Category Overview

8/15/07

Evaluation

Answering the performance questions

In the early 1900s, telephone engineers were addressing the challenges of building a telephone network. At the time, the telephone network consisted of little more than wires connecting every subscriber to a local switching office. A switchboard operator plugged cords into jacks to make connections between pairs of subscribers. Telephone engineers were seeking to automate the switches so that calls would go through faster, there would be fewer errors, and the entire system could be scaled up to a large number of subscribers. Automatic switches consisted of stepping relays that moved one notch for each pulse from a rotary dial. There are N(N-1)/2 connections possible among N subscribers; a large number of stepping relays would be needed to allow them all. Therefore, the amount of switching hardware would be proportional to the square of the number of subscribers. A community of 100 subscribers, for example, would need enough switches to allow up to 4950 calls to be in progress simultaneously. That is a lot of equipment. For a town with 10,000 subscribers, the equipment needed to support all possible 50 million connections simultaneously would be staggering.

Telephone engineers knew from experience that only a small fraction of all the possible connections were in use at any given time. Is there a way to design the switch to support an actual number of connections just slightly larger than the number that would be busy anyway? In 1909 A K Erlang invented a mathematical model to predict the number of connections actually in use. His model supposed that the available equipment would support at most N active calls (connections). With the model he could calculate the probability p(N) that the next caller will find the switching center full (no dial tone). (In that case the caller has to come back later and try again.) Then it was just a matter of choosing N so that p(N) met an availability objective. For example, if the availability objective were 98%, and p(30) is 98%, a switch that can support only 30 calls would pose a 2% chance that a subscriber could pick up a phone and not get a dial tone. That is less than 1/165 of the amount of switching equipment needed to allow for 4950

simultaneous calls (100% availability). The "cost" to get the last 2% availability is 165 times larger than the "cost" to get the first 98%.

Erlang's model had just two parameters: the arrival rate and the call completion rate. He validated his model by gathering field data showing that the histograms of times between arrivals and of call lengths met the assumptions of his model and thus the model calculations could be trusted.

Erlang's model found great favor among telephone engineers. They found that for typical values of the arrival and completion rates the actual number of active calls was indeed quite small compared to the number of subscribers. This was a win-win because costs to subscribers were kept down and the switching offices could scale up well to large subscriber communities.

Erlang's basic model was a major discovery. He showed that accurate and useful statements could be made about a system whose key variables are random. The idea that we could accurately estimate waiting times if we know only the means of the variables was new. This idea is of major importance today. With its help, we can choose the capacities of various network servers so that the means of the key metrics and the network availability are all within targeted ranges. Without it, we would be unable to design usable networks and computing systems.

Resource Sharing

Erlang's model illustrates another important principle: sharing resources is always more efficient than partitioning them among users. The equipment needed in a switching center can be drastically reduced if all callers share switch instead of having their own dedicated, mostly dormant potential connections.

This principle manifests in many settings. Inside a computer, the multitasking protocol loads multiple programs into the single RAM. The amount of RAM needed is dramatically reduced if RAM is allocated dynamically among the tasks to guarantee their working sets, rather than when each task is allocated all the RAM it will ever need.

Inside a digital communication network, signals are encoded into packets, which are sent to the receiver as they are generated. Because the bandwidth of the channel is much higher than the rate of new packet generation, it is much more efficient to share the channel among many generators than to give each generator its own circuit to its recipient.

We can use statistical methods such as Erlang's model to quantify the amount of resource saved by sharing.

Going Statistical

Designers of software and computing systems must inevitably face two fundamental questions:

Correctness: Does the software or system work?

Performance: Does the software or system produce its results soon enough to be useful?

Because computing systems cannot function without software, it is tempting to conclude that the correctness question is the more fundamental. In fact, the performance question is equally fundamental. Correct software that takes too long is of no value.

Programmers and algorithms analysts are no strangers to performance issues. Computational complexity dominates performance of algorithms on ideal computers -- flat memories and fast CPUs. We classify algorithms by the order of the growth of their computation time with size of input -- for example, linear-time algorithms, quadratic-time algorithms, or exponential-time algorithms. Knowledge that an algorithm is in one of these categories only allows us to rank the time of the algorithm relative to others of the same or different orders. A more careful analysis is needed before we can say what the running time will be for particular input data.

In reality, software is never a single algorithm run alone on a single computer. Software is almost always a mixture of algorithms of different running times executed on machines shared with other users. User processes (also called "jobs") compete for limited CPU, memory, disk, and networking resources. Contention for these resources adds delay above and beyond what the software would require if it were able to operate alone on the machine. Performance, therefore, is almost always formulated in a systems context, where the analyst must provide answers such as:

- What is the throughput (jobs per second completed)?
- What is the response time to a user request?
- Where are the bottlenecks?
- How much capacity is needed to meet throughput and response time objectives?

The evaluation category includes the basic principles with which we answer these questions about our computing systems and networks, just as Erlang did for telephone systems a century before.

Objects of Study

In many parts of computing, the algorithm is the object of study. We have learned to answer a large number of questions about algorithms, their expression and their complexity, and their translatability from one language to another. Our tools sets for algorithms analysis include discrete mathematics, logic, formal language theory, and computability theory.

When we put real software on real computing systems, competing with other software for limited resources, we must switch to computations as objects of study. Even the apparently simple throughput question depends on knowledge of the demands of every job for CPU, disks, and other servers. Those demands vary from one job to another and even one phase to another of a single job. A job places varying demands on resources and will be delayed when needed resources are not available. We characterize the varying demands with probabilities of the different values that will be observed. Our tool sets for analyzing systems of computations include probability, statistics, queueing theory, some discrete mathematics, and some calculus.

The computing field thus studies both kinds of objects in detail: the static algorithm and the dynamic computation. Evaluation is concerned with the performance questions about computations.

Principal Methods

Answers to performance questions depend on the random interactions of many computations using the same servers and networks, interacting in unpredictable ways. The fundamental quantities such as numbers of arrivals, numbers of service completions, service times, response times, or queue lengths, are all random variables. Modeling, simulation, experiment, and statistical analysis are our sources of answers to performance questions.

A model is a simplified description of a system that enables calculation of performance quantities. Three kinds of model for performance questions are benchmarking, simulation, and analytic. A benchmark is a suite of programs representing a typical workload; by running the benchmark on the system and measuring the performance quantities, we can get a pretty good idea which system configurations will work

best. A simulation is a model that imitates the real system by tracing system states that result from jobs requesting, obtaining, and releasing resources; performance quantities are measured by counting events and adding up the holding times of states entered by those events. An analytic model sets up equations whose solutions are the performance quantities of the system.

Benchmarks are used for quick comparisons of hardware and software configurations. Simulations are used when the system is so complex that no benchmark is reliable and no analytic model is feasible. Analytic models are the most common.

Analytic Performance Models

We use analytic models in two basic modes, *calculation* and *prediction*. Calculation means that the model is used to calculate metrics when given the values of parameters. For example, when we are told a server's throughput and mean service time, we can exactly calculate the utilization (fraction of time the server is busy). Calculation minimizes the number of quantities that need to be measured, since everything else can be calculated from them. A model is considered to be of good quality if the calculated metrics are consistently close to the actual values in the real system. Model validation consists in running many experiments to demonstrate model quality.

Prediction means that the model is used to predict performance metrics in the future. We are interested in what happens when we change some aspects of the system -- for example, we add more users, we install a larger disk, or we add more CPUs. Predictions proceed in three steps:

- We measure the model parameters in a base observation period.
- We project the values of parameters into the future period.
- We calculate the performance metrics for the future period using the projected parameters and the model.

There are two sources of error: the quality of our projections of parameters, and the quality of the model itself. If we are working with a validated model, the main source of error is the parameter projection. Typically, we project parameters by retaining the values of those that are not changed and modifying the values of those that are.

Network of Servers Model

We can visualize Erlang's model by imagining that customers walk up to the switching office and join a waiting line. Their arrivals correspond to initiating calls and their departures to completing (hanging up). The number in the line is the number of calls actually in progress; it is a random variable that grows and shrinks with arrivals and departures. An availability objective of 98% would correspond to a waiting area of sufficient capacity to hold all waiting customers 98% of the time.

Today's networks involve many service centers, instead of just one in the Erlang model. But the visualization is much the same. Imagine a flea market with many booths. Each customer comes with a list of things to look for and visits the booths strictly in the listed order one at a time. As customers circulate in the system, each booth will develop its own waiting line. Some booths will be very popular and have long lines. The most popular booth of all will be called the bottleneck. Today's network-of-server models extend Erlang's to compute the joint probabilities of the simultaneous line lengths at all the servers.

The network-of-servers model deals with any system configured as a set of service centers, wherein a job consists of a list of visits to the centers. There are three main parts of the model:

- **Workload:** a job consists of a sequence of requests for various kinds of service -- for example, CPU time-slices, reading disk sectors, or printing files. In the model, a job makes a random choice of next server rather than following a predetermined list. A full workload consists of a set of jobs. Requests of the same job cannot be executed in parallel; requests of different jobs can. The size of the workload (number of jobs) is denoted N.
- **Servers:** individual devices respond to specific kinds of requests: for example, disks respond to requests to read or write sectors, CPUs to requests to provide time-slices of execution, printers to requests to print files. A server includes a queueing mechanism that stacks requests and an internal service mechanism picks up the requests from the queue. A "single-server" can serve only one request at a time. A "multi-server" can serve several requests in parallel. Servers are numbered i = 1,2,...,K. The mean service time per request for server i is denoted Si; for example, a typical disk service time is Si = 5 msec.
- **Topology:** the set of paths by which jobs travel between servers. When a job finishes a request at one server, it places its next

request at another server, thereby making a transition from the one server to the other. The effect of the allowable paths is that jobs visit some servers multiple times and some more than others. The visit ratio Vi is the number of times a job visits server i.

The parameter set for this basic model is: N, the total number of jobs in the system; {Si}, the mean service times of requests for server i; and {Vi}, the mean numbers of visits (requests made) per job to server i. As with Erlang's original model, it is possible with just these parameters to estimate performance metrics such as throughput, response time, and queue lengths.

To simplify the mathematics of his model, Erlang assumed that the histograms of inter-arrival times and call lengths were both exponential. In an exponential distribution, the probability that the arrival time will exceed T is e^{-aT}, where 1/a is the mean arrival time. Erlang confirmed from data on telephone calls that the arrival and call length distributions were both exponential. Thus his model validated perfectly and gave accurate answers.

For most networks of servers, most of the service time distributions are not exponential. Therefore the model calculations are only approximate and the model must be validated carefully. In practice, the models have validated well: they typically predict throughput within 5% and response times within 25%.

It took nearly 50 years after Erlang proposed his model (1909) until queueing theorists successfully generalized it to the network of servers. Unfortunately the extended model languished for many years because the only known algorithms for computing its metrics were highly intractable (2^{N+K}, (N+K)!, or worse) -- computing a throughput for a typical system with 100 jobs and 10 servers would take many times the age of the universe.

In 1971, in a major breakthrough, Jeffrey Buzen discovered an algorithm that computed all performance metrics in time proportional to N or N^2 for a fixed number of servers. Buzen's algorithm made it possible to solve large networks on a hand calculator or spreadsheet. Thereafter the network-of-servers model became the exceedingly popular gold standard for performance prediction and capacity planning.

The models and their computational algorithms can be generalized from a single workload (all jobs have the same basic parameters Si and Vi) to multiple workloads (each job class has its own set of

parameters Si and Vi). In this case Buzen's algorithm slows down slightly to order N^2.

Fundamental Laws

All networks of servers (not just the model, but the real systems) obey a set of fundamental laws. These are relationships among performance quantities that hold in every system and every observation period. The three most fundamental are:

Utilization Law says that a server's utilization (fraction of time in use) is the product of its throughput and mean service time per request. (Ui = Xi*Si)

Little's Law says that the mean number in a server is the product of its throughput and mean response time per request. (Qi=Xi*Ri)

Forced Flow Law says that the throughput at any server is the product of the system throughput and the visit ratio for that server. (Xi = Vi*X)

To illustrate these laws, consider again the flea-market example. The fresh produce booth is serving 1 customer every 2 minutes and each customer transaction takes 1 minute; therefore the utilization of that booth is (0.5 customer/min)*(1 min) or 50%. That means that 50% of the time someone is there being served, and the other 50% the booth is idle.

Suppose that we find that produce customers spent an average of 6 minutes from when they joined the line until when they completed their transactions. The average number in the line is (0.5 customers/min)*(6 min) or 3 customers. Note that the line length varies, and must be zero half the time according to our previous calculation.

Suppose that that customers visit the produce booth twice because (perhaps they chronically forget a vegetable). But they only visit the fruit booth once. The throughput at the fruit booth must be half that of the produce booth, or 0.25 customers per minute.

These and other fundamental laws allow us to deduce many other properties of the system, such as bottlenecks (next section), and help us discover errors in measurement procedures.

Bottlenecks

A bottleneck is a choke point in the system: a server that is so slow relative to others that it constrains the processing rate of the entire system.

By combining the three fundamental laws, we find that the system throughput must be Ui/(Vi*Si) for any server i. The quantity Vi*Si is the total service required at server i. The server with the largest of these quantities must be the bottleneck because it is accumulating more delay than any other server. If the load on the system becomes high enough, the utilization of that server will saturate at 100% and limit the system throughput to 1/Vi*Si.

In the flea market, we find that the produce server has the largest total demand: (2 visits)*(1 min/visit) or 2 mins. Therefore if enough people come to the market, the produce server will saturate with throughput 1 customer per minute. That will limit the market's throughput to 0.5 customer per minute since customers visit the produce server twice on average. The bottleneck also affects the fruit server, where the customer transaction time also averages 1 minute. In this case, the throughput at the fruit server will be only 0.5 customer per minute, even though the server's capacity is 1 per minute, because most people are in the waiting line at the saturated produce server.

Capacity Planning

A common design problem is to specify the numbers of servers and their service times so that given throughput and response time targets can be met for the given workload.

In the flea market, the produce server is a bottleneck that saturates with throughput 1 customer per minute, and limits the whole market's throughput to 0.5 customer per minute. No one is satisfied with this, least of all the produce owner. Suppose the owner agrees to hire an additional clerk; now he has 2. That means that the server can now complete a customer on average every 0.5 min, doubling the saturation throughput to 2 customers per min. If the faster produce server is still the bottleneck, the market's total throughput will double to 1 customer per minute.

However, the faster produce server may no longer be the bottleneck. This might happen if there is a florist, who is visited once for 90 seconds per transaction, giving it a total demand of (1 visit)*(1.5 mins/visit) or 1.5 mins. After the produce server hires the second

clerk, its total demand dropped to 1 min, changing the bottleneck to the florist. Now the system through is limited to 0.67 customers per min.

These examples show that capacity planning is not simply a matter of choosing individual capacities of the servers, it is also involved the configuration of (potential) bottlenecks. The best capacity planning results when the full network-of-servers model is used to calculate throughput and response time as a function of all the servers.