Lightweight Middlebox TCP

Developing a fast and easy-to-use framework to create middleboxes

Romain Gaillard

Supervisors: Prof. Laurent Mathy, Tom Barbette

Master thesis submitted for the degree of
Msc in Computer Science

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

Acknowledgments

Here will be the acknowledgements.

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 behaviors, for instance Network Address Translation (NAT), traffic filtering, deep packet inspection (DPI) or security strengthening. Often, a middlebox implements multiple of these features at the same time, being therefore a complex network actor that can have a great 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 network layer and above [3], 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 it to the destination the client wanted to reach. Thanks to the encapsulation made by the middleboxes, the VPN specificities are invisible for the end points. 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 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 connecting 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.[5][12][14]

Proxy

A proxy server is an intermediary between a client and the server he 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 is 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[4].

There exists 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 ones that leave *the application layer unmodified* and simply transfer its content.

Intrusion Detection System

An Intrusion Detection System (IDS) is a device that 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. Rather, it 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 textitall 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 networking device 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 set 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 take 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 one by hiding

it. Such a firewall thus does not only rely on the headers to take a decision and allows more fine-grained criteria.

Domain Name System liar

The Domain Name System (DNS) is a service that allows to map a domain name into an IP address, among other pieces of information. When a user wants to reach, say *www.google.fr*, the first step is to resolve the domain name in order to determine the associated IP address that will be used to establish the connection.

Although a lying DNS resolution does generally not involve middleboxes, but simply a DNS server that will process the user's request and reply with deliberately false information, it is possible to achieve the same goal by using a middlebox that will intercept the reply and modify it *on the fly*. If the middlebox is located on a link that the traffic coming from the user will necessarily use, it allows to ensure that the reply will be modified regardless of which DNS the user is trying to contact. For this purpose, the *application layer* has to be modified.

1.1.3 End-to-end principle

The **end-to-end** principle is described by J. H. Saltzer et al.[11] as the argument supporting that the intelligence of the network should be implemented at its end points. They state that implementing the various functions on the internal nodes can be redundant and not efficient. Indeed, in many cases, the end points will still have to implement the functions, which leads to redundancy. This results in higher costs and lower performances.

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 end points, 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."[2]

By definition, middleboxes contravene the end-to-end principle as they are providing functions at the middle of the network instead of leaving the responsibility to the end points of implementing them. The implications and the interpretations of the end-to-end principle have evolved over the years, as described by Kempf et al.[7] 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 end points) analysing the content of the packets going through them, 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 anymore.

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 discard or the connection closed without notifying any of the protagonists[13]. 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. 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 their quality as well. 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 performances of networks. It is thus crucial to ensure that middleboxes do not become network bottlenecks. Furthermore, as they are generally more tools than actors that produce useful content in a connection, it is reasonable to consider that they should not significantly decrease the performances of the connection. It may even be conceivable to use middleboxes to improve overall performances in some cases.

1.3 Goals

The goal of this work is to develop a fast and easy-to-use 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 performances of the network management system, as well as the specificities of the lower layer protocols.

Using recent tools such as Netmap[9], it 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 performances as low as possible and to cope with gigabits of traffic. 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 this 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 the Click Modular Router[8], 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-know 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 large number of elements, options and extensions that makes 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

A middlebox framework

2.1 Netmap

Netmap is a framework aimed at providing the best performances for fast packet I/O. It has been developed by *Luigi Rizzo* as a result of the observation that general purpose OSes offer a network API that was designed to be a 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[10]. 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 anymore.

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 a negative impact on the performances 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.

In this section, we analyse the various improvements that netmap offers in order to achieve high rate packet I/O.
TODO
[1]

2.2 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.

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 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 and provides 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 elements 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 to replace the *Discard* component, which is a click built-in element that drops all the traffic, 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.

TODO

[8]

2.3 Middleclick

TODO

2.4 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 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 is arose when creating the *TCPReorder* element that will be described later on. 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, some of the network specificities can be tricky to deal with and considering difficulties one at a time helped to stay focus on them. Moreover, it allowed to define and reach multiple milestones during the development and test the state of the framework on real traffic.

2.5 Data structures

To achieve the goal of developing a fast and lightweight middlebox framework, it is mandatory to carefully select and implement the data structures that will be at the heart of the system. In order to cope with gigabits of traffic per second, 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.

2.5.1 Memory pool

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 anymore. Therefore, an efficient implementation of the memory management is of the highest importance. In order to be able to cope with gigabits of traffic per second, 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 degradation of the performances and thus have an impact on the traffic.

To be able to fulfil the need of a fast memory allocation mechanism, a common approach is to use *memory pools*. This data structure actually provides a space-time trade-off. 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 anymore, instead of freeing it, the memory pool just add it back in the list of free memory chunks so that it is available for a further request. If the need for memory is greater than the expected one and no more memory chunks are available to fulfil a request, the memory pool exceptionally performs a new memory allocation so that the pool will grow.

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 performances are 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) 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 in front of the list.

To minimize the memory consumption, each node of the list is represented as an *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.

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 thus there will be no need to allocate memory. In this case, the functions only manipulates pointers to return the current head of the list and set the new one. Moreover, putting an element back in the list because it is not use anymore is also $O(1)$. Here, the structure just adds the elements it gets back in front of the list, which is done by modifying two pointers.

2.5.2 Red-black tree

2.5.3 Byte stream maintainer

2.6 Framework elements

In this section, we describe the framework from the point of view of the click elements it provides, describing their functioning, peculiarities and the data structures and algorithms they use. The

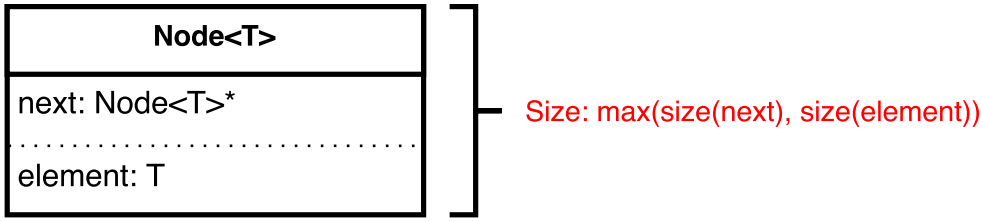

Figure 2.1: Representation of a node in the memory pool. A node is the union between a pointer to the next node and the element it stores, meaning that its size is equal to its biggest member.

section thus provides an exhaustive list of the elements a developer can use to implement a middlebox. Some of them are directly usable in a click configuration while some others are abstract and only used to develop other components.

2.6.1 TCPReorder

A critical component regarding the network management of middleboxes is the TCP reordering mechanism. Reordering packets is indeed necessary in many applications, depending on their needs. We will enumerate 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 an insertion, and what is assumed to be a retransmission in the case of a deletion. 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 a addition or a deletion of bytes if 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.

   The figure 2.2 depicts an example of data insertion and how the receiver perceives it. In this second case (*receiver's view (b)*), we can see that the receiver perceives the first 100 bytes of the second packet as overlapping with the end of the first packet because the middlebox did not update the sequence number of the second packet. To avoid this case, the packet $x+1$ must be transmitted by the middlebox after the packet x has been processed, which actually applies not only to the packet $x+1$ but to every packet labelled with a packet number $> x$.

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 come just before in the stream order is not a straightforward task that 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 2.3.

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*)[6]. Ensuring that the packets arrive in the right order is thus very important because in this way, the application can check the first packet 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 take 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 performances of the whole system.

Even though we have seen that a 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 doing it in the *TCPIn* element, 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 components related to the middlebox, if his router includes other features. Moreover, the user may decide to process the packets unordered to increase the efficiency of the middlebox if he knows that the application will not meet one of the three criteria above.

One interesting thing to notice regarding the implementation of the *TCPReorder* element is that making him compatible with packet batching, after implementing a first version that received one packet at a time, was not as straightforward as we might think. Indeed, the simple approach that consists in taking an element that works with one packet at a time and loop to process all the packets 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 recipient. The list can contain gaps, meaning that packets with a sequence number smaller than some of the list are yet to be received. The element 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 sequence number of the packet is checked to determine if it has already been transmitted before. In case of a retransmission, the packet is dropped. 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.

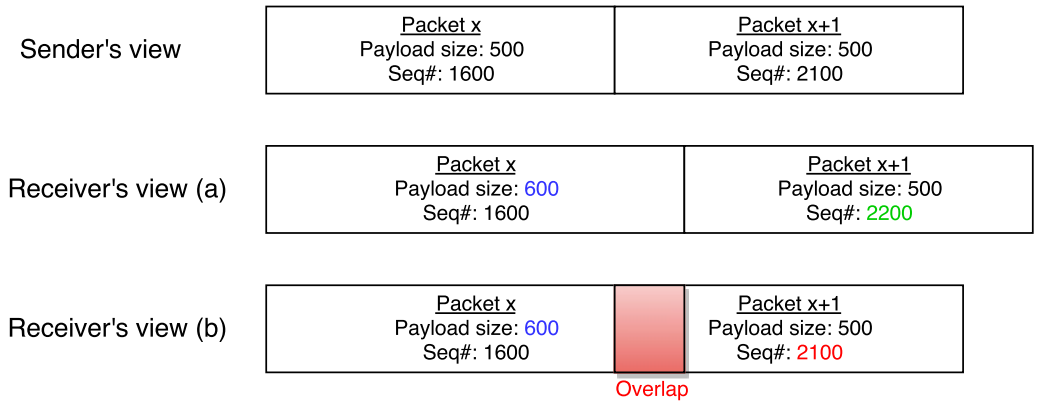

Figure 2.2: Points of view of the sender and the receiver when the middlebox adds 100 bytes of data in the packet x. The view **(a)** corresponds to the case where the middlebox is able to update the sequence number of the next packet. The view **(b)** corresponds to the case where the middlebox did not update the sequence number of the next packet.

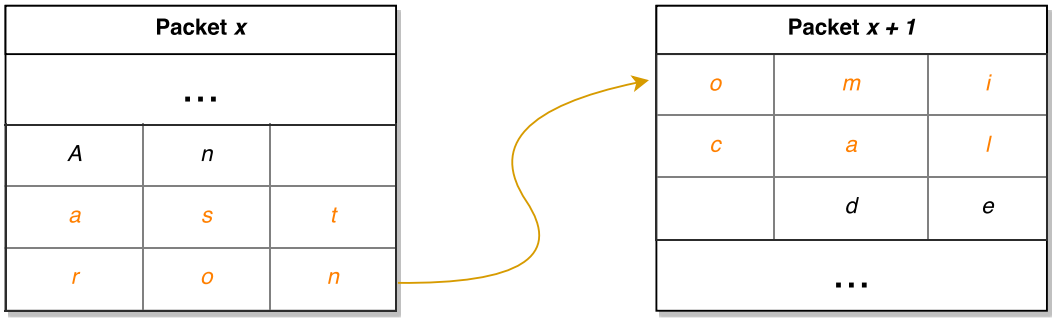

Figure 2.3: Example of a pattern split over two packets. Here, the middlebox searches for the word *astronomical* that begins at the end of the packet x and ends at the beginning of the packet $x + 1$. Only the payloads are represented.

2. 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 *SYN* 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.

3. Next, the packet is added in the list of waiting packets, taking care to keep it 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 2.4.

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 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 always the most efficient and thus requires to perform some analyses in order to determine when it is suited or not.

If we look closer to 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 an element 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 2.6. 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 were 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 list. 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 and the packets are sent to the destination, meaning that when the next batch will be processed, the list of waiting

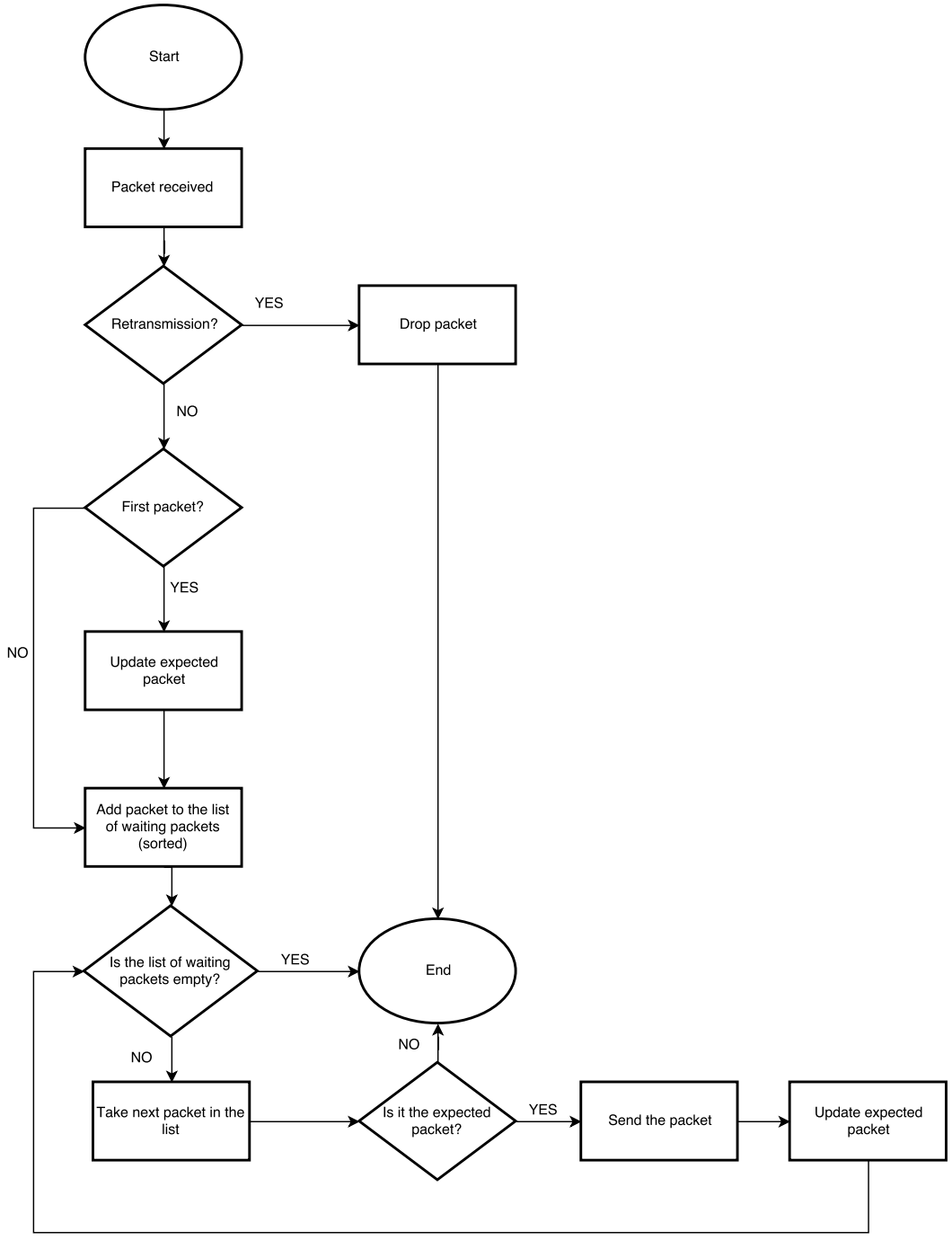

Figure 2.4: Algorithm used by the first implementation of TCPReorder, processing one packet at a time instead of batches.

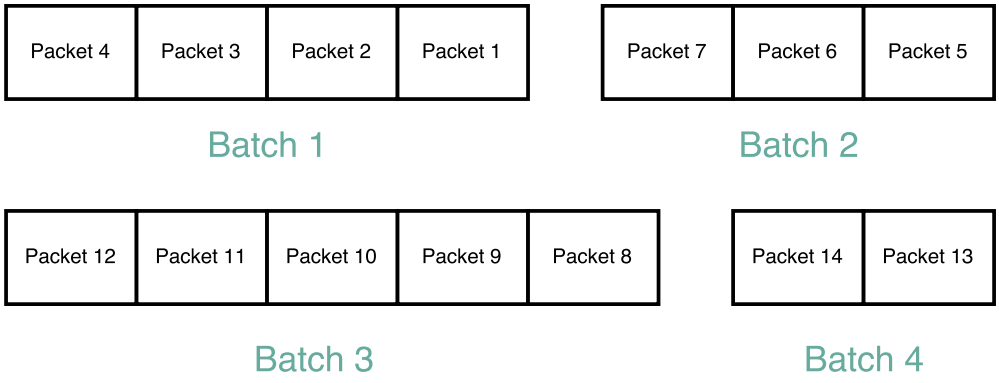

Figure 2.5: Example of situation that leads to the best case for the first version of the algorithm that includes packet batching in TCPReorder

packets will be empty. In this case, since the packets in the batch list are in order and added one after the other in the list, each one added at the end of the current list, the time complexity is $O(k^2)$, which is not the worst case. 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 get rid of the advantage given by the fact that packets arrive in order: the waiting list is flushed after each batch. This 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 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 the 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 have also seen that, 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 for each packet of the batch. We can improve this situation if the 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 arrive globally in order so 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 situation that leads to this case is depicted in figure 2.5

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 it to consider an alternative to this approach that involves a better sorting algorithm and determine if we can improve the performances.

A good choice when it comes to sorting linked list, a reasonable choice is to perform a *merge sort*. This algorithm works as follows: TODO
This algorithm is depicted on figure 2.7.

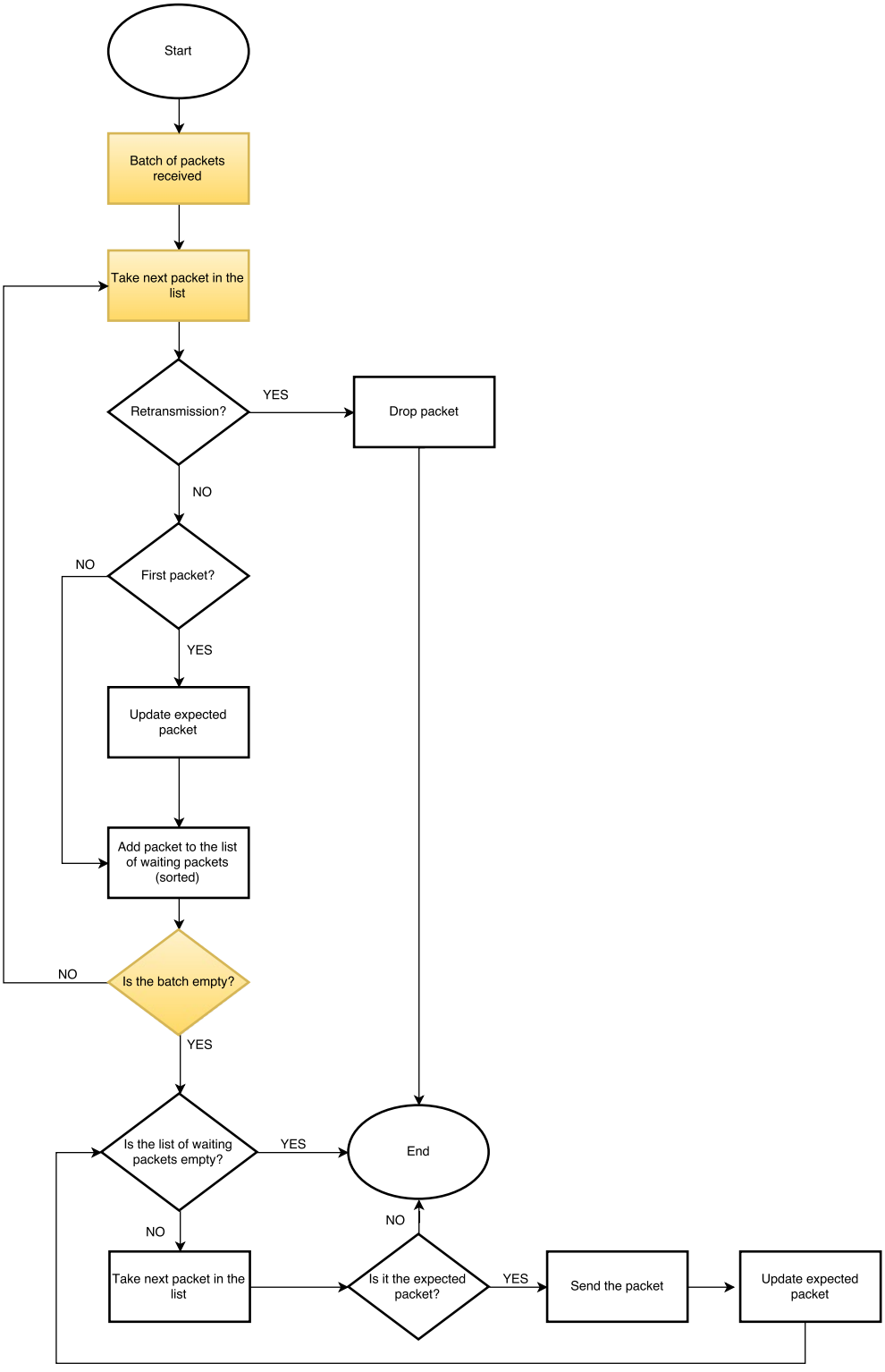

Figure 2.6: First version of the algorithm that integrates packet batching in TCPReorder. The new elements are in yellow.

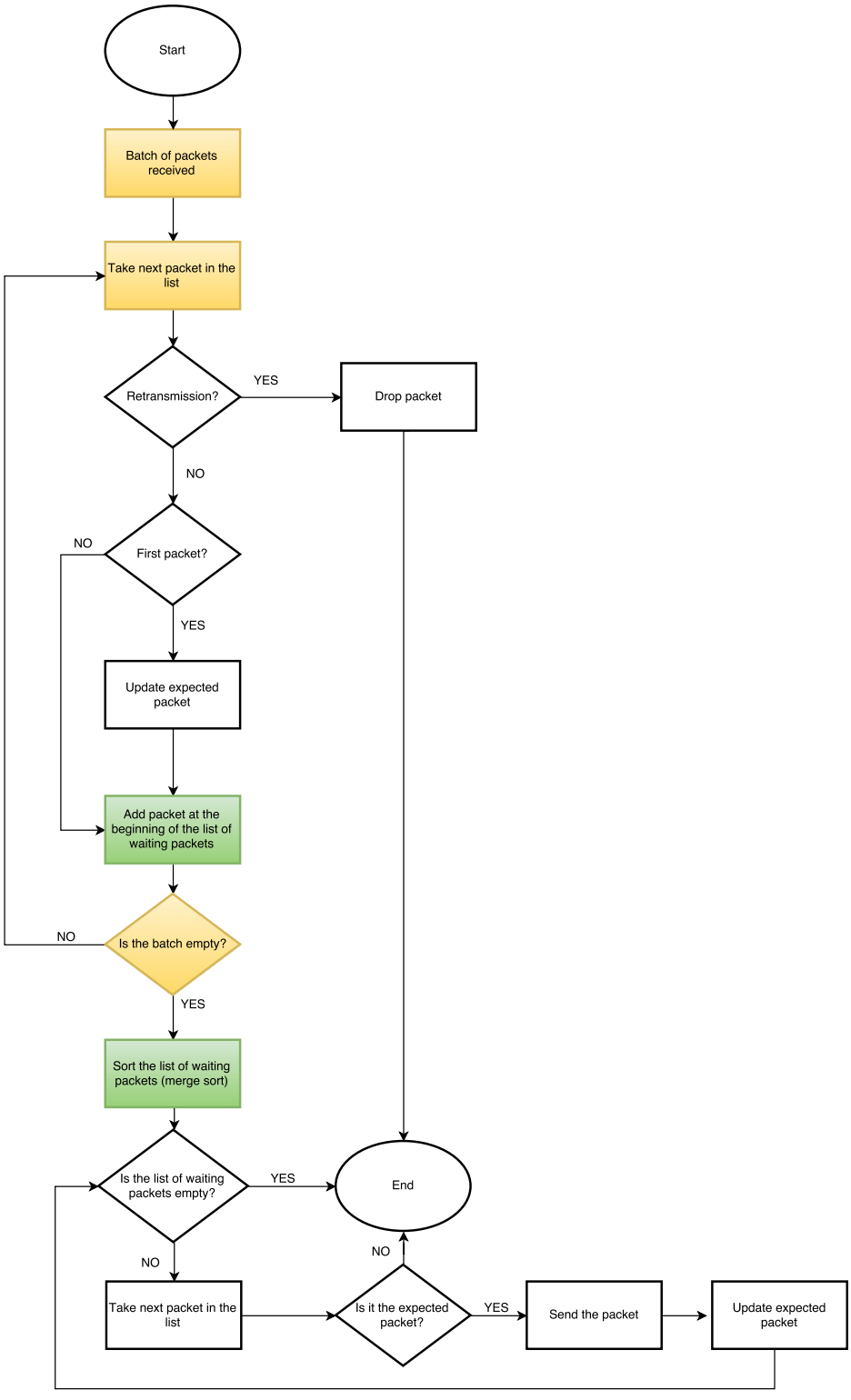

Figure 2.7: Second version of the algorithm that integrates packet batching in TCPReorder. The modified elements are in green.

Chapter 3

Results

Chapter 4

Conclusion

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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 03/28/2016).

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

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