Commit History: 111550022 施采緹

<first commit>

Commits on May 2, 2025

support task1~4

HelloHe110 committed yesterday 962d3ad

In the first commit, I uploaded 5 files to support task1 to task4. So that talks about what I've done in each task one by one. In task1, I copied the `network.graph` to the workspace of lab3 for getting the transmitting schedule based on "or-tools" and "greedy" approaches. Therefore, after running both method with the new `network.graph`, I can get 2 file, which are `network.ortools.out & network.greeedy.out`.

```
132  int main(int argc, char *argv[]) {
173      ParseLinkRates(graphFile);
174      ParseAssociation(inputFile);
```

```
void ParseLinkRates(string filename) {
  ifstream in(ns3_path_head + filename);
  int numGs, numSat, dummy;
  in >> numGs >> numSat >> dummy;

  int gs, sat, rate;
  while (in >> gs >> sat >> rate) {
    // printf("gs: %d, sat: %d, rate: %d\n", gs, sat, rate);
    linkRates[make_pair(gs, sat)] = to_string(rate) + "kbps";
  }
}
```

```
void ParseAssociation(string filename) {
  printf("============================================\n");
  ifstream in(ns3_path_head + filename);
  double dummy;
  in >> dummy;
  int gs, sat, cnt = 0;
  while (in >> gs >> sat) {
    if (cnt >= 20) break;
    cnt++;
    satToGsQueue[sat].push_back(gs);
    assocMap[sat].push_back(gs);
    gsStarted[gs] = false;
    printf("gs: %d, sat: %d\n", gs, sat);
  }
  printf("============================================\n");
}
```

Now, let's move to task2.1 – `SendPacket(int gsId, int satId)`. The first thing to do is to retrieve the NS-3 Node objects corresponding to the GS and SAT. And I also setup a counter `sat_cnt` for each SAT in order to trace how many packet has it processed (will be useful when calling `GetTotalRx()` which record the overall data that SAT has collected. After that, as the requirement, the data rate is set up as the value provided in `network.graph`, which is stored in map – linkRates.

```
utNet.Get(gsNode->GetId())->GetObject<MockNetDevice>()->SetDataRate(linkRates[make_pair(gsId, satId)]);
utNet.Get(satNode->GetId())->GetObject<MockNetDevice>()->SetDataRate(linkRates[make_pair(gsId, satId)]);
```

And we can see that the transmission should use TCP protocol, set MaxBytes to 125000 (Bytes) and set SendSize to 512 (Bytes) according to the spec.

```
Ipv4Address dstAddr = satNode->GetObject<Ipv4>()->GetAddress(1, 0).GetLocal();

BulkSendHelper source("ns3::TcpSocketFactory", InetSocketAddress(dstAddr, port));
source.SetAttribute("MaxBytes", UintegerValue(125000));
source.SetAttribute("SendSize", UintegerValue(512));
```

After that, we can install and immediately start the application (i.e., the packet sender) on the ground station. And I also record some information for further condition testing and final output like gsStartTime records the transmission start time for the GS. gsStarted record a Boolean flag about whether this GS start the transmission. As the SendPacket() function being called, the gsStarted will be set to true. And satBusy which represents whether the SAT is busy or not is set as true.

```
ApplicationContainer app = source.Install(gsNode);
app.Start(Seconds(Simulator::Now().GetSeconds()));

gsStartTime[gsId] = Simulator::Now().GetSeconds();
gsStarted[gsId] = true;
satBusy[satId] = true;
```

Now, let's move on to task2.2 - Call SendPacket() in main(). This part is responsible for scheduling the next available GS-to-satellite transmission when a satellite becomes available. It's a core part of the simulation's loop to manage packet transmissions based on satellite availability and queue order.

```
map<int, vector<int>> satToGsQueue;
```

```
satToGsQueue[sat].push_back(gs);
```

```
for (auto &[sat, q] : satToGsQueue) {
  // auto it = min_element(q.begin(), q.end());
  if (!q.empty() && !gsStarted[q.front()]) {
    // Simulator::Schedule(Seconds(0), &SendPacket, q.front(), sat);
    SendPacket(q.front(), sat);
    // printf("Sending packet from gs%d to sat%d\n", *it, sat);
  }
}
```

The key is to check if (1) the SAT's queue isn't empty. (2) the GS at the front of the queue (i.e., the next one to send data) hasn't started sending yet. This prevents duplicate transmission and enforces the queue order.

The following is Task3 - EchoRx().

```
int nodeId = stoi(GetNodeId(context));
if (nodeId >= (int)satellites.GetN()) return;
```

Extracts the satellite node ID from the simulation context string. And if the ID is out of range (i.e., not a satellite), the function exits.

```
Ptr<PacketSink> sink = DynamicCast<PacketSink>(satellites.Get(nodeId)->GetApplication(0));
if (!sink) return;
```

Grabs the TCP sink application on the satellite. If it's not found (e.g., wrong type or index), exit early.

```
uint64_t totalRx = sink->GetTotalRx();
if (totalRx < (uint64_t)(125000 * (sat_cnt[nodeId]))) return;
```

Checks how many bytes the satellite has received so far. If the satellite hasn't yet finished receiving all data from the current GS (125000 bytes per transmission × completed GS transmissions so far), return early.

```
for (int gsId : assocMap[nodeId]) {
  if (gsEndTime.find(gsId) == gsEndTime.end() && gsStartTime.count(gsId)) {
    gsEndTime[gsId] = Simulator::Now().GetSeconds();
    satFinishTime[nodeId] = gsEndTime[gsId];
    satCollectTime[nodeId] += gsEndTime[gsId] - gsStartTime[gsId];
    printf("nodeId: %s from %s\n", GetNodeId(context).c_str(), context.c_str());
    printf("Satellite %d finished receiving data at time: %f\n", nodeId, satFinishTime[nodeId]);
    break;
  }
}
```

Looks through all GSs associated with this satellite. Finds the **first GS that has started but not yet finished transmission**. Records the GS's **reception end time**, satellite's **latest finish time**, and **adds this transmission duration to the satellite's collection time**. Exits loop after processing just one GS.

```
for (int gsId : assocMap[nodeId]) {
  if (gsStartTime.find(gsId) == gsStartTime.end()) {
    Simulator::ScheduleNow(&SendPacket, gsId, nodeId);
    break;
  }
}
```

Scans through associated GSs again. Finds the next GS that hasn't started, and schedules it to begin transmission immediately via SendPacket.

Finally, it's time to print the result out – task4!

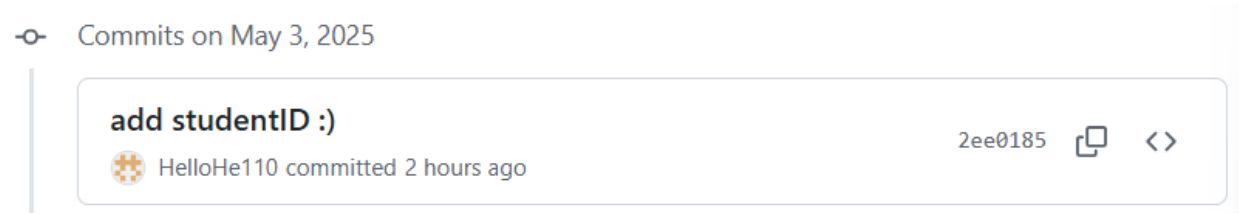
```cpp
ofstream out(ns3_path_head + outputFile);
double totalTime = 0;
for (auto &[_, t] : gsEndTime) totalTime = max(totalTime, t);
out << totalTime << endl;
for (int i = 0; i < 60; ++i) {
  if (satCollectTime.count(i)) {
    out << i << " " << satCollectTime[i] << endl;
  } else {
    out << i << " 0" << endl;
  }
}
for (int gs = 0; gs < 20; ++gs) {
  if (gsStartTime.count(gs) && gsEndTime.count(gs)) {
    out << gs << " " << gsStartTime[gs] << " " << gsEndTime[gs] << endl;
  } else {
    out << gs << " 0 0" << endl;
  }
}
```

Scans all ground station transmission end times and takes the **latest one** as the **total simulation time**. Writes it as the first line of the output. For each of the 60 satellites (indexed 0–59): If the satellite collected data, write: satellite_id total_collection_time. If not, write: satellite_id 0.

For each ground station (0–19): If it has both a start and end time: gs_id start_time end_time; Otherwise, write: gs_id 0 0.

And I could get the 2 output files: `lab4.greedy.out & lab4.ortools.out.`

<second commit>



In the second commit, I add the studentID file containing my studentID to fit the spec's requirements.

<third commit>

The third commit is about 2 things, the first is to push the report into the repo, and the other is much more important!

```
91    ApplicationContainer app = source.Install(gsNode);
92    // app.Start(Seconds(Simulator::Now().GetSeconds()));
93    app.Start(Seconds(0));
```

I realized that the if I run the original (commented) code, it means that it will delay “The now second” and start instead of start the simulation simultaneously. Therefore, I modify the code to let app start at second(0), which means it will start immediately!
Due to the modification of the lab4-leo.cc code, the result file of both ortools and greedy will also be modified!

Questions:

1. **Compare lab4.greedy.out and lab4.ortools.out.**
   About total completion time:
   The **ORTools-based scheduler finishes ~17× faster** than the greedy scheduler. This suggests that ORTools found a far more efficient way to schedule transmissions.
   About pre-SAT collection time:
   Below is a satellite-wise comparison (Satellite ID from 0 to 9):

SAT	Greedy Time (s)	ORTools Time (s)	Faster in
0	0	0.26545	Greedy
1	0.108083	0.278296	Greedy
2	0.220336	0.254399	Greedy
3	0.452024	0.235131	ORTools
4	0.228448	0.230025	Greedy
5	0.337939	0.225653	ORTools
6	0.220952	0.220952	same
7	0	0.246872	Greedy
8	0.682102	0.222106	ORTools
9	0	0.237353	Greedy

   We can see that Greedy leaves SAT 0, 7, 9 unused, and offloads most traffic to SAT 8 (0.682102s). On the other hand, ORTools balances load well across all 10 satellites, keeping all collection times under 0.3s.
   About transmission schedule per GS:
   A few examples of GS scheduling:

GS ID	Greedy Start-End (s)	ORTools Start-End (s)
0	0 → 0.108083	0 → 0.113262
1	0 → 0.107541	0.113262 → 0.26545
2	0.107541 → 0.220336	0 → 0.118675
3	0 → 0.11425	0.118675 → 0.278296
...	...	...

Observations:

- **Greedy sometimes serializes many GS transmissions onto a single satellite.**
- **ORTools parallelizes more effectively, allowing overlapping transfers.**

Feature	Greedy Approach	ORTools Approach
Total Completion Time	0.682102s	0.278296s (✓ much faster)
Load Balancing	Poor (e.g., SAT 8 overloaded)	Excellent (all SATs used effectively)
Parallelism	Limited, mostly sequential per SAT	High, overlapping GS transmissions
Utilization of SATs	7/10 SATs used	All 10 SATs used
Scheduling Heuristics	Likely first-fit or static matching	Global optimization via MILP (ORTools)

2. **Explain why the collection time will not be the same as the solutions in network.xxx.out?**

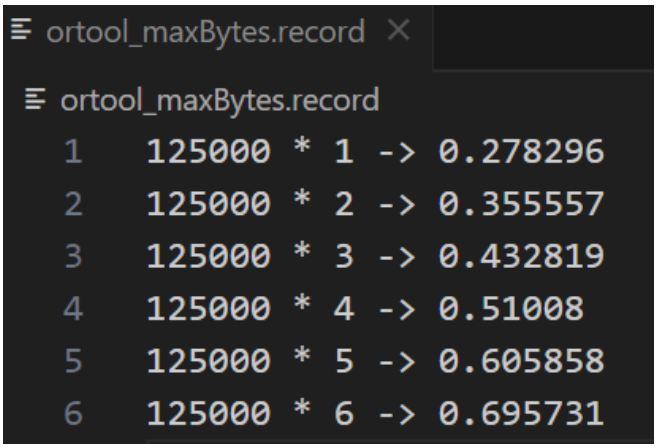
   In practice, the simulation's measured collection time (0.06705 s for OR-Tools, 0.19384 s for Greedy) will always exceed the idealized values computed by lab3's solver (network.ortools.out: 0.0670526 s; network.greedy.out: 0.193835 s) because the solver's outputs assume a perfect, zero-overhead transfer model. In contrast, the ns-3 simulation actually emulates TCP/IP protocol behavior, wireless/channel effects, and event scheduling. Here are the main sources of extra time:

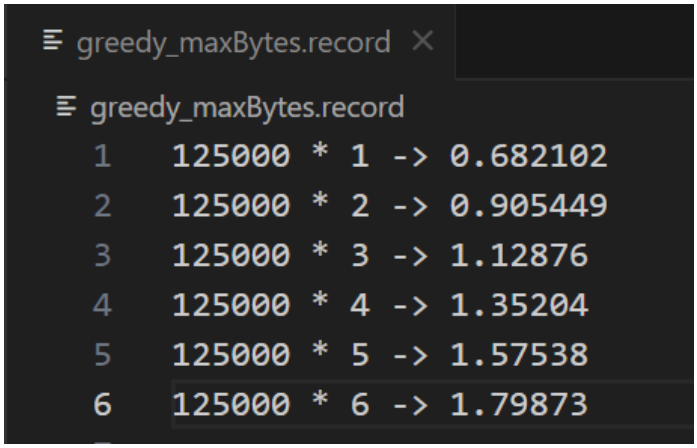
   - **TCP Handshake & Slow-Start:** Before any data begins flowing, each BulkSendHelper socket performs a three-way handshake (SYN, SYN-ACK, ACK), which adds at least one round-trip time. The TCP slow-start algorithm ramps up the congestion window gradually (starting with one segment), so

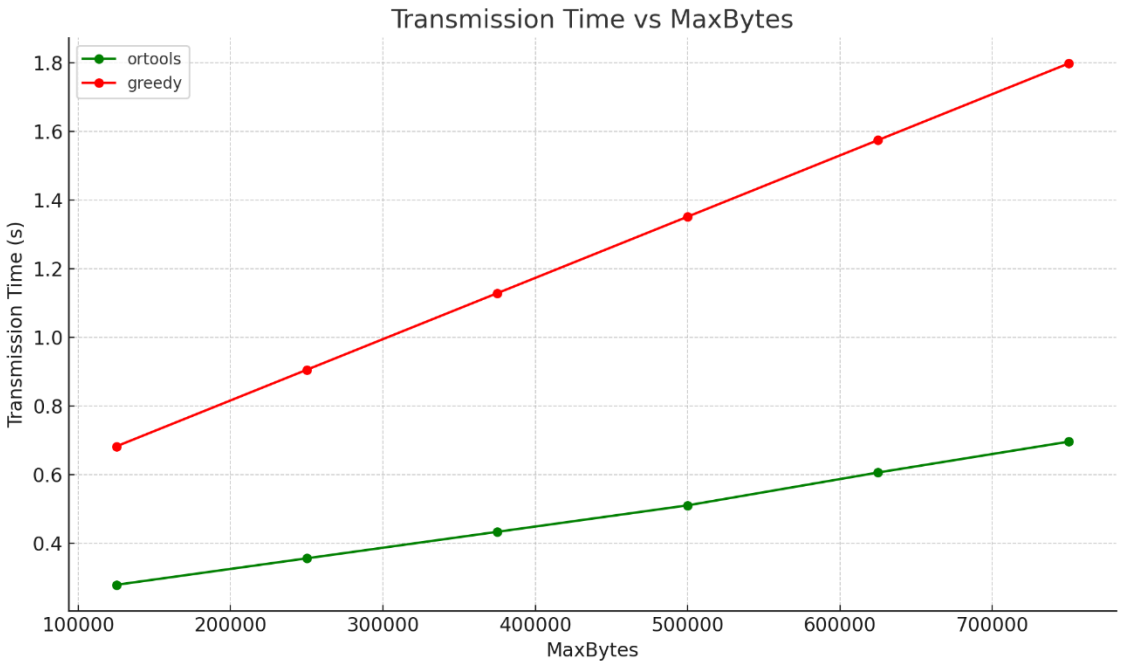
the first few packets are spaced out by the initial congestion window growth rather than being sent all at once.

- **Protocol Overhead:** Every TCP segment carries a 20-byte header and every IP packet adds another 20 bytes, plus link-layer framing. Those extra bytes consume airtime, slightly reducing the effective payload rate below the nominal "rate = payload/time" assumed by the solver. ACK packets sent by the satellite also consume channel time and can introduce small scheduling gaps.
- **ns-3 Event Scheduling Granularity:** In a discrete-event simulator, each packet transmission, reception, and callback (e.g., your EchoRx) is scheduled as a separate event. The processing and queuing of these events takes non-zero simulation time and can serialize actions that the solver treated as truly simultaneous.

3. **Explain the meaning of MaxBytes and SendSize in BulkSendHelper.**
   MaxBytes is UintegerValue. The meaning of it is the total number of bytes the application will attempt to send over the life of the connection.
   Behavior: If you set it to, say, 125000, the app will send exactly 125 000 bytes (in multiple chunks) and then gracefully close the socket. If you set MaxBytes = 0, BulkSendApplication interprets that as "send **unlimited** data" until the peer closes or the simulation ends.
   SendSize is also an UintegerValue. The meaning of it is the size (in bytes) of each individual write issued on the TCP socket.
   Behavior: If you set it to 512, the application will chunk your total payload into 512-byte writes. Internally, each SendSize-byte chunk becomes one TCP segment (subject to the TCP segment size you've configured). Thus, a larger SendSize means fewer, larger writes (and fewer protocol-level segments), while a smaller SendSize means more frequent writes and more overhead per byte.
4. **Adjust the value of MaxBytes and observe the changes in transmission time.**







1. Transmission Time Growth

ortools:

- Grows **linearly but with small increments**.
- Each step increases by approximately **~0.08–0.09 seconds**.
- Example:
  - From 1× to 2×: 0.355557 − 0.278296 = **0.077261 s**
  - From 2× to 3×: 0.432819 − 0.355557 = **0.077262 s**

greedy:

- Also grows linearly, but **at a much higher rate**.
- Each step increases by approximately **~0.22–0.23 seconds**.
- Example:
  - From 1× to 2×: 0.905449 − 0.682102 = **0.223347 s**
  - From 2× to 3×: 1.12876 − 0.905449 = **0.223311 s**

2. Efficiency Comparison

- **ortools is significantly more efficient** in terms of transmission time.
- At MaxBytes = 125000 × 6, ortools takes **~0.696 s**, while greedy takes **~1.799 s** — **2.6× slower**.

3. Interpretation

- The ortools approach likely includes optimized scheduling or batching, reducing overhead per unit of data.
- The greedy algorithm seems to suffer from cumulative inefficiencies, possibly treating each data chunk independently without coordination.