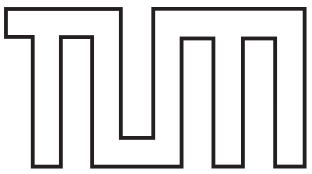


Technische Universität München
Chair of Media Technology
Prof. Dr.-Ing. Eckehard Steinbach

Engineer Practice

Priority Based Buffer Flushing With the Click Modular Software Router

Author:	Arturo Buitrago Méndez
Matriculation Number:	03646899
Address:	Türkenstr. 58a
	80799 München
Advisor:	Christoph Bachhuber
Begin:	23.05.2016
End:	31.08.2016

With my signature below, I assert that the work in this thesis has been composed by myself independently and no source materials or aids other than those mentioned in the thesis have been used.

München, August 26, 2016 _____

Place, Date Signature

München, August 26, 2016 _____

Place, Date Signature

Contents

Chapter 1

Introduction

The aim of this Engineer Practice is the implementation of a buffer flusher mechanism. Its main goal is to give certain video frames, called *"keyframes"*, a higher transmission priority, by *"flushing"* the buffer containing the outgoing packets of any belonging to non-keyframes, thus allowing the more important packets to be transmitted immediately. This would ideally result in a lower end-to-end transmission time in real systems. The concept for this mechanism was born out of research at the Chair for Media Technology at the Technical University of Munich, with collaboration from Martin Reisslein at Arizona State University. The router devised worked as intended at the user level but kernel integration proved to be too laborious for the scope of this work.

The buffer flusher itself was implemented through the Click Modular Router, a virtual router creation software with Linux kernel integration. The corresponding Click Element files are included in the repository of the project, as well as the scripts used to create and send the packages. Tests were then run in a variety of different transmission conditions in order to gauge the effectiveness of the application.

The rest of this report includes a comment on why Click was chosen and a more in-depth explanation of the software and the buffer flusher specifically. It is then followed by a discussion of the prospect of kernel integration. The testing environment is introduced and the test results are presented and discussed.

Chapter 2

Click Modular Router

The **Click Modular Router** is a project that was born out of a paper by Eddie Kohler and other collaborators in 2000 [1]. It is still being given very sporadic, but important, updates by Kohler himself in his personal github repository. It is designed to be a software architecture for building configurable routers. These routers are composed of simple packet routing modules, called *elements*, that perform relatively straightforward functions. Click works by concatenating these elements (with possibly multiple inputs and outputs), in a meaningful way. This property makes it particularly easy to expand the software with user-defined elements, which was the case in this project.

Click configuration files, which specify the flow of packets by listing the elements and their respective connections, are written in their own, simple syntax. Element files in themselves use C++, with its own implementation of the C++ Standard Library, which contains equivalents to most of the STL data structures.

The software is capable of running the router either at the user level or as a Linux kernel thread. Be advised that not all elements included in the software distribution can be executed at a kernel level. One of the original goals of this project was kernel integration of the resulting application, but unforeseen errors lead to debugging at a kernel level being out of the time scope devised for it. For further details, please refer to subsection 3.1.2.

Click has a significant learning curve, especially due to lack of documentation and online resources, so perseverance is advised.

2.1 Why Click?

The Click Modular Router software had already been identified as the option with most potential from before the start of the practice in itself by my supervisor, Mr. Bachhuber.

However, part of the literature research needed included evaluating other options for the implementation of the application.

Alternatives seemed to be relatively scarce and deplorably out of date. The main ones identified were:

- X-kernel
- Scout
- OpenVZ

X-kernel is a project that was mostly maintained by researchers from the University of Arizona. Its aim is to be an "Object-based framework for implementing network protocols". Most of its functionality was aimed at the TCP/IP protocol, and was deemed to lack the flexibility needed in order to process packets that did not expressly follow the protocol. I was also unable to find a reference or paper from this millennium, which further made it seem nonviable for the realization of the flusher.

Scout was devised as a spin-off of X-kernel, also with support from University of Arizona personnel. It was described as being a "communication-oriented operating system targeted at network appliances". Despite being operating system, which would probably grant some greater measure of speed or expediency, it was also seen as not fitting the specifications. This was in no small part due to the woeful lack of updates and the sorry state of its host website and documentation.

OpenVZ, as discussed more thoroughly in a research paper by Doriguzzi et al [2], is a "modified Linux kernel tree supporting virtualization, isolation and resource management", and it is mostly used for virtualization of Linux containers. It was found by the authors to be too limiting compared to Click in different aspects for their simulation work on multi-hop wireless networks.

Adding to these considerations, the paper published by Bianco et al [3], which aims to prove the feasibility of a high performance IP router, which was more in line with our expectations of the software. It compared Click favorably in terms of reception and transmission (bit and packet) rate, among other measures, in a series of evaluations against the standard Linux network stack implementation. It especially attributes the differences to elements such as better buffer management in Click.

Regarding buffer flushing or other forms of frame dropping in general, despite being a relatively common occurrence in video delivery for bandwidth-constrained applications, as in [4] and [5], the actual action is glossed over quickly and given little attention in the available literature.

Chapter 3

Buffer Flusher

3.1 Click Element

The buffer flusher was ultimately implemented as a single element in the Click software. Like all Click elements, it consists of a header file, in which the overall specifications of the element are given, and an element file, which contains the actual code. The header file contains the information that Click needs in order for it to know in what contexts the element makes sense, how many ports it has, how it reacts to packets being pulled or pushed, etc. In this file, the flusher has a very similar profile to that of a queue, with one input and one output port, that can handle any number of pushed packets and returns the appropriate packet when pulled, if there are any present.

The element file contains the actual implementation of the buffer flusher. It was modeled from the file that belonged to the default QuickQueue implementation. The definition for the pull function is left unchanged, with the element returning the first packet in the queue when any are present and returning a Null pointer when empty. Buffer flushes can only occur when a new packet is pushed into the queue.

The push function starts by setting a pointer to the byte in the packet header which indicates whether the packet in question belongs to a *keyframe*, a video frame which was deemed to be of higher relevance and thus marked for faster forwarding. Note that the buffer flusher has no functionality to actually tell whether a frame is a *keyframe* or not, that distinction should be made by a different algorithm in another part of the process altogether, before splitting the frame itself into packets. The position of the *keyframe* identifier was set beforehand (see the section dealing with packet creation), but the position of the pointer can be easily changed in the element file code.

If the frame is not a *keyframe*, the frame gets pushed into the awaiting queue and the number of *non-keyframes* since a flush is incremented by one. If, on the contrary, the packet does identify itself as having high transmit priority, another pointer into the packet

checks whether the packet is the start of a frame or not. This makes the flushing function happen only for the start of frames, assuming of course that the whole frame has the same priority. With an average of 10 packets per frame, this small optimization should result in the flush taking place for only 10% of the packets processed. Should that not be an issue, then the second pointer can be altogether ignored.

At this point, a flush of the buffer is called. This leads to two possible cases for flushing:

1. Queue is full of *non-keyframes*
2. Queue has a mixture of *keyframes* and *non-keyframes*

In the first instance, the whole queue is simply purged. All incoming *keyframes* are pushed to the start of the queue and can be pulled immediately. In the second case, the algorithm drops a number of packets at the back of the queue equal to the number of *non-keyframe* packets since the last flush. Since all *keyframes* cause a buffer flush unless the queue is empty, this should lead the prioritized packets to be pushed to the first slot not occupied by the same kind of parcels. Thus we have an element that implements the buffer flushing according to specifications.

3.1.1 Example

Given the following packet flow, with 0 denoting a *non-keyframe* and 1 a *keyframe*, with the first packet pushed beginning on the right:

$$0 \rightarrow 0 \rightarrow 0 \rightarrow 1 \rightarrow 0 \rightarrow 1 \rightarrow 0 \rightarrow 0$$

The buffer flusher would execute the following with each packet, assuming no pull requests reach the buffer flusher:

1. Push *non-keyframe* packet, 1 packet since last flush.
2. Push *non-keyframe* packet, 2 packets since last flush.
3. Buffer flush (Case 1, whole queue discarded), push *keyframe* packet. No packets since last flush.
4. Push *non-keyframe* packet, 1 packet since last flush.
5. Buffer flush (Case 2, 1 packet flushed), push *keyframe* packet. No packets since last flush.
6. Push *non-keyframe* packet, 1 packet since last flush.
7. Push *non-keyframe* packet, 2 packets since last flush.
8. Push *non-keyframe* packet, 3 packets since last flush.

This leaves the queue in the following order, with queue front at the right:

$$0 \rightarrow 0 \rightarrow 0 \rightarrow 1 \rightarrow 1$$

3.1.2 Kernel integration

As previously mentioned, one of the goals of the project was the integration of the application into the Linux kernel. The relative ease with which Click allows this was also one of the reasons why it was chosen as an appropriate software. The development and early testing of the buffer flusher occurred at the user level. After the correct functionality of the application was tested at this level, I tried to run it as a kernel thread, but ran into all sorts of trouble.

Running the router configuration in the kernel level causes Linux to intercept the thread and lock the CPU in which it is running for a certain amount of time. This loops after the timeout expires. I managed to isolate the problem to the buffer flusher element in itself, but could get no further without a deep understanding of the kernel and how it works with Click.

The painstaking process of kernel thread debugging was deemed too laborious for the scope of this engineer's practice, as well as too complex for my current skill set. My supervisor and I decided jointly to run the tests at the user level and leave any kernel integration for possible future projects.

3.2 Packet creation

Part of the scope of the project included the creation of the packets from the original trace files provided by my supervisor. To preserve the integrity of test results and the functionality of the application, I sought to build the packages in a way that was unobtrusive and maintained both the size specified in the trace and all the information indicated in the files.

I decided on building the packets with a simple script with the help of *"Scapy"*, a Python-based packet manipulation program and corresponding module with especially accessible forging of parcels. The information was codified in simple fields of four bytes and then filled with dummy data to achieve the correct size. These would then be sent from one machine to another through a UDP socket in order to crudely simulate the traffic coming in from the video encoder. This also significantly facilitates the packet capture by Click itself. The machine receiving the packages would then run the flushable buffer with the incoming information. For more details on packet structure, please see Appendix A.

Chapter 4

Testing

The testing was centered around evaluating the way the buffer flusher deals with different volumes and patterns of incoming packet traffic. This meant simulating different situations of channel saturation for the path between the creation of the packages and the buffer flusher. Evaluation of the forwarding of the packages at the output of the buffer flusher was deemed unimportant and as such was not as thoroughly investigated. The test environment was as follows:

Packet Creator → Buffer Flusher → Packet Capture

Path A is a channel emulated with the *Netropy Network Emulator* in the institute Robotics lab, under a series of bandwidths and loads. Path B is a virtual channel emulated with a certain bandwidth that is included in the router configuration file for Click. It routes the output packages of the buffer flusher into a packet capture software (e.g. *Wireshark*) and stores them as a *.pcap* dump file.

The following bandwidths (among others) were investigated in certain relevant combinations for both paths:

- 1 Mbps
- 4 Mbps
- 16 Mbps

For readability and legibility, only certain combinations are presented and analyzed in this report. The rest of the data is contained in the *test_results* repository.

These bandwidth combinations were chosen for illustrative purposes and do not necessarily represent maximum or minimum values for correct function of the buffer flusher.

- Path A : 1 Mbps, Path B : 4 Mbps

This configuration is meant to test the case in which the outgoing packet rate to the flusher exceeds the income rate, which should not be the norm for most. There should be little need for flushing due to very short queues in the flusher itself.

- Path A : 16 Mbps, Path B : 1 Mbps

  This configuration tests a high incoming packet rate with a low outgoing rate, a more realistic example. This environment should trigger the flushing mechanism at fairly regular intervals and evidence a much higher percentage of *keyframe* packages after the flusher.

- Path A : 16 Mbps, Path B : 16 Mbps

  Lastly, both income and outcome rate are roughly equal. A similar result to the first configuration should be observed in terms of package count, but with much higher expediency.s

Additionally, the following channel loads or cross-traffic patterns were investigated.

- No channel load
- Random cross-traffic
  - 25% channel load
  - 50% channel load
  - 75% channel load
- Sample pattern of cross-traffic

The random cross-traffic patterns produce Poisson-distributed bursts of packets of 1500 Bytes of length, around the size of the average packet length.

The sample pattern of cross-traffic was a random packet flow synthesized with packet lengths similar in size to the video traces used in the experiment.

Two video traces were used and had the following properties:

Video traces			
Channel type	Total packages	*Keyframe* packages	*Keyframe* percentage
Slow Channel	1465	687	46.9%
Medium Channel	15677	632	4.0%

Table 4.1: Video traces used for testing

4.1 Test Results

4.1.1 No cross-traffic

The tests run with no cross-traffic serve primarily to set a benchmark to compare other testing conditions with, as well as corroborate the function of the application.

BW A (Mbps)	BW B (Mbps)	Total packages	*Keyframe* packages	Keyframe %	*Non-Keyframe* Packages	Time elapsed (sec)
Slow Channel						
1	4	1465	687	46.9%	778	17.4
16	1	697	687	98.6%	10	1.1
16	16	1465	687	46.9%	778	1.1
Medium Channel						
1	4	15677	632	4.0%	15045	186.3
16	1	1163	632	54.3%	531	11.6
16	16	15677	633	4.0%	15046	11.7

Table 4.2: No cross-traffic

Looking at the numbers Table 4.2, it becomes apparent that my expectations were more or less correct for the measurements taken. In the first path configuration, all packages sent are received and the percentage of *keyframes* remains unchanged; this means that no buffer flushing was necessary during processing of the packets. It does not matter whether Path B has the same bandwidth or indeed a larger one by an order of magnitude, the queue implemented in the flusher is quite capable of handling the incoming traffic without a hitch. This is also the case for the third path configuration, with the only significant difference between the two being the time elapsed for packets to be received by the packet sniffer at the end of the path. This, of course can be explained quite easily by the difference in bandwidths.

The second path configuration, as expected, is much more interesting in terms of function of the buffer flusher.

A preliminary look at the data for this path shows a very important difference with the video traces before entering the flusher. In the case of the slow channel, the keyframe percentage jumps from 46.9% of all incoming packets to 98.6%, more than a twofold increase.

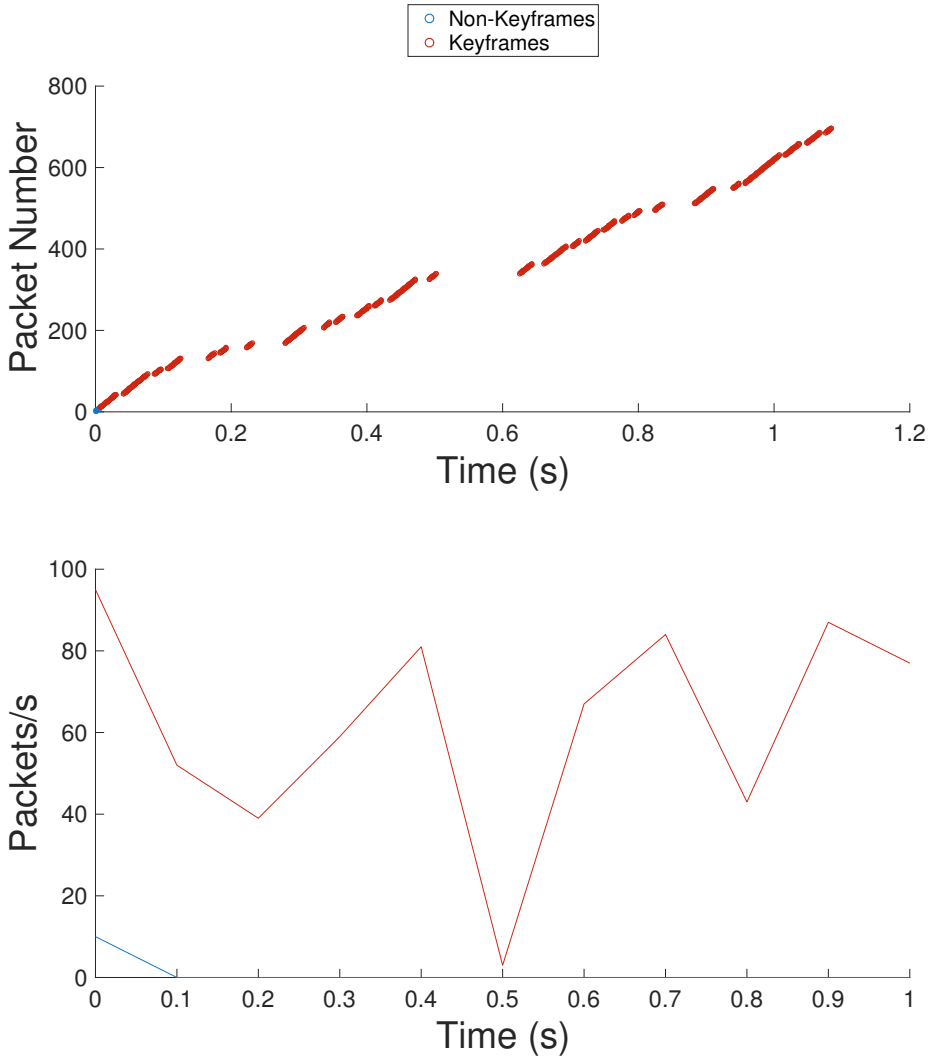


Figure 4.1: No cross-traffic; Slow Channel (Path A : 16 Mbps, Path B : 1 Mbps), measured at the end of Path B

In the case of the medium channel the increase is even more drastic, going from 4.0% all the way up to 54.3%, thus increasing by a factor larger than 12.

Despite this, the amount of *keyframes* remains constant, correctly sending the packet sniffer all packets that it receives. The time elapsed also suffers no perceptible change compared to the case with no buffer flushing, despite the high activity of the flusher.

A particularly interesting observation in both Figure 4.1 and 4.2 is the large gap of packets

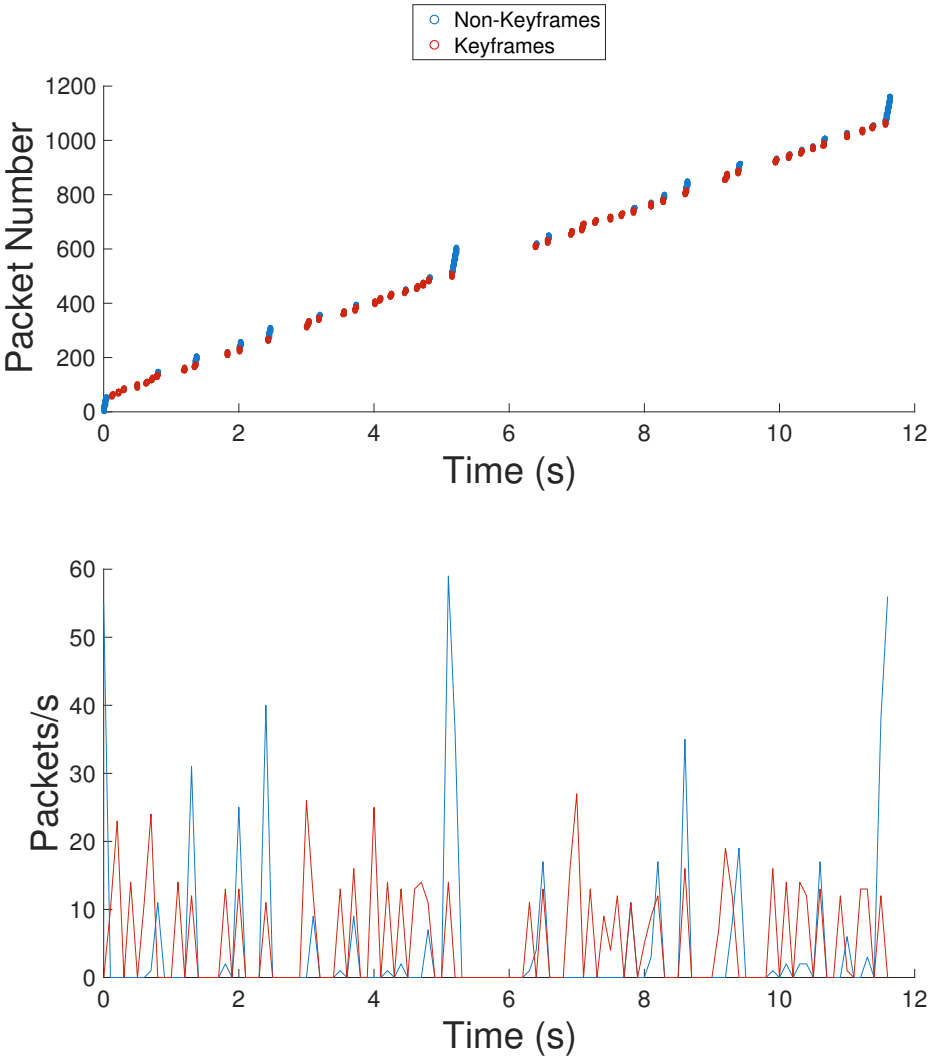


Figure 4.2: No cross-traffic; Medium Channel (Path A : 16 Mbps, Path B : 1 Mbps), measured at the end of Path B

sent around the middle of the time axis. This corresponds to one of the largest buffer flushes that take place, a fact that one can easily corroborate by looking at the amount of *non-keyframes* that fill that gap in other bandwidth combinations (see Appendix B). In terms of video frames, this would probably signify a relatively long scene showing very similar images from frame to frame.

4.1.2 Random cross-traffic

As previously mentioned, these tests were run by occupying a certain percentage of the channel capacity with random bursts of packets. In this section we discuss the case of 50% channel occupation to illustrate overall observations. The other results are included in the repository *test_results* and exhibit very similar results.

BW A (Mbps)	BW B (Mbps)	Total packages	*Keyframe* packages	Keyframe %	*Non-Keyframe* Packages	Time elapsed (sec)
Slow Channel						
1	4	1465	687	46.9%	778	35.6
16	1	710	687	96.8%	23	2.3
16	16	1465	687	46.9%	778	2.2
Medium Channel						
1	4	15677	632	4.0%	15045	371.8
16	1	2195	632	28.8%	1563	23.8
16	16	15677	633	4.0%	15046	23.7

Table 4.3: Random cross-traffic, 50% channel occupation

As can be ascertained from Table 4.3, the operation of the flusher is extremely similar to the case without cross-traffic. The similarity between the first and third bandwidth combinations is roughly maintained, including the ratio between their two elapsed times.

The first of the two largest differences is the differences in time elapsed, both in the slow and the medium channel. The transmitted video traces take around twice as long to be ready to be transmitted, compared to the first case observed. This is, naturally, due to the channel occupation rate, evident in the neat correspondence between a twofold increase in time and a halving of the available channel.

The second difference lies in the case with active buffer flushing. Note the fact that in no occasion is the absolute amount of *keyframe* packets lower than expected: all of them continued to be ready to be transmitted as expected, with no drops. It is the relative percentage of *keyframe* packets that sinks compared to the first set of measurements.

In the slow channel, the relatively low amount of overall packages masks this phenomenon, with a change of only 1.8% between both sets. In the medium channel, however, the difference is 25.5%. The reason lies in the effectively lower bandwidth with which the

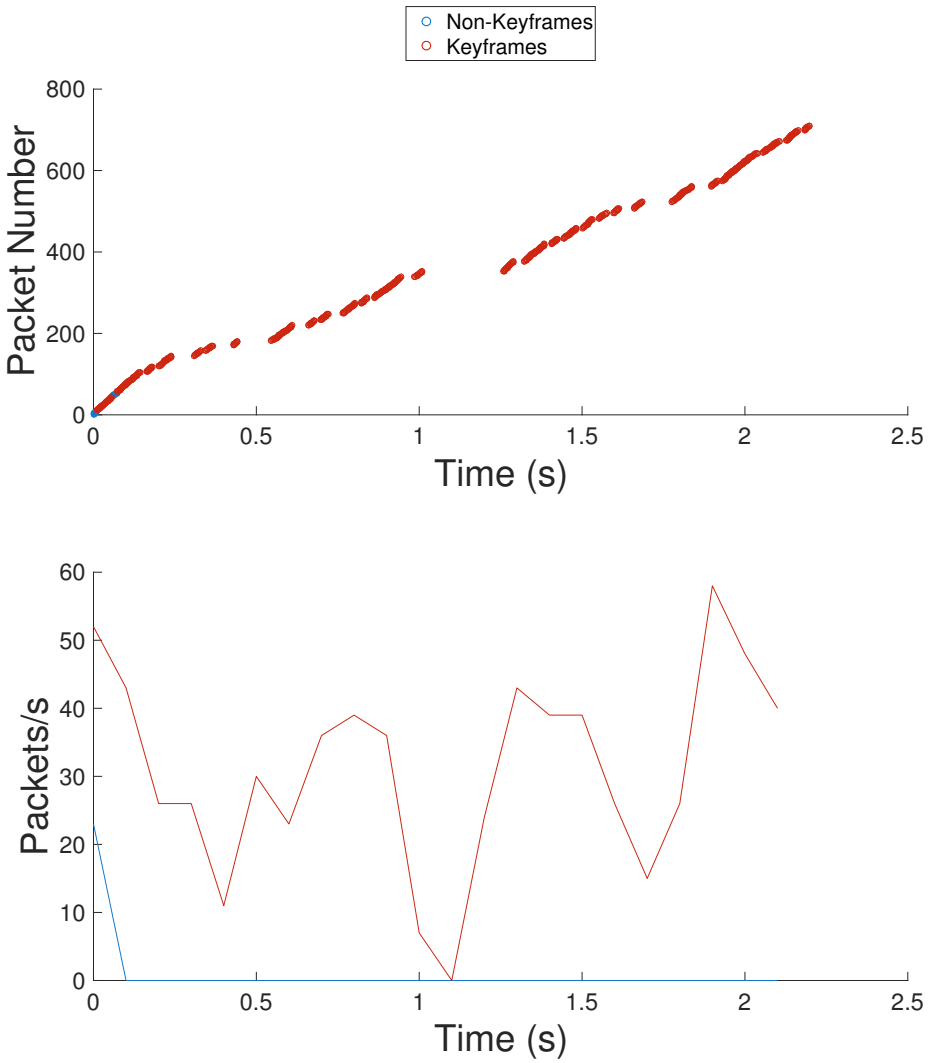


Figure 4.3: Random cross-traffic; Slow Channel (Path A : 16 Mbps, Path B : 1 Mbps), 50% channel occupation rate, measured at the end of Path B

buffer flusher receives packages from the channel. With a lower incoming rate, the flusher triggers less frequently, thus letting more *non-keyframe* packages through. Thus, despite small disruption, the application still exhibits the expected function.

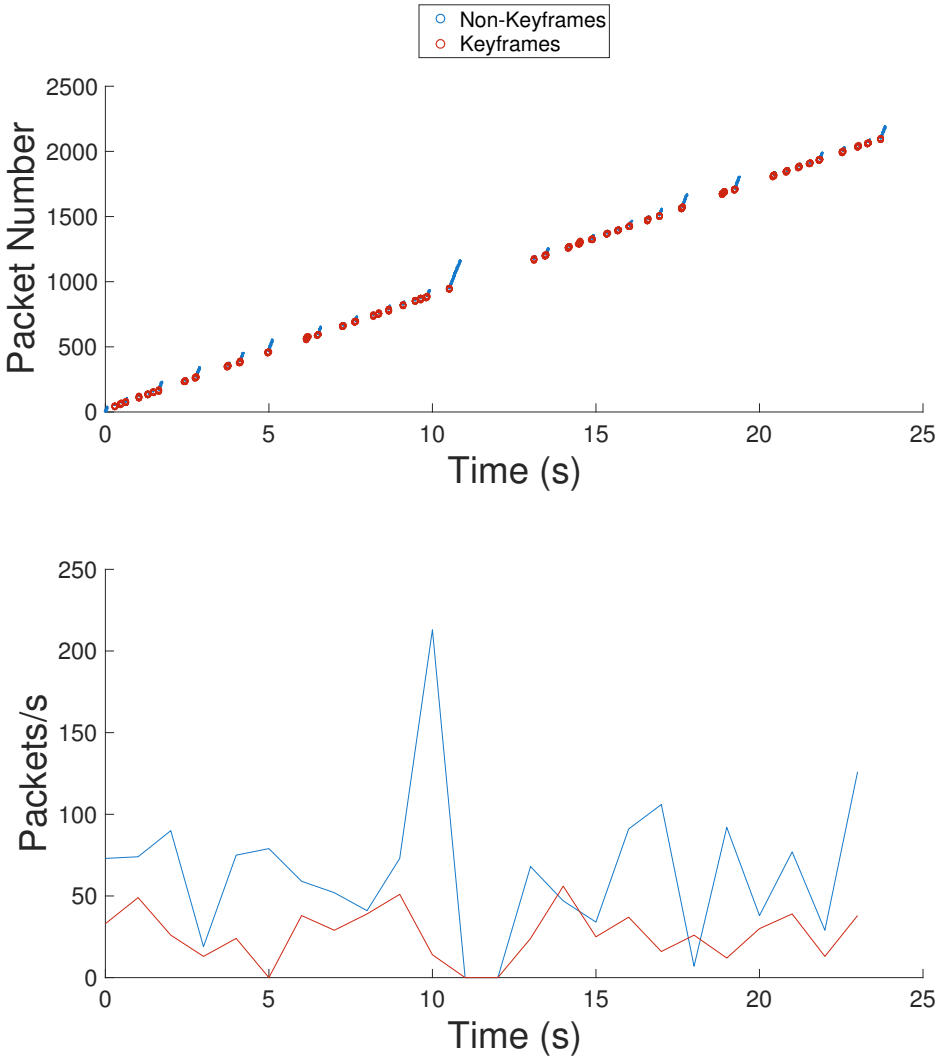


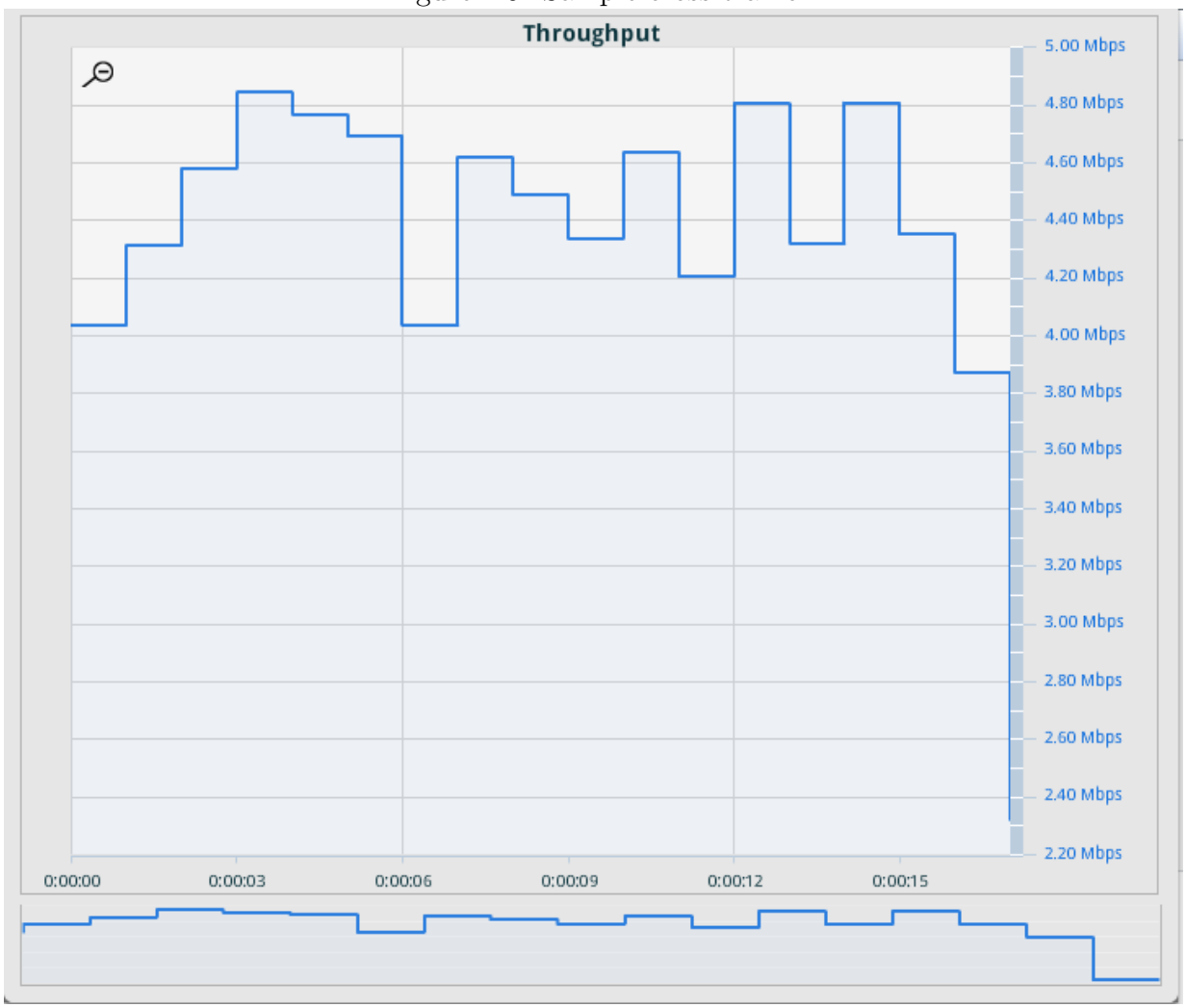
Figure 4.4: Random cross-traffic; Medium Channel (Path A : 16 Mbps, Path B : 1 Mbps), 50% channel occupation rate, measured at the end of Path B

4.1.3 Sample pattern of cross-traffic

Finally, the third set of measurements was performed with background channel traffic corresponding to a packet capture synthesized especially for this purpose and designed to resemble a normal usage of the channel by some other application. The file containing the capture was uploaded to the network emulator, where it was replayed along with the packet transfer.

A graphic representation of the sample can be seen in Figure 4.5.

Figure 4.5: Sample cross-traffic



A glancing look at Table 4.4 reveals very familiar numbers, evidencing the similarity between this measurement set and the first.

In this set, the phenomena already discussed in the previous two are replicated. The amount of *keyframe* packets remains the same, without any drops. The percentage of

BW A (Mbps)	BW B (Mbps)	Total packages	*Keyframe* packages	Keyframe %	*Non-Keyframe* Packages	Time elapsed (sec)
Slow Channel						
1	4	1465	687	46.9%	778	17.5
16	1	710	687	96.8%	23	1.2
16	16	1465	687	46.9%	778	1.2
Medium Channel						
1	4	15677	632	4.0%	15045	191.5
16	1	1234	632	51.2%	602	12.4
16	16	15677	632	4.0%	15046	12.5

Table 4.4: Sample cross-traffic

keyframes compared to the overall packets is much closer to that of the first, unhindered transmission, with a very small dent that can be explained by the rate of channel utilization.

The graphs in Figures 4.6 and 4.7 exhibit a slightly more erratic behavior compared both to the first and second measurement sets, most probably due to the fact that the background traffic exhibits more variation than in the Poisson-distributed random cross-traffic case. Despite this, the overall trends remain constant.

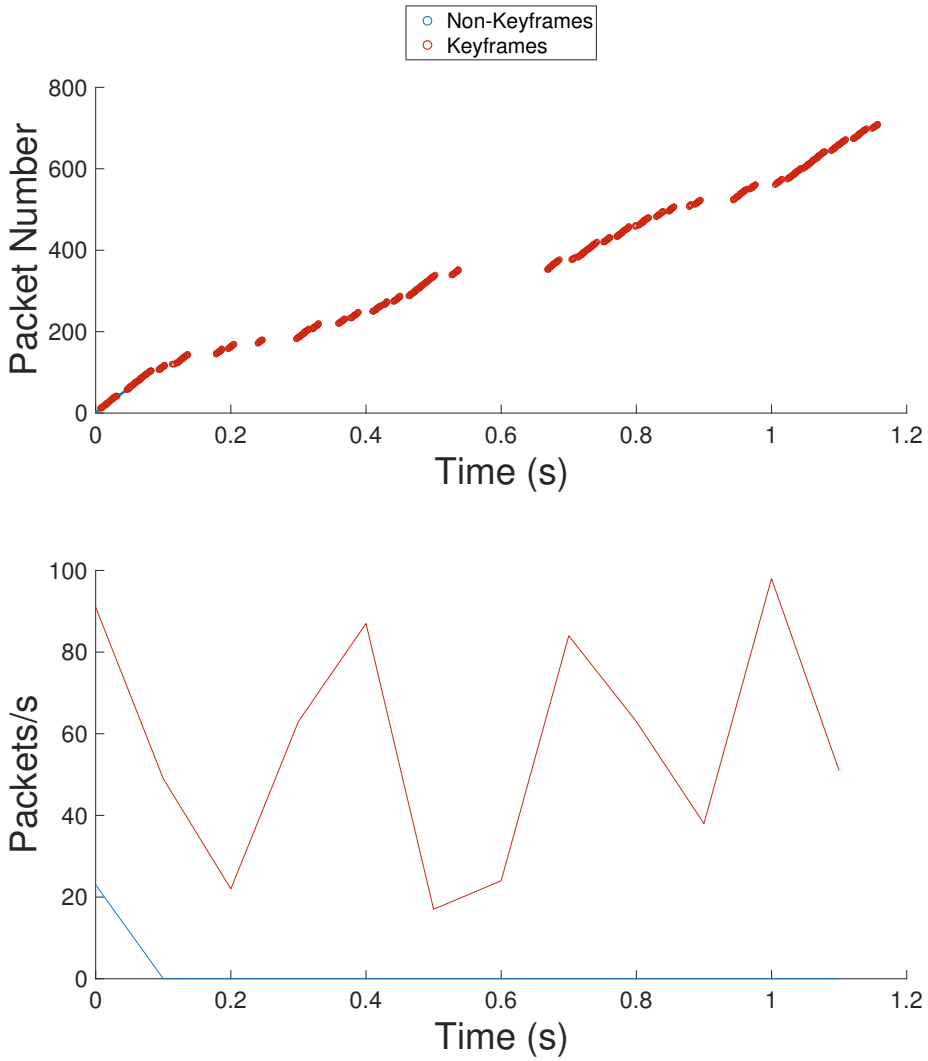


Figure 4.6: Sample cross-traffic ; Slow Channel (Path A : 16 Mbps, Path B : 1 Mbps), measured at the end of Path B

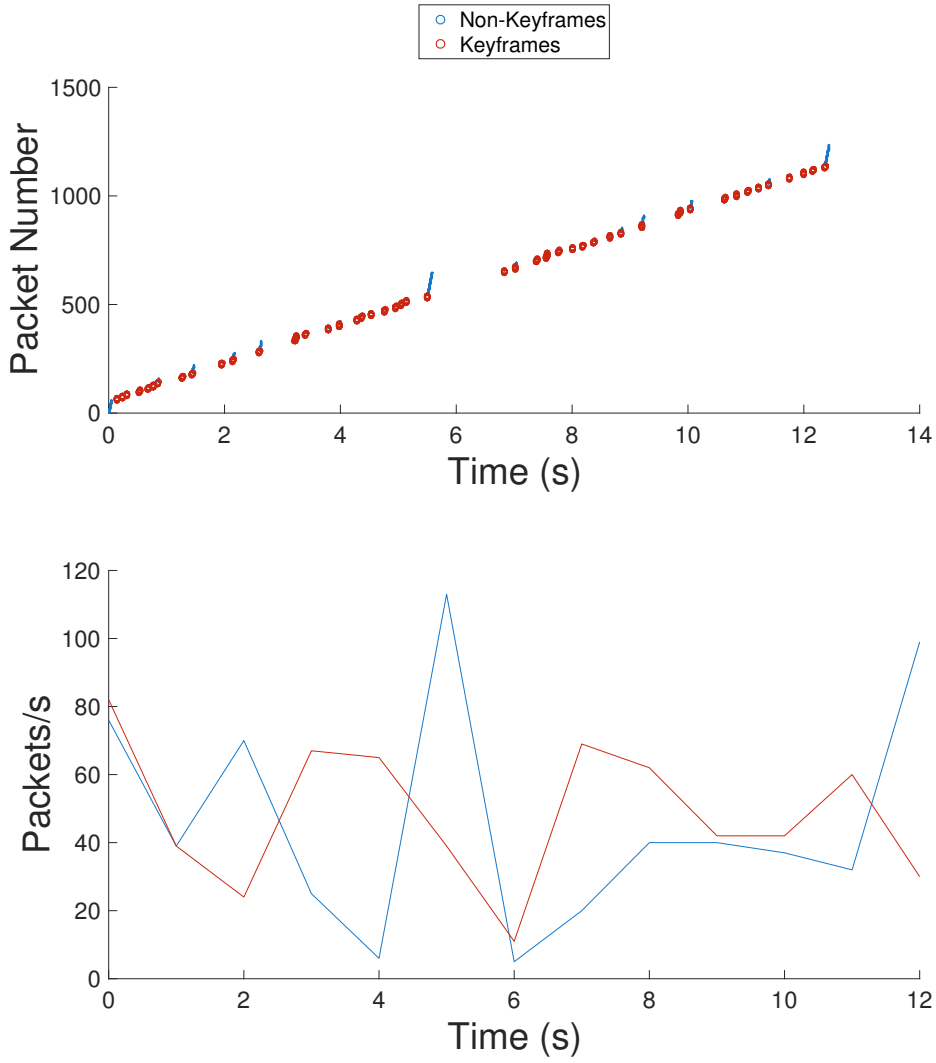


Figure 4.7: Sample cross-traffic ; Medium Channel (Path A : 16 Mbps, Path B : 1 Mbps), measured at the end of Path B

Chapter 5

Conclusions

In the course of our testing, we have observed the performance of a buffer flusher for packets of video frames under different constraints. These have included different combinations of incoming and outgoing data rates as well as various forms of channel occupancy between packet creation and the buffer flusher. An effort was also made to leave behind documentation (both for Click and the packet creation scripts) in order to facilitate any other further work in this topic.

The developed buffer flusher has demonstrated the capacity to process large amounts of packets in varying time frames, effectively flushing those with lower priority when the need arises. Additionaly, none of the *keyframes* were dropped in any experiment.

The effectiveness of the ultimate goal, which is the reduction of the overall transmitted data necessary to transfer all the *keyframes* from one service to the other, can be easily calculated. Table 4.5 illustrates this using the results from the second measurement set.

As is especially visible in files with a higher percentage of *non-keyframes*, the difference between unadulterated and flushed files is significant, reaching up to a 92.6% reduction in total file size to be transferred in the case with no cross-traffic.

All of the observations described in this report also reflect the measurements not discussed here due to concerns for repetitiveness or readability. For the packet capture files of these other measurements please refer to the *test_results repository*.

Thus we have fulfilled all the initial goals of the project except for the integration with the Linux kernel, which was deemed out of the time scope of this Engineer practice.

Channel Type	Total packets	*Keyframe* percentage	File size
Video traces before flushing			
Slow Channel	1465	46.9%	2.20 MB
Medium Channel	15677	4.0%	23.52 MB
No cross-traffic			
Slow Channel	697	98.6%	1.05 MB
Medium Channel	1163	54.3%	1.74 MB
Random cross-traffic			
Slow Channel	710	96.8%	1.07 MB
Medium Channel	2195	28.8%	3.30 MB
Sample cross-traffic			
Slow Channel	710	96.8%	1.07 MB
Medium Channel	1234	51.2%	1.85 MB

Table 5.1: File size to be sent after buffer flusher

Appendix A

Packet structure

Table A.1: Breakdown of packet structure by byte

Packet Structure		
Content	Bytes	Description
Ethernet Header	0 - 13	Header due to packaging.
IPv4 Header	14 - 33	Header due to packaging.
UDP Header	34 - 41	Header due to packaging.
Frame Identifier	42 - 45	Binary variable coded as integer, either *keyframe* or not.
Frame Number	46 - 49	Used to monitor that order of packages is not scrambled. Coded as integer.
Segment Number	50 - 53	Can be kept track of to detect any drops, coded as an integer.
Packet Size	54 - 57	For most packets it equals 1538 bites. Coded as long.
Event Occurred	58 - 61	Uninteresting for our purposes. Coded as float.
Sampling time	62 - 65	Same as above. Coded as a float.
Dummy Data	66 onward	Necessary filler to achieve specified length. Filled with zeroes.

Appendix B

Other test results

B.1 No crosstraffic

B.2 Random Cross-traffic

B.3 Sample Crosstraffic

Figure B.1: No cross-traffic ; Slow Channel (Path A : 1 Mbps, Path B : 4 Mbps)

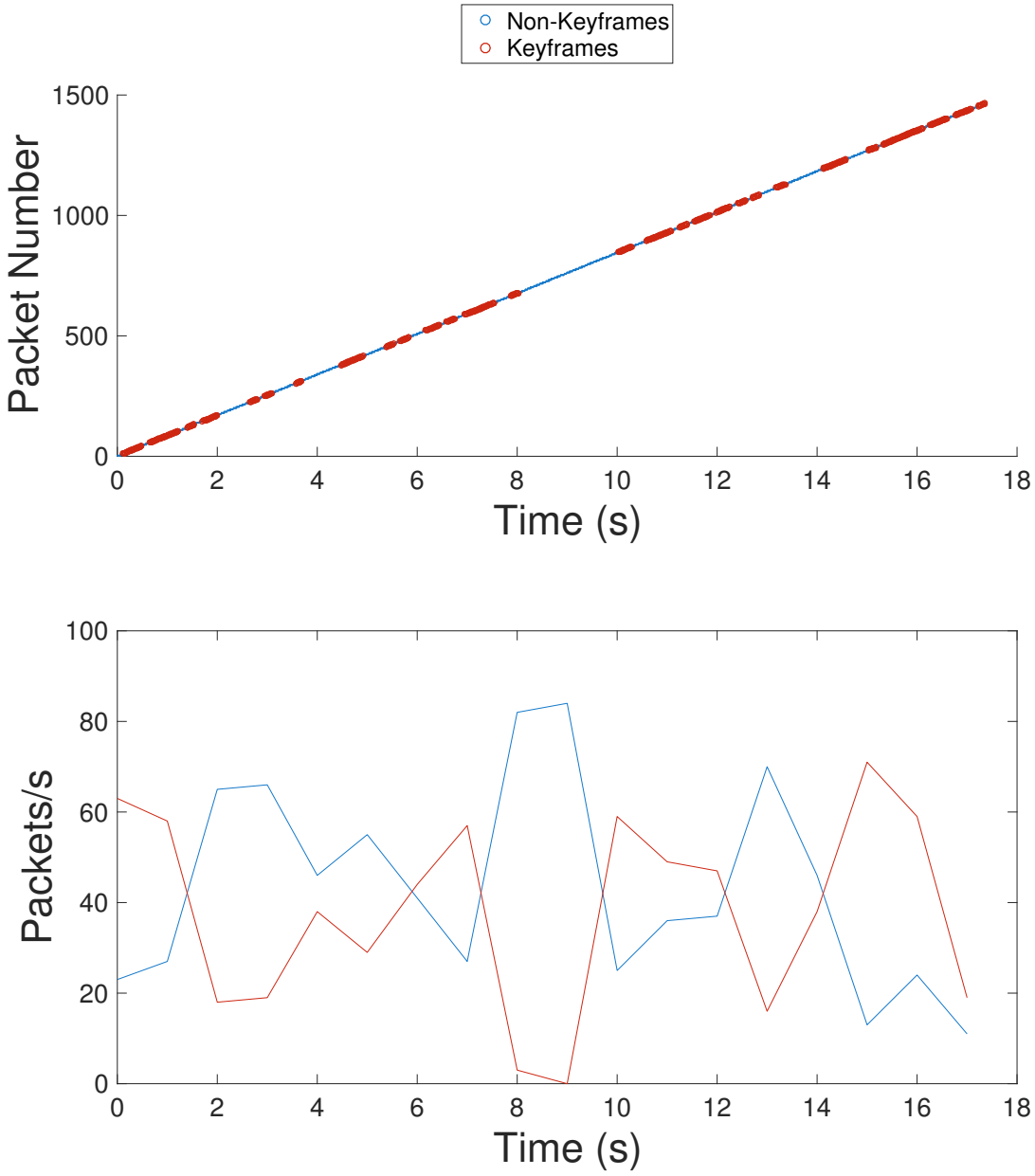


Figure B.2: No cross-traffic ; Medium Channel (Path A : 1 Mbps, Path B : 4 Mbps)

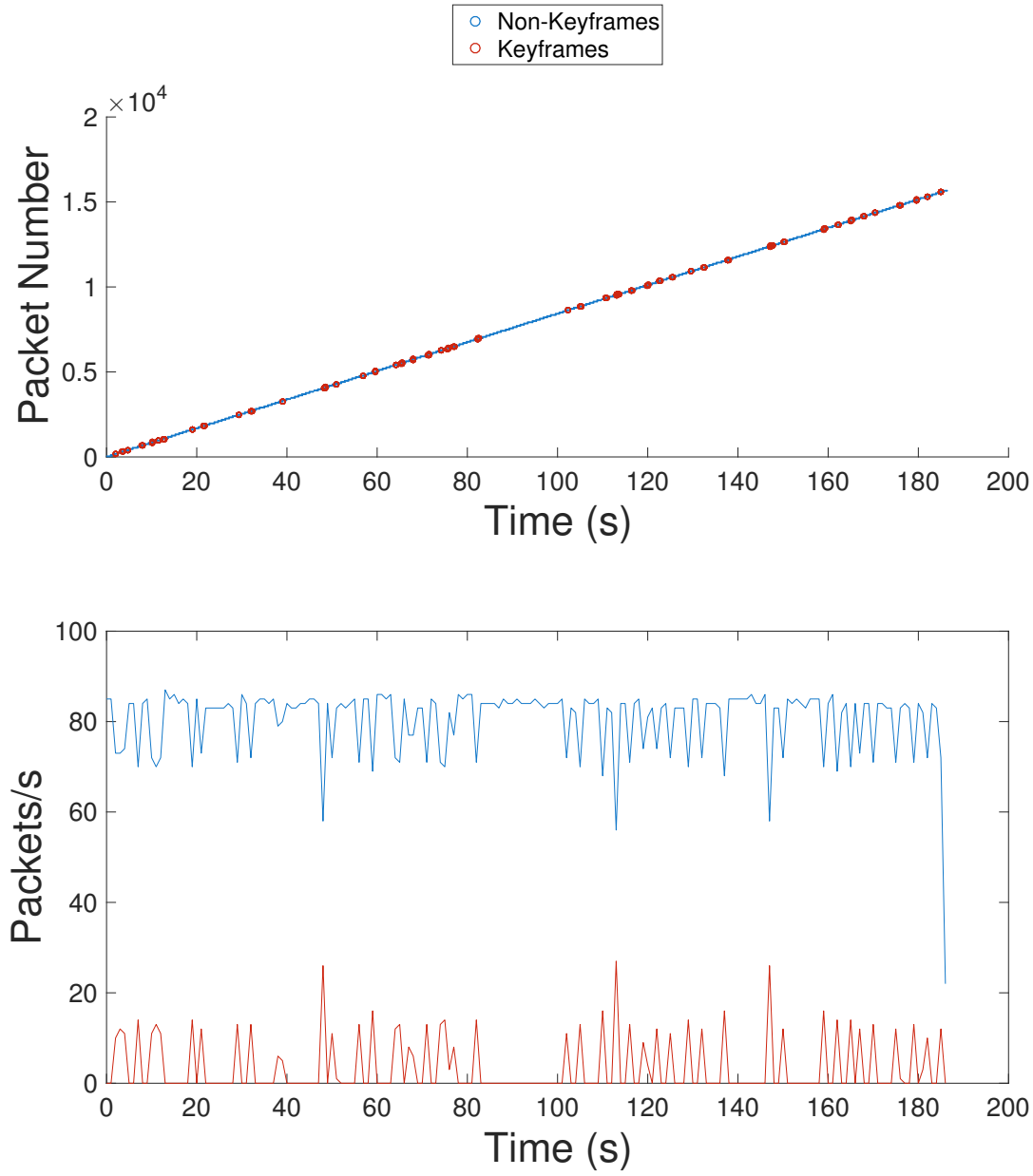


Figure B.3: No cross-traffic ; Slow Channel (Path A : 16 Mbps, Path B : 16 Mbps)

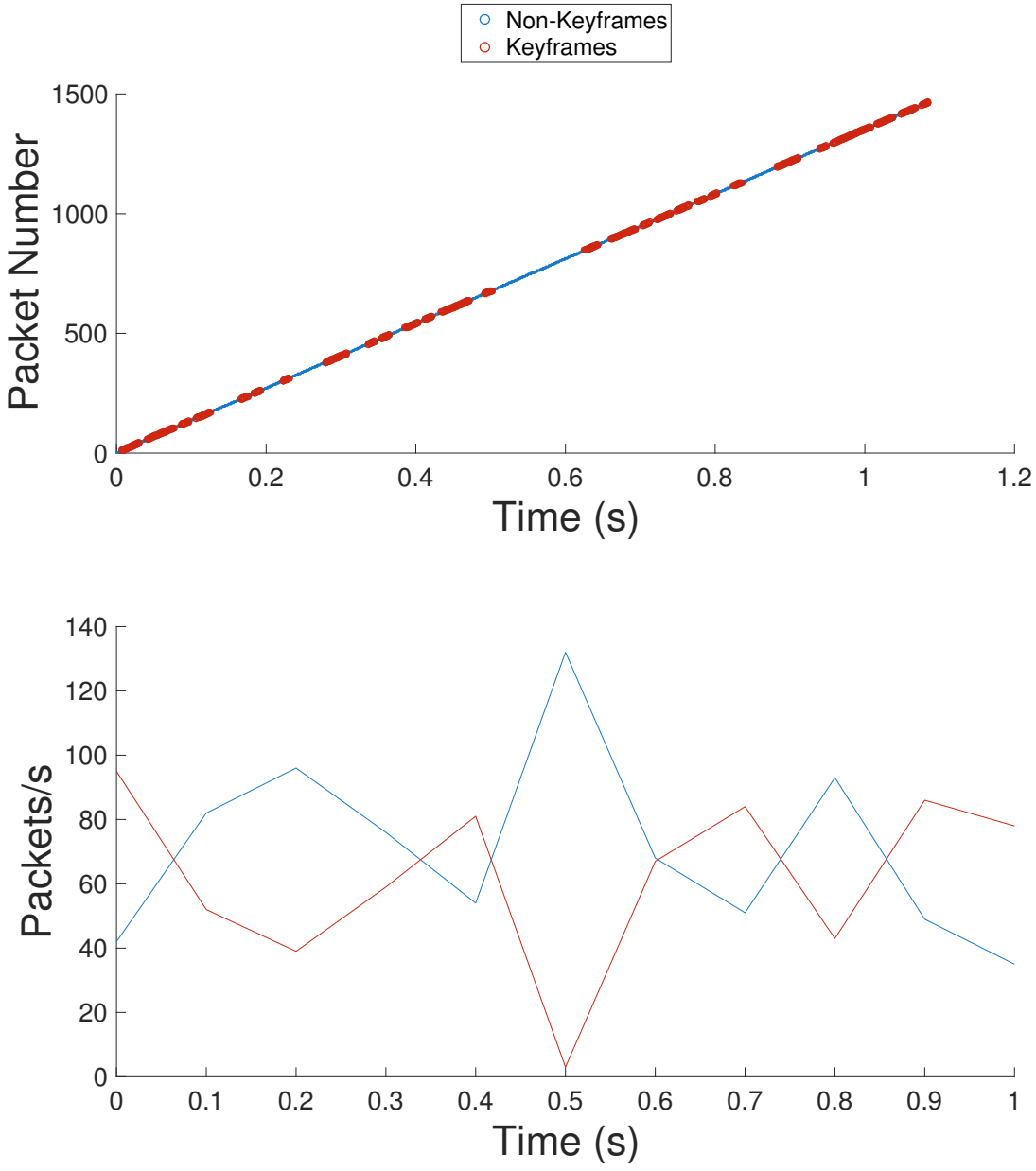


Figure B.4: No cross-traffic ; Medium Channel (Path A : 16 Mbps, Path B : 16 Mbps)

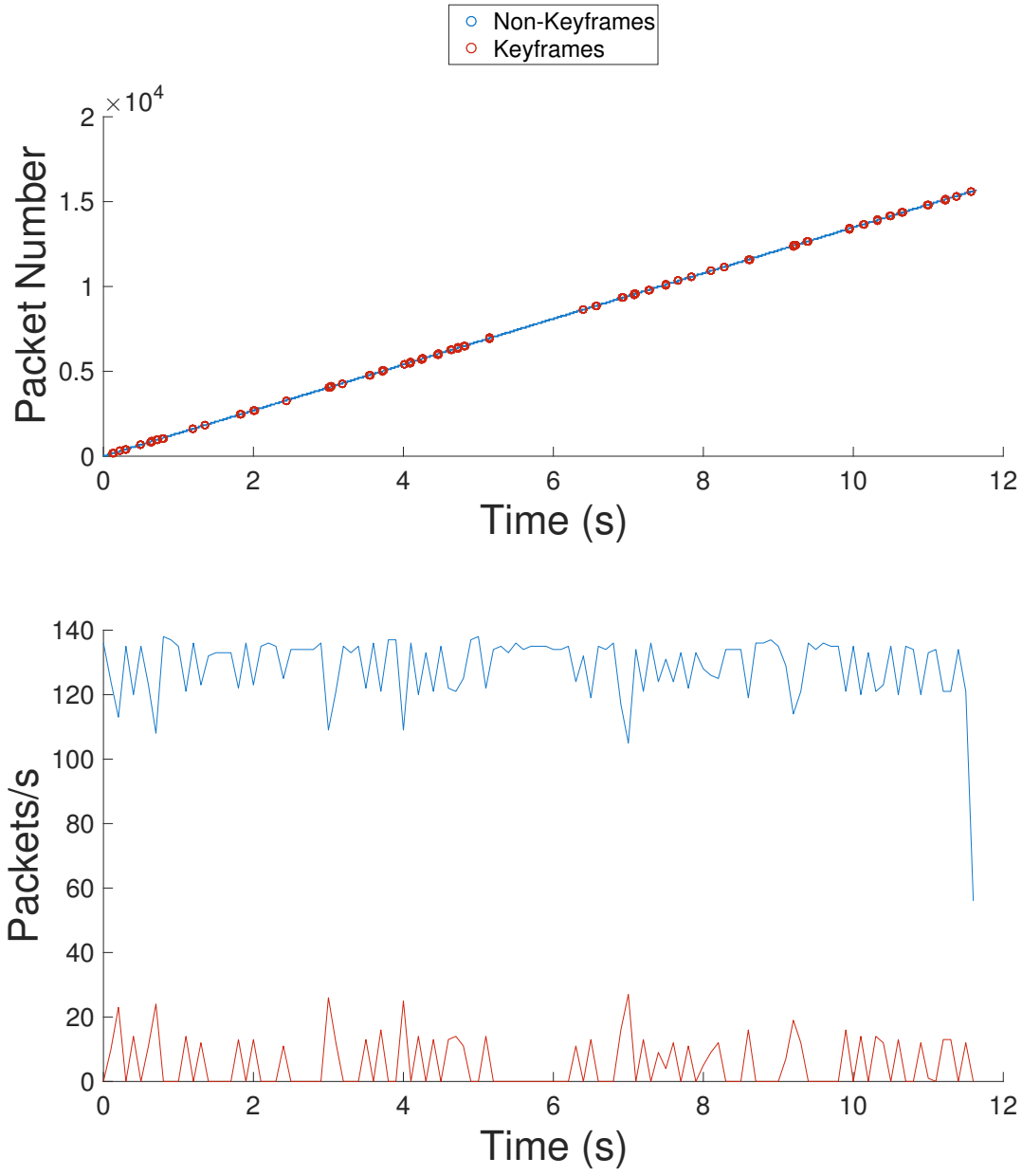


Figure B.5: Random cross-traffic ; Slow Channel (Path A : 4 Mbps, Path B : 1 Mbps), 50% channel occupation

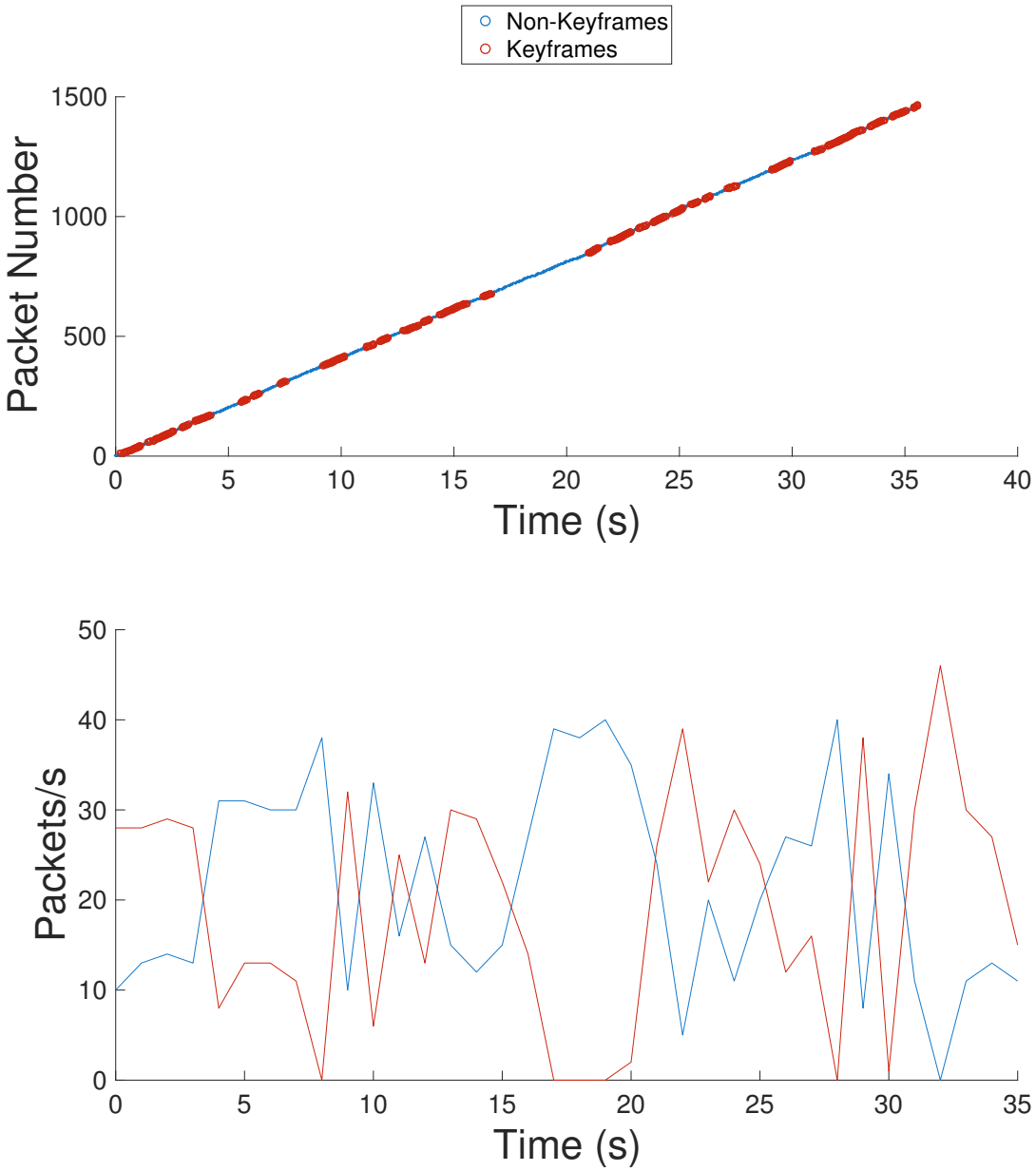


Figure B.6: Random cross-traffic ; Medium Channel (Path A : 4 Mbps, Path B : 1 Mbps), 50% channel occupation

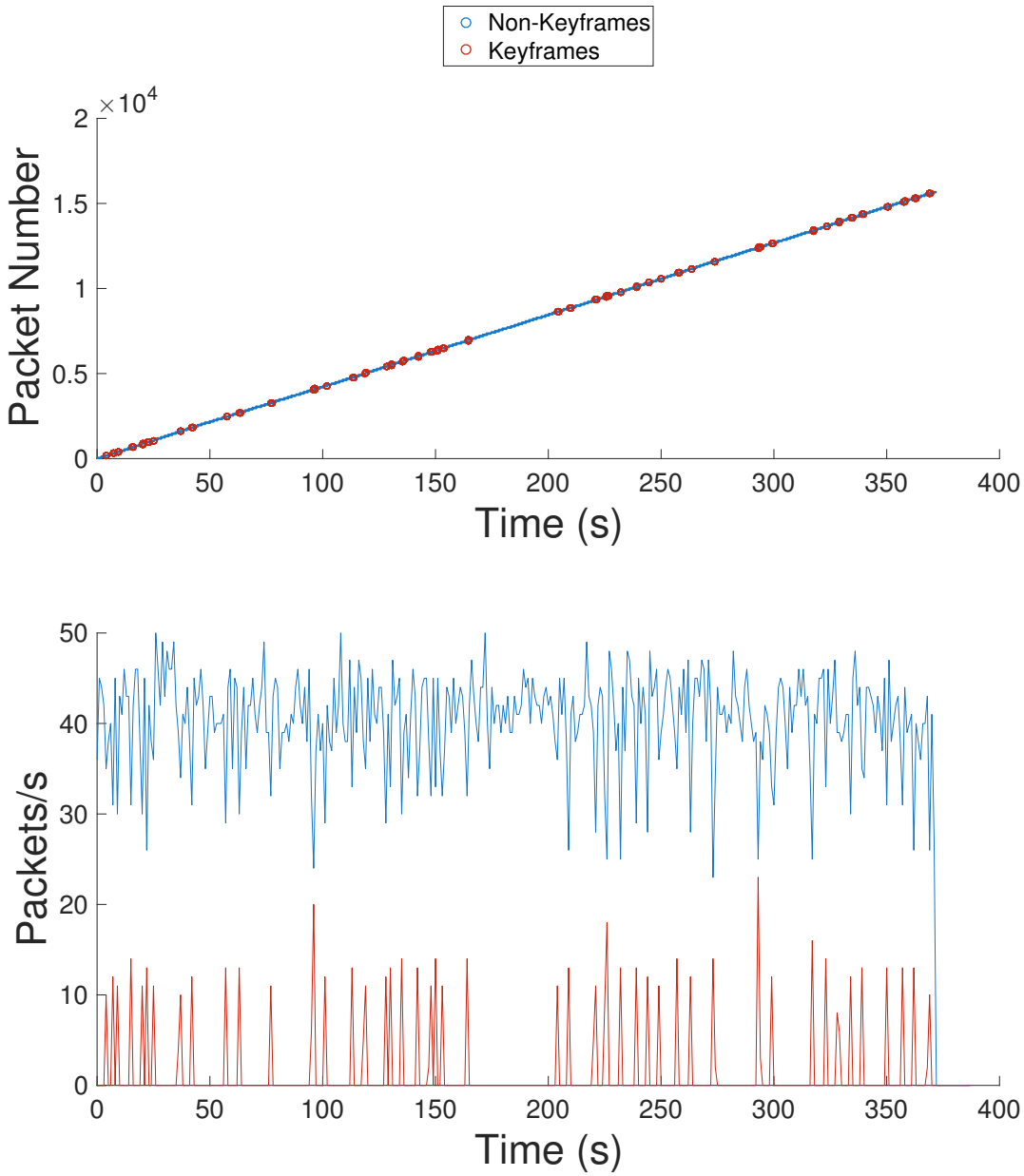


Figure B.7: Random cross-traffic ; Slow Channel (Path A : 16 Mbps, Path B : 16 Mbps), 50% channel occupation

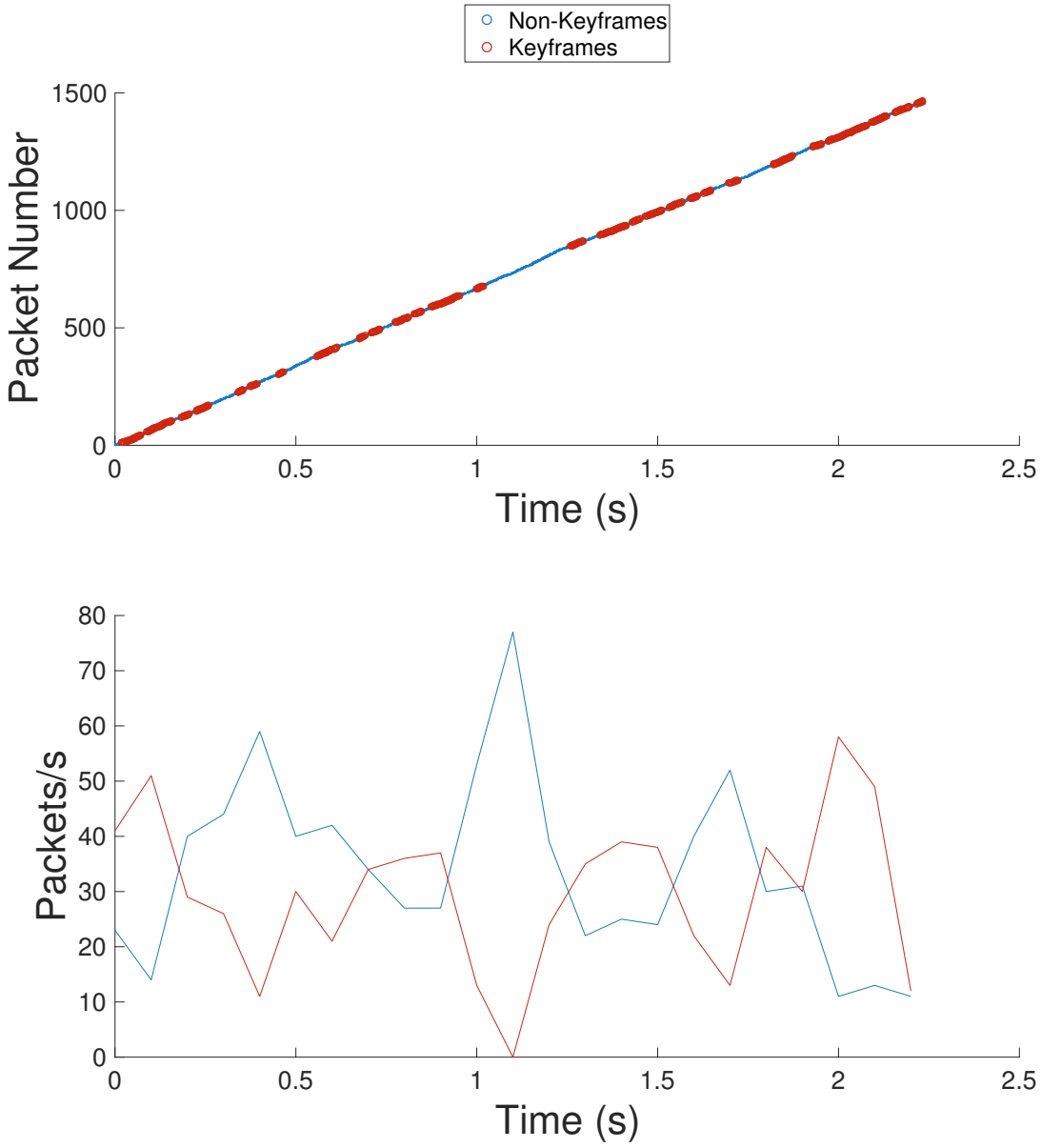


Figure B.8: Random cross-traffic ; Medium Channel (Path A : 16 Mbps, Path B : 16 Mbps), 50% channel occupation

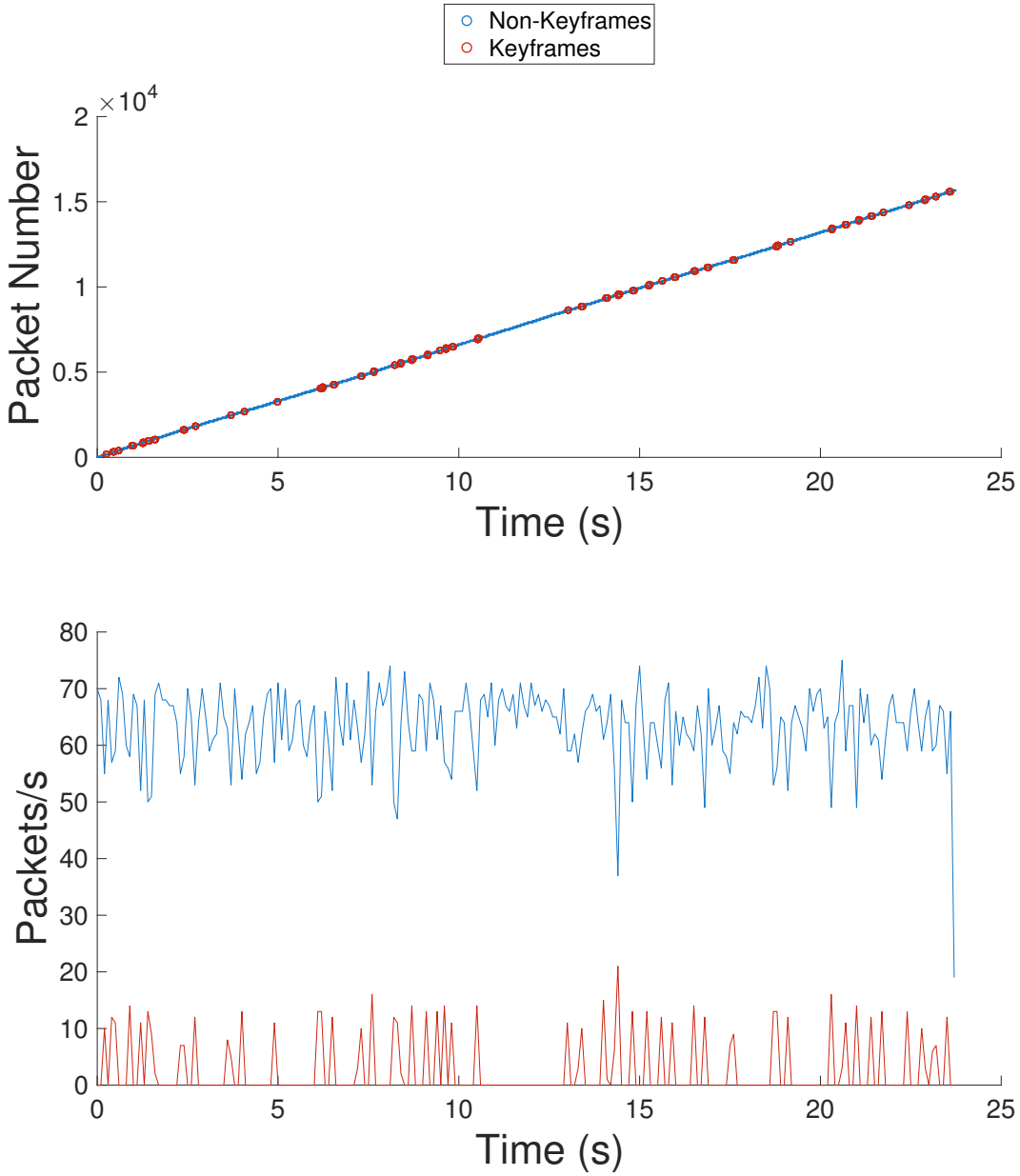


Figure B.9: Sample cross-traffic ; Slow Channel (Path A : 4 Mbps, Path B : 1 Mbps)

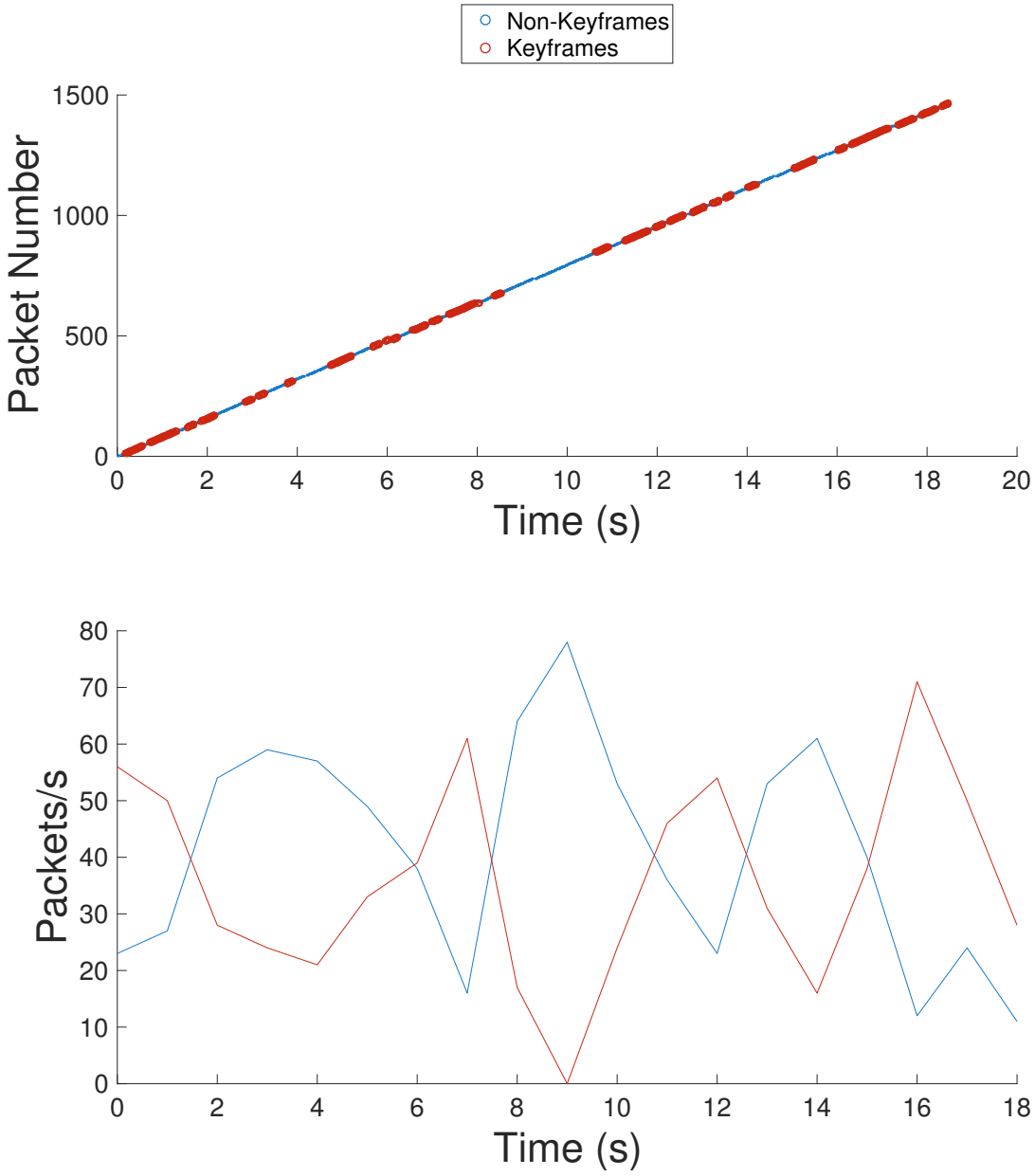


Figure B.10: Sample cross-traffic ; Medium Channel (Path A : 4 Mbps, Path B : 1 Mbps)

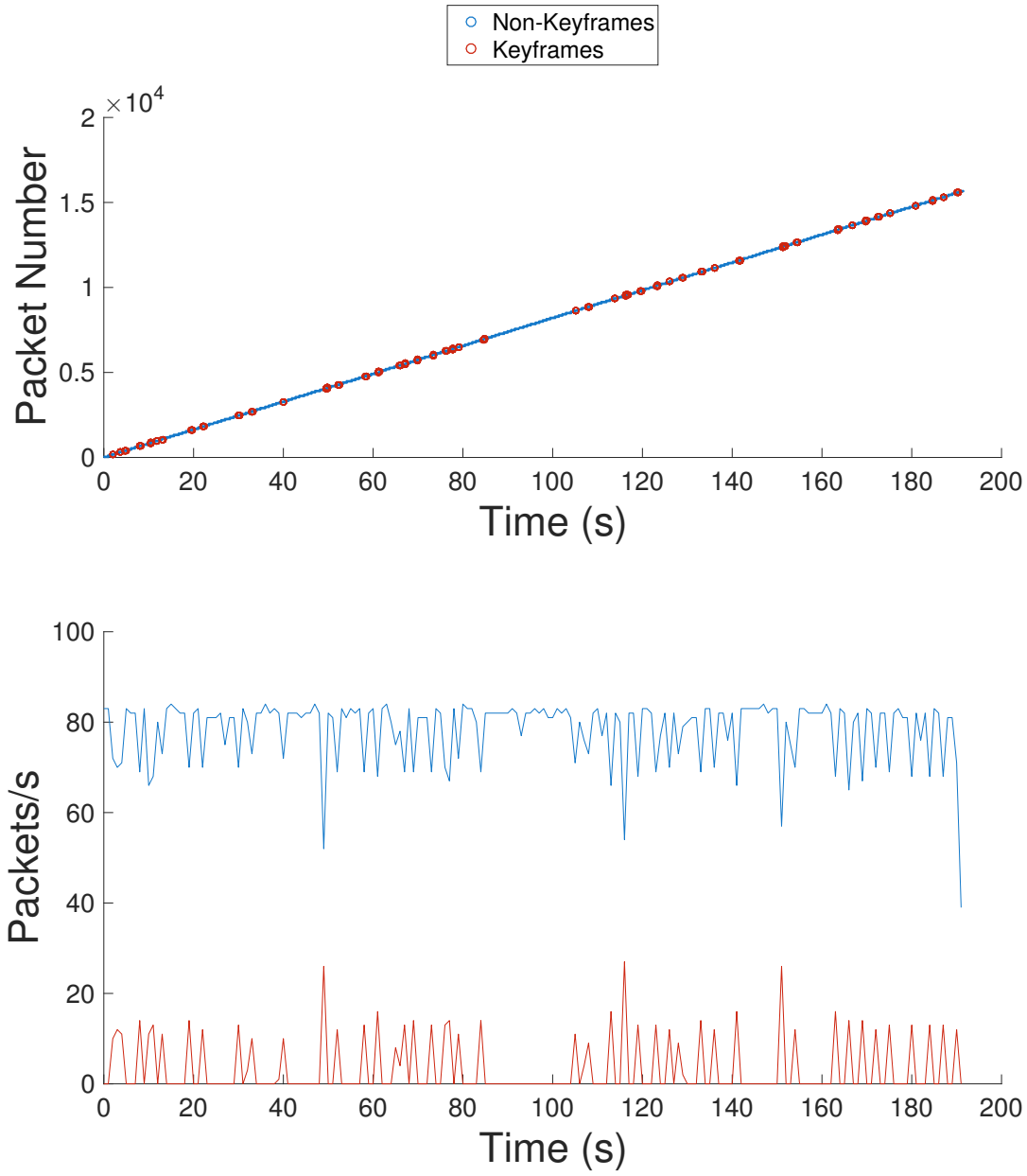


Figure B.11: Sample cross-traffic ; Slow Channel (Path A : 16 Mbps, Path B : 16 Mbps)

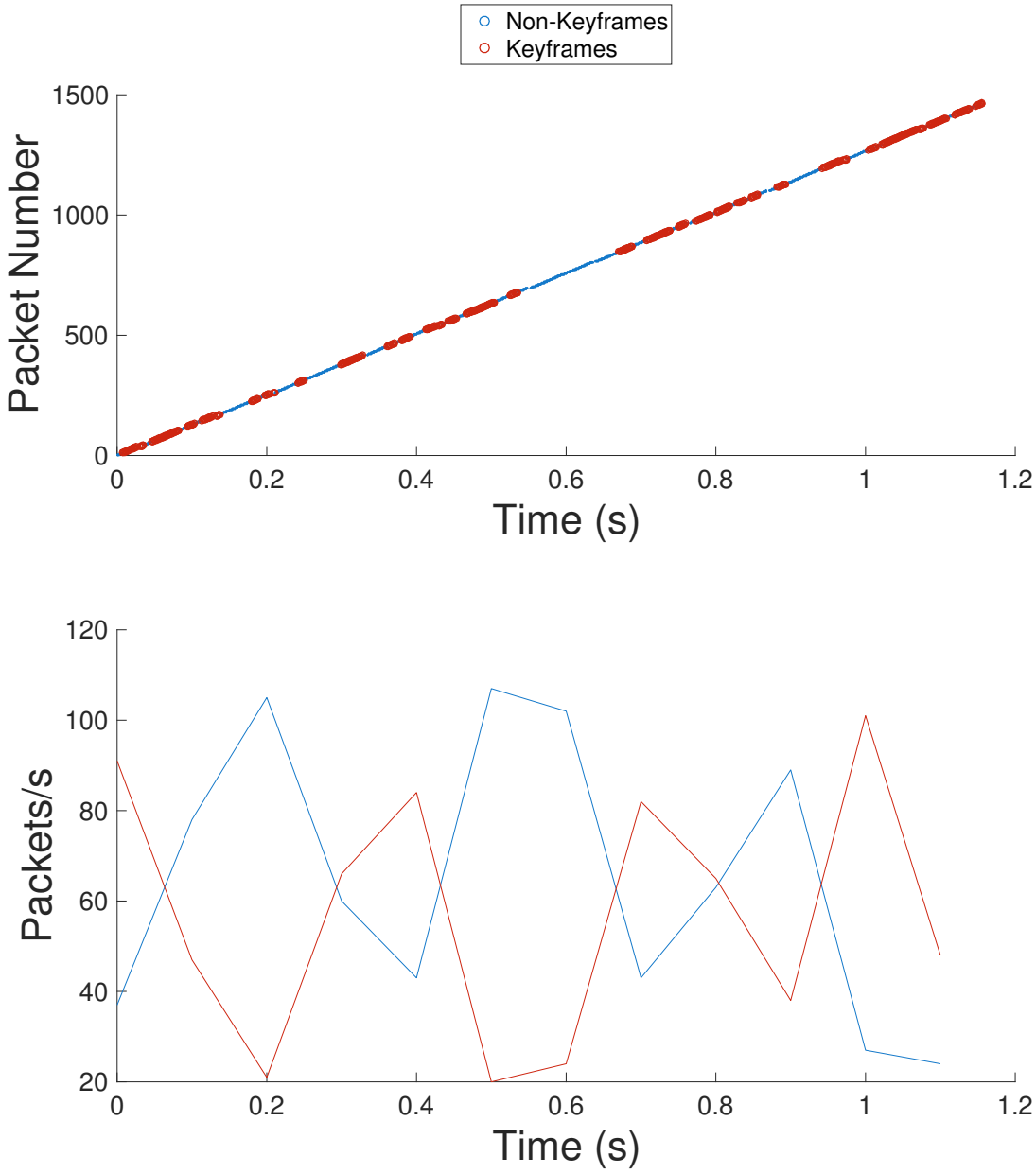
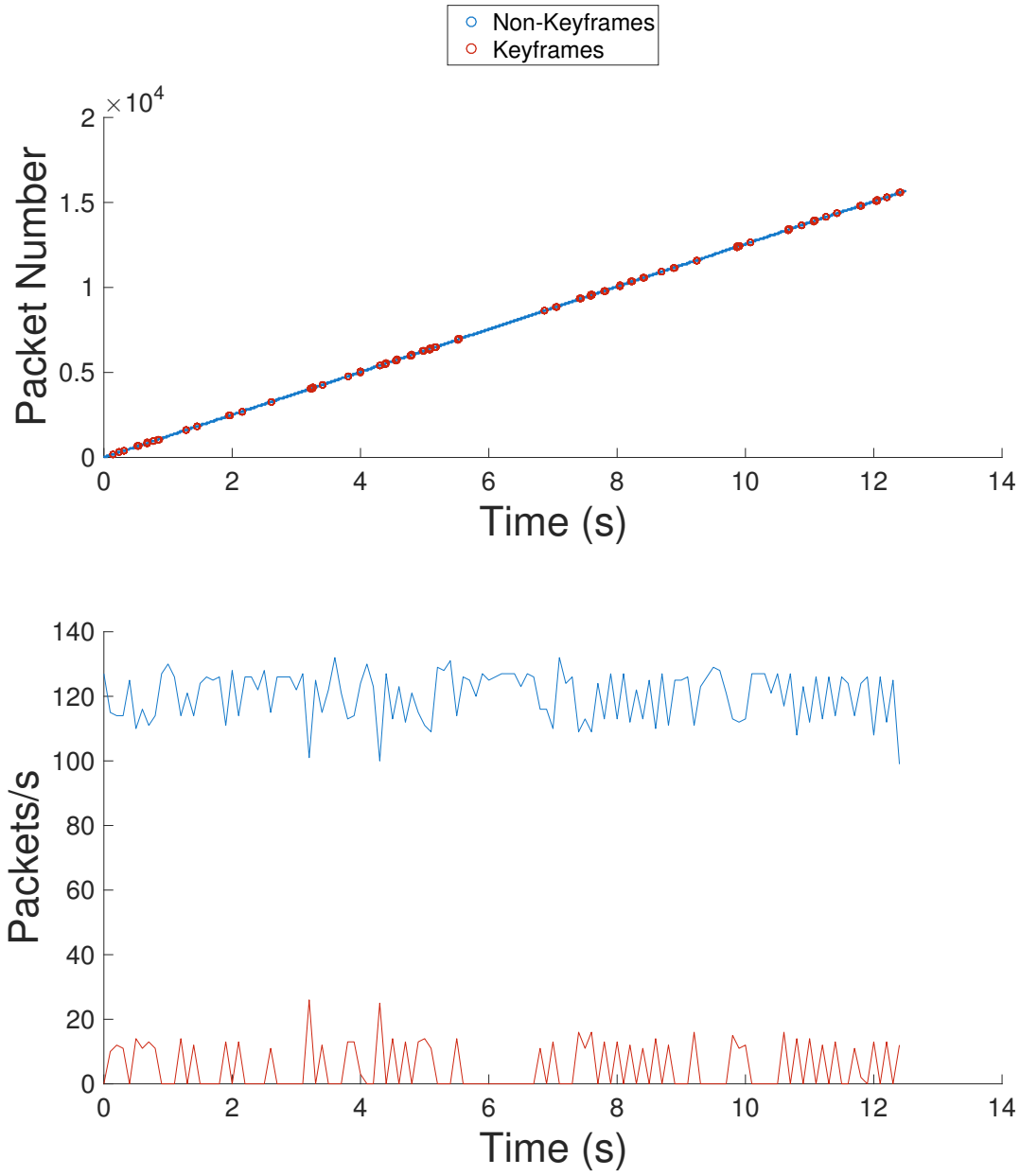


Figure B.12: Sample cross-traffic ; Medium Channel (Path A : 16 Mbps, Path B : 16 Mbps)



List of Figures

List of Tables

Bibliography

[1] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(212):263–297, 2000.

[2] Roberto Doriguzzi Corin, Roberto Riggio, Daniele Miorandi, and Elio Salvadori. AiroLAB: Leveraging on virtualization to introduce controlled experimentation in operational multi-hop wireless networks. *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST*, 46:331–344, 2011.

[3] A. Bianco, R. Birke, D. Bolognesi, J.M. Finochietto, G. Galante, M. Mellia, M.L.N.P.P. Prashant, and F. Neri. Click vs. Linux: two efficient open-source IP network stacks for software routers. *HPSR. 2005 Workshop on High Performance Switching and Routing, 2005.*, pages 18–23, 2005.

[4] Tiecheng Liu and Chekuri Choudary. Real-time content analysis and adaptive transmission of lecture videos for mobile applications. *ACM Multimedia 2004 - proceedings of the 12th ACM International Conference on Multimedia*, pages 400–403, 2004.

[5] Zhi-Li Zhang, Srihari Nelakuditi, Rahul Aggarwal, and Rose P Tsang. Efficient Selective Frame Discard Algorithms For Stored Video Delivery Across Resource Constrained Networks. *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, pages 472–479, 1999.