FEBRUARY 23, 2016

SIMPLE ROUTING PROTOCOL

DESIGN DOCUMENT

DEBARUN DAS (DED59)

NANNAN WEN (NAW66)

SUPERVISOR: DR. TAIEB ZNATI

CS 2520 TERM PROJECT

1.Introduction

This document serves as the Design Document of the term project "Simple Routing Protocol" in CS 2520. The routing protocol required to be implemented is a *Simple Link State Routing Protocol (SLSRP).* This documents gives a brief overview of the different stages of implementation and the design choices and decision making strategies for efficient interplay between the different components of the system. A Link State Routing Protocol consists of an autonomous system of Routers which exchange routing information via a common routing protocol. Every router has a local copy of the entire topology and each router independently calculates its own best path. Further, the overall performance of the topology is tested by sending files from one *end system* to another that are connected to each other by a series of routers.

The various stages of SLSRP include configuration of each router using the Configuration file. Also, the user is allowed to issue commands and interact with the system. After a router is configured, it establishes adjacency relationship with neighboring routers. Then, the cost of link with each of its neighbors is calculated. Periodic messages are sent to check the status of its neighbors at regular intervals. This followed by the generation and flooding of *Link State Advertisements (LSA),* building the Link State database and ultimately using the *Link State Database (LSD)* to generate the routing table by using Dijkstra's Shortest Path Algorithm.

The entire document is divided into the following sections. Section 2 to Section 7 describes the design choices and strategies for implementing the different stages of the protocol. Section 8 describes the user interface, Section 9 describes implementation of the protocol using threads, Section 10 explains detection different errors and ways to solve them, Section 11 describes the connection type and format of the packets that are generated, Section 12 describes the requirements for implementing the protocol, Section 13 discusses the different testing and analysis techniques to measure the effectiveness of the implementation and finally, Section 14 lists the references.

2.Configuration Of The Router

A configurable file helps to remove the need for hard-coding in the implementation by placing them somewhere accessible by the implementation globally. A configurable item can be instilled in the system by three possible ways – by placing them in a *header file* or an *external configurable file*, or by taking *user input*. Since, for router configuration in the real world, taking *user input* for router is not a feasible choice for most parameters, so we include configuration using an external configurable file. However, for testing and experimentation, we also use user inputs for changing the router configuration. For simplicity, we plan to place the configurable items in an *external configurable file* and the *header file* will have a pointer to this file.

1. The router gets its own IP address (referred to as *Rip*) and an available Port number dynamically. One thing to be noted as that the IP address is not the loopback IP address of the machine. For design of the router topology in the real world, each interface of the Router is identified with a unique IP address. However, since this project demands implementation of a simpler version of the Link State Routing protocol, we associate

only a single IP address with a single router and indicate the different interfaces with different available port numbers in the machine. After this, router determines it Router Id and registers itself in a *Name Server*.

- **Router ID (*RId*):** This is ideally a 32-bit number that uniquely identifies a router in an Autonomous System. Ideally, in a router with many interfaces that are uniquely identified by an IP address, the smallest IP interface address is chosen as *RId*. However, for our implementation, since each router is associated with a single IP address, so we are left with two options – using a *unique integer* or using the *RIp(Router's IP address)*. The problem with using an integer as Router ID is that it will cost additional overheads to ensure that the *RId* is unique, as the router will need to flood *RId* to its neighbors so that it is not used by any of them. To avoid such unnecessary overheads, we plan to choose the IP address of the router as the *RId*as it will be unique for a single router.

- **Name Server:** The system bootstraps by initializing the Name Server first. It is used by the routers to *register* and *resolve*. The Name Server maintains a nested Hash Table with the *Router Id* as the key and the value is another hash table with the Port Number as the key and the functionality as the value. We do not store the IP address because a router's *RId* is same as its IP address.

2. The router is then configured using the external configuration file and the commands provided by the user at the User Interface. The list of commands available to the user are described in Section 8.The following are the list of configurable items to be provided in the *external configurable file*:

- ***RBlackListSend*:**This is the list of *RId* and port numbers of all the routers in the topology, which the current router cannot send *Be_Neighbors_Request*.

- ***RBlackListReceive:*** This is the list of *RId* and port numbers of all the routers in the topology, whose *Be_Neighbors_Request* is not accepted by the current router.

- ***SequenceNo***: This represents the sequence number of the first *LSA* (Link State Advertisement) packet that is flooded by the router after it initiates. Ideally, its value is zero.

- ***UInterval***: This represents the *Update Interval.* The amount of time determines the time after which *LSA* packets are flooded. 3 minutes.

- ***HInterval*:** This represents the amount of time the router waits before sending out *Alive* messages to its routers. This is generally given in seconds. The value of the *Hello Interval* should not be very high as we plan to send multiple *Alive* messages to verify if a neighbor has failed.It is ideally lesser than the *UInterval*.

- ***VersionNo:*** This parameter represents the Routing Protocol version number that is implemented by the router. The protocol version number that is followed by the router is exchanged with the other routers during the *Neighbor Acquisition* stage. In our implementation, if two routers have different protocol version numbers, then they will not have interface to a common network (i.e. they cannot be neighbors).

- ***InitialSequenceNo*****:** This parameter represents the sequence number of the first Link State Advertisement packet that is generated by the router. Details about choosing the Sequence number is explained in details in Section 6.1. Ideally, *InitialSequenceNo* is set to 1.

- ***MaxAge:*** This represents the maximum value of the LS Age field in seconds. Ideally, it is 3600 seconds (1 hour).

- ***LSRefreshTime:*** This is the time interval after which new LSAs are generated by a router with age zero and the next incremented Sequence Number. Ideally, it is 1800 seconds (30 minutes).

- ***InfTransDelay:*** The value by which the age of a LSA packet is incremented with every hop. Ideally, *InfTransDelay* is 1.

- ***MaxAgeDiff:***Ideally its value is 900 seconds (15 minutes). Its use is explained in Section 6.1.

- ***MaxLSAPacketSize:*** This is the maximum size of a LSA packet in bytes.

- ***Maximum Hop Count:*** If the *Hop Count* (see Section 10)field of a data packet reaches this value, then loop is detected.

- ***MaxCost:*** The maximum value to which the actual Link cost is transformed to (see Section 5). Ideally, 10 is chosen as the *MaxCost* value.

3.Neighbor Acquisition Protocol

After the configuration of a router with initial parameters in the external configuration file, the first step is discovery of neighboring routers.The fact that routers are neighbors is not sufficient to guarantee an exchange of Link State Advertisements (LSA), they must form adjacencies to exchange LSAs.A collection of packets (together referred to as the *Hello packets*sometimes) are sent out of each functioning router interface to discover and maintain relationship with neighbor routers. Before describing the types of messages that are transmitted for neighbor acquisition, we describe the design of a data structure that will store the list of neighbors and some of their related fields, for each router in the Autonomous system.

3.1. Neighbor_Table

A Neighbor_Table represents a data structure that is maintained by each router in the Autonomous System. This table will maintain the required details of the neighbors of the router. It is initialized with the list of all the Routers in the Name Server to, which are not present in list of Routers in *RBlackListSend*. It has the following fields:

- *NeighborRId*: This field stores the router ID of the neighbor. Since the Router ID is identical to the IP address of the router, this field essentially stores the IP address of the neighboring router.

- *InterfacePort:* This field stores the port number of the particular interface of the router that connects to the neighbor.

- *NeighborPort:* This field represents the port number of the Neighbor router which receives the neighbor messages. It is generally the port number of the router with type *Receive_Message*in the nested Hash Table maintained by the Name Server(see Section 2 and Section 9).

- *SendNFlag:* This field will store either 0 or 1. The value 1 indicates that *Be_Neighbors_Request*packet has been sent to this router. Otherwise, its value is 0.

- *WaitACKNFlag:* This field will store either 0 or 1. Initially, it stores 0. The value 1 indicates that the router is waiting for an acknowledgement after sending out the *Be_Neighbors_Request* packet.

- *ACKNFlag:* This field will store either 0 or 1. A value of 1 indicates that the router has received the *Be_Neighbors_Confirm* message as acknowledgement from the router it has sent *Be_Neighbors_Request* to.

- *SendCFlag:* This field holds either 0 or 1. It is set to 1 when the router sends a *Cease_Neighbors_Request* to the neighboring router.

- *ACKCFlag:* This field holds either 0 or 1. It is set to 1 when the router receives a *Cease_Neighbors_Confirm* message from the neighboring router to which it previously sent *Cease_Neighbors_Request* packet.

- *Cost:* This field contains the cost of the link to the neighbor in the network topology. A value of 0 indicates that the Link Cost is yet to calculated

3.2. Design Choice

We plan to implement our Neighbor_Table using a Hash Tablewith the combination of*NeighborRId* and *InterfacePort* as the key and the rest of the fields as value. One advantage of using a hash table is that unlike arrays (which suffers from overhead of shifting elements while deleting an element in the middle), it allows faster removal of a Hash Table entry, based on its key. Also, the retrieval of an element in hash table is ideally faster than in arrays and we need not use integer keys to elements (unlike arrays). Since we plan to implement using C++, so we will use a*map* container for implementing this.

After its configuration, the router can *send* the following types of neighbor acquisition messages:

- **Be_Neighbors_Request:** This message packet is sent to all the routers whose *SendNFlag* and *SendCFlag* are 0 in the *Neighbor_Table*. After successfully sending the *Be_Neighbors_Request* packet, the *SendNFlag* and the *WaitACKNFlag*corresponding to the recipient router in the*Neighbor_Table* is set to 1.

- **Cease_Neighbors_Request:** This message is sent to a router to which the sending router does not want to be neighbors with, anymore. However, this message can only be sent to routers whose *ACKNFlag*is set to 1 in the *Neighbor_Table*. The *SendCFlag* of the *Neighbor_Table* is set to 1.

- **Be_Neighbors_Confirm:** This message can be sent as an acknowledgement to the message *Be_Neighbors_Request*. There are two criteria which are checked before a router sends a *Be_Neighbors_Request*. The version number of the protocol that is run by the two routers should be identical and the router which sends the *Be_Neighbors_Request* should not be in the *RBlackListReceive* of the router which sends *Be_Neighbors_Confirm*. After a *Be_Neighbors_Confirm* message is received by a router as acknowledgement to *Be_Neighbors_Request*, the new neighbor is added to its *Neighbor_Table* with the ACKNFlag set to 1.

- **Be_Neighbors_Refuse:** This message can be sent as an acknowledgement to the router which sends the message *Be_Neighbors_Request*. Ideally, there are two instances in which this is done. First, if the router that requests to be neighbor is in the *RBlackListReceive* of the receiving router and second, when the protocol version of the two routers do not match. The requesting router, on receiving the *Be_Neighbors_Refuse* message resets the *SendNFlag* and *WaitACKNFlag* of the *Neighbor_Table* to zero.

- **Cease_Neighbors_Confirm:** This message is sent as an acknowledgement of *Cease_Neighbors_Request* that is received from a router. The router that sends the *Cease_Neighbors_Request*, on receiving the *Cease_Neighbors_Confirm* message, resets the value of *SendNFlag* and ACKNFlag of *Neighbor_Table* to zero. Also, the values of *SendCFlag* and *ACKCFlag* are set to 1.

NeighborRId	InterfacePort	NeighborPort	SendNFlag	WaitACKNFlag	ACKNFlag	SendCFlag	ACKCFlag	Cost
RId_2	1800	2000	1	0	1	0	0	10
RId_3	1801	3800	1	1	0	0	0	0
RId_4	1802	2500	0	0	0	0	0	0

Figure 1: Sample Neighbor_Table of A Router With Router ID RId_1

In Figure 1, we see a sample Neighbor_Table of RId_1 with combination of *NeighborRId* and *InterfacePort* as the key to the Hash Table. In the table, RId_2 is confirmed as an adjacent neighbor of RId_1, while we are still waiting for an acknowledgement to the *Be_Neighbors_Request* sent to RId_3. And, the *Be_Neighbors_Request* was not accepted by RId_4.

4.Alive Messages

Alive messages are periodic messages that are sent to all the routers with which neighboring relationship has been established. This is essentially the procedure for monitoring the network status that contributes to robustness.

4.1.Strategy

Alive messages will be sent out after a fixed amount of time called the Hello Interval (*HInterval* in the configuration file). We plan to send multiple *Alive* messages in a *session* before concluding about the state of a neighboring router. Each such *session* will initiate every time a timer for the Update Interval (*UInterval*) starts and Alive Messages will be sent to the neighbor routers after every *HInterval*for the entire duration of the *UInterval*, until it receives an acknowledgement (message of type *Alive_ACK*) to one of the *Alive* Messages that it sent out to a particular neighboring router. If no acknowledgement message is received from a neighbor router in a particular session, then the neighbor router is assumed to be *dead* and new LSA packets are generated to be flooded out at the end of the *UInterval*. Additionally, the *Neighbor_Table* is modified to reflect that the adjacency with the unreachable adjacent router is broken. Generally, entry for the unreachable adjacent router is erased from the *Neighbor_Table*. Also, the Hash Table that maintains the record of all the routers in the topology is also updated.

Sending multiple *Alive* messages before coming to a conclusion about the state of a neighboring router ensures that the network is not highly sensitive and more stable.

5.Link Cost Messages

Link Cost Messages are sent after the Neighbor Acquisition phase to determine the cost of a link to a neighboring router. The following strategy is used for measuring the cost of a link to a neighboring router. A series of messages of the type *LinkCost* are sent sequentially to every neighboring router from the*LinkCost* thread (see Section 9).Half of the total time for sending a *LinkCost* message and receiving an acknowledgement message *LinkCost_ACK*(or the Round Trip Time) is ideally the cost of the link to an adjacent router. However, we plan to send multiple Link Cost Messages (around 20) and find the mean of the Round Trip Times (RTTs)to determine the cost of a link.

A simple strategy to implement this is by using two timestamp variables named *start* and *end*placed before sending the message and after receiving an Acknowledgement respectively. These two variables will store time before sending and after receiving Acknowldegement using *gettimeofday()* function in C.

RTT = $\sum(end_i - start_i)$) for i = 1 to n (n= no. of *LinkCost* Messages sent with acknowledgement)

So, Link Cost = RTT/2.

For simplification, we apply a transformation function this Link cost obtained to transform it to a positive integer value. We plan to have cost values ranging from 1 to *MaxCost* depending on the range in which the actual Link Cost lies.

The final Link Cost that we get for a particular neighbor is stored in the *Cost* field of the *Neighbor_Table.*

6.Generation And Flooding Of Link State Advertisement

Each router in the Autonomous System originates one or more Link State Advertisement (*LSA*) packets. A LSA is to describe a router's local part of the routing topology domain. After a router has established its adjacencies and determined the cost of each of its link, a router can build its LSAs that contain the link-state information about its links.

6.1.Building LSA

A Link State Advertisement packet consist of the following fields:

- **Router ID:**This is a 32 bit field. This field contains the Router ID (*RId*) of the router that originates the LSA.

- **LSA Type:** This field indicates the format and function of the LSA. In Link State Routing protocols like OSPF Version 2, a LSA packet can be of different types (like router LSA, network LSA etc). However, in this project, we will deal with only one type of LSA which can just be indicated by *'LS_1'*.

- **LS Sequence Number:** This is ideally a 32-bit unsigned number that represents the sequence number of a LSA packet. We choose unsigned integer over a signed integer for

simplicity in testing the implementation.The sequence number of the first LSA packet that is flooded by the router that generates it, is equal to the value of the parameter *InitialSequenceNo* in the external configuration file. The sequence number is incremented each time a new instance of the LSA is generated.

An advantage of associating a Sequence Number to each LSA packet is that it increases the speed of convergence of the Autonomous System by rejecting duplicate LSA packets. This is ensured because whenever a router receives a LSA packet, it checks the sequence number of the received LSA packet and the sequence number of the LSA packet (of the same originating router) that it maintains in its Link State Database (*LSD*, which is explained in details in Section 6.3.2). The router will accept a new LSA only if its sequence number is greater than the sequence number of the LSA that is maintained in its local Link State Database.
However, there are several problems that can arise because associating a Sequence Number with a LSA packet. They are as follows:

a) Routers can get confused if the Sequence Number wraps around after reaching the maximum(which is $2^{32} – 1$).
b) If router crashes, it loses track of its latest sequence number. The LSA packets will be generated starting from *InitialSequenceNo*.
c) The sequence number is corrupted.

This gives rise to the use of another field called LS Age.

- **LS Age:** This field is a 16-bit unsigned integer which represents the age of an LSA packet in seconds. It must be incremented by the parameter *InfTransDelay* in the external configuration file on every hop during the *flooding* procedure. Ideally, the value of *InfTransDelay* is 1. The LSAs are also aged as they are held in a router's database. However, the age of an LSA cannot go beyond the value of the parameter *MaxAge* that is specified in the configuration file. So, adding the field *LS Age* in addition to *LS Sequence Number* helps in the following ways:

  a) If the present sequence number of a LSA is the maximum sequence number (which is 2^{32} -1) and a new instance of the LSA must be created, the router must first flush the old LSA from all databases. This is done by the originating router by setting the LS Age of the existing LSA to *MaxAgeprematurely*and re-flooding it over all adjacencies. As soon as all adjacent neighbors have acknowledged the prematurely aged LSA, the new instance of the LSA with a sequence number of *InitialSequenceNo* may be flooded by the originating router. This solves the problem when the Sequence Number wraps around after reaching the maximum.

  b) If a LSA reaches *MaxAge*, they are flushed from the entire domain. So, there must be a mechanism for preventing legitimate LSAs from reaching *MaxAge* and being flushed. This is achieved by using the mechanism called link-state refresh. After

every interval of *LSRefreshTime* (in configuration file), the router that originated the LSA floods a new copy of the LSA with an incremented sequence number and an age of zero. Upon receipt, the other routers replace the old copy of the LSA and begin aging the new copy.

c) If the age of the LSA that is received by a router and the age of the LSA copy that is maintained in the local LS Database of the router, differ by more than the interval *MaxAgeDiff*, the LSA with the lower age is more recent. This can occur when a router restarts and loses track of the LSA's previous LS sequence number.

- **Number of Links:** This represents the number of links that are reported in the LSA.

- **Link Id:** This field uniquely identifies the Link to an adjacent router. For convenience in identifying the link, we choose it uniquely as the combination of the port number of the interface of the originating router that originates the link and the *RId*of the adjacent router that is connected by that link. This combination is chosen over integers because it helps in uniquely identifying the link and can be directly used in building the Link State Database.

- **Link Data:** This field contains the cost of the link.

- **LS Checksum:**This field is the checksum of all the contents of the LSA, except the LS Age field, so that the LS Age field can be incremented without updating the checksum. So, the difference between the length of the LSA and the length of the LS Age field is the amount of data to checksum. It can never have the value of zero (which indicates a checksum failure).We implement this by using CRC (see Section 10).

6.2. Flooding LSAs

Flooding is part of the protocol that distributes LSA packets and synchronizes the Link State Database between all the routers in the Autonomous System. The LSA that is generated by a router is sent to all its adjacent neighbor router. In turn, each received LSA is copied and forwarded to every neighbor except to the one that sent the LSA. *Flooding* is a reliable process, that is, we use the protocol Automatic Repeat Request (ARQ). Thus, when a router receives LSAs from a neighboring router, it has to send an acknowledgement message (called *LSAACK*) to the router which sent the LSA.

For each LSA that is received by a router from any of its adjacent routers, the following steps are taken to determine if the LSA will be accepted or not:

a) The LSA’s checksum is validated. If it is invalid, then it is discarded. No Acknowledgement is sent to the sending router so that it resends the LSA packet with the correct checksum.

b) The LSA's type is validated. If it is invalid, then the LSA is discarded. In our implementation, a valid LSA type is 1.No acknowledgement is sent back is such a case so that retransmission occurs.

   If the above two checks are satisfied, then if the router's Link State Database does not contain any LSA of the same originating router, then the received LSA is stored in the Link State Database. However, if a copy of the same originating router's LSA is contained in the local Link State Database of the router, then the following checks are done to determine which is more recent of the two LSAs:

   i. The LSA's sequence number is compared with sequence number ofthe copy of the LSA in the router's local link state database. If the sequence number of the received LSA is lesser than that of the LSA already contained in the local Link State Database, then it is discarded and an acknowledgement is sent to the sender so that the LSA is not retransmitted again.

   ii. If the sequence numbers are equal, then their checksums are compared. The LSA with the highest unsigned checksum is more recent and stored in the Link State Database. Again, an acknowledgement is sent to the sender.

   iii. If the checksums are equal, then their LS Age is compared. If only one of the LSAs has an age of *MaxAge*, it is considered most recent and stored in the Link State Database.

   iv. If the age of the LSAs differ by more than *MaxAgeDiff*, then the LSA with lower LS Age is considered most recent and is stored in the Link State Database.

   v. If none of the above conditions are met, the two LSAs are considered identical and the received LSA is discarded. An acknowledgement is sent to the sending router so that the LSA is not retransmitted.

6.3.Data Structures To Be Used For Efficient Flooding

The LSA packets that are accepted are first stored in a linked list of LSA packets called *Incoming_List*. We choose a linked list over a queue because the number of elements that can be stored in a Linked List is not fixed, unlike a queue, which is fixed in size. There are two other data structures that are maintained for this purpose. They are described as follows.

6.3.1.LSA_Status Table

This is a table that will maintain the status of each LSA that is flooded out to the neighbor routers. We use a nested hash table to implement this. The following diagram shows the structure of the table.

LSA RId	Neighbor 1		Neighbor 2		Neighbor 3		Neighbor 4	
	Send	ACK	Send	ACK	Send	ACK	Send	ACK
RId_1	1	0	1	0	1	0	1	0
RId_3	1	1	1	1	1	1	1	1
RId_6	1	0	0	0	1	0	1	1
RId_2	1	1	0	0	1	1	0	1

Figure 2:Example Of A LSA_Status Table Maintained By A Router

*LSA*_Status Table stores the *RId* of router that originates a LSA that is flooded, as the key to a row. So, the key of the hash table *LSA_Status* is the *RId* of the LSA that is flooded. The value of *LSA_Status* is another internal hash table that has the *RId*of each of the neighbors as the key and the valueconsists of two fields (named *Send* and *ACK*) that store either 0 or 1. When a LSA is sent to an adjacent neighboring router, the value of the *Send* field is 1(otherwise its 0) and when it receives an acknowledgement message back from its neighbor, then the value of the *ACK* field is 1 (otherwise, it is 0). The nested hash table will be implemented by using nested map in C++.

If the router which sent the LSA is a neighbor of the receiving router, then the values of *Send* and *ACK* for that neighbor router is set to 1 after an acknowledgement is sent to the receiving router. This is done so that the LSA is not flooded to the sender.

Figure 2 shows an example of LSA_Status table, where the LSAs that are/is to be flooded are the LSAs originating from the routers RId_1, RId_3, RId_6 and RId_2. The LSAsthat are/to be flooded to the four neighbors that the router has have their fields *Send* and *ACK*set to values that reflect their respective states.

6.3.2.Design Of The Link State Database(LSD)

Every router will maintain the LSAs as a series of records. The LSD is designed as a Hash Table named *LSD* with the combination of the *RId* of the router that originates it and a Link ID as the key of the Hash Table. The values of the Hash Table *LSD* are all the fields of a LSA packet: LS Sequence Number, LS Type, LS Age, LS Checksum, Cost(which is contained in the Link Data field of a LSA) and Number Of Links.

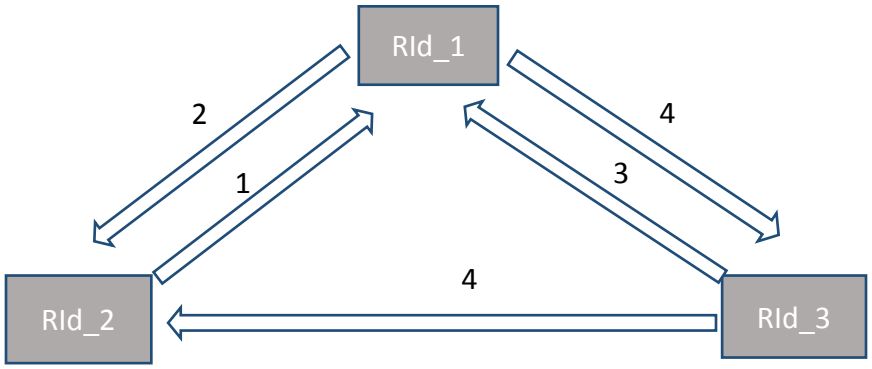

Figure 3.a: An Autonomous System Of Three Routers With The Value On Top Of The Edges Representing The Cost Of The Link

Router_ Id	Link_ ID	Cost	Sequence	Age	Type	Checksum	No. Of Links
RId_1	RId_2	2	1	3	LS_1	C1	2
RId_1	RId_3	4	1	3	LS_1	C1	2
RId_2	RId_1	1	5	2	LS_1	C2	1
RId_3	RId_2	4	2	1	LS_1	C3	2
RId_3	RId_1	3	2	1	LS_1	C3	2

Figure 3.b: Link State Database For The Topology In Figure 2a.

The above Figure 3.b shows a tentative LSD that is formed for the routing topology in Figure 3.a. In the LSD of Figure 3.b, the columns Link_ID and Router_ID represent the key of the Hash table that represents the LSD. For simplicity, the Link_Id in Figure 3.b is represented as the Router Id of the neighbor. However, in our implementation, the Link_Id will be similar to the one described in Section 6.1. A Hash table is a proper choice for designing this because we take a combination of both the Link_ID and Router_ID as the key, which results in faster access.

6.4.Strategy For Forwarding LSAs To Adjacent Routers

The LSA packets that are stored in the Linked List *Incoming_List* (see Section 6.3 .for details) are used to update the *LSA_Status* table (described in Section 6.3.1) and the *LSD* (see Section 6.3.2) before flooding to the neighbors other than the sending router. The LSAs in the *Incoming_List* are flooded at the end of every *UInterval*. When a self-generated LSA is to be

flooded, it is added to the *Incoming_List*. As Acknowledgements are received from each of the Neighbor Routers, the *LSA_Status* table is updated and the respective entry in the *Incoming_List* is removed. After every Update Interval (*UInterval*), the LSAs in the *Incoming_List* are flooded to the neighbors. This takes care of the fact that LSA packets that were not acknowledged are re-transmitted. We wait for the Update Interval to flood messages to avoid a 'Flood Storm'.

6.5.Sending and Receiving Link State Acknowledgements

The LSA Acknowledgement packets (referred to as *LSA_ACK*) contain the fields Router_Id (*RId* of the originating router), Type (which is set to *LSA_ACK*), LS Checksum. When a message of type *LSA_ACK* is received, the originating Router ID is checked. If it matches with any of the originating router IDs of a LSA in the *Incoming_List*, then it is removed from the list and updates are made in the *LSA_Status* Table. If the required LSA is not in the *Incoming_List*, then we simply discard the LSA_ACK packet.

7.Computing The Shortest Path And Generating The Routing Table

The Link State Database (LSD) is used to generate a two dimensional weighted adjacency matrix *WAdj[][]*, that represents the weighted digraph of the entire topology. This matrix *WAdj[][]* is a n*n matrix (where n is the number of routers in the entire topology). The rows represent the source *RId* and the column represents the Destination Router Id. The elements in *WAdj[][]* represents the cost of any direct link from the source *RId* to the destination *RId*. It is infinity (a very large constant integer for implementation) if there if there is no direct link and 0 if the source and destination *RIds* are the same.

Further, we give the weighted adjacency matrix *WAdj[][]* as input to Dijkstra's Algorithm to generate the Shortest Path Tree *SPTree*. The *SPTree* represents the shortest paths from the source node to all the other nodes. The *SPTree*is used to calculate the Routing Table.

7.1. Design Choice For Routing Table

The Routing Table will be implemented as a Hash Table with the key as the *Destination_RouterID* and the value as the fields *Next_Hop* and *Cost*. The *Next_Hop* field indicates the Router ID to which the next hop is to be made to reach the *Destination_RouterID* using the shortest path. A basic structure of the routing table is shown in Figure4b of the router RId_1 in the routing topology shown in Figure 4a. In the routing table of Figure 4b, the key is the column Destination_RId.

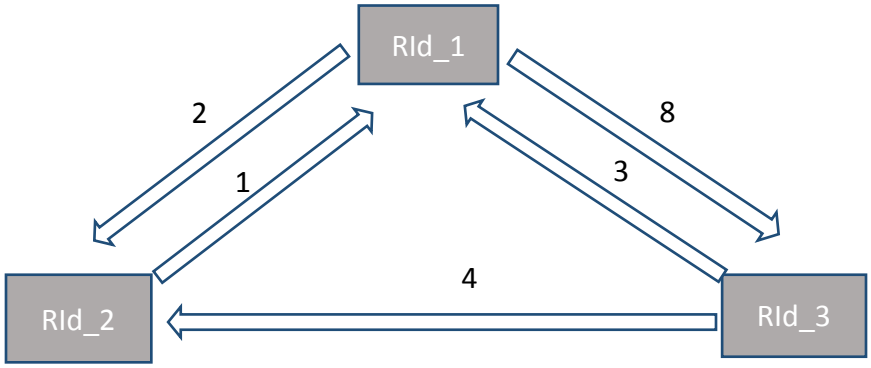

Figure 4a: An Autonomous System of three routers with respective link costs between them

Destination_RId	Next_Hop	Cost
RId_3	RId_2	6
RId_2	RId_2	2

Figure 4b: Routing table of Router RId_1 for the topology in Figure 4a

8.User Interface

The user interface is exposed to the end users which allows them to interact with the underlying system. We plan to implement this as a simple Command Line Interface.The user can issue several commands, as described below:

8.1.User Interface Commands For Routers

- *HELP*: This allows the user to view all the commands that can be issued from the user interface.

- *NEIGHBORS:* This commands allows the user to view all the adjacent routers of the current router.

- *UPDATEINTV <value>:* This command allows user to specify the Update Interval of the router in seconds. This value will override the default *UInterval* value in the configuration file.

- *HELLOINTV <value>:* This command allows user to specify the Hello Interval in seconds. Overrides the *HInterval* value of the configuration file.

- *ROUTING_TABLE:* This command allows the user to view the routing table that is maintained by the router at that instant.

- *SHORTEST_PATH <destination>:* This command allows the user to view the Router Ids (starting from the current Router Id) that contributes to the shortest path to the destination Router Id.

- *MAXAGE:* This command allows the user to set the Max LS Age of the self-generated LSAs.

- *MAXCOST:* This allows the user to specify the Maximum cost value to which the actual link cost will be transformed to. It will override the *MaxCost* field in the configuration file.

- *LSREFRESH:* This command allows the user to set the LS Refresh Time that will override the *LSRefreshTime* in the configuration file.

- *QUIT:* This command is used to terminate all the connections of the router and bring the router down.

- *KILL <interface>:* This allows the user to terminate a particular interface of the router, where *<interface>* is the port number that identifies an interface.

8.2. User Interface Commands For End Systems:

- *SEND <filename>:* This command allows the user to upload a file to send a file named *<filename>*.

- *TIMERECEIVE:* This command allows the user to view the total time it takes for sending the file and receiving an acknowledgement to it.

9.Implementing The Protocol Using Threads

Every router that implements the routing protocol goes through the following basic implementation steps:

Configuration Thread

This is the main thread that configures the router initially. After router configuration, new child threads will be generated for each of the different stages of the protocol. Each thread will use a new socket connection with a new port number. All the data structures – *Neighbor_Table, Inccoming_List, LSA_Status table, LSD, Shortest Path Tree* and the *Routing Table* are maintained in this thread. Thefollowing Threads are generated:

i. Neighbor Acquisition Thread(s).
ii. Alive Thread
iii. Link Cost Thread
iv. LSA Generation Thread
v. LSA Flooding Thread
vi. Routing Table Generation Thread
vii. Receive Thread

Neighbor Acquisition Threads

There is one Neighbor Acquisition thread for each adjacent link to a neighbor. Initially, a *Neighbor Acquisition Thread* registers itself in the Name Server with functionality as *Neighbor_Acquisition*. Then, it establishes neighborhood relationships with another router. For this, it *resolves* the available neighbor router IP and port in the Name Server and sends *Be_Neighbors_Request* to a router that is not in its *RBlackListSend*. And updates the *Neighbor_Table*. Depending on the user command at the User Interface, it may perform other tasks like sending a *Cease_Neighbors_Request*.

Alive Thread

The *Alive Thread* registers itself in the Name Server with its port number and functionality as *Alive*. Then it *waits* till there is a valid neighbor entry in the *Neighbor_Table*. Once it finds a valid neighbor in the *Neighbor_Table*, it sends *Alive* messages using the strategy described in Section 4.1.

Link Cost Thread

This thread registers itself in the Name Server with functionality as *LinkCost*. It remains idle until it finds a valid neighbor entry in the *Neighbor_Table*. It determines the Link Cost to all the

adjacent neighbors and continues executing till all the valid neighbors in the Neighbor_Table have a valid *Cost* entry.

LSA Generation Thread

This thread registers in the Name Server with the functionality *LSAGen*. It remains idle initially until there is a valid neighbor entry in the *Neighbor_Table.* However, it will also generate new LSA on other events like when an adjacent router fails or restarts etc. The events in which new LSAs will generated are mentioned in details in Section 6. After a LSA is generated, it is stored in the Linked List *Incoming_List*, described in Section 6.3.

LSA Flooding Thread

This thread registers in Name Server with functionality *LSAFlood*. It remains idle until there is an entry in the *Incoming_List.* It is also responsible for updating the *LSD* and the*LSA_Status Table*. The LSAs are flooded after every Update Interval as described in Sections 6.2, and 6.4.

Routing Table Generation Thread

This thread registers itself in the Name Server with functionality *Routing_Table*. It remains idle until there is an entry in the LSD. It generates the routing table after every Update Interval.

Receive Thread

This thread is responsible for receiving different types of messages. It registers itself in the Name Server using its port number and functionality *Receive_Message*. The different messages that it can receive include the *Be_Neighbors_Request*, *Cease_Neighbors_Request*, *Alive* message, *LinkCost* message and a *LSA* message.

Since the Receive Thread has to handle multiple types of messages, so we can instill load-distribution by using multiple Receive threads.

Figure 5 below shows the relationship between a router and the Name Server and the relation between the user interface, the configuration file with the threads of the router.

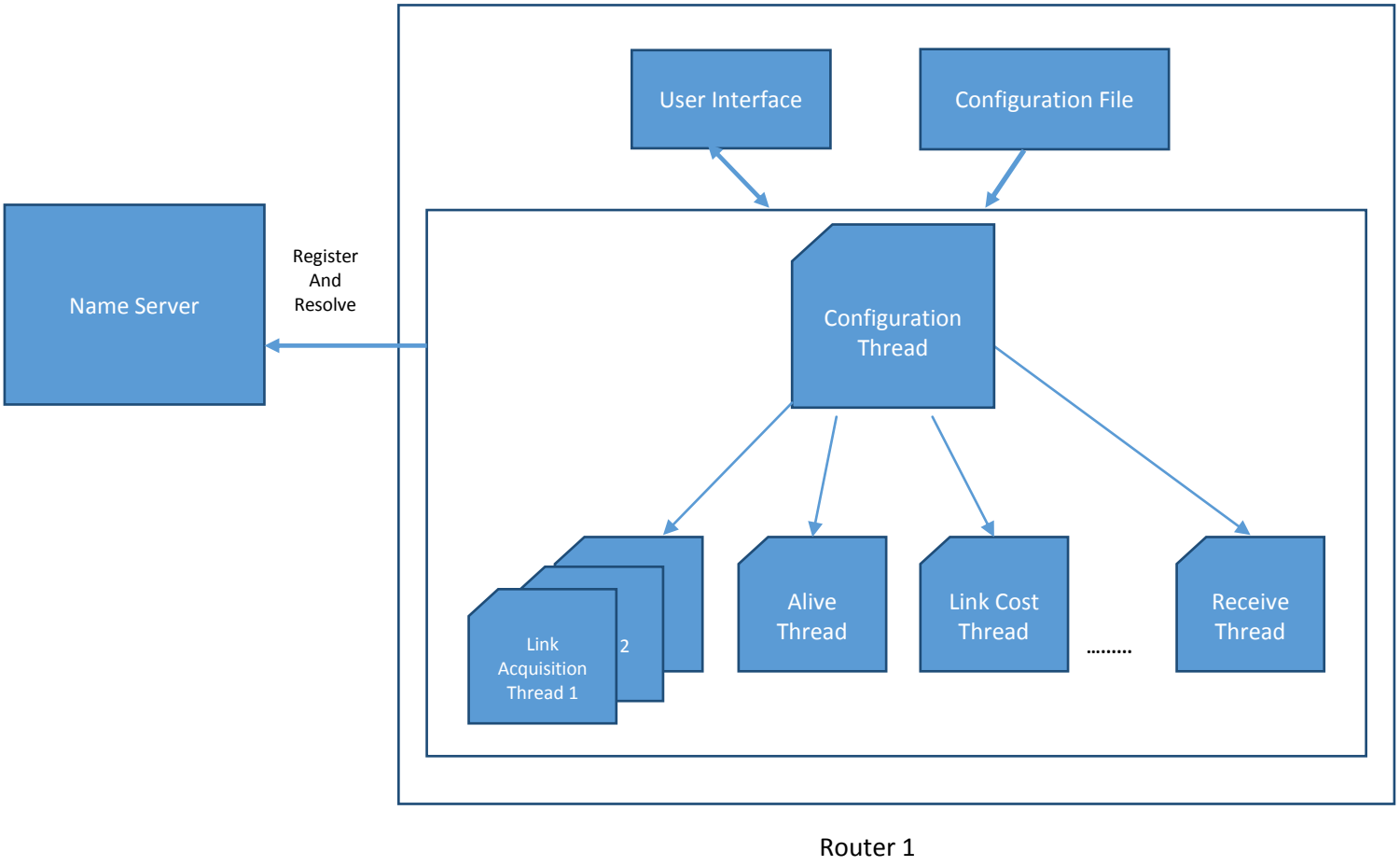

Figure 5: A Configuration Of A Single Router And Registration Of The Threads In The Name Server

10.Error Detection And Solution

- **Packet Error Detection:** As mentioned in the project description, errors are introduced in the packet by altering at least one bit with probability P_e (that is generated randomly). So, for detecting this error, we use *CRC*.
  Prior to transmitting a packet to R2, a router R1 computes and appends the *CRC* to the packet, forming a codeword. When the receiver R2 receives the packet, R2 performs a *CRC* on the whole codeword and compares the resulting check value with an expected residue constant, if the check values do not match, then the packet contains a data error, then R2 should request the retransmission of the packet.

- **Network Congestion Detection:** A router drops a LSA packet with a probability P_c before transmitting it. This results in inconsistencies in the database and possibly a loop.

  Handling loops And Database Inconsistency: Generally, in Link State Routing protocol, a routing loop disappears as soon as the new LSA is flooded to all the routers in the neighborhood and since the convergence rate is generally fast in Link State Routing, so this happens very quickly. However, we can still associate a field called the *Hop Count* in the Data packets that are transmitted by the End Systems. When the *Hop Count* reaches a *Maximum Hop Count* value, then a loop is detected and new LSA are generated and flooded by all the routers with involved in the loop. With flooding of the new LSAs the inconsistent Link State Databases of the routers are modified, which will remove the loop.

- **When A Router Fails And Restarts Again:** This has the potential to create Link State Database inconsistencies. The method to solve this is mentioned in Section 6.1 under LS Age.

- **Adding A New Router/ Removing A Router:** When a new router is added, then the LSDs of all the other routers will be updated as soon as this new router's LSAs are flooded. On removing a router, the failed router is detected by using *Alive* messages and the LSDs are updated as soon as new LSAs reflecting this topology change are generated and flooded.

11.Connections And Packet Format:

We use TCP because reliability is a major issue in Link State Routing Protocol. Moreover, most of the connections like the connections for Neighbor Acquisition, measuring Link Cost, Flooding LSAs etc are short lived, so resources are kept dedicated to ensure reliability. The packet format of different messages are shown below.

Source	Destination	RouterId	PacketType	Data	Checksum

Figure 5: A general packet format for Packet Types Be_Neighbors Request, Cease_Neighbors Request, Alive Message and Link Cost Message

In Figure 5, the Data field will generally contains the port number of the Router Interface and the current IP address of the sending Router.

Source	Destination	RouterId	Type	Checksum

Figure 6: A general packet format of the Acknowledgement Messages

In Figure 6, the Type field will hold *Be_Neighbors_Confirm, Be_Neighbors_Refuse, Cease_Neighbors_Confirm, LSA_ACK, LinkCost_ACK, Alive_ACK*. There is no Data field and the Router ID field is the Router Id of the sending router.

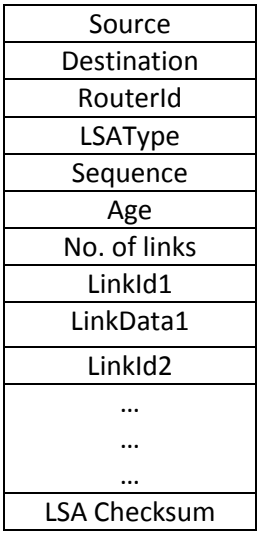

Figure 7: LSA packet format.

Figure 7 represents the LSA packet format and its details are described in Section 6.1.

12.Requirements For Implementation Of The Protocol

This section illustrates the requirement for the implementation of the protocol.

Platform Of Implementation: Unix or Linux platform.

Language for Implementation: We are going to implement the project using C++.

Tools:

Documentation Tool:

MS Word – to create documents for design proposal, final report and User Manual.

Statistical Analysis Tool:

MATLAB and MS EXCEL – to analyze statistical results.

Build Tool:

Make – to build the software bundle automatically.

13.Testing And Analysis Of The Proposed Implementation And Protocol

13.1. Test Strategy

We will test our components individually and independently. This is followed by integration testing. For integration testing purpose, tests will be performed on a simple file transfer application to demonstrate how data packets are routed correctly between different routers. Files to be transferred between two end systems connected by a series of routers are classified based on different test scenarios.

Dijkstra's SPF Algorithm Correctness Testing

1. Create a known topology, calculate the shortest path between routers manually.

2. Using user interface to compare the shortest paths generated by Dijkstra's Algorithm.

Neighbor Acquisition Correctness Testing

1. Set different version of the Link State Routing Protocol between router R1 and router R2 in the configuration file, and see if R1 will send *Be_Neighbors_Refuse* message or not after receiving *Be_Neighbors_Request* message from R2.

2. Set router R1 in the *RBlackListReceive*ofrouter R2'sconfiguration file, and see if R2 will send *Be_Neighbors_Refuse* message after receiving *Be_Neighbors_Request* message from R1.

3. Set router R1 and R2 on each other's listof acceptable routers, and with the same version of Link State Routing Protocol in the configuration file.Then, check if R1 sends Be_Neighbors_Confirm message after receiving Be_Neighbors_Request message from R2.

Alive Messages Correctness Testing

1. If routers R1 and R2 are neighbors, make R1 down, after a period of time (larger than Update_Interval). Check to see if R1 is deleted in R2's LSD.

2. Restart router R1 up after a period of time.Check to see if R1 reappears in R2's LSD.

Packet Error Detection Testing on the Router

Make R1 transmit a LSA packet with errors.Check the response of receiver R2.

Network Congestion Testing on the Router

Make R1 transmit a LSA packet, then drop the packet, check the response of receiver R2.

13.2.Statistical Analysis

Performance with different error rates

Make a set of folders, each containing files of different sizes, using different packet error rate Pe, and different Pc as the probability of dropping the pocket, run the test for each folder. For every response, plot the Round Trip Time and convergence time.

Considering processing delay and transmission delay for ACK to be very small, we can say that Round Trip Time = 2 * D_{proc} + D_{trans} where D_{proc} and D_{trans} are the processing and transmission delay respectively.

Plot the graph of Round Trip Time against file sizes for each of the P_e using Matlab.

Plot the graph of Round Trip Time against file sizes for each of the P_c using Matlab.

Confidence Interval

Make a set of topologies, each containing different number of routers, collect the convergence time for each topology, plot the convergence time for different topology. And then change a certain number of routers' status from up to down. Then, calculate the number of router that goes down and the corresponding convergence time. We then plot the convergence time against the number of fails per time unit of the router.

We calculate the confidence interval for convergence time and check for the optimal topology using the plot.

14. Bibliography:

1. https://supportforums.cisco.com/discussion/11060926/ospf-lsa-aged-maxage

2. https://www.ietf.org/rfc/rfc2328.txt

3. http://people.cs.pitt.edu/~znati/wans.html

4. http://www.ciscopress.com/articles/article.asp?p=24090&seqNum=4

5. https://en.wikipedia.org/wiki/Checksum

6. https://en.wikipedia.org/wiki/Convergence_(routing)

7. https://en.wikipedia.org/wiki/Cyclic_redundancy_check

8. Data Communications And Networking (4th Edition) — Behrouz A. Forouzan