Anthony Poerio (adp59@pitt.edu)
University of Pittsburgh
CS1538, Spring 2016
Airport Simulation

Overview

This document outlines the implementation details and design decisions I made while creating an airport check-in simulation in Python for the 4th assignment in CS1538 at the University of Pittsburgh. It goes on to outline the results from a series of experiments performed on the completed system. Finally, recommendations are made for the best way to staff the airport check-in system, based the results of these experiments. The full project specifications can be found online, at: http://people.cs.pitt.edu/~lipschultz/cs1538/homework4.html

Objectives

The objective of this experiment is to help the airline determine the best way to achieve these goals:

1. Maximize profit
2. Minimize check-in wait time
3. Maximize likelihood that an international passengers catches the plane
4. Minimize the post-security screening wait time for commuter passengers
5. Minimize the agents' idle time

Experiment Design

Random Variable Generation

I began my implementation with the random variable generation methods. In order to maintain consistency throughout the entire program, I decided to generate all times in terms of minutes.

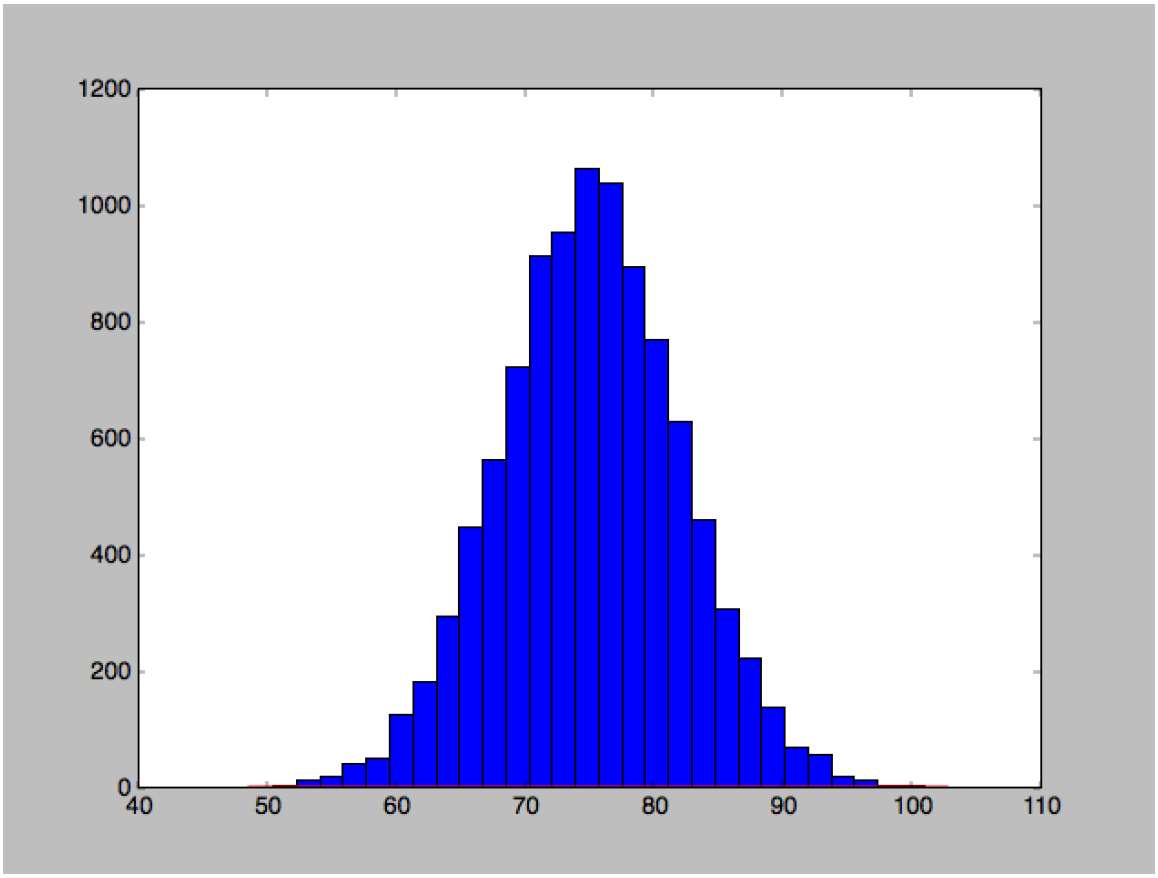
To start, I created the method for **commuter arrivals**. This follows a **Poisson Process** with an arrival rate of 40 people per hour. Converting to minutes, we have 40 arrivals every 60 minutes, or an arrival rate (lambda) of 40/60, which reduces to 2/3**.** Because the Exponential PDF can be used to model waiting times between two successive Poisson events, I used the Exponential PDF with **lambda=2/3** to model this arrival. This can be done simply by returning: **-log(1.0 – U[0,1)) / lambda**

Next, I created the method for generating **international arrivals**. These arrivals follow a Normal Distribution with **mean=75 minutes** and **variance=50 minutes^2**. In order to achieve this: I used the Polar Coordinates method of generating variables that follow the normal distribution, to get a Standard Normal random variate. I then transformed this value into one that follows our desired distribution performing this calculation on each resulting value:

StdNormal_Value * (Std_Deviation) + Mean
StdNormal_Value * 50 + 75

I chose this method because it reliably approximated values according to the Normal distribution. An example of the output can be seen in the histogram below:

Polar Coordinates Method Output



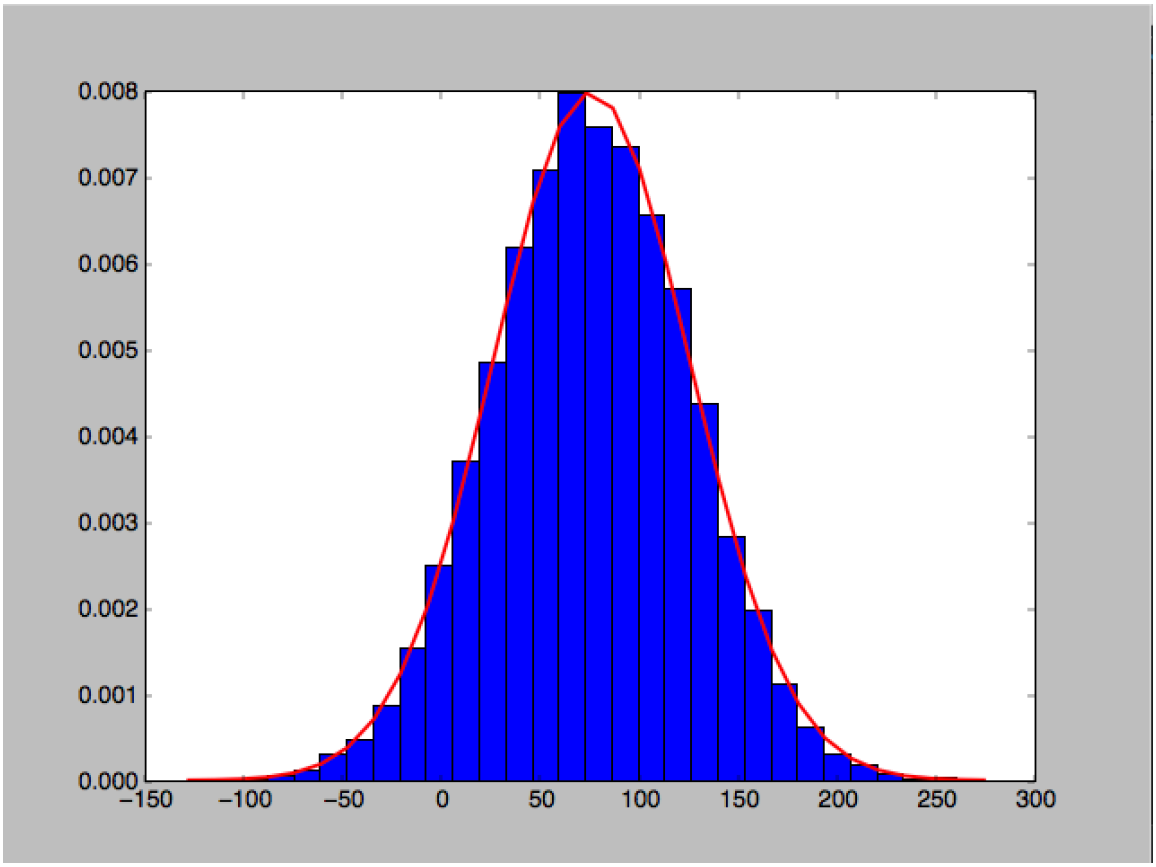
X-Axis = Random Value Generated
Y-Axis = Number of Values Generated At this value

I also attempted to generate this data using the **Random Normal Generation** method supplied by the **NumPy library**, but this method generated a large number of values lower than 0, because of the large variance. For my simulation, this would mean that the passengers arrived AFTER their flight left. And because we assume that these passengers don't make any logical decisions, in order to simplify the experiment, this would mean that these passengers simply have no chance of making their plane, but sit in the line and clog up the queue anyway. I wanted to avoid this, since it doesn't make logical sense, and so ***I decided to use the Polar Coordinates method to model International Arrivals***. The main thing to note here is that this method does not pass all statistical tests confirming that the variance is accurate, but I decided to go with this version anyway, because I did not have a good

way to handle negative arrival times, and they just don't make much sense for this simulation, in my opinion.

Confirmation of the output from NumPy's Standard Normal function using our parameters can be seen below. Again, I decided that using these values did NOT make sense, since many of the arrivals would simply new have a chance to make their plane at all.

NumPy Standard Normal Output



X-Axis = Random Value Generated
Y-Axis = Percentage of Values Generated At this value

Generation of **international seats sold and carry on items**—both for Coach and First Class—were all modeled using **NumPy's built in Binomial** function. For example.

- Coach ticket sales are modeled using this function call:
  - [np.random.binomial(150,0,85)]
- First Class ticket sales are modeled using this function call:
  - [np.random.binomal(50, 80)]

Similar calls for carry on items can be seen in the attached source code. And the **total number of carry on items** were modeled via the **Geometric Distribution**. In

order to achieve this using python, I simply counted the number of Binomial trials until a success, all within a while loop. The code snippet looked like this:

```
counter = 0
successful_trial = 0
while successful_trial == 0:
    successful_trial = bernoulli_trial_commuter()
    counter += 1

return counter
```

All **methods for service time in the queuing processes** themselves were generated very simply, using this Exponential Distribution formula:

- **-log(1.0 – U[0,1)) / lambda**
  - Where lambda=arrival rate for "service time"

Queues

Next, I **made a class to handle each type of Queu**e. In both classes, I created only methods for **adding to** and **removing from** our queue. Underneath the hood, I'm using Python's Queue class, which helped me model this smoothly. There are a number of function calls that need to happen on both addition and removal, and I encapsulated all of that in the class methods to make these queues easier to work with.

Servers

After that, I **made a Class to handle each type of Server**. For my purposes, I considered both the Check-In agents and Security machines to be "servers". Each server class generates a service time as soon as a customer is added to it, and then counts down as the system time increments. By encapsulating these key parts of the system as their own classes, I intended to abstract away as much of the design as possible, then test it, so I knew that each of my components worked. This turned out to be a good idea, and this design pattern helped everything work smoothly.

Time Advancement System

This airport simulation is created in Python, using **a Fixed-increment time advancement** system, made from scratch. At each "step" in the simulation, we advance the simulation clock by $1/100^{th}$ of a minute. In practice, that means we increment the system time by the floating-point number: **0.01**. Every time we update the system time, we also update the all the **servers**, and if they are no longer busy, we then grab the next Object waiting to be served, and begin to serve them.

Because international flights leave every 6 hours, I **decided to handle arrival generation on six hour intervals**, feed those all into the system, and repeat the process for as long as the user desires.

Finally, I **decided that all simulations will be run for a specified number of days**. This makes sense because we are generating arrivals on six hour intervals, so each day of simulation results in four sets of arrivals being created and fed into the system.

Assumptions Stated

This section outlines assumptions I made as I interpreted the assignment prompt:

- Both Check-in Agents and Security Machines will be modeled as "servers" in the system, but ONLY check-in agents get paid hourly.
- We cannot increase the number of security machines (though I did this later to stabilize the system), but we CAN increase the number of check-in agents, up to 6.

These are not many assumptions, but I'm noting them mostly because *I am not counting **cost of security machines** in my wages, and I wanted to be clear as to why.*

Key Statistics

In order to determine the overall health and effectiveness of the system, I collected a wide variety of metrics, all of which can be seen by running the accompanying Python simulation program. However, **the most useful of these can be viewed readily in the attached Excel File named "*ADP59-CS1538-AirportSimulation_Data.xlsx*"**.

In this file, I've collated and organized all the data from every experiment I ran, and it can be easily viewed. For completeness, please note that the data points I used for my analysis are:

- Total Profit
- Refunds Issued
- Number of Passengers who WOULD get a refund if they missed their flight (Max Refunds)
- Money Lost to Refunds
- Wages Paid
- AVG Time in System
- Longest Time in System
- AVG Time in System 1st Class
- AVG Time in System Coach
- Number of International Passengers who Made their Flight
- Number of International Passengers who MISSED their flight
- Number of Commuter Passengers Moved to Commuter Gate
- Number of Passengers Currently En-queued
- Post-Security Wait Time for Commuter Passengers
- Agents' Average Idle Time Per Day

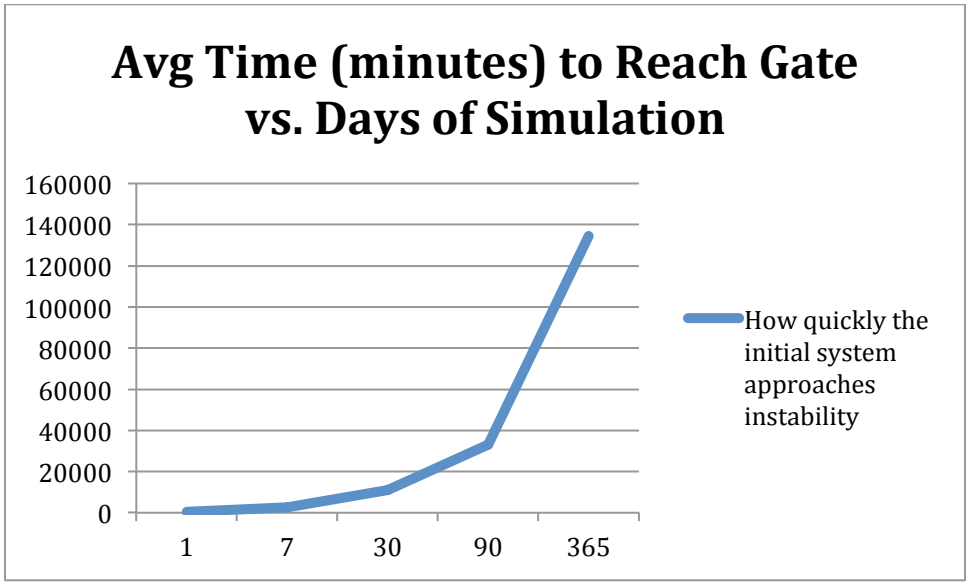
Initial Results

My first few runs of the experiments used the settings prescribed in the project prompt:

- Check-in Station
  - 1 Server for First Class
  - 2 Servers for Coach
- Security Station
  - 1 Server for First Class
  - 2 Servers for Coach
- Departures
  - Commuters: Every 60 minutes
  - International: Every 6 hours

I found that the system prescribed by these settings was highly unstable, and the customer wait time quickly grew towards infinity, spiraling out of control. To demonstrate this, I've plotted the average customer wait time (Y-Axis) as a function of Simulation time. At first, the system appears reasonable, though un-optimized. The **average wait time for a customer after one-day is 395 minutes.** But before long, we have times that are not possible in real-life. **After 30 days the average wait time has grown to 11,117 minutes**. And it keeps going up from there. The graph below illustrates this.

Average Wait Time



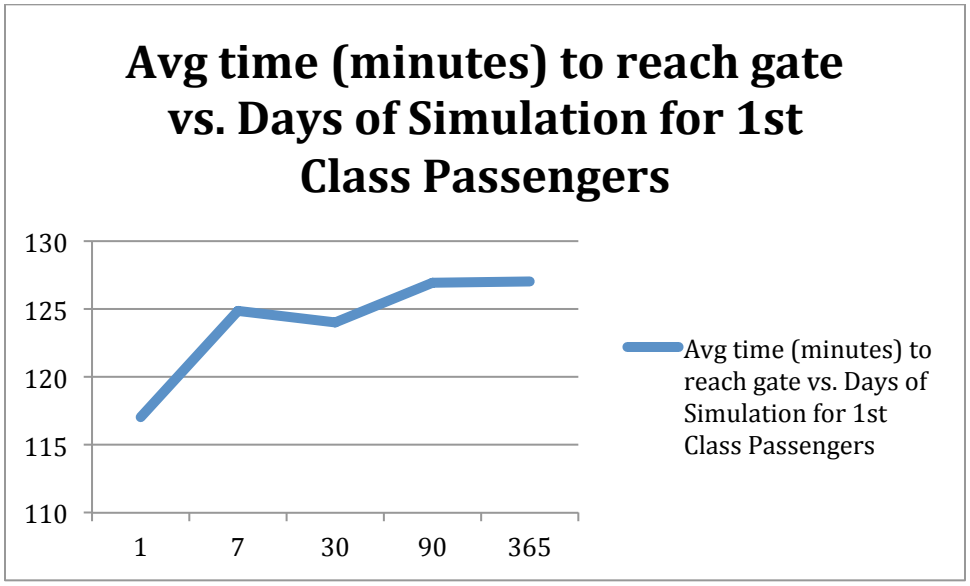
X-Axis: Simulation Time in Days
Y-Axis: Average wait time in Minutes

Because of this instability**, the data quickly grows unreliable**, because lots of people simply never finish. The system isn't stable and the size of the check-in queue will grow towards infinity.

This affects all aspects of the system. For example, we see that after 365 days of simulation, only 955 refunds were issued. But the number of passengers who WOULD get a refund if they made it to their gate is 4178. AND, the wait time is always growing towards infinity. A this point, no international passenger will ever catch their plane again, and so all of these refunds will be issued. **But the system doesn't account for that because no one ever makes it to their gate. This means we desperately need more servers, and 3 at each station isn't sufficient.**

However, on the brighter side—while the overall average wait time is dismal, nearly all of this seems to come from passengers waiting for line in coach. **The First Class passengers**, who have their own dedicated servers (and who are a much smaller group overall), **have a completely stable wait time**, at all points in the simulation. And that wait time is reasonable, **averaging about 127 minutes per customer**.

First Class Wait Times



X-Axis: Simulation Time in Days
Y-Axis: Average wait time in Minutes

Now, given that first class is stable and the wait time is reasonable, we know it's possible to make this system work, as long as we add more servers for the commuters. But the question is: How many servers do we need to add?

At this point I decided to figure how first how to stabilize this system, and then, once that was complete, optimize from there. The rest of this document outlines my attempts to stabilize the system, my optimizations, and finally my recommendation.

Attempted Improvements

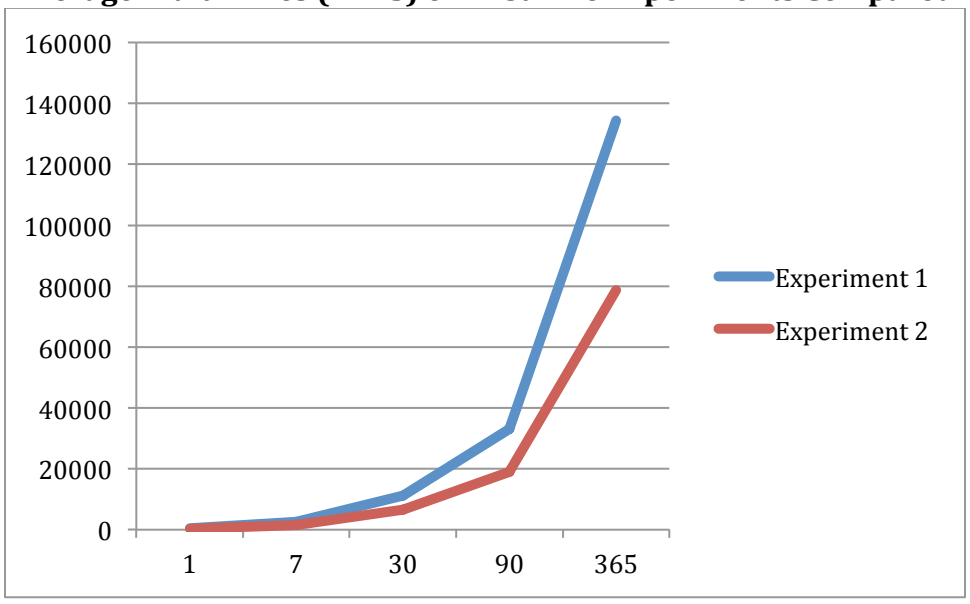
Experiment 2 – Use Max Check-in Agents

My first attempted improvement was to increase the number of check-in agents to the maximum allotted amount. The parameters for this experiment became:

- Check-in Station
  - 1 Server for First Class
  - **5 Servers for Coach**
- Security Station
  - 1 Server for First Class
  - 3 Machines For Coach
- Departures
  - Commuters: Every 60 minutes
  - International: Every 6 hours

Here, I used all six possible check-in agents, but changed nothing else. Here's how the results stacked up.

Average Wait Times (MINS) of First Two Experiments Compared

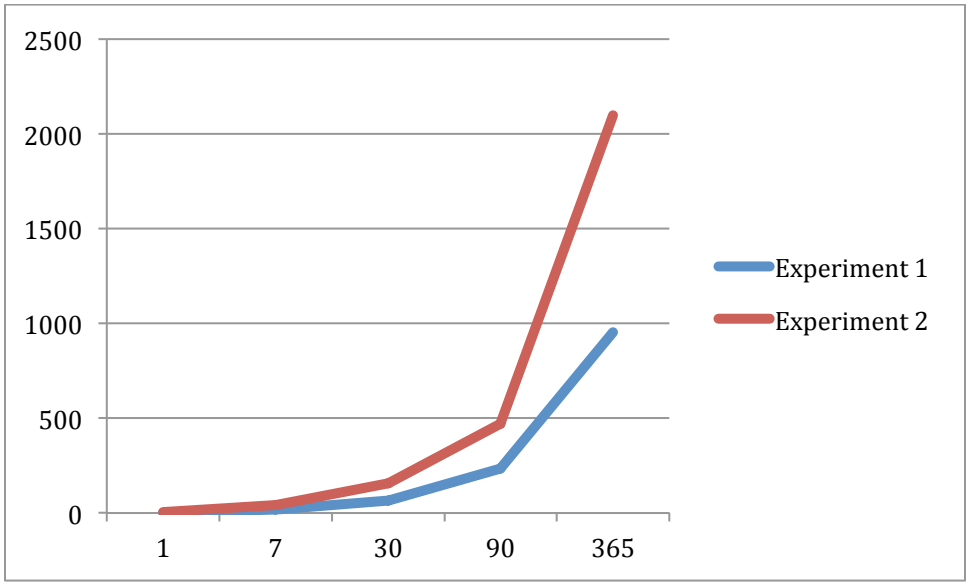


Y-Axis: Avg Wait Time in Minutes
X-Axis: Simulation Time in Days

So we're already cutting down the wait time, but we're still not at a reasonable number yet. **The system is still unstable, but the wait times just approach just approach infinity more slowly.**

Another somewhat surprising result: As we push more people through the system, **we start to issue more refunds**. This is because fewer people were finishing at all before.

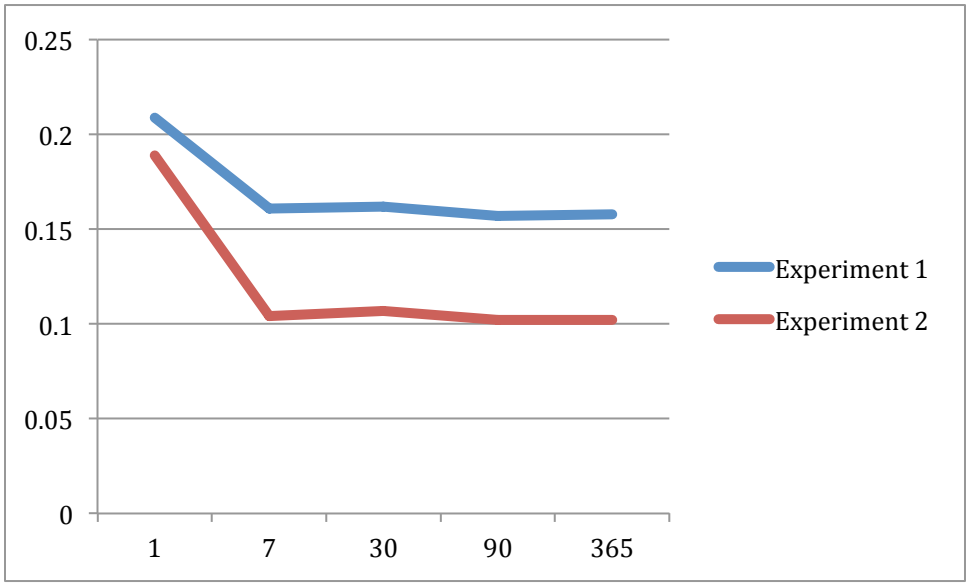
Number of Refunds Issued—First Two Experiments



X-Axis: Simulation Time in Days
Y-Axis: Number of Refunds Issued

And more: the likelihood that an international passenger makes his flight goes down as well.

Likelihood of Making Your Flight as an International Passenger



X-Axis: Simulation Time in Days
Y-Axis: Probability of Making your flight

Again, we're seeing this likelihood go down, because we're getting an increasingly accurate view of the system, which is still unstable.

But we've now hit a problem: We can't optimize it any further within the specified parameters. We're maxed out on the allowable number of check-in agents and security machines, but no one is making it through the system. From this we can make a conclusion.

Conclusion

These data show that it's **not possible to stabilize the system** without either:

1. Using more than **6-total** Check-in Agents
2. Using more than **3-Total** Security Machines

And because the system is not stabilized, we can't achieve many of the other goals easier. For example: it doesn't matter how long a commuter passenger waits at his gate if he never makes it through the queuing system. And no agents will ever be idle for long, but the system is itself an impossible entity.

Therefore: We must seek other ways to stabilize this system. I chose to do this by adding even more check-in agents and security machines**. I modeled both the check-in agents and security machines as "servers" in my simulation, so going forward I'm using "server" as a catch-all term for both of these entities**. My goal is to find out how many servers are necessary overall. I also want to find what is optimal.

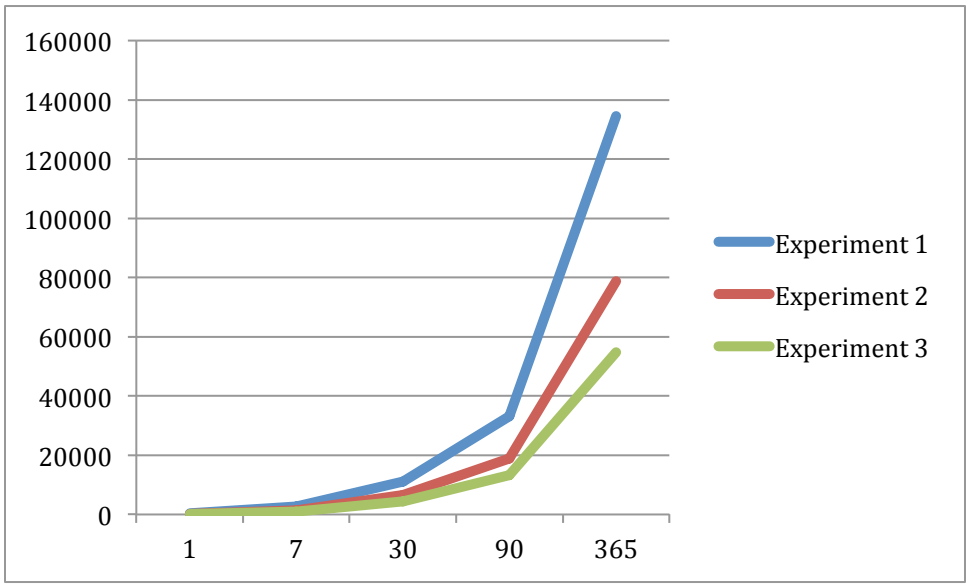
Experiment 3 – Add More Security Machines

In order to stabilize the system, I first started by adding more security machines. A 1-to-1 ratio is stable for First Class passengers, so I decided to first try the same thing for Coach.

Thus, the parameters for this experiment were:

- Check-in Station
  - 1 Server for First Class
  - **5 Servers for Coach**
- Security Station
  - 1 Server for First Class
  - **5 Machines For Coach**
- Departures
  - Commuters: Every 60 minutes
  - International: Every 6 hours

Average Wait Times (MINS) of First Three Experiments Compared



Y-Axis: Avg Wait Time in Minutes
X-Axis: Simulation Time in Days

Again, an improvement, but we're not quite stable yet. We are still approaching infinity, but more slowly. In the next experiment, I decided to stabilize the system at any cost, just to establish that it's possible.

Experiment 4 – Stabilize the System at Any Cost

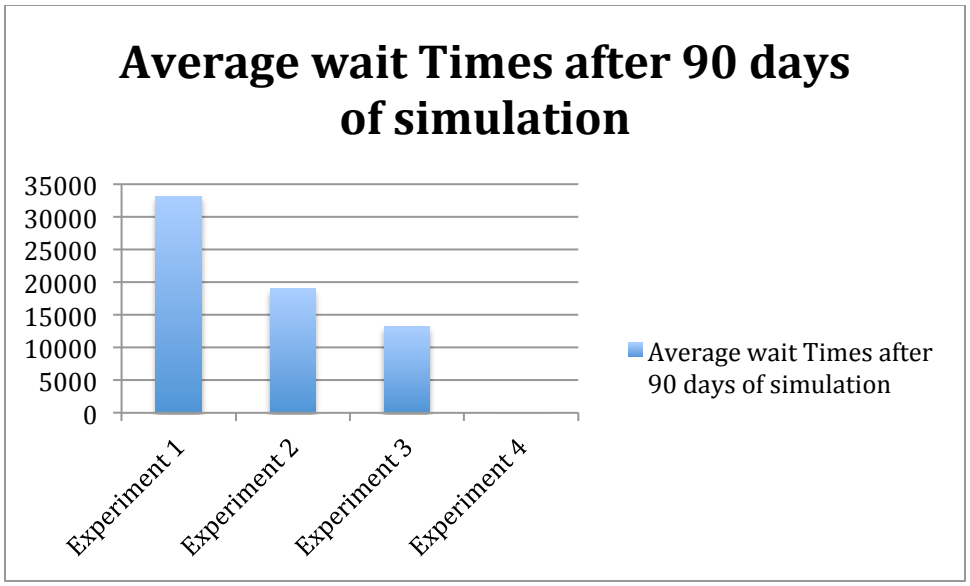
For this experiment, I quadrupled the number of servers at both stages of the queuing process. There were 24 total servers for check-in and 24 servers for security.

Thus, the parameters for this experiment were:

- Check-in Station
  - 1 Server for First Class
  - **24 Servers for Coach**
- Security Station
  - 1 Server for First Class
  - **24 Machines For Coach**
- Departures
  - Commuters: Every 60 minutes
  - International: Every 6 hours

I decided to run the experiment for 90 days to start, because by this point we have usually seen strong indicators of whether the system will stabilize or not. **Luckily, in this case, I found that the system was stable.**

A results comparison for all experiments can be seen below:



Overall, the average wait time for all users after 90 days of simulation was shown to be **25 minutes.**

This was a great result, but now I turned my attention to optimizing the system.

Experiments 5 and 6 – Finding the Sweet Spot

Over the next two experiments, I lowered the number of total servers to determine perfect amount.

Experiment 5 was run with the following parameters:

- Check-in Station
  - 1 Server for First Class
  - **12 Servers for Coach**
- Security Station
  - 1 Server for First Class
  - **12 Machines For Coach**
- Departures
  - Commuters: Every 60 minutes
  - International: Every 6 hours

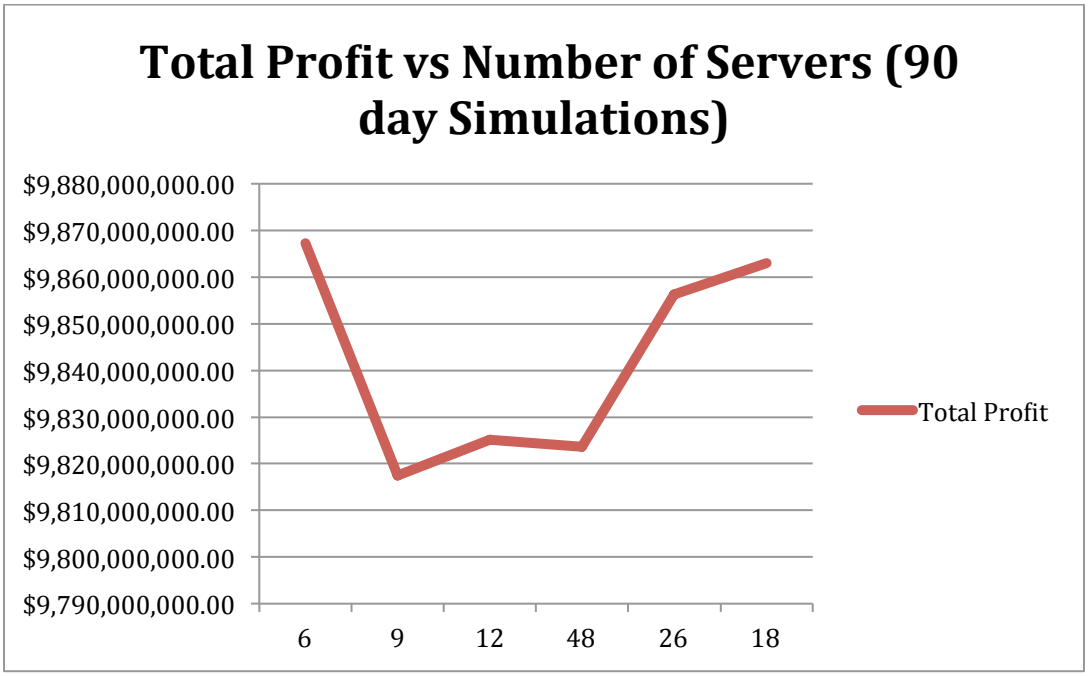
At this point, I saw that the results were essentially the same as before, so I cut the server size even more aggressively. **And in experiment 6 I found the inflection point.** Comparative data for all six experiments will be shown below.

Please note that all graphs are plotted against "number of servers" on the X-Axis, but I'm keeping this in chronological order (Experiment Number), just for clarity.

Experiment 6 was run with these parameters:

- Check-in Station
  - 1 Server for First Class
  - **8 Servers for Coach**
- Security Station
  - 1 Server for First Class
  - **8 Machines For Coach**
- Departures
  - Commuters: Every 60 minutes
  - International: Every 6 hours

At this point, I started to see inflection points in many of the curves, most importantly, PROFIT:



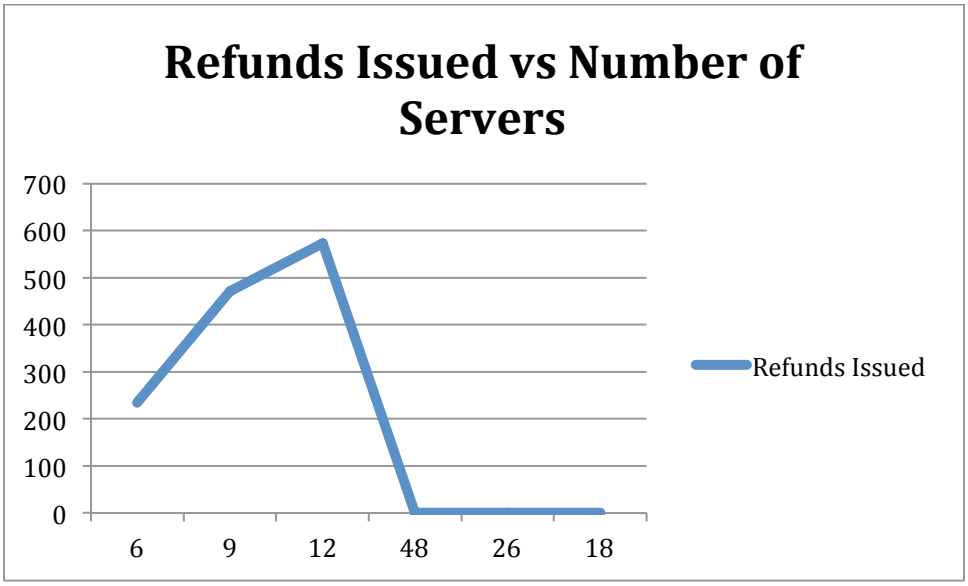
X-Axis: Number of servers, but this can also be seen as experiment number.
Y-Axis: Total Profit after 90 days of Simulation

Here, we can see that profit is high for 6 servers total (Experiment 1), but this is because many of the users who should be getting refunds simply **never exit the system**. Profit takes a downturn from this point, as we add servers, mostly because people leave the system faster, and we see the effects of its instability earlier.

However, once we hit 48 servers (our max) profit moves up sharply, and keeps going all the way until 18 total servers (final experiment), where we can see it start to dip again. **This is because: with 18 total servers, the ONLY people who miss their flights are people who arrived more than 90 minutes late for their flight.**

So we never issue refunds. This is the sweet spot in the trade-off between staff size and profitability.

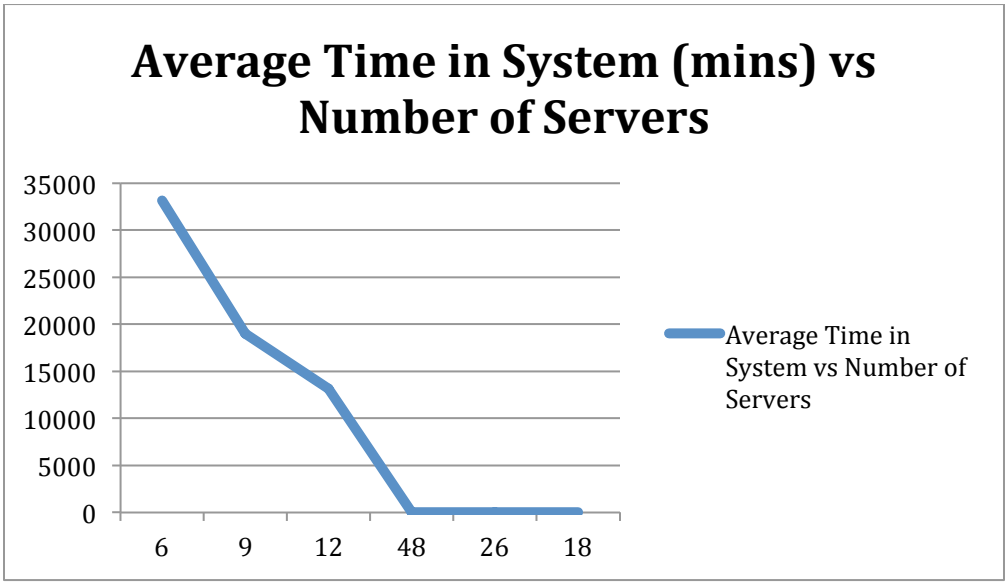
This data is further illustrated in the following graph.



X-Axis: Number of servers, ordered by experiment number.
Y-Axis: Number of refunds after 90 days of Simulation

So, there's no payoff when using more than 18 servers. Refunds hit zero at this point, and stay that way.

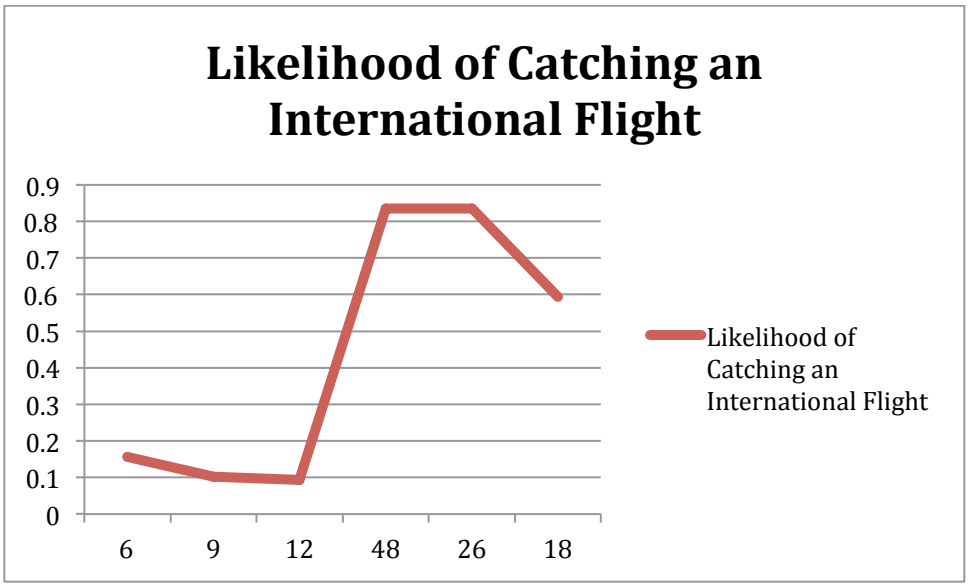
The underlying reason no one gets a refund after staffing 18 servers is further shown here:



X-Axis: Number of servers, ordered by experiment number.
Y-Axis: Average time in system (mins) after 90 days of Simulation

The actual value for average system time after 90 days for is **51 minutes**. And this is the value at which the system stabilizes. I confirmed this by running a year-long simulation as well. This means we never issue refunds because the wait time is safely below 90 minutes, and it also seems like a reasonable amount of time to wait, overall.

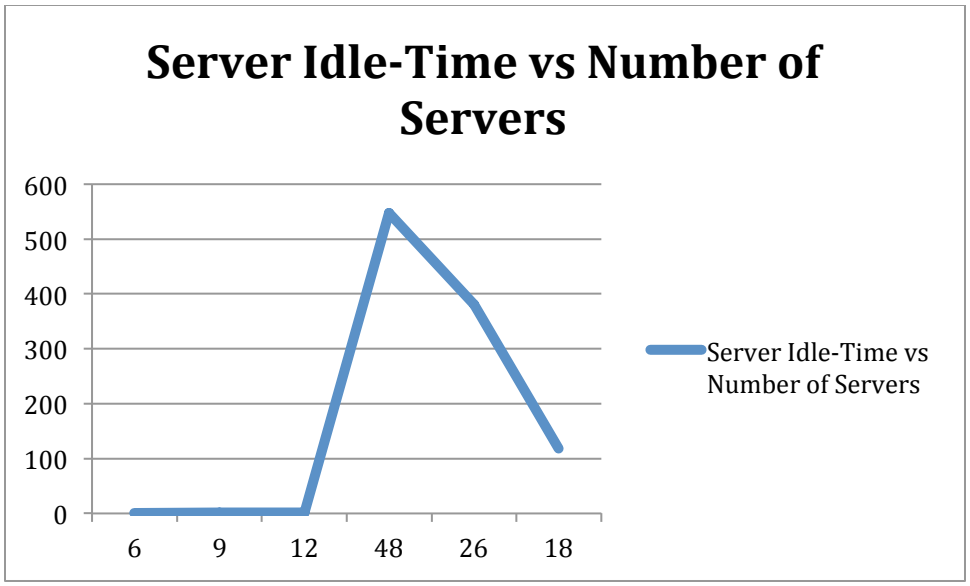
The main area of concern I'd have is in the likelihood of catching an International Flight:



X-Axis: Number of servers, ordered by experiment number.
Y-Axis: Probability of Catching Flight, as an International Passenger

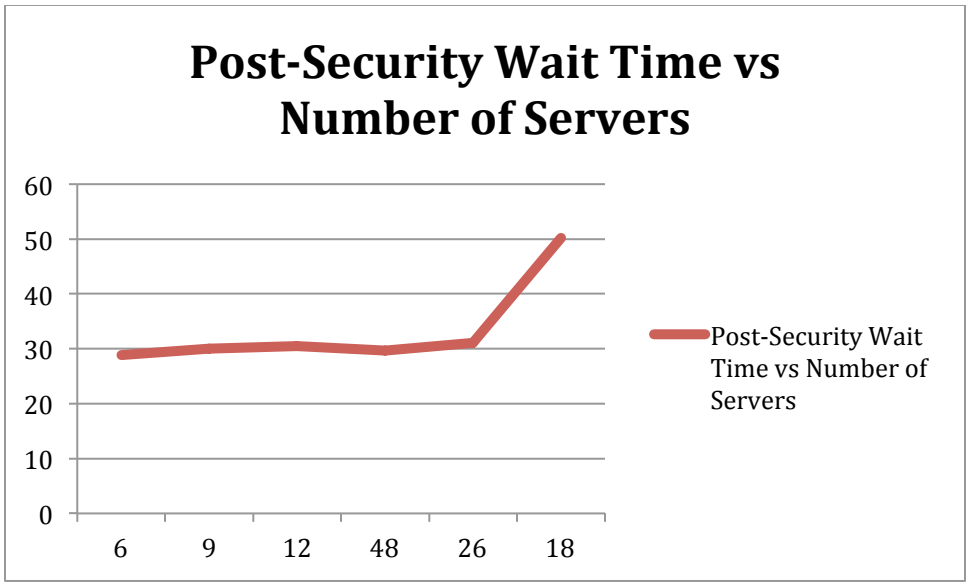
We start to see a downturn in this likelihood after decreasing the number of servers to 18—**but**, we do NOT see an increase in the number of refunds issued. It remains steadily at zero. This because the average wait time for all passengers is below 90 minutes. (126 for first class, and 51 overall). **So while fewer passengers make their flight, it's only the passengers who are more than 90 minutes late who ever miss.** And that does NOT affect profitability, so we have no real incentive as an airline to worry about this.

The other key metrics to watch for are also at reasonable levels with 18 servers. **Server Idle time is not at its lowest** (120 minutes per day), but even at 120 minutes per day, it's not affecting our profitability any more than issuing refunds would be. So I think it is a fair tradeoff.



X-Axis: Number of servers, ordered by experiment number.
Y-Axis: Server idle time (mins/day)

Average Post-Security Wait Time is also reasonable. There's actually an overall uptick in wait time when we transition to 18 servers, in the last experiment. But this is a mark of overall system efficiency, if anything. These customers are being shuttled through the system quickly, and a little bit more wait time at the back end of process seems preferable to waiting in line for very long periods.



X-Axis: Number of servers, ordered by experiment number.
Y-Axis: AVG Post-Security Wait time for Commuter Passengers (mins)

Increasing the frequency of commuter flights could further optimize this wait time, but at under an hour, this seems very reasonable. And moreover, increasing flight frequency would hurt our bottom-line profitability, rather than help it. **We're not**

at the point where supply is outpacing demand, so there's no need to increase flight frequency.

Recommendations

System Parameters

Taking this data into account, my recommendation would be to use the parameters outlined in experiment 6:

- Check-in Station
  - 1 Server for First Class
  - **8 Servers for Coach**
- Security Station
  - 1 Server for First Class
  - **8 Machines For Coach**
- Departures
  - Commuters: Every 60 minutes

At this point, we are highly profitable, passengers make it through the system quickly, and passengers are not waiting un-reasonably long at any stage. Furthermore, while there is some idle time for our servers, it does not affect our bottom line profitability any more than issuing more refunds would be. And since this money would be going to employees who can use it to help build a career and feed their families, I think it's preferable to spend this money on staffing rather than refunds.

This plan requires adding 5 more security machines and going over our maximum number of check-in agents, but I think it is the best balance.

Simulation Running Time

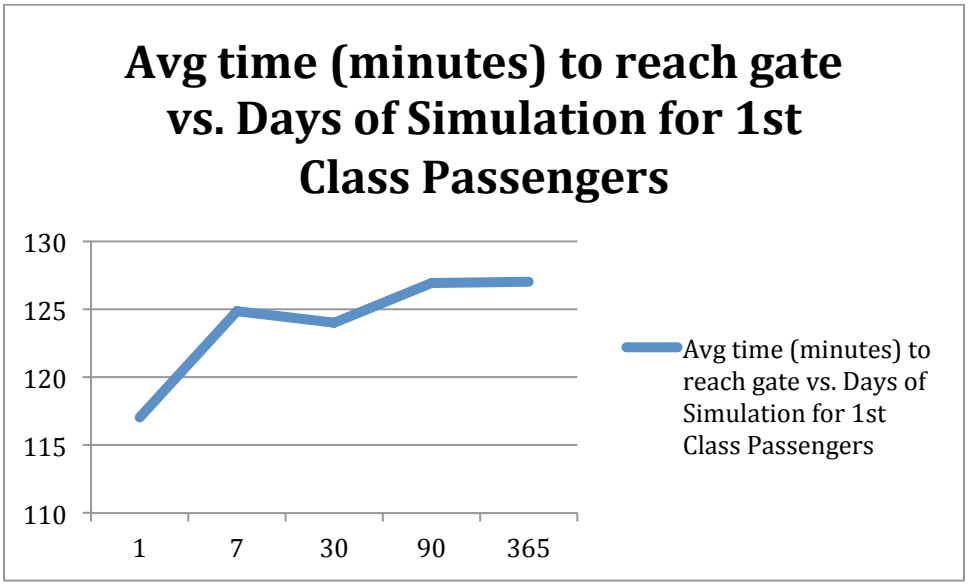
Based on my experiments**, I would recommend running the simulations for 90 days.** At this point the system has stabilized, or we can tell that queues are building infinitely. For all of my data sets (fully available in the attached spreadsheet in this package), my wait times were stable after 90 days, and my other values were too.

For this reason, I don't think there's much to gain from running the simulation for longer. But we DO need to run the simulation for at least 90 days to get a clear picture of what's going on. And because there's not much difference in output from 90 days and 365 days (a full year), I think it's fine to start gather statistics immediately. The conditions of the empty system are quickly remedied (after a day or two), and this anomaly doesn't affect the long-term averages with any real significance.

Moreover, I would recommend **running 2-3 trials of these 90–day experiments** to confirm your results. Most of my data is from a single trial, just for brevity, but I ran each trial about 5 times to ensure that the data collected was representative. I

found that there isn't much significant difference between trials after 90 days, so in practice, I used the values from one 90 day long experiment for most of my comparisons at the end of this process, and that's annotated in the text above. Overall, after 90 days we get a good idea of what's going on in the system, so for me, that was the optimal experiment time.

A quick visual confirmation of this can be seen in First Class arrival times, which are stable over all experiments:



Please note that this stabilizes around day 90 and does not change thereafter.