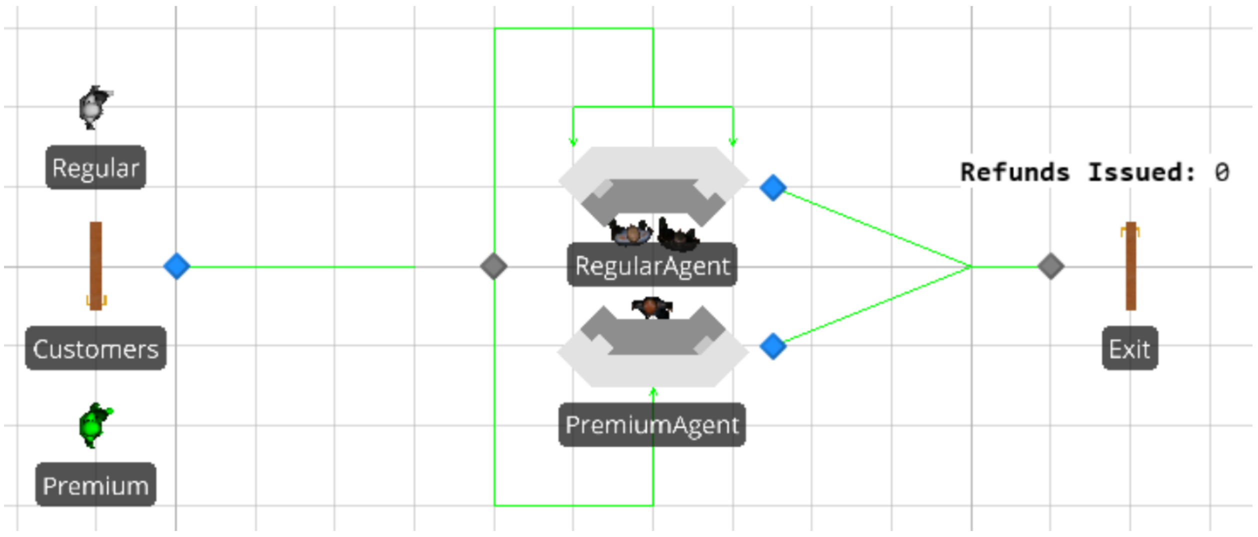
DATA 604 Assignment 10: Movement and Animation

Dan Smilowitz

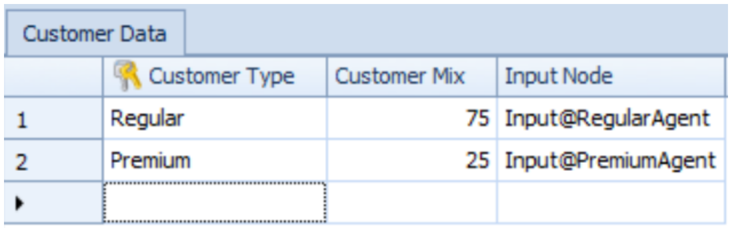
April 8, 2017

Model Setup

A model is created with a single source, two servers, a single sink, two entities, and no links. The input nodes for the servers are colocated. A floor label is created to display the number of refunds issued.

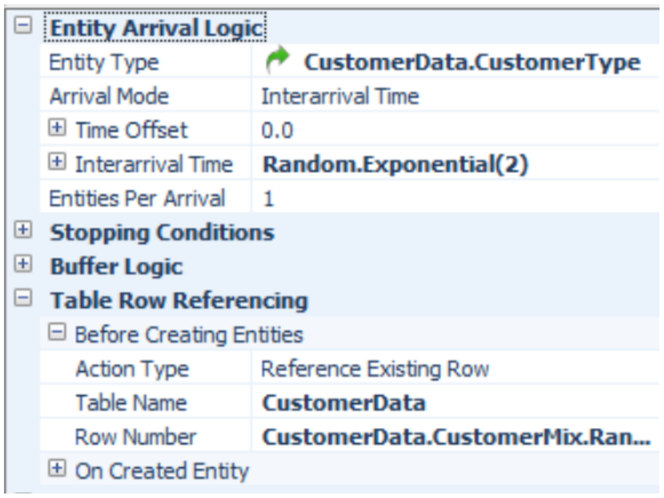


The customer mix is set using a table and references on the source object, as shown below:



Customer Data

	Customer Type	Customer Mix	Input Node
1	Regular	75	Input@RegularAgent
2	Premium	25	Input@PremiumAgent



Entity Arrival Logic	
Entity Type	**CustomerData.CustomerType**
Arrival Mode	Interarrival Time
Time Offset	0.0
Interarrival Time	**Random.Exponential(2)**
Entities Per Arrival	1
Stopping Conditions	
Buffer Logic	
Table Row Referencing	
Before Creating Entities	
Action Type	Reference Existing Row
Table Name	**CustomerData**
Row Number	**CustomerData.CustomerMix.Ran...**
On Created Entity	

Tracking Refunds

To identify when premium customers are eligible for a refund, a continuous state variable `NumRefunds` is created. It is assumed that the "wait" referenced in the problem refers to the time between the customer entering the office and beginning processing with an agent. As such, when an agent begins processing, it performs the state assignment

```
NumRefunds = NumRefunds + Is.Premium * (ModelEntity.TimeInSystem * 60 >= 5)
```

This checks if the customer is a premium customer, then checks if it has been in the system for 5 minutes or more; if so, the state variable is incremented by 1.

Revenue and Cost

An additional continuous state variable, `Revenue` is created. This variable is incremented by $ 8 every time a customer enters the sink. It reflects the net revenue from a successful transaction, not counting any refunds. The total of this variable is reported through the use of an output statistic `GrossRevenue`.

To track costs, the final value of `NumRefunds` is collected in the output statistic `RefundCount`; this is multiplied by $ 15 to give the output statistic `RefundCost`. The staff of cost is gathered by setting the idle cost rate and usage cost rate for each server to $ 55 per hour; this is collected in the output statistic `CostStaff`.

Finally, the net income at the end of a run is gathered by creating an output statistic:

```
NetIncome = GrossRevenue.Expression - RefundCost.Expression - Cost
```

Entity Movement

To handle entity movement through free space, the output nodes of the two servers given the following routing logic:

- Outbound Travel Mode: *Free Space Only*
- Entity Destination Logic: *Specific*
- Node Name: `Input@Exit`

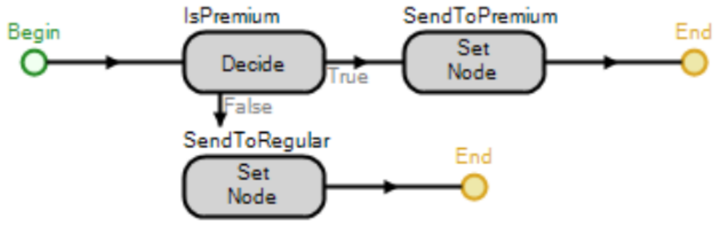
Entity movement from the source to the servers is investigated under multiple assumptions below.

Server Processing of Customers

The rules dictating how servers process customers of different types may affect servers' utilization, customers' wait times, and the number of refunds issued. These rules are implemented in Simio by changing the routing logic of entities and the processing logic of servers.

Dedicated Agents

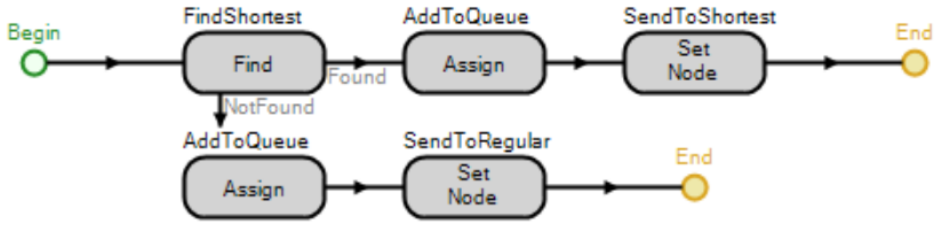
The first assumption on customer handling is that the premium agent handles premium customers, and the two regular agents handle regular customers. This is implemented by setting the destination node for entities based on their type; this fairly straightforward add-on process involves a *Decide* step on the condition `Is.Premium` followed by a *SetNode* step under each condition (one setting to `Input@PremiumAgent` and the other to `Input@RegularAgent`):



Shortest Line

If customers are not required to go to the agent dedicated to their customer type, they are likely to choose the shortest line when determining their destination. In order to track the line at each server, a new state variable `ServerQueue` is created as a real vector with 2 rows (one for each agent). This variable is used to track the number of customers currently being served or waiting to be served at each server. Since the Premium server has only one agent, while the Regular server has two, the values are doubled for the Premium server.

When a customer is deciding which server to go to, they first find the shortest line through a *Find* step; a new dummy real state variable `Index` is created for this purpose. The appropriate `ServerQueue` is then incremented to reflect the addition of the new customer in an *Assign* step. The customer's destination is then set using a *SetNode* step which references the final column of the `CustomerData` table shown above. If the expression `ServerQueue[Index]` can not be minimized (possibly due to a tie), the customer is sent to the Regular server since it has two agents.



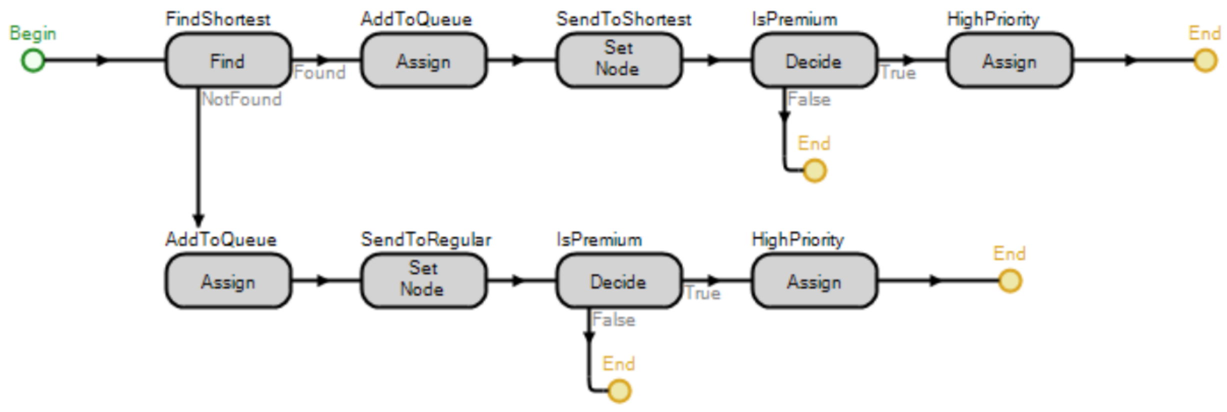
To make the values in `ServerQueue` reflect the shortening of lines as customers are served, the servers performs the following state assignment after processing:

- RegularAgent: `ServerQueue[1] = ServerQueue[1] - 1`
- PremiumAgent: `ServerQueue[2] = ServerQueue[2] - 2`

As mentioned above, the Premium server decreases the value by 2 since one processed customer likely has double the impact (at least in perception) there as compared to the Regular server.

Introducing Priority

The final alternative considered is a modification of the shortest line approach, where priority is introduced. Under this scenario, customers still choose their destination based on the shortest line, but Premium customers are served before Regular customers. This is done by adding *Decide* and *Assign* steps to the previous process, checking if an entity `Is.Premium` and assigning it `Priority = 2` if so. The servers' process logic setting are changed to reflect this as well, using a ranking rule of *Largest Value First* keyed off of `Entity.Priority`.



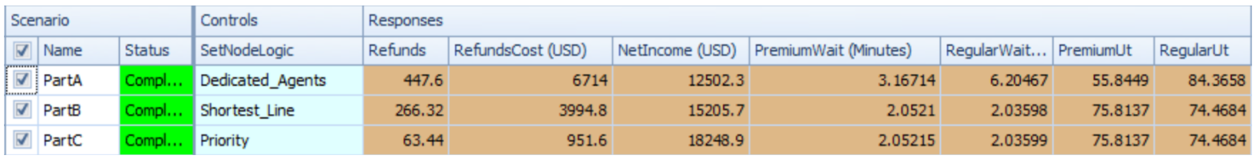
Comparing Results

In order to compare the results of the different routing and processing rules, the `Entered` add-on process at `Output@Customers` is set to a referenced property `SetNodeLogic`. This can then be used to handle the customer node selection as different scenarios in the same experiment. Since all entities are defined an initial priority of 1, the process logic of the servers is set to be based on largest value first, as this defaults to *First In, First Out* under ties.

An experiment is created with 3 scenarios, each executed for 50 replications. Each replication is run for 10 days, with a warmup period of 8 hours. Seven responses are defined:

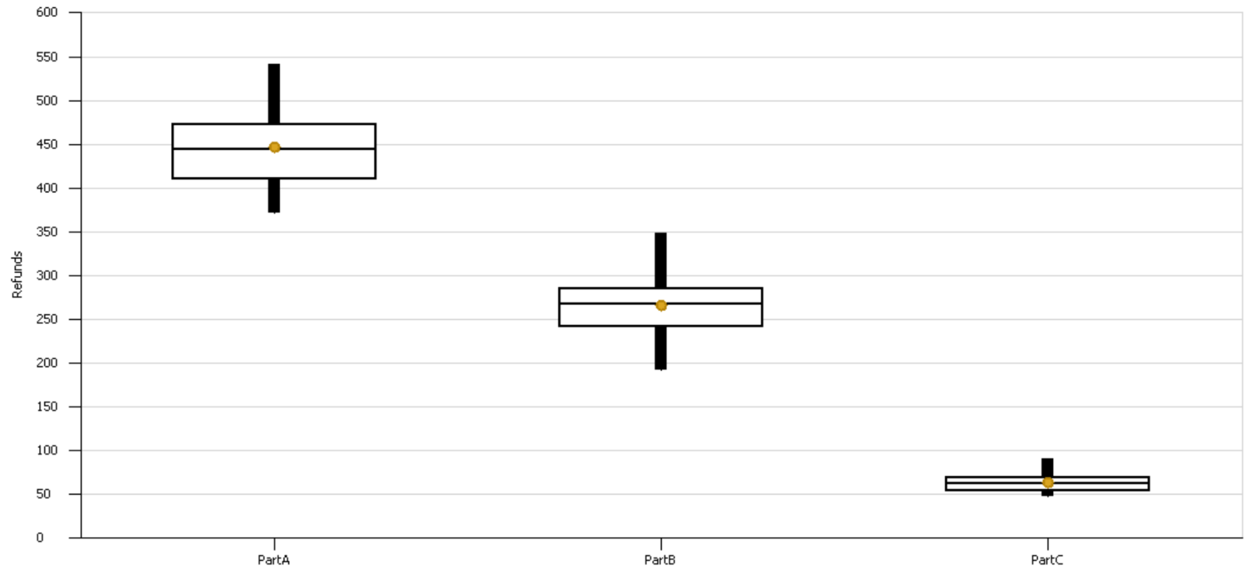
- Number of refunds issued:
  - **Refunds** = `RefundCount.Value`
- Cost of refunds issued:
  - **RefundsCost** = `RefundCost.Value`
- Net income earned:
  - **NetIncome** = `NetIncome.Value`
- Waiting time for Premium customers:
  - **PremiumWait** = `PremiumAgent.InputBuffer.Contents.AverageTimeWaiting`
- Waiting time for Regular customers:
  - **RegularWait** = `RegularAgent.InputBuffer.Contents.AverageTimeWaiting`
- Utilization of Premium server:
  - **PremiumUt** = `PremiumAgent.Capacity.ScheduledUtilization`
- Utilization of Regular server:
  - **PremiumUt** = `RegularAgent.Capacity.ScheduledUtilization`

The results of the experiment are presented below:



Scenario			Controls	Responses						
☑	Name	Status	SetNodeLogic	Refunds	RefundsCost (USD)	NetIncome (USD)	PremiumWait (Minutes)	RegularWait...	PremiumUt	RegularUt
☑	PartA	Compl...	Dedicated_Agents	447.6	6714	12502.3	3.16714	6.20467	55.8449	84.3658
☑	PartB	Compl...	Shortest_Line	266.32	3994.8	15205.7	2.0521	2.03598	75.8137	74.4684
☑	PartC	Compl...	Priority	63.44	951.6	18248.9	2.05215	2.03599	75.8137	74.4684

It can clearly be seen that moving from dedicated agents to shortest path to priority increases net profit and decreases the number of refunds. Investigation of the SMORE plots shows that the variability of the number of refunds issued also decreases:


Refunds
PartA
PartB
PartC