2020

Assignment Four

DATA7202

ROBERT WILLIAMSON (S4575121)

Table of Contents

Question 1

This question concerns an example from Efron and Tibshirani (1993).

When a drug company introduces new medications, they are sometimes required to show "bioequivalence". Bioequivalence demonstrates that the new drug is not substantially different than the current treatment.

The table below shows eight subjects who used medical patches to infuse a certain hormone into the blood. Each subject received three treatments: placebo, old-patch, new-patch.

Table 1: results from eight subjects that used medical patches to infuse a certain hormone into their blood. Each subject received treatment using a placebo, a new-patch and the old-patch.

subject	placebo	old	new	old – placebo	new – old
1	9,243	17,649	16,449	8,406	-1,200
2	9,671	12,013	14,614	2,342	2,601
3	11,792	19,979	17,274	8,187	-2,705
4	13,357	21,816	23,798	8,459	1,982
5	9,055	13,850	12,560	4,795	-1,290
6	6,290	9,806	10,157	3,516	351
7	12,412	17,208	16,570	4,796	-638
8	18,806	29,044	26,325	10,238	-2,719

Let:

- Z = old – new
- Y = new – old

The Food and Drug Administration (FDA) requirement for bioequivalence is that $|\theta| \leq 0.20$, where $\theta = \frac{\mathbb{E}[Y]}{\mathbb{E}[Z]}$.

Plug-in estimate for θ

The "plug-in" estimate for θ is -0.0713 for the data given in Table 1.

Bootstrap

Based on the outcome of the plug-in estimate from Table 1 alone, the drug satisfies bioequivalence, $|-0.07| = 0.07 \leq 0.2$. However, the bootstrap method can be used to determine a confidence interval for $\hat{\theta}$ to determine if the result is reliable.

The bootstrap method is used with B=1000 replications to determine the 95% bootstrapped confidence interval (CI) of $\hat{\theta}$.

The bootstrapped mean and standard – deviations are -0.060 and 0.104 respectively. The 95% CI is:

$$(-0.060 - 1.96 \times 0.104, -0.060 + 1.96 * 0.104) = (-0.26, 0.14)$$

The bootstrapped 95% CI of -0.26 to 0.14 for bioequivalence contains a portion outside of the FDA requirement, so it cannot be stated with 95% confidence that $|\theta| \leq 0.20$. On this basis the new treatment should probably be rejected.

Question 2

Problem summary and objective

Air Secure are undertaking an investigation to determine the resourcing requirement to meet customer satisfaction standards. They have determined that the standard is to ensure in the long run that 90% of customers do not wait longer than 8 minutes before being served.

Air Secure will conduct a discrete event simulation supported by research outcomes of customer service demand and customer behaviour to determine the resource requirement to satisfy this standard.

Objective

Determine the minimum number of services desks that *Air Secure* must make available to ensure that 90% of customers do not wait longer than 8 minutes to be served.

Customer Service Demand

Demand is a combination of expected inter-arrival times of customers and the time required to service them. Research by *Air Secure* has provided a sample of 1,000 customer inter-arrival times and 1,000 customer service times, see Figure 1 for histograms of these parameters.

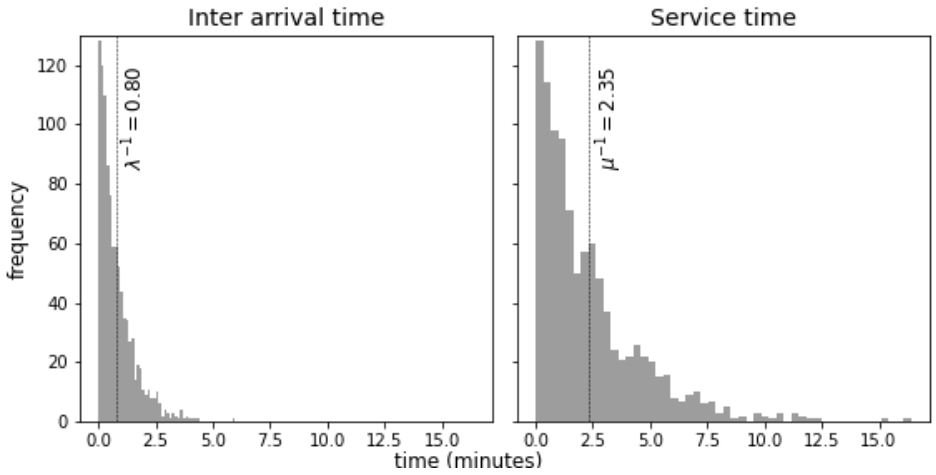

Figure 1: Histograms of the inter-arrival times and service times collected by *Air Secure*. λ^{-1} represents the expected inter-arrival time and μ^{-1} is the expected service time.

From analysis of the inter-arrival and service times the minimum resource requirement can be estimated to ensure the expected service time is less than the expected inter-arrival time.

$$\mu^{-1} < \lambda^{-1}, or\ \lambda < \mu$$

Taking the inverse of the expected inter-arrival and service times gives $\lambda = 1.25$ and $\mu = 0.425$, so to satisfy $\lambda < n_{min}\mu$, where n_{min} is the minimum number of desks, we would need at least 3 desks for the system to be stable. Simulation will determine the requirement to meet the customer satisfaction standard.

Customer behaviour

Further research by *Air Secure* has determined that on arrival customers will always choose the smallest queue and remain in that queue until being served.

This will be important in determining how customers are allocated to desks in the simulation.

Study variables

Inputs

Inter-arrival time and service time are provided as samples and can be used as empirical cumulative distribution functions (ECDF) as given in (1) and plotted in Figure 2.

$$\hat{F}(x) = \frac{\sum_{i=1}^{n} I\{X_i \leq x\}}{n} \quad (1)$$

Where $\boldsymbol{I}$ is an indicator variable given by (2) and n is the number of samples.

$$\boldsymbol{I}(\boldsymbol{x}) = \begin{cases} 1, X_i \leq x \\ 0, X_i > x \end{cases} \quad (2)$$

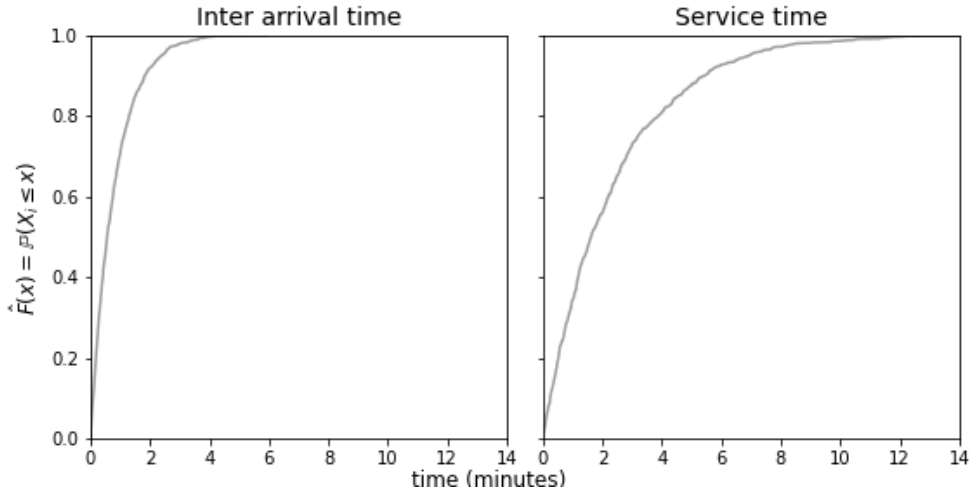

Figure 2: ECDFs of inter arrival and service times of customers

Customer service indicator

Customer queue time is used to determine if customer standards are adhered to for a range of scenarios. Each scenario is defined by the number of service desks made available. Queue time is derived from the time stamps recorded when a customer joins and leaves a queue. This is used to derive an indicator variable I_{cs} defined in (3).

$$\boldsymbol{I_{cs}}(\boldsymbol{y}) = \begin{cases} 1, Y_i \leq 8 \\ 0, Y_i > 8 \end{cases} \quad (3)$$

Where Y_i is customer waiting time in minutes. Taking the expectation of $\boldsymbol{I_{cs}}$ will provide an estimate of the probability of customers waiting no more than 8 minutes to be served. This is given in (4). Finally, the 95% confidence interval (CI) can be calculated via (5) and (6).

$$\hat{\ell} = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{I_{cs\,i}} \quad (4)$$

$$S^2 = \frac{1}{N-1}\sum_{i=1}^{N}\left(\boldsymbol{I_{cs\,i}} - \hat{\ell}\right)^{\mathbf{2}} \quad (5)$$

$$\left(\hat{\ell} - 1.96 \times \frac{S}{\sqrt{N}}, \hat{\ell} + 1.96 \times \frac{S}{\sqrt{N}}\right) \quad (6)$$

Where N is the number of customers served.

System Dynamics

A discrete event simulator (DES) was developed for *Air Secure* with the following key considerations of customer behaviour and parameters:

1. Generate customers at inter-arrival times from the data provided
2. Distribute customers to desks with the smallest queue, or else evenly distribute customers
3. Service customers with service times from the data provided
4. Record queuing and service times to derive $\hat{\ell}$ with a 95% CI

Figure 3 shows a process flow diagram of events that occur in the DES and includes an example system state at a snapshot in time (representative only). In the DES diagram customers are implemented as “clients” and service desks as “servers”.

Initialise System

Upon starting the simulation, system components are initialised including:

- **An event queue:** to keep track of scheduled events and prioritise the event with the earliest scheduled time
- **Several service desks:** n service desks are created with a unique id, a queue, a pointer to the event queue and the service time data
- **A distributor:** to distribute customers to the service desk with the smallest queue if none are available, or uniformly random otherwise.
- **An event logger:** to record events. E.g. when a customer is added to or removed from a queue or service desk
- **A network:** to store pointers to the system components, the entry point for customers and record their connections. I.e. where customers should flow from and to.

The entry point for customers is the distributor to determine what service desk customers choose.

The system is set to run for simulation time $T = 3000$ minutes as per the project specifications. All time units are in minutes.

Schedule first event

The first event is scheduled by generating a customer at $t = 0 + t_1$, where t_1 is sampled from the inter-arrival times provided. This is scheduled as an “*inject*” event which will “*put*” the customer in the distributor at time t_1. Events are scheduled by putting them in the event queue.

Event loop

The loop will run until system time t is greater than T. It takes the event at the head of the event queue and executes the actions of the event according to the event details.

Generally, after an event is executed another event will be scheduled or consequential events will occur. E.g. after a customer is *put* into a service desk the customer’s finish time will be scheduled in the event queue and after the event loop *gets* a customer from a service desk, the next customer will be *put* into service.

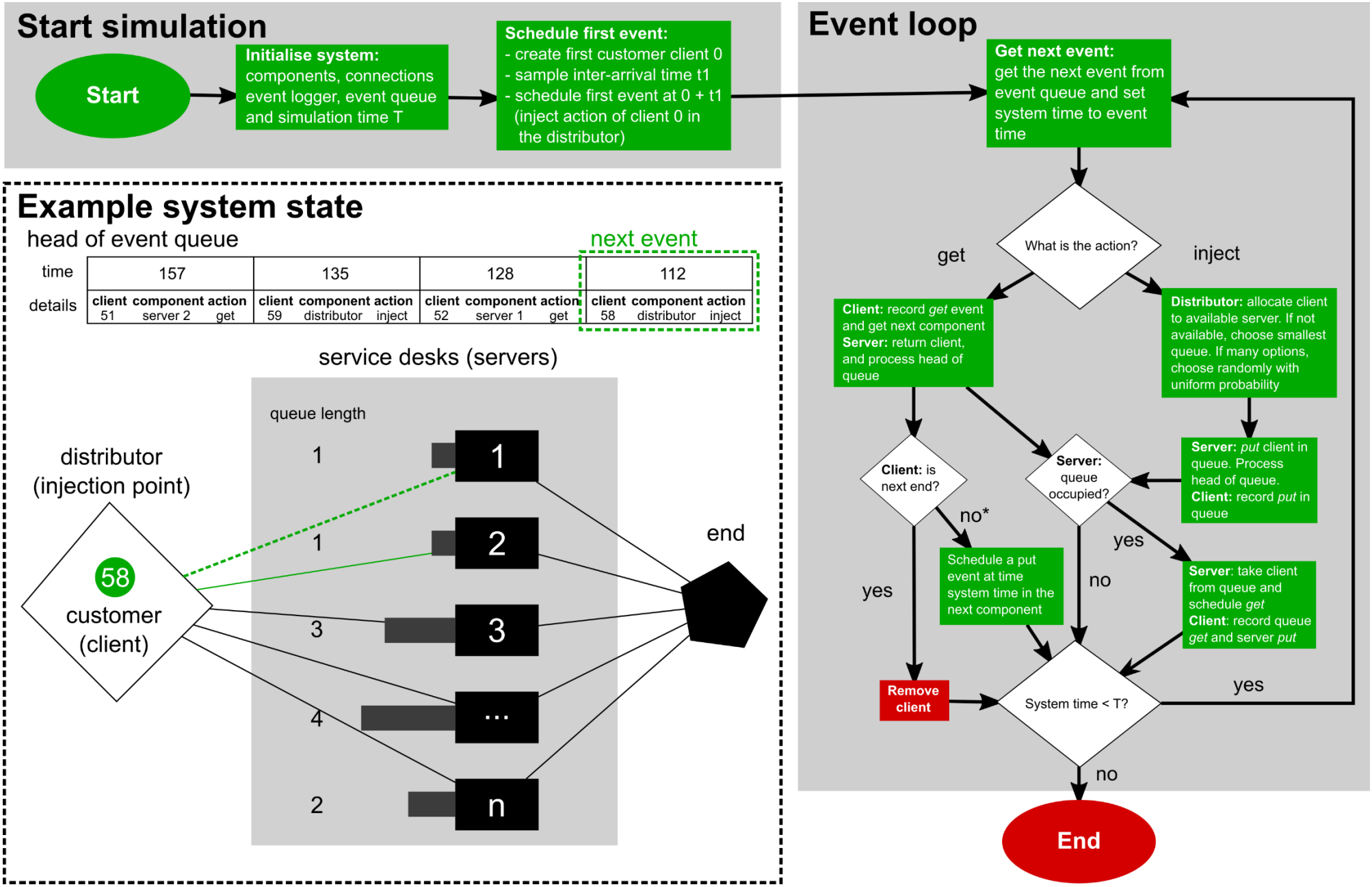

Figure 3: Diagram of the discrete event simulation system dynamics. Including an example system state showing the physical representation of connected nodes

* For the *Air Security* configuration only inject and get events are scheduled into the event queue. On *inject*, the distributor will allocate a client to a service desk. The service desk will schedule a *get* and because all the service desks are connected to an end node, after a *get* is executed from the service desk the client is removed from the system.

Results and analysis

Figure 4 shows the simulation results for a range of service desks. The left part of the figure provides ECDFs for customer waiting time with 1 to 10 service desks available. The right part of the figure shows $\hat{\ell}$ with a 95% CI (probability of waiting 8 minutes or less for service) versus the number of desks available.

The results show that 4 service desks should be made available to ensure that at least 90% of wait no more than 8 minutes for service.

Further, the results confirm our analysis of the expected inter-arrival and service times that at least 3 desks are required for system stability. As, the ECDFs for queuing time for scenarios with 1 and 2 service desks have very little if not any customers serviced to the required standard.

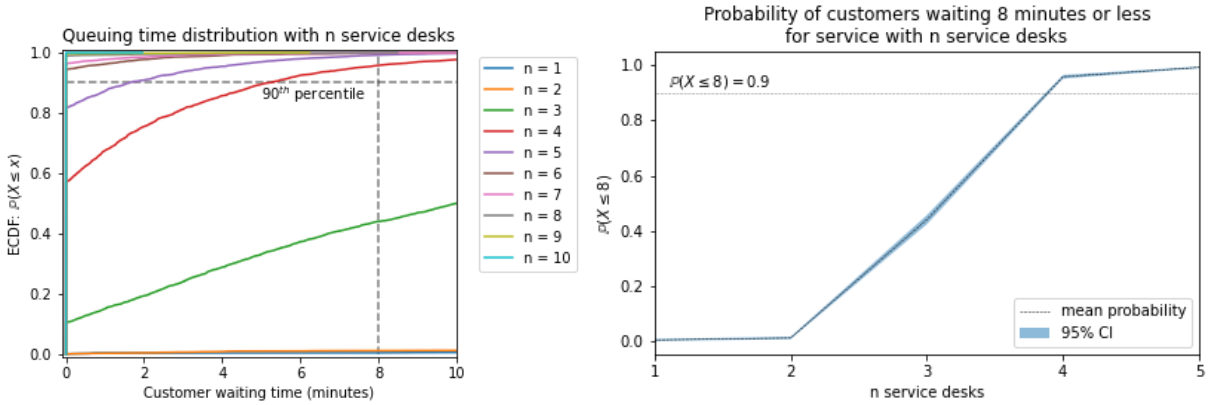

Figure 4: (left) ECDFs of customer queuing time for each scenario - the dotted lines show the 90th percentile line and 8 minute line. (right) Plot of the probability of customers waiting 8 minutes or less to be served versus the number of service desks.

Table 2 shows the 95% CIs for $\hat{\ell}$ and system throughput and utilisation. Here, we can see why *Air Secure* would like to understand the number of service desks to make available. Under a stable system with 3 desks, throughput is reasonable with 3,572 customers serviced in 3,000 minutes. With only two desks throughput drops by 29% to 2,527. To meet customer satisfaction standards with 4 desks, throughput remains high, but utilisation drops to 74%. This means there is a cost trade-off of paying for additional service staff and desks for un-utilised time to achieve customer service standards. Increased resourcing to 5 desks would goes above and beyond with customer service ensuring 99% of customers wait no longer than 8 minutes, but resources are under-utilised at reduces 59%.

Table 2: Probability of customers waiting 8 minutes or less for service with a range of service desks made available, throughput and utilization.

Number of desks	Probability of waiting 8 minutes or less (95% CI)	Throughput (customers served)	Utilisation (service time / total time)
1	0.00, 0.01	1,294	100%
2	0.01, 0.02	2,527	100%
3	0.42, 0.46	3,572	95%
4	0.95, 0.96	3,693	74%
5	0.99, 0.99	3,717	59%
6	1.00, 1.00	3,673	47%
7	1.00, 1.00	3,824	44%
8	1.00, 1.00	3,696	37%
9	1.00, 1.00	3,777	33%
10	1.00, 1.00	3,693	28%

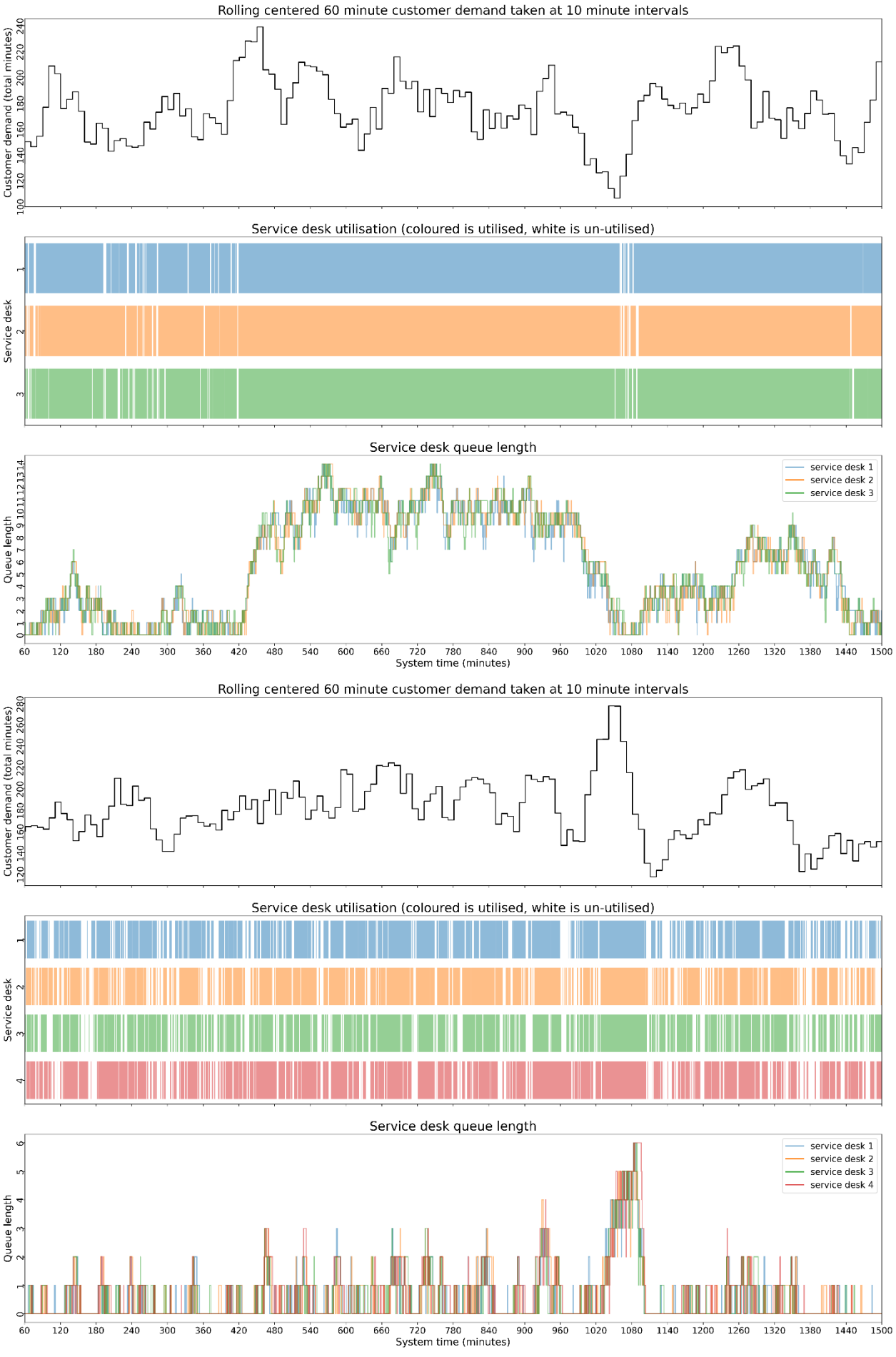

Figure 5: System snapshots showing customer demand, service desk utilisation and queue length over 24 hours. Two configurations are shown with 3 desks (top) and 4 desks (bottom)

System snapshots

Figure 5 shows snapshots of the system dynamics for 3 desks (top) and 4 desks (bottom). While the system is stable with 3 desks, long queues tend to form up to 14 customers deep and utilisation is very high with infrequent and small breaks in operation in periods of low demand only.

However, with 4 desks, queue lengths remain low reaching 6 customers deep at one point during a period of high demand and there are longer more frequent breaks in operation.

Figure 5 also indicates even utilisation and queue length of desks, demonstrating the system is behaving as expected.

Conclusion

Air Secure determined that 90% of customers should not have to wait longer than 8 minutes for service as a minimum customer service standard. Simulation was undertaken to determine the minimum number of service desks required to guarantee this standard.

Based on the simulation outcomes and system assumptions provided, *Air Secure* should ensure that 4 service desks are made available.

The simulation results indicate with 95% confidence that with 4 service desks between 95% and 96% of customers will be serviced after waiting no longer than 8 minutes. If only 3 desks are made available, performance reduces to between 42% and 46%.

4 desks provides a good balance to achieve the required customer service standards and maintain reasonable utilisation of resources and assets. Increasing to 5 desks reduces utilisation to 59% without any added benefit.

Appendix

Discrete event simulator code files

BabySim is a simple simulation tool developed to undertake the *Air Secure* discrete event simulation.

Table 3: Summary of discrete event simulator modules in BabySim

Module	Description
Main file *babysim/run.py* Executes actions in the event queue and schedules the next action	Initiation This file initiates that system parameters and runs the simulations. 10 scenarios are developed with 1 to 10 servers made available for customers. First an event queue is instantiated, then n servers are instantiated each with a simple queue, an id and pointers to the service time samples and the event queue. Next a distributor is instantiated with the list of servers. A dictionary of server ids and pointers is created as nodes along with a list of origin destination tuples representing their connections. These are used to instantiate a network representing the *Air Secure* system. Start simulation The function *run_simulation* is executed with the system and event queue as input. This function schedules the first event and executes the event loop. First the simulation run time T is set and the event logger is instantiated along with the first client. The client is instantiated with a unique id and a pointer to the event logger to record events (actions) as they occur. Finally, the first inject is scheduled and the simulation enters the event loop. Event loop The event loop repeats while t < T and takes the top event from the event queue. Events contain the time of the event and the event details. Details consist of a client, component and action, which is either: • **Inject** – *put* a client into the injection point (first node), create a new client, sample the inter-arrival time and schedule the next *inject* action. • **Put** – *put* the client in the component • **Get** – *get* a client from the component, *get* the next component from the network – if the next component is "end", remove the client or else schedule its *put* action in the next node (*puts* are never scheduled as the next node is always "end" in the *Air Secure* System) With each event the system time is set to the event time and the event loop will continue until $t \geq T$ – as the event time is set in the loop it will execute one final loop when $t \geq T$. After exiting the event loop, the event logs are returned and saved to file.
Network (system) *babysim/network/network.py* Contains system nodes (components) and edges (connections)	Represents the nodes and edges that make up the system. It is used to determine the injection point (where new clients enter the system when an *inject* action is scheduled) and the next node a client is to go after a *get* action is executed from a server and a *put* action is scheduled. The next node is returned by executing the *next_node* method.

Module	Description
Event queue *babysim/event_queue/event_queue.py* Priority queue that prioritises earliest event	A priority queue that maintains the time and details of scheduled events. Events are prioritised by the earliest time. Details include a client, component and action, to determine which client is actioned where. The event queue ensures non-concurrent events by incrementing a subsequent event by 1 nanosecond, if two events share an event time. Events are scheduled via the *put_event* method and the top event is returned via the *get_event* method.
Logger *babysim/event_logger/logger.py* Keeps a record of all *put* and *get* events	A global event logger – each client receives a pointer to the event logger that is accessed by the simple queue or server components to record when a client's action is executed with that component (i.e. *put* or *get*). Events are recorded via the *record* method by creating a dictionary with the time, client id, component id and action of each event. The dictionary is added to a list and all the records are returned as a pandas dataframe via the *get_logs* method.
Client *babysim/component/client.py* Customer representation	Created when an *inject* action is scheduled with a unique identifier and given a pointer to the logger to record *put* and *get* actions in components. It interacts with components through the *record* method to record events taking the event time, component id and action.
Distributor babysim/components/distributor.py Allocates clients to components when there is a choice	The distributor is instantiated with a list of servers it is connected to and interacts with the event loop via the *put* method. This method takes a client and event time. It randomly distributes customers to service desks when more than one is available. If not, the desk with the smallest queue will be allocated as per the customer behaviour findings. If more than one queue has the minimum length, a random choice will be made between those.
Server *babysim/components/server.py* Service desk representation	Represents a system component that processes clients. On creation a server is allocated a service time distribution, a queue and a pointer to the event queue. The service time distribution can be provided as an exponential distribution parameter μ or a sample of service times. Service times are chosen when a client goes into service, either uniformly at random from the samples provided or sampled from the exponential distribution. The server interacts with the system loop through the *put* and *get* methods. A *put*, puts the client straight in the queue and simultaneously processes the head of the queue. If the server is available, the client at the head of the queue is retrieved, it's *put* action is recorded and it's *get* action is scheduled in the event queue. If the server is in service, it stays in the queue. When a *get* is executed in the system loop, the event is recorded with the client and the client is returned to the event loop. If the simple queue is occupied, the head of the queue is processed to *put* the next client into service.
Simple queue *babysim/components/queue.py* Service desk queue representation	A container that holds clients as a first in first out queue. It has a *put* and *get* method. The *put* method takes a client and time, appends the client to the tail of the container and records a *put* event with the client. The *get* method takes a client from the head of the queue and records a *get* event with the client.

Analysis code files

Analysis code for questions 1 and 2 are contained in Assignment_4.ipynb. This includes analysis and visualisation of simulation inputs and results.