5. Performance Evaluation and Models

The various performance metrics we have examined so far enable us to speak of many aspects of a system's performance (e.g., utilization, response time, jitter, fairness, availability, etc.) However, given a system, how do we obtain these metrics?

5.1 Evaluation Approaches

There exist two main approaches to evaluate the performance of a system: *post-mortem* evaluation and *predictive* evaluation. We hereby summarize the main aspects of both.

5.1.1 Post-Mortem Evaluation

Of course, if the system is already "built" or developed, we can just simply measure these metrics. Indeed, once a system is designed and implemented, it is customary to characterize its performance by providing "data sheets" that would give those using such systems a good idea of what to expect of the system.

Obtaining performance metrics from a developed "real" prototype (implemented instance) of the system will clearly give us the most accurate quantification of these metrics, but it may not be so useful or practical after all[1]! In particular, if the purpose of evaluating these performance metrics is to choose between competing designs, then it is not practical to go ahead and develop all competing

[1] In the way of an analogy, consider weather forecasting. The most accurate measurement of (say) temperature would be of the actual (current) temperatures. But, besides measuring the current temperatures for purposes of record-keeping, measuring the current temperature is of little use. What we need is the ability to "forecast" (i.e. predict) what the temperature will be in the future. To do so, we resort to the use of analytical weather models and to the use of sophisticated simulations.

designs in order to measure their performance! Instead, what we need is the ability to "predict" what the system's performance will be **without** actually building it.

5.1.2 Predictive Evaluation

Rather than implementing the system, we may study its performance by simulating its behavior. In order to simulate a system, we must have a fairly good model of the system (e.g., what are the various processes involved, how they interact, etc.) Clearly, the ability of a simulator to predict the behavior of a system hinges on the accuracy of the models used for that system. Models may be quite simplistic – reflecting very rudimentary behaviors – or they may be quite elaborate – reflecting significant details. In practice, not all aspects of a system are modeled at the same level of details. For instance if we are building a simulator to study network performance, we may opt to have a very detailed model of how packets are queued, scheduled, and transmitted through a given router in the network. But, we may also elect to have a fairly simplistic model of other aspects of the system – say, what happens at other routers on the path between the sender and receiver. In general, any simulation model will be detailed with respect to the parts of the system "under the microscope," while simplifying assumptions will be made about other parts of the system.

Building accurate simulators and performing simulation experiments – while much cheaper than building the "real" system – requires quite a bit of investment, which may be warranted in later stages of a system's design, but not in early stages of system conception. At such stages, what a designer needs is a quick sense of how the system would behave under different configurations or workloads – what he/she needs is the ability to come up with "back of the envelop" calculations or "quick answers." For that, we would resort to the use of mathematical analysis of a model of the system.

Whether our prediction of the system's performance will be based on analysis or simulation, it is clear that we must develop a model of the system. System modeling (and analysis thereof) has been and continues to be a very rich and active area of research in computer science (and indeed in all applied science and engineering disciplines). System models could be thought of as "abstractions" that enable us to focus on what is important while "abstracting out" (or hiding) details that may not be important. Clearly, what we decide to reflect in our models will determine what sort of questions we will be able to answer through analysis of the models we construct. For example, in a model that only reflects system behavior at steady state (i.e., it abstracts out transient behaviors) we can only answer questions related to steady-state performance. Similarly, in a model that does not distinguish between various (say) classes of jobs in the system, we would not be able to speak of such things as "fairness" amongst various classes of jobs.

As we have alluded earlier in this course, one may view a computing system at any point in time as a collection of queues, reflecting different stages in which processes are awaiting service from various resources. Indeed, a very powerful tool that computer system designers often employ in modeling and analyzing computing systems is "queuing analysis." Queuing analysis enables us to make concrete statements related to the performance of the system at steady state[2].

[2]The concept of reaching a steady state is a complicated one, and in general is not well defined for real systems (since many assumptions made in abstract models of such systems do not hold). For now, we can assume that a system reaches

5.1.3 Nuts and Bolts of Queuing Analysis

Of all the analysis techniques that enable us to do "back of the envelope" estimation of computing systems performance, queuing analysis is by far the most important.

The basic entities in a queuing system are:

- **Customers:** These are the individual requests for service (e.g. a request for I/O, or a request by a customer in a bank, etc.).
- **Queues:** These are waiting areas where requests for service wait for servers (e.g. the ready queue of processes waiting for the CPU, or the waiting room at a doctor's office).
- **Servers:** These are the entities or resources that are capable of satisfying the service requests (e.g. CPU, disk, bank teller, etc.).

In addition to the above entities, we must discuss a number of other issues, including:

- **Dispatching Discipline:** Once a server is done serving a customer, it must pick the next customer out of some queue. The algorithm used to do so is termed the dispatching discipline. Possibilities (e.g. for scheduling purposes) include First-Come-First-Serve (FCFS), also called First-In-First-Out (FIFO), Shortest-Job-First, Earliest-Deadline-First, etc. We will see the impact of these scheduling techniques later in the course. For the purposes of this course we will restrict our analysis to FCFS.
- **Distribution of Arrivals:** For the purposes of this class we will restrict our analysis to a Poisson process for the arrival of customers (from the outside world) to the system. The rate of arrivals is λ. In other words, the number of customers coming into the system every period T is $\lambda \cdot T$. A Poisson arrival process implies that the arrivals to a system are independent.
- **Distribution of Service Time:** How long does it take a server to service a customer's request? Well, the service time may be the same for all customers (e.g. all customers request exactly the same service and it takes exactly the same time to serve each and every one of them). Alternatively, and more realistically, the service time is likely to be variable. For the purposes of this class we will restrict our analysis to the case in which the service time is a random variable that is exponentially distributed with a mean service time T_s. If the average time it takes a server to service a request is T_s, then it follows that the average rate of service (if the server has an infinite supply of requests to work on) would be $\mu = \frac{1}{T_s}$.

For the purposes of our introductory treatment of queuing theory, we make the following assumptions:

1. **Population:** We assume that Requests for service are generated from an infinite population. The significance of this assumption is that the arrival of a request to the system does not influence "future" arrivals. For example, the likelihood of a new customer walking into the bank in a given interval of time has nothing to do with other customers walking in and out of the bank.
2. **Queue Size:** For now we assume that queues have infinite capacity. The significance of this

(or is in) a "steady-state" at some time t, if computing an average performance metric for the system in one (long-enough) period of time after t will yield the same average when the same metric is computed at any other later (long-enough) period of time.

assumption is that it will never be the case (in our analysis, that is) that requests for service will be "lost" or will affect the likelihood of other requests joining the queue. For example, we exclude the likelihood that a patient will not be able to see a doctor because there was no waiting space for them in the doctor's office. A more computing-system-related example would be the arrival of IP packets to the buffer of a router. Under our assumption, packets can always be buffered and thus cannot be lost! Of course, any real buffer has a finite capacity. While it is possible (albeit more complex) to analyze queuing systems with finite queues, this assumption simplifies analysis greatly and provides an acceptable approximation.

5.2 Single-Queue, Single-Server System

We are interested in studying the performance of the system depicted in Figure 5.1 in the steady state.

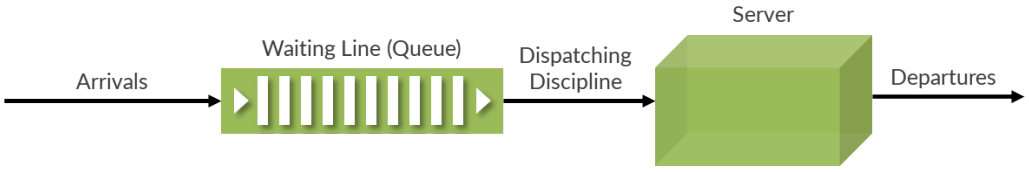

Figure 5.1: A single queue / single-server system.

5.2.1 Notation

The following notation will be used throughout the course to identify a number of important parameters and observed quantities (metrics).

- Let w denote the number of customers waiting in the queue;
- Let q denote the total number of customers in the system (waiting + being served);
- Let T_s denote the service time (the time it takes the server to serve a customer);
- Let T_w denote the waiting time in the queue, i.e. before starting service;
- Let T_q denote the turnaround time, or response time (waiting time + service time);
- Let ρ denote the utilization of the system, which is the ratio between the rate of arrivals and the rate of service $\rho = \lambda \cdot T_s = \frac{\lambda}{\mu}$.

In the steady state, the rate at which requests are queued cannot exceed the rate at which the server is able to serve them. Thus, it must hold that:

$$\lambda < \mu \implies \rho < 1 \tag{5.1}$$

The notation is summarized in Figure 9.1.

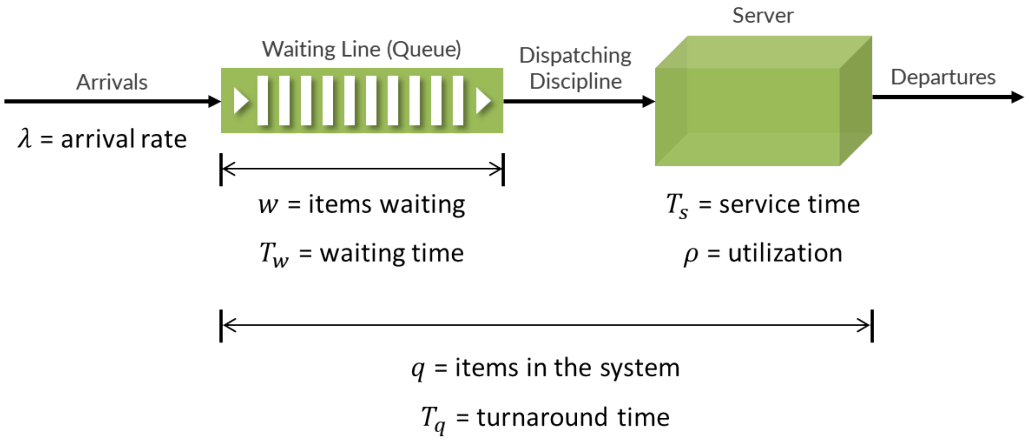

Figure 5.2: A single queue / single-server system with the corresponding notation.

5.2.2 Little's Law

The following two relationships are true of any "steady state" queuing system (i.e. a queuing system in equilibrium). They are known as **Little's Formulae** (or **Little's Theorem**, or **Little's Law**).

$$q = \lambda \cdot T_q \quad \text{and} \quad w = \lambda \cdot T_w \tag{5.2}$$

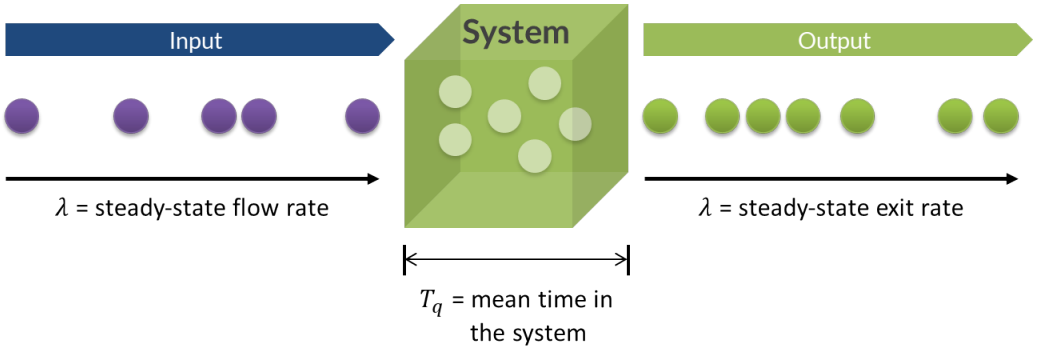

Figure 5.3: Illustration of Little's Law.

The intuition behind these two formulas is that over a period of time T, the number of arrival in the system is $\lambda \cdot T$. At equilibrium, every customer will wait (on average) an amount of time T_q in the system – see Figure 5.3. From the instant a customer arrives and until that customer leaves (i.e. after T_q units of time) $\lambda \cdot T_q$ "new" customers would have joined the system (on average). Thus at the time the customer leaves the system a total of $\lambda \cdot T_q$ customers would be in the system. Obviously, the same applies at the time any other customer leaves the system (and in general at any instant of time!).

5.2.3 Relationship between Turnaround, Queuing, and Service Times

In a queuing system, a customer's time is spent either waiting for service or getting service. Thus, we get the following additional (obvious) relationship:

$$T_q = T_w + T_s \tag{5.3}$$

Multiplying the above equation by the arrival rate λ and applying Little's formula, we get:

$$q = w + \lambda \cdot T_s = w + \frac{\lambda}{\mu} = w + \rho \tag{5.4}$$

Despite its seeming simplicity, Little's Law allows us to answer many questions related to a system's performance. The following are a few examples:

- If you are told that a request to a web server experiences an average response time of 0.5 seconds and that the average rate of arrivals for requests is 100 requests per second, then you can immediately figure out that (on average) the web server would be managing a total of $100 \cdot 0.5 = 50$ concurrent requests.
- If you are told that the multiprogramming level (number of processes in a system) is 30 on average and that the rate with which processes are created is 2 per minute, then you may conclude that the average "lifetime" of a process is 30/2 = 15 minutes.
- If you are told that the rate of arrival of packets to a router is 1000 per second and that it takes 0.5 msec to service each one of these packets and that the average number of packets queued at the router is 3, then you may conclude that the delay through this router will be 0.0005 + 3/1,000 = 3.5 msec.

Little's Law allows us to make some very general statements about the relationship between process arrival rates, number of processes in the system and turnaround time – knowing any two of these three variables, Little's Law allows us to figure out the third (under steady state). Clearly, we need to go beyond that. Specifically, we would like to figure out the turnaround time given some information about the process arrivals and the service times.

To be able to answer such questions, we need to *model* these characteristics. It is not enough to just talk about averages (such as average arrival rates). We can explain this with an example.

Assume that you are told that packets arrive to a router at the average rate of 100 packets per second. Many arrival processes may give you this average. For example, it could be that packets arrive precisely once every 10 msec, or it could be that they arrive in a batch of 10 every 100 msec, or they may arrive in some non-deterministic random fashion but somehow add up to 100 packets every second on average. Each one of these processes will result in a very different behavior at the queue. If all packets arrive in a burst, we expect the queue to build up quickly, and if they arrive in a well-space manner, then we don't expect too much oscillation in the queue size.

Thus it is clear that in order to analyze the queuing system introduced above, we will need to "model" how packets arrive to the system and also how they are served. For that we will rely on probabilistic modeling and analysis.