# CSE160: Computer Networks

## Project 1 -- High Level Design

**2020-09-08**

Professor
Alberto E. Cerpa

# Design Choices

- In the following slides I propose a high level design scheme for project 1
- **HOWEVER**, this is not the **ONLY** way you can design project 1
- The bottom line is that whatever it is your design, you should try to work on:
  - Scalability: modular design with clean interfaces → easy to use by other modules
  - Efficiency: reduce the overhead as much as possible (not at the expense of scalability)
- Ultimately, **you** are in control of your code

# Flooding Design

- The cleanest design involves having a module and component just for flooding
- This can be used by other applications whenever they need to use the service (**Hint: project 2!!**)
- This means having your own flooding header information that should include:
  - Source addr of the node initiating the flood
  - Monotonically increasing sequence number to uniquely identify the packet
  - A TTL field to avoid the packet looping forever

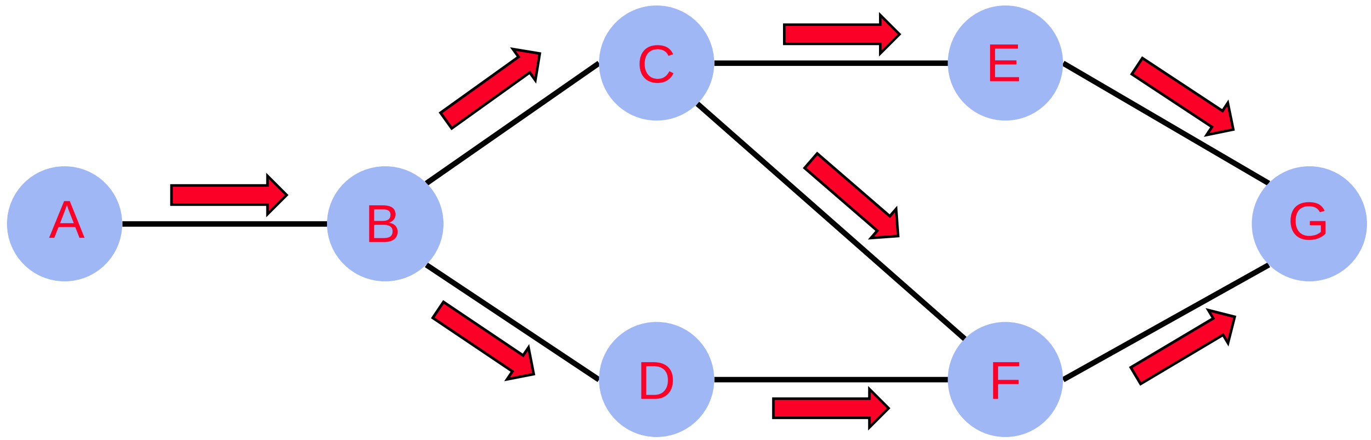
# Flooding Design (2)

- It is important to know from which neighbor you receive a packet
- I would recommend adding a Link Layer module with source and destination addresses
  - This changes hop by hop as the packet get flooded over the network (see slide 9)
- Having a node table (one entry per node) should be useful to implement a cache
  - The cache contains the largest sequence number seen from any node's flood

# Flooding Design (3)

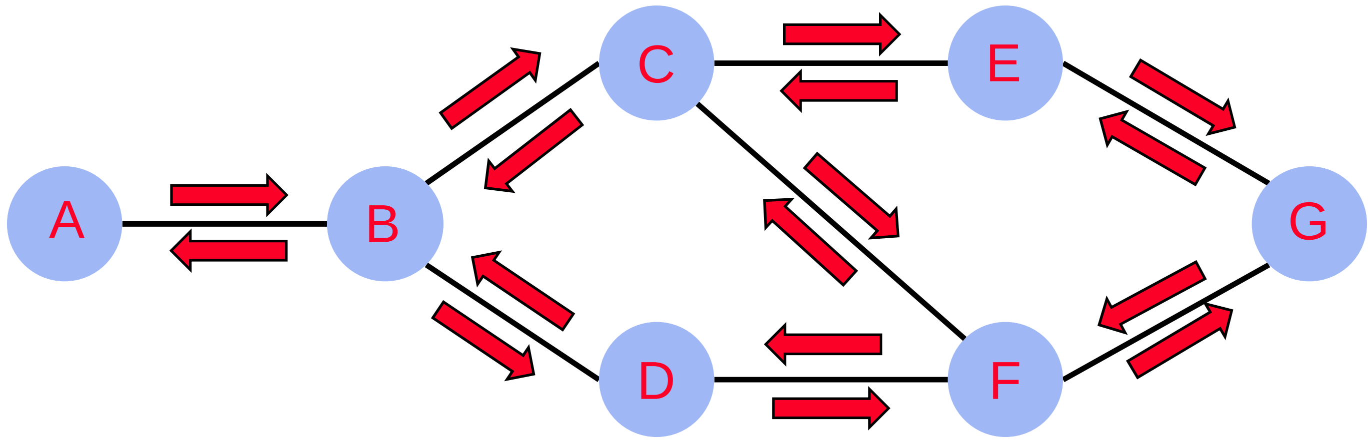
- To access the payload of any packet your header structures should end with a zero length array
  - Please use the trick/method described in slide 27 (Why a zero length array?) of the C Tutorial (Files → Projects → C-Tutorial.pdf)
- Two options for local level addressing
  - Wireless: broadcast medium, so you should take advantage and broadcast to send 1 packet only
  - Wired: you should forward packets to each of your neighbors serially

# Ideal Flooding



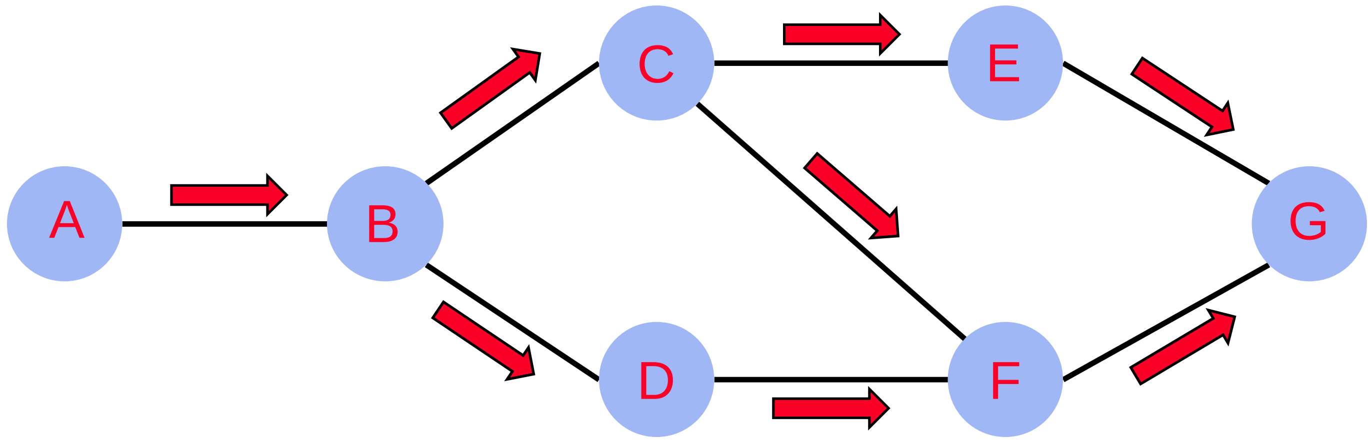

- Only one flood packet per link
- Nodes send packets on each interface to each neighbor (do not broadcast to all interfaces)
- Nodes use the cache to check for duplicates
  - F does NOT send two flood packets to G
  - F does NOT send C's flood packets back to D
  - F does NOT send D's flood packet back to C
  - G does NOT send any flood packet to neither E or F

# Wireless Flooding using Broadcast




- Two flood packet per link (double overhead of ideal case)
- Sending nodes must check the cache
  – All flood packets they sent are being received back
  – Each node must constantly check the cache to avoid sending duplicates

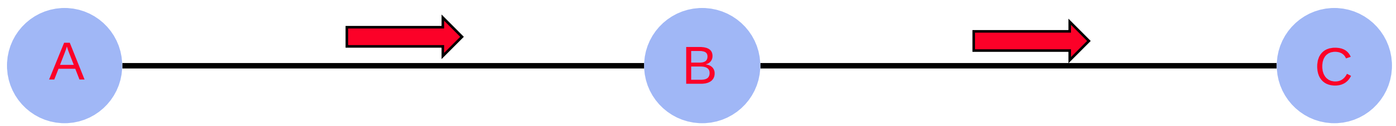
# Wired Flooding




- Only one flood packet per link (like ideal case)
- Nodes send packets on each interface to each individual neighbor (do not broadcast)
- Each node MUST know their neighbors (Neigh. Discovery)
  - Send one flood packet to each neighbor
  - Do NOT send to the neigh you received the flood packet from
    - Use the link layer interface to figure out which neighbor sent the packet
- Nodes use the cache to check for duplicates

# Putting it all together

| | | | |
|---|---|---|---|
| **Application Payload** | Payload | | |
| **Flooding Header** | Flood Source | Sequence Number | Time to Live |
| **Link Layer Header** | Source Address | Destination Address | |




| | | | |
|---|---|---|---|
| **APP** | Hello | | |
| **FH** | A | 120 | 15 |
| **LLH** | A | B | |

If NOT in the cache (check FH source), forward

| | | |
|---|---|---|
| Hello | | |
| A | 120 | 14 |
| B | C | |

- B checks the cache, and forwards the flood message to other neighbors (not A) → implies we know neighbors (Neigh. Discovery)
- Note that the Link Layer header src and dst addresses change hop-by-hop
- Also note that the Flooding header does NOT change, except for the TTL, which is decreased at each hop

# Putting it all together (2)

- Note that C node can figure out the flood's source since it is in the FH
  - This is the index to the Flooding cache (source node address)
- If the Flood sequence number is NOT in the cache (or larger than one in the cache), we do the following:
  - Update the cache with a new value
  - Forward the message
- The Link Layer knows the neighbor that sent the message
  - LL must provide this info when passing the payload to modules
  - We avoid sending the flood message back to node that sent it
- It also knows which other neighbors to send the messages (we assume that Neighbor Discovery is running)
- The logic above slightly changes if you are broadcasting messages at each hop

# Neighbor Discovery Design

- Design involves having a module and component just for neighbor discovery
- This can be used by other applications whenever they need to use the service (**Hint: flooding and project 2!!**)
- This means having your own flooding header information that should include:
  - Request or Reply field
  - Monotonically increasing sequence number to uniquely identify the packet
  - Source address information can be obtained from the link layer

# Neighbor Discovery Design (2)

- You need to send neighbor discovery packets periodically
  - Nodes could die at any time
  - The quality of the links might degrade
  - Use a **timer** and *post a task* to do this periodically
    - What is a good timer to avoid overloading the network?
- Upon reception of a neighbor discovery packet, the receiving node must reply back
- The mechanics are very similar to Ping and Ping Reply, you could copy or reuse the code
- When getting a reply back, the node should gather statistics

# When is a node a neighbor?

- If a neighbor node replies to 1 out of 100 neigh discovery packets, is that node a neighbor?
- If a neighbor node has failed to reply to the last 5 neigh discovery packets, is that node active?
- Each neighbor discovery module should gather statistics about what is the quality of the link to each of its neighbors
  - E.g. responded to 10 out of 20 packets → 50% link
- It should also have thresholds to determine when a neighbor node is no longer responding
  - E.g. failed to respond to the last 5 consecutive packets, neighbor might be dead or link is gone

# How to gather statistics?

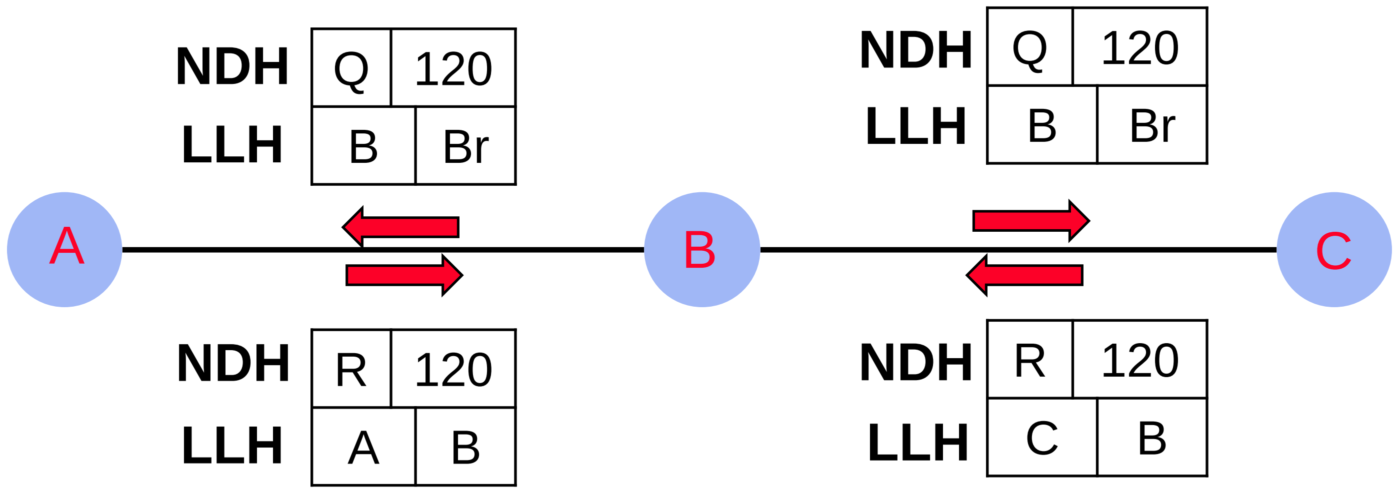
- TinyOS has a statically allocated data model
  - Not possible to dynamically allocate memory
- In this context, the simplest thing to implement is a running average to gather link quality stats
  - We have two variables per neighbor:
    - X: total packet received
    - Y: total packet sent
  - Each neighbor discovery packet header has a monotonically increasing sequence number
  - If a skip in the sequence, it means loss packet(s)
    - E.g.: last seq. received: 116, next seq.: 120 → this means three losses, 117, 118, and 119
    - $X(t) = 59 \rightarrow X(t+1) = 60$; $Y(t) = 116 \rightarrow Y(t+1) = 120$
    - Link quality $(t+1) = X(t+1) / Y(t+1) = 60 / 120 = 50\%$

# Neighbor Table

- You should have a neighbor table with three (3) fields:
  - Neighbor address
  - Quality of the link (percentage)
  - Active neighbor (yes/no)
- You may also remove from table neighbor that are no longer active
  - Based on the previously defined threshold
- Also note that the neighbor table has **VERY** important information for a lot of applications
  - Make your design such that other components can query the content of the neighbor table
  - Routing (project 2!!) will need to access the content of this table

# Putting it all together

| | | |
|---|---|---|
| **Neigh Discovery Header** | reQuest or Reply | Sequence Number |
| **Link Layer Header** | Source Address | Destination Address |




- When timer fires off, B broadcasts (Br) a neighbor Request (Q) discovery packet increasing the last sent sequence number
- Each neighbor node that receives a Request (Q), sends back a Reply (R) immediately
- Upon reception of R messages, B updates the neighbor table with the new neighbor info and statistics