**Single Server Queuing System**

Table of Contents

[Step 01: Goals and Objectives 4](#_Toc87177861)

[Step 02: Conceptual Model 5](#_Toc87177862)

[System Diagram 5](#_Toc87177863)

[System State and State Variable 5](#_Toc87177864)

[Events 6](#_Toc87177865)

[Input Variables 7](#_Toc87177866)

[Output Variables 7](#_Toc87177867)

[Step 03: Specification Model 8](#_Toc87177868)

[State Equations 8](#_Toc87177869)

[State Space 9](#_Toc87177870)

[Feasible States 9](#_Toc87177871)

[Output Equation 9](#_Toc87177872)

[Output Statistics 12](#_Toc87177873)

[Input Data Modelling 17](#_Toc87177874)

[Step 04: Computational Model 20](#_Toc87177875)

[Components 20](#_Toc87177876)

[Flowcharts 24](#_Toc87177877)

[Main Program 24](#_Toc87177878)

[Arrival Events 26](#_Toc87177879)

[Departure Events 27](#_Toc87177880)

[Visualizing Changes 28](#_Toc87177881)

[Actual Program 34](#_Toc87177882)

A **Single Server Queuing System** is one where the main service is being provided by a single source. For example, if there is a single cashier in a bank, that system would be a Single Server Queuing System. The simulation of a Single-Server Queuing System begins with a **problem statement**, which is provided by the people running the simulation.

Consider that customers are arriving at **random time-points** and seeking services that require **random amounts of time**. These times follow some **probability distribution**. If a customer finds the server **idle**, they are immediately served. Otherwise, they enter the **queue**. When a customer **departs**, and the queue is **not empty**, a customer from the queue is served.

For our evaluation, we are interested in:

* **Service quality** improvement
* Customer **satisfaction**
* Effective use of **system resources**

The points we are interested in will be provided.

## Step 01: Goals and Objectives

We need to ask a few **questions** about what we are trying to achieve before we get started. The questions might be Boolean, numeric or qualitative.

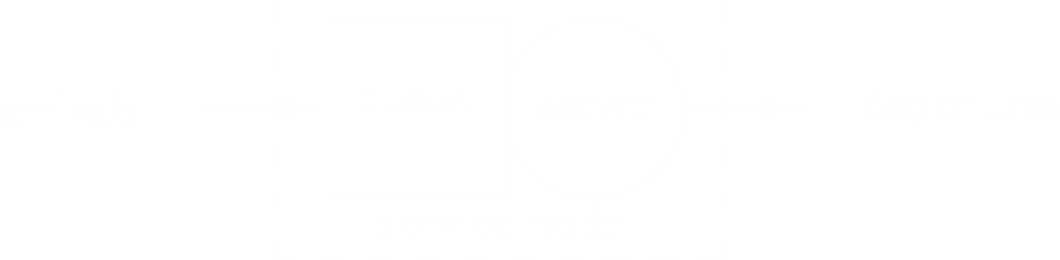
* Boolean: Do we need an additional server?
* Numeric: How many parallel servers do we need?
* Qualitative: Do we need to decrease the customer waiting time? Do we need to increase it (in case we have idle servers)?

By answering these questions, it may be possible to find the **output variables**, such as waiting time, or server utilization.

## Step 02: Conceptual Model

The Conceptual Model helps us find the details of the simulation. There are actually several parts to this.

### System Diagram



In the diagram above, each server is represented by a **circle** and the queue is represented by an **open-ended rectangle**. The open-end implies that the queue has no limit. If there was a limit to the queue size, it would be **closed-ended**. In that case, customers who arrive once the queue is full would have to be denied service.

Right from the start, we can see that the service time will be dependent on the arrival rate and the departure rate .

### System State and State Variable

The **state** should tell us the complete characterization of the system at an instance of time. In our case, this could be things like how many customers are in the **queue** or whether or not the server is **idle**.

The **state variables** are quantitative, which assigns a numerical value to a **system state**. The set of state variables is the system state.

For this system, we can use the state variables to answer a few questions. For example, what does a new customer find when they arrive? Is the server idle or busy? How long is the queue if the server is busy?

Thus, we need **two** state variables, **server status** and **queue length**. There is also a **relationship** between these state variables. If the queue length is **non-zero**, the server must be **busy**. If the server is **idle**, the queue length must be **zero**.

There is an alternative way we could have represented the system using just **one** state variable, the **total customers** in the system. The two state variables we mentioned earlier can be derived from this. If there are no customers, the server must be idle and the queue must be empty. If the number of customers is non-zero, the server must be busy and the queue length must be one less than the number of customers.

Queue Length

Server Status

### Events

An **event** is the occurrence of **actions** that may change the **system state**. An event does not necessarily change the system state, but the system state cannot change without an event.

In our case, a **customer arrival or departure** are events. These events potentially cause the **server status** to change from idle to busy or vice versa or change the **queue length**. The **total customer** count also changes.

### Input Variables

In our example, the input variables are:

* **Arrival times** of customers
* **Departure times** of customers

The departure time will depend on how long the customer had to wait plus how long the actual service took, i.e. the total **system time**, or sojourn time, for the customer.

The arrival and departure times could be provided as **absolute** times or **relative** times. If we use relative times, then we only need to keep track of when a particular customer will arrive once the customer before them has arrived, i.e. once one customer arrives, then we start worrying about when the next one will arrive. This is easier than having to track exactly when each customer will arrive right from the beginning.

### Output Variables

The **output variables** we will need are:

1. To judge **customer satisfaction**:
   * Average waiting time
   * Average queue length
   * Maximum queue length
2. For **management decisions**:
   * Server utilization
   * Optimum number of servers
   * Optimum queue length

## Step 03: Specification Model

### State Equations

The first step of creating the specification model is defining the **state equations**. We will be directly using these equations when writing the algorithm.

As discussed, before, we can do this using

* Two variables, , the **status of the server**, and , the **length of the queue**
* One variable, , representing the **number of customers** in the system

In the first case,

In the second case,

Note that the equation would not apply if , since a **departure cannot occur** under that circumstance. This is not a feasible event.

### State Space

The **state space**, , defines the set of all possible values of the state variables.

When the state variables are and , . Notice that if , the only possible value of is .

When the state variable is , .

### Feasible States

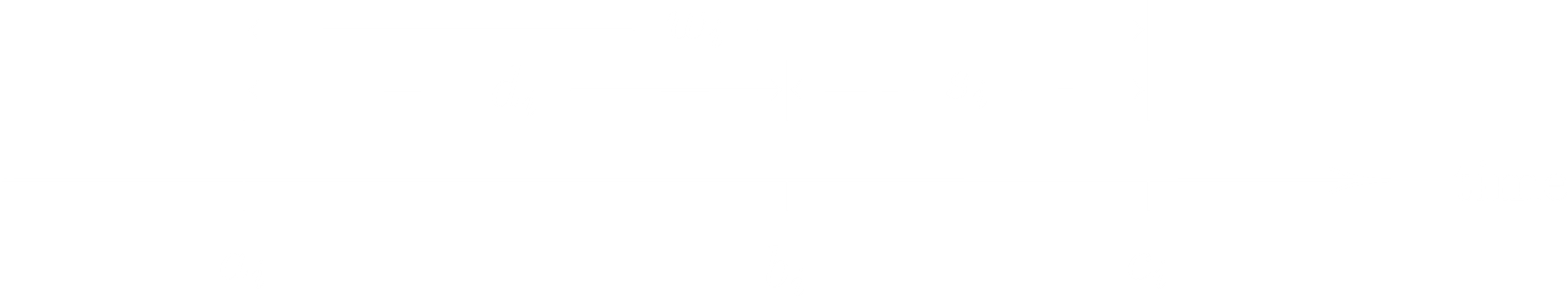
If we are considering and , at the only feasible event is a **customer arriving**, while at every other state both **arrivals and departures** are feasible.

If we are considering , the only feasible event at is an **arrival**, while at every other state, both **arrivals and departures** are feasible.

### Output Equation

The **output equation** depends on several variables:

1. The **arrival time**, .
2. The **delay** in the queue, , where .
3. The **service start time**, .
4. The **service time**, , where .
5. The **system time**, .
6. The **departure time**, .



As previously discussed, the **arrival times** can be given as **absolute times**, () or as **interarrival times** (). If the interarrival times are given, then

Assuming ,

The **delay** in the queue will be given as a set of **arrival times** (or interarrival times) and **service times**. From this information, we can calculate the delay.

The way we do this depends on the **queuing discipline** we decide to apply. For simplicity, let us consider it is FIFO.

For a FIFO queue, there can be two cases.

First, if the th job arrives **before** the th job departs, i.e. , then

Second, if the th job arrives **after** the th job departs, i.e. , then

A simple program to calculate this delay (using some pseudocode) is given below:

c[0] = 0.0f;  
i = 0;  
  
while (remainingJobs > 0)  
{  
 i++;  
 a[i] = GetArrival();

if (a[i] < c[i-1]) d[i] = c[i-1] - a[i];  
 else d[i] = 0.0f;

s[i] = GetService();  
 c[i] = a[i] + d[i] + s[i];  
}  
n = i;  
return d[1], d[2], …, d[n];

PSEUDOCODE

The program above is **trace driven**, meaning the input data will be **provided**. If this were not the case, then the input data, arrival times and the service times in this case, would need to be randomly generated. If we are using randomly generated input data, then we need to ensure that we change the **seed** value every time we run the program. Otherwise, we will end up using the same set of random values every time we run the program. Generally, the **system time** is used as the seed value. The mathematics behind this will be covered later.

The data below can be used as sample data to test the program above.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |

In the above example, the completion time of the last event, , is .

### Output Statistics

**Output statistics** are statistics based on the output data we get. What statistics we are interested in depends on who we are. From the customer’s perspective, the **average delay** is the most important factor. For them, the best-case scenario is to never have to wait. From the management’s perspective, the **resource utilization** is the most important factor. For them, the best-case scenario is for the server to never be idle. These two scenarios cannot co-exist, and so a compromise must be made. The **optimal case** is that the system is busy for most of the time and the delay is very short.

There are two ways in which we can average our data, **job-average** and **time-average**. Job-average is a simple arithmetic average over the total number of jobs. Time average uses the total time, but is a little more complicated.

For **job-average** statistics, the possible average values are:

* Average **inter-arrival** time:
* Average **service** time:
* Average **delay**:
* Average **system** time:

We know that the number of jobs in the system at time is given by , the number of jobs in the queue at time is and the number of jobs in service at time is . Logically, . Since these are functions of time, we can find the **time-average** for these variables.

For example, if we find the time average of the total number of jobs in the system, then we will find the expected number of jobs at any given moment in time. If the system is running for time units, and there are customers for the first time units and customers for the last time units, then the average is .

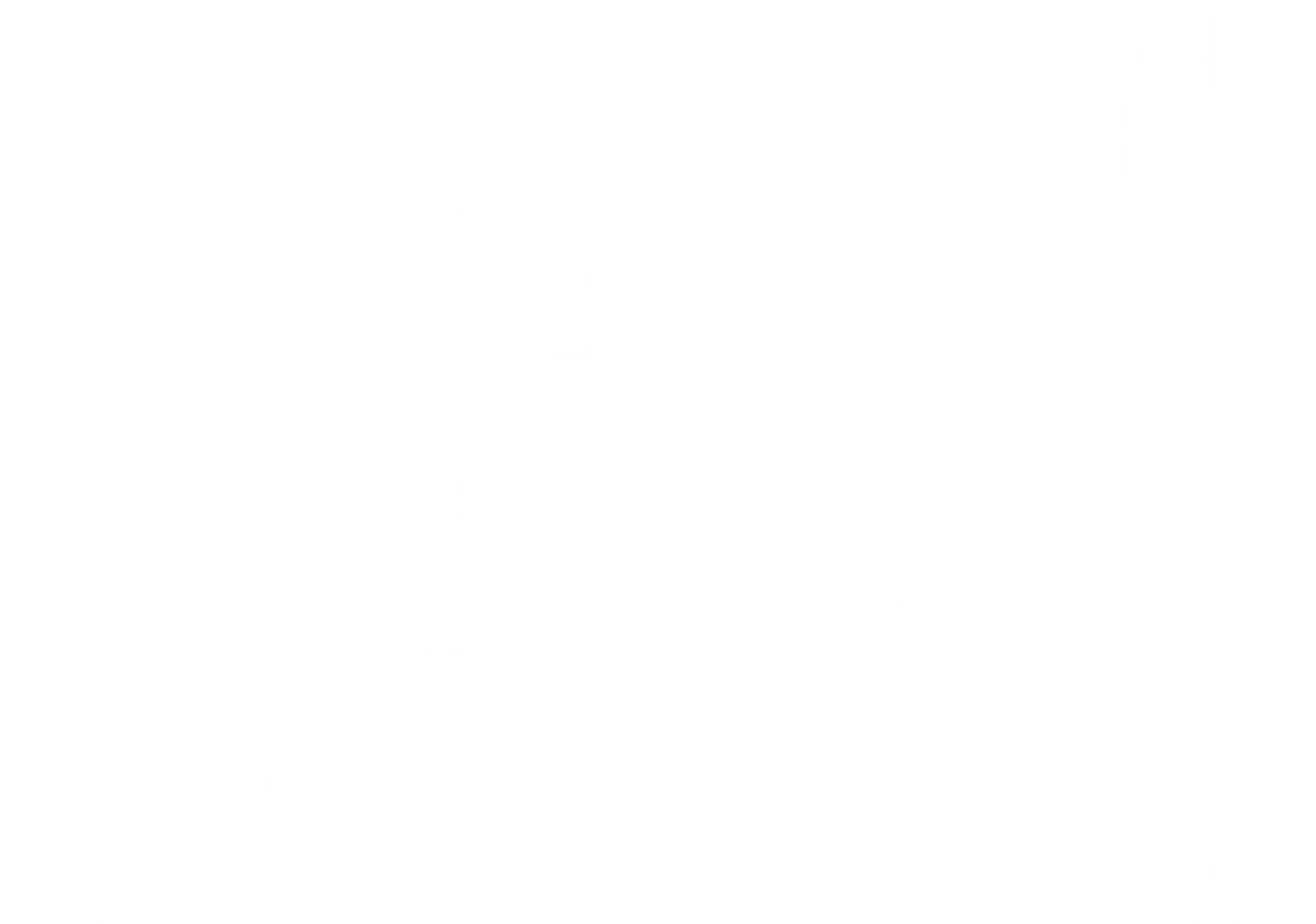
However, time is **continuous**, so we cannot simply average it over time units. Instead, we find the time-average number of jobs in the system for the period using integration:

Similarly, we can also find the time-average number of jobs in the queue and the time-average number of jobs in service. The latter quantity is also called the **server utilization**, or the portion of time for which the server is busy.

Since , for all , .

#### Little’s Formula

**Little’s Formula** describes how job-average quantities are related to time-average quantities.



Consider the diagram above. The upper curve tells us the **cumulative number of arrivals** at a given point of time while the lower curve tells us the **cumulative number of departures**. At any given moment of time, the difference between these two values will tell us the **total number of customers** in the system.

For the th customer, tells us the service time. We could also use the **area** of the rectangle containing , since the height is . Using this, if we find the area between the upper and lower curves, we will find the **total delay**, i.e. .

Above, we essentially drew a few horizontal lines to divide the smaller rectangles and find the area. We can also find the area using vertical lines to divide the smaller rectangles. This is similar to what we did earlier, using time units to find the time average.

If we make the distance between the vertical lines infinitely small, we can approximately say that the area of a rectangle that starts near time will give us the total number of customers in the system at time , i.e. . Since the rectangles in this case are infinitely narrow, we need to perform **integration** to sum them up. Thus, the total number of customers in the system for a time period can be found by .

However, the actual area that we found using both approaches should still be the same. This suggests that,

Similarly, we can also derive two more relationships:

From this, Little’s Formula derives that:

Here, is the **average arrival rate**.

Similarly,

### Input Data Modelling

Whenever we are running a simulation, we need to provide it some **input data**. In our case, this would be things like the arrival times, the departure times and the service times. As discussed before, there are two ways in which we could do this, using **absolute time** or **relative time**.

Another question is how exactly is the data being provided? Is the data from a **sample data collection**, or is it being **randomly generated**?

#### Sample Data vs Randomly Generated Data

What methods we use will actually vary from situation to situation. It may be tempting while creating a simulation to just use random values, but random values are generated by default as a uniform distribution. We need to consider that our data may not actually be uniformly distributed.

This in turn brings up the question of how we are even supposed to know how our data is meant to be distributed. In reality, we need to **collect random sample data** from the real system to determine this. We also need to find the parameter values for the distribution we identify, which can be achieved using **MLE**.

Once all of this is done, then we can generate random data. Alternatively, we could have just collected all the data we needed from the real system. That process is more difficult though, since sample data is not always easy to collect. Also, it restricts us to using absolute timing.

#### Absolute vs Relative Timing

If we choose to use **absolute timing**, every timestamp will be provided with respect to the simulation start time, as opposed to **relative timing**, where each timestamp is provided with respect to the timestamp before it.

All of the timing information must be available before the simulation starts. It is not possible to generate timing information during the simulation if we are using absolute timing, since we could end up with a timestamp that is before the current time. Thus, we need to randomly generate data at the beginning. We also need to **sort** the data we receive so that we do not end up randomly jumping around.

Again, this is opposed to relative timing, where we simply randomly generate how long we need to wait before the next event instead of having all the timestamps available right from the start. This helps in many cases where we do not know how much data we will need. Clearly, using relative timing is far more beneficial. However, there are some cases where it is better to use absolute timing.

#### Generating Random Data

Consider that we have been provided job interarrival times that are **exponentially distributed** with parameter . This means that a new customer arrives every minute on average. For this distribution, the PDF is

If we take a **random number**, , in the range , the **random value** we need can be generated using the formula:

This formula is applicable for exponential distributions only. We will be studying how to derive the formulae for all the different distributions later on.

We can now add the random value that we get to the previous absolute time to get a new, random arrival time. In this way, if we run the program **multiple times**, each time with a different **seed value**, we will get a set of results that we can then find the arithmetic average of.

Next, consider that **service times** are also **exponentially distributed** with the parameter . Thus,

Now that we have a way to generate both arrival times and service times, we can integrate these changes to the simple program we saw earlier that allowed us to calculate the delays. Instead of using provided values, we can now use randomly generated values. Once we have the different delay values, we can use **Little’s Formula** to find the average values we need.

The ratio is more commonly called the **traffic intensity**. If we vary the values of the two variables, we will find that the closer this ratio gets to , the more impossible the situation becomes. We will essentially be heading towards a point where we will never be able to serve all our customers. If the ratio ever crosses , the queue size will eventually become infinite. Thus, we can derive that this ratio needs to be far less than to be a sustainable system.

## Step 04: Computational Model

The **Computational Model** converts the Specification Model into a computer algorithm. Initially, we will be using **pseudocode**, and once we have the program in hand, we will convert it into an actual **computer program** using a language of our choice. It is best to use an object-oriented language, since big programs are easier to handle. However, we will begin with a structured programming language.

### Components

#### Simulation Clock

For **discrete-event systems**, one of the features is that the system has events that occur randomly over time. This means we need to keep track of time. We do this using a **simulation clock**. This is a variable that will keep track of the simulated time, which starts at when the simulation starts. The unit is irrelevant.

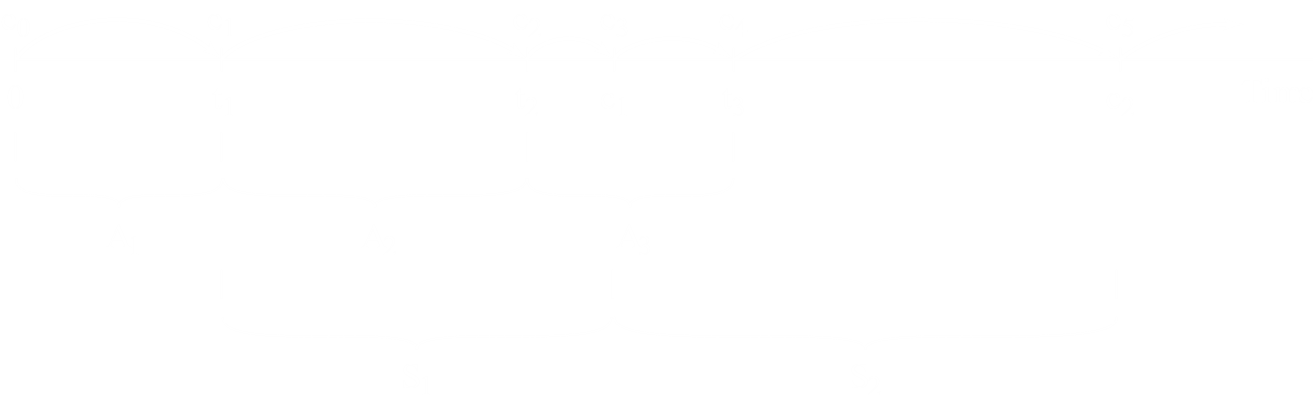
The clock allows us to synchronize the different events. An interesting thing here is, it is easier to use absolute times in this case. Once we have generated a random time for the next event to occur relative from the previous event, if we do not convert this to an absolute time, we would need to repeatedly calculate how much time is remaining. With an absolute time, we would just need to check if the simulation clock has the same time.

#### Time-Advancement Mechanism

Related to the simulation clock is a **time-advancement mechanism**, which will increment time at fixed intervals. Time will be changed from one event’s occurrence time to the next feasible event’s occurrence time. This is done because anything in between is of no consequence to the simulation. This approach is called the **next event time advancement**. Time advancement is dependent on the event list and vice versa.

Another approach could be **fixed increment time advancement**, where time is repeatedly incremented a fixed amount and it is checked whether an event should occur at that time. This approach results in unnecessary time wasted in between events, especially if the simulation needs to simulate a large amount of time. Time advancement has nothing to do with events occurring.

The two concepts above should become a little clearer with the diagram below.



indicates the interarrival time of the th event. These are the values that we generate randomly relative to the previous event. is the absolute arrival time of the th event. We will convert to once is generated.

Similarly, we have , the service time of the th customer, which we will convert to , the absolute time at which service for the th customer is complete.

The time in the program will change from to if the th customer leaves before the th customer arrives, or from to if the th customer arrives before the th customer leaves. Thus, the time is changing from one event to the next, from to .

#### Event List

The above points also mean we need to have an **event list**, which contains the discrete times at which every event occurs. This also requires a **list maintenance mechanism** to exist, which will figure out which events are feasible at the given state. In most cases, events are sorted based on time.

#### Event Scheduling

We also need **event scheduling**, which essentially causes the next event to occur. All actions related to an event are handled by the scheduler. It inserts the events into the event list and specifies when each event will occur.

Note that, when specifying the time of occurrence, even though we prefer to generate the time randomly relative to the previous event, it is easier to store the time as an absolute time. This is because if we store it as an absolute time, we can just check that the simulation clock has reached that time and fire the event.

Once the time has arrived, the event handler is started by the scheduler. The event handler changes state variables as required.

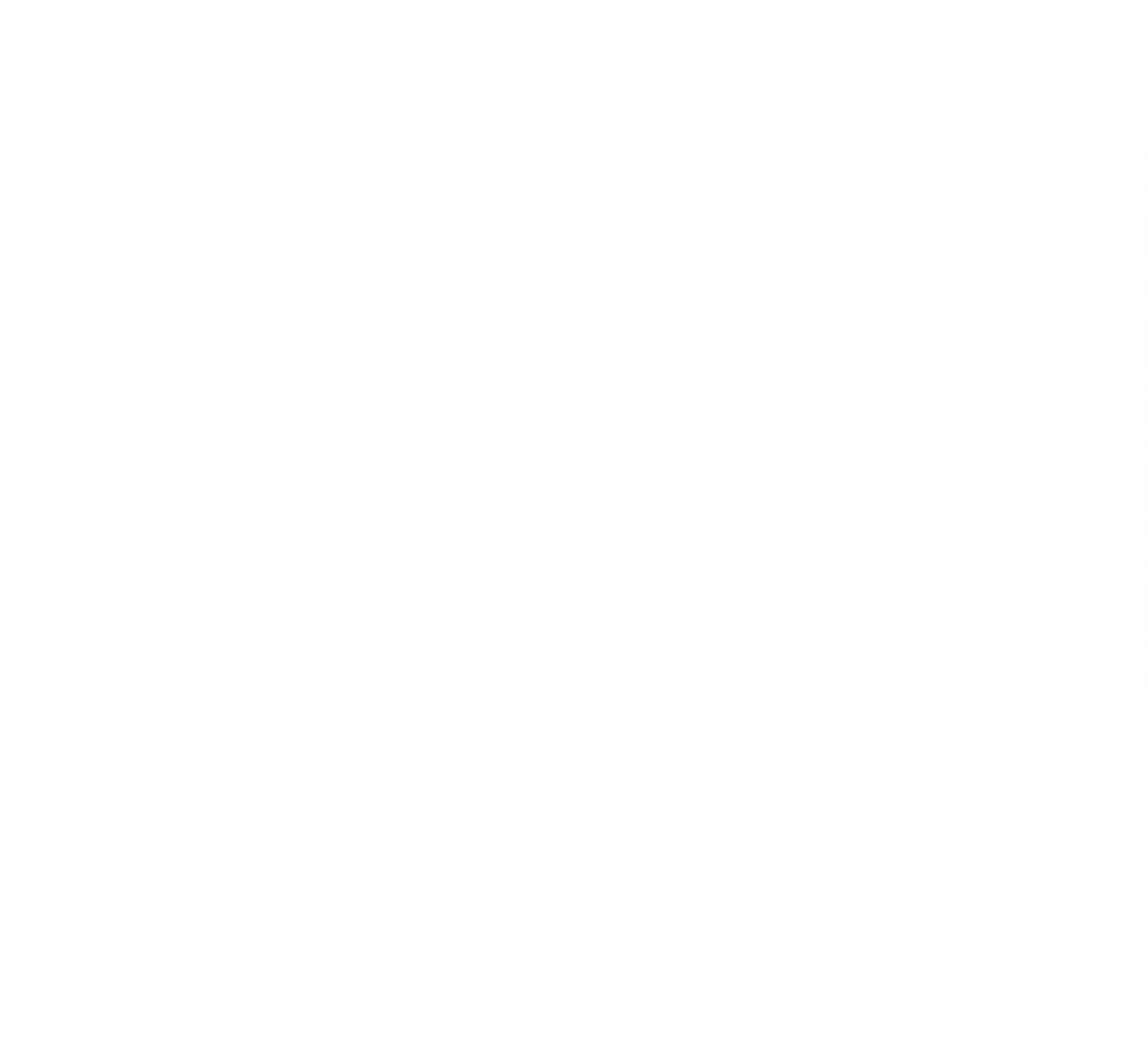
#### Other Objects

Finally, we have several **objects of the system** that we need to keep track of such as:

* State variables
* Set of events
* Output variables
* Input variables

## Flowcharts

### Main Program



The flowchart above shows how a possible simulation program might work.

First, the **main program** calls an **initialization routine**. This routine essentially sets everything up. It sets the simulation clock to , initializes all the state variables and counters and initializes the event list. Then control is passed back to the main program. The initialization routine is aware of how long we want the simulation to run or how many events we want to simulate and it initializes everything based on that.

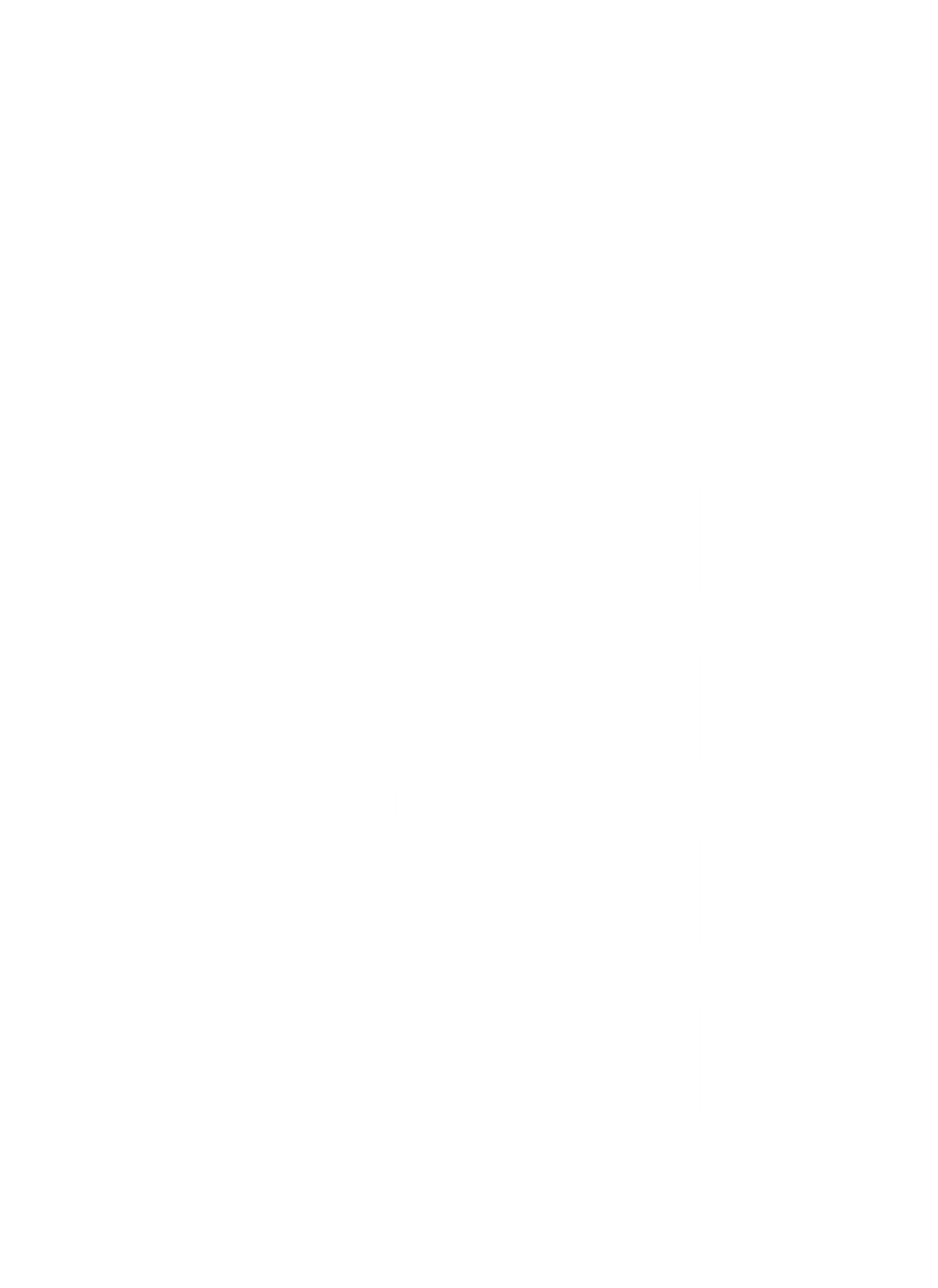
Next, we enter a loop. There are two routines, the **timing routine** and the **event routine**. The timing routine essentially takes us to the time of the next event, while the event routine executes the event. These two routines are alternately called until the program ends. Again, the ‘end’ of the program depends on how long we have told it to run or how many events we have told it to process.

The **timing routine** determines the type of the next event. It then advances the simulation clock to the time of that event. For every type of event (in our case there can only be two types, an arrival or a departure), a different event routine will be called.

The **event routine** updates the state variables and counters as required by an event. It is also possible that the event routine will need to generate more events in case we are randomly generating events as we go instead of all at once at the beginning. In this case, it will need to use **library routines**.

Once an event occurs, we need to check if the simulation has ended. If it has not, we go back to the main program. If it has, we generate a **report**. In the report, we compute the estimates we are interested in. Finally, the program ends.

### Arrival Events

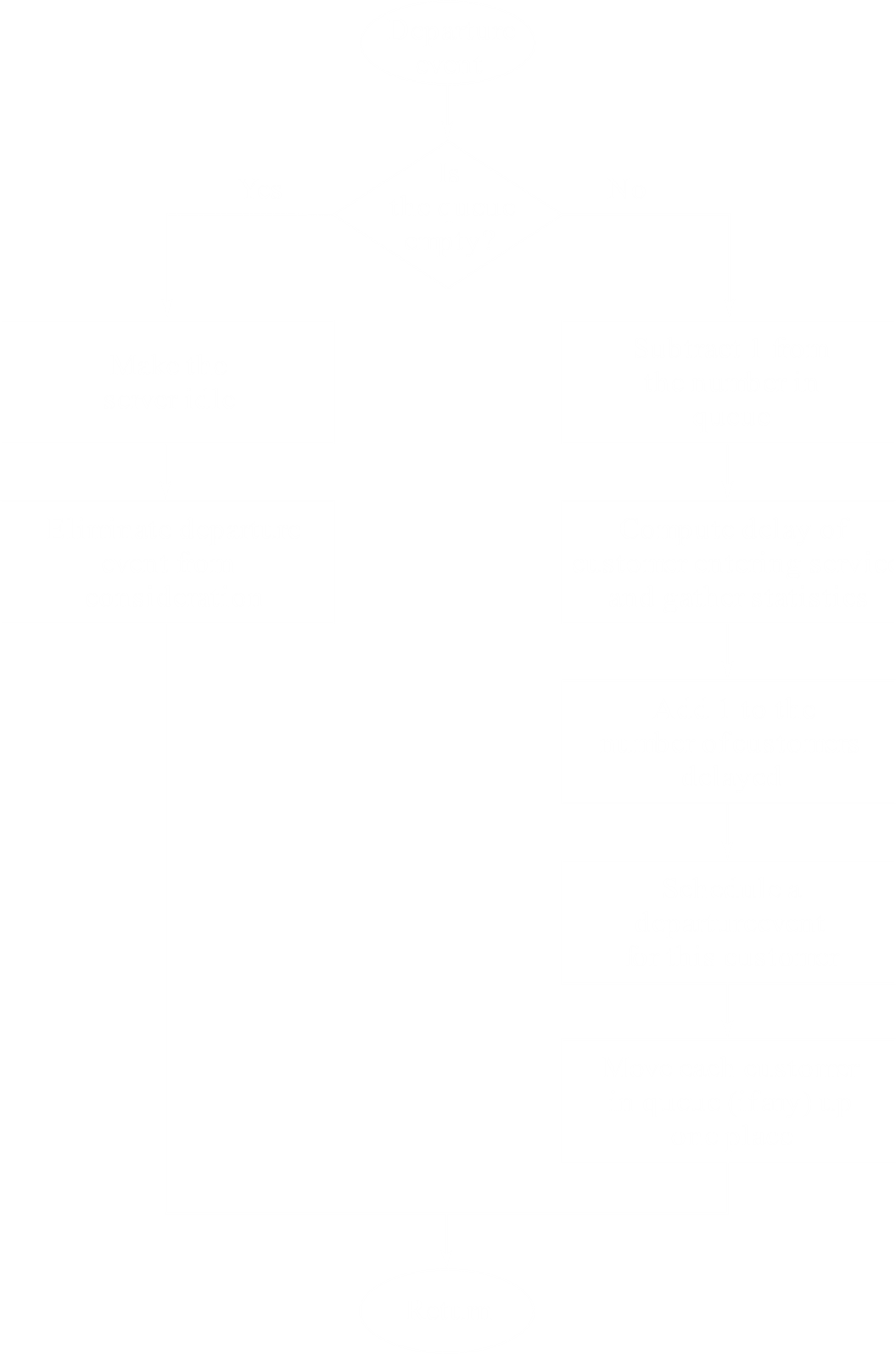


The first thing we do is set up the **next arrival event**. The rest of our work depends on whether the server is busy or not.

If the server is **busy**, we increment the **queue length**. If this causes the queue to become **full**, we have to show an error and stop the program. Otherwise, we store the **arrival time** of the current customer.

If the server is **not busy**, we set the **delay** of this customer to and make the server busy. We also set up the **departure event** for this customer.

### Departure Events



If the queue is **empty**, we mark the server as **idle**.

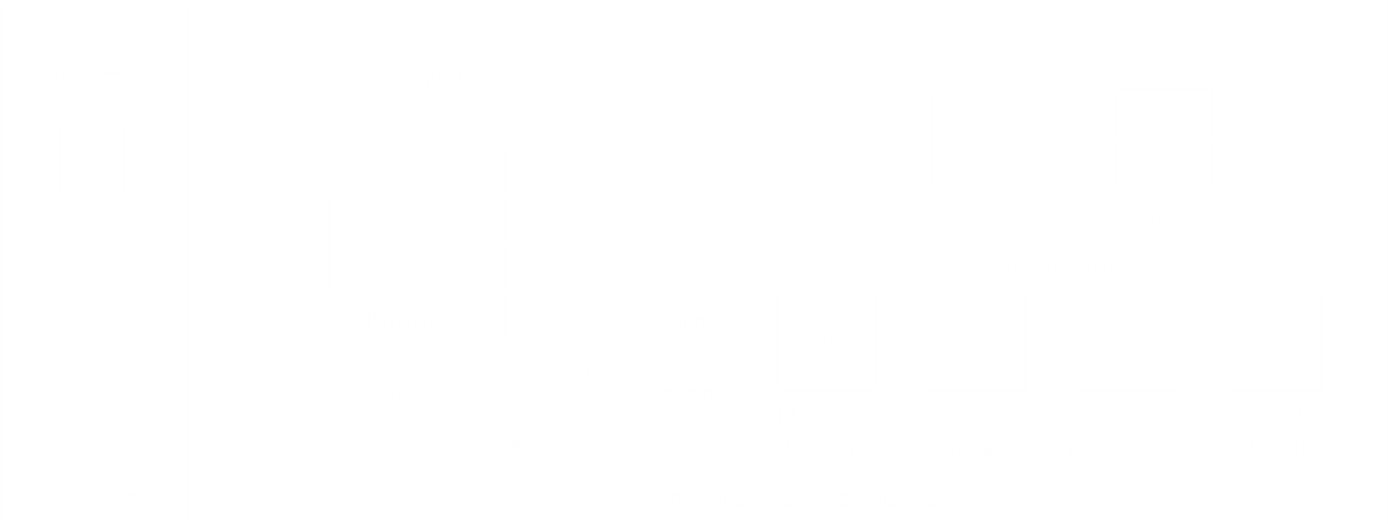
If the queue is **not empty**, the queue length is **decremented**, the **delay** faced by the next customer is computed (based on their arrival time and the current departure time), the **departure event** of the next customer is scheduled and every other customer in the queue is **moved up** by one position.

## Visualizing Changes

Consider that we have the following set of arrival and departure times. Interarrival times will be generated on the go and the corresponding absolute times (shown in brackets) will be calculated and used in the program.

|  |  |
| --- | --- |
| Interarrival Time | Service Time |
| () | () |
| () | () |
| () | () |
| () | () |

The following diagrams should make it easier to visualize how the system changes as the program runs.



The list on the left is like a graphical representation of the queue, the rectangle being the server.

The block of system states contains the variables that are used by the program, i.e. the server status, the queue length, the arrival times of customers in the queue and the time at which the last event occurred.

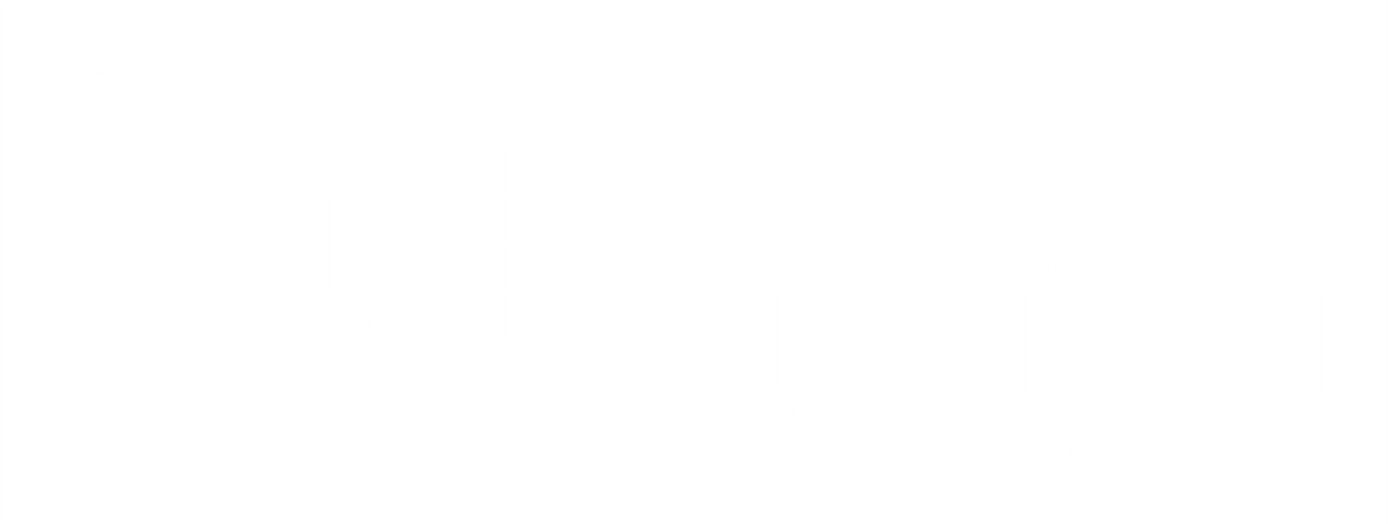
The block on the top right simply has the simulation counter and the times for the next arrival and the next departure. Remember that we are generating the arrival and departure times as we go, so we only need to store one of each.

The statistical counters tell us about values we may be interested in, such as how many customers were served, how much delay they faced in total, the area under queue length graph and the area under the server busy time graph.

The first event that we know about is the arrival at , so the clock jumps to this time. We also schedule the next arrival time and the departure time for the current customer.

Since the queue is empty and the server is idle, this customer is immediately served. The server status is set to busy.

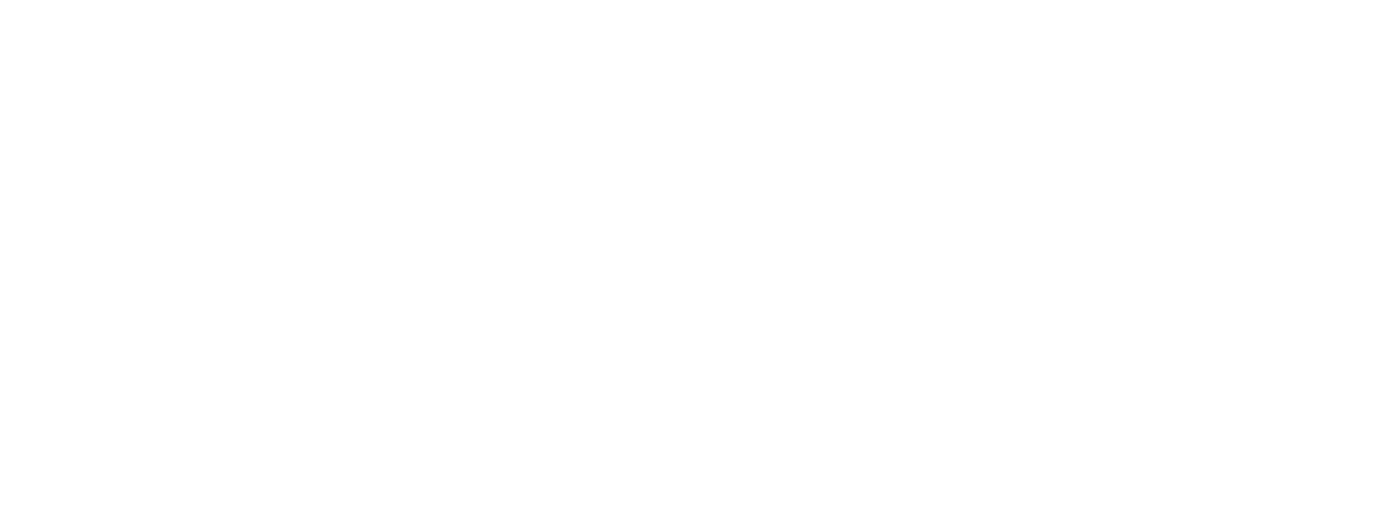
The number of customers served is incremented. The total delay does not change, since this customer did not experience any delay.



The next event occurs at . The previous departure event has not yet occurred, so we only generate a new arrival time.

This customer finds that the server is busy, so the queue length is incremented. The arrival time for the customer is also stored.

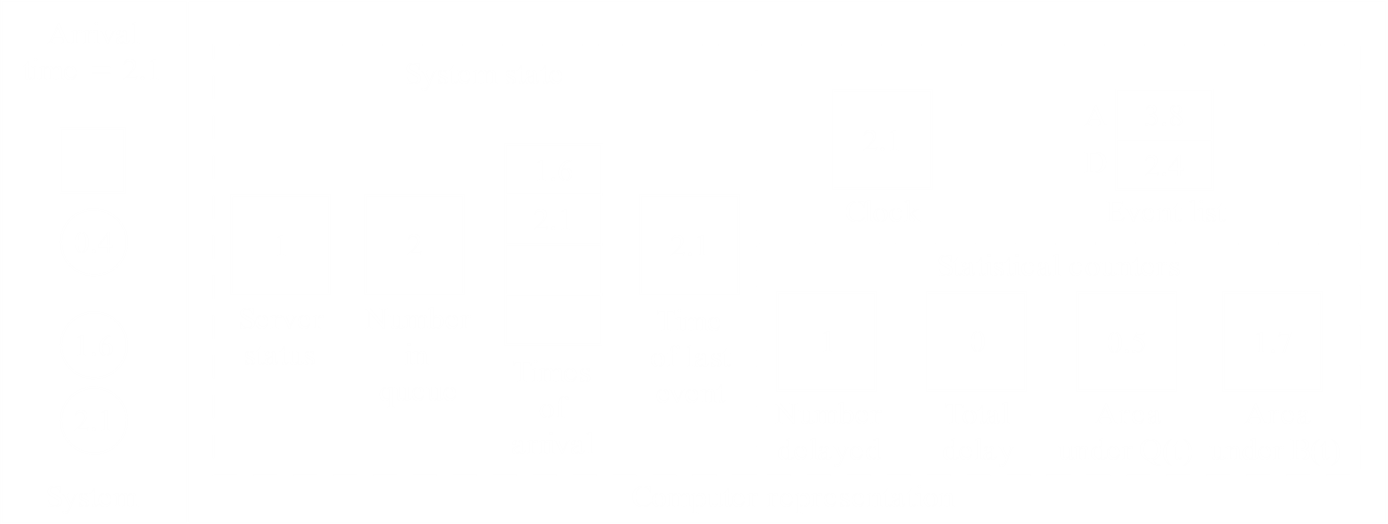
Notice that, in the diagram, the area under , which tells us about server utilization, has increased since the server has been busy between and .



The next event is also an arrival, at . Again, we generate another arrival event time.

The queue length is incremented and the arrival time of the current customer is stored.

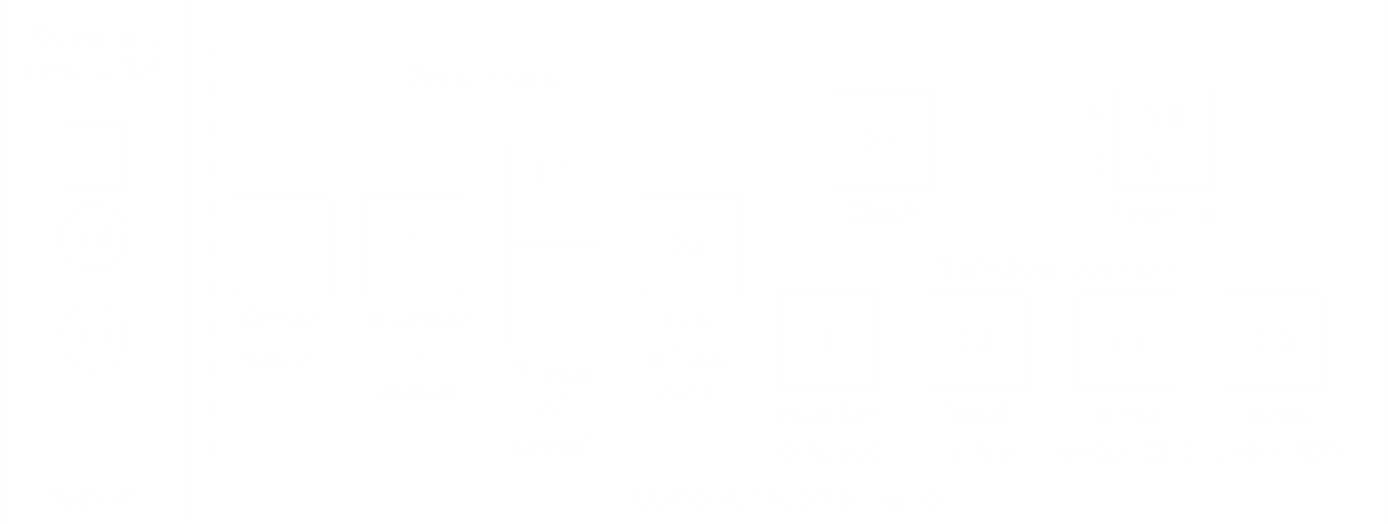
Again, the area under has increased since the server was busy between and . The area under has also increased, since the queue length was between and .



The next event is a departure at . This means we need to generate a new departure event time.

This event causes the queue length to be decremented. The arrival time of the customer that is to be served is also removed from the list, since we no longer need it.

The number of customers served becomes . This customer arrived at and is being served at , so the total delay increases to . The area under has increased to , since the queue length was between and . The area under has increased to , since the server was busy between and .



The next event is another departure, at . Thus, we generate another departure event time.

The queue length is decremented again and the arrival time of the next customer is removed from the list.

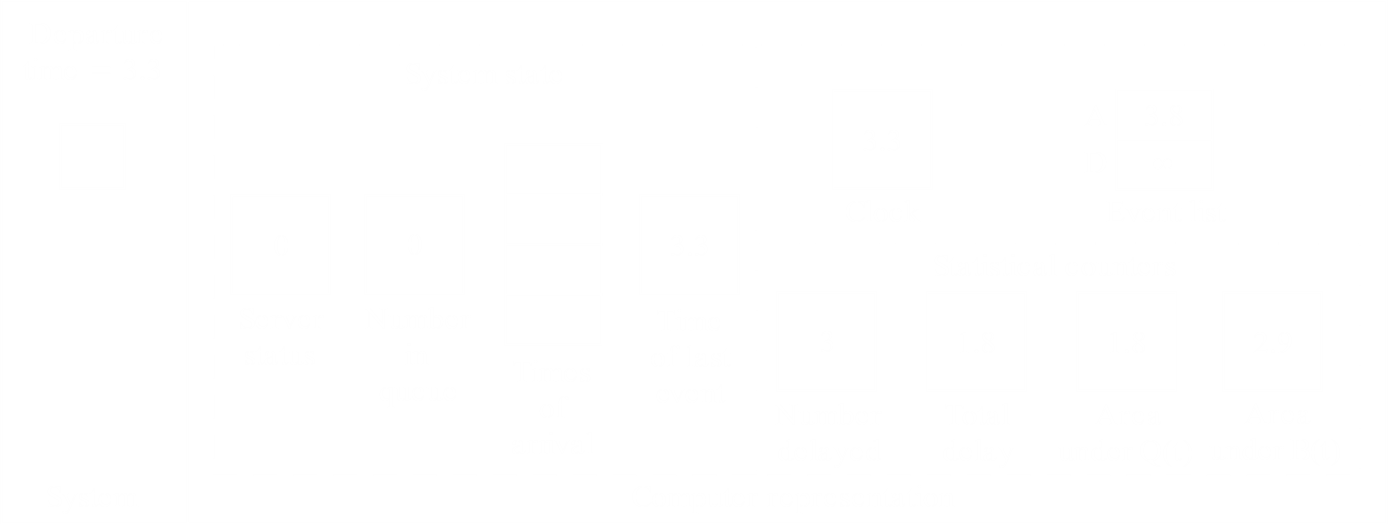
The number of customers served is incremented. The total delay increases, since the current customer arrived at and is being served at . The area under increases, since the queue length was between and . The area under increases since the server was busy between and .



The next event is another departure at . Since the queue is empty now, we do not generate another departure event time, since yet another departure cannot occur.

The server status is set to idle.

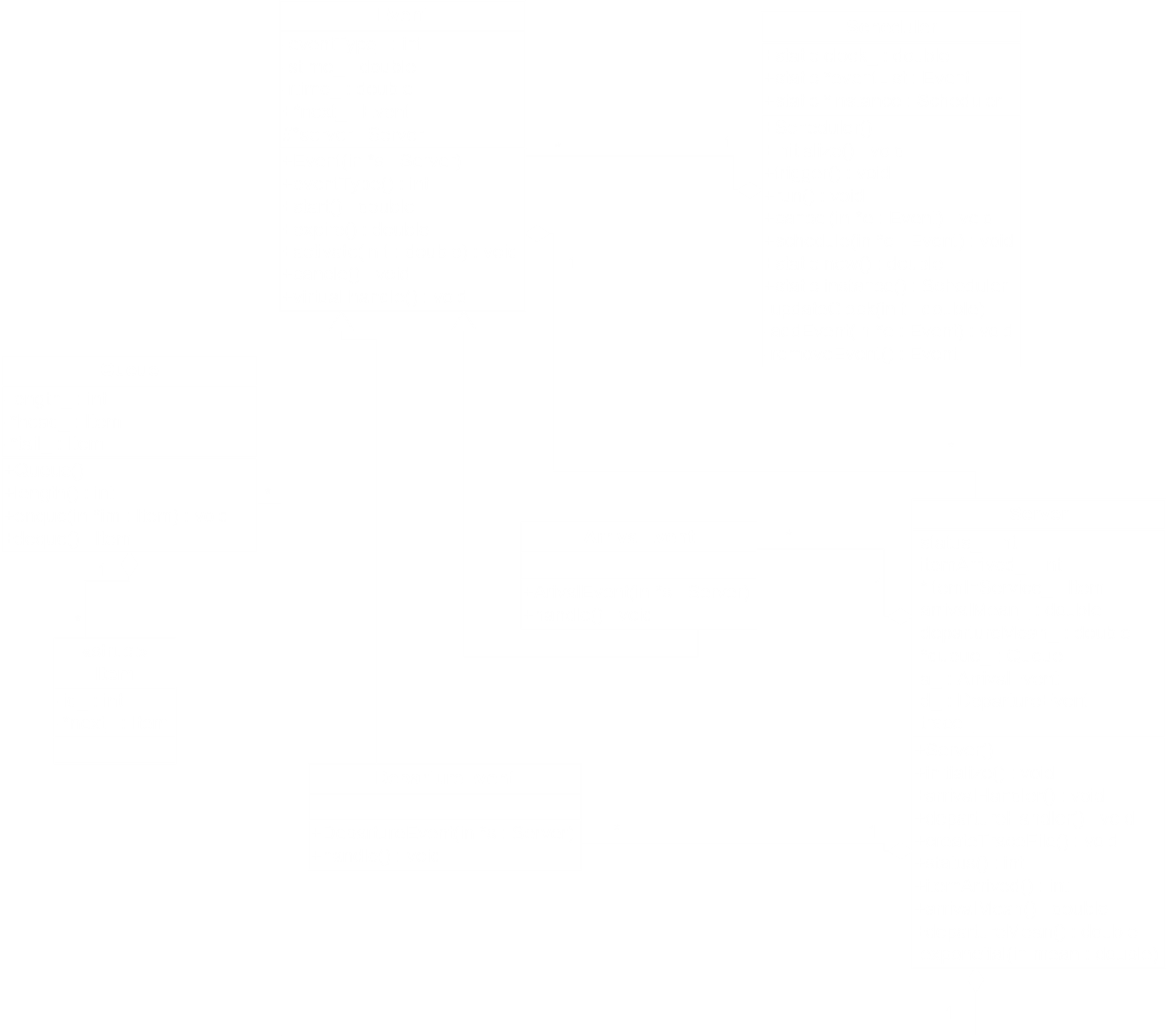
The area under is the only counter to change, since the server was busy between and .



At this point, every possible situation in the system has been analysed. Starting with the next arrival at , the system will continue to run in the same manner, so it is pointless to analyse each of those events in depth.

## Actual Program

In the actual program, we will have several classes.



This class diagram is huge, but we do not need to worry about all of it at once. Essentially, we have an Event class, from which the ArrivalEvent and DepartureEvent classes derive. We also have a Scheduler class, which schedules events, and a Server class, which acts as the server.

Whenever we create a new event, we add it to the eventList in the Scheduler. The run function in the Scheduler removes events from this list one by one. As long as the list is not empty, it will keep removing events. For each event, the clock is updated and the corresponding handler is called, which is in the Server.