Lightweight Middlebox TCP

Auteur : Gaillard, Romain
Promoteur(s) : Mathy, Laurent
Faculté : Faculté des Sciences appliquées
Diplôme : Master en sciences informatiques, à finalité approfondie
Année académique : 2015-2016
URI/URL : http://hdl.handle.net/2268.2/1626

Lightweight Middlebox TCP

Romain Gaillard

Supervisor: Prof. Laurent Mathy
Committee members: T. Barbette, Prof. B. Boigelot, Prof. G. Leduc

Master's thesis submitted for the degree of
Msc in Computer Science

University of Liège
Faculty of Applied Sciences
Academic year 2015 - 2016

Abstract

Lightweight Middlebox TCP

Master's thesis submitted for the degree of Msc in Computer Science

Author: Romain Gaillard
Supervisor: Prof. L. Mathy
Academic year 2015 - 2016

Nowadays, middleboxes are important actors of the Internet and they are used in many contexts such as network address translation, firewalls, load balancers, and intrusion detection systems, among others. Consequently, their implementation can have a great impact on the performance of networks and it is thus crucial to ensure that they do not become bottlenecks.
The objective of this work is to develop a lightweight and middlebox-oriented TCP stack that takes into account the specificities of the context in which middleboxes work in order to provide them with the ability to inspect and modify the traffic, as well as inject packets. All of this on the fly. This manuscript first describes the architecture of the developed framework, listing its components and functionalities, and how to use them to create middleboxes.
We then continue by providing information about the development and the design of the framework, describing the underlying data structures. In addition, we detail the algorithms at the heart of the TCP stack as well as the corresponding time complexities and we explain why they are important to achieve our goals.
The results indicate that it is possible to use this framework to implement a middlebox that performs deep packet inspection with a small and constant overhead. On the other hand, when the middlebox starts modifying the flows, the overhead becomes linear regarding the size of the content. We show that the modularity of the framework we developed allows the users to mitigate the induced overhead by selecting only the features they need.
Finally, we list some elements that could not be implemented in this work, proposing some improvements to the present work that could be made in the future in order to extend it.

Acknowledgements

I would like to start by expressing my sincere gratitude to my supervisors, Prof. Laurent Mathy and Tom Barbette. They spent much time sharing their experience and counsels with me and they were always available to assist me. I had the chance to work on a enthralling subject in which they are very involved and this work would have definitely not been possible without their help.

I also thank my friends and family for their support and their interest, in particular Soraya for proofreading this document.

Contents

Chapter 1

Introduction

1.1 Middleboxes

1.1.1 Definition

A middlebox, as the name suggests, is a networking device located in the middle of a connection that performs several functions on the traffic that passes through it. Such a device can have various behaviours, for instance Network Address Translation (NAT), traffic filtering, deep packet inspection (DPI) or security strengthening. A middlebox often implements multiple of these features at the same time, being therefore a complex network actor that can have a considerable influence on the traffic.

1.1.2 Usages and classification

As the definition of middleboxes is extremely wide, it includes a large number of network protagonists with highly different purposes and behaviours. Therefore, classifying middleboxes is not a straightforward process and it can be done in many different ways. Here, we provide a non-exhaustive list of the goals that can be achieved by using middleboxes, trying to classify them according to their behaviour regarding the traffic that flows through them. The criterion we use to do so is the network layers impacted by the middlebox. Middlebox are considered to act on the network layer and above [4], and to have a more intrusive behaviour when a higher layer is involved in the functioning of the middlebox.

Virtual Private Network

A Virtual Private Network (VPN) is, in short terms, a private network built upon a public network. The point of using a public infrastructure such as the Internet, instead of wiring a dedicated private network is generally to reduce costs and allow flexibility. A VPN can be implemented with two middleboxes creating an IP tunnel, the first one is located at the first edge of the private network. It encapsulates the traffic and sends it to a second middlebox, located at the other end of the private network. This second middlebox decapsulates the data it receives, and sends them to the destination the client wanted to reach. Thanks to the encapsulation made by the middleboxes, the VPN specificities are invisible for the endpoints. Everything is seen as if the two

subnetworks were directly connected. In this case, the middleboxes alter *the third layer*, in order to provide the encapsulation mechanism.

Network Address Translator

A Network Address Translator (NAT) is a network device located at the edge of a local network that makes the mapping between the private IP addresses used behind the NAT with unique public addresses used outside of the NAT. One of the reasons why they have become more and more used is to provide a solution to the massive growth of the Internet and the exhaustion of the IPv4 address space. Indeed, it quickly became obvious that providing a unique IP address to each host connected to the Internet would require a bigger address space. NATs helped mitigate this problem by allowing to assign one public IP address to an entire local network. Behind the NAT, the IP addresses are private and do not need to be unique regarding other local networks, they are not advertised outside of the local network. The middlebox implementing the NAT function modifies *the third layer*, namely the network layer, to translate the IP addresses, and can also impact the fourth layer (*transport layer*) to manipulate the port numbers[6][21][26].

Proxy

A proxy server is an intermediary between a client and the server it wants to reach. In this configuration, instead of having two endpoints directly communicating as in the normal case, we have a middle protagonist in the communication process, the proxy. The latter is in charge of forwarding the client's requests to the destination. For the endpoint server, everything is seen as if the proxy server was the initiator of the requests; it never communicates directly with the client, all the traffic passes through the proxy. In fact, the proxy acts as a client for the server and as a server for the client[5].

There exist multiple kinds of proxies, often depending on the protocol they are designed to manage. Those proxies can modify *all the network layers*, including the application one, but there also exist more generic proxies such as SOCKS proxies that leave *the application layer* unmodified and simply transfer its content.

Intrusion Detection System

An Intrusion Detection System (IDS) analyses the traffic that passes through it in order to detect and prevent illicit activities. Its role is passive and it does not interfere with the network activity. It rather detects specific patterns and signatures in the packets in order to trigger alerts when a suspicious element is found so that countermeasures can be set up. To perform the analysis, the IDS can monitor *all the network layers*, but in a read-only fashion. On the other hand, Intrusions Prevention Systems (IPS) do not only log and notify the suspicious content, they can also try to block them by modifying or discarding the malicious packets.

Firewall

A firewall is a network device or program that monitors the traffic in order to determine if it is permitted or not, according to a set of rules. A firewall therefore has a role of

classification regarding the traffic, determining if it belongs to the *allowed* or *blocked* category. The set of rules can contain criteria based on the IP addresses of the source and the destination, the protocols and the port numbers, for instance.

A firewall can be **stateless**, or **stateful**, in which case it has memory and can take a decision based on previously seen packets and events. As an example, a stateful firewall can determine if the *three-way handshake* has been performed before allowing other TCP packets. In both cases, firewalls generally act on the first four layers, but more advanced ones can also analyse the *application layer* in order to make better decisions. For instance, to determine if the protocol used is indeed the expected one and if an attacker does not try to use a forbidden protocol by hiding it. Such a firewall hence does not only rely on the headers to make a decision and allows more fine-grained criteria.

1.1.3 End-to-end principle

The **end-to-end** principle is described by J. H. Saltzer et al. (1984)[20] as the argument supporting that the intelligence of the network should be implemented at its endpoints. They state that implementing the various functions on the internal nodes can be redundant and not efficient. Indeed, in many cases, the endpoints will still have to implement the functions, which leads to redundancy. This results in higher costs and lower performance.

Moreover, the internal nodes do not always have all the information needed to apply the required functions in an efficient way and should thus leave the responsibility to the endpoints, which are the main actors of the communication process. This principle has been applied to the architecture of the Internet, assuming that "an end-to-end protocol design should not rely on the maintenance of state (i.e. information about the state of the end-to-end communication) inside the network" (Ben Carpenter, 1996)[3].

By definition, middleboxes contravene the end-to-end principle since they are providing functions in the middle of the network instead of leaving the responsibility of implementing them to the endpoints. The implications and the interpretations of the end-to-end principle have evolved over the years, as described by Kempf et al.[10] and it is the subject of more and more pressures.

1.1.4 Deep Packet Inspection

Deep packet inspection (DPI) refers to an internal node of the network (as opposed to endpoints) analysing the content of the packets going through it, including the payload and not only the various headers necessary for the good forwarding of the packet. It may be done for statistical purpose, eavesdropping, intrusion detection, filtering or even for censorship reasons. Middleboxes that manipulate the application layer are said to perform deep packet inspection. Obviously, performing DPI violates the end-to-end principle as the intelligence of the network is not only located to the end nodes any more.

Nowadays, DPI is a hot topic, mainly because it can have a huge impact on the privacy and the Internet neutrality. As an example, some governments use it to prevent

people from accessing a list of websites. To do so, the TCP payload, corresponding to the application layer content, is analysed, searching for sensitive keywords and in the case of a match, the packet may be discarded or the connection closed without notifying any of the protagonists[25]. However, DPI is a very broad technique that has many applications and can be used in a large number of contexts, not only for controversial reasons.

1.2 Context of this work

As the Internet grows, the amount of traffic that flows everyday becomes more and more important, leading to the need to carefully design the actors involved in the network. Avoiding congestion is thus of the highest importance, in particular at the dawn of the Internet of things, which results in the exponential growth of the number of actors exchanging data.

Nowadays, middleboxes play a central role regarding the Internet. They are in fact necessary in many aspects. For instance, it is usual for big companies that receive a lot of traffic to use load balancers to share the charge between multiple servers. It is also common to use a cache system to reduce the number of heavy requests. In addition, some Internet providers use several methods to decrease the bandwidth consumption, including cache systems but also compression. Indeed, it is not uncommon, in particular for mobile connections, to compress images included in web pages, reducing their size, but generally also their quality at the same time. The argument given by the providers is that, as the screen size is generally lower on mobile devices, reducing the quality should not have an influence on the perceived quality of the image. These are a few examples of the reasons why middleboxes not only have many theoretical applications, as described in section 1.1.2 (*Usages and classification*), but are actually used to cope with the need of higher and higher throughputs.

Those two observations highlight the fact that middleboxes are now important actors of the Internet; their implementation has therefore a great impact on the performance of networks. It is thus crucial to ensure that middleboxes do not become network bottlenecks. Furthermore, as they are generally rather tools than actors that produce useful content in a connection, it is reasonable to consider that they should not significantly decrease the performance of the connection. It may even be conceivable to use middleboxes to improve overall performance in some cases.

1.3 Goals

The goal of this work is to develop a fast and easy-to-use TCP framework targeted to developers so that they can create lightweight TCP middleboxes. This framework is meant to handle the low-level problems in such a way that developers can focus on the functionalities they need to implement instead of worrying about the performance of the network management system, as well as the specificities of the lower layer protocols. Moreover, the implementation of the TCP protocol must be lightweight and specifically designed for middleboxes, allowing to inspect, modify the traffic and inject packets on

the fly, all of this seamlessly for the endpoints and by interfering as low as possible with the TCP connection.

Using recent tools such as Netmap[18], this work must take advantage of dedicated techniques, including zero-copy, kernel bypassing and packet batching to get the maximum potential of the network. The expected result is to have an impact on the performance as low as possible. Regarding its implementation, the focus has to be made on providing a modular, flexible and lightweight tool that is able to handle some of the challenges raised by the growing size and complexity of the Internet. The algorithms and data structures used to achieve this goal are thus of the highest importance and must be selected according to memory consumption as well as time efficiency criteria in order to avoid bottlenecks. Thus, this work must also focus on giving arguments and comparisons that lead to the final result, which will consist in a series of elements and libraries compatible with an extended version of the Click Modular Router[14], a piece of software that provides a convenient and flexible way to implement and configure router functions. Using those elements, a developer will be able to create a configuration of modules that manage the TCP stack and some well-known protocols such as HTTP. Moreover, the various tools and libraries provided must allow to create custom elements that implement behaviours corresponding to specific needs.

In addition to describing the implementation of the provided framework, this work must also focus on giving clues and points of attention to extend it. Indeed, as we have seen, middleboxes have a huge number of applications, in very distinct fields and, moreover, the TCP protocol is vast and contains a broad number of mechanisms, options and extensions that make it hard to integrate in an exhaustive way. Thus, we must explore the possible improvements and weaknesses that could not be assessed during the development of the presented framework, hoping to provide a good starting point to the development of a more exhaustive tool and to describe the errors to avoid when developing it.

Chapter 2

Architecture

In this chapter, we present the developed tool from a high level point of view, describing its various components and their peculiarities.

2.1 Frameworks used

The developed program takes advantage of several pieces of software in order to achieve its purpose. This section provides a short description of these tools, their contributions, and how they are used in the context of this work.

2.1.1 Netmap

Netmap is a framework aimed at providing the best performance for fast packet I/O. It was developed by *Luigi Rizzo* as a result of the observation that general purpose OSes offer a network API that was designed to be generic, easy to use and adapted for all situations, but with tradeoff considerations that are now 30 years old. At that time, the link speed was also much slower than it is now and parallel processing was not as important as it currently is[19]. Nowadays, the situation that led to the design of such an API has changed and the tradeoffs that were relevant before are not necessarily appropriate any more.

For this reason, among others, it is now clear that the network API provided by the OSes on our computers is not the most efficient to develop high throughput applications. In the context of this work, where the goal is to have an impact on the performance as low as possible, netmap helps by providing a framework that takes benefit of several state-of-the-art techniques to provide high rate packet I/O.

Using Netmap instead of the classical kernel API allows to decrease the packet-processing cost and to achieve better performance by taking advantage of many improvements such as zero-copy, I/O batching and kernel bypass[18][2]. In the context of this work, this framework is used, via its integration in FastClick (c.f. section 2.1.3) as a shortcut to send and receive raw packets with as small a cost as possible.

2.1.2 Click Modular Router

Click is a modular software architecture that allows to create routers in a very convenient and flexible way. A click router is based on a configuration file containing elements that are linked together as a directed graph. The traffic flows from one or more entry points (generally *FromDevice* elements) to one or more exit points (generally *ToDevice* elements) and passes through various elements, following various paths according to how elements process packets[14].

In the case of this framework aimed at developing middleboxes, Click is ideal as it offers its flexibility to the users. Indeed, they can easily use the elements provided along with this work to create chains of middleboxes that implement various functions, only including the needed components so that it stays as lightweight as possible. Moreover, it is easy to extend or create new elements on top of the others, so that developers can create their very own ones to meet specific needs. Finally, the configuration files used to create click routers are easily editable and a user can modify the parameters of the elements in a short time, without needing to recompile them, which appears to be very practical for debugging purpose in addition to provide a high adaptability to their product.

Imagine that a developer has created an element that classify the traffic according to some criteria (for instance, whether or not the payload contains specific keywords). Let us assume that this element has two outputs: the first one corresponding to the traffic that meets the criteria and the second output corresponds to the traffic that does not meet them. If the developer is currently dropping all the traffic arriving on the second output but wants to change it to send this traffic on a dedicated network card, he just has to edit the configuration file in order to replace the *Discard* component, which is a click built-in element that drops all the packets, by a *ToDevice* element. With this simple example, we can see how powerful and modular Click is, and why it is the ideal candidate for the implementation of our framework.

A very simple example of a configuration is depicted on figure 2.1. In this configuration, every packet coming on the interface *eth0* is sent to the element **Strip**, which is a component that strips the number of bytes given as an argument from the beginning of the packet. It is often used to get rid of the Ethernet header by configuring it to remove *14 bytes*. Next, the packet is given to the **CheckIPHeader** element which performs some tests on the packet in order to determine if its IP header is correct. This element has one input and can have either one or two outputs. If only the first output is used, packets with a valid IP header are sent on it and the others are discarded. If two outputs are used, instead of being discarded, the packets with an invalid IP header are sent on the second output. In this example, only the first input is used and it is connected to the **Print** element that will display the message *OK* when it receives a packet, and transmit it on its output. Finally, the packets arrive in the **Discard** element where they are discarded. This configuration is translated straightforwardly into a click configuration file as listing 2.1 shows.

Figure 2.1: Simple example of Click configuration (adapted from the documentation[12])

```
1 FromDevice(eth0) -> Strip(14) -> CheckIPHeader() -> Print(OK)
2     -> Discard();
```

Listing 2.1: Simple example of Click configuration file

A second example of configuration is showed on figure 2.2. It the the same as the first example, except that the second output of **CheckIPHeader** is used and connected to a **Print** element that displays the message *NOK* when it receives a packet. The packets are, as in the first example, discarded after the *Print* elements. Once again, this configuration is easily translated into a click configuration file, showed on listing 2.2. Note that *cip* is just the name we chose to give to the instance of **CheckIPHeader** so that we can configure its second output.

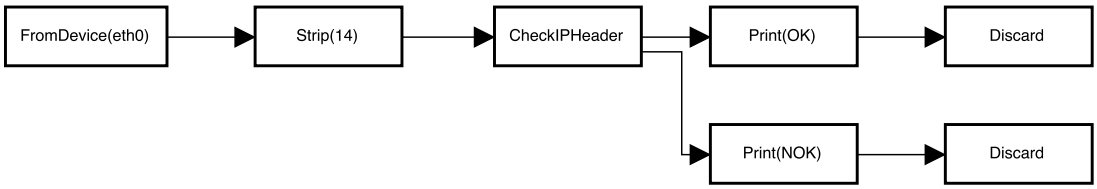

Figure 2.2: Second simple example of Click configuration (adapted from the documentation[12])

```
1 FromDevice(eth0) -> Strip(14) -> cip::CheckIPHeader()[0] -> Print(OK)
2     -> Discard();
3
4 cip[1] -> Print(NOK) -> Discard();
```

Listing 2.2: Second simple example of Click configuration file

These two examples show how modular and easy to maintain a click configuration is. It is indeed simple to modify or create new paths that the packets will follow. This advantage is used in the context of this work as the final result is provided in the form of click elements that can be used to create middleboxes.

2.1.3 FastClick

FastClick is an extended version of Click, developed at the University of Liege, which integrates many improvements to increase the performance[2]. Here is a non-exhaustive list of the improvements provided by FastClick that directly benefit to this work:

- **Enhanced integration of Netmap:** in the vanilla version of Click, Netmap is polled in order to obtain an I/O batch of packets that will be sent, one by one, as

a burst, to the click elements. After the burst, Click will reschedule this task and repeat the process if there are packets available for I/O. FastClick improves this behaviour by avoiding rescheduling the task if a full I/O batch is not available, which allows to process bigger batches and therefore take better advantage of the I/O batching mechanism[2].

- **Zero copy:** Taking advantage of the ability to swap Netmap's buffers from the receive and the transmit rings, FastClick gets rid of the necessity to copy the content from the receive buffer to the structure Click uses to process the packets, and at the end of the path, from this structure to the transmit buffer.

- **Compute batching:** In Click, along the path of elements, packets are passed one by one from the output of an element to the input of the next one. FastClick modifies this behaviour by introducing the mechanism of packet batches in Click. They are represented by linked lists of packets and can have various sizes, depending on the number of packets available when the batch is built. This allows each element to process an entire batch of packets before passing it to the next element, instead of processing packets one by one. The strong advantage of the packet batching implementation provided by FastClick is that it is fully backward compatible with the old Click's elements. Thus, it is possible to benefit from the advantages of packet batching by making elements compatible with this mechanism incrementally.

More information about FastClick and the various improvements it provides to Click can be found in the paper *Fast Userspace Packet Processing*, written by Tom Barbette, Cyril Soldani and Laurent Mathy[2].

2.1.4 MiddleClick

MiddleClick is a fork of FastClick, also developed at the University of Liege, aimed at providing Click with flow management mechanisms. It is based on the observation that even though some middleboxes such as NATs are packet oriented, most of the functions achieved by middleboxes require to process flows and not only single packets. As an example, a middlebox managing HTTP contents may have to process data split over multiple packets, or process headers that require to have a view of the entire flow of data in order to make a decision (such as *Content-Length* which requires to know the size of the HTTP payload). Moreover, for frameworks that use raw packets and do not rely on the operating system to handle flows, they need to manage the flow classification by themselves. This may be problematic when middleboxes are pipelined because it means that the flow classification is done multiple times. Therefore, Middleclick aims at solving these problems and enhancing the support of middleboxes in Click.

Most importantly for this work, MiddleClick provides the router elements with the ability to define per-flow memory chunks, called *scratchpads*. In concrete terms, this means that when an element receives a packet (or a packet batch), it also receives a pointer to the scratchpad belonging to the flow associated to the packet (or the packet batch, since in MiddleClick, all the packets of a batch belong to the same flow). Therefore, the elements do not have to take care about flow management as the memory they

use is automatically associated to the flow; because this memory is not shared among the flows, the elements have the impression to work on a single flow of data.

As an example, imagine we want to create an element that we will call *TCPTime*, which, when the connection is closed, displays the time during which it was open. To do so, we add in the scratchpad a timestamp field that we will initialize to the current time when a *SYN* packet is received. Finally, when a *FIN* packet is received, we compute the difference between the current time and the value of the timestamp in the scratchpad and we display it. Doing this is enough to ensure that our component will be able to perform this operation on any number of flow at the same time as MiddleClick will always give the right scratchpad, along with the packets to process, to our element. As you can see, *TCPTime* does not have to be aware of the flow mechanism, it just has to work with the data contained in the per-flow memory area MiddleClick gives it.

An example of how this mechanism could be implemented in *TCPTime* is given in pseudocode in listing 2.3. We can see that thanks to the mechanism of scratchpad offered by MiddleClick, it is really easy and convenient to implement elements that work seamlessly on flows.

```
1 Function ProcessPacket(packet, scratchpad)
2     if CheckFlag(packet, SYN) then
3         scratchpad.timestamp = Now();
4     else if CheckFlag(packet, FIN) then
5         duration = Now() - scratchpad.timestamp;
6         Display("Duration of the TCP connection: " + duration);
7     end if
```

Listing 2.3: Pseudocode of the packet processing function in TCPTime

Therefore, all the elements provided with this work in order to achieve the expected goals were developed to be integrated in MiddleClick and take advantage of its flow management system.

2.2 Framework architecture

As stated in Chapter 1, this work is meant to provide developers with a fast and easy-to-use TCP framework to create middleboxes. To achieve this purpose, it takes the form of a set of MiddleClick elements, developed in C++, that will handle the network management part of the middlebox system, allowing the developer to focus on the higher level aspects of the implementation, without having to worry about the specificities of the network. Note that in Click, elements receive raw packets, so it is the role of the framework developed in the context of this master's thesis to handle the network-related content in order to let the middlebox developers focus on the useful content of the packets. For this purpose, a lightweight implementation of a TCP/IP stack was developed, specifically designed to be used within the context of middlebox development, therefore trying to interfere as low as possible with the TCP connections.

Generally, the simplest way to implement a middlebox that will see all the traffic that passes through it is to use a device with two interfaces, say *eth0* and *eth1*. This

device may be located at the edge of the network to ensure that all the traffic must go through it in order to enter or exit the network. In this case, the middlebox receives data on an interface, processes them, and puts them back on the other interface, and vice versa for the other direction. The figure 2.3 shows an example of such a topology. In this situation, the middlebox is completely invisible for the client and the server, it is seen as a simple wire. Note that with this configuration, the interfaces of the middlebox are configured in promiscuous mode to be able to intercept the frames intended for the client or the server.

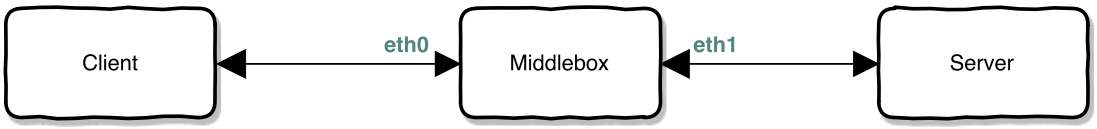

Figure 2.3: Example of a simple topology that can be used to implement a middlebox

Since the role of the elements of this framework is to carry out the functions of the protocol stack for middlebox applications, a path of elements representing a middlebox will be referred to as *the middlebox stack*. Once again, the advantage of having a TCP stack in the form of Click elements allows for flexibility and modularity: developers can choose not to use some functionalities by not using the corresponding elements, or they can decide to perform some operations at precise points of the *middlebox stack* (or even outside of the stack for some elements such as *TCPReorder* which can be used anywhere in a configuration).

The figure 2.4 depicts a minimal example of a TCP stack instance. Note that for simplicity purpose, the arguments used to configure the elements of the stack are not shown on the diagram. Moreover, in a topology such as the one depicted on figure 2.3, the left side of the configuration is ultimately connected to the interface *eth0* while the right side of the configuration is connected to the interface *eth1*. Therefore, the top path manages packets coming on the interface *eth0* and puts them back on the interface *eth1*, while the bottom path manages packets flowing in the other direction. In such a configuration, the two paths are the two directions of the TCP connections. For the sake of convenience, the second path is often referred to as the *return path* of the first. As you can see, the configuration depicted on this example does not do anything special and does not implement *middlebox functions*, the packets just go through the *middlebox stack* which leaves them intact.

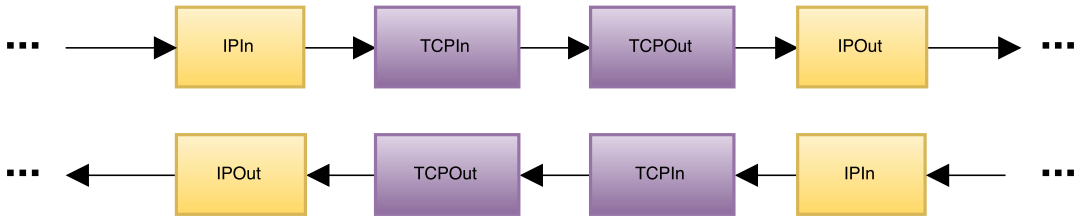

Figure 2.4: Minimal example of a TCP stack instance. The arguments for the elements of the stack are not shown for simplicity purpose

To implement a specific middlebox function (for instance *remove all the insults in web pages*) with the help of this framework, a developer has to create a new MiddleClick element that inherits from *StackElement*, which is the abstract element developed in the context of this work from which every element of the *middlebox stack* inherits. It allows the newly created element to use specific functions of the stack (described in the section 2.4) in order to insert or remove bytes in the flow, for instance. Moreover, at any point of the stack, the developer can use a method called *getPacketContent*, defined in *StackElement*, to obtain the current useful content of the packet. If this method is called in a element located after a *IPIn* element, it will return the IP payload of the packet. If it is called in an element located after *TCPIn*, it will return the TCP payload. Finally, if it is called after an *HTTPIn* element, it will return the HTTP payload. This technique is used to allow the elements to be agnostic of the protocols used by the lower layers of the network. The developer will then create a Click configuration and put his or her element in the middle of the *middlebox stack* (namely, between *TCPIn* and *TCPOut* in the example depicted on figure 2.4).

2.3 List of components

This section presents and describes the framework elements developed in the context of this work. A summary of their characteristics is provided at the end of the section, in table 2.1.

2.3.1 IPIn

IPIn is the entry point of the stack for IP packets. All the IP packets must pass through this element before their IP content is processed. This element has one input and one output and does not take any argument for configuration.

When a packet goes through an *IPIn* element, it is annotated with the position of the IP payload so that any further call to *getPacketContent* on this packet will return the IP payload.

2.3.2 IPOut

The *IPOut* element is the twin of *IPIn*, it is the exit point for IP packets in the stack. All packets that went through an *IPIn* element must go through an *IPOut* element before they exit the middlebox stack. This element has one input and one output. It does not take any argument.

IPOut recomputes the IP checksum of the packets to ensure that it is still correct after the modifications that occurred during their path in the middlebox stack.

2.3.3 TCPIn

As for the IP protocol, there is an entry point for TCP packets in the middlebox stack and it corresponds to the *TCPIn* element. It also has one input and one output but takes three arguments:

- **FlowDirection (mandatory):** ID of the direction of the flow (0 or 1). One of the two paths must have the ID 0 and its return path the ID 1. For instance, in the example given on figure 2.4, the top path could have the ID 0 and the bottom path the ID 1 (or vice versa). It is used internally by the stack to be able to access the structures shared by the two directions of the TCP connection.
- **OutName (mandatory):** The name of the *TCPOut* element associated to this element (which is on the same path).
- **ReturnName (mandatory):** The name of the *TCPIn* element on the return path.

Note that the arguments requiring the name of an element expect the name the user gave it in the Click configuration file. Remember the example given in listing 2.2 where we gave the name *cip* to the instance of *CheckIPHeader* to be able to configure its second output.

As for the *IPIn* element, *TCPIn* annotates the packets to indicate the beginning of their TCP payload so it can be accessed immediately by subsequent elements. It also performs multiple operations on the incoming packets:

- It manages the TCP options to determine the maximum segment size (MSS) and whether the window scale option is used, and if so, it records its value. It also automatically removes the *SACK-PERMITTED* option if it finds it, because the stack is not compatible with it. Therefore, by removing this option from the header of *SYN* packets, it makes the receiver think that the other protagonist is not compatible with selective acknowledgements. Thus, no one will use it because both endpoints think that the other does not support it.
- It updates the ACK number of incoming packets, according to modifications done by the other side of the connection. It also keeps up-to-date and computes the various parameters of the TCP connection (congestion window size, sender's window size, number of duplicate ACKs received, last ACK received, ...). Those mechanisms will be described in more details in chapter 3.
- It processes the requests of the elements located downstream in the stack. As it will be described in section 2.4, stack elements can issue requests (such as *remove bytes*, *insert bytes*, *close the connection*, ...) that will go back up in the middlebox stack until an element is able to achieve the requested operation or answer the given question. *TCPIn* therefore handles the TCP-related requests.

Updating the sequence and ACK numbers of the packets according to the modifications performed by the middlebox elements is crucial. The figure 2.5 depicts an example of data insertion and how the receiver perceives it. In this situation, an element of the middlebox stack adds 100 bytes in the packet x. We can see on *receiver's view (b)* that if the middlebox does not update the sequence number of the packet $x + 1$, the receiver perceives the first 100 bytes of the second packet as overlapping with the end of the first packet. On the other hand, if the middlebox updates the sequence number of the packet

$x+1$ according to the modification (*receiver's view (a)*), the receiver will not notice this modification. Note that when a modification is done in a packet with a sequence number equal to x, every packet with a sequence number $> x$ must be updated accordingly. Thus, for a given packet, we have to apply to its sequence number the result of all the modifications done in the flow before this point.

Updating the ACK number must also be done. In the example shown on figure 2.5, the receiver would send an ACK with a value of 2700. Obviously, this number must be modified to 2600 because the sender is not aware of the data added in the flow by the middlebox. This will be done by the *TCPIn* element located on the return path. In summary, the TCP stack must always perform the mapping between the original flow, which is the data sent by the source, and the modified flow which is the data as they are received by the destination. This mapping must be done in both directions of the connection. Remember that the TCP stack of the middlebox is invisible for the two endpoints of the connection; there is no such thing as a TCP connection between the endpoints and the middlebox. Instead, there is one TCP connection between the two protagonists, and the TCP stack of the middlebox works on the existing flow, ensuring that both endpoints continue to work and do not notice the modifications performed on the flow.

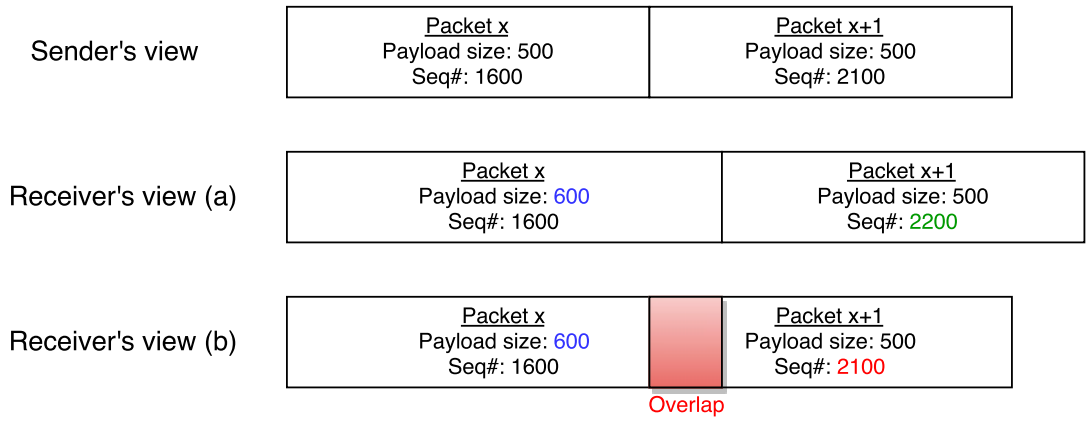

Figure 2.5: Points of view of the sender and the receiver when the middlebox adds 100 bytes of data in the packet x. The view **(a)** corresponds to the case in which the middlebox updates the sequence number of the next packet according to the modification. The view **(b)** corresponds to the case in which the middlebox does not update the sequence number of the next packet.

2.3.4 TCPOut

The *TCPOut* element is used as the exit point for TCP packets in the stack. It has one input and can have one or two outputs. The first output is the normal path for the packets of the stack and the second one, if used, allows the TCP stack to craft and send packets back to the source. It is used for instance to acknowledge a given packet. In this case, the TCP stack crafts the ACK corresponding to the given packet and sends it on its second output. Thus, the source of the given packet will immediately receive

the ACK. It is also used when a stack element makes a request to close the connection, in which case a *FIN* or *RST* packet (depending on the parameters of the request) is crafted and sent to the source. The figure 2.6 illustrates how this second output is used. Note that this second output is required in most of the configurations; the criteria will be listed in the section 2.4. Regarding its configuration, *TCPOut* does not allow any argument.

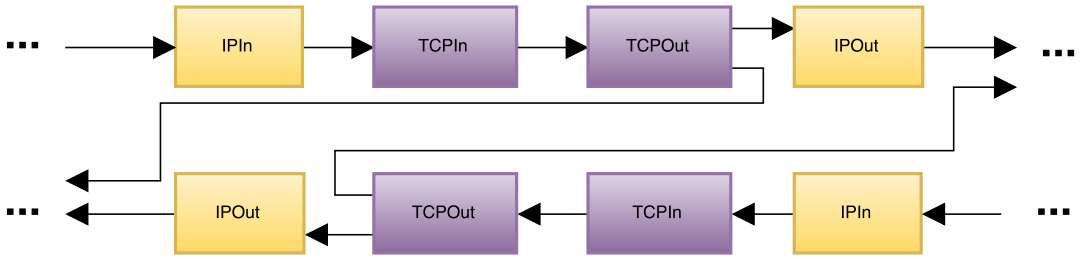

Figure 2.6: Illustration of the usage of TCPOut's second output

The role of *TCPOut* is to update the sequence number of packets according to previous modifications done in the flow, before the packets are sent to the destination. It also detects packets whose entire TCP content has been removed by previous elements, in which case it sends an ACK to the source for the initial content and discards the packet. This is required as, otherwise, the destination would receive an empty packet that it would consider as a simple acknowledgement. It would therefore not send an acknowledgement to the source in which packet we could update the ACK number as we usually do to make the source believe that the destination received the data it sent. *TCPOut* also recomputes the TCP checksum of the packets and contributes to keep the metadata about the TCP connections up-to-date.

2.3.5 TCPReoder

As the name suggests, *TCPReorder* reorders the TCP packets. It has one input and can have either one or two outputs. As we may expect, the reordered packets exit the element on the first output. Depending on whether the second output is used or not, TCPReorder handles retransmission differently. First of all, it is important to notice that retransmitted packets cannot simply go through the middlebox stack as they would be processed again, which requires useless computations, but also because it could leave the elements in an inconsistent state (they are supposed to receive ordered packets and therefore to see the data stream moving forward). Therefore, if the second output is used, retransmissions are sent on it. On the other hand, if it is not used, the retransmissions are discarded. The figure 2.7 shows how to use *TCPReorder*.

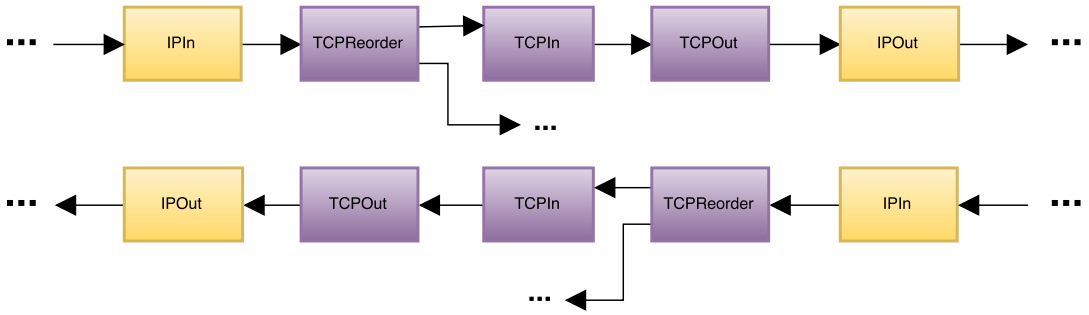

Figure 2.7: Usage of TCPReorder. The second output is optional and may be used to process the retransmissions.

TCPReorder can take up to two arguments:

- **FlowDirection (mandatory):** ID of the direction of the flow (0 or 1)
- **MergeSort (optional):** A boolean used to disable the merge sort version of the reordering algorithm. Default value: *true*. This will be explained in more details in chapter 3.

The TCP reordering mechanism is a critical component regarding the network management of middleboxes. Reordering packets is indeed necessary in many applications, depending on their needs. The following list provides the criteria that require to process the packets in order.

1. **The application modifies the stream by adding or removing data**

   If the application needs to add or remove data in the stream, the packets must be ordered. Indeed, when the size of a packet is modified because some bytes are inserted or deleted, the middlebox has to modify accordingly the sequence number of all the packets with a greater sequence number than the offset at which the modification occurs, in addition to update the corresponding acknowledgements on the return path. If this is not the case, the recipient will see a gap in the case of a deletion, and what is assumed to be a retransmission in the case of an insertion. Obviously, the middlebox can only perform this operation if the packets whose sequence number must be updated are processed after the modification occurs. Thus, a packet can never be the subject of a modification that consists in an addition or a deletion of bytes if some packets that come after it in the stream order have already been transmitted. For this reason, the packets have to be reordered before they are processed by the middlebox to ensure that it never happens.

2. **The application searches for patterns that could be split over multiple packets**

   Most of the time, an application searching for patterns in a stream will have to do it over multiple packets. For instance, the end of a packet can contain a part

of the pattern and the following packet the rest of it. This is very common with the HTTP protocol for instance, where pages are returned over several packets. If the packet containing the end of the pattern is processed first, the application will probably not be able to detect it, unless it uses dedicated algorithms to perform this task. But still, waiting for the next packet and ensuring that it is the one that comes just before in the stream order is not a straightforward task. It should not be done by the application as it requires to manipulate the TCP header, which has already been processed by a dedicated component. An example of this case is depicted on figure 2.8.

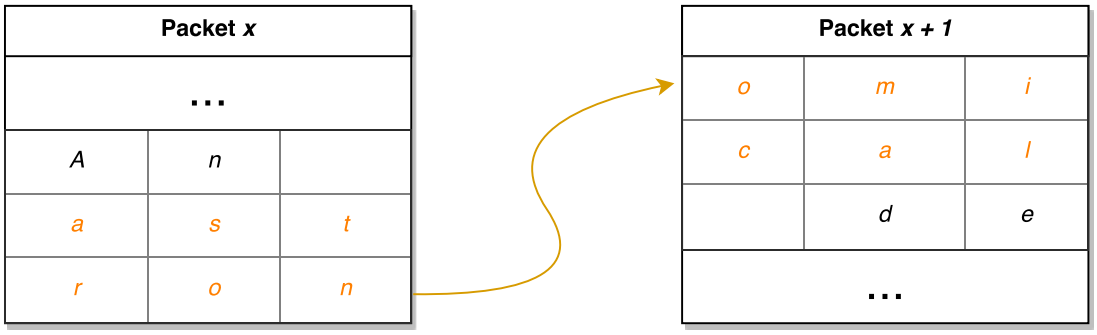

Figure 2.8: Example of a pattern split over two packets. Here, the middlebox searches for the word *astronomical* that begins at the end of the packet x and ends at the beginning of the packet $x + 1$. Only the payloads are represented.

3. **The application uses a protocol that requires it**

   A third possibility takes place when an application uses a protocol that requires to process the packets in order. Imagine a middlebox that analyses the HTTP headers in order to determine if the request has to be blocked or not. In this case, the HTTP protocol specifies that the headers will be at the very beginning of the request, just after the *start line* (which can be a *request line* or a *status line*)[7]. Ensuring that the packets arrive in the right order is thus very important because, in this way, the application can check the first packets only, without potentially buffering the packets that do not come in order, which would make the implementation much more complex. Some protocols may also require to know the content of the previous packets in order to be able to make a decision for a given packet. For instance an application could want to remove or replace the occurrences of a given word if this word already appeared a given number of times in the past.

According to those 3 criteria, we can see that most of the applications will require to get the packets in order. The reordering component is thus of the highest importance in this framework and must therefore be as efficient as possible. A bad implementation could have a large impact on the performance of the whole system.

Even though we have seen that the applications will generally meet one of the three criteria and thus use the TCP reordering feature, we decided to implement it as a

separate Click element instead of automatically reordering the packets in *TCPIn*, the component of the middlebox framework that manages incoming TCP connections. This is, once more, for modularity purpose. Indeed, the user may want to reorder the TCP packets before the traffic enters the middlebox stack, if the router configuration includes other features. Moreover, the user may decide to process the packets unordered to increase the efficiency of the middlebox if he or she knows that the application will not meet one of the three above criteria.

2.3.6 TCPRetransmitter

The *TCPRetransmitter* element is the part of the stack that manages retransmissions, data buffering and ensures congestion and flow control. It has two inputs: the first one is for the packets that were processed by the middlebox stack and the second is for retransmitted packets, generally provided by the second output of *TCPReorder*. On its unique output will exit all the TCP packets, retransmissions or not. Note that the mechanisms implemented by *TCPRetransmitter* are not directly integrated to *TCPOut* for modularity purpose. Indeed, managing retransmissions is not mandatory if the user knows that packets will only be read and never modified, and if the packets do not necessarily need to be in order before entering the middlebox stack. Therefore, one can choose not to use this component, and this is the reason why *TCPOut* and *TCPRetransmitter* are two separate elements.

This element may be configured using one argument:

- **InitialBufferSize (optional):** Initial size of the circular buffers used to store the data waiting to be acknowledged. Note that it is only the initial size of the buffers, they will automatically grow, if needed, to adapt to the amount of data *in flight*. Default value: 65535 bytes.

On the figure 2.9, you can see an example of usage of *TCPRetransmitter*. The second output of *TCPReorder* is connected to the second input of *TCPRetransmitter* to be able to handle retransmissions.

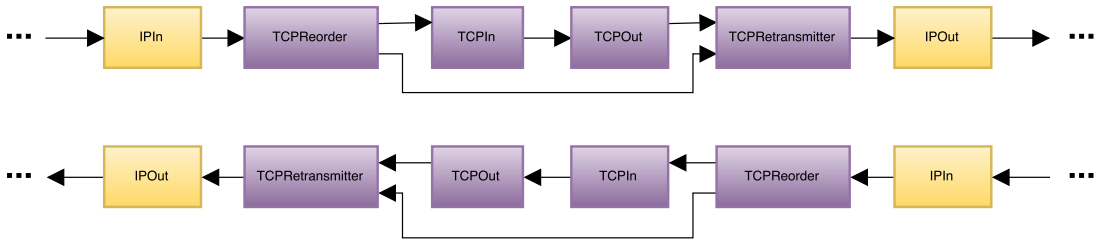

Figure 2.9: Example of usage of TCPRetransmitter.

When packets have been processed by the elements of the middlebox stack, they arrive on the first input of *TCPRetransmitter*. The content of the packets is then stored in a buffer, and the packets are transmitted to their destination. If a packet loss occurs between the middlebox and the destination, the sender will never receive the corresponding ACK and will retransmit the packets. This retransmission will be detected by

TCPReorder, which will put the retransmitted packets on the second input of *TCPRetransmitter*. The latter will analyse the sequence number of the retransmitted packet, as well as its length, and perform a mapping to determine to which data it corresponds into the flow modified by the middlebox stack. It will then retransmit it from the buffer. It is thus important to notice that the size of the content retransmitted to the destination is not necessarily the size of the content retransmitted by the sender. Indeed, as an example, imagine that the content of the packet retransmitted by the sender corresponds to content removed from the flow by the middlebox stack. In this case, when performing the mapping from the *initial flow* (as seen by the sender) to the *modified flow* (as seen by the receiver), *TCPRetransmitter* will notice that there is in fact nothing to retransmit. Actually, this example is a special case in which *TCPRetransmitter* will need to send an ACK for the retransmitted content to the sender, or it will continue to retransmit the same content forever as we are not retransmitting anything to the destination that will therefore not send an ACK by itself.

Another problem that *TCPRetransmitter* must handle is that when an element of the middlebox stack puts packets in a buffer and therefore uses the *requestMorePackets* request, the TCP stack sends an ACK to the source to ensure it continues to send packets. The consequence is that the data we just acknowledged is now under our responsibility. Indeed, the source thinks that the destination correctly received the data and will therefore never retransmit them, even if the data get lost between the middlebox and the destination. Thus, *TCPRetransmitter* has in its buffer data that it must take care of and ensure it is correctly received by the destination. To handle this problem, *TCPRetransmitter* implements the TCP retransmission mechanisms for the data under its responsibility. To do this, it follows the guidelines provided by the standards of the TCP protocol[8][17][9] and implements a *retransmission timer*. *TCPRetransmitter* constantly computes the round-trip time (RTT), according to Karn's algorithm, between the middlebox and the destination to determine the *retransmission timeout*. When the timer fires, *TCPRetransmitter* considers that the data between the middlebox and the destination were lost and it resends them.

The last issue that *TCPRetransmitter* handles is once again related to packet buffering. We have seen that, when an element puts packets in a buffer, the TCP stack becomes responsible for the data in question. The problem occurs when an element flushes its buffer and releases the packets in the middlebox for further processing. All these packets will actually arrive at the end of the TCP stack at the same moment, all at once. When possible, the TCP stack of the middlebox lets the two endpoints manage the connection as much as possible, but in this case, it has to implement the flow and congestion control mechanisms of TCP to avoid sending a large amount of data on the network, without ensuring that it does neither exceed the receiver's window size nor congestion the network. As an example, imagine the case of an element that needs to put in a buffer the whole content of a web page before making a decision. When the end of the web page is received, it will release its entire content as a burst. Thus, *TCPRetransmitter* implements TCP Tahoe, including the slow start algorithm, to ensure flow and congestion control to the data it is responsible for. On the other hand, for the data that it is not responsible for, *TCPRetransmitter* lets the source manage the transmission rate and does not interfere with it.

2.3.7 HTTPIn

Creating an HTTP stack was not really in the scope of this work, but an elementary one has nonetheless been developed to be able to test the TCP/IP stack in real conditions. Hence, the *HTTPIn* element is the entry point of HTTP packets in the middlebox stack. Regarding its configuration, it is very simple: it has one input, one output and does not take any argument.

The role of *HTTPIn* is, as it is the case for *IPIn* and *TCPIn*, to annotate the packets to indicate the beginning of their payload, which is located after the HTTP headers. *HTTPIn* also modifies the HTTP versions in requests to set it to *HTTP 1.0* as it is not compatible with the mechanisms implemented in further HTTP versions, such as chunked transfer encoding. It also removes the *Accept-Encoding* header to ensure that the server will not reply with encoded data, in order to compress them for instance. Finally, *HTTPIn* gets the value of the *Content-Length* header so that it can determine when the entire web page has been received.

The limitation of the HTTP stack implemented in the context of this work is that all the headers must be included in the same packet in order for *HTTPIn* to be able process them, which is fortunately the case most of the time.

2.3.8 HTTPOut

HTTPOut is the exit point of HTTP packets in the middlebox stack. It has one input and one output. It does not take any argument and its role is to modify the *Content-Length* header so that it corresponds to the new length of the content, after it has been processed by the middlebox. This allows the receiver to be able to process the HTTP content properly. An example of the usage of *HTTPIn* and *HTTPOut* is shown on the figure 2.10.

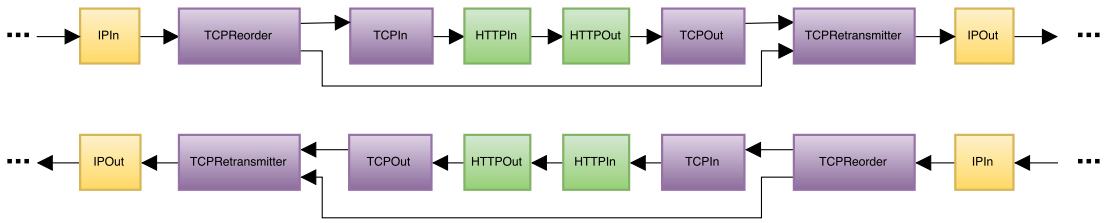

Figure 2.10: Example of usage of HTTPIn and HTTPOut.

2.3.9 InsultRemover

InsultRemover is an example of an element that would be developed by a user of this framework. Its role is to remove insults from web pages, or to block the web pages returned by the server by replacing the requested page by an error message, if it contains an insult. It is provided with this framework to show the developers how to use the framework to implement the middlebox functions they want.

This element has one input and one output. It can take the following argument:

- **CloseConnection (optional):** This argument is used to indicate the behaviour to engage in when an insult is found in a web page. By default, *InsultRemover* removes the insults in the web pages before sending them to the destination. If this parameter is enabled, the content of the web pages is replaced by an error message, informing the user that the web page has been blocked. In this case, the connection is closed after this message has been sent.

Thanks to the protocol stack provided by the developed framework, *InsultRemover* is totally agnostic of the network protocols used to carry out the content it analyses. It simply checks the content of the web pages and asks the protocol stack to remove or insert data at given positions, as though it was manipulating a string. It is the role of the stack to take care of problems like updating the checksums, taking care of the acknowledgement and sequence numbers, ensuring that it does not send too much data for the network, and so on. Therefore, developing an element such as *InsultRemover* becomes straightforward and does not require more efforts than if it was developed in a classic environment using the kernel stack. In other words, we can say that developing a middlebox function with this framework does not require any knowledge of the TCP/IP protocols, and the elements such as InsultRemover should work out of the box with a different stack that uses for instance UDP and provides elements such as *UDPIn* and *UDPOut*. The figure 2.11 depicts an example of the usage of *InsultRemover* in the middlebox stack. Note that in this example, *InsultRemover* has been put only on the path taken by the packets sent by the web server, not the path followed by the request emitted by the client. If the user knows that web pages can be returned on both paths, he or she can put an *InsultRemover* on the other path too.

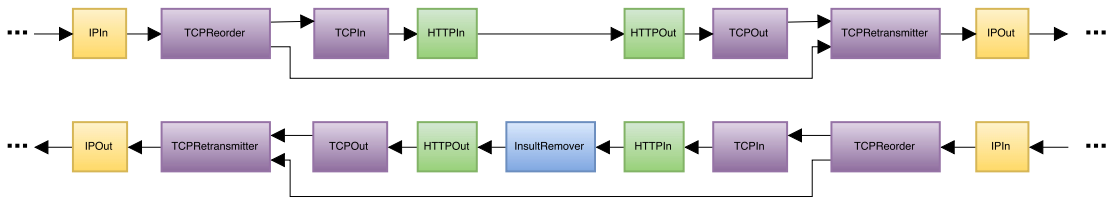

Figure 2.11: Example of InsultRemover usage.

2.3.10 PathMerger

In a Click configuration, it is possible to use built-in elements such as *Classifier* to allow packets to take different paths according to some criteria. For instance, it is possible to create two paths for the packets and make them follow the one corresponding to the protocol they use. In the middlebox stack, when an element issues a stack request such as *request more packets*, *insert bytes*, and so on, the request is transmitted upstream to the previous element on the path and it continues to be relayed element by element until someone is able to handle it. In the case of a request such as *request more packets*, it will be handled by the *TCPIn* upstream, for instance. The problem when there exists multiple paths in the middlebox stack occurs when the paths are merged, generally at the end of the stack. In this case, the element located after the merge does not know,

when it receives a request for a packet, to which upstream element it must transmit it, as it has no means to know from which element a given packet came from. This is problematic as some requests must handled by the elements located on the path from which the packet comes from. As an example, imagine that the stack contains a path for HTTP packets and a path for another TCP protocol. Those two paths lead to the same *TCPOut* element. If an element after *TCPOut* issues a request for an HTTP packet that must be handled by *HTTPIn*, how does *TCPOut* know to which element it must transmit the request, as there are two possibilities?

To solve this problem, a special element called *PathMerger* has been created. It has two inputs, the two paths to merge, and one output, the merged path. When it receives a packet, it keeps track of which of the two elements connected to it sends the packet. Therefore, when it receives a request for a given packet, it is able to determine from which element it came from and to transmit the request to it. *Pathmerger* does not accept any argument.

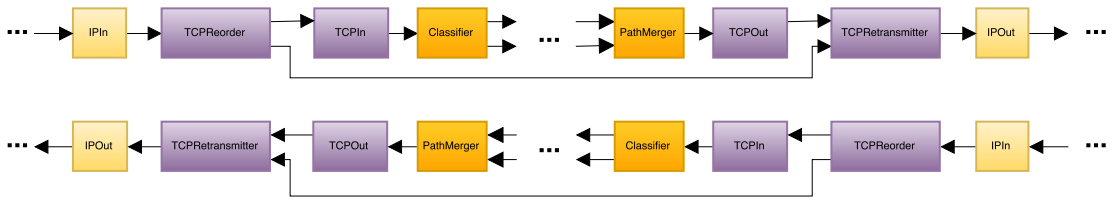

Figure 2.12: Example of PathMerger usage.

2.3.11 TCPMarkMSS

When an element of the middlebox stack adds content into a packet, it may happen that the packet becomes bigger than the *MTU* or that its payload is bigger than the *MSS* of the flow. For simplicity purpose, it was decided to use a built-in Click element called *TCPFragmenter* to ensure that it does not happen. *TCPFragmenter* is an element that can be configured with the *MTU* and which will split TCP packets bigger than this value into multiple packets. It is also possible to annotate packets with a given value that will be used by *TCPFragmenter* to determine if the packet must be split. Therefore, *TCPMarkMSS* was created to annotate the packets with the *MSS* of the flow so that *TCPFragmenter* can use both this value and the *MTU* to determine if it must split the packets. The figure 2.13 depicts an example of usage of *TCPMarkMSS* and *TCPFragmenter* to ensure the size of the packets complies with the MTU and the MSS.

TCPMarkMSS has one input and one output. It can take up to three arguments:

- **FlowDirection (mandatory):** ID of the direction of the flow (0 or 1)
- **Annotation (mandatory):** Offset of the annotation used to mark the MSS. In Click, each packet contains an annotation area that can be used by elements to associate information to a packet. The same offset must be used to configure *TCPFragmenter*

- **Offset (optional):** Offset to apply to the MSS before setting its value into the annotations of the packet. Can be positive or negative. Default value: 0.

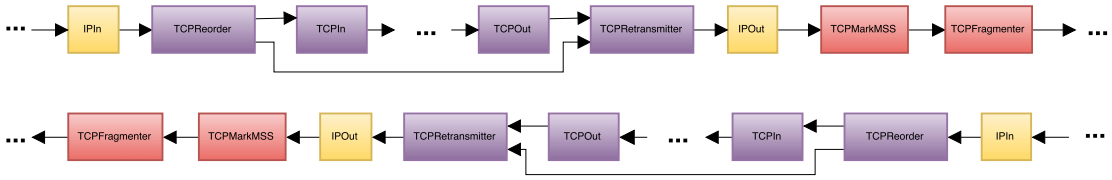

Figure 2.13: Example of the usage of TCPMarkMSS and TCPFragmenter together to ensure the size of the packets complies with the MTU and the MSS.

2.3.12 Summary

Table 2.1 provides a summary of the main characteristics of the components of the framework.

Name	Inputs	Outputs	Arguments	Short description
IPIn	1	1	0	Entry point of IP packets in the stack
IPOut	1	1	0	Exit point of IP packets in the stack
TCPIn	1	1	3	Entry point of TCP packets in the stack
TCPOut	1	1 or 2	0	Exit point of TCP packets in the stack
TCPReorder	1	1 or 2	From 1 to 2	Reorders TCP packets
TCPRetransmitter	2	1	From 0 to 1	Manages TCP retransmissions and manual transmissions
HTTPIn	1	1	0	Entry point of HTTP packets in the stack
HTTPOut	1	1	0	Exit point of HTTP packets in the stack
InsultRemover	1	1	From 0 to 1	Blocks insults in web pages
TCPMarkMSS	1	1	From 2 to 3	Annotates TCP packets with the MSS of the flow
PathMerger	2	1	0	Merges two paths in the middlebox stack

Table 2.1: Summary of the main characteristics of the elements provided by the framework

2.4 Stack functions

We have seen previously that elements of the middlebox stack can issue requests that will be propagated upstream until an element is able to handle it. This mechanism is

very powerful in the context of a modular stack such as the one developed for this thesis. Indeed, the elements issuing the requests do not need to know anything about the lower layer protocols or the context in which they are used. They simply send the request agnostically of their environment, and they know that it will be handled somehow, in the right way. A good example for this point is the request used to determine if a given packet is the last useful for the current protocol.

2.4.1 Determine if a given packet is the last useful one for the current layer

Elements of the middlebox stack can determine, thanks to this request, if a given packet will be the last containing useful information for the current layer. When an element puts packets in a buffer to be able to make a decision when it has received the full content, it needs to know when to flush the buffer, otherwise packets would stay forever in it, never reaching their destination. For this purpose, when elements receive a packet, they can issue the request called *isLastUsefulPacket* to determine if this packet is the last packet containing useful data for the current layer. As an example, *HTTPOut* puts every packet in a buffer until it has received the full web page. Once it has been done, it computes the new size of the HTTP payload to set the *Content-Length* header. Thanks to this request, *HTTPOut* can determine, when it receives a packet, if it has now received the full web page, and act accordingly.

In the current set of elements provided along with this work, there are two elements able to answer this request: *TCPIn* and *HTTPIn*. The first one to receive the request will answer it, and this answer will thus depend on the location of the element issuing the request. Indeed, we have seen that the request will be relayed upstream element by element, so if the element issuing the request is located between an *HTTPIn* and an *HTTPOut* element, we know that the request will be transmitted to *HTTPIn* before *TCPIn*, and therefore, the request will be handled by the former. On the other hand, if the element is located in a path that does not imply the HTTP protocol, or if it is located before an *HTTPIn* element, the request will be handled by the *TCPIn* element of the path.

Here is how the request is handled, depending on the element receiving it:

- *TCPIn*: To determine if a packet is the last one to contain useful content for the TCP protocol, *TCPIn* simply checks whether the packet has the *FIN* or *RST* flags enabled.

- *HTTPIn*: When it receives the packet containing the headers, *HTTPIn* determines the value of *Content-Length*. It then uses a counter called *contentSeen* which is incremented, for every packet, by the size of the HTTP content in the packet. When the counter reaches the value retrieved in the *Content-Length* header, it annotates the packet to indicate that it is the last one containing HTTP content. Therefore, when it receives a request, *HTTPIn* simply checks if the given packet has the corresponding annotation.

2.4.2 Remove bytes

When an element wants to remove bytes in a packet, it must issue the *removeBytes* method and provide it with the starting position and the number of bytes to remove. Note that an element gives a position relative to the content it sees. For instance, for *InsultRemover*, the position 0 corresponds to the first byte of the HTTP payload in the packet. As always, the elements are totally agnostic of lower layers and therefore do not see the content related to those protocols. This request is always handled by *TCPIn* which will map the position given by the element that issued the request into a position in the TCP payload and remove the requested bytes from the packet. *TCPIn* handles all the operations needed to achieve the request and keeps track of the modification in the flow to be able to map further sequence and acknowledgement numbers, as we have seen previously. After the request is complete, the element that issued the request gets the given packet back, modified in such a way that the requested content has been removed.

Note that using this request in a middlebox stack requires to use the second output of *TCPOut* as it may have to send acknowledgements to the source for the removed content, in particular if the entire content of the packet is removed during its processing.

2.4.3 Insert bytes

In addition to removing bytes in a packet, it is also possible for elements to perform the opposite operation and add bytes. This can be done via the *addBytes* request, by providing it with the packet in question, the number of bytes to add, and the position where to add them. The result of this request is very similar to the one obtained by removing bytes, except that here, the packet is returned with the added bytes initialized to 0, and the element issuing the request can set their content afterwards.

Once again, *TCPIn* will handle the request and keep track of the modification in the flow to be able to process further sequence and acknowledgement numbers.

2.4.4 Request more packets

As discussed previously, if an element adds a packet in a buffer instead of sending it immediately to the next element, it is supposed to issue a *requestMorePackets* request. Indeed, if it was not the case, as the packet would not be sent to the destination, the sender would not receive the corresponding acknowledgement. The consequence is that it would not send any further packets but instead, it would retransmit the buffered packet, thinking it has been lost.

When it receives a *requestMorePackets* request, *TCPIn* sends an acknowledgement for the given packet to the source. To achieve this, it crafts a packet with an acknowledgement number equal to the sequence number of the packet added to the length of its initial payload and sends it via the second output of *TCPOut*. Therefore, to use this request, user must have configured the second output of the latter element.

We have seen that performing such an operation implies that the TCP stack of the middlebox becomes responsible for the data in question, in particular ensuring that those

data will be received by the destination (and therefore it must manage their retransmission if the data get lost) and guaranteeing that they will be transmitted according to the flow and congestion management standards of TCP when the buffers will be flushed by the concerned elements.

2.4.5 Close the connection

Finally, the last request that can be used by the elements of the stack is called *closeConnection*. As the name suggests, it closes the TCP connection and prevents any further packet from reaching the destination.

This request can be configured via two boolean values, namely *bothSides* and *graceful*. If *bothSides* is set to false, only the path on which the element issuing the request is will be closed. On the other hand, if it is set to true, the connection will also be closed for the return path. The second parameter, *graceful*, indicates the method that must be used by the TCP stack to close the connection. If it is set to true, *FIN* packets will be used and if it is set to false, *RST* packets will be used instead.

Once the connection has been closed, the TCP stack automatically acknowledges any packet coming from the sender of the closed path and discards it, so that it is not sent to the destination for which the connection is considered to be closed. Therefore, if only one path is closed, the source continues to send data, thinking that the destination is still listening.

Note that using this request requires to configure the second output of *TCPOut*.

Chapter 3

Development and design

This chapter focuses on the design and the development of the framework, including the data structures and some elements presented in the chapter Architecture.

3.1 Development methodology

To achieve the various goals of this work and ensure that everything works as expected during the development, the methodology was a key point. To carry out this project, we used a constructive approach. The first step consisted in familiarizing with Click, developing the most basic elements, with the most basic features, not taking advantage of advanced techniques such as packet batching. This is noteworthy as it appears that some of the algorithms used when processing one packet at a time may not always be the most efficient when processing batches of packets.

A good example of this arose when creating the *TCPReorder* element. Fortunately, this constructive approach, although it sometimes required to be careful and have a critical thinking, never required to rethink from scratch what had been done before. Retrospectively, this approach helped a lot and was quite fitted to achieve the goals. Indeed, it turned out to be extremely helpful to implement, one by one, the required features of the TCP protocol. Some of the network specificities can be challenging to deal with and considering difficulties one at a time helped to stay focused on them. Moreover, it allowed to define and reach multiple milestones during the development and to test the current state of the framework on real traffic, and hence avoiding regressions.

3.2 Data structures

To achieve the goal of developing a fast and lightweight middlebox TCP framework, it is mandatory to carefully select and implement the data structures that will be at the heart of the system. In order to achieve the best performance, the time and space complexities were essential criteria to take into account during the development. In this section we describe the most important data structures and we justify the choices we made.

3.2.1 MemoryPool

Most of the data structures used by the elements of the framework need to allocate memory to store information about the packets and the flows, and to release this memory once they do not need it any more. Therefore, an efficient memory management is of the highest importance. In order to be as efficient as possible, a middlebox cannot simply request memory with a mere *malloc* each time it needs to store information, which can happen for every packet. Indeed, a heavy usage of *malloc* and *free* would create a bottleneck and slow down the packet processing system, resulting in a degradation of the performance which will thus have an impact on the traffic.

To be able to fulfill the need of a fast memory allocation mechanism, a common approach is to use *memory pools*. This data structure actually provides a space-time tradeoff. When the structure is created, memory for a given number of fixed-size elements is preallocated in order to be available immediately when requested. When the memory is not needed any more, instead of freeing it, the memory pool just adds it back in the list of free memory chunks so that it is available for a further request. If the need of memory is greater than the expected one and no more memory chunks are available to fulfill a request, the memory pool exceptionally performs a new memory allocation so that the pool grows.

In the context of this work, we implemented a generic memory pool mechanism that can handle any data type. The main concerns regarding its design were to ensure that its performance was as good as possible, obviously, but we also wanted to reduce the impact of the space-time trade-off by limiting its space footprint. To do so, we chose a very simple design:

1. The memory chunks are linked together as in a **linked list**. The pool stores a pointer to the first element of the list
2. When the pool is **created**, a given number (specified as a parameter) of chunks are allocated and added to the list
3. When a chunk is **requested**, the head of the list is returned if it exists (otherwise, a new memory chunk is allocated) and removed from the list
4. When a chunk is **released**, it is added at the beginning of the list.

To minimize the memory consumption, each node of the list is represented as a *union* between **a pointer to the next node** and **the element**. Thanks to the union, the node only uses the memory corresponding to its biggest member, which is almost always *the element it stores* in this case. Thus, the list only uses memory corresponding to its elements and does not induce any overhead. The union is perfectly suited here because either the node is in the list and thus we only need memory for the pointer to the next element, or the element is used by the application after it requested it and therefore, the pointer to the next element is irrelevant. In fact, the only additional memory needed is the pointer to the head of the list, stored by the memory pool, which makes it very efficient in terms of memory consumption.

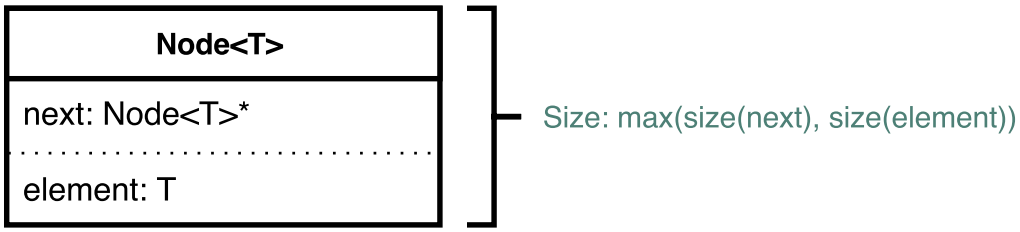

Figure 3.1: Representation of a node in the memory pool. A node is the union between a pointer to the next node and the element it stores, meaning that its size is equal to its biggest member.

Regarding the time complexity, the data structure is also as efficient as it can be. Indeed, requesting a memory chunk is $O(1)$ and thus performed in a constant time. Most of the time, the pool will not be empty and there will be no need to allocate memory. In this case, the methods only manipulate pointers to return the current head of the list and set the new one. Moreover, putting an element back in the list if it is not used any more is also $O(1)$. Here, the structure just adds the elements it gets back to the front of the list, which is done by modifying two pointers.

Note that this implementation of *MemoryPool* requires the support of *C++11* since the pools can store complex objects. Indeed, the support of unrestricted unions that can contain non-static members with non-trivial constructor and destructor is only there since *C++11*[23].

3.2.2 BufferPool

One limitation of the *MemoryPool* data structure is that it is limited to fixed-size memory chunks, and that this size must be known at compile time. This allows to use concepts such as the *union* to provide a very convenient interface to use them. However, sometimes, the limitations engendered are not compatible with the requested usage. For instance, the buffers used by *TCPRetransmitter* must grow if they become too small to store all the data *in flight*. Moreover, their initial size should be configurable by using an argument. This is typically an example of an usage that cannot be achieved via the *MemoryPool* data structure. As a consequence, *BufferPool* was developed. It allows to create a pool of buffers that can be resized at any moment and whose initial size may be configured at run time.

To implement resizeable memory chunks, *BufferPool* uses *Vector*[13], a container quite similar to the one provided by the C++ Standard Template Library (STL). This container is growable and uses a contiguous memory area to store its content. Therefore, it can be used as a buffer that can be resized.

The interface provided by *BufferPool* is very similar to the one offered by *MemoryPool*, except that since it cannot benefit from the *union* mechanism as the size of the elements it stores is not fixed, it cannot directly return a pointer to the memory chunk when the user requests it. Instead, it returns a *BufferPoolNode* element that can be

used to access the buffer. Additionally, when users want to put the buffer back into the pool, they must provide the *BufferPoolNode* they obtained.

As it is the case for memory pools, *BufferPool* avoids allocating new memory each time a memory chunk is needed. It only needs to perform such an operation if the pool of buffers is empty or if a buffer is resized with a size greater than the capacity of the underlying *Vector*.

Requesting a buffer from the pool and putting it back is, like for the memory pools, an operation with a time complexity of $O(1)$. Resizing a buffer corresponds to a linear time complexity regarding the difference between the old size and the new one.

3.2.3 Common TCP structure

We have seen previously that when a modification is done in the flow, not only the sequence number of the subsequent packets coming on the same direction will have to be modified, but also the acknowledgements coming back on the return path. This implies that the two directions of a TCP connection have to exchange data. Actually, the two sides of the connection exchange many pieces of information, for instance the maximum segment sizes, the congestion window sizes, the round-trip-times, and so on. In MiddleClick, two TCP packets are considered to belong to the same flow if they share the same *(ip_source, ip_destination, port_source, port_destination)* tuple. Therefore, the two sides of a connection do not share the same scratchpad as they are considered to be two distinct flows. As a consequence, a common memory area in which they can exchange information had to be created.

When *TCPIn* receives a *SYN* packet, it allocates memory (via a dedicated memory pool) for a common structure corresponding to this TCP connection. It then puts a pointer to this structure in a hashtable with the tuple *(ip_source, ip_destination, port_source, port_destination)* as a key. Finally, it sets in the scratchpad given by MiddleClick, which corresponds to the flow, the pointer to the common TCP structure. In this way, it will be able to access the common structure in the future just by using the pointer in the scratchpad. On the other hand, when it receives a *SYNACK* packet, *TCPIn* asks the *TCPIn* element on the return path to get a pointer to the common memory, as it knows that it was created when it received the *SYN* packet. To do so, it requests to *TCPIn* the common structure corresponding to the tuple *(ip_**destination**, ip_**source**, port_**destination**, port_**source**)*. Finally, it also sets a pointer to this structure in its own scratchpad. As a consequence, after the *three-way handshake*, the two scratchpads corresponding to the two directions of the connection have a direct pointer to the shared structure.

3.2.4 ModificationList

As already mentioned, the TCP stack must update the acknowledgement and sequence numbers according to the modifications performed in the flows. For this purpose, a data structure that stores the modifications performed in each packet has been created. Its role is to keep track of the modifications in such a way that each element of the middlebox stack can perform modifications agnostically of what has been done before.

Such a data structure is not straightforward to set up because the underlying algorithms had to be developed specifically in the context of this work.

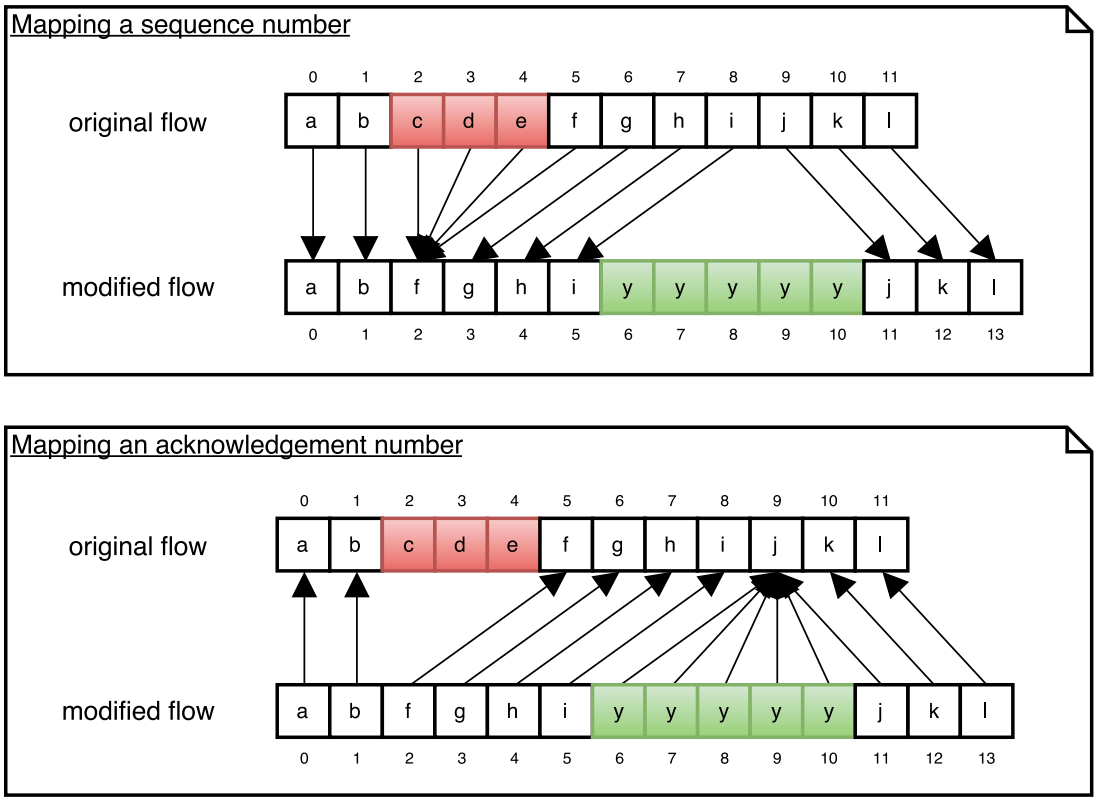

Figure 3.2: Example of the mapping between an original flow and the corresponding modified flow. The mapping can be done in the two directions in order to map both the sequence and the acknowledgement numbers. Red cells correspond to data removed from the original flow and green cells correspond to data added in the modified flow.

The figure 3.2 shows an example in which the following modifications are done to the original flow by the elements in the middlebox stack:

1. 3 bytes are removed at the position 2 via a *removeBytes(2, 3)* request, removing *cde* from the flow .

2. 5 bytes are inserted at the position 6 via an *insertBytes(6, 5)* request, adding *yyyyy* in the flow.

As you can see, the new bytes are inserted at the position 6, which corresponds to the position in the modified flow, not the original flow. This is due to the fact that an element is working on a flow that may have been modified previously, and it thus manipulates the positions relatively to the current state of the flow. The positions given in the requests therefore depend on the order in which the modifications are performed. If the modifications had been done in the reverse order, the request corresponding to the insertion would have been *insertBytes(9, 5)* instead, and the request corresponding to the deletion would have stayed the same as these positions are not affected by the

insertion. Note that both orders result in the exact same modified flow. This is the role of *ModificationList* to take into account such a specificity and to allow the elements to perform the modifications in any order, sometimes performing an operation that overlaps another one done previously.

You can also see that for a flow, there are in fact two types of mapping, one for the sequence numbers (from the original flow to the modified flow) and one for the acknowledgement numbers (from the modified flow to the original flow). Remember that the sender sees the original flow and therefore sends packet with sequence numbers relative to it while the receiver sees the modified flow and sends packets with acknowledgement numbers relative to the latter.

Another important point depicted on the figure 3.2 is that, in this case, to map a sequence number, the positions corresponding to the removed bytes are all mapped to the position 2, corresponding to *f*. Indeed, if the sender starts a retransmission with a sequence number equals to, for instance, 3, therefore trying to retransmit data from *d*, the mapped retransmission will start at *f*. This is because the beginning of the retransmission corresponds to removed data. On the other hand, when mapping an acknowledgement number, all the added data point to j in the original flow. This is due to the fact that if the destination has correctly received data between i and the last y, the sender must start retransmitting from j. Note that it will be the responsibility of *TCPRentransmitter* to ensure that *yyyyy* is retransmitted if necessary.

ModificationList uses a linked list of modifications, represented by a position (the position at which the modification occurred) and an offset that corresponds to the number of bytes modified. This offset is negative if bytes are removed and positive if bytes are added. Because the requests to perform a modification in a flow use positions relative to the current state of the flow, it is required to convert them first into positions based on the same reference. For this purpose, all the positions are first converted into positions in the original flow before being added to the modification list. To achieve this goal, the modification list is browsed and the offsets are applied to the position given in the request. As soon as the positions in the modification list become greater than the position we are computing, we stop because, since the list is sorted according to the positions, we know that further modifications do not have an influence on the position we are processing. The new modification is then added at the current place in the list, so that it stays sorted according to the positions of the modifications.

After adding a new modification in the list, the final step consists in merging nodes that represent overlapping deletions. The figure 3.3 depicts an example of such a situation. To achieve this goal, the algorithm inspects every node representing a deletion and checks if the next node in the list is also a deletion. If it is the case, it checks the offset of the first of the two deletions to determine if there is an overlap with the second modification. In such a case, the node corresponding to the second deletion is removed from the list and its offset is added to the one of the first deletion in order to combine their effects.

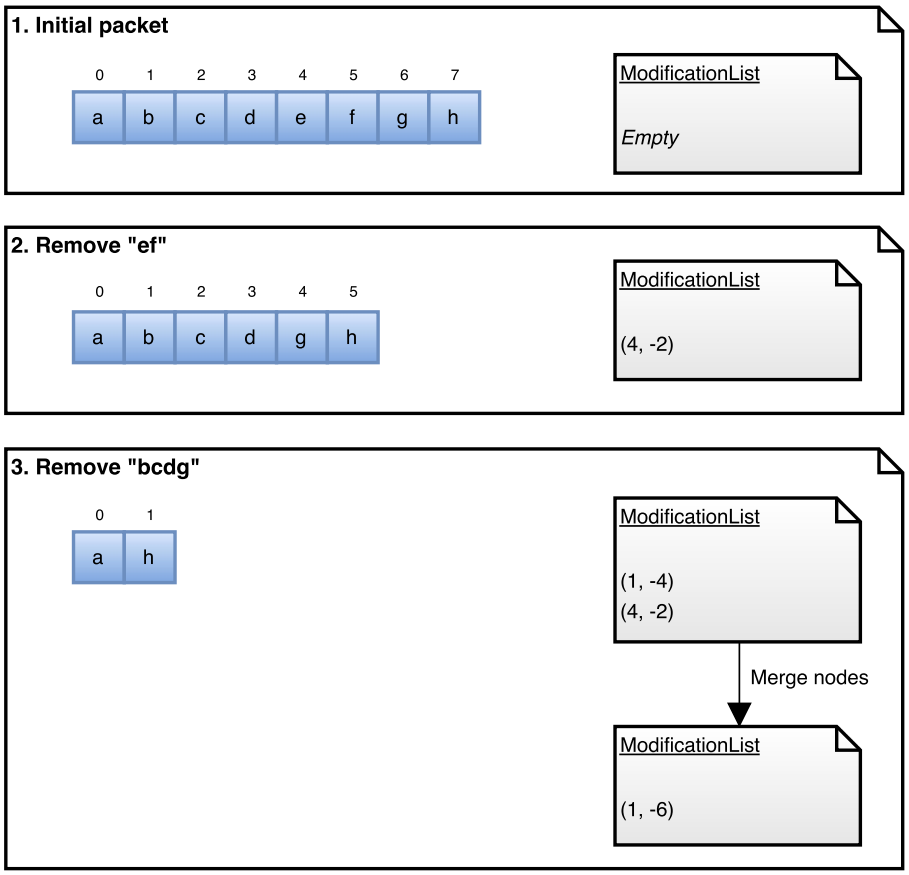

Figure 3.3: Example of two successive deletions that lead to a merge because they overlap

Regarding the complexities, adding a new modification in the list requires to browse it in order to compute the value of the position relatively to the initial flow and to put the new node to the right place. Therefore the time complexity is linear with respect to the number of modifications in the list. And because a modification list is associated to a packet, the time complexity to add a new modification is therefore $O(n)$ where n is the number of modifications in the packet.

The modification lists are created and managed by *TCPIn* that handles the requests used to modify the content of the packets.

3.2.5 Red-black trees

In the context of this work, a red-black tree implementation was needed in order to be able to develop data structures with the best time complexities. It was therefore decided to adapt the implementation of Emin Martinian[15] to fit the needs of the current framework. For this purpose, the latter implementation was modified in order to enhance the red-black trees with the concept of *red-black tree manager*. This mechanism allows to associate an instance of a red-black tree to a *red-black tree manager* that will take care of the memory used by the tree as well as the comparisons between the keys. Consequently, it allows to have a very generic implementation that can be adapted to

every situation. As an example of the benefits it brings to this work, the *red-black tree managers* used for the implementation of the various data structures of this framework manage the memory used by the trees they are associated to by using memory pools.

In addition to the mechanism of *red-black tree manager*, several operations described in table 3.1 were added to the initial implementation.

Operation	**Time complexity**
Obtain the node with the greatest key less or equal to the given value	$O(log(n))$
Find the minimum	$O(log(n))$
Find the maximum	$O(log(n))$
Prune the tree and remove nodes with a key less than a given threshold	$O(k * log(n))$

Table 3.1: Operations added to the RBT implementation. Regarding the time complexities, n corresponds to the number of nodes in the tree and k corresponds to the number of nodes to remove

3.2.6 ByteStreamMaintainer

As we have seen previously, *ModificationList* stores the modifications performed on a packet, but it does not provide a straightforward way to perform a mapping on a sequence or an acknowledgement number. The role of *ByteStreamMaintainer* is to provide an interface that allows to perform such a mapping immediately and straightforwardly. For this purpose, it uses two red-black trees, one for the mapping of sequence numbers and the other for the mapping of acknowledgement numbers.

The keys of the nodes in the red-black tree used for the mapping of the sequence numbers are positions relative to the original flow, as the goal is to map a sequence number sent by the source to a sequence number that will be understood by the destination. Therefore, it maps a position from the original flow into a position in the modified flow. On the other hand, it is quite the opposite for acknowledgement numbers and the positions are relative to the modified flow. Regarding the value stored in the nodes of these trees, it is the **cumulative offset** of the modifications, so that we do not need to browse the entire tree to determine the offset to apply to our sequence or acknowledgement number.

As a matter of fact, when a packet is ready to exit the TCP stack (in *TCPOut*), the modifications listed in its *ModificationList* are committed to the *ByteStreamMaintainer* of the flow. We do that because we know that the content of the packet will not be modified any more and thus its *ModificationList* contains the final list of modifications for this packet. Working directly in the trees when a modification is performed would be possible but much more complex since the problems such as overlapping modifications and offset updating are easier to handle in a linked list. Moreover, it would not improve the linear time complexity provided by *ModificationList* as adding a modification in the trees would require to browse the nodes with a greater position to update the value of their *cumulative offset*, which also corresponds to a linear time complexity. In fact, it is even better to work first in *ModificationList* as the associated complexity

is linear regarding the number of modifications in *the corresponding packet* while with *ByteStreamMaintainer*, it is linear regarding the number of modifications *in the whole flow*. Note that working directly in the trees also requires to modify both of them when a modification is done in the flow. On the other hand, using *ByteStreamMaintainer* to perform the mapping allows to benefit from the $O(log(n))$ time complexity of the search operation in red-black trees. The solution that consists in storing the modifications in a structure that allows to manipulate them easily and to move them in a second one that allows a straightforward mapping afterwards is therefore ideal and allows to have simpler algorithms without impacting the performance.

Commit the modifications of a packet in the trees

As said previously, the *ModificationList* of a packet is committed into the *ByteStreamMaintainer* of the flow once it is about to exit the middlebox stack. Here is the algorithm used to add the modifications into the tree used to map sequence numbers:

1. The cumulative offset corresponding to the greatest key in the tree is retrieved. Because we process the packets in order, we know that this cumulative offset is the result of all the modifications performed in the flow before this packet.
2. For each modification in the list:
   (a) We add its offset to the cumulative offset
   (b) We create a new node in the tree with the key *"position in the original flow"* and we set its value to *the current value of the cumulative offset*. As the modification list stores positions relative to the original flow, we do not have to perform any modification on it.

On the other hand, here is the algorithm used to add the modifications in the tree used to map the acknowledgement numbers:

1. As for the sequence tree, the cumulative offset corresponding to the greatest key in the tree is retrieved. The only difference is that this time, we take the opposite of this value. Indeed, the offsets stored by the acknowledgement tree have the opposite value of the ones stored by the sequence tree. This is because each tree stores the same operations but in the reverse direction (adding data in the modified flow is perceived as if we removed these data in the original flow, from the point of view of the modified flow). Therefore, taking the opposite value of the cumulative offset in the the acknowledgement tree corresponds to taking the raw value of the cumulative offset in the sequence tree. As a consequence, both algorithms start with the same value for the cumulative offset.
2. For each modification in the list:
   (a) We add its offset to the cumulative offset
   (b) We map the position stored in the modification list into a position relative to the modified flow by applying to it the current value of the cumulative offset

(c) We create a new node in the tree with the key *"position in the modified flow"* and we set its value to *the opposite of the current cumulative offset* (to cancel the effects of the operation performed at the first step that reversed the sign of the cumulative offset).

The difference between the two trees is therefore that they use keys with positions relative to the flow they map. The sequence tree uses positions relative to the *original flow* while the acknowledgement tree uses positions relative to the *modified flow*. Additionally, their nodes store the same values, but they are of opposite sign.

On figure 3.4, you can see the result, along with the corresponding *ModificationList*, after the following operations are performed on a packet:

1. Remove *defg*
2. Add *yyy* before *j*
3. Remove *lm*

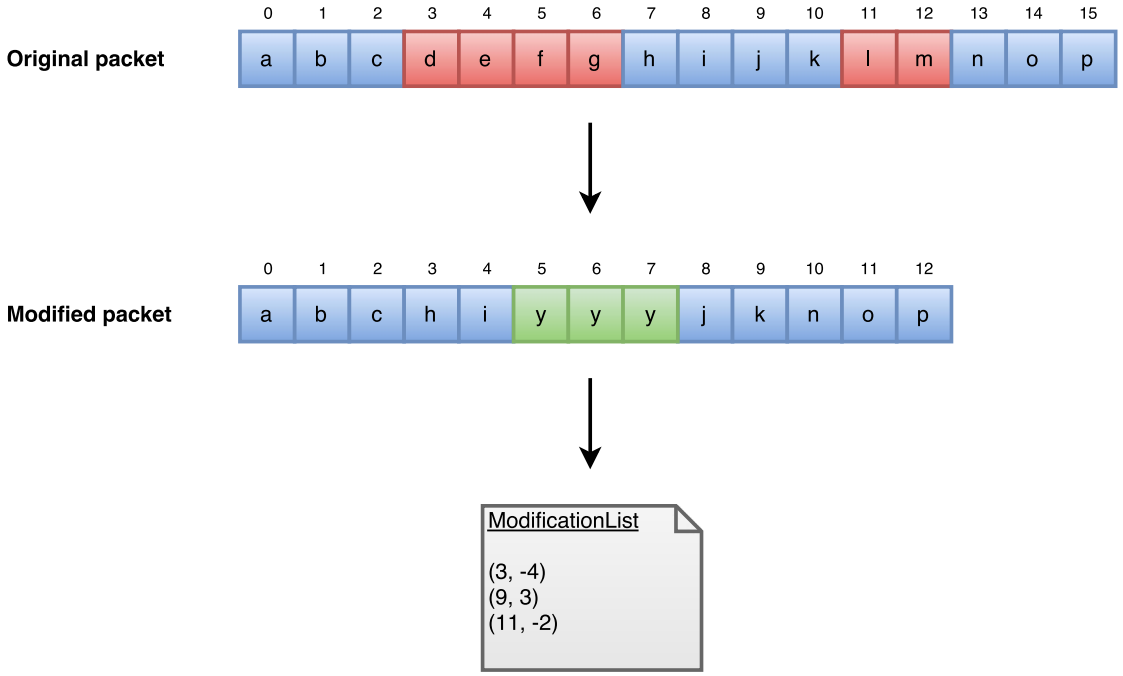

Figure 3.4: Example of three modifications in a packet and the resulting ModificationList. Red cells correspond to data removed from the initial flow and green cells correspond to data added in the modified flow.

Assuming that this packet is the first in the flow, the figure 3.5 shows the sequence and acknowledgement trees after the corresponding *ModificationList* is committed. Note that committing a *ModificationList* in the corresponding tree has a time complexity of $O(k * log(n + k))$ where k is the number of modifications in the list and n is the number of nodes in the tree before the commit.

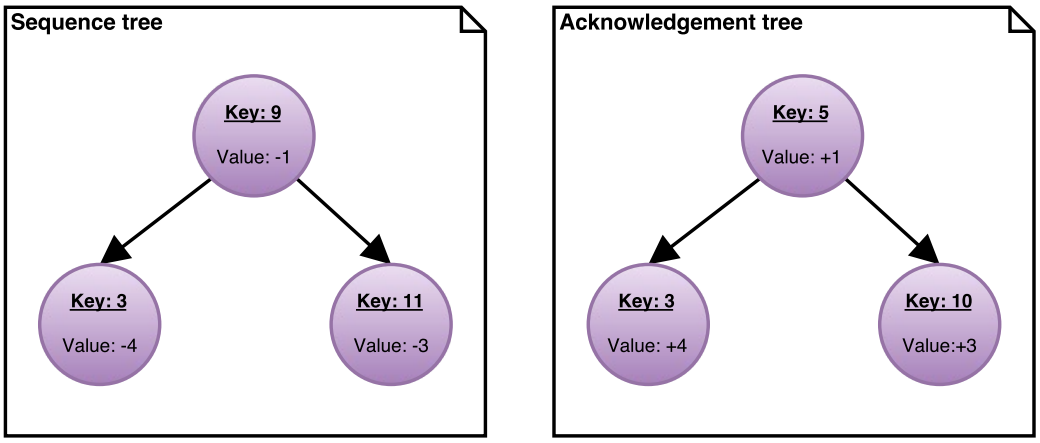

Figure 3.5: Sequence and acknowledgement trees resulting from the modifications shown on figure 3.4, assuming that the modified packet was the first in the flow.

Perform the mapping

To map a sequence or an acknowledgement number, the following algorithm is used:

1. The node with the greatest key less or equal to the given number is retrieved in the corresponding tree.
2. The offset associated to the node is applied to the given sequence or acknowledgement number
3. We then ensure that the computed number is not below the threshold obtained via the predecessor, for this purpose we perform the following operations:
   (a) The predecessor of the retrieved node is obtained
   (b) Its offset is applied to the key of the node we obtained in the first step. If the value we computed at step 3 is less than this threshold, we modify it to be equal to this bound.
4. Finally, we return the computed value

Checking that the computed value is not below the threshold obtained by adding the key of the node to the offset of its predecessor is used to manage the deletions. Remember that all the positions inside a deletion in the original flow (or an insertion in the modified flow) are mapped to the position just after it. Note that if a node is not found in the tree, is it considered to have the key 0 and the offset 0. The time complexity of the mapping is $O(log(n))$, where n is the number of nodes in the tree.

Here are some examples of mappings performed on the flows depicted on figure 3.4 and whose trees are shown on figure 3.5:

- If we want to map b from the original flow to the modified flow, we take its position, 2, and we search the corresponding node in the sequence tree. It does not exist so the corresponding offset is 0 and the position thus stays 2, which also corresponds to b in the modified flow

- To map e (position 4) from the original flow to the modified one, we take the node with the key 3 and we apply its offset (-4) to the position. We obtain 0. However, when we take the corresponding predecessor (which does not exist, so the key and the offset are considered to have the value 0), and we add its offset to the key of the node we first retrieved, we obtain $3 + 0 = 3$, therefore, the mapped position is 3, which indeed corresponds to h, the character after the deletion.
- To map the first y of the modified flow, which has the position 5, we take the node with the key 5 in the acknowledgement tree. We obtain $5 + 1 = 6$, but the predecessor gives a threshold of $5 + 4 = 9$, so the mapping corresponds to j, the first character after the insertion
- Finally, to map o from the modified flow to the original one, we add its position to the offset in the node with the key 10 and we obtain $11 + 3 = 14$. Here, the threshold obtained via the predecessor is 11 and has therefore no influence. The position 14 in the original flow does indeed correspond to o.

Prune the trees

When acknowledgements are received, the trees are pruned in order to improve the performance of the search operation. We can do this because we know that we will not have to perform mapping on acknowledged data.

More information on ByteStreamMaintainer

In addition to perform the mapping of sequence and acknowledgement numbers, *ByteStreamMaintainer* also stores information about the corresponding direction of the TCP connection. Among others, it stores information such as the value of the last ACK received, the last ACK sent, the size of the sender's window, the size of the congestion window, the number of duplicate ACKs received, and so on.

One *ByteStreamMaintainer* for each side of the connection is put in the common TCP structure, so that each side can access the *ByteStreamMaintainer* of the other side. This is used, for instance, when an acknowledgement is received: the ACK number is modified using the *ByteStreamMaintainer* of the other direction (because this is the one containing the information about the modifications performed on the other flow).

3.2.7 CircularBuffer

CircularBuffer is a data structure used to buffer data so that they can be retrieved later. The table 3.2 shows its interface.

Operation	**Time complexity**
Add data at the end of the buffer	$O(n)$
Remove data at the beginning of the buffer	$O(n)$
Get data	$O(n)$

Table 3.2: Operations provided by CircularBuffer. Regarding the time complexities, n refers to the length of the data, not the size of the entire structure.

We can see that the time complexities of these operations do not depend on the size of the data stored in the buffer, only on the size of the data we want to add/remove/retrieve. Note that even though the *get data* operation allows for random access in the buffer, it does not apply to the two other operations. This could be the case but, because *CircularBuffer* is only used in a context in which the data we want to add will be located at the end of the buffer and the data we want to remove at its beginning, it was decided not to provide such an interface as it would be useless.

To store the data, *CircularBuffer* uses an array of contiguous memory and two pointers. One pointer is called *start* and indicates the beginning of the data in the array and the other pointer is called *end*. The size of the circular buffer corresponds to the length of the data it stores (therefore the number of bytes between *start* and *end*) while its capacity is the size of the underlying array, thus, the length of the data it can store. Note that *"circular"* in the name of the data structure refers to the fact that data may wrap to the beginning of the array if needed. In this case, *end* will point to a lower index than *start*.

The peculiarity of *CircularBuffer* is that it uses a memory chunk obtained via *BufferPool*, and as you may remember, these buffers can grow if needed. Therefore, if the capacity of *CircularBuffer* becomes too small to store new data, it will increase the size of its buffer and therefore its own capacity.

When the size of the underlying buffer is increased, *CircularBuffer* faces two cases:

1. In the best case, the data stored in the array did not wrap and *end* points to a greater index than *start*. In this case, increasing the size of the buffer will not affect the data in the buffer, as depicted on figure 3.6.

2. However, if the data wrapped, *end* points to a lower index of the array than *start* and increasing the size of the buffer will therefore break the logic of the circular array as the new slots for data will be inserted between the *start* and *end* pointers, as depicted on figure 3.7. The solution is to move the data that were between the *start* pointer and the previous end of the array to the right, by an offset equals to the number of slots inserted in the array, so that the data stay at the end of the array, as previously, and the wrapping logic remains correct.

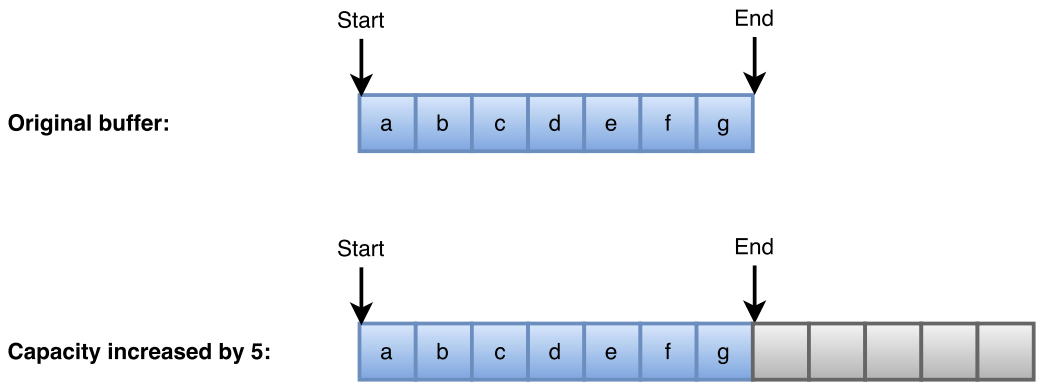

Figure 3.6: Best case obtained when the capacity of the circular buffer is increased

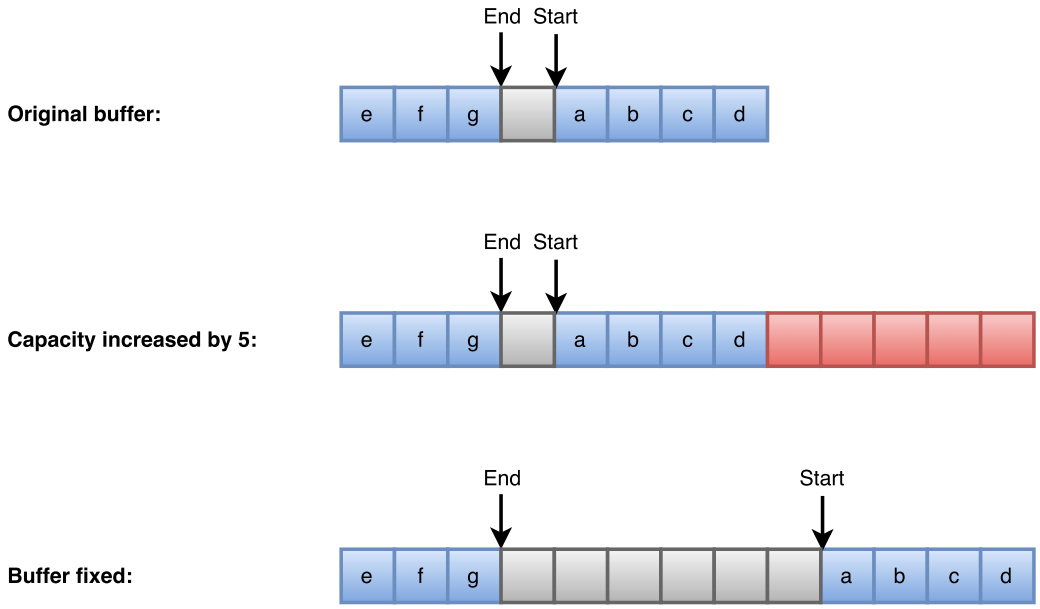

Figure 3.7: Case where increasing the capacity of the buffer leads to the need of fixing it. The cells in red are the added slots in the buffer and we can see that because they are after the start pointer, they are considered to be part of the data currently stored.

An important thing to notice is that increasing the size of the buffer may be costly and is not expected to be needed under normal conditions. The user of this data structure has to carefully select the initial capacity of the buffer used by *CircularBuffer* in order to avoid this situation as much as possible.

3.2.8 FlowBuffer

As the elements of a Click configuration manipulate packets, it is not convenient for them to perform some operations such as searching a specific pattern in the flow. Indeed, this pattern could be split over multiple packets, as we have seen on figure 2.8. The role of *FlowBuffer* is to provide the elements of the middlebox stack with a way to put packets in a buffer, and perform operations on it as if it were a continuous flow.

The table 3.3 lists the operations that can be performed on the content stored by *FlowBuffer*.

Operation	**Time complexity**
Search a pattern	$O(n * k)$
Replace a pattern	$O(n * k)$
Remove a pattern	$O(n * k)$

Table 3.3: Operations provided by FlowBuffer. All of them can work on patterns split over multiple packets. Regarding the time complexities, n corresponds to the number of bytes of payload in the buffer and k the size of the pattern.

FlowBuffer stores the packets as a linked list. To be able to fulfill its role that consists

in manipulating the packets as a continuous flow, a special iterator for this linked list has been developed. At any moment, this iterator points to a given byte of the content of a packet. When the iterator is increased, it either moves to the next byte of the content of the same packet, or to the first byte of the content of the next packet if it reached the end of the first one. Therefore, it is possible to rebuild the flow by increasing the iterator and dereferencing it to get the byte it points to until we reach the end of the last packet. This is exactly what the operations provided by *FlowBuffer* do. They loop on the iterator to search the given pattern as if they were manipulating contiguous data. The abstraction provided by this iterator allows to browse the flow seamlessly.

Note that the operations *replace* and *remove* automatically insert or remove content in the right packets to achieve their purpose. Moreover, all the operations return three different kinds of feedback:

1. The pattern has been found (and removed or replaced if it applies)
2. The pattern has not been found but it appears that the content at the end of the last packet of the buffer may be the beginning of the pattern. Therefore, adding the next packet in the buffer is required in order to be able to make a decision about the pattern
3. The pattern has not been found and cannot appear in the packets of the buffer. Therefore, the latter can be flushed.

3.2.9 RetransmissionTiming

RetransmissionTiming is the helper class used by *TCPRetransmitter* to manage the timings related to the retransmissions. There is one *RetransmissionTiming* for each side of the connection and both of them are stored in the common TCP structure. Among other tasks, *RetransmissionTiming* manages the retransmission timer. For this purpose, it constantly performs measures of the RTT between the middlebox and the destination. It then uses this value to compute the *retransmission timeout* that corresponds to the time after which a transmission from the middlebox to the destination will be considered as lost. Remember that because the middlebox has to acknowledge data when the elements buffer packets, it becomes responsible for these data and must ensure that they are correctly received by the destination.

The figure 3.8 depicts an example of a scenario in which the middlebox computes the two RTTs it uses. Here, *Endpoint 1* starts by sending a packet containing one byte of payload to *endpoint 2*. When the packet is about to exit the middlebox, the latter detects that it contains data and stores the timestamp at which the packet exits the middlebox. Once the corresponding acknowledgement is received, *RetransmissionTiming* is informed by the return path and computes the elapsed time to obtain the RTT between the middlebox and *Endpoint 2*. In this scenario, the packet acknowledging the data sent by *Endpoint 1* also contains one byte of payload. Therefore, the *RetransmissionTiming* of the second side of the connection will record the time at which the packet exits the middlebox so that when the corresponding acknowledgement is received, it can deduce the RTT between the middlebox and *Endpoint 1*. Note that *RetransmissionTiming* uses

Karn's algorithm to compute the RTTs and therefore ignores the retransmissions to avoid ambiguities.

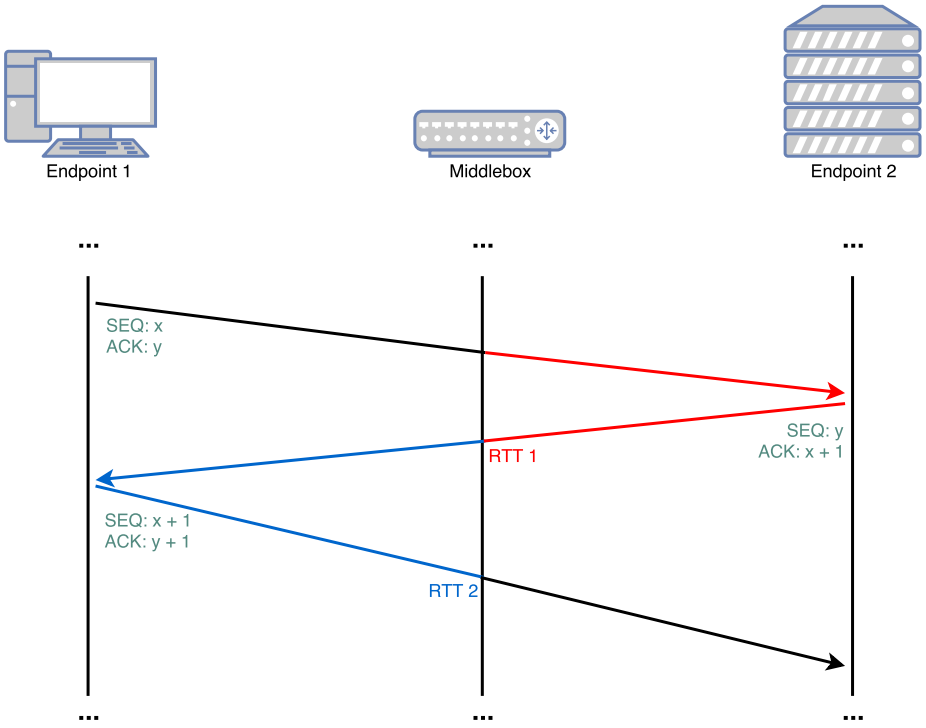

Figure 3.8: Example of RTT measurement by the middlebox. In this situation, *Endpoint 1* sends one byte of data to *Endpoint2* and *Endpoint2* sends in turn the acknowledgement along with 1 byte of data to *Endpoint 1*. The middlebox measures the RTT between itself and each endpoint.

To compute the *retransmission timeout* from the corresponding RTT, each *RetransmissionTiming* complies with the RFC2988 (*'Computing TCP's Retransmission Timer'*)[17]. This means that the following set of rules is used:

- Before the first measure of the round-trip time, the *retransmission timeout* is initialized to 3 seconds
- After the first measure of the RTT, the following computations are done:

$$smoothedRTT = measuredRTT$$
$$variationRTT = \frac{measuredRTT}{2}$$
$$retransmissionTimeOut = smoothedRTT + max(G, K \times variationRTT)$$

Where G is the clock granularity (in seconds) and K is set to 4.

- After each subsequent measure of the RTT, the values are updated as follows:

$$\begin{aligned} variationRTT &= (1-\beta) \times variationRTT \\ &\quad + \beta \times |smoothedRTT - measuredRTT| \end{aligned}$$

$$\begin{aligned} smoothedRTT &= (1-\alpha) \times smoothedRTT \\ &\quad + \alpha \times measuredRTT \end{aligned}$$

$$retransmissionTimeOut = smoothedRTT + max(G, K \times variationRTT)$$

  Where K and G have the same values as before. Additionally, α is set to $\frac{1}{8}$ and β is set to $\frac{1}{4}$, as suggested in *'Congestion avoidance and control'*[9].

- When the retransmission timer expires, the *retransmission timeout* is doubled before the timer is restarted
- The value of *retransmission timeout* is never below 1 second and never above 1 minute.

3.3 More information about some framework elements

In this section, we focus on important aspects of the implementation of some elements provided with this framework for which additional explanations are relevant.

3.3.1 StackElement

StackElement is the abstract element from which the components of the middlebox stack must inherit in order to use the stack interface. It allows to use the requests described in the section *Stack functions* as well as methods such as *getPacketContent* in order to have a direct access to the content corresponding to the current layer of the protocol stack. It also provides the method *processPacket* that can be overridden by the elements to implement the behaviour to engage in when a packet is received. Its advantage is that it works seamlessly with batching, so that elements that do not need to manage packet batches in a specific way do not have to change anything in order to be compatible with batching. On the other hand, if an element needs to implement batching in a different way, more complex than just repeating the same process on all the packets of the batch, it is still possible to do it.

3.3.2 TCPReorder

One interesting thing to notice regarding the implementation of the *TCPReorder* element is that making it compatible with packet batching, after implementing a first version that received one packet at a time, was not as straightforward as we might have thought. Indeed, the simple approach that consists in taking an element that works with one packet at a time and loops to process all the packets making up the batch may not be the most efficient in all cases.

First implementation

The first implementation of TCPReorder was done without taking advantage of the packet batching improvement provided by Middleclick in order to have a first working draft. This element works with a linked list of packets, sorted according to their sequence number, waiting to be sent to the destination. The list can contain gaps, meaning that packets with a sequence number smaller than some of the list are yet to be received. *TCPReorder* also keeps track of the sequence number of the next expected packet, in order to determine whether an incoming packet arrives in order or not.

When a packet is received, the following algorithm is performed:

1. The packet is analysed in order to determine if it is the first one of the flow in this direction. To do so, the TCP flags are checked to see if the *SYN* flag has been set. If it is the case, the variable containing the sequence number of the expected packet is initialized to the sequence number of the current packet, the first of the flow in this direction.

2. The sequence number of the packet is checked to determine if it has already been transmitted before. If it is a retransmission, the packet is either dropped or sent on the second output of the element. To determine whether a packet has already been transmitted before, its sequence number is compared to the expected one. If it is smaller, it is considered as a retransmission.

3. Next, the packet is added in the list of waiting packets at the right position, ensuring that the list remains sorted

4. Then, the list of waiting packets is browsed. For each packet, there are two possibilities:

   - The current element of the list is not the expected packet. Because retransmission are not allowed, it can only mean that we are still waiting for a packet with a smaller sequence number, it is therefore not useful to continue browsing the list as it is sorted and other packets will necessarily have a greater sequence number. In this case, we have to wait for the expected packet to arrive.
   - The current element of the list is the expected one. In this case, we update the variable containing the sequence number of the expected packet, we remove the current element from the list and send it to the next Click element for processing. In this case, the exploration of the list continues until a gap is found.

The algorithm is described on figure 3.9.

Making the element compatible with packet batching

The straightforward solution to take into account packet batching consists in repeating the first part of the process (from *receiving a packet* to *adding it at the right position in*

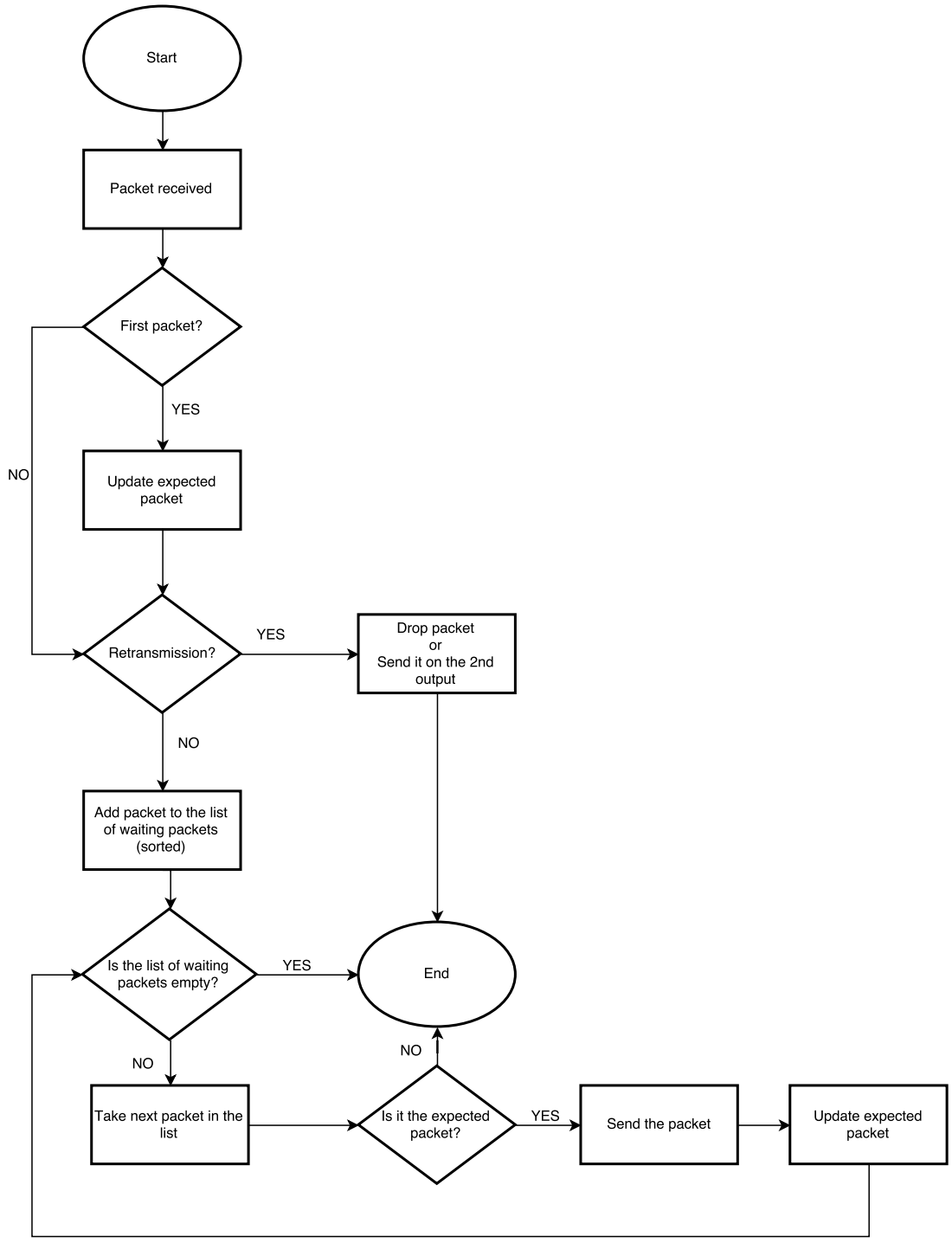

Figure 3.9: Algorithm used by the first implementation of TCPReorder, processing one packet at a time instead of batches.

the list of waiting packets) for every packet in the batch. The next part of the process, *exploring the list of waiting packets to send them in order*, stays the same. This solution, which consists in repeating the first implementation of the algorithm on every packet of the batch, is easy and fast to implement. Moreover, it is most of the time perfectly suitable to use it to make an element compatible with packet batching. However, we will see that, in this case, this approach is not necessarily the most efficient and thus requires to perform some analyses in order to determine when it is suited or not.

If we take a closer look at this version of the algorithm, we can see that we are sorting a list of packets, according to their sequence number. The packets are sorted as in the insertion sort algorithm: for each packet to add, we browse the list until we find a packet with a sequence number greater than the one of the packet we want to add. Since the list is sorted, we have found the position of the packet to add. These steps are then repeated for every packet in the batch in order to add them at the right position in the list. This algorithm is depicted on figure 3.10. Because this approach is the one used by the well-known sorting algorithm *insertion sort*, we know that we will have a quadratic time complexity. Let us analyse it in more details:

- **Time complexity:** The time complexity of this approach is $O(k * (n + k))$. Indeed, for each packet of the batch (of size k), we browse the list of waiting packets (starting at n elements and growing when we add elements from the batch, therefore bounded by $n + k$) to determine where to add it. This time complexity is indeed quadratic, as we expected.

- **Worst case:** The worst case occurs when the list of waiting packets has to be explored entirely to add a packet from the batch. This occurs when the sequence number of the packet we want to add is the largest in the waiting list. This is due to the fact that this list is sorted in increasing order. Thus, when the packets arrive in order, the algorithm needs to check all the elements in the waiting list to add the packets. However, if the packets arrive in order, the list is flushed after each batch as the packets are sent to the destination, meaning that, when the next batch is processed, the list of waiting packets will be empty. In this case, since the packets in the batch are in order and added one after the other in the waiting list, each one added at the end, the time complexity is $O(k^2)$. This is not the worst case, it is even better than the average one, $O(k * (n + k))$. To be in the worst possible conditions, we have to avoid the advantage given by the fact that packets arrive in order: the waiting list is flushed after each batch. The worst case therefore occurs when the packets arrive in order, **but**, occasionally, a packet is lost. In this configuration, after a packet has been lost, the packets continue to arrive in order and they are added at the end of the list of waiting packets, which is not flushed between each batch. The awaiting packets will indeed be sent when the lost packet will be retransmitted. In this configuration, we have an $O(k * (n + k))$ time bound, which gives the worst case complexity.

- **Best case:** We have seen that, when packets arrive in order, we have the strong advantage that the list of waiting packets is reset between each batch. This gives a clue to determine the best conditions for this algorithm. We also have seen that,

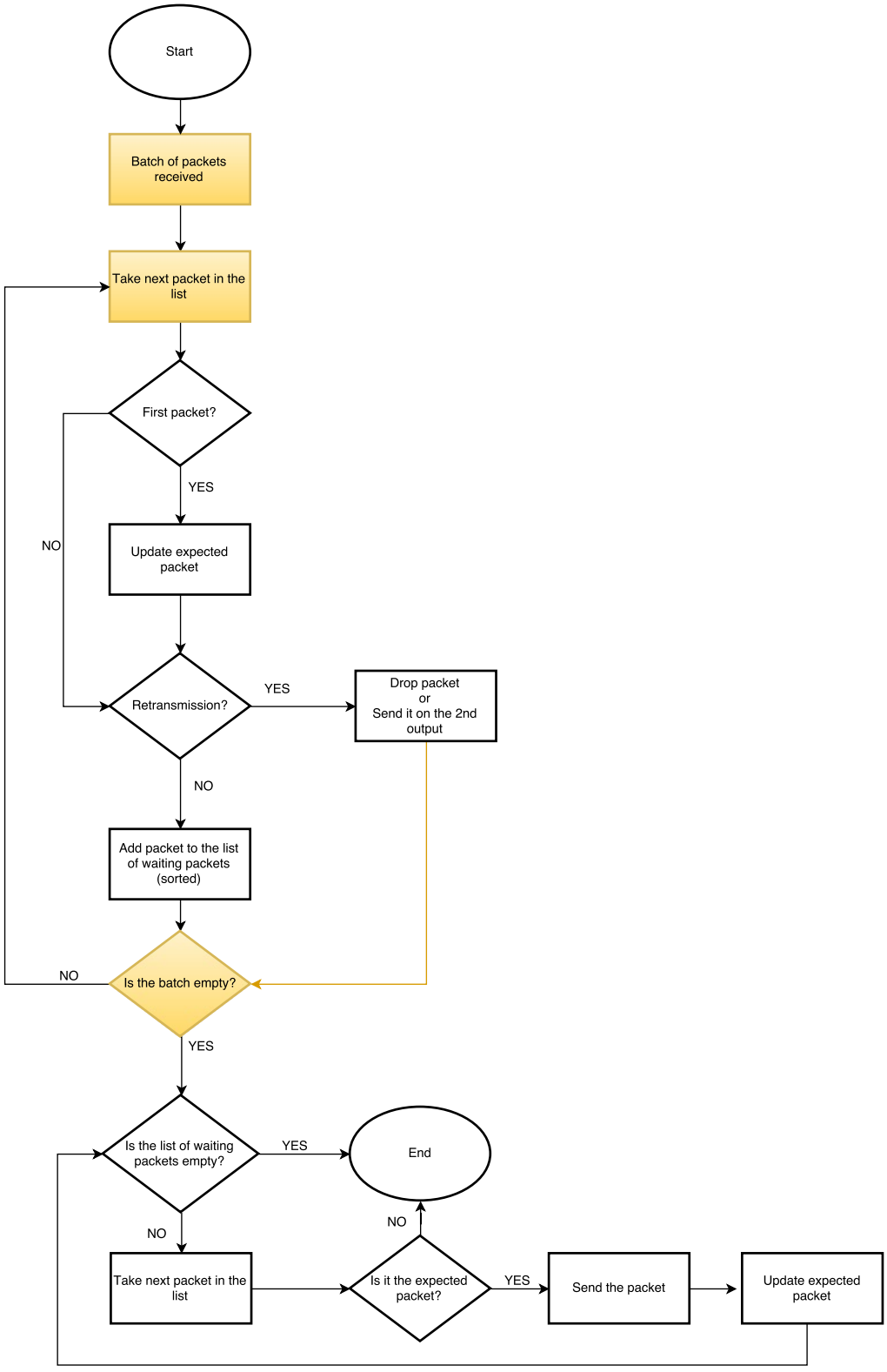

Figure 3.10: First version of the algorithm that integrates packet batching in TCPReorder. The new elements are in yellow.

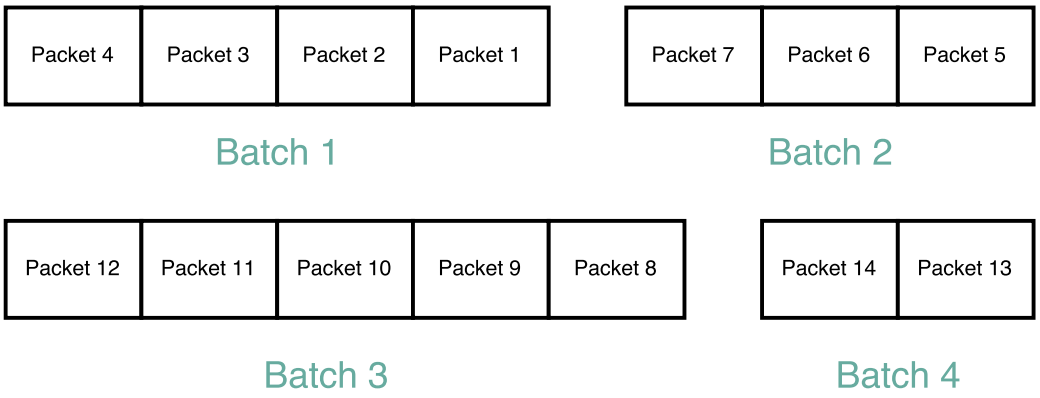

Figure 3.11: Example of situation that leads to the best case for the first version of the algorithm that includes packet batching in TCPReorder

in this case, because the batch is sorted in increasing order, each element is added at the end of the list of waiting packets, which requires to explore it entirely for each packet of the batch. We can improve this situation if each batch is sorted in reverse order. Indeed, the list of waiting packets will not have to be explored in this case because every packet of the batch will be added at its beginning. Thus, we can see that the best situation occurs when the packets globally arrive in order so that the list is flushed after each batch, but each batch is in reverse order so that the packets are added at the beginning of the list. In those conditions, we obtain an $O(k)$ time complexity. An example of a situation that leads to this case is depicted on figure 3.11

This average time complexity is not the best we can obtain regarding sorting algorithms. Indeed, we know that there exist algorithms able to sort a linked list with an $O(n * log(n))$ time bound. Thus, it may be worth considering an alternative to this approach that involves a better sorting algorithm and determine if we can improve the performance.

When it comes to sorting linked list, a reasonable choice is to perform a *merge sort*. It is indeed possible to implement this algorithm in-place, using only a small and constant extra space. Moreover, merge sort has a time complexity of $O(n * log(n))$, even in the worst case. Thus, it was decided to implement it to improve the performance as it provides excellent space and time bounds. The implementation is based on the work of Simon Tatham[22].

The new version of the algorithm therefore simply adds each packet of the batch at the beginning of the list of waiting packets, which is an $O(1)$ operation, and, at the end, sorts the list using merge sort, which is $O((n+k) * log(n+k))$ as we are sorting a list of n elements to which we just added k new elements. This algorithm is depicted on figure 3.12.

Finally, as depicted on figure 3.13, the merge sort version of the algorithm quickly becomes better as the values of k and n increase and it thus provides good performance

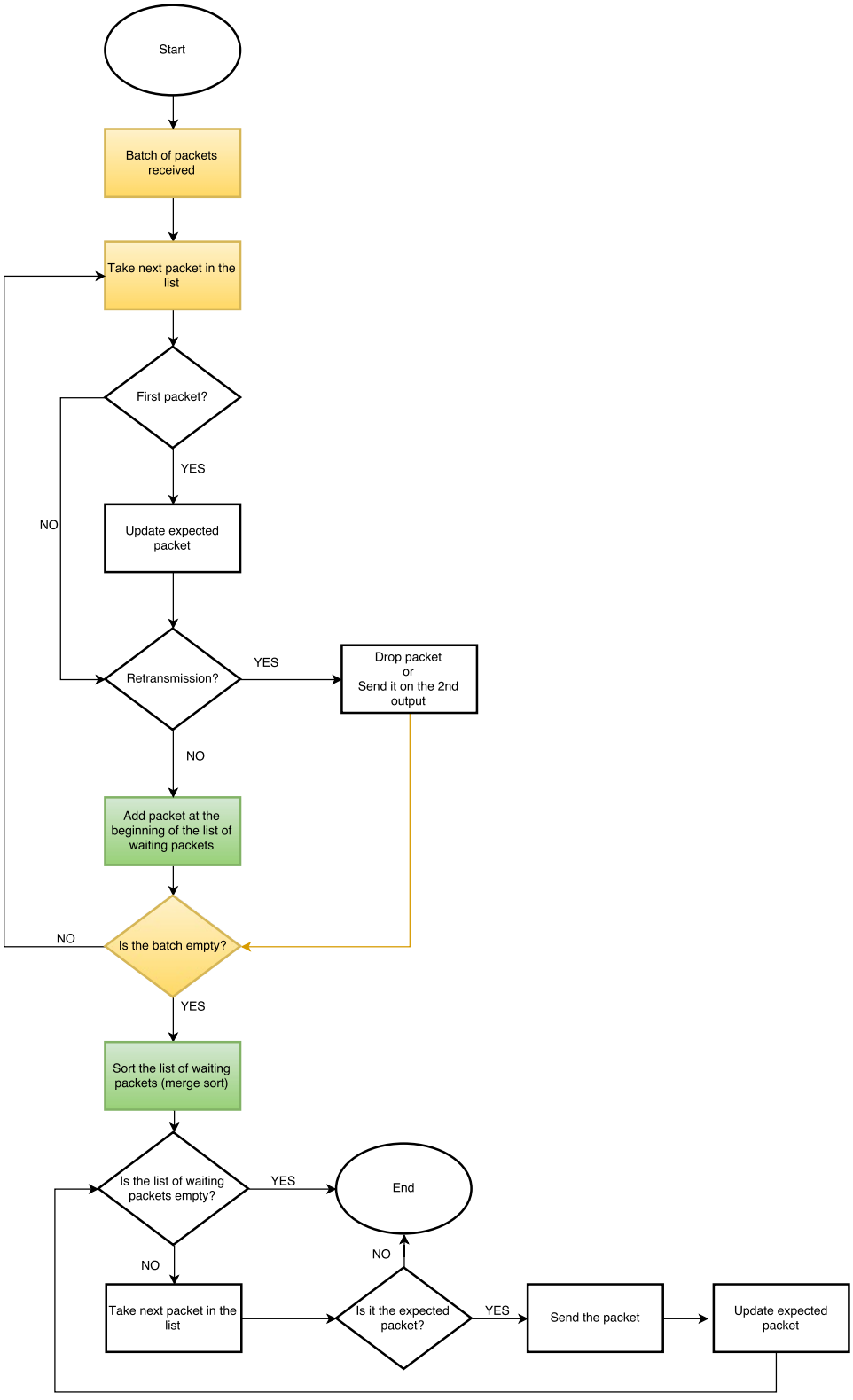

Figure 3.12: Second version of the algorithm that integrates packet batching in TCPReorder. The modified elements are in green.

improvements compared to the naive approach. Note that, as stated in chapter Architecture, the users of the framework can still decide to use the first version of the algorithm by disabling the *MERGESORT* parameter during the configuration of *TCPReorder*, depending on their needs.

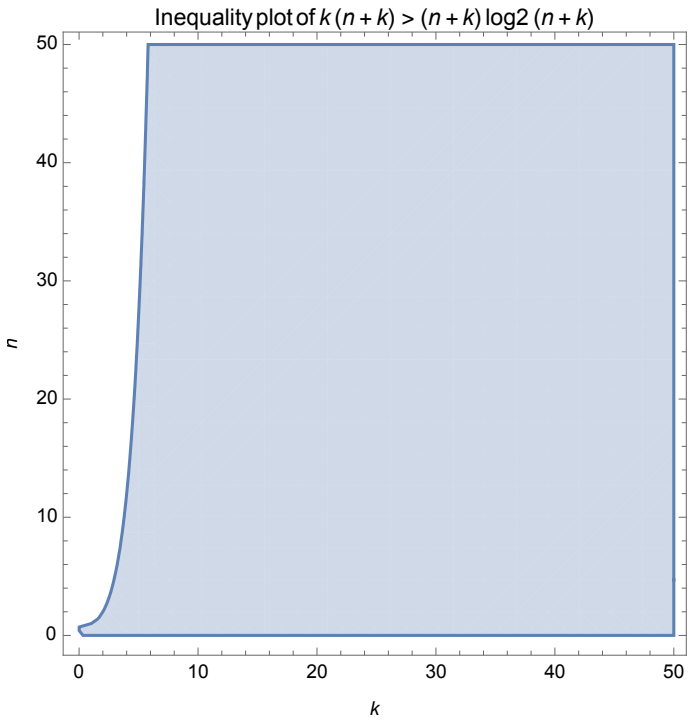

Figure 3.13: Inequality plot of $k * (n + k) > (n + k) * log2(n + k)$. The blue area corresponds to the values for which the merge sort has a theoretical advantage over the naive approach.

3.3.3 TCPRetransmitter

When data have been processed and arrive on the first input of *TCPRetransmitter*, the first step consists in putting them at end of the circular buffer (remember that packets arrive in order if we are in such a situation). Additionally, if we are responsible for these data, meaning that the middlebox has sent an acknowledgement for them, we perform the following operations:

1. We determine if we can send them right now by comparing the amount of data *in flight* with the size of the congestion window and the receiver's window size.
2. If we can send data, we start the retransmission timer, if it was not yet running, and we send them to the destination.

Then, when an acknowledgement is received, we apply this procedure:

1. We check if the acknowledgement is related to data we are responsible for. In such a case, there are two possibilities:

- All the data in flight we are responsible for have been acknowledged, in which case we stop the retransmission timer
- Or, it acknowledged only a part of these data. In this case, we restart the retransmission timer.

2. The acknowledged data are removed from the circular buffer
3. If there are data in the buffer waiting to be transmitted, we determine which amount of them we can send regarding the data *in flight*, the congestion window, and the receiver's window size.

Another interesting point regarding the implementation of *TCPRetransmitter* is how it manages the size of the congestion window. As stated in chapter 2, an implementation of TCP Tahoe has been developed in *TCPRetransmitter* to ensure that it will not congestion the network when the elements in the middlebox stack flush the entire content of a buffer. The following set of rules is therefore used to avoid such an issue and to implement the *Slow Start algorithm*[1]:

1. The initial value of the congestion window (also called hereafter *cwnd*) is set to one MSS
2. The initial value of the slow start threshold, called *ssthrehold* is set to 65535 bytes (as recommended in *'TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms'*[24])
3. Each time an acknowledgement for new data is received, there are two possibilities:
   (a) The size of the congestion window is less than *sshthreshold*: we are in *slow start* mode. In this case, we increase the size of the congestion window by one MSS
   (b) The size of the congestion window is greater than or equal to *sshthreshold*. In this case, we are in *congestion avoidance* mode. Here, the new size of the congestion window becomes: $cwnd = cwnd + MSS \times \frac{MSS}{cwnd}$
4. When the *retransmission timer* fires, the values are updated as follows:
   (a) $ssthreshold = \frac{cwnd}{2}$
   (b) $cwnd = MSS$

Moreover, the *fast retransmit* mechanism of TCP Tahoe is implemented in such a way that when 3 duplicate ACKs are received, we do not wait for the *retransmission timer* to fire and we start to retransmit immediately.

Finally, when packets arrive on its second input, *TCPRetransmitter* maps the sequence number of the packet retransmitted by the sender to determine where to start the retransmission in the modified flow. It also computes the sum of the initial sequence number and the size of the retransmitted TCP payload to determine the end of the retransmission in the original flow. Once again, *TCPRetransmitter* maps this position to know where to stop the retransmission in the modified flow. If it appears that there are indeed data to retransmit, they are obtained from the circular buffer according to what has been computed.

3.3.4 TCPFragmenter

As stated previously, *TCPFragmenter* allows to ensure that packets going out of the middlebox stack do not have a size that exceeds the *MTU* or the *MSS*. It is a built-in element and it was therefore not developed in the context of this work. However, it has been modified in order to be compatible with packet batching so that the entire path used by the middlebox is compatible with this mechanism, which improves the performance of the system.

3.3.5 PathMerger

To remember from which input a packet came from, *PathMerger* uses a hashtable. The key corresponds to the sequence number of the packet and the value stored is the index of the input. When the packet is about to exit the middlebox and we are therefore sure that no further requests will be made for it, *PathMerger* receives a last special request for this packet that will remove the entry in the hashtable in order to save memory.

3.3.6 InsultRemover

To be able to process the content of a web page as a flow and to get rid of the problems related to patterns split across multiple packets, *InsultRemover* uses *FlowBuffer*. When a new packet arrives, it is put in the buffer. Then, for each insult, the method used to remove a pattern in the buffer is called until the insult cannot be found any more. If the feedback obtained for every insult indicates that it is impossible for the last packet to contain the beginning of an insult, the buffer is flushed. Moreover, *InsultRemover* performs a request to determine if the current packet is the last one containing HTTP content and if it is the case, it flushes the buffer. On the other hand, if the current packet is not the last one and at least one feedback indicates that an insult may start at the end of the last packet, the buffer is not flushed right now and we will wait the next one to make a decision.

This element has been developed to test and show the features of the TCP/IP stack provided with this work. It is for instance able to prevent users from accessing web pages containing insults by replacing their content with an error message, as stated in chapter 2. Therefore, the focus has not been made on the efficiency in priority and *InsultRemover* actually achieves terrible performance. It indeed uses the *remove in flow* method of *FlowBuffer* for each insult and loops until it is not in the flow any more. As a result, the time complexity is quartic regarding the length of the flow, the length of the insults, the number of different insults and the number of their occurrences. This behaviour has to be improved by using algorithms such as *Knuth–Morris–Pratt*[11] before the element is used in real conditions.

3.4 Multithreading

To improve the performance of the framework, it has been developed to be compatible with the multithreading system of Click.

In MiddleClick, all the packets coming from the same flow, therefore the same direction of the TCP connection, are managed by the same thread. Thus, it means that a scratchpad corresponding to a flow cannot be accessed by multiple threads at the same time. As a consequence, all the data specific to the scratchpads, not shared among the two sides of the connection, are automatically thread-safe. However, the common TCP structure can be accessed by both threads at the same time and it therefore requires to feature a mutual exclusion mechanism. For this purpose, this common structure also contains a reentrant spinlock that must be acquired before accessing the shared data.

In addition, some elements such as *TCPIn* contain data that are not flow-specific, for instance, the memory pools for the modification lists of the packets. These pools can be accessed by any flow as they are associated to an instance of *TCPIn*, not a flow. This allows to have only one memory pool for all the flows going through a given element. To ensure that those pools are thread-safe, it was decided to make them local to the threads. Consequently, each instance of *TCPIn* has x instances of the memory pool for the modifications lists, where x is the number of threads in use. To achieve this, it uses the *per_thread* template of MiddleClick that allows to have an instance of a variable per thread. Note that the same principle applies for all the memory pools used by the components of this framework.

Finally, the last structure that required some extra precautions is the hashtable described in the section 3.2.3 (*Common TCP Structure*), because it stores the common structure for each connection and they will try to access it concurrently during their *three-way handshake*. To ensure the mutual exclusion on this data structure, a reentrant spinlock is used.

Chapter 4

Results

The first interesting result regarding this work is that it successfully achieves its purpose. It is indeed able to modify the flows on the fly and none of the endpoints notice the modifications performed by the middlebox. Indeed, both of them act as though the flows were not modified. The TCP/IP stack makes all the adjustments needed in order to ensure that the connection still works in spite of the alterations. When using an element such as *InsultRemover*, the client is not aware that some content was removed from the web page it received. Additionally, the server does not know that the client received a modified version of the web page it sent.

Besides, in order to assess the impact of the developed framework on the performance of the TCP connections, it was decided to measure the completion times of fixed-size flows under multiple conditions. To do so, six web pages containing random data were created. Each web page has a fixed content size so that we have pages of *1 kB*, *10 kB*, *50 kB*, *100 kB*, *150 kB*, and *200 kB*. These values have been selected because it appears that a large majority of the TCP flows are short, with a size of about a few kilobytes only[16]. Using the topology presented on figure 2.3, an Apache server and *wget* as a client, we measure the mean completion time of the flows under the following conditions:

- First, the completion times are measured without the middlebox to have a point of comparison
- Next, a middlebox containing only the basic elements related to the developed TCP/IP stack is used. The content of the payload is not modified and the stack only performs the basic operations (determining the parameters of the connection, recomputing the checksums, removing the *SACK-permitted* option, ...). The elements used are the ones depicted on figure 2.4 (namely *IPIn*, *TCPIn*, *TCPOut*, and *IPOut*).
- The third test includes the same elements, but *TCPReorder* is added.
- For the fourth test case, an element removing all the occurrences of the character 'a' in the flow, as well as *TCPRetransmitter*, are added. In this configuration, the stack has to take into account the fact that the content of the flow is modified and to perform the corresponding mappings. The content removed by the middlebox for these tests represents approximately 2% of the initial content of the flows. Note

that the elements of the HTTP stack are not used here as they would buffer the packets to determine the final *Content-Length*. This configuration corresponds to the one a middlebox could use in real conditions, provided that it works directly on the TCP payloads.

- The last test uses the configuration shown on figure 2.10, namely the TCP/IP stack elements, including *TCPReorder* and *TCPRetransmitter*, along with the HTTP stack elements. Here the HTTP payload is not modified but the HTTP stack performs the usual modifications on the headers so that the flows are nonetheless modified. In this configuration, the stack becomes responsible for the content of the flows when it receives it as the HTTP stack buffers the packets.

The results are listed in table 4.1 and the corresponding chart is depicted on figure 4.1. Note that the mean completion times have been computed on 30 samples for each case. The first thing we can notice is that, in every case, using the middlebox increases the completion time. Therefore, it cannot be used as a mean to improve the performance. However, we also see that when we use the TCP/IP stack alone, or even with the reordering component, the overhead is rather limited and does not seem to increase according to the size of the flow. This is in itself a good result as in such a configuration, an element of the middlebox stack can read the content of the flow and make decisions such as closing the connection if some criteria are met. Thus, this result corresponds to real applications.

	Without middlebox	**TCP/IP stack**	**TCP/IP stack reordering**	**Remove all 'a's**	**HTTP stack**
1 kB	1.6 ms	1.7 ms	1.75 ms	1.8 ms	1.9 ms
10 kB	2.7 ms	3 ms	3.2 ms	3.6 ms	3.9 ms
50 kB	6 ms	6.3 ms	6.8 ms	7.4 ms	8.2 ms
100 kB	12.5 ms	12.8 ms	13.4 ms	15 ms	15.5 ms
150 kB	18 ms	18.8 ms	19.3 ms	22 ms	23 ms
200 kB	23.9 ms	24.2 ms	24.6 ms	28 ms	30 ms

Table 4.1: Impact of the elements of the framework on the mean completion time of the flows. The rows represent the flow sizes (in kilobytes) and the columns the operations performed on the flows.

On the other hand, the tests featuring the HTTP stack induced the greatest overhead. Here, the packets are modified by the stack to ensure that the HTTP headers comply with what the HTTP stack expects. Moreover, the whole content of the TCP payload is buffered in *HTTPOut* so that when the entire page has been processed, it is able to compute the new *Content-Length*. In such a case, the TCP/IP stack acts as an intermediary between the two endpoints: it acknowledges the data it receives and ensures that the destination correctly receives them. The fact that, in this configuration, the middlebox plays the role of a complete actor explains why it provides the biggest overhead.

Finally, the most interesting case is the one in which we modify the content of the flow seamlessly for both endpoints, but without buffering the packets like the HTTP stack

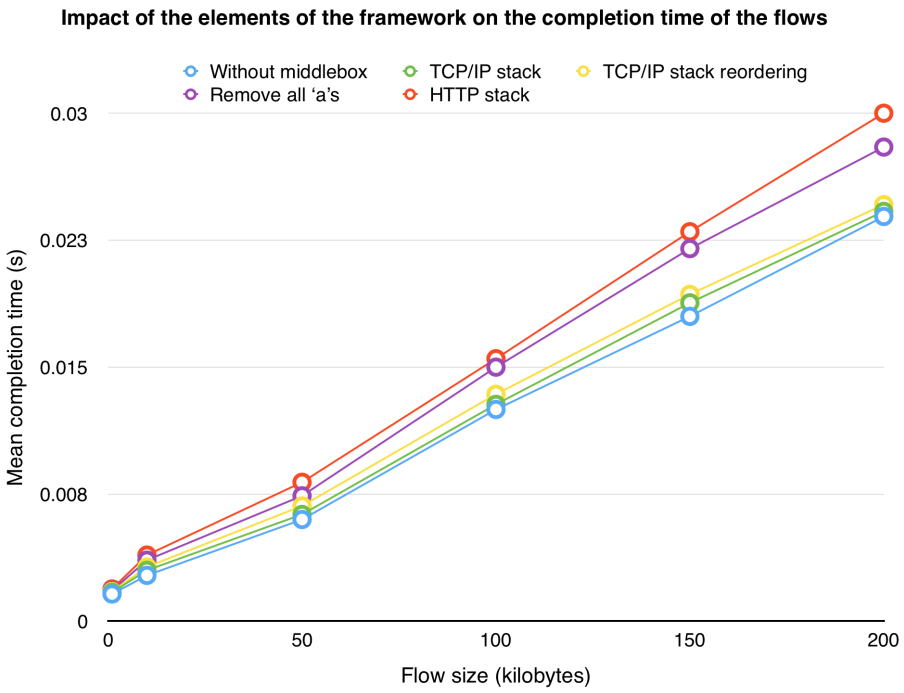

Figure 4.1: Chart depicting the impact of the framework elements on the completion time of the flows.

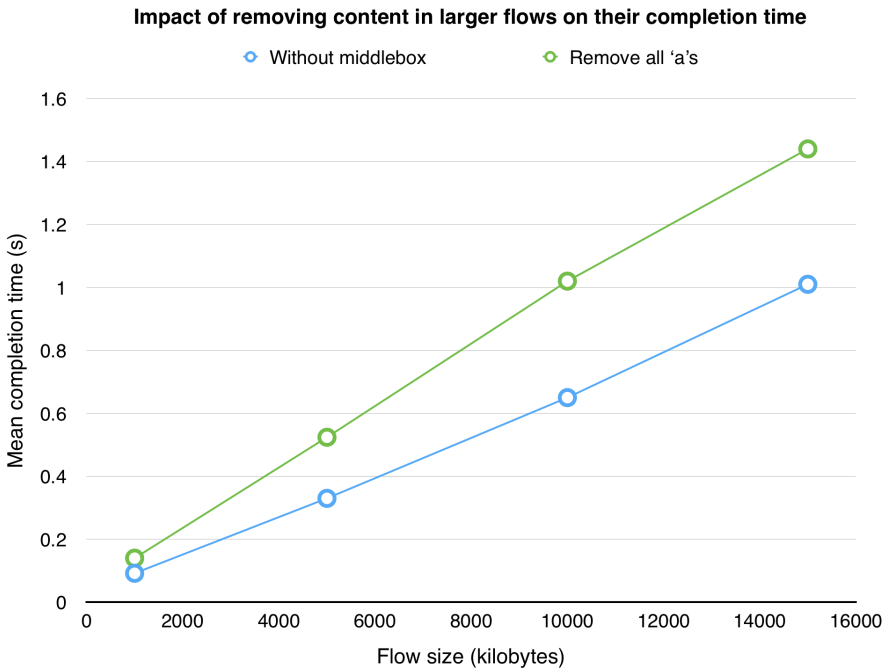

Figure 4.2: Chart depicting the impact of removing content in larger flows on their completion time.

does. It indeed provides the most significant results for the majority of the applications as HTTP is an exception regarding the fact that it must first buffer the entire content before being able to finish processing it. In the configuration of this test case, the entire content is analysed and approximately 2% is modified by the middlebox. Regarding the times listed in table 4.1, the average overhead induced on the completion time is 21%, with respect to the situation in which the middlebox is not used. This seems rather high. However, in the topology used for the tests, the round-trip time between the two endpoints is very small and the percentage of the overhead induced by the middlebox is therefore higher than in a topology where the network delay contributes for a bigger part of the completion time. Moreover, the overhead induced by this configuration seems to grow linearly with respect to the flow sizes. It is confirmed by the figure 4.2 which depicts the impact of such a configuration on larger flows. This is the expected result. Indeed, remember that in this test configuration, we remove a fixed percentage of the flow (approximately 2%), therefore, the amount of data removed by the middlebox also grows linearly with respect to the size of the flows.

These results show that the modularity of the implemented TCP/IP stack allows the users to select only the features they need for their application, avoiding the overhead induced by other elements. However, there is clearly still room for improvement, in particular in the case where the HTTP stack is used. Some hints about the improvements that could be made will be given in the next chapter.

Chapter 5

Future work

Since this work and MiddleClick were developed in parallel, the presented framework was in fact developed within FastClick, planning to eventually integrate it into MiddleClick, when both would be ready. However, at the end of this project, it appeared that Middleclick was not yet stable enough and that it would not be possible to make the integration in time for the submission of this work. Therefore, as it currently runs into FastClick, which does not provide the expected flow management system, it can only be used with one flow at a time since every packet received is considered to be part of the same flow. Note that this limitation will automatically disappear when the integration is done. The first perspective of improvement would therefore be to finalise the integration into Middleclick when it becomes possible.

Moreover, there are several points that can be explored to improve the framework developed in the context of this work:

- The TCP stack currently does not support the *SACK-permitted* option. As described previously, it automatically disables it for connections going through the middlebox so that it is not a real a problem, but implementing this option would improve the performance of the system by avoiding unneeded retransmissions.
- The HTTP stack provided with this work in order to be able to test elements such as *InsultRemover* is rather basic and could be improved to take into account some of the new mechanisms of HTTP 2.
- It could be possible to improve the performance of the TCP stack by using techniques such as gathering the acknowledgements together, in particular when an element puts the packets in a buffer. This would limit the amount of data sent on the network.
- It is possible to improve the multithreading implementation by using finer-grained mutual exclusions or *lock-free data structures*, especially on the *common TCP structure* shared by both directions of the connection.

Moreover, TCP is a very complex protocol that features a large number of mechanisms. It is therefore really difficult to ensure that everything works as expected in all situations, and that every corner case has been taken into account. This is especially

true in the context of this work in which we have to manipulate the flows on the fly, being ignored by the endpoints which will therefore not comply with what we expect. Some network protagonists may indeed have exotic behaviours and thus be challenging to predict. In addition, some data structures and the corresponding algorithms had to be developed especially in the context of this work, such as the mapping mechanism performed by *ByteStreamMaintainer*. As a consequence, it is very difficult to assess the stability of such a large piece of work during its development. Ideally, the framework should be tested extensively, in real life conditions, to ensure that all the corner cases have been taken into account and to be able to adapt some mechanisms according to the results.

Finally, it would be interesting to assess the performance of the framework once it has been integrated to Middleclick. This would allow to determine its efficiency when a large number of flows are processed at the same time by the middlebox and to determine its impact on the global throughput.

Chapter 6

Conclusion

In this thesis, we started by introducing the concept of middlebox, providing examples and classifying them according to the layers they impact. We then explained why developing a lightweight TCP framework specifically designed for middleboxes is important regarding their peculiarities and the fact that they must have an impact on the performance of the network which is as low as possible.

For this purpose, we presented the framework developed in the context of this thesis which consists in a set of elements that can be used in Middleclick, the enhanced version of the Click modular router aimed at providing it flow management mechanisms. These elements can be used by developers to handle the network management part of their middlebox so that they can focus on implementing the higher level functionalities. As this framework has been developed specifically to take into account the specificities of middleboxes, the implemented TCP/IP stack allows to modify flows on the fly, seamlessly for the two endpoints of the connection.

We saw that a middlebox becomes responsible for the data it acknowledges, which occurs when packets are buffered to delay their processing. As a consequence, an implementation of TCP Tahoe was developed to ensure that the stack complies with the congestion and flow control mechanisms of TCP. We also described how the TCP/IP stack manages the retransmissions of the data it is responsible for, according to the standards of TCP. The various data structures developed to be able to perform the mapping between the original flow and the one resulting of the modifications made by the middlebox have been described, trying to justify the choices we made by providing an analysis of their time complexities.

The results showed that it is possible to use this framework to implement a middlebox that performs deep packet inspection with a limited and constant overhead. However, a larger additional cost is induced when the middlebox modifies the content of the flows. Even though this overhead seems to be linear according to the size of the content modified, there is still room for improvements regarding the performance of the framework.

We finally proposed some improvements that could be made in the future to extend the present work. Although the framework achieves its goals, the TCP protocol is vast and there are always new features that can be added or handled in a better way.

Appendix A

Complete example of configuration

The listing A.1 shows an example of a complete Click configuration that can be used to create a middlebox with the elements provided in this framework.

```
// Left side of the connection
define($leftMac 08:00:27:db:83:16)
ipINLeft :: IPIn();
// 0 is the id of the flow direction
tcpINLeft :: TCPIn(0, tcpOUTLeft, tcpINRight);
httpINLeft :: HTTPIn();
ipOUTLeft :: IPOut();
tcpOUTLeft :: TCPOut();
httpOUTLeft :: HTTPOut();
// 0 is the id of the flow direction
reorderLeft :: TCPReorder(0);
retransmitterLeft :: TCPRetransmitter();

// Right side of the connection
define($rightMac 08:00:27:27:b5:9a)
ipINRight :: IPIn();
// 1 is the id of the flow direction
tcpINRight :: TCPIn(1, tcpOUTRight, tcpINLeft);
httpINRight :: HTTPIn();
ipOUTRight :: IPOut();
tcpOUTRight :: TCPOut();
httpOUTRight :: HTTPOut();
// 1 is the id of the flow direction
reorderRight :: TCPReorder(1);
retransmitterRight :: TCPRetransmitter();

// Left path
inLeft:: FromNetmapDevice(netmap:eth0, PROMISC true) -> Strip(14)
    -> chIPLeft :: CheckIPHeader()[0]
    -> chTCPLeft :: IPClassifier(ip proto tcp, -)[0]
    -> ipINLeft -> reorderLeft[0] -> tcpINLeft
    -> httpINLeft -> httpOUTLeft -> tcpOUTLeft
    -> [0]retransmitterLeft -> ipOUTLeft
    -> TCPMarkMSS(0, 24, OFFSET 40)
    -> TCPFragmenter(MTU 1500, MTU_ANNO 24)
    -> EtherEncap(0x800, $leftMac, $rightMac)
    -> outLeft :: ToNetmapDevice(netmap:eth1);
```

```
38
39 // Right path
40 inRight:: FromNetmapDevice(netmap:eth1, PROMISC true) -> Strip(14)
41     -> chIPRight :: CheckIPHeader()[0]
42     -> chTCPRight :: IPClassifier(ip proto tcp, -)[0]
43     -> ipINRight -> reorderRight[0] -> tcpINRight
44     -> httpINRight -> InsultRemover()
45     -> httpOUTRight -> tcpOUTRight[0]
46     -> [0]retransmitterRight -> ipOUTRight
47     -> TCPMarkMSS(1, 24, OFFSET 40)
48     -> TCPFragmenter(MTU 1500, MTU_ANNO 24)
49     -> EtherEncap(0x800, $rightMac, $leftMac)
50     -> outRight:: ToNetmapDevice(netmap:eth0);
51
52 // Retransmissions detected by TCPReorder go to TCPRetransmitter
53 reorderLeft[1] -> [1]retransmitterLeft;
54 reorderRight[1] -> [1]retransmitterRight;
55
56 // Left path for generated packets that go back to the source
57 etherLeft :: EtherEncap(0x800, $rightMac, $leftMac);
58 tcpOUTLeft[1] -> etherLeft -> ToNetmapDevice(netmap:eth0);
59
60 // Right path for generated packets that go back to the source
61 etherRight :: EtherEncap(0x800, $leftMac, $rightMac);
62 tcpOUTRight[1] -> etherRight -> ToNetmapDevice(netmap:eth1);
63
64 // Non-TCP packets bypass the middlebox
65 bpLeft :: Unstrip(14) -> outLeft
66 bpRight :: Unstrip(14) -> outRight;
67
68 chIPLeft[1] -> bpLeft;
69 chIPRight[1] -> bpRight;
70 chTCPLeft[1] -> bpLeft;
71 chTCPRight[1] -> bpRight;
```

Listing A.1: Complete example of Click configuration that can be used to create a middlebox

List of Figures

List of Tables

Listings

Bibliography

[1] M. Allman, Paxson V. and Stevens W. *TCP Congestion Control*. RFC 2581. IETF, Apr. 1999, pp. 1–14. URL: https://tools.ietf.org/html/rfc2581.

[2] Tom Barbette, Cyril Soldani and Laurent Mathy. 'Fast Userspace Packet Processing'. In: *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ANCS '15. Oakland, California, USA: IEEE Computer Society, 2015, pp. 5–16. ISBN: 978-1-4673-6632-8. URL: http://dl.acm.org/citation.cfm?id=2772722.2772727.

[3] B. Carpenter. *Architectural Principles of the Internet*. RFC 1958. RFC Editor, June 1996, pp. 1–8. URL: http://www.rfc-editor.org/rfc/rfc1958.txt.

[4] B. Carpenter and S. Brim. *Middleboxes: Taxonomy and Issues*. RFC 3234. RFC Editor, Feb. 2002, pp. 1–27. URL: http://www.rfc-editor.org/rfc/rfc3234.txt.

[5] Benoit Donnet. 'Introduction to Computer Security – Chapter 2: Proxy'. 2015. URL: http://www.montefiore.ulg.ac.be/~bdonnet/info0045/files/slides/Network_Chap2.pdf (visited on 28/03/2016).

[6] K. Egevang and P. Francis. *The IP Network Address Translator (NAT)*. RFC 1631. RFC Editor, May 1994, pp. 1–10. URL: http://www.rfc-editor.org/rfc/rfc1631.txt.

[7] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. RFC Editor, June 1999, pp. 1–176. URL: http://www.rfc-editor.org/rfc/rfc2616.txt.

[8] Internet Engineering Task Force. *Requirements for Internet Hosts – Communication Layers*. RFC 1122. IETF, Oct. 1989, pp. 1–112. URL: https://www.ietf.org/rfc/rfc1122.txt.

[9] Van Jacobson. 'Congestion avoidance and control'. In: *ACM SIGCOMM computer communication review*. Vol. 18. 4. ACM. 1988, pp. 314–329.

[10] J. Kempf and R. Austein. *The Rise of the Middle and the Future of End-to-End: Reflections on the Evolution of the Internet Architecture*. RFC 3724. RFC Editor, Mar. 2004, pp. 1–14. URL: http://www.rfc-editor.org/rfc/rfc3724.txt.

[11] Donald E Knuth, James H Morris Jr and Vaughan R Pratt. 'Fast pattern matching in strings'. In: *SIAM journal on computing* 6.2 (1977), pp. 323–350.

[12] Eddie Kohler et al. 'Click: Example Configurations'. 2008. URL: http://www.read.cs.ucla.edu/click/examples (visited on 29/07/2016).

[13] Eddie Kohler et al. 'Vector< T > Class Template Reference'. 2011. URL: http://www.read.cs.ucla.edu/click/doxygen/class_vector.html (visited on 05/07/2016).

[14] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti and M. Frans Kaashoek. 'The Click Modular Router'. In: *ACM Trans. Comput. Syst.* 18.3 (Aug. 2000), pp. 263–297. ISSN: 0734-2071. DOI: 10.1145/354871.354874. URL: http://doi.acm.org/10.1145/354871.354874.

[15] Emin Martinian. 'Red-Black Tree C Code'. 2005. URL: http://web.mit.edu/~emin/Desktop/ref_to_emin/www.old/source_code/red_black_tree/index.html (visited on 15/04/2016).

[16] Marco Mellia, Hui Zhang and Ion Stoica. 'TCP model for short lived flows'. In: *IEEE Communications Letters* 6.2 (2002), pp. 85–87.

[17] V. Paxson and M. Allman. *Computing TCP's Retransmission Timer*. RFC 2988. IETF, Nov. 2000, pp. 1–8. URL: https://www.ietf.org/rfc/rfc2988.txt.

[18] Luigi Rizzo. 'netmap: A Novel Framework for Fast Packet I/O'. In: *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012, pp. 101–112. ISBN: 978-931971-93-5. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/rizzo.

[19] Luigi Rizzo. 'Revisiting Network I/O APIs: The Netmap Framework'. In: *Queue* 10.1 (Jan. 2012), 30:30–30:39. ISSN: 1542-7730. DOI: 10.1145/2090147.2103536. URL: http://doi.acm.org/10.1145/2090147.2103536.

[20] J. H. Saltzer, D. P. Reed and D. D. Clark. 'End-to-end Arguments in System Design'. In: *ACM Trans. Comput. Syst.* 2.4 (Nov. 1984), pp. 277–288. ISSN: 0734-2071. DOI: 10.1145/357401.357402. URL: http://doi.acm.org/10.1145/357401.357402.

[21] P. Srisuresh and M. Holdrege. *IP Network Address Translator (NAT) Terminology and Considerations*. RFC 2663. RFC Editor, Aug. 1999, pp. 1–30. URL: http://www.rfc-editor.org/rfc/rfc2663.txt.

[22] Simon Tatham. 'Mergesort For Linked Lists'. 2001. URL: http://www.chiark.greenend.org.uk/~sgtatham/algorithms/listsort.html (visited on 20/07/2016).

[23] *Union declaration - cppreference.com*. URL: http://en.cppreference.com/w/cpp/language/union (visited on 05/08/2016).

[24] Stevens W. *TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms*. RFC 2001. IETF, Jan. 1997, pp. 1–6. URL: https://tools.ietf.org/html/rfc2001.

[25] Ben Wagner. *Deep Packet Inspection and Internet Censorship: International Convergence on an 'Integrated Technology of Control'*. 23rd June 2009. URL: http://ssrn.com/abstract=2621410.

[26] Lixia Zhang. 'A retrospective view of network address translation'. In: *IEEE Network* 22.5 (2008), pp. 8–12.