Howard Zhang

06/12/2019

CSC 536

Final Project: Distributed Ticket Kiosk

**Introduction:**

The distributed ticket kiosk consists of an application where a set of clients attempts to purchase tickets from a group of kiosks. The issues we must deal with are:

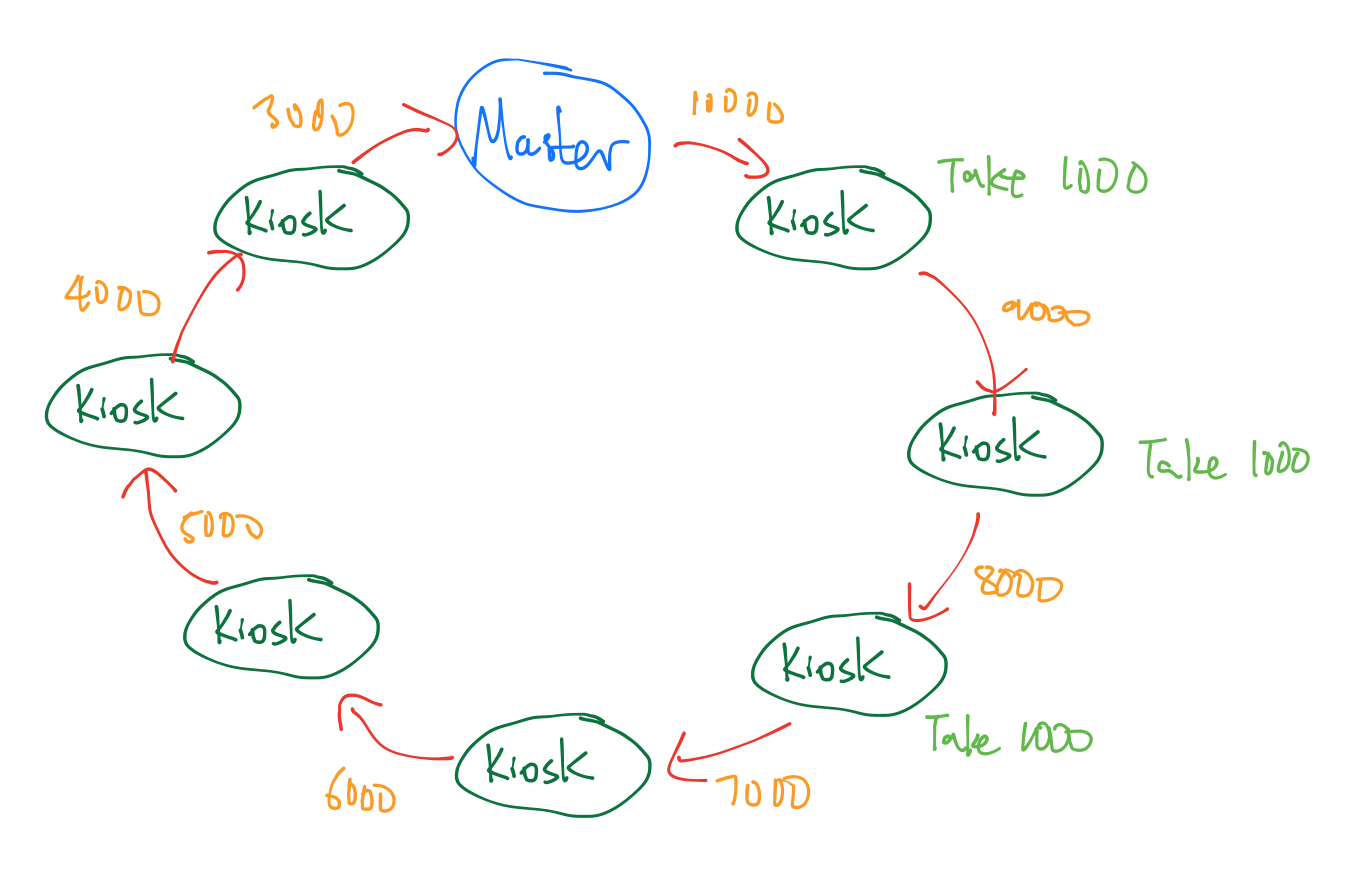
1. Liveness: All clients that want tickets should be able to buy one as long as tickets are available.
2. Safety: We cannot sell more tickets than are available. A stronger condition, though not implemented here, is that the same ticket (event, date, seat) should not be sold twice
3. Fairness: Clients should be issued tickets in the order that they accessed the kiosk

**How to Run:**

1. cd into the folder containing build.sbt
2. sbt
3. run

**Overall System Design:**

Kiosks are arranged in a message passing ring with a master kiosk. Each kiosk knows the location of its clockwise neighbor in the ring. The master kiosk initiates a token containing the number of tickets left for each event which is passed along the ring in a clockwise direction. Each time a kiosk runs out of tickets, it takes a designated number of tickets (chunk) from the ring and passes the token minus the chunk it took.



**Ticket Exchange:**

Once the master has run out of free tickets, we will need a way to exchange tickets between kiosks. There are many ways to implement this. I will first discuss the method I chose, and then other methods I considered.

case class MasterToken(numTickets: Int)

case class ExchangeToken(status: Array[Int], numTickets: Int)

My implementation choice was to create a separate token called an “Exchange” token. The exchange token acts like Robin Hood in that it steals from the rich and gives to the poor. Once the master kiosk runs out of initial tickets, it will start sending the Exchange Token around the ring instead. The behavior is below:

1. If a kiosk receives the exchange token and owns more than N tickets, then the exchange token will steal the surplus from the kiosk.
2. If a kiosk has fewer than N tickets, then the exchange token will give tickets to the kiosk until it has N tickets

Note that N is an arbitrary number that should ideally be calculated based on the number of servers and the size of each chunk.

The Exchange Token has the additional job of tracking whether the event is sold out. Each exchange token contains a “status” variable which is an array with a size corresponding to the number of kiosks. A kiosk will set the array index corresponding to its kioskID if:

1). The number of tickets in the kiosk is 0 and

2). The number of tickets in the token is 0.

If the master kiosk receives an exchange token such that all the values in the status array are set, then it will know that the event is sold out and will stop sending the exchange token. If not, then the values are reset and a new exchange token is sent.

**Other approaches I considered:**

1. Have each kiosk broadcast the number of tickets it owns to each other kiosk. Kiosks that run out of tickets request tickets from the kiosk with the highest number of tickets.
2. Each kiosk requests tickets from its clockwise neighbor only.
3. The master keeps track of how many free tickets each kiosk has, and each time a kiosk runs out of tickets, it asks the master for which kiosk to request tickets from.

The benefit of my approach is that it is simple and guarantees liveness. Simplicity is achieved because kiosks do not need to know about the state of every other kiosk. We also do not need to worry about consistency between kiosks.

Liveness is guaranteed (there are no unsold tickets) because the exchange token is a singleton. This means that there is no possibility of other kiosks exchanging tickets amongst each other which the token is traveling. If the token reaches the master and all the bits are set, indicating no more tickets, then we are certain that no tickets were missed.

The drawbacks of my approach is that my implementation is slower and less scalable than the other approaches. Having a singleton exchange token means that if we have 1000 kiosks, in the worst-case scenario, a kiosk with no more tickets left may have to wait until the token has passed through 999 other kiosks before receiving more tickets. The key is to choose of value of N such that the kiosk will likely still have tickets left before the token comes around.

**Replication:**

Replication of data is achieved through using a log file. This simulates servers storing their data to a hard drive. Each kiosk writes a ticket transaction to a log file before issuing the ticket. This way, if the kiosk crashes, the master actor or a supervisor actor can restart the kiosk by loading the log file corresponding to the kioskID and recreating its database without losing tickets.

private def issueTicket(): Ticket = {

println(self.path.name + " sold a ticket")

val file = new File("Kiosk"+kioskID+".txt")

val bw = new BufferedWriter(new FileWriter(file, true))

val ticketID = kioskID \* 10000 + ticketsSold

bw.write("Ticket id: %05d \n".format(ticketID))

bw.close()

this.ticketsAvailable -= 1

this.ticketsSold += 1

val newTicket = new Ticket(ticketID, new Event("2Cellos", "Tomorrow"))

return newTicket

}

Other techniques I considered was replicating its data to the next N neighbors, as is the approach used by Amazon’s Dynamo. The issue with that is that each kiosk stores a large amount of data such that replicating every ticket transaction multiple times is not efficient.

Second, this will bring in issues of consistency. If we want to replicate the data to the next N servers, do we choose strict consistency or eventual consistency? If we choose strict consistency, then ticket transactions will be slow, because we must have N servers agree on the transaction before we can issue a ticket. If we use eventual consistency, then we must worry about different versions of the same data replicated over different kiosks. This can be resolved using **Vector Timestamps**, but then I would have to implement those into my system.

**Fault Tolerance:**

There are two main situations that we must account for with a message passing ring.

1. If a kiosk crashes, how do we guarantee the token can still be passed through the ring?
2. If a kiosk crashes while in possession of the MasterToken / ExchangeToken, how to we guarantee that the token is not lost?

private var nextKiosk: ActorRef = null; // clockwise neighbor

private var kioskList: IndexedSeq[ActorRef] = null; // list of all kiosks in case neighbor goes down

The way I deal with the first situation is that each kiosk contains a list of actorRefs to each other kiosk. This way, each Kiosk knows its position in the message passing ring, and can bypass its neighbor if needed.

For example, suppose that Kiosk8 goes down. Kiosk7 passes a message to Kiosk8 but the message fails, so Kiosk7 will pass the token to Kiosk9 instead. To find Kiosk9, it simply looks for kioskList(kioskID + 1) in the List of ActorRefs. In this manner, it could bypass as many neighbors as necessary to deliver the token.

What about is the Kiosk goes down while it holds the token? Though I do not have this fault tolerance step implemented, this is how I would do so:

1. Each Ticket token has a counter of how many times it has been passed around the ring (similar to a Vector Timestamp, but not quite)
2. Each Kiosk stores the cycle count of the last token, and the number of free tickets available after it donated or took its share from the token
3. The Master Kiosk broadcasts a heartbeat message to every kiosk if the token does not return in a certain period of time. Each kiosk replies with the cycle count and free tickets of the last Token it received. The kiosk with the highest cycle count and kioskID will reinitiate the token.