FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Individual Project (Project 18)

Multicast Based NFV Resource Information Synchronization for a Distributed NFV Orchestration

Submitted by:	Devan Premchandrakumar Nair
First examiner:	Prof.Dr.Armin Lehmann
Date of start:	19 June 2020
Date of submission:	16 September 2020

Statement

Content

1 Introduction

In highly fault prone communication systems environment, it is necessary to have an infrastructure that is resilient to failures of elements of the network architecture. Moreover, such systems should not contain any centralized points of access or control. Such systems then inevitably demand dynamic provisioning and allocation of network resources which are agnostic of the underlying physical topology as far as possible. Network Functions Virtualization (NFV) enables for such on-demand provisioning and decommissioning of network resources suitable for such an application. The limitations of the traditional ETSI defined standards of NFV Framework are considered and analyze alternatives for a more resilient approach to utilization of virtualized network functions. In such applications, failures will be unavoidable. Therefore, communication protocols and operational mechanisms that are capable of handling dynamic situations to ensure continuously working infrastructure are required. In addition, technical requirements, and mechanisms to isolate and react to scenarios of failure need to be identified.

The possibility of a single point of failure with a logically centralized Network Function Virtualization Orchestrator (NFVO) prevents the integration of NFV into such application. With the control of resources designated to such NFVO's it is highly likely that in the event of a failure ,collapse of an NFVO may lead to compromised usability of large number of network resources. Thus, there is a need to look for mechanisms to distribute functionality of traditional NFVO's to multiple physical resources to improve resiliency. This study aims to looks for a mechanism which is directly associated with one such technical requirement namely the identification and distribution of resource information in clustered hierarchical network. A mechanism to isolate and identify elements involved in a fault situation that may arise in such a network is considered. The clustered architecture and the logical separations associated therewith are discussed in the subsequent sections. A multicast-based resource information distribution system as it will enable logical separation and efficient means of broadcasting network resource information in high resilient systems was employed. The analysis of multicast routing protocols or the procedure of separation of logical layers from the underlying physical topology is not considered in this study. A suitable mechanism for distribution of information of individual NFVO resources to other logically equivalent elements in the network and isolation of faults and disruptions that may happen at that logical layer is provided. The solution proposed aims to be generalized independent of implementation and underlying logical/physical topology that may have been used to demonstrate/analyze the requirements for this study.

This study is comprised of three major sections. In Chapter 2, the theoretical limits to the operability of our traditional solutions and look for improvements to that along with identification of possible metrics that may need to be considered to test the fidelity of the solution are explored. In Chapter 3, the general and technical requirements that our solution might require to fulfill its demands along with a more concrete representation of test scenarios/use cases for the solution and response that is expected from desired functionalities is isolated. In Chapter 4, the implementation that was designed with message sequences and transactions that were required to achieve the desired state are analyzed. The Appendix provides some information for setting up the emulation model on any system and discusses the steps involved in usage of solution modules.

2 Theoretical Background

This section discusses the theoretical elements involved in the construction of the implementation strategy and the underlying principles that may be involved in design of the system. Broadly the solution will deal with two major elements, the concept of virtualized resource management deployment and operations as per ETSI framework which forms the basis for Distributed Orchestration model discussed in the next chapter and the algorithm which was used to estimate the connectivity or partitioning of the topology that may occur in a highly dynamic network. The algorithm is discussed with an example in Realization chapter, here only the procedural part of the algorithm will be discussed.

2.1 ETSI NFV Management and Orchetsration Architecture

Network Functions Virtualization Orchestration (NFV-Orchestration) is responsible for coordination of network and resources for cloud based applications. Providers of cloud based services would have structured utilities of fundamental network components or network elements as a Virtual Network Function (VNF) rather than as physical hardware or specialized devices. In a similar context there can be standard physical hardware which can run specialized software designed as a VNF with a customized NFVO coordinating deployments and management of these resources (ETSI, 2015-12). Because NFV requires a lot of virtualized resources it requires a sophisticated mechanism for its resource management. Management tasks associated with VNF involves in case of a provider network- billing, lifecycle management, maintenance, operating support systems, inventory mechanisms etc. whereas for self-contained NFV implementations there might be redundancy systems, monitoring ,operations support systems (OSS) etc. The NFV Management and Orchestration (NFV-MANO) architecture shown in Figure.2.1 consists of three functional blocks: Virtual Infrastructure Manager (VIM), VNF (Virtual Network Function) Manager (VNFM), NFV Orchestrator (NFVO) and data repositories for VNF service catalog and infrastructure description. A VIM manages and controls NFVI physical and virtual resources in a single domain. This implies that an NFV architecture may contain more than one VIM, each of them managing or controlling NFVI resources from a given infrastructure resource provider. Each VNF instance is assumed to have an associated VNFM. The VNFM is responsible for the management of the lifecycle of VNFs. A VNFM may be assigned with the management of a single or multiple VNF instance of the same or different type, including the possibility of a single VNFM for all active VNF instances for certain domain. (Frick, et al., 2018)

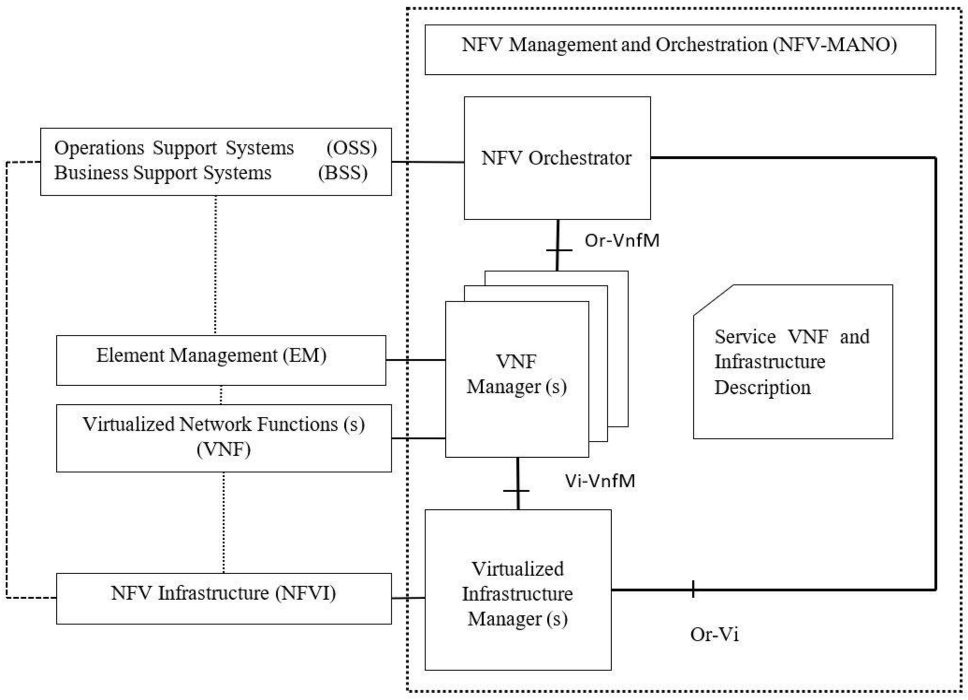

NFV Management and Orchestration (NFV-MANO)
Operations Support Systems (OSS)
Business Support Systems (BSS)
NFV Orchestrator
Or-VnfM
Element Management (EM)
VNF Manager (s)
Service VNF and Infrastructure Description
Virtualized Network Functions (s) (VNF)
Vi-VnfM
NFV Infrastructure (NFVI)
Virtualized Infrastructure Manager (s)
Or-Vi

Figure 2.1:ETSI NFV Architecture

2.2 Determination of connectedness and resource information states

Different nodes in a network topology may need to estimate whether it is connected to a particular node or not. So, a node can employ a connectivity estimation algorithm known as Breadth First Search (BFS) to compute connectivity of a node in a synchronized partition/topology. The steps involved in BFS is outlined in Figure.2.2. It basically consists of a topology tree which is pruned when a node is repeated as queue proceeds in search of extended neighbors to list. It therefore acts as an Open Shortest Path First (OSPF) (Moy, n.d.) algorithm with all the costs associated with a path being equal. However instead this method employs a primitive metric to prune its neighborhood scope trees, namely by eliminating if the node was already accounted for while searching during previous iterations.

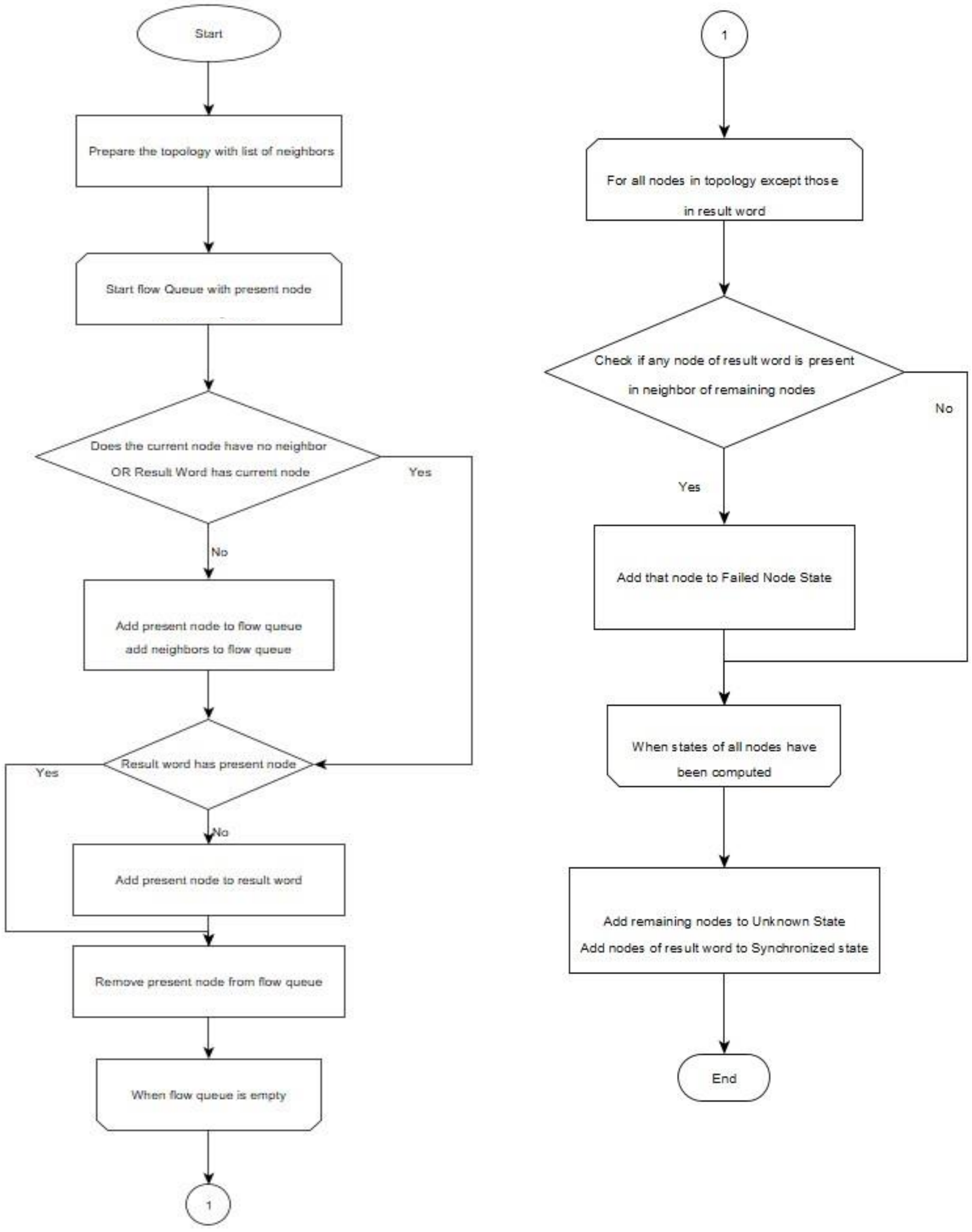

Start
Prepare the topology with list of neighbors
Start flow Queue with present node
Does the current node have no neighbor OR Result Word has current node
Yes
No
Add present node to flow queue add neighbors to flow queue
Result word has present node
Yes
No
Add present node to result word
Remove present node from flow queue
When flow queue is empty
1
1
For all nodes in topology except those in result word
Check if any node of result word is present in neighbor of remaining nodes
No
Yes
Add that node to Failed Node State
When states of all nodes have been computed
Add remaining nodes to Unknown State
Add nodes of result word to Synchronized state
End

Figure 2.2:Breadth First Search Algorithm

2.3 Constrained Application Protocol (CoAP)

CoAP is a service layer protocol that is intended for use in resource-constrained internet devices, such as wireless sensor network nodes. CoAP is fundamentally a ReST(Representative State Transfer) based session protocol over UDP with multicast capability. Multicast, low overhead, and simplicity are extremely important for Machine-to-Machine (M2M) devices, which tend to be embedded and may have limited power supply than traditional internet devices have. Therefore, efficiency is very important. CoAP can run on most devices that support UDP or a UDP analogue. CoAP provides a request/response interaction model between application endpoints, supports built-in discovery of services and resources, and includes key concepts of the Web such as URIs and Internet media types. (Shelby, et al., n.d.). Some of the important fields of a CoAP message are –

- Code: A 8-bit unsigned integer, split into a 3-bit class (most significant bits) and a 5-bit detail (least significant bits). The class can indicate a request (0), a success response (2), a client error response (4), or a server error response (5). (All other class values are reserved.) As a special case, Code 0.00 indicates an Empty message. In case of a request, the Code field indicates the Request Method (e.g. GET,POST etc.); in case of a response, a Response Code (e.g. 2.05 , 2.04 etc.).
- Message ID (MID): A 16-bit unsigned integer in network byte order. Used to detect message duplication and to match messages of type Acknowledgement/Reset to messages of type Confirmable/Non-confirmable.
- Token : The header is followed by the Token value, which may be 0 to 8 bytes, as given by the Token Length field. The Token value is used to correlate requests and responses.

For two nodes A and B communicating via CoAP, directly connected to each other, a multicast interaction may look like Figure.2.3. For a GET type request as shown in first half, a request is sent to a URI under a multicast IP address. As both nodes are listening to the multicast address both are capable to respond to such a request. In the second half a POST request between A and B is shown. In both the cases the nodes send requests to multicast address with unique generators for Message ID (MID) and token index, which are used to add a layer of reliability to the datagram payloads (Shelby, et al., n.d.).

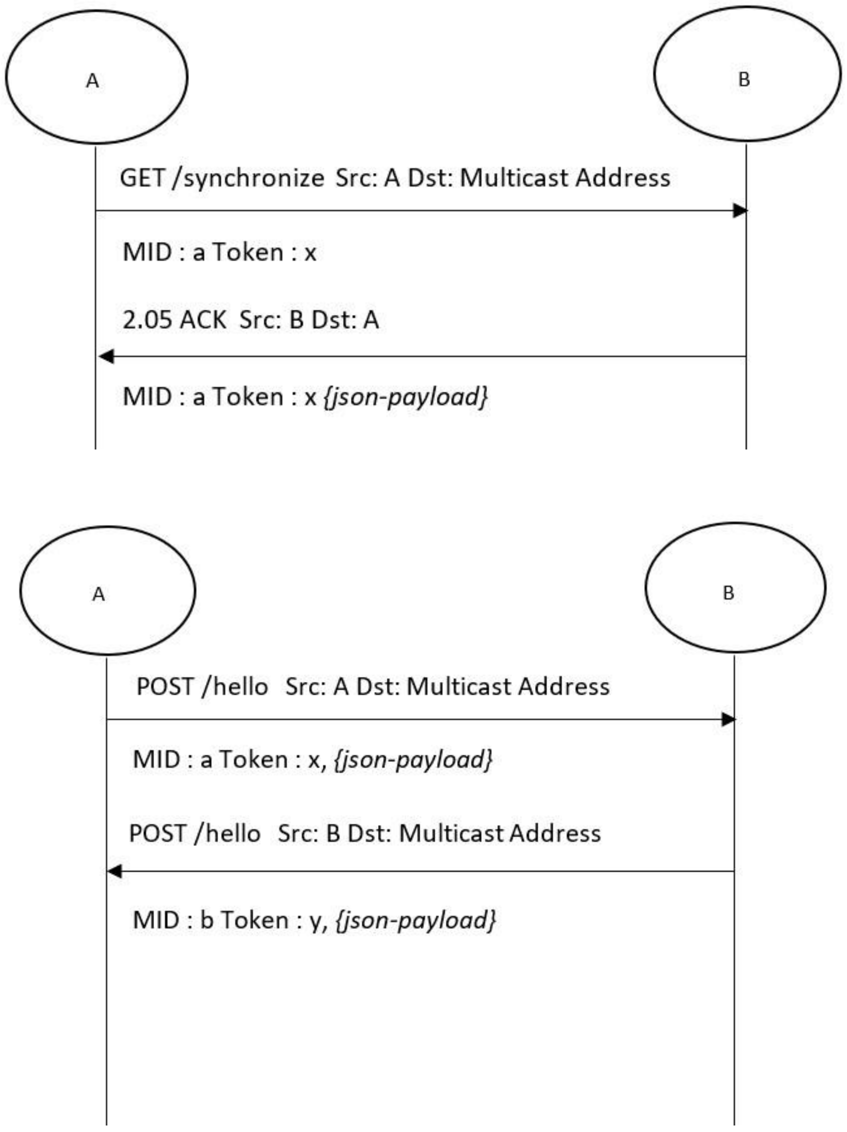

A
B
GET /synchronize Src: A Dst: Multicast Address
MID : a Token : x
2.05 ACK Src: B Dst: A
MID : a Token : x *{json-payload}*
A
B
POST /hello Src: A Dst: Multicast Address
MID : a Token : x, *{json-payload}*
POST /hello Src: B Dst: Multicast Address
MID : b Token : y, *{json-payload}*

Figure 2.3:CoAP message examples

3 Requirements Analysis

3.1 General Objectives

One of the major issues with ETSI definitions of NFV Framework is the centralization of control and access of a network architecture at the NFV Orchestrator. The possibility of failure of nodes in highly dynamic and risk environments limits the resiliency of such network architectures. To overcome such a difficulty a distributed NFV Orchestration scheme was proposed (Frick, et al., 2018). To maintain such a NFVI ,distributed orchestration must be aware of all available resources in a network. Another issue that may arise in such systems is highly dynamic changes in the network topology (Frick, et al., 2019) .The changes can either be failure of nodes/links or insertion of new nodes into the network as part of scaling or replacement for failed nodes. So, the resultant orchestration system must be able to adapt to such conditions. Such a decentralized architecture eliminates possibility of a single point of failure and enables fault tolerant means of orchestration. The possibility of a breakup of cluster arrangements arising because of multiple Cluster NFV Orchestrators failing could also be independently managed in a limited environment without losing control to the network. Such a scenario demands that these isolated sectors (network partitions) of Cluster NFV Orchestrators can dynamically identify and isolate the partitions that are caused in the global NFVO layer and adapt their resource information state tables accordingly. The resultant architecture for distributed NFV orchestration could be summarized as shown in Figure.3.1 . It consists of multiple logical layers with each layer providing a specific functionality to the architectural requirements . In this study, the functionality of synchronizing resource information in Global NFVO layer and identifying and isolating partitions that may arise in dynamic situations are analyzed.

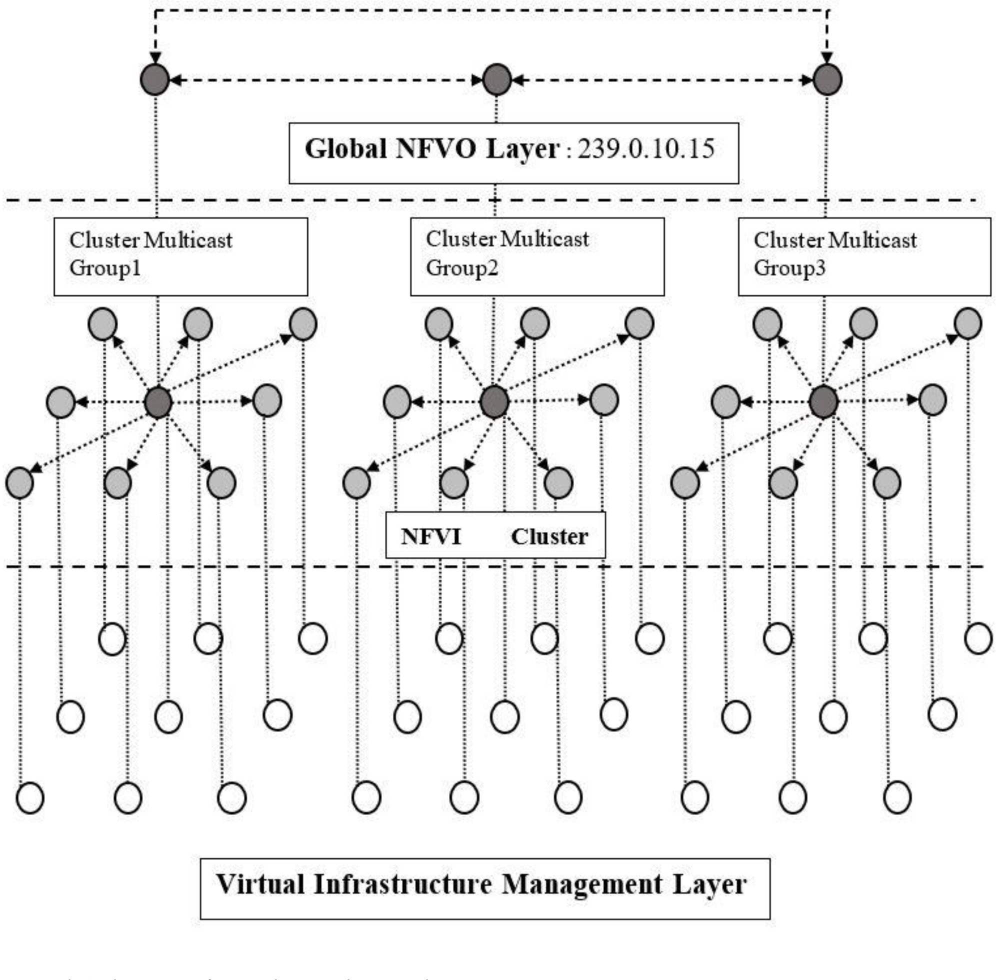


Figure 3.1:Network Architecture of a Wireless Mesh Network.

An application level solution must be conceptualised and implemented to ensure that information of services and network resources available on individual orchestrators over a cluster network is synchronized across all the orchestrators. Also, the solution should be able to identify when instances of orchestrators fail and denote them under appropriate states. The network comprises of orchestrators which are connected to one multicast group which corresponds to a logical layer. The application solution should be periodically scheduled so that dynamical changes in resource information in one or more nodes at a given time is distributed across the whole network in a stable fashion. The application must provide suitable endpoints or API's to access the resource and network information specific to an individual node and its knowledge about resources of all the other orchestrator nodes in the cluster, on each orchestrator node in the network. A suitable flooding strategy must be designed which would propagate updates/changes to network information/states to all orchestrator nodes in the network. At the network level there should be two sets of messages , one is communicated only to neighboring nodes and represents a keep alive message indicating validity of orchestrator node to the synchronization procedure with relevant information to represent the connection and the other is a reactive message that is flooded in the entire network when any orchestrator node makes an update to its resource information. The implementation solution must be agnostic of the underlying logical/physical topology and should track synchronization states of all nodes executing the application. There should be a common implementation of the services across all the orchestrator node in the network. The network level solution must ensure the orchestrators use only multicast messages to initiate a transaction. The application should be configurable at each node based on the network interfaces that are available and the network and service resources that the orchestrators have access to/can provision. The network must be capable of synchronizing any amount of resource and service information in the network, without any size restrictions of the synchronizing messages passed between the nodes .Each node must maintain its own database or a persistent solution to store synchronized resource information and state of synchronization of every node i.e. resource information should be synchronized even after an application reboot on an individual node (Not necessarily a simulation restart) .

3.2 Clarifying the Requirements

The application will use CoAP for multicast message requests and responses (Shelby, et al., n.d.). The resources associated with a node will be stored in a configuration file and user may be able to modify contents of these configuration files via API's and the solution will ensure that changes to configuration file will be distributed across all the nodes in the network . The user may be able to remove any service or network resource add or update using these API's on any node in the network .The implementation must flood the resource update/synchronize messages throughout the network, but send the keep-alive messages to the neighbors. Each orchestrator node must have two configuration sections *NETWORK SERVICES* and *APPLICATION SERVICES* referring to network resources and services that an individual orchestrator node can provide. The node must have a *HOSTID* section in configuration indicating a character string ID for the orchestrator node. The *HOSTID* is unique in the entire cluster. The network topology will consist of multiple orchestrator nodes in a mesh configuration such that failure/collapse of one node will not lead to an instance of a network partition. In addition each node must be able to estimate the states of orchestrators as 'synchronized', 'connected to failed nodes' and 'unknown' or equivalent semantic representations symbolizing nodes having the capability to synchronize orchestrator resource/s, links to the synchronized sector that have failed and nodes partitioned from the present cluster of orchestrators and there is no network path to verify their resource synchronization, respectively. Each node which runs the application will have files to persist information about the resource information about each node in the cluster with the relation *HOSTID -> NETWORK SERVICES , APPLICATION SERVICES*. Each node will flood message whenever the resource information for any of the node changes. The solution will ensure that there are fail-safe transactions to maintain same version of information in a synchronized sector .The multicast messages in the whole network should be non-confirmable to prevent unnecessary repeated requests. The requests and response in synchronization message sequence should be JSON formatted. A working CORE emulator model implementing the scenario with appropriate messages and constraints must be implemented on a suitable mesh topology (Anon., n.d.).

3.3 Target State

An example mesh network shown in Figure.3.2 was chosen to test different aspects of the solution. In the desired final state , each partition in the mesh should be synchronized with the latest resource information of all nodes in that sector. In case of a network partition, each node in the partition should mark the host ID's of all nodes synchronized with its information , the host ID's whose states cannot be synchronized because the network connectivity to those nodes have been lost and the list of nodes ,to which links of the synchronized sector has gone down thereby causing the partition. As a result of this state estimation each node in the network must be capable of determining the connectedness of each node with every other node whose resource information it is synchronizing.

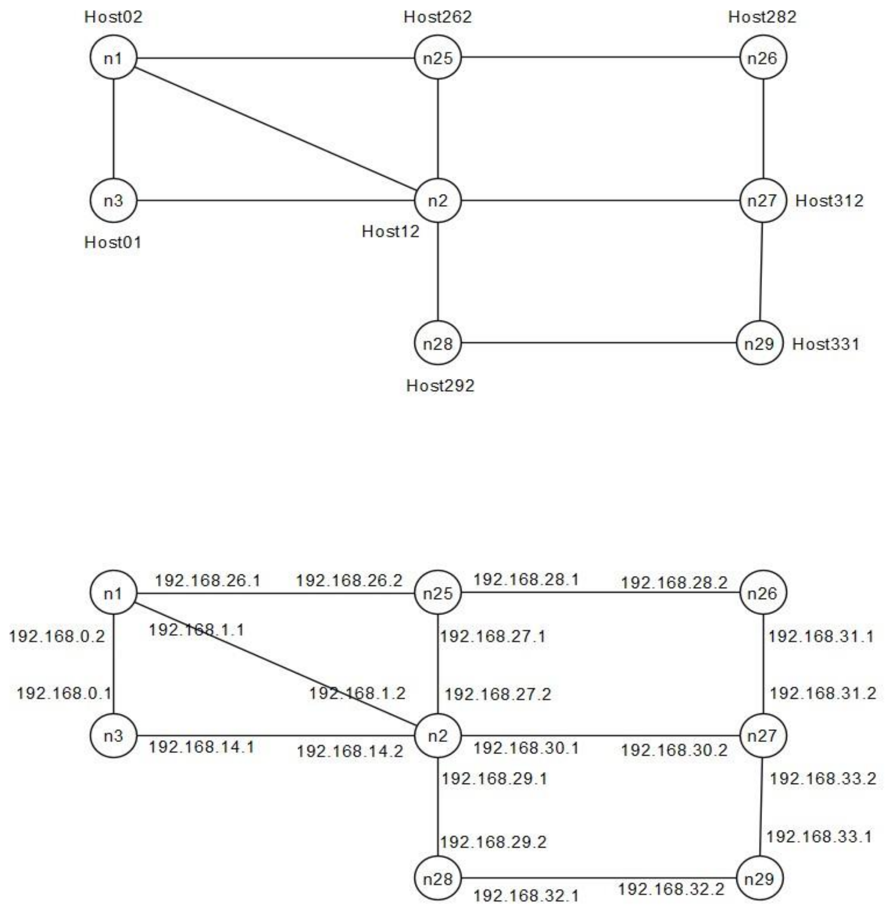


Figure 3.2:Example Topology used to analyze operation of the solution

3.4 Use Cases for the Prototype

3.4.1 Updating Resource Information of a node at runtime

In the given topology, the resource information of node n3 is changed at runtime and the rest of the nodes in the network receives the flooded Update message for the new resource information which has to be replaced instead of the old one, as shown in Figure.3.3. Node n3 sends each resource information with a version/sequence ID which should also be noted by all nodes to keep track of resource information and to make sure that nodes do not get desynchronized.

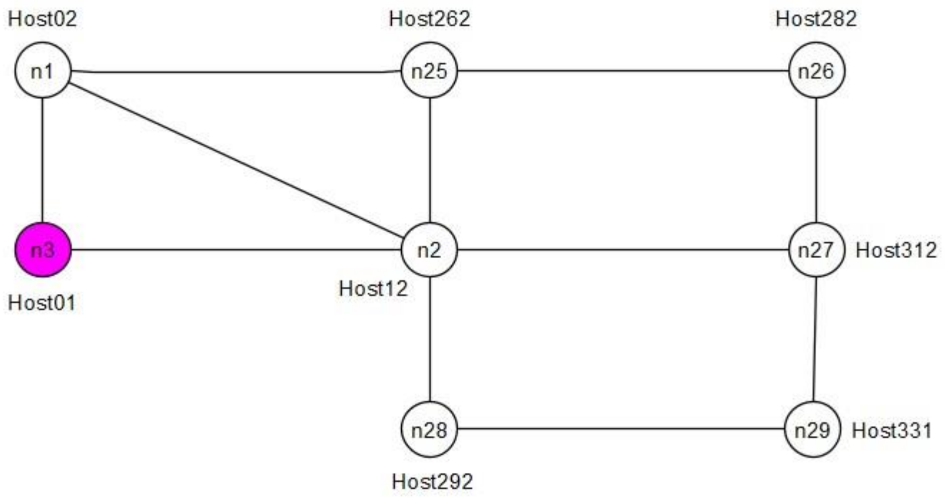


Figure 3.3:Node n3 changes resource information at runtime

3.4.2 Node injected into an already synchronized network

The network initially starts up and synchronizes without the application running on node n29 as indicated in Figure.3.4. So, the entire network except n29 synchronizes all its information. The network does not have knowledge of existence/connectedness to n29 at this point. After a while, the application is started on n29. Some time after this, n29 has resource information of all the nodes in the network and the rest of the network have updated their resource tables to include an added node n29/Host331 and estimates connectedness to the newly injected node into the network.

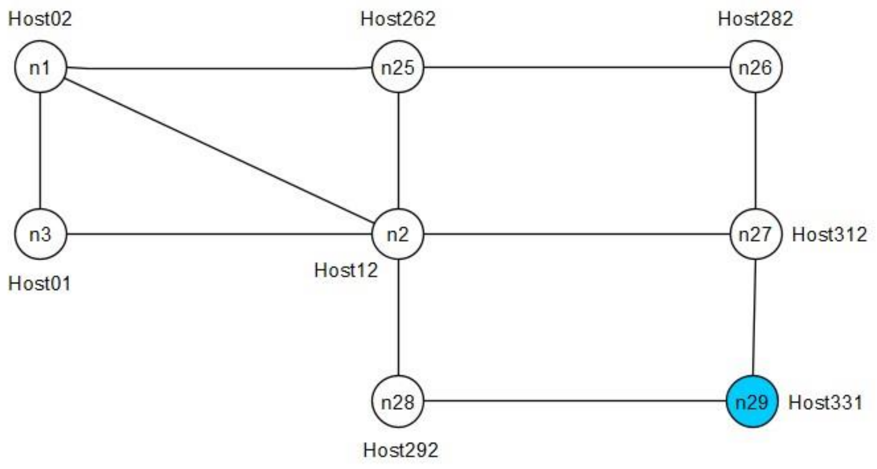


Figure 3.4:Node n29 injected into the network at runtime

3.4.3 Node Failure

In this scenario, node n25 shuts down all its interfaces at runtime after the entire network has synchronized with resource information of all the nodes in the topology. After some time, all the nodes in the network estimate n25 to be unconnected in the topology and mark interfaces to it having failed or a node-failed state incapable of being synchronized with resource information updates of the rest of the network, as shown in Figure.3.5.

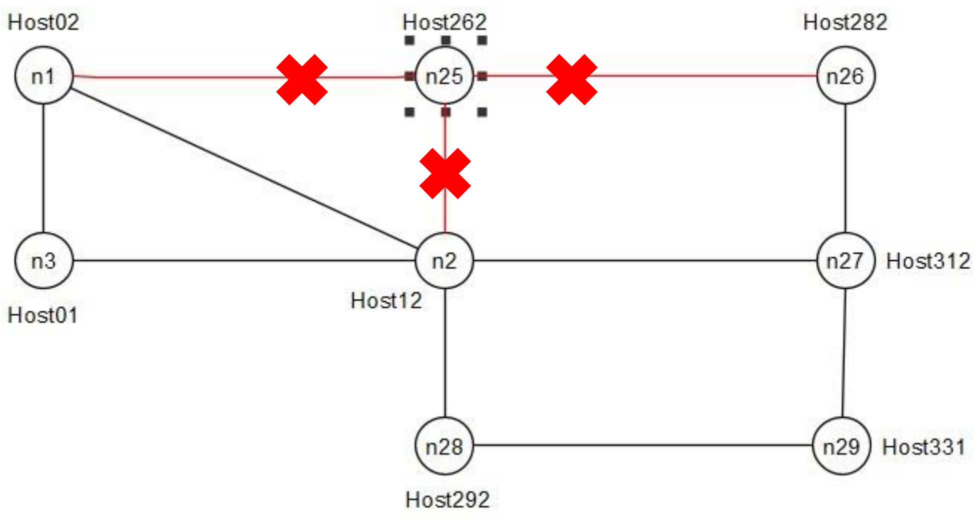


Figure 3.5:Instance of all interfaces of a node failing without causing a network partition

3.4.4 Interfaces of multiple nodes failing causing Network Partition

In this case all the interfaces of nodes n25 and n2 fail resulting in a network partition scenario as shown in Figure.3.6. The two partitions that can be observed are A(n1,n3) and B(n26,n27,n29 &n28). All nodes in each partition must mark its members as synchronized while the members of the other partition as unknown. It should also mark the host ID's of nodes whose links have failed causing the synchronized segment to be unconnected from the unknown state nodes.

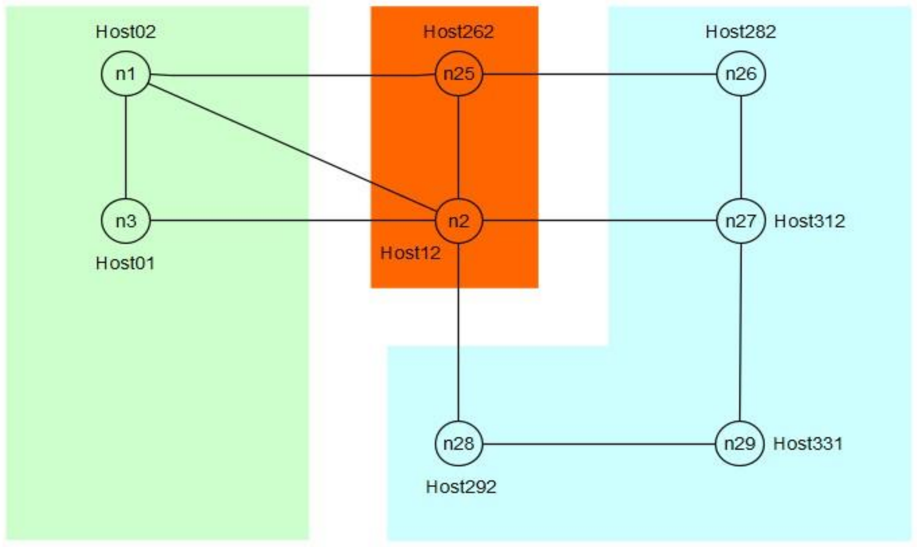


Figure 3.6:Possible case of a network partition to be dynamically detected by all nodes

3.4.5 Interface of node fails without causing Network Desynchronization

In this scenario the interface of n29 connected to node n27 is brought down at runtime some time after the network has synchronized. The entire network still manages to synchronize the resource information/keep alive statuses of n27 from the other interface. All the resource information will be synchronized and none of the node will mark a network desynchronization with n29, as indicated in Figure.3.7.

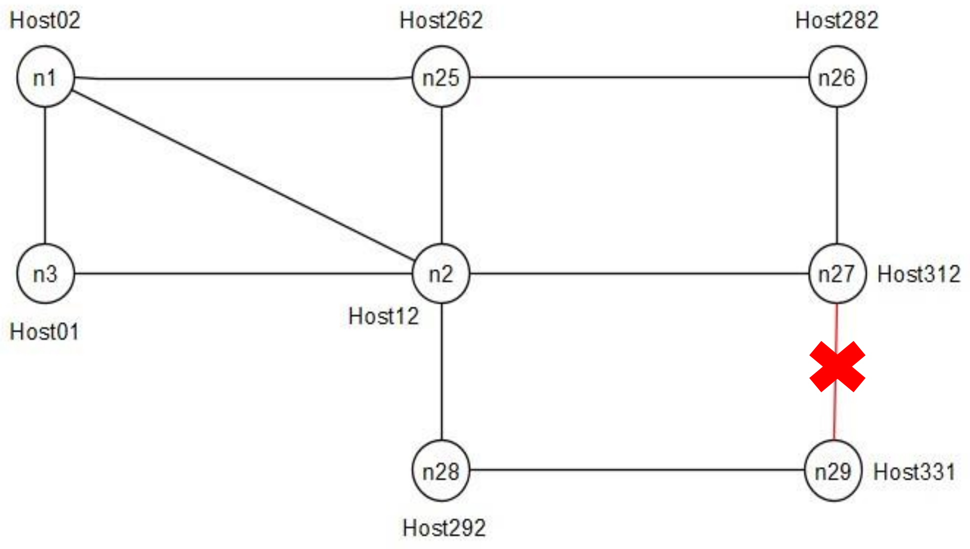


Figure 3.7:One of the interfaces of node n29 fails during runtime

3.4.6 Merging of Distinct Network Partitions

In this case initially all the interfaces of nodes n25 and n2 fail resulting in twos network partition as shown in Figure.3.8. The two partitions that can be observed are A(n1,n3) and B(n26,n27,n29 &n28). All nodes in each partition must mark its members as synchronized while the members of the other partition as unknown. After some time, all interfaces of Host262/node n25 is activated. The system should then resynchronize all the network information of partitions A and B along with Host262/node n25. Host12 is still unsynchronized with updates to resource information of the rest of the network.

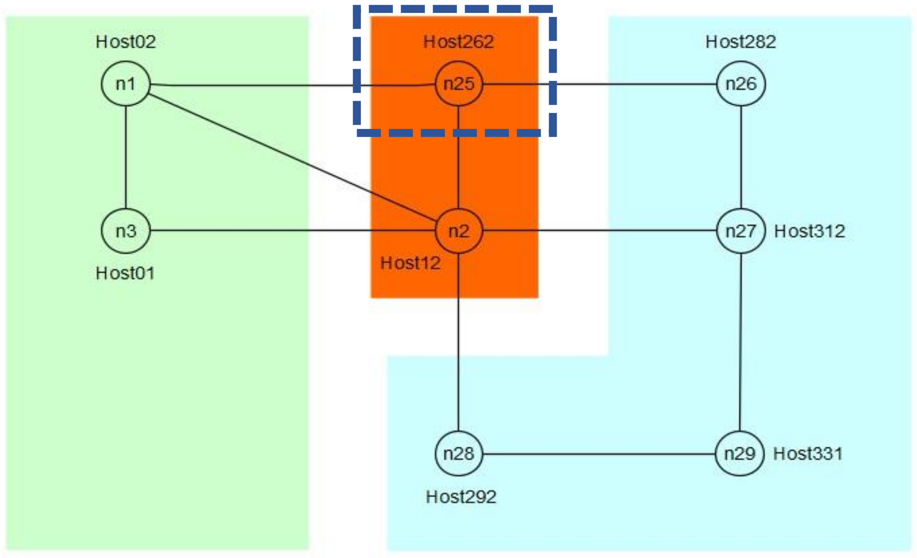


Figure 3.1:Merging of two network partitions to resynchronize all nodes by activating Host262

4 Realisation

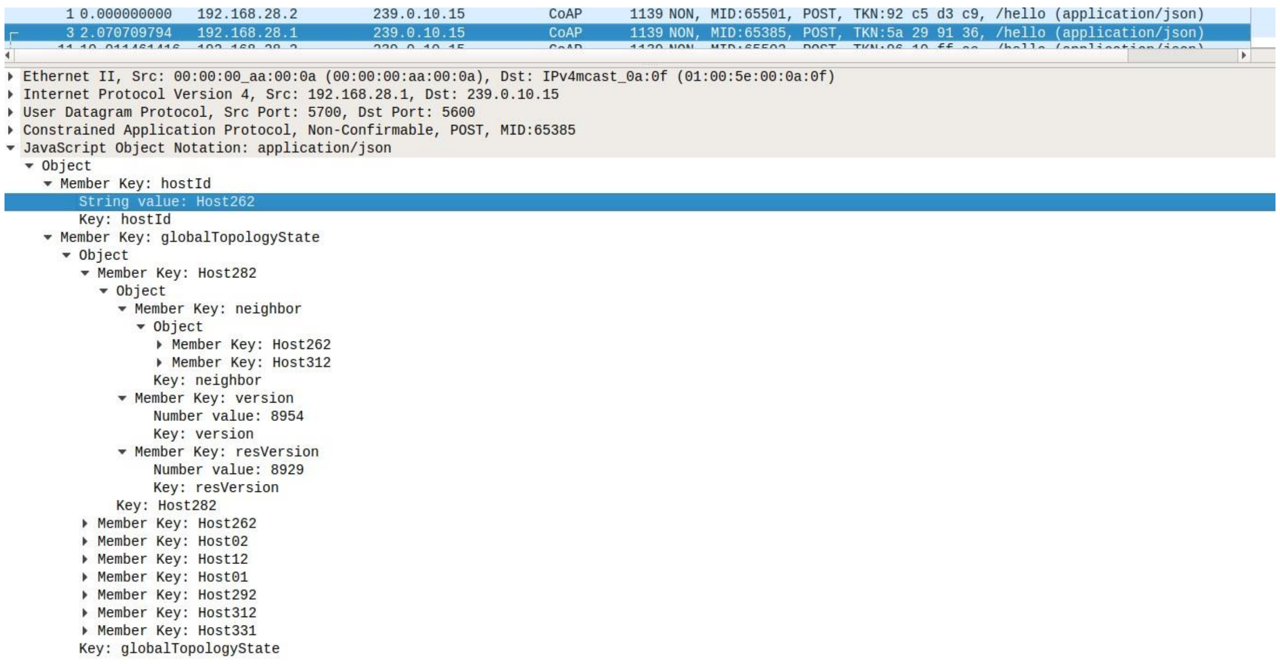
The solution fundamentally demands the knowledge about topology at each node and resources associated with each Cluster NFVO node. So, there are fundamentally two targets to be achieved : Dynamically understand topology of the entire network and update to rest of the network any changes to resource information of any individual CNFVO node. The solution employs two messages for this purpose : NS-Hello and NS-Synchronize. NS-Hello is serving two purposes : It acts as a keep-alive message to neighboring node which in turn sends that information to its downstream neighbors thereby enabling all nodes to be aware of all the nodes that are active in the network, secondly it enables all the nodes to have an asymmetrical representation of global topology (Further discussed in Section.4.3). NS-Synchronize is a reactive POST/GET type message that is sent only in two cases : When it finds that a node's local resource information has updated (POST-type) or when it detects that version of resource information of some node in network is ahead of present node's global catalog of all resource information of the nodes (GET-type). These aspects are further discussed in Section.4.2.2 and Section.4.3.

4.1 Messages and Fields

4.1.1 NS-Hello (Network Synchronization-Hello)

NS-Hello is a POST like keep alive message used to inform a neighbor's active state in synchronization process. Each node keeps track of the versions of resource information of all the nodes in the network using the NS-Hello message. It is basically a list of resource information version and neighborhood graph for all nodes in the network and the host ID of the node that is sending the message. So, each entry for a node will have primarily have a *hostID* field and a *globalTopologyState* field. For all nodes in the *globalTopologyState* field, there are primarily 3 sub-fields.

- *neighbor:* For all the nodes in the network , this will keep track of list of immediate neighbors. Only the node which is sending the NS-Hello message can modify its own entry in the field and update the nodes entire knowledge of the topology in this section and forward it to all its neighbors.
- *version:* This keeps track of the NS-Hello message version (abbr. as HV henceforth in the figures) for the given entry of host ID. Each node which receives this message from neighbor uses this version field to update its knowledge of topology of the nodes it is not directly connected to. Every node updates this field at each iteration corresponding to its own entry in the *globalTopologyState* . This field is primarily used as a marker to drop duplicate/old versions of messages from being sent in the network.
- *resVersion:* This field keeps track of the resource information version for that entry of host ID (abbr. as RV henceforth in the figures). Every time a node makes an update to its resource information ,it must update this entry in its local queues and also in this message so that other nodes in the network (e.g. newly added nodes to network) may keep track of the current version of resource information for each node.



```
  1 0.000000000   192.168.28.2        239.0.10.15         CoAP      1139 NON, MID:65501, POST, TKN:92 c5 d3 c9, /hello (application/json)
  3 2.070709794   192.168.28.1        239.0.10.15         CoAP      1139 NON, MID:65385, POST, TKN:5a 29 91 36, /hello (application/json)
```

```
▸ Ethernet II, Src: 00:00:00_aa:00:0a (00:00:00:aa:00:0a), Dst: IPv4mcast_0a:0f (01:00:5e:00:0a:0f)
▸ Internet Protocol Version 4, Src: 192.168.28.1, Dst: 239.0.10.15
▸ User Datagram Protocol, Src Port: 5700, Dst Port: 5600
▸ Constrained Application Protocol, Non-Confirmable, POST, MID:65385
▾ JavaScript Object Notation: application/json
  ▾ Object
    ▾ Member Key: hostId
        String value: Host262
        Key: hostId
    ▾ Member Key: globalTopologyState
      ▾ Object
        ▾ Member Key: Host282
          ▾ Object
            ▾ Member Key: neighbor
              ▾ Object
                ▸ Member Key: Host262
                ▸ Member Key: Host312
                Key: neighbor
            ▾ Member Key: version
                Number value: 8954
                Key: version
            ▾ Member Key: resVersion
                Number value: 8929
                Key: resVersion
            Key: Host282
        ▸ Member Key: Host262
        ▸ Member Key: Host02
        ▸ Member Key: Host12
        ▸ Member Key: Host01
        ▸ Member Key: Host292
        ▸ Member Key: Host312
        ▸ Member Key: Host331
        Key: globalTopologyState
```

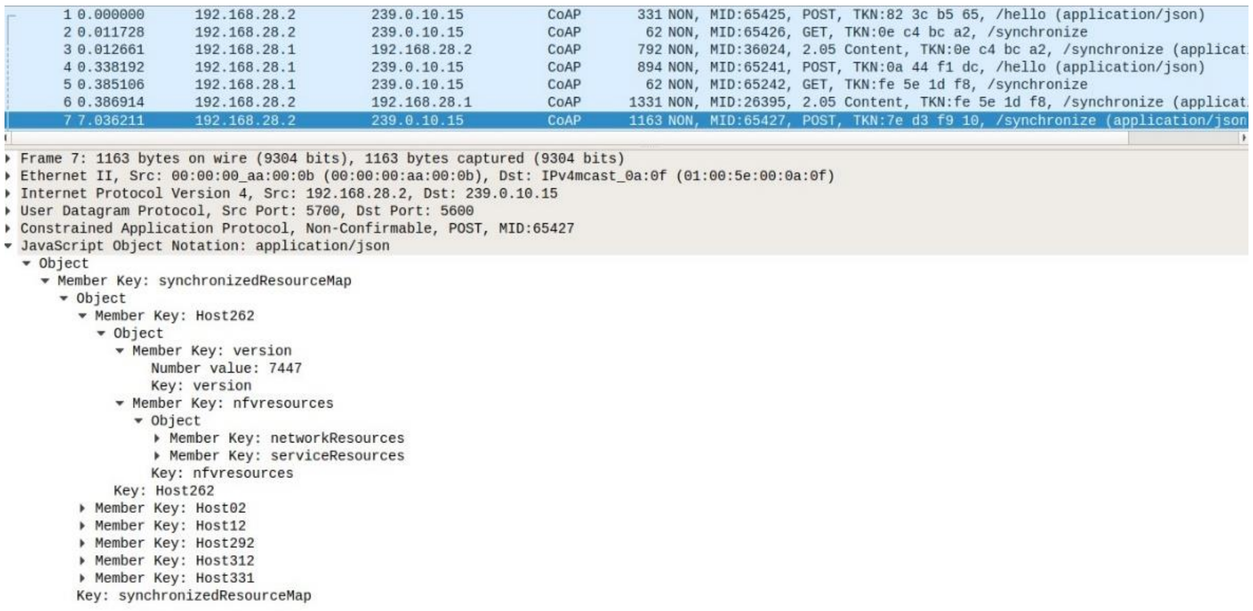
Figure 4.1:Example of a NS-Hello message sent by node n25 in Figure.3.2

4.1.2 NS-Synchronize (Network Synchronization-Synchronize)

NS-Synchronize message is sent by a node only when it detects a change of resource information in the present node. This message is flooded throughout the network with the help of a flooding queue and a scheduler in each orchestrator node. So if a node receives a NS-Synchronize message then it checks its local queue of resource information and its version and if the received information is more recent then it updates its local resource information table and pushes that entry to the flooding queue, else it drops that particular message which is possibly a duplicate. The scheduler during its scheduled time-slice for firing will check flooding queue and if there are entries present it will forward it through all of its interfaces (The process is explained in subsequent sections in more detail). Primarily the message is a list of *hostID* and *nfvresources* for all the messages in queue of a node.

- *hostID:* Host ID of the node which fired the resource information update message.
- *nfvresources:* Resource Information of the node that must be distributed throughout the network.
- *version:* Version of the Resource Information being flooded/updated. This field is equal to the *resVersion* that is used in *globalTopologyState* of NS-Hello messages corresponding to a node's entry.

In addition to the POST-like version shown in Figure.11, there is also a request type version of this message, which is queried to all immediate neighbors at the multicast address, whenever a node detects an updated version for a particular node in the NS-Hello message but has a less recent version of the resource information in its local queue. This is primarily meant as a fail-safe transaction to prevent any potential desynchronizations arising in part due to possibility of resource information updates happening during an instance of network desynchronization.



```
   1 0.000000    192.168.28.2    239.0.10.15     CoAP     331 NON, MID:65425, POST, TKN:82 3c b5 65, /hello (application/json)
   2 0.011728    192.168.28.2    239.0.10.15     CoAP      62 NON, MID:65426, GET, TKN:0e c4 bc a2, /synchronize
   3 0.012661    192.168.28.1    192.168.28.2    CoAP     792 NON, MID:36024, 2.05 Content, TKN:0e c4 bc a2, /synchronize (applicat
   4 0.338192    192.168.28.1    239.0.10.15     CoAP     894 NON, MID:65241, POST, TKN:0a 44 f1 dc, /hello (application/json)
   5 0.385106    192.168.28.1    239.0.10.15     CoAP      62 NON, MID:65242, GET, TKN:fe 5e 1d f8, /synchronize
   6 0.386914    192.168.28.2    192.168.28.1    CoAP    1331 NON, MID:26395, 2.05 Content, TKN:fe 5e 1d f8, /synchronize (applicat
   7 7.036211    192.168.28.2    239.0.10.15     CoAP    1163 NON, MID:65427, POST, TKN:7e d3 f9 10, /synchronize (application/json

▸ Frame 7: 1163 bytes on wire (9304 bits), 1163 bytes captured (9304 bits)
▸ Ethernet II, Src: 00:00:00_aa:00:0b (00:00:00:aa:00:0b), Dst: IPv4mcast_0a:0f (01:00:5e:00:0a:0f)
▸ Internet Protocol Version 4, Src: 192.168.28.2, Dst: 239.0.10.15
▸ User Datagram Protocol, Src Port: 5700, Dst Port: 5600
▸ Constrained Application Protocol, Non-Confirmable, POST, MID:65427
▾ JavaScript Object Notation: application/json
  ▾ Object
    ▾ Member Key: synchronizedResourceMap
      ▾ Object
        ▾ Member Key: Host262
          ▾ Object
            ▾ Member Key: version
                Number value: 7447
                Key: version
            ▾ Member Key: nfvresources
              ▾ Object
                ▸ Member Key: networkResources
                ▸ Member Key: serviceResources
                Key: nfvresources
            Key: Host262
        ▸ Member Key: Host02
        ▸ Member Key: Host12
        ▸ Member Key: Host292
        ▸ Member Key: Host312
        ▸ Member Key: Host331
        Key: synchronizedResourceMap
```

Figure 4.2:An NS-Synchronize message sent by node n26 in the Figure.3.2 topology at the interface 192.168.28.2

4.2 Solution Architecture

4.2.1 Software Component Description

The software solution fundamentally consists of a Synchronization Interface (SI) module, which is a CoAP handler and resource bundle which is triggered whenever the physical interface associated with a CNFVO node receives an NS-Hello or NS-Synchronize CoAP message. The scheduler and Breadth First Search Algorithm (BFS) modules are responsible for scheduling message/state table updates and identifications for partitions in the network topology. The component structure is represented in Figure.4.3

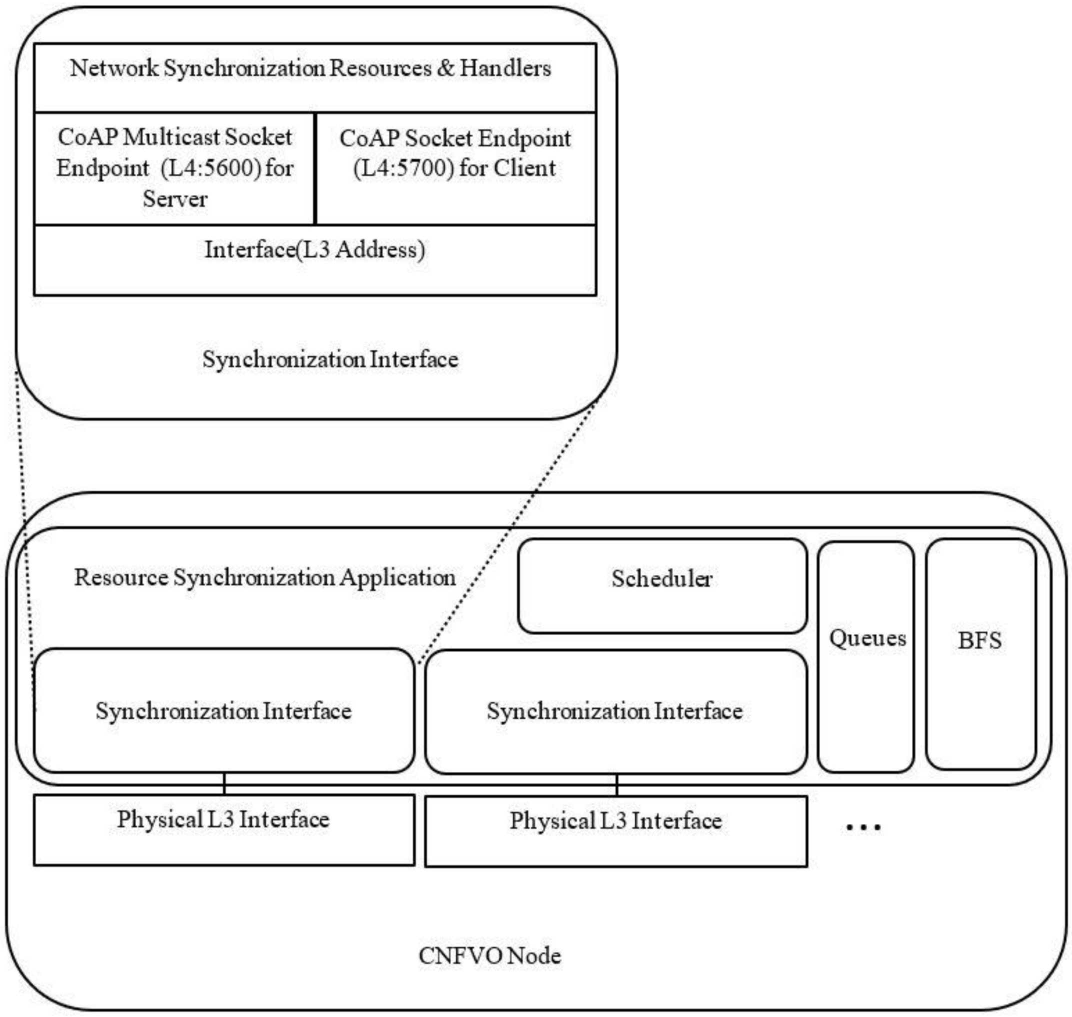

Network Synchronization Resources & Handlers
CoAP Multicast Socket Endpoint (L4:5600) for Server
CoAP Socket Endpoint (L4:5700) for Client
Interface(L3 Address)
Synchronization Interface
Resource Synchronization Application
Scheduler
Queues
BFS
Synchronization Interface
Synchronization Interface
Physical L3 Interface
Physical L3 Interface
...
CNFVO Node

Figure 4.3:Interface and Component Overview

The Synchronization Interface(SI) module corresponding to the physical interfaces which must be enabled with the synchronization procedure. Each SI has a series of CoAP event handlers and triggers to initiate processing or sending of incoming and outgoing messages, respectively. The message sending and state estimation are triggered via a scheduler which fires up these modules within the application. The scheduler also periodically checks for updates to its local resource information file and fires up the flooding mechanism whenever it detects a change in its local resource information. The information regarding topology and resource information are stored in queues and maps within the application which are persisted by the scheduler during periodic intervals or whenever an update happens. The BFS module is used to compute the connectedness of the topology information available to the node and to estimate the states of different nodes within the network which are used to identify if a network partition has taken place or the list of nodes that are synchronized in the current partition.

The UML diagram in Figure.4.4 shows the class diagram of the software components of *Hello* message structure, *Synchronize* message structure, the class hierarchy of *UDPMulticastEndpoint* multicast datagram sockets (Matthias Kovatsch, n.d.) associated with *SynchronizationInterface* structure (within the *CoapEndpoint* fields) . The *BFSAlg* structure represents the method fields to compute node connectivity and network-partitions.

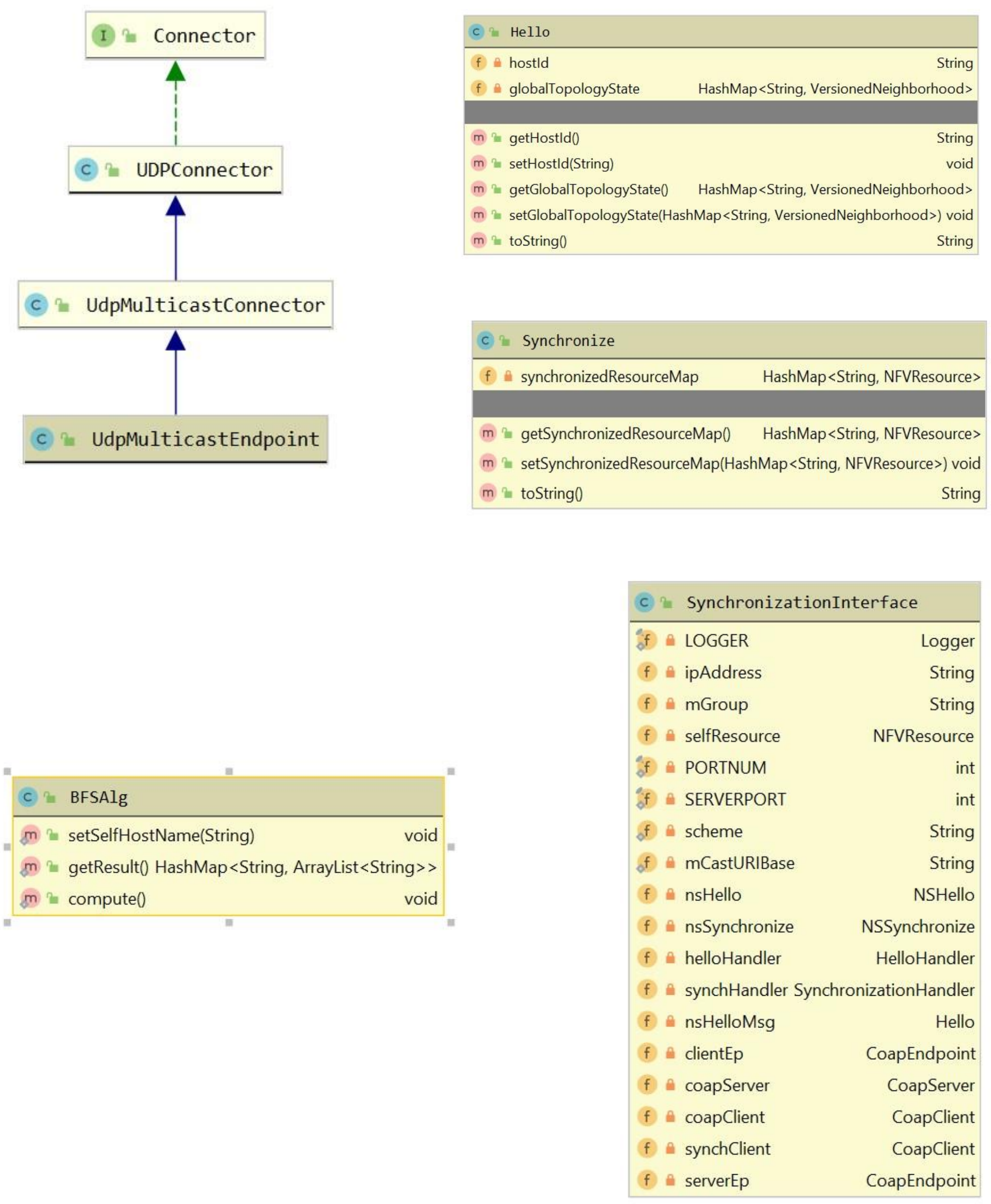

Connector
UDPConnector
UdpMulticastConnector
UdpMulticastEndpoint
Hello
hostId String
globalTopologyState HashMap<String, VersionedNeighborhood>
getHostId() String
setHostId(String) void
getGlobalTopologyState() HashMap<String, VersionedNeighborhood>
setGlobalTopologyState(HashMap<String, VersionedNeighborhood>) void
toString() String
Synchronize
synchronizedResourceMap HashMap<String, NFVResource>
getSynchronizedResourceMap() HashMap<String, NFVResource>
setSynchronizedResourceMap(HashMap<String, NFVResource>) void
toString() String
SynchronizationInterface
LOGGER Logger
ipAddress String
mGroup String
selfResource NFVResource
PORTNUM int
SERVERPORT int
scheme String
mCastURIBase String
nsHello NSHello
nsSynchronize NSSynchronize
helloHandler HelloHandler
synchHandler SynchronizationHandler
nsHelloMsg Hello
clientEp CoapEndpoint
coapServer CoapServer
coapClient CoapClient
synchClient CoapClient
serverEp CoapEndpoint
BFSAlg
setSelfHostName(String) void
getResult() HashMap<String, ArrayList<String>>
compute() void

Figure 4.4:UML diagrams of relationships of Software components

4.2.2 Flooding Mechanism

The scheduler in a node fires up the flooding mechanism at periodic intervals(in the current implementation every ~10 sec) if any NS-Synchronize message is present in the flooding queue. Each node on an update of its local resource information updates its resource version ID and sends the updated resource information to all its neighbors. The receiving node in turn checks the resource information version and compares it with the version present in its own local map of resource information. If it finds that resource information with the given version is present ,it drops it. If the received packet is of a later version then, it updates its own local resource information queues and maps and pushes the information to the flooding queue.

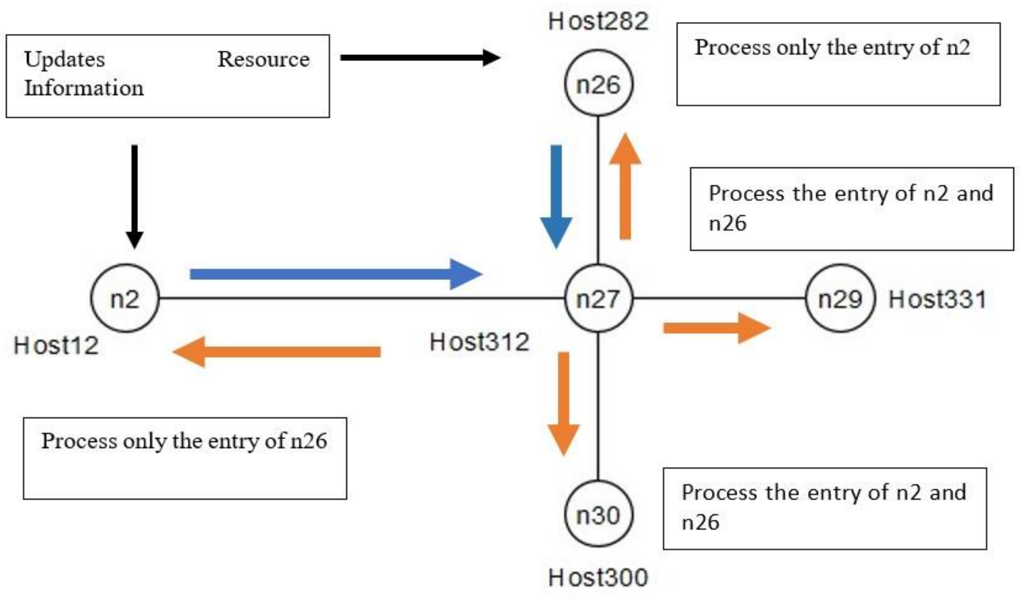

Updates Resource Information
Host282
n26
Process only the entry of n2
Process the entry of n2 and n26
n2
n27
n29
Host331
Host12
Host312
Process only the entry of n26
n30
Process the entry of n2 and n26
Host300

Figure 4.5:Mechanism and scheduling of flooding queue

During the scheduled time-slice for flooding it will forward all the packets in the queue (in a concatenated message) to all its neighbors. These neighbors will in turn check version of entries to determine if they should be pushed to the flooding queue or not. In Figure.4.5, the process of n27's scheduler when it receives an NS-Synchronize update message from n26 and n2 is shown. It updates its own local information tables and floods two entries to all the nodes in its neighborhood. Nodes n26 and n2 receive duplicated copies of its own information which it drops, but updates the resource information of the other entry, meanwhile n27,n29 and n30 processes both the resource information update and floods as mentioned above. At steady state , all the nodes will have updated its resource information tables and would no longer flood the NS-Synchronize messages.

4.2.3 Connectedness and State Estimation

To estimate whether all the nodes which are present in the global resource information view of a cluster orchestrator element are still actively synchronized , each node must be able to independently compute its connectedness to every other Cluster NFVO (CNFVO) node. The way in which this study proposes to achieve this is by creating an asymmetric connected graph of network topology at global NFVO layer. Then the graph is pruned to estimate a single connected path based on a metric of equal cost of traversal. In that regard it may be considered comparable to OSPF used in network layer routing (Moy, n.d.), with the cost of traversal being equal among all the paths. In this case however pruning needs to be done via the simplest factor possible which is to check if the path has already been factored in the previous iteration of the computation. However, the standard interpretation of the algorithm on a symmetric graph makes it unsuitable to determine elements involved in network partition and classifying them. So, the topology view that each node persists must be modified to enable a deeper analysis of the underlying topology than the one which is offered by a consistent symmetric graph of the network. Each node persists as a result a view of topology that consists of neighbors which are connected directly to it and aggregates this view from all nodes of the network thereby generating a global view of the network topology. However in an instance of separation of network segments the elements consisting in the foreign sector wont be able to communicate its loss of connectivity to the present sector, so each sector on its own, will view itself as being disconnected from the foreign sector but the entries connected to the foreign sector will still appear as connected to the present sector, thereby creating an asymmetric representation of topology that may be used by every node to estimate nodes that lost synchronization to the present sector and the nodes lying downstream to the foreign sector whose resource information cannot be possibly identified as there is no path to the foreign sector. In any other representation of topology one can only estimate the set of CNFVO's that are not desynchronized thereby clubbing them into just one category, however in this asymmetric view of the topology, one can estimate the desynchronization that was responsible for the partition to occur in the first place and differentiate them from the nodes that lie downstream to that desynchronized node.

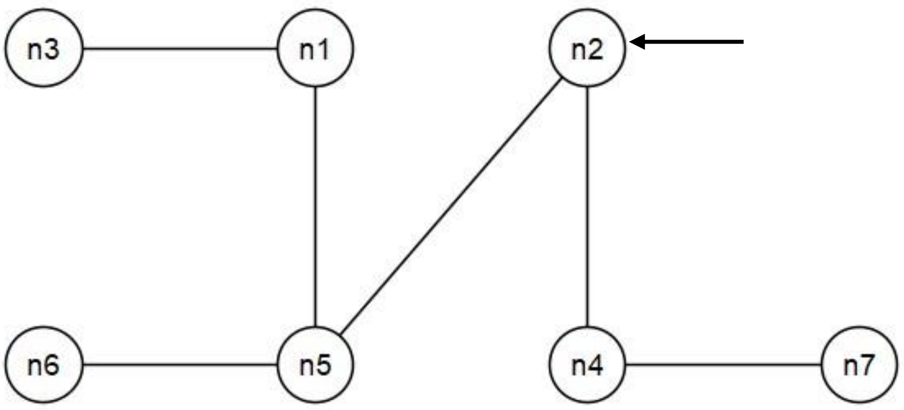


Figure 4.6:Example topology for BFS algorithm computation demonstration at n2

Table.4.1 shows global topology view of node n2 for the topology represented in Figure.4.6 , which is used for computation of connectivity and state estimation by the BFS module. The procedure that BFS module undergoes is explained in Table.4.2.

Table 4.1:Topology view at n2

Node	**Neighbors**
n2	n4,n5
n1	n5,n3
n5	n2,n6,n1
n4	n2,n7
n3	n1
n6	n5
n7	n4

Table.4.2 shows the procedure of BFS algorithm which maintains a flow queue and a result word queue which categorizes neighbors and entries of the global topology view. The entries of result word queue are the unique elements of the flow queue and flow queue iteratively fills itself with all neighbors eliminating only the ones which are not present in result word queue. The resulting connectivity estimation will look as shown below.

Table 4.2:Queue Entries for connectedness computation at n2

Flow Queue	**Neighbors**	**Result Queue**
n2	n4,n5	n2,n4,n5
n4	n2,n7	n7
n5	n2,n6,n1	n6,n1
~~n2~~		
n7	n4	
~~n2~~		
n6	n5	
n1	n3,n5	n3
~~n4~~		
~~n5~~		
n3	n1	
~~n5~~		
~~n1~~		

The algorithm terminates when there are no new elements in the flow queue. Aggregated Result Queue word found as a result would be (n2,n4,n5,n7,n6,n1,n3) ,thus demonstrating that the node n2 is connected to all the nodes in the network.

Figure.4.7 shows a scenario which represents a network partition scenario when looked at from the point of view of node n2.

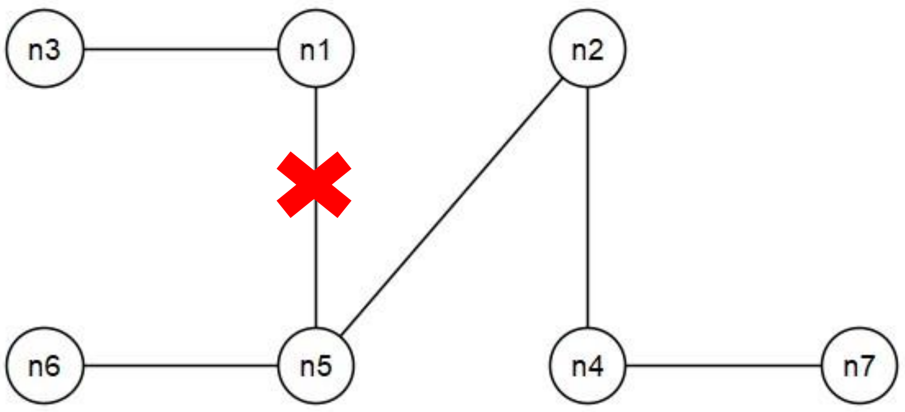


Figure 4.7:Network Partition occurring in the topology and resulting connectedness estimation

Table.4.3 represents what an asymmetric global topology view looks like. The topology state table is considered from the point of view of n2.The entry of neighbors for node n5 does not represent n1 as its neighbor as the connection is disabled between those two nodes, but node n1 shows n5 as a neighbor as n2 lost connection to n1 and there is no way by which node n1 could have indicated to n2 that it lost connection to n5. So n2 maintains the old entry for n1's neighbor which leads to a resultant asymmetric view of the global topology shown in Table.4.3. The procedure involved in constructing the global topology view is explained in detail in Section.4.3

Table 4.3:Network Topology view at n2 during an instance of network partition

Node	Neighbors
n2	n4,n5
n1	n5,n3
n5	n2,n6,~~n1~~
n4	n2,n7
n3	n1
n6	n5
n7	n4

Table.4.4

Table 4.4:Queue Entries for connectedness computation at n2 during a network partition

Flow Queue	Neighbors	Result Queue
n2	n4,n5	n2,n4,n5
n4	n2,n7	n7
n5	n2,n6	n6
~~n2~~		
n7	n4	
~~n2~~		
n6	n5	
~~n4~~		
~~n5~~		

The algorithm terminates when there are no new elements in the flow queue. Aggregated Result Queue word found as a result would be (n2,n4,n5,n7,n6) ,thus demonstrating that the node n2 belongs to a partition in the network and there would be set of nodes in the topology table of n2 which would be desynchronized to resource information changes in the partition of n2. So, the secondary sequence is activated which determines which nodes were responsible for partition to occur and which nodes are lying downstream of partition segment.

Table 4.5:Determination of states of nodes not belonging to n2's network partition segment.

Node	Neighbors
n2	n4,n5
n1	n5,n3
n5	n2,n6

n4	n2,n7
n3	n1
n6	n5
n7	n4

By comparing nodes of the topology view with the nodes found in the partition , it is observed that n1 contains one of asymmetric entry of neighbors as n5 which indicates that the present node's (node n2) partition was desynchronized from n1 which resulted in the partitioning of the network. The second connected node to the partition namely n3 is lying downstream from the present partition segment thus getting identified as a possible node in foreign partition.

4.3 Transactions and Sessions

4.3.1 NS-Hello

The message interactions involved in synchronizing topology information throughout the network is analyzed with the topology shown in Figure.4.8. The system consists of three nodes A,B and C which needs to understand the linear topology via the NS-Hello messages.

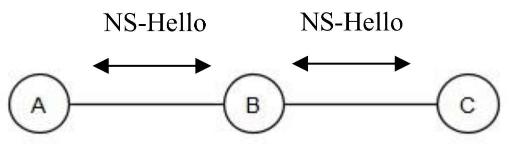


Figure 4.8: Example Topology for NS-Hello procedure.

Figure.4.9 shows the *globalTopologyState* field entries for NS-Hello messages sent between A,B and C. Assume the interactions are taking place in order as shown, however for any other order of events happening as well the end result would be the same but not all of those permutations can be possibly discussed. In the first NS-Hello message sent by A to B, A is not aware of any neighbors so its neighbor list is denoted as empty in the *globalTopologyState* field, it sends the other fields as per individual configurations namely Hello Version, Resource Version and Host ID. In the same manner node C also sends an NS-Hello message to its neighbor namely B. So, in the end node B would have computed A and C as its direct neighbors in its topology state vector. The entries for A and C would still be empty since neither A nor C have received NS-Hello message from node B which is the directly connected neighbor to those nodes. Now node B sends NS-Hello message to A and C with its topology state vector listing A and C as neighbors for node B. Then nodes A and C computes respectively B as direct neighbor to itself but A would not be aware of node B being direct neighbor to node C and vice versa for the case of node C's interpretation of node A. So, the resultant topology state vectors look like Figure.4.10.

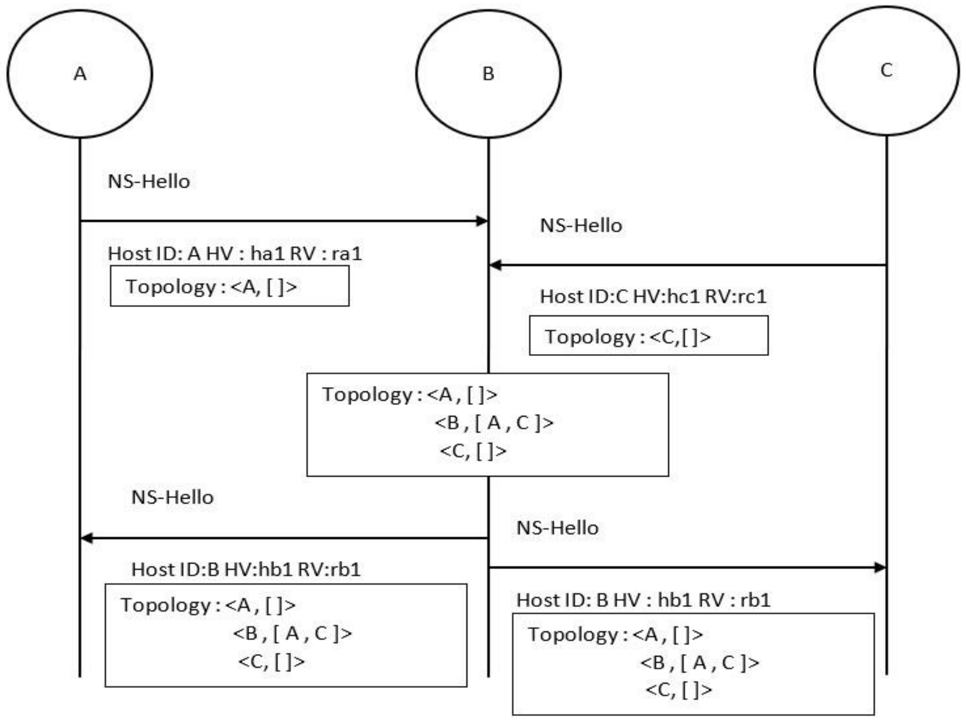

A
B
C
NS-Hello
Host ID: A HV : ha1 RV : ra1
Topology : <A, [ ]>
NS-Hello
Host ID:C HV:hc1 RV:rc1
Topology : <C,[ ]>
Topology : <A , [ ]> <B , [A , C]> <C, [ ]>
NS-Hello
Host ID:B HV:hb1 RV:rb1
Topology : <A , [ ]> <B , [A , C]> <C, [ ]>
NS-Hello
Host ID: B HV : hb1 RV : rb1
Topology : <A , [ ]> <B , [A , C]> <C, [ ]>

Figure 4.9:NS-Hello Operation

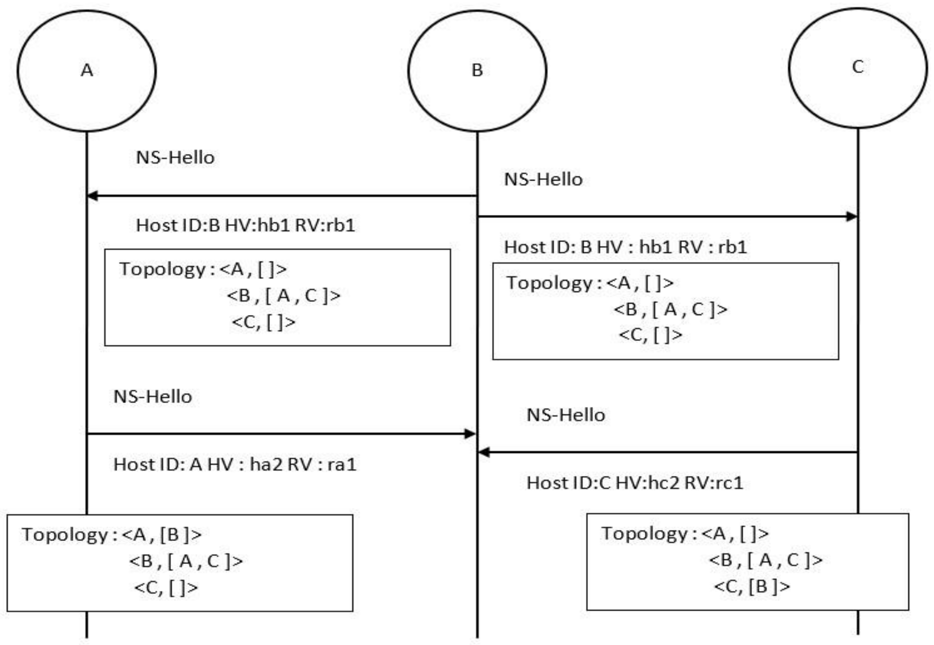

A
B
C
NS-Hello
Host ID:B HV:hb1 RV:rb1
Topology : <A , [ ]> <B , [A , C]> <C, [ ]>
NS-Hello
Host ID: B HV : hb1 RV : rb1
Topology : <A , [ ]> <B , [A , C]> <C, [ ]>
NS-Hello
Host ID: A HV : ha2 RV : ra1
Topology : <A , [B]> <B , [A , C]> <C, [ ]>
NS-Hello
Host ID:C HV:hc2 RV:rc1
Topology : <A , [ ]> <B , [A , C]> <C, [B]>

Figure 4.10:NS-Hello Operation

Figure.4.11 shows the final stage in identification of topology across all the nodes. Node B identifies the entire topology and in the resultant NS-Hello enables nodes A and C to identify neighbors of all nodes in the network.

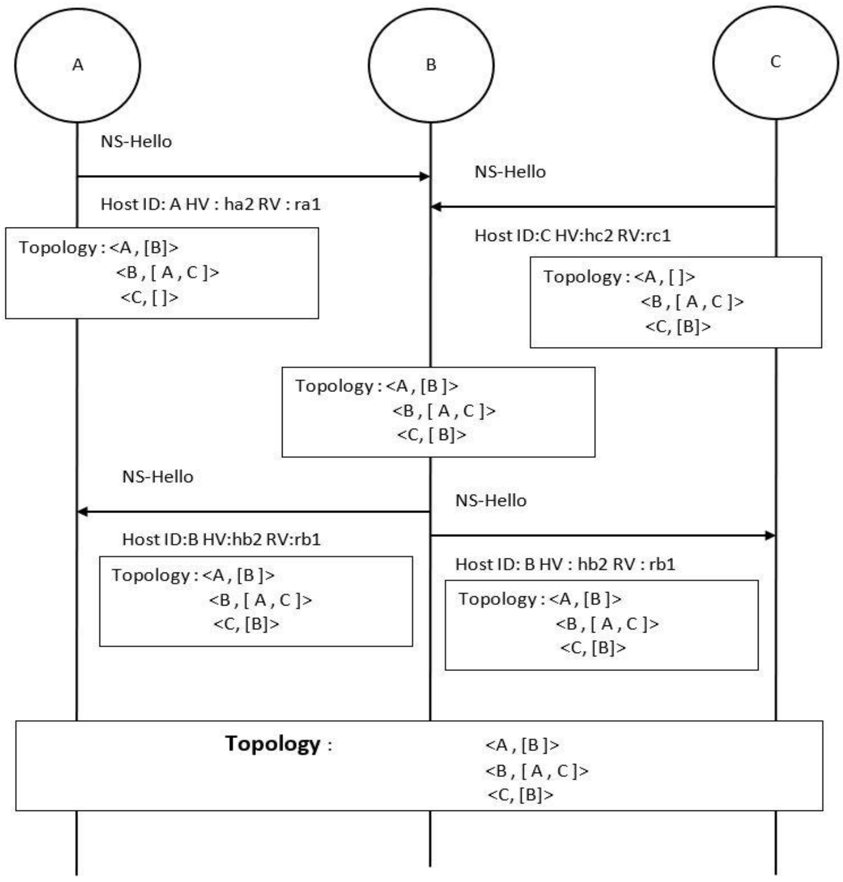

A
B
C
NS-Hello
Host ID: A HV : ha2 RV : ra1
NS-Hello
Host ID:C HV:hc2 RV:rc1
Topology : <A, [B]> <B, [A, C]> <C, [ ]>
Topology : <A, [ ]> <B, [A, C]> <C, [B]>
Topology : <A, [B]> <B, [A, C]> <C, [B]>
NS-Hello
Host ID:B HV:hb2 RV:rb1
NS-Hello
Host ID: B HV : hb2 RV : rb1
Topology : <A, [B]> <B, [A, C]> <C, [B]>
Topology : <A, [B]> <B, [A, C]> <C, [B]>
Topology : <A, [B]> <B, [A, C]> <C, [B]>

Figure 4.11:NS-Hello Operation

4.3.2 NS-Synchronize

Figure.4.12 shows an example scenario explaining the messages exchanged when node D updates its resource information and the application attempts to synchronize the information to rest of the nodes. So, node D synchronizes its resource information update via flooding. So, the black marked NS-Synchronize message is sent to the neighbor of D which is flooded to the rest of the network via the indigo marked NS-Synchronize flooding messages. Each node drops the duplicates which are received and floods only those which are not present in a node's global resource information table.

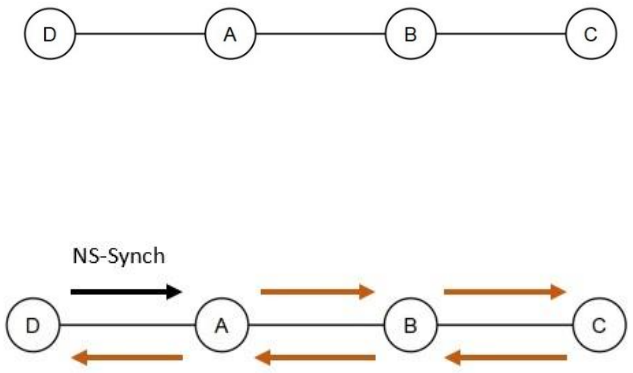
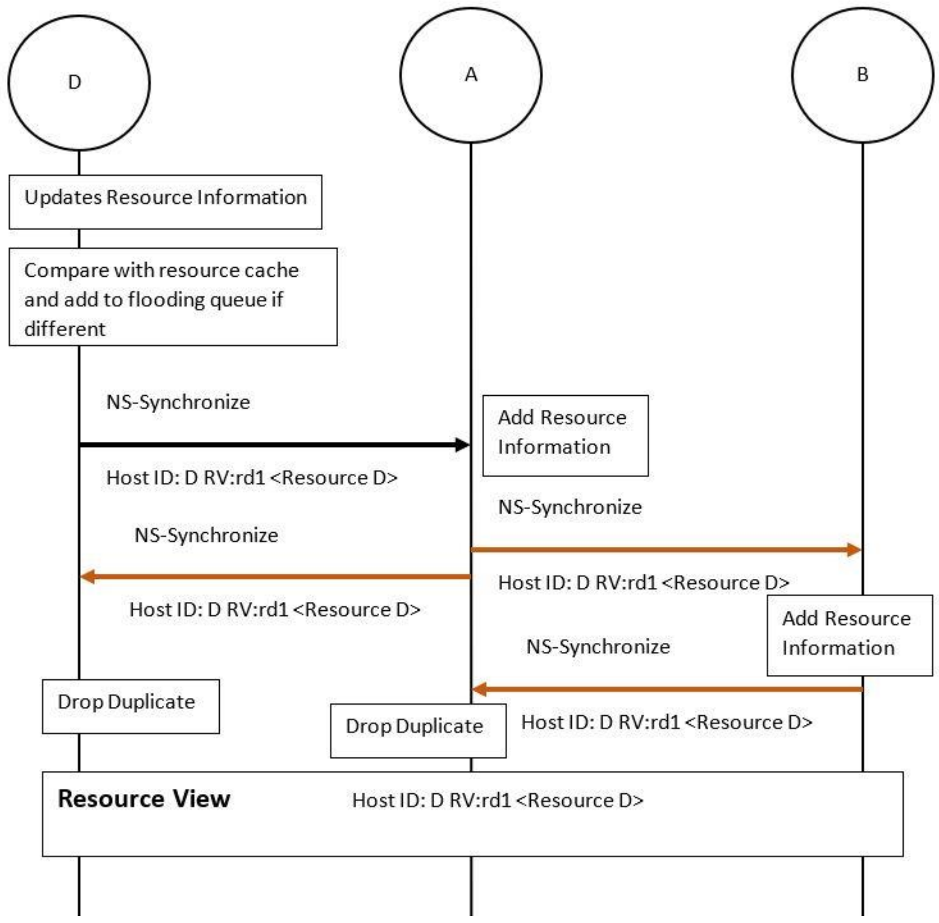


Figure 4.12:NS-Synchronize Operation Example topology

Figure.4.13 shows how each of the nodes in the network updates its resource information table with the flooded messages. It shows how nodes A and B react to the NS-Synchronize messages representing update of node B's resource information and Figure.4.13 shows similarly how the resource information is updated across A-B-C sector. The indigo marked arrows represent the flooded message and black marked ones represent the original update message sent by node D to its neighbor (in this case node A) as represented in 4.12.


D
A
B
Updates Resource Information
Compare with resource cache and add to flooding queue if different
NS-Synchronize
Host ID: D RV:rd1 <Resource D>
Add Resource Information
NS-Synchronize
Host ID: D RV:rd1 <Resource D>
NS-Synchronize
Host ID: D RV:rd1 <Resource D>
Add Resource Information
NS-Synchronize
Host ID: D RV:rd1 <Resource D>
Drop Duplicate
Drop Duplicate
Resource View
Host ID: D RV:rd1 <Resource D>

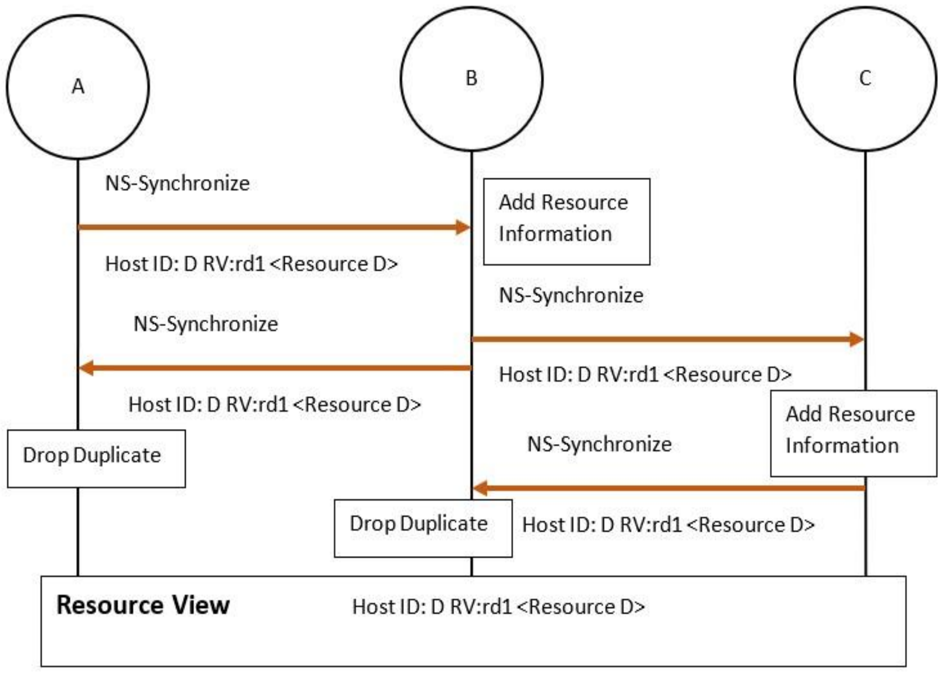

A
B
C
NS-Synchronize
Host ID: D RV:rd1 <Resource D>
Add Resource Information
NS-Synchronize
Host ID: D RV:rd1 <Resource D>
NS-Synchronize
Host ID: D RV:rd1 <Resource D>
Add Resource Information
NS-Synchronize
Host ID: D RV:rd1 <Resource D>
Drop Duplicate
Drop Duplicate
Resource View
Host ID: D RV:rd1 <Resource D>

Figure 4.13:NS-Synchronize Operation

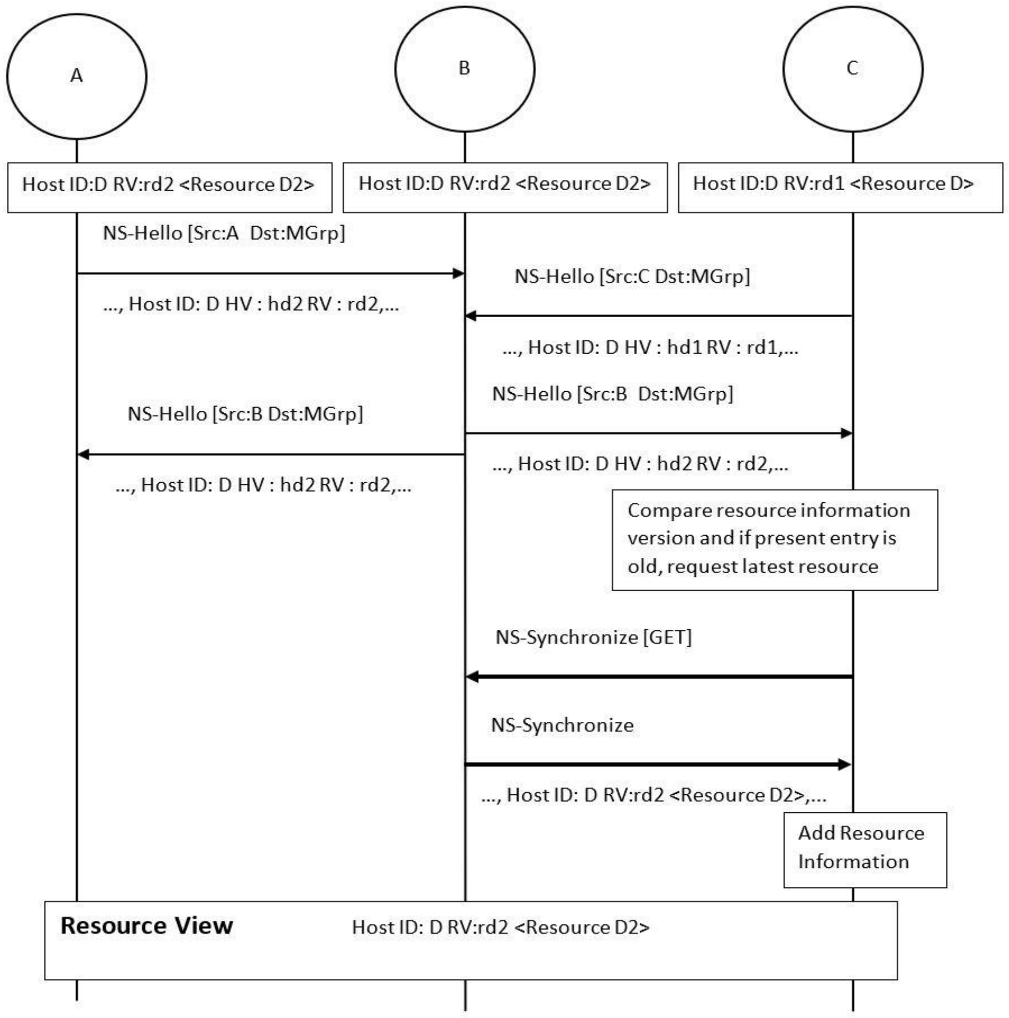

A
B
C
Host ID:D RV:rd2 <Resource D2>
Host ID:D RV:rd2 <Resource D2>
Host ID:D RV:rd1 <Resource D>
NS-Hello [Src:A Dst:MGrp]
..., Host ID: D HV : hd2 RV : rd2,...
NS-Hello [Src:C Dst:MGrp]
..., Host ID: D HV : hd1 RV : rd1,...
NS-Hello [Src:B Dst:MGrp]
..., Host ID: D HV : hd2 RV : rd2,...
NS-Hello [Src:B Dst:MGrp]
..., Host ID: D HV : hd2 RV : rd2,...
Compare resource information version and if present entry is old, request latest resource
NS-Synchronize [GET]
NS-Synchronize
..., Host ID: D RV:rd2 <Resource D2>,...
Add Resource Information
Resource View
Host ID: D RV:rd2 <Resource D2>

Figure 4.14:NS-Synchronize Operation

Figure.4.14 shows an alternate path which may be taken by application on a node to update its resource information, if the version update information is propagated via the NS-Hello transactions first and then the resource information happens via NS-Synchronize request type of messages.

4.4 Analysis of Use Cases

4.4.1 Updating Resource Information of a node at runtime

When n3 makes a change to its resource information, it floods it to the rest of the network using NS-Synchronize messages. Node n3 would send an update NS-Synchronize message to its direct neighbors n2 and n1which in turn floods it as per flooding mechnism to rest of the network. Then in the NS-Hello procedure all the resource information version would be updated in the subsequent keep-alive messages.

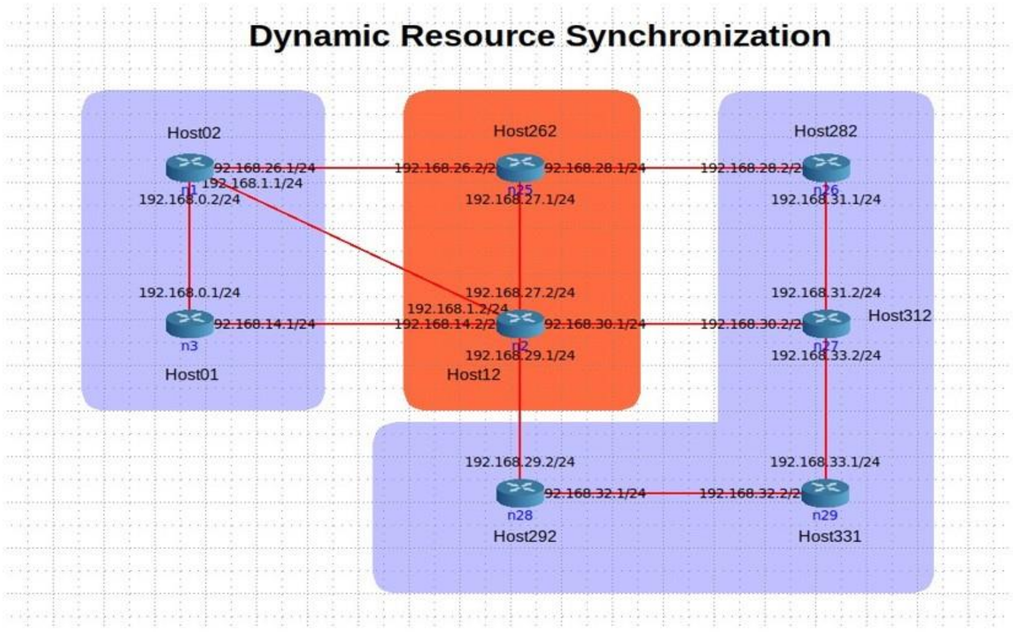


Figure 4.15:Node n3 (Host01) updates resource information at runtime.

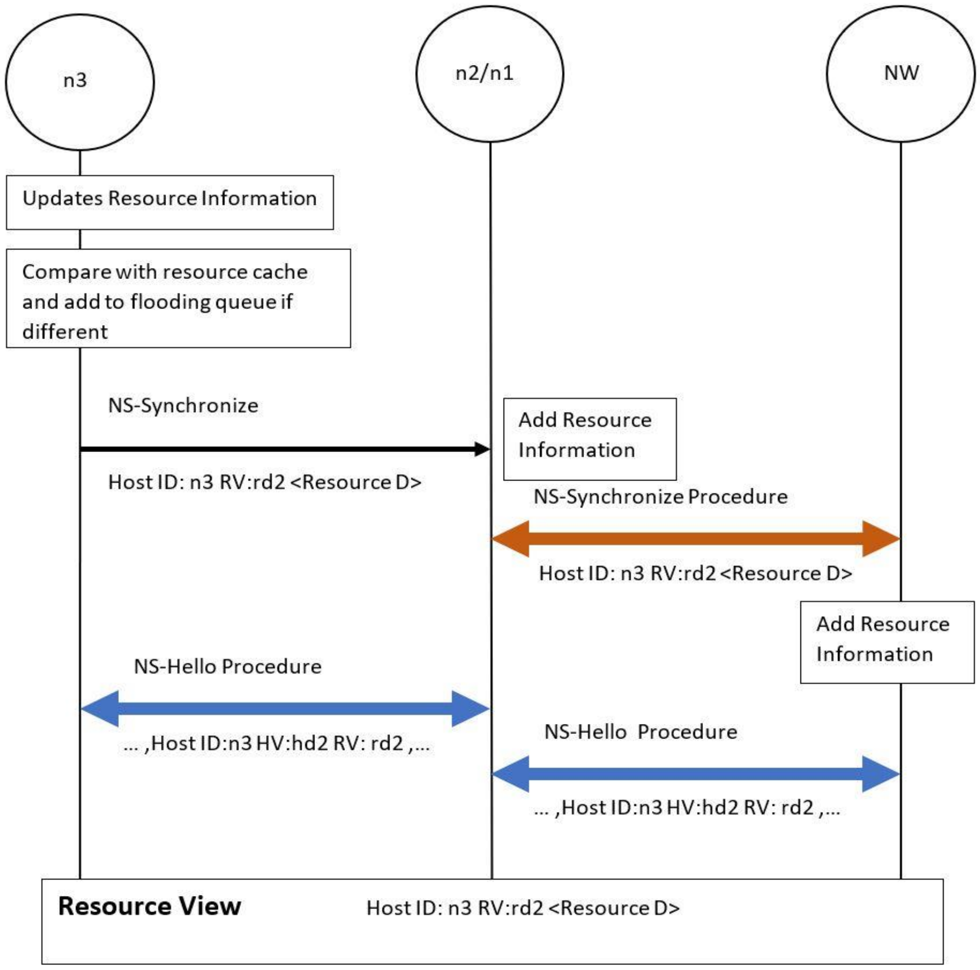
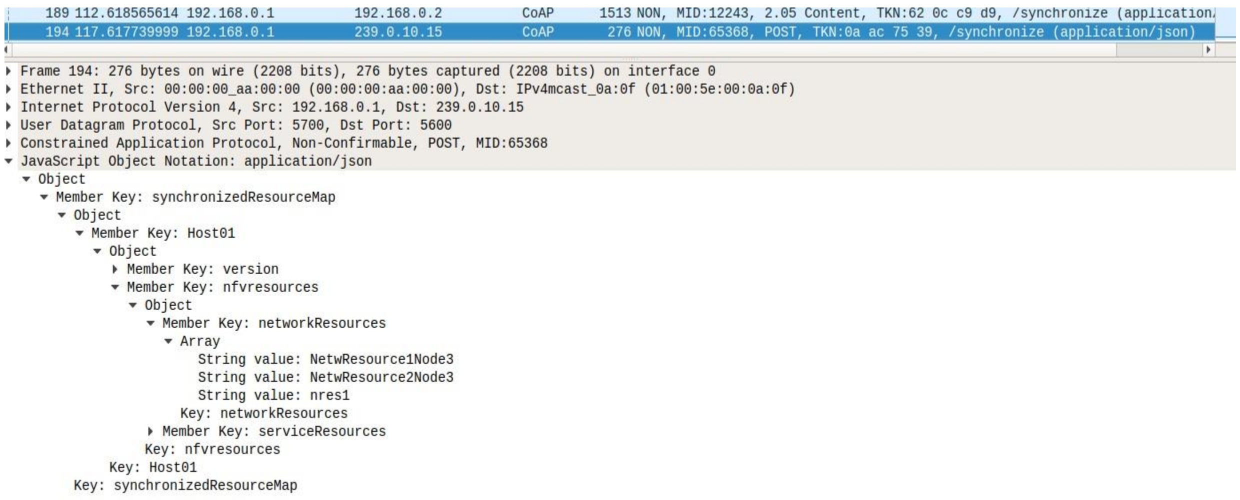

n3
n2/n1
NW
Updates Resource Information
Compare with resource cache and add to flooding queue if different
NS-Synchronize
Host ID: n3 RV:rd2 <Resource D>
Add Resource Information
NS-Synchronize Procedure
Host ID: n3 RV:rd2 <Resource D>
Add Resource Information
NS-Hello Procedure
... ,Host ID:n3 HV:hd2 RV: rd2 ,...
NS-Hello Procedure
... ,Host ID:n3 HV:hd2 RV: rd2 ,...
Resource View
Host ID: n3 RV:rd2 <Resource D>

Figure 4.16:Transactions involved in a resource information update at runtime. (NW denotes rest of the network)



```
  189 112.618565614 192.168.0.1          192.168.0.2          CoAP      1513 NON, MID:12243, 2.05 Content, TKN:62 0c c9 d9, /synchronize (application
  194 117.617739999 192.168.0.1          239.0.10.15          CoAP       276 NON, MID:65368, POST, TKN:0a ac 75 39, /synchronize (application/json)

▸ Frame 194: 276 bytes on wire (2208 bits), 276 bytes captured (2208 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_aa:00:00 (00:00:00:aa:00:00), Dst: IPv4mcast_0a:0f (01:00:5e:00:0a:0f)
▸ Internet Protocol Version 4, Src: 192.168.0.1, Dst: 239.0.10.15
▸ User Datagram Protocol, Src Port: 5700, Dst Port: 5600
▸ Constrained Application Protocol, Non-Confirmable, POST, MID:65368
▾ JavaScript Object Notation: application/json
  ▾ Object
    ▾ Member Key: synchronizedResourceMap
      ▾ Object
        ▾ Member Key: Host01
          ▾ Object
            ▸ Member Key: version
            ▾ Member Key: nfvresources
              ▾ Object
                ▾ Member Key: networkResources
                  ▾ Array
                      String value: NetwResource1Node3
                      String value: NetwResource2Node3
                      String value: nres1
                    Key: networkResources
                ▸ Member Key: serviceResources
                Key: nfvresources
            Key: Host01
        Key: synchronizedResourceMap
```

Figure 4.17:Resource Information update NS-Synchronize message sent by n3 to n1 when n3 adds a network resource 'nres1' denoted by black arrow in Figure.4.16.

4.4.2 Node injected into an already synchronized network

When a node n29 is injected into an already synchronized network, it floods the network with its resource information using NS-Synchronize messages. Node n29 gets information of the rest of the network from NS-Synchronize request type message sent to multicast address of its neighbor which was already synchronized with rest of network's information. In the end the entire network is aware of n29's resources and its neighborhood relationships with n27 & n28.

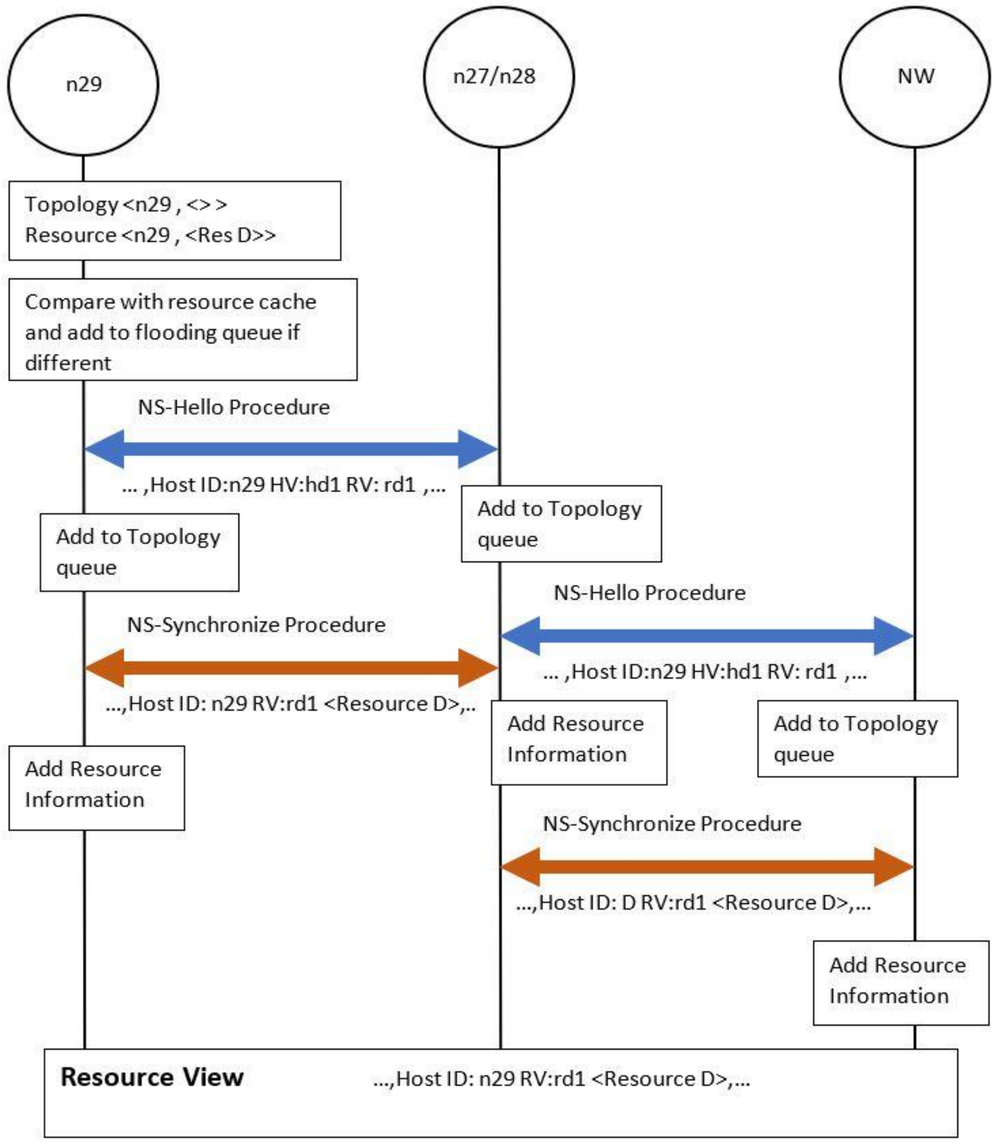
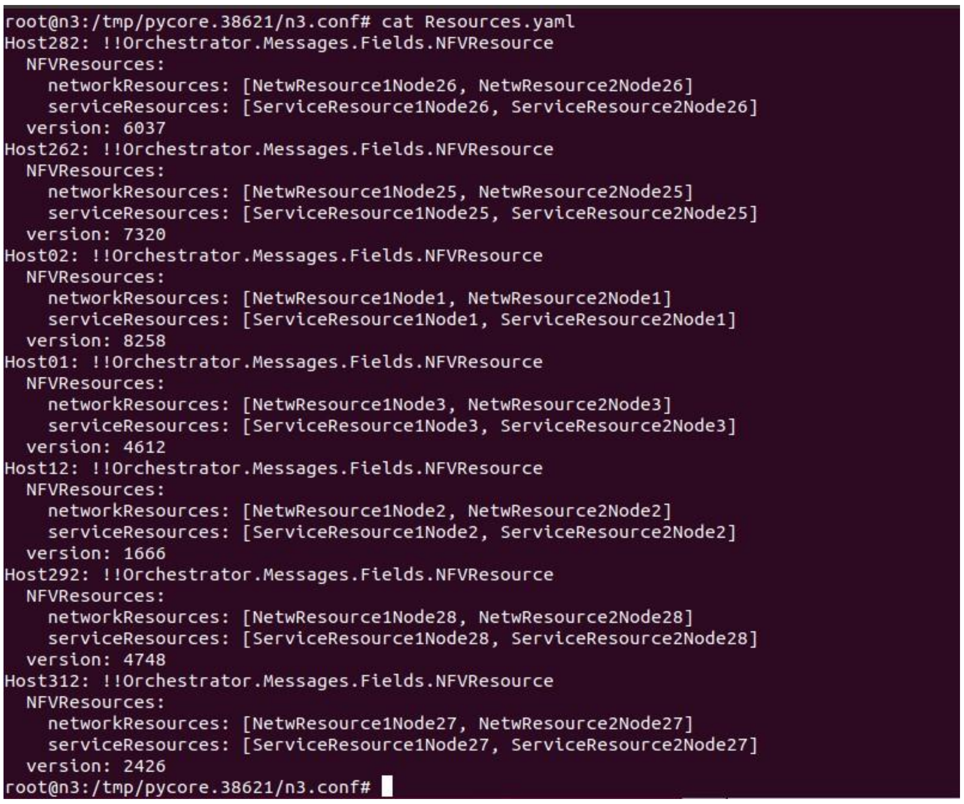
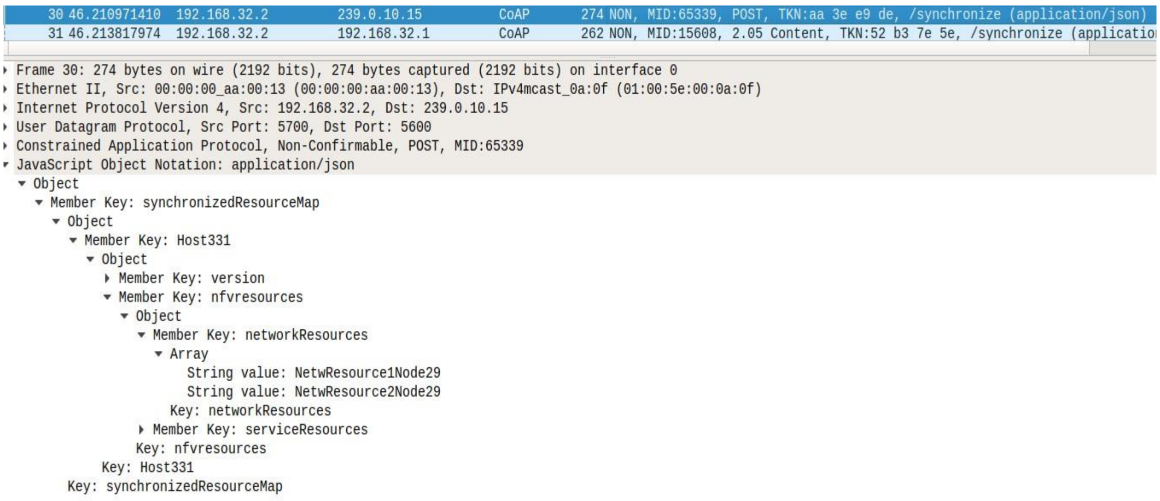


Figure 4.18:Transactions involved in injecting a new node into an already synchronized network. (NW denotes rest of the network)



```
root@n3:/tmp/pycore.38621/n3.conf# cat Resources.yaml
Host282: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node26, NetwResource2Node26]
    serviceResources: [ServiceResource1Node26, ServiceResource2Node26]
  version: 6037
Host262: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node25, NetwResource2Node25]
    serviceResources: [ServiceResource1Node25, ServiceResource2Node25]
  version: 7320
Host02: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node1, NetwResource2Node1]
    serviceResources: [ServiceResource1Node1, ServiceResource2Node1]
  version: 8258
Host01: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node3, NetwResource2Node3]
    serviceResources: [ServiceResource1Node3, ServiceResource2Node3]
  version: 4612
Host12: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node2, NetwResource2Node2]
    serviceResources: [ServiceResource1Node2, ServiceResource2Node2]
  version: 1666
Host292: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node28, NetwResource2Node28]
    serviceResources: [ServiceResource1Node28, ServiceResource2Node28]
  version: 4748
Host312: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node27, NetwResource2Node27]
    serviceResources: [ServiceResource1Node27, ServiceResource2Node27]
  version: 2426
root@n3:/tmp/pycore.38621/n3.conf#
```

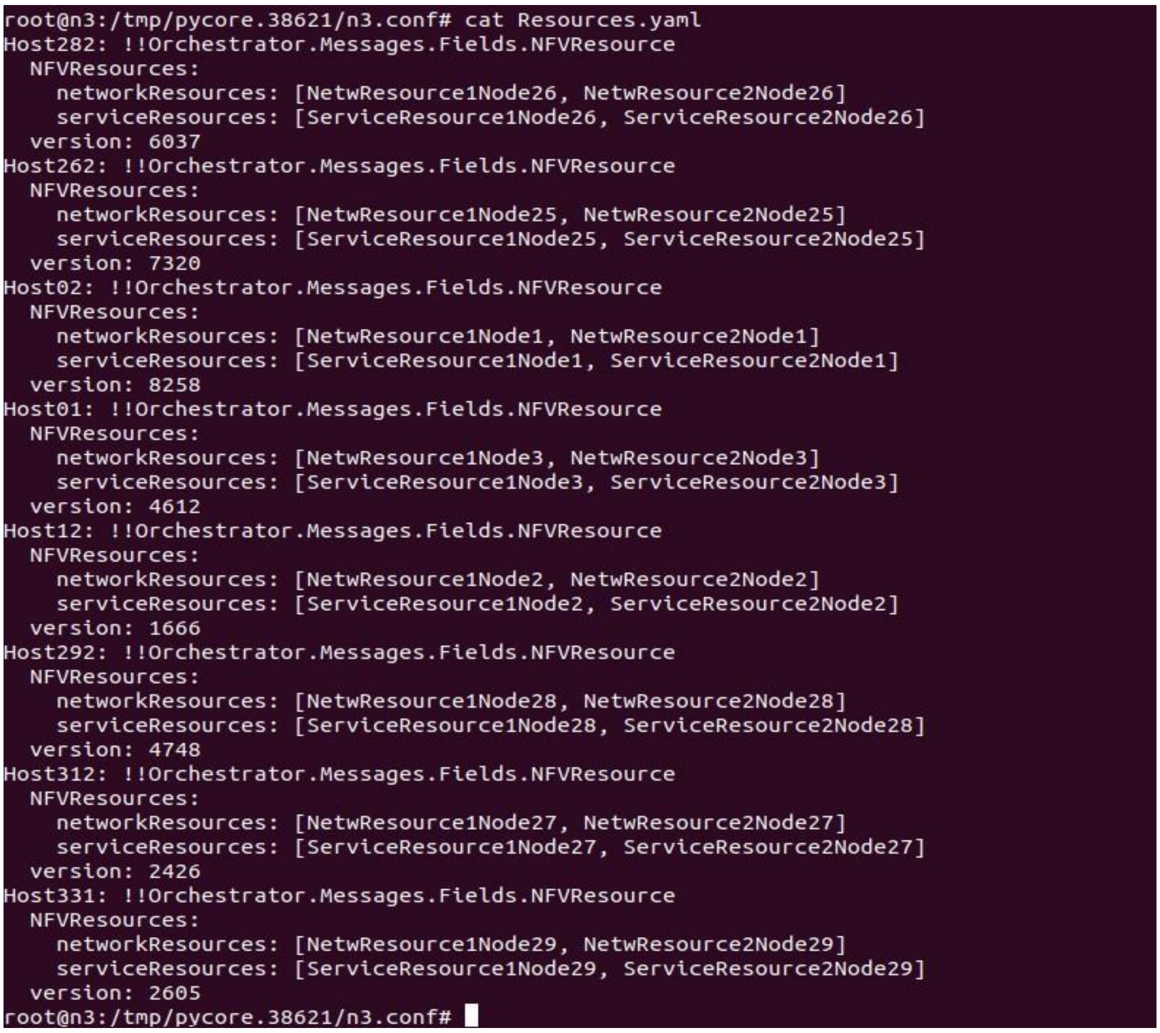
Figure 4.19:Global Resource Information Table view at node n3(Host01) before n29 is injected into the network.



```
    30 46.210971410  192.168.32.2        239.0.10.15         CoAP      274 NON, MID:65339, POST, TKN:aa 3e e9 de, /synchronize (application/json)
    31 46.213817974  192.168.32.2        192.168.32.1        CoAP      262 NON, MID:15608, 2.05 Content, TKN:52 b3 7e 5e, /synchronize (applicatio

▸ Frame 30: 274 bytes on wire (2192 bits), 274 bytes captured (2192 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_aa:00:13 (00:00:00:aa:00:13), Dst: IPv4mcast_0a:0f (01:00:5e:00:0a:0f)
▸ Internet Protocol Version 4, Src: 192.168.32.2, Dst: 239.0.10.15
▸ User Datagram Protocol, Src Port: 5700, Dst Port: 5600
▸ Constrained Application Protocol, Non-Confirmable, POST, MID:65339
▾ JavaScript Object Notation: application/json
  ▾ Object
    ▾ Member Key: synchronizedResourceMap
      ▾ Object
        ▾ Member Key: Host331
          ▾ Object
            ▸ Member Key: version
            ▾ Member Key: nfvresources
              ▾ Object
                ▾ Member Key: networkResources
                  ▾ Array
                      String value: NetwResource1Node29
                      String value: NetwResource2Node29
                    Key: networkResources
                ▸ Member Key: serviceResources
                  Key: nfvresources
              Key: Host331
          Key: synchronizedResourceMap
```

Figure 4.20: NS-Synchronize message of the NS-Synchronize Procedure shown in indigo solid arrow in Figure.4.18 between n29 and n28.



```
root@n3:/tmp/pycore.38621/n3.conf# cat Resources.yaml
Host282: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node26, NetwResource2Node26]
    serviceResources: [ServiceResource1Node26, ServiceResource2Node26]
  version: 6037
Host262: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node25, NetwResource2Node25]
    serviceResources: [ServiceResource1Node25, ServiceResource2Node25]
  version: 7320
Host02: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node1, NetwResource2Node1]
    serviceResources: [ServiceResource1Node1, ServiceResource2Node1]
  version: 8258
Host01: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node3, NetwResource2Node3]
    serviceResources: [ServiceResource1Node3, ServiceResource2Node3]
  version: 4612
Host12: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node2, NetwResource2Node2]
    serviceResources: [ServiceResource1Node2, ServiceResource2Node2]
  version: 1666
Host292: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node28, NetwResource2Node28]
    serviceResources: [ServiceResource1Node28, ServiceResource2Node28]
  version: 4748
Host312: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node27, NetwResource2Node27]
    serviceResources: [ServiceResource1Node27, ServiceResource2Node27]
  version: 2426
Host331: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node29, NetwResource2Node29]
    serviceResources: [ServiceResource1Node29, ServiceResource2Node29]
  version: 2605
root@n3:/tmp/pycore.38621/n3.conf# █
```

Figure 4.21:Global Resource Information Table view at node n3(Host01) after n29 is injected into the network.

4.4.3 All Interfaces of a node failing without causing Network Partition

For the example topology consider when all interfaces of node n25 fail. Then a neighborhood relationship graph as shown in Figure.23 for the rest of the network is achieved. By applying the BFS Algorithm, the connected nodes include all nodes in network except n25. And since n25 could not communicate its own topology change to other nodes, the other nodes may estimate its desynchronization by checking the neighbors of any of the synchronized nodes. In this case the neighbor of n2,n1 and n26 i.e. n25 is desynchronized. In the diagram below shows a connectivity graph which is a diagrammatic equivalent of neighborhood and result queue tables of Section.4.2.3. It is shown from point-of-view of node n26 (Host282) , the red lines denote the estimation links that were pruned during BFS implementation. The connected nodes are represented along the black lines demonstrating all nodes are connected except node n25 which is the result that is expected.

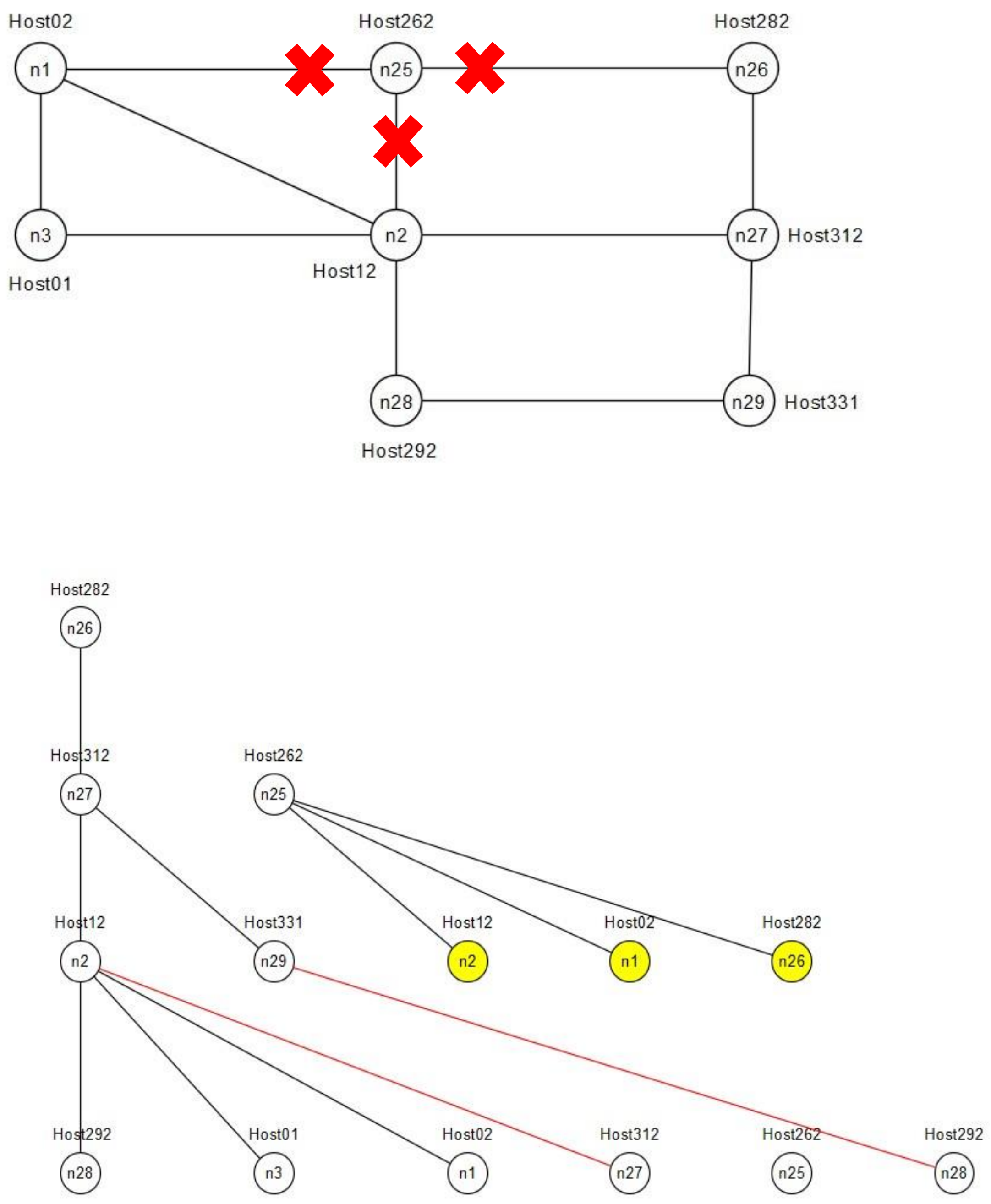
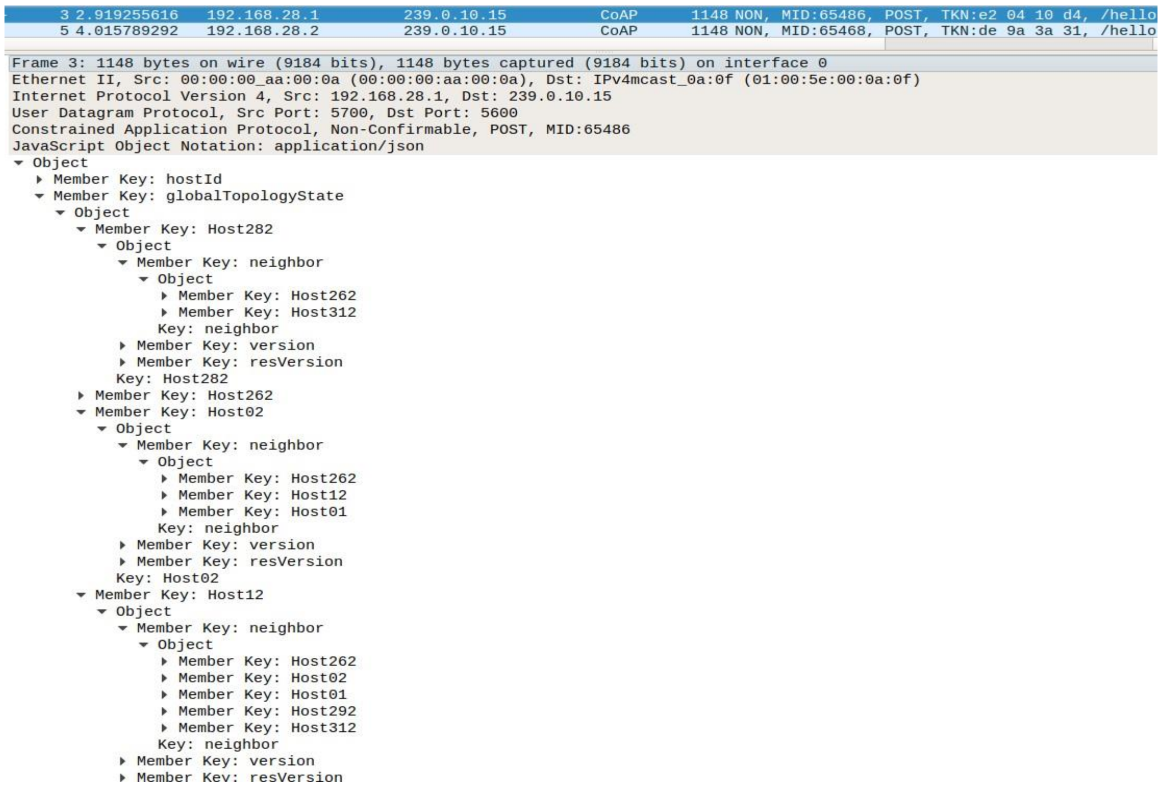


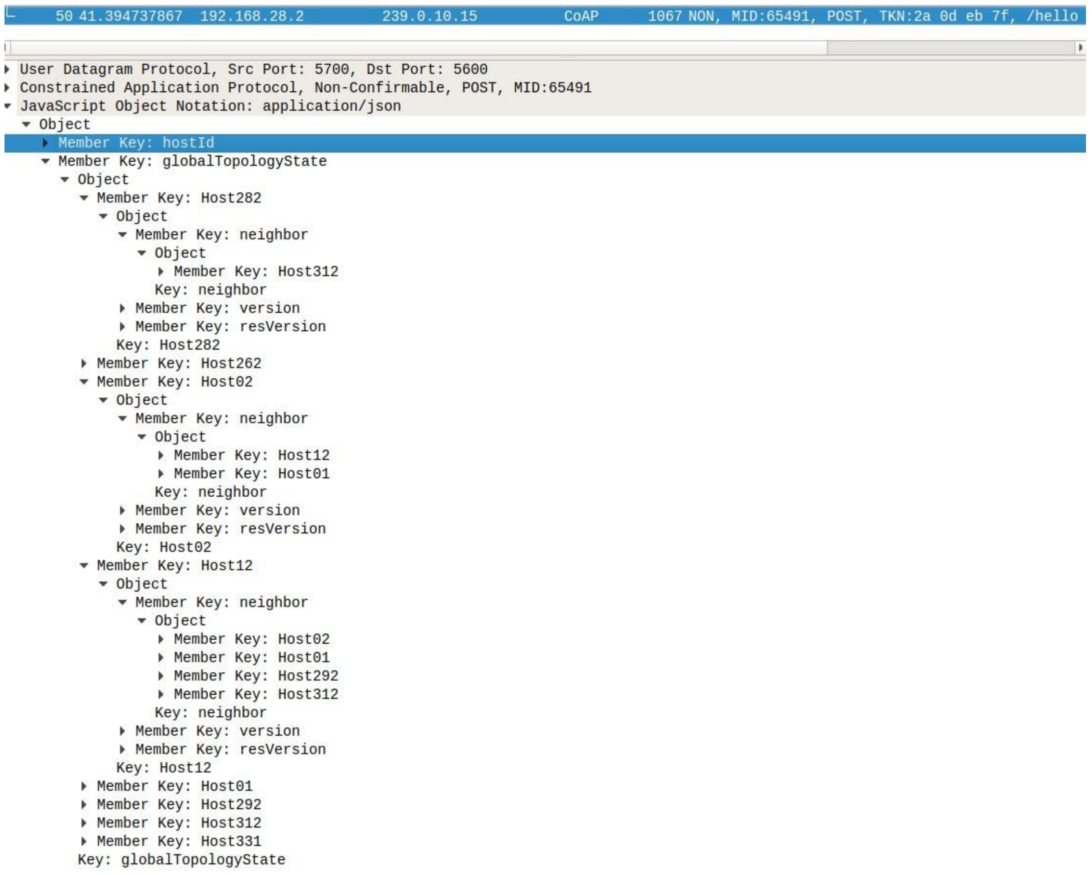
Figure 4.22:Network Graph for connectedness estimation.



```
    3 2.919255616   192.168.28.1        239.0.10.15        CoAP      1148 NON, MID:65486, POST, TKN:e2 04 10 d4, /hello
    5 4.015789292   192.168.28.2        239.0.10.15        CoAP      1148 NON, MID:65468, POST, TKN:de 9a 3a 31, /hello

Frame 3: 1148 bytes on wire (9184 bits), 1148 bytes captured (9184 bits) on interface 0
Ethernet II, Src: 00:00:00_aa:00:0a (00:00:00:aa:00:0a), Dst: IPv4mcast_0a:0f (01:00:5e:00:0a:0f)
Internet Protocol Version 4, Src: 192.168.28.1, Dst: 239.0.10.15
User Datagram Protocol, Src Port: 5700, Dst Port: 5600
Constrained Application Protocol, Non-Confirmable, POST, MID:65486
JavaScript Object Notation: application/json
▾ Object
  ▸ Member Key: hostId
  ▾ Member Key: globalTopologyState
    ▾ Object
      ▾ Member Key: Host282
        ▾ Object
          ▾ Member Key: neighbor
            ▾ Object
              ▸ Member Key: Host262
              ▸ Member Key: Host312
              Key: neighbor
          ▸ Member Key: version
          ▸ Member Key: resVersion
          Key: Host282
      ▸ Member Key: Host262
      ▾ Member Key: Host02
        ▾ Object
          ▾ Member Key: neighbor
            ▾ Object
              ▸ Member Key: Host262
              ▸ Member Key: Host12
              ▸ Member Key: Host01
              Key: neighbor
          ▸ Member Key: version
          ▸ Member Key: resVersion
          Key: Host02
      ▾ Member Key: Host12
        ▾ Object
          ▾ Member Key: neighbor
            ▾ Object
              ▸ Member Key: Host262
              ▸ Member Key: Host02
              ▸ Member Key: Host01
              ▸ Member Key: Host292
              ▸ Member Key: Host312
              Key: neighbor
          ▸ Member Key: version
          ▸ Member Key: resVersion
```

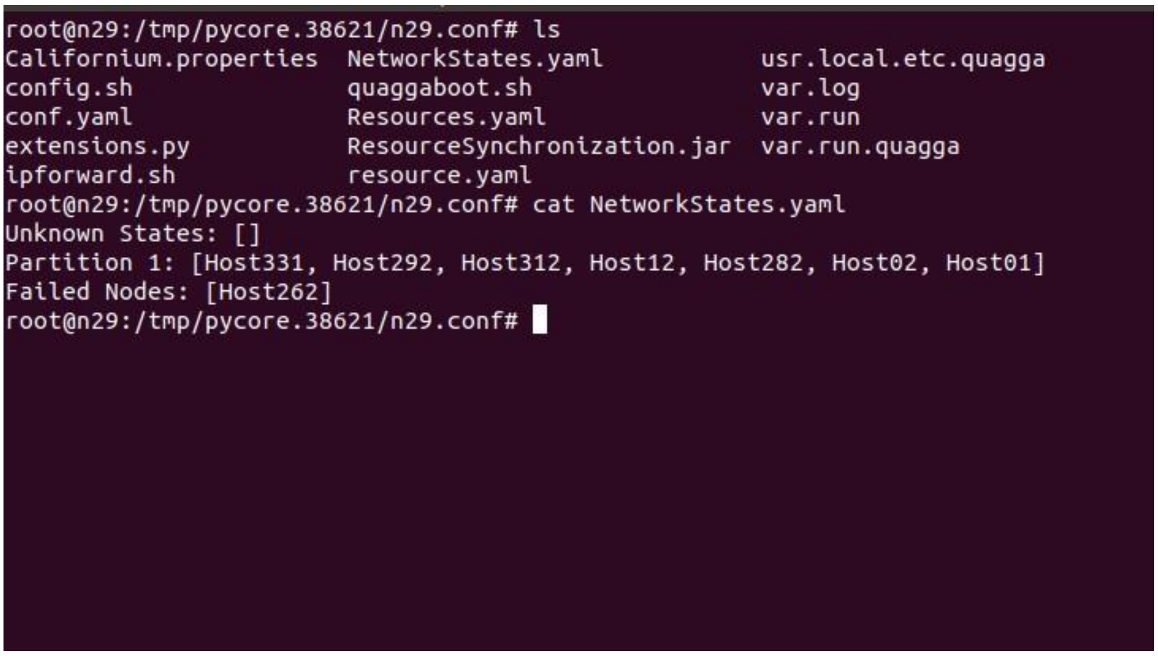
Figure 4.23:NS-Hello globalTopologyState vector before n25(Host262) failed.



```
   50 41.394737867  192.168.28.2          239.0.10.15          CoAP       1067 NON, MID:65491, POST, TKN:2a 0d eb 7f, /hello

▸ User Datagram Protocol, Src Port: 5700, Dst Port: 5600
▸ Constrained Application Protocol, Non-Confirmable, POST, MID:65491
▾ JavaScript Object Notation: application/json
  ▾ Object
    ▸ Member Key: hostId
    ▾ Member Key: globalTopologyState
      ▾ Object
        ▾ Member Key: Host282
          ▾ Object
            ▾ Member Key: neighbor
              ▾ Object
                ▸ Member Key: Host312
                Key: neighbor
            ▸ Member Key: version
            ▸ Member Key: resVersion
            Key: Host282
        ▸ Member Key: Host262
        ▾ Member Key: Host02
          ▾ Object
            ▾ Member Key: neighbor
              ▾ Object
                ▸ Member Key: Host12
                ▸ Member Key: Host01
                Key: neighbor
            ▸ Member Key: version
            ▸ Member Key: resVersion
            Key: Host02
        ▾ Member Key: Host12
          ▾ Object
            ▾ Member Key: neighbor
              ▾ Object
                ▸ Member Key: Host02
                ▸ Member Key: Host01
                ▸ Member Key: Host292
                ▸ Member Key: Host312
                Key: neighbor
            ▸ Member Key: version
            ▸ Member Key: resVersion
            Key: Host12
        ▸ Member Key: Host01
        ▸ Member Key: Host292
        ▸ Member Key: Host312
        ▸ Member Key: Host331
      Key: globalTopologyState
```

Figure 4.24: NS-Hello globalTopologyState vector after n25(Host262) failed.



```
root@n29:/tmp/pycore.38621/n29.conf# ls
Californium.properties  NetworkStates.yaml           usr.local.etc.quagga
config.sh               quaggaboot.sh                var.log
conf.yaml               Resources.yaml               var.run
extensions.py           ResourceSynchronization.jar  var.run.quagga
ipforward.sh            resource.yaml
root@n29:/tmp/pycore.38621/n29.conf# cat NetworkStates.yaml
Unknown States: []
Partition 1: [Host331, Host292, Host312, Host12, Host282, Host02, Host01]
Failed Nodes: [Host262]
root@n29:/tmp/pycore.38621/n29.conf#
```

Figure 4.25:Identification of failure of n25(Host262).

4.4.4 Interfaces of multiple nodes failing causing Network Partition

The same scenario as mentioned above when coupled with all interfaces of n2 also failing demonstrates a case of network partition as shown below. The graph is calculated at n26, so the resulting connected nodes computation shows the partition which is synchronized with n26. In the same fashion as previous case if the direct neighbors of the nodes of the synchronized sector are computed, the result should be the nodes whose interface failure has caused this network partition. The second connected neighbor to that would be the nodes whose states cannot be determined (hence marked unknown) because there is no UDP path to those nodes(In this case node n1 & n3). Looking from the point of view of n26, as shown in Figure.4.18 the partition to which n26 is synchronized to is the set (n26, n27, n29, n28). Then the BFS module computes the failed nodes by finding the first neighbors to the nodes marked in yellow, hence n25 and n2 are estimated as nodes whose failure caused the network partition and the remaining nodes (n1 and n3) possibly belong to foreign partition.

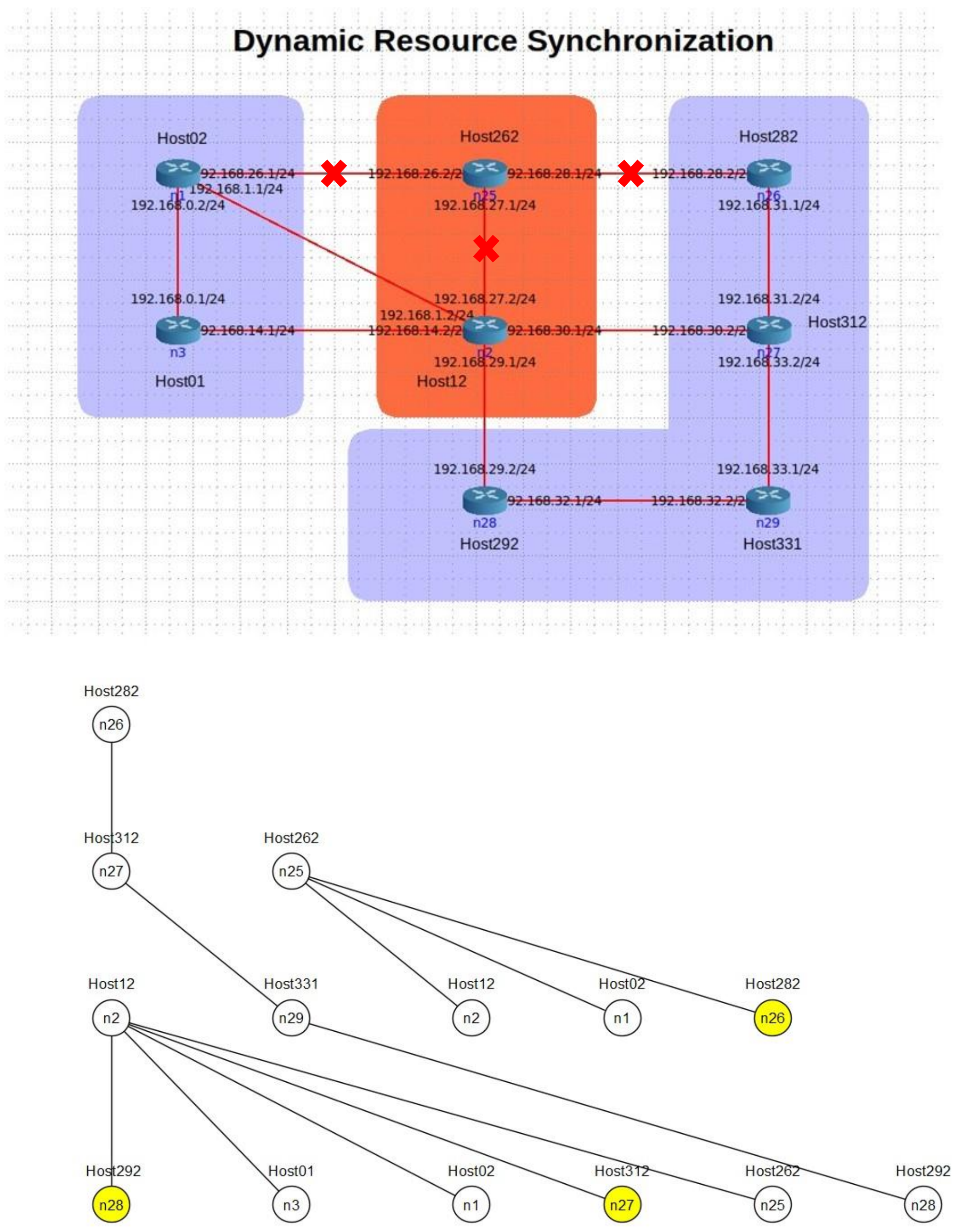
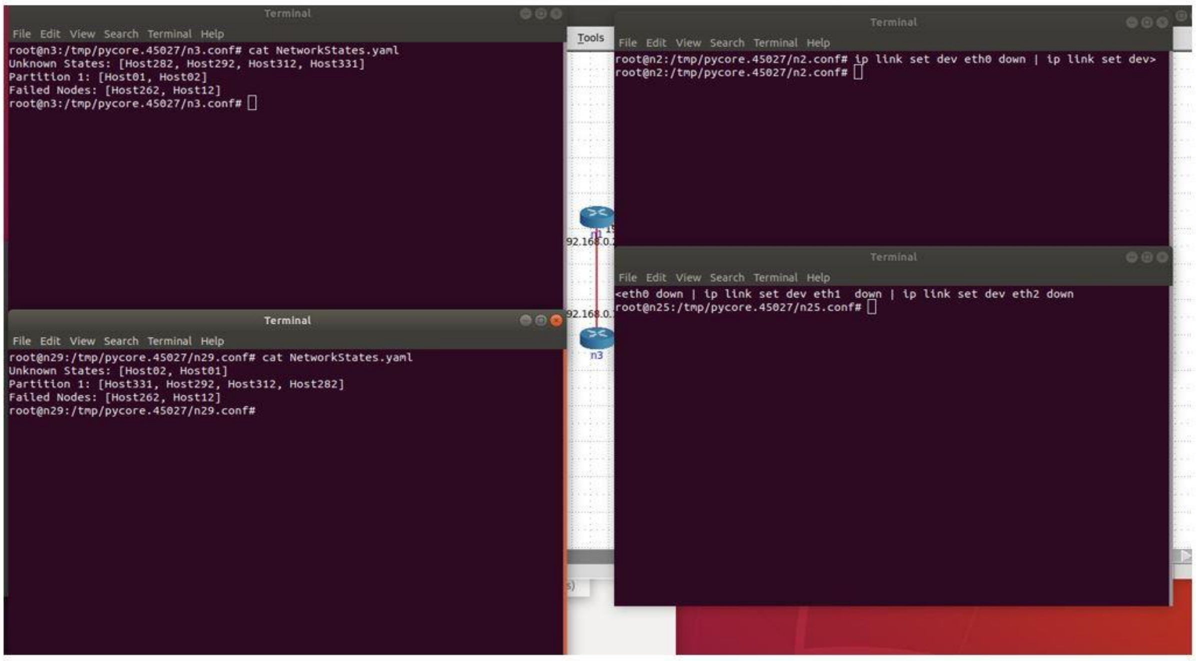


Figure 4.26:Network Graph for connectedness estimation in case of network partition identifying desynchronized nodes and unknown nodes of distant partition.



```
root@n3:/tmp/pycore.45027/n3.conf# cat NetworkStates.yaml
Unknown States: [Host282, Host292, Host312, Host331]
Partition 1: [Host01, Host02]
Failed Nodes: [Host262, Host12]
root@n3:/tmp/pycore.45027/n3.conf#
```

```
root@n29:/tmp/pycore.45027/n29.conf# cat NetworkStates.yaml
Unknown States: [Host02, Host01]
Partition 1: [Host331, Host292, Host312, Host282]
Failed Nodes: [Host262, Host12]
root@n29:/tmp/pycore.45027/n29.conf#
```

```
root@n2:/tmp/pycore.45027/n2.conf# ip link set dev eth0 down | ip link set dev>
root@n2:/tmp/pycore.45027/n2.conf#
```

```
<eth0 down | ip link set dev eth1  down | ip link set dev eth2 down
root@n25:/tmp/pycore.45027/n25.conf#
```

Figure 4.27:Demonstration of a network partition scenario identification on nodes n3 and n29 (on the left) when node n25(Host262) and n2(Host12) fails.

4.4.5 Interface of node fails without causing Network Desynchronization

Consider the scenario where the interface between node n27 and n29 collapses then, the two nodes deregister each other from the neighborhood relationships in NS-Hello's topology sector. Then at any given node one may find equivalents of the graph below. Since there is another UDP path connecting n27 and n29 to the rest of the network, neither of the nodes get desynchronized from resource synchronization procedure and the connectedness estimation computes a fully connected network as shown below. Like previous scenario the diagrammatic representation of state connectedness shows in black lines the connectivity estimation and the red lines represents the eliminated nodes of algorithm implementation throughout the graph. So, the diagram shows the entire network is synchronized with resource information.

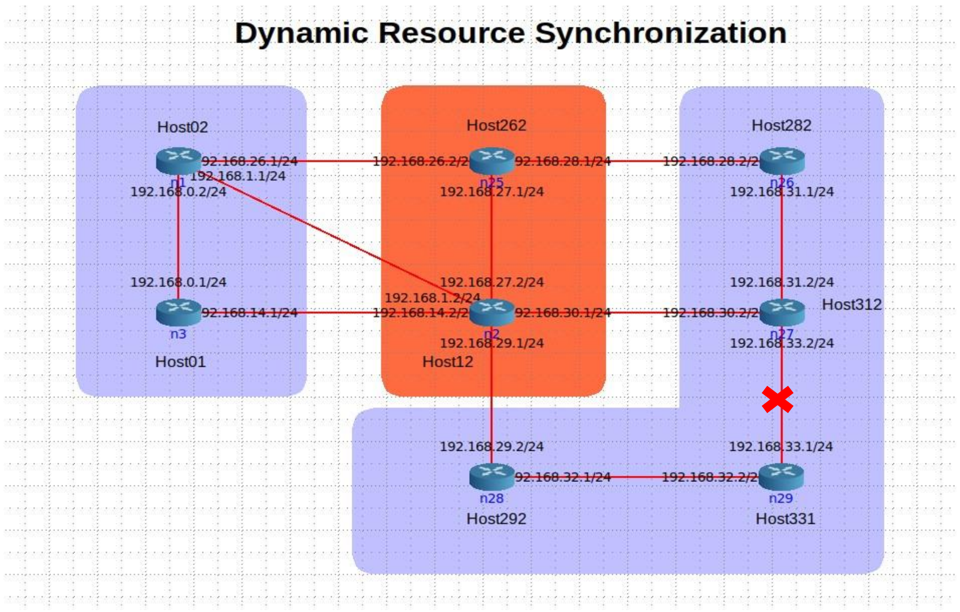


Figure 4.28: Link between n29(Host331) and n27(Host312) fails during runtime.

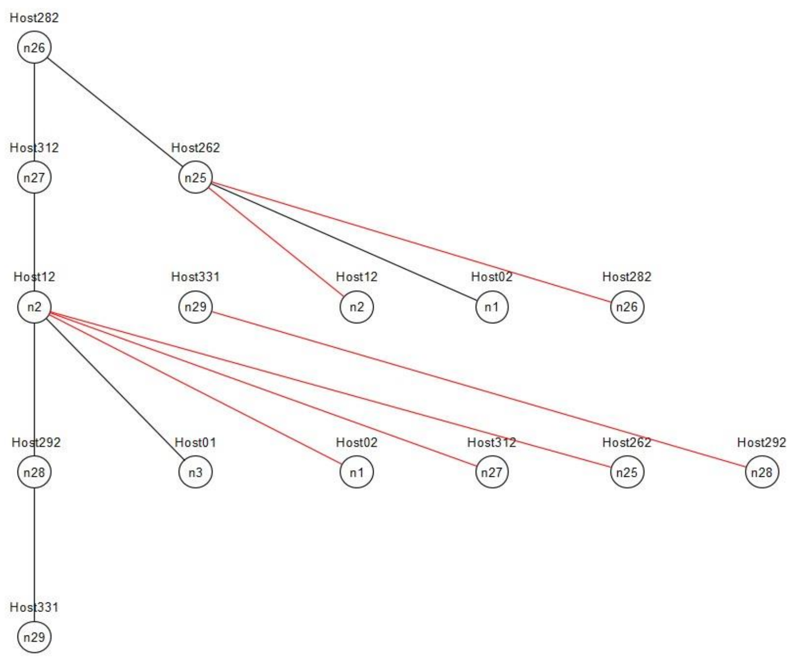
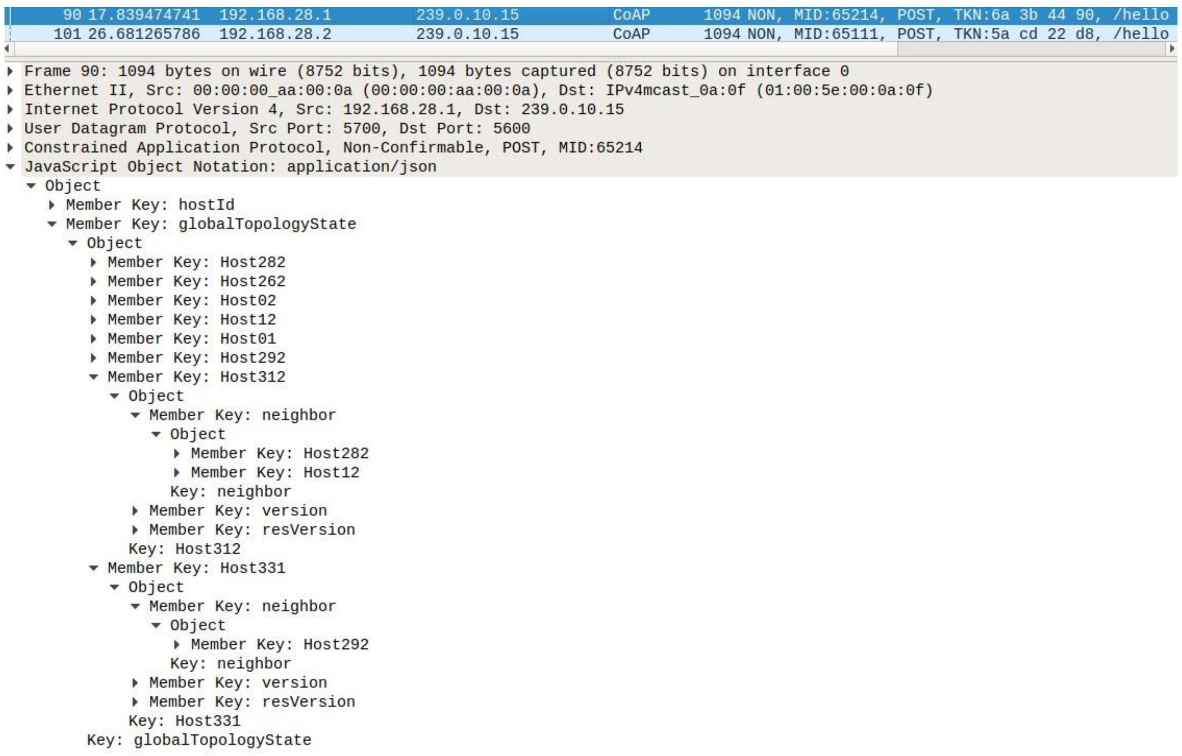
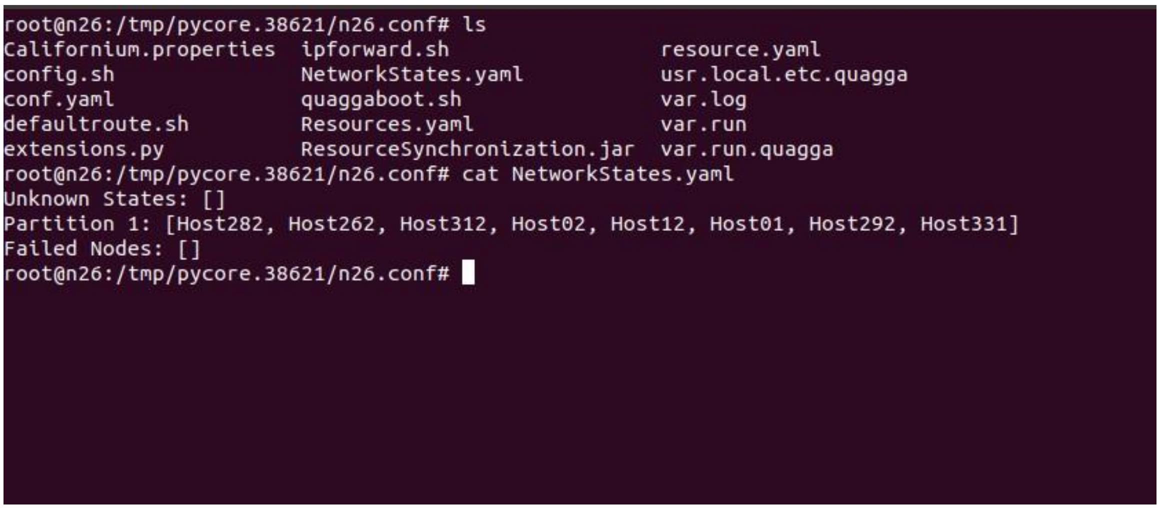


Figure 4.29:Network Graph for connectedness estimation.



```
90 17.839474741  192.168.28.1    239.0.10.15    CoAP    1094 NON, MID:65214, POST, TKN:6a 3b 44 90, /hello
101 26.681265786  192.168.28.2    239.0.10.15    CoAP    1094 NON, MID:65111, POST, TKN:5a cd 22 d8, /hello
▸ Frame 90: 1094 bytes on wire (8752 bits), 1094 bytes captured (8752 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_aa:00:0a (00:00:00:aa:00:0a), Dst: IPv4mcast_0a:0f (01:00:5e:00:0a:0f)
▸ Internet Protocol Version 4, Src: 192.168.28.1, Dst: 239.0.10.15
▸ User Datagram Protocol, Src Port: 5700, Dst Port: 5600
▸ Constrained Application Protocol, Non-Confirmable, POST, MID:65214
▾ JavaScript Object Notation: application/json
  ▾ Object
    ▸ Member Key: hostId
    ▾ Member Key: globalTopologyState
      ▾ Object
        ▸ Member Key: Host282
        ▸ Member Key: Host262
        ▸ Member Key: Host02
        ▸ Member Key: Host12
        ▸ Member Key: Host01
        ▸ Member Key: Host292
        ▾ Member Key: Host312
          ▾ Object
            ▾ Member Key: neighbor
              ▾ Object
                ▸ Member Key: Host282
                ▸ Member Key: Host12
                Key: neighbor
            ▸ Member Key: version
            ▸ Member Key: resVersion
            Key: Host312
        ▾ Member Key: Host331
          ▾ Object
            ▾ Member Key: neighbor
              ▾ Object
                ▸ Member Key: Host292
                Key: neighbor
            ▸ Member Key: version
            ▸ Member Key: resVersion
            Key: Host331
      Key: globalTopologyState
```

Figure 4.30: NS-Hello globalTopologyState field corresponding to the BFS diagrammatic equivalent of Figure.4.29



```
root@n26:/tmp/pycore.38621/n26.conf# ls
Californium.properties  ipforward.sh                resource.yaml
config.sh               NetworkStates.yaml          usr.local.etc.quagga
conf.yaml               quaggaboot.sh               var.log
defaultroute.sh         Resources.yaml              var.run
extensions.py           ResourceSynchronization.jar  var.run.quagga
root@n26:/tmp/pycore.38621/n26.conf# cat NetworkStates.yaml
Unknown States: []
Partition 1: [Host282, Host262, Host312, Host02, Host12, Host01, Host292, Host331]
Failed Nodes: []
root@n26:/tmp/pycore.38621/n26.conf# 
```

Figure 4.31: Network connectedness estimation proving that network was not partitioned even though there was an interface failure.

4.4.6 Merging of Distinct Network Partitions

Consider the scenario of partition created by failure of nodes n25 and n2 as shown below and then reunification of the partitioned segments by restarting n25. So, both the segments will synchronize their resource information via n25, and the keep-alive messages will enable all the nodes to compute the asymmetric connectivity graph. From the point of view of n26, BFS computation will enable connectivity estimation graph as shown in Figure.4.20, which shows pruned segment of n26 denoted in red and the connected graph shown in black which represents all the enabled nodes in the synchronized segment (n1, n3, n25, n26, n27, n28, n29).

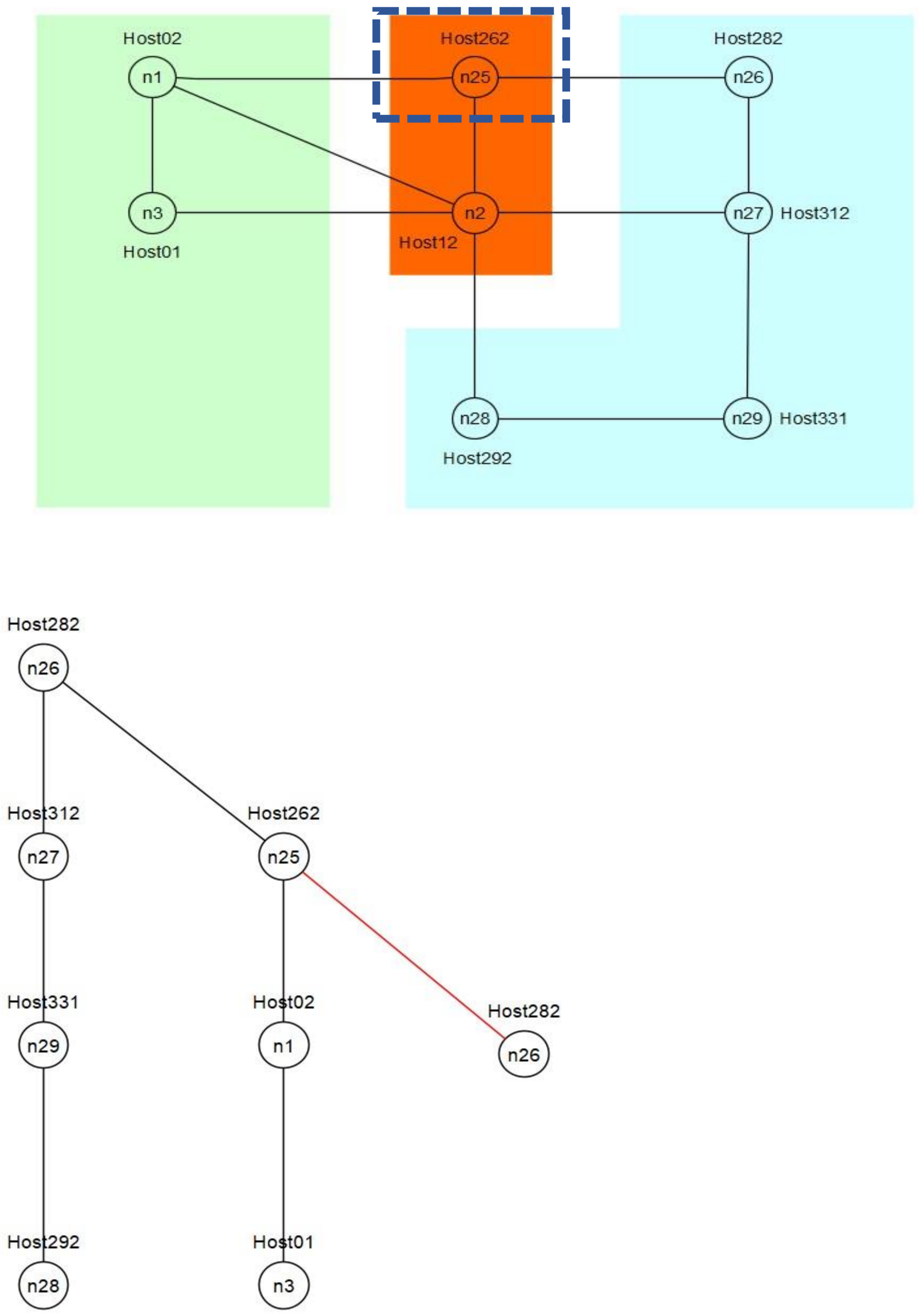
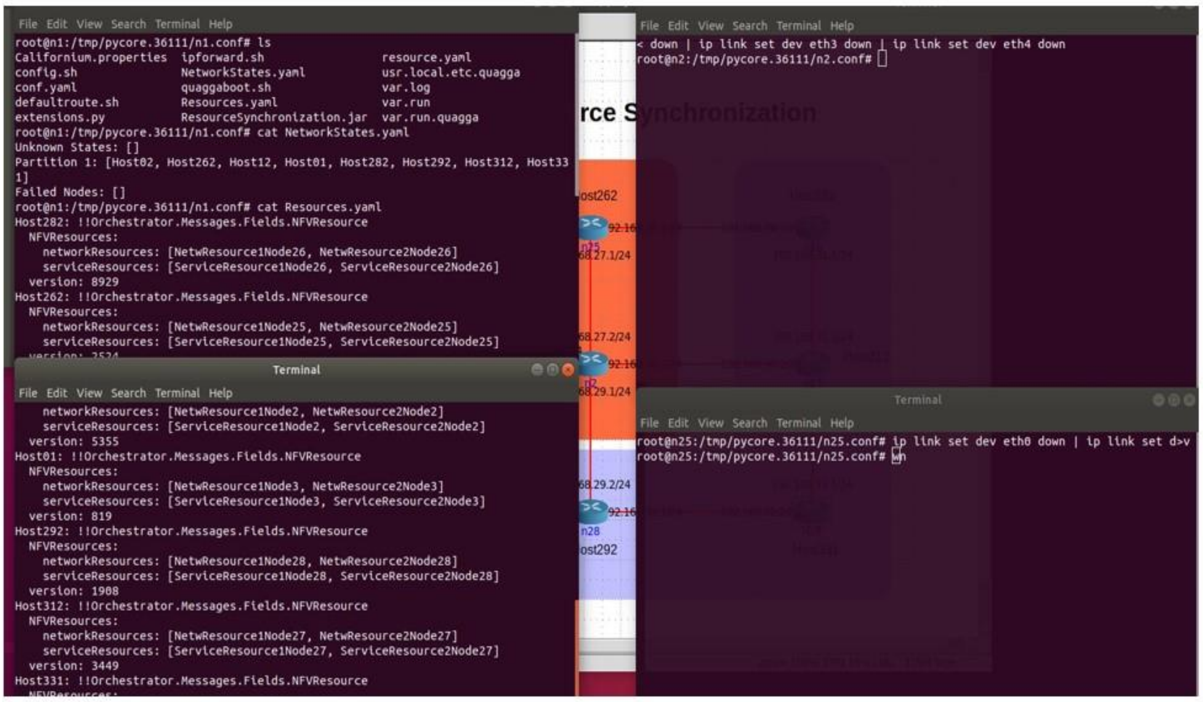


Figure 4.22:Merging of distinct partitions



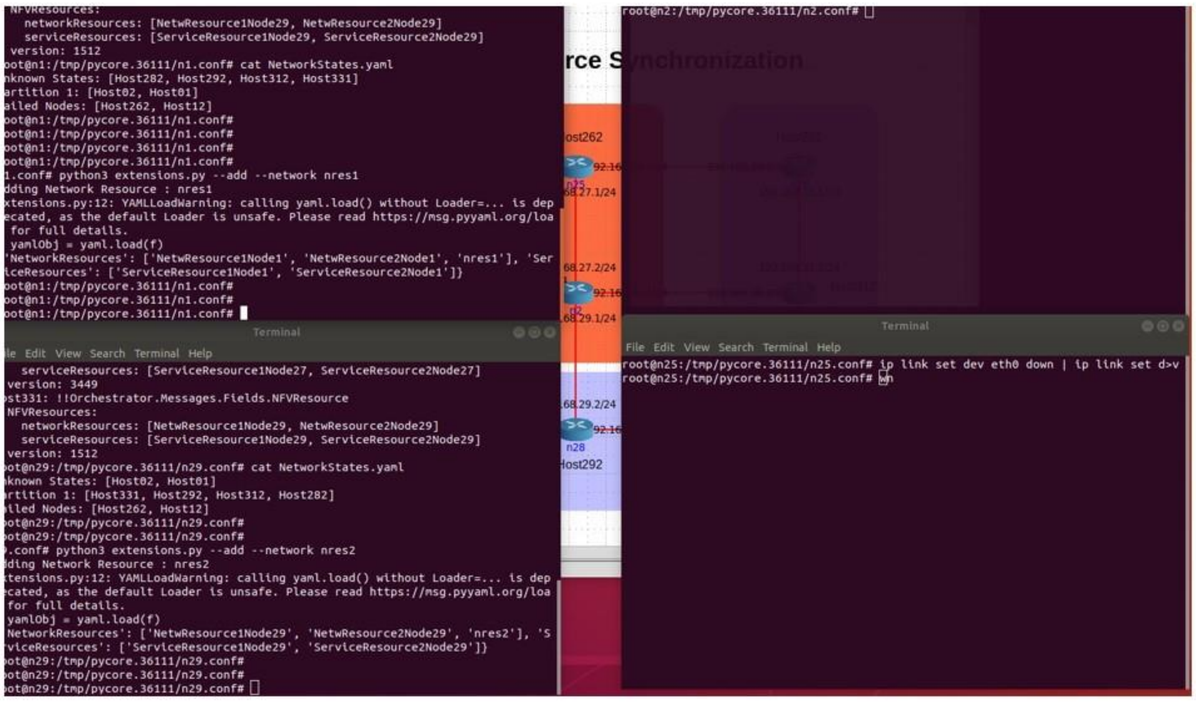
```
root@n1:/tmp/pycore.36111/n1.conf# ls
Californium.properties  ipforward.sh               resource.yaml
config.sh               NetworkStates.yaml         usr.local.etc.quagga
conf.yaml               quaggaboot.sh              var.log
defaultroute.sh         Resources.yaml             var.run
extensions.py           ResourceSynchronization.jar  var.run.quagga
root@n1:/tmp/pycore.36111/n1.conf# cat NetworkStates.yaml
Unknown States: []
Partition 1: [Host02, Host262, Host12, Host01, Host282, Host292, Host312, Host33
1]
Failed Nodes: []
root@n1:/tmp/pycore.36111/n1.conf# cat Resources.yaml
Host282: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node26, NetwResource2Node26]
    serviceResources: [ServiceResource1Node26, ServiceResource2Node26]
  version: 8929
Host262: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node25, NetwResource2Node25]
    serviceResources: [ServiceResource1Node25, ServiceResource2Node25]
```

```
    networkResources: [NetwResource1Node2, NetwResource2Node2]
    serviceResources: [ServiceResource1Node2, ServiceResource2Node2]
  version: 5355
Host01: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node3, NetwResource2Node3]
    serviceResources: [ServiceResource1Node3, ServiceResource2Node3]
  version: 819
Host292: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node28, NetwResource2Node28]
    serviceResources: [ServiceResource1Node28, ServiceResource2Node28]
  version: 1908
Host312: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node27, NetwResource2Node27]
    serviceResources: [ServiceResource1Node27, ServiceResource2Node27]
  version: 3449
Host331: !!Orchestrator.Messages.Fields.NFVResource
```

```
< down | ip link set dev eth3 down | ip link set dev eth4 down
root@n2:/tmp/pycore.36111/n2.conf#
```

```
root@n25:/tmp/pycore.36111/n25.conf# ip link set dev eth0 down | ip link set d>v
root@n25:/tmp/pycore.36111/n25.conf# wn
```

Figure 4.33:Merging of distinct Partition : Blocking the interfaces of n25 (Host 262) and n2 (Host 12)



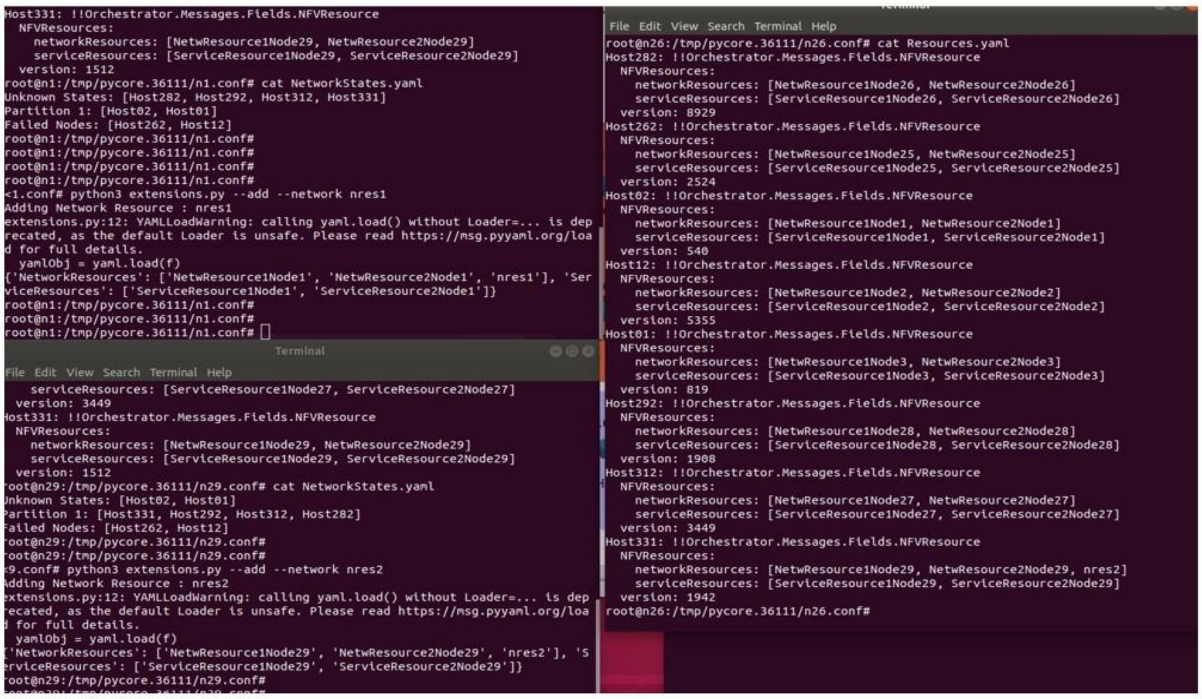
```
    networkResources: [NetwResource1Node29, NetwResource2Node29]
    serviceResources: [ServiceResource1Node29, ServiceResource2Node29]
  version: 1512
oot@n1:/tmp/pycore.36111/n1.conf# cat NetworkStates.yaml
nknown States: [Host282, Host292, Host312, Host331]
artition 1: [Host02, Host01]
ailed Nodes: [Host262, Host12]
oot@n1:/tmp/pycore.36111/n1.conf#
oot@n1:/tmp/pycore.36111/n1.conf#
oot@n1:/tmp/pycore.36111/n1.conf#
oot@n1:/tmp/pycore.36111/n1.conf#
1.conf# python3 extensions.py --add --network nres1
dding Network Resource : nres1
xtensions.py:12: YAMLLoadWarning: calling yaml.load() without Loader=... is dep
ecated, as the default Loader is unsafe. Please read https://msg.pyyaml.org/loa
 for full details.
 yamlObj = yaml.load(f)
'NetworkResources': ['NetwResource1Node1', 'NetwResource2Node1', 'nres1'], 'Ser
iceResources': ['ServiceResource1Node1', 'ServiceResource2Node1']}
oot@n1:/tmp/pycore.36111/n1.conf#
oot@n1:/tmp/pycore.36111/n1.conf#
oot@n1:/tmp/pycore.36111/n1.conf#
```

```
   serviceResources: [ServiceResource1Node27, ServiceResource2Node27]
 version: 3449
st331: !!Orchestrator.Messages.Fields.NFVResource
 NFVResources:
   networkResources: [NetwResource1Node29, NetwResource2Node29]
   serviceResources: [ServiceResource1Node29, ServiceResource2Node29]
 version: 1512
ot@n29:/tmp/pycore.36111/n29.conf# cat NetworkStates.yaml
known States: [Host02, Host01]
rtition 1: [Host331, Host292, Host312, Host282]
iled Nodes: [Host262, Host12]
ot@n29:/tmp/pycore.36111/n29.conf#
ot@n29:/tmp/pycore.36111/n29.conf#
.conf# python3 extensions.py --add --network nres2
dding Network Resource : nres2
tensions.py:12: YAMLLoadWarning: calling yaml.load() without Loader=... is dep
cated, as the default Loader is unsafe. Please read https://msg.pyyaml.org/loa
 for full details.
 yamlObj = yaml.load(f)
 NetworkResources': ['NetwResource1Node29', 'NetwResource2Node29', 'nres2'], 'S
viceResources': ['ServiceResource1Node29', 'ServiceResource2Node29']}
ot@n29:/tmp/pycore.36111/n29.conf#
ot@n29:/tmp/pycore.36111/n29.conf#
ot@n29:/tmp/pycore.36111/n29.conf#
```

```
root@n2:/tmp/pycore.36111/n2.conf#
```

```
root@n25:/tmp/pycore.36111/n25.conf# ip link set dev eth0 down | ip link set d>v
root@n25:/tmp/pycore.36111/n25.conf# wn
```

Figure 4.34:Merging of Partitions: Identification of Partitions shown in terminals to the left under NetworkStates.yaml.

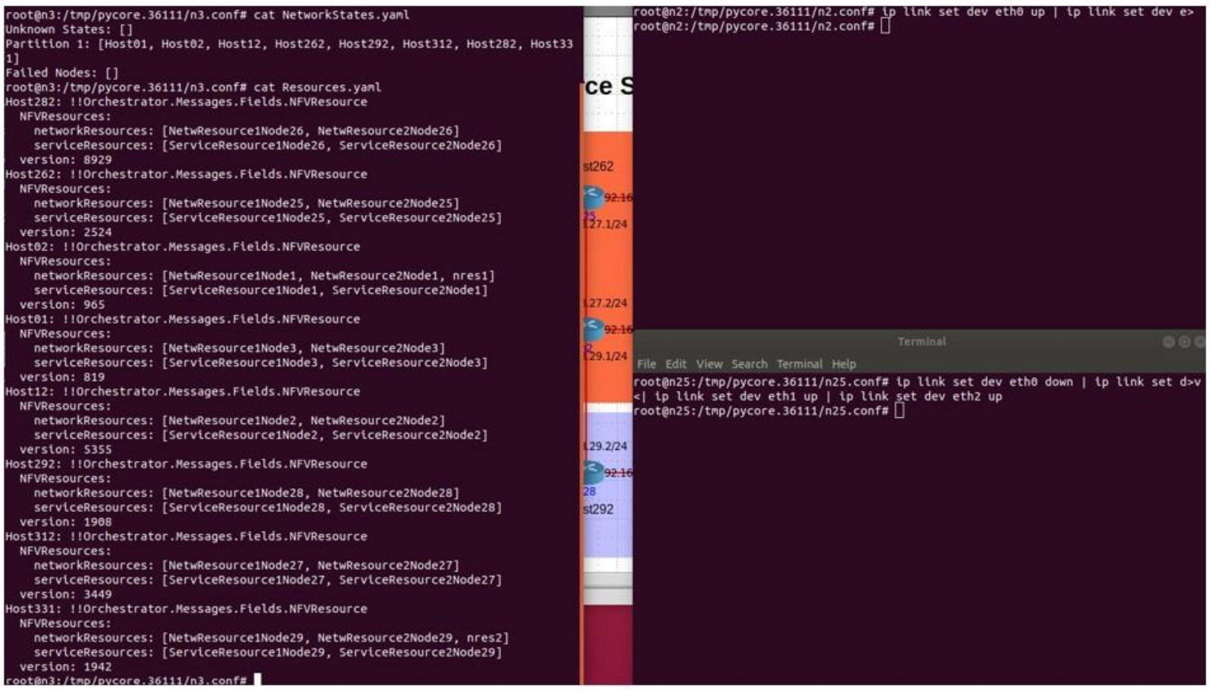


```
Host331: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node29, NetwResource2Node29]
    serviceResources: [ServiceResource1Node29, ServiceResource2Node29]
  version: 1512
root@n1:/tmp/pycore.36111/n1.conf# cat NetworkStates.yaml
Unknown States: [Host282, Host292, Host312, Host331]
Partition 1: [Host02, Host01]
Failed Nodes: [Host262, Host12]
root@n1:/tmp/pycore.36111/n1.conf#
root@n1:/tmp/pycore.36111/n1.conf#
root@n1:/tmp/pycore.36111/n1.conf#
root@n1:/tmp/pycore.36111/n1.conf#
<1.conf# python3 extensions.py --add --network nres1
Adding Network Resource : nres1
extensions.py:12: YAMLLoadWarning: calling yaml.load() without Loader=... is dep
recated, as the default Loader is unsafe. Please read https://msg.pyyaml.org/loa
d for full details.
  yamlObj = yaml.load(f)
{'NetworkResources': ['NetwResource1Node1', 'NetwResource2Node1', 'nres1'], 'Ser
viceResources': ['ServiceResource1Node1', 'ServiceResource2Node1']}
root@n1:/tmp/pycore.36111/n1.conf#
root@n1:/tmp/pycore.36111/n1.conf#
root@n1:/tmp/pycore.36111/n1.conf# 
```

```
   serviceResources: [ServiceResource1Node27, ServiceResource2Node27]
 version: 3449
Host331: !!Orchestrator.Messages.Fields.NFVResource
 NFVResources:
   networkResources: [NetwResource1Node29, NetwResource2Node29]
   serviceResources: [ServiceResource1Node29, ServiceResource2Node29]
 version: 1512
root@n29:/tmp/pycore.36111/n29.conf# cat NetworkStates.yaml
Unknown States: [Host02, Host01]
Partition 1: [Host331, Host292, Host312, Host282]
Failed Nodes: [Host262, Host12]
root@n29:/tmp/pycore.36111/n29.conf#
root@n29:/tmp/pycore.36111/n29.conf#
<9.conf# python3 extensions.py --add --network nres2
Adding Network Resource : nres2
extensions.py:12: YAMLLoadWarning: calling yaml.load() without Loader=... is dep
recated, as the default Loader is unsafe. Please read https://msg.pyyaml.org/loa
d for full details.
 yamlObj = yaml.load(f)
{'NetworkResources': ['NetwResource1Node29', 'NetwResource2Node29', 'nres2'], 'S
erviceResources': ['ServiceResource1Node29', 'ServiceResource2Node29']}
root@n29:/tmp/pycore.36111/n29.conf#
```

```
root@n26:/tmp/pycore.36111/n26.conf# cat Resources.yaml
Host282: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node26, NetwResource2Node26]
    serviceResources: [ServiceResource1Node26, ServiceResource2Node26]
  version: 8929
Host262: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node25, NetwResource2Node25]
    serviceResources: [ServiceResource1Node25, ServiceResource2Node25]
  version: 2524
Host02: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node1, NetwResource2Node1]
    serviceResources: [ServiceResource1Node1, ServiceResource2Node1]
  version: 540
Host12: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node2, NetwResource2Node2]
    serviceResources: [ServiceResource1Node2, ServiceResource2Node2]
  version: 5355
Host01: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node3, NetwResource2Node3]
    serviceResources: [ServiceResource1Node3, ServiceResource2Node3]
  version: 819
Host292: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node28, NetwResource2Node28]
    serviceResources: [ServiceResource1Node28, ServiceResource2Node28]
  version: 1908
Host312: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node27, NetwResource2Node27]
    serviceResources: [ServiceResource1Node27, ServiceResource2Node27]
  version: 3449
Host331: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node29, NetwResource2Node29, nres2]
    serviceResources: [ServiceResource1Node29, ServiceResource2Node29]
  version: 1942
root@n26:/tmp/pycore.36111/n26.conf#
```

Figure 4.35:Merging Partitions : Modifying resource information in n29 (Host 331) and n1 (Host 02) shown in terminals on the left.

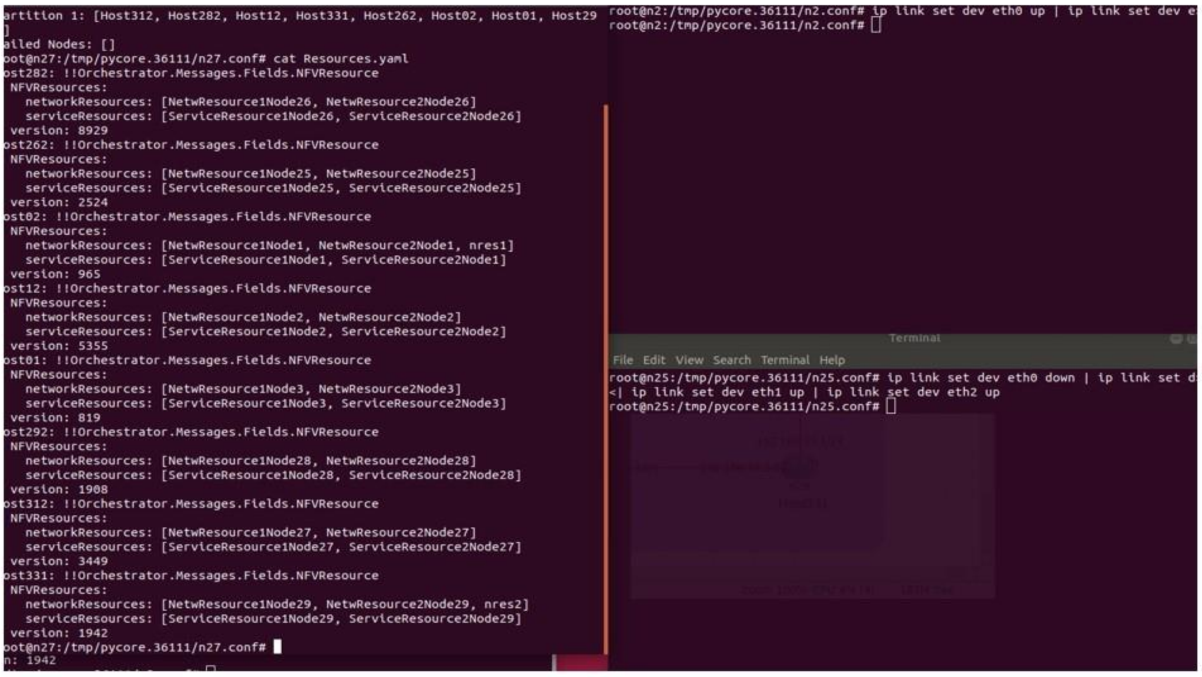


```
root@n3:/tmp/pycore.36111/n3.conf# cat NetworkStates.yaml
Unknown States: []
Partition 1: [Host01, Host02, Host12, Host262, Host292, Host312, Host282, Host33
1]
Failed Nodes: []
root@n3:/tmp/pycore.36111/n3.conf# cat Resources.yaml
Host282: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node26, NetwResource2Node26]
    serviceResources: [ServiceResource1Node26, ServiceResource2Node26]
  version: 8929
Host262: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node25, NetwResource2Node25]
    serviceResources: [ServiceResource1Node25, ServiceResource2Node25]
  version: 2524
Host02: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node1, NetwResource2Node1, nres1]
    serviceResources: [ServiceResource1Node1, ServiceResource2Node1]
  version: 965
Host01: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node3, NetwResource2Node3]
    serviceResources: [ServiceResource1Node3, ServiceResource2Node3]
  version: 819
Host12: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node2, NetwResource2Node2]
    serviceResources: [ServiceResource1Node2, ServiceResource2Node2]
  version: 5355
Host292: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node28, NetwResource2Node28]
    serviceResources: [ServiceResource1Node28, ServiceResource2Node28]
  version: 1908
Host312: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node27, NetwResource2Node27]
    serviceResources: [ServiceResource1Node27, ServiceResource2Node27]
  version: 3449
Host331: !!Orchestrator.Messages.Fields.NFVResource
  NFVResources:
    networkResources: [NetwResource1Node29, NetwResource2Node29, nres2]
    serviceResources: [ServiceResource1Node29, ServiceResource2Node29]
  version: 1942
root@n3:/tmp/pycore.36111/n3.conf#
```

```
root@n2:/tmp/pycore.36111/n2.conf# ip link set dev eth0 up | ip link set dev e>
root@n2:/tmp/pycore.36111/n2.conf# 
```

```
root@n25:/tmp/pycore.36111/n25.conf# ip link set dev eth0 down | ip link set d>v
<| ip link set dev eth1 up | ip link set dev eth2 up
root@n25:/tmp/pycore.36111/n25.conf# 
```

Figure 4.36:Merging Partitions : Merging Partitions : Merged the separated partitions and resource information modifications. View from node n3.



```
artition 1: [Host312, Host282, Host12, Host331, Host262, Host02, Host01, Host29
]
ailed Nodes: []
oot@n27:/tmp/pycore.36111/n27.conf# cat Resources.yaml
ost282: !!Orchestrator.Messages.Fields.NFVResource
 NFVResources:
   networkResources: [NetwResource1Node26, NetwResource2Node26]
   serviceResources: [ServiceResource1Node26, ServiceResource2Node26]
 version: 8929
ost262: !!Orchestrator.Messages.Fields.NFVResource
 NFVResources:
   networkResources: [NetwResource1Node25, NetwResource2Node25]
   serviceResources: [ServiceResource1Node25, ServiceResource2Node25]
 version: 2524
ost02: !!Orchestrator.Messages.Fields.NFVResource
 NFVResources:
   networkResources: [NetwResource1Node1, NetwResource2Node1, nres1]
   serviceResources: [ServiceResource1Node1, ServiceResource2Node1]
 version: 965
ost12: !!Orchestrator.Messages.Fields.NFVResource
 NFVResources:
   networkResources: [NetwResource1Node2, NetwResource2Node2]
   serviceResources: [ServiceResource1Node2, ServiceResource2Node2]
 version: 5355
ost01: !!Orchestrator.Messages.Fields.NFVResource
 NFVResources:
   networkResources: [NetwResource1Node3, NetwResource2Node3]
   serviceResources: [ServiceResource1Node3, ServiceResource2Node3]
 version: 819
ost292: !!Orchestrator.Messages.Fields.NFVResource
 NFVResources:
   networkResources: [NetwResource1Node28, NetwResource2Node28]
   serviceResources: [ServiceResource1Node28, ServiceResource2Node28]
 version: 1908
ost312: !!Orchestrator.Messages.Fields.NFVResource
 NFVResources:
   networkResources: [NetwResource1Node27, NetwResource2Node27]
   serviceResources: [ServiceResource1Node27, ServiceResource2Node27]
 version: 3449
ost331: !!Orchestrator.Messages.Fields.NFVResource
 NFVResources:
   networkResources: [NetwResource1Node29, NetwResource2Node29, nres2]
   serviceResources: [ServiceResource1Node29, ServiceResource2Node29]
 version: 1942
oot@n27:/tmp/pycore.36111/n27.conf# █
n: 1942
```

```
root@n2:/tmp/pycore.36111/n2.conf# ip link set dev eth0 up | ip link set dev e
root@n2:/tmp/pycore.36111/n2.conf# 
```

```
root@n25:/tmp/pycore.36111/n25.conf# ip link set dev eth0 down | ip link set d
<| ip link set dev eth1 up | ip link set dev eth2 up
root@n25:/tmp/pycore.36111/n25.conf# 
```

Figure 4.37:Merging Partitions : Merging Partitions : Merged the separated partitions and resource information modifications. View from node n27.

5 Summary and Perspectives

The proposed solution is only intended as a prototype without any specific assumption of virtual resource provider's architecture/implementation requirements. Therefore the resource information that is synchronized is meant only as a placeholder with the possibility of classifications and not associated with any service provider's designs. It is important to note that there is a resource information catalog that is being persisted and synchronized, such a design will have to be critically analyzed with regards to the security aspects necessary for real-life deployments. The solution aims at identifying the necessary transactions at the service layer for proper achieving the target state, therefore to some extent the solution is agnostic of the structure of the global resource information catalog and may employ additional security or classification encoding if needed. Improvements in that regard may include but not limited to byte encoded solution to topology identification mechanism. The classification of network resources that may be used in such high resilient systems also needs to be identified and if possible, those communication for resource information should occur through a custom protocol designated to accommodate these resource classifications. A deeper analysis of the operational mechanism for logical separation of different physical elements should also be conducted and integrated with the results of this study, possibly in a further iteration.

Another observation from the analysis of an asymmetric topology construct is the possibility to construct a topology transition history. If a topology transitions to a final state from an initial state, it may have multiple possibilities of intermediate topologies possible. As a result of that one may achieve different outcomes for network-state analysis done by the BFS module. Such an outcome is expected and is mathematically justified, however its extent and reasons underlying are too complicated for the analysis of the present study. However that observation may be utilized to come to a qualitative estimation of the intermediate topology changes that happened based on the differing outcomes for network-state analysis. Such an observation may be utilized to construct a transition history for a topology in addition to estimation of network connectedness and partition identification.

The proposed prototype incorporates CoAP on the service layer, however that is coincidental and one may have pure UDP implementations after isolation of network resources and synchronization requirements for a particular deployment. The only requirement is capability to provide multicast listening on the interfaces for any Wireless Mesh Network(WMN) node and associated network layer routing protocols for unicast communication. Although the simulation architecture looks at the topology from a wired connection perspective, after isolation of suitable wireless routing protocols the solution can be extended to dynamic WMN systems. The transitioning methods have not been discussed in this solution owing to its broad scope unrelated with the specificities of the present problem statement.

6 Abbreviations

BFS:

Breadth First Search (Algorithm)

CoAP:

Constrained Application Protocol

CNFVO:

Cluster Network Function Virtualization Orchestrator

NFV:

Network Function Virtualization

NFVI:

Network Function Virtualization Infrastructure

NFVO:

Network Function Virtualization Orchestrator

OSPF:

Open Shortest Path First (Algorithm)

VIM:

Virtual Infrastructure Management

WMN:

Wireless Mesh Network

7 References

1. Anon., n.d. *CORE: Common Open Research Emulator.* s.l.:Boeing Company.
2. ETSI, 2015-12. *ETSI Report on SDN Usage in NFV Architectural Framework. ETSI GS NFV-EVE 005 v1.1.1.* s.l.:s.n.
3. Frick, G. et al., 2018. Distributed NFV Orchestration in a WMN-based Disaster Network. *IEEE.*
4. Frick, G. et al., 2019. NFV Resource Advertisement and Discovery Protocol for a Distributed NFV Orchestration in a WMN-based Disaster Network. *IEEE.*
5. Matthias Kovatsch, M. L. D. I. O. D. P. K. H. A. K., n.d. *Eclipse Californium.* s.l.:s.n.
6. Moy, J., n.d. *Request for Comments: 2328 OSPF Version 2.* s.l.:s.n.
7. Shelby, Z., Hartke, K. & Bormann, C., n.d. *The Constrained Application Protocol (CoAP) (RFC 7252),* s.l.: s.n.

8 Appendix

8.1 CORE Setup, usage and debugging

Directory structure for the simulation to successfully startup , the jar executable has to be placed in the folder : */home/core-vm/CoRE-Models/DistributedOrchestration/ex/* and the name of the executable is *ResourceSynchronization.jar*. For modification of resource information during runtime a python script is provided which can be used as an extension API to add or remove those details. They have to be placed under the directory structure */home/core-vm/CoRE-Models/DistributedOrchestration/ex/* under the name *extensions.py*. They could be used within a simulation as follows

1. Open a terminal of the node whose resource information is to be changed. Execute the following command , ***python3 extensions.py --add --network {resource_name}*** . The ***--add*** also has a ***--del*** switch and ***--network*** also has a ***--service*** switch.

In case of external validation of proper functioning of the Synchronization application , the validation script could be used. The validator script *validator.py* needs the container files for all the nodes in the simulation. The following steps may be employed to successfully validate the resource information externally for verification.

1. In the current directory where the validation script is located , open a shell terminal and copy the CORE-Emulator temporary container folder (typically located at /tmp/pycore.xxx, [for version 6.0.0] where xxx denotes a number sequence used by emulator's daemon which changes with each simulation execution) to a folder named ***pwd*** using command ***cp -r /tmp/pycore.xxx pwd***
2. Then run the validation script as ***python3 validator.py***.

Potential problems with setting up the simulation environment may involve the following -

- Improper parsing of the original .imn file if CORE Emulator version does not match. The .imn file provided with this solution works for version 6.0.0, if there are any parsing issues consider switching to the above version.
- In case of the executable not functioning during simulation startup, consider restarting UserDefinedServices by right clicking on the node where you wish to start the synchronization application, select UserDefinedService and click start option. Alternatively the commands shown in Figure.8.3 could be manually executed from terminal of a node. In case there are any problems with associating ports, check if UDP ports 5600 and 5700 are free and if there were any processes previously running (highly unlikely but still to be on safer side !). The assigned ports can be checked by executing command ***lsof -i -P*** and noting the pid of the process associated with those ports. If there is a process associated with those ports preventing a start/restart of the synchronization application , consider killing those processes first before restarting application by executing ***kill -9 {pid-of-the-application}.***
- If the extension API's don't work, consider if the host VM/system has python3 and python yaml parsers (Unlikely since CORE also demands these modules for its daemon functionality). The procedure to install them in case they are not present are standard and may be found on CORE documentation.
- The software solution is internally managing redundancy checks to make sure that every node has latest resource information of the corresponding partition, but one may employ the external validator to check the same. The external validator transitively checks resource information of node n1 as shown in Figure.8.1 with the remaining nodes to give a result as shown in Figure.8.5. Make sure that the CORE runtime container temp information files are stored in the same folder as *validator.py* and within a folder named ***pwd***. So the resulting directory structure relative to validator.py may look like –

```
--Validation
        |
        | -- validator.py
        | -- pwd
              |
              | -- {Container temp information files}
```

Make sure that simulation runs for a few minutes for the synchronization process to complete before checking with the help of validators .

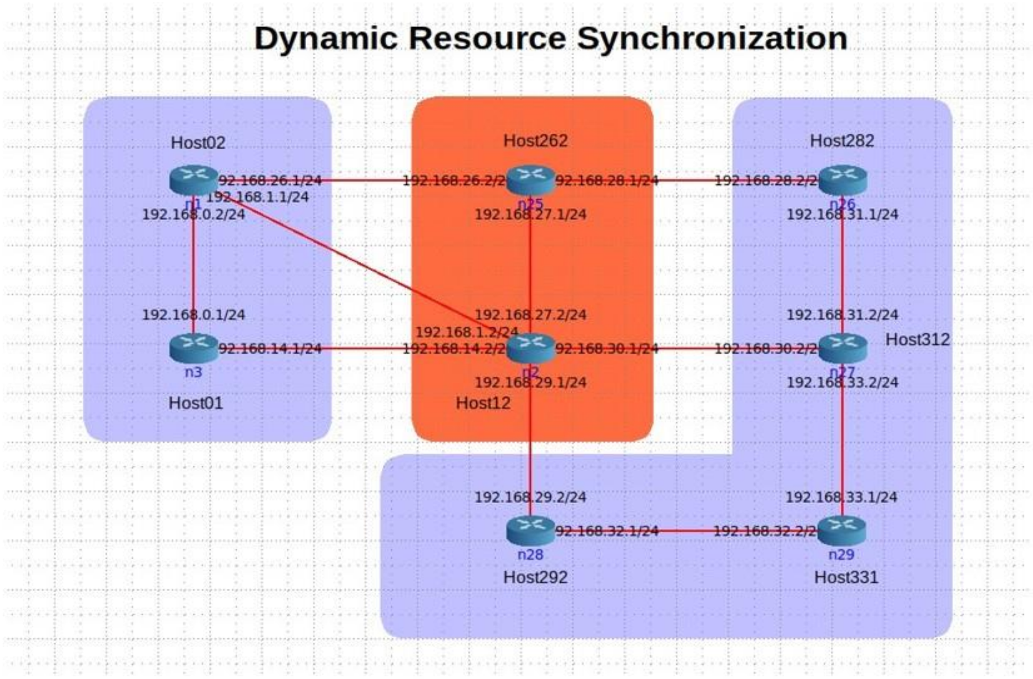
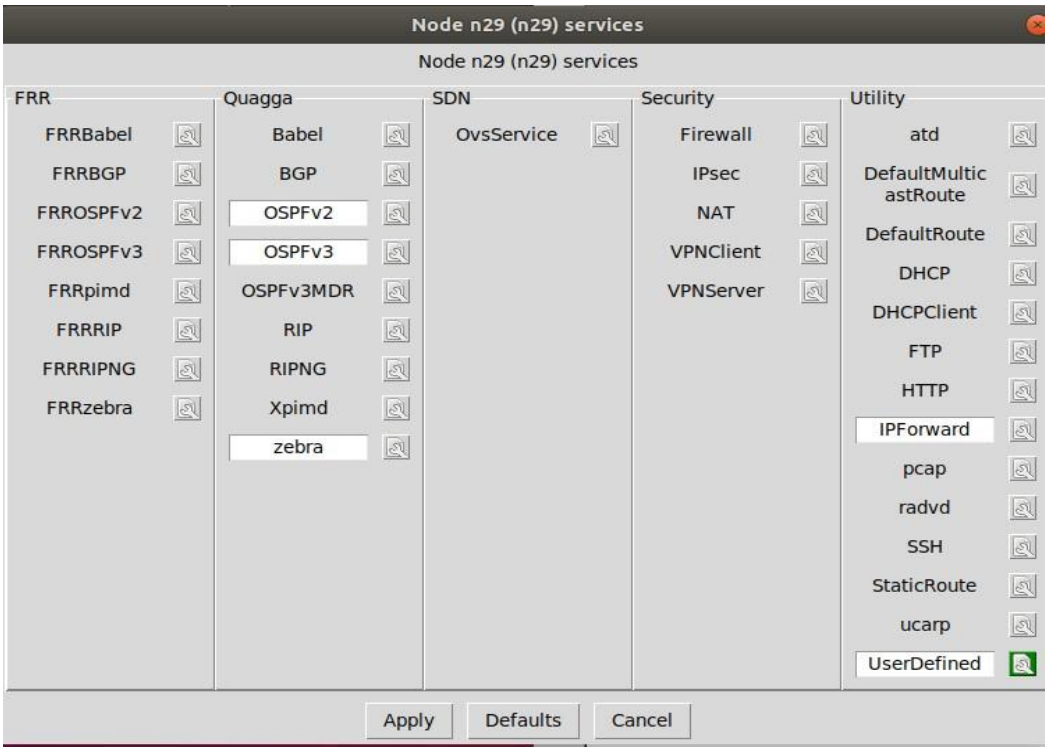


Figure 8.1:CORE Emulator topology model

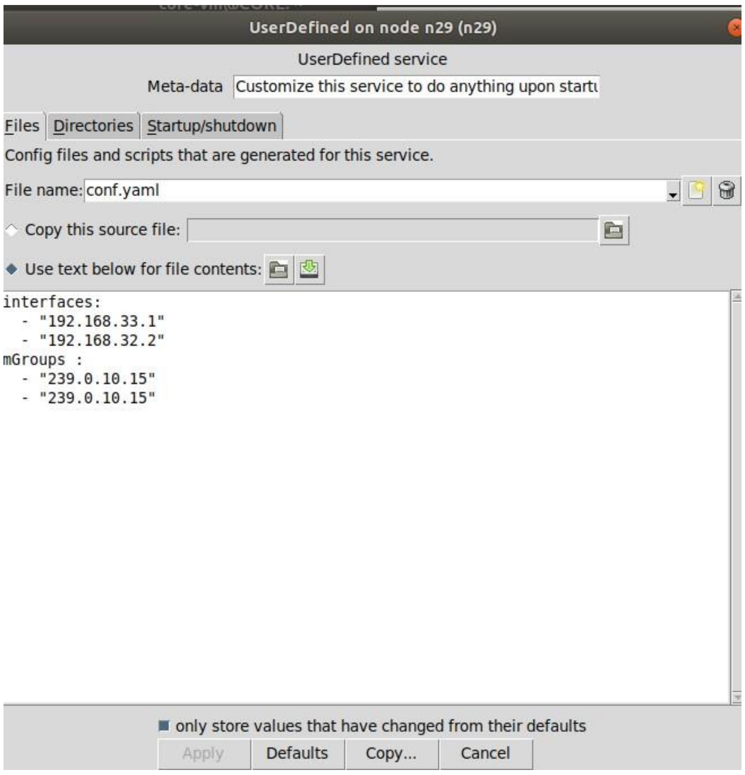


Node n29 (n29) services

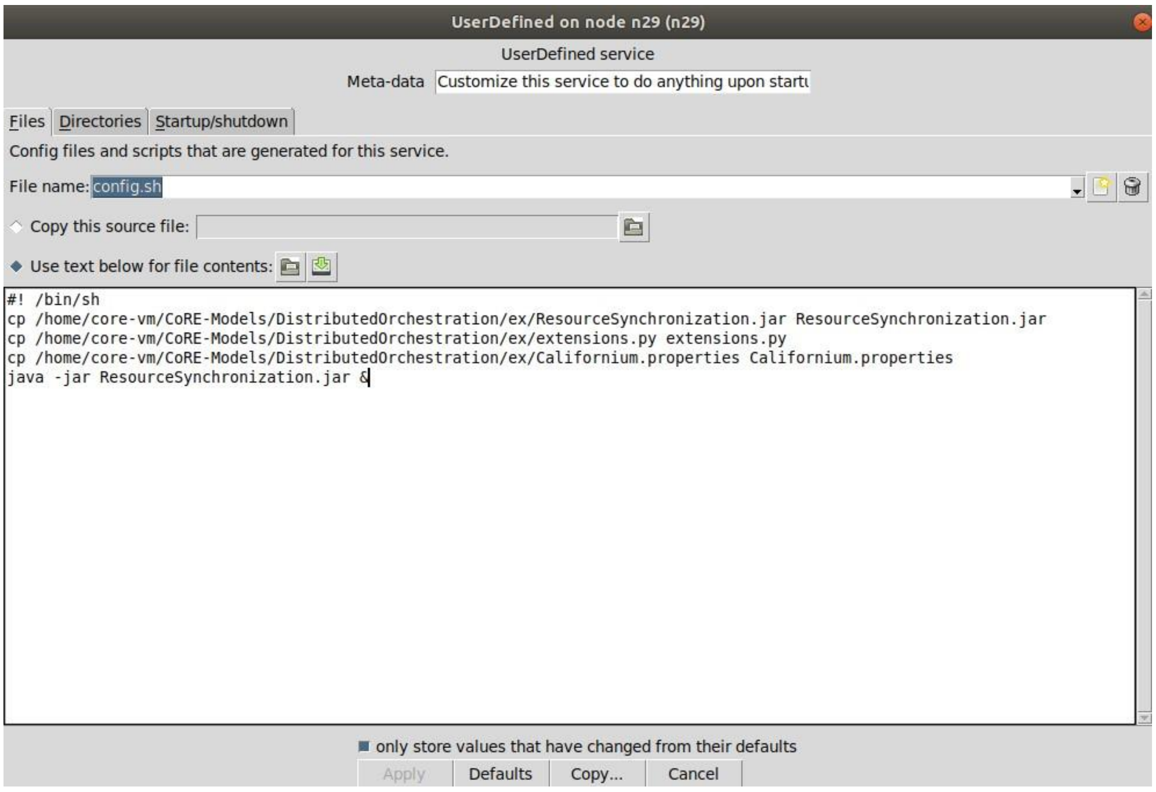
Node n29 (n29) services

FRR	Quagga	SDN	Security	Utility
FRRBabel	Babel	OvsService	Firewall	atd
FRRBGP	BGP		IPsec	DefaultMulticastRoute
FRROSPFv2	[✓ OSPFv2]		NAT	DefaultRoute
FRROSPFv3	[✓ OSPFv3]		VPNClient	DHCP
FRRpimd	OSPFv3MDR		VPNServer	DHCPClient
FRRRIP	RIP			FTP
FRRRIPNG	RIPNG			HTTP
FRRzebra	Xpimd			[✓ IPForward]
	[✓ zebra]			pcap
				radvd
				SSH
				StaticRoute
				ucarp
				[✓ UserDefined]

Figure 8.2:Configuration of a router node in CORE Emulator mesh network.

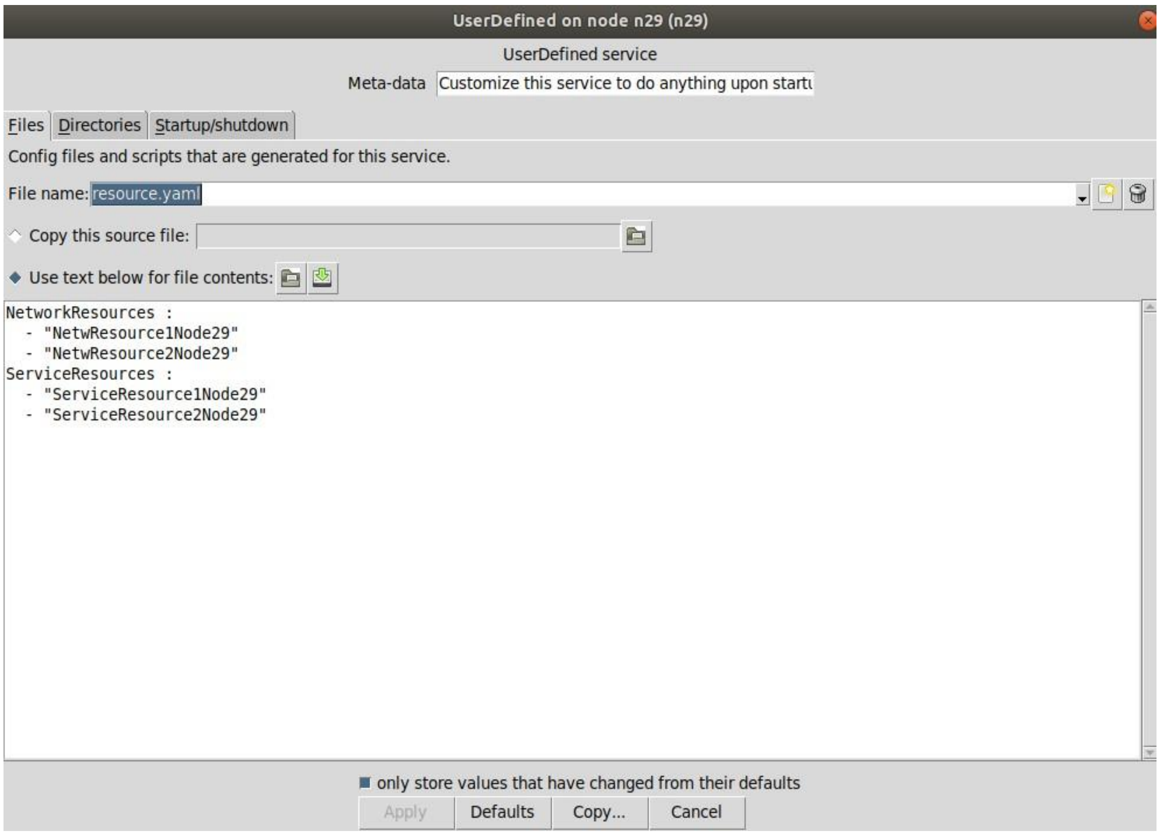

UserDefined on node n29 (n29)
UserDefined service
Meta-data
Customize this service to do anything upon startu
Files
Directories
Startup/shutdown
Config files and scripts that are generated for this service.
File name: conf.yaml
Copy this source file:
Use text below for file contents:
only store values that have changed from their defaults
Apply
Defaults
Copy...
Cancel

```
interfaces:
  - "192.168.33.1"
  - "192.168.32.2"
mGroups :
  - "239.0.10.15"
  - "239.0.10.15"
```


UserDefined on node n29 (n29)
UserDefined service
Meta-data
Customize this service to do anything upon startu
Files
Directories
Startup/shutdown
Config files and scripts that are generated for this service.
File name: config.sh
Copy this source file:
Use text below for file contents:
only store values that have changed from their defaults
Apply
Defaults
Copy...
Cancel

```
#! /bin/sh
cp /home/core-vm/CoRE-Models/DistributedOrchestration/ex/ResourceSynchronization.jar ResourceSynchronization.jar
cp /home/core-vm/CoRE-Models/DistributedOrchestration/ex/extensions.py extensions.py
cp /home/core-vm/CoRE-Models/DistributedOrchestration/ex/Californium.properties Californium.properties
java -jar ResourceSynchronization.jar &
```

Figure 8.3:Configuration and setup files for application.



Meta-data: Customize this service to do anything upon startu

Config files and scripts that are generated for this service.

File name: resource.yaml

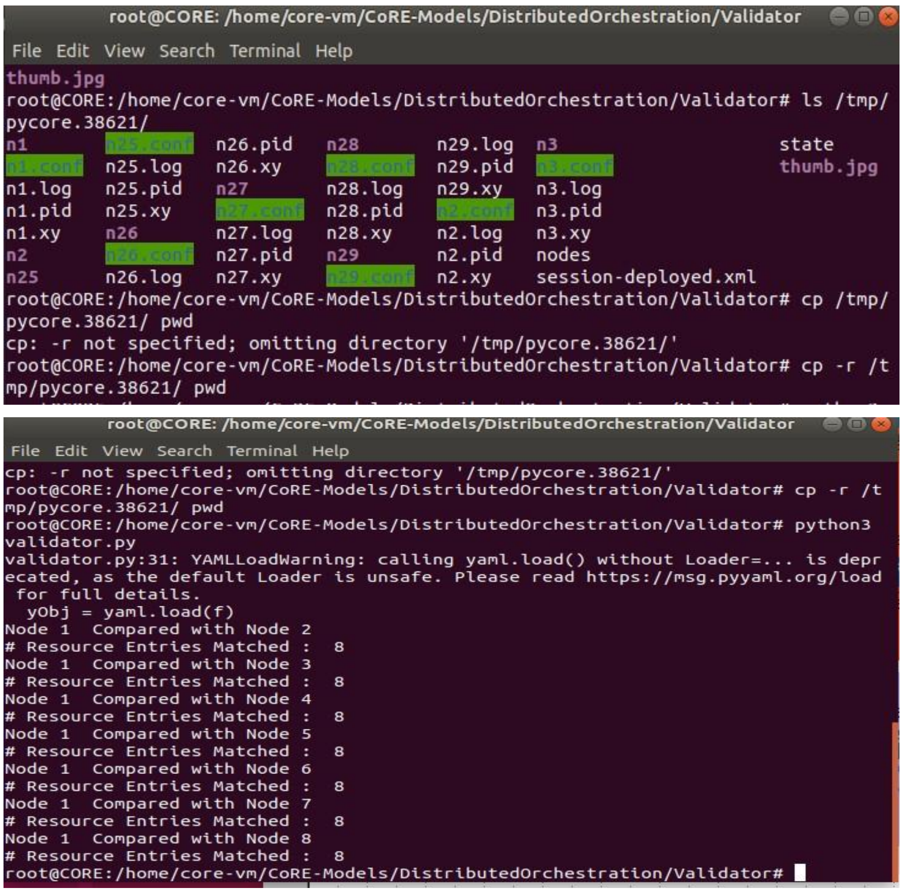
Copy this source file:

Use text below for file contents:

```
NetworkResources :
  - "NetwResource1Node29"
  - "NetwResource2Node29"
ServiceResources :
  - "ServiceResource1Node29"
  - "ServiceResource2Node29"
```

only store values that have changed from their defaults

Figure 8.4:Example of resource information configuration yaml file


root@CORE: /home/core-vm/CoRE-Models/DistributedOrchestration/Validator
File Edit View Search Terminal Help

```
thumb.jpg
root@CORE:/home/core-vm/CoRE-Models/DistributedOrchestration/Validator# ls /tmp/
pycore.38621/
n1          n25.conf  n26.pid   n28        n29.log  n3              state
n1.conf     n25.log   n26.xy    n28.conf   n29.pid  n3.conf         thumb.jpg
n1.log      n25.pid   n27       n28.log    n29.xy   n3.log
n1.pid      n25.xy    n27.conf  n28.pid    n2.conf  n3.pid
n1.xy       n26       n27.log   n28.xy     n2.log   n3.xy
n2          n26.conf  n27.pid   n29        n2.pid   nodes
n25         n26.log   n27.xy    n29.conf   n2.xy    session-deployed.xml
root@CORE:/home/core-vm/CoRE-Models/DistributedOrchestration/Validator# cp /tmp/
pycore.38621/ pwd
cp: -r not specified; omitting directory '/tmp/pycore.38621/'
root@CORE:/home/core-vm/CoRE-Models/DistributedOrchestration/Validator# cp -r /t
mp/pycore.38621/ pwd
```

```
cp: -r not specified; omitting directory '/tmp/pycore.38621/'
root@CORE:/home/core-vm/CoRE-Models/DistributedOrchestration/Validator# cp -r /t
mp/pycore.38621/ pwd
root@CORE:/home/core-vm/CoRE-Models/DistributedOrchestration/Validator# python3
validator.py
validator.py:31: YAMLLoadWarning: calling yaml.load() without Loader=... is depr
ecated, as the default Loader is unsafe. Please read https://msg.pyyaml.org/load
 for full details.
  yObj = yaml.load(f)
Node 1  Compared with Node 2
# Resource Entries Matched :  8
Node 1  Compared with Node 3
# Resource Entries Matched :  8
Node 1  Compared with Node 4
# Resource Entries Matched :  8
Node 1  Compared with Node 5
# Resource Entries Matched :  8
Node 1  Compared with Node 6
# Resource Entries Matched :  8
Node 1  Compared with Node 7
# Resource Entries Matched :  8
Node 1  Compared with Node 8
# Resource Entries Matched :  8
root@CORE:/home/core-vm/CoRE-Models/DistributedOrchestration/Validator#
```

Figure 8.5: Usage of external validation scripts.