures and operating systems. With this arises a need to synchronize data between the devices in a fast and efficient way.

These devices can be thousands of kilometers apart and in different internet networks – some of them may not even be connected to the global internet.

1.2 Basic architecture of Fata Morgana

Each device in the Fata Morgana platform runs some kind of FM software. Currently, this SW runs on top of the operating system – but this is not required, and in the future, some IoT devices running only with very light firmware are being considered. Fata Morgana software is composed of various company-developed libraries. Currently, every FM device requires at least two libraries – FMCore, containing basic data types used, and FMLink, to allow

connection to other devices. Other components are added depending on the application requirements.

1.3 Synchronization Requirements

The main purpose of the Fata Morgana platform is to provide remote presence services in virtual and augmented reality. This fundamentally brings the challenge of transferring large volumes of data. By the request of the Pocket Virtuality, FMLink will need to provide two services – data transfer and data synchronization.

1.3.1 Data transfer

Platform applications often need a simple way to transfer data from their device to some other device. This data can be of any quality – it can be text, video, raw binary data, or serialized objects. Clients (applications built upon synchronization library) provide the data messages and the service ensures the other side receives them.

1.3.2 Data synchronization

Another service clients will need is data synchronization. In this case, the application exposes its data to the synchronization service and the service ensures that the data is synchronized to other participants in communication. When the data changes on any device, the changes must propagate to other devices, so all devices have the same data. Any side of the communication can change the synchronized data and the change must be transferred to other clients automatically.

1.4 FMCore

As mentioned before, devices in the Fata Morgana platform need to expose and transfer many different data types. Every application keeps part of its internal data in the form of a tree, with nodes, edges, and attributes. All these and more are defined in the FMCore library. This library was already developed by Pocket Virtuality, but it will need to be expanded to fulfill the needs of the FMLink solution.

1.4.1 Node

The tree structure used by the FM applications is defined in the FMCore library. The main class used is FMCore::Node, which represents nodes of the data tree. The tree has bidirectional links – each node has a reference to its parent and vice versa. Application data can be represented by the tree

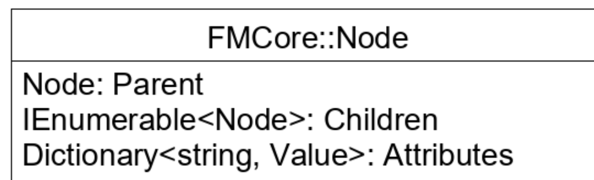

Figure 1.1: FMCore::Node class diagram

structure itself or as part of node attributes. Each node can have a set of different attributes. An attribute is characterized by its name and Value.

Value is class comprised of three important members – enum value, specifying the value type of the payload, serial number, used for uniquely identifying the concrete value and the payload itself. The value type allowed can be of any predefined data type – the data types allowed for the attributes are also defined and listed in the FMCore library. They can be of very simple nature – scalars, vectors, fixed-size matrices – or complex ones – textures, XML documents, etc. The synchronization solution for Fata Morgana will need to know how to serialize and process these data types.

Some applications store whole 3D models as trees in their data context (each vertex representing a Node), thus the tree can have millions of nodes and attributes.

1.4.1.1 Node addresses

The location of each node in a tree can be characterized by a string address. An example of such an address is "/foo/bar", which describes a node named "bar", that is a child of a node named "foo". This syntax is very similar to the Linux syntax of locating files – it also shares the usage of '.' and '..' for specifying "local" and "parent" respectively.

1.4.2 Project

The main data tree of a given application is encapsulated in a structure called "Project" and is represented by the FMCore::Project class. Project is the main data type synchronization is based around and it is the only object the synchronization service needs to access from the application. It has two main members: the root of the data tree and the messenger. The project also contains a basic execution engine.

1.4.3 Messenger

FMCore::Messenger is a class used to route messages inside of one application instance. It can be used to notify different components of activity from other components. The messages are characterized by their type ID, subtype ID,

and other optional parameters. Different components of an application can send messages to the Project's messenger. Components can then subscribe to messages from the messenger and only receive messages of the requested type. FMLink can use this messenger to notify other components that the data under synchronization has been modified by the other party.

1.4.4 Execution engine

The Project has an ability to accept commands (usually expressed as lambda expressions in c# or c++) and execute them. This is used so the Project can control access to the data tree. Every action that accesses or modifies the data tree should be bundled and executed as "Command", represented by the FMCore:Command class. Project takes these Commands and queues them for execution, and when possible, executes them in order. Commands may or may not be "exclusive". If a Command modifies the data inside the tree, it should be marked as exclusive – during its execution, no other Commands will be executed. If the Command only reads the data inside the tree, it should be marked as non-exclusive – in this case, the execution engine will allow it to run in parallel to other non-exclusive commands that are successive in the Command queue.

1.5 Environment

Fata Morgana platform is heavily distributed – it runs on many devices. Currently, the devices are mainly Microsoft Hololens, Windows Desktops, and Windows Server. The devices can be connected using various connections, such as Wi-Fi, LTE, or Ethernet. Some devices may not even be permanently connected to the Internet. The quality of connectivity between devices (mainly Wi-Fi) can be highly degraded since the platform might be deployed inside a factory, mining site, or other challenging environments.

1.6 Supported Devices

Fata Morgana applications need to run on many different device types, even different CPU architectures. Currently, only the Windows devices, but other operating systems will be considered in the future.

1.6.1 Microsoft Hololens

> *"Microsoft HoloLens 2 is an untethered holographic computer. It refines the holographic computing journey started by HoloLens (1st gen) to provide a more comfortable and immersive experience paired with more options for collaborating in mixed reality."[4]*

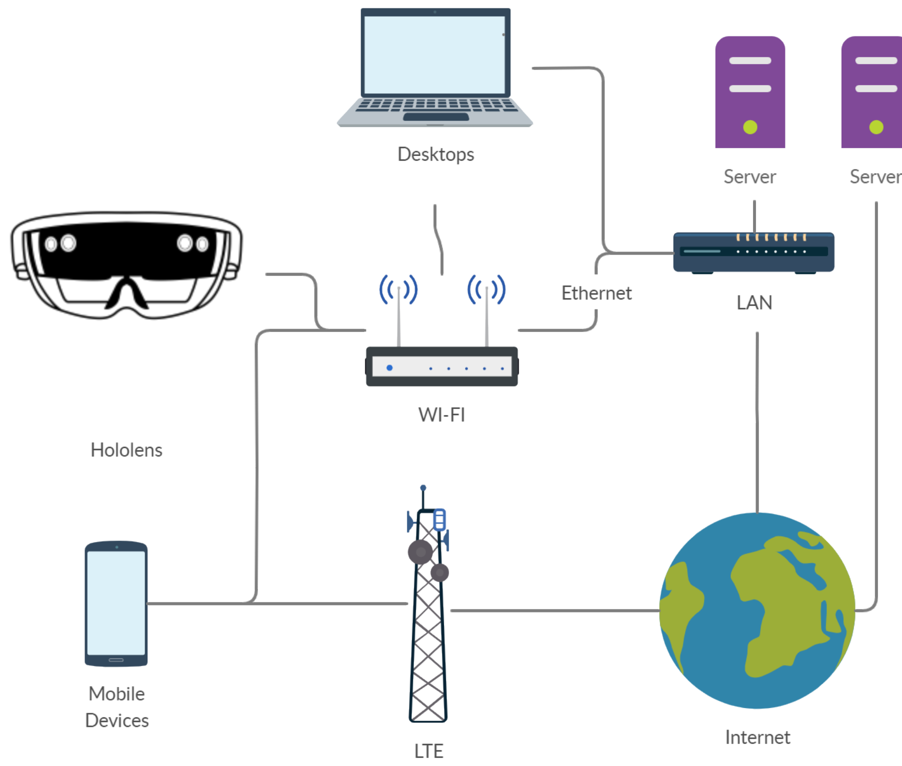

Figure 1.2: Example of network conditions.

Hololens is a device from Microsoft for augmented reality applications. Its main feature is a pair of holographic displays allowing it to display digital objects in the real world. It is a completely stand-alone device – it includes its own processing unit, which is running Windows Holographic Operating System (a modified version of Windows 10).

There are currently versions 1 and 2 of Hololens, which are both supported by Fata Morgana. The main difference from the FMLink perspective is CPU architecture. Hololens 1 uses old Intel Atom x86 CPU and Hololens 2 uses Qualcomm Snapdragon 850 ARM CPU. ARM and x86 instruction sets are not mutually compatible, therefore FMLink needs to take this into account.

1.6.2 Desktops and Servers

These devices are classic desktops and servers running Windows 10 and Windows Server 2019. They all run on the x86-64 architecture, and thus can use all the common tools available to these platforms. Some of the ”desktops” might have very little computing power – one of them is Intel Compute Stick. This device includes Intel Atom x5-Z8300, a quad-core 1.44 GHz x86-64 CPU,

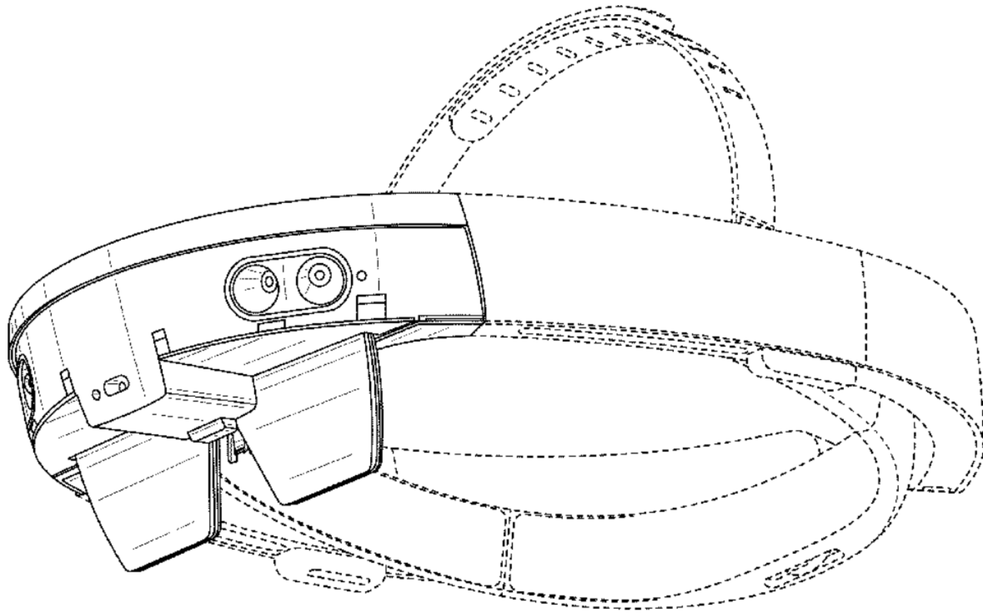

Figure 1.3: Drawing of Microsoft Hololens 1 (from patent[1])

and only 2 GB of RAM. This might seem like a lot, but this device is running full Windows 10 OS, so there is not much left for applications. The main advantages of this device are its size, which is slightly larger than an average USB flash drive, and power requirements – it can be powered by a standard USB charger.

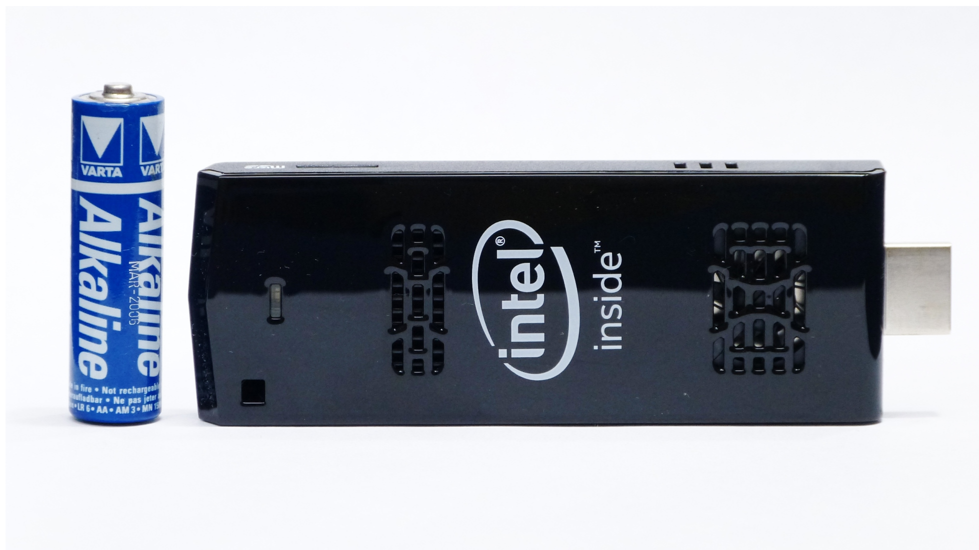

Figure 1.4: Intel Compute Stick

1.7 Software platforms

Since there are many different device types, several Fata Morgana components are being developed for two different platforms - Universal Windows Platform and Microsoft .NET.

1.7.1 Universal Windows Platform

Universal Windows Platform – UWP for short – was chosen as a platform for native development for Windows. UWP was chosen mostly because of Hololens and other low power devices. FMCore and other existing Fata Morgana libraries are written using C++/WinRT runtime, so FMLink will also use these tools to target the UWP platform. These tools natively support both ARM and x86 CPUs.

> *"C++/WinRT is an entirely standard modern C++17 language projection for Windows Runtime (WinRT) APIs, implemented as a header-file-based library, and designed to provide you with first-class access to the modern Windows API. With C++/WinRT, you can author and consume Windows Runtime APIs using any standards-compliant C++17 compiler."[5]*

1.7.2 .NET

Another platform that Fata Morgana targets is .NET. This platform was previously chosen for its powerful capabilities in building server applications and faster and simpler development, compared to C++/WinRT.

1.7.2.1 .NET Framework vs .NET Core

At the start of the FMLink development, Fata Morgana libraries were targeting .NET Framework version 4.7.2. During the development, the decision was made to switch the whole Fata Morgana platform to .NET Core 3.1. This brought many advantages – .NET Core offers much better portability to other operating systems. This was almost necessary since version 4.8 of the .NET Framework is probably the last one and only the .NET Core will continue its development.

1.8 Legacy solutions

Before the deployment of the FMLink, the Fata Morgana platform used various legacy techniques to synchronize data. The most common was very basic HTTP pooling from clients to servers. This was very inefficient and slow. Since the polling was executed at fixed intervals, there was a necessary latency created by this polling interval. Every time there was even a small change

in the application data, the whole application state had to be transferred. Combined with very inefficient data serialization, this was limited by both CPU computing power and network throughput. Applications also had to include a lot of ”boilerplate” code for the handling of HTTP communication and manual serialization.

Chapter 2

Requirements

In this chapter, I will discuss the functional and non-functional requirements of FMLink, which have been identified from the analysis phase. The final library will need to consider and satisfy these requirements. The term ”user” will be used, which represents the application that will include and use the FMLink library.

2.1 Functional Requirements

The main purpose of FMLink is to provide a simple way of connecting to other Fata Morgana devices, synchronize selected subtrees of a Project and send and receive any arbitrary data. Functional requirements need to take this into account.

2.1.1 Create connections between Fata Morgana devices

Users of the FMLink library need a simple way to connect to other Fata Morgana devices. User should supply only the hostname and port of the other device and FMLink should handle the rest and establish a connection which can be used further. When the user wants to close the connection, FMLink should notify the other party and gracefully close the connection on both ends. The user should also be notified if the connection gets to a broken state (other party not responding, network disconnection, etc.).

2.1.2 Sending of messages

Once the connection is established, users of the library should be able to send messages to the other party. The messages can contain any data, as soon as the data is immutable and serializable. The messages can be expected to be very large, up to single gigabytes in extreme situations. The library needs to

return control to the user immediately when the user tries to send a message, so it doesn't block the execution of the user thread.

If the user tries to send a very large message on a slow connection, he should still be able to send other messages with higher priority, even if the first large message hasn't been fully transmitted yet. Once these have been transmitted, the original large message should continue the transfer. This implies that there needs to be some mechanism in place which allows the user to assign different priorities to messages and FMLink needs to respect that.

2.1.3 Project Synchronization

The application using FMLink can have some of its data stored inside the Project. FMLink needs to be able to synchronize this data with the other party, making sure that the other party receives a correct copy of this data. Since it is in the form a tree, the user needs to be able to select which subtree to synchronize to the other party's destination subtree. The remote path of the synchronized subtree might be different from the original local path of the subtree.

If the user makes any changes in the subtree marked for synchronization, the user will notify FMLink of it – FMLink then needs to ensure, that any changes made to the subtree are replicated on the remote party's subtree. This is required both ways – if a remote party makes such changes to their subtree, the changes should also reflect locally.

To easily process changes made to users Project tree, FMLink should notify the user when it received and applied the changes from the other party. It should apply the changes in a way that will not corrupt any data in the Project tree or cause undefined behavior.

2.2 Non-Functional Requirements

From the analysis chapter, several major non-functional requirements emerged. These requirements are probably the largest factor of why FMLink became it's own and large project – if the following requirements were relaxed, the development of synchronization solution for Fata Morgana would take only a small fraction of development efforts. They can be grouped into several categories: portability, security, compliance, reliability, and efficiency.

2.2.1 Compatibility

As mentioned before, Fata Morgana is currently targeting .NET and UWP platforms and its libraries and applications are programmed in C# and C++/WinRT respectively. Therefore, UWP users should be able to use native FMLink and .NET users should be able to use the .NET library. Not

only that, but client applications should be able to communicate with other client applications running on different platforms.

2.2.2 Security

Customers using the Fata Morgana platform might be transferring sensitive and business-critical data. Therefore FMLink needs to provide all aspects of a well-secured communication protocol.

The first is access control – devices using FMLink must be able to properly authenticate and authorize themselves and also other devices. In this context, authentication is the process of retrieving and verifying of an identity. Authorization then specifies what access and privileges the given identity has. In general, this means that FMLink must be able to defend itself against an unauthorized attacker.

FMLink also needs to be able to encrypt the data it transfers, since it can be very sensitive. The encryption must be fast since the devices can have very little processing power. It also needs to be resistant to potential attacks from well-funded attackers.

2.2.3 Compliance

Since the Fata Morgana platform will have access to sensitive customer data, customers will probably need some assurance that this solution is safe to install into their environment. One of the common ways to achieve this is to get the product audited by an independent organization specializing in security audits of IT solutions. This needs to be taken into account when designing security aspects of FMLink since they will be a large part of a security audit.

2.2.4 Reliability

Fata Morgana devices might be deployed into an environment, where network conditions might be very poor. FMLink needs to account for this and provide means to reliably transfer information. It must prevent the potential delivery of corrupted information to other devices, which might cause catastrophic consequences. FMLink should also be to reasonable work in conditions, where information loss during transfer is common and account for this. Messages delivered to other parties by FMLink need to be delivered in an order that the user sending them expects. This means that FMLink needs to be designed in a way where delivery of messages is always in deterministic order, regardless of network conditions.

2.2.5 Efficiency

FMLink needs to utilize network resources efficiently. Network latency and especially throughput needs to be considered since the amount of data a given

network is able to transfer can be a very finite resource. FMLink should not add unnecessary delay and latency to communication in a network with significant delay. FMLink also needs to be data-efficient – when synchronizing changes made to the Node tree, only the changes in the tree should be transferred.

Chapter 3

Design

When simplified a lot, the basic purpose of FMLink is synchronization of user-provided Node trees and sending other messages to the other party. Let's start by decomposing the main problem, synchronization of the Node trees.

3.1 StartSync and Notify

At the top level of the FMLink API, two main methods used for subtree synchronization will be provided, StartSync, and Notify. StartSync takes a Node tree as a parameter – this way FMLink obtains a reference to the tree user wants to have synchronized with other party and it instructs FMLink to start checking the tree for changes. It is expected the user will call the Notify method every time he makes a change to the tree – this way FMLink doesn't have to periodically check for changes in the tree, only when the Notify method is called. This also allows it to be more predictable about when will FMLink send synchronization messages to the other party.

3.2 Node Shadows

Each time the user calls the Notify method, FMLink needs to know what changed from the last known state of the tree that was selected for synchronization. The basic solution to this problem would be to keep a copy of the whole tree. Every time the user would notify FMLink, it would check it against its own copy of the tree, find the differences and then make a new copy of the current tree for the next checking of differences. It becomes obvious that this solution brings a lot of copying, memory allocation, and would be highly ineffective since it would basically double the memory requirements of the Node tree.

As a solution for this, a concept of serial numbers in FMCore needs to be explained first. Each node attribute is described by its name and Value. Value

is defined in FMCore documentation as follows :

> Value is immutable. If one desires to modify the Value, a new one must be created and then the particular node attribute has to be replaced. A Value holds a payload, its simplified type, and a serial number (for detecting modifications).

This means that every time an attribute has changed, it is possible to detect this by checking the serial number of the attribute's Value.

FMLink takes advantage of this and creates a new concept of "Shadow Nodes". These nodes are very similar to the normal Node tree structure except for one difference – Values of attributes have an empty payload. In order to create Shadow Node tree, FMLink copies the supplied Node tree marked for synchronization and creates a "Shadow Copy". This copy has exactly the same structure as the original tree, but it omits the actual payload of the attribute Values while keeping the serial numbers and ValueTypes of the Values. The payloads of Values account for the majority of the data, so this structure uses much less memory space.

Using this Shadow Nodes, when a user first marks some Node tree for synchronization (using the StartSync method), FMLink makes a shadow copy of this tree, but also it also keeps a reference to the original tree. When the user notifies the FMLink of possible changes, FMLink can check for changes by checking the tree structure and attribute Serial numbers against its own copy in the form of Shadow tree. When the check and synchronization are completed, the Shadow tree is actualized to reflect the current state of the user's Node tree.

3.3 Finding tree changes

Now that FMLink has access to two Node trees (one that is a reference to the current tree and one that is Shadow copy of the last known state of the tree), it needs to compare them against each other and find the differences. Then, it can serialize these differences, bundle them as a message, and send them to the other connected party. Finally, when the other party applies these differences to their Node tree, this will effectively synchronize the trees to hold the same data and have the same structure.

3.3.1 Diff Utils

To find the actual differences between the two trees, I have designed DiffUtils. This utility takes two trees (the old, shadow copy, and the new, full Node tree with Values) and compares their structure and serial numbers of attribute Values and produces a "Patch". The patch fully describes the changes the user made to the new tree, compared to the old one. This patch can then be

applied to the tree on the other side of the connection and it will transform it and thus synchronizes it.

3.4 Serialization

FMLink needs to provide a simple means of transporting the tree structure and changes in the tree. In order to do this, it needs to be able to turn this data into a stream of bytes and reconstruct this data from the provided stream of bytes – a process called serialization and deserialization.

Node attributes in the Project tree have 39 predefined possible Value types in the FMCore library. There are numerical value types, variable-length arrays, fixed sized matrices, and other types commonly used in applications working with computer graphics. The full list is listed in the ValueType enum in the FMCore library.

Fortunately, FMCore already contains a means of serializing and deserializing these types, so FMLink can leverage it. FMCore includes a serializer and deserializer that takes a Value with a given ValueType and platform-dependent binary reader/writer and uses it as byte storage.

3.4.1 FMCore BinaryArchive

FMCore library contains its own binary format for storing Node trees called FMBinaryArchive. This binary format is able to store a single tree in a given byte stream, including all its attribute values and the structure of the tree itself. It is quite a simple format, which works recursively. The algorithm can be described in 3 parts :

1. Write begin marker
2. Recursively go through the whole tree and write each node.

   Each node written consists of several fields :

   a) Node start marker
   b) Node address in the tree
   c) Node name
   d) List of attributes, where each attribute consist of its value type, name, and actual serialized value
   e) Node end marker
3. Write end marker

During the deserialization process, as the algorithm reads the stream of bytes, it deserializes the nodes contained and gradually builds the whole tree. Since the tree is serialized recursively, each time a node is read during deserialization,

it is certain that its parent was already deserialized and thus the Node can be added to its list of children.

This format works well for storing complete information about a given tree. However, FMLink needs something more. It needs to maintain a synchronized copy of a tree on the other side of the connection. If it sent the whole tree every time a single number changed in a very big tree, it would be sending a lot of redundant information. Therefore some system needs to be designed, which detects actual changes made to the tree and transfers only those.

3.4.2 FMPatchatableArchive

As part of this thesis, a new type of archive type has been proposed and implemented in order to support the requirements of FMLink. This format is formally version 2 of the existing FMBinaryArchive format. It was designed to have additional ability to create archives that store tree ”patches”, which can be applied to an existing tree and modify it. This extended format is able to specify nodes and attributes to be removed from an already existing tree. More information about this format is discussed in the Implementation chapter.

3.5 Choosing an internet layer

Since FMLink needs to run over the internet, the decision was needed which network protocols it is going to use. This decision needs to take into account the requirements specified. If the layer chosen is too high, the protocol might have issues achieving some goals set by the requirements, since the higher the abstraction, the lower the amount of control possible. For example, if HTTP was chosen, which is running on an application layer, FMLink would lose a lot of control and for example, it wouldn't be possible to get finer control of security, latency, frame sizes, etc. However, if the layer chosen is too low, it might be needed to do an unnecessary large amount of development, which might not even provide better results than already existing solutions. In the following description, I was following the TCP/IP model of the Internet, as opposed to the OSI model.

The lowest TCP/IP layer in the TCP/IP model is the link layer. Since this layer is not routed through the Internet, which is a requirement of FMLink, this layer was not selected. A level higher is the Internet Protocol layer, with common implementations such as IPv4, IPv6, ICMP, IPsec, and more. Since protocols such as IPv4 and IPv6 are already widely used and supported, it would be unfeasible to easily develop custom Internet layer protocols that would get routed through the Internet.

The third layer is the Transport Layer. The most common protocols of this layer are UDP and TCP. In theory, it would be possible to develop custom Internet Layer protocol, these protocols would even be routed through the

OSI Model	TCP/IP Model	Protocols
Application layer	Application Layer	DNS, DHCP, FTP, HTTP, IMAP, LDAP, NTP, POP3
Presentation Layer		JPEG, MIDI, MPEG, TIFF
Session Layer		NrtBIOS, NFD, PAP, SCP, SQL, ZIP
Transport Layer	Transport Layer	TCP and UDP
Network Layer	Internet Layer	ICMP, IGMP, Ipsec, IPv6, IPX
Data Link Layer	Link Layer	ARP, ATM, CDP, FDDI, Frame-Relay, HDLC, PPP, STP, Token ring
Physical Layer		Ethernet, DSL, ISDN, Bluetooth

Figure 3.1: Internet protocol suite model[2]

Internet. However, UDP and TCP already offer enough functionality and performance for use by FMLink, so the decision was to use these two protocols.

Higher-level protocols and messaging systems, such as Kafka, ZeroMQ, or RabbitMQ were not used, since the communication protocol needs to be highly integrated with other FM components and the company wants full control over how the protocol works and behaves. It would be possible to adapt some already existing networking framework, but this would bring an unnecessary and unwanted dependency to additional external software solutions and additional undesired complexity.

3.6 Messages

As opposed to stream protocols, FMLink is a protocol based on the concept of messages, a so-called messaging protocol. Every bit of information that FMLink sends or receives is represented by a message. These messages need some way to be represented in the FMLink library and also in network communication.

3.6.1 IMessage

At lower levels, all messages sent through FMLink are implemented as classes with the IMessage interface. This interface has only two public methods: serialize and deserialize. Every type of message sent through FMLink needs to implement this interface, so it can be sent via a serial link. This interface guarantees that every instance of a message can be serialized and deserialized just by calling the interface methods. However, since FMLink doesn't handle serialization of these message objects, every class needs to implement the serialization by itself. This brings the benefit that the serialization can store the data very efficiently since it is written individually for each class as opposed to universal serialization.

3.6.1.1 Transport Message

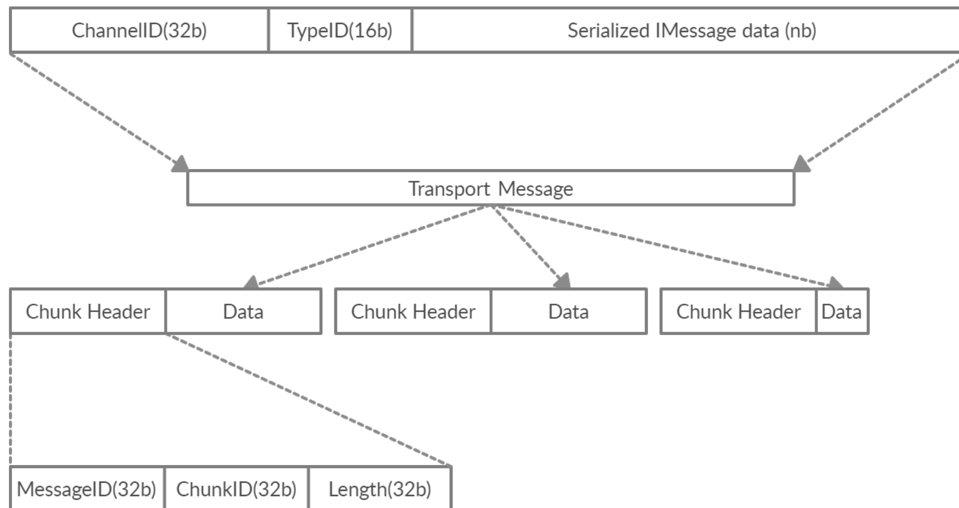

Figure 3.2: Transport Chunks and Transport Messages

After the IMessage object is serialized, some additional information is added in front of the stream of serialized data in form of a header. The header contains ChannelID (the concept of channels will be explained later) and TypeID. TypeID is used mainly during the deserialization and it tells the deserializer the actual class type of the message being deserialized. Each message class implementing the IMessage interface has assigned a unique TypeID.

The whole header and the serialized IMessage object are together called TransportMessage in the context of FMLink and consist of a single array of bytes ready to be sent.

3.6.2 Transport Chunks

FMLink will often need to send very large message objects through the network. On the internet, the fundamental unit of data is the packet. These packets are often very small, in most conditions, these are only 1500 bytes (2304 B for WLAN, 1500 for Ethernet, 1492 for PPPoE). When sending any data through the Internet, the information will be divided into packets and then reconstructed on the other side. This applies to every network solution using the Internet. The trivial solution would be to use some existing protocol's ability to divide and reconstruct the data and just send the whole stream of bytes at once. This is exactly what TCP can do since it is a streaming protocol and it hides the concept of packets from the user.

During the design phase of FMLink, it was decided to send the data in smaller chunks, even if not required by the underlying network protocol. This brings one big advantage of greater control of sending and receiving data.

One such advantage is the ability to assign different priorities to messages. Let's say we have two messages to be sent, one very large (for example 3D model), but a low priority, and another one much smaller, but higher priority (text message). Using traditional TCP connection, if the user would first try to send the big, low-priority message, he would have to wait until it is finished sending in order to send the high priority message. In the context that FMLink operates in, this is unacceptable, since high-priority massages couldn't be sent very quickly.

FMLink splits the messages it wants to send into smaller chunks. These chunks should be the same size as MTU of the network in order to minimize protocol overhead. In computer networking, the maximum transmission unit (MTU) is the size of the largest protocol data unit (PDU) that can be communicated in a single network-layer transaction.[6]

Each chunk is a part of a serialized TransportMessage and consists of two parts – header data and actual payload data. The header has three fields: MessageID, ChunkID, and Lenght. Length is straightforward, it denotes the length of the chunk's data. MessageID is used so that chunks are correctly assigned to the right TransportMessage during the receiving process and ChunkID is used so that chunks are assembled in the correct order. Since the chunks have the order number inside them, they can even arrive in a different order they were sent in. TCP would not allow this, since it checks if the messages being sent are delivered in the correct order and it might require retransmission. UDP would not detect packets in the wrong order at all. FMLink is able to detect this and still use the data if the only issue is that packets are being delivered in the wrong order since it contains the Chunk and Message IDs and uses them to correctly reassembly the message.

3.7 Sending Messages

This section will discuss the process of sending the actual TransportMessages and TransportChunks. For this reason, two interfaces and their implementations were created: IFMLinkSocket and IFMLinkClient.

3.7.1 IFMLinkSocket

The main purpose of this interface is to send and receive small bundles of binary data. It was created as an interface (or abstract class in C++), in order to allow FMLink to be network protocol agnostic and to allow different implementations and easy unit-testing. Actual implementations are allowed to use any specific network protocol, such as UDP, TCP, or even something

higher-level such as ZeroMQ or WebSocket, as soon as the protocol allows sending and receiving binary data.

This interface is not responsible for ensuring the message actually gets delivered, since that is handled at higher levels of FMLink. It is also not responsible for splitting, merging, or serializing the data. However, implementations of this interface need to handle the creation and closure of the network connection if required by the underlying protocol.

The interface contains two public methods, Send and Receive. It is expected that chunks will be sent one by one in a blocking fashion, so the Send method should not return until it has finished transmitting the binary data. The same goes for the Receive method, which accepts the number of bytes as an argument and the method will return only when the requested number of bytes has been received.

Only one IFMLinkSocket instance will be used by the FMLink connection and it is the only point in the whole FMLink library where the actual platform-dependent network APIs are called, therefore IFMLinkSocket is a bridge between FMLink and network layer APIs available on a given platform.

3.7.2 IFMLinkClient

Interface IFMLinkClient was designed to create the actual instance of IFMLinkSocket object and to send and receive TransportChunk objects, for which it uses the created IFMLinkSocket object. It was designed as an interface, but in this case, the only reason to do so was to allow easy unit-testing so that the actual implementation can be swapped for a mock class.

FMLinkClient was designed to take TransportChunk instances, serialize them and send them through IFMLinkSocket. This process just serializes the TransportChunk header data (chunk size, ChunkID, and MessageID), prepends it to the TransportChunk binary payload, and forwards the whole data bundle to IFMLinkSocket. This process works similarly for the Receive method, which receives binary data through IFMLinkSocket. First, it tries to receive header data of a fixed size (3 integers – 12 bytes). This tells the FMLinkClient the length of the incoming chunk data, which then tries to receive the payload of the chunk. After the whole chunk including its header was received, this data is stored back in the TransportChunk instance. Similarly to IFMLinkSocket, the Receive method blocks until a TransportChunk is actually received and returned to the caller.

This interface also exposes interface methods for getting the connection state. IFMLinkClient can have four states - Created, Open, Failed, and Closed.

Every IFMLinkClient starts in the Created state. As soon as the IFMLinkClient obtains IFMLinkSocket instance with an open connection, it changes the state to Open and it signals that the client is ready to use. If the connection fails during creation or in any other phase,the client goes into Failed state permanently and new IFMLinkClient should be created (and thus a new

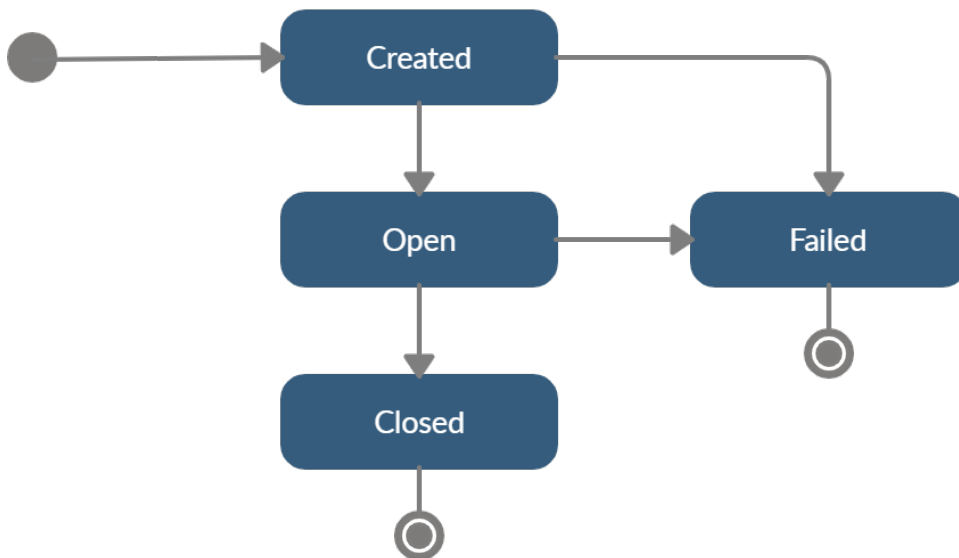

Figure 3.3: FMLinkClient connection state machine

connection opened). When the client is gracefully closed successfully, it goes into Closed state.

3.8 Channels

When sending messages through FMLink, the user might want to send some messages with high priority and some with lower priority. In order to enable doing that, the concept of channels was introduced to FMLink – FMLinkChannel. A channel is a virtual link that can be opened from one side of an already established communication and it will be opened on both sides. A channel is characterized by its name and channel ID. Channels were designed as an exclusive way to send messages, every message needs to be sent and received through some channel.

For this reason, every open FMLink connection always has one channel created on both sides by default. This channel has the name ”Main” and an ID of 0 and is always present. It is not visible or usable by the user, since its purpose is for sending and receiving messages created by the FMLink itself. It is used for commanding the other connected FMLink instance to open channels, starting/stopping of synchronizations, or to close the connection. Messages received from this channel are handled exclusively by FMLink and not passed down to the user.

Channels were introduced so that the user can send messages with different parameters. For now, only the priority parameter can be changed for the chan-

nel, however, in the future, channels might feature optional data compression, optional encryption, or automatic persistent storage of messages. FMLink architecture with channels allows me to easily implement these features later when needed.

Each FMLink connection has a list of currently open channels. This collection is managed by the ChannelManager class. Users of FMLink are able to open and close the channels. When a user opens the channel, a new FMLinkChannel instance is created and inserted into collection managed by ChannelManager. The other party is then notified (through the "Main" channel) about this new channel, so the other party also creates their FMLinkChannel instance.

FMLinkChannel is the main entry point for sending messages through FMLink. For this reason, FMLinkChannel was designed with a method to send an IMessage instance and an event that notifies the user that some message has been received on that particular channel from the other party. However, when the user calls the SendMessage method on FMLinkChannel, the message is not sent immediately. Instead, it is serialized (into the TransportMesage object), placed in the channel's outgoing buffer queue, and waits for further processing.

3.9 FMLinkQueue

FMLinkQueue is a class designed as a bridge between FMLinkChannels and FMLinkClient. The main problem this class was designed to solve was that there can be many FMLinkChannels created but there should always be just one FMLinkClient for a given connection, which means that channels have to compete for the outgoing connection. FMLinkQueue decides, which channel should be able to send its message – this message is already serialized and in the form of a TransportMessage, which contains the serialized IMessage data.

The process starts with FMLinkQueue deciding, which channel should be selected for transmission, depending on various criteria. It then takes the front TransportMessage of the selected channel's outgoing message queue. Each TransportMessage object keeps track of how much of it was transferred. Taking this information into consideration, FMLinkQueue takes a TransportChunk out of this message, forwards it to IFMLinkClient and upon successful transmission, it marks the given TransportMessage that the aforementioned chunk was transmitted. If the TransportMessage is fully transmitted, it is popped from the outgoing message queue of the channel. This process then repeats again in an infinite loop.

It must be noted that a different channel might be chosen for transmission, even if its front message is not yet fully transmitted, with some chunks still remaining for transmission. Since the chunks are small, this allows the FMLink to quickly transmit messages from high priority channels even if

other messages haven't finished transmitting yet. This design also allows for a pseudo-simultaneous transmission of multiple messages at once on a single(TCP) connection.

3.10 FMLinkSession

Since all the basic building blocks of FMLink design have been described, FMLinkSession can be introduced. It is the main point of contact for the user to use the FMLink API. It exposes most of the main methods the user will need, so most of the architecture, such as clients, sockets, and queues can be hidden from the user(private). Since the target of FMLink is to provide synchronization and messaging services, it's interface reflects this. It exposes methods for channel creation and retrieving and methods supporting synchronization.

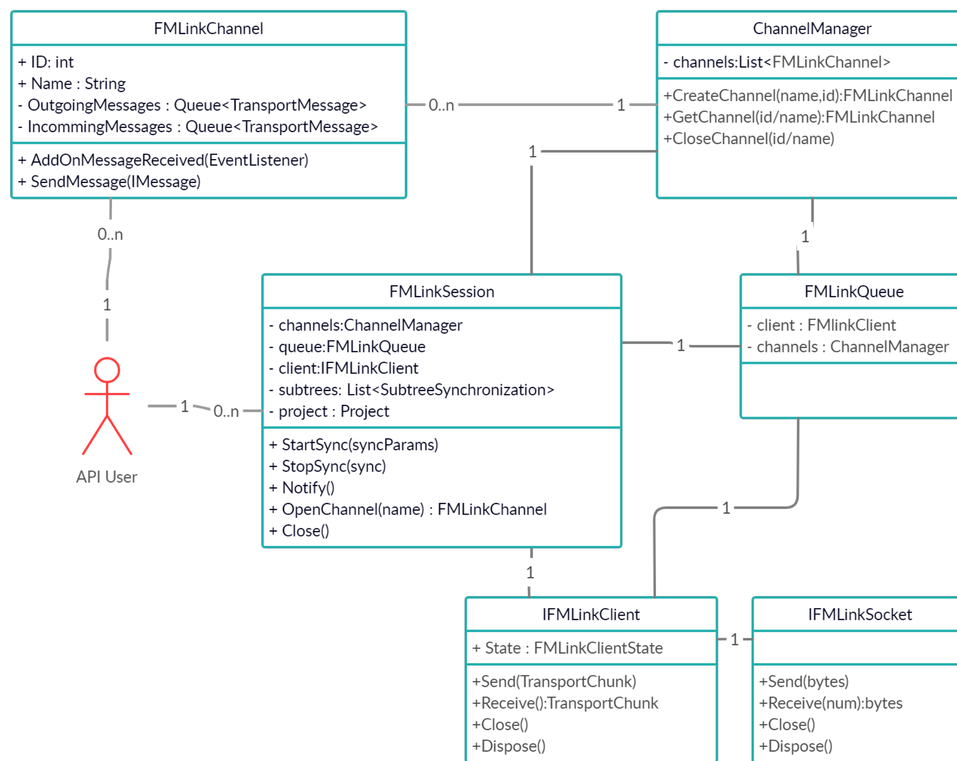

Figure 3.4: Simplified internal structure of FMLink

FMLinkSession represents an active FMLink connection. It is built as a central point for FMLink and contains references to FMLinkClient, FMLinkQueue, and all the FMLinkChannels. It manages these objects and it

also handles the synchronization services – it keeps a list of trees marked for synchronization and executes commands that keep these trees synchronized.

To recapitulate previous sections, I will show what happens when a user tries to create a channel and send a text message.

1. User will request a new channel from FMLinkSession.
2. FMLinkSession will inform the other party a new channel has been created and provides the user with the new FMLinkChannel.
3. User will create a new instance of a message with the IMessage interface, that contains the data he wishes to send
4. User sends this message to the provided FMLinkChannel
5. FMLinkChannel serializes this message (by calling the Serialize method of the IMessage interface), bundles it in the TransportMessage object and adds this message to its queue of outgoing messages
6. When it gets its turn, FMLinkQueue will take this message from the front of the channel's queue and it will send a chunk from this message through FMLinkClient
7. FMLinkClient adds chunk header data to the chunk binary data and forwards the final binary data (of the mentioned small size, approximately 1500 B) to the IFMLinkSocket
8. IFMLinkSocket transmits the binary data via the selected network interface (TCP, UDP...)
9. This repeats until all the chunks of the message are sent

Chapter 4

Implementation

4.1 Platforms

Since FMLink was needed to be developed for two platforms, Universal Windows Platform and .NET, the implementation process needed to reflect this. The primary development started on the .NET platform since I was personally most experienced with this platform and it was easiest for me to prototype and develop on. After the implementation of the .NET version in C# has reached a phase where basic functionality was working, I started porting the .NET version to UWP, using WinRT/C++. WinRT/C++ will be from now on be referenced only as C++ since it is based on the C++17 language standard.

4.1.1 Memory managemenent

The largest difference between .NET and C++ that I encountered during the development phase, is that .NET provides garbage collection, while in C++, the programmer needs to manage the program memory.

> *".NET's garbage collector manages the allocation and release of memory for your application. Each time you create a new object, the common language runtime allocates memory for the object from the managed heap. As long as address space is available in the managed heap, the runtime continues to allocate space for new objects. However, memory is not infinite. Eventually, the garbage collector must perform a collection in order to free some memory. The garbage collector's optimizing engine determines the best time to perform a collection, based upon the allocations being made. When the garbage collector performs a collection, it checks for objects in the managed heap that are no longer being used by the application and performs the necessary operations to reclaim their memory."[7]*

In order to simplify the porting process, smart pointers from the C++ language standard were chosen as the best solution to imitate the architecture of FMLink in .NET. In C#/.NET, object instances of classes are always stored as references. To simulate this, everywhere in the .NET version where there was a reference to an object, a shared_ptr was used instead containing the object. This approach worked most of the time, except for cyclical references. A shared_ptr uses reference counting to decide when to delete the referenced object :

> *"std::shared_ptr is a smart pointer that retains shared ownership of an object through a pointer. Several shared_ptr objects may own the same object. The object is destroyed and its memory deallocated when either of the following happens:*
>
> *the last remaining shared_ptr owning the object is destroyed; the last remaining shared_ptr owning the object is assigned another pointer via operator= or reset(). The object is destroyed using delete-expression or a custom deleter that is supplied to shared_ptr during construction."[8]*

This works well for most of the scenarios, except when two objects reference each other. While the garbage collector in .NET can detect this and remove these objects when nothing else references them, a solution using shared_ptr would never delete this objects. This conflict of resolved by making one of the references weak_ptr and the other one shared_ptr. This way the reference counting cycle is broken because weak_ptr isn't using reference counting.

A smart decision needs to be made in every case, which one is going to be shared_ptr and which one weak_ptr. In most cases, this can be quite simple, since one object usually creates the other, in which case the parent object should be shared_ptr.

4.1.2 Interfaces

Up until now, I have been referring to IMessage and others as interfaces. However, only C# supports interfaces, C++ doesn't have a direct language construct to represent them. Since interfaces are a very common design construct, C++ supports them via abstract classes. These classes are defined when at least one of the class methods is defined as a pure virtual method with no implementation. When a class wants to implement this interface, it just extends this class. This is enabled by the fact that as opposed to C#, C++ supports multiple inheritance and thus a C++ class can implement multiple interfaces.

4.2 Usage example

To better illustrate how the final implementation of FMLink is used, a code example can be seen in figure 4.1. The client first specifies the IP address and port, at which the server listens for incoming connections. ClientConnectionHandler then creates a new FMLinkSession and returns it to the user.

```
//create a project, the tree contains one child
//node named "foo", with empty attribute "bar"
var project = new Project();
project.Root.EnsureChild("foo").SetAttributeDirect("attr","hello_1");

//connect the client to localhost server
var clientConfig = new ClientConnectionConfiguration()
{
    RemoteAddress = "127.0.0.1",
    RemotePort = 5556,
    UseTLS = false
};
var clientSession = ClientConnectionHandler.Connect(clientConfig);

//start synchronization of the "foo" subtree
clientSession.SetProject(project);
await clientSession.StartSyncProjectAsync(new SynchronizationParams
  ("/foo", "/foo"));

//change attribute of the node and request sync to server
await project.ExecuteAsync(true, (p)
    => p.Root.GetChild("foo").SetAttributeDirect("attr","hello_2"));
await clientSession.NotifyAsCommand();

//open a channel a send a text message to client
var channel = clientSession.OpenChannel("messages");
channel.MessageReceived += ((sender,arg) => Console.WriteLine
  ("Received message"));
channel.SendMessage(new ASCIITextMessage("Hello from client !"));
```

Figure 4.1: Example usage of FMLink for .NET

The user then specifies the target Project and Node paths for synchronization. FMLink adds these to its list of synchronized subtrees. After that, the user modifies the Node attribute and Notifies the FMLink of the change. Note, that the user doesn't need to specify what exactly changed, since FMLink can automatically detect the scope of change.

Another functionality the example demonstrates is the sending of messages. The user requests a new channel ”messages”. FMLink then notifies the server of this channel and the channel is opened and ready to be used on both sides.

```
#include "pch.h"
#include <string>
#include <iostream>
#include "../FMLink/FMLink.h"

using namespace winrt;
using namespace std;
using namespace FMLink;

int main()
{
    //Connect to localhost server
    FMLink::ClientConnectionConfiguration clientConfig(L"127.0.0.1", L"5556");
    shared_ptr<FMLinkSession> clientSession =
      FMLink::ClientConnectionHandler::connect(clientConfig);

    //Open channel "messages"
    ChannelRef channel = clientSession->open_channel(L"messages");
    channel->add_message_received_listener(on_message);

    //send message in the "messages" channel
    FMLinkMessageRef mes = make_shared<ASCIITextMessage>("Hello from C++ !");
    channel->send_message(mes);
}
void on_message(ChannelRef channel, FMLinkMessageRef message)
{
    shared_ptr< ASCIITextMessage> txtMessage =
      dynamic_pointer_cast<ASCIITextMessage>(message);
    wcout << L"Received message : "<< txtMessage->get_txt();
}
```

Figure 4.2: Example usage of FMLink for UWP

The user then sends a text message by calling the ”SendMessage” method on the provided channel. If a server sends a message via this channel, a ”Received message” text is displayed in the console window.

4.3 FMPatchableArchive

In order to support the new functionality of creating ”patch” updates for the tree, a new version of the FMBinaryArchive was introduced. The V2 format makes use of the Node start marker in the V1 format in order to add the new features. It adds a new Node delete marker, which instructs the deserializer to remove the marked Node from the tree. In order to remove an attribute from a Node, a special new value type ”Delete” was created, which in turn instructs the deserializer to remove the attribute from the Node. It doesn’t matter that the actual value type of the deleted attribute is not specified, since Nodes cannot have attributes with duplicate names.

4.4 Messages

FMLink can send various messages that include the IMessage interface. Some messages can be of an internal character, only the FMLink itself is expected to send and use them and they are sent over the ”Main” channel. These messages inform the other party of the synchronization trees, channel creation/destruction, etc. In order to simplify the handling of these messages on arrival, they not only include the data they transfer, but also an action that needs to be performed when they arrive. For this purpose a new interface has been created, IExecutableMessage, that extends the IMessage interface and adds one more method, Apply(). When the message arrives, FMLink will call the Apply method and supply the FMLinkSession object as a parameter. This means that messages implementing this interface have access to the whole internals of FMLink and can manipulate them.

4.4.1 Synchronization Messages

When a user requests a synchronization of some subtree from FMLink, the other connected party needs to be informed of this. For this purpose, StartSynchronizationMessage and StopSynchronizationMessage message types were implemented. They contain the local and remote addresses of the subtree to be synchronized. These messages are executable, when a Start message arrives, it creates a new entry in a list of synchronized subtrees. The Stop message does the opposite, it removes the entry from the list of synchronized subtrees from the FMLinkSession.

4.4.2 TreeDiffMessage

This type of message is sent by the FMLink each time the user calls the Notify method. The Notify method will search for things that changed in the subtree marked for synchronization and if any changes are found, they are serialized into FMPatchableArchive and send a part of TreeDiffMessage. This message also implements the IExecutableMessage interface. On arrival at the other side of the connection, it is executed and the serialized tree patch is applied to the local copy of the subtree a therefore it synchronizes the contents of these two subtrees.

4.4.3 Channel Messages

For the purpose of channel creation and destruction, ChannelAddedMessage and ChannelRemovedMessages were created. When one party opens or closes a channel in their FMLinkSession, FMLink creates an instance of this message and sends it to the other party, informing it of that action. For that, these messages also implement the IExecutableMessage interface, so that they can be executed and create/destroy a channel on arrival.

4.4.4 Data Messages

For the purpose of transferring data from the user, ASCIITextMessage and BinaryDataMessage were created. They are only containers for the data and therefore are not executable. When the user wants to send a text message via FMLink, he needs to create a new ASCIITextMessage instance, initialize it with the desired string contents and send it via some existing channel. ASCIITextMessage and BinaryDataMessage messages are currently the only message types accessible by a user and not sent by FMLink internally.

4.5 Synchronization

For synchronization purposes, most of the synchronization orchestration is handled by the FMLinkSession class. For this purpose, three methods were implemented, StartSync, Notify, and StopSync.

4.5.1 Start/Stop Sync

Each FMLinkSession holds a list of subtrees that need to be kept synchronized. Each subtree marked for synchronization is represented as a SubtreeSynchronization object. This object contains the Shadow copy of the Node tree, local and remote addresses in the Project tree where subtree is located in, and an FMLinkChannel instance. Every time a user starts synchronization by calling the StartSync function, a new SubtreeSynchronization instance is stored inside a list of synchronized subtrees and StartSynchronizationMessage is sent to the other party. A new channel is also created for this specific subtree. This means that if the user synchronizes multiple subtrees, multiple channels will be created. Since channels can have different priorities, synchronization of different subtrees can also have different priorities.

4.5.2 Notify

The Notify function is called every time a user wants to update the synchronization of the selected subtrees. The function goes over a list of SubtreeSynchronizations and checks for changes in each of them using DiffUtils. When there is a change detected in the subtree, the serialized archive produced from the DiffUtils is sent as a TreeDiffMessage to the other party.

4.5.3 Detecting change

For detecting the change in a subtree and serializing it into a patch, DiffUtils provides the function WriteDiff. This function creates an FMPatchableArchive and recursively iterates through each Node of the subtree and compares it to the Shadow tree. If a Node is found that is missing from the Shadow tree, this Node is added to the archive and the same goes for attributes. If there is a

node present in a Shadow tree but not present in the current Node tree, this Node is considered to be removed by the user and written as deleted to the archive. The new FMPatchableArchive format supports this, since it not only has markers for adding nodes/attributes as the old format did, but it also has markers for node/attribute deletion and attribute change.

In order to deserialize the FMPatchableArchive, the DiffUtils class provides the ReadDiff class. This class reads the archive a applies the changes described in the archive to the user's Node subtree. In the process, it also changes the Shadow copy of the subtree to reflect the changes.

4.6 Interaction with Project

As mentioned before, every operation using the Project Node tree should use the Project's execution engine. This engine provides the Execute method, which takes a function delegate and places it in the command queue. When ready, this function is executed and the function can interact with the Node tree. The Execute function can be called as exclusive or non-exclusive. When the command is marked as non-exclusive, it is expected that it will execute only read-only operations on the Node tree. This allows the execution engine to run multiple non-exclusive commands in parallel since no data race can happen and this can increase performance.

During the development of FMLink, one downside of the execution engine was encountered – it is not known by default, when the command actually starts execution and when the command fully executes. Since FMLink needs to wait for the execution of some of the commands, a new function was introduced, ExecuteAsync. This new function returns the user a Task object immediately and when the command actually completes, the Task object is notified. This means that the user just needs to await this Task. Both C# and C++17 support asynchronous Task programming in the current versions. In the C++ implementation, the IAsyncAction interface from the Windows Runtime was used to facilitate asynchronous programming.

4.6.1 Multiple variants of API

Some of the public methods of FMLinkSession class are expected to be changing or reading the contents of the Project tree and there this code needs to be executed inside a Project Command. Two main functions that require this are Notify and StartSync. The issue with running the Commands is that they cannot be executed and waited on inside another Command, since that would cause a deadlock scenario. And when running some code, it is currently impossible to tell if the code is already running as a Command.

For this reason, the Notify and StartSync functions have three variants. The first variant NotifyAsCommand creates it's own Command and executes the Notify function inside it. It returns an awaitable task that tracks the

progress of this Command, so the function returns immediately and it cannot be run inside another command. Since this method returns a Task, it can be waited on until the Notify function is completed.

The second variant is NotifyUnsafe. The Unsafe postfix suggests that as opposed to NotifyAsync, this function doesn't create its own Command but assumes it is already running inside one. Therefore, this method must be called from inside the Command.

The third variant is Notify (without any prefix or postfix). This variant also creates a new Command, but it doesn't wait on it and returns immediately. Although this version has a benefit that it can be called from both the inside and outside of the Command, it cannot be waited on and therefore it is harder to tell when the function has completed the execution. One solution is to schedule a Command right after the Notify method call. As soon as the scheduled command is executed, we know, that Notify has been already executed, since the Execution engine respects the order of Commands.

All three variants are also created for the StartSync function since it also accesses the Project tree, similarly to the Notify function.

4.7 Implementing the IFMLinkSocket and choosing network layer

At the lowest level, the decision needed to be made of which network protocol to use. The first proposal was UDP.

4.7.1 UDP

UDP offers many advantages over TCP which the FMLink could use, such as lower protocol overhead.

> *"The main difference is that UDP doesn't require the recipient to acknowledge that each packet has been received. Any packets that get lost in transit are not resent. This enables computers to communicate more quickly, but the data received might not exactly match the data sent.*
>
> *UDP packets don't have sequence numbers, so they can arrive out of order. They do have checksums, though, so the packets that do arrive are protected against corruption or modification in transit."[9]*

However, UDP also brings many disadvantages. The most common difference between UDP and TCP people remember is that UDP is unreliable and fragments can arrive out of order. Both of these are solved by using the TransportChunks, so this is not an issue for FMLink. The much larger issue I have encountered was that UDP doesn't handle congestion control – this

means that the user needs to control how fast he sends UDP datagrams to the network. If he sends them too fast, a lot of them will get lost, if he sends them too slowly, the full capability of the network will not be used. I have discovered that congestion control algorithms are too complicated for the scope of the project and after a long discussion in the company, the decision was made to use only TCP.

4.7.2 TCP

TCP was chosen instead of UDP mainly because it already includes a network congestion-avoidance algorithm. This means it was possible to develop FMLink much faster since TCP already offers many services that FMLink would need to reimplement if it would use UDP. However, since IFMLinkSocket is an interface and any implementation can be used, it is possible to add support for UDP later in the future, if it's deemed worthwhile.

TCP also satisfies the reliability requirements of FMLink. It includes checksum, which prevents the possibility of receiving a corrupted piece of information. It also includes out-of-order preventions and automatic retransmission of lost data.

4.8 Establishing connections

TCP is a connection-oriented protocol and it requires one party to act as a client and one party to act as a server to establish a connection. This means that FMLink inherits this trait and when establishing a connection over TCP, one instance of FMLink needs to act as a server and listen for incoming connections and one needs to act as a client and connect to a server. This behavior is implemented in ClientConnectionHandler and ServerConnectionHandler classes.

The ServerConnectionHandler contains only a single static public method "AcceptClientSession". This method creates a new instance of FMLinkClient that creates and starts a TCP listener on the desired IP endpoint based on provided arguments. When a client connects to this endpoint and TCP connection is established, FMLinkClient contains a connected FMLinkTCPSocket (TCP implementation of IFMLinkSocket) instance. A new instance of FMLinkSession is then created, with the FMLinkClient as a parameter. This instance is then returned by the AcceptClientSession method and is considered to be initialized and ready to be used.

The ClientConnectionHandler works similarly, but instead of listening, the FMLinkClient tries to connect to a TCP listener on a given IP endpoint. After the connection is successful, ClientConnectionHandler also returns an initialized instance of FMLinkSession.

It is important to mention, that although for establishing a connection, one FMLink instance needs to assume a role of a server and one assumes a

role of a client, after the connection is established, both FMLink endpoints are equal and no longer categorized as server/client.

4.9 Security

From the functional and non-functional requirements, a need to encrypt the actual network traffic emerged. I considered several security solutions, but in the end, I decided on Transport Layer Security (TLS), the most standard and probably the most secure standard available.

The TLS standard is widely supported on all major platforms and operating systems. It handles all the encryption automatically and the application using TLS doesn't have to choose the specific encryption to be used. When opening a connection between a client and server, the TLS protocol automatically checks for the strongest encryption methods on both sides and uses it. This means, that in the future, if the currently used cipher becomes compromised, TLS will stop using it and customers will only need to update their operating system or runtime in order to keep the system secure.

The actual implementation used in FMLink doesn't use a specific version of TLS – it automatically uses the newest version available for the runtime. As of the time of writing, .NET core build of FMLink was using TLS version 1.2 but it is expected that during the following year, the TLS version should automatically get updated to version 1.3 with advanced security features without any additional work needed on FMLink.

The implementation of TLS connection handling is in class FMLink-TLSSocket, which implements the IFMLinkSocket interface. This means that FMLinkClient can easily use secure TLS communication just by using a different implementation of the IFMLinkSocket interface.

When the TLS handshake was inspected using Wireshark, FMLink was using Elliptic-curve Diffie–Hellman algorithm for key exchange and then AES with a 256-bit key to encrypt the TCP traffic. These security standards are currently considered highly secure and they are supported by all major operating systems and platforms. AES encryption can be hardware-accelerated on most modern CPUs, thus adding only very little CPU overhead.

In order to prevent man-in-the-middle (MITM) attacks, FMLink uses certificates for TLS authentification. Each computer that is desired to be an FMLink server, needs to own a private/public key pair to enable TLS authentification. If the certificates are self-signed, they also need to be installed on each client device, so the client devices recognize and trust the public certificate of the server. The self-signed certificated can be generated using the following Powershell script :

```
New-SelfSignedCertificate -CertStoreLocation
Cert:\LocalMachine\My -DnsName "HOST_NAME"
-FriendlyName "MySiteCert"
```

```
-NotAfter (Get-Date).AddYears(10)
```

The DnsName parameter needs to have a real hostname as a value. This hostname can be easily obtained by running the ”hostname” command in Windows Powershell. When the client connects to the server, it will compare the computer hostname and the hostname from the provided public certificate, which must match, otherwise, the TLS handshake will fail.

In order to not require the installation of every server certificate on each client, the server must be signed by some certificate authority (CA). This authority can be created internally by the company. In that case, only the CA certificate needs to be installed on clients and each client will automatically trust every server certificate signed by the company's CA.

Another common solution is to get a certificate from some well-known certificate authority. These kinds of certificates are usually a paid option. Since public certificates of well known CA's are already pre-installed in certificate stores of most operating systems, the clients will not need installation of any additional certificates.

Since using TLS for key-exchange and encryption is the most standard way of securing communication, it was easiest to integrate with FMLink. In the future, when FMLink will be tested for security by outside auditing, the use of standardized security solution will greatly simplify this process.

Chapter **5**

Testing

5.1 Wireshark

During the implementation and testing phase of FMLink, various network analyzing tools were used to debug the network protocol. One of them was Wireshark, a highly popular network analyzing tool.

> *"Wireshark is the world's foremost and widely-used network protocol analyzer. It lets you see what's happening on your network at a microscopic level and is the de facto (and often de jure) standard across many commercial and non-profit enterprises, government agencies, and educational institutions. Wireshark development thrives thanks to the volunteer contributions of networking experts around the globe and is the continuation of a project started by Gerald Combs in 1998."[10]*

However, since FMLink is a binary format, vanilla Wireshark was not very useful for inspecting the contents of FMLink packets, it was even hard to find FMLink communication in the huge amount of traffic that usually flows through a network adapter. Luckily, Wireshark is open-source and expandable via its plugin system.

Wireshark uses the concept of dissectors. Dissector is a component that parses and extracts information from the relevant part of the network frame – each frame can be parsed by several dissectors at different layers. The dissector at the application level was needed since FMLink works at that level.

There are two major ways of developing Wireshark plugins/dissectors – native plugins and Lua plugins. In order to develop a native plugin, it is required to download the whole Wireshark source code and learn how to compile it. This proved to be an overly difficult task, so an easier solution was required. I personally am not too familiar with Lua scripting, so this option was also not viable. Finally, there is a third, less common option of writing Wireshark dissectors – WSGD.

5.1.1 Wireshark Generic Dissector

Wireshark Generic Dissector is a plugin developed for the easy creation of Wireshark dissectors without programming in C++ or Lua. WSGD instead uses a descriptive approach when parsing network protocols. It requires the protocol to be described using the WSGD syntax and uses this description to parse relevant information and display it in the Wireshark Interface.

It must be noted that WSGD has one large downside, it has a very small and chaotic documentation. A trial and error approach had to be used in order to write a very basic dissector for FMLink.

Each dissector written for WSGD needs two files – one with .wsgd extension and one with .fdesc extension. As far as I understood from the context, the .wsgd file provides metadata information and basic structure, while .fdesc file is used for actual parsing of the protocol data. The .fdesc file format is a custom format of WSGD, however, some parts of it look like C++ language. It uses a concept of data structures, where each field has a defined data type. This is an excerpt from the .fdesc file for the FMLink protocol :

```
struct  T_msg_header_type_wsgd
{
    byte_order big_endian;
    uint32 message_id;
    uint32 chunk_id;
    uint32 chunk_length;
    uint32 channel_id;
    T_FMLink_msg_type Type;
    hide var uint32 real_size = chunk_length-6;
    hide var string     Msg_Title = print ("message id = %d,chunk id = 0x%x, size = %d,
        channel = %d, type = %s",  message_id,chunk_id,real_size,channel_id,Type);
}
```

Figure 5.1: Part of .fdesc FMLink format

Currently, this dissector properly dissects only FMLink messages that fit into one frame. If the message is divided into multiple chunks, this dissector will properly parse the chunk ID and length, but other information will be parsed incorrectly. It would be possible to write a proper dissector in WSGD, but since the documentation is very insufficient, this would take a lot longer and in the end, it might be easier to just write a native dissector.

However, for the testing purposes of FMLink, even the basic dissector created has proven to be highly useful, since it displays the message and chunk IDs of the messages and the type of messages, as opposed to raw binary data that was originally displayed. For some message types, it even displays the parsed message content.

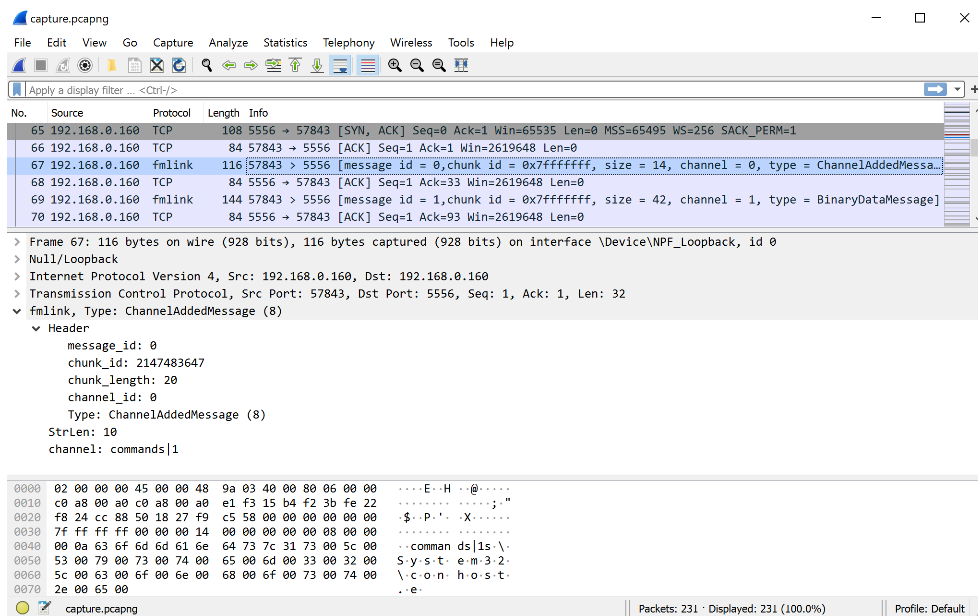

Figure 5.2: Wireshark Interface with FMLink plugin

5.2 Creating various test conditions

One of the basic objectives of FMLink is resiliency to degraded network conditions since the Fata Morgana platform will be deployed in environments, where the devices might be connected to each other via an unreliable network. Wi-Fi wavebands are highly vulnerable to disruption since they operate at 2.4 and 5 GHz. These radio frequencies are populated by many other wireless protocols, which often compete for these radio bands. They also have issues with penetrating solid objects such as walls. In this case, 2.4 GHz usually has a longer range and can more easily pass through walls, but at the cost of lower network throughput than 5 GHz Wi-Fi and higher radio frequency occupancy.

These wireless network conditions are hard to replicate and it would be hard to repeatedly and deterministically test FMLink for these conditions. However, software proxy servers that can emulate these conditions exist and can be used. The most common issues with these network connections that could affect FMLink and need to be emulated are delay and packet loss scenarios.

5.2.1 Test environment

In order to perform the tests, a testing environment was required to be set up. In this case, these tests are going to be performed manually. In the future, automated tests will be developed to automatically test the code.

The testing environment consists of two computers, both connected in one

Machine	Lenovo ThinkPad P52	Dell Precision T1600
Type	Laptop	Desktop Workstation
CPU	Intel i7-8750H 2.2-4.1 GHz, 6-Core/12 Threads	Intel Xeon E3-1270 3.4 GHz, 4-Core/8 Threads
Memory	24 GB, DDR4	8 GB, DDR3
OS	Windows 10, Build 18362	
.NET	.NET Core, Build 3.1.202	

Table 5.1: Test Machine Configurations

network. One computer was a Lenovo Laptop and the second one a Desktop Workstation. The exact specifications are listed in table 5.1.

These two computers were then connected to the TP-LINK TL-SG108 gigabit network switch via Gigabit Ethernet. This switch was then connected via Gigabit Ethernet to the TP-LINK Archer C1200 Wi-Fi router, which was connected to the Internet. The drawing of the network can be seen in figure 5.3. Since the router has four LAN ports and the test scenario requires only two LAN ports (for the laptop and desktop), the switch might seem unnecessary. The reason for the inclusion of the switch between the devices and router was to enable faster and more reliable communication between the desktop and the laptop. I have had issues with fully saturating the gigabit network interface in the past when communicating on the local network since the router doesn't have enough processing power. The deployment of the switch helped to resolve these issues and allows the traffic to flow only via the switch and outside the slower router.

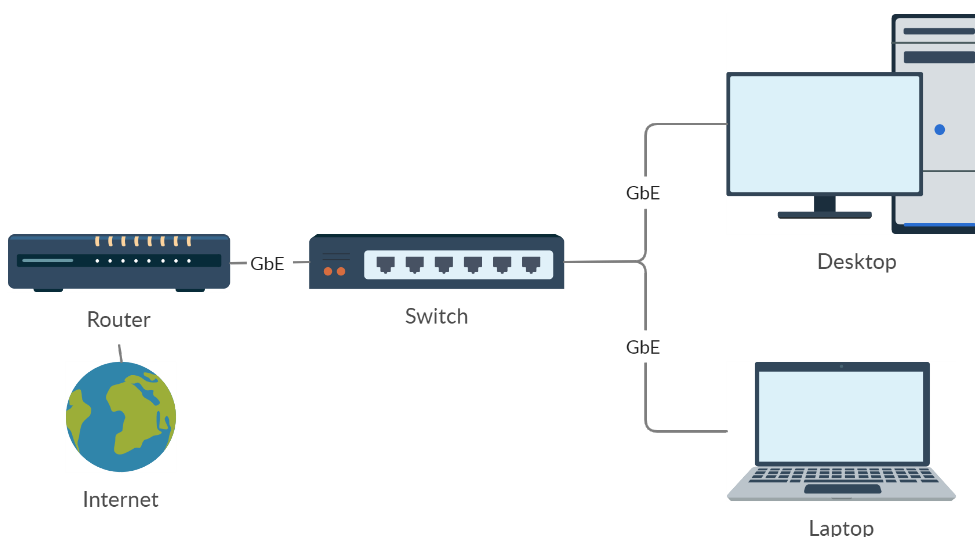

Figure 5.3: Scheme of the test network

5.2.2 Testing the environment

To test the quality of the network connection between the two computers, psping utility from the company sysinternals was used. This networking utility allowed me to easily measure latency and bandwidth of the connection between two computers. An instance of the psping program was executed on both computers, one as a server and one as a client.

5.2.2.1 Bandwidth test

For the first network test, network bandwidth was measured. The utility was configured to send 1000 requests, each with a size of 4 MB. These requests were configured to be using TCP protocol. When executed, an average speed of 113.9 MB/s was measured, which accounts for 911 Mbit/s of TCP data. After accounting for the protocol overhead, the OS was reporting an average transfer rate of 980 Mbit/s, which confirms that the connection can indeed deliver a gigabit per second of bandwidth.

```
PS C:\psping> .\psping -b -l 4m -n 1000 -h 20 192.168.0.149:1234

PsPing v2.10 - PsPing - ping, latency, bandwidth measurement utility
Copyright (C) 2012-2016 Mark Russinovich
Sysinternals - www.sysinternals.com

Setting warmup count to match number of outstanding I/Os: 16
TCP bandwidth test connecting to 192.168.0.149:1234: Connected
1016 iterations (16 warmup) sending 4194304 bytes TCP bandwidth test: 700697100%

TCP sender bandwidth statistics:
  Sent = 1000, Size = 4194304, Total Bytes: 4257218560,
  Minimum = 99.54 MB/s, Maximum = 120.48 MB/s, Average = 113.90 MB/s
```

Figure 5.4: Output of the bandwidth test

5.2.2.2 Latency test

The second network test was measuring the latency when sending a message from one computer to the other and back, counting the time of the whole round-trip. The utility was configured to send 1000 requests over the TCP protocol. The utility measured a minimum latency of 0.14 ms, maximum of 0.66 ms, and an average of 0.24 ms. Psping also allows to print the measured data as histogram buckets. This data has been plotted in figure 5.5.

5.2.3 Emulating network conditions

For the purpose of emulating degraded network conditions, various software solutions were tested. These solutions offer different functionality and often work very differently. I will name a few of the solutions that were tested and used.

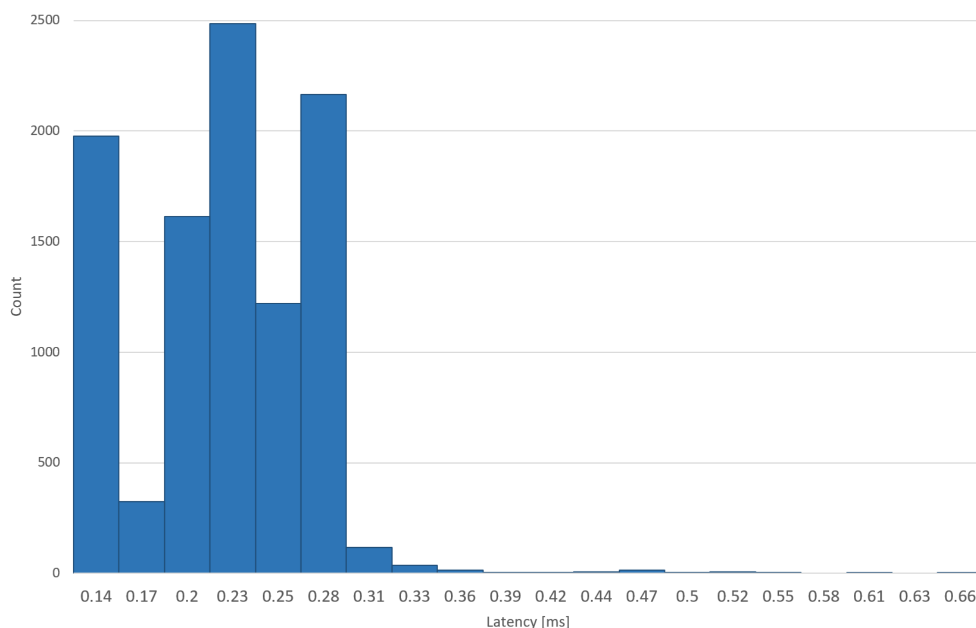

Figure 5.5: Network latency histogram

5.2.4 Toxiproxy

Toxiproxy is a proxy server developed by Shopify to test for different network conditions.

> *"Toxiproxy is a framework for simulating network conditions. It's made specifically to work in testing, CI and development environments, supporting deterministic tampering with connections, but with support for randomized chaos and customization. Toxiproxy is the tool you need to prove with tests that your application doesn't have single points of failure."[11]*

This proxy server can be run without root access to the system, doesn't have to be installed, and is multiplatform, with binaries available for Windows and Linux. This is important so that Toxiproxy can be easily integrated into existing testing environments. It can be operated via an HTTP interface, so it can be easily manipulated from the test environment.

Toxiproxy is a TCP proxy and allows to manipulate mainly delay and bandwidth of the connection. Unfortunately, it doesn't have the capability to simulate a random packet loss scenario. Another issue with Toxiproxy was limiting the bandwidth, which accounted only for TCP payload data. When sending a lot of small frames over Ethernet, the headers from the lower layers of the network (Ethernet, IPv4) could add a lot of data. This meant that when the bandwidth was limited to a certain amount, the actual bandwidth over

the wire was larger and variable since this proxy doesn't account for the lower layers of the network. Because of this limitation, it was impossible to emulate a network connection with fixed bandwidth.

Toxiproxy was initially integrated into the test toolkit, but since it doesn't support packet loss emulation, it had to be paired with another solution that adds that capability. Since other solutions that offer packet loss emulation also offer bandwidth and delay emulation, Toxiproxy was redundant and it was removed from the test toolkit.

5.2.5 Clumsy

As opposed to Toxiproxy, Clumsy works as a transparent or invisible proxy – the client is not aware of such proxy and doesn't require a special configuration of the client. Clumsy leverages the WinDivert library, which "allows user-mode applications to capture/modify/drop network packets sent to/from the Windows network stack"[12]. This means that Clumsy affects not only the TCP layer but works on the packet level of the Windows network stack, which gives it more capabilities than a TCP proxy. It is able to add time delay to the packets and randomly drop them. It is also possible to specify a probability of dropping a packet, a setting, which is useful when testing network applications.

Clumsy was originally used in testing the FMLink protocol for both delayed and lossy connection, but I have discovered a big issue with its packet delaying capabilities. Although it was possible to set an amount in milliseconds by which each packet should be delayed, I have discovered that this amount is highly imprecise, as opposed to other software products and it was delaying packets by a higher amount than requested. In order to get more precise data, I had to use a different software product. Clumsy was kept for the tests because it was the only solution I tested that could reliably drop packets.

5.2.6 TMnetsim

TMnetsim is a software product very similar with its capabilities to Clumsy. Compared to Clumsy, it also allows the user to delay the packets, but it is much more accurate. It works as a proxy server, so the client application needs to be aware of this and needs to connect to a proxy, instead of the server. The proxy server then forwards these packets to the FMLink server on another computer. When specifying the delay for the packets, it can also specify the jitter parameter, which is variability in delay. It can specify the statistical distribution of this variability to Gaussian, Normal, and Markovian. Since FMLink doesn't need to be tested extensively for jitter, only a brief jitter test with Normal distribution will be performed.

5.2.7 NetBalancer

"NetBalancer is a Windows application for local network traffic control and monitoring". NetBalancer is a complex product with many capabilities, but for my use case, only the network monitoring and limiting capabilities were used. NetBalancer is able to monitor and limit traffic on a per-application basis. This allowed me to set a network bandwidth limit for a certain application and not worry that some other application running (such as Spotify, Steam, or Windows Update) might take this bandwidth from the test application and invalidate the test results.

NetBalancer also works directly with the Windows networking stack, and so the set bandwidth limit represents the actual bandwidth flowing through the wire (other applications were often displaying incorrect readings). In the following tests, NetBalancer will be used to limit the network bandwidth an application can use and also for monitoring the amount of network traffic.

5.3 FMLinkTester

For the purpose of testing various test scenarios, FMLinkTester was developed. It is a very simple command-line program, that can be run as a server or as a client. This program can recreate two simple scenarios and it can test, how the FMLink protocol behaves in problematic network conditions with the use of previously mentioned programs Clumsy, TMnetsim and NetBalancer.

5.3.1 Ping-Pong Test Scenario

For the purpose of testing scenarios, where two clients are connected to each other over a link with very high delay, I developed a very simple protocol, built using FMLink and integrated into the FMLinkTester. When executed, one instance of the application needs to be run as a server and one instance needs to be run as a client. After a server starts listening for incoming FMLink connections, the client should connect to the server and the protocol communication can start.

The client first opens an FMLinkChannel and sends a text message to the server with the text "Ping:{counter}" over this channel, with the counter set to initial value of 1. When the server accepts the ping message, it extracts the counter number and sends back a text message "Pong:{counter}" with the same counter number. When the client receives the pong message, it checks if the counter number is valid. It then increments the counter value and sends another ping message. This repeats in an infinite loop.

In a separate thread, a monitoring loop is executed. This method monitors the counter number in a specified time interval (in this case 5 seconds), which allows it to compute the rate of ping-pong cycles per second. Each cycle is one ping message from the client and one pong response from the server.

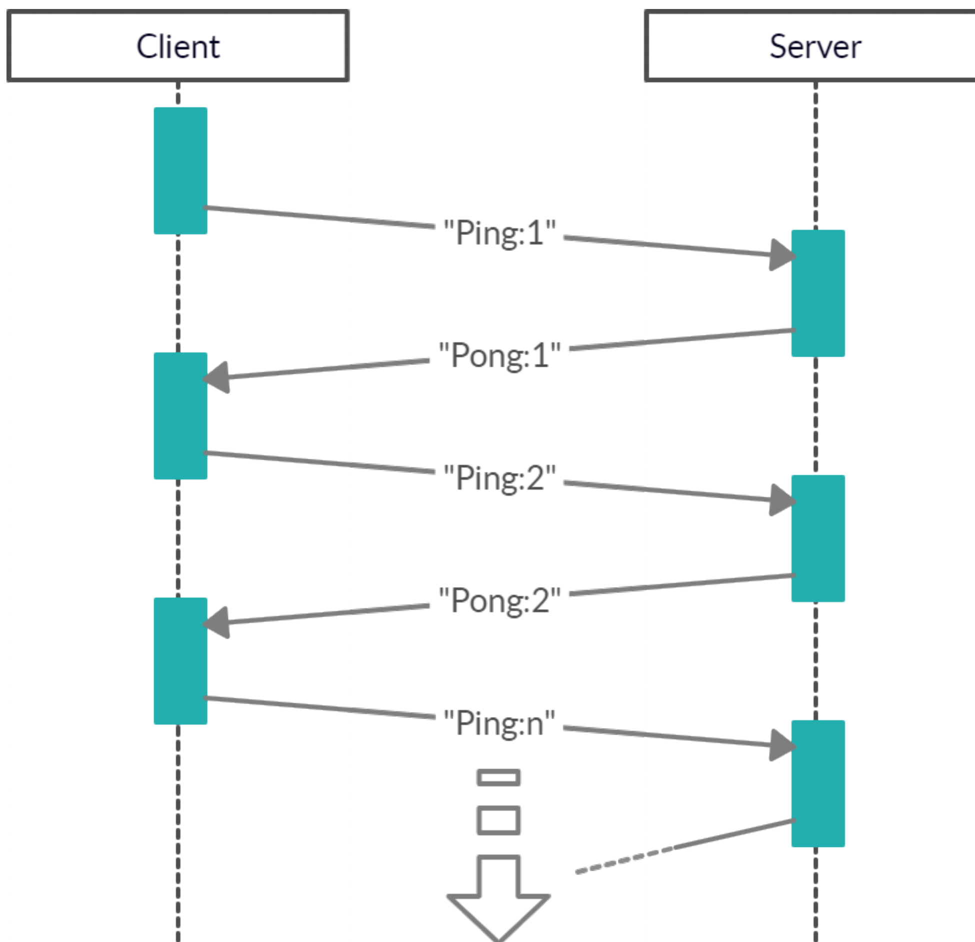

Figure 5.6: Ping-Pong protocol

5.3.1.1 Test results without network modifications

When running the Ping-Pong test scenario without any network modifications, FMLink averaged around 1600 cycles per second, with a peak of 2000 cycles and a minimum of 1000 cycles. This means that one cycle from one FMClient to another and back took on average around 0.62 ms on a local network. This can be considered a good result since it approaches the average latency of the network, which was previously measured to be 0.24 ms.

5.3.2 Testing delayed connection

For this test, TMnetsim proxy was used for the emulation of delay, and the client test app connected to the proxy instead. For the first part, the delay was set to 50 ms on both uplink and downlink. The theoretical round-trip time for this case is 100 ms. When the Ping-Pong scenario was executed, it

achieved 9.8 cycles per second with no significant variance in the number of cycles per second.

For the second part, the delay of the network was set to 1000 ms, which can be considered a very significantly degraded network. In this case, FMLink achieved stable 0.5 cycles per second, which is comparable to theoretical achievable result. This confirms that in scenarios, where FMLink is deployed in a network with very large delay, FMLink will not add any significant delay to the communication.

5.3.3 Testing a connection with jitter

In this test, TMnetsim was used again. The emulated delay was set to 50 ms, with 50 ms jitter with normal distribution. This meant, that packets were delayed with times between 25 ms a 75 ms, averaging a 50 ms delay. When I executed the Ping-Pong test with these settings, I have achieved averaged 9.8 cycles per second, almost exactly as without the jitter. This proved that FMLink can work in a network with significant jitter.

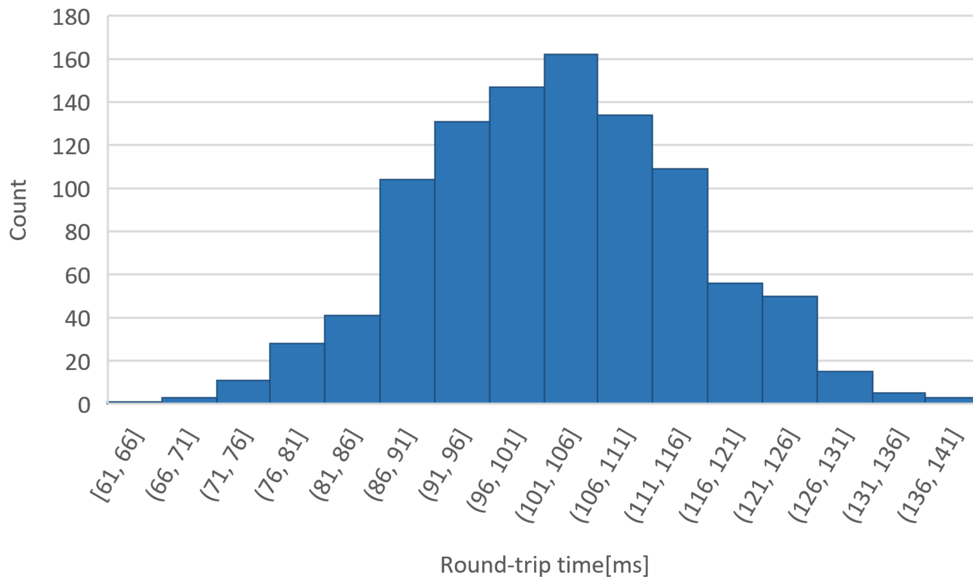

Figure 5.7: Distribution of round-trip times over an emulated network with delay and jitter

It must be noted, that although FMLink doesn't increase jitter, it will not decrease it. Figure 5.7 shows a distribution of 1000 recorded round-trip times over FMLink and it approximates the normal distribution of delay times emulated by the TMnetsim proxy.

5.3.4 Testing connection with packet loss

Another scenario in which the Ping-Pong protocol over FMLink was tested was a network with significant packet loss. In order to simulate random packet loss, Clumsy was chosen. I have tested 6 different scenarios, with increasing packet loss percentage. The packet loss was set at 0.1 % for the lowest value and then the test was repeated with 0.5 %, 1 %, 2 %, 6 % and 10 % packet loss probabilities. The test was executed for 100 sec each time and I recorded the response time of each ping-pong cycle. After 100 seconds, the cycle time measurements were written to file. These measurements were then used to create histograms of cycle times and they can be seen in figure 5.8.

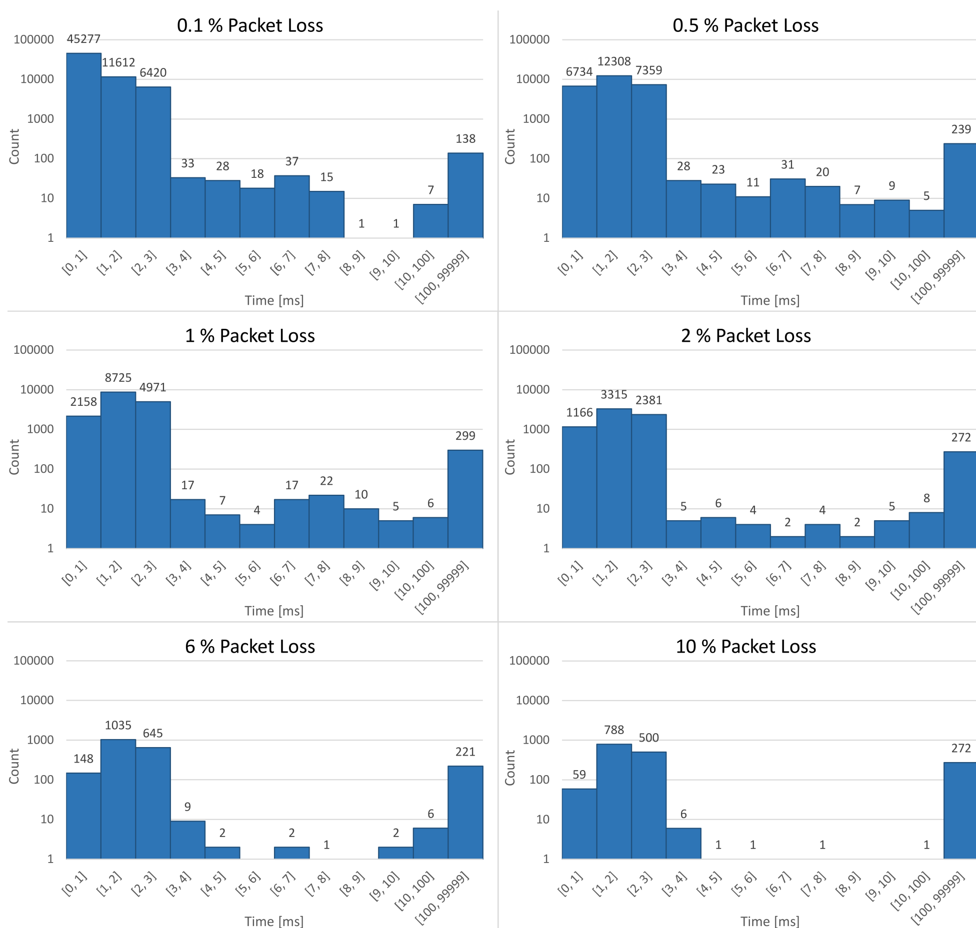

Figure 5.8: Histograms of cycle times in a network with packet loss

When the packet loss was set to 0.1 %, the average cycle time was 1.64 ms and 71 % of cycles were under 1 ms. 63587 ping-pong cycles were sent over the network, which accounts to around 635 cycles per second. Network with 0.1 % packet loss can be considered already pretty unreliable, but since

FMLink currently uses TCP, the underlying transport protocol handled the retransmission of missing messages, and no cycles were lost permanently, the counter protocol was still increasing one-by-one.

After the increase of packet loss probability to 0.5 %, the average cycle time dropped to 3.91 ms, with 26774 cycles during the test, less than half of the previous test. With 267 ping-pong cycles per second, this is still usable for most applications. I was gradually increasing the packet loss probability and the number of cycles per second was significantly dropping. At a 10 % packet loss ratio, which is considered to be highly disrupted network and unusable for most activities, the ping-pong scenario recorded an average of 16 cycles per second. As seen from the histogram, although the average was pretty good, 16 % of cycles took more than 200 ms.

Even with 10 % packet loss, the ping-pong test scenario kept working as expected. This proved the resiliency of FMLink to packet loss conditions. During each test, FMLink reported no errors, such as missing cycles or ping-pong counters out of order. The reason that many cycles took over 200 ms was the way TCP handles retransmissions. For each ACK (confirmation of delivery) packet, TCP sets a time limit it expects this ACK to arrive in. For many packets, this time limit, called RTO (retransmission timeout) was set to 200 ms and after this period, TCP sent the packet again. The RTO time limit cannot be easily set manually and TCP protocol uses Karn's and Jacobson's algorithms to estimate RTO values. If the FMLink would be implemented over UDP, it would need to implement some form of RTO estimation algorithm, which would greatly exceed the scope of the project.

5.3.5 Testing tree exchange

Another feature of FMLink that was needed to be tested was the capability to synchronize Node trees. For this reason, I have implemented another part of the test application, that simulates tree synchronization workflow.

This test scenario again first establishes a connection between the server and the client. The client then creates a Project with just one Node. The Node has one attribute, that has a Value of type byte array. This allows the attribute to have variable and potentially very large size. The client fills the byte array with random values but considers the first value to be a "counter" (similar to the ping-pong scenario), so it sets this first value to 0. The client then calls Notify on the FMLinkSession, so that the provided Project is synchronized to the server. After the server receives the Node tree, it reads the first value from the byte array attribute of the aforementioned Node and treats it as the counter from the Ping-Pong protocol. It then sends back an ASCIITextMessage over some FMLinkChannel, which contains the counter. After the client receives the text response, it can extract the counter value from the message and confirm that it is equal to the first value of the byte array attribute and that the tree synchronization was successful. The client then increases the counter, sets a

new randomly generated byte array to the Node attribute a Notifies the session again. This repeats in an infinite cycle.

With each Notify called by the client, FMLink should send the whole byte array from the attribute, since it's value will be different each time and it will require synchronization to the server so that the server has an up-to-date copy of the Node tree. Since FMLink uses FMPatchableArchive as a container to transport the data and it currently doesn't support any compression of byte array attributes, the actual number of transferred bytes by FMLink can be easily obtained from the length of the byte array.

This scenario will allow to test how efficient is FMLink in utilizing the provided network bandwidth.

5.3.5.1 Testing maximum possible network utilization

For the first test of this scenario, both client and server were connected directly, each running on a different computer (the same as before, one running on the laptop, and another one on the desktop). The client app was configured to synchronize a tree, where the payload attribute has a size of 1 GB. I have then measured, how much does the FMLink utilize the network resources during the synchronization.

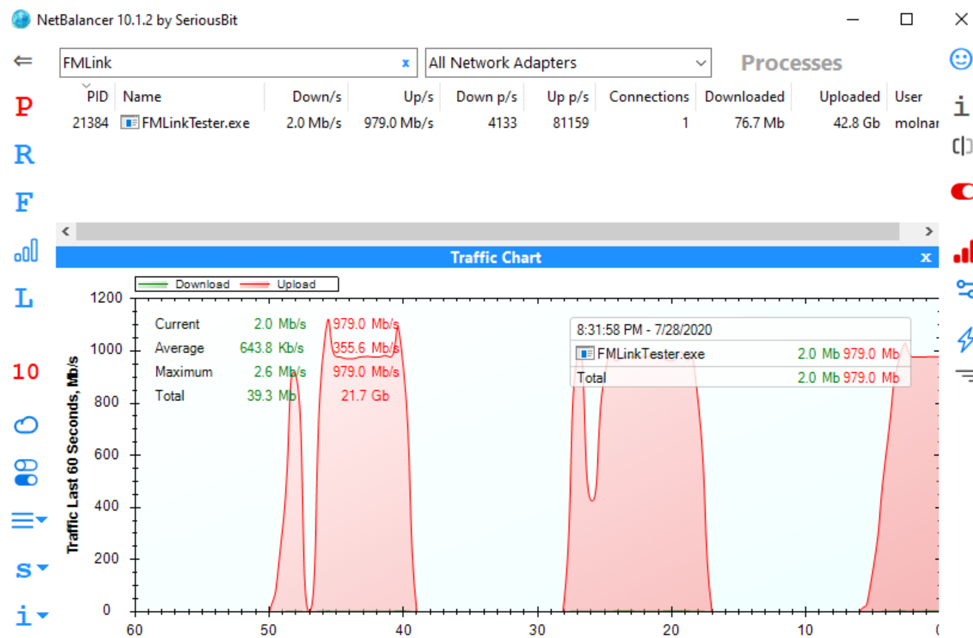

Figure 5.9: Utilization of the network

As seen from the measurements in figure 5.9, the measurements from the NetBalancer monitor proved that FMLink is able to fully utilize the provided gigabit network connection up to its limit. Blank intervals between uploads

Test Number	Packet Loss	Average[ms]	Std Dev
1	0 %	4829	299
2	0.1 %	5113	128
3	0.5 %	5036	271
4	1 %	5317	533
5	2 %	6211	1072
6	3 %	8654	884
7	4 %	27520	2631

Table 5.2: Test result of tree synchronization with different packet loss scenarios

seen in the graph are caused by the client generating the large byte array, serialization, and deserialization on the server – all these are limited by the CPUs on the machines.

It is worth noting one interesting aspect seen in the screenshot, which is that FMLink keeps open only one TCP connection, even though internally it provides two communication channels available to the user (one for the tree synchronization and one for the text messages for confirmation). This is implied by the fact that the FMLinkClient uses only one FMLinkSocket for sending and receiving the data while allowing the prioritization of messages from different channels.

5.3.5.2 Tree synchronization over lossy network

For the next test, I have tried to emulate a scenario, in which the client is connected to the server over a connection with packet loss and slower network speed. The network throughput was limited to 50 MBit/sec by the NetBalancer and on one of the computers, Clumsy was executed. For each test, a tree update with a size of 25 MB was sent to the server thirty times and each tree update was waiting for the confirmation from the server that the previous update was successful. For each tree update, a stopwatch was started at the time of Notify command from the client, and time was recorded to file when a successful response was received from the server.

The test was executed 7 times, each time with different packet loss percentages and the results can be seen in table 5.2. The results tell us that packet loss up to 1 % made very little difference to the synchronization speed. What is interesting, is that when the packet loss was increased further, the transfer speed achieved started dropping considerably. At 4 % packet loss, it took several times longer to transfer the same amount of data. It must be again noted, that even though the transfer took longer, it was still successful and the application was working as expected.

When I tried to increase the packet loss to 10 %, the achieved transfer speed dropped to single kilobits per second and the test had to be terminated.

For this reason, I was testing only packet loss up to 4 % in this test scenario. This is still considered a good result since such high packet losses are not common in modern internet networks (when testing the network without any packet loss emulation, no packet loss at all have been detected).

Conclusion

The main subject of this thesis is the synchronization of complex data over the network in complicated situations and environments. The main target of this thesis was to develop the FMLink library for the company Pocket Virtuality.

FMLink needed to be highly integrated into the existing Fata Morgana platform and in order to do that, I first had to analyze the existing software environment used for this platform. I described the basic building blocks and needs of applications in the Fata Morgana ecosystem. This allowed me to populate a clear list of requirements, that needed to be met to fulfill the mission of the FMLink library.

The current FMLink library was designed with all these requirements in mind and the implementation achieved all the functionality that was expected from this solution. Comprehensive tests in the testing chapter proved that FMLink is able to function even in highly degraded network conditions while still working as expected. FMLink was tested not only by the tests described in this thesis but also by countless hours of usage by other software developers, which helped me to create a solution that the company Pocket Virtuality can rely on.

From a personal perspective, I have chosen this topic for my thesis because it allowed me to develop a software solution, that has real-world applications. FMLink is currently already deployed and used by many Fata Morgana applications. During the whole development process, which took almost a year, I was constantly engaging in conversations with other Pocket Virtuality employees, in order to develop the solution that will meet their needs.

FMLink is still ongoing active development and it is expected that this networking solution will be used for years to come. Some of the expected improvements that will be further developed are the support of the UDP protocol, discovery protocols to find other devices on the network, a security audit, and many others.

Bibliography

[1] Bosveld et al. Optical Display Element For a Headset. June 2015. Available from: https://pdfpiw.uspto.gov/.piw?Docid=D0740813

[2] OSI model vs TCP/IP model. Mar. 2020. Available from: https://www.routexp.com/2020/03/osi-model-vs-tcpip-model.html

[3] Fata Morgana. 2020. Available from: https://www.pocketvirtuality.com/products.php

[4] HoloLens 2 hardware. Sept. 2019. Available from: https://docs.microsoft.com/en-us/hololens/hololens2-hardware

[5] C++/WinRT - UWP Applications. Apr. 2019. Available from: https://docs.microsoft.com/en-us/windows/uwp/cpp-and-winrt-apis/

[6] Postel, J. Internet Protocol. Sept. 1981. Available from: https://tools.ietf.org/html/rfc791#page-25

[7] .NET Garbage collection. Apr. 2020. Available from: https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/

[8] std::shared_ptr. Available from: https://en.cppreference.com/w/cpp/memory/shared_ptr

[9] Bischoff, P. UDP vs TCP: What are they and how do they differ? Jan. 2019. Available from: https://www.comparitech.com/blog/vpn-privacy/udp-vs-tcp-ip/

[10] About Wireshark. 2020. Available from: https://www.wireshark.org/index.html#aboutWS

[11] Shopify. Shopify/toxiproxy. July 2020. Available from: https://github.com/Shopify/toxiproxy

[12] Windows Packet Divert. Available from: https://reqrypt.org/windivert.html