Comprehensive Report on Distance Vector Routing Protocol Implementation with Poisoned Reverse and Split Horizon

Table of Contents

Introduction

Routing protocols are essential for determining optimal paths for data transmission across networks. The **Distance Vector Routing (DVR)** algorithm is a foundational protocol where routers share their routing tables with immediate neighbors, enabling dynamic calculation of shortest paths. However, the basic DVR algorithm can suffer from the **count-to-infinity problem**, leading to routing loops and slow convergence.

To mitigate these issues, techniques like **Poisoned Reverse** and **Split Horizon** are used. This report details the implementation of the DVR algorithm along with these two mechanisms. It provides an in-depth explanation of every function in the code, how they fit into the overall flow, and how they interact with each other to achieve the desired outcomes.

Overall Flow of the Program

The program simulates a network of routers using the Distance Vector Routing algorithm. The overall flow is as follows:

1. **User Input and Initialization:**
    - The program prompts the user for the number of nodes and edges.
    - The user inputs the details of each edge (source, destination, cost).
    - The user selects the method to use (None, Poisoned Reverse, or Split Horizon).
    - Nodes and edges are initialized based on the input.
2. **Initial Distance Vector Computation:**
    - The initial distance vectors and next-hop tables for all nodes are computed.
    - The DVR algorithm runs iteratively until convergence, updating distance vectors based on neighbor information.
3. **Displaying Initial Routing Tables:**
    - The routing tables for each node are displayed after the initial convergence.
4. **Simulating Link Failure:**
    - A specific link between two nodes is removed to simulate a link failure.
    - Neighbor lists and distance vectors are updated accordingly.
5. **Recomputing Distance Vectors After Failure:**
    - The DVR algorithm is re-run to compute new distance vectors post-failure.
    - The program checks for the count-to-infinity problem during updates.
6. **Displaying Updated Routing Tables:**
    - The routing tables for each node are displayed after re-convergence.
7. **Analysis and Conclusion:**
    - The program outputs whether the count-to-infinity problem was detected.
    - Observations are made based on the selected method.

Part 1: Implementing the Distance Vector Routing Algorithm

3.1 Overview

The basic DVR algorithm involves each router maintaining a distance vector, which holds the shortest path distances to all other routers. Routers share their distance vectors with their immediate neighbors, and iteratively update their own distance vectors based on the information received.

3.2 Data Structures

Edge Structure

```
struct Edge {
    int src;    // Source node
    int dest;   // Destination node
    int cost;   // Cost of the edge
};
```

- **Purpose:** Represents a bidirectional link between two nodes with an associated cost.

Node Structure

```
struct Node {
    int id;                             // Node identifier
    map<int, int> distanceVector;       // Destination -> Cost
    map<int, int> nextHop;              // Destination -> Next Hop
    vector<int> neighbors;              // List of neighbor nodes
};
```

- **Components:**
  - `id`: Unique identifier for the node.
  - `distanceVector`: Stores the minimum cost to each destination node.
  - `nextHop`: Stores the next hop node for each destination.
  - `neighbors`: Lists directly connected neighboring nodes.

3.3 Detailed Function Explanations

1. `initializeNodes()`

```
void initializeNodes(vector<Node>& nodes, vector<Edge>& edges, int N);
```

- **Purpose:** Initializes the nodes and their neighbors based on the provided edges.
- **Parameters:**
  - `nodes`: Reference to the vector of nodes.
  - `edges`: Vector of edges representing the network links.
  - `N`: Number of nodes.
- **Process:**
  - Resizes the `nodes` vector to accommodate `N` nodes (1-based indexing).
  - Assigns an `id` to each node.
  - Iterates over each edge to populate the `neighbors` list for each node.
- **Role in Overall Flow:** Sets up the initial network topology by defining nodes and their immediate neighbors, which is crucial for subsequent computations.

2. `initializeDistanceVectors()`

```
void initializeDistanceVectors(vector<Node>& nodes, vector<Edge>& edges, int N);
```

- **Purpose:** Initializes the `distanceVector` and `nextHop` maps for each node.

- **Parameters:**
  - `nodes`: Reference to the vector of nodes.
  - `edges`: Vector of edges.
  - `N`: Number of nodes.
- **Process:**
  - Clears any existing data in `distanceVector` and `nextHop` maps.
  - For each node:
    - Sets the cost to itself (`distanceVector[i][i]`) as `0`.
    - Sets the `nextHop` to itself (`nextHop[i][i]`) as `i`.
    - Initializes the cost to other nodes as `INFINITY`.
    - Sets the `nextHop` to `-1` (unknown) for other nodes.
  - For each edge:
    - Updates the `distanceVector` and `nextHop` for directly connected neighbors based on the edge cost.
- **Role in Overall Flow:** Establishes the initial state of each node's routing table, which is essential for the DVR algorithm to start from a known baseline.

3. `updateDistanceVectors()`

```
bool updateDistanceVectors(vector<Node>& nodes, int N, int method);
```

- **Purpose:** Updates the distance vectors of all nodes based on their neighbors' information.
- **Parameters:**
  - `nodes`: Reference to the vector of nodes.
  - `N`: Number of nodes.
  - `method`: Indicates the method to use (1: None, 2: Poisoned Reverse, 3: Split Horizon).
- **Process:**
  - Initializes a flag `updated` to track if any distance vector changes.
  - For each node `i`:
    - For each neighbor `neighbor` of node `i`:
      - Creates a copy of `nodes[i].distanceVector` as `dvToSend`.
      - **Method-Specific Adjustments:**
        - **Method 1 (None):** No adjustments; the full distance vector is sent.
        - **Method 2 (Poisoned Reverse):** For destinations where `neighbor` is the next hop, set `dvToSend[dest]` to `INFINITY`.
        - **Method 3 (Split Horizon):** Remove destinations from `dvToSend` where `neighbor` is the next hop.
      - **Neighbor Updates:**
        - For each destination `dest` in `dvToSend`:
          - Calculates `newCost = nodes[neighbor].distanceVector[i] + dvToSend[dest]`.

        - If `newCost` is less than `nodes[neighbor].distanceVector[dest]`:
            - Updates `nodes[neighbor].distanceVector[dest]` to `newCost`.
            - Sets `nodes[neighbor].nextHop[dest]` to `i`.
            - Sets `updated` to `true`.

- **Returns:** `true` if any node's distance vector was updated; `false` otherwise.
- **Role in Overall Flow:** This is the core function that iteratively updates the routing tables until convergence, reflecting changes in network topology and path costs.

4. `printRoutingTables()`

```
void printRoutingTables(const vector<Node>& nodes, int N);
```

- **Purpose:** Displays the routing tables (distance vectors and next hops) for each node.
- **Parameters:**
    - `nodes`: Reference to the vector of nodes (const, as it's not modified).
    - `N`: Number of nodes.
- **Process:**
    - For each node `i`:
        - Prints the header for node `i`'s routing table.
        - For each destination `j`:
            - If `nodes[i].distanceVector.at(j)` is greater than or equal to `INFINITY`, prints `INF` and `-`.
            - Else, prints the cost and next hop.
- **Role in Overall Flow:** Provides a visual representation of the current state of each node's routing table, aiding in understanding the network's routing decisions.

5. `checkCountToInfinity()`

```
bool checkCountToInfinity(const vector<Node>& nodes, int N);
```

- **Purpose:** Checks if any node's distance to a destination exceeds a predefined threshold (e.g., 100), indicating the count-to-infinity problem.
- **Parameters:**
    - `nodes`: Reference to the vector of nodes.
    - `N`: Number of nodes.
- **Process:**

- Initializes a flag `countToInfinity` to track if the problem is detected.
- For each node `i`:
  - For each entry in `nodes[i].distanceVector`:
    - If the cost `entry.second` is greater than 100 and less than `INFINITY`:
      - Prints a message indicating the node and destination.
      - Sets `countToInfinity` to `true`.

- **Returns:** `true` if the count-to-infinity problem is detected; `false` otherwise.
- **Role in Overall Flow:** Monitors the algorithm for signs of the count-to-infinity problem, allowing the simulation to detect and report routing issues.

6. `main()` Function

```
int main();
```

- **Purpose:** Entry point of the program; orchestrates the overall flow.
- **Process:**
  - **User Input:**
    - Prompts for the number of nodes (`N`) and edges (`M`).
    - Collects edge information (source, destination, cost).
    - Asks the user to select the method to use (1: None, 2: Poisoned Reverse, 3: Split Horizon).
  - **Initialization:**
    - Creates a vector of `Node` objects.
    - Calls `initializeNodes()` and `initializeDistanceVectors()` to set up the network.
  - **Initial DVR Execution:**
    - Runs `updateDistanceVectors()` in a loop until no updates occur.
    - Prints the routing tables using `printRoutingTables()`.
  - **Simulating Link Failure:**
    - Removes a specified edge to simulate a link failure.
    - Updates neighbor lists for the affected nodes.
    - Re-initializes distance vectors by calling `initializeDistanceVectors()`.
  - **Post-Failure DVR Execution:**
    - Runs `updateDistanceVectors()` again until convergence or the count-to-infinity problem is detected.
    - Uses `checkCountToInfinity()` to monitor for the problem.
    - Prints the routing tables after re-convergence.
- **Role in Overall Flow:** Coordinates the entire simulation, ensuring that each step is executed in the correct order and that user interaction is handled.

3.4 Algorithm Execution Flow

1. **Initialization:**
   - Nodes and edges are initialized based on user input.
   - Distance vectors and next-hop tables are set up.

2. **First DVR Iteration:**
    - Nodes exchange distance vectors with neighbors.
    - Distance vectors are updated based on the Bellman-Ford equation.
    - Iteration continues until no updates occur (network converges).
3. **Displaying Routing Tables:**
    - Routing tables after initial convergence are displayed.
4. **Simulating Link Failure:**
    - A link is removed from the network.
    - Neighbor lists and distance vectors are updated.
5. **Second DVR Iteration:**
    - Nodes re-exchange distance vectors.
    - Updates occur to reflect the new network topology.
    - The count-to-infinity problem is monitored.
6. **Displaying Updated Routing Tables:**
    - Routing tables after re-convergence are displayed.

3.5 Simulating Link Failure

- **Edge Removal:**
    - The edge between Node 4 and Node 5 is removed from the `edges` list.
- **Neighbor Updates:**
    - Node 4 and Node 5 remove each other from their `neighbors` lists.
- **Re-initialization:**
    - Calls `initializeDistanceVectors()` to reset distance vectors.
- **Re-execution:**
    - The DVR algorithm is re-run to observe changes due to the link failure.

3.6 Observations

- **Without Any Mechanism (Method 1):**
    - After the link failure, nodes may enter the count-to-infinity problem.
    - Distances to the unreachable node may increase indefinitely.
    - The simulation detects and reports when the problem occurs.

Part 2: Implementing the Poisoned Reverse Mechanism

4.1 Overview

The Poisoned Reverse technique aims to prevent routing loops by setting the distance to a destination as `INFINITY` when sending updates to a neighbor if that neighbor is the next hop to the destination.

4.2 Modifications to the Algorithm

- **Inclusion of Method Parameter:**
    - The `method` parameter in `updateDistanceVectors()` determines if Poisoned Reverse is applied.

- **Adjustments in `updateDistanceVectors()`:**
  - When sending the distance vector to a neighbor, destinations where the neighbor is the next hop have their distances set to `INFINITY`.

4.3 Detailed Function Explanations

Modified `updateDistanceVectors()` Function

- **Implementation of Poisoned Reverse:**

```
if (method == 2) { // Poisoned Reverse
    for (auto& entry : dvToSend) {
        int dest = entry.first;
        // If neighbor is the next hop to dest, poison the route
        if (nodes[i].nextHop[dest] == neighbor && dest != neighbor) {
            dvToSend[dest] = INFINITY; // Set distance to infinity
        }
    }
}
```

- **Process:**
  - For each destination `dest` in `dvToSend`:
    - Checks if `neighbor` is the next hop to `dest` from node `i`.
    - If so, sets `dvToSend[dest]` to `INFINITY`, effectively poisoning the route when sending to that neighbor.
- **Role in Overall Flow:** Prevents the neighbor from considering node `i` as a route to `dest`, thus avoiding potential routing loops and the count-to-infinity problem.

4.4 Algorithm Execution Flow

- **Initialization and First DVR Iteration:**
  - Same as in Part 1, but with `method` set to `2` (Poisoned Reverse).
- **Link Failure Simulation:**
  - The same link is removed as before.
- **Second DVR Iteration with Poisoned Reverse:**
  - Nodes exchange poisoned distance vectors.
  - Updates occur, but routes that could cause loops are poisoned.
- **Monitoring for Count-to-Infinity:**
  - The `checkCountToInfinity()` function verifies that the problem does not occur.

4.5 Simulation Results

- **After Implementing Poisoned Reverse:**
  - Nodes do not enter the count-to-infinity problem.
  - Routing tables converge quickly.
  - Unreachable destinations are correctly marked with `INFINITY`.

4.6 Analysis

- **Effectiveness:**

    - Poisoned Reverse successfully prevents routing loops and the count-to-infinity problem.
- **Trade-offs:**
    - Increases the size of updates slightly due to the inclusion of poisoned routes.
- **Suitability:**
    - Effective in networks where the overhead of poisoned updates is acceptable.

Part 3: Implementing the Split Horizon Mechanism

5.1 Overview

The Split Horizon technique prevents a router from advertising a route back to the neighbor from which it was learned, reducing the chance of routing loops.

5.2 Modifications to the Algorithm

- **Adjustments in `updateDistanceVectors()`:**
    - Routes learned via a neighbor are removed from the updates sent to that neighbor.

5.3 Detailed Function Explanations

Modified `updateDistanceVectors()` Function

- **Implementation of Split Horizon:**

```
else if (method == 3) { // Split Horizon
    for (auto it = dvToSend.begin(); it != dvToSend.end();) {
        int dest = it->first;
        // If neighbor is the next hop to dest, remove the route
        if (nodes[i].nextHop[dest] == neighbor && dest != neighbor) {
            it = dvToSend.erase(it); // Remove entry and update iterator
        } else {
            ++it; // Move to the next entry
        }
    }
}
```

- **Process:**
    - Iterates over `dvToSend` using an iterator.
    - If the neighbor is the next hop to a destination, removes that entry from `dvToSend`.
    - Ensures that routes are not advertised back to the neighbor from which they were learned.
- **Role in Overall Flow:** Prevents routing information from being sent back in the direction it came from, effectively avoiding routing loops.

5.4 Algorithm Execution Flow

- **Initialization and First DVR Iteration:**

- Similar to previous parts, but with `method` set to `3` (Split Horizon).

- **Link Failure Simulation:**
  - The same link is removed.
- **Second DVR Iteration with Split Horizon:**
  - Nodes exchange distance vectors with certain routes omitted.
  - Updates occur without propagating routes back to their sources.
- **Monitoring for Count-to-Infinity:**
  - The algorithm ensures that the count-to-infinity problem does not occur.

5.5 Debugging and Fixing the Segmentation Fault

- **Issue Encountered:**
  - A segmentation fault occurred due to improper modification of the `dvToSend` map during iteration.
- **Cause:**
  - Erasing elements from a map while iterating invalidates the iterator.
- **Solution:**
  - Use an iterator (`it`) and update it properly after erasure:
    - When erasing, set `it = dvToSend.erase(it);`.
    - Else, increment `it` normally.
- **Result:**
  - The segmentation fault was resolved, and the Split Horizon mechanism functioned correctly.

5.6 Simulation Results

- **After Implementing Split Horizon:**
  - Routing tables converge without entering the count-to-infinity problem.
  - Unreachable destinations are correctly identified.
  - The network adapts efficiently to the link failure.

5.7 Analysis

- **Effectiveness:**
  - Split Horizon effectively prevents routing loops and the count-to-infinity problem.
- **Advantages:**
  - Reduces the size of routing updates by omitting certain routes.
- **Considerations:**
  - May be preferred in networks where minimizing update sizes is important.

Conclusion

The implementation and simulation of the Distance Vector Routing algorithm, along with the Poisoned Reverse and Split Horizon mechanisms, demonstrate the following key points:

1. **Basic DVR Algorithm Limitations:**
   - Susceptible to the count-to-infinity problem.
   - Can result in routing loops and slow convergence after topology changes.

2. **Poisoned Reverse Mechanism:**
    - Effectively prevents routing loops by poisoning routes when sending updates to specific neighbors.
    - Ensures rapid convergence without incorrect route propagation.
3. **Split Horizon Mechanism:**
    - Also prevents routing loops by not advertising routes back to the neighbor from which they were learned.
    - May reduce overhead by sending smaller update messages.
4. **Implementation Challenges:**
    - Modifying data structures during iteration requires careful handling to prevent runtime errors.
    - Proper initialization and clearing of data structures are essential for accurate simulations.
5. **Overall Flow Integration:**
    - Each function plays a critical role in the simulation.
    - Functions are designed to work together seamlessly, enabling the simulation to accurately reflect network behavior under different conditions.
6. **Simulation Insights:**
    - Both mechanisms significantly improve the reliability and efficiency of the DVR protocol.
    - The choice between them depends on specific network characteristics and performance requirements.

References

1. **Computer Networking Textbooks:**
    - Kurose, J. F., & Ross, K. W. *Computer Networking: A Top-Down Approach*. Pearson.
    - Stallings, W. *Data and Computer Communications*. Prentice Hall.
2. **Networking Protocols Documentation:**
    - RFC 1058: *Routing Information Protocol*.
    - RFC 2453: *RIP Version 2*.
3. **Algorithm Resources:**
    - Explanations of the Bellman-Ford Algorithm.
    - Tutorials on routing algorithms and network simulations.

Usage Instructions:

1. **Compile the Code:**

```
g++ -o dvr_simulation dvr_simulation.cpp
```

2. **Run the Executable:**

```
./dvr_simulation
```

3. **Input Data:**

    - Enter the number of nodes (`N`) and edges (`M`).
    - Input the edges with source, destination, and cost.
    - Choose the method:
        - `1` : None (Basic DVR)
        - `2` : Poisoned Reverse
        - `3` : Split Horizon

4. **Observe the Output:**

    - The program displays the routing tables before and after the link failure.
    - It indicates whether the count-to-infinity problem occurs based on the selected method.

Final Remarks:

Understanding and implementing these routing protocols and mechanisms is crucial for efficient network design and operation. The detailed explanations and code annotations aim to provide clarity on how each function contributes to the overall program, and how they interact to achieve reliable routing in dynamic network environments.

Feel free to experiment with different network configurations and observe how the Poisoned Reverse and Split Horizon techniques affect the network's behavior.
