user: bhupc
user: shreyg

Exception Handling

We introduced exception handling in our software at relevant places (Illegal Range for new types, NULL values, illegal attributes). The exceptions which could be dealt with at the rep layer were handled by printing the relevant error and continuing the execution. This is done in those cases were default value or previous value of an attribute would enable the simulation to go forward.

Scheduled changes to the fleet object's attributes

The attributes of the fleet can be set depending on the time of the day the fleet is going to operate. The two time slots we have chosen are:

1. "slot 1" = 8AM-8PM
2. "slot 2" = 8PM-8AM

Any fleet attribute can be set for these time slots, making the simulation closer to reality. Following in an example of how this could be done:

```
fleet->attribute("Truck, speed, slot 1", "1");  // slot 1 =  8AM-8PM
fleet->attribute("Truck, speed, slot 2", "2");  // slot 2 = 8PM-8AM

fleet->attribute("Truck, capacity, slot 1", "10");  // slot 1 =  8AM-8PM
fleet->attribute("Truck, capacity, slot 2", "7");  // slot 2 = 8PM-8AM

fleet->attribute("Truck, cost, slot 1", "10");  // slot 1 =  8AM-8PM
fleet->attribute("Truck, cost, slot 2", "10");  // slot 2 = 8PM-8AM
```

Real Time and Virtual Time activity Managers

We implemented the both real time and the virtual time activity managers. Appropriate activity manager can be selected by setting the attribute "scaleFactor" for the activity manager instance. The value of scaleFactor attribute should be a double value. The scaleFactor determines the amount of time the activity manager should sleep for a given duration of activity time. The exact equation is:

```
 ____________________________________________________
|                                                    |
|sleep_time = 100000us * scaleFactor * real_time     |
|____________________________________________________|
```

Example code to do this is:

This would make activity Manager to behave as a virtual manager:

```
activityManager->scaleFactorIs(0.0);
```

Following would make activity Manager to behave as a realtime activity manager which sleeps for 200,000 us for each unit of activity time

```
activityManager->scaleFactorIs(2.0);
```

Shipment Transfer Implementation

The shipment transfer is implemented through the LocationReactor and ShipmentActivityReactor interfaces. Every location has a location reactor. A LocationReactor is solely responsible for scheduling activities at a given location. A LocationReactor also does the work of an inject Activity Reactor. The way we have implemented inject ativity reactor is to always inject a timeout activity, with a nextTime attribute set to 24 hrs, into the activity manager queue. When this activity is run to completion, it notifies it's reactor, ie. the LocationReactor which reschedules this inject activity for the next day. Note that only Customer locations can have inject activities since they inject new shipments everyday.

The ShipmentActivityReactor reacts to activities scheduled by LocationReactor other than the inject activity. The "onStatus" method of the shipment activity reactor is called once the activity is over. ShipmentActivityReactor keeps complete information of the activity it is responsible for. So, there is always a one-one relationship between a ShipmentActivityReactor and it's activity. SAR keeps a reference to the segment on which this activity is scheduled to happen. Once the activity is over, SAR signals the segment and the end point of the segments that the activity is over and thus increments and decrements the package count at the two end point of the segment. Any location triggers it's LocationReactor on its "onPackageInc" method. The location reactor for this location then schedules the activities for the new bunch of packets received.

Note:: We don't decline shipments

Routing Algorithms

The LocationReactor schedules a new activity based on the destination. Since, there could be multiple paths to reach it's destination, an activity can be scheduled on multiple segments starting at a given intermediate node on the path. Since, every node is computing the path to the destination of the activity to schedule, our shipping simulation is robust to changing network topology during to the simulation. However, it's not completely robust as it does not take care of packets that might be lost because of previous scheduling on a bad path.

There are two algorithms we have implemented based on which every node can decide which is the next segment on which this activity could be scheduled:

1. **Least Time Dijkstra**: An intermediate node calculates the least time path using the "Dijkstra" algorithm. The first segment on this path is where it will schedule the new activity.

2. **Least Time Greedy:** An intermediate node calculates the least time segment towards the destination. So, this is greedy and might not be the least cost path to the destination, but node can make such choices because they might not have information about the complete network graph.

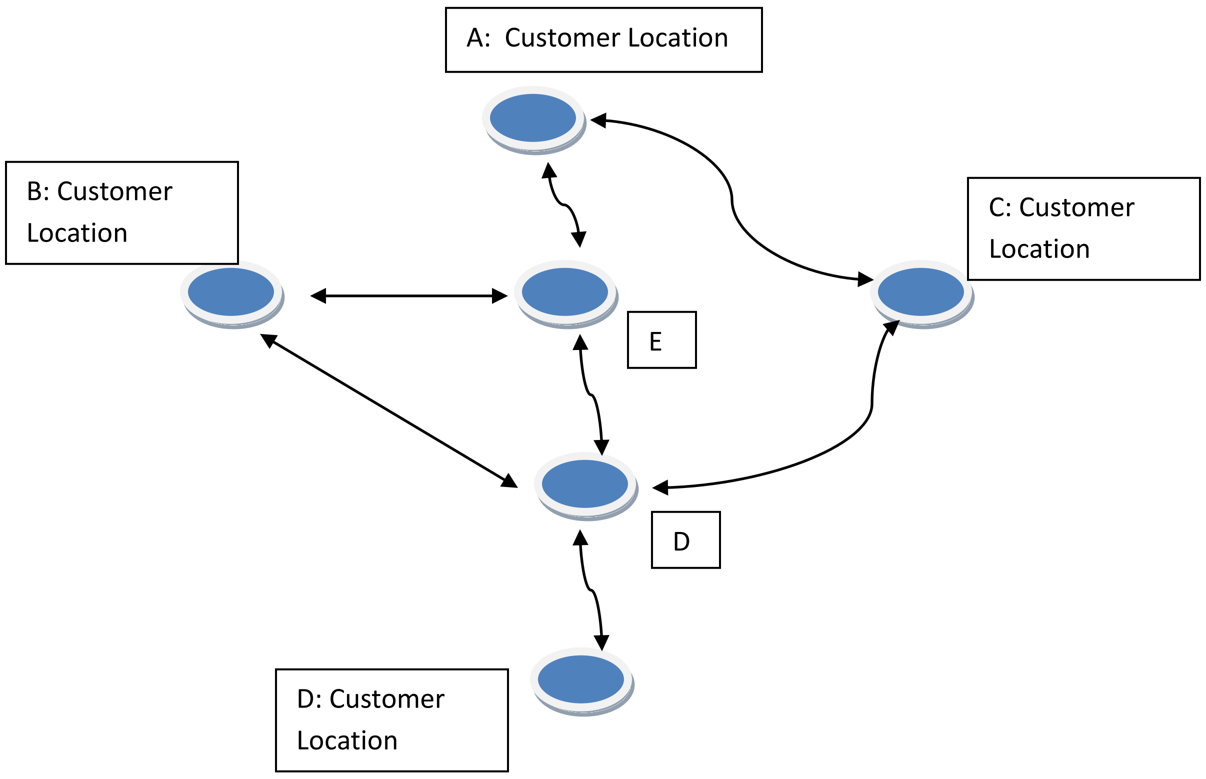
Client is free to chose any one of the two routing algorithms. This can be done by setting the "routing algorithm" attribute of the connectivity instance. Example code to do this is:

```
conn->attributeIs("1"); // THis is for Dijkstra
conn->attributeIs("2"); // This is for Greedy
```

Simulations -

Verification.cpp

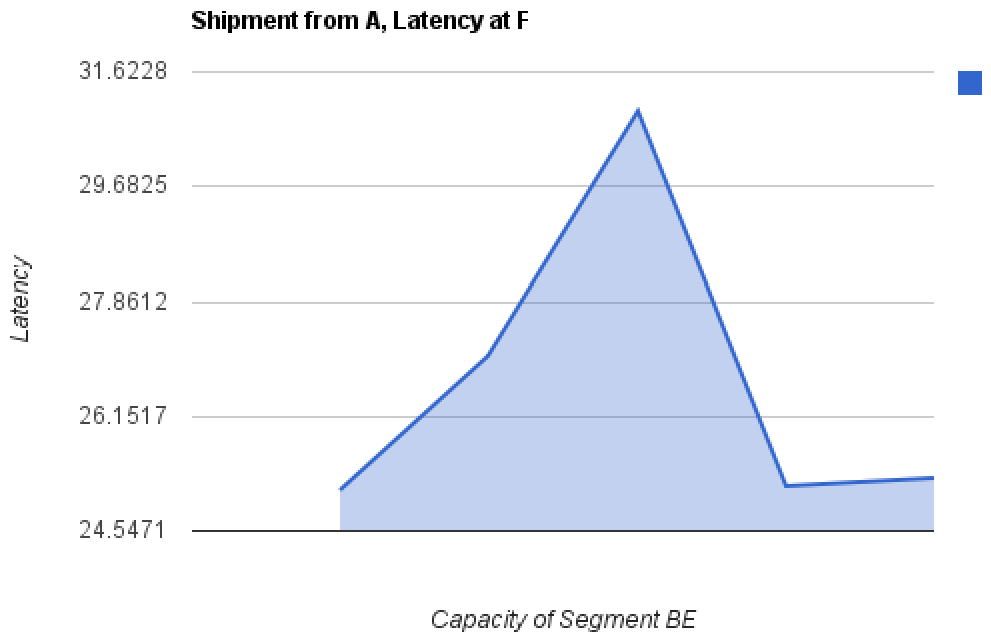
Following is the diagram of our test network:



We use the above topology as our verification topology.

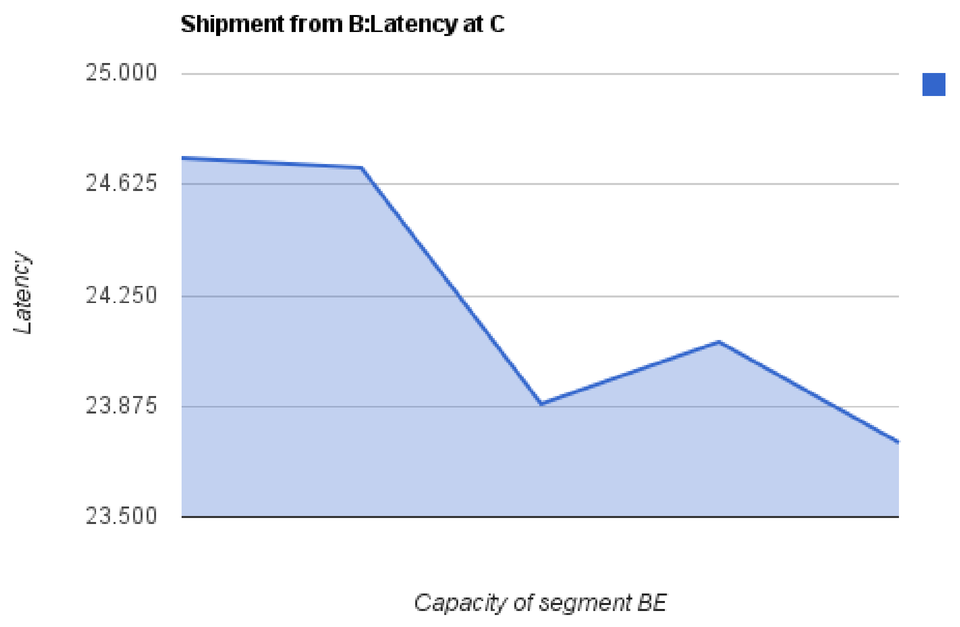
We study the characteristics of network transfer from **A to F and B to C**. We keep the cost of the segment BD and AC extremely high, forcing the topology to route via E and D. This causes congestion at E and we study this behavior.

Observations –

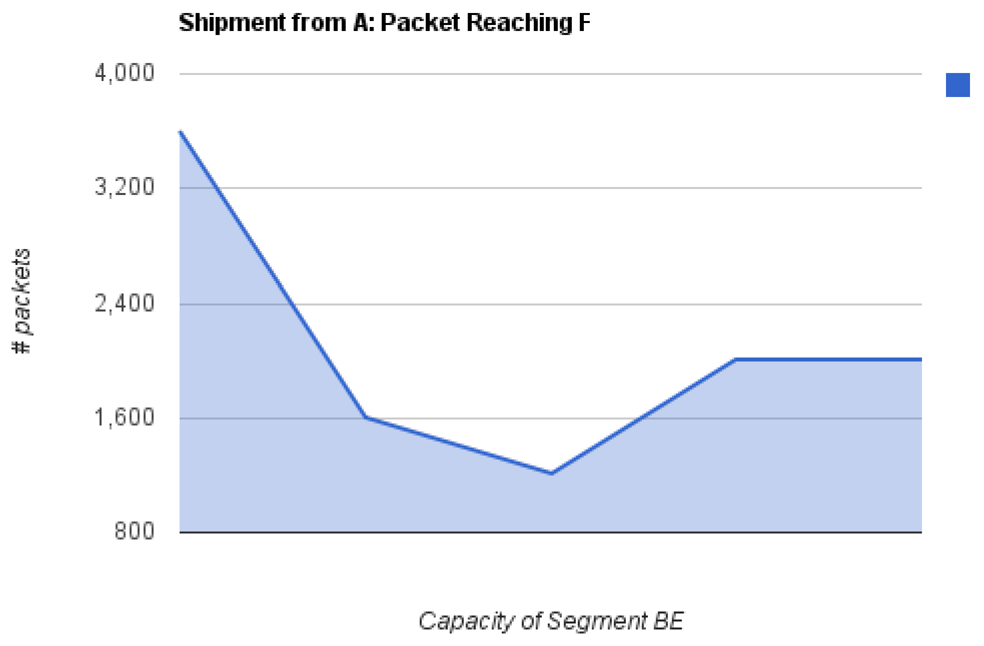


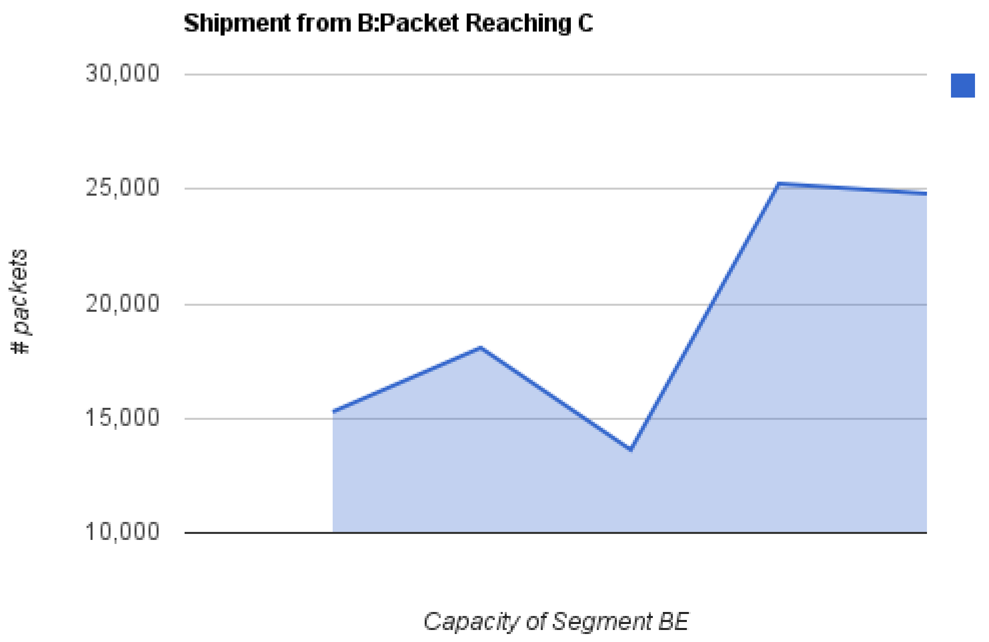
We can observe that the latency of A increases initially as capacity of BE is increased. This is due to the fact that at E, packages from B are queued before those from A.

The subsequent decrease can be explained by the fact that if capacity of BE is increased beyond a threshold, the queuing of B happens much before queuing from A starts and hence we see a drop in latency at C.



The latency at C, for shipments from C decreases as expected with increase in capacity of BE. Note that the decrease in latency dies down after a level, as the capacity of segment ED is unchanged and it remains a bottleneck.





We can see that the behavior is complementary to the latency observations. Minor variations are also due to the simulation time frame (number of days, day/night slots etc.)

- **Impact of Relative Scheduling -** One of the interesting observations we made during our simulations was the role of task scheduling algorithm at intermediate nodes. Though our software ensures that we find the most cost (or time) efficient path in the network. The issue of picking up one shipment job over the other was handled on FCFS basis.
  Usually in real world, each shipment would have a associated priority and that has to be used while deciding the order of scheduling.

Experiment.cpp

The results for the simulation are (T = 30 hours)

Shipment Size == 100

Total packets received at = 10200
Average packet latency at d = 0.11
Total delivery cost at d = 31800.00
Total packets received on critical segment = 200

Random Shipment Size

Total packets received at = 3853
Average packet latency at d = 0.12
Total delivery cost at d = 13620.00
Total packets received on critical segment = 227

In the first case with constant shipment size, we observe that we receive all the packages (100*100) from the left hand side of the network and there is no bottleneck there but only 200 of 10,000 packages from the 100 sources on the right, due to the bottle neck segment connecting the Destination to a single truck terminal from which the network further fans out.

In the second case, we can observe that the critical segment had a better performance, probably due to the fact that the random size resulted in shorted queues at intermediate nodes and hence the performance of critical segment increased by 13.5%. We see a decrease in overall packets delivered but that is only due to the fact that random shipment size choices (1-1000) resulted in lower number of packages from those sources.

We again ran the simulation in which we kept the size of shipment from sources on left as 100 and chose a random value for sources on right. The results obtained were consistent as all packets from left sources reached the destination.

Total packets received at = 10223
Average packet latency at d = 0.11
Total delivery cost at d = 32340.00
Total packets received on 1 = 223

*Note – The **refused_packages** metric is not reported as there is only single path to route here (from every source to Destination),so there cannot be any re-routing, and also we do not drop any packages.

EXTRA CREDIT:

We have implemented a shipment acknowledgement mechanism in which the source of the shipment gets the acknowledgement back from the destination of the shipment once the destination receives the shipment. For, this we added a new attribute to the source customer location called "Acknowledegments Received" , the value of which would be the number of shipments for which this source has received acknowledgement packets.

```
source->attribute("Acknowledgements Received");
```