**Chapter 02: Modelling Complex Systems**

Table of Contents

[Time-Shared Computer Model 2](#_Toc87178287)

[Output Statistics 4](#_Toc87178288)

[Events 4](#_Toc87178289)

[State Variables 5](#_Toc87178290)

[Event Handlers 6](#_Toc87178291)

[Determining Events 10](#_Toc87178292)

[Unnecessary Events 11](#_Toc87178293)

[Initialization Edges 12](#_Toc87178294)

[Multi-Teller Bank with Jockeying 15](#_Toc87178295)

[State Variables 15](#_Toc87178296)

[Events 15](#_Toc87178297)

[Arrival Handler 16](#_Toc87178298)

[Departure Handler 17](#_Toc87178299)

[Jockeying 17](#_Toc87178300)

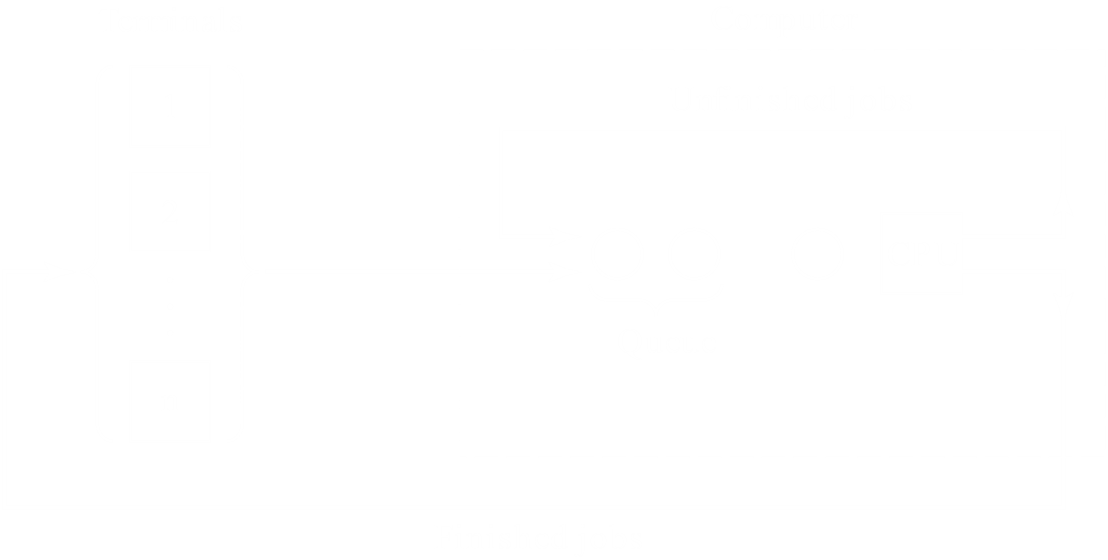
[Termination Handler 19](#_Toc87178301)

[Job-Shop Model 20](#_Toc87178302)

## Time-Shared Computer Model

A long time ago, there used to exist large **CPUs** that served several **terminals**. The terminals did not have any **resources** to themselves, but rather received the resources from the CPU.

In a **Time-Shared Computer Model**, each of the terminals submits their **jobs** to the CPU, where they join a **round-robin queue**.



In a round-robin queue, there is a **time quantum**, . Each job in the queue is given the specified amount of time, and after that time is finished, the job must stop running. If the job is **unfinished**, it is **paused** and placed at the **end** of the queue. If the job **finishes** within the time, it is sent back to its respective terminal. For our example, we will consider that .

We also need to consider the amount of time required to perform a **context switch**, called the **switching time**, . This is considered to be .

For this system, the **initial conditions** are assumed to be:

* The **terminals** are **idle**.
* The **queue** is **empty**.
* The **CPU** is **idle**.

Each terminal is operated by a **human operator**. Before they can **submit a job** using their terminal, they need to **think** for a while. The time they require to do this is called the **thinking time**. Let the thinking time of the th operator be . For this example, let us consider that .

After this time, the operator **creates a job** and submits it to the CPU. The jobs being created are considered to have a **service time**, , that has the distribution .

Since a job might not necessarily be finished in a **single run**, we also need to keep track of the **remaining service time** of each job, . Thus, initially, . At the end of each quantum, we check if . If it is, it means the job finished within that quantum and can be returned to its terminal. If it is not, then the job is still unfinished. In this case, is decremented by and the job is returned to the queue.

The simulation **terminates** after the first jobs.

### Output Statistics

We want to measure the **average response time**, which is the time between the moment a job is submitted to the CPU and the moment the job is returned to its terminal.

This is also the **waiting time** for the operator of that terminal, i.e. a new job is not submitted by the operator until the old one is returned.

It can be calculated for all jobs and for each of the terminals, which will help verify **fairness**, i.e. whether or not the average response times for each terminal are nearly identical to the average response time for the overall system. The system will be unfair if we use a priority queue.

We should generally try to keep the average response time below . We can vary the number of terminals to see how it affects the average response times and adjust our system accordingly.

Other output statistics we might be interested in include the **time-average number of jobs in the queue** and the **CPU utilization**.

### Events

The **events** are:

* Job Arrival Event
* End-of-Time-Quantum Event
* Termination Event

Notice that the **Start-of-Time-Quantum Event** is not being considered. This is because the start of a time quantum is triggered when a job arrives or departs.

If an arriving job finds the CPU idle, an End-of-Time-Quantum Event is scheduled after time. If, as a job is departing, it is seen that the queue is not empty, another End-of-Time-Quantum Event is scheduled after time.

Similarly, a **Departure** Event is not being considered either. At the end of each quantum, the serviced job will either leave the CPU or re-join the queue.

There can also be **multiple arrival events** simultaneously, from multiple terminals. This means, in our code, we cannot simply reschedule a single arrival event object repeatedly. We need to actually have  **terminal objects**, each with their own arrival event.

### State Variables

The **state variables** are:

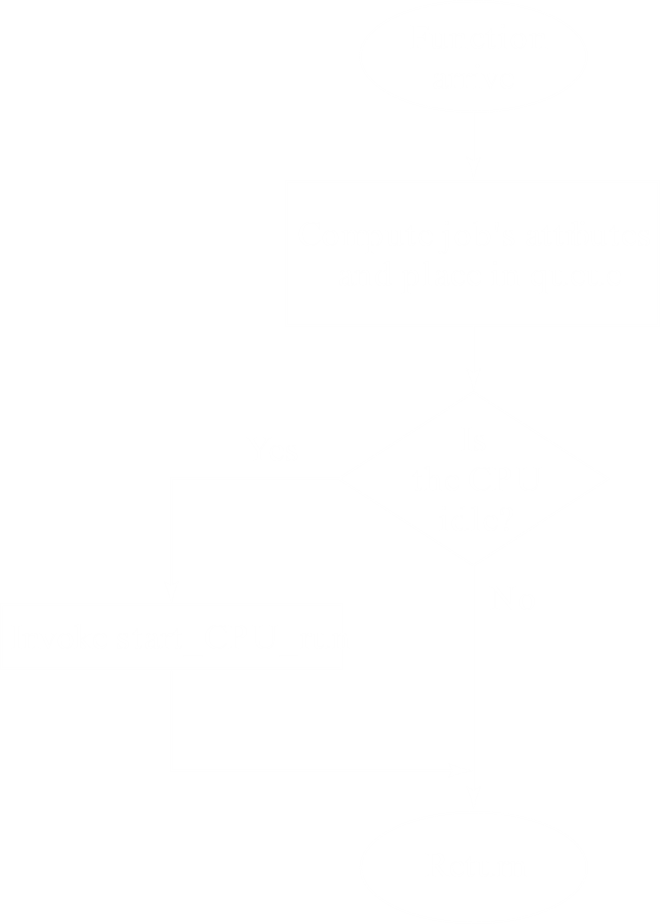
* CPU Status
* Queue Length

Each job must also maintain:

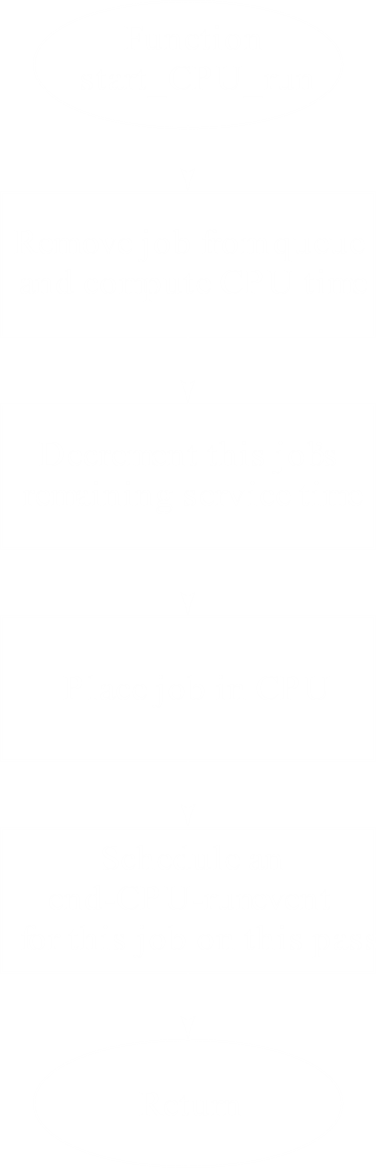
* Start Time
* End Time
* Remaining Time
* Total Time
* Total Queueing Delay

### Event Handlers

Arrival Event Handler:

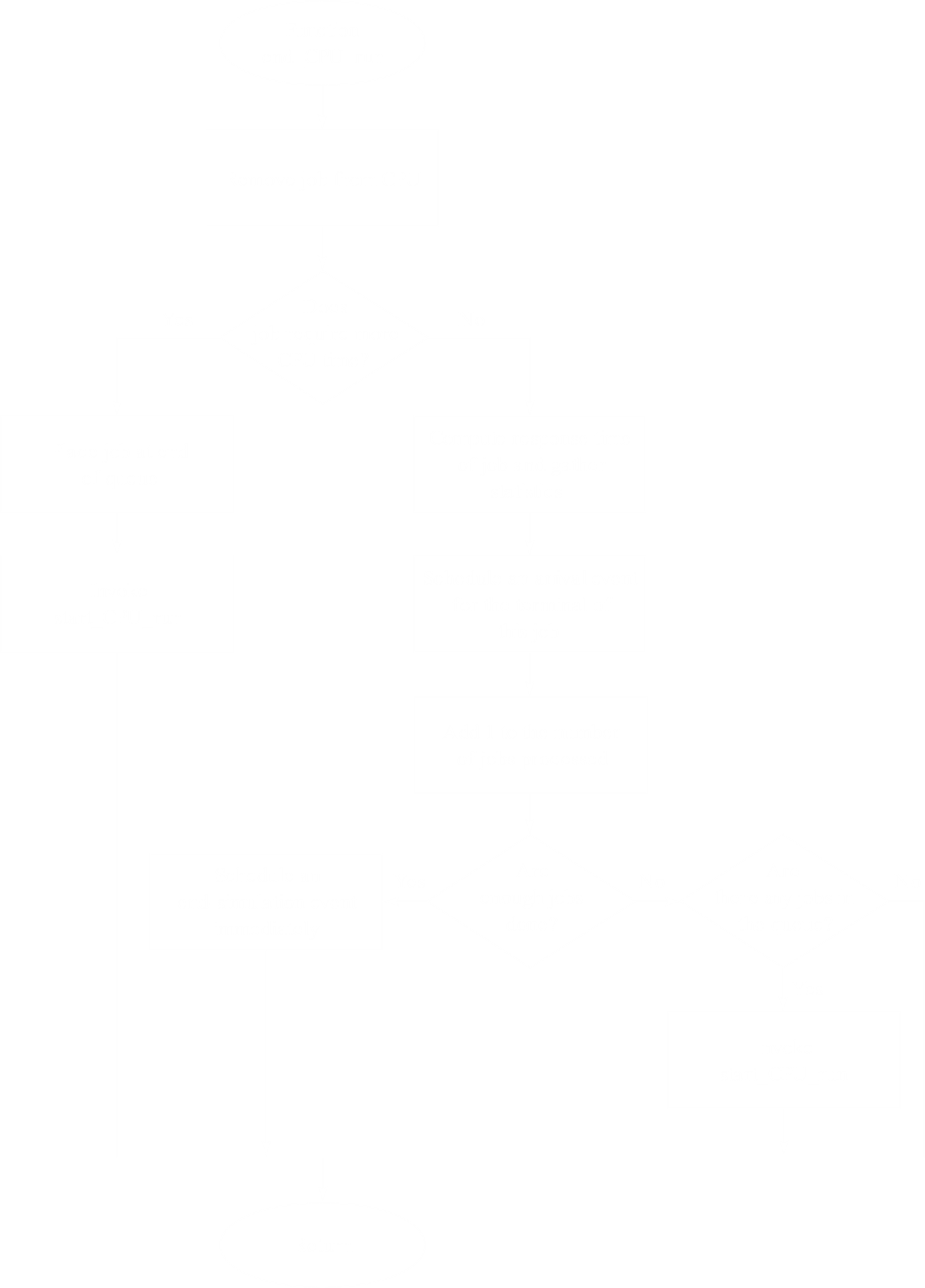


In this flowchart, we are **creating a job** and **placing it in the queue** first. Notice that an **arrival event** is **not scheduled**, since the terminal that sent this job must wait. Then we check if the **CPU is idle**, and if it is, we start the **time quantum**. It would also have been acceptable to check if the CPU is idle first and place the job in the queue if it was not.



Starting the **time quantum** causes the first job in the queue to be **dequeued**. That job’s **remaining service time** is decremented by the value . It is placed on the CPU, which for the simulation’s purposes means nothing, and the **End-Time-Quantum Event** is scheduled. The time for this event is actually , taking into account the **switching time** as well.

End-Time-Quantum Event Handler:



The End-of-Time-Quantum event is more complicated. We first check if the job requires more time, i.e. whether the **remaining service time** is more than . It if is, the job is placed at the **end of the queue** and the next time quantum is started.

If the job is **finished**, we calculate the statistics for the job and schedule an **arrival event** for the corresponding terminal. Then we check if we have processed the **maximum number of jobs**. If we have, we schedule a **termination event**. Otherwise, we check the **queue**, and if there is a job, we start the next time quantum.

## Determining Events

There are several ways in which we can make sure that the **events** that we are selecting for a particular simulation are correct. One of these is to use an **event graph**.

An event graph consists of events as **nodes** and **directed edges**, which represent the connections between events. If we have an edge from Event A to Event B, it means that Event A **schedule’s** Event B.



Note that by **schedule**, we mean that the event is placed in the event list of the scheduler, as opposed to **trigger**, which would mean that the event is fired, meaning the event handler is called.

Edges can be of three types:

1. A **thick edge** means that there is a delay between the moment an event is scheduled and the moment an event is triggered.



1. A **thin edge** means that the event is trigged instantaneously after it is scheduled.



1. A **jagged edge** means that the event is initialized at the start of the simulation.



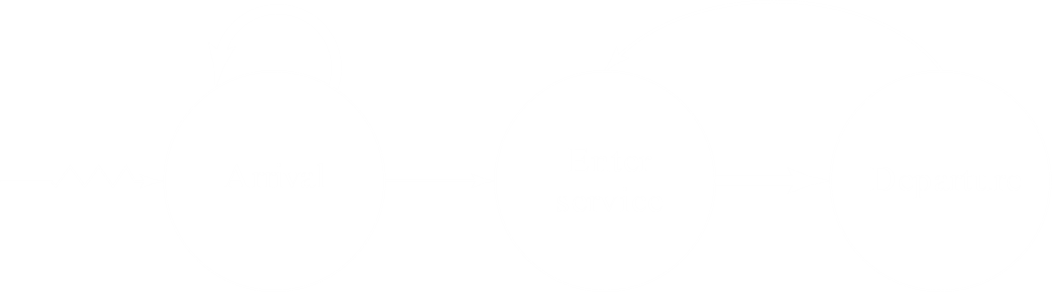
### Unnecessary Events

If we consider an **SSQS**, the event graph would look like this:



* An **arrival event** schedules another **arrival event**. There is a **delay**.
* An **arrival event** might schedule a **departure event**, in case the arriving item finds the **server idle**. Again, there is a **delay**.
* A **departing event** might schedule another **departing event**, if the **queue is not empty**. Again, there is a **delay**.
* At the **start** of the simulation, an **arrival event** is scheduled.

Remember that in an SSQS, we ignore the ‘event’ where an item **enters service**. This is because that ‘event’ is triggered **instantaneously** by one of the other two events. If an arriving item finds the server idle, it instantaneously enters service. If a departing item finds a non-empty queue, another item instantaneously enters service. But consider what would happen if we **did not ignore** that event.



* An **arrival event** schedules an **enter service** event, which triggers **instantaneously**.
* The **enter service event** schedules a **departure event**, which triggers after a **delay**.
* The **departure event** schedules an **enter service** event if the queue is not empty, which triggers **instantaneously**.

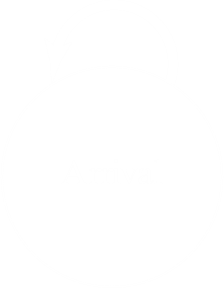
If the event graph still exists and is **accurate**, then how did we know that this extra event could be **eliminated**? The rule is that if **all incoming edges are thin and smooth**, then the event can be **eliminated**.

### Initialization Edges

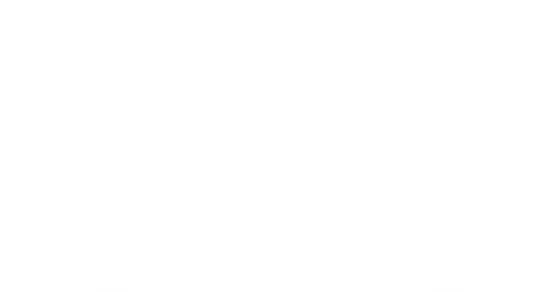
Another question is when do we add the **initialization edge**? To understand that, we need to define **strongly connected components**.

A strongly connected component is one where we can **travel from any node to all other nodes**. This essentially means that the component forms a **cycle**, but for single nodes, the node itself will suffice (without any self-loop).

If we consider the original graph for an SSQS, the arrival event alone forms a cycle and the departure event alone forms a cycle, so these are both strongly connected components.

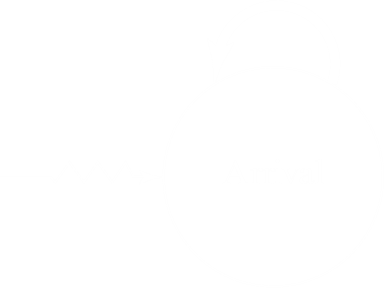
 

For the second graph of an SSQS, the enter service and departure events together form a cycle, so they form a single strongly connected component.



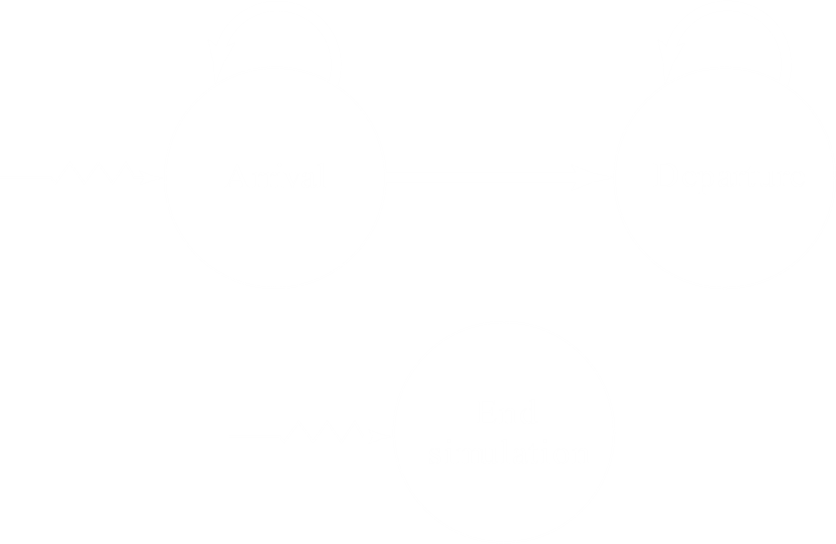
For a strongly connected component, if there are **no edges entering** the component **from the outside**, then we need to add an **initialization edge**.

For example, in the original graph of an SSQS, the departure event had an edge entering from outside, from the arrival event, so that did not require any initialization edge, but the arrival event did not, so that did.



The same situation applies for the second form of the SSQS graph we saw.

Note that just because an initialization edge **schedules** an event right at the start of the simulation, it does not necessarily mean that the event will be **triggered** immediately. It might even be triggered at the end, for example with a termination event.



## Multi-Teller Bank with Jockeying

We will now consider a scenario where customers are being served by **multiple servers**, each with their **own queue**. For our example, we will consider that there are **five servers**. In addition, the customers are allowed to **jockey**, i.e. move from one queue to another, if it seems that this will get them served faster.

The bank **opens at 9AM** and **closes at 5PM**. No customers are allowed **after 5PM**, but customers who are already inside **will be served**.

The **arrival times** are in the set , meaning the arrival events are **exponential distributed** with a mean of **1 minute**.

The **service times** are in the set , meaning the departure events are **exponential distributed** with a mean of **4.5 minutes**.

Note that this means and meaning .

### State Variables

* **Status** of the th server,
* **Queue length** of the th server,

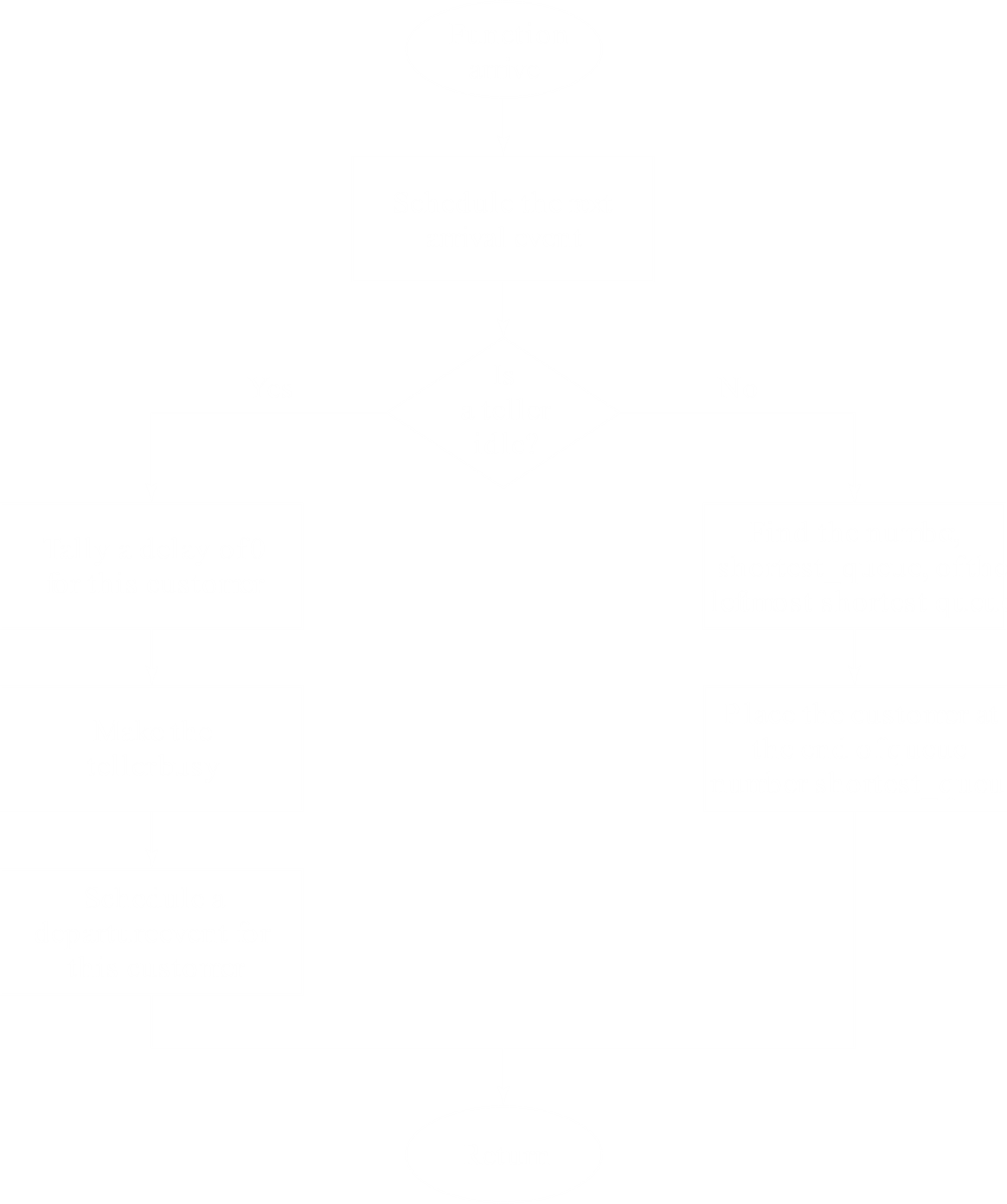
### Events

As before, there are just two events, an **arrival event** and a **departure event**. We also have the **termination event**.

### Arrival Handler

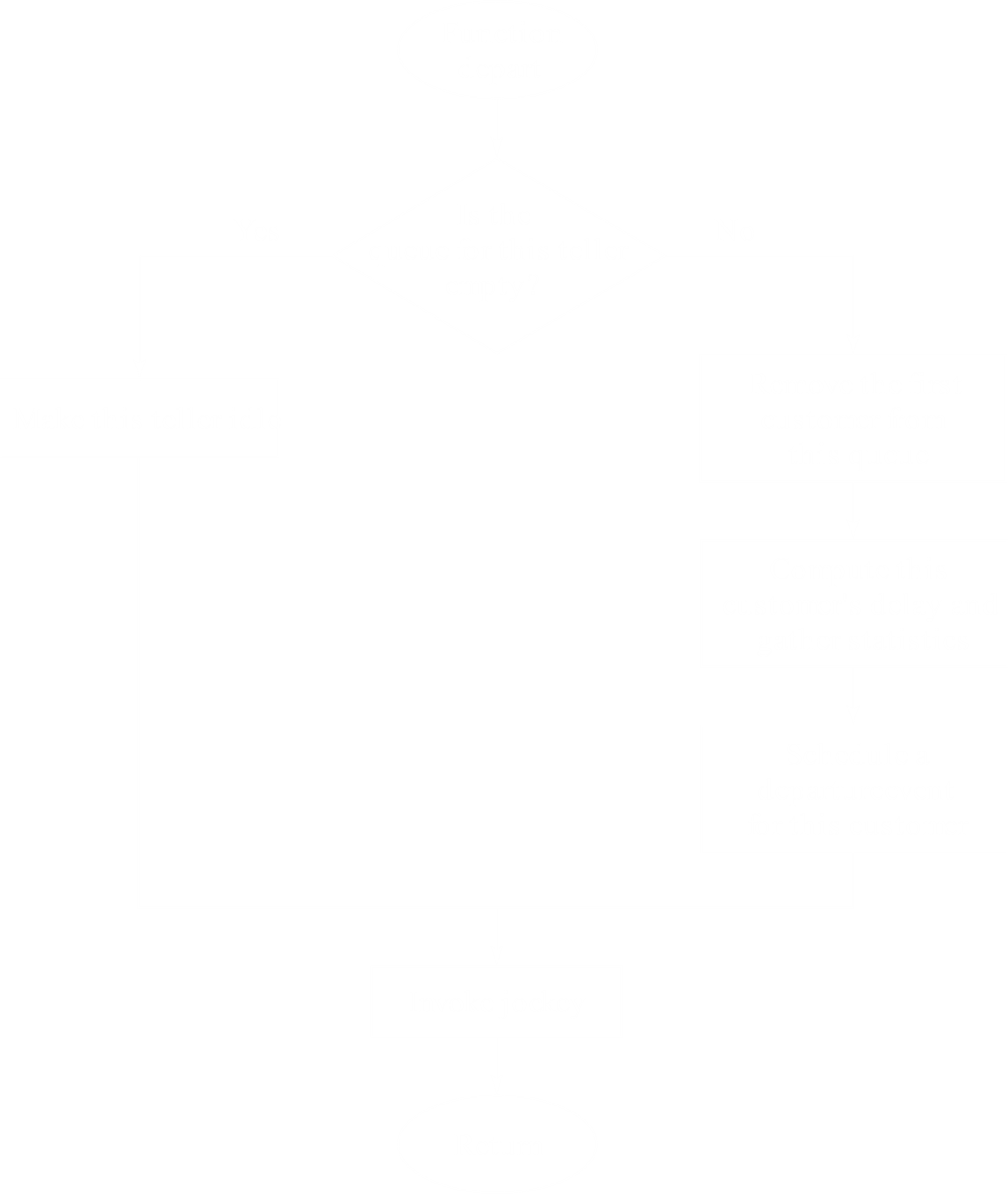
An **arriving customer** will first check for a **free server**. If so, the customer will take service from that server.

If there are **no free servers**, they will join the **shortest queue**. If there are **multiple queues** with the same shortest size, they will join the **leftmost queue**.



### Departure Handler

When a **departure** occurs, the **next customer** in the corresponding **queue** gets service.

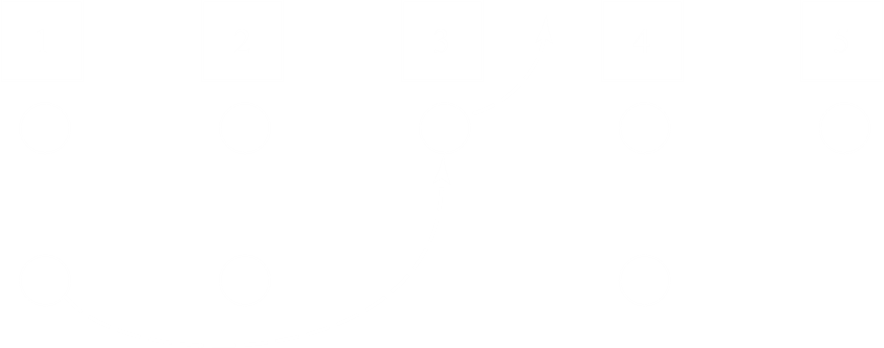


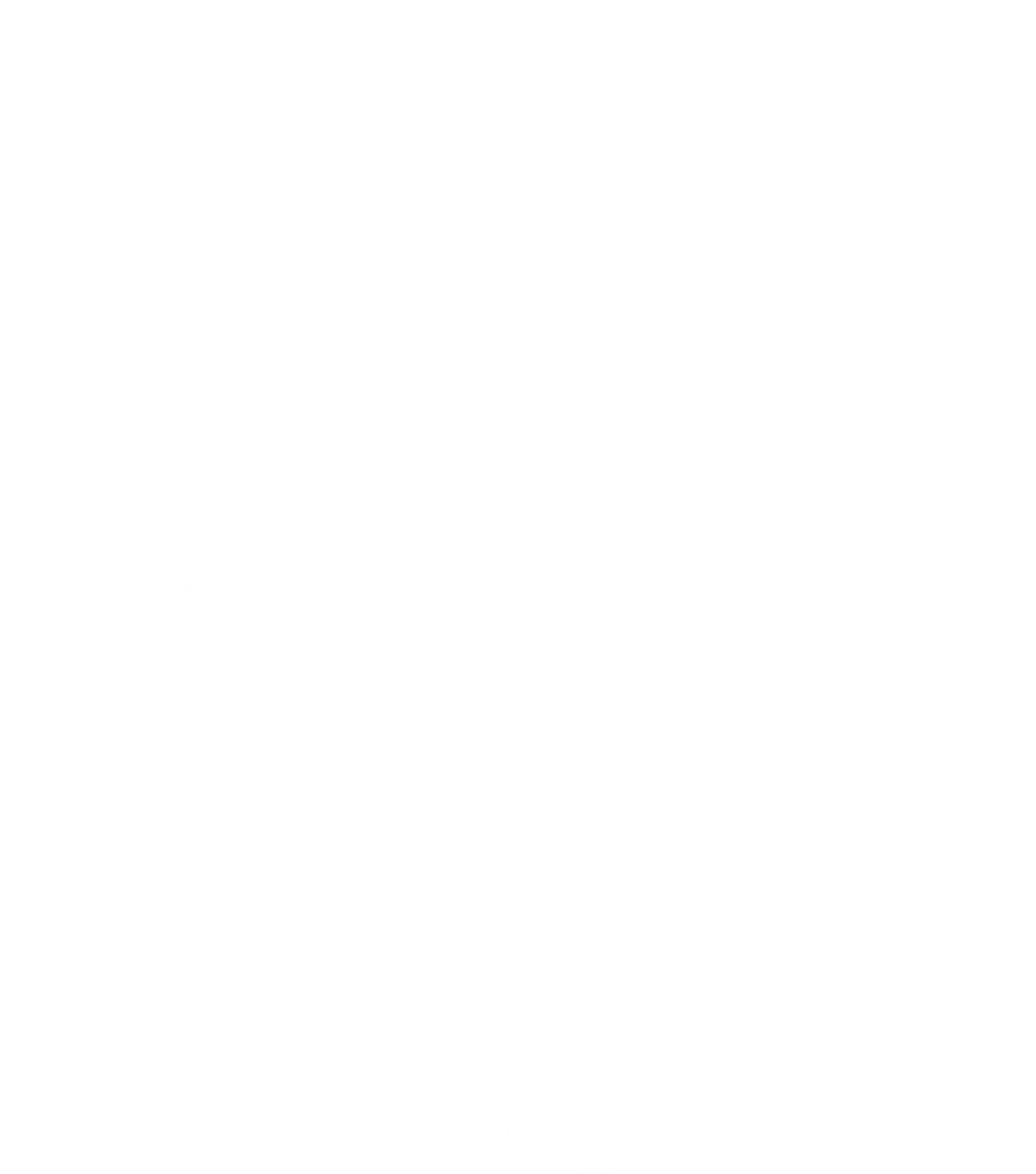
### Jockeying

A **departure** can also cause **jockeying**. After a customer has departed from the server and the next customer (if any) from the corresponding queue is called, we start checking the other queues.

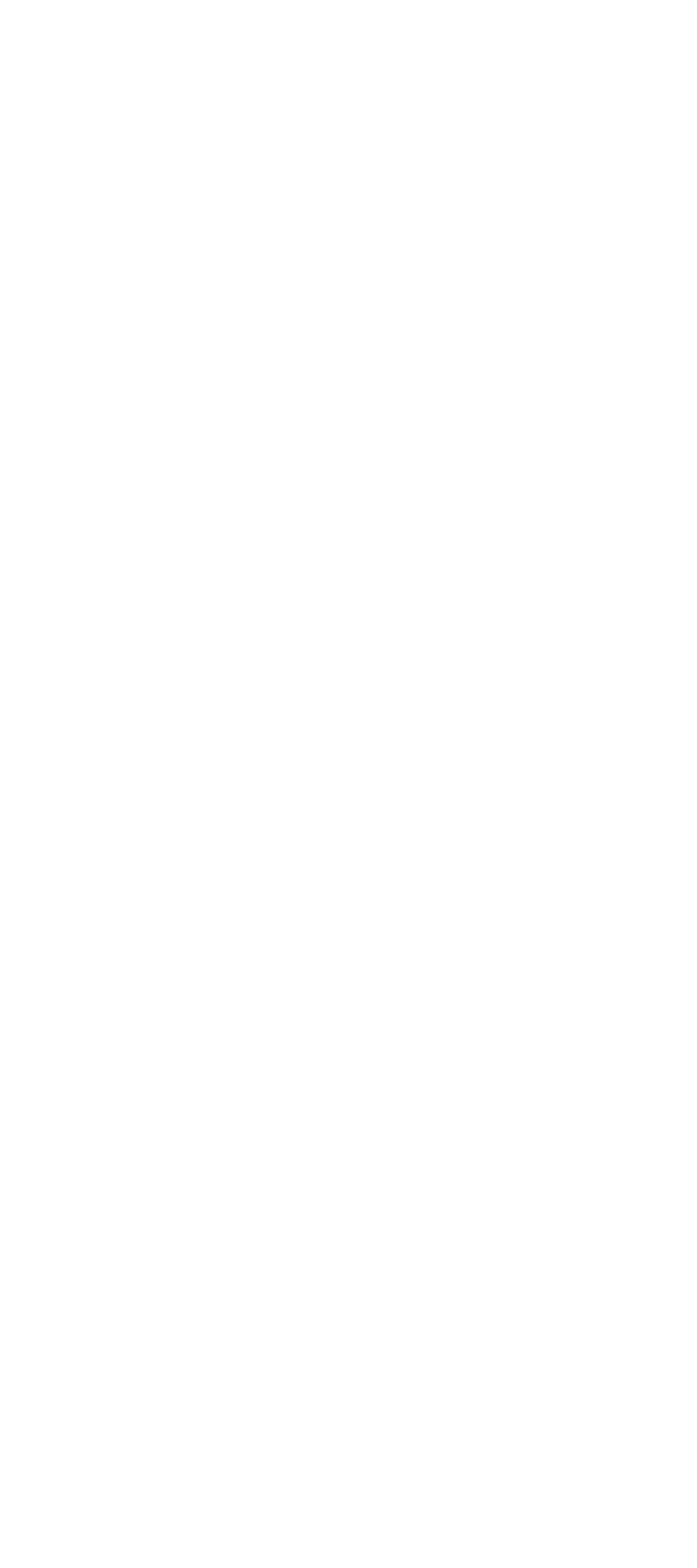
For the th server, if the **total customers** for that server (in service and in queue) is and we find that , where is the total customers for the th server, then **one customer** from the th server will go to the server or join the queue of the th server.

If there are **multiple servers** that satisfy the condition required for **jockeying**, then the **leftmost server** will have its customers shifted.





### Termination Handler

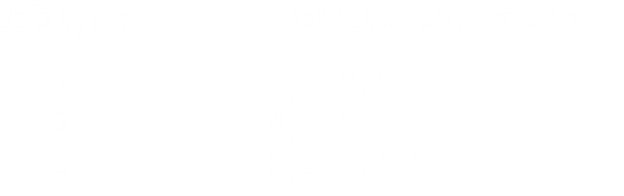


## Job-Shop Model

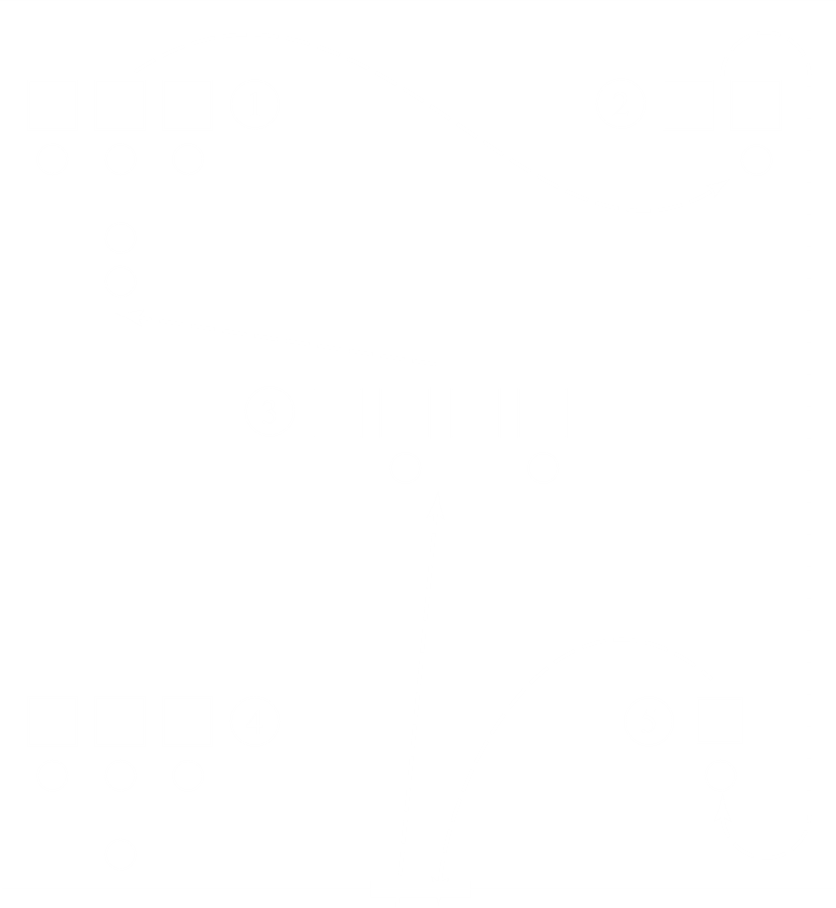
Consider that we have a system with **five workstations** and **three types** of jobs. The **probabilities** of a newly arrived job being a Type 1 job, a Type 2 job or a Type 3 job are , and respectively.

Each workstation has a number of **identical machines**, with 3, 2, 4, 3 and 1 machines in each of the workstations respectively. Thus, each of the machines can serve the corresponding number of jobs **simultaneously**. If a machine is given more jobs than it can serve, the extra jobs enter a **single queue**.

Each **job type** must follow a **specific order** in which to navigate the machines, as given below:



Thus, for a Type 1 job, the path might look like this:

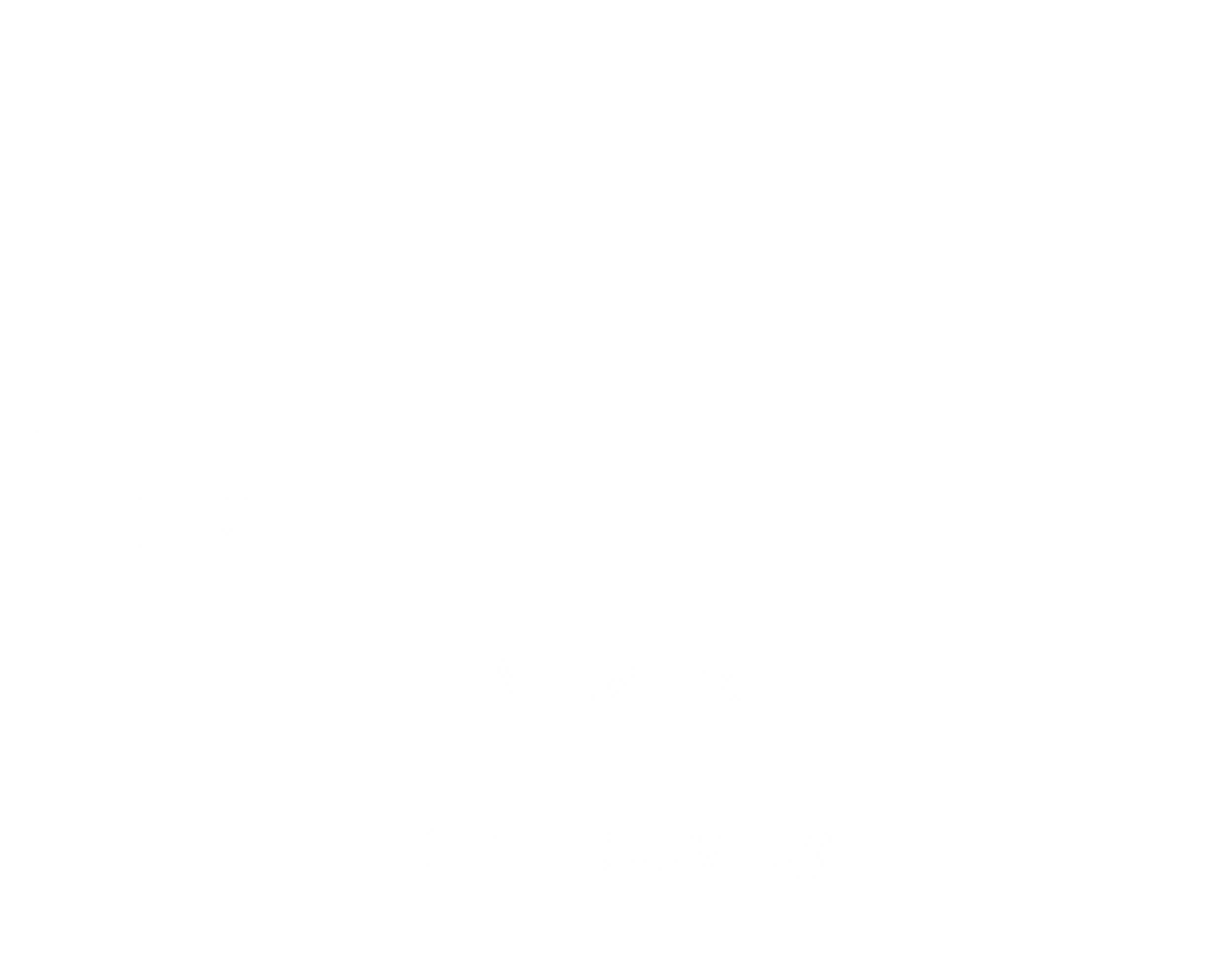


Clearly this is very complicated. To deal with it, we will essentially create a **network**.

Consider that we have a **WorkStation class**. This class has a ‘connectivity layer’, which **sends and receives jobs** to and from a connected **channel**.

On top of this, we have a ‘network layer’, which has a **routing table** of sorts. For each workstation, based on the table above, we can determine the **next workstation** depending on the **job type**.

The ‘network layer’ will add jobs to either the **machines** or the **queue**. Once a job is done being served, it may **exit** the system, if this workstation was the last one for it, or it may return to the network layer, to be send to the **next workstation**.



Each of the ‘layers’ have their own **state variables** and **events**.

Coming back to the **channel**. The **channel** will simply forward the job from one workstation to a common **switch**. The sending workstation’s ‘connectivity layer’ will inform the switch while sending the packet where to send the packet next.

