

ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ КЪМ  
ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

## ДИПЛОМНА РАБОТА

Тема: Мрежов анализатор с възможност за отдалечен анализ  
посредством AngularJS 2 клиент

Дипломант: Ивайло Арнаудов

Научен ръководител: Стоил Стоилов

София, 2017

## Списък на съкращения

|     |                         |
|-----|-------------------------|
| VPN | Virtual Private Network |
| IP  | Internet Protocol       |

# Todo list

|                                |    |
|--------------------------------|----|
| аргументация . . . . .         | 20 |
| refactor and explain . . . . . | 34 |
| refactor and explain . . . . . | 34 |
| explain . . . . .              | 39 |

# Съдържание

|          |   |           |
|----------|---|-----------|
| 0.1      | Компютърни мрежи . . . . .  | 6         |
| 0.2      | Приложения на компютърните мрежи . . . . .  | 6         |
| 0.2.1    | Приложения на компютърните мрежи в бизнеса . . . . .  | 6         |
| 0.2.2    | Приложения на компютърните мрежи в дома . . . . .   | 7         |
| 0.3      | Изисквания към мрежов анализатор . . . . .  | 7         |
| <b>1</b> | <b>Методи и технологии за реализиране на мрежови анализатори</b>  | <b>8</b>  |
| 1.1      | Основни принципи, технологии и развойни среди за реализиране на мрежови анализатори . . . . .                 | 8         |
| 1.1.1    | Основни принципи . . . . .  | 8         |
| 1.1.2    | Основни технологии . . . . .  | 13        |
| 1.1.3    | Основни развойни среди . . . . .  | 14        |
| 1.2      | Съществуващи решения и реализации . . . . .   | 14        |
| 1.2.1    | Wireshark . . . . .   | 15        |
| 1.2.2    | tcpdump . . . . .   | 15        |
| <b>2</b> | <b>Проектиране на структурата на мрежов анализатор</b>  | <b>18</b> |
| 2.1      | Функционални изисквания към мрежов анализатор . . . . .   | 18        |
| 2.1.1    | Намиране и избиране на физически интерфейси . . . . .   | 18        |
| 2.1.2    | Интерпретиране на данни . . . . .   | 18        |
| 2.1.3    | Филтриране на данни . . . . .   | 18        |
| 2.1.4    | Отдалечен анализ . . . . .  | 18        |
| 2.1.5    | Сигурност на отдалечения анализ . . . . .   | 19        |
| 2.1.6    | Споделяне на анализа . . . . .  | 19        |
| 2.1.7    | Сигурност на споделяне на анализа . . . . .   | 19        |
| 2.1.8    | Графичен интерфейс и визуализация . . . . .   | 19        |
| 2.2      | Съображения за избор на програмни средства и развойна среда . . . . .   | 19        |
| 2.2.1    | C++ . . . . .   | 19        |
| 2.2.2    | libpcap . . . . .   | 19        |
| 2.2.3    | WebSocket . . . . .   | 20        |
| 2.2.4    | Angular 2 . . . . .   | 20        |
| 2.3      | Проектиране на архитектура на мрежов анализатор с отдалечен достъп . . . . .                                  | 20        |
| 2.3.1    | Цялостна архитектура . . . . .  | 20        |
| 2.3.2    | Архитектура на общи класове . . . . .   | 20        |
| 2.3.3    | Архитектура и алгоритъм на мрежовия анализатор . . . . .  | 21        |
| 2.3.4    | Архитектура на сървър . . . . .   | 25        |
| 2.3.5    | Архитектура на клиент . . . . .   | 26        |
| <b>3</b> | <b>Програмна реализация на мрежов анализатор</b>  | <b>28</b> |
| 3.1      | Имплементация на общи класове <code>ConfigurationManager</code> и <code>SerializationManager</code> . . . . . | 28        |
| 3.1.1    | Имплементация на клас, енкапсулиращ конфигурация . . . . .  | 28        |

|          |  |           |
|----------|--|-----------|
| 3.1.2    | Имплементация на <code>ConfigurationManager</code> . . . . .   | 28        |
| 3.1.3    | Имплементация на <code>FileStoragePolicy</code> . . . . .  | 29        |
| 3.1.4    | Имплементация на <code>JsonFormattingPolicy</code> . . . . .   | 30        |
| 3.1.5    | Имплементация на клас, енкапсулиращ сериализиран обект . . . . .                                     | 30        |
| 3.1.6    | Имплементация на <code>SerializationManager</code> . . . . .   | 31        |
| 3.2      | Имплементация на мрежов анализатор . . . . .   | 32        |
| 3.2.1    | Имплементация на намиране на физически интерфейси . . . . .  | 32        |
| 3.2.2    | Имплементация на <code>LayerStack</code> и <code>Layer</code> класове . . . . .                      | 34        |
| 3.2.3    | Имплементация на <code>SniffedPacket</code> класа . . . . .  | 36        |
| 3.2.4    | Имплементация на <code>ReceptionHandler</code> . . . . .   | 36        |
| 3.2.5    | Имплементация на <code>PacketSniffer</code> и <code>PcapPacketSniffer</code> . . . . .               | 37        |
| 3.2.6    | Имплементация на <code>Header</code> и <code>HeaderMetadata</code> класовете . . . . .               | 39        |
| 3.2.7    | Имплементация на <code>PayloadInterpreter</code> и <code>HexAsciiPayloadInterpreter</code> . . . . . | 39        |
| 3.3      | Имплементация на сървър . . . . .  | 40        |
| 3.3.1    | Имплементация на <code>Server</code> . . . . .   | 40        |
| 3.3.2    | Имплементация на <code>WebSocketServer</code> . . . . .  | 40        |
| 3.3.3    | Имплементация на <code>WebSocketServerEventHandler</code> . . . . .                                  | 41        |
| 3.3.4    | Имплементация на <code>Command</code> . . . . .  | 43        |
| 3.3.5    | Имплементация на <code>ServerCommandInvoker</code> . . . . .   | 44        |
| 3.4      | Имплементация на клиент . . . . .  | 45        |
| 3.4.1    | Имплементация на услуги . . . . .  | 45        |
| 3.4.2    | Имплементация на компоненти . . . . .  | 49        |
| <b>4</b> | <b>Ръководство на потребителя</b>  | <b>51</b> |
| 4.1      | Deployment . . . . .   | 51        |
| 4.2      | Работа с приложението . . . . .  | 51        |
| 4.2.1    | Въвеждане на парола . . . . .  | 51        |
| 4.2.2    | Избор на физически интерфейс . . . . .   | 51        |
| 4.2.3    | Въвеждане на филтриращ израз . . . . .   | 52        |
| 4.2.4    | Стартиране на сесия . . . . .  | 52        |
| <b>5</b> | <b>Заклучение</b>  | <b>54</b> |
|          | <b>Библиография</b>  | <b>55</b> |
|          | <b>Списък на фигурите</b>  | <b>56</b> |
|          | <b>Списък на таблиците</b>   | <b>57</b> |

# Увод

## 0.1 Компютърни мрежи

Всеки от последните три века бива доминиран от някаква нова технология. Пример за това е ерата на механичните системи съпътстващи Индустриалната революция през XVIII век. За XIX век пък е характерен парния двигател. През XX век, ключовата технология е събирането, обработката и дистрибуцията на информация. С развитието ѝ човечеството става свидетел на инсталацията на глобални телефонни мрежи, изобретяването на радиото и телевизията, експоненциалния растеж на развитието на компютърната индустрия и, разбира се, Интернет. Като резултат от огромния технологичен прогрес в сферата на информационните технологии, през XXIV. разликите между съхраняване, транспортиране и обработка на информация изтъняват, а успоредно с това растат и изискванията на крайния потребител към комуникационните услуги.

Въпреки крехката възраст на компютърната индустрия, тя прави значителен прогрес. През първите две десетилетия от съществуването им, компютърните системи са били силно централизирани. Университет или средно голяма фирма биха имали един или два компютъра, а по-големите институции — по няколко. Идеята за съществуването на малки устройства тип смартфон, които са взаимосвързани, е била по-скоро утопична.

Обединяването на компютрите и комуникациите оказва голямо влияние върху организацията на самите компютърни системи. Старият модел при който един компютър изпълнява заявките на цялата организация бива заменен от нов модел при който голямо количество отделни, но взаимосвързани компютри извършват обработка на дадена информация. Тези системи се наричат *компютърни мрежи*. [Tanenbaum and Wetherall \(2011\)](#)

## 0.2 Приложения на компютърните мрежи

### 0.2.1 Приложения на компютърните мрежи в бизнеса

Обикновено повечето компании имат голямо количество компютри, най-често по един за всеки служител. Изначално, те биха могли да работят в изолация един от друг, но в даден момент идва необходимост те да бъдат свързани с цел служителите да извършват работата си по-пълноценно чрез колаборация помежду си.

Един от основните проблеми, който решават компютърните мрежи е споделянето на ресурси. Целта е информацията да бъде достъпна от всеки в мрежата независимо от физическото му местоположение.

По-важен проблем, който те решават, е споделянето на информация. Компаниите са фундаментално зависими от дигиталната информация. Повечето компании имат записи за клиенти, за продукти и т.н. онлайн. Мрежите предоставят механизми за по-богати форми на виртуална комуникация – споделяне на екрана, видеоконференции, споделена обработка на документи. Компютърните мрежи отварят вратите и за нов бизнес модел,

наречен *електронна търговия (e-commerce)*, който се развива с големи темпове и става де факто стандарт при търговията от всякакъв тип. [Tanenbaum and Wetherall \(2011\)](#)

## 0.2.2 Приложения на компютърните мрежи в дома

В началото на компютърната индустрия причините за покупка на компютър от крайния потребител са се свеждали до нужда от обработка на текст и игри. През XXI век, основната причина е нуждата за достъп до Интернет. Аналогично на компаниите, крайните потребители могат да достъпят отдалечена информация, да комуникират посредством *социални мрежи*, да купуват продукти и услуги чрез *e-commerce* системи, да използват електронно банкиране, да споделят мултимедия и софтуер, да колаборират посредством *wiki* сайтове.

Друго напоследък развиващо се приложение на мрежите е концепцията за *Internet of Things*. Основната ѝ характеристика е, че електронните устройства на крайните потребители се включват в компютърните мрежи. Например, душа в банята, който традиционно не е компютър, би могъл да записва какво количество вода е използвано и да праща информацията на приложение, което изчислява как водата да бъде използвана по-ефективно.

## 0.3 Изисквания към мрежов анализатор

Споменатият етап на развитие на компютърните мрежи предразполага към по-комплексни инструменти и процеси за мониторинг и отстраняване на проблеми в мрежата. *Мрежовия анализатор* е инструмент, който помага на мрежовия администратор, предоставяйки услугата *анализ на пакети (packet analysis, packet sniffing)*. Анализът на пакети описва процеса на заснемане и интерпретиране на данни в реално време (т.е в момента на преминаване през преносвателната среда). Изискванията към един такъв анализатор е да улесни мрежовия администратор със изпълнението на следните задачи:

- Идентифициране и задълбочено разбиране на процесите в мрежата
- Идентифициране на участниците в мрежата, потенциални причинители на атаки или злонамерена активност
- Изследване кой или какво използва наличния капацитет на преносвателната среда, както и на моментите на максимално използване (load) на мрежата

Представеното в текущата дипломна работа приложение има за цел да спази тези изисквания като, вземайки предвид вече описаното развитие на компютърните мрежи с оглед необходимостта от колаборация и децентрализираност, предостави на мрежовия администратор и възможност за отдалечено достъпване на мрежовия анализатор както и споделен преглед на анализа с други администратори.

# Глава 1

## Методи и технологии за реализиране на мрежови анализатори

### 1.1 Основни принципи, технологии и развойни среди за реализиране на мрежови анализатори

#### 1.1.1 Основни принципи

Процеса на анализ на пакети включва кооперация между софтуера и хардуера и може да бъде разделен в следните три стъпки:

- **Събиране** В началната фаза на работата си, анализатора събира 'сурови', неинтерпретирани данни в двоичен вид директно от проводника. Типично, това става като съответно избрания мрежови интерфейс за анализ бива превключен в т.нар. **promiscuous mode**. В този режим, мрежовата карта може да 'слуша' за всевъзможен трафик по дадения мрежови сегмент, а не просто такъв, адресиран до станцията.
- **Конвертиране** В следващата фаза на работата си, анализатора конвертира събраните данни в разбираем формат за крайния потребител. След тази стъпка, данните събрани от преносвателната среда са във вид, който може да бъде интерпретиран на много основно ниво; останалата по-голяма част от анализа се оставя на крайния потребител.
- **Анализ** В третата и финална фаза, мрежовият анализатор извършва реалния анализ на събраната и конвертирана информация. Анализатора взема събраните данни, отчита използвания мрежови протокол, базирайки се на извлечените до момента данни, и започва да анализира конкретните свойства на протокола.

С цел да бъде разбран процеса на работа, а и съответно модела на реализация на мрежов анализатор, е необходимо дефиниране на основните принципи на комуникация между компютърните системи.

#### Протоколни архитектури

За да се намали комплексността на решенията, повечето мрежи са организирани като стек от *слоеве* или *нива*, всеки изграден върху слоя под него. Броя на слоевете, имената на всеки от тях, съдържанието на всеки и функциите, които изпълнява са различни за различните мрежи. Целта на всеки слой е да предостави конкретни услуги на



разположените по-високо от него в йерархията слоеве като им спестява детайлите около имплементацията на тези услуги.

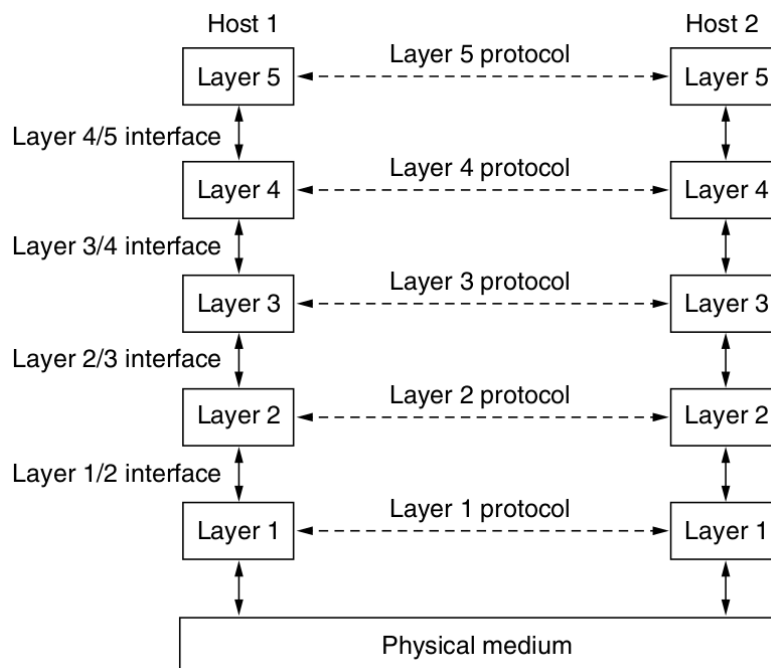
Тази концепция е широко популярна в компютърните науки, още известна е като криене на имплементационни детайли, абстрактни типове от данни, енкапсулация на данни. Фундаменталната идея е, че дадена част от софтуера (или хардуера) осигурява услуга на потребителите си, но скрива от тях детайлите на вътрешното си състояние и алгоритмите.

Когато слой  $n$  на една машина е във връзка със слой  $n$  на друга машина, правилата и конвенциите използвани в тази връзка се наричат *протокол на  $n$ -ти слой*. На практика, протокол е договореност между комуникиращите страни относно това как протичат процесите на комуникацията между тях. Tanenbaum and Wetherall (2011) Протоколите могат да бъдат прости или комплексни. Някои от общите свойства, които традиционно споделят, макар и абстрактно представени Sanders (2011), са:

- **Инициация на връзка** Дефинира кой инициира връзката, например клиентът или сървърът. При инициране на връзка може да е необходима и допълнителна служебна информация преди да протече обмен на полезна информация.
- **Договаряне на параметрите на връзката** Дефинира процес, в който двете страни се разбират — дали връзката е криптирана, как се пренасят ключовете за декриптиране, какъв тип е връзката (full/half duplex) и др.
- **Форматиране на данните** Дефинира подредбата на данните, в каква последователност се обработват от приемащата страна и др.
- **Откриване на грешки и корекция на грешки** Дефинира какво се случва при загуба на данни, как едната страна на връзката реагира при липса на отговор от другата и др.
- **Терминиране на връзка** Дефинира как дадено крайно устройство сигнализира на друго че връзката е приключила, каква финална информация трябва да бъде предадена преди успешния край на връзката и др.

Традиционно, тези протоколи не 'живеят' сами, а са основополагащи за цялостния процес на комуникация. Например, на Figure 1.1 е представен пет слоен модел на комуникация между два софтуерни процеса. Реално данни не се предават директно от слой  $n$  на едната машина до слой  $n$  на другата: комуникацията е *виртуална* (означена с прекъснати линии на Figure 1.1). Вместо това, всеки слой предава данни и контролна информация на този под него докато не се достигне най-ниския слой. Под първия слой е физическият, т.е. преносвателната среда, през която реалната комуникация се случва. (означено с непрекъснати линии на Figure 1.1)

Между всяка съседна двойка слоеве има *интерфейс*. Този интерфейс дефинира какви операции и услуги долният слой предлага на горния. Интерфейсите между слоевете трябва да бъдат ясно дефинирани. Това впоследствие би улеснило замяната на един слой с напълно различен протокол или имплементация (например смяна на телефонни линии със сателитни такива), защото единственото, което се очаква от новия протокол, е да предлага *точно* същото множество услуги на горния слой като стария. Аналогично, този механизъм позволява един протокол да се промени в даден слой без знанието на слоевете под и над него.



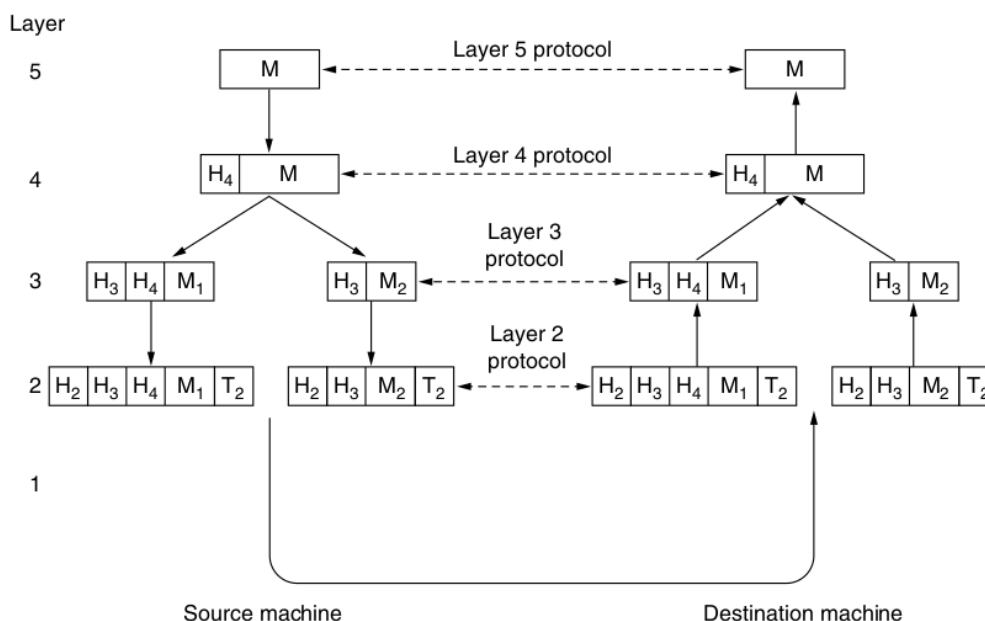
Фигура 1.1: Модел на пет слойна мрежа

### Основен механизъм на междуслойна комуникация. Капсулиране и декапсулиране.

Основния механизъм на междуслойна комуникация е фундаментален за имплементацията на представения в дипломната работа мрежов анализатор. Той може да бъде описан с помощта на пет слойния модел представен във Figure 1.2. В случая, даден процес иска да изпрати съобщението  $M$ . От петия слой, съобщението бива предадено на четвъртият слой за изпращане. Четвъртият слой слага т.нар. *header* в началото на съобщението и предава резултата към трети слой. Този *header* включва служебна информация, например адреси, за да може съответния четвърти слой на приемната машина да достави съобщението. Други примери за служебна информация могат да бъдат *числови поредици* (*sequence numbers*) — често използвани когато слоят на по-ниско ниво няма функционалност за запазване на последователността на съобщенията, както и размери и времена.

В много мрежи, често няма граница относно размера на съобщения на четвърти слой, но почти винаги има такава относно размера от самия протокол на трети слой. Следователно, третият слой трябва да раздели идващите съобщения на по-малки единици — пакети, като успоредно с това добавя *header* към всеки пакет. В случая на Figure 1.2, съобщението  $M$  се разделя на две части:  $M_1$  и  $M_2$ .

Третият слой аналогично предава пакетите на втория слой, който от своя страна освен че добавя *header*, добавя и *опашка* (*trailer*). Резултата се предава на първия слой, който се занимава с физическия пренос на данните. Описаният процес е още известен като *капсулация* (*encapsulation*). В приемащата страна, съобщението се *декапсулира* (*decapsulation*) като всеки *header* се отделя успоредно с 'изкачването' на съобщението нагоре по слоевете. Нито един *header* за слоевете под  $n$ -тия не достига до  $n$ -ти слой. Едновременно с това, на Figure 1.2 ясно проличават виртуалната и реална комуникация, както и разликите между протоколи и интерфейси. Например, на четвърти слой процесите концептуално интерпретират комуникацията си като хоризонтална, използвайки протокола на четвърти слой, и биха имали функции от типа на `send()` и `recieve()`, макар че в действителност те комуникират с по-ниските слоеве през 3/4 интерфейса.



Фигура 1.2: Междуслойна комуникация. Капсулиране и декапсулиране.

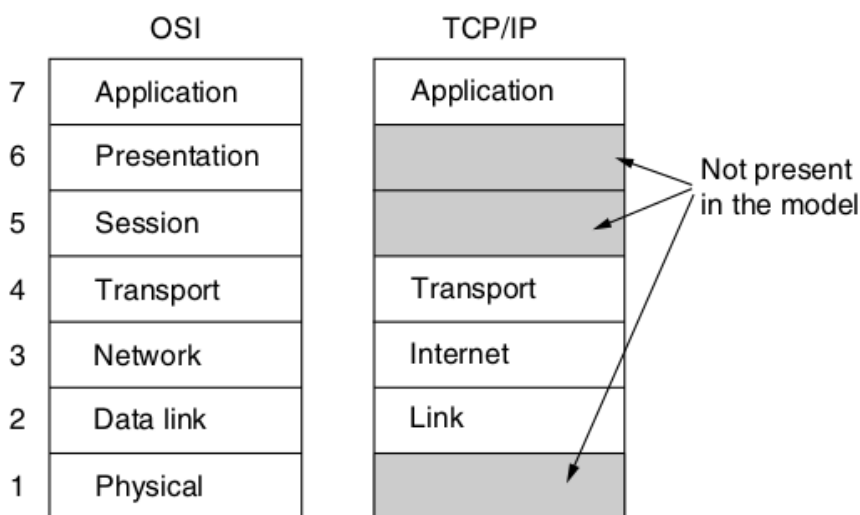
### Имплементации на протоколни архитектури

**Open Systems Interconnection (OSI)** представлява модел на свързване на отворените системи, т.е системи отворени за комуникация с другите такива. Развит от Международната организация по стандартизация (ISO), той има седем слоя:

- **Physical** Физическият слой се занимава с предаването на битове информация в чист вид посредством електрически сигнали.
- **Data Link** Каналният слой се занимава с контрола на достъпа до споделената преносителна среда и създаването на абстракция върху чистия пренос на данните. Това става чрез разделянето на изпращаните данни в единица, наречена *frame* и изпращането ѝ последователно.
- **Network** Мрежовият слой се занимава с маршрутизацията на пакети от подател към получател.
- **Transport** Транспортният слой се занимава с разделянето на получените данни от слоевете над него, подаването им към мрежовия слой и гарантиране на успешния им пренос.
- **Session** Сесийният слой се занимава с установяването на *сесии* между потребителите на две машини. Примерни услуги са контрол на диалога и синхронизация.
- **Presentation** Представителният слой се занимава с синтаксиса и семантиката на изпратената информация.
- **Application** Приложеният слой се занимава с преноса на данни на ниво приложения, типични представители за такива протоколи са HTTP, FTP и т.н.

**TCP/IP** представлява първият модел на работа на компютърните мрежи, разработен първоначално през 1974г. за да опише функционалността на ARPANET. OSI моделът на практика представлява негово разширение. TCP/IP моделът е съставен от 4 слоя:

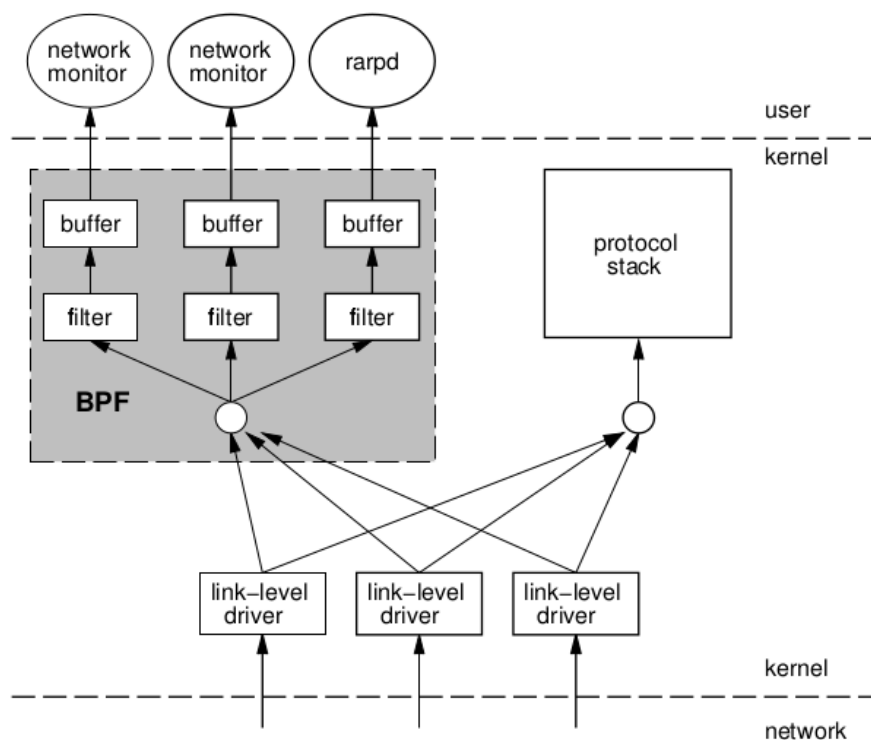
- **Link** Каналният слой описва какви връзки да бъдат използвани, например серийни или класически Ethernet, като по-скоро представлява не слой, а интерфейс между преносвателната среда и машините.
- **Internet** Мрежовият слой се занимава с маршрутизацията на пакети от подател към получател. Класически примери за протоколи, опериращи на слоя са IP (Internet protocol) и ICMP (Internet Control Message Protocol).
- **Transport** Транспортният слой се занимава с разделянето на получените данни от слоевете над него, подаването им към мрежовия слой и гарантиране на успешния им пренос. В зависимост от нуждата за надеждност на преноса има два типични протокола: TCP (Transmission Control Protocol) и UDP (User Datagram Protocol).
- **Application** Приложеният слой се занимава с преноса на данни на ниво приложения, типични представители за такива протоколи са HTTP, FTP и т.н.



Фигура 1.3: Сравнение между OSI и TCP/IP

**Основен принцип за заснемане на данни от преносвателната среда**  
Обикновено, когато мрежовата карта получи frame, тя проверява дали адресът на получателя съвпада с нейния собствен. Ако съвпада, тя генерира прекъсване към процесора. Функцията, обработваща прекъсването, е драйверът за мрежовата карта в ядрото на операционната система. Драйверът 'закача' *времева щампа (timestamp)* върху приетите данни и копира данните от буфера на картата в заделен блок от памет в ядрото на операционната система. След това системния протоколен стек обработва данните чрез процеса на декапсулация и ги предава към потребителското приложение.

При типичната имплементация на мрежов анализатор, пакетите следват аналогичен път, но с малка промяна: драйверът копира приети или изпратени данни в част от ядрото на операционната система наречено *пакетен филтър (packet filter)*. По подразбиране, пакетните филтри пропускат всеки пакет, но поддържат комплексни филтри. Важно е да се отбележи, че в следствие от местоположението си, пакетните филтри изискват административни привилегии, тъй като копирането на получени/изпратени пакети предполага риск за сигурността. На Figure 1.4 е илюстриран описаният процес с помощта на Berkley пакетния филтър (BPF), механизъм, стоящ в основата на libpcap.



Фигура 1.4: Преглед на имплементацията на мрежов анализатор на ниво ядро на операционната система

### 1.1.2 Основни технологии

#### libpcap

**libpcap** е библиотека с отворен код, която осигурява интерфейс на високо ниво за системи за прихващане на пакети от преносвателната среда. Създадена е през 1994г. от МакКейн, Лиърс и Якобсън — изследователи в Lawrence Berkeley National Laboratory към University of California, като част от научен проект за изследване и подобрене на TCP. Интерфейсът на libpcap е основно достъпен на програмните езици C и C++. Съществуват и голямо количество библиотеки енкапсулиращи функционалността ѝ на езици като Perl, Python, Java, C или Ruby.

#### WebSocket

**WebSocket** представлява комуникационен протокол, позволяващ двустранна (full-duplex) връзка в реално време между клиент и сървър върху TCP. Типичното му приложение е в уеб браузърите и уеб сървърите (тъй като при началното ръкостискане се използва HTTP Upgrade заявката), но на практика може да се приложи от всеки тип клиент-сървър приложение. Протокола е стандартизиран през 2011г. в RFC 6455.

#### Angular 2

**Angular 2** представлява framework за разработка на едностранни (SPA) уеб приложения. Наследникът на AngularJS имплементира компонентно-базирана архитектура като акцентира още повече върху улесняването на разработката на този тип приложения. Главен фокус са и разработката на мобилни приложения, модулърността и тестваемостта.

### 1.1.3 Основни развойни среди

#### Eclipse CDT

**Eclipse CDT** предоставя напълно функционална развойна среда за програмиране на езиците C и C++, базирана на платформата Eclipse. По-забележителните характеристики са: поддръжка на множество системи за автоматизиране на процеса на компилация, навигация на изходния код, йерархия на типовете, граф на извикванията, навигация на макроси, рефакториране и генерация на програмен код, инструменти за дебъгване, включително такива за преглед на паметта, регистрите и деасемблиране.

#### Qt Creator

**Qt Creator** е мултиплатформена среда среда за разработка, част от SDK (Software Development Kit) на популярния framework за разработка на графични интерфейси Qt. Някои от забележителните характеристики на средата са: поддръжка на визуално дебъгване на програмния код, интерактивен дизайнер на подредбата на графичния интерфейс, поддръжка на множество системи за автоматизация на процеса на компилация, поддръжка на множество системи за контрол на изходния код (VCS), поддръжка на инструменти за симулация на приложението за мобилни устройства. Разбира се, средата е предимно подходяща за разработка, когато се използва Qt.

## 1.2 Съществуващи решения и реализации

Преди да разгледаме съществуващите решения и реализации е важно да споменем критериите, необходими за оценяване на полезността на един мрежов анализатор:

- **Поддържани протоколи** Всички мрежови анализатори могат да интерпретират множество от протоколи. Повечето могат да интерпретират най-основните протоколи от мрежовия слой – напр. IPv4 и ICMP; от транспортния слой – TCP и UDP, както и от приложения – DNS и HTTP. Не всички обаче поддържат нетрадиционни или нови протоколи (напр. IPv6).
- **Потребителски интерфейс** От значение е цялостния изглед на приложението, колко лесно се инсталира, колко лесно се извършват необходимите операции посредством интерфейса. От значение е и опита на анализиращия – типично, по-опитният анализиращ би предпочел анализатор използващ командния ред, начинаещият – анализатор с графичен интерфейс.
- **Цена** Голямо количество от мрежовите анализатори са свободно-използваеми и конкуриращи се с платени такива. Обикновено разликата между платените и свободно-използваемите е при прегледа на анализа, който е по-пълноценен при платените.
- **Програмна поддръжка** Обикновено при изникването на проблем анализиращият трябва да има солидна база от източници на решения – документация на анализатора, публични форуми, блогове и т.н. Фундаментално при избора на анализатор е до колко са налични тези източници.
- **Поддръжка на операционната система** Не всеки анализатор поддържа всяка операционна система. Основополагащо за избора на анализатор е операционната система под която се очаква той да функционира.

## 1.2.1 Wireshark

Wireshark има дълга история. Програмата е оригинално създадена от Джералд Комбс, студент по компютърни науки. Първата версия на приложението на Комбс се нарича *Ethereal* и за първи път е пуснато през 1998г. под GNU Public License (GPL). Осем години след пускането на *Ethereal*, той напуска работа, но за съжаление неговият работодател има пълни права върху името *Ethereal*. Така през средата на 2006г. се ражда *Wireshark*.

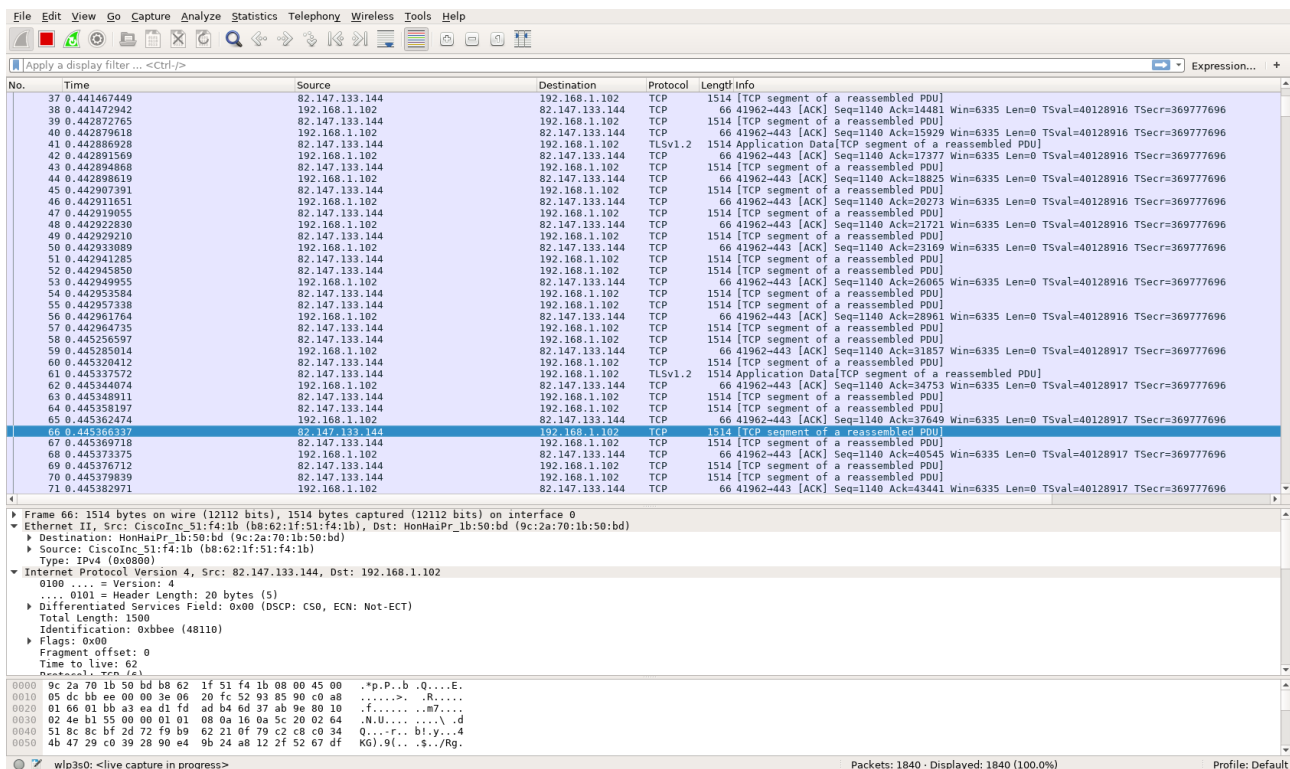
*Wireshark* предлага няколко предимства които я правят подходяща за всекидневна употреба. Програмата таргетира както по-напредналите с мрежовия анализ, така и начинаещи.

- **Поддържани протоколи** Поддържа над 850 различни протокола — от по-популярните като IP и DHCP, до по-комплексни частни протоколи като AppleTalk и BitTorrent. Вземайки предвид факта че *Wireshark* се разработва на принципа на отворения код, нов протокол се добавя с всяка нова версия. В случай че необходим на анализиращия протокол не е имплементиран, той може да бъде имплементиран като разширение и изпратен до разработчиците на *Wireshark*.
- **Потребителски интерфейс** Един от най-лесните за употреба потребителски интерфейси. Типичен Graphical User Interface (GUI) с ясно описани контекстни менюта и интуитивно оформление. Поддържа цветово кодиране спрямо протокола както и детайлен преглед на неинтерпретираните данни. За разлика от *tcpdump*, *Wireshark* GUI е идеална за начинаещия в мрежовия анализ. Разделен е на три компонента: Packet List (списък от пакети), Packet Details (детайли за избрания пакет), Packet Bytes (преглед на пакета в неинтерпретиран вид). Цялостен изглед на интерфейса може да бъде видян на Figure 1.5.
- **Цена** Тъй като програмата се разработва на принципа на отворения код, тя е изцяло безплатна и може да се използва според GPL лиценза.
- **Програмна поддръжка** Тъй като програмата се разработва на принципа на отворения код, тя няма формална поддръжка — тя се базира на обществото от потребители на програмата. Уеб страницата на *Wireshark* има връзки към няколко форми за поддръжка: онлайн документация, wiki за разработчици, FAQ, както и връзки към официалната мейл листа.
- **Поддръжка на операционната система** Поддържа всички модерни операционни системи, вкл. Microsoft Windows, Mac OS X и Linux базираните операционни системи.

## 1.2.2 tcpdump

Програмата е написана през 1987г. от Ван Якобсен, Крейг Леърс и Стивън МакКейн, които работят като изследователи в Lawrence Berkeley Laboratory. Макар че съществуват модерни анализатори с графичен интерфейс, **tcpdump** е изчистен, универсален и ефективен инструмент работещ в командния ред. Едно от основните предимства на програмата е удобството на ползване — използва се една единствена команда, за да се пусне; работи през SSH сесия, няма нужда от window manager и е широкодостъпна на различни платформи. Понеже използва класически конвенции свързвания с командния ред (напр. писане в stdout) може да се използва в голямо количество ситуации.

- **Поддържани протоколи** Поддържа значително по-малко количество протоколи от *Wireshark* — Ethernet (FDDI/Token Ring), IP, IPv6, ARP, RARP, TCP, UDP.



Фигура 1.5: Изглед на потребителския интерфейс на Wireshark.

- **Потребителски интерфейс** Минималистичен command-line interface (CLI). Типичната структура на една команда е представена на Figure 1.6. Поддържа различни нива на детайлност на анализа. На Figure 1.7 е представен средно-детайлен преглед чрез -vv опцията.

```
tcpdump -i eth1 -vvn icmp or udp
```

options
expression

Фигура 1.6: Структура на tcpdump команда.

- **Цена** Тъй като програмата се разработва на принципа на отворения код, тя е изцяло безплатна и може да се използва според BSD лиценза.
- **Програмна поддръжка** Тъй като програмата се разработва на принципа на отворения код, тя няма формална поддръжка — тя се базира на обществото от потребители на програмата.
- **Поддръжка на операционната система** Поддържа повечето UNIX-базирани операционни системи: Linux, Solaris, BSD, HP-UX и др. Microsoft Windows се поддържа през **WinDump**, която използва **WinPcap** — Microsoft Windows имплементацията на **libpcap**.



```

192.30.253.124.https > 192.168.1.102.50882: Flags [.], cksum 0x9687 (correct), seq 1, ack 1, win 31, options [nop,nop,TS val 1639104246 ecr 45537667],
length 0
14:50:00.180575 IP (tos 0x0, ttl 1, id 0, offset 0, flags [DF], proto UDP (17), length 383)
192.168.1.1.ssdp > 239.255.255.250.ssdp: [udp sum ok] UDP, length 355
14:50:00.181683 IP (tos 0x0, ttl 64, id 62855, offset 0, flags [DF], proto UDP (17), length 74)
192.168.1.102.44862 > home-77-70-13-1.megalan.bg.domain: [udp sum ok] 59512+ PTR? 250.255.255.239.in-addr.arpa. (46)
14:50:00.183767 IP (tos 0x0, ttl 1, id 0, offset 0, flags [DF], proto UDP (17), length 320)
192.168.1.1.ssdp > 239.255.255.250.ssdp: [udp sum ok] UDP, length 292
14:50:00.186880 IP (tos 0x0, ttl 1, id 0, offset 0, flags [DF], proto UDP (17), length 311)
192.168.1.1.ssdp > 239.255.255.250.ssdp: [udp sum ok] UDP, length 283
14:50:00.190519 IP (tos 0x0, ttl 1, id 0, offset 0, flags [DF], proto UDP (17), length 375)
192.168.1.1.ssdp > 239.255.255.250.ssdp: [udp sum ok] UDP, length 347
14:50:00.194025 IP (tos 0x0, ttl 1, id 0, offset 0, flags [DF], proto UDP (17), length 359)
192.168.1.1.ssdp > 239.255.255.250.ssdp: [udp sum ok] UDP, length 331
14:50:00.197282 IP (tos 0x0, ttl 1, id 0, offset 0, flags [DF], proto UDP (17), length 320)
192.168.1.1.ssdp > 239.255.255.250.ssdp: [udp sum ok] UDP, length 292
14:50:01.275923 IP (tos 0x0, ttl 64, id 14711, offset 0, flags [DF], proto TCP (6), length 52)
192.168.1.102.36614 > sof02s18-in-f33.1e100.net.https: Flags [.], cksum 0x8958 (correct), seq 3122062306, ack 2706027691, win 1444, options [nop,nop,
TS val 45552128 ecr 2919903715], length 0
14:50:01.275981 IP (tos 0x0, ttl 64, id 31657, offset 0, flags [DF], proto TCP (6), length 52)
192.168.1.102.42908 > sof02s18-in-f46.1e100.net.https: Flags [.], cksum 0x42fc (correct), seq 2715452946, ack 4244149933, win 1444, options [nop,nop,
TS val 45552128 ecr 3055914973], length 0
14:50:01.277676 IP (tos 0x0, ttl 64, id 63131, offset 0, flags [DF], proto UDP (17), length 72)
192.168.1.102.37669 > home-77-70-13-1.megalan.bg.domain: [udp sum ok] 9446+ PTR? 33.212.58.216.in-addr.arpa. (44)
14:50:01.280412 IP (tos 0x0, ttl 58, id 43610, offset 0, flags [none], proto TCP (6), length 52)
sof02s18-in-f33.1e100.net.https > 192.168.1.102.36614: Flags [.], cksum 0xe699 (correct), seq 1, ack 1, win 472, options [nop,nop,TS val 2919949791 e
cr 45483150], length 0
14:50:01.280949 IP (tos 0x0, ttl 59, id 23273, offset 0, flags [none], proto TCP (6), length 52)
sof02s18-in-f46.1e100.net.https > 192.168.1.102.42908: Flags [.], cksum 0x9ef1 (correct), seq 1, ack 1, win 358, options [nop,nop,TS val 3055961049 e
cr 45483596], length 0
14:50:01.283832 IP (tos 0x0, ttl 61, id 37950, offset 0, flags [none], proto UDP (17), length 493)
home-77-70-13-1.megalan.bg.domain > 192.168.1.102.37669: [udp sum ok] 9446 q: PTR? 33.212.58.216.in-addr.arpa. 2/6/11 33.212.58.216.in-addr.arpa. PTR
sof02s18-in-f33.1e100.net., 33.212.58.216.in-addr.arpa. PTR sof02s18-in-f1.1e100.net. ns: 216.in-addr.arpa. NS x.arin.net., 216.in-addr.arpa. NS z.arin.
net., 216.in-addr.arpa. NS y.arin.net., 216.in-addr.arpa. NS u.arin.net., 216.in-addr.arpa. NS r.arin.net., 216.in-addr.arpa. NS arin.authdns.ripe.net. a
r: u.arin.net. A 204.61.216.50, u.arin.net. AAAA 2001:500:14:6050:ad::1, y.arin.net. A 192.82.134.30, y.arin.net. AAAA 2001:500:127::30, x.arin.net. A 19
9.71.0.63, x.arin.net. AAAA 2001:500:31::63, z.arin.net. A 199.212.0.63, arin.authdns.ripe.net. A 193.0.9.10, arin.authdns.ripe.net. AAAA 2001:67c:e0::10
, r.arin.net. A 199.180.180.63, r.arin.net. AAAA 2001:500:f0::63 (465)

```

Фигура 1.7: Изглед на потребителския интерфейс на tcpdump.

## Глава 2

# Проектиране на структурата на мрежов анализатор

## 2.1 Функционални изисквания към мрежов анализатор

### 2.1.1 Намиране и избиране на физически интерфейси

Преди да започне какъвто и да било анализ, мрежовия администратор трябва да получи списък с интерфейси поддържани от мрежовата карта. Списъкът трябва да съдържа освен имената на интерфейсите (напр. `eth0`), конфигурираният IP адрес, мрежова маска с цел по-лесното ориентиране в случай на голям списък. Анализатора трябва да предостави възможност на администратора за избор на конкретен интерфейс, който да бъде анализиран.

### 2.1.2 Интерпретиране на данни

Мрежовия анализатор трябва да може да интерпретира прихваната от преносвателната среда поредица от байтове. Тъй като типично тя е в двоичен вид, информацията за съдържанието на пренесения PDU трябва да е в разбираем от администратора вид, т.е. представена като символен низ. Фундаментална е функционалността за интерпретиране на най-често срещаните протоколи в TCP/IP стекът, а именно Ethernet, IP, TCP, UDP.

### 2.1.3 Филтриране на данни

Прихващането на трафик директно от преносвателната среда предполага огромно количество данни, особено в случай на голямо количество станции. Филтрирането позволява на администратора да се абстрахира от ненужния му трафик като настрои анализатора да приема конкретен тип трафик, напр. единствено IP трафик. В противен случай анализирането би отнело ненужно време и ресурси.

### 2.1.4 Отдалечен анализ

Посредством client-server архитектура, анализатора трябва да поддържа отдалечен анализ. Клиентът трябва да има възможност да задава команди на сървърът, а той да ги изпълнява, делегирайки част от тях на анализатора за изпълнение. Комуникацията между клиента и сървъра трябва да е full-duplex и в реално време с оглед принципа на работа на мрежовия анализатор.

### 2.1.5 Сигурност на отдалечения анализ

С оглед на допълнителна сигурност, анализатора трябва да използва библиотека за комуникация между сървърът и клиента поддържаща SSL, тъй като евентуалната липса на поддръжка на криптиран трафик би довела до eavesdropping на комуникацията между двете и следователно до анализирания трафик, достигащ до сървъра.

### 2.1.6 Споделяне на анализа

Анализатора трябва да има възможност за споделяне на анализа, т.е. да поддържа списък от клиенти, които едновременно да получават прихванатия трафик.

### 2.1.7 Сигурност на споделяне на анализа

Анализатора трябва да поддържа метод за автентикация, който позволява единствено авторизирани мрежови администратори да достъпват текущата сесия на анализ. Така потребител (независимо злонамерен или не), достигнал случайно до адреса на уеб сървър, не би имал възможност да проследи целия трафик на машината, което би било еквивалентно на огромна дупка в сигурността.

### 2.1.8 Графичен интерфейс и визуализация

Анализатора трябва да поддържа лесен за използване и интуитивен графичен интерфейс. При създаване на нова сесия на анализ, мрежовия администратор трябва да има възможност за въвеждане на парола, за избиране на физически интерфейс, за въвеждане на филтри и стартиране на анализатора. Прозорецът на сесията трябва да съдържа списък с прихванати пакети и кратка информация за тях, както и компонент показващ детайлното съдържание на избран пакет, представяйки цялата му структура в интуитивен за разбиране от начинаещ вид.

## 2.2 Съображения за избор на програмни средства и развойна среда

### 2.2.1 C++

Езикът C++ е език с общо предназначение, използван предимно в сферата на системното програмиране. Създаден през 1979г. от Бьорн Струструп, езикът е комбинация от механизмите за абстракция на Simula и бързината и ефективността на C. Отчитайки поддръжката на системно програмиране, програмния код написан на C++ лесно взаимодейства със софтуер написан на други езици. Тази необходимост от взаимодействие е отчетена още от началния етап на дизайна на езика и поддръжката на C, Assembler и Fortran не изисква допълнително процесорно време или преобразуване на структурите от данни. [Stroustrup \(2013\)](#) Тъй като `libc++` е имплементирана на C, а C++ предоставя широка гама от механизми за абстракция, езикът е натурален избор, взимайки предвид необходимото бързодействие при работата с високоскоростни мрежови връзки.

### 2.2.2 `libc++`

`libc++` е библиотека, предоставяща платформенно-независим API с цел елиминиране на нуждата от системно-зависими модули в приложенията на по-високо ниво, тъй като всяка операционна система имплементира свои собствени такива механизми. Откъм

операционни системи, `libpcap` се поддържа на повечето UNIX-базирани операционни системи — Linux, Solaris, BSD и др. Съществува и разновидност за Microsoft Windows, наречена **winpcap**. Взимайки предвид разновидността от операционни системи, чрез интерфейса си `libpcap` предоставя на мрежовия анализатор универсалност спрямо ОС на която се изпълнява, което го прави широко приложим.

### 2.2.3 WebSocket

WebSocket протокола предоставя ефикасно решение на проблема за комуникация в реално време, тъй като при него не е необходимо отварянето на множество HTTP връзки (например посредством `XMLHttpRequest` обекта, `<iframe>` или *long polling*). Предимствата на такъв тип комуникация са значими за ефективната реализация на отдалечен преглед на анализа в реално време.

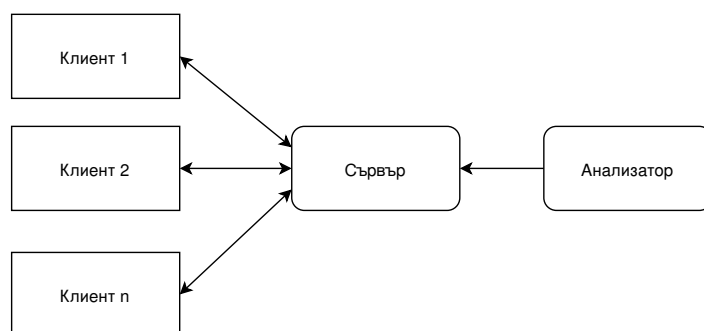
### 2.2.4 Angular 2

аргумент

## 2.3 Проектиране на архитектура на мрежов анализатор с отдалечен достъп

### 2.3.1 Цялостна архитектура

На Figure 2.1 е представен абстрактен поглед върху цялостната архитектура на приложението. Със стрелките е означен потока от данни между клиентите (мрежовите администратори), сървърът и анализаторът. След като е прихванат от преносвателната среда, пакетът се обработва, подава на сървър и бива изпратен до всички автентикирани клиенти (broadcast). Междувременно комуникацията между сървър и клиентите е двупосочна (full-duplex) — те могат да изпращат различни команди, например да се автентикират към сървър, да проверят дали съществува сесия, да стартират сесия и т.н.



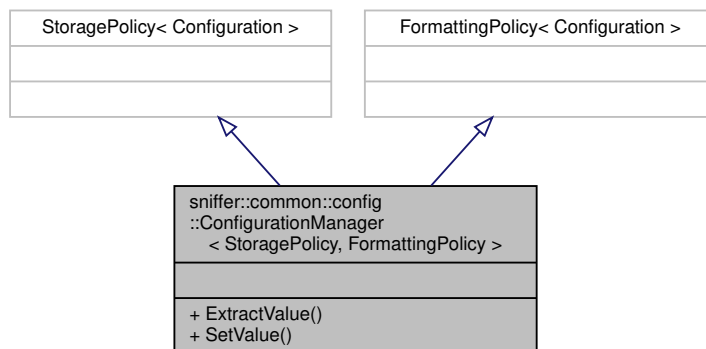
Фигура 2.1: Абстрактен поглед върху архитектурата.

Важно е да се отбележи, че приложението е многонишково: освен основната нишка, съществува нишка, обработваща събития на сървъра (например получаване на съобщение), както и нишка, която извършва реалния анализ.

### 2.3.2 Архитектура на общи класове

Цялостната архитектура на приложението не би била напълно описана без споменаването на двата класа `ConfigurationManager` и `SerializationManager`. Те се

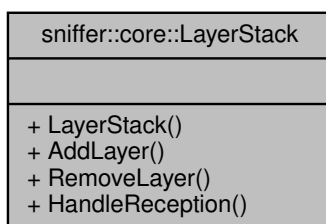
занимават съответно с конфигурацията на сървъра и със сериализацията на данните, готови за изпращане до клиента. Те представляват характерен пример за т.нар. *дизайн, базиран на политики (policy-based design)*, според когото биха се характеризирали като *host* класове. Парадигмата е еквивалентна на класическия **Strategy** шаблон за дизайн [Gamma \(1995\)](#), отчитайки предимството, че стратегиите при нея се определят по време на компилация, което намалява разхода на процесорно време и памет за извикване на виртуални методи. [Alexandrescu \(2001\)](#). На Figure 2.2 е представена йерархията на `ConfigurationManager`. Йерархията на `SerializationManager` е аналогична.



Фигура 2.2: UML диаграма на `ConfigurationManager` класа.

### 2.3.3 Архитектура и алгоритъм на мрежовия анализатор

В основата на архитектурата на мрежовия анализатор са класовете `LayerStack` и `Layer`. Двата класа представляват частично модифицирана имплементация на шаблона за дизайн *протоколен стек*<sup>1</sup>. Решението, освен разделяне (*decoupling*) на отговорността на слоевете, позволява динамична промяна на слоевете на стека, например ако е нужно 'подпъхване' на междинен слой между два вече съществуващи — имплементация, аналогична на двусвързан списък. Публичният интерфейс на `LayerStack` е описан на Figure 2.3

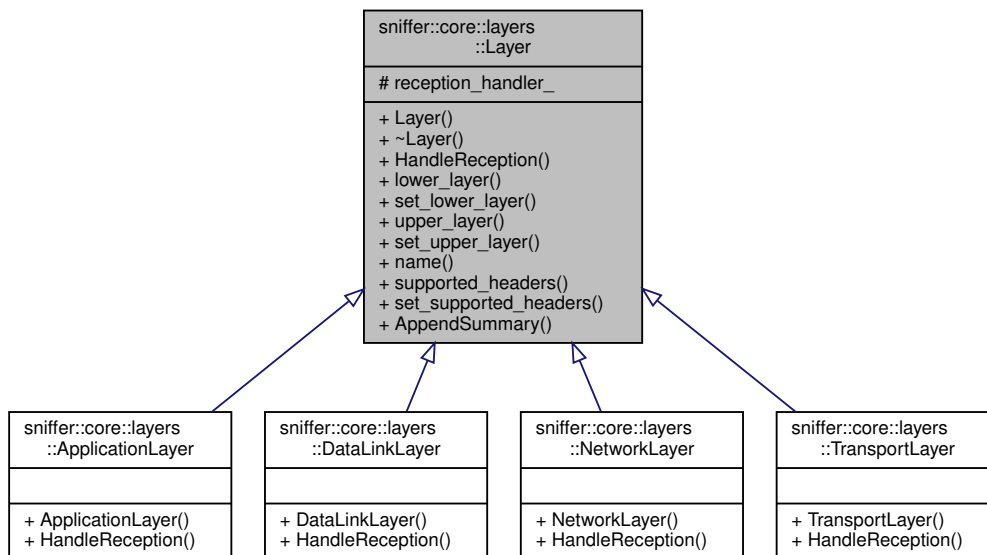


Фигура 2.3: UML диаграма на класа `LayerStack`.

`Layer` е базов клас за всички слоеве, както е представено на Figure 2.4. Индивидуалните слоеве се достъпват чрез указатели от този тип. Предимството е, че конкретния тип на горния и долния слой е неизвестен за имплементацията на даден слой.

Всеки обект от типа `Layer` има списък от поддържани `Header` обекти посредством обекти от типа `HeaderMetadata`. Например, този списък за `TransportLayer` би съдържал обекти от типовете `UserDataagramHeaderMetadata` и `TransmissionControlMetadata`. Всеки `HeaderMetadata` клас описва някаква метаинформация за `Header`, например дали съответния `Header` е с променлива дължина, името на `Header` класа, който да се инстанцира, минималната дължина на съответния `Header` обект и т.н. Всеки

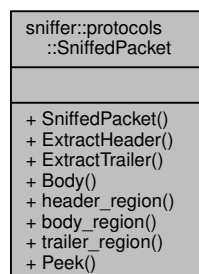
<sup>1</sup>[http://www.eventhelix.com/RealtimeMantra/PatternCatalog/protocol\\_stack.htm](http://www.eventhelix.com/RealtimeMantra/PatternCatalog/protocol_stack.htm)



Фигура 2.4: UML диаграма на класа **Layer** и наследниците му.

**HeaderMetadata** клас съдържа списък на съответствие между уникален идентификатор на конкретния **Header** и името на **Header** класа, за когото този идентификатор е валиден.

На Figure 2.5 е показан публичният интерфейс на **SniffedPacket** класа. Той представлява частично модифицирана имплементация на шаблона за дизайн *протоколен пакет*<sup>2</sup>, още известен като *протоколен буфер* или *многослоен буфер*. Взимайки предвид вече описания модел на протоколна комуникация, всеки слой добавя или премахва свой header или trailer. Това предразполага към имплементация, в която всеки слой заделя или освобождава нов буфер отчитайки новия размер. Изложения шаблон дава просто и ефективно решение на този проблем.



Фигура 2.5: UML диаграма на класа **SniffedPacket**

Буферът стандартно се разделя на три области: header, body и trailer. Алгоритъма на декапсулиране, представен визуално на Figure 2.6, е следния:

1. Полученият пакет се създава с всички байтове в body областта. В този начален момент, header и trailer областите имат нулева дължина.
2. Слой 1 изважда своите header и trailer области, двете области се 'отрязват' от body областта. Размера на body областта се намалява.
3. Слой 2 също изважда своите header и trailer области, двете области отново се 'отрязват' от body областта. Размера на body областта се намалява.

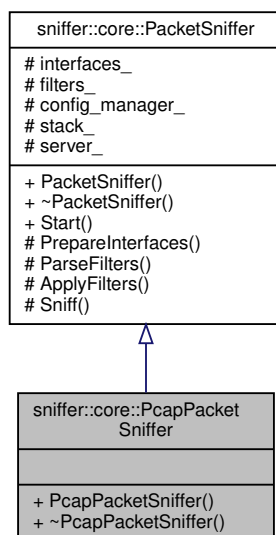
<sup>2</sup>[http://www.eventhelix.com/RealtimeMantra/PatternCatalog/protocol\\_packet\\_design\\_pattern.htm](http://www.eventhelix.com/RealtimeMantra/PatternCatalog/protocol_packet_design_pattern.htm)

- Аналогичен е процесът при трети слой, след който се получава изначалната body област.



Фигура 2.6: Диаграма на разпределение на областите при получаване на пакет.

Представен на Figure 2.7 е абстрактния клас `PacketSniffer` и имплементацията му `PcapPacketSniffer`. Абстракцията е необходима с цел поддръжка на библиотеки на ниско ниво различни от `libpcap`. Публичният интерфейс на класа е тривиален — метод, който стартира анализатора, като в имплементацията си извиква последователно частни виртуални методи за подготовка на физическите интерфейси, за интерпретиране на зададените филтри и прилагането им, което на практика имплементира шаблона за дизайн *шаблонен метод* (*template method*). [Gamma \(1995\)](#) Това позволява различните имплементации на мрежов анализатор да предефинират поведението на описаните методи необходими за стартиране. Всеки обект от типа `PacketSniffer` е композиран от обект от типа `LayerStack`. С оглед имплементацията на мрежов анализатор описана в първа глава, `LayerStack` е де факто аналогичен, паралелен на системния протоколен стек, но локален за мрежовия анализатор.



Фигура 2.7: UML диаграма на класа `PacketSniffer`.

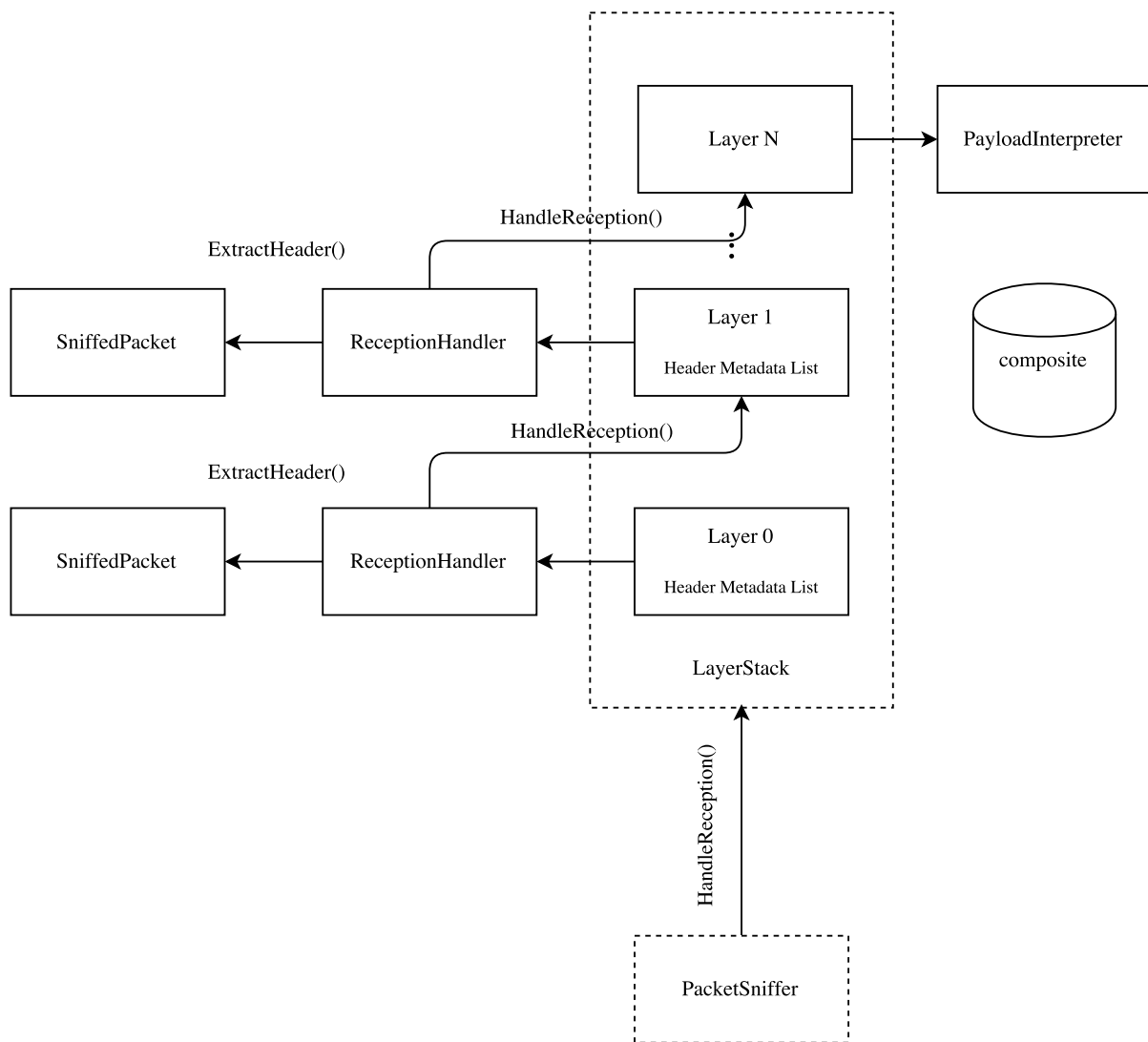
## Намиране на физически интерфейси за анализ

В основата на функционалността за намиране на физически интерфейси за анализ стои класът `InterfaceRetriever`. Публичният му интерфейс се състои от един метод — `Retrieve()`, който предоставя списък от интерфейси (`Interface` обекти) на извикващия. Всеки от тези интерфейси съдържа в себе си списък от `InterfaceAddress` обекти, представляващи съответните адреси за интерфейса, например настроеният IP адрес,

мрежова маска, IPv6 link-local адрес или broadcast адрес (аналогично на `ifconfig` командата в UNIX операционните системи).

`InterfaceRetriever` съдържа обект от типа `IpAddressFactory`. Целта на `IpAddressFactory` е да инстанцира обекти от типа `IpAddress` — типична имплементация на шаблона за дизайн *завод (factory)* [Gamma \(1995\)](#). Класът `IpAddress` е базов за класовете `Ipv4Address` и `Ipv6Address`, съответно енкапсулиращи данни за IPv4 и IPv6 адрес.

## Основен алгоритъм на работа



Фигура 2.8: Основен алгоритъм на работа.

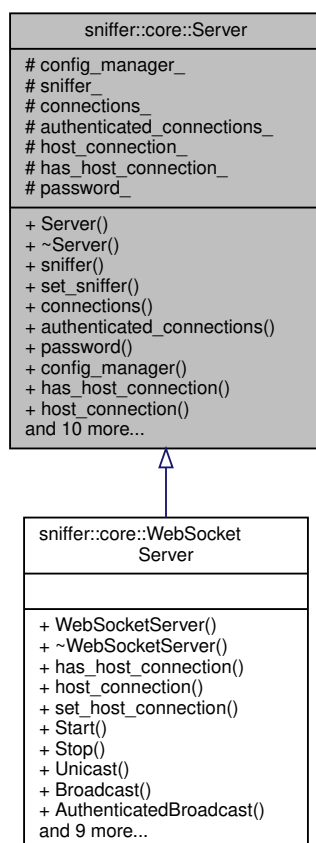
Основния алгоритъм се базира на вече описания принцип на работа на протоколната комуникация, както и на вече описаните основни класове. Класът `PacketSniffer` има *callback* метод, който се извиква от в момента на получаване на пакет по преносвателната среда. Инстанцира се обект от типа `SniffedPacket`, който бива предаден на протоколния стек. Той го препредава на най-ниския слой. При предаването, освен `SniffedPacket` обекта, се предават три параметъра — `composite`, който представлява дървовидна йерархична структура, позволяваща 'наслаждане' на сериализираните полета на всеки `Header`, името на предишния `Header` обект, обработил пакета, и текущия уникален идентификационен номер, извлечен от предишния `Header`. `ReceptionHandler` обекта



проверява дали на текущия слой съществува **HeaderMetadata** със зададено съответствие между името на предишния **Header** обект и текущия уникален идентификационен номер. Ако съществува, инстанцира обект от типа **Header**, който извлича себе си от **SniffedPacket** обекта, сериализира го и го наслагва върху **composite** обекта, препредавайки същите параметри на горния слой, където процеса е аналогичен. На последния слой същинските данни, енкапсулирани от слоевете, се интерпретират в някакъв формат и също се наслагват върху **composite** обекта, който бива изпратен до всички автентикирани клиенти на сървъра. Ако не съществува, **SniffedPacket** обекта се определя като невалиден и понататъшната обработка се прекъсва.

### 2.3.4 Архитектура на сървър

В ядрото на сървърната част на цялостната архитектура е **Server** класа. Неговият интерфейс описва абстрактна функционалност за добавяне на нови връзки, за множествоно изпращане (broadcast), за автентикация, за поддръжка на host на сесията (т.е. мрежовия администратор, иницирал сесия на анализ). Абстракцията на този клас предполага лесна подмяна на **WebSocketServer** класа, представен на Figure 2.9, с друг тип сървър.

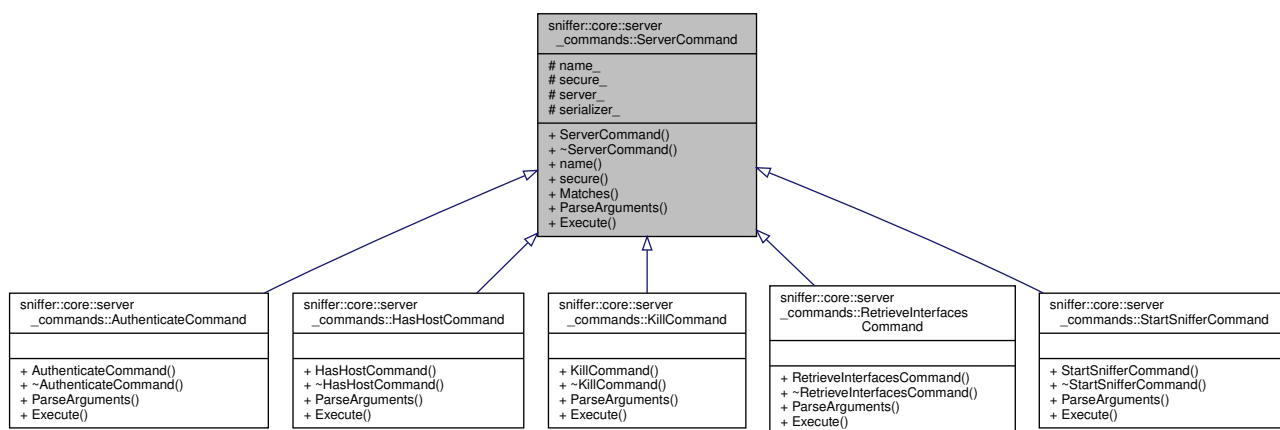


Фигура 2.9: UML диаграма на класовете **Server** и **WebSocketServer**.

### Поддръжка на команди

Комуникацията между клиентите и сървъра може да бъде разгледана като команди, които клиентите изпращат и биват съответно изпълнени върху сървъра. Това предполага имплементацията на класическия шаблон *команда* (*command*). С това се цели разделяне (decoupling) на обекта, извикващ изпълнението (**ServerCommandInvoker**) и обекта, действително изпълняващ командата (**Command**). Това улеснява добавянето на нови

команди чрез просто наследяване на **Command** класа, както изложено на Figure 2.10. Gamma (1995).



Фигура 2.10: UML диаграма на класа **ServerCommand** и наследниците му.

## Модели на отговор при изпълнени команди

Изпълняването на дадена команда върху сървъра предполага отговор за състоянието след изпълнение на командата към клиента. С оглед разделянето на отговорности, тези класове са обособени в *модели на отговор* (*response models*). *Модел* тук може да се разглежда като аналог на моделите при архитектурните шаблони за дизайн от типа на MVC, MVVM и т.н. Всеки от тях описва някакво състояние, например **RetrieveInterfacesResponseModel** има списък от **Interface** обекти.

Тривиално, те поддържат метод за сериализацията на модела на отговора с оглед необходимостта от пренасянето на данните по канала между клиента и сървъра.

### 2.3.5 Архитектура на клиент

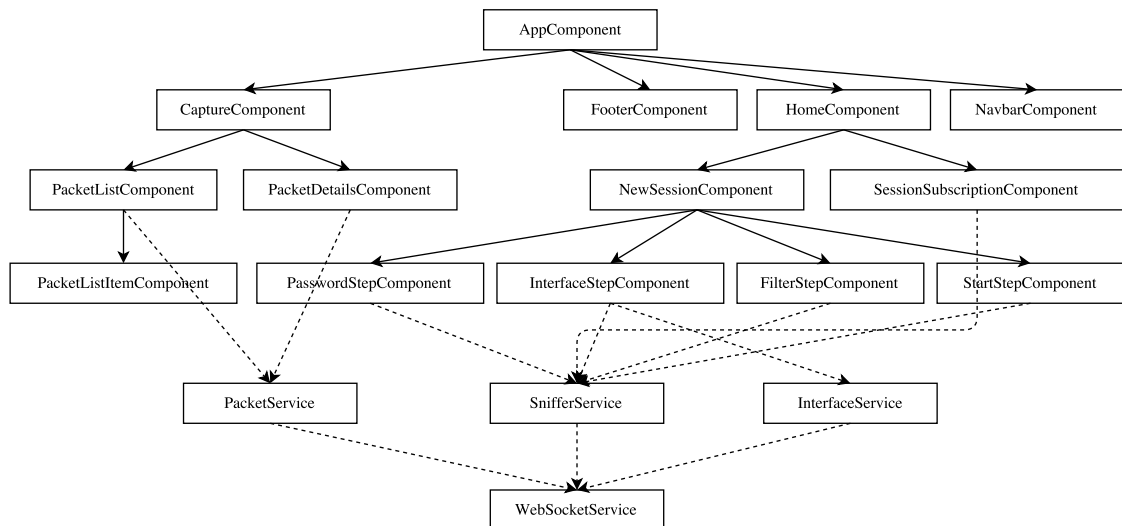
#### Компонентно дърво

Базирайки се на компонентната архитектура на Angular 2, архитектурата на приложението има следния йерархичен вид:

Потребителския интерфейс е съставен от два основни компонента: **HomeComponent** и **CaptureComponent**. **HomeComponent** има две деца: **NewSessionComponent** и **SubscriptionComponent**. Първото представлява компонента за създаване на нова сесия на анализ, т.е. когато клиентът е установил, че никой друг не е иницирал такава. То поддържа подстъпки за въвеждане на парола, избиране на интерфейси, въвеждане на филтър и стартиране на сесията на анализ, които са енкапсулирани в отделни компоненти. **SubscriptionComponent** компонента се инициализира в случаите, когато сесия на анализ съществува и втория (в общия случай *n*-тия) мрежов администратор я достъпва. Стандартно този компонент предоставя възможност за въвеждане на парола.

**CaptureComponent** представлява реалния екран на анализ, т.е. като подкомпоненти поддържа списък с пакетите, които са получени от сървъра (и съответно анализатора), и детайли за избран пакет. Екрана е почти сходен на основния екран в Wireshark.

На Figure 2.11 също са маркирани (с прекъсната линия) зависимостите на компонентите от съответните *услуги* (*services*). Услугите енкапсулират т.нар. *бизнес логика* като съответно биват разделени (decoupled) от генерирането на потребителския интерфейс. Това улеснява компонентното тестване на всеки от тях, както и подмяната им (mocking).



Фигура 2.11: Компонентно дърво на потребителския интерфейс и зависимости на компонентите от услугите.

Ядрото на клиента е `WebSocketService` услугата. Тя енкапсулира извикванията към `WebSocket API` на браузъра в удобен интерфейс. От нея зависят услугите, от които клиентът достъпва получените пакети (`PacketService`) и физическите интерфейси (`InterfaceService`). `PacketService` също поддържа възможност за избиране на 'наблюдаван пакет', който например се настройва когато администратора избере един пакет от листа. `SnifferService` представлява абстракция върху командите, които могат да се изпълнят на сървъра, т.е. метода `retrieveInterfaces()`, изпълнен върху обект от тип `SnifferService`, би изпратил команда за намиране и извличане на физическите интерфейси. Отговора от командата би бил наличен през `InterfaceService`.

## Глава 3

# Програмна реализация на мрежов анализатор

### 3.1 Имплементация на общи класове ConfigurationManager и SerializationManager

#### 3.1.1 Имплементация на клас, енкапсулиращ конфигурация

Конфигурацията е имплементирана чрез класът `Configuration`, чиято единствена функционалност е да енкапсулира съдържанието на конфигурацията във вида на символен низ.

Listing 3.1: Дефиниция на `Configuration` класа

```
1 class Configuration {
2     public:
3         explicit Configuration(const std::string& data);
4
5         std::string data() const;
6
7     private:
8         std::string data_;
9 };
```

Имплементацията на класа е тривиална, съдържаща инициализация на частното поле, както и функция за достъпването на полето (getter), поради което не е представена.

#### 3.1.2 Имплементация на `ConfigurationManager`

Класът `ConfigurationManager` е *host* клас за различни политики. Целта му е да предостави функционалност за извличане и задаване на стойност от/на конфигурация. Типично, една конфигурация представлява йерархична структура от вида:

```
1 {
2     "sniffer": {
3         "snap_len": 1518
4     },
5     "server": {
6         "port": 1903,
7         "password": "foo",
8     }
9 }
```

В случая е използван **JSON (Javascript Object Notation)** форматът, но в действителност гъвкавата имплементация позволява използването на аналогичен йерархичен формат.

Следвайки класическата имплементация на подобен тип класове, класът приема две политики като аргументи на шаблона: **StoragePolicy** и **FormattingPolicy**. **StoragePolicy** дефинира начина, по който конфигурацията се запазва — във файл, в мрежата и т.н. **FormattingPolicy** дефинира формата на конфигурационния файл — JSON, XML и т.н.

Класът има два метода: **ExtractValue()** и **SetValue()**. Методът **ExtractValue()** приема три аргумента: тип на стойността, която да прочете, името на обекта и името на ключа, от които да я извади. След като вземе пътят към ресурса от политиката за запазване на конфигурацията и го конкатенира с характерното разширение за политиката на форматиране на конфигурацията, конфигурацията се чете от политиката за запазване. Като резултат от метода се извиква политиката за форматиране, към която се делегират параметрите.

Listing 3.2: Имплементация на **ExtractValue()** метода

```
1  template <typename U>
2  U ExtractValue(const std::string& object, const std::string& key) const {
3      auto resource_path = StoragePolicy<Configuration>::resource_path() +
4                          FormattingPolicy<Configuration>::extension();
5
6      auto configuration = StoragePolicy<Configuration>::Read(resource_path);
7
8      return FormattingPolicy<Configuration>::template ExtractValue<U>(
9          configuration, object, key);
10 }
```

Методът **SetValue()** има три параметъра: типа на стойността, която да запише, името на обект и името на ключът, в които да се запише самата стойност. След като вземе пътят към ресурса от политиката за запазване на конфигурацията и го конкатенира с характерното разширение за политиката на форматиране на конфигурацията, старата конфигурация се чете от политиката за запазване. Новата конфигурация се генерира посредством политиката на форматиране и базирайки се на старата конфигурация. Накрая се запазва посредством политиката на запазване.

Listing 3.3: Имплементация на **SetValue()** метода

```
1  template <typename U>
2  void SetValue(const std::string& object, const std::string& key,
3               U value) const {
4      auto resource_path = StoragePolicy<Configuration>::resource_path() +
5                          FormattingPolicy<Configuration>::extension();
6
7      auto old_config = StoragePolicy<Configuration>::Read(resource_path);
8
9      auto new_config = FormattingPolicy<Configuration>::template SetValue<U>(
10         old_config, object, key, value);
11
12     StoragePolicy<Configuration>::Write(new_config, resource_path);
13 }
```

### 3.1.3 Имплементация на **FileStoragePolicy**

Класът **FileStoragePolicy** представлява политика за запазване на обект от даден тип **T** (в случая **Configuration**) на файловата система. Той има три метода: **Read()**, **Write()** и **resource\_path()**. **Read()** метода отваря файла във входен поток, чете съдържанието

му, с което инстанцира обект от типа `T` (в случая `Configuration`). `Write()` метода отваря изходен поток към файла, чете съдържанието на обекта от тип `T` и го пренасочва към изходния поток. Накрая затваря потока. `resource_path()` връща пътя към файла, в който да се запази обекта. Методите използват стандартните потоци за вход и изход, затова и не са представени.

### 3.1.4 Имплементация на `JsonFormattingPolicy`

Класът `JsonFormattingPolicy` представлява политика за форматиране на обекти от тип `T`, в случая `Configuration`, в JSON. Той е своеобразен *wrapper* клас върху JSON библиотеката `nlohmann/json`, към която делегира същинската обработка. Има два метода: `ExtractValue()` и `SetValue()`.

`ExtractValue()` извлича стойност от даден обект (в случая `Configuration`) като го интерпретира в обект от типа `nlohmann::basic_json`, върху когото прилага `operator[]()`<sup>1</sup> и шаблонния метод `.get<U>()`<sup>2</sup>.

Listing 3.4: Имплементация на `ExtractValue()` метода на `JsonFormattingPolicy`

```
1  template <typename U>
2  U ExtractValue(const T& config, const std::string& object,
3                const std::string& key) const {
4      auto json_obj = nlohmann::json::parse(config.data());
5      return json_obj[object][key].template get<U>();
6  }
```

`SetValue()` задава стойност на даден обект (в случая `Configuration`), като го интерпретира в обект от типа `nlohmann::basic_json`, върху когото прилага вече споменатият `operator[]()`, присвоявайки стойността.

Listing 3.5: Имплементация на `SetValue()` метода на `JsonFormattingPolicy`

```
1  template <typename U>
2  T SetValue(const T& config, const std::string& object, const std::string& key,
3            U value) const {
4      auto json_obj = nlohmann::json::parse(config.data());
5
6      json_obj[object][key] = value;
7
8      T config_obj{json_obj.dump()};
9      return config_obj;
10 }
```

### 3.1.5 Имплементация на клас, енкапсулиращ сериализиран обект

Концепцията за сериализиран обект е имплементирана чрез класът `SerializedObject`, чиято единствена функционалност е да енкапсулира съдържанието на обекта във вида на символен низ. Имплементацията на класа е тривиална: съдържаща инициализация на частното поле, както и методи за достъпването на полето, поради което не е представена.

<sup>1</sup>[https://nlohmann.github.io/json/classnlohmann\\_1\\_1basic\\_\\_json\\_a233b02b0839ef798942dd46157cc0fe6.html](https://nlohmann.github.io/json/classnlohmann_1_1basic__json_a233b02b0839ef798942dd46157cc0fe6.html)

<sup>2</sup>[https://nlohmann.github.io/json/classnlohmann\\_1\\_1basic\\_\\_json\\_a16f9445f7629f634221a42b967cdcd43.html](https://nlohmann.github.io/json/classnlohmann_1_1basic__json_a16f9445f7629f634221a42b967cdcd43.html)

### 3.1.6 Имплементация на SerializationManager

Класът `SerializationManager` е *host* клас за различни политики. Целта му е да предостави общ интерфейс за сериализация на обекти: създаване на обект, задаване на свойствата му, добавяне на други обекти и т.н.

Следвайки класическата имплементация на подобен тип класове, класът за момента приема една политика като аргумент на шаблона: `SerializationPolicy`, която дефинира конкретния начин на сериализация на обектите — JSON, XML и т.н.

Имплементацията на класа включва единствено делегиране на функционалността си към `SerializationPolicy` класа, затова и не е представена. Обяснение на принципа на работа на методите на класа е изложено в имплементация на `SerializationPolicy` — `JsonSerializationPolicy`.

#### Имплементация на JsonSerializationPolicy

Класът `JsonSerializationPolicy` представлява политика за сериализация на обекти от тип `T`, в случая `SerializedObject`. Той е своеобразен *wrapper* клас върху JSON библиотеката `nlohmann/json`, към която делегира същинската обработка. Методите `Create()`, `ObjectExists()`, `IsEmpty()` представляват тривиални методи за създаване на нов обект, проверка дали свойство на обект съществува и дали на обект липсват свойства. Методите `ExtractValue()` и `SetValue()` представляват подобни по функционалност методи на вече описаните такива в `JsonFormattingPolicy`. Методът `SetObject()` дава функционалност за вложен обект, т.е един `SerializedObject` може да бъде добавен като свойство на родителски `SerializedObject`. `AppendObject()` има аналогична функционалност с разликата, че при него вложения `SerializedObject` се добавя към свойство на родителския обект от тип масив.

По-интересна е имплементацията на методите `AppendVariableDepthObject()` и `InsertObjectAtNthDepth()`. Спрямо обяснения основен алгоритъм на реализация, обработвания пакет се 'изкачва' по слоевете на стека. Тези методи служат за генериране на дървовидната структура на `composite` обекта, представляващ сериализирания вариант на пакета.

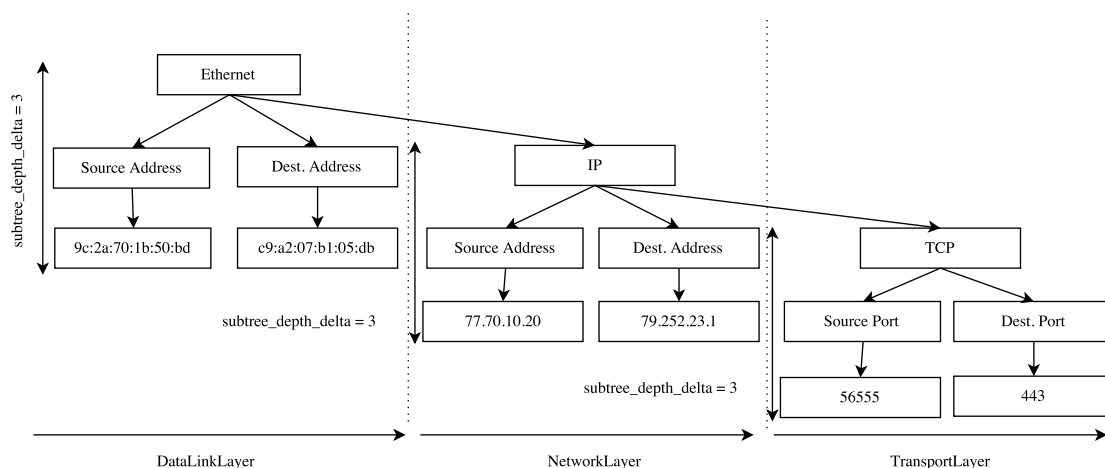
В `AppendVariableDepthObject()` се извиква `InsertObjectAtNthDepth()`, където всъщност се имплементира логиката по добавяне на обекта.

Listing 3.6: Имплементация на `AppendVariableDepthObject()` метода

```
1 void AppendVariableDepthObject(const std::string& root_key_name,
2                               const std::string& children_key_name,
3                               int subtree_depth_delta,
4                               const T& serialized_obj,
5                               T* serialized_parent_obj) const {
6     auto json_pobj = nlohmann::json::parse(serialized_parent_obj->data());
7     auto json_obj = nlohmann::json::parse(serialized_obj.data());
8
9     InsertObjectAtNthDepth(children_key_name, subtree_depth_delta, json_obj,
10                           &(json_pobj[root_key_name]));
11
12     serialized_parent_obj->set_data(json_pobj.dump());
13 }
```

В `InsertObjectAtNthDepth()` се имплементира рекурсивно обхождане на дървото, като при всяко рекурсивно извикване на функцията се калкулира височината му. Целта е обекта да се запише в списъка с деца на обекта, намиращ се на разстояние `subtree_depth_delta` от листото на дървото. На Figure 3.1 е представена проста визуализация на процеса. Например, IP обекта би се записал като дете на Ethernet обекта,

тъй като в действителност **Ethernet** енкапсулира IP. Генерирането на такъв тип структура впоследствие позволява удобна визуализация на структурата на цялостния frame.



Фигура 3.1: Процес на добавяне на нов Header обект към вече изграденото дърво.

Описания процес има следната имплементация:

Listing 3.7: Имплементация на InsertObjectAtNthDepth() метода

```

1  int InsertObjectAtNthDepth(const std::string& children_key_name ,
2                             int subtree_depth_delta ,
3                             nlohmann::json obj_to_insert ,
4                             nlohmann::json* root) const {
5      if (root->empty()) {
6          return 0;
7      }
8
9      int height = 0;
10
11     for (auto& obj : (*root)[children_key_name]) {
12         height = std::max(
13             height , InsertObjectAtNthDepth(children_key_name , subtree_depth_delta ,
14                                             obj_to_insert , &obj));
15     }
16
17     if (height + 1 == subtree_depth_delta) {
18         (*root)[children_key_name].push_back(obj_to_insert);
19     }
20
21     return height + 1;
22 }

```

## 3.2 Имплементация на мрежов анализатор

### 3.2.1 Имплементация на намиране на физически интерфейси

#### Имплементация на InterfaceRetriever

В представената дефиниция класа има само един метод — `Retrieve()`, който е публичен виртуален метод. Целта му е конкретната имплементация на наследника да определи по какъв начин се намират интерфейсите на ниво операционна система.

Listing 3.8: Дефиниция на InterfaceRetriever класа



```

1 class InterfaceRetriever {
2 public:
3     explicit InterfaceRetriever(
4         const sniffer::common::addressing::IpAddressFactory& factory)
5         : ip_addr_factory_{factory} {}
6
7     virtual std::vector<Interface> Retrieve() = 0;
8
9 protected:
10     sniffer::common::addressing::IpAddressFactory ip_addr_factory_;
11 };

```

## Имплементация на IpAddressFactory

В представената имплементация на `IpAddressFactory` методът `Parse()` получава указател към структура от тип `sockaddr`. В зависимост от стойността на полето `sa_family` се инстанцира правилния обект, съответно енкапсулиращ IPv4 или IPv6 адрес. Методът връща `std::unique_ptr`, тъй като създавания `IpAddress` обект е отговорен за неговото унищожаване. [Meyers \(2014\)](#)

Listing 3.9: Дефиниция на `IpAddressFactory` класа

```

1 std::unique_ptr<IpAddress> IpAddressFactory::Parse(struct sockaddr* sockaddr) {
2     if (sockaddr) {
3         switch (sockaddr->sa_family) {
4             case AF_INET:
5                 return std::make_unique<Ipv4Address>(sockaddr);
6             case AF_INET6:
7                 return std::make_unique<Ipv6Address>(sockaddr);
8             default:
9                 break;
10        }
11    }
12
13    return nullptr;
14 }

```

## Имплементация на IpAddress

`IpAddress` представлява интерфейс съставен от един метод: `data()`, връщащ указател към началото на масив от символи в паметта, представляващи четим от човек формат на IP адрес (най-често *dot-decimal* нотация).

Listing 3.10: Дефиниция на `IpAddress` интерфейса

```

1 class IpAddress {
2 public:
3     virtual ~IpAddress() {}
4
5     virtual const char* data() const = 0;
6 };

```

Двете му имплементации: `Ipv4Address` и `Ipv6Address` съответно енкапсулират IPv4 и IPv6 формат на адрес. Тъй като имплементациите са сходни са представена само нетривиалната част от имплементацията на `Ipv4Address`. Интерфейсът представлява класически интерфейс на *RAII* (*Resource Acquisition Is Initialization*) клас: конструктор, деструктор, копиращ конструктор и *copy assignment*. [Stroustrup \(2013\)](#). Имплементацията на последните три е стандартна и следователно не е представена. В конструктора се

извършва *typecast* на подадения адрес на `sockaddr` структура към `sockaddr_in`. След извикването на полулярната `inet_ntop` функция, указателят `buffer_` сочи към масив от символи, съдържащ IPv4 адреса в *dot-decimal* формат.

Listing 3.11: Имплементация на конструктора на `Ipv4Address` класа

```
1 Ipv4Address::Ipv4Address(struct sockaddr* sockaddr) {
2     struct sockaddr_in* addr_in = (struct sockaddr_in*)sockaddr;
3     buffer_ = new char[INET_ADDRSTRLEN];
4     inet_ntop(AF_INET, &(addr_in->sin_addr), buffer_, INET_ADDRSTRLEN);
5 }
```

Разликата между `Ipv4Address` и `Ipv6Address` е в размера на буфера, който се заделя в динамичната памет, както и типа на структурата, към която се извършва *typecast* на `sockaddr` указателя.

## Имплементация на `InterfaceAddress`

### Имплементация на `Interface`

refactor  
and  
explain

### Имплементация на `PcapInterfaceRetriever`

refactor  
and  
explain

Класът `PcapInterfaceRetriever` е конкретна имплементация на вече описания `InterfaceRetriever` за намиране на интерфейси посредством `libpcap` библиотеката. В имплементацията на метода `Retrieve()`, след намирането на поддържаните мрежови интерфейси от операционна система посредством `pcap_findalldevs()`, характерните за `libpcap` структури `pcap_if_t` и `pcap_addr_t`, които абстрактно представляват едносвързан списък, се итерират и се създават съответните `Interface` и `InterfaceAddress` обекти (последните посредством `IpAddressFactory`).

## 3.2.2 Имплементация на `LayerStack` и `Layer` класове

### Имплементация на `LayerStack`

Дефиницията на `LayerStack` класа включва методи за добавяне на слой, за премахване на слой и за препредаване на получен пакет 'нагоре' по стека, енумерацията `Position`, дефинираща къде да бъде поставен даден слой, и частни полета, пазещи указатели съответно към най-високия и най-ниския слой на стека.

Listing 3.12: Дефиниция на `LayerStack` класа

```
1 class LayerStack {
2     public:
3         enum class Position { TOP, ABOVE, BELOW };
4
5         LayerStack();
6
7         void AddLayer(layers::Layer* layer, layers::Layer* existing = nullptr,
8                     Position pos = Position::TOP);
9
10        void RemoveLayer(layers::Layer* layer);
11
12        void HandleReception(std::string prev_header_name, int current_header_id,
13                            sniffer::protocols::SniffedPacket* packet,
```

```

14         sniffer::common::serialization::SerializedObject* composite);
15
16     private:
17         layers::Layer* highest_layer_;
18         layers::Layer* lowest_layer_;
19     };

```

Методът `AddLayer()` добавя слой в стека, като ако такъв не съществува, новият бива определен като най-високия и най-ниския. Ако такъв съществува, се взима предвид исканата позиция и съществуващия слой, според които новият слой се позиционира. Имплементацията е аналогична на имплементацията на двусвързан списък.

Методът `RemoveLayer()` премахва слой от стека, като при премахването свързва слоя над премахнатия с този, който е под него.

Методът `HandleReception()` има проста имплементация: той предава получения пакет, както и допълнителните параметри, описани в основния алгоритъм, на най-ниския слой.

## Имплементация на Layer

Дефиницията на `Layer` класа включва метод за получаване на пакет на слоя (`HandleReception()`), методи за задаване на горния и долния слой, метод, връщаш поддържаните `Header` обекти от слоя (`supported_headers()`), и метод, грижещ се за наслабяването на обобщена информация на полетата на даден `Header` (`AppendSummary()`) (обобщената информация впоследствие се използва при показване на листа с пакети на клиента).

## Имплементация на Layer класове

С оглед изискванията за поддържани протоколи до момента, мрежовия анализатор има имплементирани четири слоя — `DataLinkLayer`, `NetworkLayer`, `TransportLayer` и `ApplicationLayer`. Имплементацията на първите три е сходна, затова е представена имплементацията на `DataLinkLayer`.

Listing 3.13: Дефиниция на `DataLinkLayer` класа

```

1 class DataLinkLayer : public Layer {
2     public:
3         DataLinkLayer(
4             const std::string& name,
5             const sniffer::common::serialization::SerializationMgr& serializer);
6
7         void HandleReception(
8             std::string prev_header_name, int current_header_id,
9             sniffer::protocols::SniffedPacket* packet,
10            sniffer::common::serialization::SerializedObject* composite) override;
11 };

```

Класът освен конструктор приемащ името на слоя и `SerializationManager` класа, който да се използва, има един единствен метод: `HandleReception()`. Неговата единствена функция е да делегира приемането на пакета към `ReceptionHandler` класа, където се случва реалната обработка. Именно това е методът, който `ReceptionHandler` обекта на долния слой извиква, когато иска да обработи `Header` предназначен за горния слой.

По-интересна е имплементацията на `ApplicationLayer` класа. Тъй като стандартно той е най-горния слой на стека, в него се случва обработката на полезната информация, носена от пакета (payload). Като зависимост (dependency) конструкторът на приложния слой приема `PayloadInterpreter`, чиито метод `Interpret()` извиква за да интерпретира съдържанието на полезната информация. След което тя се сериализира и съответно наслабва върху `composite` обекта.

Listing 3.14: Имплементация на `HandleReception()` метода на `ApplicationLayer`

```

1 void ApplicationLayer::HandleReception(
2     std::string prev_header_name, int current_header_id,
3     sniffer::protocols::SniffedPacket* packet,
4     sniffer::common::serialization::SerializedObject* composite) {
5     int payload_length = packet->payload_length();
6
7     if (payload_length > 0) {
8         auto serializer = reception_handler_.serializer();
9         auto payload_obj = serializer.CreateObject();
10
11         auto interpreted_payload =
12             interpreter_->Interpret(packet->Body(), payload_length);
13
14         serializer.SetValue<std::string>("contents", interpreted_payload, &payload_obj);
15         serializer.SetValue<int>("length", payload_length, &payload_obj);
16         serializer.SetObject("payload", payload_obj, composite);
17     }
18 }
19 }

```

### 3.2.3 Имплементация на `SniffedPacket` класа

Както вече описано, `SniffedPacket` класа е в основата на декапсулацията на получените по преносвателната среда пакети. Това е имплементирано чрез структура, описваща обяснените вече области, дефиницията на която включва полетата `offset` и `length`, съответно представляващи изместване на областта и дължината ѝ:

Listing 3.15: Дефиниция на `PacketRegion`

```

1 struct PacketRegion {
2     PacketRegion();
3     PacketRegion(int offset, int length);
4
5     int offset;
6     int length;
7 };

```

По-нетривиалните методи на класа `SniffedPacket` са `ExtractHeader()`, `ExtractTrailer()` и `Peek()`. `ExtractHeader()` се занимава с извличането на заглавна област, като съответно формира заглавна област, намалява размера на `body` областта и увеличава изместването ѝ. Аналогично, `ExtractTrailer()` намалява размера на `body` областта и формира `trailer`, започващ от сумата на изместването на `body` областта и дължината ѝ. Методът `Peek()` 'наднича' `n` байта в `body` областта и връща указател, сочещ след тези `n` байта. Той се използва за извличане на дължините на `header`-и с променлива дължина, т.е. такива, при които дължината е кодирана като поле на `header`-а, преди да бъде инстанциран конкретния `Header` клас.

### 3.2.4 Имплементация на `ReceptionHandler`

Класът `ReceptionHandler` се грижи за изкачването на пакета 'нагоре' по слоевете и следователно е в центъра на основния алгоритъм. Методът `Handle()` имплементира основната част от логиката. Първоначално той проверява дали текущия слой поддържа `Header` клас, проверявайки дали на слоя съществуват метаданни с подадените (от долния слой) `prev_header_name` и `current_header_id`. В случай, че се поддържа, от метайнформацията се взима името на класа на конкретния наследник на `Header`,

изчислява се дължината му (ако е с променлива дължина, тя се взима чрез `SniffedPacket::Peek()`), проверява се дали изчислената дължина не е по-малко от допустимата, кодирана в метайнформацията. В случай че е, конкретното PDU бива изхвърлено и понататъжния процес на декапсулация спира. В случай, че не е, се инстанцира конкретен `Header` обект, полетата му се сериализират и биват добавени към `composite` дървото чрез `AppendVariableDepthObject`. В случай че инстанцирания `header` поддържа обобщен вариант на някои негови полета, той също се добавя към `composite` обекта чрез `AppendSummary()` метода на `Layer`. В края на процеса, ако слой за когото е създаден `ReceptionHandler` обекта има указател към слой над него, пакета продължава пътят си 'нагоре' по стека като се извиква `HandleReception()` метода на горния слой и му се подават името на текущия `Header` и ID на следващия.

### 3.2.5 Имплементация на PacketSniffer и PcapPacketSniffer

#### Имплементация на PacketSniffer

`PacketSniffer` представлява абстрактен клас за мрежов анализатор. В конструктора той получава необходимите зависимости (dependencies) за анализатора — `LayerStack`, който да ползва при анализирането на пакетите, указател към `Server` обекта, интерфейс, филтър и т.н. `Start()` методът има следната експресивна имплементация:

Listing 3.16: Имплементация на `Start()` метода на `PacketSniffer`

```

1 void PacketSniffer::Start() {
2     PrepareInterfaces();
3
4     if (!filter_.empty()) {
5         ParseFilters();
6         ApplyFilters();
7     }
8
9     Sniff();
10 }
```

Методите `PrepareInterfaces()`, `ParseFilter()` и `ApplyFilters()` и `Sniff()` представляват виртуални методи, поведението на които конкретните имплементации на мрежов анализатор презаписват (override).

#### Имплементация на PcapPacketSniffer

`PcapPacketSniffer` класа имплементира виртуалните методи на `PacketSniffer` чрез `libpcap`. Методът `PrepareInterfaces()` първо намира IPv4 адреса и мрежовата маска за зададения физически интерфейс чрез `pcap_lookupnet()`, а след това инициализира сесия на анализ чрез `pcap_open_live()` функцията, която има следния прототип:

```

1 pcap_t *pcap_open_live(char *device, int snaplen, int promisc, int to_ms,
2     char *ebuf)
```

В случая, на функцията се подава избрания физически интерфейс, броя байтове, които да се заснемат от преносвателната среда, дали интерфейса да е в *promiscuous* режим, милисекунди за чакане преди да се прочете пакет и указател към буфер, който се запълва в случай на грешка. Накрая чрез `pcap_datalink()` се прави проверка на типа на протокола поддържан на каналния слой, в случая всеки различен от Ethernet (DLT\_EN10MB) води до изключение, тъй като е единствения поддържан от анализатора.

Преди да се приложи даден филтър, той се компилира в `ParseFilters()` метода чрез `pcap_compile()` функцията:

```
1 int pcap_compile(pcap_t *p, struct bpf_program *fp, char *str, int optimize,
2                 bpf_u_int32 netmask)
```

Тя приема указател към вече стартирана сесия, указател към мястото, където да се запази вече компилирания филтър, самия израз за филтрация, дали да се оптимизира, както и мрежовата маска, върху която да се приложи (която се извлича в `PrepareInterfaces()`). Причината да се ползва този метод вместо стандартни условни конструкции е, че този филтър се оптимизира вътрешно, тъй като се предава на BPF пакетния филтър.

В `ApplyFilters()` метода компилирания филтър се прилага чрез `pcap_setfilter()` функцията, приемаща указател към сесия и указател към компилиран филтър:

```
1 int pcap_setfilter(pcap_t *p, struct bpf_program *fp)
```

След компилация на филтъра, следва пускането на сесията. Това се случва в метода `Sniff()`. Той извиква основна функция на `libpcap` — `pcap_loop()`:

```
1 int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user)
```

Първият аргумент е указател към сесията, вторият — колко пакета трябва функцията да заснеме преди да върне резултат (отрицателна стойност означава заснемане докато се случи грешка), третият аргумент е името на *callback* функция, която `libpcap` да извика, когато заснеме пакет, а последния: указател към аргументите, които да се подадат на *callback* функцията.

В случая, `pcap_loop()` се извиква с *callback* функция `OnPacketReceived()`, която е статичен метод за класа:

#### Listing 3.17: Имплементация на `Sniff()` метода на `PcapPacketSniffer`

```
1 void PcapPacketSniffer::Sniff() {
2     pcap_loop(handle_,
3               config_manager_.ExtractValue<int>("sniffer", "packets_count"),
4               OnPacketReceived, reinterpret_cast<u_char*>(this));
5 }
```

Важно е да се отбележи, че като аргумент на функцията се подава `this` указателя, тъй като в `OnPacketReceived()` функцията се извиква метод на класа: `OnPacketReceivedInternal()`. В него се инициализира обект от типа `SniffedPacket`, `body` областта, `composite` обекта, чиито адреси се подават 'нагоре' по стека.

#### Listing 3.18: Имплементация на `OnPacketReceivedInternal()` метода

```
1 void PcapPacketSniffer::OnPacketReceivedInternal(
2     const struct pcap_pkthdr* header, const u_char* packet) {
3     namespace protocols = sniffer::protocols;
4
5     protocols::PacketRegion body{0, static_cast<int>(header->caplen)};
6     protocols::SniffedPacket sniffed_packet{packet, body};
7
8     sniffer::common::serialization::SerializedObject composite{"{}"};
9 }
```

```

10 // http://www.tcpdump.org/linktypes.html
11 stack_.HandleReception("physical", 1, &sniffed_packet, &composite);
12
13 // Discard malformed/partly parsed packets.
14 if (sniffed_packet.valid()) {
15     server_ ->AuthenticatedBroadcast(composite.data());
16 }
17 }

```

### 3.2.6 Имплементация на Header и HeaderMetadata класовете

explain

### 3.2.7 Имплементация на PayloadInterpreter и HexAsciiPayloadInterpreter

#### Имплементация на PayloadInterpreter

PayloadInterpreter представлява интерфейс с един метод: Interpret(), който получава указател към началото на полезната информация в пакета (payload) и дължината ѝ;

Listing 3.19: Имплементация на Interpret() метода на PayloadInterpreter

```

1 class PayloadInterpreter {
2 public:
3     virtual std::string Interpret(const u_char* payload, int length) = 0;
4
5     virtual ~PayloadInterpreter() {};
6 };

```

#### Имплементация на HexAsciiPayloadInterpreter

HexAsciiPayloadInterpreter е клас, интерпретиращ полезната информация на пакета като байтове, представени в шестнадесетична бройна система и ASCII символи. Той има два метода: Interpret() и InterpretLine(). Interpret() метода проверява дали дължината на полезната информация е по-малка от предефинирана константа за брой на байтове на ред (kBytesPerLine), ако е, интерпретира само един ред извиквайки InterpretLine(). Ако дължината е по-голяма, цялата дължина се дистрибутира на редове, дълги kBytesPerLine, като всеки ред се интерпретира поотделно. Всеки ред има изместване (offset), което се определя увеличава с kBytesPerLine. Резултата от интерпретирането на всеки ред се конкатенира в поток от символни низове и е от вида {offset hex ascii}, {offset hex ascii}, .., {offset hex ascii}.

Listing 3.20: Имплементация на Interpret() метода на HexAsciiPayloadInterpreter

```

1 std::string HexAsciiPayloadInterpreter::Interpret(const u_char* payload,
2                                                    int length) {
3     if (length <= 0) {
4         return "none";
5     }
6
7     const u_char* index_ptr = payload;
8     int offset = 0;
9
10    if (length <= kBytesPerLine) {

```



```

11     auto res = InterpretLine(index_ptr, length, offset);
12     return res;
13 }
14
15 int remaining_length = length;
16 int current_line_length = 0;
17 std::ostream out;
18
19 while (true) {
20     current_line_length = kBytesPerLine % remaining_length;
21
22     out << InterpretLine(index_ptr, current_line_length, offset);
23
24     remaining_length -= current_line_length;
25     index_ptr += current_line_length;
26     offset += kBytesPerLine;
27
28     if (remaining_length <= kBytesPerLine) {
29         out << InterpretLine(index_ptr, remaining_length, offset);
30         break;
31     }
32 }
33
34 return out.str();
35 }

```

`InterpretLine()` интерпретира масив от байтове във вида: *offset hex ascii*. Алгоритъма използва тривиална указателна аритметика, като единствено инкрементира указателя `index_ptr`, за да трансформира *i*-тия байт чрез `std::hex` функцията от стандартната библиотека. Трансформация не се налага ако *i*-тият байт е ASCII символ.

## 3.3 Имплементация на сървър

### 3.3.1 Имплементация на Server

`Server` класа има сравнително богат публичен интерфейс: методи за пускане и спиране, за задаване на `host` връзката (т.е клиентът, иницирал сесията на анализ), за изпращане на съобщение към конкретен клиент, към всички клиенти, към всички автентифицирани клиенти, за добавяне на клиент, за премахването му, за автентифицирането на клиент и проверката дали клиент е автентифициран. Важно е да се спомене, че клиентът е представен чрез `ID` от целочислен тип, т.е конкретните имплементации могат да използват друг тип, стига той да бъде съпоставим (например чрез `std::map`) със съответното `ID`. Клиентите представляват списък от тези числа, а автентификацията е имплементирана просто: класът има втори списък с клиенти, който обаче представя само автентифицираните такива. Описаните вече методи добавят или премахват елементи от съответния списък. По-детайлно обяснение на методите е представено в наследник на `Server` класа — `WebSocketServer`.

### 3.3.2 Имплементация на WebSocketServer

`WebSocketServer` класа е *thread-safe* имплементация на сървър, комуникиращ посредством `WebSocket` протокола. Класът е своеобразен `wrapper` върху `websocketpp::server` класа и реализира основната си функционалност чрез `websocketpp` библиотеката. Освен `websocketpp::server`, той съдържа указател към `WebSocketServerEventHandler` (клас, енкапсулиращ функционалността за обработка



на събития от типа на свързване на нов клиент, пращане на съобщение от клиент и т.н), съответствие между `websocketpp::connection_hdl` (структура, имплементираща концепцията за клиент в `websocketpp`) и целочисления тип, идентифициращ клиенти според интерфейса на `Server`, и брояч, определящ следващия ID на клиент. Състоянието на обект от тип `WebSocketServer` изисква манипулация от няколко нишки. Типичен случай, в който *race condition* би се появил е ако нишката, отговорна за анализа и декапсулация, реши да изпрати обработения пакет чрез сървъра, докато нишката за обработка на събитията добавя клиент. С цел избягването на конкурентно модифициране на споделена памет, представените в класа структури биват предпазени с помощта на характерния принцип за взаимно изключване (*mutual exclusion*).

В конструктора на класа се инициализира `Boost.Asio` библиотеката, която `websocketpp` ползва за имплементацията си. На `websocketpp::server` обекта се задават като *callback* методи методите за обработка на събития на `WebSocketServerEventHandler`. `Start()` методът създава отделна нишка за обработка на събитията и извиква `Run()` методът, частен за класа, който от своя страна стартира `websocketpp::server` сървър. Важно е да се отбележи, че по този начин на имплементацията методите за обработка на събития на `WebSocketServerEventHandler`, без `Handle()` методът, който се изпълнява на новосъздадената нишка, биват изпълнени на основната нишка на програмата. След създаването на новата нишка в `Start()`, се изчаква нейното приключване чрез `join()` метода на `std::thread`. Williams (2012) Цялостната логика на останалите методи на класа е да изолират критичната секция чрез мютекс, извиквайки имплементацията на базовия клас (`Server`). Например, методът `AddClient()` добавя клиент по подаден `websocketpp::connection_hdl`.

Listing 3.21: Имплементация на метода `AddClient()` на `WebSocketServer`

```
1 void WebSocketServer::AddClient(websocketpp::connection_hdl handle) {
2     std::unique_lock<std::mutex> map_lock(ws_connections_map_lock_);
3     ws_connections_map_[next_connection_id_] = handle;
4     map_lock.unlock();
5
6     AddConnection(next_connection_id_);
7
8     std::lock_guard<std::mutex> ncid_guard(next_connection_id_lock_);
9     next_connection_id_++;
10 }
```

В случая, списъка на съответствия между `websocketpp::connection_hdl` се заключава, новия клиент се добавя, след което структурата от данни се отключва. Важно е да се отбележи, че в случая член променливата `next_connection_id_` няма нужда от механизъм за взаимно изключване, тъй като `AddClient()` е единствената функция, променяща стойността ѝ. При инкрементирането ѝ в края на метода обаче, механизмът е необходим тъй като има вероятност тя да бъде прочетена в същото време от друга нишка.

### 3.3.3 Имплементация на `WebSocketServerEventHandler`

Вече споменатият `WebSocketServeEventHandler` клас е клас, който се занимава с логиката по обработка на събитията, свързани със сървъра. Едно събитие има следната дефиниция:

Listing 3.22: Дефиниция на структурата `WebSocketServerEvent`

```
1 struct WebSocketServerEvent {
2     WebSocketServerEvent(WebSocketServerEventType event_type,
3                          websocketpp::connection_hdl handle);
4 }
```

```

5   WebSocketServerEvent(WebSocketServerEventType event_type,
6                       websocketpp::connection_hdl handle,
7                       websocketpp::config::asio::message_type::ptr message);
8
9   WebSocketServerEventType type;
10  websocketpp::connection_hdl handle;
11  websocketpp::config::asio::message_type::ptr message;
12 };

```

Вторият конструктор се използва в случай, че събитието е от типа съобщение, в такъв случай събитието освен тип и клиентът, за когото е валидно, съдържа и указател към съобщението. Типа на събитията е енумерация със следната дефиниция:

Listing 3.23: Дефиниция на енумерацията `WebSocketServerEventType`

```

1 enum class WebSocketServerEventType : char { kConnect, kDisconnect, kMessage };

```

Основните методи на класа са `OnConnectionOpened()`, `OnConnectionClosed()`, `OnMessageReceived()` и `Handle()`. Първите три метода имат сходна имплементация, затова е представен само първият:

Listing 3.24: Имплементация на метода `OnConnectionOpened()`

```

1 void WebSocketServerEventHandler::OnConnectionOpened(
2     websocketpp::connection_hdl handle) {
3     {
4         std::lock_guard<std::mutex> guard(event_lock_);
5         events_.push(
6             WebSocketServerEvent(WebSocketServerEventType::kConnect, handle));
7     }
8
9     // Wake up the processing thread
10    event_cond_.notify_one();
11 }

```

Класът `WebSocketEventHandler` има абстрактна структура от данни опашка, съставена от събития. При нова връзка, `websocketpp` извиква въпросният метод като му подава `websocketpp::connection_hdl` за новия клиент. Опашката се заключва чрез осигурен за нея мютекс, новото събитие се добавя към опашката, тя се отключва, а нишката за обработка на събитията се сигнализира ('събужда') посредством популярния механизъм на имплементация с *условна променлива* (condition variable).

`Handle()` методът, фрагмент от който е представен, изчаква върху условната променлива главната нишка да сигнализира за пристигнало събитие. При сигнализация, мютексът се заключва, първото събитие (във FIFO опашката) се изважда, след това мютексът се отключва, а събитието бива обработено според типа му. Ако типът му е съобщение, то се интерпретира като изпратена команда, за изпълнението на която се грижи `ServerCommandInvoker`.

Listing 3.25: Част от имплементацията на метода `Handle()`

```

1     std::unique_lock<std::mutex> lock(event_lock_);
2
3     // The lambda prevents from spurious wake-ups
4     event_cond_.wait(lock, [this] { return !events_.empty(); });
5
6     WebSocketServerEvent current_event = events_.front();
7     events_.pop();
8
9     lock.unlock();

```

### 3.3.4 Имплементация на Command

Следвайки цялостната архитектура, върху сървъра може да се изпълняват команди. Концепцията за команда съответства на `Command` класа, а конкретните команди просто го наследяват, изменяйки поведението му. Освен указател към `Server` обект, върху когото да се изпълни командата, една команда има име и степен на сигурност (т.е. дали може да се изпълнява от неавтентикиран клиент). Най-фундаменталните методи за класа са: `Matches()`, `ParseArguments()` и `Execute()`, като последните два са виртуални и конкретните команди определят поведението им. `Matches()` метода проверява дали даден символен низ е команда, като десериализира низа и чрез описаните вече методи за извличане на стойност в `SerializationManager` провери дали обектът има свойство с името на командата. `ParseArguments()` връща двойки от ключ-стойност с името на аргумента на командата и стойността му, по подразбиране връща празен `std::map`. `Execute()`, логично, изпълнява командата.

#### Имплементация на AuthenticateCommand

`AuthenticateCommand` сравнява подадената като аргумент на командата парола с предефинираната парола на сървъра. Инстанцира модел на отговор, подавайки резултата от предиката, а в случай на съвпадение — автентикира даден клиент към сървъра, т.е. го добавя към вътрешния списък с автентикирани клиенти през публичния интерфейс на `Server`.

Listing 3.26: Имплементация на `Execute()` метода на `AuthenticateCommand`

```
1 void AuthenticateCommand::Execute(  
2     int connection_id, std::map<std::string, std::string> args) {  
3     auto is_authenticated = (server_ -> password() == args["password"]);  
4  
5     sniffer::core::response_models::AuthenticateResponseModel model{is_authenticated};  
6     auto model_obj = model.Serialize(serializer_);  
7  
8     if (is_authenticated) {  
9         server_ -> Authenticate(connection_id);  
10    }  
11  
12    server_ -> Unicast(connection_id, model_obj.data());  
13 }
```

#### Имплементация на HasHostCommand

`HasHostCommand` проверява дали вече е иницирирана сесия на анализ, резултата от проверката подава на съответния модел на отговор, сериализира го и изпраща резултата на клиента, извикал командата.

#### Имплементация на KillCommand

`KillCommand` спира сървъра като извиква `Stop()` метода му.

#### Имплементация на RetrieveInterfacesCommand

`RetrieveInterfacesCommand` е командата, която връща списък от физически интерфейси, поддържащи възможност за анализ. Те са именно интерфейсите, които се извличат посредством `InterfaceRetriever` класа. Имплементацията на `Execute()` метода е тривиална: извличат се всички интерфейси чрез указателя към `InterfaceRetriever`,

който е член променлива за командата, инициализира се модел на отговор, на когото се подават извлечените интерфейси, той се сериализира и съответно бива изпратен на клиентът, изпратил командата.

Listing 3.27: Имплементация на `Execute()` метода на `RetrieveInterfacesCommand`

```
1 void RetrieveInterfacesCommand::Execute(  
2     int connection_id, std::map<std::string, std::string> args) {  
3     auto interfaces = interface_retriever_ -> Retrieve();  
4  
5     sniffer::core::response_models::RetrieveInterfacesResponseModel model{  
6         interfaces};  
7     auto model_obj = model.Serialize(serializer_);  
8  
9     server_ -> Unicast(connection_id, model_obj.data());  
10 }
```

### Имплементация на `StartSnifferCommand`

`StartSnifferCommand` инстанцира обект от тип `PacketSniffer`, подавайки му името на интерфейса и филтъра, който да бъде приложен. В конкретната имплементация е използван `PcapPacketSniffer`, но в идеалния случай аргументът на командата би приемал символен низ, представляващ конкретния тип `PacketSniffer`, който да се инстанцира. Изпратения командата клиент бива определен като инициатор на сесията за анализ. В нова нишка се извиква `Start()` метода на анализатора.

Listing 3.28: Имплементация на `Execute()` метода на `StartSnifferCommand`

```
1 void StartSnifferCommand::Execute(int connection_id,  
2                                   std::map<std::string, std::string> args) {  
3     auto interface = args["interface"];  
4     auto filter = args["filter"];  
5  
6     std::unique_ptr<PacketSniffer> sniffer = std::make_unique<PcapPacketSniffer>(  
7         interface, filter, server_ -> config_manager(), layer_stack_, server_);  
8  
9     server_ -> set_sniffer(std::move(sniffer));  
10    server_ -> set_host_connection(connection_id);  
11  
12    std::thread sniffing_thread{&sniffer::core::PacketSniffer::Start,  
13                                server_ -> sniffer()};  
14    sniffing_thread.detach();  
15 }
```

### 3.3.5 Имплементация на `ServerCommandInvoker`

`ServerCommandInvoker` класът има списък от команди, като основния му метод е `Invoke()`. Той получава указател към `Server` обект, върху когото да изпълни командата, ID на клиента, за когото да я изпълни, и съобщението получено от клиента в чист вид. Имплементацията на метода е сравнително експресивна:

Listing 3.29: Имплементация на `Invoke()` метода на `ServerCommandInvoker`

```
1 void ServerCommandInvoker::Invoke(Server* server, int connection_id,  
2                                   const std::string& message) {  
3     for (auto command : server_commands_) {  
4         if (command -> Matches(message)) {  
5             if (command -> secure()) {
```

```

6         if (server->IsClientAuthenticated(connection_id)) {
7             command->Execute(connection_id, command->ParseArguments(message));
8         }
9     } else {
10        command->Execute(connection_id, command->ParseArguments(message));
11    }
12 }
13 }
14 }

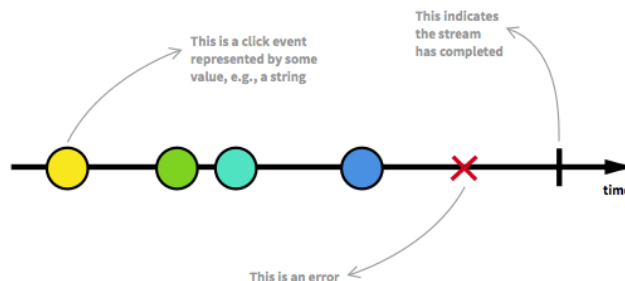
```

Проверява се дали полученото съобщение съвпада с регистрирана команда. Ако командата изисква клиента да е автентикиран, командата се изпълнява тогава и само тогава, когато клиентът е автентикиран. Ако не изисква автентикация, тя се изпълнява директно.

## 3.4 Имплементация на клиент

### 3.4.1 Имплементация на услуги

Голяма част от имплементацията на услугите е базирана основно на парадигмата *реактивно програмиране* и т.нар. реактивни разширения (reactive extensions), реализирани чрез RxJS библиотеката, вградена в Angular 2. На практика, те позволяват програмиране с асинхронни потоци от събития или данни, като предимството е, че в тях е застъпена функционалната парадигма, следователно тези потоци могат да се съединяват (merge), филтрират, трансформират (map) и др. Пример за поток може да бъде стандартно click събитие:



Фигура 3.2: Визуализация на асинхронен поток от данни.

Всяко събитие се прихваща асинхронно чрез дефиниране на callback функция, изпълняваща се при излъчване на стойност, на друга подобна функция, изпълняваща се когато е излъчена грешка, и на друга такава, когато е излъчено събитие за край на потока (completed). Тези функции се наричат изследващи състоянието (observers), а потока, който се изследва — изследваем (observable). В обектно-ориентирания дизайн този шаблон е известен като *наблюдател* (observer). [Gamma \(1995\)](#)

### Имплементация на WebSocketService

Класът `WebSocketService` се занимава с установяването и поддръжката на двустранната комуникация със сървъра. Той има два метода: публичния `connect()`, и частния `create()`. Методът `connect()` приема URL, към който да се свърже, създавайки обект от типа `WebSocket`, дефиниран от `WebSocket API`-а на браузъра.

Класът има поле от тип `Rx.Subject`, което представлява едновременно изследваем (observable) и изследващ обект (observer) от събития, включващи съобщение

(`MessageEvent`). Всяка излъчвана стойност или събитие се изпраща на всички изследващи обекти (subscribers). Това поле се зарежда мързеливо (lazy load) от `create()` метода.

Listing 3.30: Имплементация на `connect()` метода на `WebSocketService`

```
1 public connect(url: string): Rx.Subject<MessageEvent> {
2     this._ws = new WebSocket(url);
3     this.activeConnection = Rx.Observable.fromEvent(this._ws, 'open');
4
5     if (!this._subject) {
6         this._subject = this.create(url);
7     }
8
9     return this._subject;
10 }
```

В `create()` метода се създава изследваем обект (observable), своявайки (bind) callback функциите на изследващия го обект с тези, извиквани по подразбиране за `WebSocket` обекта. Така този изследваем обект на практика приема данните от канала на комуникация и ги 'предава' на изследващите: когато по канала се получи съобщение, се извиква `next` callback функцията на изследващия, когато се получи грешка — `error` callback функцията, когато се затвори канала — `completed` callback функцията.

Listing 3.31: Имплементация на `create()` метода на `WebSocketService`

```
1 private create(url: string): Rx.Subject<MessageEvent> {
2     let wsInstance = this._ws;
3
4     let observable = Rx.Observable.create((observer: Rx.Observer<MessageEvent>) => {
5         wsInstance.onmessage = observer.next.bind(observer);
6         wsInstance.onerror = observer.error.bind(observer);
7         wsInstance.onclose = observer.complete.bind(observer);
8
9         return wsInstance.close.bind(wsInstance);
10    });
11
12    let observer = {
13        next: (data: Object) => {
14            if (wsInstance.readyState === WebSocket.OPEN) {
15                wsInstance.send(JSON.stringify(data));
16            }
17        }
18    };
19
20    return Rx.Subject.create(observer, observable);
21 }
```

## Имплементация на `SnifferService`

`SnifferService` поддържа връзка към сървър, използвайки вече описания `WebSocketService`. Вика се `connect()` метода му с адреса на сървър, определен от средата на изпълнение (development или production), а при отговор потока трансформира получения `MessageEvent` обект към JSON преди да достигне крайния абонат (subscriber).

Listing 3.32: Част от имплементация на конструктора на `SnifferService`

```
1 this._remoteConnection = <Subject<Object>>this.wsService
2     .connect(Config.WS_SERVER_ADDRESS)
3     .map((response: MessageEvent) : Object => {
4         return JSON.parse(response.data);
5     });
```

5       });

Услугата предоставя и методи за 'настройване' на параметрите на анализатора: тя се инжектира като зависимост (dependency) на стъпковите компоненти, където последователно тези параметри се настройват — това са методите `interfaceName()` и `filterExpression()`. Освен това, тя де факто описва интерфейс към командите на сървъра, например извикването на `start()` метода на `SnifferService` води до изпращане на команда за стартиране на анализатора:

Listing 3.33: Част от имплементация на конструктора на `SnifferService`

```
1 public start(): void {
2     let argumentsObject: Object = {
3         'interface': this._interfaceName,
4         'filter': this._filterExpression
5     };
6
7     let commandObject: Object = {
8         'start-sniffer': argumentsObject
9     };
10
11     this._remoteConnection.next(commandObject);
12 }
```

Аналогични са имплементациите на `retrieveInterfaces()`, `sendAuthenticate()` и `sendHostCheck()` методите. Отговорите на тези команди представляват потоци:

Listing 3.34: Част от имплементация на конструктора на `SnifferService`

```
1 public start(): void {
2     let argumentsObject: Object = {
3         'interface': this._interfaceName,
4         'filter': this._filterExpression
5     };
6
7     let commandObject: Object = {
8         'start-sniffer': argumentsObject
9     };
10
11     this._remoteConnection.next(commandObject);
12 }
```

Те са от типа `HostInfo` и `AuthenticationInfo`. Това на практика са модели, аналогични на моделите на отговор, имплементирани на сървъра. Потока за отговори на `retrieveInterfaces()` е енкапсулиран в друга услуга — `InterfaceService`.

## Имплементация на `InterfaceService`

Услугата представлява енкапсулация на потока на отговори на командата за намиране на физически интерфейси, чийто тип е масив от `Interface` обекти:

Listing 3.35: Имплементация на `InterfaceService`

```
1 @Injectable()
2 export class InterfaceService {
3
4     public interfaces: Subject<Array<Interface>>;
5
6     constructor(private wsService: WebSocketService) {
7         this.interfaces = <Subject<Array<Interface>>>this.wsService
8             .connect(Config.WS_SERVER_ADDRESS)
```



```

9      .map((response: MessageEvent) : Array<Interface> => {
10          let data = JSON.parse(response.data);
11          return <Array<Interface>>data.interfaces;
12      });
13  }
14
15  }

```

**Interface** е модел, който дефинира свойствата на един физически интерфейс. Той е аналогичен по свойства на обектите, към който **Interface** обекта на сървъра се сериализира. Има следната дефиниция:

Listing 3.36: Дефиниция на **Interface** модела

```

1  export interface Interface {
2      name: string;
3      description: string;
4      addresses: Array<InterfaceAddress>;
5  };

```

Подобно на **Interface**, **InterfaceAddress** модела на клиента има свойствата на обекта, резултат от сериализацията на **InterfaceAddress** обектите за даден физически интерфейс на сървъра:

Listing 3.37: Дефиниция на **InterfaceAddress** модела

```

1  export interface InterfaceAddress {
2      addr: string;
3      bcastAddr: string;
4      dstAddr: string;
5      netMask: string;
6  };

```

## Имплементация на **PacketService**

Класът **PacketService** предоставя основния поток от пакети, които се изпращат по комуникационния канал между сървъра и клиента. Стандартно, той бива трансформиран като се извлекат данните от **MessageEvent** обекта и се десериализират в конкретен обект — **Packet**. Това става в конструктора на класата:

Listing 3.38: Имплементация на конструктора на **PacketService** модела

```

1  constructor(private wsService: WebSocketService) {
2      this.packets = <Subject<Packet>>this.wsService
3          .connect(Config.WS_SERVER_ADDRESS)
4          .map((response: MessageEvent) : Packet => {
5              let data = JSON.parse(response.data);
6              return <Packet>data;
7          });
8  }

```

Услугата има и отговорността да пази изследвания пакет за цялото приложение, чрез полето **\_observedPacket**, което е от тип **BehaviorSubject<Packet>**. Разликата между използвания вече **Rx.Subject** клас и **Rx.BehaviorSubject** е, че последния излъчва стойности (в случая **Packet** обекти) дори когато в потока не съществуват такива (т.е по подразбиране, ако пакет не е избран, потока връща като стойности празни обекти).



### 3.4.2 Имплементация на компоненти

#### Имплементация на HomeComponent

**HomeComponent** проверява дали на сървъра съществува сесия на анализ посредством **sendHostCheck()** метода на **SnifferService**. В конструктора компонента се абонира за потока от отговори на командата и при отговор пренасочва към правилния компонент. Ако сесия не съществува, клиента се пренасочва към **NewSessionComponent**, ако такава съществува — **SessionSubscriptionComponent**.

#### Имплементация на NewSessionComponent

**NewSessionComponent** няма съществена програмна логика, но съдържа шаблона, в който се изобразяват стъпките за настройване на параметрите на анализатора, които се изобразяват в `<router-outlet></router-outlet>`.

#### Имплементация на PasswordStepComponent

В **PasswordStepComponent** се съдържа логиката за четене на паролата на администратора, което се в шаблона се имплементира със стандартно **input** поле, и автентикацията. При натискането на **Next** бутона се извиква метода на компонента **handleStep()**, който чете въведената парола и я изпраща чрез **sendAuthenticate** метода на **SnifferService**. Тъй като в конструктора на компонента той се абонира към поток от отговори на съобщения за автентикация, при положителен отговор клиентът се пренасочва към компонента за избиране на физически интерфейс.

#### Имплементация на InterfaceStepComponent

**InterfaceStepComponent** е компонент, който предоставя на мрежовия администратор възможността да избере чрез **typeahead** падащо меню конкретния физически интерфейс, на който да слуша анализатора. В конструктора си компонента изпраща команда за намиране на интерфейсите чрез **retrieveInterfaces()** метода на **SnifferService**, след това се абонира към потока от отговори на командата от **InterfaceService** и когато получи такива, попълва списъка, от когото се генерира падащото меню. След като се натисне **Next** бутона се извиква **handleStep()** метода, който настройва интерфейса като параметър на **SnifferService** и пренасочва администратора към следващата стъпка — **FilterStepComponent**.

#### Имплементация на FilterStepComponent

**FilterStepComponent** представлява компонент, аналогичен по имплементация на **PasswordStepComponent** — **input** поле, чиято стойност се прочита, но вместо да се изпрати се записва като параметър на **SnifferService**.

#### Имплементация на StartStepComponent

**StartStepComponent** има тривиална имплементация — в шаблона изобразява систематизирано в таблица обобщение на зададените параметри на **SnifferService**. При натискането на **Start** бутона се извиква **handleStep()** метода на услугата, който чрез **start()** метода на **SnifferService** инстанцията изпраща команда за пускане на анализатора с вече зададените параметри.

## **Имплементация на SessionSubscriptionComponent**

`SessionSubscriptionComponent` компонента се инициализира в случай, че вече съществува сесия на анализ. Неговата единствена задача е да прочете паролата от *n*-тия свързан мрежов администратор, да я изпрати на сървъра чрез `sendAuthenticate()` метода на `SnifferService` и при успешна автентикация, да пренасочи администратора към `CaptureComponent`.

## **Имплементация на PacketListComponent**

## **Имплементация на PacketDetailsComponent**

## **Имплементация на PacketBytesComponent**

# Глава 4

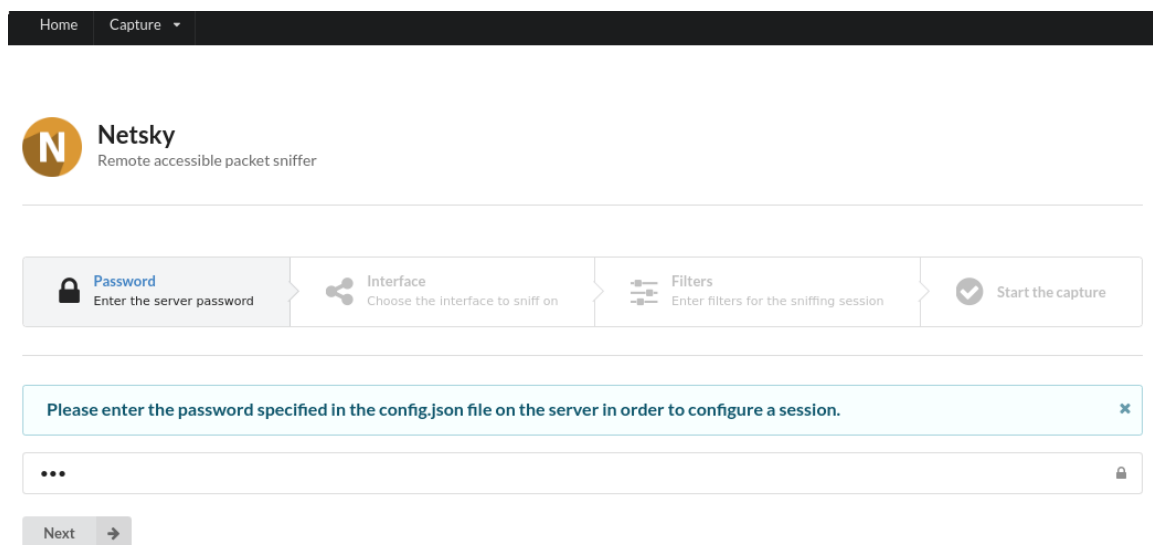
## Ръководство на потребителя

### 4.1 Deployment

### 4.2 Работа с приложението

#### 4.2.1 Въвеждане на парола

Преди да има възможност за избор на интерфейс, мрежовият администратор трябва да бъде автентикиран към сървъра. За тази цел той трябва да въведе парола. След въвеждането на парола, **Next** бутона отвежда към следващата стъпка — избора на физически интерфейс.

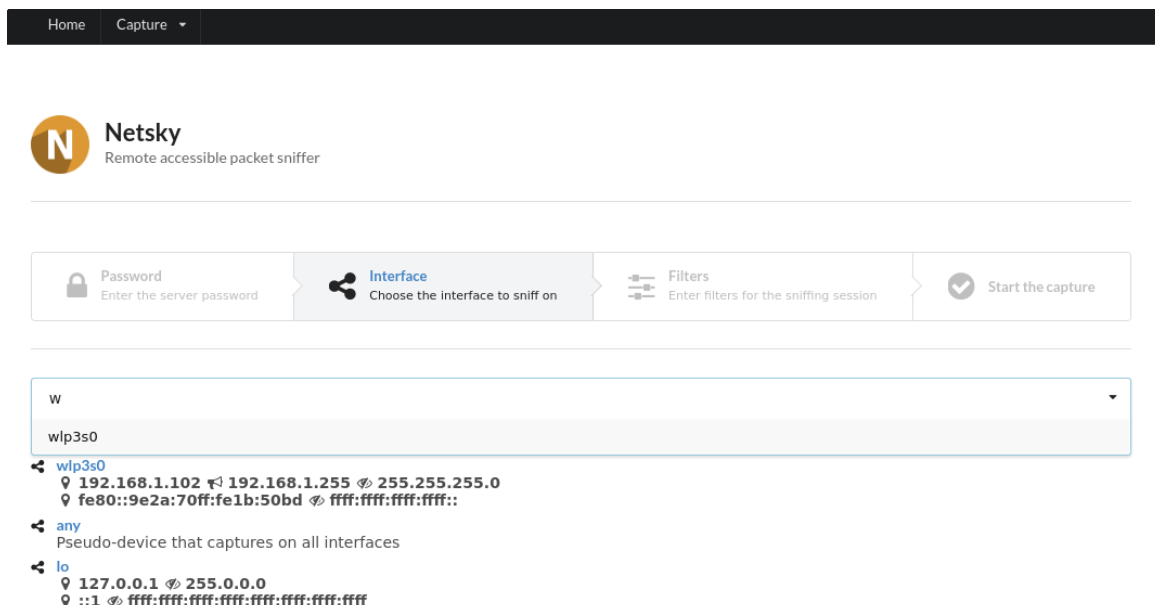


The screenshot shows the Netsky web interface. At the top, there is a navigation bar with 'Home' and 'Capture' (with a dropdown arrow). Below this is the Netsky logo and the text 'Remote accessible packet sniffer'. The main content area has a horizontal flow of four steps: 'Password' (selected, with a lock icon and the text 'Enter the server password'), 'Interface' (with a network icon and 'Choose the interface to sniff on'), 'Filters' (with a list icon and 'Enter filters for the sniffing session'), and 'Start the capture' (with a checkmark icon). Below these steps is a light blue message box that says 'Please enter the password specified in the config.json file on the server in order to configure a session.' with a close button (X). Underneath the message box is a password input field with a lock icon on the right. At the bottom left, there is a 'Next' button with a right arrow.

Фигура 4.1: Въвеждане на парола за сесията.

#### 4.2.2 Избор на физически интерфейс

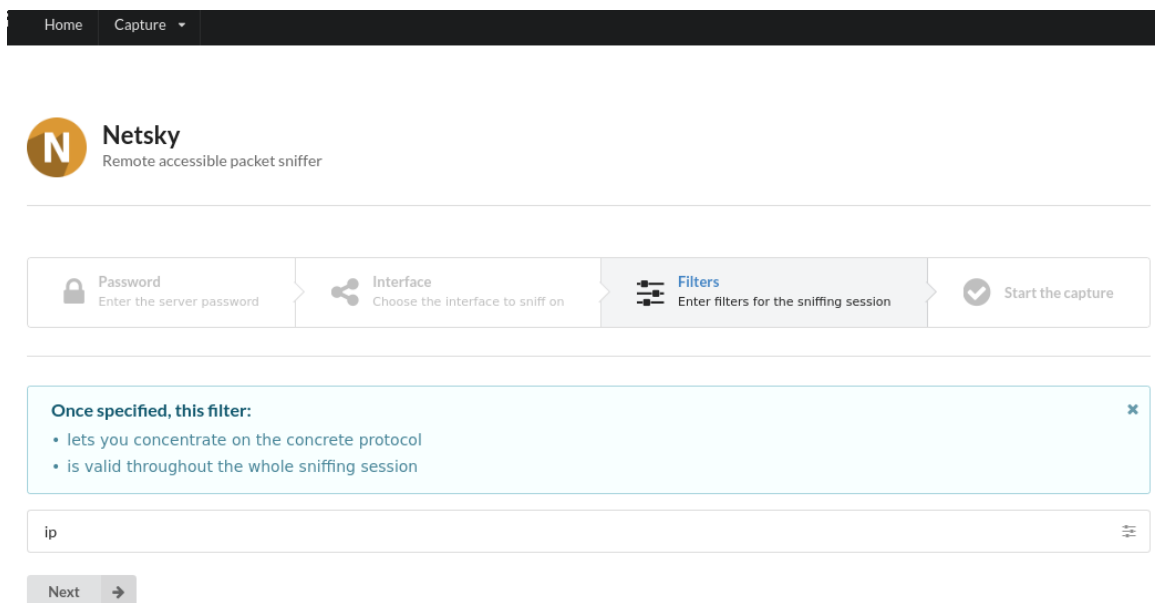
На мрежовия администратор се предоставя списък от имена на интерфейси, както и адресите на тези, които имат такива: IP адрес, мрежова маска и т.н. За избор на интерфейс е достатъчно да се започне въвеждането на името на интерфейса, след това от падащото меню се избира чрез мишката или Enter. След избора на интерфейс, **Next** бутона отвежда към следващата стъпка — въвеждането на филтриращ израз.



Фигура 4.2: Избор на физически интерфейс.

### 4.2.3 Въвеждане на филтриращ израз

Мрежовия администратор може да въведе филтър, който да е валиден за цялата сесия на анализ. Например, филтърът `ip` позволява само IP трафик. Съществуват по-прости<sup>1</sup> филтри, но и комплексни<sup>2</sup> такива, с които може да се филтрира на ниво битове.



Фигура 4.3: Въвеждане на филтриращ израз.

### 4.2.4 Стартиране на сесия

След настройването на необходимите параметри на сесията, на мрежовия администратор се представя екран на кратък обзор на конфигурираните параметри. След натискането на **Start** бутона, сесията започва.

<sup>1</sup><http://alumni.cs.ucr.edu/~marios/ethereal-tcpdump.pdf>

<sup>2</sup>[https://www.wains.be/pub/networking/tcpdump\\_advanced\\_filters.txt](https://www.wains.be/pub/networking/tcpdump_advanced_filters.txt)

Home

Capture ▾

N

Netsky

Remote accessible packet sniffer

🔒

Password

Enter the server password

🔗

Interface

Choose the interface to sniff on

🔍

Filters

Enter filters for the sniffing session

✓

Start the capture

| Configuration entry   | Selected value |
|---|----------------|
| <div><div>🔗</div><div><div>Interfaces</div><div>Physical interfaces to sniff on</div></div></div> | wlp3s0         |
| <div><div>🔍</div><div><div>Filters</div><div>Limit the output</div></div></div>                   | ip             |

Start

Фигура 4.4: Стартиране на сесия.

## Глава 5

### Заклучение

Представената дипломна работа реализира прост мрежов анализатор с възможност за отдалечен преглед на анализа от няколко мрежови администратори. Въпреки че не се цели в имплементация на голямо множество протоколи, текущата имплементация предлага принципен модел на реализация, който се стреми да направи добавянето на такива лесно чрез прилагане на вече доказани добри практики в обектно-ориентирания дизайн. Текущата имплементация включва в себе си използването на библиотеки на ниско ниво (libpcap), използването на вече установени езици поддържащи механизми за абстракция (C++11), както и модерни технологии в лицето на Angular 2, TypeScript и WebSocket протокола.

# Библиография

- A. Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. C++ in-depth series. Addison-Wesley, Boston, MA, 2001. ISBN 978-0-201-70431-0.
- E. Gamma, editor. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley professional computing series. Addison-Wesley, Reading, Mass, 1995. ISBN 978-0-201-63361-0.
- S. Meyers. *Effective modern C++: 42 specific ways to improve your use of C++11 and C++14*. O'Reilly Media, Beijing ; Sebastopol, CA, first edition edition, 2014. ISBN 978-1-4919-0399-5. OCLC: ocn884480640.
- C. Sanders. *Practical packet analysis: using Wireshark to solve real-world network problems*. No Starch Press, San Francisco, Calif, 2. ed edition, 2011. ISBN 978-1-59327-266-1. OCLC: 755869776.
- B. Stroustrup. *The C++ programming language*. Addison-Wesley, Upper Saddle River, NJ, fourth edition edition, 2013. ISBN 978-0-321-56384-2.
- A. S. Tanenbaum and D. Wetherall. *Computer networks*. Pearson Prentice Hall, Boston, 5th ed edition, 2011. ISBN 978-0-13-212695-3. OCLC: ocn660087726.
- A. Williams. *C++ concurrency in action: practical multithreading*. Manning, Shelter Island, NY, 2012. ISBN 978-1-933988-77-1. OCLC: ocn320189325.

# Списък на фигурите

|      |   |    |
|------|---|----|
| 1.1  | Модел на пет слойна мрежа . . . . .   | 10 |
| 1.2  | Междуслойна комуникация. Капсулиране и декапсулиране. . . . .                                       | 11 |
| 1.3  | Сравнение между OSI и TCP/IP . . . . .  | 12 |
| 1.4  | Преглед на имплементацията на мрежов анализатор на ниво ядро на<br>операционната система . . . . .  | 13 |
| 1.5  | Изглед на потребителския интерфейс на Wireshark. . . . .  | 16 |
| 1.6  | Структура на tcpdump команда. . . . .   | 16 |
| 1.7  | Изглед на потребителския интерфейс на tcpdump. . . . .  | 17 |
| 2.1  | Абстрактен поглед върху архитектурата. . . . .  | 20 |
| 2.2  | UML диаграма на <b>ConfigurationManager</b> класа. . . . .  | 21 |
| 2.3  | UML диаграма на класа <b>LayerStack</b> . . . . .   | 21 |
| 2.4  | UML диаграма на класа <b>Layer</b> и наследниците му. . . . .                                       | 22 |
| 2.5  | UML диаграма на класа <b>SniffedPacket</b> . . . . .  | 22 |
| 2.6  | Диаграма на разпределение на областите при получаване на пакет. . . . .                             | 23 |
| 2.7  | UML диаграма на класа <b>PacketSniffer</b> . . . . .  | 23 |
| 2.8  | Основен алгоритъм на работа. . . . .  | 24 |
| 2.9  | UML диаграма на класовете <b>Server</b> и <b>WebSocketServer</b> . . . . .                          | 25 |
| 2.10 | UML диаграма на класа <b>ServerCommand</b> и наследниците му. . . . .                               | 26 |
| 2.11 | Компонентно дърво на потребителския интерфейс и зависимости на<br>компонентите от услугите. . . . . | 27 |
| 3.1  | Процес на добавяне на нов <b>Header</b> обект към вече изграденото дърво. . . . .                   | 32 |
| 3.2  | Визуализация на асинхронен поток от данни. . . . .  | 45 |
| 4.1  | Въвеждане на парола за сесията. . . . .   | 51 |
| 4.2  | Избор на физически интерфейс. . . . .   | 52 |
| 4.3  | Въвеждане на филтриращ израз. . . . .   | 52 |
| 4.4  | Стартиране на сесия. . . . .  | 53 |



## Списък на таблиците