COMP 6216
Discrete-event simulations

thanks to Jason Noble, Brendan Neville

Gradual change or specific events?

Continuous-time simulations work well when modelling processes of gradual change: an accelerating car or a growing population.

What about systems where specific events lead to relatively sudden changes in state?

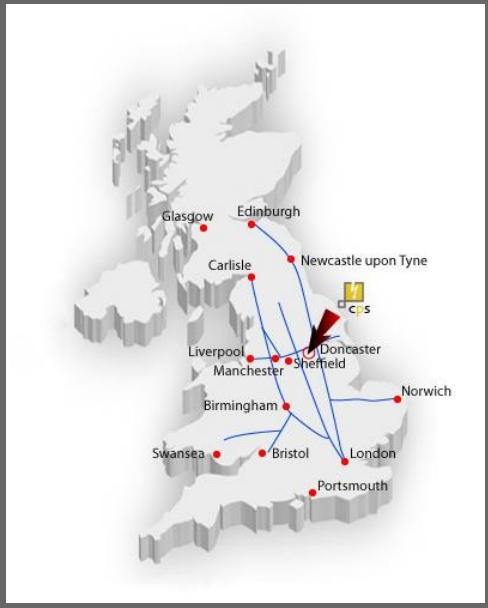
Consider a fleet of trucks used to pick up goods from suppliers, store them in a central depot, and deliver them to stores.

Or patients moving through a hospital: a single patient might arrive in casualty, be moved to an operating theatre, then a recovery room, then a ward.

Disadvantages of continuous time

What would happen if we tried to model a truck delivery network using continuous time?

T = 0 sec, truck leaves Sheffield
T = 1 sec, truck 1.2 m closer to London
...
T = 11340 sec, truck arrives in London



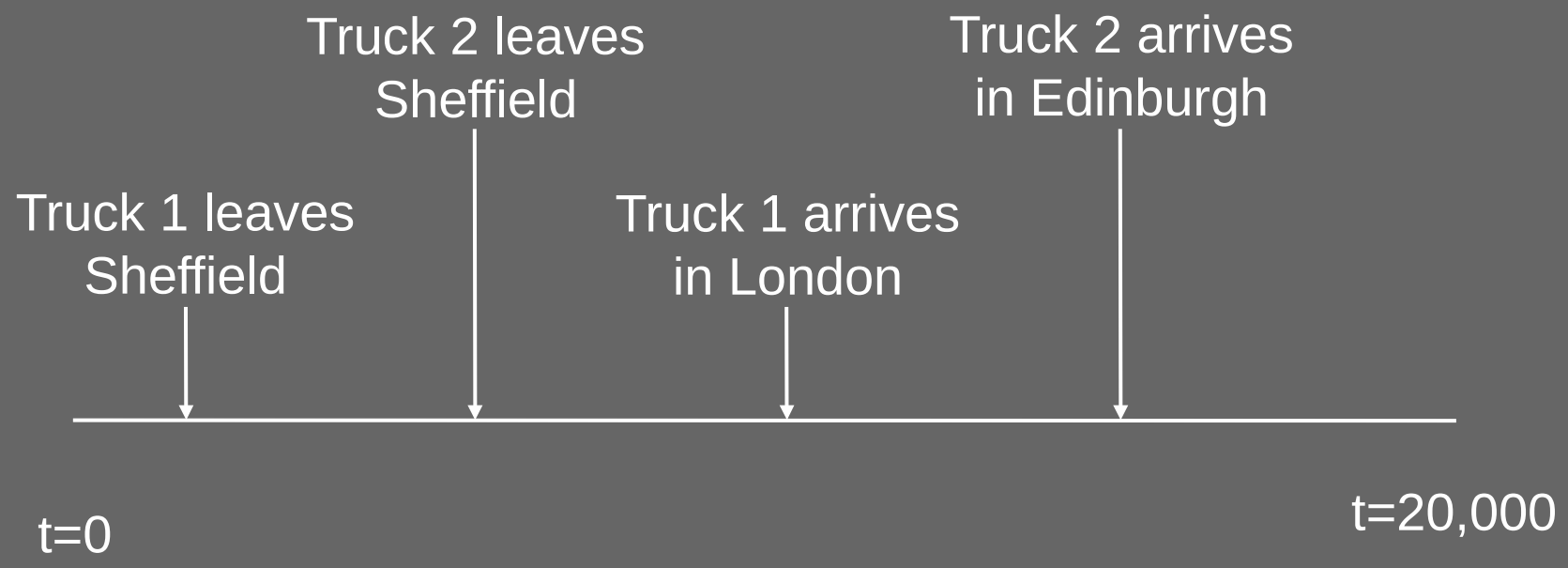
Continuous-time has a downside: empty clock ticks where nothing happens.

Real computation time wasted on thousands of updates that do nothing.

Focusing on distinct events

What if we could fast-forward to the interesting bits?

We can if we handle time using an *event list:* this is the key concept in discrete-event simulation.



Focusing on distinct events

The simulation skips forward to the next event.

Each event changes the state of the system.

We get big gains in computational efficiency, but handling time in this way means:

1. we need to think about distributions that describe the length of events and the duration of gaps between events.
2. some of the statistical housekeeping gets trickier, because there's no regular clock-tick.

Fast food restaurant problem

Fast food restaurant problem

You're the owner of a fast-food restaurant. You don't want to make customers unhappy with long queueing times, but you also don't want to employ more people than you need.

Right now you have six service tills, and 30 tables. About 120 customers enter every hour. Getting served, if there's no queue, takes on average 1min 40sec. Eating your meal at a table takes on average 10 min.

Six queues for service, but people can hop to shorter queues. One queue (if needed) for getting a table. Everyone wants to eat in. Customers facing queues > 10 people will walk out.

Fast food restaurant events

Note the presence of queues and relatively independent processes. This is a typical case for discrete-event simulation.

We need to identify the events we will simulate:

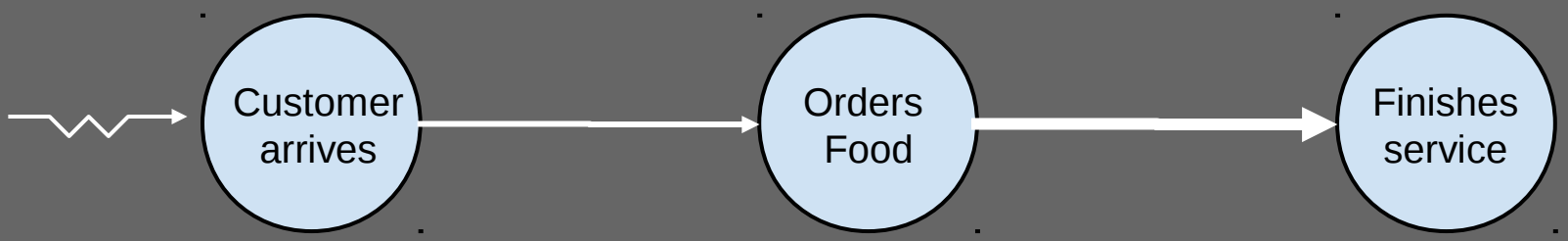
1. arriving
2. joining a service queue
3. reaching the head of the queue
4. placing your order
5. getting your food
6. joining the queue for a table
7. getting a table and sitting down to eat
8. leaving the restaurant once you've finished

Fast food restaurant events

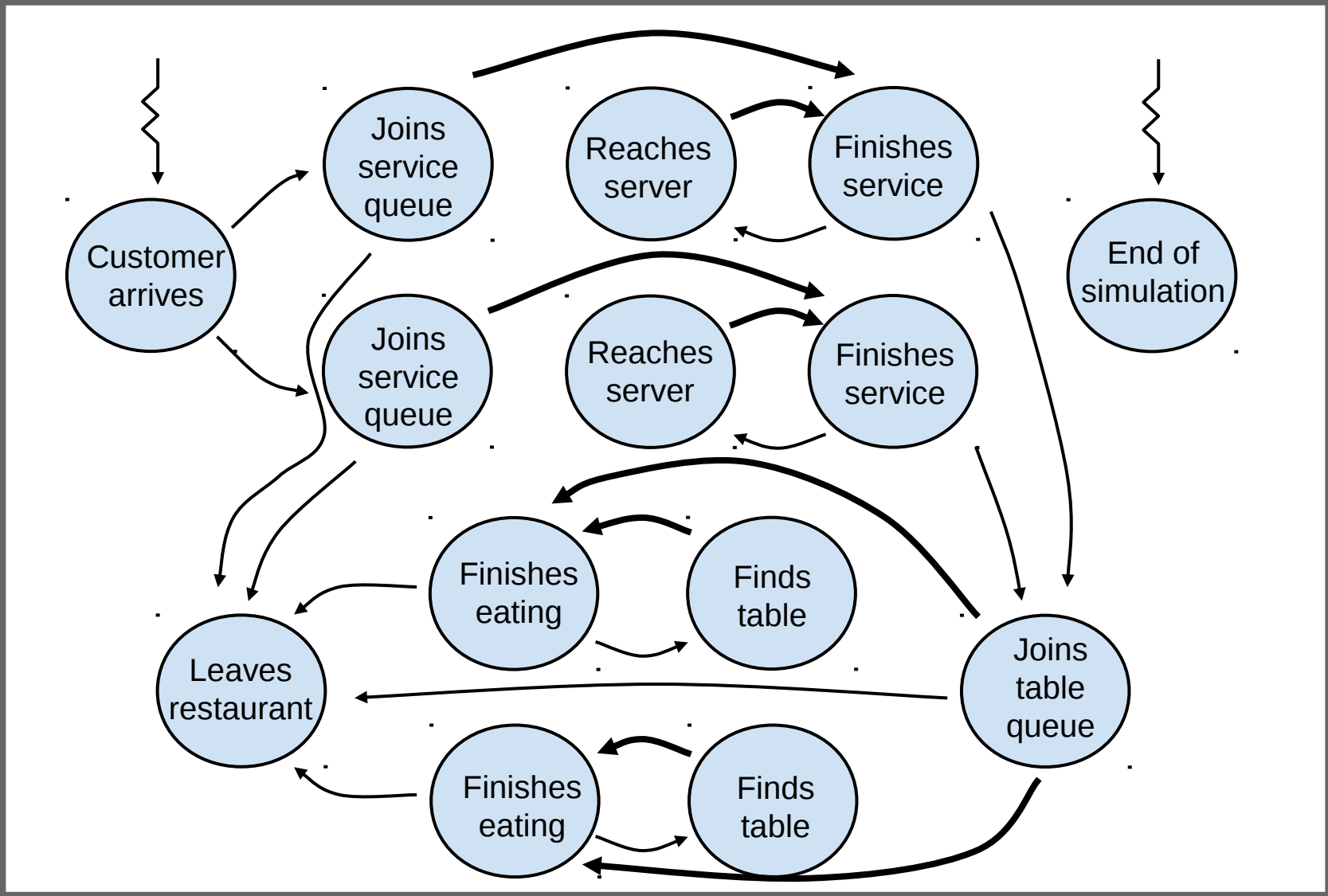
Next step is to display events using an *event graph*.

Conventions: thin lines join events that happen simultaneously. Thick lines join events that will be scheduled at some point in the future. Jagged lines indicate events that must be scheduled as part of the initialization of the program.

We need to cover all of the events, but to keep the program as simple as possible, it turns out that we can merge events that are only connected by thin lines.



Fast food restaurant event diagram



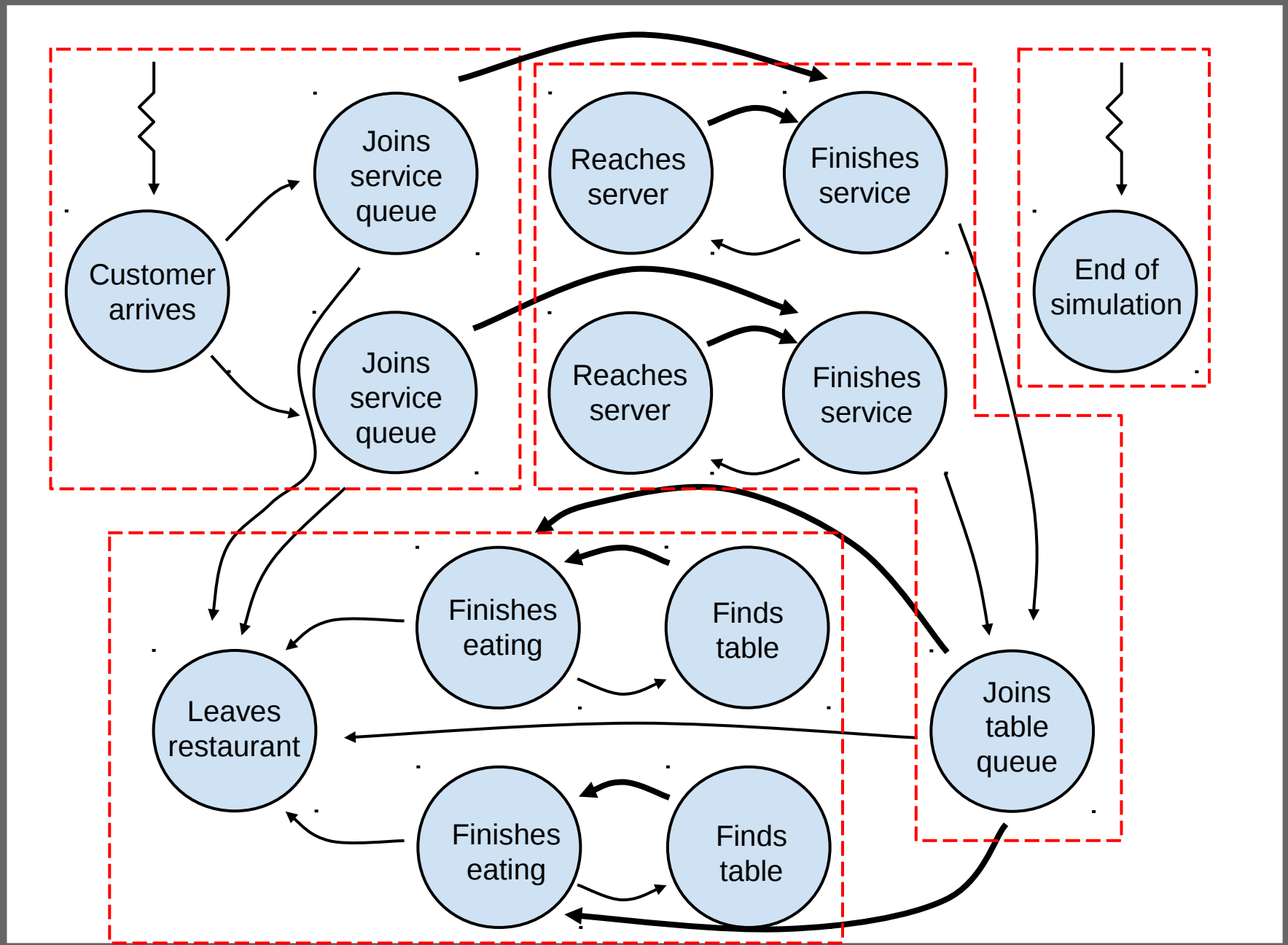
Fast food restaurant events

We can get by with only 4 general classes of event:
1.arrival
2.server completion (one event type for each server)
3.table departure (one event type for each table)
4.end of simulation event

Code for a simulation of the restaurant is available:
http://users.ecs.soton.ac.uk/jn2/simulation/discrete.html

Download "fastfood.py" or "fastfood.c".

Fast food restaurant event diagram



Components of discrete-event sims

1. The system state (many variables)
2. The simulation clock (skips rather than ticks)
3. The event list
4. An initialization routine (setting up at the start)
5. A timing routine (what event comes next?)
6. A routine for each class of event (what does it do?)
7. Statistical counters

```
int main ( void )
{
                        /* Set statistical counters to zero, etc. */
  initialize();

  while(1)                  /* An infinite loop.  Needs to be broken by an "end of simulation" event */
    {
      timing();                         /* Determine the type of event that occurs next */
      updateTimeAvgStats();         /* Update relevant counters */

                        /* Perform the updates appropriate to the event */
      if ( nextEvent == arrivalEvent )
        arrival();
      else if ( nextEvent >= serverCompletionEvent && nextEvent < tableDepartEvent )
        serverCompletion(nextEvent - serverCompletionEvent);
      else if ( nextEvent >= tableDepartEvent && nextEvent < endOfSimulationEvent )
        tableDepart(nextEvent - tableDepartEvent );
      else if ( nextEvent == endOfSimulationEvent )
        break;
    }

  report();                         /* Display output that summarizes the run */
}
```

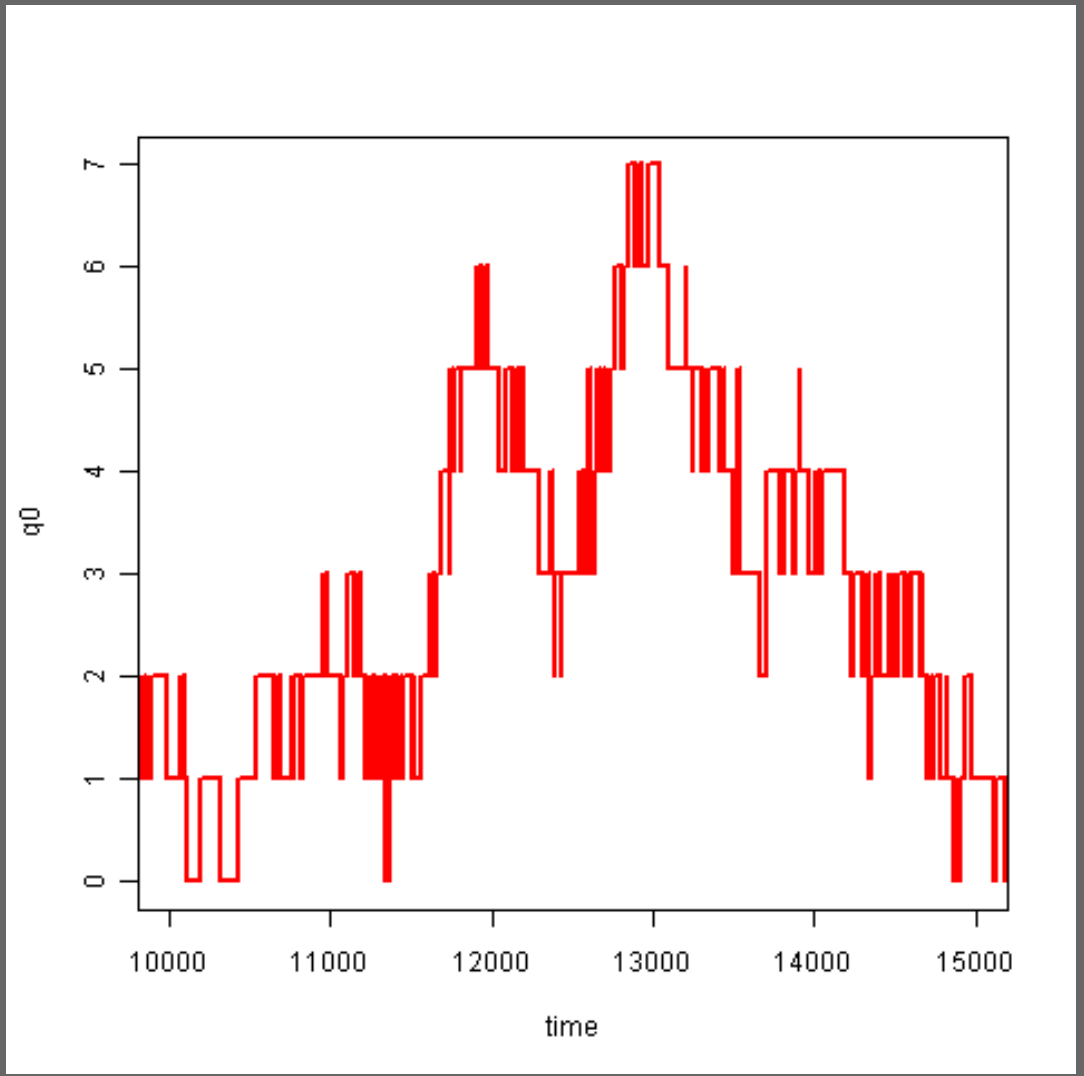
Some results

With 6 servers, the average waiting time before a customer gets to eat is 2min. Servers are busy 56% of the time, and table occupancy is 67%.

With only 4 servers, waiting time increases to 5min 51sec. Servers are busy 85% of the time, and table occupancy is 64%.

Where could the owner save money?

Average queue length (4 servers, queue 1)



Scheduling and processing events

Discrete-event simulation makes us think harder about time.

Looking at the code for the arrival event, we see that the event re-schedules itself, i.e., when one person arrives, we post a future event that represents the arrival of the next person.

We said that 120 people arrive per hour, i.e., one every 30 seconds. But they won't walk in *exactly* one every 30 seconds on the dot.

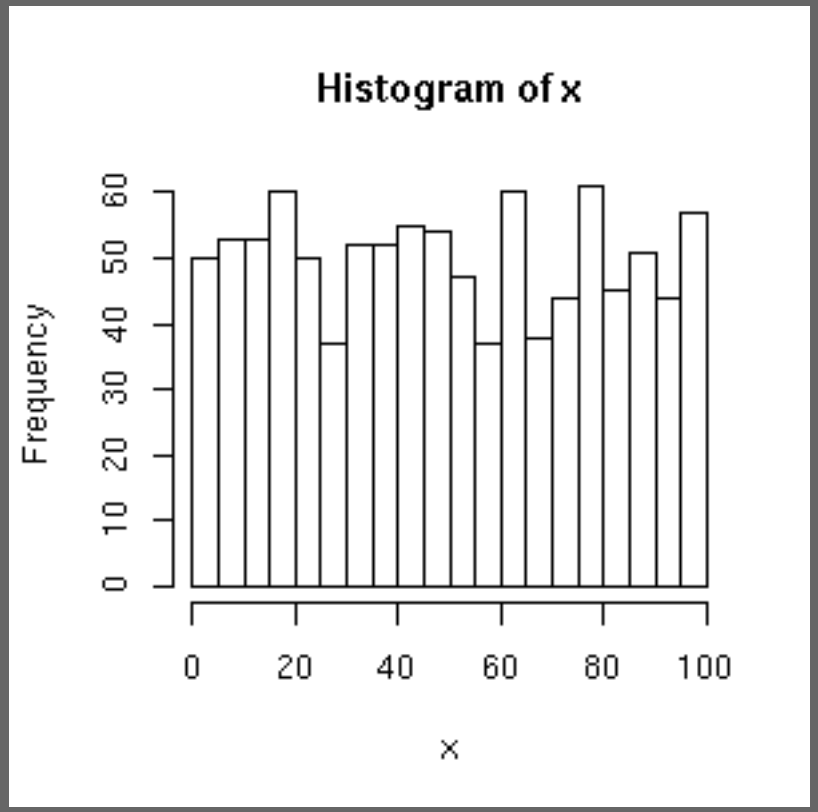
How should the inter-arrival times be distributed? We need to know, to post the next arrival to the event list.

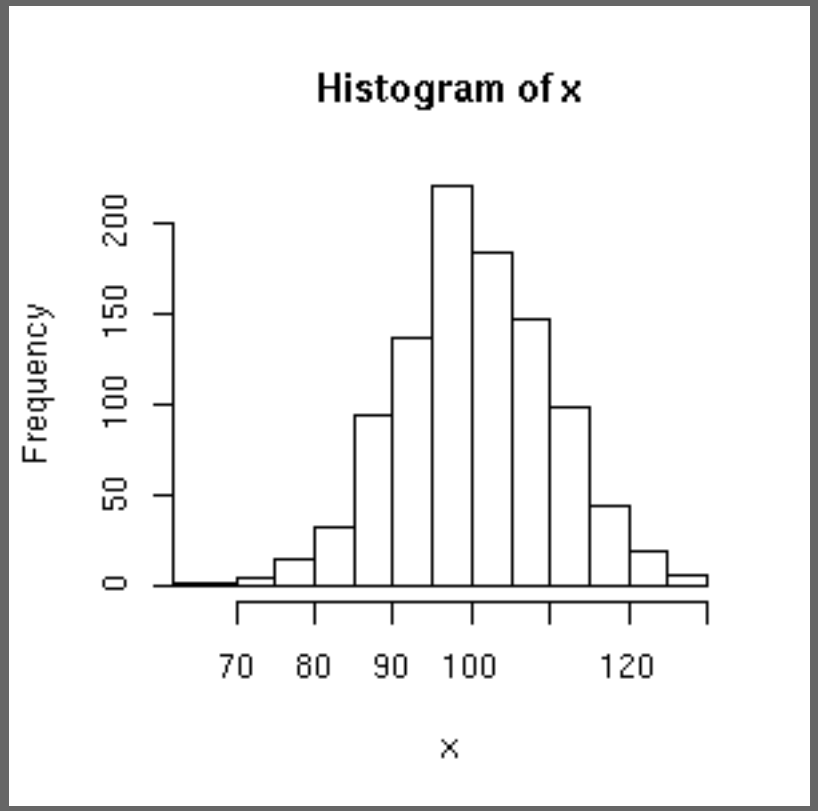
```
int main ( void )
{
                        /* Set statistical counters to zero, etc. */
  initialize();

  while(1)                  /* An infinite loop.  Needs to be broken by an "end of simulation" event */
    {
      timing();                         /* Determine the type of event that occurs next */
      updateTimeAvgStats();         /* Update relevant counters */

                        /* Perform the updates appropriate to the event */
      if ( nextEvent == arrivalEvent )
        arrival();
      else if ( nextEvent >= serverCompletionEvent && nextEvent < tableDepartEvent )
        serverCompletion(nextEvent - serverCompletionEvent);
      else if ( nextEvent >= tableDepartEvent && nextEvent < endOfSimulationEvent )
        tableDepart(nextEvent - tableDepartEvent );
      else if ( nextEvent == endOfSimulationEvent )
        break;
    }

  report();                         /* Display output that summarizes the run */
}
```

```
void arrival ( void )
{
  int i, shortQueue;
  int lengthShortQueue = MAX_SERVER_QUEUE + 1;

  if ( VERBOSE )
    printf("%.2f: someone arrives.\n", simTime );

  totalArrivals++;

                          /* Schedule the next arrival event */
  timeNextEvent[arrivalEvent] = simTime + expon( MEAN_INTERARRIVAL_TIME );

                          /* Check the length of the queues: find the shortest */
  for ( i = 0 ; i < NUM_SERVERS ; i++ )
    {
      if ( serverQueue[i] < lengthShortQueue )
        {
          lengthShortQueue = serverQueue[i];
          shortQueue = i;
        }
    }

                          /* Three possibilities: all queues too
                             long and they go away, they join
                             the short queue, or the short queue
                             is empty and they go straight to
                             the counter */
```

```
  if ( lengthShortQueue >= MAX_SERVER_QUEUE )
   {
     if ( VERBOSE )
       printf("All queues are too long and they leave immediately.\n");

     totalTurnedAway++;
   }
  else if ( lengthShortQueue > 0 )
   {
     serverQueueArrivalTime[shortQueue][lengthShortQueue+1] = simTime;
     serverQueue[shortQueue]++;

     if ( VERBOSE )
       printf("They go to place %d in queue %d.\n",
              lengthShortQueue+1, shortQueue );
   }
  else
   {
     serverQueueArrivalTime[shortQueue][lengthShortQueue+1] = simTime;
     serverQueue[shortQueue]++;

                          /* This server is now busy; schedule server completion event */
     serverStatus[shortQueue] = BUSY;
     timeNextEvent[serverCompletionEvent+shortQueue] =
       simTime + erlang(MEAN_SERVICE_TIME, 2);

     if ( VERBOSE )
       printf("They walk up to server %d and order.\n", shortQueue );
   }
}
```

The uniform and normal distributions





Neither of these two familiar distributions is appropriate.

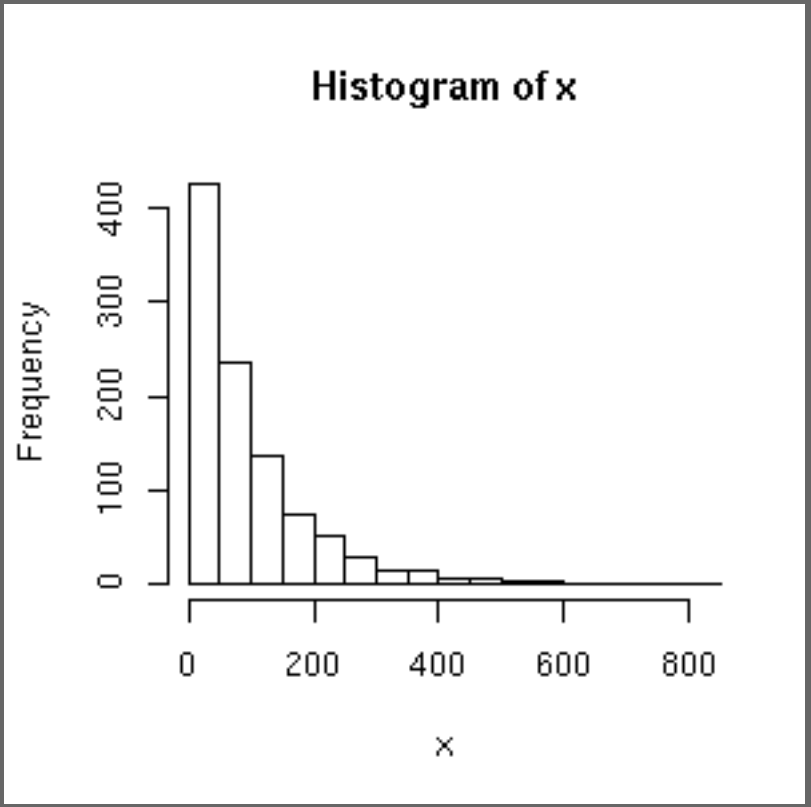
They both have "memory", i.e., the imminent arrival of the next customer could be predicted as you near the maximum.

The exponential distribution

The exponential is the only "memory-less" distribution, due to its self-similar shape.

If it has been 400sec since the previous customer, the chances of one walking in during the next 10sec remains the same.

Thus the exponential is used in discrete-event simulation to model the duration of gaps between independent events.



Modelling the duration of events

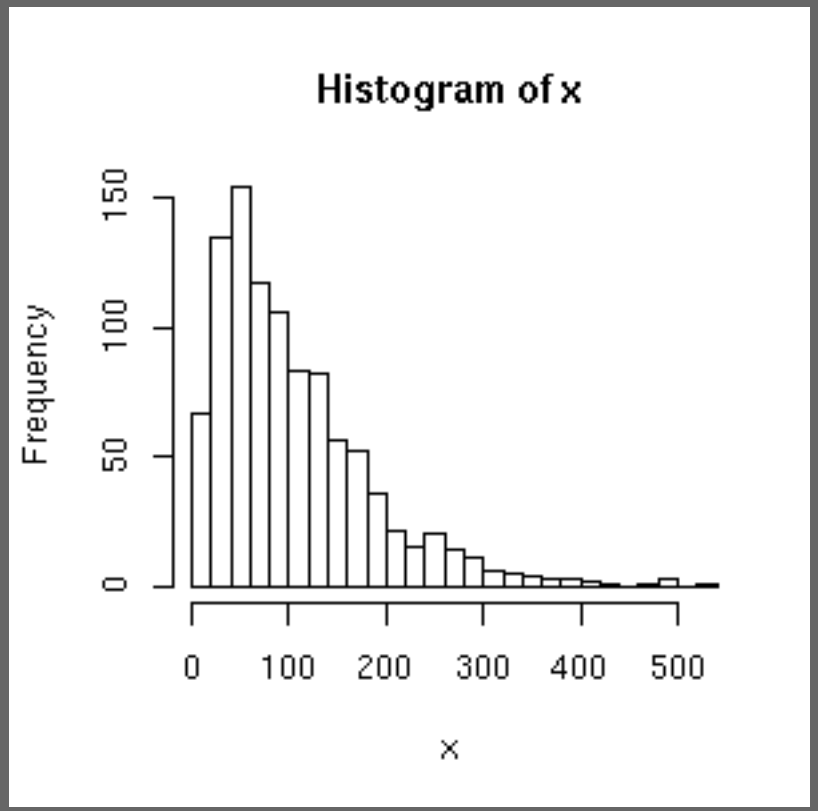
We also need to know how long things take.

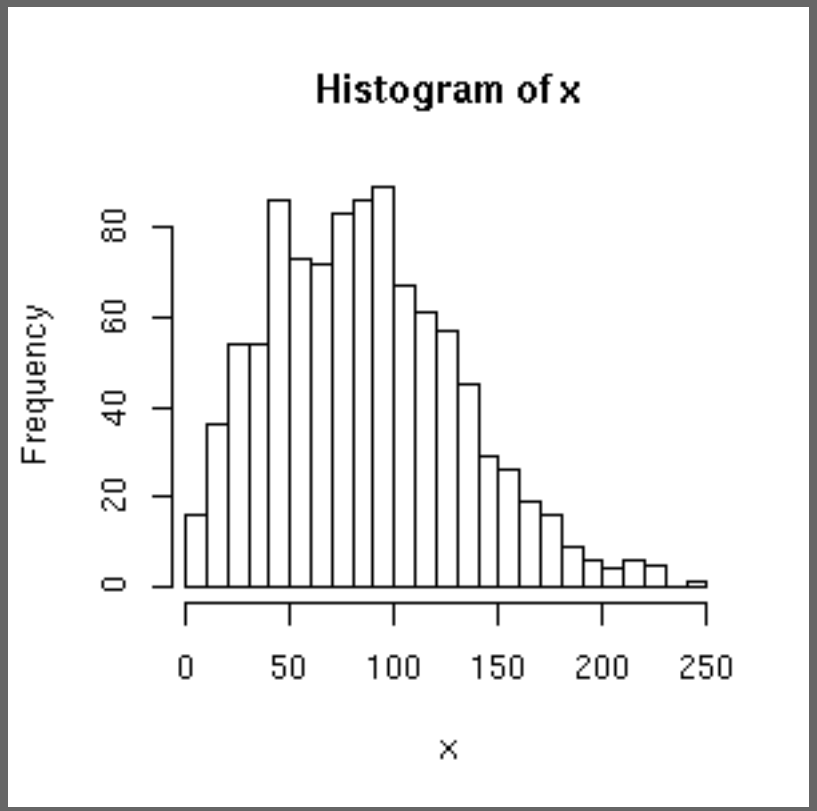
What is the distribution of durations for cooking a burger, eating a burger, repairing a truck, conducting an operation, etc?

Most times will cluster around the median value. There will be a minimum practical length and nothing can happen in zero seconds or less. But some service durations will be extremely long, as anyone who has ever stood in a queue at the bank knows.

There are some distributions that capture these properties.

The Erlang and Weibull distributions





The fast food simulation uses an Erlang distribution of shape parameter 2 (as above, left) to model service durations and meal durations.

Keeping track of statistics gets harder

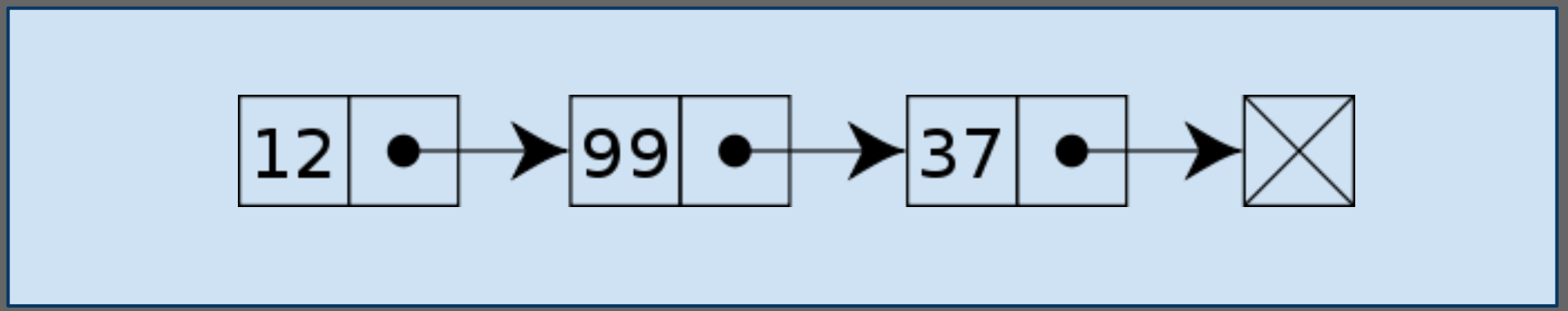
```
void updateTimeAvgStats( void )
{
  int i;
  double timeSinceLastEvent;

                          /* Calculate time since last event,
                             and update the marker for the last
                             event */
  timeSinceLastEvent = simTime - timeLastEvent;
  timeLastEvent = simTime;

                          /* Update the area under the curve for
                             the time-averaged statistics */
  for ( i = 0 ; i < NUM_SERVERS ; i++ )
    {
      areaServerStatus[i] += (double)serverStatus[i] * timeSinceLastEvent;
      areaServerQueue[i] += (double)serverQueue[i] * timeSinceLastEvent;
    }

  for ( i = 0 ; i < NUM_TABLES ; i++ )
    {
      areaTableStatus[i] += (double)tableStatus[i] * timeSinceLastEvent;
    }

  areaTableQueue += (double)tableQueue * timeSinceLastEvent;
```

Representing event lists

In fastfood.c there are few event types, and so the event list is managed simply by keeping track of the time at which the next one of each kind is due.

The timing routine notes which event type is most imminent.

A more general way of constructing event lists is to use *linked list* data structure.



Representing event lists

A high-level language like Python makes all this very easy.

In fastfood.py the event list is simply a Python list, storing objects of class Event in temporal order, with new objects being inserted at the right point based on their .time field.

References

A comprehensive text on discrete event simulation:

Law, A.M. and Kelton, W.D. (2000). Simulation Modelling and Analysis. 3rd edn. McGraw Hill.