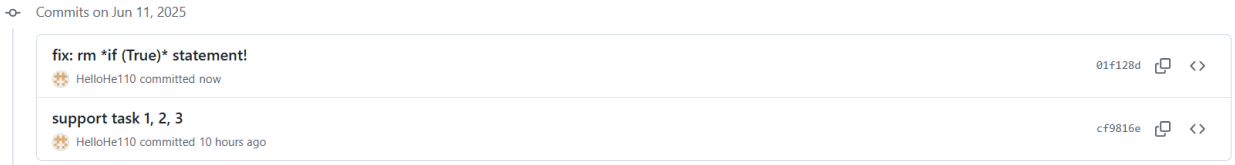
Lab6 – report **111550022 施采緹**

<commit history>

In the first commit, I uploaded the student ID to the repo.



In the second commit, I uploaded the workable version of task1~3. And the following is how I implemented these tasks. And the third commit only modify the redundant code without changing the program logic!

Task1:

For task 1-1, I added $string\ m\_pathFile$ as private variable and $void\ setPathFile(string)$ as public function in "nix-vector-helper". And the implementation follows the step TA provided in spec! *{add var, add func, func: store pathname, call func in Copy constructor}*

```
107 +   // Task 1-1:
108 +   void SetPathFile (std::string pathFile);
109 +
110 + private:
111 +   // Task 1-1:
112 +   std::string m_pathFile;
```

$nix - vector - helper.h$

```
32 + // Task 1-1:
33 + template <typename T>
34 + void NixVectorHelper<T>::SetPathFile (std::string pathFile) {
35 +   m_pathFile = pathFile;
36 + }
```

$nix - vector - helper.cc$

```
54 + template <typename T>
55 + NixVectorHelper<T>*
56 + NixVectorHelper<T>::Copy (void) const
57 + {
58 +   // Task 1-1:
59 +   auto helper = new NixVectorHelper<T> (*this);
60 +   helper->SetPathFile (this->m_pathFile);
61 +   return helper;
62 + }
```

$nix - vector - helper.cc$

Also in task1-1, I modify “nix-vector-routing”! And what I’d done is also the same as what being mentioned in the spec: add variable – Table, add function SetPaths to read path line by line from file and store it in Table, call SetPaths in Create function after creating NixVectorRouting object.

```
137 +   // Task 1-1:
138 +   void SetPaths (std::string pathFile);
139 +
140 + private:
141 +   // Task 1-1:
142 +   std::map<std::pair<int, int>, std::vector<int>> Table;
```

$nix - vector - routing.h$

```
54  + // Task 1-1:
55  + template <typename T>
56  + void NixVectorRouting<T>::SetPaths (std::string pathFile) {
57  +   std::ifstream file(pathFile);
58  +   if (!file.is_open()) {
59  +     NS_LOG_ERROR("Failed to open path file: " << pathFile);
60  +     return;
61  +   }
62  +
63  +   Table.clear();
64  +
65  +   std::string line;
66  +   // std::cout << "Reading path file: " << pathFile << std::endl;
67  +   while (std::getline(file, line)) {
68  +     std::istringstream iss(line);
69  +     int src, dst, path_len;
70  +     iss >> src >> dst >> path_len;
71  +     std::vector<int> path(path_len);
72  +     for (int i = 0; i < path_len; i++) {
73  +       iss >> path[i];
74  +     }
75  +     Table[{src, dst}] = path;
76  +   }
77  +   file.close();
78  + }
```

$nix - vector - routing.cc$

```
64  + template <typename T>
65  + Ptr<typename NixVectorHelper<T>::IpRoutingProtocol>
66  + NixVectorHelper<T>::Create (Ptr<Node> node) const
67  + {
68  +   Ptr<NixVectorRouting<IpRoutingProtocol>> agent = m_agentFactory.Create<NixVectorRouting<IpRoutingProtocol>> ();
69  +   agent->SetNode (node);
70  +   node->AggregateObject (agent);
71  +   // Task 1-1:
72  +   agent->SetPaths (m_pathFile);
73  +   return agent;
74  + }
```

$nix - vector - helper.cc$

Let’s move on to task1-2! In

```
206 + template <typename T>
207 + Ptr<NixVector>
208 + NixVectorRouting<T>::GetNixVector (Ptr<Node> source, IpAddress dest, Ptr<NetDevice> oif) const
209 + {
```

in file “nix-vector-routing.cc”, it calls BFS originally. Therefore, to use the path in the path file, I ran the following self-defined parent vector instead of the BFS approach!
In the code, I double checked whether this source-destination pair is in Table or not. If yes, then I constructed the parentVector based on the path in Table. And then I called BuildNixVector with this parentVector to return the nixVector!

```
219 +   // ============================== Task 1-2 START ==============================
220 +   std::vector< Ptr<Node> > parentVector;
221 +   auto it = Table.find({source->GetId(), destNode->GetId()});
222 +   if (it != Table.end())
223 +   {
224 +       std::vector<int> path = it->second;
225 +       std::vector<Ptr<Node>> parentVector(NodeList::GetNNodes(), Ptr<Node>(0));
226 +       for (size_t i = 1; i < path.size(); ++i)
227 +       {
228 +           uint32_t parentId = path[i - 1];
229 +           uint32_t childId = path[i];
230 +           parentVector[childId] = NodeList::GetNode(parentId);
231 +       }
232 +       // std::cout << "Using path: ";
233 +       // for (int nid : path) std::cout << nid << " ";
234 +       // std::cout << std::endl;
235 +
236 +       if (BuildNixVector(parentVector, source->GetId(), destNode->GetId(), nixVector))
237 +       {
238 +           return nixVector;
239 +       }
240 +       else
241 +       {
242 +           NS_LOG_ERROR("Path found in table but BuildNixVector failed for " << source->GetId() << " -> " << destNode->GetId());
243 +           return 0;
244 +       }
245 +   }
246 +   // ============================== Task 1-2 END ==============================
```


In task 1-3, the goal is to disable the cache! Therefore, l let function - $GetNixVectorInCache$ and $GetIpRouteInCache$ in “nix-vector-routing.cc” return false and 0 directly as follows:

```
289 + // Task 1-3:
290 + template <typename T>
291 + Ptr<NixVector>
292 + NixVectorRouting<T>::GetNixVectorInCache (const IpAddress &address, bool &foundInCache) const
293 + {
294 +   // NS_LOG_FUNCTION (this << address);
295 +
296 +   // CheckCacheStateAndFlush ();
297 +
298 +   // typename NixMap_t::iterator iter = m_nixCache.find (address);
299 +   // if (iter != m_nixCache.end ())
300 +   //   {
301 +   //     NS_LOG_LOGIC ("Found Nix-vector in cache.");
302 +   //     foundInCache = true;
303 +   //     return iter->second;
304 +   //   }
305 +
306 +   // not in cache
307 +   foundInCache = false;
308 +   return 0;
309 + }
```

```
311 + // Task 1-3:
312 + template <typename T>
313 + Ptr<typename NixVectorRouting<T>::IpRoute>
314 + NixVectorRouting<T>::GetIpRouteInCache (IpAddress address)
315 + {
316 +   // NS_LOG_FUNCTION (this << address);
317 +
318 +   // CheckCacheStateAndFlush ();
319 +
320 +   // typename IpRouteMap_t::iterator iter = m_ipRouteCache.find (address);
321 +   // if (iter != m_ipRouteCache.end ())
322 +   //   {
323 +   //     NS_LOG_LOGIC ("Found IpRoute in cache.");
324 +   //     return iter->second;
325 +   //   }
326 +
327 +   // not in cache
328 +   return 0;
329 + }
```

Task2, 3:

In task2 and task3, we can move back to file – "leo-lab6.cc". And I'll describe them together! The duration is set to 100 `20 + double duration = 100;` .

And the send packet function is modified:

```
23    - void SendPacket(Ptr<Node> src, Ptr<Node> dst);
24    + void SendPacket(int srcId, int dstId, std::vector<Ptr<PacketSink>>& sinks);
```

The modification is to support task3 – throughput calculation!

In *int main*(), I add some if-else conditions check to direct task2 and task3 for supporting different workflow! Task2 for only send for 36 to 38, and Task3 for sending 3 pkt and calculate the total throughput!

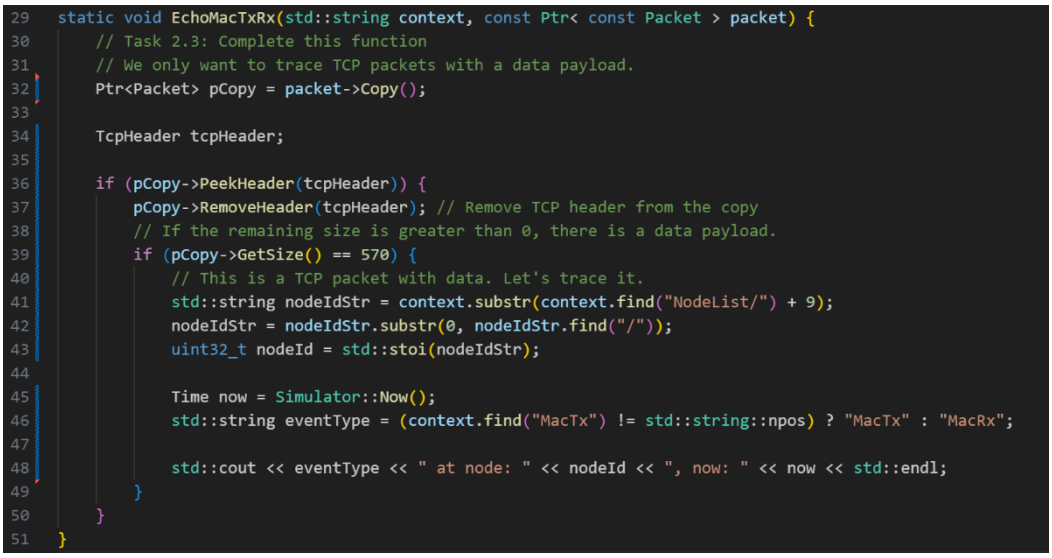
```
120  174        // Task 2.2 & Task 3.2 : Call SendPacket()
     175  +     std::vector<Ptr<PacketSink>> sinks;
     176  +     if (Task == 2) {
     177  +         SendPacket(36, 38, sinks);
     178  +     }
     179  +     else if (Task == 3) {
     180  +         SendPacket(36, 38, sinks);
     181  +         SendPacket(37, 40, sinks);
     182  +         SendPacket(39, 41, sinks);
     183  +     }
121  184
122       -     // Task 3.3 : Install sink applications for each destination
     185  +     // Task 3.3 : Install sink applications for each destination is handled in SendPacket
132  195        // Task 3.4 : Calculate throughput
     196  +     if (Task == 3) {
     197  +         double totalBytes = 0;
     198  +
     199  +         // Throughput for 36 -> 38
     200  +         double throughput1 = sinks[0]->GetTotalRx();
     201  +         std::cout << "36->38: " << throughput1 << std::endl;
     202  +         totalBytes += throughput1;
     203  +
     204  +         // Throughput for 37 -> 40
     205  +         double throughput2 = sinks[1]->GetTotalRx();
     206  +         std::cout << "37->40: " << throughput2 << std::endl;
     207  +         totalBytes += throughput2;
     208  +
     209  +         // Throughput for 39 -> 41
     210  +         double throughput3 = sinks[2]->GetTotalRx();
     211  +         std::cout << "39->41: " << throughput3 << std::endl;
     212  +         totalBytes += throughput3;
     213  +
     214  +         std::cout << "Total throughput: " << totalBytes << std::endl;
     215  +     }
```

And let's move on to the ***send packet function***:

```
33        - void SendPacket(int srcId, int dstId) {
      65  + void SendPacket(int srcId, int dstId, std::vector<Ptr<PacketSink>>& sinks) {
34    66        // Task 2.1: Complete this function
35    67        // Task 2.1: Set MaxBytes to 512 & Task 3.1: Set MaxBytes to 0
      68  +     Ptr<Node> srcNode = NodeList::GetNode(srcId);
      69  +     Ptr<Node> dstNode = NodeList::GetNode(dstId);
      70  +
      71  +     // Install PacketSink on destination
      72  +     PacketSinkHelper sinkHelper("ns3::TcpSocketFactory", InetSocketAddress(Ipv4Address::GetAny(), port));
      73  +     ApplicationContainer sinkApp = sinkHelper.Install(dstNode);
      74  +     sinkApp.Start(Seconds(0.0));
      75  +     sinkApp.Stop(Seconds(duration));
      76  +     // Keep track of sink for throughput calc
      77  +     sinks.push_back(StaticCast<PacketSink>(sinkApp.Get(0)));
      78  +
      79  +     // Resolve IPv4 address of destination
      80  +     Ptr<Ipv4> ipv4 = dstNode->GetObject<Ipv4>();
      81  +     Ipv4Address dstAddr = ipv4->GetAddress(1, 0).GetLocal();
      82  +
      83  +     // Configure BulkSend application
      84  +     BulkSendHelper sourceHelper("ns3::TcpSocketFactory", InetSocketAddress(dstAddr, port));
      85  +     uint32_t maxBytes = (Task == 2 ? 512 : 0);
      86  +     sourceHelper.SetAttribute("MaxBytes", UintegerValue(maxBytes));
      87  +     ApplicationContainer sourceApp = sourceHelper.Install(srcNode);
      88  +     sourceApp.Start(Seconds(0.0));
      89  +     sourceApp.Stop(Seconds(duration));
36    90    }
```

The SendPacket function sets up a TCP communication between a source node and a destination node in the simulation. It installs a PacketSink application on the destination node to receive data, and a BulkSendHelper on the source node to send data using TCP. The destination's IP address is resolved for proper socket setup. For **Task 2**, the source sends a single 512-byte packet (MaxBytes = 512), while for **Task 3**, it sends continuously (MaxBytes = 0). The function also stores the sink in a vector for later throughput calculation in Task 3.

And in ***Echo Tx Rx function***:



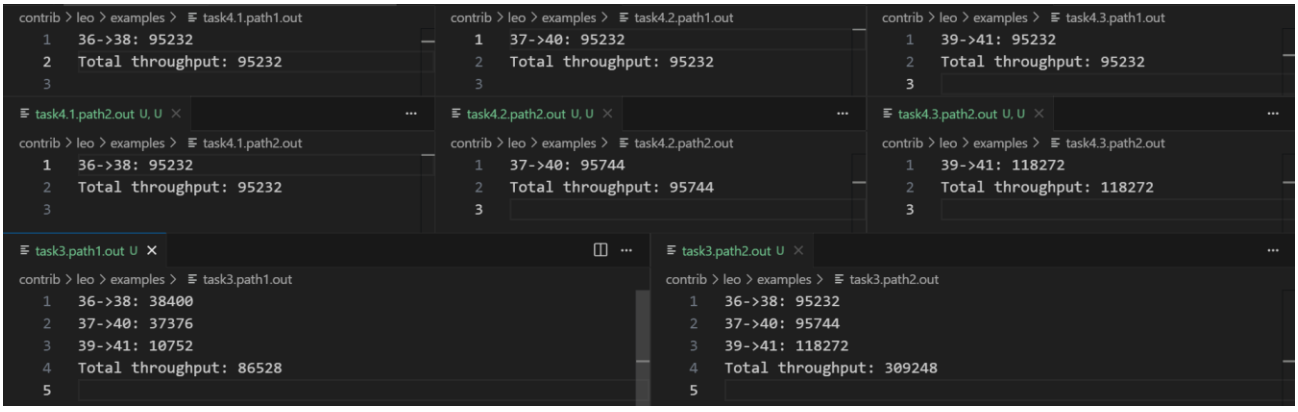
```
static void EchoMacTxRx(std::string context, const Ptr< const Packet > packet) {
    // Task 2.3: Complete this function
    // We only want to trace TCP packets with a data payload.
    Ptr<Packet> pCopy = packet->Copy();

    TcpHeader tcpHeader;

    if (pCopy->PeekHeader(tcpHeader)) {
        pCopy->RemoveHeader(tcpHeader); // Remove TCP header from the copy
        // If the remaining size is greater than 0, there is a data payload.
        if (pCopy->GetSize() == 570) {
            // This is a TCP packet with data. Let's trace it.
            std::string nodeIdStr = context.substr(context.find("NodeList/") + 9);
            nodeIdStr = nodeIdStr.substr(0, nodeIdStr.find("/"));
            uint32_t nodeId = std::stoi(nodeIdStr);

            Time now = Simulator::Now();
            std::string eventType = (context.find("MacTx") != std::string::npos) ? "MacTx" : "MacRx";

            std::cout << eventType << " at node: " << nodeId << ", now: " << now << std::endl;
        }
    }
}
```

The EchoMacTxRx function is a callback used to trace MAC layer transmission and reception events for TCP packets with a data payload. When a packet is sent or received at the MAC layer, this function checks whether it contains a TCP header and whether the payload size is 570 bytes (indicating actual data, not just headers or ACKs). If so, it extracts the node ID from the context string, determines whether the event is a transmission (MacTx) or reception (MacRx), and prints the event type, node ID, and current simulation time. This is primarily used in **Task 2** to verify the actual routing path by tracing the packet as it travels through the network.

And the third commit will be uploading the report!

<Questions>

1. **Explain how parentVector in nix-vector-routing describe a path**
   The parentVector in nix-vector-routing is a node-based data structure that encodes the path from the source to the destination as a reversed parent-child relationship for each hop along the route. Specifically, for each node on the path, it stores a pointer to its parent (i.e., the previous node in the path), such that parentVector[child] = parent. This structure is then used by the BuildNixVector() function to reconstruct the full path from the destination back to the source, allowing the NixVector to be built correctly. It effectively captures the routing path in a tree-like form that enables efficient encoding of the forwarding decisions into the NixVector bit sequence.
2. **Explain why you get different total throughputs for paths1.in and paths2.in? Does congestion occurs in paths1.in and paths2.in?**
   Because in **paths1.in** all three TCP flows (36→38, 37→40, 39→41) funnel through the same subset of satellites—most notably node 35 and its inter-satellite links—those shared links become a bottleneck at the MAC layer. Since each link has limited capacity (11 kbps on the UT channel and whatever the ISL rate is), the flows contend, queue, and experience collisions or backoffs, driving down each flow's effective throughput (and hence the total). In contrast, **paths2.in** assigns each flow to largely disjoint hops (different intermediate satellites and links), so they don't compete for the same resources. As a result, you observe minimal contention and much higher per-flow and aggregate throughput in paths2. Thus, yes—congestion is significant in paths1 but largely absent in paths2.
3. **Please provide more experiments to clarify your answer in Q2. (Hint: Try to transmit each SD pair separately)**



By comparing each flow's throughput when run alone (task4) against when all three run together (task3), we see a stark difference between the two path sets:

- **paths1.in (shared-hop)**

    - *Solo runs* (task4.1–4.3): each SD pair achieves 95 232 B.
    - *Combined* (task3): 36→38 drops to 38 400 B, 37→40 to 37 376 B, and 39→41 to 10 752 B (total 86 528 B).
      This shows that when they compete for the same intermediate satellites (notably node 35), they severely throttle one another—none attains even half its solo rate.
- **paths2.in (disjoint-hop)**
    - *Solo runs* match the *combined* results almost exactly (36→38: 95 232 B, 37→40: 95 744 B, 39→41: 118 272 B; total 309 248 B).
      Here, since each flow travels over different links, there's no contention and each reaches its full capacity both alone and together.

  Conclusion: the solo-vs-combined comparison confirms that **paths1.in** suffers significant congestion on its shared ISLs, whereas **paths2.in** does not.