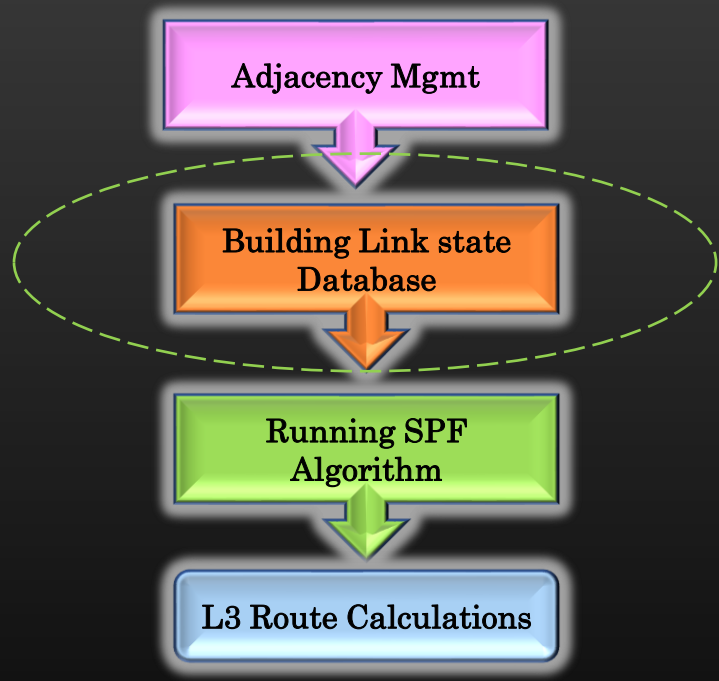
Agenda

- We will be going to implement a simplified Routing Protocol in this course
- Routing protocol chosen – Interior gateway protocol (IGP , ex OSPF, ISIS)
  - Don't know about it – don't worry, we shall cover theory first before any implementation
- A typical IGP (link state) protocol functionality is divided into 4 distinct parts :

1. Adjacency Management (Each device know its neighbours)
   - Sending and Receiving hello packets periodically
   - Update neighborship state machine
2. Building Link State Database (Each device internally creates a view of topology - Graph)
   - Building Link State packets
   - Flooding link state packets
   - Build a Graph – a view of network topology
3. Running SPF algorithm (Dijkstra) on LSDB
   - Process the LSDB through the algorithm
   - Compute Results and store
   - Algorithmically challenging
4. L3 Route Calculations
   - Use Results of 3 to compute final L3 routes and update Routing Table
   - Algorithmically challenging



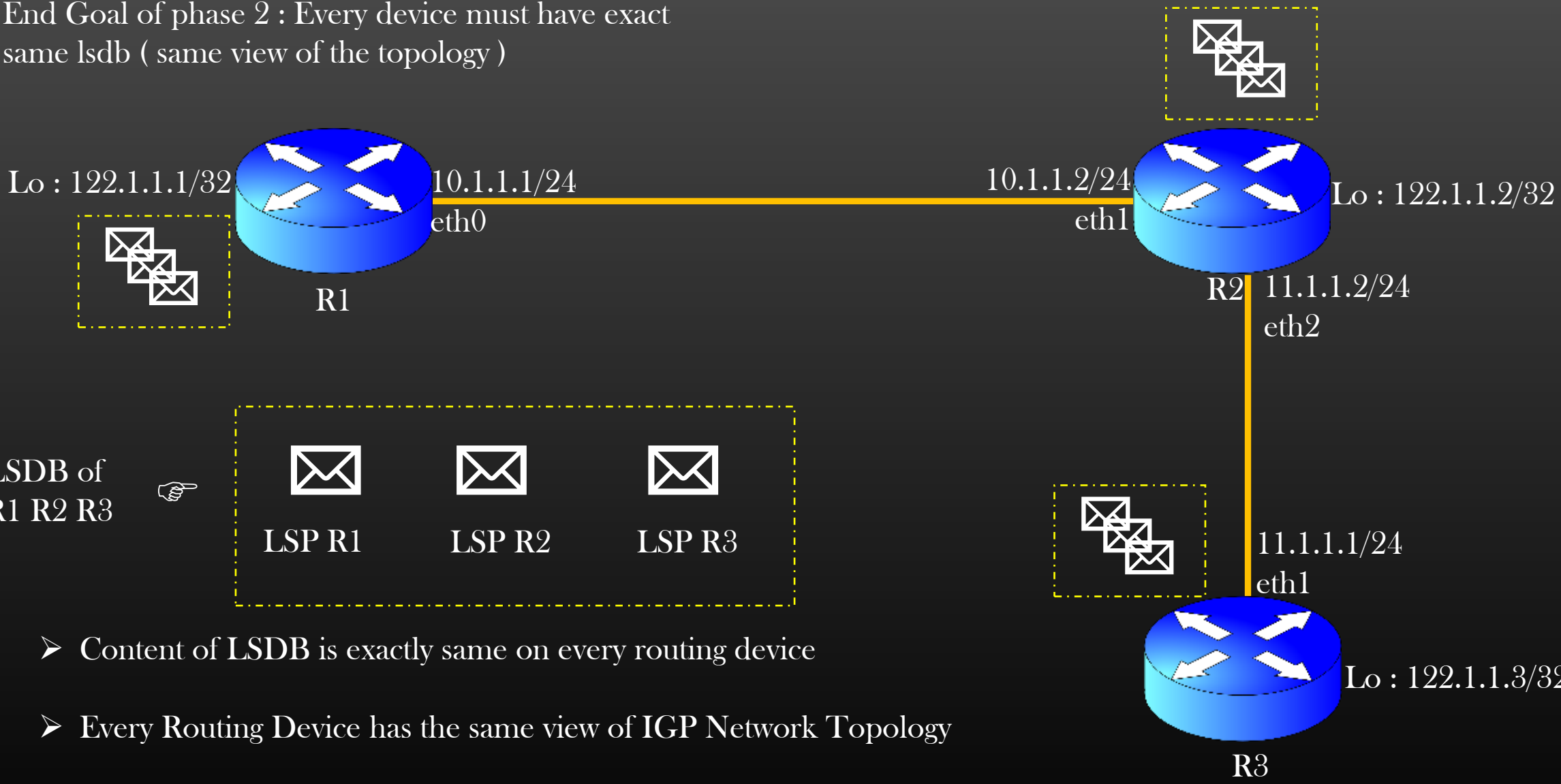
We shall be going to implement all 4 parts in this course series

Along the journey we shall implement various sub-features within the protocol

Network Protocol Development → Phase 2 : LSDB Mgmt

- Congratulations for successfully completing the Adj Mgmt (Phase 1) of the project
- You are marching toward being a pro programmer ! ☺
- In this Phase, we shall be going to implement Link-State Database Mgmt
- Device running IGP protocols Generates periodic link state packets
- Link state packets contains :
  - Information about Originator Device
    - Rtr ID, Host Name
    - Nbr information (Adjacencies in UP state)
    - Other attributes
  - ISIS LSPs are also TLV-ised packets
  - Every device share/flood its own LSPs throughout topology
    - Every device recv LSP of every other device in the topology
    - Result : Every device has a collection of LSP of every other device this is called Link-state-database (lsdb)
    - LSDB represents a graph in which:
      - Nodes represent the IGP devices
      - Edges represent UP adjacencies
    - This Graph represent the topology altogether
    - End Goal of phase 2 : Every device must have exact same lsdb (same view of the topology)

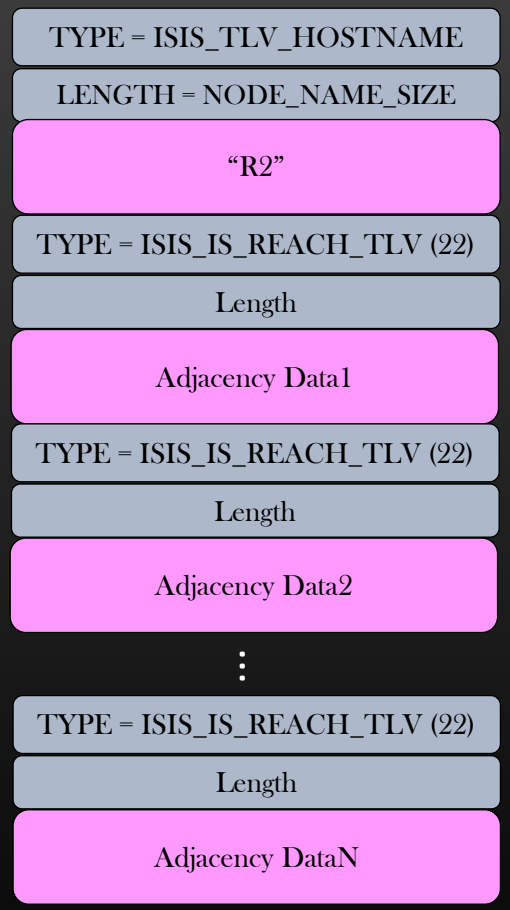
- End Goal of phase 2 : Every device must have exact same lsdb (same view of the topology)

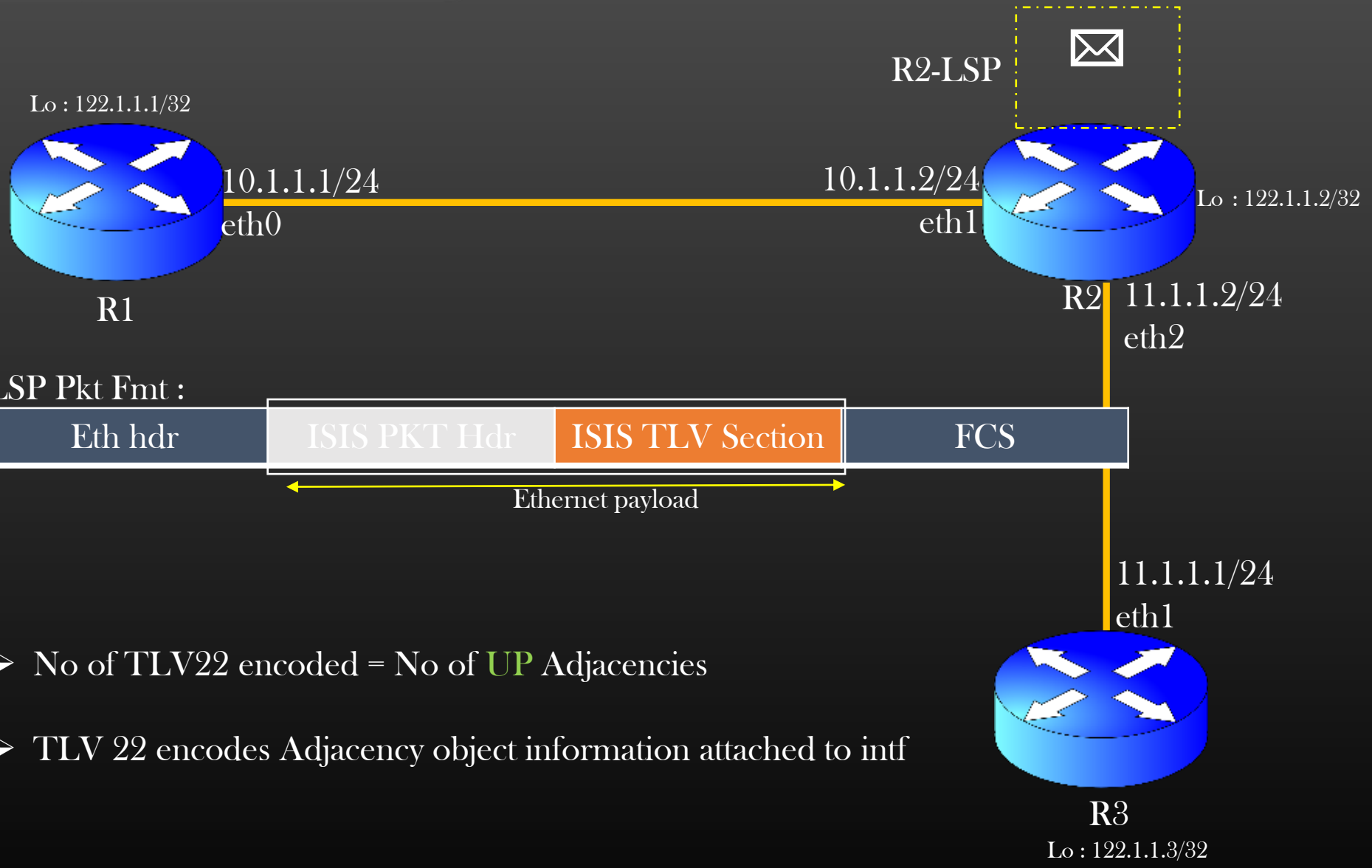


- Content of LSDB is exactly same on every routing device
- Every Routing Device has the same view of IGP Network Topology
- Every Routing Device then Runs several Algorithms on Local LSDB to compute results such as its local Routing Table

Topics :

1. LSP format
2. Generation of LSPs
3. Periodic Flooding of LSPs
4. Generation of LSPs on several Events
5. Processing LSPs
6. Populating LSDB
7. Reconciliation (LSP DB Fast Synch)
8. Mini Project – Interface Groups

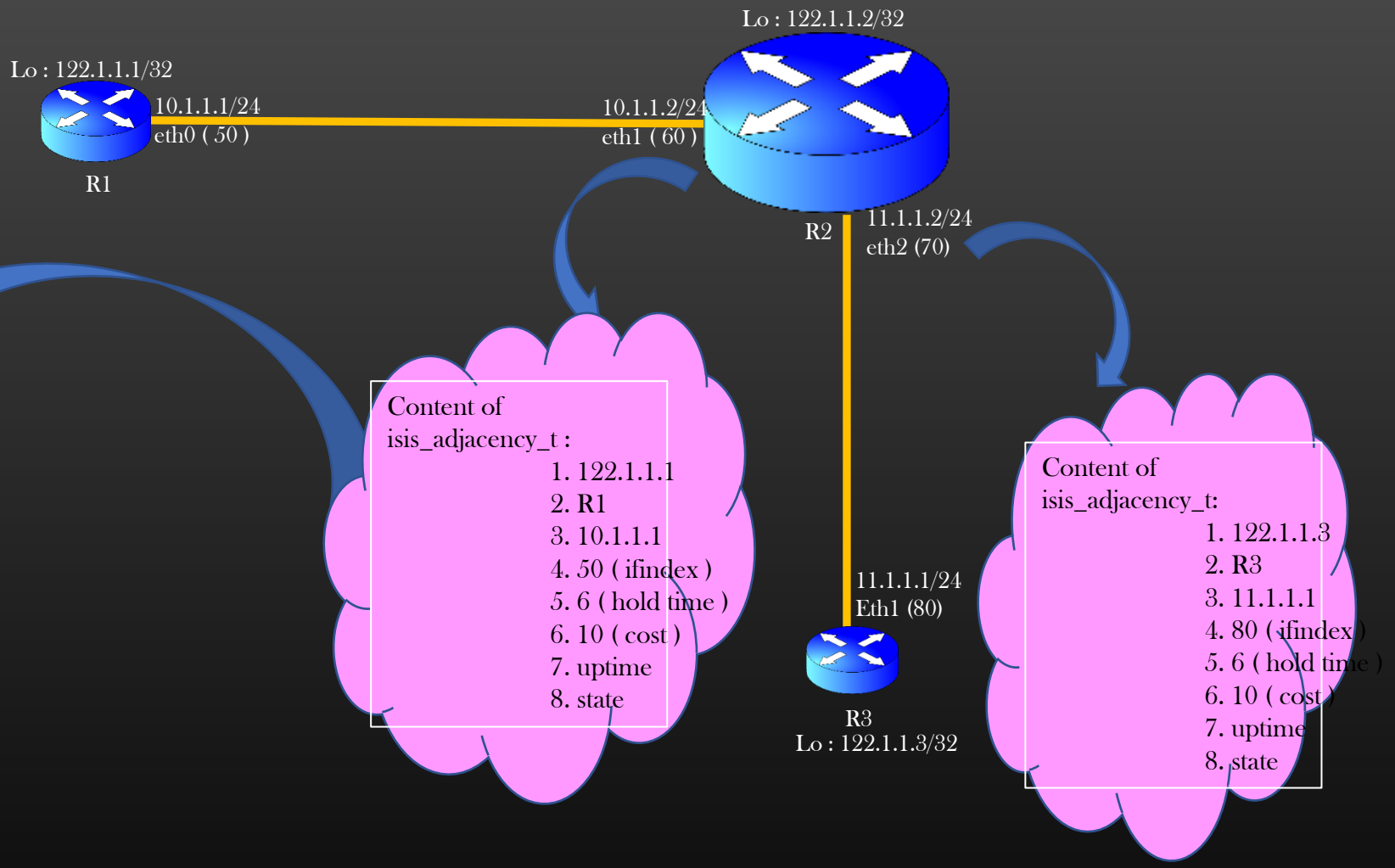




- No of TLV22 encoded = No of UP Adjacencies
- TLV 22 encodes Adjacency object information attached to intf

TYPE = ISIS_TLV_HOSTNAME	1
LENGTH = NODE_NAME_SIZE	1
"R2"	16
TYPE = ISIS_IS_REACH_TLV (22)	1
Length = 31	1
122.1.1.1 (Nbr lo addr)	4
10 (metric)	4
Sub TLV Length = 22	1
SUB TYPE = ISIS_TLV_IF_INDEX (4)	1
Length = 8	1
60 (lcl if index)	4
50 (remote if index)	4
SUB TYPE = ISIS_TLV_LOCAL_IP (6)	1
Length = 4	1
10.1.1.2	4
SUB TYPE = ISIS_TLV_REMOTE_IP (8)	1
Length = 4	1
10.1.1.1	4

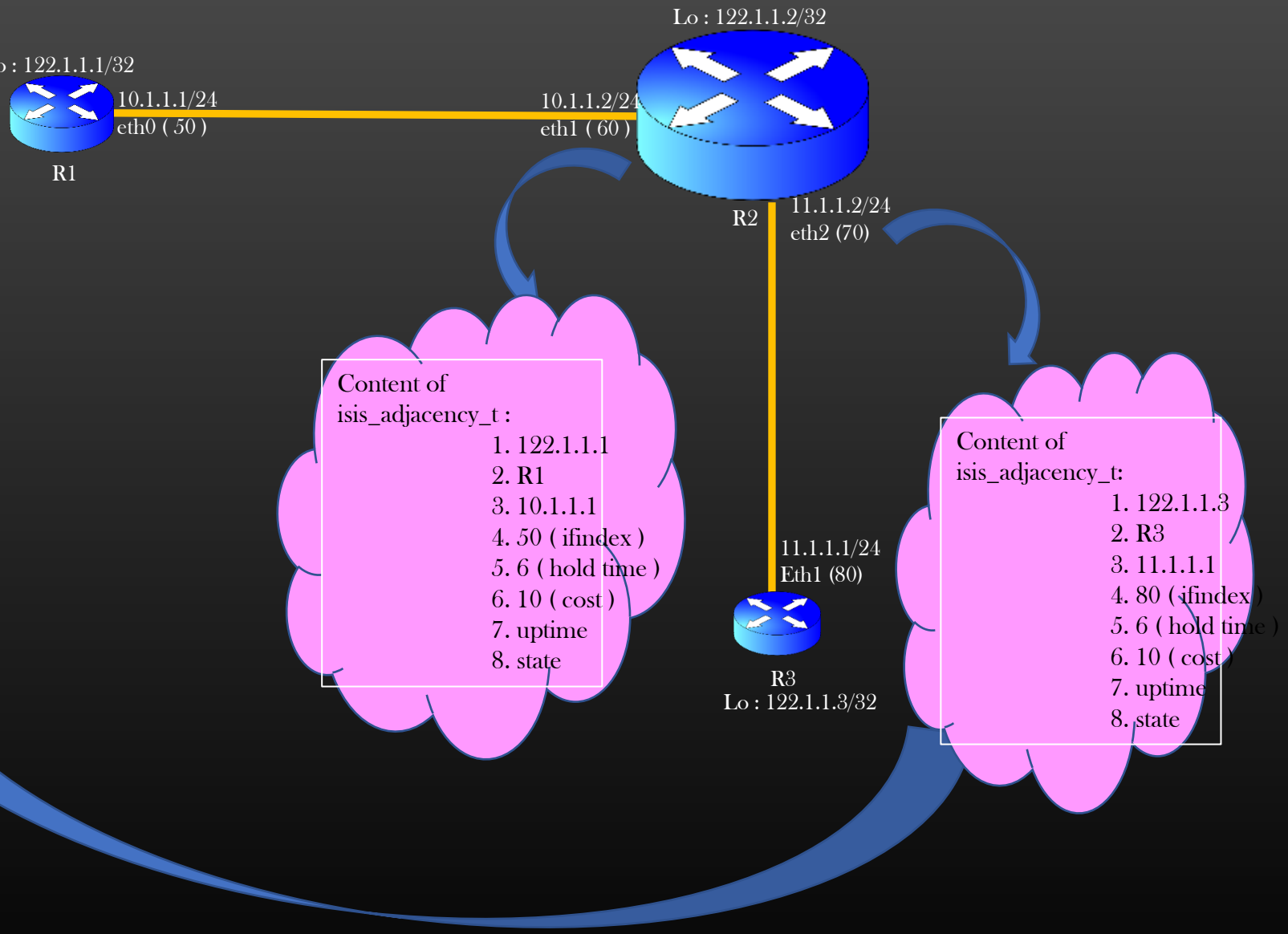
... Next TLV 22



TLV 22 = Nbr lo Addr + Metric + SubTLV 4 + SubTLV 6 + SubTLV 8

... Next TLV 22

TYPE = ISIS_IS_REACH_TLV (22)	1
Length = 31	1
122.1.1.3 (Nbr lo addr)	4
10 (metric)	4
Sub TLV Length =22	1
SUB TYPE = ISIS_TLV_IF_INDEX (4)	1
Length = 8	1
70 (lcl if index)	4
80 (remote if index)	4
SUB TYPE = ISIS_TLV_LOCAL_IP (6)	1
Length = 4	1
11.1.1.2	4
SUB TYPE = ISIS_TLV_REMOTE_IP (8)	1
Length = 4	1
11.1.1.1	4



TLV 22 = Nbr lo Addr + Metric + SubTLV 4 + SubTLV 6 + SubTLV 8

Given as Assignment

- TLV 22 is encoded from a given isis_adjacency_t object
- Write below APIs in isis_adjacency.c/.h

```
uint8_t      /* Returns the total length of TLV 22 in bytes */
isis_nbr_tlv_encode_size(
                        isis_adjacency_t *adjacency,
              uint8_t *subtlv_len);  /* Total length of all SUBTLVs with in a TLV 22 */
```

```
byte *       /* Returns end of the buffer ptr after encoding/inserting a TLV 22 */
isis_encode_nbr_tlv(
                isis_adjacency_t *adjacency,
    byte *buff,                          /* Output buffer ptr to encode tlv in */
    uint16_t *tlv_len);    /* length encoded ( tlv overhead + data len)*/
```

Content of isis_adjacency_t :
1. 122.1.1.1
2. R1
3. 10.1.1.1
4. 50 (ifindex)
5. 6 (hold time)
6. 10 (cost)
7. uptime
8. state

TYPE = ISIS_TLV_HOSTNAME	1
LENGTH = NODE_NAME_SIZE	1
"R2"	16
TYPE = ISIS_IS_REACH_TLV (22)	1
Length = 31	1
122.1.1.1 (Nbr lo addr)	4
10 (metric)	4
Sub TLV Length = 22	1
SUB TYPE = ISIS_TLV_IF_INDEX (4)	1
Length = 8	1
60 (lcl if index)	4
50 (remote if index)	4
SUB TYPE = ISIS_TLV_LOCAL_IP (6)	1
Length = 4	1
10.1.1.2	4
SUB TYPE = ISIS_TLV_REMOTE_IP (8)	1
Length = 4	1
10.1.1.1	4

... Next TLV 22

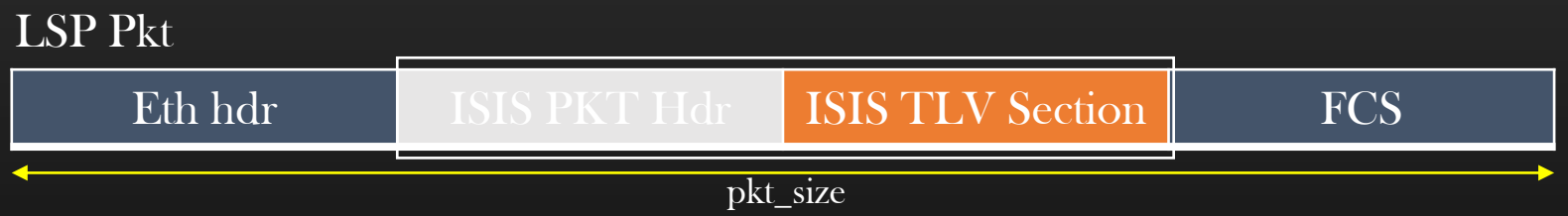
TLV 22 = Nbr lo Addr + Metric + SubTLV 4 + SubTLV 6 + SubTLV 8

- ISIS protocol would have to manage LSP Pkts (its own, as well as remote)
  - Update self-LSP contents
  - Periodic flooding
  - Install/un-install in LSDB
  - LSDB synchronization

- Therefore, It is better to have a proper structure to govern the management of LSPs

isis_pkt.h

```
typedef struct isis_pkt_ {

    /* The wired form of pkt */
    byte *pkt;
    /* pkt size, including ethernet hdr */
    size_t pkt_size;
} isis_lsp_pkt_t;
```



- Whenever the device initializes the ISIS protocol , it initialize a uint32_t sequence no to 0

```
typedef struct node_info_ {
        ...
        uint32_t seq_no;
        ...
} isis_node_info_t;
```

- When device generates a new LSP pkt from scratch, this sequence number is incremented first and then fed into lsp pkt hdr

```
node_info->seq_no++;
lsp_pkt_hdr->seq_no = node_info->seq_no;
```

```
typedef struct isis_pkt_hdr_{

    isis_pkt_type_t isis_pkt_type;
    uint32_t seq_no; /* meaningful only for LSPs */
    uint32_t rtr_id;
    isis_pkt_hdr_flags_t flags;
} isis_pkt_hdr_t;
```

- This Sequence number shall be used to control the flooding of LSPs in the topology, we shall revisit
- Higher the sequence no, younger is said to be the LSP (most recent or latest)

Topics :

1. LSP format
2. Generation of LSPs
3. Periodic Flooding of LSPs
4. Generation of LSPs on several Events
5. Processing LSPs
6. Populating LSDB
7. Reconciliation (LSP DB Fast Synch)
8. Mini Project – Interface Groups

- ISIS Protocol, after creating self lsp pkt, will cache its own LSP packet so that :
  - It has a quick access to its own LSP packet
  - Re-transmit LSP packet as and when required

```
typedef struct node_info_ {

    . . .
    /* pointer to self LSP pkt */
    isis_lsp_pkt_t *self_lsp_pkt;
    . . .

} isis_node_info_t;
```

isis_pkt.c/.h

```
void
isis_create_fresh_lsp_pkt (node_t *node);
```

- Create a new LSP pkt
- Discard the old one
- Cache the new one

- TLV Section of ISIS Lsp pkt must contain ISIS_TLV_HOSTNAME, and all TLV22s
- Protocol must trigger new fresh LSP pkt generation as a result of admin config change, Adjacency state change, in request of other devices (on demand), periodically etc.

- It make sense to have some show commands in our bag asap, which could display the LSP contents
- It would help in catching early bugs and proceed further

cli : show node <node-name> protocol isis lsdb

For now, just call the fn to generate new lsp pkt and just print the self – lsp pkt contents


```
uint32_t
isis_show_one_lsp_pkt_detail (byte *buff, isis_pkt_hdr_t *lsp_pkt_hdr, size_t pkt_size);
```

call to : isis_create_fresh_lsp_pkt ( )

if buff is NULL, print it on console, else print it in buff using sprintf

return number of output bytes

Use the fn isis_show_one_lsp_pkt_detail ( ) to trace lsp packets by tracing infra

Demo !

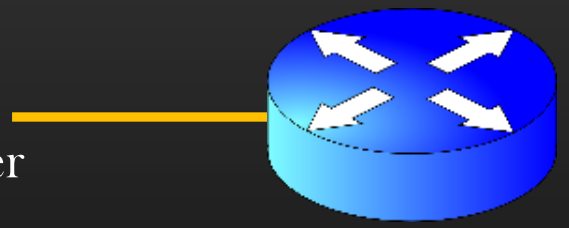
Introducing Asynchronous Programming

- For further development of our project, we shall switch to Asynchronous Programming
- We would be able to reap benefits of Asynchronous programming in our development
- Development made easy through Asynchronous Programming
- We shall realize the benefits of async programming as we progress more into the course
  (telling all benefits right here would only confuse you)
- First, let us get ourself introduced to the world of async programming and understand a subset of problems it solves for us, and understand why *sync programming alone* is a no go in big software projects
- We shall be going to use external library called Dispatch Queue which will allow us to program our protocol asynchronously. Technically it is also as called as Event Loop
- Asynchronous programming is a broad term, Dispatch Queue is just one programming techniques to realize a part of Asynchronous Programming
- Async Programming is also realized using Multithreading, Coroutines, etc

- Before we proceed further, first we need to understand the terms clearly and understand the difference between them
- Pls refer to Appendix Section B to understand basics of A(Synchronous) Programming Model
- Appendix B shall also explain a pit fall we would fall into in our project if we don't switch to async programming asap
- Proceed to next lecture video only after covering the appendix Section
- A Part of Async Programming which we shall going to use in our project is Work Deferral, as I explains in the next lecture video

- Let us discuss straightaway the problem we shall face if we do not have Async Programming (esp work deferral) in place

- Consider a device running our implementation of ISIS protocol. Let us say, device running 3 LSPs from remote nodes of the topology
  - Update Link State Database
  - Run SPF Algo
  - Calculate all Routes
  - Update Routing Table



- So, if you notice, device has to do lot of work whenever it recvs an LSP pkt , and that for every LSP pkt

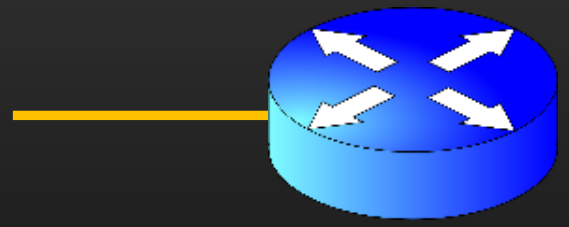
- If LSP incoming pkt rate is constant and last long, device CPU would be high (100 %)

- Device is overloaded !!

- If the device synchronously process all incoming LSP pkts, sine time to process one LSP pkt take significant time, rest of LSP packets may throttle the incoming queue and get dropped

- Solution :

- If device could have deferred the SPF computation for a short while, and only update the LSDB with all 3 LSPs and trigger SPF only after 3 LSP is updated in LSDB, then device would not have to redundant work (save CPU)



Note : SPF algo + Route cal/installation is quite a computationally intensive task and lot of research has been done to minimize the number of SPF runs

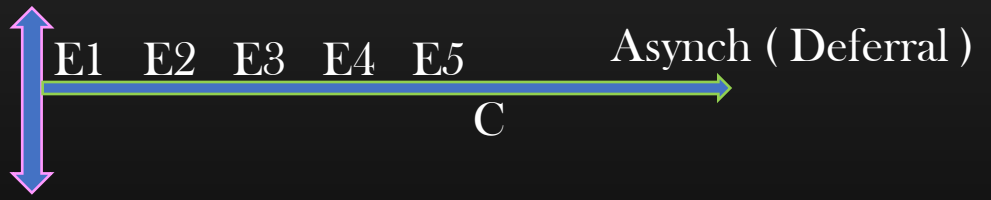
- As the size of our protocol grows, there shall exist several external stimuli which revokes ISIS protocol to trigger computation

- For example :
  - At time t1, say, a new LSP pkt recvd → device trigger SPF synchronously
  - At time t2, say, some adjacency goes up → device regen new LSP → trigger spf synchronously

  - Now if t1 and t2 are temporally very close, then we end up same problem, spf triggered by LSP pkt reception shall be redundant work

- Solution :

- If device could have deferred the SPF computation and trigger it only after t2, then only one SPF run suffice



In general : Events which are temporally close enough and invoke same computation C should invoke only one instance of computation C (relate it to real life)

In General : Work Deferral comes under asynchronous programming world, and is very powerful Linux kernel use work deferral techniques such as tasklets, work Queues, timers, waitQueues

- The problem of redundant computation is just not specific to ISIS protocol, but common to software applications whose design somewhat matches to ISIS protocol

  - Packet Reception at high rate
  - External events reception at high rate (IPCs)
  - Various code paths leading to same computation
  - Common to Network Protocols

- Problems becomes more pronounced as software size grows (many code paths leads to invocation to same computation) and software scale grows.

- In our project, any logically isolated non-trivial computation shall be performed asynchronously (by deferring it from main flow)
  - LSP packet generation
  - Flooding LSPs out of all local interfaces
  - Triggering SPF Algo
  - Route calculations and installation

- TCP-IP stack library provides the mechanism using which application can perform computation asynchronously

computation = F + ARG

instead of traditional fn call : F(ARG)

invoke a function F asynchronously :

```
task_t *task = task_create_new_job(ARG, F, TASK_ONE_SHOT);
```

Eg : `task_t *bar_task = task_create_new_job (arg, bar, TASK_ONE_SHOT);`

Synchronous Call

```
foo ( ) {

. . .
printf("this ");
bar (arg);
printf("is my fav ");
. . .
}
```

Asynchronous Call

```
foo ( ) {

. . .
printf("this ");
task_t *bar_task =
            task_create_new_job(arg, bar);
printf("is my fav ");
. . .
}
```

```
bar( void *arg) {

            printf ("bar");
}
```

- To cancel the already scheduled job :

```
task_cancel_job(bar_task);
bar_task = NULL;
```

- We wrote an API :
  - isis_create_fresh_lsp_pkt ( )

  - This API must be invoked when :

    - ISIS hello pkt recvd has different Hostname, if-index or metric value then what is there in local adjacency object on recipient interface [*isis_update_interface_adjacency_from_hello ( )*]
    - User config changed which results in content of LSP pkt to update
      - Interface up/down [*isis_handle_interface_up_down ( )*]
      - IP address change [*isis_handle_interface_ip_addr_changed ( )*]

  - Adjacency goes from init → UP , Or from UP → DOWN [*isis_change_adjacency_state ( )*]
  - Adjacency in UP state is deleted via CLI : clear node <node-name> protocol isis adjacency [*isis_delete_adjacency ( )*]
  - Protocol Initialization at node level
    - [isis_init ( )]
  - Periodically (later)

  - Remove the API call which we placed in show CLI backend handler, not required anymore

- Let us first integrate this API at all appropriate places in a code so that protocol always advertise the most up to date information in its LSP
  > Only Adjacencies in UP state as TLV 22

Network Protocol Development → Phase 2 : LSDB Mgmt → Sync Method of Generating LSP Packets → Testing

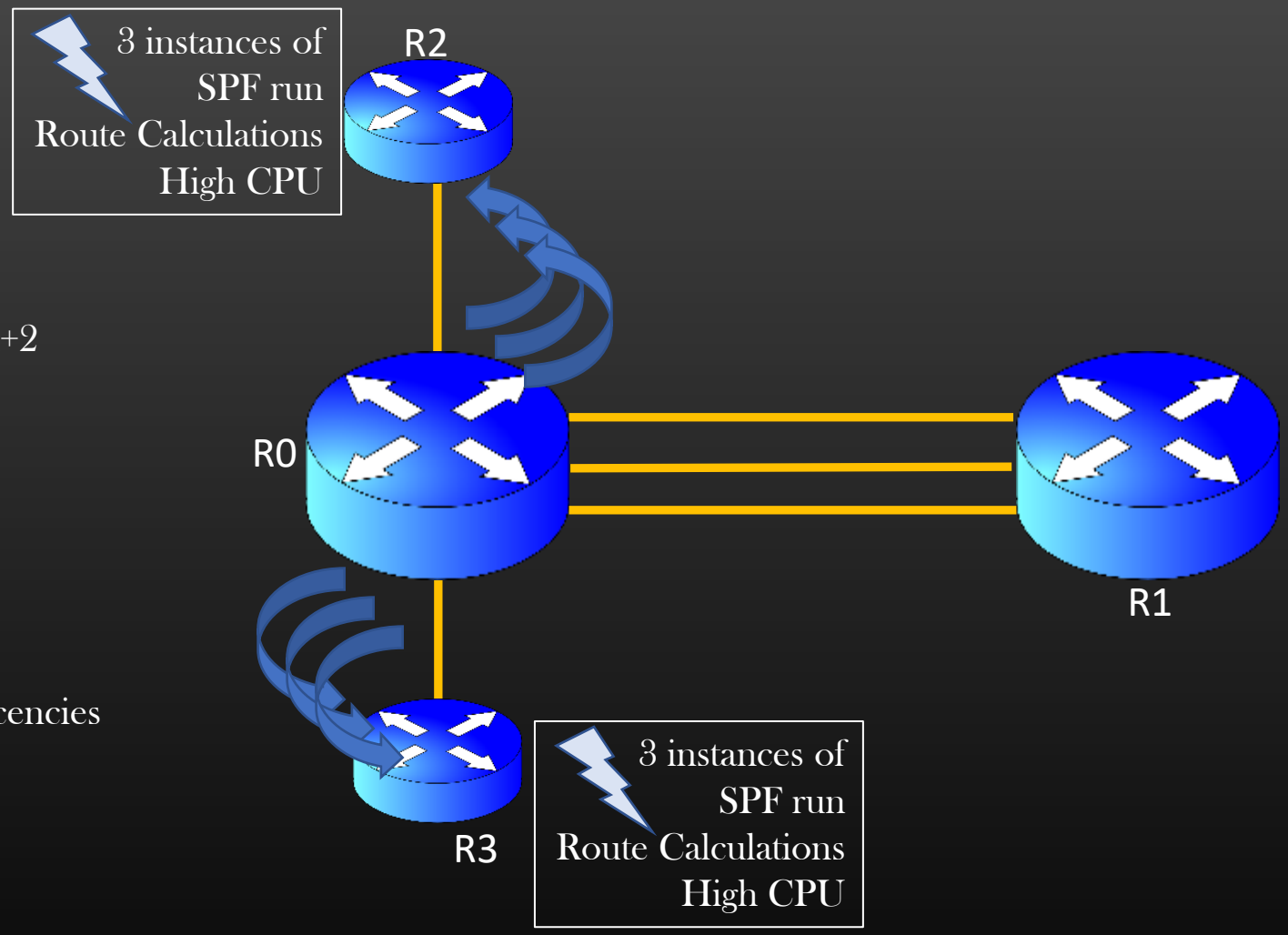
Redundant Computation Example Demo

Event Sequence :

1. Disable protocol on R1
2. 3 Adj on R0 time out almost same time
3. R0 generate 3 LSPs and flood with seq x, x+1, x+2
4. R0, R2 and R3 recvs 3 LSPs and trigger SPF 3 times
5. All routers – R0, R2, R3 are momentarily overloaded (~100% CPU)

Somewhat similar to *Denial of Service* Attack

Situation is more pronounced with more no of adjacencies between R1 and R0



- Invoking isis_create_fresh_lsp_pkt( ) asynchronously using API isis_schedule_lsp_pkt_generation( )
- Since LSP Gen is a node level work, therefore isis_node_info should contain a new member of type task_t *
- Any work that if deferred to be done future is called as task

```
protocol() {
        . . . . .
        printf ( "Adjacency 1 goes down");
        . . .
        isis_schedule_lsp_pkt_generation(node);
        . . .
        printf ( " Adjacency 2 goes down");
        printf ( " Adjacency 3 goes down");
        printf ( " Adjacency 4 goes down");
}
```

Adjacency 1 goes down
Schedule lsp pkt gen task
Adjacency 2 goes down
Adjacency 3 goes down
Adjacency 4 goes down
LSP pkt is generated

deferred

☜ Asynchronous LSP pkt Generation

```
protocol() {
        . . . . .
        printf ( "Adjacency 1 goes down");
        . . .
        isis_create_fresh_lsp_pkt(node);
        . . .
        printf ( " Adjacency 2 goes down");
        isis_create_fresh_lsp_pkt(node);
        printf ( " Adjacency 3 goes down");
        isis_create_fresh_lsp_pkt(node);
        printf ( " Adjacency 4 goes down");
        isis_create_fresh_lsp_pkt(node);
}
```

Adjacency 1 goes down
Lsp pkt generated
Adjacency 2 goes down
Lsp pkt generated
Adjacency 3 goes down
Lsp pkt generated
Adjacency 4 goes down
Lsp pkt generated

☝ Synchronous LSP pkt Generation

End Result is same : Most recent LSP pkt would contain same content as per latest view of topology

Periodic Generation of LSP

- An ISIS node not only have to generate self LSP periodically, but also must disburse it
  - This is done to ensure that all nodes of the topology has their LSDB in sync

- Disburse means – send the new LSP pkt out of all protocol enabled interfaces which has adjacency in UP state

- So let us implement the mechanism of periodic generation of self LSP

  - Timer must start as soon as protocol is enabled on node level
  - Timer must be stopped only when protocol is disabled on a node
  - Flooding Time interval : #define ISIS_LSP_DEFAULT_FLOOD_INTERVAL 120 // but take 15 sec for testing purpose

APIs : isis_flood.c/.h (new files)

```
void
isis_start_lsp_pkt_periodic_flooding(node_t *node);

void
isis_stop_lsp_pkt_periodic_flooding(node_t *node);
```

You are already familiar with the Timer library, pls try to do this by yourself.
Use show command to verify that self LSP pkt's sequence number increases by 1 periodically

Disburse of LSP

➢ Sending out the LSP packet out of all protocol enabled interfaces of a device which has Adj in UP state

```
isis_node_info_ {
    . . . .
    isis_lsp_pkt_t *self_lsp_pkt;
    . . . .
} isis_node_info_t;
```

```
isis_intf_info_ {
    . . . .
    glthread_t lsp_xmit_list_head;
    task_t *lsp_xmit_job;
    . . . .
} isis_intf_info_t;
```

☞ Every interface has a Queue which maintains list of LSPs pending to be sent out of that interface

☞ Every interface has a task/job whose responsibility is to Disburse the pending lsp queue

isis_flood.h

```
typedef struct isis_lsp_xmit_elem_ {

    isis_lsp_pkt_t *lsp_pkt;
    glthread_t glue;
} isis_lsp_xmit_elem_t;
```

Acts as a *glue* to attach LSP into Intf lsp xmit Queue

Disburse of LSP

- Sending out the LSP packet out of all protocol enabled interfaces of a device which has Adj in UP state

Overall Procedure :

1. LSP to be disbursed is Queued into pending lsp queue for all eligible interfaces

2. Task/job to process pending lsp queue for each interface is scheduled

3. When task/job of an interface is triggered, it dequeues LSPs from interface's pending queue one by one and send them out of the interface

4. When pending queue of an interface is empty, task/job is finished

5. When sending out a given LSP out of all eligible interface is completed, then LSP is said to have completed its disbursement procedure

Highly Asynchronous Design !

Disburse of LSP

1

```
void
isis_queue_lsp_pkt_for_transmission(
        interface_t *intf,
        isis_lsp_pkt_t *lsp_pkt) ;
```

isis_flood.h/.c

4 Steps :

Step 1 : Queue the lsp_pkt into interface lsp xmit Queue

```
isis_lsp_xmit_elem_t *lsp_xmit_elem =
                                calloc(1, isis_lsp_xmit_elem_t);
init_glthread(&lsp_xmit_elem->glue);
lsp_xmit_elem->lsp_pkt = lsp_pkt;
glthread_add_last(&intf_info->lsp_xmit_list_head,
                &lsp_xmit_elem->glue);
```

Step 2 : Fork out the new task to dispatch lsp pkt out of this Queue, if not already

```
if (!intf_info->lsp_xmit_job) {

    intf_info->lsp_xmit_job =
        task_create_new_job (intf, isis_lsp_xmit_job,
                                        TASK_ONE_SHOT);
}
```

Disburse of LSP

2

```
static void
isis_lsp_xmit_job(void *arg, uint32_t arg_size);
```

isis_flood.c

Step 3 :
Dequeue LSPs one by one from interface's lsp xmit queue
and send_pkt_out( ) it out of the interface

isis_flood.h/.c

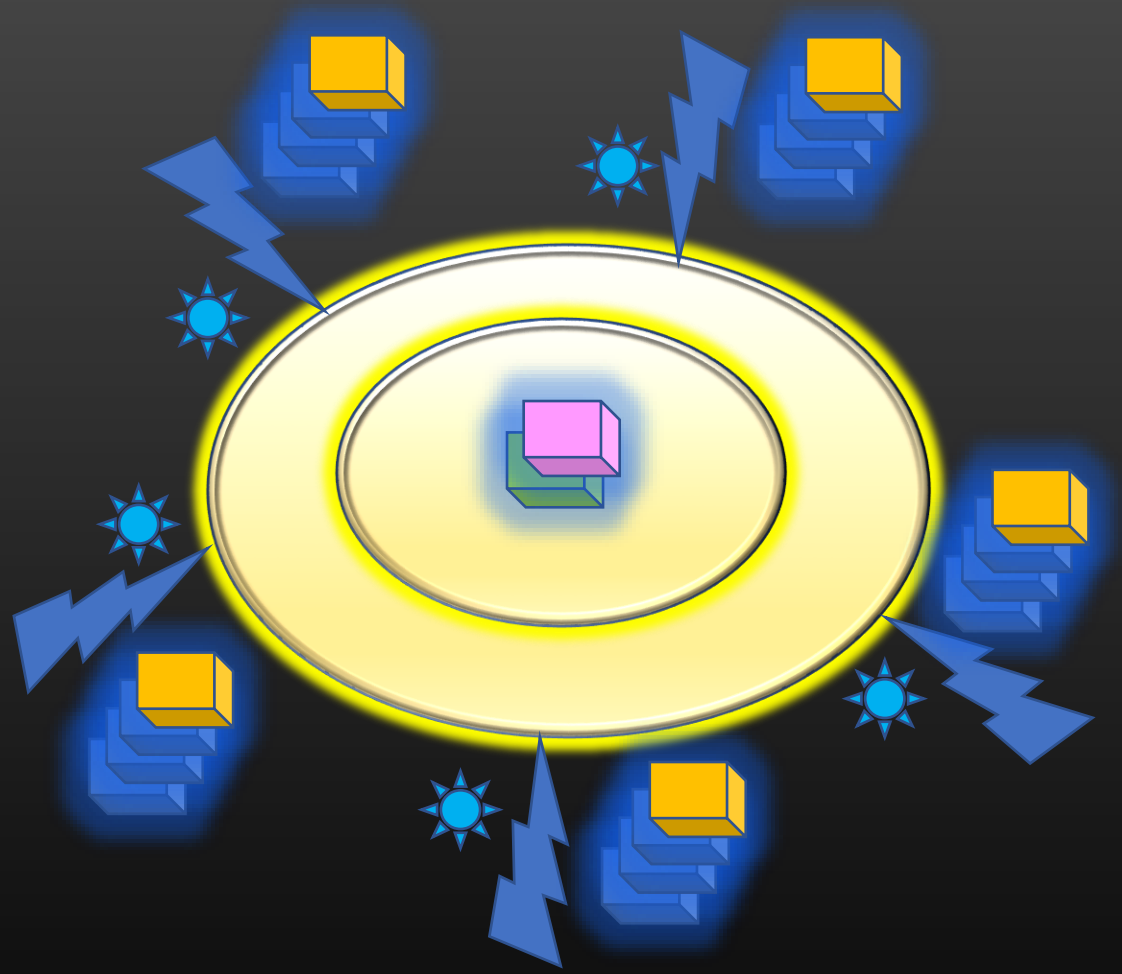
3

```
void
isis_schedule_lsp_flood(node_t *node,
                        isis_lsp_pkt_t *lsp_pkt,
                        interface_t *exempt_iif );
```

☜ Highest Level API to be used

Step 4 :
Iterate over all interfaces, and Queue LSP pkt
for transmission for each interface

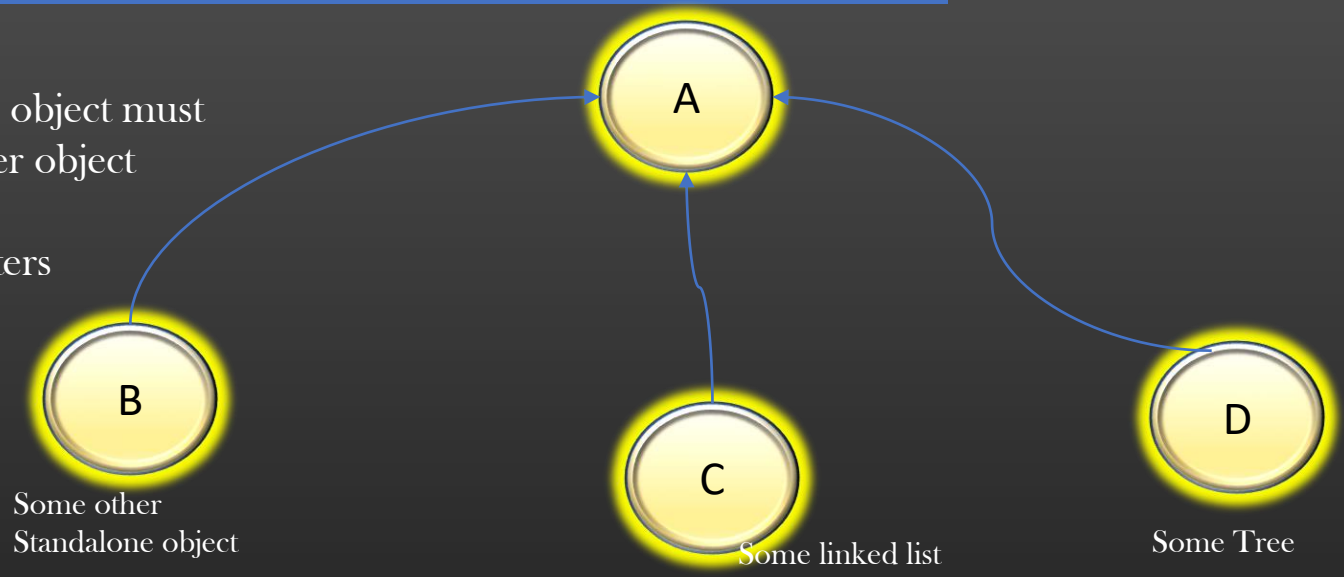
```
ITERATE_NODE_INTERFACES_BEGIN(node, intf) {

        if (intf == exempt_intf) continue;
         isis_queue_lsp_pkt_for_transmission(intf, lsp_pkt);

} ITERATE_NODE_INTERFACES_END(node, intf);
```



- Consider the protocol has Scheduled up its own LSP in LSP Xmit Queue of interfaces during LSP disbursement, at time t1
- At time t2, assume some Adjacency goes up (or down)
- Protocol will delete its own old LSP, and re-create a new one
- LSP xmit Queue would now hold the pointer to freed LSP packet (dangling reference), performing any operation on it would result in program crash, unexpected behaviour
- Moreover , if the old LSP is installed in LSDB, then LSDB would corrupt (freeing the node already inserted in a tree)
- Solution :
  - Protocol must free the lsp pkt only when no other object hold the reference to it
- Let us understand this problem in general and discuss solution

Reference Counting

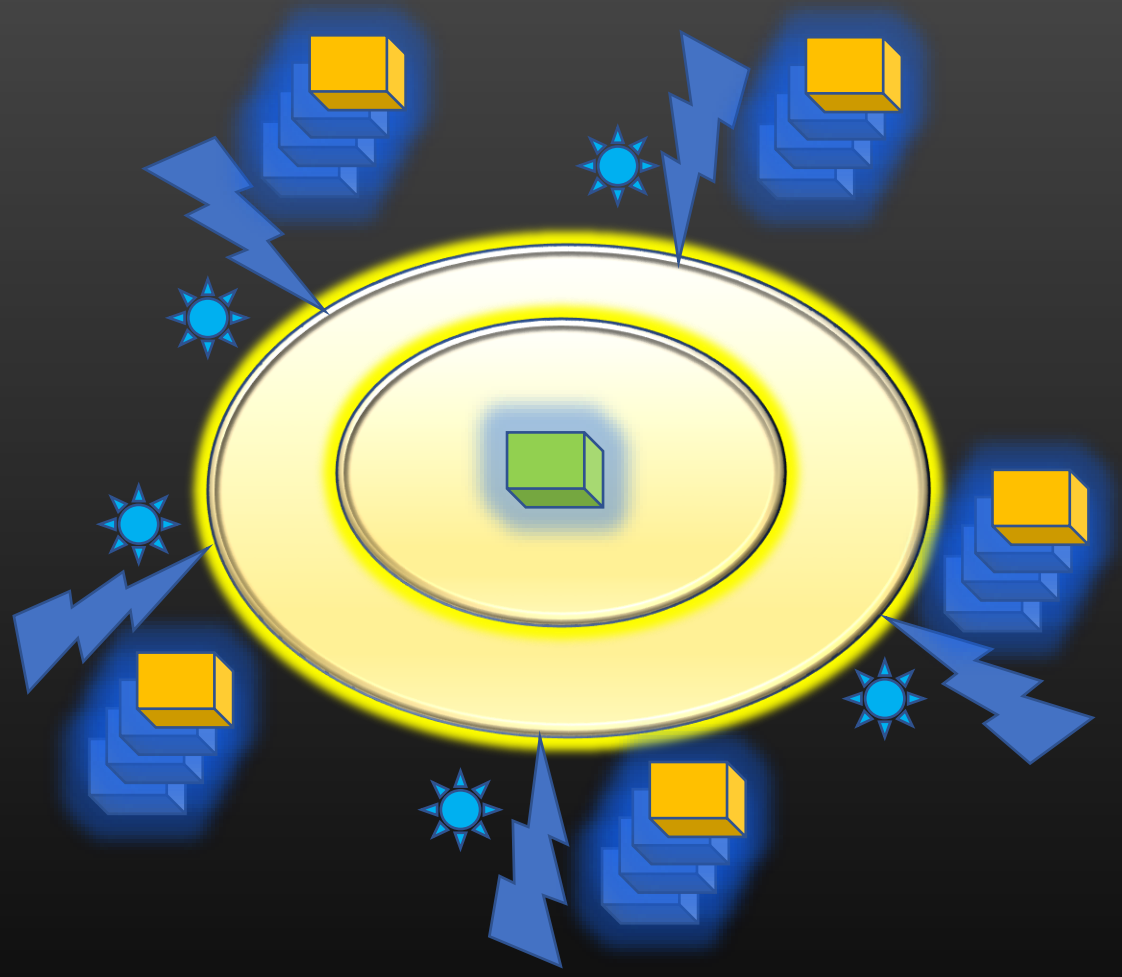
- When the application has a multi-reference object then such an object must be freed only when it is no more referenced by any other object
- Pre-mature free of such an object would result in dangling pointers
- Solution : Program must destroy the object A only when no other object has a reference to it
- Reference count



  - Maintain an integer ref_*count* of A which keep tracks regarding how many other objects are referencing to A
  - In this case it is 3
  - Malloc the object A with ref_count = 0
  - Increase the ref_count of A when some object holds the pointer to it (reference (A))
  - Decrease the *count* of A, when other object stop referencing object A (dereference (A))
  - Free the object A when ref_count of A reduces to 0
  - Never invoke free() on such objects directly, always free it through dereference( ) it

```
void reference(A) {
        A->ref_count++;
}
```

```
void dereference(A) {
        A->ref_count--;
        if (A->ref_count == 0 ) free(A);
}
```



- lsp_pkt->ref_count = 6 as per diagram

- When protocol creates a new LSP pkt, ref_count of old lsp_pkt = 5
  (node_info->self_lsp_pkt will now point to new LSP pkt)

- When old lsp pkt is disbursed out of interface I1, ref_count = 4
- When old lsp pkt is disbursed out of interface I2, ref_count = 3
- When old lsp pkt is disbursed out of interface I3, ref_count = 2
- When old lsp pkt is disbursed out of interface I4, ref_count = 1
- When old lsp pkt is disbursed out of interface I5, ref_count = 0
  - Old_lsp_pkt is freed !

isis_pkt.c/.h

```
void
isis_ref_isis_pkt (isis_lsp_pkt_t *lsp_pkt) ;

void
isis_deref_isis_pkt (isis_lsp_pkt_t *lsp_pkt);
```

So, now our disbursement logic is full proof

- First of all, we should be able to verify whatever we have implemented so far is bug-free
- So, lets work towards our short-term goal to show link state packet contents using show command

- LSPDB is a database of Link State Packets the device has recvd + its own LSP
  - Node level data structure

- We shall model LSPDB as a balanced AVL Tree, keyed by router id
  - Install
  - Delete
  - Lookup
  - Walk

- We shall install/replace LSP packets in LSPDB
- Homework : Before proceeding to next lecture video, pls go through Appendix A section

- Implement a show CLI to print the link-state database content

  show node <node-name> protocol isis lsdb [<rtr-id>]

- LSDB is modelled as AVL tree
  - Define new AVL tree as member in isis_node_info_t structure
- LSDB will contain isis_lsp_pkt_t type objects keyed by router id.
  - Define new Members in isis_lsp_pkt_t
- Initialize LSDB in isis_init ( ) , and destroy in isis_de_init( )
- APIs for LSDB Mgmt will go in isis_lsdb.c|.h (new files)
- Let us discuss what all APIs we would be needing to manage lsdb and implement them

isis_lsdb.h/.c

LSDB Integration Steps:

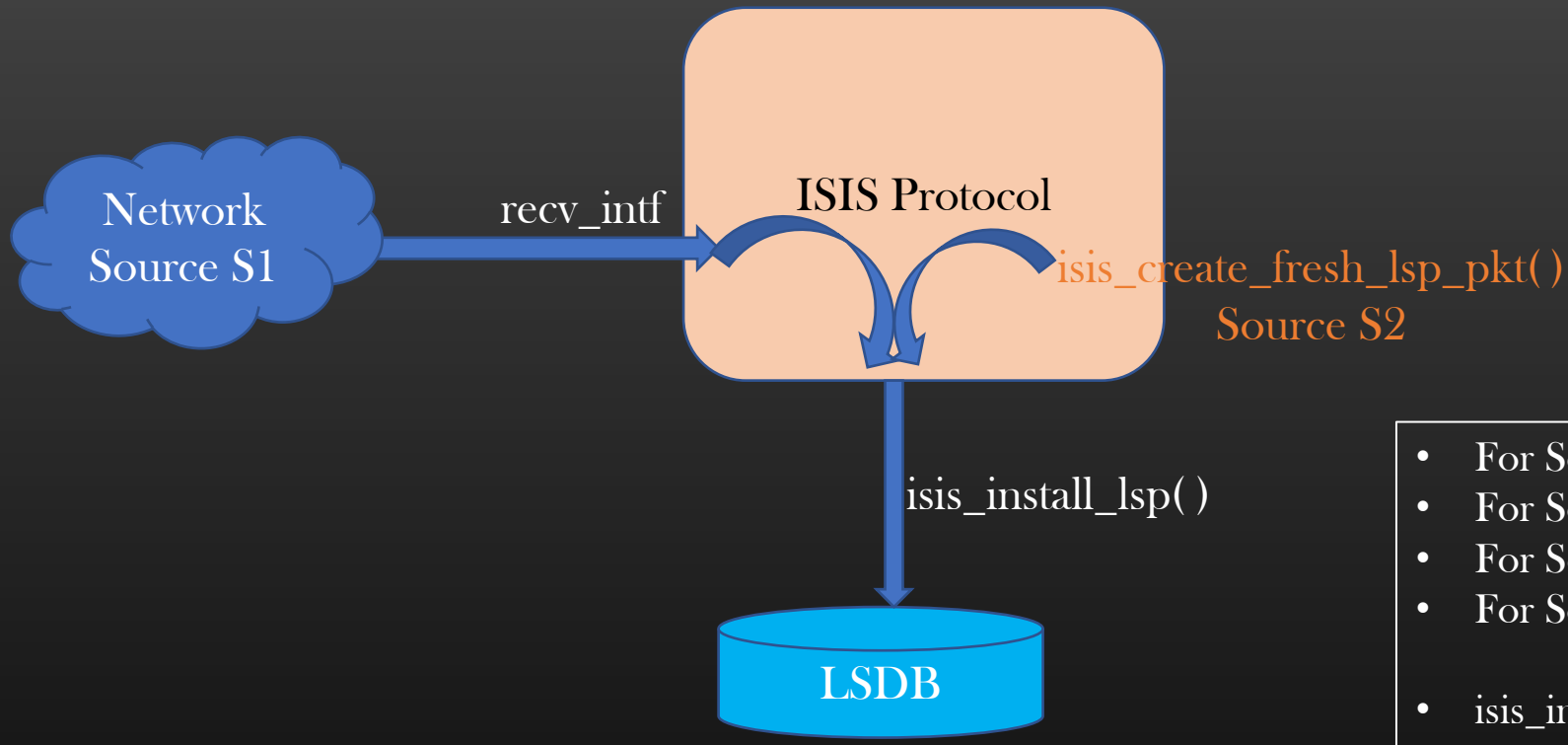
- Initialize isis_init ( )
  > avltree_init( )

- de_initialize in isis_de_init ( )
  - void isis_cleanup_lsdb(node_t *node);

- Install API : bool isis_add_lsp_pkt_in_lspdb(node_t *node, isis_lsp_pkt_t *lsp_pkt) ;
- Remove API : void isis_remove_lsp_from_lspdb(node_t *node, uint32_t rtr_id);
  : void isis_remove_lsp_pkt_from_lspdb(node_t *node, isis_lsp_pkt_t *lsp_pkt);

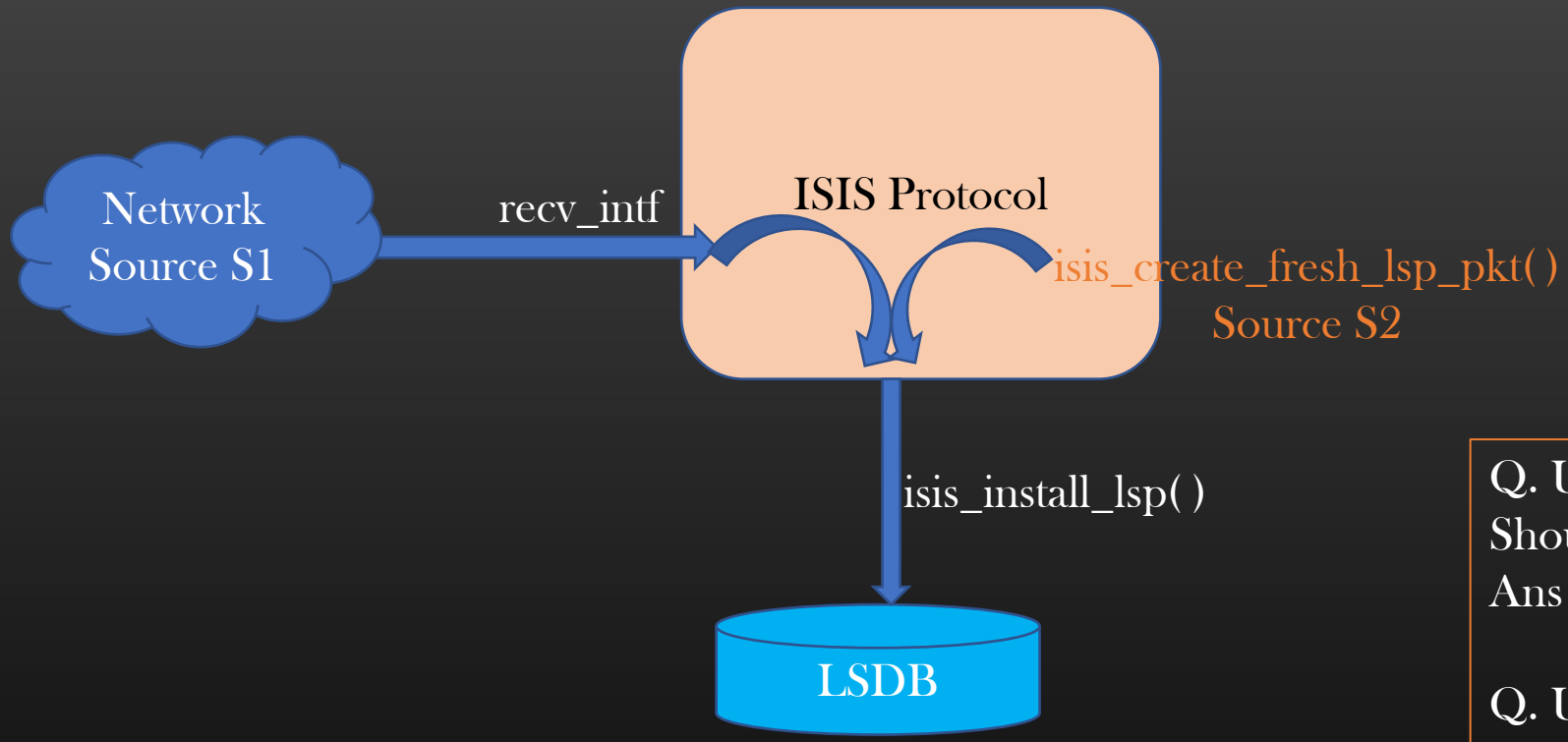
- Validation : bool isis_is_lsp_pkt_installed_in_lspdb(isis_lsp_pkt_t *lsp_pkt);
- Util : byte* isis_print_lsp_id(isis_lsp_pkt_t *lsp_pkt);
- Util : bool isis_our_lsp(node_t *node, isis_lsp_pkt_t *lsp_pkt) ;
- Show : void isis_show_one_lsp_pkt_detail(node_t *node, char *rtr_id_str);
- Show : void isis_show_lspdb(node_t *node);
- Lookup : isis_lsp_pkt_t * isis_lookup_lsp_from_lsdb(node_t *node, uint32_t rtr_id) ;
- Look up : avltree_t * isis_get_lspdb_root(node_t *node);

- All ISIS nodes has to process the LSP packets it recvs over interface Or self-originates



- For Source S2, recv_intf = NULL
- For Source S2, LSP is always self-lsp
- For Source S1, recv_intf != NULL
- For Source S1, remote node's LSP or self-LSP can also be recvd
- isis_install_lsp( ) is an interface between ISIS core and LSDB
- isis_install_lsp( ) API implements algorithm to look after LSP installation in LSDB
  - Remove the obsolete one
  - Trigger LSP flood
  - Regenerate new self LSP

- Revisiting Ref count of LSPs (Covered as Assignment)



Q. Under normal protocol operation, what Should be the ref count of self LSP ?
Ans : ?

Q. Under normal protocol operation, what Should be the ref count of remote LSPs ?
Ans : ?

Normal Operation : No LSP is Queued for disbursement by protocol

- Remember LSPs are multi-reference objects
- we need to increase the ref count of LSP if it is installed in lsdb
- Dec the ref count if it is removed from lsdb
- Code changes
- We shall soon going to write a show command to verify ref counts of LSPs are correct

Network Protocol Development → Phase 2 : LSDB Mgmt → LSP Flooding Algorithm

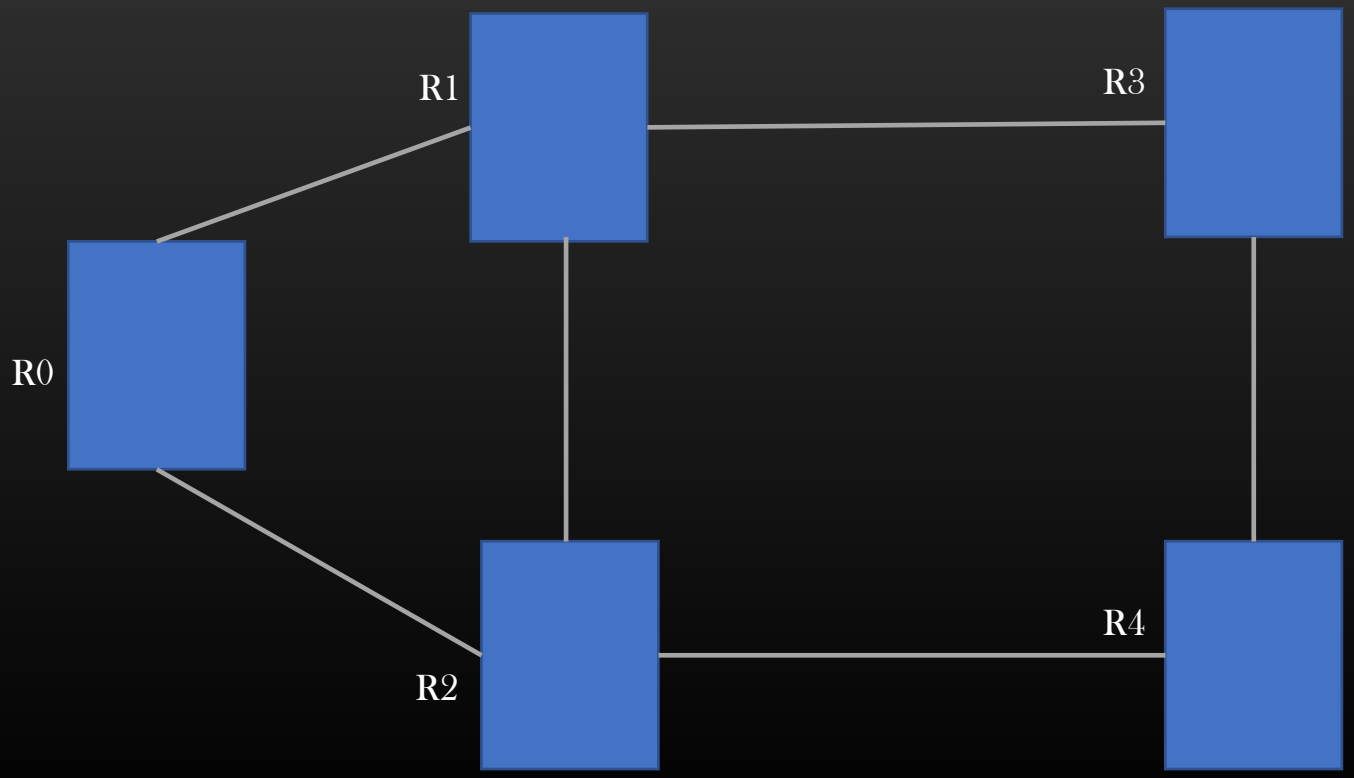
- Ever ISIS node has to distribute its own LSP throughout the ISIS topology

- This is done via Flooding Algorithm
  - Ensure Every Node's LSP Pkt is distributed to every other node
  - Ensure the LSDB of all nodes is in sync and accurate
  - Ensure there is no infinite distribution of LSPs (infinite loop)
  - Ensure Addition/Deletion of new nodes in the topology (Later)

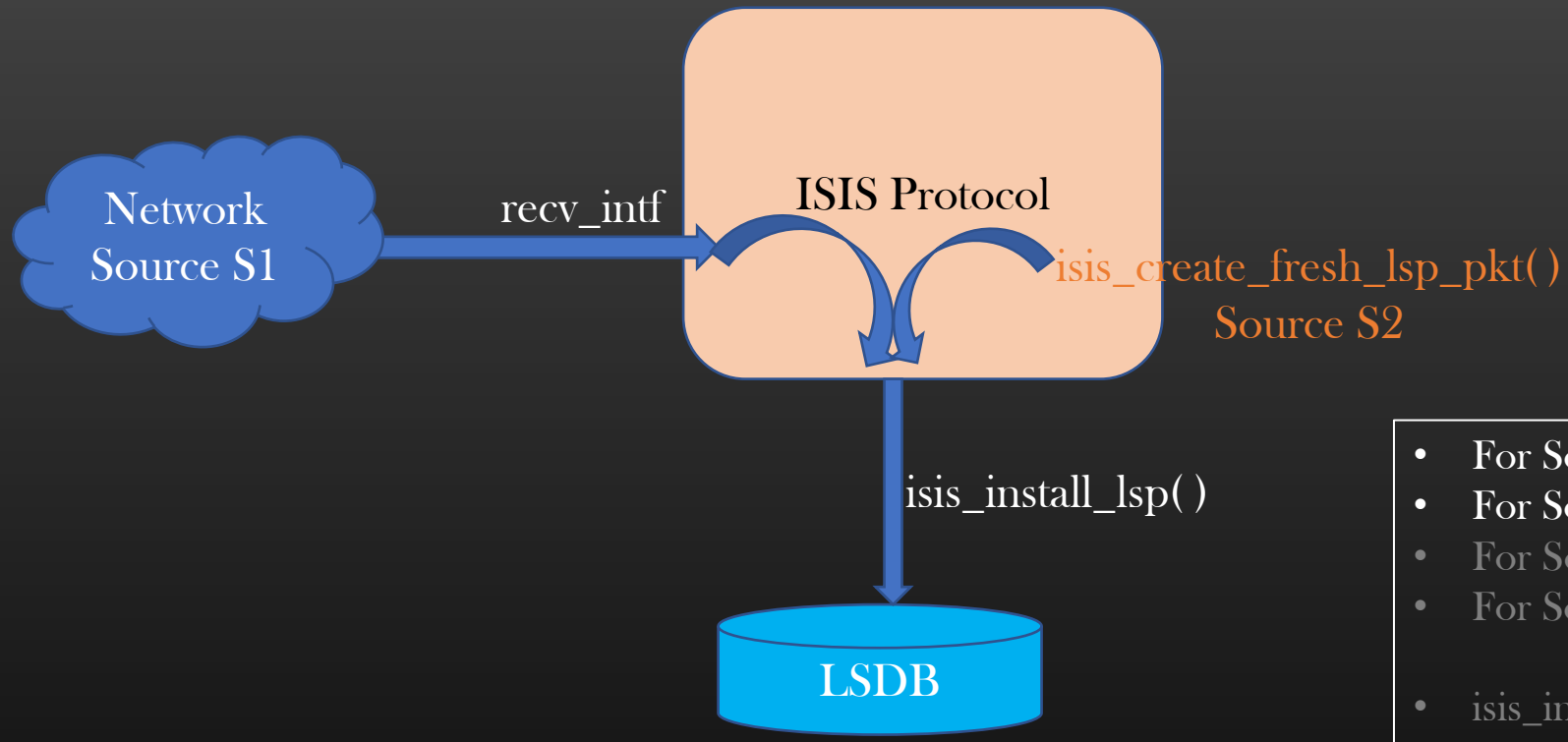
- The LSP's Sequence no plays a key role in flooding Algorithm Implementation

Two variations of LSP flooding :

- Forward flooding
- Blind flooding



- All ISIS nodes has to process the LSP packets it recvs over interface Or self-originates

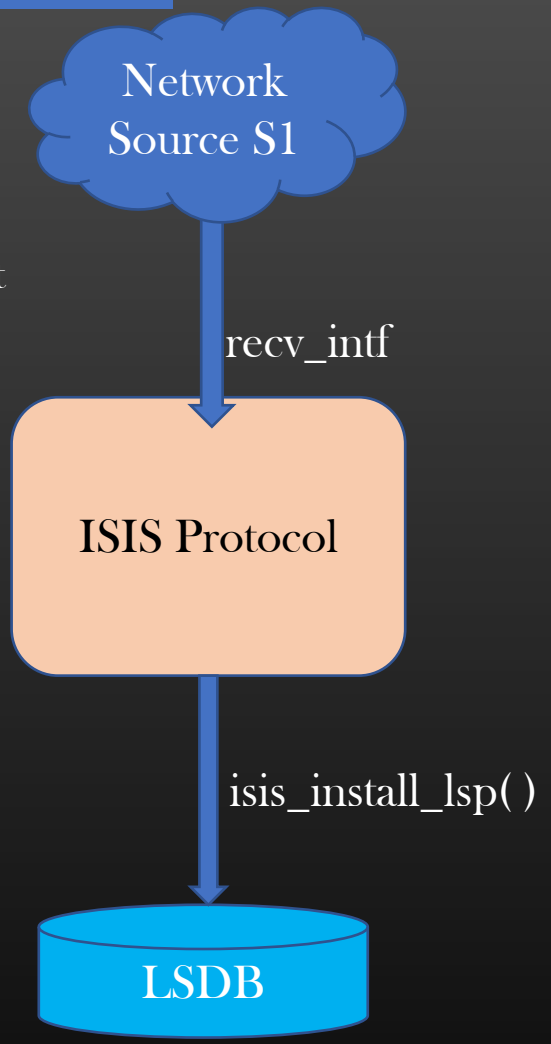


- For Source S2, recv_intf = NULL
- For Source S2, LSP is always self-lsp
- For Source S1, recv_intf != NULL
- For Source S1, remote node's LSP or self-LSP can also be recvd
- isis_install_lsp( ) is an interface between ISIS core and LSDB
- isis_install_lsp( ) API implements algorithm to look after LSP installation in LSDB
  - Remove the obsolete one
  - Trigger LSP flood
  - Regenerate new self LSP

Network Protocol Development → Phase 2 : LSDB Mgmt → Self LSP Installation Rules

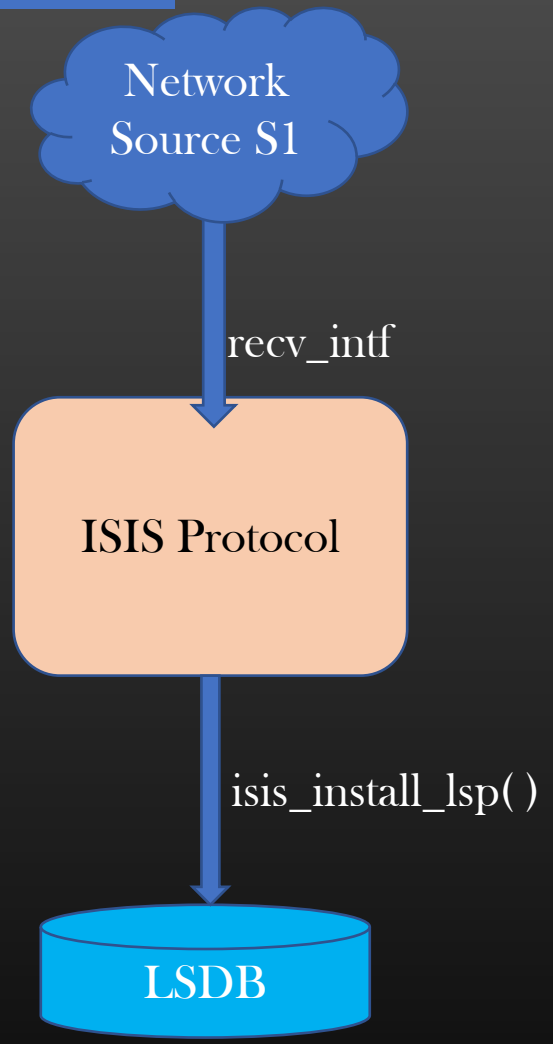
- ISIS have to maintain LSDB which store all remote LSPs recvd + Self own LSP
- It is ISIS responsibility that LSPs installed in LSDB is correct and accurate
- When ISIS regenerates its own LSP, it should install the new self LSP in LSDB replacing the old one if exist
- Node may recv its own LSP from Remote Nodes as well as a part of flooding Algorithm

Event	Criteria	Recvd via interface	Action	Remark
isis_event_self_duplicate_lsp	isis_our_lsp(node, new_lsp) == TRUE && new_lsp->seq_no == old_lsp->seq_no	Yes	Ignore the LSP	As a part of flooding algorithm, Node may recv its own duplicate lsp from other node. Since node already has it, no action
		No	assert(0);	Node never generate a new LSP pkt with same seq no as before
isis_event_self_fresh_lsp	isis_our_lsp(node, new_lsp) == TRUE && old_lsp == NULL	No	Install the LSP in LSDB, forward flood it	Propagate the LSP further as a part of flooding algorithm
		Yes	Ignore the LSP, regenerate self LSP pkt with higher seq no and blind flood	Somebody trying to game you ! Node receiving its own LSP when it itself doesn't have one, Node would try to overwrite its LSP in other node's lsdb
isis_event_self_new_lsp	isis_our_lsp(node, new_lsp) == TRUE && new_lsp->seq_no > old_lsp->seq_no	Yes	Ignore the LSP, regenerate self LSP pkt with higher seq no and blind flood	Somebody trying to game you ! Node receiving its own obsolete LSP, Node would try to overwrite its LSP in other node's lsdb
		No	Replace the new_lsp in LSDB with old_lsp, blind flood the new_lsp	Node is refreshing its own LSP with higher sequence no.
isis_event_self_old_lsp	isis_our_lsp(node, new_lsp) == TRUE && new_lsp->seq_no < old_lsp->seq_no	yes	Ignore the LSP, blind flood own LSP	Node would blind its own LSP so as to overwrite its own old lsp in other node's lsdb
		No	assert(0);	Node cannot generate new LSP with lower seq no



- ISIS have to maintain LSDB which store all remote LSPs recvd + Self own LSP

Event	Criteria	Recvd via interface	Action	Remark
isis_event_non_local_duplicate_lsp	isis_our_lsp(node, new_lsp) == FALSE && new_lsp->seq_no == old_lsp->seq_no	Yes	Ignore the LSP	As a part of flooding algorithm, Node may recv own duplicate remote lsp from nbr node
		No	assert(0);	Node never generate remote LSPs
isis_event_non_local_fresh_lsp	isis_our_lsp(node, new_lsp) == FALSE && old_lsp == NULL	No	assert(0)	Node never generate remote LSPs
		Yes	Add LSP in DB, Forward flood it	Node recvd LSP of remote node for the first time, install it
isis_event_non_local_new_lsp	isis_our_lsp(node, new_lsp) == FALSE && new_lsp->seq_no > old_lsp->seq_no	Yes	Replace the old_lsp with new_lsp in lsdb, forward flood new lsp	Node recvd more recent remote LSP, update the lsdb with new LSP
		No	assert(0)	Node never generate remote LSPs
isis_event_non_local_old_lsp	isis_our_lsp(node, new_lsp) == FALSE && new_lsp->seq_no < old_lsp->seq_no	yes	Ignore the LSP , shoot back the LSP already in LSPDB on receiving interface	Node would try to overwrite the older remote LSP in other node LSDB by advertising the newer remote LSP is has
		No	assert(0);	Node never generate remote LSPs



- We implemented isis_install_lsp( ) which implements the
  - Local LSDB database update
  - LSP flooding
  - LSDB synchronization

- Any bug introduced in this fn can have devastating effect :
  - LSPs can go in infinite flooding
  - LSPs can replicate to millions (like bacteria , causing system going OOM)
  - System hang, process killed
  - Functional bugs – LSPs missing in LSDB, LSDBs is not in sync state across topology
  - What about Memory leaks ?

- You need to test out the API starting from smaller topologies
  - 1 point-to-point link between two devices
  - Gradually increase topology size by adding nodes and links
  - Test out your code for at-least 5 days
  - Use show CLIs to verify the functionality

- Bug Fixes :
  - We had introduced 2 bugs in isis_install_lsp ( ) , lets exercise how did I resolve them
  - You will suffer if you had not put traces in your code, specially in this function (dynamic feature)
  - You would have to depend on logs/tcp_log_file to resolve bugs related to lsdb mgmt, gdb wont help much

  - I give you some instances how to resolve bugs related to dynamic features ! Same Technique applies to production code

- Phew !! The core part of the proto dev is finished ! Congratulations once again.
- But the show is not finished yet, we have completed approx 70% of the protocol development
- And Our Protocol is growing in size .. Have you counted # of lines we have coded so far in isis dir
- It is desirable to introduce some event counters which helps user keeps a track regarding how many times what events have gener
  triggered by the protocol during the course of its operation in the network

  Eg :
  Counter to count no of self LSP pkt generation
  Counter to count no of Adj state transitions
  Counter to count LSP recv events
  Counter to count no of times adjacency attributes changed
  and many more ...

- These counters give an idea to the user how the ISIS protocol is reacting to the network, health check of the network as well indi
- Very helpful in debugging as well
- Addition of more features, introduce more related counters in future

```
tcp-ip-stack> $ show node R2 protocol isis event-counters
Parse Success.
Event Counters :
 ISIS EVENT ADJ STATE CHANGED : 4
 ISIS EVENT ADMIN CONFIG CHANGED : 1
 ISIS EVENT NBR ATTRIBUTE CHANGED : 0
 ISIS_EVENT_UP_ADJACENCY_DELETED : 0
 ISIS EVENT SELF DUPLICATE LSP : 0
 ISIS EVENT SELF FRESH LSP : 1
 ISIS EVENT SELF NEW LSP : 6
 ISIS EVENT SELF OLD LSP : 0
 ISIS EVENT NON LOCAL DUPLICATE LSP : 59
 ISIS EVENT NON LOCAL FRESH LSP : 5
 ISIS EVENT NON LOCAL NEW LSP : 27
 ISIS EVENT NON LOCAL OLD LSP : 0
 ISIS EVENT PERIODIC LSP GENERATION : 1
 ISIS EVENT ADMIN ACTION DB CLEAR : 0
CLI returned
```

```
typedef struct isis_node_info_ {
. . .
   /* event counts */
    uint32_t isis_event_count[isis_event_max];
. . .
} isis_node_info_t;
```

- If user disables the protocol ISIS on a node of the topology, then rest of the nodes in the topology must almost immediately delete the LSP pkt of the former node, and must not rely on timeout of LSP timer

- This is achieved via Purge LSP

- A purge LSP is an LSP in which there is no TLVs except ISIS_TLV_HOSTNAME and only isis_pkt_hdr_t, with purge bit set in isis_pkt_hdr_t->flags

  #define ISIS_LSP_F_PURGE_LSP (1) in isis_const.h

- When other routers in the topology recvs this purge LSP pkt, they immediately delete the existing LSP pkt from their LSDB and forward-flood the purge LSP

- Purge LSP is also flooded using same LSP flooding algorithm, whichever node recvs it, delete the copy the LSP pkt from its LSDB from forward-flood it

- Result : All nodes in the topology have got rid of LSP packet of a dead ISIS router ! LSDB of all nodes end up in sync
- Demo

```
void
isis_create_and_flood_purge_lsp_pkt_synchronously (node_t *node);
```

- isis_create_fresh_lsp_pkt(node); // create purge LSP
- isis_flood_lsp_synchronously(node, node_info->self_lsp_pkt); // flood the purge LSP

Read flags

```
void
isis_de_init (node_t *node) {

    interface_t *intf;
    isis_node_info_t *isis_node_info = ISIS_NODE_INFO(node);

    if (!isis_node_info) return;

    isis_create_and_flood_purge_lsp_pkt_synchronously(node);

    . . .
    . . .
```

```
typedef struct isis_node_info_ {
. . .
    /* control flags to generate LSP */
    uint8_t lsp_gen_flags;
. . .
} isis_node_info_t;
```

```
/* LSP pkt generation flags in isis_pkt.h */
#define ISIS_LSP_PKT_CREATE_PURGE_LSP   1
```

```
void
isis_create_and_flood_purge_lsp_pkt_synchronously (node_t *node);
```

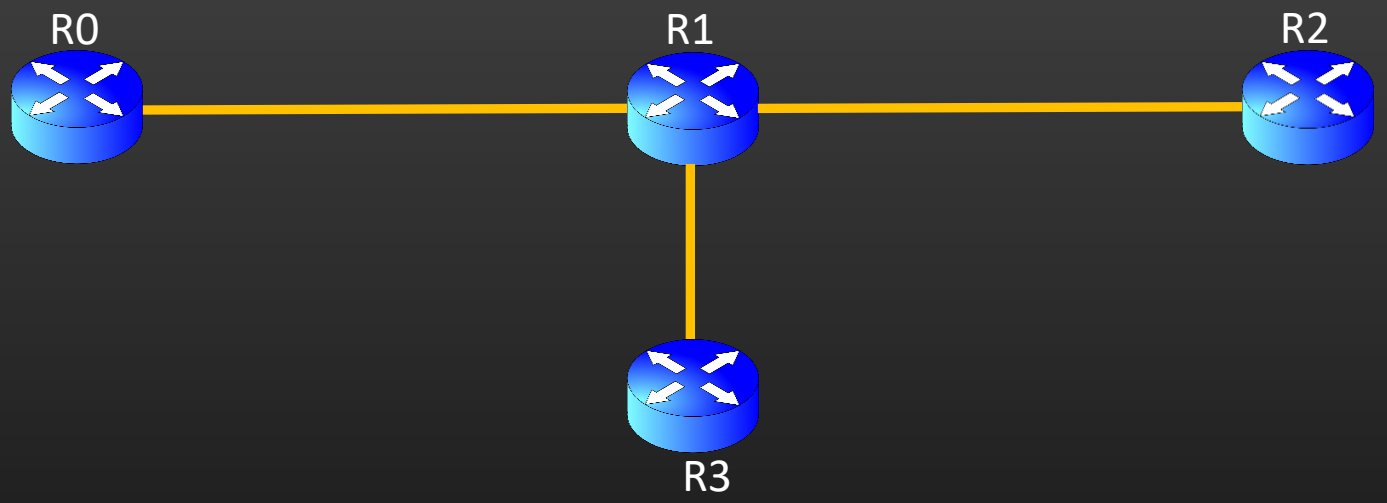
- isis_create_fresh_lsp_pkt(node); // create purge LSP
- isis_flood_lsp_synchronously(node, node_info->self_lsp_pkt); // flood the purge LSP

```
void
isis_flood_lsp_synchronously (node_t *node, isis_lsp_pkt_t *lsp_pkt) {

    interface_t *intf;
    isis_intf_info_t *intf_info ;

    ITERATE_NODE_INTERFACES_BEGIN (node, intf) {

        if (!isis_node_intf_is_enable((intf))) continue;

        intf_info = ISIS_INTF_INFO(intf);

        if (intf_info->adjacency &&
            intf_info->adjacency->adj_state == ISIS_ADJ_STATE_UP) {

                send_pkt_out (lsp_pkt->pkt, lsp_pkt->pkt_size, intf);
        }

    } ITERATE_NODE_INTERFACES_END (node, intf);
}
```

- Protocol Convergence
  - When the protocol is disturbed (adj gone up/down, admin config etc), it triggers the entire computation cycle :
    - Regenerate self LSP and disburse
    - Run SPF algorithm
    - Compute routes
    - Update Routing Table
  - Time interval beginning from the point when protocol was stimulated (t1), to the point when protocol has updated it Routing table again (t2) is called protocol convergence time (t2 – t1)
  - Lesser convergence time is desirable
  - Until the protocol converges, Routing table has stale routes which results in traffic loss, traffic loop etc (Explained in theory sec)
  - Protocol undergoes convergence are of two types :
    - Event based triggered convergence (Fast)
    - Timer based triggered convergence (Slow)

- Protocol Convergence

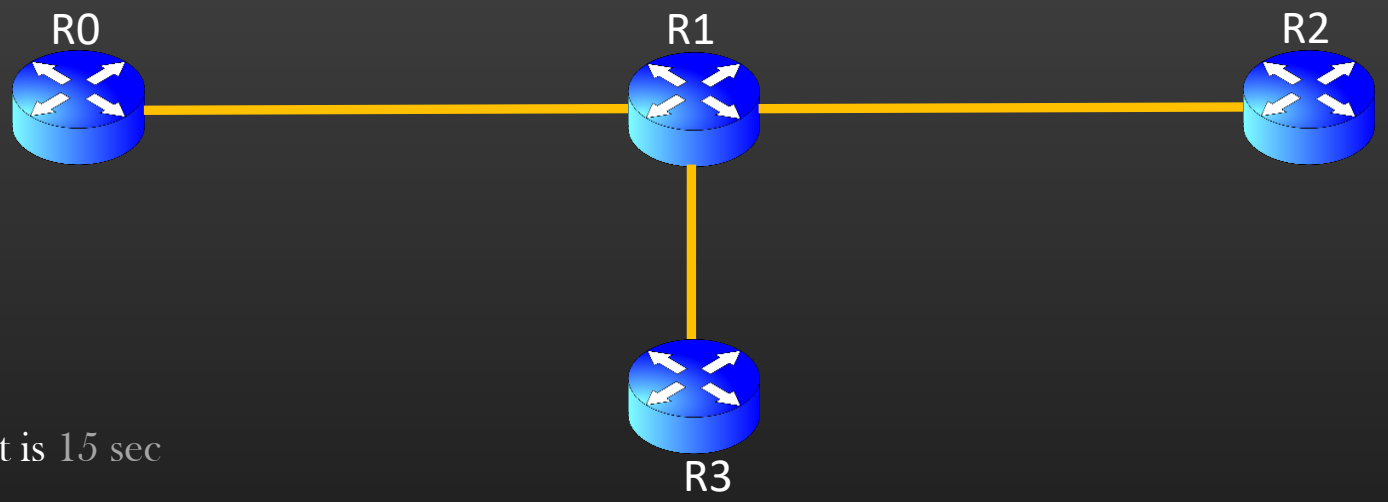
  Ex of Event based convergence (Fast)



  - All routers recvs the update almost instantly

➢ Protocol Convergence

Ex of Timer based convergence (Slow)



Assume flood time interval set is 15 sec

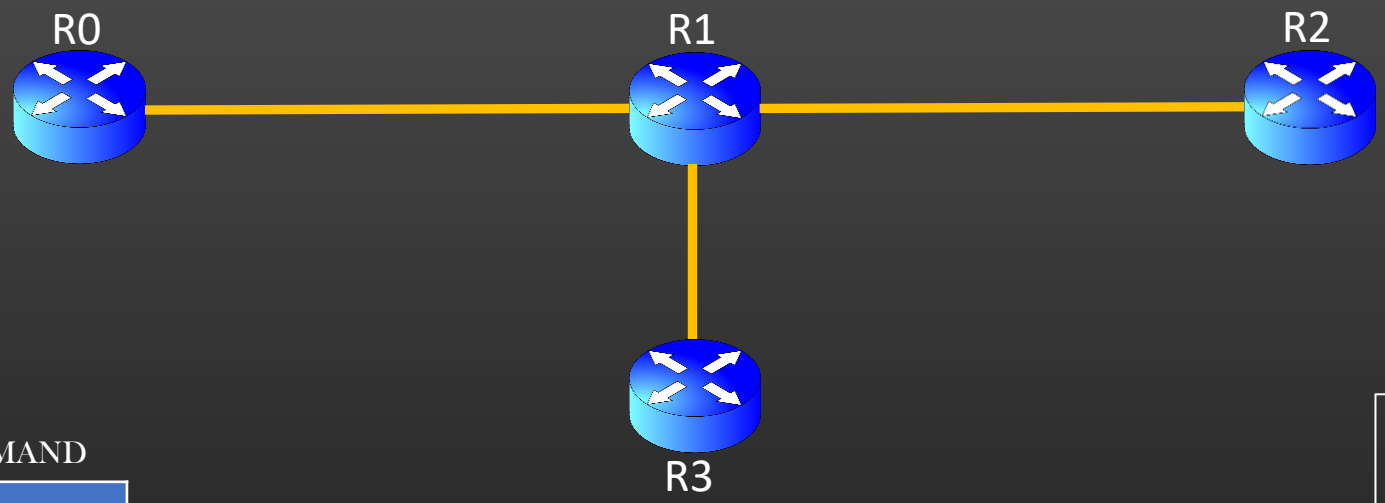
1. At t1 R1->R0 Adj goes up
   R1 regen LSP-R1 and flood
   R2 and R3 recvs and update lsdb
   R0 ignores because R0->R1 adj is not up yet

2. At t2, R0->R1 Adj goes up
   1. R0 regen LSP-R0 and flood
   2. R1 recvs and update lsdb
   3. R1 forward flood
   4. R2 and R3 recvd and update lsdb

☠R0 would not recv LSPs of R1, R2 & R3 until they periodically floods it

- Higher the flood time interval, higher the convergence time
- Setting flood time interval too rigorous would result frequent LSP flooding which would eat up lot of network resources in big topologies (Scaling problem)
- Solution : Reconciliation

Network Protocol Development → Phase 2 : LSDB Mgmt → Reconciliation Explained



ISIS_TLV_ON_DEMAND

Type = 111
1
1 Or 0

```
typedef struct isis_reconc_data_ {

   /* is reconciliation going on */
    bool reconciliation_in_progress;
    /* reconciliation timer */
    timer_event_handle *reconciliation_timer;
} isis_reconc_data_t;
```

- When the ISIS router detects its adjacency has gone UP, it enters into reconciliation phase (void isis_enter_reconciliation_phase(node_t *node))
- In reconciliation phase, router generates its self-LSP pkts at an interval of 2 sec (default)
- On demand TLV is inserted into LSP pkt when router is operating in reconciliation phase
- When other router recvs LSP pkt with OD TLV, it blind floods its own LSP in addition to processing the LSP as per the normal LSP flooding algorithm, provided that recipient router is not running in reconciliation phase
- When router enters reconciliation phase, it also starts the reconciliation timer (10 sec) (void isis_start_reconciliation_timer(node_t *node))
- When reconciliation timer expires, router exits the reconciliation phase and resume self lsp pkt periodic flooding as normal (void isis_exit_reconciliation_phase(node_t *node))
- When router operating in reconciliation phase detects adj transitions, it restarts the reconciliation timer (void isis_restart_reconciliation_timer(node_t *node))
- Router operating in Reconciliation phase process remote router's LSPs as normal (no change)

```
void isis_stop_reconciliation_timer(node_t *node)
bool isis_is_reconciliation_in_progress(node_t *node)
```

Network Protocol Development → Phase 2 : LSDB Mgmt → Reconciliation Explained

- Router operating in Reconciliation phase has the following special behaviour :
  - Generate self LSPs in every 2 sec , overriding the flood LSP interval value
  - Runs the Reconciliation timer
  - Restart the reconciliation timer, when any Adj goes UP

- Whether or not Router is operating in Reconciliation timer, it always responds to On-Demand TLV

- The purpose of the Reconciliation phase is to enable router to snatch the LSPs from other routers in the topology !
  - Like its your first day at school and you are cracking funny jokes to make/attract new friends in the first week ☺

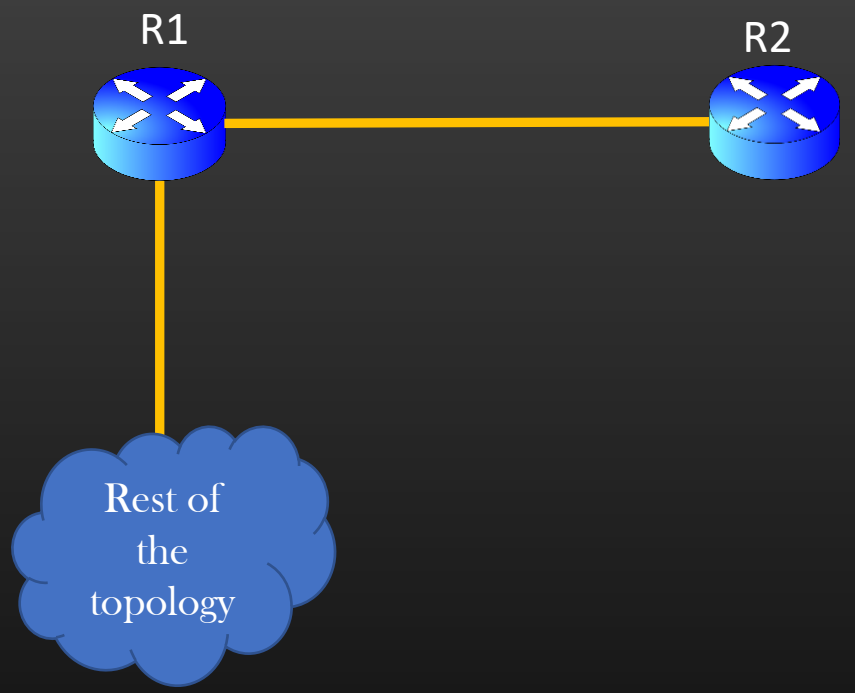
- LSDB sync no more depends on periodic LSP flooding due to reconciliation

- Lets Code Reconciliation !

- When other router recvs remote LSP pkt with OD TLV :
  - if recipient router is executing in reconciliation phase
    - Process LSP as per the LSP flooding algorithm (isis_install_lsp ( ))
    - ignore OD TLV
  - Else
    - Process LSP as per the LSP flooding algorithm (isis_install_lsp ( ))
    - blind floods its own LSP with higher seq no

Solution :

```
isis_install_lsp( ) {
                ...
if ( !isis_is_reconciliation_in_progress (node) &&
    !self_lsp &&
    isis_present_on_demand_tlv(new_lsp_pkt) &&
    !node_info->lsp_gen_task) {

     isis_schedule_lsp_pkt_generation(node);
}
}
```

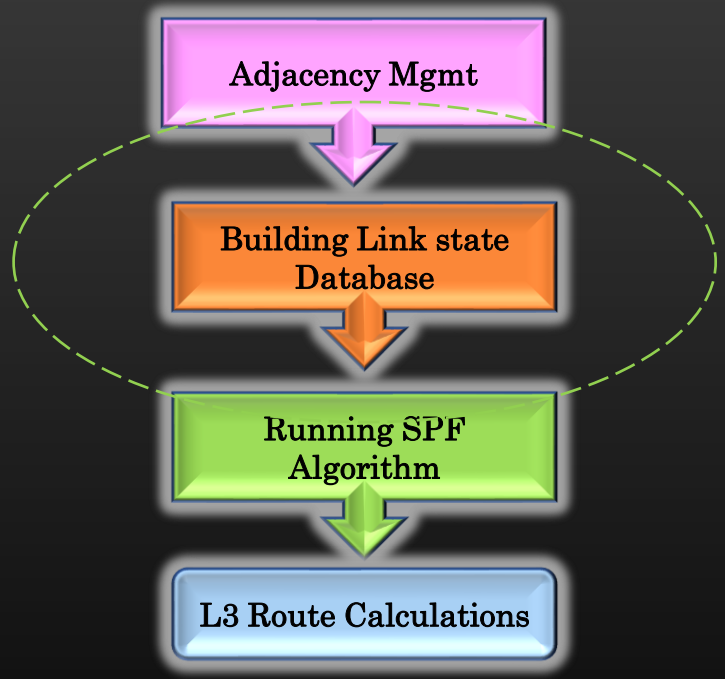


☠If two routers are in reconciliation phase, then they form LSP storm in the network, that's the reason they should ignore OD TLV

Where to go from here ?

- We will be going to implement a simplified Routing Protocol in this course
- Routing protocol chosen – Interior gateway protocol (IGP , ex OSPF, ISIS)
  - Don't know about it – don't worry, we shall cover theory first before any implementation
- A typical IGP (link state) protocol functionality is divided into 4 distinct parts :

1. Adjacency Management (Each device know its neighbours)
   - Sending and Receiving hello packets periodically
   - Update neighborship state machine
2. Building Link State Database (Each device internally creates a view of topology - Graph)
   - Building Link State packets
   - Flooding link state packets
   - Build a Graph – a view of network topology
3. Running SPF algorithm (Dijkstra) on LSDB
   - Process the LSDB through the algorithm
   - Compute Results and store
   - Algorithmically challenging
4. L3 Route Calculations
   - Use Results of 3 to compute final L3 routes and update Routing Table
   - Algorithmically challenging



We shall be going to implement all 4 parts in this course series

Along the journey we shall implement various sub-features within the protocol

Network Protocol Development → Phase 2

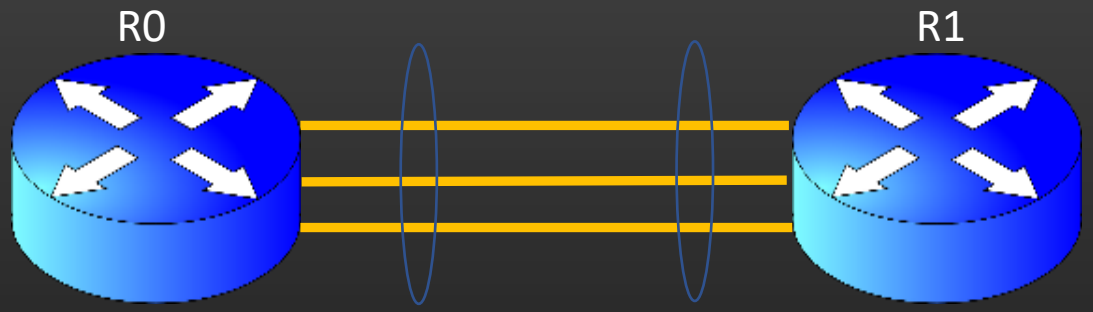
Network Protocol Development → Phase 2 : Mini Project – Interface Groups

- We have completed the core development of phase 2 of the project and now its time to add one feature to the protocol

- This is optional feature – even if you skip, further project development is not impacted

- Interface groups – A feature to optimize LSP flooding
  - Pledge that
    - You have thoroughly tested LSDB Mgmt and LSP flooding
    - You have thoroughly completed all assignment
    - You have all show/config CLIs in place which we have discussed so far
    - You are confident that there are no major bugs in the project
    - Your code is stable and ready for further development

- There is a lot of redundant flooding of LSPs whenever there are parallel links in the topology
- Waste Processing, eats up network bandwidth



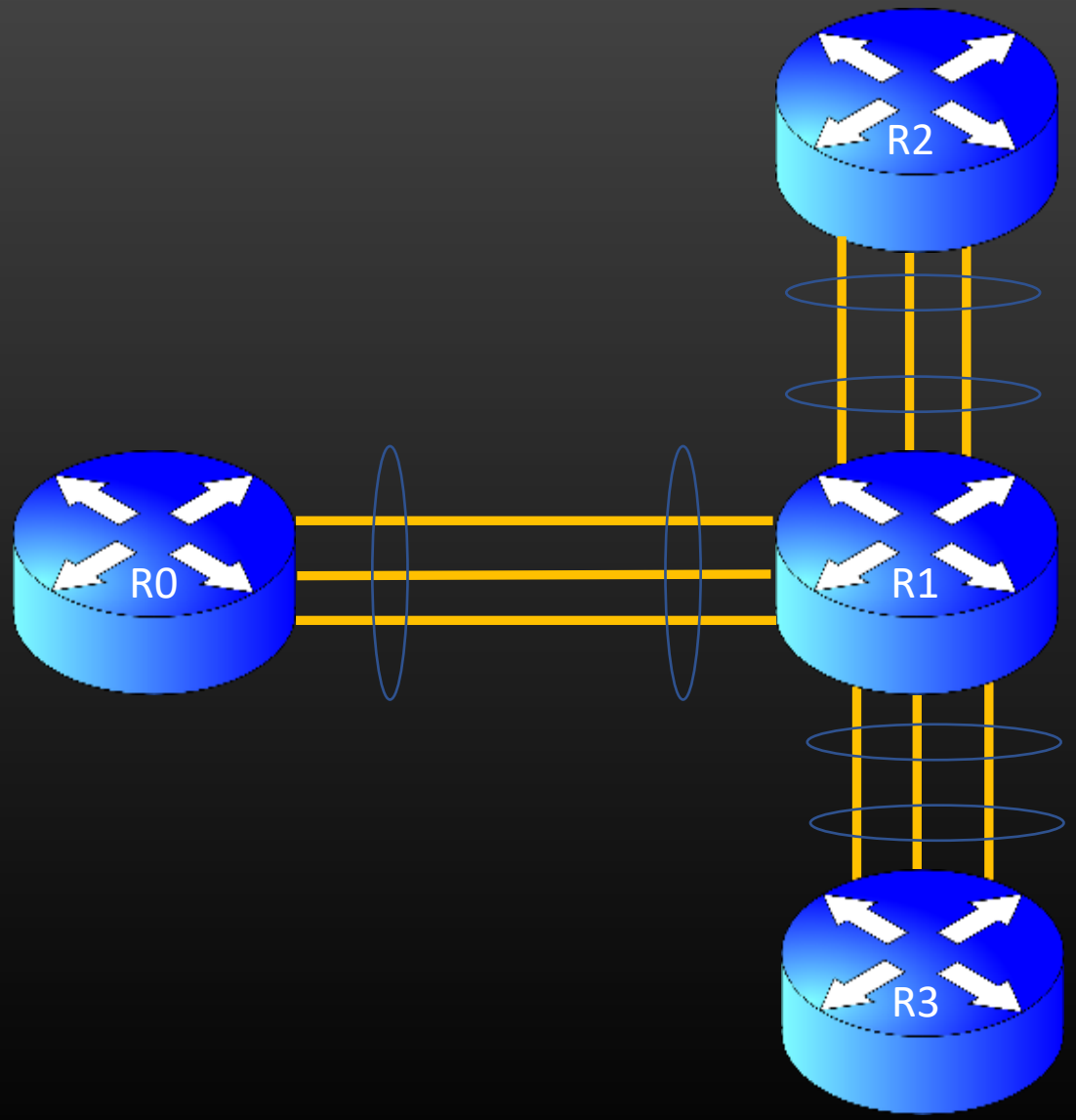
isis_event_non_local_new_lsp
isis_event_non_local_duplicate_lsp

- Flooding LSP on exactly one qualified parallel link is enough
- We can suppress the redundant event *isis_event_non_local_duplicate_lsp* through interface group feature
- LSP should be sent out on exactly one parallel link – whichever is available at the time of LSP disbursement

- All parallel PTP links between two devices are treated as one group called interface groups
- Interface groups is a group of local interfaces which connects to same nbr node
- Protocol sees parallel links as just one link and send out LSP pkt on available member sub-link

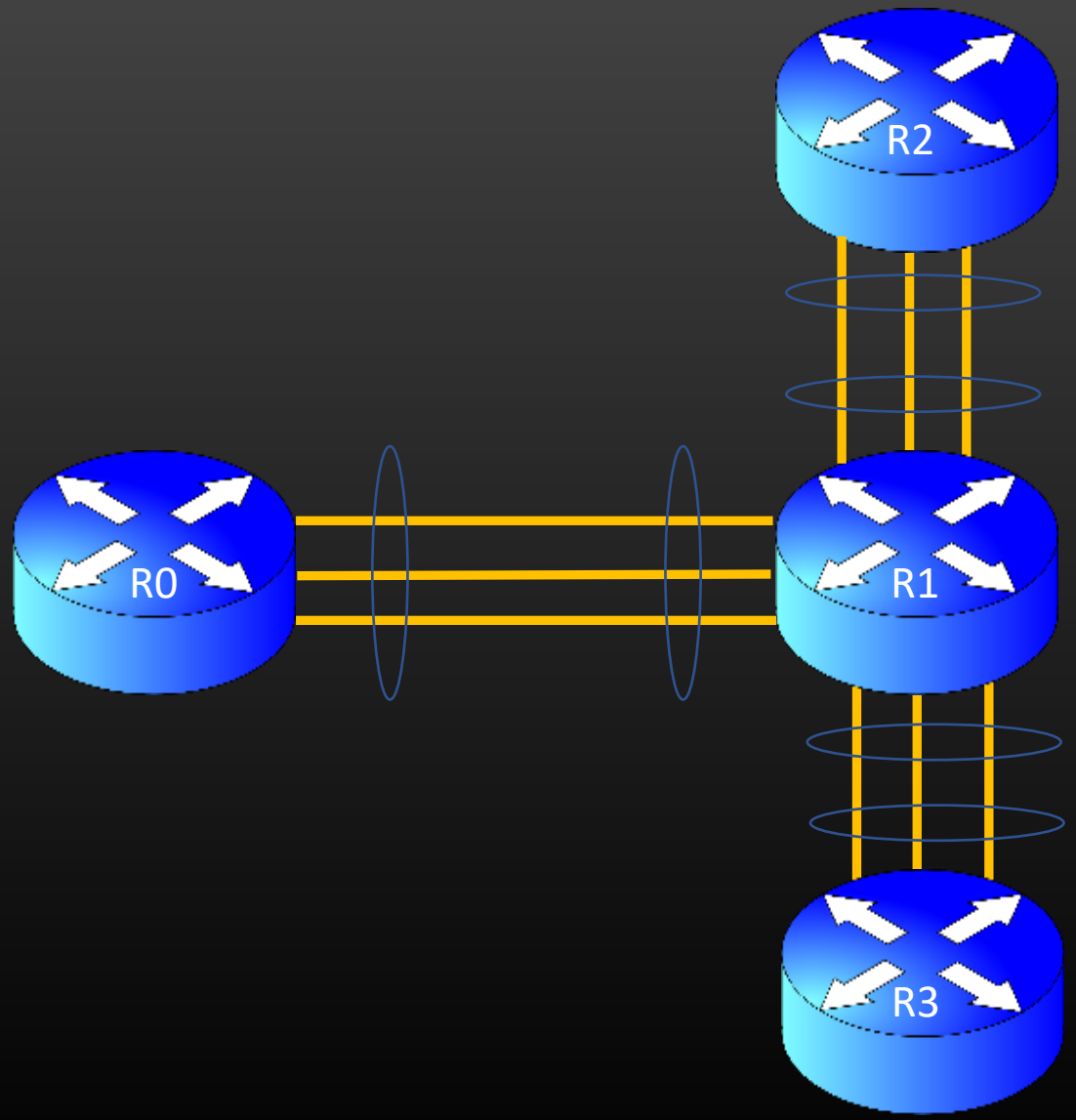
- Can you think of the Data structure / Design / Algorithm to implement this functionality !
- This is how you shall be implementing various new features on existing protocol in the industry.

- LSP flooding with Interface Groups



- LSP pkt recvd on a interface Group is not flooded back on that interface group
- Saves lot of redundant LSP flooding and Network Resources
- A node can have multiple interface Groups
  - Ex, R1 has 3 interface groups
- Interface groups is a group of local interfaces which connects to same nbr node
- How can a given node name multiple interface groups such as interface groups are unique for a given device ?
  > Soln : Using Nbr rtr id as identifier of interface groups

- Interface Groups Design



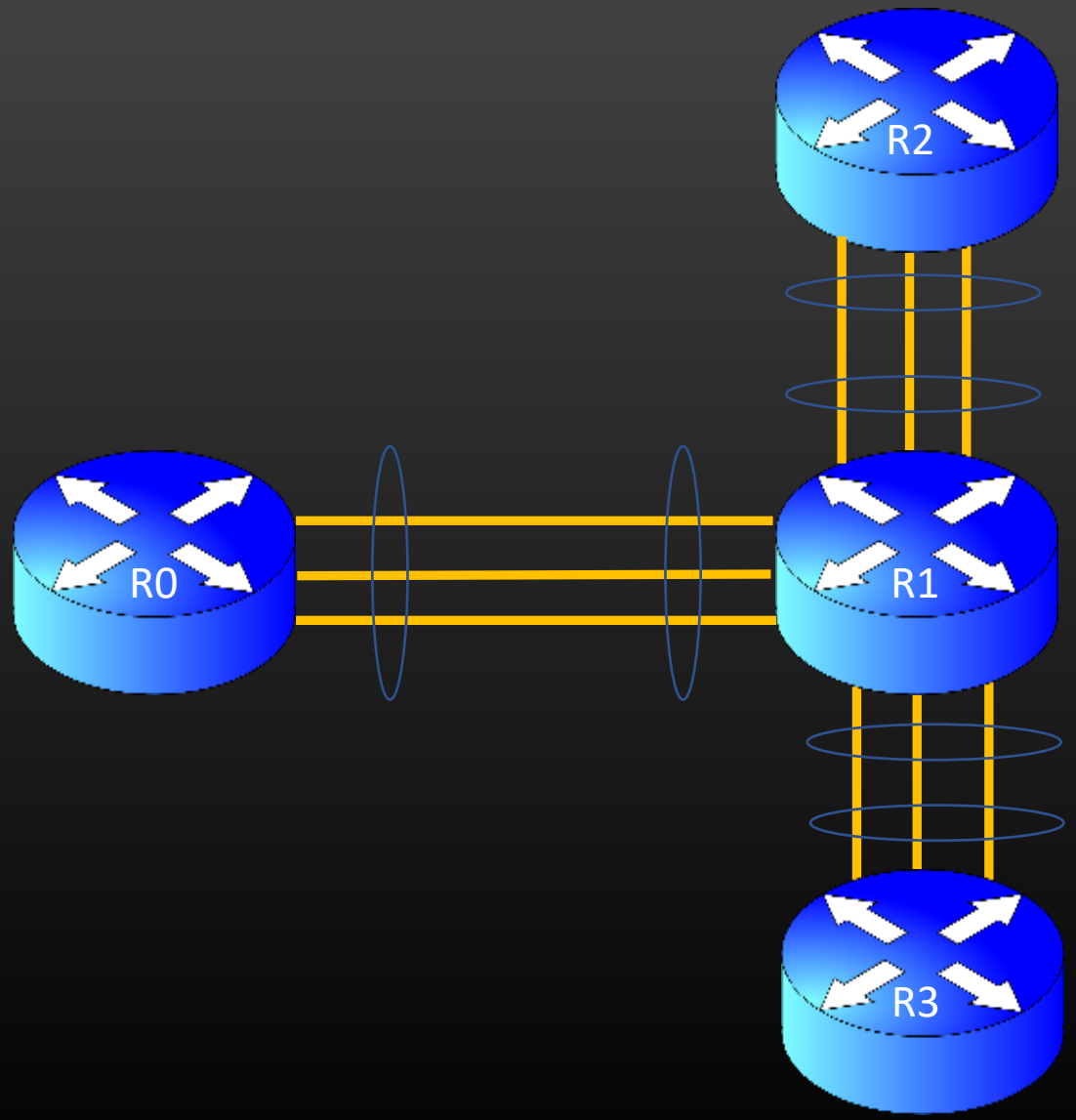
- Every interface Group is identified by nbr rtr id
- Interface Grp is a collection of parallel interfaces
- A Node can have multiple interface grp, therefore maintain a db of interface groups

```
typedef struct isis_intf_group_ {

    char name[ISIS_INTF_GRP_NAME_LEN];  /* key */
    glthread_t intf_list_head;
    avltree_node_t avl_glue;
} isis_intf_group_t;
```

```
typedef struct isis_node_info_ {
. . .
/* Tree of interface Groups */
    avltree_t intf_grp_avl_root;
/* Dynamic intf grp */
    bool dyn_intf_grp;
. . .
} isis_node_info_t;
```

```
typedef struct isis_intf_info_ {
. . .
 /* glue to add to interface group*/
    glthread_t intf_grp_member_glue;
    isis_intf_group_t *intf_grp;
. . .
} isis_node_info_t;
```

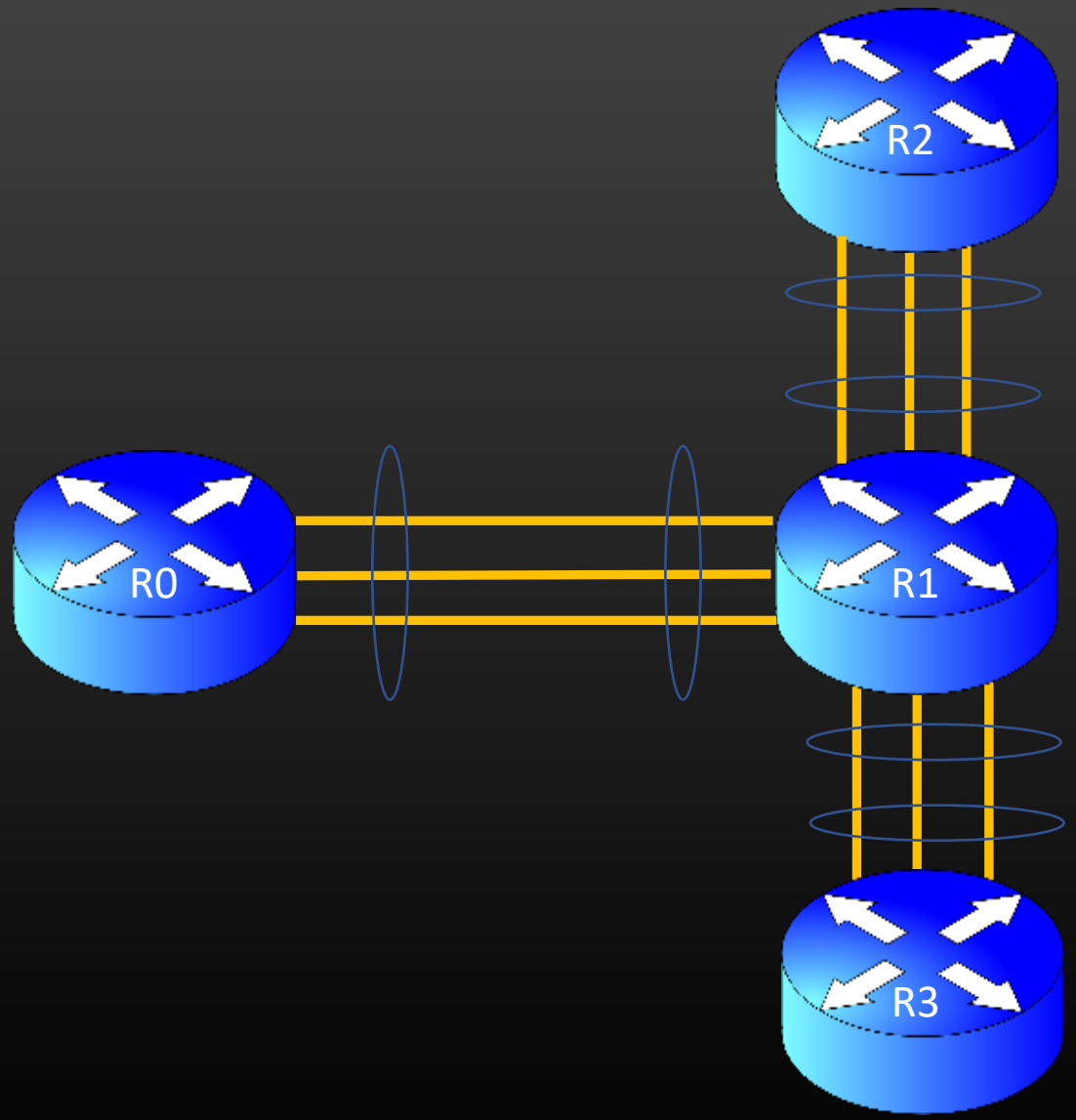
- Interface Groups Design -> Algorithm



- When a node will create a new interface group and add interface
- Steps :
  - Node R1 brings up Adjacency with node R2 on local interface R1-if1
  - Node R1 search in intf grp db if there exist intf_grp with name 2.2.2.2
  - If yes, add R1- if1 in interface grp's intf list
  - If Not, create a new intf grp "2.2.2.2" , add it to intf grp db and, add R1-if1 in interface grp's intf list

- When a node will delete an interface from interface grp
- Steps :
  - Node R1 brings down Adjacency with node R2 on local interface R1-if1
  - Node R1 search in intf grp db if there exist intf_grp with name 2.2.2.2
  - If not found, done
  - If found, remove R1-if1 from intf grp intf list. If intf list is empty, delete interface group also

- Interface Groups Design -> Algorithm

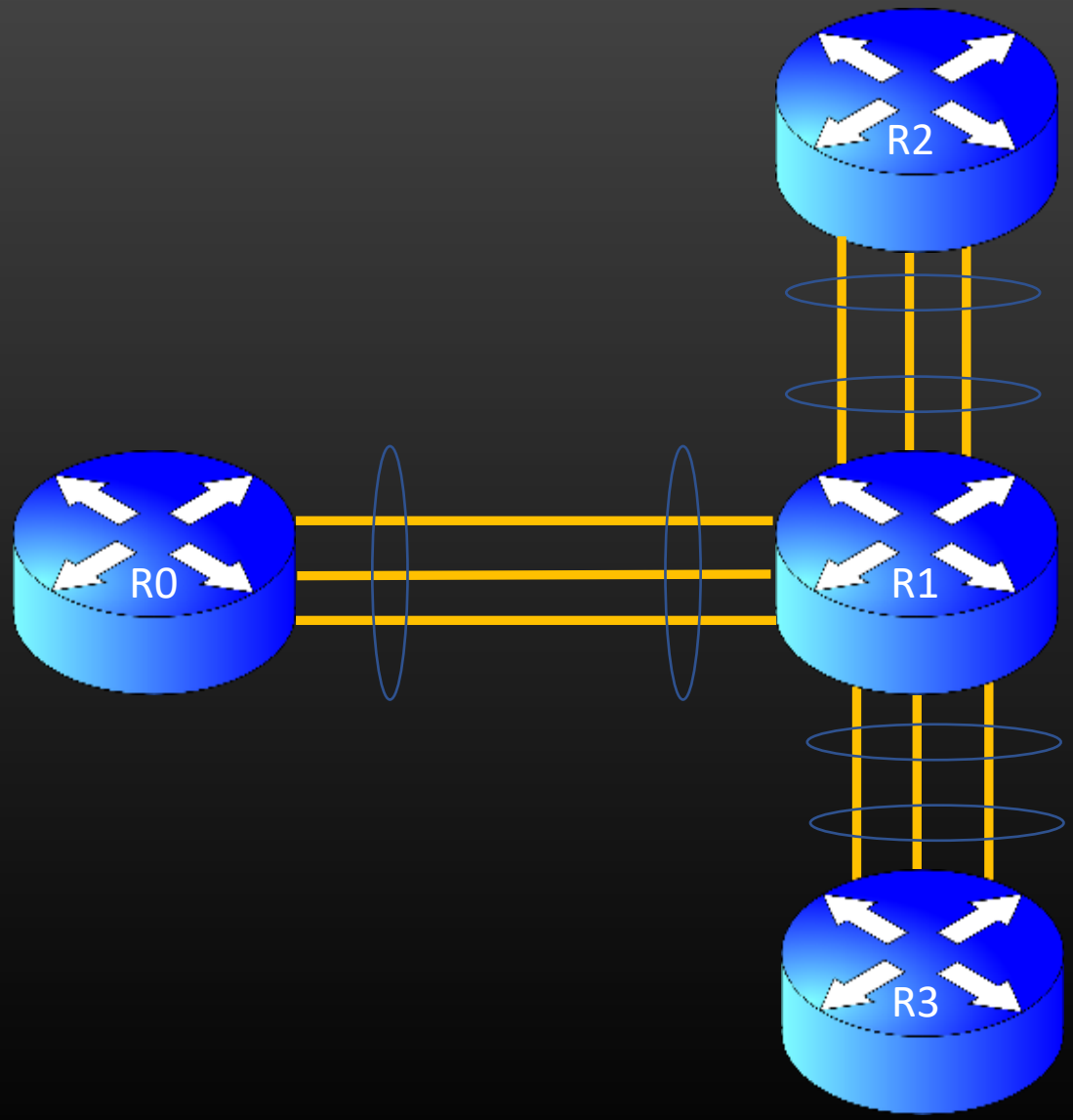


isis_schedule_lsp_flood( ) to be updated !

- How to do controlled LSP flooding
  - Blind Flooding Steps :
    - Iterate over all interface groups
    - Select the Ist interface in interface-grp list
    - Schedule lsp xmit on this interface
  - Forward Flooding Steps:
    - Let *exempt_iif* is the exempted interface (LSP pkt recvd on this interface)
    - Iterate over all interface groups
    - If intf_grp has member *exempt_iif* , skip this intf_grp
    - Select the Ist interface in interface-grp list
    - Schedule lsp xmit on this interface

Note : The lsp flooding behaviour must fallback to Old behaviour if interface group feature is disabled on a node

- CLIs



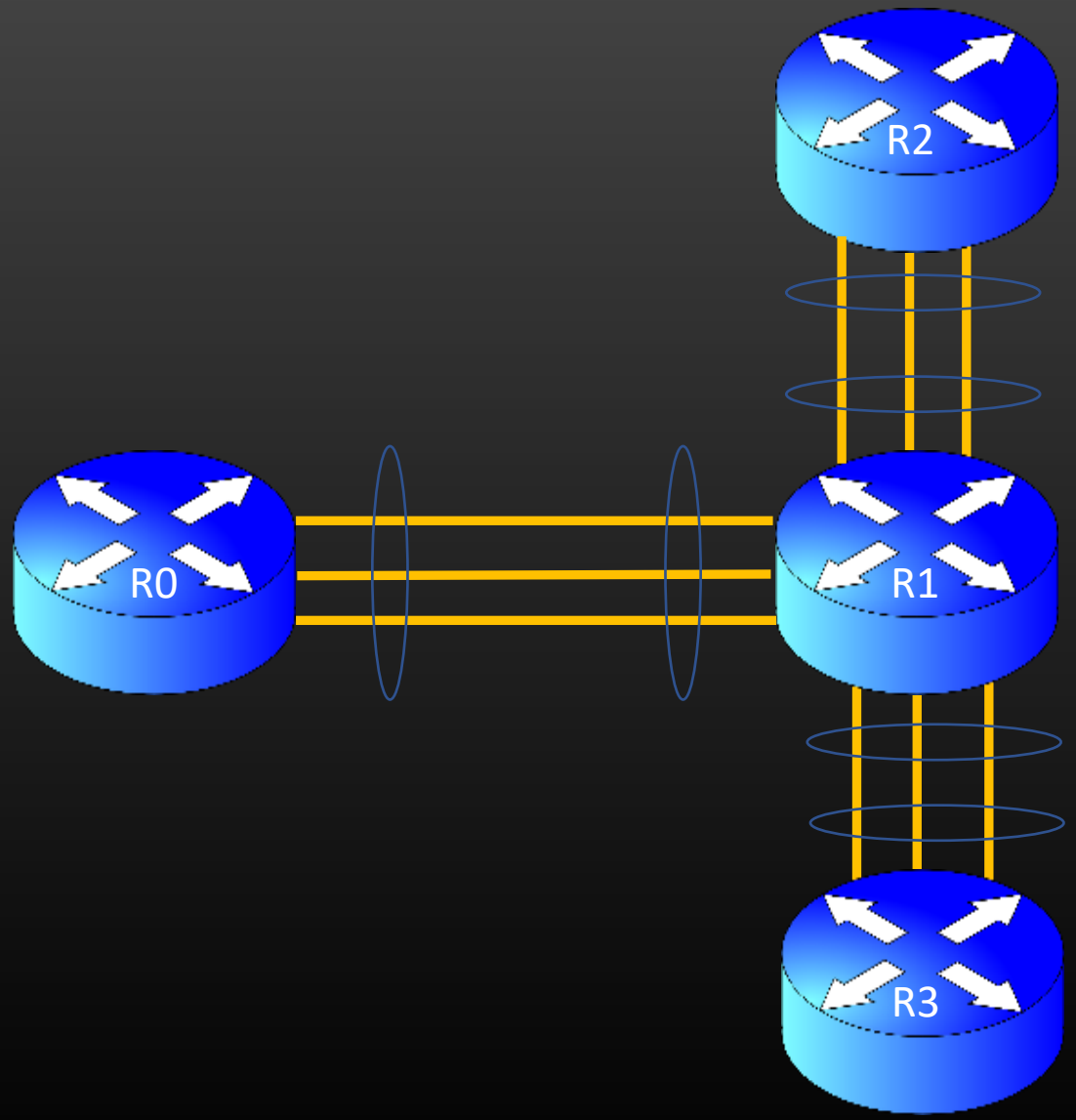
Show :

Show node <node-name> protocol isis interface-groups
< Output attached in Resource Section >

Config :

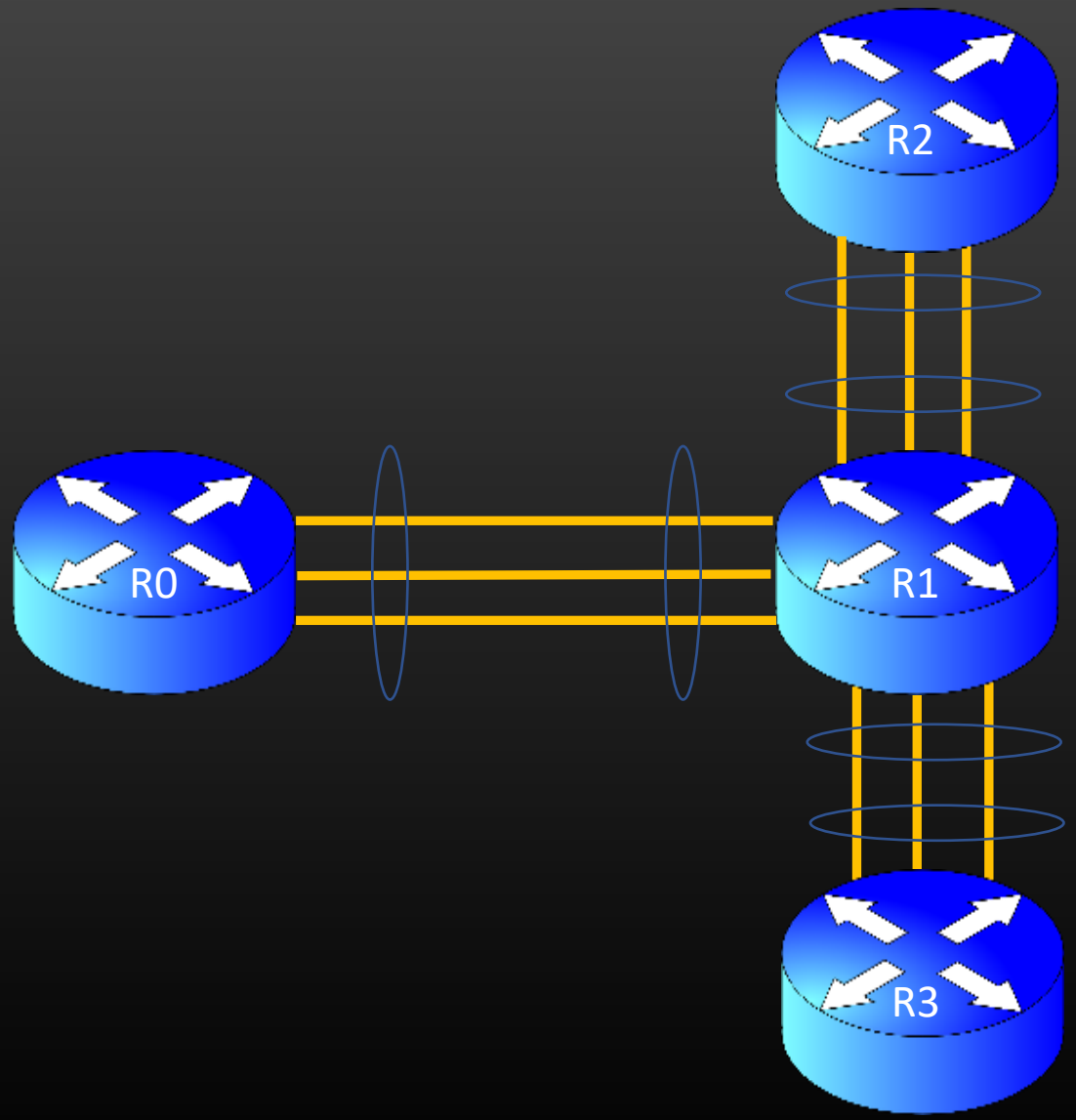
config node <node-name> [no] protocol isis interface-groups
[Enabled by default]

- APIs



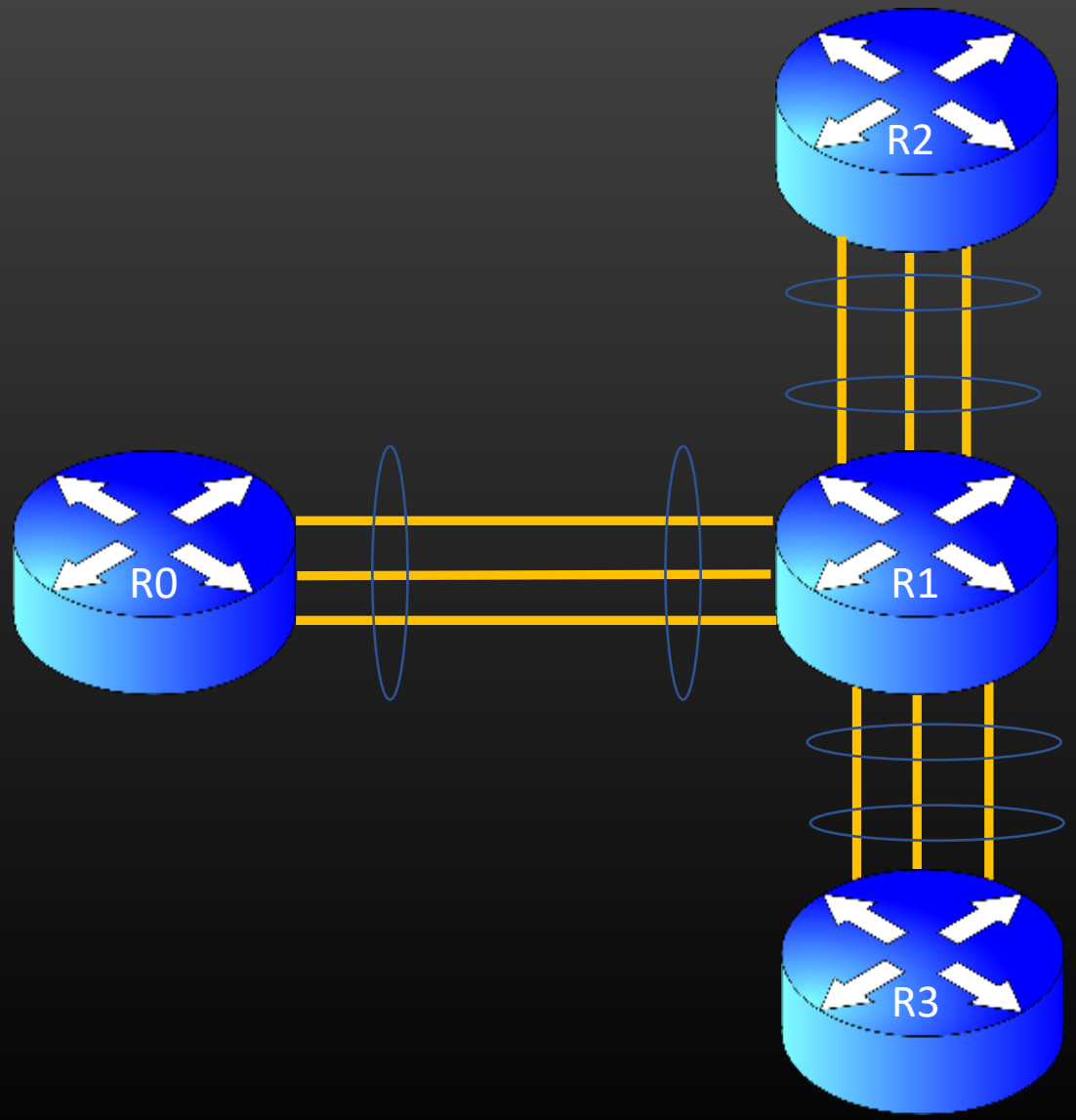

```
int
isis_config_dynamic_intf_grp(node_t *node) ;

int
isis_un_config_dynamic_intf_grp(node_t *node) ;

void
isis_dynamic_intf_grp_update_on_adjacency_up (
                isis_adjacency_t *adjacency);

void
isis_dynamic_intf_grp_update_on_adjacency_down (
                isis_adjacency_t *adjacency);

void
isis_dynamic_intf_grp_build_intf_grp_db(node_t *node);

void
 isis_intf_grp_cleanup(node_t *node) ;

void
isis_init_intf_group_avl_tree(avltree_t *avl_root);

isis_intf_group_t *
isis_intf_grp_look_up(node_t *node, char *intf_grp_name);
```

- APIs




```
bool
isis_intf_group_insert_in_intf_grp_db(node_t *node,
        isis_intf_group_t *intf_grp);

isis_intf_group_t *
isis_intf_group_create_new(char *grp_name);

bool
isis_intf_group_delete_by_name_from_intf_grp_db(
        node_t *node, char *intf_grp_name);

void
isis_intf_group_remove_from_intf_grp_db(
        node_t *node, isis_intf_group_t *intf_grp);
```

- Testing



- Ensure that unnecessary LSP flooding do not happen
- ISIS LSDB is in sync across all topologies
- Use show CLIs to ensure LSP pkts are not being transmitted over redundant parallel links
- clear node <node-name> protocol isis adjacency must also delete all interface grps, but intf-grps will form again as soon as adjacency starts forming ag due to continuous hellos
- Take care of protocol shut-down procedure
  - Delete all interfaces from interface grp intf list
  - Delete interface groups
  - Add assert check in isis_check_and_delete_intf_info( ) to ensure that interface is not Queued up in intg grp interface list
  - Add assert check in isis_check_delete_node_info( ) to ensure that interface grp DB is empty

Hope You Enjoyed this Course !
END of Phase 3