EVENT-DRIVEN SIMULATION SAMPLE CODE DESCRIPTION

YE XIA

1. Description

1.1. **The Model.** The sample code simulates the following system, which may resemble a datacenter with a large number of servers (see Fig. 1).

Requests arrive according to a Poisson process with rate λ (requests per second). In other words, the inter-arrival times of the requests are IID exponential random variables with mean $1/\lambda$.

Each request i will generate a workload W_i. We assume W_i's are IID. We can use W to represent a generic workload, with distribution function F_W.

In the first stage of processing, a request is immediately dispatched to an idle server for processing, for instance, to gather the requested file. We assume there is an infinite number of servers and each server can only process one request at a time.

In the second stage, after a server finishes servicing a request, the requested file enters a FIFO queue with transmission rate μ. We assume the buffer capacity is infinite.

Effectively, we have an $M/G/\infty$ queue in the first stage, followed by a FIFO queue in the second stage.

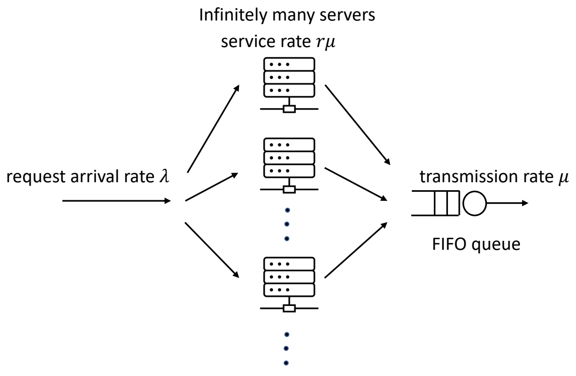

FIGURE 1. Simulation Model

1.2. **Parameters.** In the code, the request arrival rate λ is a user-specified input parameter. The system load, denoted by ρ, is another user-specified input. Since there is an unlimited number of servers, the system load is defined with respect to the FIFO transmission rate μ.

In the code, the transmission rate μ does not show up directly. Here are the details. Suppose the random variable S represents the size of a generic requested file in bits, and let $\overline{S}$ be the mean file size. Then, $\lambda\overline{S}$ is the arrival rate to the FIFO queue in bits per second. If μ is measured in bits per second, then the system load is $\rho = \lambda\overline{S}/\mu$.

The workload W is defined as $W = S/\mu$, which has the unit of seconds. It is the amount of time needed to transmit a generic requested file out of the FIFO queue. The mean of the workload, denoted by $\overline{W}$, is $\overline{W} = \overline{S}/\mu$. Then, we have

$$\rho = \lambda\overline{W}.$$

In the code, we do not specify μ or S, but we work with the workload W. The user specifies the request arrival rate λ and the system load ρ in the input file. Then, the mean of the workload is calculated as:

$$\overline{W} = \rho/\lambda. \tag{1}$$

The user also specifies the distribution W, which can be either an exponential distribution or a Pareto distribution. In the exponential case, the workload mean is the only parameter needed for drawing samples from the distribution.

We now discuss the Pareto case. The CDF for an Pareto distribution is

$$F_W(x) = 1 - (K/x)^{\alpha}, \quad x \geq K,$$

where K and α are parameters. The mean exists for $\alpha > 1$ and the variance exists for $\alpha > 2$. Under the condition $\alpha > 1$, the mean is $\overline{W} = \frac{\alpha K}{\alpha - 1}$. Therefore, among $\overline{W}$, K and α, only two of them are needed and the third one can be calculated. In the code, the user specifies α in the input file. Then, K is calculated as $K = \overline{W}(\alpha - 1)/\alpha$, where $\overline{W}$ is calculated using (1).

Side Note: For $\alpha \in (1, 2]$, you will see that the simulation results vary a lot for different runs using different random seeds. This is because the variance is infinite when $\alpha \in (1, 2]$.

The server rate of each server is assumed to be proportional to μ, i.e., $r\mu$, where r is a user-specified input parameter. In the code and the input file, we actually call r `ServerRate`, as it is the normalized server rate. Again, there is no need of μ. If a request generates a workload w seconds, then it takes w/r seconds to process it at a server. For scheduling an event in the simulation, only the service time is needed.

The following are the mapping between the symbols used in this document and their variable names in the code.

- λ: `RequestRate`; user-specified input.
- ρ: `SystemLoad`; user-specified input.

- α: `Pareto_Alpha`; user-specified input.
- r: `ServerRate`; user-specified input.
- $\overline{W}$: `WorkloadMean`; calculated.
- K: `Pareto_K`; calculated.

2. How to Use

Library: For random number generation, we use the gnu scientific library (gsl). You will need to download the library and install it on your system.

Compilation: A Makefile is provided. On my Linux system, I type 'make' to compile the code. If it doesn't already work on your system, you may need to make some small changes to the Makefile.

Input: Most of the parameters are set by an input file. A sample input file is provided, called `in1`. You can create copies of it and make changes to the parameters to simulate different scenarios.

To run the program, you need to supply the input file name and a seed (nonnegative integer) for random number generation. The command line is something like:

```
./main in1 2
```

In the above, the number 2 is the seed number. For the same seed number and the same input file, different simulation runs will produce identical results. You vary the seed number to produce different realizations of the results, for the same input file. As you vary the seed number, you will be able to see if the collected statistics are stable over different runs. If not, you need to run the program for either more requests or longer simulated time. This is particularly relevant for the Pareto case with $\alpha <= 2$.

In the input file, you will see two input lines for two variables: `TotalRequests` and `TotalTime`. The simulation ends when both the following conditions are satisfied: the total number of finished requests exceeds `TotalRequests` and the total simulated time exceeds `TotalTime`. Usually, you only need to increase the value for `TotalRequests` to excecute more events.

The workload distribution may either be exponential (with a value 1) or Pareto (with a value 2).

Note: Empty lines in the input file may cause problems. If you need to have something like an empty line for better formatting, use # or +, followed by nothing.

Output: The simulation output are the averages of various statistics.

- Average number of requests in service: This is the average number of requests in the first-stage servers, seen by the arrivals of requests to the system. It is also the average number of servers in use.

- Average number of requests in queue: This is the average number of requested files in the FIFO queue, seen by the arrivals to the queue.
- Average queueing delay: Unit is in seconds. This is the average queueing delay, averaged over the finished requests.
- Average response time: Unit is in seconds. The response time of a request is from the time the request arrives at the system, till the time the request is finished (i.e., the requested file left the FIFO queue).

There are not much surprises with the averages. You will see various quantities are related by Little's Law (check it out on line). The more interesting statistics, which are not collected by the program, may be the histograms of the delay, response time or the number of requests in the first or second stage. Recall that a histogram is the empirical version of a probability mass function. From a histogram, you will be able to estimate the variance and the tail probabilities.

For your own project, you should consider collecting some histograms.

3. Code Internals

The code can serve as a basis for your other simulators. Most of it are straight-forward. The following briefly describes the source files.

- `event.c, event.h`: Memory allocation and management for the event data structure; various interfaces to use the event priority queue.
- `packet.c, packet.h`: Memory allocation and management for the `Packet` data structure. Throughout, a `Packet` is a data structure that keeps track of a request as it goes through the system, including various significant times and the workload.
- `packqueue.c, packqueue.h`: Implement the FIFO queue as a queue of packets.
- `sim.h` Define/declare the data structures and global variables specific to this program that are shared by multiple source files.
- `main.c`: Main code for the simulator. Three types of events are defined: `request_arrival_event`, `end_of_service_event`, `end_of_transmission_event`.
- `input.c, input.h`: Read input parameters from an input file.
- `freelist.c, freelist.h`: These files implement a generic free list. To reduce the overhead for memory allocation, the program uses free lists for events and packets.
- `splay.c, splay.h`: Implement the priority queue using a splay tree. There is little need to change the code in these files.

Finally, for your own simulator, you will need to modify the `Event` and `Packet` data structures (in `sim.h`), or create something like those.