1 Data retrieval

First of all, we need to retrieve the data from the simulation. For that, we first run the below script. It runs the simulation for every possible loads (request/s), first for 1 AP and 1 server, then for 1 AP and 2 server, and so on. Each measure is made 3 times, to avoid at most some “unlucky” or “too lucky” simulation that would distort the analysis.

```python
NUM_THREADS = 10
URL = "http://tcpip.epfl.ch/output.php"

class MyThread(threading.Thread):
    """
    Subclass to threading.Thread for multithreading purpose.
    """

    def __init__(self, ap, servers, thread_id):
        super().__init__()
        self.ap = ap
        self.servers = servers
        self.thread_id = thread_id
```
```python
self.data = []
self.client = 0

def run(self):
    for self.client in range(1, 1001):
        payload = {
            "sciper": "236079",
            "clients": self.client,
            "apoints": self.ap,
            "servers": self.servers
        }
        for k in range(3):  # take each measure 3 times
            resp = requests.post(URL, data=payload)
            soup = bs4.BeautifulSoup(resp.content)
            table = soup.find("table")
            rows = table.find_all("tr")
            data_point = []
            for row in rows:
                cols = row.find_all("td")
                cols = [ele.text.strip() for ele in cols]
                if not "values" in cols[0]:
                    data_point.append(cols)
            self.data.append(data_point)
            if self.client % 100 == 0:
                print("{{}} {} - {} - {{}}".format(self.thread_id, self.ap, self.servers, self.client))

    try:
        df = pd.read_csv('pandas_data.csv', index_col="index")
        df['Access points'] = pd.to_numeric(df['Access points'])
        df['Servers'] = pd.to_numeric(df['Servers'])
        empty = False
    except FileNotFoundError:
        df = pd.DataFrame()
        empty = True
    for ap in range(1, 11):  # for each AP configuration
        threads_list = []
        for num_thread in range(NUM_THREADS):
            servers = num_thread + 1
            if not empty:
                specific_df = df[(df['Servers'] == servers) & (df['Access points'] == ap)]
                if len(specific_df) >= 1000:
                    threads_list.append(specific_df)
        if threads_list:
            threads = Pool(NUM_THREADS)
            threads.map(worker, threads_list)
```

#if already enough data, don't launch thread
print("ignoring combination {}-{}".format(ap, servers))
continue
print("Launching new thread for {} AP and {} servers".format(ap, servers))
thread = MyThread(ap, servers, num_thread)
thread.start()
threads_list.append(thread)

#Join threads
for thread in threads_list:
    print("Joining thread {}".format(thread.thread_id))
data = thread.join()
#read joined data, and append to current DataFrame
for i in data:
    df = df.append(pd.Series([x[1] for x in i], [x[0] for x in i]),
                   ignore_index=True)
#save intermediate result
df.to_csv("pandas_data.csv", index_label="index")

1.1 Results exploration

In [3]: df.head()

Out[3]:
Access points Collision probability Delay Packets per second
index
0 1 0.000013 0.143996 3.245
1 1 0.000012 0.115127 0.000
2 1 0.000014 0.123098 0.000
3 1 0.000014 0.147046 3.894
4 1 0.000013 0.138540 9.439

Requests per second Sciper ID provided Servers Theta
index
0 1 236079 1 0.972
1 1 236079 1 0.944
2 1 236079 1 0.993
3 2 236079 1 1.943
4 2 236079 1 1.883

In [4]: df.describe()

Out[4]:
Access points Collision probability Delay
count 300000.000000 300000.000000 300000.000000
mean 5.500000 0.279587 155.781832
std 2.872286 0.419075 333.378683
min 1.000000 0.000009 0.021352
25% 3.000000 0.000261 0.623686
50% 5.500000 0.006141 2.555264
75% 8.000000 0.783901 30.798532
max 10.000000 1.000000 1082.849096
<table>
<thead>
<tr>
<th>Packets per second</th>
<th>Requests per second</th>
<th>Sciper ID provided</th>
</tr>
</thead>
<tbody>
<tr>
<td>count 300000.000000</td>
<td>300000.000000</td>
<td>300000.0</td>
</tr>
<tr>
<td>mean 2176.337510</td>
<td>500.500000</td>
<td>236079.0</td>
</tr>
<tr>
<td>std 1284.483024</td>
<td>288.675471</td>
<td>0.0</td>
</tr>
<tr>
<td>min 0.000000</td>
<td>1.000000</td>
<td>236079.0</td>
</tr>
<tr>
<td>25% 1206.448750</td>
<td>250.750000</td>
<td>236079.0</td>
</tr>
<tr>
<td>50% 2039.977000</td>
<td>500.500000</td>
<td>236079.0</td>
</tr>
<tr>
<td>75% 3120.555750</td>
<td>750.250000</td>
<td>236079.0</td>
</tr>
<tr>
<td>max 5523.136000</td>
<td>1000.000000</td>
<td>236079.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Servers</th>
<th>Theta</th>
</tr>
</thead>
<tbody>
<tr>
<td>count 300000.000000</td>
<td>300000.000000</td>
</tr>
<tr>
<td>mean 5.500000</td>
<td>404.318171</td>
</tr>
<tr>
<td>std 2.872286</td>
<td>249.965069</td>
</tr>
<tr>
<td>min 1.000000</td>
<td>0.930000</td>
</tr>
<tr>
<td>25% 3.000000</td>
<td>197.328750</td>
</tr>
<tr>
<td>50% 5.500000</td>
<td>366.695500</td>
</tr>
<tr>
<td>75% 8.000000</td>
<td>588.684000</td>
</tr>
<tr>
<td>max 10.000000</td>
<td>999.000000</td>
</tr>
</tbody>
</table>

2 Question 1

Clearly, making the same simulation multiple times will not yield every time the same results. As we can see two cells above, the “simplest” case (1 AP, 1 server, 1 request/second) yields different outputs (e.g. delays of 5.288, 11.005 and 0). As in every simulation, there are a lot of hidden factors at play. Hopefully, they are nothing but nuisance factors, and only create a small variance in every result, that is evenly distributed in each simulation.

3 Question 2

First, we need to make a script to easily plot and visualize the representation

In [5]: def plot_comparison(df, nums_servers, nums_aps):
    """
    Plots multiple configurations on a single figure
    """
    Example:
    plot_comparison(df, [1,2], [3,4]) will create a graph with 2 configuration:
    first with 1 server and 3 access points,
    second with 2 servers and 4 access points.
    args:
    df -- pandas.DataFrame: dataframe where to select the data
    nums_servers -- list[int]: numbers of servers for each
    configuration to display (order matters)
nums_aps -- list[int]: numbers of access points for each configuration to display

```python
if len(nums_servers) != len(nums_aps):
    print("Error, length of configurations must be the same")
    return

ylabel = {
    "Theta": "Theta [requests/s]",
    'Collision probability': "Collision probability [%]",
    'Delay': "Delay [ms]",
    'Packets per second': "Packets per second [packets/s]"
}

fig = plt.figure(figsize=(15,10))
for num, metric in enumerate(ylabel):  # iterate through metrics
    fig.add_subplot(2,2,num+1)  # iterate through configurations
    for serv, ap in zip(*nums_servers, nums_aps):
        # get data
        df_temp = df[(df['Servers'] == serv) & (df['Access points'] == ap)]
        plt.scatter(df_temp['Requests per second'],
                    df_temp[metric] * (100 if "Collision" in metric else 1),
                    s=0.5,
                    label="{} serv / {} AP.".format(serv, ap))

    if metric == "Theta":  
        # plot ideal theta line if needed
        plt.plot(np.arange(1000), label="Ideal Theta")

    plt.title("{}".format(metric))
    plt.ylabel(ylabel[metric])
    plt.xlabel("Load [requests/s]"
    plt.legend(markerscale=9.)
    plt.tight_layout()

In [6]: plot_comparison(df, [1], [1])
```
3.1 First observations

3.1.1 Delay

We see that the delay first is consistently small, then grows exponentially until reaching an horizontal asymptote. The asymptote is at 1’000 ms, because above that the request is dropped, otherwise we would have an ever increasing load (as each request is re-made every second). Ideally, we would like to keep the delay as low as possible (ideally close to 0)

3.1.2 Collision probability

As before, the collision probability is close to 0%, but at some threshold (between 120 and 160 requests/s) it explodes and comes consistently close to 100%. Again, we are wishing for a probability close to 0%

3.1.3 Theta

As expected, the number of requests served is linear and follows the “ideal” line: (almost) as many requests per second are queried and served. But once a threshold around 200 is reached, the delay augments (see 3rd plot), and thus the number of served requests per seconds can’t follow the load. Finally, a second threshold around 800 is reached, and now the server is under too much pressure, and can’t handle it: requests get dropped, but too much is lost in overhead. The service is degrading proportionally to the load, reaching the asymptote at 200 (the “max” it can handle)
### 3.2 Conclusions

It is hard to conclude anything serious with only one configuration. What we see, is that between 180 and 200 requests per second, everything starts to break and takes a new form. We’d need a second configuration to see which factor is a bottleneck at 180 requests/s.

### 4 Question 3

Now we compare our first results with a second configuration: still one server, but 2 access points.

In [7]: `plot_comparison(df, nums_servers=[1,1], nums_aps=[1,2])`

![Graphs showing comparison between configurations](image)

This yields a lot of information. * First of all, the least surprising result: the collision probability never reaches the same heights as before. It “only” goes up to 90%, then goes down. This is typical to a CSMA/CD protocol, where when a collision is detected, the first sender will wait for some time to avoid future collision. * This is coherent with the Delay graph, showing a greater delay: when a collision is detected, the access point will wait until it can transmit again. The delay is thus going up quicker, but the collision stays low. * From ~600 requests/second, the collision probability starts going up, as the delay is already too high and the protocol will transmit again, ignoring the collisions.

So doubling the number of access points is not an perfect solution, but we do have some enhancements.

Only from this, we can suppose that * with more access points, the theta grows for longer before “crashing”, * The delay augments with the access points, * Packets/s grow linearly for
longer, but “crash harder” when their limit is reached. *Collision probability benefits more access points.

We try to those claims with some more simulations:

In [8]: plot_comparison(df, [1,1,1,1], [1,2,3,4])

It becomes clear that more access points will decrement the collision probability goes down, no matter what. But if one more access points “unlocked” some performances, they are now locked, probably bottlenecked by the server count, as every access point count > 1 follows the same performances in all the metrics (except collision probability)

In order to understand better, we plot two new configurations to compare: 2 servers / 1 AP and 2 servers / 2 AP

In [9]: plot_comparison(df, [1,1,2,2], [1,2,1,2])
With this, we can draw some thumb rules: * A server can serve up to ~380 requests per second  
* But an AP can only serve ~190 requests per second  

The other interesting observation we can do, is that the green configuration has the same collision probability than the blue one. So this is not a function of the servers number, but only the access points (as already asserted before)  

Then, we see in the Delay graph that the “heaviest” solution (2 AP and servers) does not lead to the best result. The more servers compared to the number of AP, the least delay we’ll obtain,
Indeed, the green configuration seems to cover the needs in terms of Theta. and packets/s. The orange configuration is mainly sufficient for 600 requests/s, as predicted.

To reduce some delay, we can add servers. We take the orange configuration as a reference.

In [11]: plot_comparison(df, [1,2,3,4], [1,4,4,4])
As predicted the delay has gone down with the number of servers, while the collision probability barely changes. Only change there: at the “break point” of our orange reference configuration, instead of absorbing the collisions and going down, it stays high.

Varying the count of access points (in reference to our 2 serv / 4 AP configuration), we obtain the following results:

**In [12]:** plot_comparison(df, [1,2,2,2], [1,4,6,8])
Which matches our expectations.

6 Conclusion

The engineering rule we can give to Joe, is our rule from above:

$$#\text{servers} = \frac{\text{load}}{380} \quad AP = 2 \cdot \text{servers}$$

This is the minimal configuration to have the maximum throughput (Theta and packets/s). If Joe wished to reduce the delay of each request, he can then augment the number of servers, and if he wishes to reduce the collision probability, he can augment the count of access points.

7 Bonus: slider of sliders to compare configurations

In [13]: @interact(n_weights=(1,7,1))
    def nice_plotter(n_weights):
        weight_sliders = [(IntSlider(
            value=1,
            min=1,
            max=10,
            step=1,
            description='serv config %d' % i,
            disabled=False,
        ) for i in n_weights]
continuous_update=False,
orientation='horizontal',
readout=True
), IntSlider(
    value=1,
    min=1,
    max=10,
    step=1,
    description='AP config %d ' % i,
    disabled=False,
    continuous_update=False,
    orientation='horizontal',
    readout=True
) for i in range(n_weights)]
serv_kwargs = {'server{}{}'.format(i):s_slider
                             for i, (s_slider, _) in enumerate(weight_sliders)}
ap_kwargs = {'ap{}{}'.format(i):ap_slider
                             for i, (_, ap_slider) in enumerate(weight_sliders)}
kwargs = {**serv_kwargs, **ap_kwargs}

def wrapper(**kwargs):
    serv = [kwargs[x] for x in kwargs if 'server' in x]
ap = [kwargs[x] for x in kwargs if 'ap' in x]
plot_comparison(df, serv, ap)

ip = interactive(wrapper, **kwargs)
for i in range(n_weights):
    display(HBox([ip.children[i],
                      # show controls in pairs
                      ip.children[(i+n_weights)%((n_weights*2)])]))
    display(ip.children[-1])#Show the output

interactive(children=(IntSlider(value=4, description='n_weights', max=7, min=1), Output()), _dom_classes="wid13")