**Prioritizing SOS Packets in an IP Network**

Interim Report Submitted in partial fulfilment towards requirements of CSC-573

Submitted By,

Anand Jayaram

Balaji Shekar

Neela Krishna Teja Tadikonda\*

Swetha Murali

Under the Guidance of

**Dr. Rudra Dutta**



North Carolina State University

Raleigh, North Carolina

**CONTENTS**

|  |  |  |
| --- | --- | --- |
| **S No** | **Topic** | **Page No** |
| 1 | Problem Statement | 3 |
| 2 | Project Components | 4 |
| 3 | High Level design | 7 |
| 4 | Low Level Design and Development | 9 |
| 5 | Per Member Responsibility | 13 |
| 6 | Project Timeline | 14 |
| 7 | Verification and Validation Plan | 14 |
| 8 | References | 17 |

**1. Problem Statement**

In times of catastrophic events, distress messages can be sent over IP networks. These distress messages should not be treated with the best effort service available to regular traffic. Our project aims to provide priority service to these messages and ensure that they are delivered in irrespective of the load on the network.

SOS or Save Our Souls is a distress message generated by a standalone sensor node or a user application in response to some emergency scenario. The message consists of just the time and identity of the transmitting node. The node can be as simple as a Fire alarm/smoke detector/app running on user’s smart phone. These nodes will be a part of a participating Local Area Network or a Wide Area Network in a campus or research facility which has its own response teams at the ready. Various Listener nodes are also present in this network which can respond to this distress signal. These may be applications running on police vehicles, 911 responder or just an alert system for building maintenance or security.

The role of the network in this system is to ensure that the SOS messages are received by every node in the system and that they get highest priority, however loaded the network is. IP Multicast will be used to send these packets to every host that is expecting SOS messages in a swift and efficient manner.

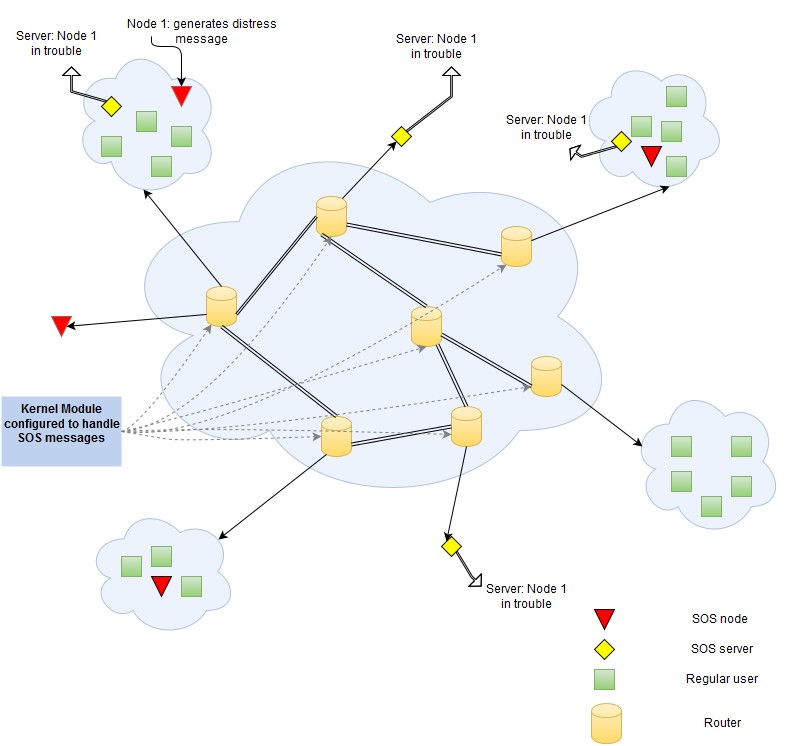


Figure 1.1: Possible Topology of the Network: The Distress message is generated at SOS node 1(red triangle). The message “Node 1 in trouble” is received at all listener nodes in the network

**2. Project Components**

**2.1 Platform**

Linux machines acting as routers and nodes running Ubuntu in GENI environment.

**2.2 Areas of project**

1. Creating a multicast routing environment – Set up a multicast environment with a source, clients, multicast routers, routing protocol (PIM) and a group management protocol (IGMP).
2. Application Development – SOS applications are written in the user space to generate and receive the SOS message and respond accordingly.
3. Classification and Prioritizing – Kernel modules loaded into the router must identify and mark SOS packets. Prioritization is done using priority queuing discipline provided by Linux Traffic Control mechanism (TC).

**2.3 Major Components**

1. Hosts running SOS applications: C1, C2, C3, C4 are user nodes (Linux machines) that will run SOS applications along with some regular traffic generators like Scapy.
2. Underlying network: R1- R7 will be Linux machines running as L3 routers with our kernel modules running on them.
3. Kernel module and Router Configuration: Each of the Routers R1-R7 will be running Kernel Modules used to identify the SOS packets. The QDisc configuration will be used to prioritize the SOS packets over the regular traffic (the overview of this particular component has been provided in sections 2.3.1 and 2.3.2 below).

**2.3.1 Kernel Module**

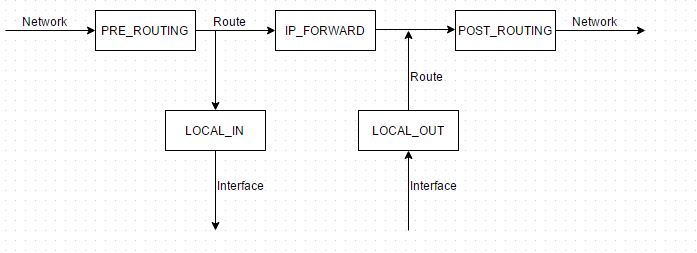
Coding Kernel Modules require understanding of the following:

1. Netfilter Hooks
2. Socket Buffer Structure

**Netfilter Hooks:** Netfilter is Linux kernel functionality which lets us analyze/modify packets at various points in the Linux network Stack. This is achieved by using 5 hooks which act like interrupt:

* NF\_INET\_PRE\_ROUTING,
* NF\_INET\_LOCAL\_IN,
* NF\_INET\_FORWARD,
* NF\_INET\_LOCAL\_OUT,
* NF\_INET\_POST\_ROUTING.

These hooks act as access points to the packets at various locations in the network stack.



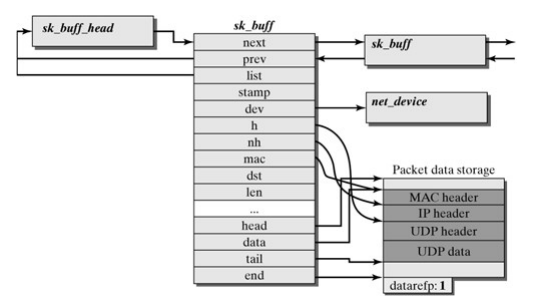
Whenever a packet gets to any of the 5 points above in the network stack, the corresponding hook functions are called if they are registered. We can register the hooks in a kernel module and use the hook functions to access the packet. The hook functions return 5 values based on how we want the Linux Kernel to further process the packets. These Return Values are

* NF\_ACCEPT : let the packet pass.
* NF\_DROP : drop the packet.
* NF\_STOLEN : take the packet and don’t let the packet pass.
* NF\_QUEUE : queue the packet, usually for user space handling.
* NF\_REPEAT : call the hook again.

**Socket Buffer Structure:** Packets in Linux are managed in the buffers called socket buffers. The implementation of socket buffers is flexible and is independent of specific protocols. The socket buffer data structure consists of 2 parts:

1. Packet data – A block of memory where the actual data that is to be transmitted over the network or received from the data-link layer is stored.

2. sk\_buff - A structure for storing management data required for processing the packet. The sk\_buff structure stores pointers that point to the headers of different network layers, timers, flags etc. It also has pointers for managing socket buffer queues. This structure is shown below:



The different pointers in the sk\_buff structure are documented below:

1. *next, prev* and *list*: Pointers related to socket buffer queue management. The next and prev pointers point to the location of the next and previous socket buffer structures in the queue. The list pointer points to the queue where the socket buffer is currently loaded.
2. *stamp*: Time stamp that specifies when the packet arrived for processing.
3. *dev*: The dev pointer points to the structure that contains the interface over which the packet arrived. It is set to the interface over which packet is to be sent once routing decision is made.
4. *h, nh* and *mac*: Pointers to the transport, network and data-link layer headers respectively.
5. *head* and *end*: Point to the start and finish locations for storing packet data.
6. *data* and *tail*: Pointers that point to the start location of the data and the tail information of the packet currently being processed.

The *datarefp* variable is not a part of the actual sk\_buff structure. It keeps track of the number of references made to the packet data unit. Since we are working in the kernel space accessing proper memory locations is a stringent requirement to ensure stability of Linux System.

**2.3.2Queuing Disciplines in Linux:**

Linux provides ways to manage the bandwidth for certain users in form of traffic control and policing using queueing disciplines. We have ingress Qdiscs, present at the entrance after entering  the routers network interface and egress Qdiscs present at the end before leaving the routers network interface. Ingress Qdiscs are used to police the incoming packets, i.e to drop out of policy packets. Egress Qdiscs are used for the traffic control.  This project is more interested in the Egress Qdisc and henceforth all Qdiscs referred from here will be understood as Egress Qdiscs. Packets are queued into several queues and each queue is scheduled depending on the type of predefined scheduling algorithm associated with the Queueing discipline. Each Network Interface can be configured with a Queueing discipline. Queues could be from simple FIFO queue (*pfifo\_fast –* usually default Qdisc in Linux) to a multi-level priority based queues (PRIO – a class based Qdisc). There can also be other types of queues and scheduling algorithm associated with those queues. Queuing disciplines can also be arranged hierarchically.  The Qdisc at beginning of the hierarchy is called root Qdisc. We have two types of Qdisc, classless Qdisc’s and classful Qdisc’s. The classful Qdisc are represented using <major number> :< minor number> notation, which are called handles. Major number is unique across the queueing discipline. Minor number is associated with each class in the Queueing discipline. Linux provides *tc* to configure these Queueing disciplines. For example to configure a Qdisc SFQ (Stochastic Fairness Queueing) for interface eth0 we use tc command as follows.

tc qdisc add dev eth0 root sfq perturb 10

Adds Qdisc SFQ to the Network Interface eth0 at the root.

tc -s -d qdisc ls

Provides the statistics dump associated with qdiscs such as sent bytes, sent packets etc., configured for the Linux system.

Stochastic Fairness Queueing is an implementation from the family of Fairness queue implementations although not very accurate to other implementations, but is fair enough. It internally divides the traffic over a limited number of queues using a hashing algorithm and dequeue using round-robin from each of them. Using this hashing multiple sessions can end up in same bucket, which will reduce the chance of sending a packet; SFQ makes sure that it changes the hashing algorithm frequently so that colliding sessions will only do from small time.

**3. High Level Description**

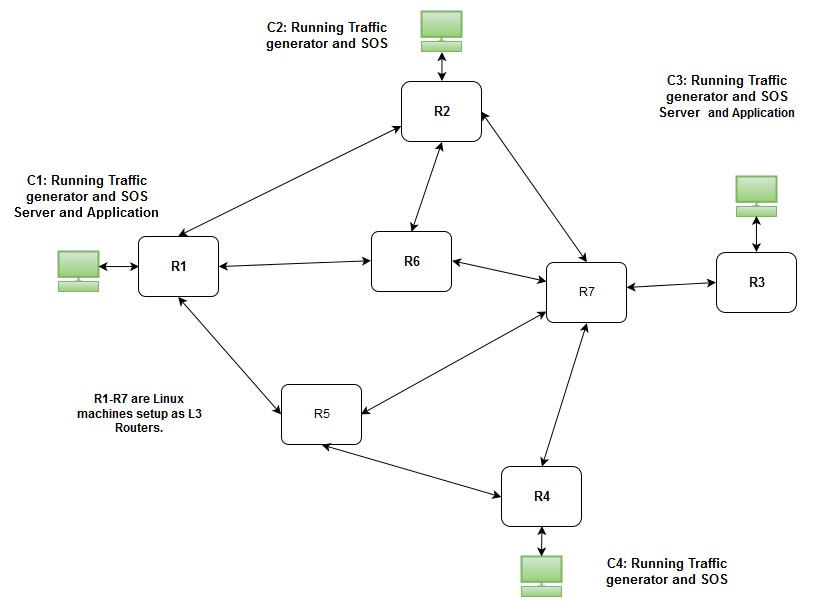


Figure 3.1: Overall Working of System

Fig 3.1 gives an overall working idea of the system. We have a set of regular users (C1,C2,C3,C4) running a SOS application that generate packets to be sent to other users who are listening (by running a similar application) for such packets to take necessary action. Next, we have the underlying network of routers that separate these hosts geographically. To keep the explanation simple and illustration of the idea more clear, we stuck to a pool of 7 Linux machines that can simulate routers (R1 through R7), connected to a LAN of computers which run the mentioned SOS application.

The flow starts when one user is in distress and begins transmitting SOS packets to other users who can help. These packets are sourced from the user's source address and sent to a multicast destination group to which all the hosts listening belong. Each packet reaches the pool of routers and is routed or flooded along all paths that lead to a multicast listener. Within a router, the SOS packets will be identified on the basis of their content and prioritized accordingly. After getting routed through the network, the packets finally reach the destination hosts where they are identified by the SOS application as distress packets.

We adopted a server-listener model, which will help keep the discussion simple. A server is the application running on a node that generates SOS messages to be sent and listeners are applications running on all the other nodes in the system waiting to receive these messages.

In the given figure, Node C1 generates distress messages (running the server program) and sends them to the underlying network to be flooded towards other nodes running the SOS application (listeners).

**4. Low Level Design and Development**

In order to detail a low-level design and the proposed development plan, we break down our project into parts that can be designed and integrated in the end to achieve our goal. These parts are:

1. Topology Setup in ExoGENI
2. Traffic Generation using Scapy (a python tool)
3. Coding Kernel Modules
4. Ensuring Packet Priority

**4.1 Topology Setup in ExoGENI**

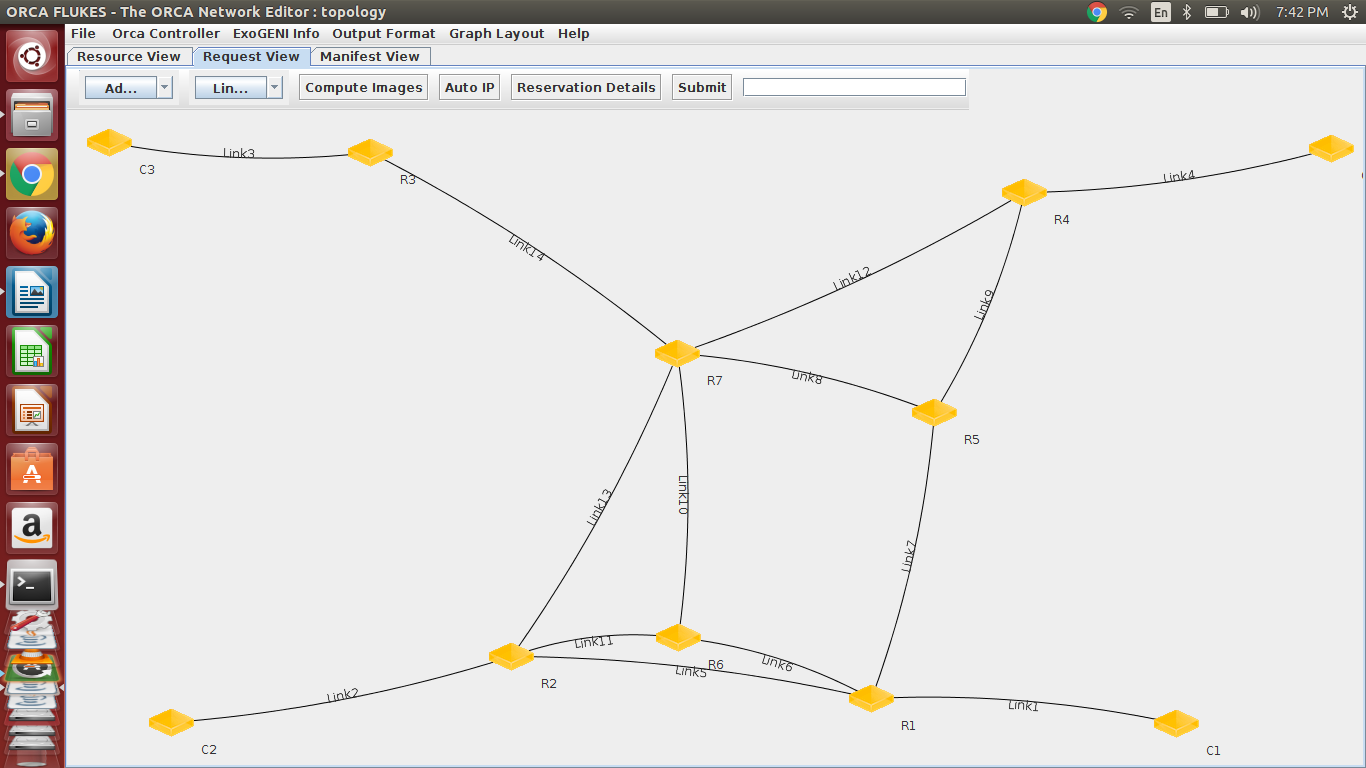
We used GENI to create a topology of hosts and routers. As illustrated in the high level design, we have 4 hosts (C1, C2, C3, C4) belonging in different LANs who participate in sending or receiving SOS packets by using the application explained in the previous section. To carry this traffic between the LANs, we have a network of 7 (R1 through R7) that each have our kernel module loaded in order to prioritize SOS packets. Figure 4.1 represents the topology.

Figure 4.1 Topology of the Network

All the nodes in the topology (hosts and routers) are Linux based hosts running Ubuntu 14.04 LTS. We selected this configuration as it includes all our required packages (quagga, pimd etc.). Here are the network parameters of the topology:

*C1's LAN: 10.0.0.0/24*

*Host C1: 10.0.0.2/24*

*Router R1's local LAN interface: 10.0.0.1/24*

*C2's LAN: 10.0.1.0/24*

*Host C2: 10.0.1.2/24*

*Router R2's local LAN interface: 10.0.1.1/24*

*C3's LAN: 10.0.2.0/24*

*Host C3: 10.0.2.2/24*

*Router R3's local LAN interface: 10.0.2.1/24*

*C4's LAN: 10.0.3.0/24*

*Host C4: 10.0.3.2/24*

*Router R4's local LAN interface: 10.0.3.1/24*

|  |  |
| --- | --- |
| Link | Network |
| R1 – R2 | 10.0.4.0/30 |
| R1 – R6 | 10.0.4.4/30 |
| R1 – R5 | 10.0.4.8/30 |
| R2 – R6 | 10.0.4.12/30 |
| R2 – R7 | 10.0.4.16/30 |
| R6 – R7 | 10.0.4.20/30 |
| R5 – R7 | 10.0.4.24/30 |
| R5 – R4 | 10.0.4.28/30 |
| R4 – R7 | 10.0.4.32/30 |
| R7 – R3 | 10.0.4.36/30 |

In the above table, in any pair Ri - Rj, Ri has the first available address in the corresponding subnet.

We used static routing inside each LAN for every host to reach its gateway and OSPF between routers to automatically advertise directly connected networks to neighbors. We used a package called quagga [6] that helped us turn nodes (R1 through R7) into OSPF routers. The package ran multiple daemons to achieve this functionality. The zebra daemon let us use the nodes' interfaces as a router's interface and the daemon ospfd gave us a CLI for OSPF related configuration.

Our application running on the hosts sends SOS data on a multicast group that we have set up using the PIMD[7] package. We used PIM-SM to create the multicast routing environment. The multicast group we are using is 239.0.0.1 and we have all our hosts (C1 through C4) join the group.

**4.2 Traffic Generation using scapy**

We used scapy to generate SOS and normal packets. Scapy is a powerful interactive packet manipulation program. It can forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies, and much more. It can easily handle most classical tasks like scanning, tracerouting, probing, unit tests, attacks or network discovery.

In our project, the emergency application sends a SOS packet which will have a special bit pattern in the beginning. This bit pattern will be used to identify the SOS packets by the kernel module. For instance, this identification can be done when the Morse code binary form for SOS is found in the payload (Morse code for SOS : ...---... , binary equivalent: 101010001110111011100010101).The listener application listens on the specified port for the arrival of the SOS packet and when it receives an emergency packet it prints a message indicating the reception of the SOS packet.

* 1. **Coding the Kernel Modules**

The current design of the project uses kernel modules to classify the incoming traffic as SOS packets or regular traffic. To do this we register an NF\_INET\_PRE\_ROUTING netfilter hook. This hook will generate an interrupt for every packet coming into the router and the module then peeks into the packet's content to see if the packet is a SOS packet. If the packet is identified as a SOS packet, the module will mark its DSCP field with our designated DSCP value for SOS. Using this DSCP value the packet will be prioritized by the Traffic Control module (details in the next section).

A basic overview of the module is as given below:

struct nf\_hook\_ops netfilter\_ops;

int hook\_function(int hooknum, sk\_buff \*skb,…..)

{

// the reference to the packet will be available in the sk\_buff \*skb

if(packet content = special pattern) //SOS Packet

modify DSCP field, return NF\_ACCEPT;

else // Regular Packet

NF\_ACCEPT;

}

int init\_module()

{ netfilter\_ops.hook = hook\_function;

netfilter\_ops.pf = PF\_INET;

netfilter\_ops.hooknum = NF\_INET\_PRE\_ROUTING;

netfilter\_ops.priority = NF\_IP\_PRI\_FIRST;

nf\_register\_hook(&netfilter\_ops);

return 0;

}

void cleanup\_module()

{

nf\_unregister\_hook(&netfilter\_ops);

}

**4.4 Ensuring Packet Priority**

The SOS traffic is classified based on the DSCP value marked by the kernel module and scheduled first from each network interface. It involves configuring each linux router with the following.

tc qdisc add dev <INTERFACE> root handle 1: prio

Creates a PRIO Qdisc with 3 classes. Here we have a SOS class (:1) with highest priority and two Non-SOS classes (:2 , :3) as PRIO Qdisc provides 3 classes. After this filters are attached to Qdisc to classify packets into respective class queues.

tc filter add dev <INTERFACE> parent 1:0 protocol ip prio 1 u32 match ip tos <DSCP VALUE> 0xff flowid 1:1

tc filter add dev eth0 protocol ip parent 1:0 prio 2 flowid 1:2

tc filter add dev eth0 protocol ip parent 1:0 prio 2 flowid 1:3

All Non-SOS packets will be classify into either into class 1:2 or 1:3.

SFQ Qdisc used in hierarchy at the end of each class in PRIO Qdisc as follows at each interface.

tc qdisc add dev <INTERFACE> parent 1:1 handle 10: sfq

tc qdisc add dev <INTERFACE> parent 1:2 handle 20: sfq

tc qdisc add dev <INTERFACE> parent 1:3 handle 30: sfq

**5. Per Member Responsibilities**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Anand | Balaji | Neel Krishna Teja | Swetha |
| Setting up the network topology |  |  |  |  |
| Creating Multicast Groups |  |  |  |  |
| Coding the SOS Application |  |  |  |  |
| Coding the Kernel Modules to handle queuing and scheduling of the SOS packets |  |  |  |  |
| Coding the traffic generators |  |  |  |  |
| Testing and verifying the prioritized delivery of SOS packets |  |  |  |  |
| Performance Analysis |  |  |  |  |

**6. Project Timeline**

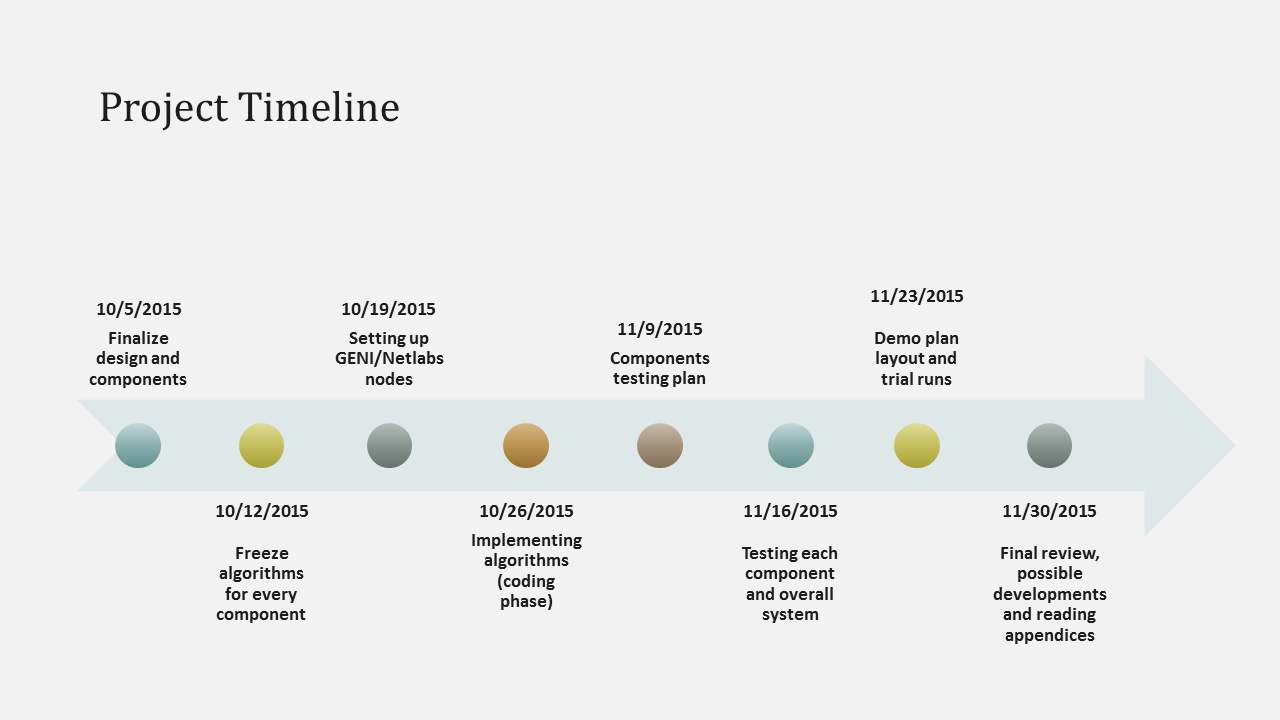


Figure 6.1 Project Timeline

1. **Verification and Validation Plan**

**7.1 Test Bed Setup**

* GENI platform with Ubuntu 14.04 installed on each node.
* Host running the SOS application which floods the network with SOS packets when an emergency occurs.
* Hosts listening for the SOS packets which performs some action when a SOS packet is received.
* GENI nodes configured as routers with the Kernel module for classification and TC for priority scheduling.
* Scapy application running on all the other host machines to generate real time non-emergency packets in the network.

**7.2 Test Plan**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test ID** | **Test Name** | **Action** | **Expected Behaviour** | **Conclusion** |
| T1 | Verifying SOS Application | Run the SOS application to send packets with DSCP value in the IP header corresponding to that of SOS packets. Capture the packets at the outgoing network interface. | The packet dump shall show the DSCP field set to the value corresponding to that of SOS packets. | The SOS application generates the SOS packets with the correct DSCP value. |
| T2 | Verifying Multicast delivery | Running the SOS application to send packets to the multicast group | Look for console alert at all listener nodes in the multicast group | The SOS application is multicasting the packets. |
| T3 | Identifying the SOS packets at router. | Load Kernel module on each router and start generating SOS packets. | The kernel module logs will show the number of SOS packets it serves. | The router has used the kernel module to identify SOS packets. |
| T4 | Assuring Higher priority for SOS packets | Generate SOS and regular traffic at the server. | At listener nodes SOS packets are delivered before regular packets. | SOS packets are being pushed to front of the Queue. |
| T5 | Increasing the traffic intensity | Increase the traffic from the scapy traffic generator. | Delay for the SOS packet is does not vary much with increase in traffic. | Irrespective of the traffic SOS packets are ensured high priority. |

**7.3 Demo Plan**

The demo will be provided using GENI. The network topology is shown in figure 4.1. The network will be flooded with a mix of SOS packets and non-emergency packets.

|  |  |  |  |
| --- | --- | --- | --- |
| Scenario-Id | Action | Expected Behaviour | Conclusion |
| S1 | Configure the network topology in Geni and connect to the network with host computers. Run the listener application on all hosts. Run the scapy applications to generate regular traffic on the network. Generate an emergency packet from one of the host computers. | Immediately observe that the Distress message has been received by all the host computers in the network. | Multicast flooding of packet is working |
| S2 | Increase the traffic load on the network using the scapy application to add multiple streams of traffic. Send a Distress signal on this loaded network. Repeat with varying loads on the network. | The Delay observed at the listener nodes will remain almost fixed. Delay is not proportional to the load | The network is prioritizing the Distress packets over the regular traffic. |

**7.3 Self Study Plan**

In a system that does not implement our kernel module, our SOS packets are neither identified nor prioritized. At best, they could be classified as priority traffic and queued in the priority class based queue. We consider the above as our base case. Our kernel module however analyses each packet coming into the router and marks it based on the content of the packet. Prioritization is achieved by creating a dedicated queue for SOS packets.

This processing on each packet's content will induce some delay in the overall transmission. We need to analyze if the delay introduced will counteract the effects of prioritizing the SOS packet i.e. does the disadvantage of added delay outweigh the advantage of SOS prioritization.

Our current design uses the previously mentioned kernel module to identify the SOS packet and mark it before the IP stack handles the packet using the TC component to prioritize these packets. It is possible however to use a different module that will let us insert SOS packets in the front of an outgoing interface's driver queue after the identification and forwarding decision phase. But this will require modifying pointers in the kernel space which will affect the overall stability of the system. We therefore consider this approach ambitious and consider it a further development case.

1. **References**
2. The Linux Networking Architecture - Design and implementation of Network Protocols in the Linux Kernel, Klaus Wehlre, Frank Pahlke, Hartmut Ritter, Daniel Muller, Marc Bechler
3. http://www.tldp.org/LDP/lkmpg/2.6/html [Linux kernel programming guide]
4. http://lxr.free-electrons.com/source/?v=3.13 [Source code for all kernel related changes across kernel versions]
5. http://www.linuxjournal.com/article/7184 [Example of using netfilter hooks to code modules]
6. http://www.netfilter.org [Home page of the netfilter project]
7. Quagga - http://packages.ubuntu.com/trusty/quagga [Ubuntu package page for quagga]
8. PIMD - http://packages.ubuntu.com/trusty/pimd [Ubuntu package page for PIMD]