UNIVERSITÀ DI PISA

Computer Engineering

Advanced Network Architectures and Wireless Systems

Distributed Message Broker

Project Documentation

Clarissa Polidori
Leonardo Poggiani
Bruno Augusto Casu Pereira De Sousa

Academic Year: 2021/2022

Table of Contents

1 | Introduction

The Distributed Message Broker project aims to present an implementation of a message distribution system, based in a Software Defined Network environment. The Broker must handle incoming messages from multiple computing nodes connected to the network, in a way that nodes subscribed to a Resource in the system will be able to receive messages published on that specific Resource (topic). The proposed system will then be divided in Computing Nodes (either servers or hosts), the Resources (Virtual Addresses managed by the SDN controller) and a User (acting as a system administrator). Figure 1 shows an overview of the system organization:

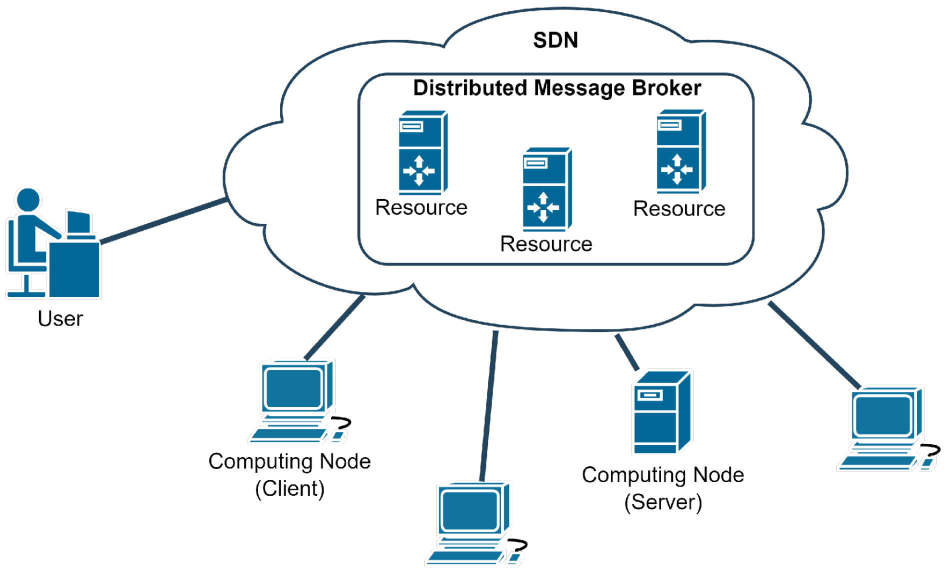


Figure 1.1: Overview of the Distributed Message Broker system

As a way to allow any computing node to publish messages on the Broker system, the Resources of the platform will be deployed as Virtual IP Addresses managed by an OpenFlow based SDN Controller. The mentioned Resources will be created or removed by a remote user, acting as a system manager. The User will also manage the subscriptions of the computing nodes that wish to receive messages published on those resources. The commands issued by the user will be sent to a REST interface exposed in the SDN Controller, allowing the management of the broker system and subscriptions. As the broker system is made stable and operational inside the SDN, a set of tests will be performed by creating a virtual network using the mininet application. This virtual network will then be used to execute a simulated environment with many hosts and resources in different network topologies, allowing the demonstration of the broker functionalities and features.

2 | Design

2.1 REST interface

In order to manage the Distributed Message Broker, a RESTful interface will be exposed to the user (system administrator), providing a set of endpoints that can be used to retrieve network information, issue configuration changes for the service and manage subscriptions; they are reachable at the following URLs:

- `http://CONTROLLER_IP:8080/db/subscribers/RESOURCE_IP/json`
- `http://CONTROLLER_IP:8080/db/resources/json`

2.1.1 Endpoint "resources"

This endpoint permits to the administrator to manage resources, allows to create a new resource, delete it or get the list of them. A resource is represented by its virtual address and an id.

- **GET**
  Retrieves the list of registered resources. The list is given in the format "IpAddress":"Id"
- **POST**
  Creates a resource assigning to it a virtual address. The module checks if the resource already exists, i.e if the virtual address is already present in the list.
- **DELETE**
  Removes a resource from the registered resources. The resource is submitted in JSON format specifying the parameter "resource". The module checks if the resource already exists.

2.1.2 Endpoint "subscribers"

This endpoint permits to the administrator to manage the users requesting the subscription to a resource. A subscriber is represented by its MAC address and its IP address.

- **GET**
  Retrieves the list of users subscribed to a resource. The list is given in the format "MACAddress":"IPAddress".
- **POST**
  Adds an user to the list of subscribed users of a resource. The user is submitted in JSON format specifying the parameters "IPAddress" and "MACAddress". The module checks if the user is already subscribed, i.e. if the MAC address is already present in the list.

- **DELETE**
  Removes an user from the list of subscribed users to a resource. The user is submitted in JSON format specifying the parameter "MAC". The module checks if the user is subscribed.

2.2 Packet filtering

To guarantee the correct functioning of our network it is necessary to adopt a policy of filtering unnecessary packets. In fact there are some packets that are not only useless for the functioning of the **distributed message broker** but that can also make it take on undesired behaviors.

In particular in our network we don't want to be possible that:

- A user send a message directly to another user. Our goal is to guarantee the communication only through the **distributed message broker** so we must filter every packet that has as destination address an address different from the one belonging to a resource.
- A user receive a message from a resource to which it is not subscribed.
- A user publish messages on a resource to which he is subscribed.

These are the conditions necessary to ensure that only hosts subscribed to a given resource can see messages posted to that resource (by hosts which are not subscribed themselves) while others do not see any incoming traffic.

In order to guarantee the expected operation of the **distributed message broker** we have implemented a packet filtering function in the controller. In this way it is possible to filter packets that are not directed to resources except ARP packets (which are forwarded in broadcast). For this reason filtering is partially implemented in the controller and partially in the forwarding module. In the forwarding module we drop and interrupt the pipeline of those ARP requests originating from messages that have a host as source and another host as destination. In fact the *forwarding* module runs before our module, because we need it to retrieve information needed for packet routing such as the *outports* inside the *attachmentPoints*.

The forwarding module has a default behavior which is to search in the network the device to which it should send the incoming message. If this device is found a *forwarding* is performed, if it is not found a *flood* is performed. The default behavior of the *forwarding* module allowed two hosts to communicate with each other because by flooding the ARP requests/responses they were able to resolve the physical address of the recipient and thus be able to forward without necessarily having to ask the controller.

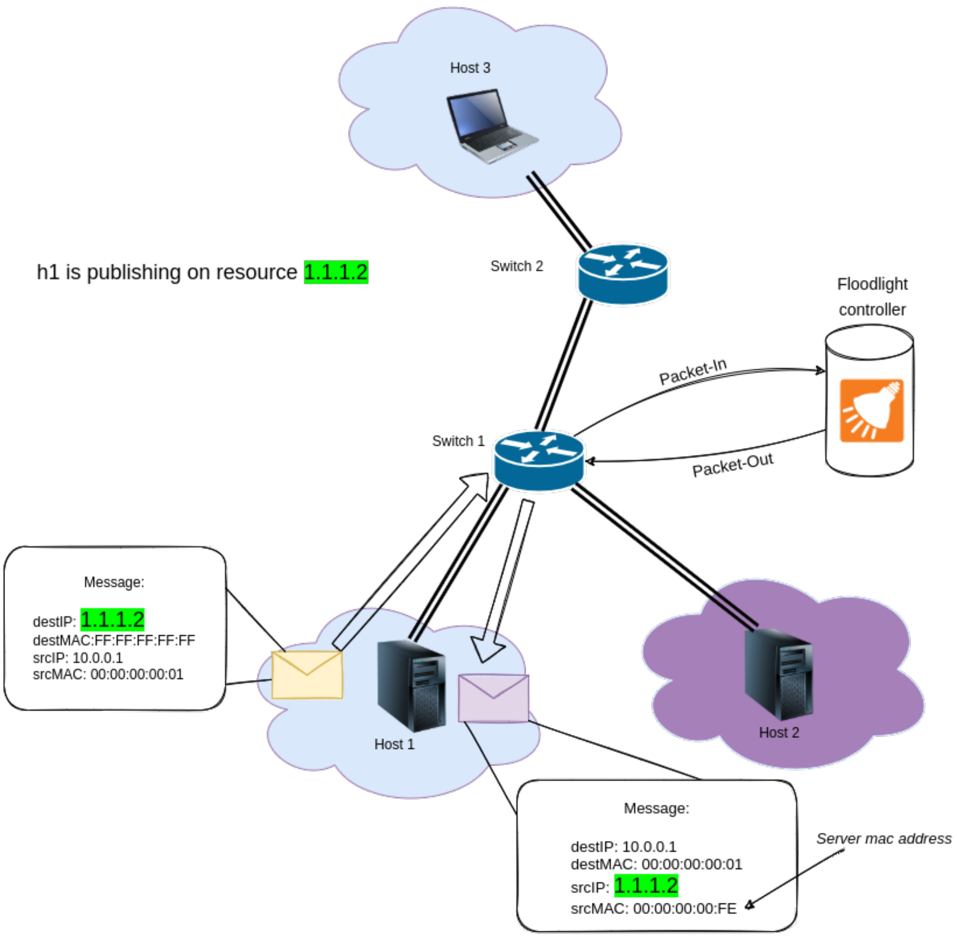
This wrong behavior for our specific application has been corrected by intervening on the module of forwarding and in particular by avoiding to flood all messages that have as destination address an address that is not present in the list of subscribers.

2.3 Arp requests

After passing the filtering we assume that only authorized packets can transit inside the network. The only requests that can arrive are those that have the destination address of a virtual resource. For this reason they are destined to fail, because there is no device inside the network with this address. The solution is to manually create an ARP reply packet for these requests with these values:

- source address: virtual address of the resource
- destination address: source address of the ARP request
- source MAC address: always *00:00:00:00:FE*, the virtual server address common to all resources
- destination MAC address: source MAC address of the ARP request

If the destination address belongs to the set of virtual resource addresses, an ARP response is created and forwarded on the port on which the request was received by means of a *packet-out*.



Message:
destIP: 1.1.1.2
destMAC:FF:FF:FF:FF:FF
srcIP: 10.0.0.1
srcMAC: 00:00:00:00:01

Message:
destIP: 10.0.0.1
destMAC: 00:00:00:00:01
srcIP: 1.1.1.2
srcMAC: 00:00:00:00:FE

Figure 2.1: ARP request and reply

2.4 IPv4 packets

IP communication within our domain is based on the idea that the virtualization of the resource is managed at the network access layer: the translation from the virtual address of the resource to the physical addresses of the hosts during the packet exchange is managed by the first switch encountered by the message during its path.

In this way only the first switch will require the translation of the virtual address of the resource to the controller, while the other switches will only be responsible for routing the packet to the correct output port and delivering it to the *subscriber* using the *forwarding* module.

Receiving a single packet from the *publisher* and having to forward it to **n** *subscribers*, it is necessary to perform **n** output actions that allow to send **n** copies of the same message. Each of these copies will have the following fields:

- **source MAC address:** server MAC address, common to all virtual resources (*00:00:00:00:00:FE*)
- **source IP address:** virtual IP address of the resource (e.g. *1.1.1.1*)
- **recipient MAC address:** The MAC address of one of the resource's subscribers, maintained in a structure within the controller.
- **recipient IP address:** IP address of one of the resource's subscribers, maintained in a structure within the controller.

2.4.1 Flow-mods

In our case, inserting *flow-mods* in the switches to route packets automatically, without the need for the controller to intervene every time, was a solution that was hardly feasible because it was necessary to find a way to send a copy of the incoming message to the switch on **n** ports, where **n** is not necessarily one or all of the ports. Flow-mods are useful when it's possible to route the outgoing packet to only one port or if it's possible to *flood*.

Although this is not our case, we could have pursued the path of the *flow-mods* through the study and use of the *Openflow Groups* but this solution involved an additional complication to our architecture (it is necessary to create a group with inside it the lists of actions to be performed, that vary according to the resource on which we are publishing) going to distract us further from the final objective, that is to create a network in which we could communicate through the publication of messages on virtual resources in a simple and reliable way.

Furthermore, the mechanism of the *flow-mods* would have led to other technical difficulties, for example making possible to continue receiving messages for a certain period of time from a resource from which one had been removed by the network administrator if all the other flow-mods had not been flushed. It would also have been impossible to receive the most recent messages from a resource you had just subscribed to for a certain period of time. This is because the *flow-mods* expire after a certain period, which can be calculated to mitigate these problems, but they would still remain.

2.4.2 Design choice

For these reasons we chose an implementation that constantly relies on the controller to check the list of subscribers and send the **n** copies of the incoming message. Our architecture is therefore based entirely on *packet-in* and *packet-out*, not installing *flow-mods* inside the switches but always requiring the intervention of the controller for packets that do not know how to handle.

Given these considerations, within our network we will have only two types of packets that can freely circulate without being filtered:

- **ARP packets** addressed to virtual resources
- **UDP packets** addressed to physical hosts from the controller.

2.4.3 UDP packets

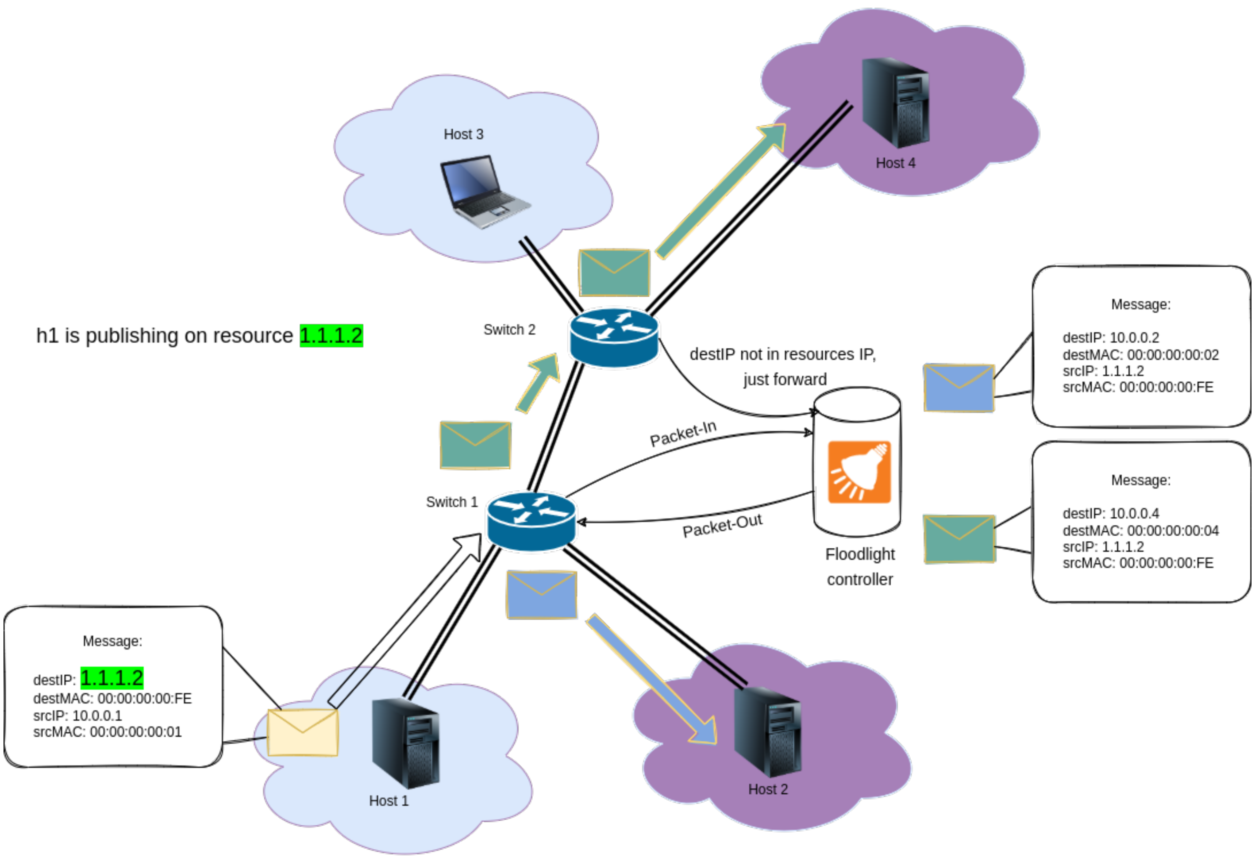

Host 3
Host 4
h1 is publishing on resource 1.1.1.2
Switch 2
destIP not in resources IP, just forward
Message:
destIP: 10.0.0.2
destMAC: 00:00:00:00:00:02
srcIP: 1.1.1.2
srcMAC: 00:00:00:00:00:FE
Packet-In
Switch 1
Packet-Out
Message:
destIP: 10.0.0.4
destMAC: 00:00:00:00:00:04
srcIP: 1.1.1.2
srcMAC: 00:00:00:00:00:FE
Floodlight controller
Message:
destIP: 1.1.1.2
destMAC: 00:00:00:00:00:FE
srcIP: 10.0.0.1
srcMAC: 00:00:00:00:00:01
Host 1
Host 2

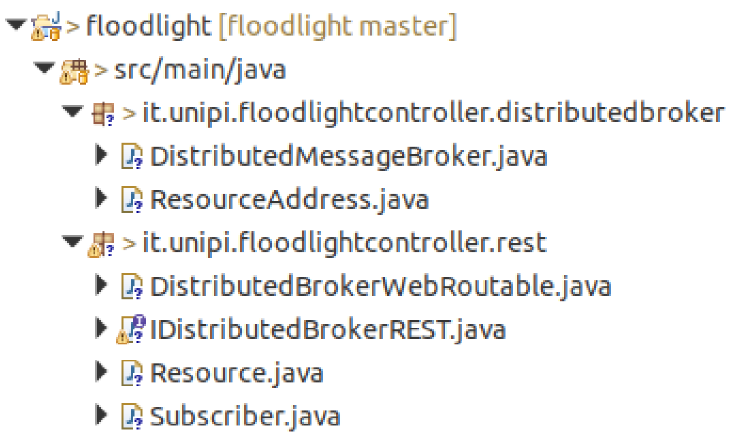
Figure 2.2: UDP packets flow

UDP packets are processed differently by the first switch they encounter than by the other switches in their path. As explained earlier, the first switch will query the controller to translate the virtual address into **n** physical addresses.

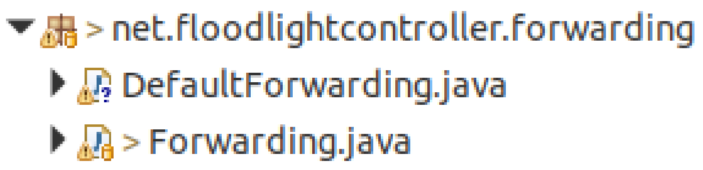
The controller will perform the operations of traffic filtering that we illustrated earlier and finally it can create **n** copies of the incoming message to be sent to the **n** subscribers of the resource. The other switches on the path only have to check the destination IP of the packet, if it is a physical address (an address not contained in the data structure where the resources with their subscribers are stored) then they have to leave the message to the *forwarding* module which will take care of it.

As mentioned above, the *forwarding* module has been patched to prevent unwanted behavior in communication between hosts. The patched version of the module does not *flood* for every packet of which it does not know the destination device, but only forwards the message on one port, the one that is part of the minimum path to the destination.

3 | Implementation



```
> floodlight [floodlight master]
  > src/main/java
    > it.unipi.floodlightcontroller.distributedbroker
      DistributedMessageBroker.java
      ResourceAddress.java
    > it.unipi.floodlightcontroller.rest
      DistributedBrokerWebRoutable.java
      IDistributedBrokerREST.java
      Resource.java
      Subscriber.java
```



```
> net.floodlightcontroller.forwarding
  DefaultForwarding.java
  > Forwarding.java
```

We implemented two packages:

- **it.unipi.floodlightcontroller.rest**
  It takes care of the REST API management described in chapter 2.
- **it.unipi.floodlightcontroller.distributedbroker**
  Contains the main module that implements the message broker system. *DistributedMessageBroker*, and a utility class *resourceAddress*

For the reasons described in Chapter 2, we also added a custom forwarding module *Forwarding*, in the original forwading module we only added a few lines in already implemented methods.

For simplicity we are going to describe only the modules that deal with the implementation of our distributed message broker system, leaving out those for the management of the REST API because they are short and easy to understand directly from the code.

3.1 DistributedMessageBroker

The main data structures defined are the following:

`private final static MacAddress SERVER_MAC MacAddress.of("00:00:00:00:00:FE");`
It is the default MAC address shared by all resources.

```
    private final Map<IPv4Address,HashMap<MacAddress, IPv4Address>>
resourceSubscribers = new HashMap<>();
```
Data structure that contains all the Resources and all subscribers for each resource.

`IPv4Address lastAddressUsed = null;`
It is the last resource address used to decide what the next one is.

The main methods defined are the following:

- `getResourceSubscribers():Map<Ipv4Address, HashMap<MacAddress, IPv4address>>`
  Returns the list of all subscribers of a resource
- `isValidPublisher(MacAddress publisherAddressMAC, IPv4Address resourceIP): boolean`
  Returns true if the publisher address is not subscribed to the resource on which it is trying to publish.
- `isSubscriber(IPv4Address resourceIpAddress, MacAddress subscribeMac):boolean`
  Returns true if the mac address is of a subscriber of this resource and false otherwise.
- `isFirstSwitch(IOFSwitch sw, MacAddress source_mac_address):boolean`
  Returns true if the switch that addressed the controller is the first hop for the packet in question, which is very important because in this case it is necessary to perform the process of translation from virtual to physical addresses of the subscribers. Returns false otherwise, i.e. the switch will have to perform normal forwarding as it is already working with physical addresses.
- `getShortestPath(DatapathId startSwitch, SwitchPort`$[] endSwitches) : Path$
  Returns the shortest path to the destination. It considers all switches attached to the destination and finds the shortest path.
- `filterPacket(IOFSwitch sw, Ethernet ethernetFrame):boolean`
  Returns true if a packet is neither an ARP nor an IP packet or if it is but is not destined for a virtual resource address. Returns false otherwise.
- `handleIpPacket(IOSwitch sw, OFPacketIn packetIn, Ethernet ethernetFrame, IPv4 ipPacket): Command`
  Handles IP packets so it takes care of discriminating using the function *isResourceAddress* if the IP packet is destined to a resource, and eventually call the method *handleRequestToResource* to handle it, otherwise it proceeds without processing the packet.
- `createArpReply(Ethernet ethernetFrame, ARP arpRequest, IPv4Address resource_virtual_address): IPacket`
  Constructs the Ethernet packet containing the ARP response for the virtual address of the resource then specifying the MAC common to all resources.

- `handleArpRequest(IOSwitch sw, OFPacketIn packetIn, Ethernet ethernetFrame, ARP arpRequest): Command`
  Constructs a packetOut for the ARP request by calling the function *createArpReply*

- `handleRequestToResource(IOSwitch sw, OFPacketIn packetIn, Ethernet ethernetFrame, IPv4 ipPacket): void`
  Creates the packetOut containing the list of actions for translating the virtual address of the resource into the physical addresses of the subscribers, it performs the translation by calling the function *translateDestinationAddfressIntoReal*

- `translateDestinationAddfressIntoReal(IOFSwitch sw, String subscriberMAC, String subscriberIP, OFPOrt outputPort): ArrayList<OFAction>`
  Performs the actual translation of the ip address and MAC address of the packet, returns a list of OFActions to replace the virtual ip of the resource with the ip of the subscriber, replace the Mac Address of the resource with the Mac address of the subscriber and set the output port.

3.2 ResourceAddress

Static class that contains only one method to check if an ip address matches a resource. It is included in both the *distributedMessageBroker* and *Forwarding* modules.

- `isResourceAddress(addressIP:IPv4Address):boolean`
  Returns true if the address is of a resource that exists and false otherwise.

4 | Testing

The Distributed Message Broker system will be tested using the Floodlight project (open source OpenFlow controller), with the addition of the developed modules. In order to simulate a network environment to test the broker platform, the Mininet application will be used, as a way to create a set of virtual hosts and switches, connected to the SDN controller, providing a customizable virtual network. Considering the project requirements and a possible application scenario, two network configurations scripts were created using the Mininet python library, as a way to validate the broker functionalities and to test its implemented features. The topologies created will then provide several hosts to be used as computing nodes (publishers and subscribers) and a number of switches, providing connectivity. To analyze the flow of messages sent in the SDN, the packets destined to the virutual addresses (resources) will be captured and displayed using the Wireshark software, as a way to demonstrate the test results.

4.1 Initial message distribution testing

To observe and test the basic functionalities of the Message Broker system, a general topology in mininet was deployed, providing 8 hosts connected using a set of 4 switches. This initial test will focus on the simple features of the system, such as publishing UDP messages in a resource and managing subscriptions. Figure 4.1 illustrate the described topology for the initial tests.

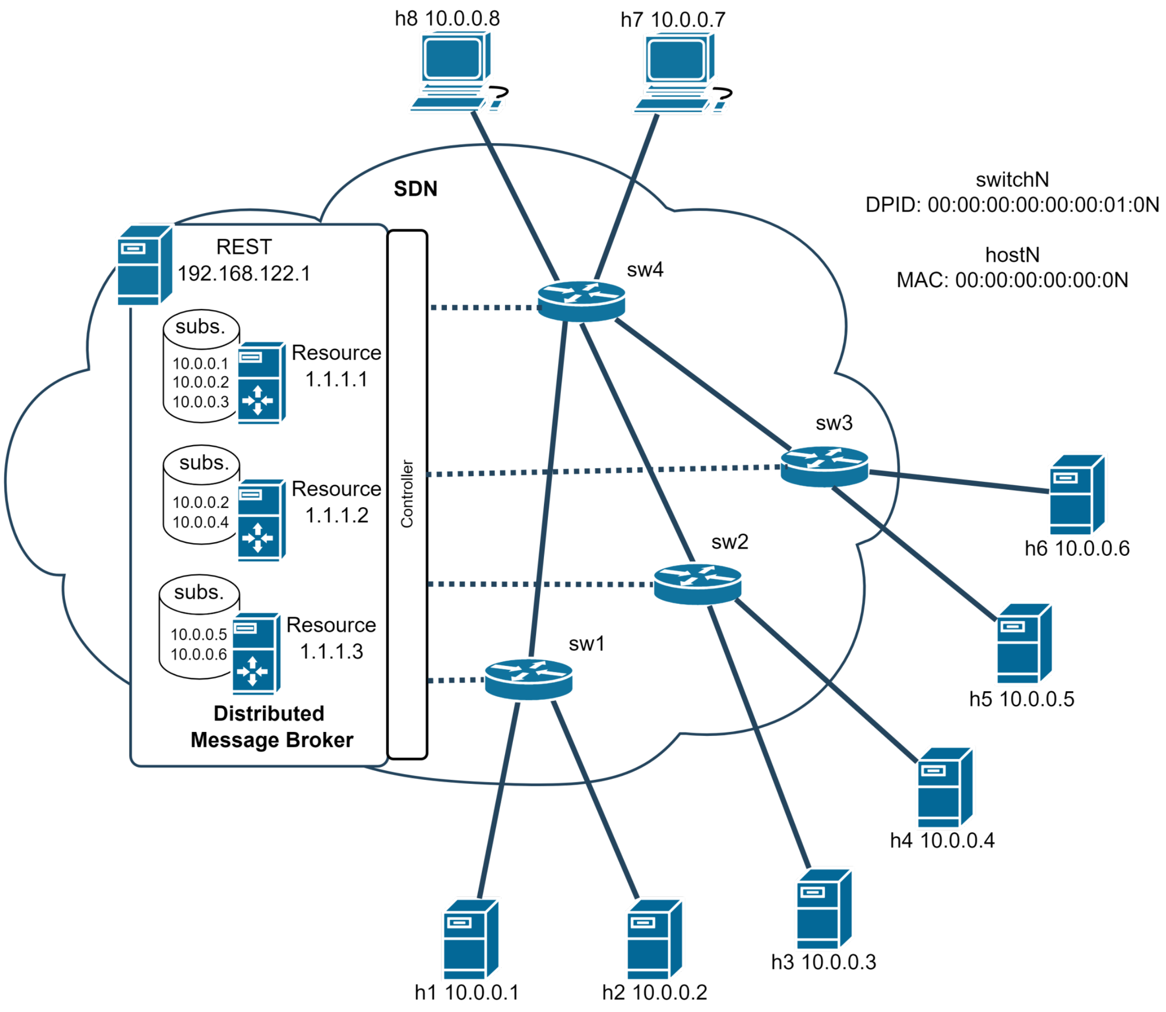


Figure 4.1: Mininet topology for initial platform testing

From the defined topology, the computing nodes were subscribed and divided in three virtual addresses. All configuration commands are issued using POST, GET and DELETE messages sent to the broker REST interface. For testing the system capabilities, node h2 was subscribed to two different resources, as the platform allow this configuration. Also, two nodes were not subscribed to any resource, acting only as publishers in the network configuration. Figure 4.2 and 4.3 illustrates the system response when sending commands for the creation of resources and computing node subscription, respectively:

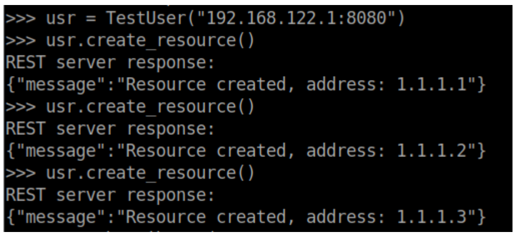
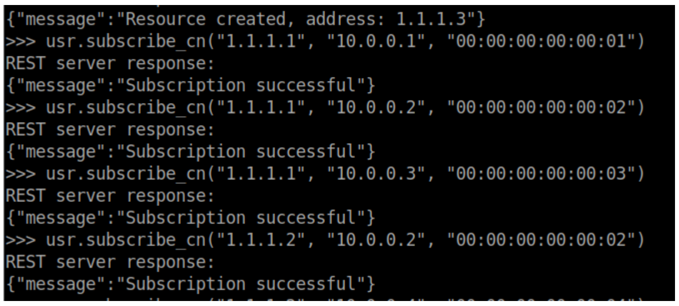

```
>>> usr = TestUser("192.168.122.1:8080")
>>> usr.create_resource()
REST server response:
{"message":"Resource created, address: 1.1.1.1"}
>>> usr.create_resource()
REST server response:
{"message":"Resource created, address: 1.1.1.2"}
>>> usr.create_resource()
REST server response:
{"message":"Resource created, address: 1.1.1.3"}
```

Figure 4.2: Creating resources on the broker system

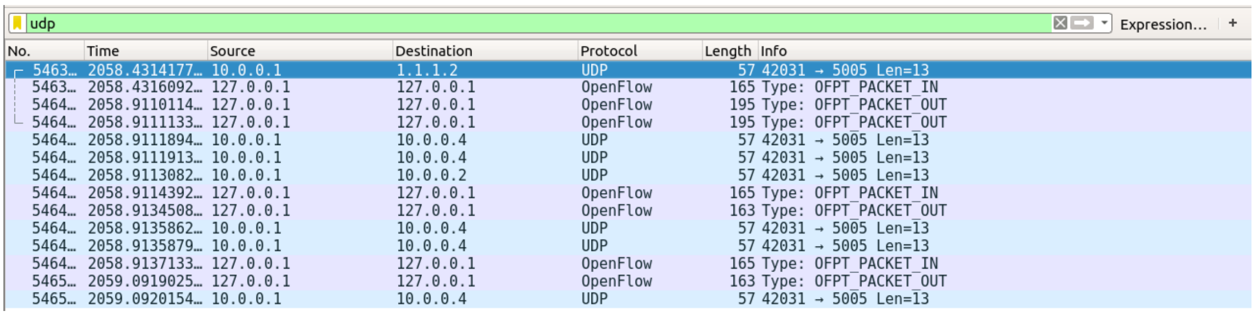


```
{"message":"Resource created, address: 1.1.1.3"}
>>> usr.subscribe_cn("1.1.1.1", "10.0.0.1", "00:00:00:00:00:01")
REST server response:
{"message":"Subscription successful"}
>>> usr.subscribe_cn("1.1.1.1", "10.0.0.2", "00:00:00:00:00:02")
REST server response:
{"message":"Subscription successful"}
>>> usr.subscribe_cn("1.1.1.1", "10.0.0.3", "00:00:00:00:00:03")
REST server response:
{"message":"Subscription successful"}
>>> usr.subscribe_cn("1.1.1.2", "10.0.0.2", "00:00:00:00:00:02")
REST server response:
{"message":"Subscription successful"}
```

Figure 4.3: Subscribing computing nodes to resources on the broker system

As the test scenario is configured, a simple UDP publisher/subscriber application was used to test the communication between the computing nodes. Some of the available mininet hosts will act as publishers, sending messages in the system using a UDP client python application (the script is executed in the mininet terminal of each host deployed). This script then sends a string message from one of the hosts to a created virtual address. For the listeners in the test scenario a UDP server python application was provided, enabling the node to bind to incoming IP addresses and retrieve the payload of the UDP messages.

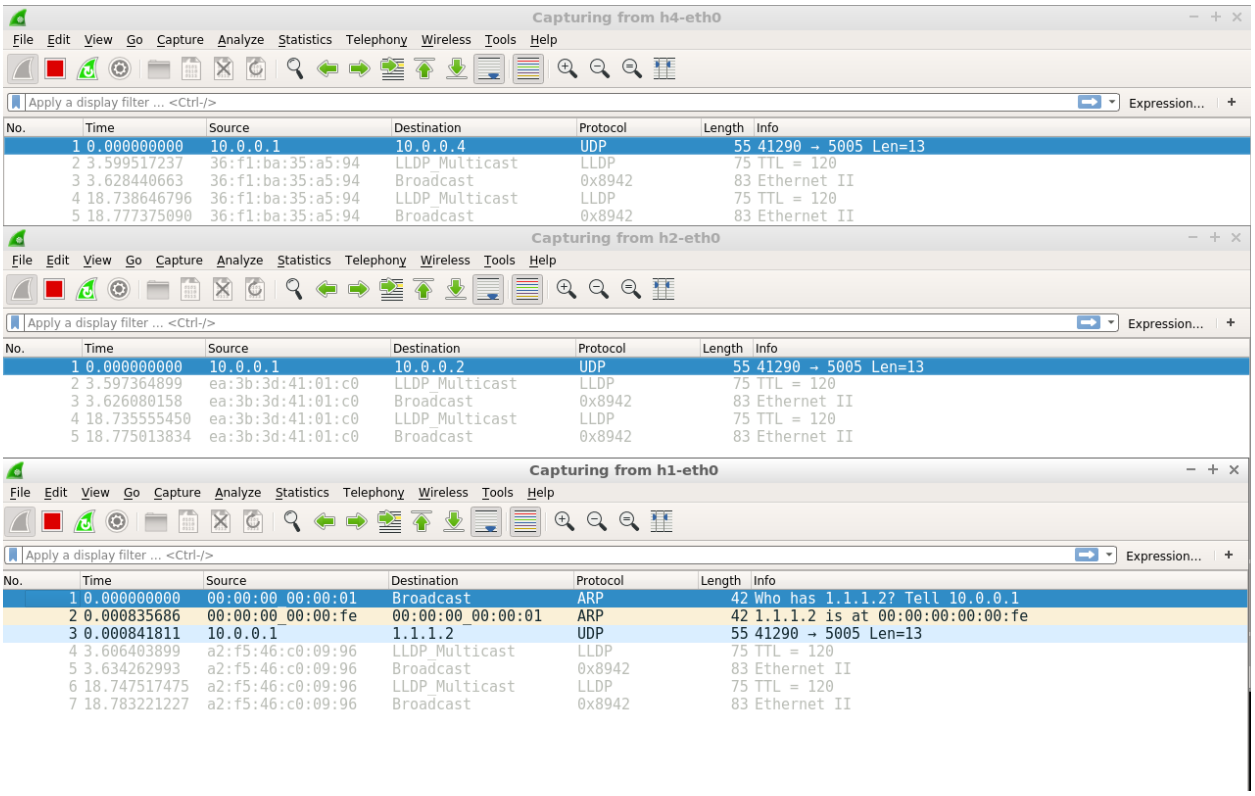
The captured packets of the first executed test are illustrated in Figure 4.5, showing the flow of UDP packets in the virtual network deployed (all interfaces with 'udp' as a filter). In this scenario computing node h1 (IP addr: 10.0.0.1) is publishing a message in the resource identified by the IP address 1.1.1.2, containing two subscribers (h2 at 10.0.0.2 and h4 at 10.0.0.4).



```
No.     Time             Source      Destination  Protocol  Length Info
5463…   2058.4314177…    10.0.0.1    1.1.1.2      UDP           57 42031 → 5005 Len=13
5463…   2058.4316092…    127.0.0.1   127.0.0.1    OpenFlow     165 Type: OFPT_PACKET_IN
5464…   2058.9110114…    127.0.0.1   127.0.0.1    OpenFlow     195 Type: OFPT_PACKET_OUT
5464…   2058.9111133…    127.0.0.1   127.0.0.1    OpenFlow     195 Type: OFPT_PACKET_OUT
5464…   2058.9111894…    10.0.0.1    10.0.0.4     UDP           57 42031 → 5005 Len=13
5464…   2058.9111913…    10.0.0.1    10.0.0.4     UDP           57 42031 → 5005 Len=13
5464…   2058.9113082…    10.0.0.1    10.0.0.2     UDP           57 42031 → 5005 Len=13
5464…   2058.9114392…    127.0.0.1   127.0.0.1    OpenFlow     165 Type: OFPT_PACKET_IN
5464…   2058.9134508…    127.0.0.1   127.0.0.1    OpenFlow     163 Type: OFPT_PACKET_OUT
5464…   2058.9135862…    10.0.0.1    10.0.0.4     UDP           57 42031 → 5005 Len=13
5464…   2058.9135879…    10.0.0.1    10.0.0.4     UDP           57 42031 → 5005 Len=13
5464…   2058.9137133…    127.0.0.1   127.0.0.1    OpenFlow     165 Type: OFPT_PACKET_IN
5465…   2059.0919025…    127.0.0.1   127.0.0.1    OpenFlow     163 Type: OFPT_PACKET_OUT
5465…   2059.0920154…    10.0.0.1    10.0.0.4     UDP           57 42031 → 5005 Len=13
```

Figure 4.4: Wireshark captured packets when computing node h1 (subscriber) publishes on resource 1.1.1.2 ('any' interface with 'udp' filter)

Also, when capturing the messages on each host interface, it tis possible to identify the ARP request sen to the network, as it is also processed by the broker, since the destination requested is a virtual addres deployed (resource).



File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...

Capturing from h4-eth0

```
No.   Time          Source              Destination      Protocol  Length Info
    1 0.000000000   10.0.0.1            10.0.0.4         UDP           55 41290 → 5005 Len=13
    2 3.599517237   36:f1:ba:35:a5:94   LLDP_Multicast   LLDP          75 TTL = 120
    3 3.628440663   36:f1:ba:35:a5:94   Broadcast        0x8942        83 Ethernet II
    4 18.738646796  36:f1:ba:35:a5:94   LLDP_Multicast   LLDP          75 TTL = 120
    5 18.777375090  36:f1:ba:35:a5:94   Broadcast        0x8942        83 Ethernet II
```

Capturing from h2-eth0

```
No.   Time          Source              Destination      Protocol  Length Info
    1 0.000000000   10.0.0.1            10.0.0.2         UDP           55 41290 → 5005 Len=13
    2 3.597364899   ea:3b:3d:41:01:c0   LLDP_Multicast   LLDP          75 TTL = 120
    3 3.626080158   ea:3b:3d:41:01:c0   Broadcast        0x8942        83 Ethernet II
    4 18.735555450  ea:3b:3d:41:01:c0   LLDP_Multicast   LLDP          75 TTL = 120
    5 18.775013834  ea:3b:3d:41:01:c0   Broadcast        0x8942        83 Ethernet II
```

Capturing from h1-eth0

```
No.   Time          Source              Destination        Protocol  Length Info
    1 0.000000000   00:00:00_00:00:01   Broadcast          ARP           42 Who has 1.1.1.2? Tell 10.0.0.1
    2 0.000835686   00:00:00_00:00:fe   00:00:00_00:00:01  ARP           42 1.1.1.2 is at 00:00:00:00:00:fe
    3 0.000841811   10.0.0.1            1.1.1.2            UDP           55 41290 → 5005 Len=13
    4 3.606403899   a2:f5:46:c0:09:96   LLDP_Multicast     LLDP          75 TTL = 120
    5 3.634262993   a2:f5:46:c0:09:96   Broadcast          0x8942        83 Ethernet II
    6 18.747517475  a2:f5:46:c0:09:96   LLDP_Multicast     LLDP          75 TTL = 120
    7 18.783221227  a2:f5:46:c0:09:96   Broadcast          0x8942        83 Ethernet II
```

Figure 4.5: Wireshark captured packets in each host eth interface

For a better illustration of the message flow, Figure 4.6 shows the path that the generated messages must go through, as well as the controller generated messages.

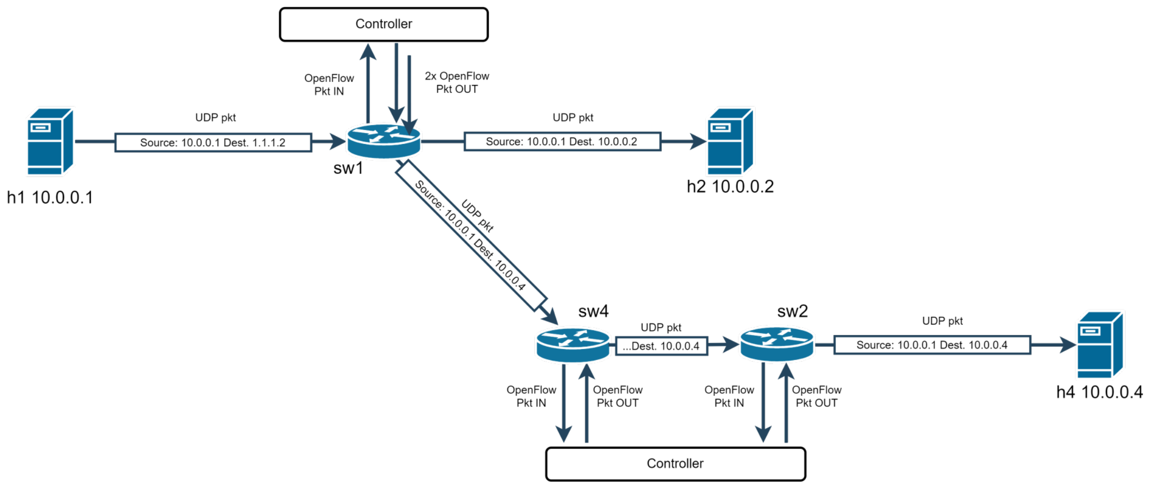
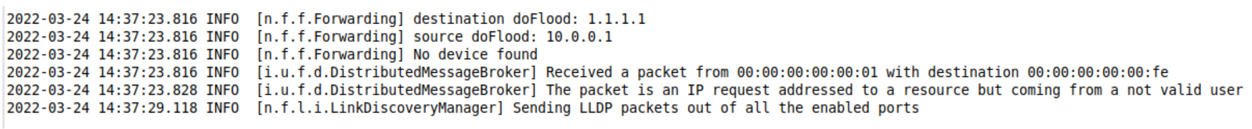


Figure 4.6: Message flow of computing node h1 publishing a message in resource 1.1.1.2

In the provided test scenario, as the first UDP packet enters the virtual network, the controller identifies that the destination IP address points to a registered virtual address (addresses are filtered), via the PACKET IN message that are generated. With this, the controller now generates two PACKET OUT messages, using the added functions for the resolution of a packet addressed to a resource. In order to reach nodes in different switches, the normal flow managed by the controller is used, allowing the standar multi hop communication implemented in floodlight.

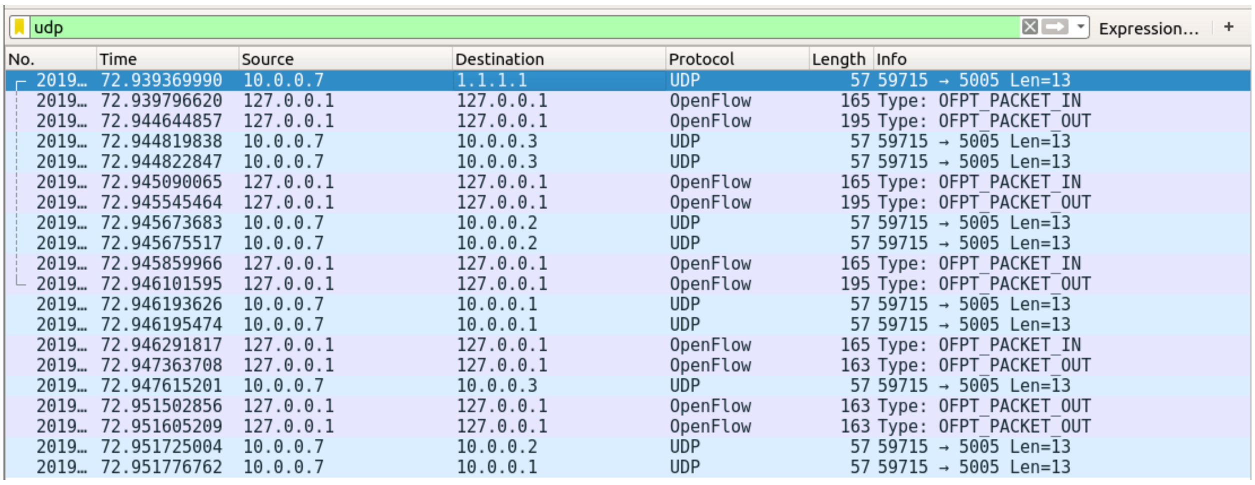
Complementing the test using h1 as a publisher, a new message was generated, destined now to the resource 1.1.1.1. This resource then contains three subscribers, including h1. Due to this distribution, h1 is prevented to publish the message, as it targets a resource where it is also a subscriber. This feature implemented in the broker system provents possible loops in the network, as well as restricts the subscribers to flood the resource address. Figure 4.7 shows the log from the Message Broker, when this test was performed:



```
2022-03-24 14:37:23.816 INFO  [n.f.f.Forwarding] destination doFlood: 1.1.1.1
2022-03-24 14:37:23.816 INFO  [n.f.f.Forwarding] source doFlood: 10.0.0.1
2022-03-24 14:37:23.816 INFO  [n.f.f.Forwarding] No device found
2022-03-24 14:37:23.816 INFO  [i.u.f.d.DistributedMessageBroker] Received a packet from 00:00:00:00:00:01 with destination 00:00:00:00:00:fe
2022-03-24 14:37:23.828 INFO  [i.u.f.d.DistributedMessageBroker] The packet is an IP request addressed to a resource but coming from a not valid user
2022-03-24 14:37:29.118 INFO  [n.f.l.i.LinkDiscoveryManager] Sending LLDP packets out of all the enabled ports
```

Figure 4.7: Computing node h1 (subscriber)publishing on resource 1.1.1.1 - publishing on a topic from a subscribed computing node (logs produced by the Distributed Message Broker module in the SDN controller)

For a second test of the message flow, the computing node h7 will publish a message in resource 1.1.1.1, as a way to test the distribution of messages across nodes connected to different switches. The result of this test is shown in Figure 4.8, as the originated message triggers the broker in sw4.



No.	Time	Source	Destination	Protocol	Length	Info
2019…	72.939369990	10.0.0.7	1.1.1.1	UDP	57	59715 → 5005 Len=13
2019…	72.939796620	127.0.0.1	127.0.0.1	OpenFlow	165	Type: OFPT_PACKET_IN
2019…	72.944644857	127.0.0.1	127.0.0.1	OpenFlow	195	Type: OFPT_PACKET_OUT
2019…	72.944819838	10.0.0.7	10.0.0.3	UDP	57	59715 → 5005 Len=13
2019…	72.944822847	10.0.0.7	10.0.0.3	UDP	57	59715 → 5005 Len=13
2019…	72.945090065	127.0.0.1	127.0.0.1	OpenFlow	165	Type: OFPT_PACKET_IN
2019…	72.945545464	127.0.0.1	127.0.0.1	OpenFlow	195	Type: OFPT_PACKET_OUT
2019…	72.945673683	10.0.0.7	10.0.0.2	UDP	57	59715 → 5005 Len=13
2019…	72.945675517	10.0.0.7	10.0.0.2	UDP	57	59715 → 5005 Len=13
2019…	72.945859966	127.0.0.1	127.0.0.1	OpenFlow	165	Type: OFPT_PACKET_IN
2019…	72.946101595	127.0.0.1	127.0.0.1	OpenFlow	195	Type: OFPT_PACKET_OUT
2019…	72.946193626	10.0.0.7	10.0.0.1	UDP	57	59715 → 5005 Len=13
2019…	72.946195474	10.0.0.7	10.0.0.1	UDP	57	59715 → 5005 Len=13
2019…	72.946291817	127.0.0.1	127.0.0.1	OpenFlow	165	Type: OFPT_PACKET_IN
2019…	72.947363708	127.0.0.1	127.0.0.1	OpenFlow	163	Type: OFPT_PACKET_OUT
2019…	72.947615201	10.0.0.7	10.0.0.3	UDP	57	59715 → 5005 Len=13
2019…	72.951502856	127.0.0.1	127.0.0.1	OpenFlow	163	Type: OFPT_PACKET_OUT
2019…	72.951605209	127.0.0.1	127.0.0.1	OpenFlow	163	Type: OFPT_PACKET_OUT
2019…	72.951725004	10.0.0.7	10.0.0.2	UDP	57	59715 → 5005 Len=13
2019…	72.951776762	10.0.0.7	10.0.0.1	UDP	57	59715 → 5005 Len=13

Figure 4.8: Computing node h7 (publisher) publishing on resource 1.1.1.1 - multi hop message to multiple subscribers

The results from this initial test then demonstrate the correct behavior of the Message Broker system, allowing the distribution of the published messages across the subscribers in the platform. The UDP packets destined to a created resource IP address were then correctly managed and translated, creating the messages for the subscribers, as it was foreseen in the Broker definition.

4.2 Application testing

With the popularity of IoT devices and solutions deployed using modern network architectures, an application for a possible temperature monitoring system was designed. The objective is to simulate a scenario where the broker platform developed would manage the data transmission, improving the communication between the monitoring units and the data providers.

In this application the main actors are a temperature sensor data provider, a set of monitoring units and a set of actuators. The tests involving this scenario will focus in the analisys of the traffic between the actors when implementing the broker message handling policies, instead of a peer to peer communication. The broker system performance can be evaluated in this exmaple considering possible improvements for links with bandwidth restrictions, or power limitation to transmit messages.

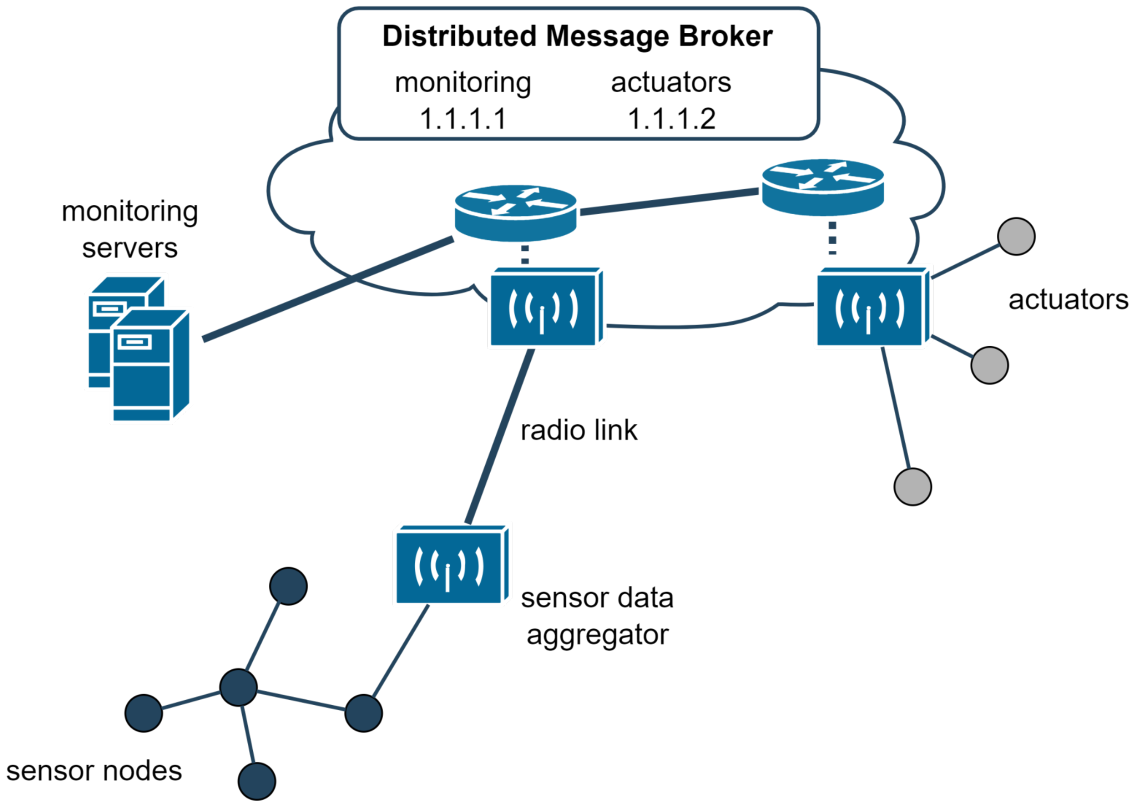


Figure 4.9: IoT Sensor Application diagram

In the proposed architecture for this sensor application, the messages from the data provider will be sent to the SDN, destined to a Monitoring virtual address, where servers (subscribers) will be monitoring the data published. Also, a second resource address will have as its subscribers a number of actuators, simulating the reactive section of the system, that will be triggered by the monitoring servers when a temperature threshold is crossed. The overall sensor application topology used in the tests is illustrated in Figure 4.10:

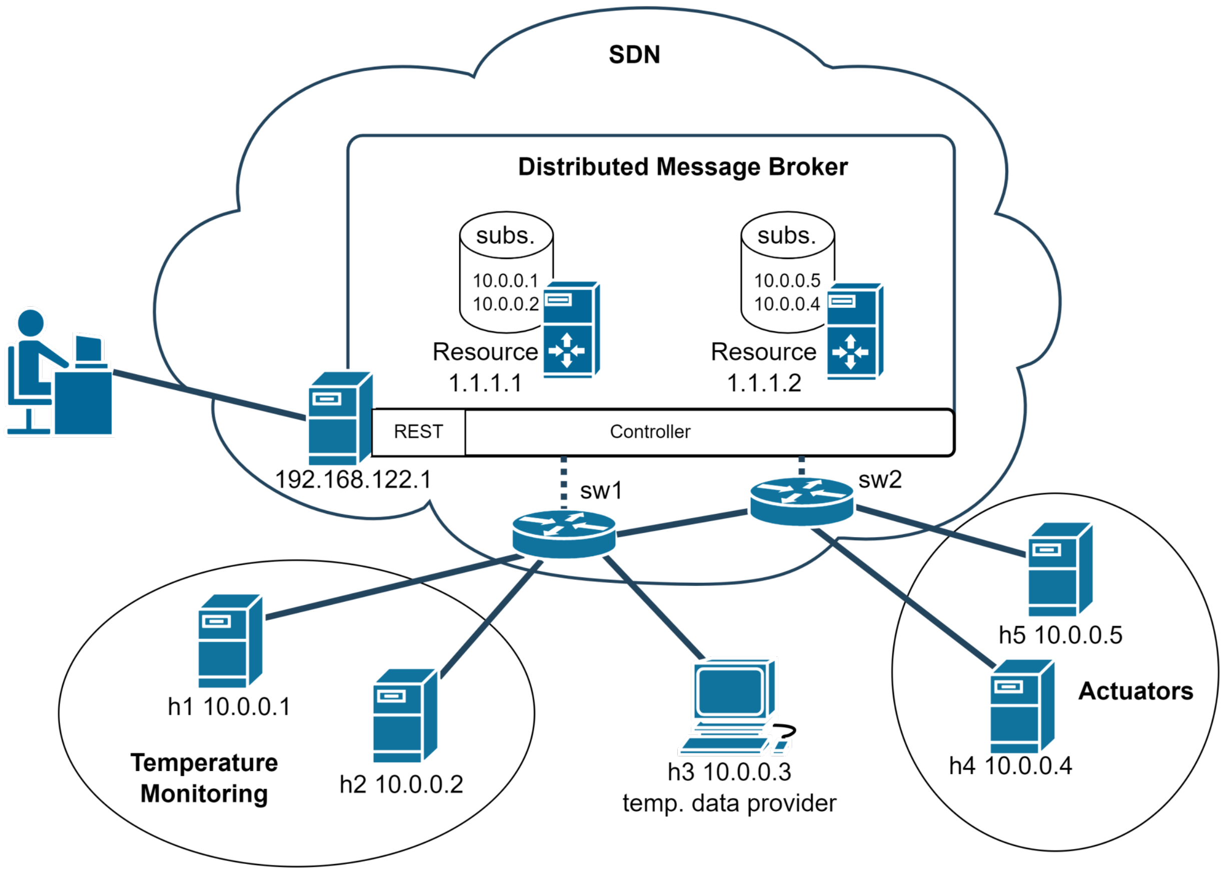
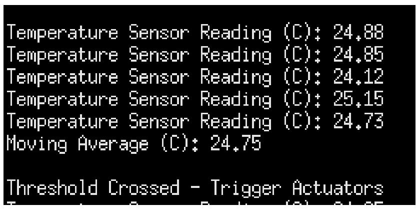


Figure 4.10: Mininet topology for the Sensor Application testing

Considering this distribution of functions for the temperature monitoring system, the defined topology was set using mininet, as well as the proper configuration of susbcribers in the broker system, using the REST interface commands. In the network, two resources were created, hosting the monitoring nodes and the actuator nodes. For the simulation, a random data generator was developed, producing and sending UDP packets within a 500 ms interval, which contain a random float value (2 decmal precision) that will represent the Temperature Reading. The data generation will be executed in the independent host h3. On hosts h1 and h2 a python server was set, allowing them to retrieve the messages that are published in the topic addressed by IP 1.1.1.1. With this, the data sent from h3 to the virtual address is distributed by the Message Broker to the monitoring servers, wich will collect and compute an average of the provided readings.

For this application, the monitoring units will use the moving avarage of the temperature readings. In h2, the moving average considers the last ten readings, and in h1 the last five readings. Also, h1 was defined to be the main monitoring server, that is, when this node calculates a moving average that is above a pre-defined temperature threshold, it will publish a trigger message in the actuators resource address (1.1.1.2). An example of the logs shown by the main monitoring server is illustrated in Figure 4.11:



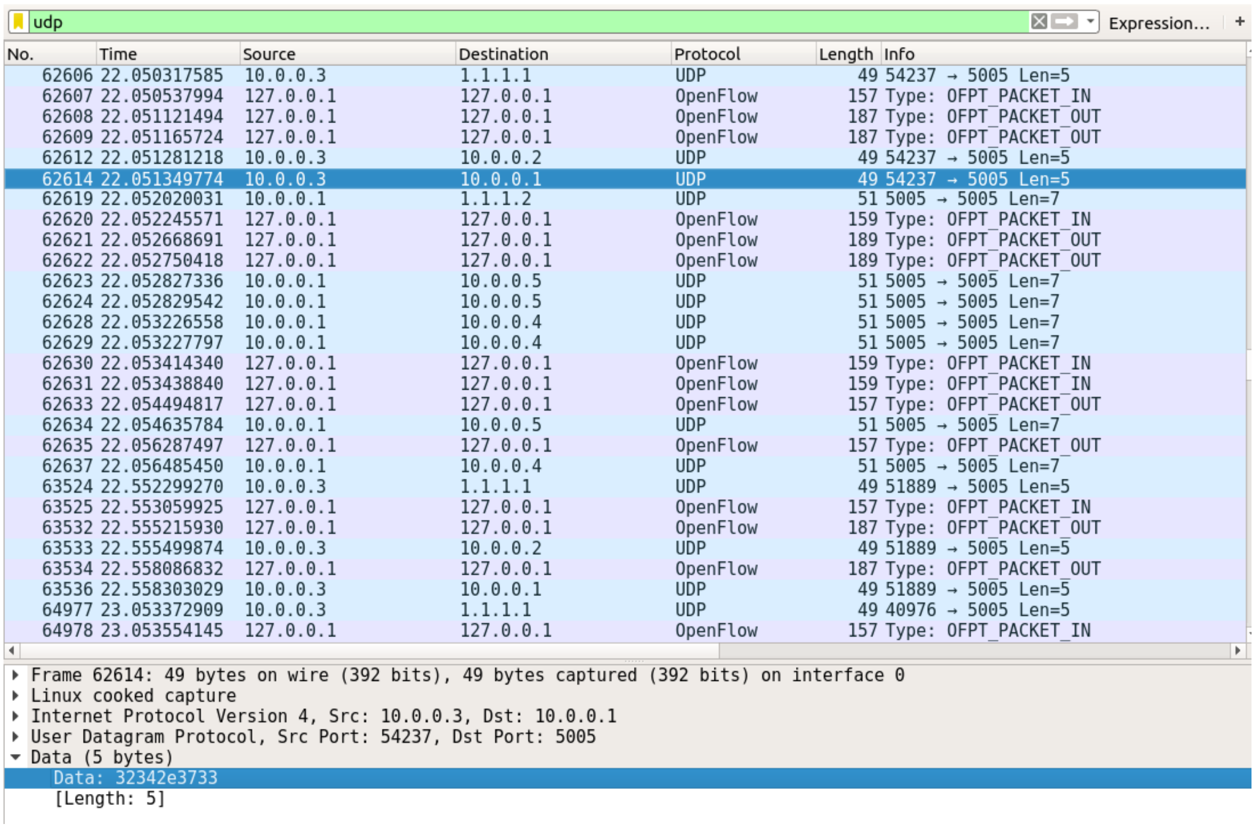
```
Temperature Sensor Reading (C): 24.88
Temperature Sensor Reading (C): 24.85
Temperature Sensor Reading (C): 24.12
Temperature Sensor Reading (C): 25.15
Temperature Sensor Reading (C): 24.73
Moving Average (C): 24.75

Threshold Crossed - Trigger Actuators
```

Figure 4.11: Main Monitoring server log - threshold crossed example

To demonstrate the full operation of the temperature management systems, a test was executed running a continuous data generation from the source node h3. Since the simulation used a random value generator, it was expected that after a while, the generator would send a sequence of readings that would cross the pre-defined threshold.

Figure 4.12 shows the captured UDP packets when that event happens.



```
No.   Time         Source     Destination Protocol Length Info
62606 22.050317585 10.0.0.3   1.1.1.1     UDP       49 54237 → 5005 Len=5
62607 22.050537994 127.0.0.1  127.0.0.1   OpenFlow 157 Type: OFPT_PACKET_IN
62608 22.051121494 127.0.0.1  127.0.0.1   OpenFlow 187 Type: OFPT_PACKET_OUT
62609 22.051165724 127.0.0.1  127.0.0.1   OpenFlow 187 Type: OFPT_PACKET_OUT
62612 22.051281218 10.0.0.3   10.0.0.2    UDP       49 54237 → 5005 Len=5
62614 22.051349774 10.0.0.3   10.0.0.1    UDP       49 54237 → 5005 Len=5
62619 22.052020031 10.0.0.1   1.1.1.2     UDP       51 5005 → 5005 Len=7
62620 22.052245571 127.0.0.1  127.0.0.1   OpenFlow 159 Type: OFPT_PACKET_IN
62621 22.052668691 127.0.0.1  127.0.0.1   OpenFlow 189 Type: OFPT_PACKET_OUT
62622 22.052750418 127.0.0.1  127.0.0.1   OpenFlow 189 Type: OFPT_PACKET_OUT
62623 22.052827336 10.0.0.1   10.0.0.5    UDP       51 5005 → 5005 Len=7
62624 22.052829542 10.0.0.1   10.0.0.5    UDP       51 5005 → 5005 Len=7
62628 22.053226558 10.0.0.1   10.0.0.4    UDP       51 5005 → 5005 Len=7
62629 22.053227797 10.0.0.1   10.0.0.4    UDP       51 5005 → 5005 Len=7
62630 22.053414340 127.0.0.1  127.0.0.1   OpenFlow 159 Type: OFPT_PACKET_IN
62631 22.053438840 127.0.0.1  127.0.0.1   OpenFlow 159 Type: OFPT_PACKET_IN
62633 22.054494817 127.0.0.1  127.0.0.1   OpenFlow 157 Type: OFPT_PACKET_OUT
62634 22.054635784 10.0.0.1   10.0.0.5    UDP       51 5005 → 5005 Len=7
62635 22.056287497 127.0.0.1  127.0.0.1   OpenFlow 157 Type: OFPT_PACKET_OUT
62637 22.056485450 10.0.0.1   10.0.0.4    UDP       51 5005 → 5005 Len=7
63524 22.552299270 10.0.0.3   1.1.1.1     UDP       49 51889 → 5005 Len=5
63525 22.553059925 127.0.0.1  127.0.0.1   OpenFlow 157 Type: OFPT_PACKET_IN
63532 22.555215930 127.0.0.1  127.0.0.1   OpenFlow 187 Type: OFPT_PACKET_OUT
63533 22.555499874 10.0.0.3   10.0.0.2    UDP       49 51889 → 5005 Len=5
63534 22.558086832 127.0.0.1  127.0.0.1   OpenFlow 187 Type: OFPT_PACKET_OUT
63536 22.558303029 10.0.0.3   10.0.0.1    UDP       49 51889 → 5005 Len=5
64977 23.053372909 10.0.0.3   1.1.1.1     UDP       49 40976 → 5005 Len=5
64978 23.053554145 127.0.0.1  127.0.0.1   OpenFlow 157 Type: OFPT_PACKET_IN

▸ Frame 62614: 49 bytes on wire (392 bits), 49 bytes captured (392 bits) on interface 0
▸ Linux cooked capture
▸ Internet Protocol Version 4, Src: 10.0.0.3, Dst: 10.0.0.1
▸ User Datagram Protocol, Src Port: 54237, Dst Port: 5005
▾ Data (5 bytes)
    Data: 32342e3733
    [Length: 5]
```

Figure 4.12: Wireshark capture when the monitoring server publishes a message in the actuator resource address (1.1.1.2)

From the logs it is possible to see the published message in the monitoring resource 1.1.1.1 being translated and forwarded to the monitoring hosts at 10.0.0.1 and 10.0.0.2 at the begining of the log. From the Wireshark interface, it was highlighted the payload of this data message, containing the value "24.73" (coded in hex values), according with the logs in 4.11. After this event, the main server then automatically publishes a trigger message in the actuator virtual address 1.1.1.2. The broker again filters the virtual addres and distributes the trigger message to the subscribed actuators h4 and h5 in the network. This then completes the monitoring cycle expected for the temperature monitoring system.

A simplified diagram for the transmission of messages in the temperature monitoring sytstem is illustrated in Figure 4.13:

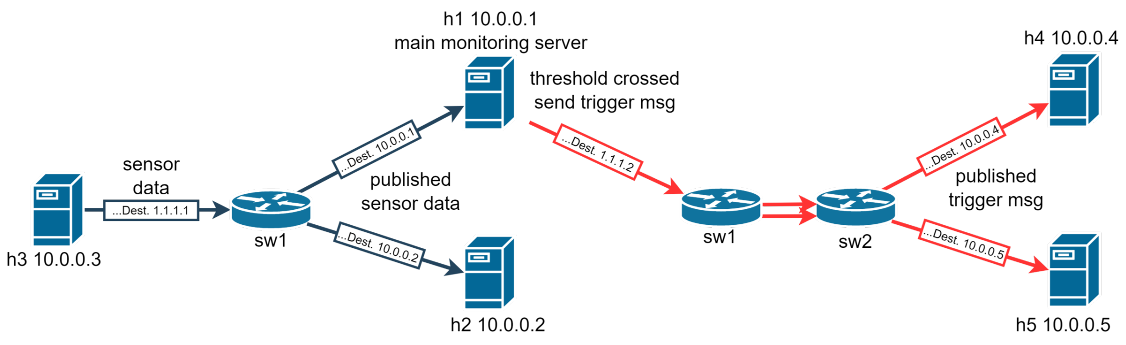


Figure 4.13: Message flow of the monitoring system using the Distributed Message Broker

4.3 Performance evaluation and conclusions

An important parameter to evaluate in the IoT application described in the tests of the broker system is the usage of the critical links in the system. In that scenario, the link with restrictions is the connection between the data provider and the main network, as the sensors are often distributed in the environment and send data to a local aggregator, which then transmits the data to the monitoring using a wireless link. Considering these characteristics, a simple analysis of the estimated usage of this link will be assessed when increasing the amount sensor data providers in the system, thus, evaluating the scalability of the monitoring system using different approaches for the messages handling (broker and P2P).

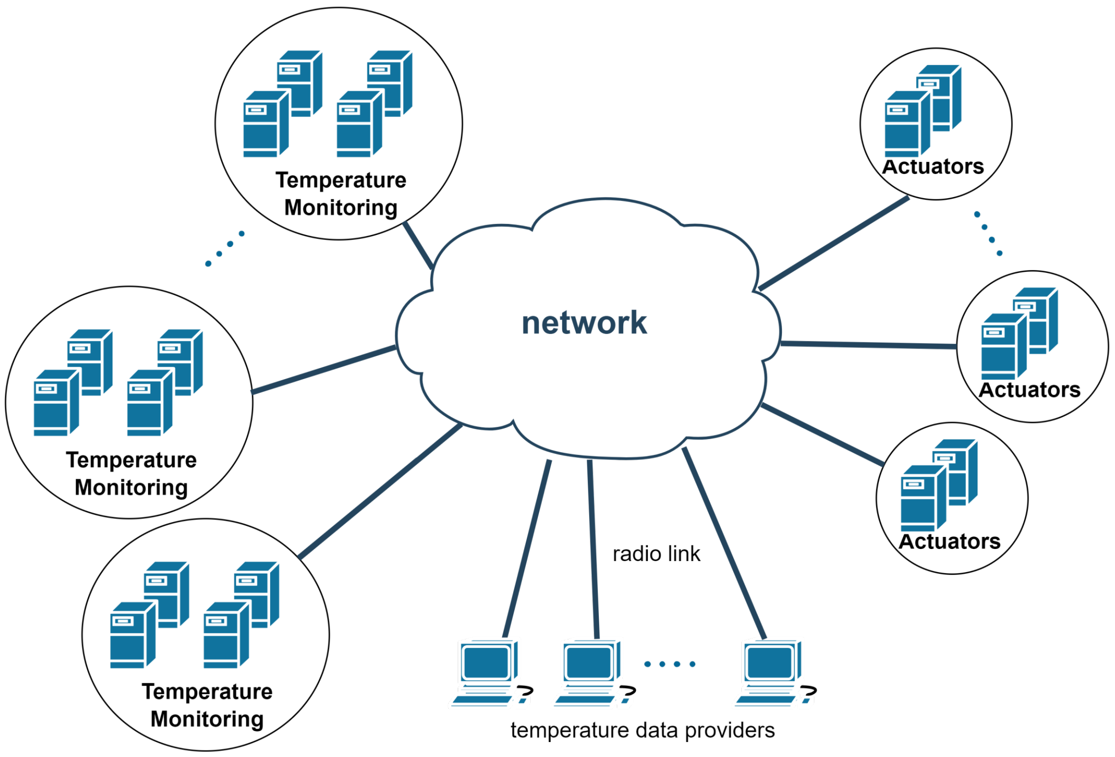


Figure 4.14: Message flow of the monitoring system using the Distributed Message Broker

The scalable monitoring temperature application than is expected to contain a number S of data generators, with a number N of monitoring clusters (with a number M of servers). With these paramters it was estimated the amount of packets trnasmitted in the radio link, when the sensors are sampled with a given rate. In the tests, the sample rate was 500 ms, that is 2 readings being generated every second or R = 2 packets/s. From this variables, the usage of the link is measured by the number of packets sent in it.

In this analysis, collisions and retransmissions are not considered, for a more detailed evaluation it is recommended to use a network simulator, such as OMNet++.

For the packet/s rate in the link evaluation, in broker implementation this number must number be proportional to the number of sensor data providers times the rate that the data is generated or simply S*R. When considering a peer to peer implementation, that is, all data providers must send a unique message to every server that is in its monitoring cluster, the amount of packets expected is then S*R*M. A plot with different workloads is shown in Figure 4.15:

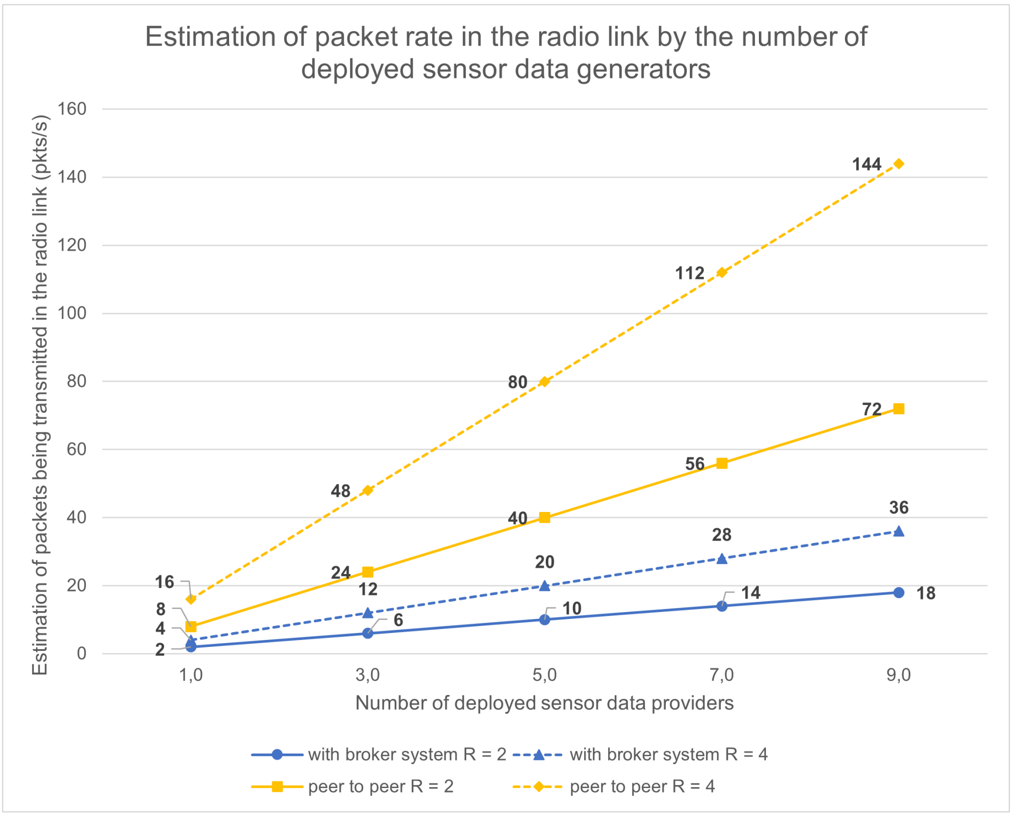


Figure 4.15: Comparison of packets traversing the radio link when using the broker system and the peer to peer communication

From this simple analysis it is possible to concluede that the broker system allows for a better scalabilty for the target application, as it can reduce drastically the amount of messages transmitted to the multiple monitoring nodes, when compared to a traditional peer to peer distribution of the messages in the network. The system than is a good alternative when implementing this type of IoT application as it can improve the link stability and reliability, as less packets are transmitted in the wireless link.

Overall the broker system developed was a consistent and proven to be efficient in the distribution of messages to the subscribers of the platform, as it was specified in the project requirements.