**Performance Analysis of a Multiple-Server Queuing System**

Table of Contents

[Events and State Variables 3](#_Toc87178555)

[Arrival Event 3](#_Toc87178556)

[Server 1 Departure Event 3](#_Toc87178557)

[Server 2 Departure Event 4](#_Toc87178558)

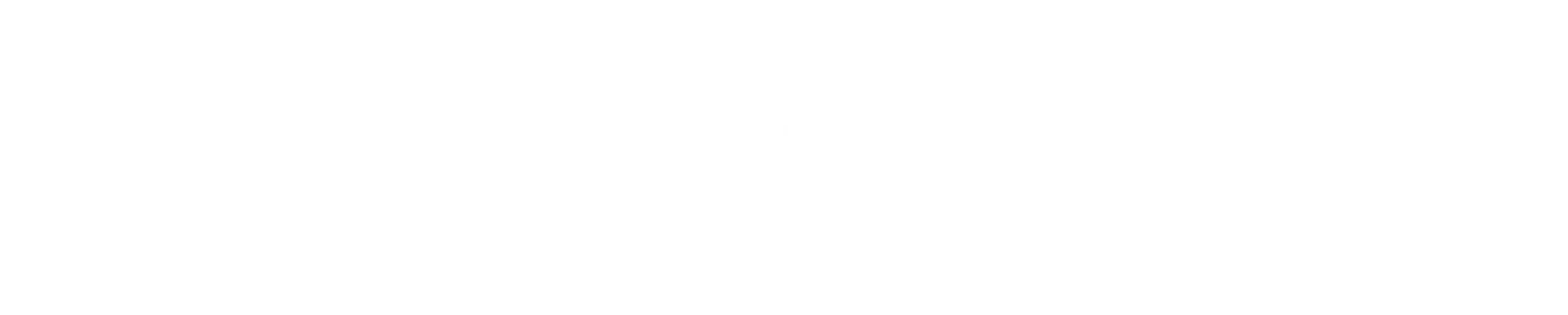
[Event Handlers 5](#_Toc87178559)

[Arrival Handler 5](#_Toc87178560)

[Extending the System 7](#_Toc87178561)

[Sending and Receiving Jobs 9](#_Toc87178562)

We will not begin the **performance analysis** of a **queueing system** with **multiple servers**. What we mean by this is that the servers are connected **one after another**, each with their own queues.



For simplicity, we will be dealing with just **two servers**, but the code we will create will be **extensible**, to accommodate as many servers as we wish.

As can be seen in the diagram above, the **initial entry** **rate** is , the **departure rate** from the **first server**, which is also the **entry rate** for the **second server**, is and the final **departure rate** from the **second server** is .

## Events and State Variables

There are **three events** in this system, the **arrival event**, the **departure event** from **server 1** and the **departure event** from **server 2**.

There are **four state variables**, q[1].length, q[2].length, s[1].status and s[2].status.

For each event, the **state variables** will change. Here, we are describing the changes that occur during each event, so that corresponding **state equations** can be created for each of the state variables.

### Arrival Event

if (s[1].status == 0) s[1].status = 1; *// if server free, make busy*else q[1].length++; *// else join queue*

PSEUDOCODE

### Server 1 Departure Event

if (q[1].length == 0) s[1].status = 0; *// if queue empty, make server free*else q[1].length--; *// else shorten queue*if (!s[1] == final\_server) *// if not the last server*{ *// handle arrival for next server* if (s[2].status == 0) s[2].status = 1;  
 else q[2].length++;  
}

PSEUDOCODE

### Server 2 Departure Event

if (q[2].length == 0) s[2].status = 0; *// if queue empty, make server free*else q[2].length--; *// else shorten queue*

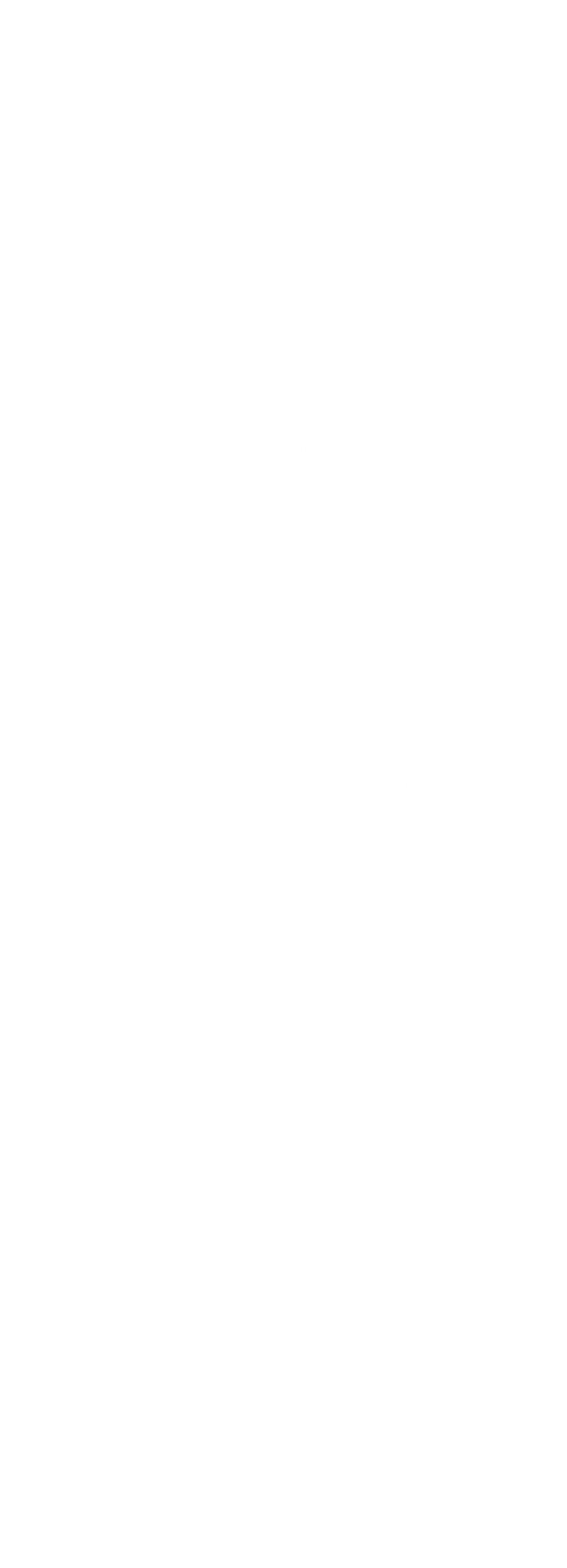
PSEUDOCODE

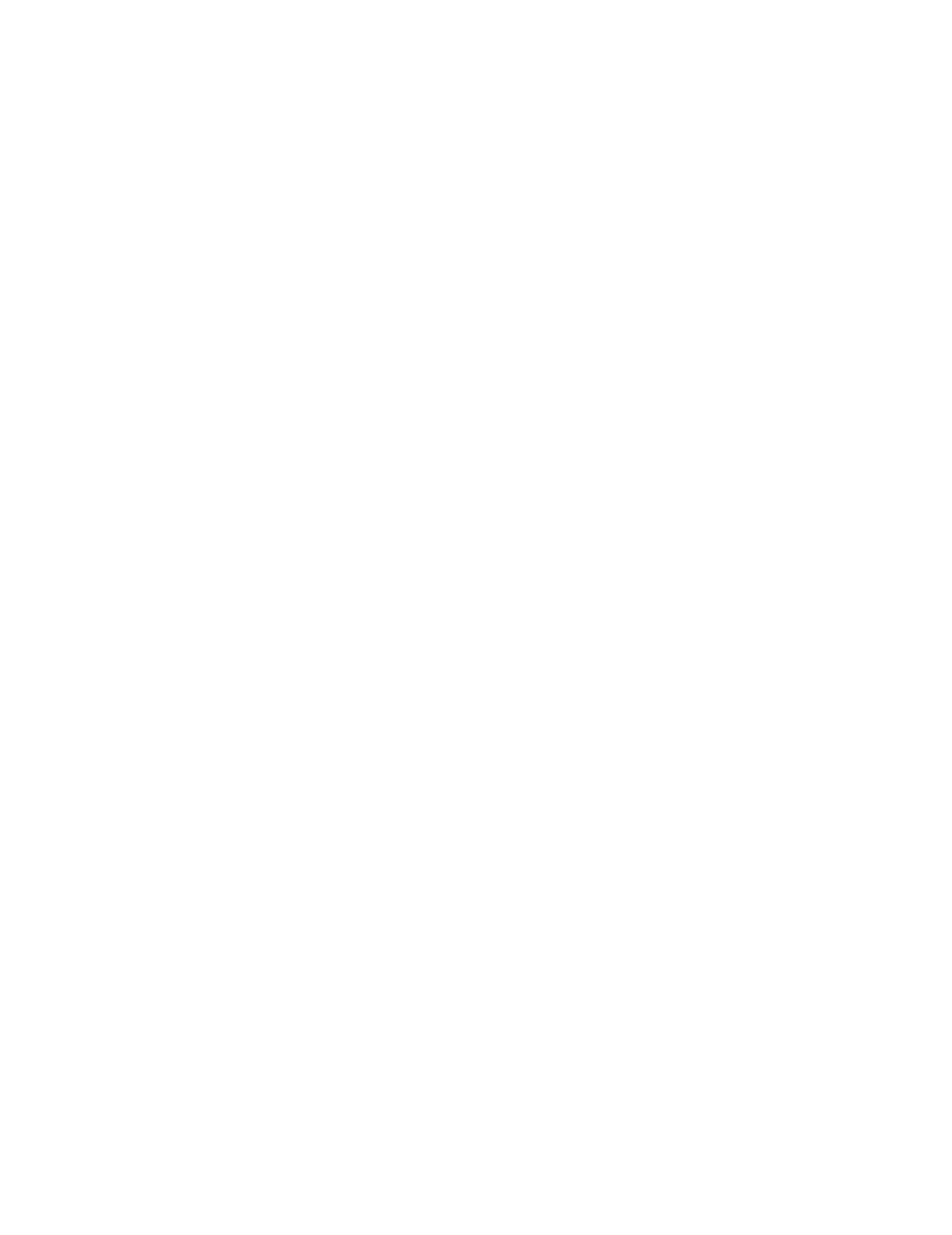
As stated before, the actual code will not have separate sections for each section, but rather will be modified to accommodate **both servers** using **reusable code**. To do this however, each **job** must keep track of which **server** it belongs to. This means each job must have some **variable**, such as server\_id.

## Event Handlers

### Arrival Handler

We need to accommodate both the **initial arrival** and movement from **one queue to another**. For this, the initial **arrival handler** will only **create the job** and then pass it to a different function, where it will be processed. This will allow us to **reuse** the code for the job processing for the other servers.

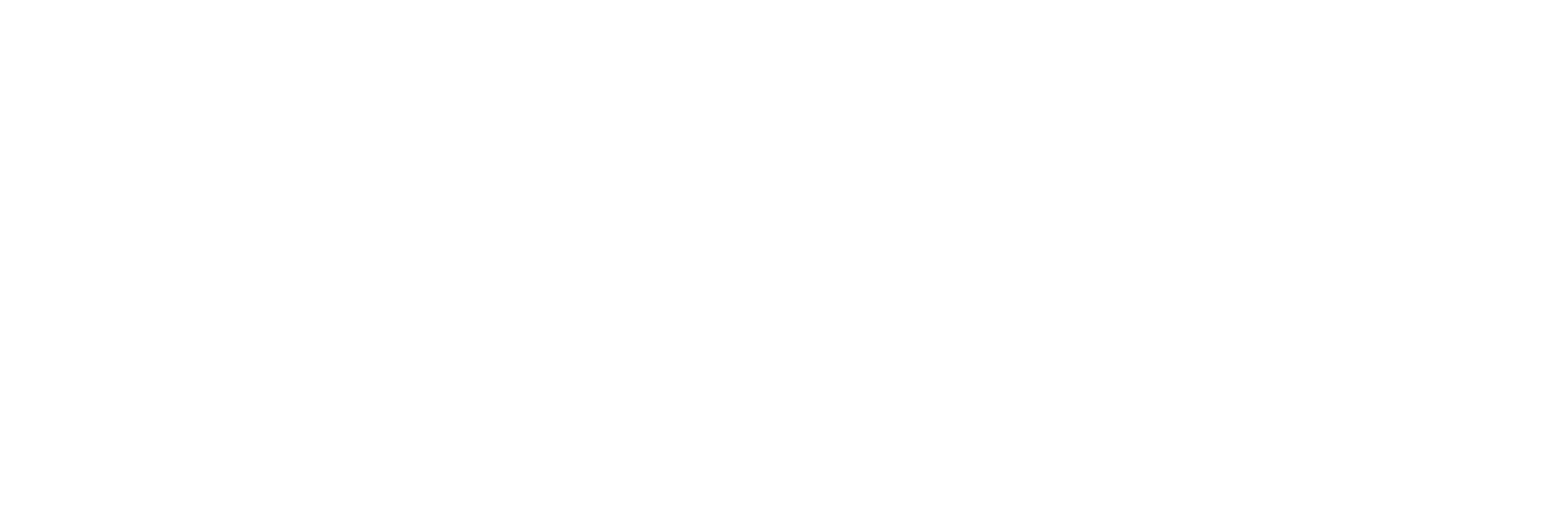




## Extending the System

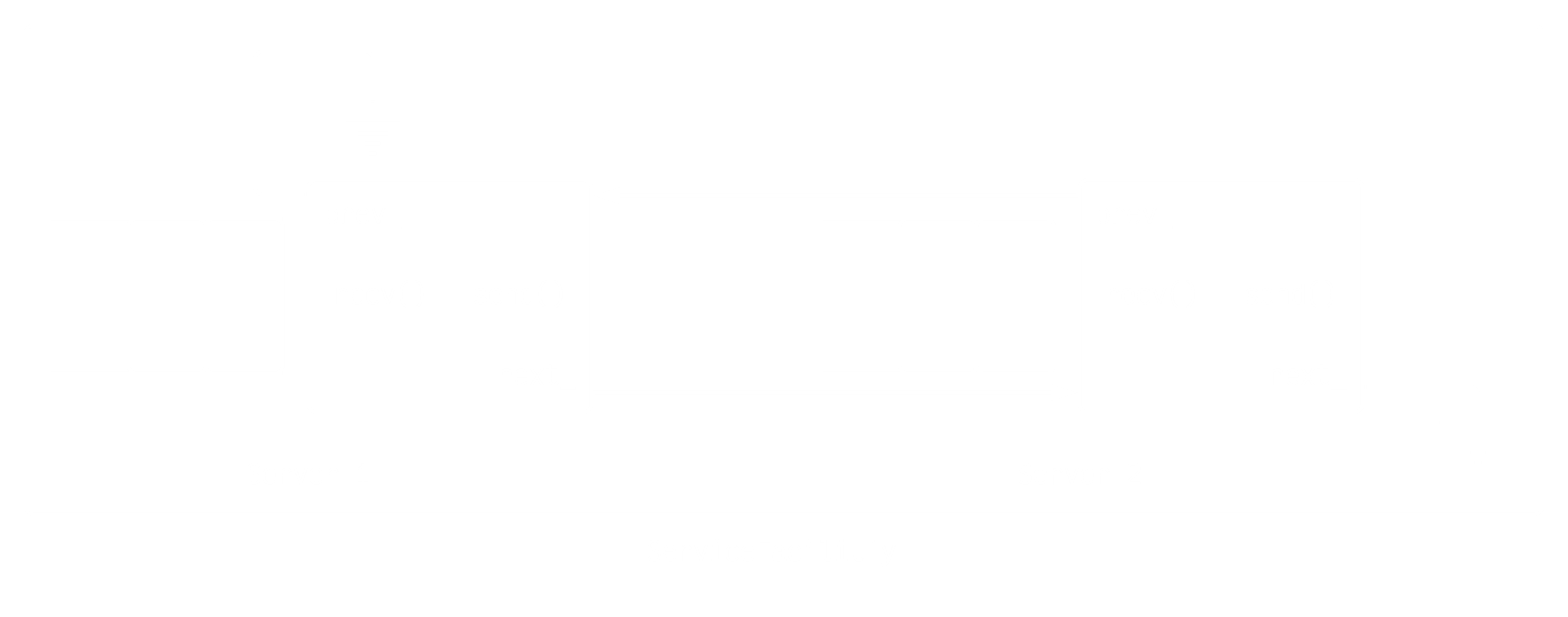
As mentioned before, we need to write out code in a way so that we can accommodate an **undefined number of objects**. The thing we are discussing right now is that we can have **multiple servers**, but the same can be true for **all entities** in the system, as we will see here.

Right now, we have **two servers**. Each server has a next pointer and a prev pointer which point to the **next server** and **previous server** respectively.



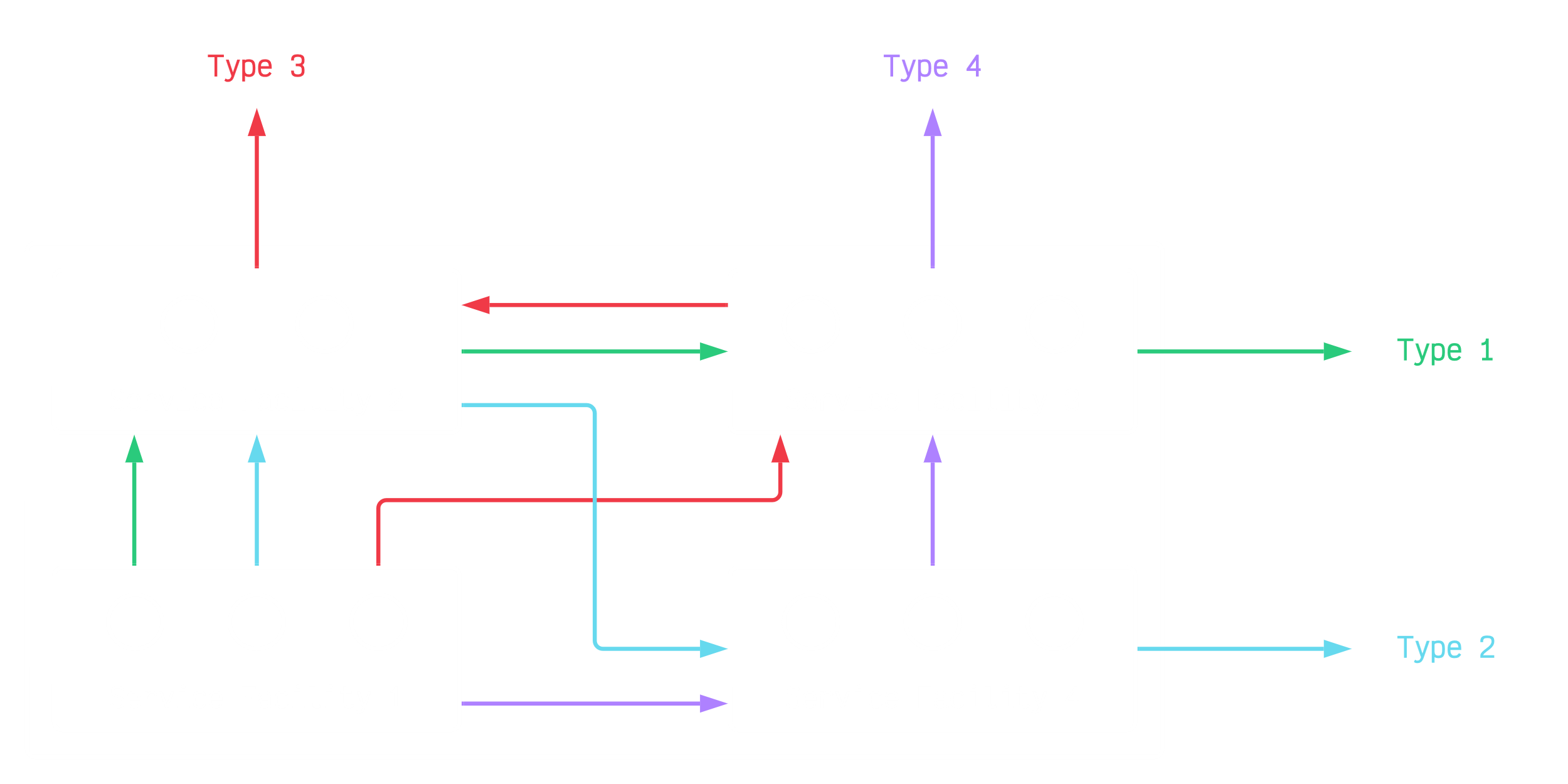
Since we want to be a bit more general, it might be beneficial to **inherit** the Server class we create from a new SimulationEntity class, which will in turn have the next and prev pointers.

Why would this be useful? Say we are dealing with a **large system** where the entire multi-server system we have created is just one of **multiple similar facilities**. Then we can **encapsulate** the entire thing in a ServiceFacility class, which also inherits from the SimulationEntity class.



Insider the ServiceFacility class we would simply create **new servers** as needed.

We can create very complicated systems from this, such as one in which we have **multiple service facilities** and jobs go to each of them in a **different order** depending on the type of the job.



### Sending and Receiving Jobs

One important aspect of extending our system like this is that the entities must be able to **communicate** with each other. This is why we have the next and prev pointers. Say one server wants to **forward a job** to the next server. The first server could call a function, send() which would internally call another function from the second server, receive(). Thus, by deriving our servers from a common parent and giving them connections to each other, we are allowing them to **communicate**.