HW3

April 1, 2019

1 Homework 3

Collaborators: * Olivier Cloux (236079) * Zoé Baraschi (219665)

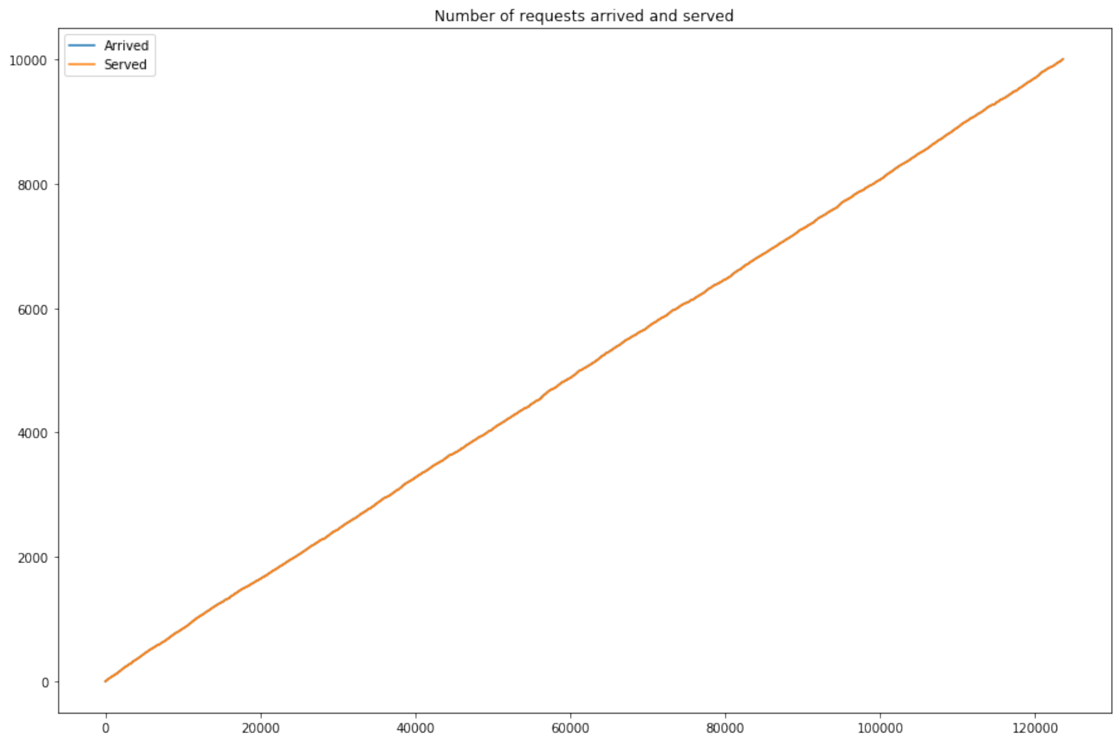
```
In [1]: import numpy as np
        %matplotlib inline
        import matplotlib.pyplot as plt
        from datetime import datetime
        import pandas as pd
```

2 Simulate

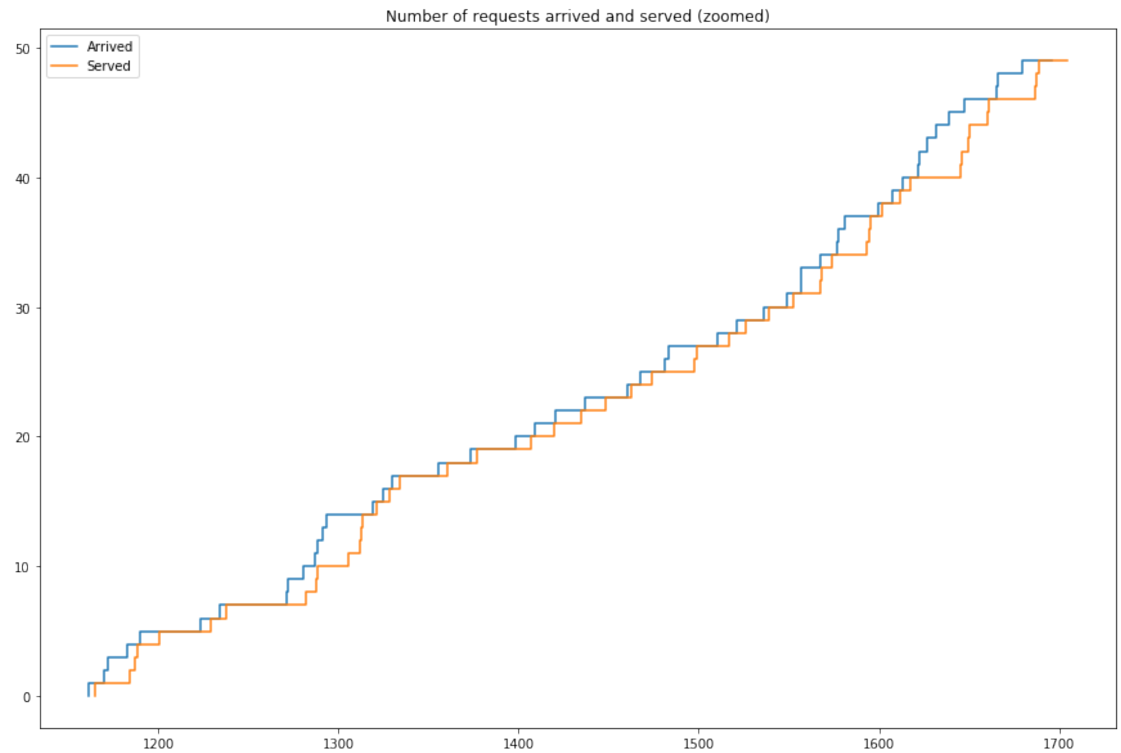
```
In [2]: from helpers import compute

In [3]: df = compute(80)

In [4]: plt.figure(figsize=(15,10))
        plt.plot(df['Start'], range(len(df)), label="Arrived")
        plt.step(df['T22'], range(len(df)),label="Served")
        plt.title("Number of requests arrived and served")
        plt.legend()
        plt.show()
```



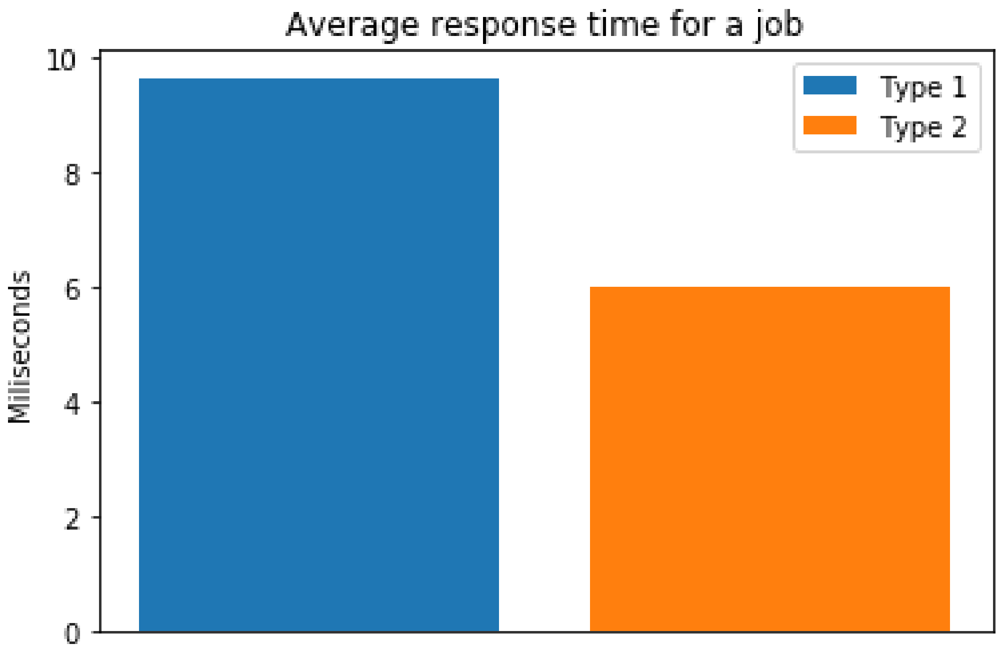

```
In [5]: plt.figure(figsize=(15,10))
        plt.step(df['Start'][100:150], range(50), label="Arrived")
        plt.step(df['T22'][100:150], range(50), label="Served")
        plt.title("Number of requests arrived and served (zoomed)")
        plt.legend()
        plt.show()
```



2.0.1 average response time (event average)

```
In [6]: #time difference between end of task, and beginning of task
        # for each type of task
        df['T1-Serv'] = df['T12']-df['Start'] # serv time for type 1
        df['T2-Serv'] = df['T22']-df['T12']  # serv time for type2
        df['Tot-Serv'] = df['T22']-df['Start']  # serv time for type2
```

```
In [7]: plt.bar([1],[df['T1-Serv'].mean()], label="Type 1")
        plt.bar([2],[df['T2-Serv'].mean()], label="Type 2")
        plt.title("Average response time for a job")
        plt.xticks([])
        plt.ylabel('Miliseconds')
        plt.legend();
```



2.0.2 Average number of jobs served per second

```
In [8]: served = []
        for sample_start_time in (df.values.max()-1000)*np.random.sample(100):
            served.append(df[(df['T21'] > sample_start_time) &
                             (df['T22'] < sample_start_time+1000)
                            ].count()['Start'])
        print("Average number of Type 1 jobs served per second: {}".format(np.mean(served)))
```

```
Average number of Type 1 jobs served per second: 82.47
```

```
In [9]: served = []
        for sample_start_time in (df.values.max()-1000)*np.random.sample(50):
            served.append(df[(df['T11'] > sample_start_time) &
                             (df['T12'] < sample_start_time+1000)
                            ].count()['Start'])
        print("Average number of Type 2 jobs served per second: {}".format(np.mean(served)))
```

```
Average number of Type 2 jobs served per second: 79.9
```

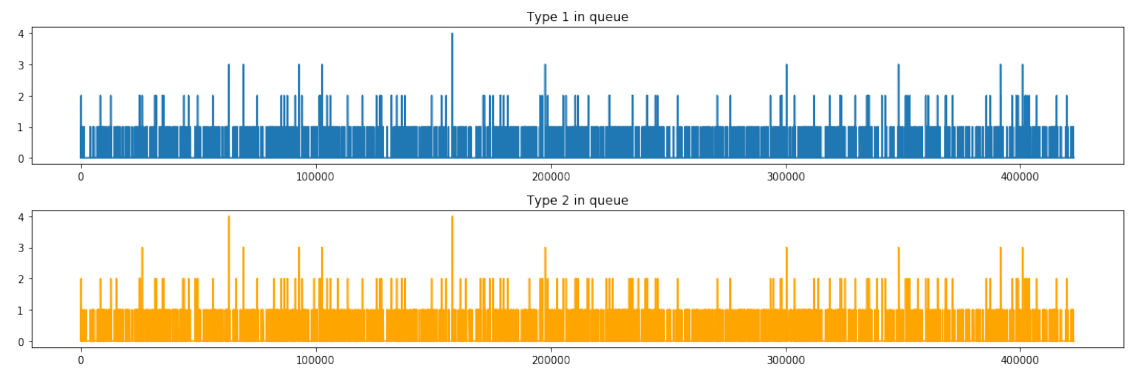
As we see, the number of jobs served per second is in both cases close to 80, as the server can keep up with the requests rate, of 80 requests/seconds. So the queue being most of the time almose empty, the rate is obviously close to 80.

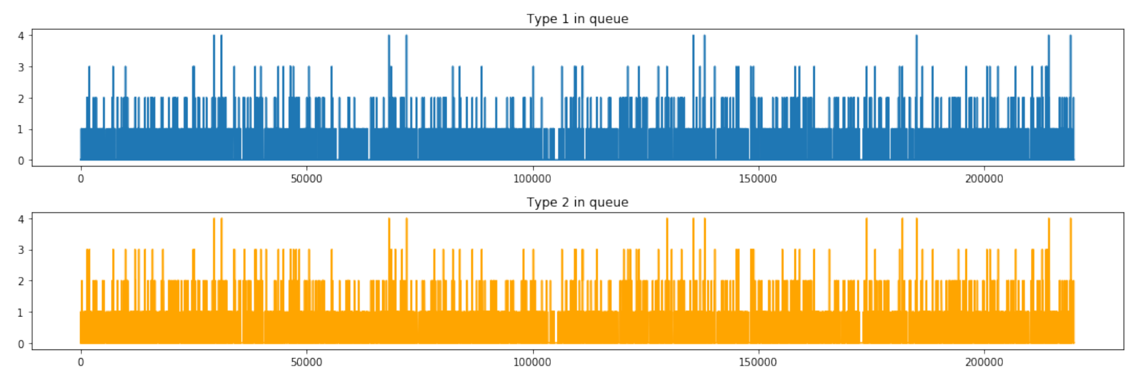
3 2

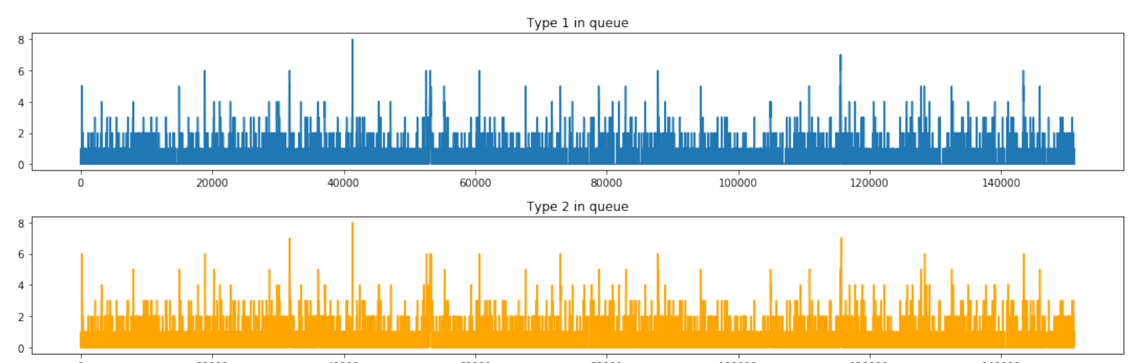
```
In [10]: lambdas = np.arange(25, 250, 25)
         queues = []
         items = []
```

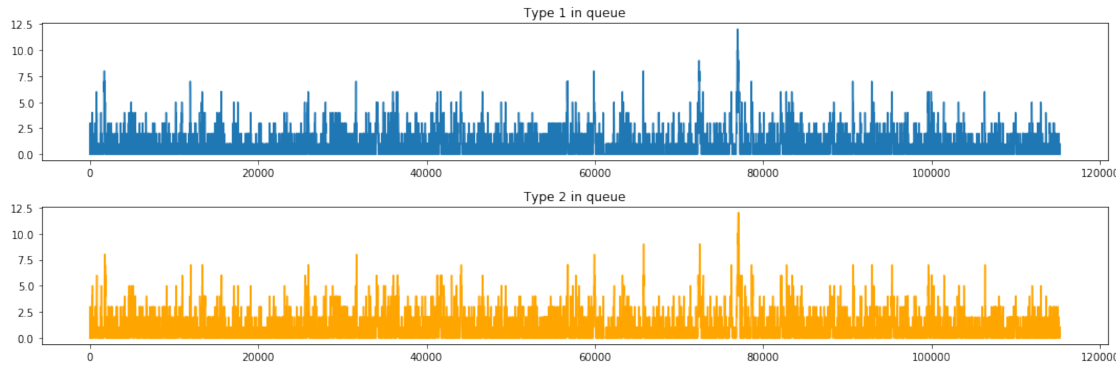
```
In [11]: from helpers import plot_and_save_jobs
```

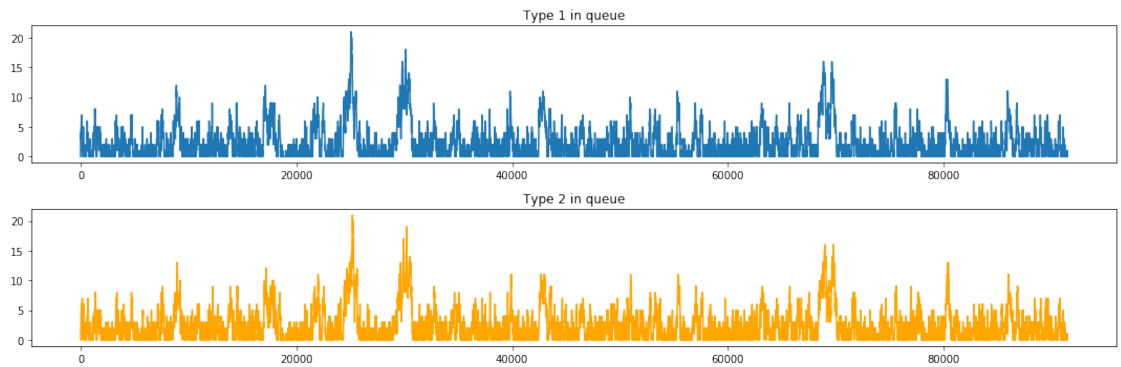
```
In [12]: for l in lambdas:
             df = pd.read_csv("data/lambda_"+str(l))
             plot_and_save_jobs(df, "fig/jobs_lambda_"+str(l))
```

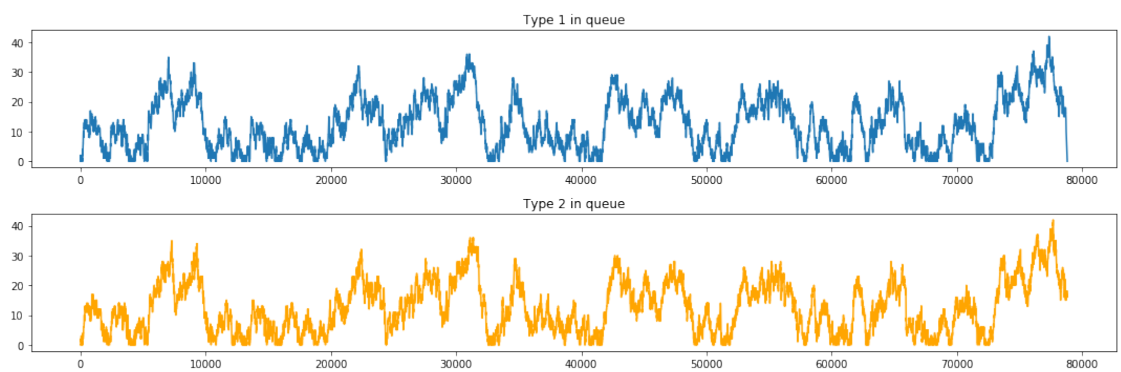


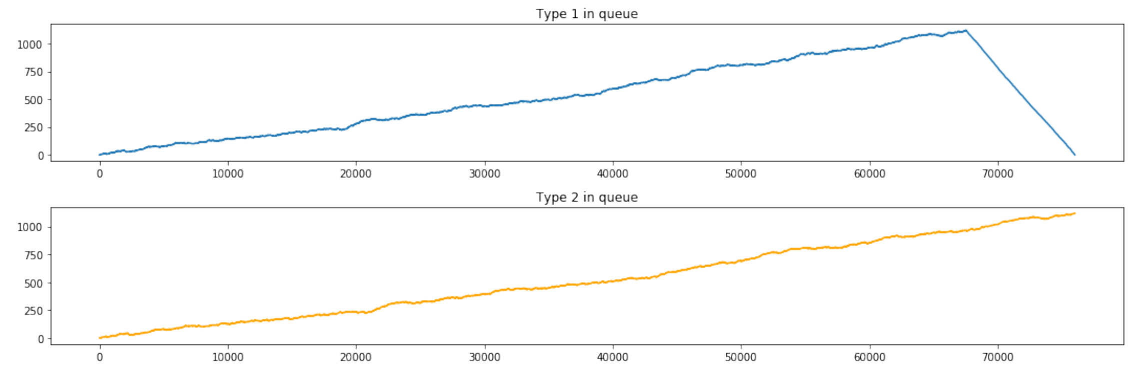


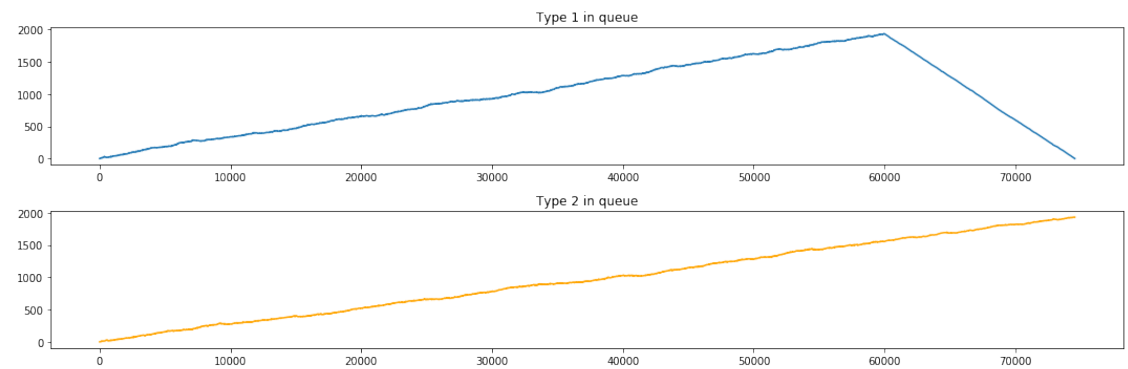


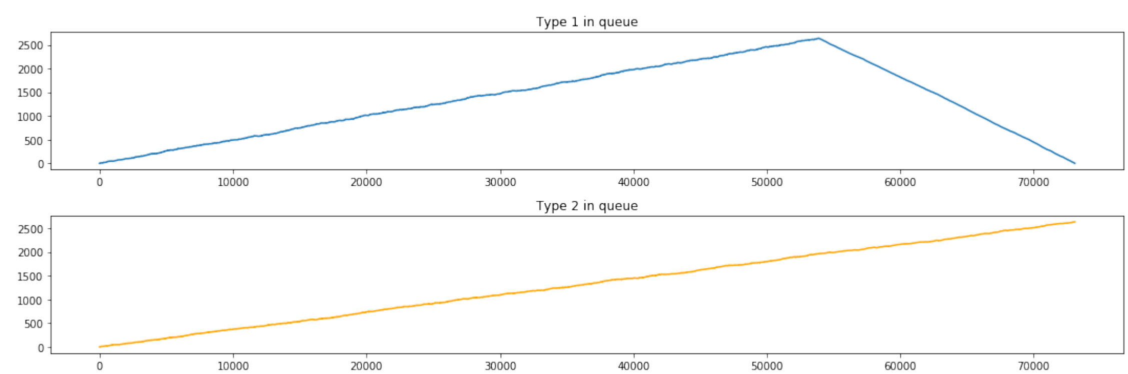

Type 1 in queue
Type 2 in queue


Type 1 in queue
Type 2 in queue


Type 1 in queue
Type 2 in queue







For which values of is the system stationary? The graphs show a stationary regime for lambda in the range (25, 150).

The system is stationary when the mean arrival time between 2 arrivals in the queue does not recede the sum of the mean of the log normal (1.5, 0.6) and the uniform distribution (0.6, 1).

As seen below, this is around 6.166 ms.

```
In [13]: m = np.exp(1.5+(0.6**2)/2) + 0.8
         print("Sum of mean of log-normal(1.5, 0.6)\
         and normal(0.6, 1) = ", m, "ms")
```

```
Sum of mean of log-normal(1.5, 0.6) and normal(0.6, 1) =  6.165555971121974 ms
```

After looping through the values, we see that the max lambda for which there is a stationary regime is 162, after that the queue is saturated and the walk to infinity begins.

```
In [14]: print("lambda = 162: ",1000/162, "requests/s,\
         lambda = 163:" , 1000/163, "requests/s")
```

```
lambda = 162:  6.172839506172839 requests/s, lambda = 163: 6.134969325153374 requests/s
```

What happens on the plots when the system is not stationary ? The system becomes unstable once the input rate is larger than the service capacity.

The buffer occupancy grows unbounded and the mean queue length increases. There is a walk to infinity.

4 3 Remove Transients

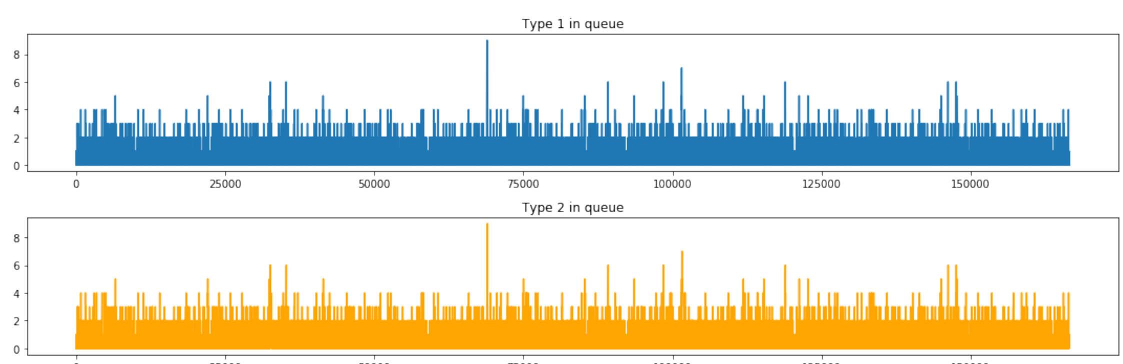
4.1 With transient present

4.1.1 $\lambda = 60$

```
In [15]: from helpers import get_type_i_in_queue, plot_type_i_in_queue
```

```
In [16]: df60 = compute(60, iterations=30)
         interesting_times60, type1_in_queue_60, type2_in_queue_60 = get_type_i_in_queue(df60)
```

```
In [17]: plot_type_i_in_queue(interesting_times60, type1_in_queue_60, type2_in_queue_60)
```



CI for median

```
In [18]: from helpers import ci_median
```

```
In [19]: l,h = ci_median(len(interesting_times60))
         a = np.sort(type1_in_queue_60)
         b = np.sort(type2_in_queue_60)
         print("The CI for median with lambda=60 are, \
         at 95% confidence, between\n\
         \ttype1: {} and {}\n\ttype2: {} and {}".format(a[l], a[h], b[l], b[h]))
```

```
The CI for median with lambda=60 are, at 95% confidence, between
        type1: 1 and 1
        type2: 0 and 1
```

CI for mean

```
In [20]: from helpers import ci_mean_large_n
```

```
In [21]: ci_means_type1 = ci_mean_large_n(type1_in_queue_60)
         ci_means_type2 = ci_mean_large_n(type2_in_queue_60)
         print("The CI for median with lambda=60 are, at 95% confidence,\
         between\n\ttype1: {:.3f} and {:.3f}\n\
         \ttype2: {:.3f} and {:.3f}".format(ci_means_type1[0], ci_means_type1[1],
                                            ci_means_type2[0], ci_means_type2[1]))
```

```
The CI for median with lambda=60 are, at 95% confidence,between
        type1: 0.565 and 0.868
        type2: 0.563 and 0.871
```

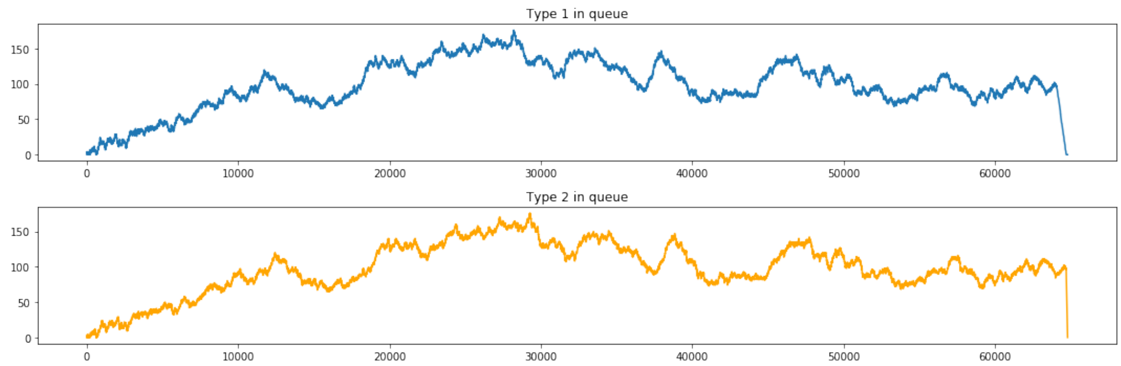
4.2 $\lambda = 160$

CI for median

```
In [22]: df160 = compute(160, iterations=30)

         interesting_times160,
         type1_in_queue_160,
         type2_in_queue_160 = get_type_i_in_queue(df160)

         plot_type_i_in_queue(interesting_times160, type1_in_queue_160, type2_in_queue_160)
```




```
In [23]: l,h = ci_median(len(interesting_times60))
         type1_160_sorted = np.sort(type1_in_queue_160)
         type2_160_sorted = np.sort(type2_in_queue_160)
         print("The CI for median with lambda=60 are, at 95% confidence,\
         between\n\ttype1: {} and {}\
         \n\ttype2: {} and {}".format(type1_160_sorted[l], type1_160_sorted[h],
                                      type2_160_sorted[l], type2_160_sorted[h]))
```

```
The CI for median with lambda=60 are, at 95% confidence,between
        type1: 95 and 96
        type2: 95 and 96
```

CI for mean

```
In [24]: ci_means_type1 = ci_mean_large_n(type1_in_queue_160)
         ci_means_type2 = ci_mean_large_n(type2_in_queue_160)
         print("The CI for mean with lambda=160 are, at 95% confidence,\
         between\n\ttype1: {:.3f} and {:.3f}\n\
         \ttype2: {:.3f} and {:.3f}".format(ci_means_type1[0], ci_means_type1[1],
                                            ci_means_type2[0], ci_means_type2[1]))
```

```
The CI for mean with lambda=160 are, at 95% confidence,between
        type1: -143.253 and 339.018
        type2: -141.591 and 337.356
```

4.3 With the transient removed

For $\lambda = 60$, the analysis is the same as before, as there seems to be no transient. Indeed, the buffer is most of the time empty or nearly empty, steadily accross time. Thus we do not repeat the calculation.

For $\lambda = 160$, we use as transient period the time until the buffer size reaches the mean of both objects (that is 97.884). Everything before this threshold is reached is considered transient period

```
In [25]: # Computing moment the threshold is first crossed
         buffer_size = np.array(type1_in_queue_160 + type2_in_queue_160)
         threshold = np.mean(buffer_size)
         threshold_crossing = np.argwhere(buffer_size > threshold)[0,0]
```

CI for median

```
In [26]: #computing median on reduced array
         med_l_ind, med_h_ind = ci_median(len(interesting_times160[threshold_crossing:]))
         #lower and higher for type 1
         med_l_type1 = type1_160_sorted[threshold_crossing+med_l_ind]
         med_h_type1 = type1_160_sorted[threshold_crossing+med_h_ind]

         med_l_type2 = type2_160_sorted[threshold_crossing+med_l_ind]
         med_h_type2 = type2_160_sorted[threshold_crossing+med_h_ind]
         print("CI for median, lambda=160, no transient, at 95% level:\n\
         \tType1: {} - {}\n\
         \tType2: {} - {}".format(med_l_type1, med_h_type1,
                                  med_l_type2, med_h_type2))
```

```
CI for median, lambda=160, no transient, at 95% level:
        Type1: 105 - 106
        Type2: 105 - 106
```

CI for mean

```
In [27]: ci_mean_l_type1, ci_mean_h_type1 = ci_mean_large_n(
             type1_in_queue_160[threshold_crossing:])
         ci_mean_l_type2, ci_mean_h_type2 = ci_mean_large_n(
             type2_in_queue_160[threshold_crossing:])
```

```
In [28]: print("CI for mean, lambda=160, no transient, at 95% level:\n\
         \tType1: {:.3f} - {:.3f}\n\
         \tType2: {:.3f} - {:.3f}".format(ci_mean_l_type1, ci_mean_h_type1,
                                  ci_mean_l_type2, ci_mean_h_type2))
```

```
CI for mean, lambda=160, no transient, at 95% level:
        Type1: -27.879 - 244.628
        Type2: -15.961 - 233.702
```

5 4 Little's Law

Little's Law: $\lambda\overline{R} = \overline{N}$, where λ is the number of customer's arriving per second, $\overline{R}$ is the average time a customer spends in the system and $\overline{N}$ is the average number of customers observed in the system.

```
In [29]: # |n - lambda* r| = 0
```

```
In [30]: #lambda = 60 (with and without transient is the same)
         df60['Tot-Serv'] = df60['T22']-df60['Start']  # serv time for type2
         r = df60['Tot-Serv'].mean()
         n = np.mean(type1_in_queue_60 + type2_in_queue_60)
         lambda_ = 60
         print(np.abs(n - lambda_*r))
```

```
696.2032113333277
```

```
In [31]: #without transient removal, lambda = 160
         df160['Tot-Serv'] = df160['T22']-df160['Start']  # serv time for type2
         r = df160['Tot-Serv'].mean()
         n = np.mean(type1_in_queue_160 + type2_in_queue_160)
         lambda_ = 160
         print(np.abs(n - lambda_*r))
```

```
203309.81944
```

```
In [32]: #with transient removal, lambda = 160
         df160['Tot-Serv'] = df160['T22']-df160['Start']  # serv time for type2
         r = df160['Tot-Serv'][threshold_crossing:].mean()
         n = np.mean((type1_in_queue_160 + type2_in_queue_160)[threshold_crossing:])
         lambda_ = 160
         print(np.abs(n - lambda_*r))
```

```
207013.93200567894
```

So apparently, Little's law doesn't verify here. This strongly suggests a mistake earlier, but unfortunately we are unable to spot it.

6 5 Parameter Estimation And Confidence Interval

A new request arriving in the system is of type 1 with probability $1-\epsilon$ and of type 2 with probability ϵ and we want to assume that ϵ is zero or almost zero.

First experiment: 10 random requests, all of type 1.

The confidence interval for p when we observe z=0 successes is $[0, p_0(n)]$, with $p_0(n) = 1 - (\frac{1-\gamma}{2})^{\frac{1}{n}}$.

For $\gamma = 0.95$ and $n = 10$, we have: $p_0(10) = 1 - (\frac{1-0.95}{2})^{\frac{1}{10}} = 0.308$

Confidence interval for ϵ: $[0, 0.308]$

Confidence interval for the stability region of the system:

Second experiment: In order to assure that $\epsilon < 1\%$ with a 95% confidence, assuming that a sample is always a type 1 request, we want $p_0(n) < 0.01$.

$1 - (\frac{1-0.95}{2})^{\frac{1}{n}} < 0.01$

$(0.025)^{\frac{1}{n}} > 0.99$

$\log 0.025^{\frac{1}{n}} > \log 0.99$

$\frac{\log 0.025}{n} > \log 0.99$
$\frac{-3.68888}{n} > -0.0100503$
$ n > 367.04$
We need to pick 368 samples in order to assure $\epsilon < 1\%$ with 95% confidence.

In [33]:
```
import math
print(1-math.pow(0.025, 0.1))
print(math.log(0.025)/math.log(0.99))
```

```
0.30849710781876083
367.0404161497511
```