DISTRIBUTING FLOWS BETWEEN SDN CONTROLLERS

Jacek Litka, MSc Eng.

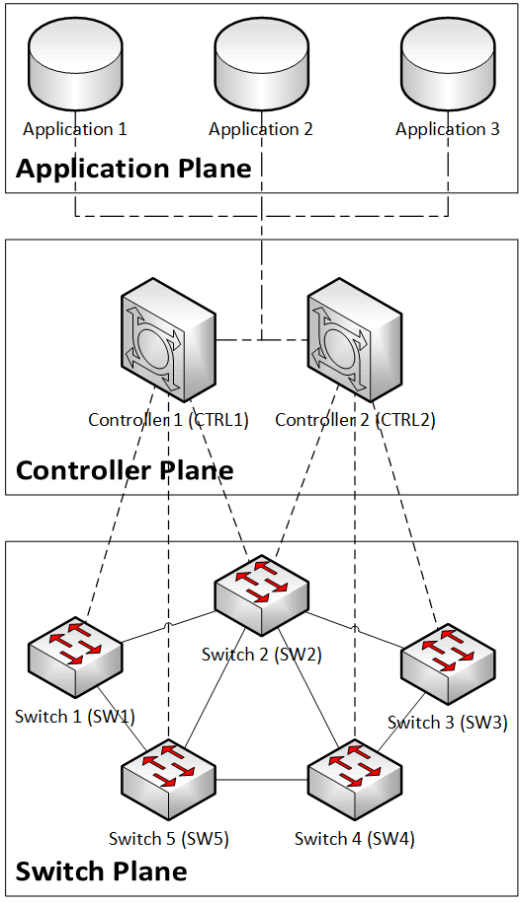
GDAŃSK UNIVERSITY OF TECHNOLOGY

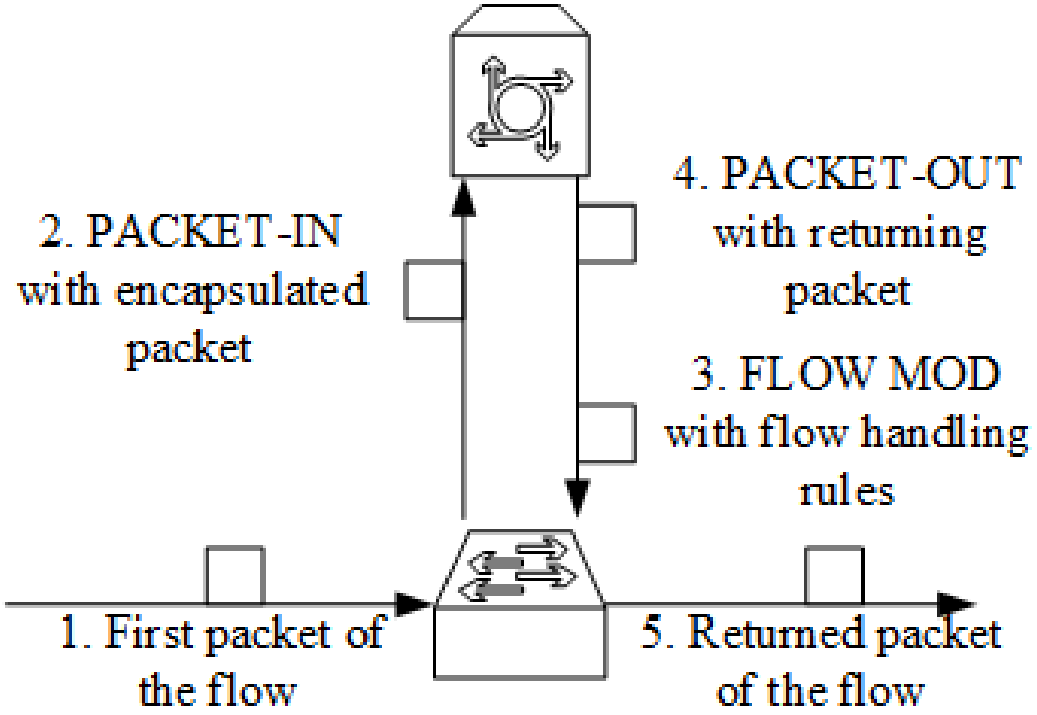
FACULTY OF ELECTRONICS, TELECOMMUNICATIONS AND INFORMATICS

AGENDA

1. Principles of Software Defined Networking (SDN)
2. SDN Performance
3. Distributing Flows, a Solution for SDN Performance
4. Current State. Are Vendors Ready?
5. Current Trend in Research. "Switching the Switches"
6. My Research. Switching Flow Handling
7. Conclusions and Further Works

1. PRINCIPLES OF SOFTWARE DEFINED NETWORKING (SDN)





2. SDN PERFORMANCE

SDN performance is not a well-defined problem – Even though recommendation documents exist, no specific guidelines have been provided.

Issues occur in both controller and switch layers – A number of them, many not correlated, disturb the efficiency of SDN architecture.

Interconnection between these two layers provides additional hurdles – Connecting controllers to switches in a production-type environment provides its own considerations.

2. SDN PERFORMANCE

Possible performance considerations in switch layer:

- switching time,
- flow table lookup time,
- flow table capacity limits,
- flow rule installation time.

Possible performance considerations in controller layer:

- **time of handling,**
- **intensity of requests,**
- buffer limits for incoming PACKET-IN messages,
- **topology reconstruction time.**

Possible performance considerations between switch and controller layers:

- limitations on switch-to-controller requests rate,
- physical distance between controllers and switches,
- placement of controllers in overall SDN network structure,
- possibility of switch not having completed the process of flow rule installation when a 2nd packet of that flows is received by it.

3. DISTRIBUTING FLOWS, A SOLUTION FOR SDN PERFORMANCE

1. Increase of controller's workload leads to increase of PACKET-IN handle time.
2. Simple solution? Decrease the workload!
3. However, we do not have the ability to change the volume of traffic in the network.
4. Complex solution? Distribute the workload!

3. DISTRIBUTING FLOWS, A SOLUTION FOR SDN PERFORMANCE

Use case #1:

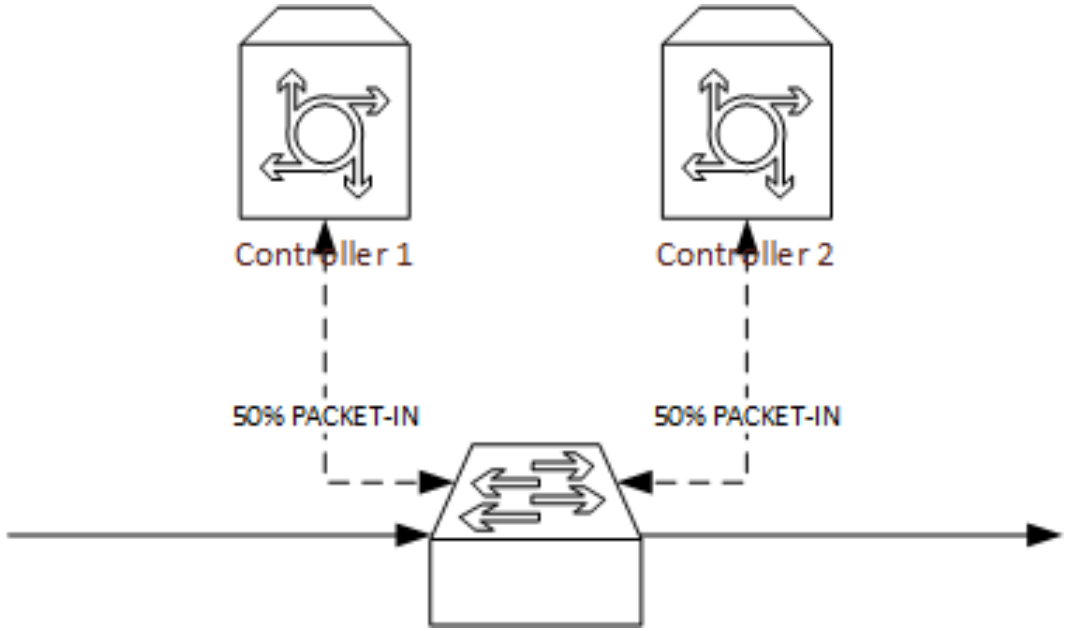
Distribute the workload equally

Benefit:

- Decreases workload of a single controller.

Drawback:

- With a small overall workload, there is a waste of resources.



3. DISTRIBUTING FLOWS, A SOLUTION FOR SDN PERFORMANCE

Use case #2:

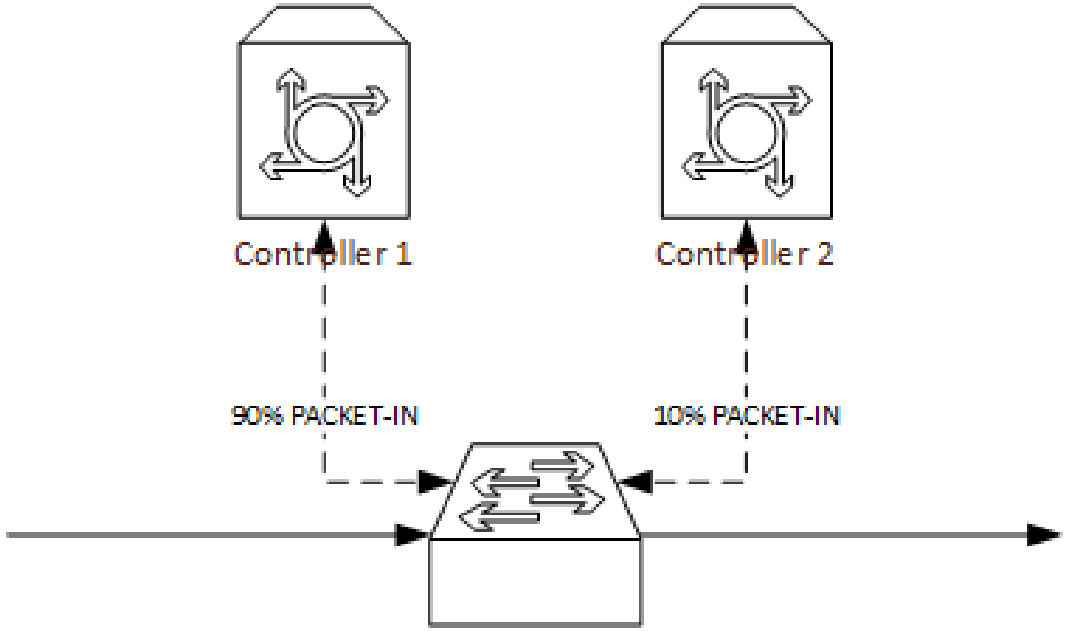
Offload some of the workload

Benefit:

- Decreases workload of the „main” controller.

Drawback:

- How big of an „offload” should there be?



3. DISTRIBUTING FLOWS, A SOLUTION FOR SDN PERFORMANCE

Use case #3:

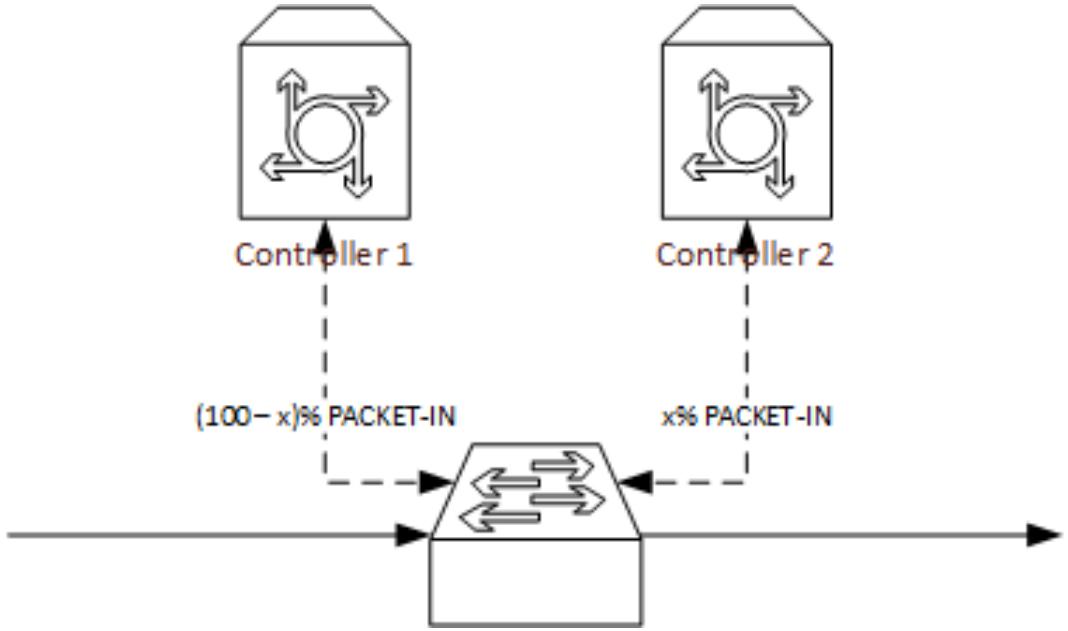
Dynamically change the offload value

Benefits:

- Decreases workload of the „main” controller.
- Mitigating unnecessary waste of resources.

Drawback:

- Requires an algorithm for calculating the offload value.



4. CURRENT STATE. ARE VENDORS READY?

Yes. **IN THEORY**.

- OpenFlow switch may work with more than one controller by assigning appropriate roles:
  - MASTER,
  - EQUAL,
  - SLAVE.
- ITU-T recommendation defines an EAST-WEST INTERFACE for communication between controllers.
- OpenFlow since version 1.3 defines „auxiliary connections” for distributing protocol messages (including PACKET-IN) between controllers.

4. CURRENT STATE. ARE VENDORS READY?

No. IN PRACTICE.

- Controller states do not work well in distributing workload scenarios.
  - MASTER – There can be only one, each of the other controllers has to work in SLAVE role. MASTER takes on all PACKET-IN messages.
  - SLAVE – Does not know the topology of the network. Is unable to design flow rules.
  - EQUAL – Every EQUAL controller receives every asynchronous message from a switch; therefore, a PACKET-IN message is generated to all controllers.
- East-West Interface is not standardized; software implementations sometimes DO NOT WORK.
- Open vSwitch, the reference open source OpenFlow switch, DOES NOT implement auxiliary connections.

4. CURRENT STATE. ARE VENDORS READY?

ORIGINAL CODE FOR CONTROLLERS' COOPERATION IN FLOODLIGHT SDN CONTROLLER.

```
29      private static final int READ_BUF_SIZE = 1024;
30      private Integer sendTO;
31      private Integer linger;
32      private SocketChannel sc;
33
34      /**
35       * Constructor should take all standard params required, like connection
36       * timeout, SO_LINGER etc.
37       */
38
39      public NioClient(Integer sndTimeOut, Integer linger) {
40          sendTO = sndTimeOut;
41          this.linger = linger;
42
43      }
44
45      public SocketChannel connectClient(String host) {
46          Integer port = Integer.valueOf(host.substring(10));
47          String host2 = host.substring(0, 9);
48
49          InetSocketAddress inet = new InetSocketAddress(host2, port);
50          try {
51              sc = SocketChannel.open(inet);
52              sc.socket().setSoTimeout(sendTO);
53              sc.socket().setTcpNoDelay(false);
54              sc.socket().setSoLinger(false, linger);
55              sc.socket().setReuseAddress(true);
56              sc.socket().setPerformancePreferences(1, 2, 0);
57              return sc;
58          } catch (Exception e) {
59              return null;
60          }
61      }
```

4. CURRENT STATE. ARE VENDORS READY?

MAKING IT ACTUALLY WORK.

```
29      private static final int READ_BUF_SIZE = 1024;
30      private Integer sendTO;
31      private Integer linger;
32      private SocketChannel sc;
33
34      /**
35       * Constructor should take all standard params required, like connection
36       * timeout, SO_LINGER etc.
37       */
38
39      public NioClient(Integer sndTimeOut, Integer linger) {
40          sendTO = sndTimeOut;
41          this.linger = linger;
42
43      }
44
45      public SocketChannel connectClient(String host) {
46          Integer delimiter = host.indexOf(':');
47          Integer port = Integer.valueOf(host.substring(delimiter + 1));
48          String host2 = host.substring(0, delimiter);
49
50          InetSocketAddress inet = new InetSocketAddress(host2, port);
51          try {
52              sc = SocketChannel.open(inet);
53              sc.socket().setSoTimeout(sendTO);
54              sc.socket().setTcpNoDelay(false);
55              sc.socket().setSoLinger(false, linger);
56              sc.socket().setReuseAddress(true);
57              sc.socket().setPerformancePreferences(1, 2, 0);
58              return sc;
59          } catch (Exception e) {
60              return null;
61          }
```

4. CURRENT STATE. ARE VENDORS READY?

OPEN VSWITCH STATEMENT ON AUXILIARY CONNECTIONS.

OpenFlow 1.3

OpenFlow 1.3 support requires OpenFlow 1.2 as a prerequisite, plus the following additional work. (This is based on the change log at the end of the OF1.3 spec, reusing most of the section titles directly. I didn't compare the specs carefully yet.)

- IPv6 extension header handling support.

  Fully implementing this requires kernel support. This likely will take some careful and probably time-consuming design work. The actual coding, once that is all done, is probably 2 or 3 days work.

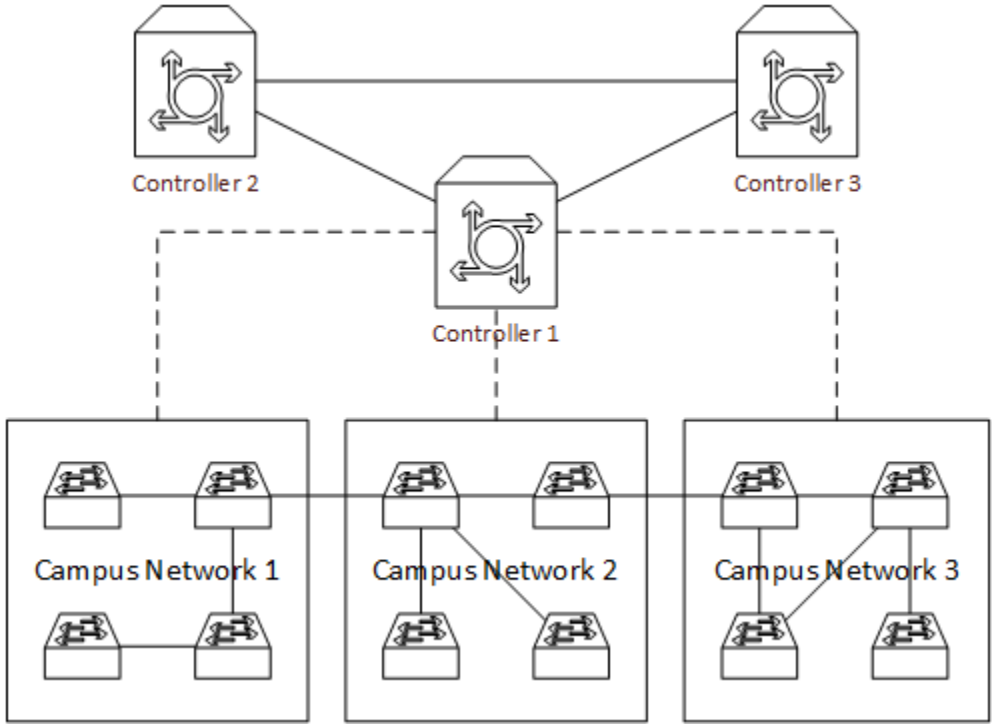
  (optional for OF1.3+)

- Auxiliary connections.

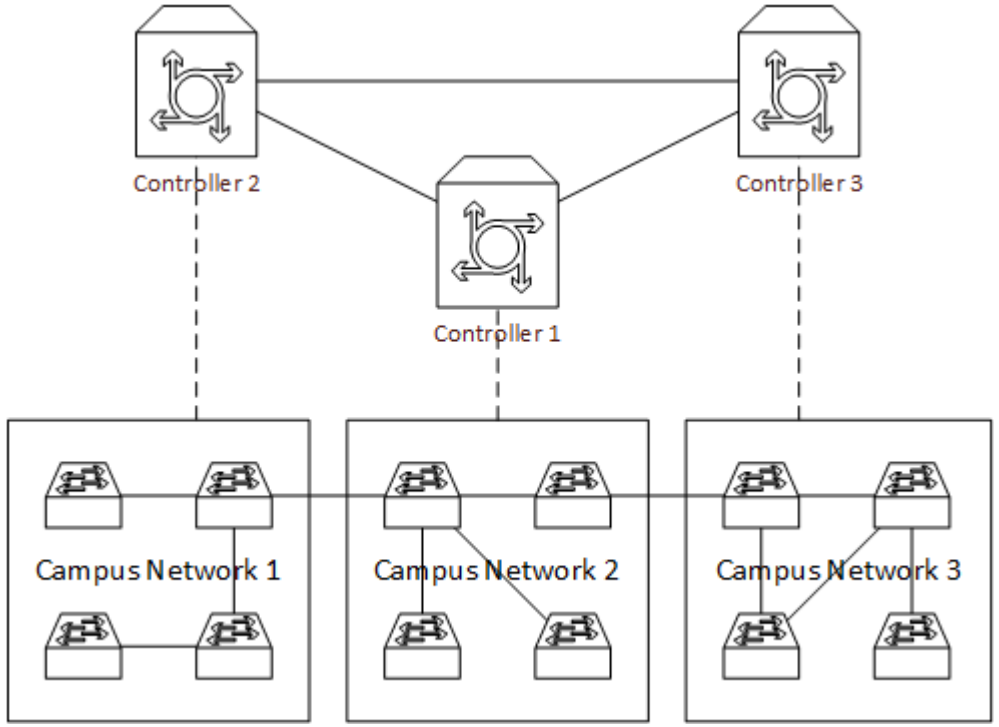
  An implementation in generic code might be a week's worth of work. The value of an implementation in generic code is questionable, though, since much of the benefit of axuiliary connections is supposed to be to take advantage of hardware support. (We could make the kernel module somehow send packets across the auxiliary connections directly, for some kind of "hardware" support, if we judged it useful enough.)

5. CURRENT TREND IN RESEARCH. "SWITCHING THE SWITCHES"

State 1: One controller handles multiple networks

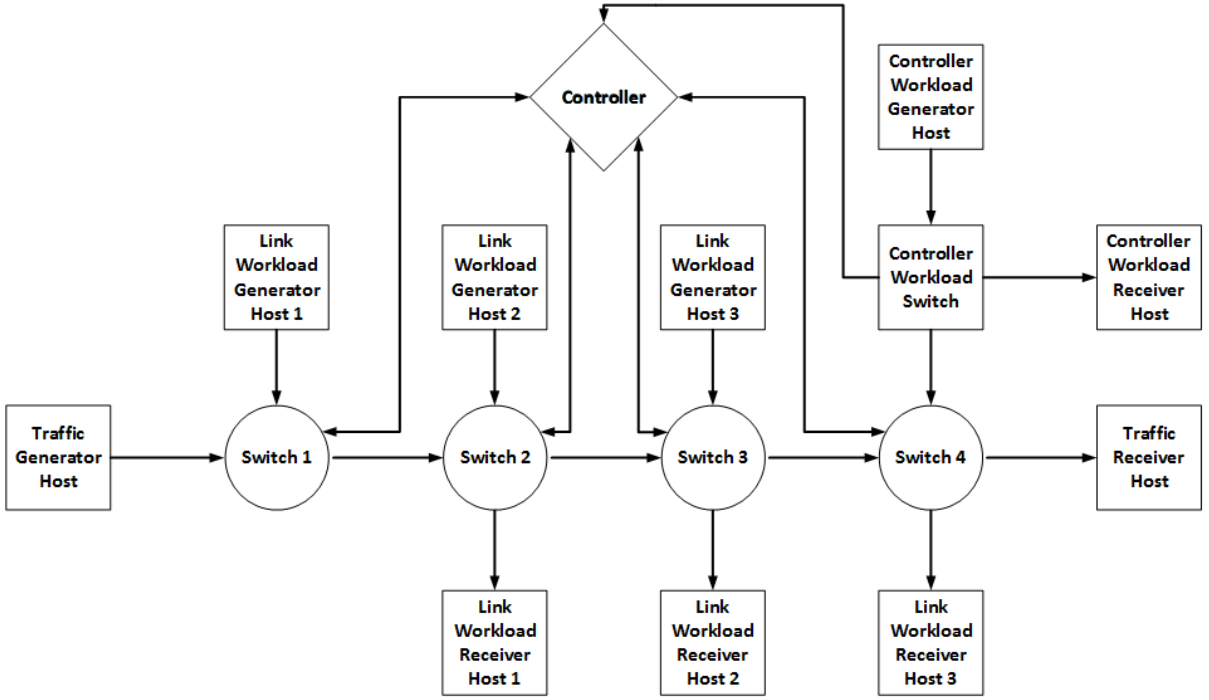


State 2: Networks are distributed among multiple controllers

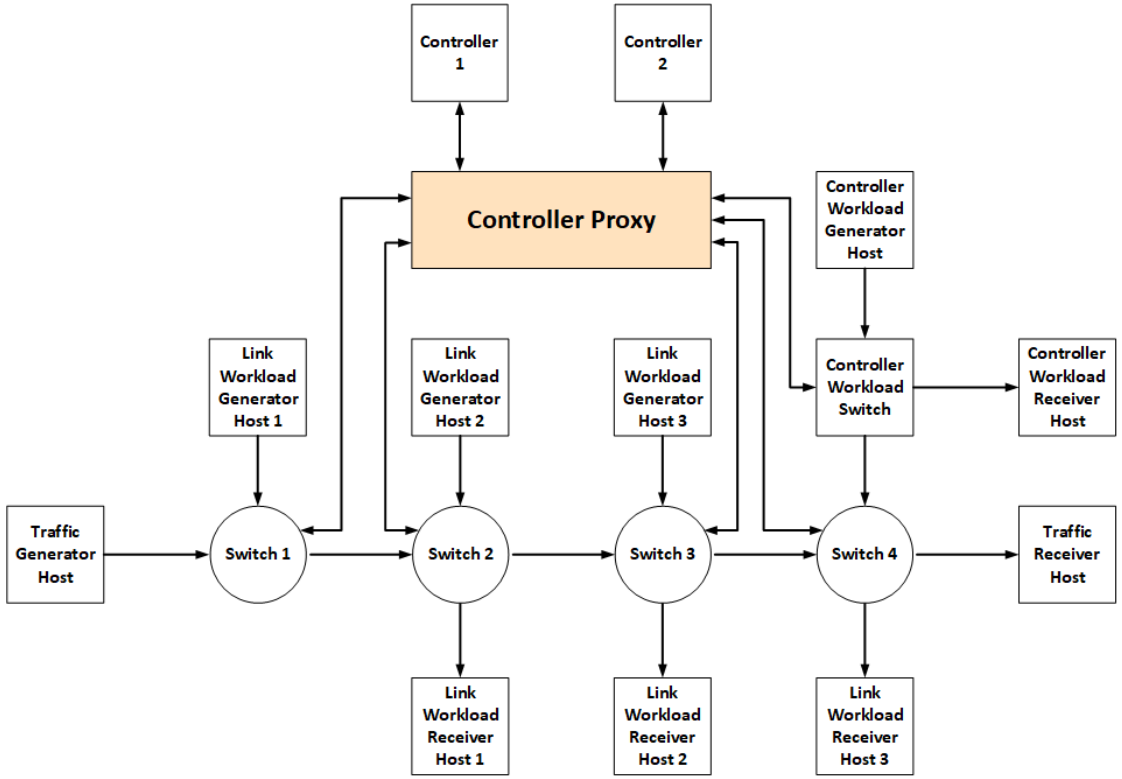


6. MY RESEARCH. SWITCHING FLOW HANDLING

Emulation environment from 2024

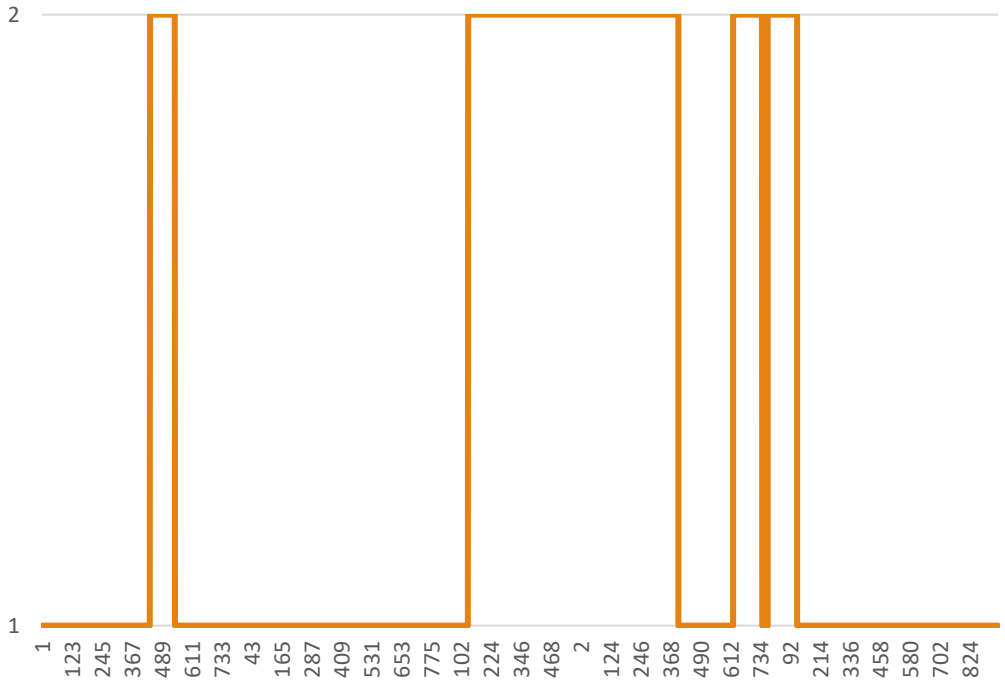


Emulation environment from 2025

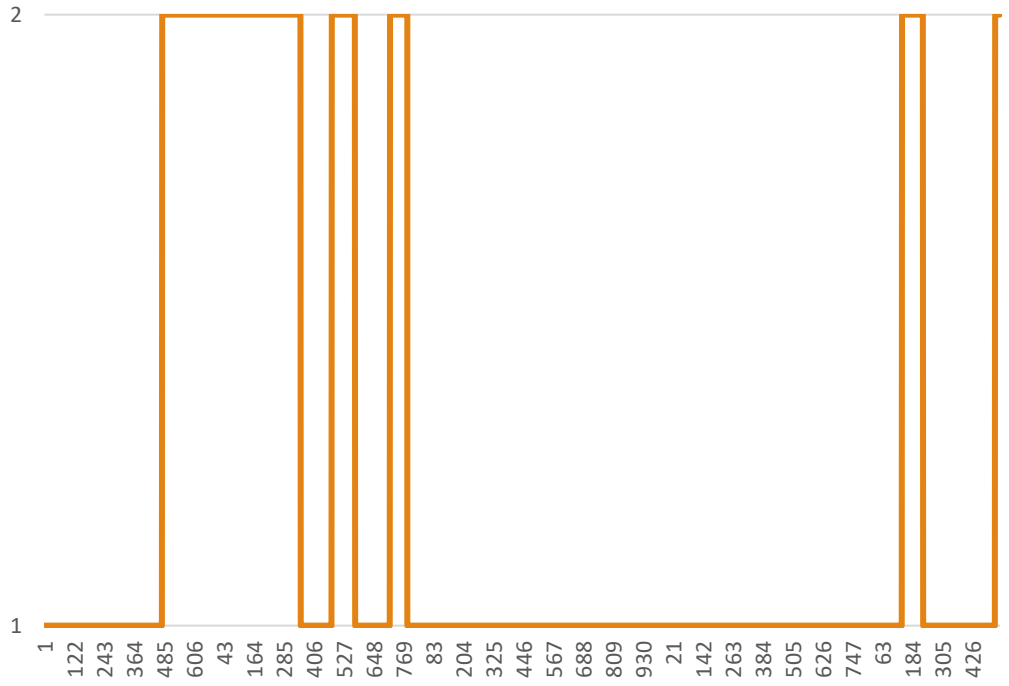


6. MY RESEARCH. SWITCHING FLOW HANDLING

PACKET-IN requests per second increase, then decrease (0/500/1250/750/250)



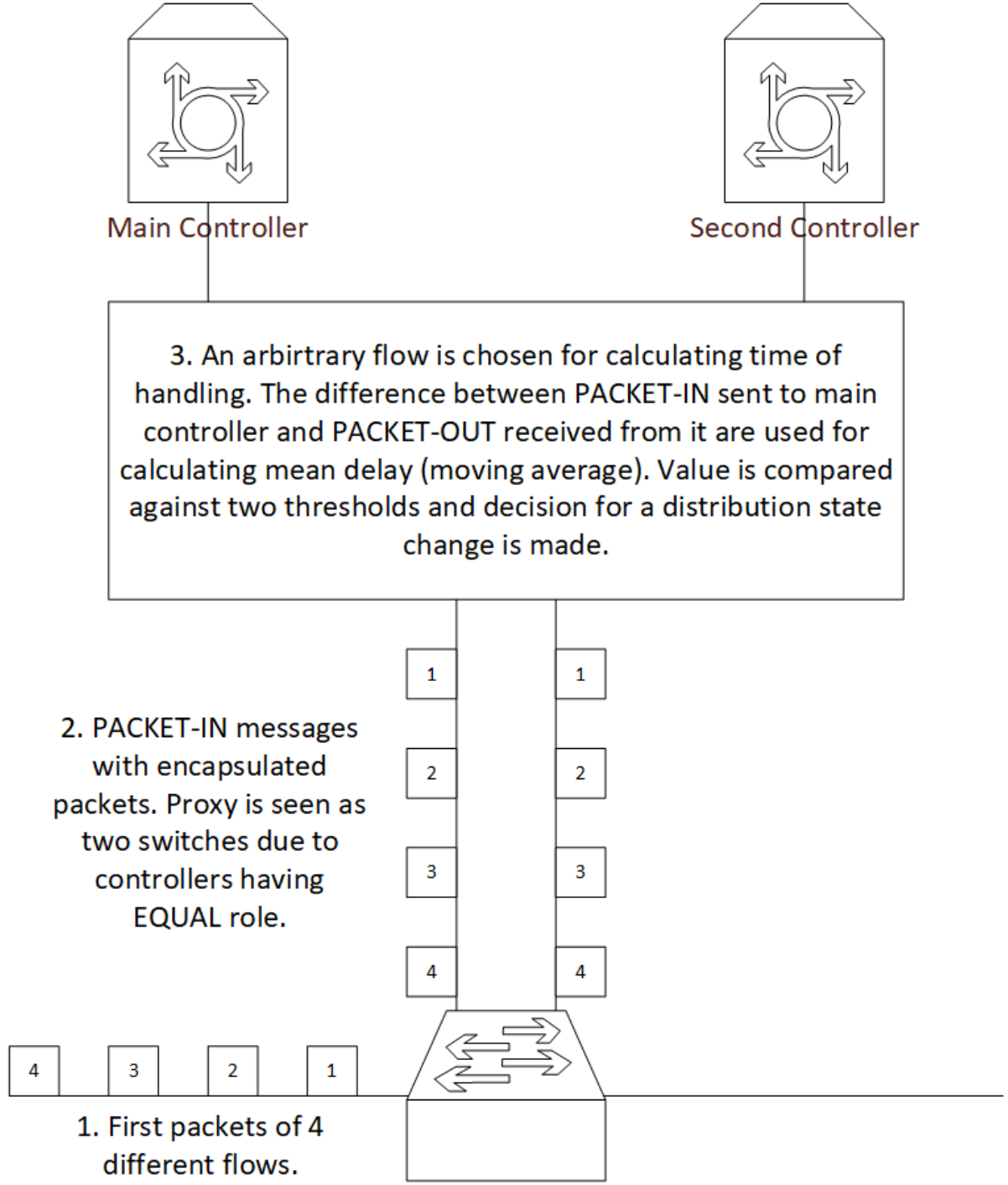
PACKET-IN requests per second decrease, then increase (1250/500/0/750/1000)



6. MY RESEARCH. SWITCHING FLOW HANDLING

My proposal for solving the issue – OpenFlow proxy:

1. Proxy works as a mediary between switch and controllers.
2. Both controllers connecting to proxy work in EQUAL role.
3. Proxy is seen as two controllers – it binds to two IP addresses, both are provided to the switch.
4. Switch will always send a pair of asynchronous messages (due to EQUAL roles), which proxy relays to both controllers.
5. Both controllers have up-to-date knowledge of the network structure without the need of East-West Interface.
6. In case of PACKET-IN messages proxy relays the request to a single controller.

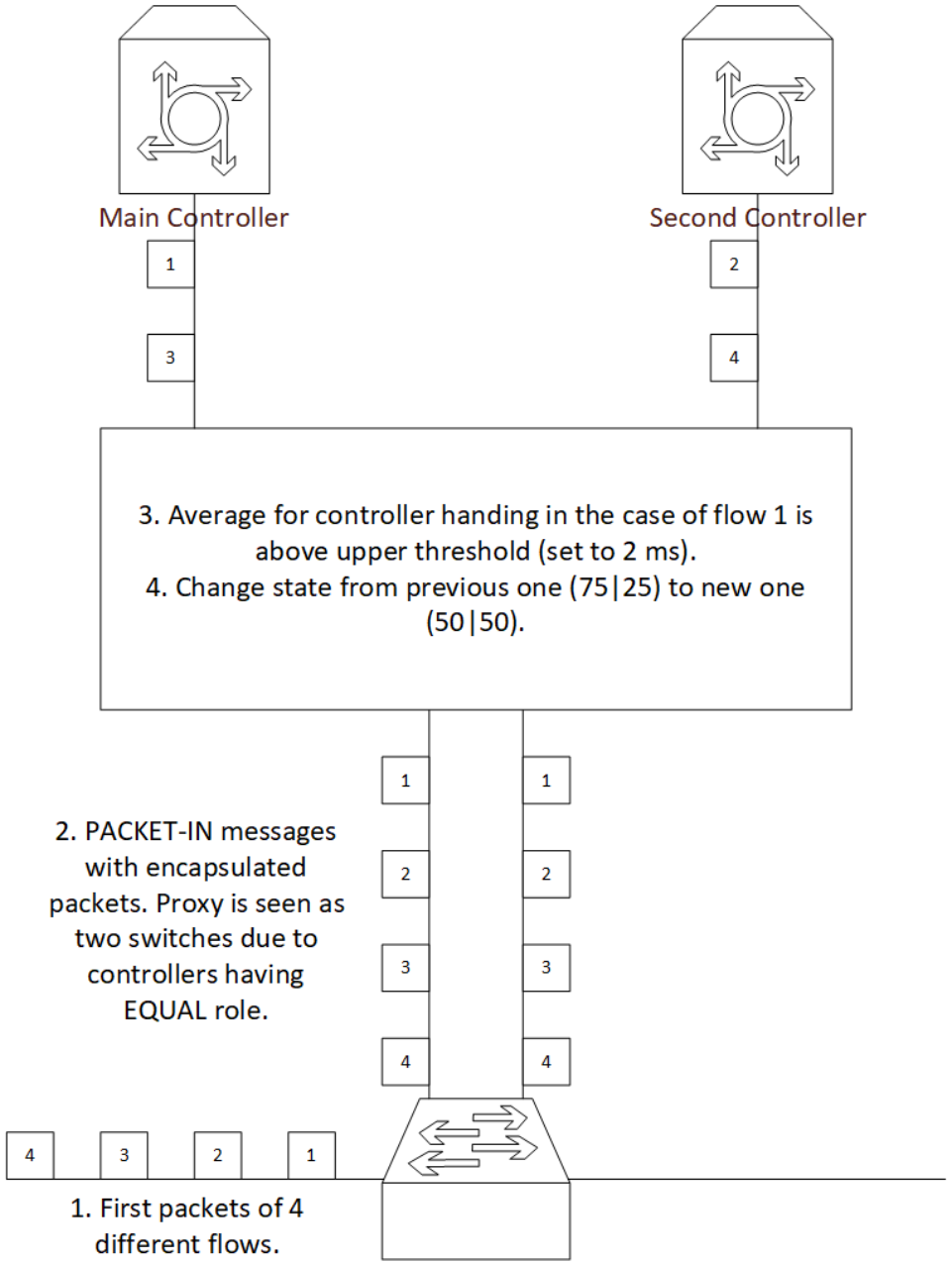

Main Controller
Second Controller
3. An arbirtrary flow is chosen for calculating time of handling. The difference between PACKET-IN sent to main controller and PACKET-OUT received from it are used for calculating mean delay (moving average). Value is compared against two thresholds and decision for a distribution state change is made.
2. PACKET-IN messages with encapsulated packets. Proxy is seen as two switches due to controllers having EQUAL role.
1
2
3
4
1
2
3
4
4
3
2
1
1. First packets of 4 different flows.

6. MY RESEARCH. SWITCHING FLOW HANDLING

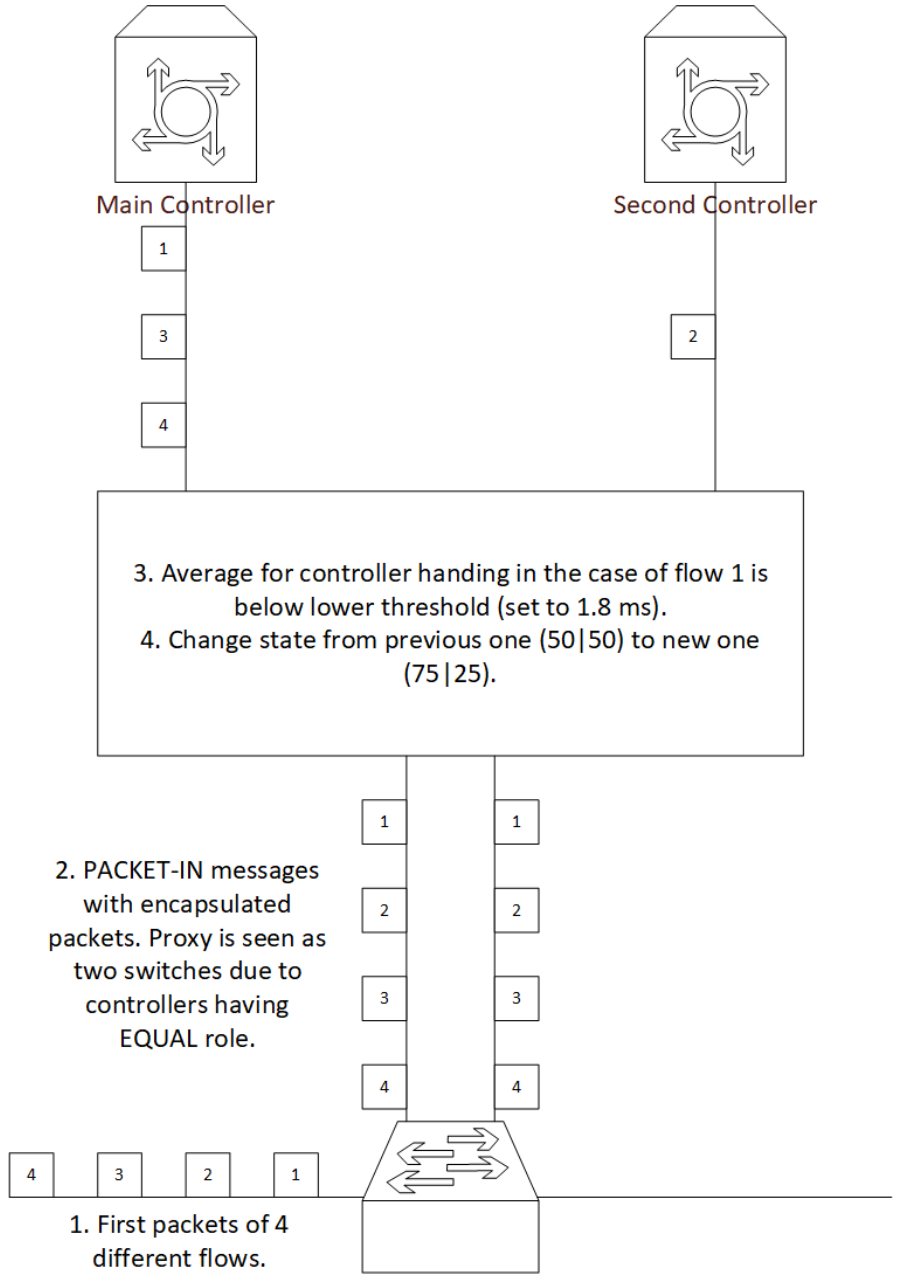
Example possible states:

- 100% | 0% - this corresponds to situations in which all of the requests are handled by the first controller, the default startup state,
- 93.75% | 6.25% - approximately 93.75% of the requests are handled by the first controller, rest is offloaded to the second,
- 87.5% | 12.5% - approximately 87.5% of the requests are handled by the first controller, rest is offloaded to the second,
- 75% | 25% - approximately 75% of the requests are handled by the first controller, rest is offloaded to the second,
- 50 % | 50% - the requests are handled almost equally between the controllers.

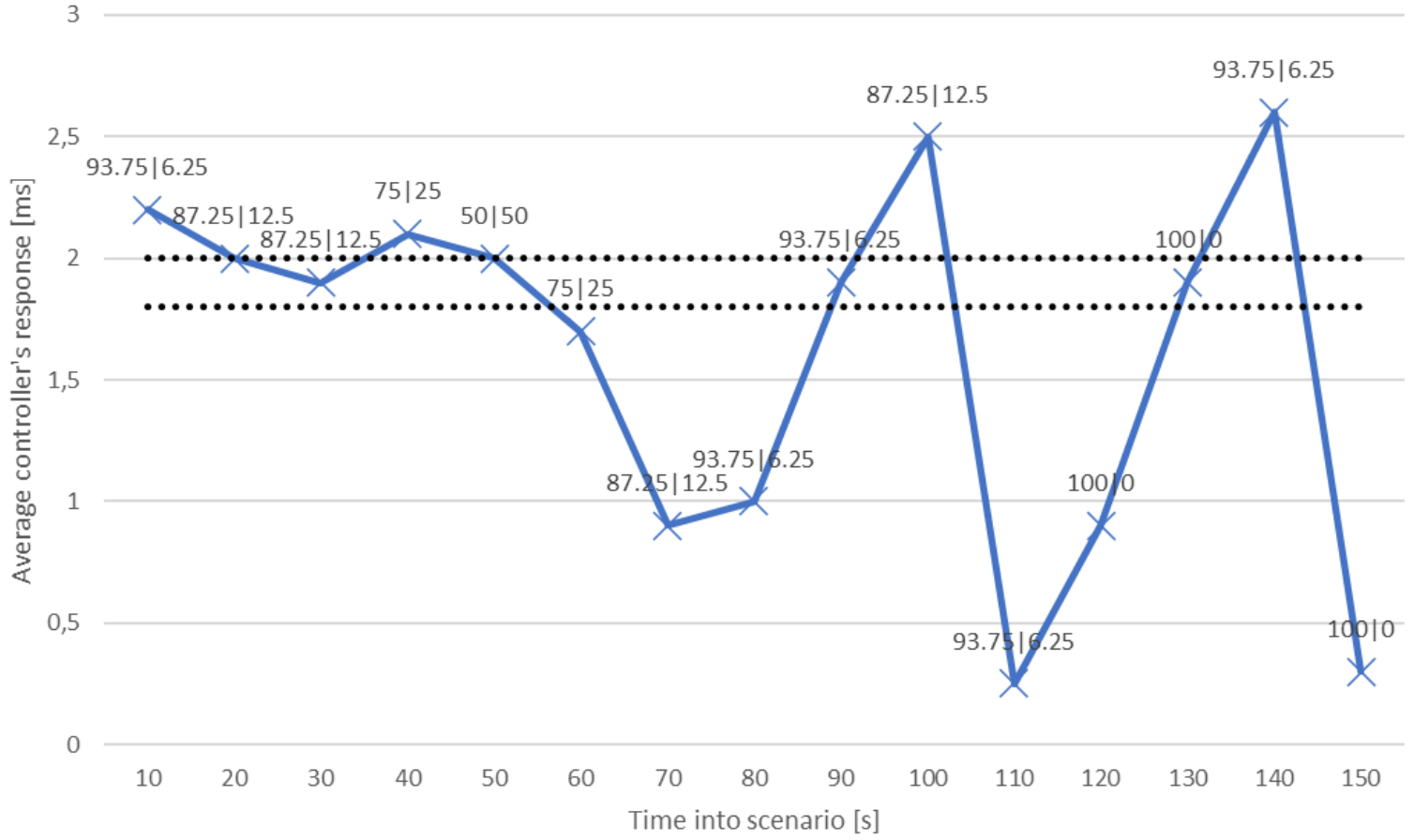
STATE #1: AVERAGE OF CONTROLLER'S HANDLING INCREASES



STATE #2: AVERAGE OF CONTROLLER'S HANDLING DECREASES



6. MY RESEARCH. SWITCHING FLOW HANDLING



6. MY RESEARCH. SWITCHING FLOW HANDLING

An interesting question arises the moment proxy is working in the system…

How much of a delay the proxy adds by itself?

- Attempt #1 (failure)
  - Proxy coded in Python.
  - Mean proxy delay: **120 ms.**
- Attempt #2 (failure)
  - Proxy coded in C++.
  - Mean proxy delay: **10 ms**.
- Attempt #3 (**success!**)
  - Proxy coded in C++.
  - Nagle's algorithm turned off in proxy.
  - Mean proxy delay: **110 µs**.

7. CONCLUSIONS AND FURTHER WORKS

CONCLUSIONS:

- Efficiency of the controller itself has a great impact on overall performance of the SDN network.
- Controller's performance issue should be solvable by distributing the workflow between a number of controllers.
- Research community focus on distributing control over switches between the controllers, not the flows themselves.
- An OpenFlow proxy is a method for distributing flows between controllers.
- **OpenFlow proxy requires a mechanism to dynamically change the distribution to mitigate resource waste. Currently, an easier, discreet one is implemented.**

FURTHER WORKS:

- Repeating last year scenarios in new environment with proxy and comparing the results.
- Making the proxy be seen as a single controller, instead of a pair of them.
- Changing the distribution mechanism to continuous one.

Thank you, for your attention!