Chapter 7
Operational Analysis and Basic Queueing Models

"All models are wrong, but some are useful."
—*George E. P. Box (1919-2013), British statistician*

In Chapter 1, we introduced the concept of system performance understood in a classical sense as the amount of useful work accomplished by a system compared to the time and resources used. Better performance normally means more work accomplished in shorter time or using less resources. To characterize the performance behavior of a system, performance metrics are used. In Chapter 3 (Section 3.3), we introduced the most common basic performance metrics used in practice: response time, throughput, and utilization.

In this chapter, we start by looking at some basic quantitative relationships, which can be used to evaluate a system's performance based on measured or known data, a process known as operational analysis (Section 7.1). Operational analysis can be seen as part of queueing theory, a discipline of stochastic theory and operations research, which provides general methods to analyze the queueing behavior of one or more service stations. In the second part of the chapter (Section 7.2), we provide a brief introduction to the basic notation and principles of queueing theory. While queueing theory has been applied successfully to different domains, for example, to model manufacturing lines or call center operation, in this chapter, we focus on using queueing theory for performance evaluation of computer systems. Nevertheless, the presented concepts and mathematical models are relevant for any processing system where the generic assumptions described in this chapter are fulfilled. The chapter is wrapped up with a case study, showing in a step-by-step fashion how to build a queueing model of a distributed software system and use it to predict the performance of the system for different workload and configuration scenarios.

S. Kounev et al., *Systems Benchmarking*, https://doi.org/10.1007/978-3-030-41705-5_7

7.1 Operational Analysis

In this section, we introduce a set of basic quantitative relationships between the most common performance metrics. These relationships are commonly known as *operational laws* and can be considered to be consistency requirements (i.e., invariant relations) for the values of performance quantities measured in any particular experiment (Menascé et al., 2004). The process of applying operational laws to derive performance metrics based on measured or known data is known as *operational analysis* (Denning and Buzen, 1978). This section introduces the most important operational laws, which are later revisited in Section 7.2 in the context of queueing theory. We refer the reader to Menascé et al. (2004) for a more detailed treatment of operational analysis.

Consider a system made up of K resources (e.g., servers, processors, storage devices, network links). The system processes requests sent by clients.[1] It is assumed that during the processing of a request, multiple resources can be used, and at each point in time, the request is either being served at a resource or is waiting for a resource to become available. The same resource may be used multiple times during the processing of a request; each time the resource is used, we will refer to this as the request *visiting* the resource. We assume that the system is observed for a finite period of time and that it is in *operational equilibrium* (i.e., steady state) during this period; that is, the number of submitted requests is equal to the number of completed requests.

We will use the notation shown in Table 7.1. Given that the system is assumed to be in operational equilibrium, the following obvious equations hold:

$$
\begin{aligned}
& S_i = \frac{B_i}{C_i}, \quad U_i = \frac{B_i}{T}, \\
& X_i = \frac{C_i}{T}, \quad \lambda_i = \frac{A_i}{T}, \\
& X_0 = \frac{C_0}{T}, \quad V_i = \frac{C_i}{C_0}, \\
& \lambda_i = X_i.
\end{aligned}
\tag{7.1}
$$

Example During a period of 1 min, 240 requests arrive at a server and 240 requests are completed. The server's CPU is busy for 36 s in this time period. If the server

[1] The term *request* here is used loosely to refer to any unit of work or processing task executed in the system, for example, an HTTP request, a database transaction, a batch job, a web service request, or a microservice invocation. Requests in this context are also commonly referred to as *customers*, *jobs*, or *transactions*.

Table 7.1: Notation used in operational analysis (Menascé et al., 2004)

Symbol	Meaning
K	Number of resources in the system
T	Length of time during which the system is observed
B_i	Total length of time during which resource i is busy in the observation period
A_i	Total number of service requests (i.e., arrivals) to resource i
A_0	Total number of requests submitted to the system
C_i	Total number of service completions (i.e., departures) at resource i
C_0	Total number of requests processed by the system
V_i	Average number of times resource i is visited (i.e., used) during the processing of a request, referred to as *visit ratio*
λ_i	Arrival rate at resource i (i.e., average number of service requests that arrive per unit of time)
S_i	Average *service time* of a request at resource i per visit to the resource (i.e., the average time the request spends receiving service from the resource excluding waiting/queueing time)
D_i	Average total service time of a request at resource i over all visits to the resource, referred to as the *service demand* at resource i
U_i	Utilization of resource i (i.e., the fraction of time the resource is busy serving requests)
X_i	Throughput of resource i (i.e., the number of service completions per unit of time)
X_0	System throughput (i.e., the number of requests processed per unit of time)
R	Average request response time (i.e., the average time it takes to process a request including both the waiting and service time in the system)
N	Average number of active requests in the system, either waiting for service or being served

uses no resources apart from the CPU, and it only has a single request class, what is the arrival rate, the CPU utilization, the mean CPU service demand, and the system throughput?

$$\begin{gathered}
K = 1, \quad T = 60\,\text{s}, \quad A_0 = A_1 = 240, \quad C_0 = C_1 = 240, \quad B_1 = 36\,\text{s}, \\
\lambda_1 = \frac{A_1}{T} = \frac{240}{60\,\text{s}} = 4\,\text{req/s}, \quad U_1 = \frac{B_1}{T} = \frac{36\,\text{s}}{60\,\text{s}} = 0.6 = 60\%, \\
S_1 = \frac{B_1}{C_1} = \frac{36\,\text{s}}{240} = 0.15\,\text{s}, \quad X_0 = \frac{C_0}{T} = \frac{240}{60\,\text{s}} = 4\,\text{req/s}.
\end{gathered} \tag{7.2}$$

In the following, we introduce the five most common operational laws providing the basis for operational analysis.

7.1.1 Utilization Law

The utilization law states that the utilization of resource i is given by the request arrival rate λ_i multiplied by the average service time S_i per visit to the resource; that is,

$$U_i = S_i \times \lambda_i = S_i \times X_i. \tag{7.3}$$

Proof

$$U_i = \frac{B_i}{T} = \frac{\frac{B_i}{C_i}}{\frac{T}{C_i}} = \frac{\frac{B_i}{C_i}}{\frac{1}{\frac{C_i}{T}}} = \frac{S_i}{\frac{1}{X_i}} = S_i \times X_i = S_i \times \lambda_i. \tag{7.4}$$

Example A program computes 190 matrix multiplications per second. If each matrix multiplication requires 1.62 billions of floating point operations, and the underlying CPU can process up to 380 GFLOPS (billions of floating point operations per second), what is the utilization of the CPU?

$$K = 1, \quad X_1 = 190,$$

$$S_1 = \frac{1.62}{360} = 0.0045\,\text{s}, \tag{7.5}$$

$$U_1 = S_1 \times X_1 = 0.0045 \times 190 = 0.855 = 85.5\%.$$

7.1.2 Service Demand Law

The *service demand* D_i (also referred to as *resource demand*) is defined as the average total service time of a request at resource i over all visits to the resource.[2] The service demand law states that the service demand of a request at resource i is given by the utilization of resource i divided by the system throughput X_0, that is,

$$D_i = \frac{U_i}{X_0}. \tag{7.6}$$

Proof

$$D_i = V_i \times S_i = \frac{C_i}{C_0} \times \frac{B_i}{C_i} = \frac{B_i}{C_0} = \frac{U_i \times T}{C_0} = \frac{U_i}{\frac{C_0}{T}} = \frac{U_i}{X_0}. \tag{7.7}$$

[2] In this book, we use the terms service demand and resource demand interchangeably.

Example A program that calculates matrix multiplications is run 180 times within 5 min. For this time period, the underlying CPU reports a utilization of 30%. What is the CPU service demand for a single program execution?

$$
\begin{aligned}
&K = 1, \quad T = 5 \times 60\,\text{s} = 300\,\text{s}, \\
&X_0 = \frac{C_0}{T} = \frac{180}{300\,\text{s}} = 0.6\,\text{runs/s}, \\
&U_1 = 30\% = 0.3, \\
&D_1 = \frac{U_1}{X_0} = \frac{0.3}{0.6} = 0.5\,\text{s}.
\end{aligned}
\tag{7.8}
$$

7.1.3 Forced Flow Law

By definition of the visit ratio V_i, resource i is visited (i.e., used) V_i times, on average, by each processed request. Therefore, if X_0 requests are processed per unit of time, resource i will be visited $V_i \times X_0$ times per unit of time. So the throughput of resource i, X_i, is given by

$$X_i = V_i \times X_0. \tag{7.9}$$

This result, known as forced flow law, allows one to compute the system throughput based on knowledge of the visit ratio and the throughput of any one resource in the system. In addition, knowing the visit ratios of all resources and the throughput of just one resource allows for calculation of the throughput of all other resources in the system.

Example A REST-based web service[3] accesses a file server five times and a database two times for every request. If the web service processes 525 requests in a 7 min interval, what is the average throughput of the web service, the file server, and the database?

[3] REST (REpresentational State Transfer) is an architectural style for developing web services, which is typically used to build lightweight web and mobile applications. Web services that conform to the REST architectural style provide interoperability between computer systems on the Internet. Nowadays, most public web services provide REST APIs (Application Programming Interfaces) and transfer data in a compact and easy-to-use data-interchange format—the JavaScript Object Notation (JSON).

$$X_{web\_service} = X_0 = \frac{C_0}{T} = \frac{525}{7 \times 60\,\text{s}} = 1.25\,\text{req/s},$$

$$X_{file\_server} = V_{file\_server} \times X_0 = 5 \times 1.25 = 6.25\,\text{req/s}, \tag{7.10}$$

$$X_{database} = V_{database} \times X_0 = 2 \times 1.25 = 2.5\,\text{req/s}.$$

7.1.4 Little's Law

Little's law states that the average number of active requests N in the system (submitted requests whose processing has not been completed yet) is equal to the average time it takes to process a request (i.e., the request response time R) multiplied by the number of requests processed per unit of time (i.e., the system throughput X_0), that is,

$$N = R \times X_0. \tag{7.11}$$

We consider Little's law in the context of a system processing requests; however, it can generally be applied to any "black box" where some entities arrive, spend some time inside the black box, and then leave. Little's law states that the average number of entities in the black box N is equal to the average residence time R of an entity in the black box multiplied by the average departure rate X (throughput); that is, $N = R \times X$. This is illustrated in Figure 7.1. We refer to Little (1961) for a formal proof. Little's law holds under very general conditions; the only assumption is that the black box does not create nor destroy entities.

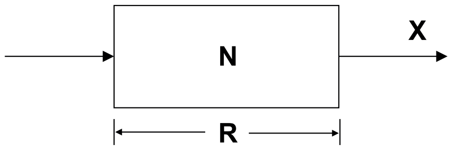


Fig. 7.1: Little's law (Menascé et al., 2004)

Example An enterprise resource planning system is implemented based on a microservice architecture consisting of many individual microservices. What is the average response time of the enterprise resource planning system if it is used by 273 employees at the same time and they execute 9,450 operations per hour?

$$T = 60\,\text{min} \times 60\,\text{s} = 3{,}600\,\text{s},$$

$$X_0 = \frac{C_0}{T} = \frac{9{,}450}{3{,}600\,\text{s}} = 2.625\,\text{ops/s}, \tag{7.12}$$

$$R = \frac{N}{X_0} = \frac{273}{2.625} = 104\,\text{s}.$$

7.1.5 Interactive Response Time Law

Assume that the system we consider is used by M clients each sitting at their own workstation and interactively accessing the system. This is an example of a *closed workload* scenario (see Chapter 8, Section 8.3.2). Clients send requests that are processed by the system. It is assumed that after a request is processed by the system, the respective client waits some time before sending the next request. We refer to this waiting time as "think time." Thus, clients alternate between "thinking" and waiting for a response from the system. If the average think time is denoted by Z, the interactive response time law (illustrated in Figure 7.2) states that the average response time R is given by

$$R = \frac{M}{X_0} - Z. \tag{7.13}$$

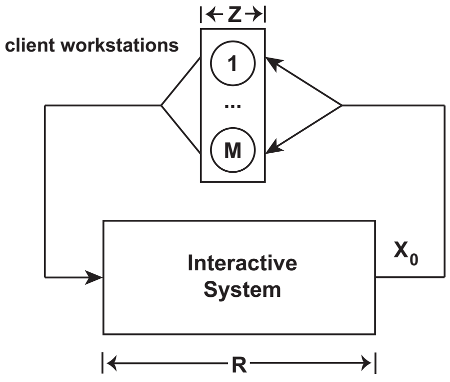


Fig. 7.2: Interactive response time law (Menascé et al., 2004)

Proof To show that the interactive response time law holds, we apply Little's law to the virtual black box composed of the client workstations and the system considered as a whole. We now consider the think time spent at the client workstation before sending a new request as being part of the respective request (e.g., preparation phase). Thus, each time the processing of a request is completed by the system and a response is sent back to the client, we consider this as one entity leaving our virtual black box and at the same time one new entity arriving at the virtual black box (corresponding to the next request). The total number of entities in our virtual black box is equal to the total number of clients M (at each point in time, each request is either at the respective client workstation or it is being processed inside the system). The rate at which requests are completed by the system is given by the system throughput X_0. The total average time a request spends in the virtual black box (i.e., client workstation plus system) is given by $Z + R$. Applying Little's law to the virtual black box, we obtain the following equation, which is equivalent to the interactive response time law:

$$M = X_0(Z + R). \tag{7.14}$$

Example A train booking and reservation system is used by 50 employees. Each of them, on average, issues a request 5 s after receiving the result of the previous request. A request has an average CPU service demand of 0.1 s; a CPU utilization of 32% is observed. How long do employees have to wait on average until a request is completed?

$$\begin{aligned} X_0 = X_{CPU} = \frac{U_{CPU}}{D_{CPU}} = \frac{0.32}{0.1\,\text{s}} = 3.2\,\text{req/s}, \\ R = \frac{M}{X_0} - Z = \frac{50}{3.2} - 5\,\text{s} = 10.6\,\text{s}. \end{aligned} \tag{7.15}$$

In summary, we introduced the following five operational laws:

Utilization law:

$$U_i = S_i \times X_i \tag{7.16}$$

Service demand law:

$$D_i = V_i \times S_i = \frac{U_i}{X_0} \tag{7.17}$$

Forced flow law:

$$X_i = V_i \times X_0 \tag{7.18}$$

Little's law:

$$N = R \times X_0 \tag{7.19}$$

Interactive response time law:

$$R = \frac{M}{X_0} - Z \tag{7.20}$$

7.1.6 Multi-Class Versions of Operational Laws

The operational laws can be extended to the case where multiple types of requests are processed by the system. The measured quantities and derived metrics are then considered on a per *request class* basis. An index c is used to distinguish between the respective classes. The following multi-class versions of the operational laws hold (Menascé et al., 2004):

Utilization law:

$$U_{i,c} = S_{i,c} \times X_{i,c} \tag{7.21}$$

Service demand law:

$$D_{i,c} = V_{i,c} \times S_{i,c} = \frac{U_{i,c}}{X_{0,c}} \tag{7.22}$$

Forced flow law:

$$X_{i,c} = V_{i,c} \times X_{0,c} \tag{7.23}$$

Little's law:

$$N_c = R_c \times X_{0,c} \tag{7.24}$$

Interactive response time law:

$$R_c = \frac{M_c}{X_{0,c}} - Z_c \tag{7.25}$$

Most of the quantities in the multi-class versions of the operational laws can normally be easily measured by means of standard system monitoring tools based on the measurement techniques presented in Chapter 6. The only exception is for the utilization $U_{i,c}$ and the service time $S_{i,c}$. For example, monitoring tools can normally provide measurements of the total resource utilization U_i. However, partitioning the total utilization on a per request class basis is not trivial. While performance profiling tools can be used for this purpose (see Section 6.3 in Chapter 6), such tools normally incur instrumentation overhead, which might lead to perturbation impacting the system behavior. Also, suitable profiling tools may not be available for the specific environment. The utilization, broken down on a per request class basis (i.e., $U_{i,c}$), is mainly relevant for obtaining the service demands $D_{i,c}$. In Chapter 17, we look at techniques for estimating service demands (also referred to as resource demands) based on easy to measure high-level metrics.

7.1.7 Performance Bounds

Now that we introduced the basic operational laws, we present some further quantitative relationships between the most common performance metrics, which provide upper and lower bounds on the performance a system can achieve. The bounds can be classified into optimistic and pessimistic bounds and are typically used for bottleneck analysis. The term *bottleneck* is normally used to refer to the resource with the highest utilization. It is assumed that this resource will first be saturated as the load increases. If a bottleneck cannot be removed (e.g., by increasing the capacity of the respective resource), the system is considered non-scalable in terms of performance. In the following two subsections, we present two sets of performance bounds on the system throughput and response time. Optimistic bounds capture the largest possible throughput (X_{opt}) and the lowest possible response time (R_{opt}), while pessimistic bounds capture the lowest possible throughput (X_{pes}) and largest possible response time (R_{pes}):

$$X_{pes} \leq X \leq X_{opt}, \qquad R_{opt} \leq R \leq R_{pes}. \tag{7.26}$$

The optimistic bounds can be derived from the service demands (Menascé et al., 2004). We assume that the service demands are load-independent,[4] which is normally implicitly assumed in the context of operational analysis. The bounding behavior of a system is determined by its bottleneck resource, which is the resource with the largest service demand. Applying the service demand law, we obtain the following upper asymptotic bound on the throughput:

$$X_0 = \frac{U_i}{D_i} \leq \frac{1}{D_i} \leq \frac{1}{\max_{i=1..K}\{D_i\}}. \tag{7.27}$$

Given that a natural lower bound for the response time R is given by the sum of the service demands at all resources, applying Little's law, we obtain another upper asymptotic bound on the throughput:

$$N = R \times X_0 \geq \left(\sum_{i=1}^{K} D_i\right) \times X_0 \Leftrightarrow X_0 \leq \frac{N}{\sum_{i=1}^{K} D_i}. \tag{7.28}$$

In summary, the upper asymptotic bounds on the throughput are given by

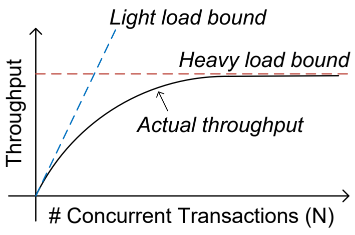
$$X_0 \leq \min\left[\frac{1}{\max\{D_i\}}, \frac{N}{\sum_{i=1}^{K} D_i}\right]. \tag{7.29}$$

From Little's law and the upper asymptotic bounds, we obtain the following lower asymptotic bounds on the response time:

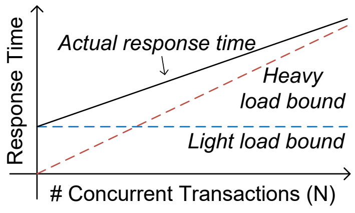
$$R = \frac{N}{X_0} \geq \frac{N}{\min\left[\frac{1}{\max\{D_i\}}, \frac{N}{\sum_{i=1}^{K} D_i}\right]} = \max\left[N \times \max\{D_i\}, \sum_{i=1}^{K} D_i\right], \tag{7.30}$$

$$R \geq \max\left[N \times \max\{D_i\}, \sum_{i=1}^{K} D_i\right]. \tag{7.31}$$

[4] A service demand is *load-independent* if it does not change as the request arrival rates and the induced utilization of system resources increase or decrease.



(a) Upper bounds on throughput



(b) Lower bounds on response time

Fig. 7.3: Asymptotic bounds

The asymptotic bounds on throughput and response time are illustrated in Figure 7.3. In addition to the asymptotic bounds, which are normally quite loose, based on a technique known as *balanced job bounds analysis*, the following tighter bounds can be derived (Menascé et al., 1994):

$$\frac{N}{\max\{D_i\}[K + N - 1]} \leq X_0 \leq \frac{N}{\text{avg}\{D_i\}[K + N - 1]}. \qquad (7.32)$$

Figure 7.4 illustrates the relationship between the asymptotic bounds and balanced job bounds.

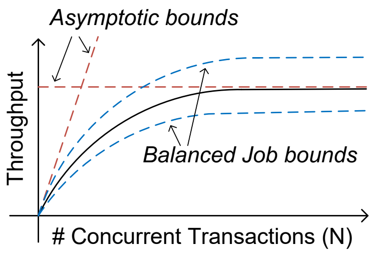


Fig. 7.4: Balanced job bounds (Menascé et al., 1994)

Example Fifty employees use an enterprise resource planning system that is implemented as a three-tier architecture. The web tier has a CPU service demand of 0.2 s, the business logic tier has a CPU service demand of 0.32 s, and the database tier has a CPU service demand of 0.15 s. Calculate the asymptotic and balanced job bounds for the throughput of the enterprise resource planning system under the assumption that all tiers use no resources apart from their CPU.

Asymptotic bounds:

$$X_0 \le \min\left[\frac{1}{\max\{D_i\}}, \frac{N}{\sum_{i=1}^{K} D_i}\right],$$

$$X_0 \le \min\left[\frac{1}{\max\{0.2, 0.32, 0.15\}}, \frac{50}{0.2 + 0.32 + 0.15}\right], \tag{7.33}$$

$$X_0 \le \min\,[3.1, 74.6] = 3.1.$$

Balanced job bounds:

$$\frac{N}{\max\{D_i\}[K + N - 1]} \le X_0 \le \frac{N}{\operatorname{avg}\{D_i\}[K + N - 1]},$$

$$\frac{50}{\max\{0.2, 0.32, 0.15\}[3 + 50 - 1]} \le X_0 \le \frac{50}{\operatorname{avg}\{0.2, 0.32, 0.15\}[3 + 50 - 1]}, \tag{7.34}$$

$$3.0 \le X_0 \le 4.3.$$

7.2 Basic Queueing Theory

The fundamental operational laws presented in the previous section can be seen as part of *queueing theory*, a discipline of stochastic theory and operations research. It provides general methods to analyze the queueing behavior at one or more service stations and has been successfully applied to different domains in the last decades, for example, to model manufacturing lines or call center operations. When analyzing the performance of a computer system, queueing models are often used to describe the scheduling behavior at hardware resources such as processors, storage, and network devices. In this section, we provide a brief introduction to the basic notation and principles of queueing theory. A detailed treatment of the subject can be found in Lazowska et al. (1984), Bolch, Greiner, et al. (2006), and Harchol-Balter (2013).

7.2.1 Single Queues

The central concept of queueing theory is a *queue*, also referred to as a *service station* or *service center*. A queue (illustrated in Figure 7.5) consists of a waiting line and a server, which serves incoming *requests*.[5] Requests arrive at the queue and are processed immediately unless the server is already occupied. In the latter case, requests are put into the waiting line. After a request has been completely processed by the server, it departs from the queue. A queue can also have several servers, assumed to be identical, in which case we speak of a *multi-server queue*. The semantics are similar; that is, whenever a request arrives, it is processed at one of the servers that is currently free. If all servers are occupied, the request is put into the waiting line.

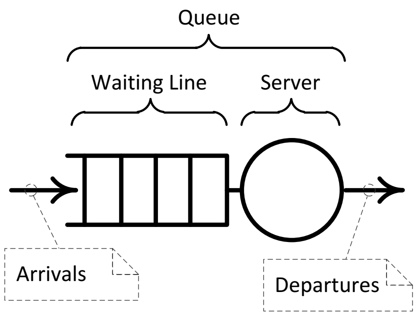


Fig. 7.5: Single queue (service station)

A number of terms are commonly used when describing the timing behavior of a queue. Requests may arrive at a queue at arbitrary points in time. The duration between successive request arrivals is referred to as *inter-arrival time*. The average number of requests that arrive per unit of time is referred to as *arrival rate*, denoted as λ. Each request requires a certain amount of processing at a server. The time a server is occupied by a request is called *service time*. The average number of requests that can be processed per unit of time at a single server is referred to as *service rate*, denoted as μ. The *mean service time* is then defined as $S = 1/\mu$ and specifies the time a server is occupied while processing a request on average. The time a request spends waiting in the waiting line is referred to as *queueing delay* or simply *waiting time*. The *response time* of a request is the total amount of time the request spends at the queue, that is, the sum of waiting time and service time.

When one request is completed, the next request to be served is selected from the requests in the queue according to a *scheduling strategy*. Typical scheduling strate-

[5] The term *request* here is used in the same way as in Section 7.1, that is, it refers loosely to any unit of work or processing task executed in the service station. Requests in this context are also commonly referred to as *customers*, *jobs*, or *transactions*.

gies are First-Come-First-Served (FCFS), where jobs are processed in the order of their arrival, Processor-Sharing (PS), where jobs are served concurrently each having an equal share of the total capacity (i.e., round-robin scheduling with infinitesimally small time slices), or Infinite-Server (IS), where all requests in the queue are scheduled immediately as if the queue were to have an infinite number of servers. When modeling computer systems, a FCFS scheduling strategy is typically used for queues representing I/O devices, whereas a PS scheduling strategy is commonly used for queues representing processors (CPUs) and an IS scheduling strategy for queues representing constant delays (e.g., average network delays).

There is a standard notation to describe a queue, known as *Kendall's notation* (Kendall, 1953). A queue is described by means of 6 parameters $A/S/m/B/K/SD$ defined in Table 7.2. The distribution components are characterized using short-hand symbols for the type of distribution, the most common of which are shown in Table 7.3. A deterministic distribution means that the respective times are constant. A general distribution means that the distribution is not known. This is commonly used for empirical distributions obtained from measurements if the underlying shape of the distribution is unknown. Parameters B and K are usually considered infinite and are thus often omitted in queue descriptions.

Table 7.2: Kendall's notation for a queue (A/S/m/B/K/SD)

Symbol	Meaning
A	Arrival process (i.e., distribution of the inter-arrival times)
S	Service process (i.e., distribution of the service times)
m	Number of servers in the service station
B	Maximum number of requests that a queue can hold (if missing, B is assumed to be infinite)
K	Maximum number of requests that can arrive at the queue, referred to as *system population* (if missing, K is assumed to be infinite)
SD	Scheduling strategy (by default FCFS)

Table 7.3: Symbols for types of distributions

Symbol	Meaning
M	Exponential (Markovian) distribution
D	Deterministic distribution (i.e., constant times without variance)
E_k	Erlang distribution with parameter k
G or GI	General (independent) distribution

In practice, many systems serve requests with different arrival and service characteristics (e.g., the service rate of read and write requests to a database may be different). In theory, it would be possible to use multi-modal distributions for such cases; however, this can complicate the parameterization and solution of queueing models (Harchol-Balter, 2013, Chapter 21). Instead, *multi-class queues* are used where multiple types of requests are distinguished, referred to as *request classes* or *workload classes*. Each workload class represents a set of requests with similar characteristics, described by their own arrival rate and service rate parameters.

For a given queue i, performance metrics can be considered for a *transient* point in time t or for the *steady state* (i.e., $t \to \infty$). Generally, a system is considered to be in a steady state if the variables that define its behavior are unchanging in time (Gagniuc, 2017). In the context of queues, it is normally assumed that after a queue has been in operation for a given amount of time (referred to as *transient phase* or *warm-up period*), it will eventually reach a steady state, in which performance metrics are stable. In the following, we are interested in the steady-state solution of a queue. More details on the transient solution of a queue can be found in Bolch, Greiner, et al. (2006). Typical performance metrics of interest include: the utilization of the queue, the queue length, the request throughput, and the request response time. The *utilization* U_i is the fraction of time in which the queue is busy serving requests. The *queue length* Q_i specifies the number of requests waiting for service (excluding those currently in service). The *throughput* $X_{i,c}$ (where c stands for workload class c) is the number of requests of class c leaving the queue per unit of time. If the maximum number of requests that arrive at a queue is unlimited, the relation $\lambda < \mu$ must hold, so that the queue is stable (i.e., a steady-state solution exists). The *response time* $R_{i,c}$ of requests from workload class c is defined as

$$R_{i,c} = W_{i,c} + S_{i,c}, \tag{7.35}$$

where $W_{i,c}$ is the time a request has to wait in the queue before being served, and $S_{i,c}$ is the service time of the request. The waiting time $W_{i,c}$ depends on a number of parameters including the scheduling strategy and the arrival and service processes (i.e., the request inter-arrival and service time distributions).

7.2.2 Queueing Networks

A queueing network (QN) consists of two or more queues (service stations) that are connected together and serve requests sent by clients. Requests are grouped into classes (workload classes) where each class contains requests that have similar arrival behavior and processing requirements. The routing of requests in the queueing network is specified by a probability matrix. Requests of class c departing from service station i will move to service station j with probability $p_{c,i,j}$ or leave the network with probability $p_{c,i,out} = 1 - \sum_j p_{c,i,j}$. Requests of class c can arrive from outside the network at service station i with a rate $r_{c,i}$.

Figure 7.6 shows an example with three queues, one multi-server queue and two single server queues. The multi-server queue represents a multicore CPU, and the two single server queues represent a disk drive and a network, respectively. The connections between the queues illustrate how requests are routed through the network of queues. An incoming request, after being processed by the CPU, is routed either to the disk or to the network. The routes are labeled with probabilities. With a probability of 0.8, a request coming from the CPU is routed to the disk queue. With a probability of 0.2, a request coming from the CPU is routed to the queue representing the network. If a request is completed at the disk or the network queue, the request either leaves the queueing network with a probability of p_{leave}, or it is immediately routed back to the CPU queue with a probability of $1-p_{leave}$. A request may visit a queue multiple times while circulating through the queueing network. A request's total amount of service time at a queue, added up over all visits to the queue, is referred to as *service demand* or *resource demand* of the request at the queue.

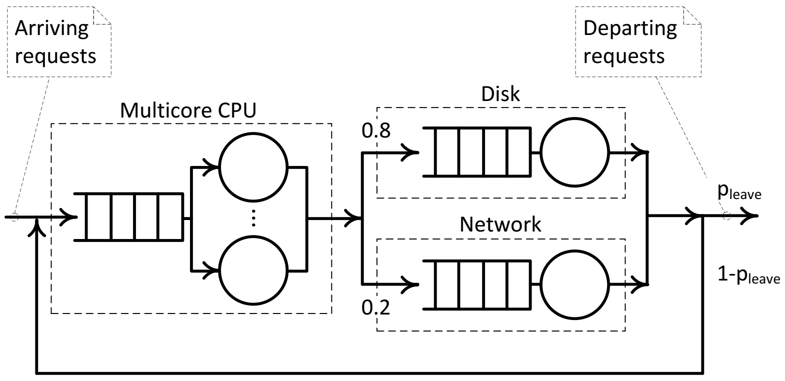


Fig. 7.6: Queueing network

A queueing network where the requests come from a source that is external of the queueing network and leave the network after service completion is referred to as *open*. A queueing network where there is no such external source of requests and there are no departing requests (i.e., the population of requests in the queueing network remains constant and is equal to the initial population) is referred to as *closed*. If a queueing network is open for some workload classes and closed for others, it is referred to as *mixed*.

In the context of queueing networks, the notation shown in Table 7.4 is typically used (similar to the notation we used in Section 7.1 when introducing operational analysis).

Table 7.4: Queueing network notation

Symbol	Meaning
K	Number of queues in the queueing network
C	Number of workload classes
λ_c	Average arrival rate of requests of class c in the queueing network (i.e., average number of requests that arrive per unit of time) (for open queueing networks)
$\lambda_{i,c}$	Average arrival rate of requests of class c at queue i
$\mu_{i,c}$	Service rate of requests of class c at queue i
$S_{i,c}$	Mean service time of requests of class c at queue i per visit to the queue (i.e., average time a request spends receiving service excluding waiting time)
$X_{i,c}$	Throughput of requests of class c at queue i (i.e., average number of service completions for class c per unit of time)
$V_{i,c}$	Average number of times queue i is visited during the processing of a request of class c, referred to as *visit ratio*
$X_{0,c}$	System throughput for class c (i.e., total number of requests of class c processed per unit of time)
U_i	Utilization of queue i (i.e., the fraction of time the queue is busy serving requests of any class)
$U_{i,c}$	Utilization of queue i due to requests of class c (i.e., the fraction of time the queue is busy serving requests of class c)
$D_{i,c}$	Service demand / resource demand (i.e., mean total service time of a request of class c at queue i over all visits to the queue)
$W_{i,c}$	Mean time a request of class c has to wait in the waiting line of queue i before being served
$R_{i,c}$	Mean response time of requests of class c at queue i (i.e., the average time it takes to process a request including both the waiting and service time in the queue)
$N_{i,c}$	Average number of requests of class c at queue i, either waiting for service or being served
$N_{0,c}$	Average number of requests of class c in the queueing network, either waiting for service or being served
Q_i	Average length of queue i (i.e., average number of requests in the queue waiting for service)
$Q_{i,c}$	Average number of requests of class c waiting for service at queue i

Given a queueing network, typical metrics of interest are the response time and throughput for each workload class and the utilization of each queue. In order to analyze a queueing network quantitatively, the queueing network's *workload* needs to be specified. For each workload class, the *workload intensity* and the *resource demands* for each visited queue have to be specified. How the workload intensity is characterized depends on whether it is a closed workload or an open workload. A closed workload is characterized by the number of requests; an open workload is characterized by the inter-arrival time of requests. A queueing network is said to be in *steady state* (or operational equilibrium) if the number of requests arriving at the queueing network per unit of time is equal to the number of requests departing from the queueing network, that is, the arrival rate is equal to the throughput. Closed formulas for the response times of requests are not easy to derive since they depend (among others) on the shape of the involved distributions (i.e., the inter-arrival time and service time distributions for each queue).

The solution of a queueing network with K service stations and C workload classes is based on deriving the steady-state probabilities $\pi(\mathbf{N_1}, \mathbf{N_2}, ..., \mathbf{N_K})$, where $\mathbf{N_i} = (n_1, n_2, ..., n_C)$ is a vector composed of the number of requests of each workload class c at service station i. Calculating the steady-state probabilities for a general queueing network requires construction of the complete state space. This can be a compute and memory-intensive task and suffers from the problem of *state space explosion* with increasing numbers of service stations and requests. However, the construction of the complete state space is not required for a special class of queueing networks called *product-form queueing networks*.

Product-form queueing networks have a special structure that allows one to compute the steady-state probabilities for the queueing network from the respective steady-state probabilities for the individual service stations using the following equation:

$$\pi(\mathbf{N_1}, \mathbf{N_2}, ..., \mathbf{N_K}) = \frac{1}{G} \left[\pi(\mathbf{N_1}) \cdot \pi(\mathbf{N_2}) \cdot \ldots \cdot \pi(\mathbf{N_K})\right], \tag{7.36}$$

where G is a normalizing constant (Bolch, Greiner, et al., 2006, p. 281). Thus, a solution of the queueing network can be obtained by analyzing the steady-state probabilities of each service station independently. Kelly showed that every queueing network with *quasi-reversible queues* and *Markovian routing* has a product-form solution (Kelly, 1975, 1976). Quasi-reversibility means that "the current state, the past departures, and the future arrivals are mutually independent" (Balsamo, 2000). Markovian routing means that the routing of requests does not depend on the current state of the queueing network.

The BCMP theorem (Baskett et al., 1975) showed that this property holds for the following types of service stations:

1. $M/M/m$ with FCFS scheduling assuming that the service rate does not depend on the workload class,
2. $M/G/1$ with PS scheduling,
3. $M/G/\infty$ with IS scheduling, and
4. $M/G/1$ with LCFS scheduling with preemption.

The service rate distribution in cases (2), (3), and (4) are required to have rational Laplace transforms. In practice, this is no limitation since any exponential, hyperexponential, or hypoexponential distribution fulfills this requirement, and all other types of distributions can be approximated by a combination of these distributions (Cox, 1955).

Furthermore, Baskett et al. (1975) showed that the product-form property holds for these scheduling strategies even with certain forms of state-dependent service rates. Among others, the service rate may depend on the number of requests at a service station. Thus, queues with multiple servers are also allowed for PS and LCFS scheduling.

7.2.3 Operational Laws

The operational laws introduced in Section 7.1 provide a quick and simple way to determine certain average performance metrics of a queue. These laws are independent of the arrival and service processes, or the scheduling strategy. Therefore, they can be applied both to a single queue and to an entire queueing network. The only assumption is that the considered queue (or queueing network) is in steady state (operational equilibrium).

Consider a multi-server queue i with m_i servers. The most fundamental law in queueing theory is *Little's law*, which applied to queue i states that the average number of requests $N_{i,c}$ of workload class c at queue i is equal to the product of the request arrival rate λ_c and the average time $R_{i,c}$ requests spent in the queue (i.e., the response time); that is,

$$N_{i,c} = \lambda_c \cdot R_{i,c}. \tag{7.37}$$

The *utilization law*, applied in the context of queue i, states that

$$U_{i,c} = \frac{X_{i,c} \cdot S_{i,c}}{m_i}, \tag{7.38}$$

where $U_{i,c}$ is the utilization of the queue due to requests of class c, $S_{i,c}$ is the service time, and $X_{i,c}$ is the throughput for requests of class c.

Finally, the *service demand law* relates the service demand $D_{i,c}$ of requests from class c with the utilization $U_{i,c}$ and the system throughput $X_{0,c}$ for class c:

$$D_{i,c} = \frac{m_i \cdot U_{i,c}}{X_{0,c}}. \tag{7.39}$$

7.2.4 Response Time Equations

The response time $R_{i,c}$ for $M/G/m$ queues with PS or preemptive LCFS scheduling, as well as for $M/M/m$ queues with class-independent service rates and FCFS scheduling, is given by

$$R_{i,c} = D_{i,c}\left(1 + \frac{1}{m_i} \cdot \frac{PB_i}{1 - U_i}\right), \tag{7.40}$$

where PB_i is the probability that all m_i servers of the queue are busy and an incoming request has to wait in the waiting line. PB_i can be calculated using the Erlang-C formula:

$$\begin{aligned} PB_i &= \frac{(m_i U_i)^{m_i}}{m_i!(1 - U_i)} \cdot \pi_{i,0} \\ \text{with } \pi_{i,0} &= \left[\sum_{k=0}^{m_i - 1} \frac{(m_i U_i)^k}{k!} + \frac{(m_i U_i)^{m_i}}{m_i!} \frac{1}{1 - U_i}\right]^{-1}. \end{aligned} \tag{7.41}$$

If a queue has IS scheduling strategy, a request never has to wait for service and the response time simplifies to

$$R_{i,c} = D_{i,c}. \tag{7.42}$$

For single server queues (i.e., $m_i = 1$), the busy probability PB_i is equal to the utilization U_i. As a result, Equation (7.40) can be simplified to

$$R_{i,c} = \frac{D_{i,c}}{1 - U_i}. \tag{7.43}$$

We refer to Bolch, Greiner, et al. (2006, p. 251) for the derivation and mathematical proof of the above equations.

The previous equations are not valid for $M/M/m$ service stations with FCFS scheduling and service rates depending on the workload class. The response time of such service stations can only be approximated. Franks (2000) compared the accuracy of different approximations and proposed the following one:

$$R_{i,c} = D_{i,c} + \frac{PB_i}{m_i} \sum_{s=1}^{C} Q_{i,c} \cdot D_{i,c}, \tag{7.44}$$

where $Q_{i,c}$ is the queue length of requests of workload class c at service station i.

7.2.5 Solution Techniques for Queueing Networks

Different solution techniques for queueing networks have been developed in the last decades. They can be broadly classified into simulation and analytic techniques. Discrete event simulation can be used to analyze arbitrarily complex queueing networks. However, it often is necessary to simulate a queueing network for a long time in order to obtain sufficiently accurate results.

Analytic techniques can provide exact solutions of a queueing network, avoiding the overhead of simulation. There are state-space and non-state-space techniques (Bolch, Greiner, et al., 2006). State-space techniques rely on the generation of the complete underlying state space of a queueing network, limiting their scalability with increasing numbers of requests, workload classes, and service stations. If certain assumptions are fulfilled, non-state-space techniques can be used instead. Given a product-form queueing network with an open workload, we can apply the equations presented in Section 7.2.4 to directly calculate performance metrics for the individual queues. Given a product-form queueing network with a closed workload, the calculation of the normalizing constant G in Equation (7.36) is nontrivial. Mean Value Analysis (MVA) (Bolch, Greiner, et al., 2006) is a recursive algorithm to calculate the queue lengths in closed product-form queueing networks, avoiding the direct determination of the normalizing constant G.

Techniques to solve queueing networks are supported by various tools, such as SHARPE (Hirel et al., 2000; Sahner and Trivedi, 1987), JMT (Bertoli et al., 2009), JINQS (Field, 2006), SPEED (Smith and Williams, 1997), and queueing-tool (Jordon, 2014).

Queueing networks provide a powerful method for modeling contention due to processing resources, that is, hardware contention and scheduling strategies. For certain classes of queueing networks, there are efficient analysis methods available. However, queueing networks are not suitable for representing blocking behavior, synchronization of processes, simultaneous resource possession, or asynchronous processing (Kounev, 2005). There are extensions of queueing networks such as *Extended Queueing Networks (EQNs)* (Bolch, Greiner, et al., 2006) that provide some support to mitigate the mentioned drawbacks.

7.2.6 Case Study

Now that we have introduced the basics of queueing network models, we present a case study—based on Kounev and Buchmann (2003)—showing how queueing networks can be used to model and predict the performance of a distributed software system. Imagine the following hypothetical scenario: A company is about to automate its internal and external business operations with the help of e-business technology. The company chooses to employ the Java EE platform[6], and it develops an application for supporting its order-inventory, supply-chain, and manufacturing operations. Assume that this application is the one provided by the SPECjAppServer benchmark.[7] SPECjAppServer models businesses using four domains: (1) *customer domain*—dealing with customer orders and interactions; (2) *manufacturing domain*—performing "just-in-time" manufacturing operations; (3) *supplier domain*—handling interactions with external suppliers; and (4) *corporate domain*—managing all customer, product, and supplier information. Figure 7.7 illustrates these domains and gives some examples of typical transactions run in each of them.

[6] Java EE platform: https://www.oracle.com/java/technologies/java-ee-glance.html

[7] SPECjAppServer benchmark: https://www.spec.org/jAppServer

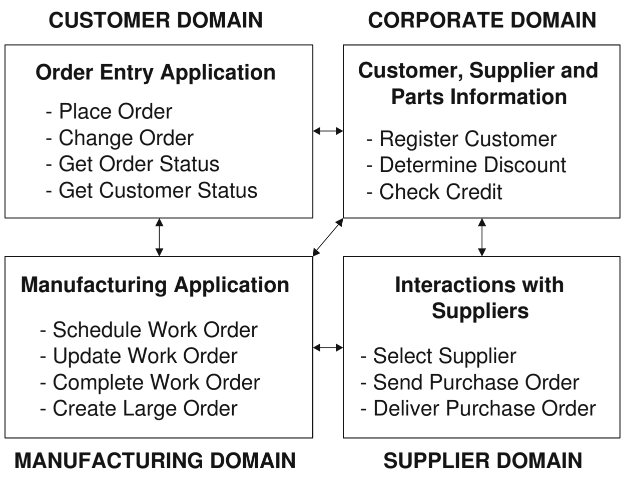


Fig. 7.7: SPECjAppServer business domains

The customer domain models customer interactions using an order-entry application, which provides some typical online ordering functionality. Orders can be placed by individual customers as well as by distributors. Orders placed by distributors are called *large orders*.

The manufacturing domain models the activity of production lines in a manufacturing plant. Products manufactured by the plant are called *widgets*. There are two types of production lines, namely *planned lines* and *large order lines*. Planned lines run on schedule and produce a predefined number of widgets. Large order lines run only when a large order is received in the customer domain. The unit of work in the manufacturing domain is a *work order*. Each work order is for a specific quantity of a particular type of widget. When a work order is created, the bill of materials for the corresponding type of widget is retrieved and the required parts are taken out of inventory. As the widgets move through the assembly line, the work order status is updated to reflect progress. Once the work order is complete, it is marked as completed and the inventory is updated. When the inventory of parts gets depleted, suppliers need to be located and *purchase orders (POs)* need to be sent out. This is done by contacting the supplier domain, which is responsible for interactions with external suppliers.

Assume that the company plans to deploy the application in the deployment environment depicted in Figure 7.8. This environment uses a cluster of WebLogic servers (WLS) as a Java EE container and an Oracle database server (DBS) for persistence. We assume that all machines in the WLS cluster are identical.

Before putting the application into production, the company conducts a capacity planning study in order to come up with an adequate sizing and configuration of the deployment environment. More specifically, the company needs to answer the following questions:

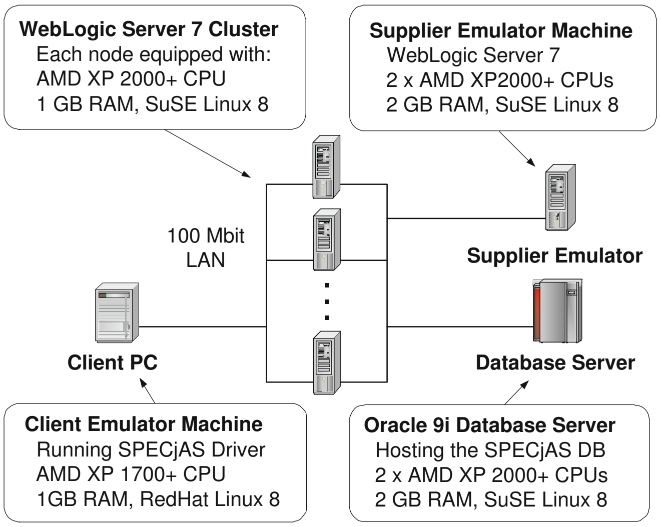


Fig. 7.8: Deployment environment

- How many WebLogic servers would be needed to guarantee adequate performance under the expected workload?
- For a given number of WebLogic servers, what level of performance would the system provide? What would be the average transaction throughput and response time? What would be the utilization (CPU/disk utilization) of the WebLogic servers and the database server?
- Will the capacity of the database server suffice to handle the incoming load?
- Does the system scale or are there any other potential system bottlenecks?

These issues can be approached with the help of queueing network-based performance models.

7.2.6.1 Workload Characterization

The first step in the capacity planning process is to describe the workload of the system under study in a qualitative and quantitative manner. This is called *workload characterization* (Menascé and Almeida, 1998), and it typically includes four steps:

1. Describe the types of requests that are processed by the system (i.e., the *request classes*),
2. Identify the hardware and software resources used by each request class,
3. Measure the total amount of service time for each request class at each resource (i.e., the *service demand*), and
4. Specify the number of requests of each class the system will be exposed to (i.e., the *workload intensity*).

As already discussed, the SPECjAppServer workload is made up of two major components: (1) the *order-entry application* in the customer domain and (2) the *manufacturing application* in the manufacturing domain. Recall that the order-entry application is running the following four transaction types:

1. *NewOrder*: places a new order in the system,
2. *ChangeOrder*: modifies an existing order,
3. *OrderStatus*: retrieves the status of a given order, and
4. *CustStatus*: lists all orders of a given customer.

We map each of them to a separate *request class* in our workload model. The manufacturing application, on the other hand, is running production lines. The main unit of work there is a *work order*. Each work order produces a specific quantity of a particular type of widget. As already mentioned, there are two types of production lines: planned lines and large order lines. While planned lines run on a predefined schedule, large order lines run only when a large order arrives in the customer domain. Each large order results in a separate work order. During the processing of work orders, multiple transactions are executed in the manufacturing domain (i.e., scheduleWorkOrder, updateWorkOrder, and completeWorkOrder). Each work order moves along three virtual stations, which represent distinct operations in the manufacturing flow. In order to simulate activity at the stations, the manufacturing application waits for a designated time at each station. One way to model the manufacturing workload would be to define a separate request class for each transaction run during the processing of work orders. However, this would lead to an overly complex model and would limit the range of analysis techniques that would be applicable for its solution. Second, it would not be of much benefit, since after all, what most interests us is the rate at which work orders are processed and not the performance metrics of the individual work order-related transactions. Therefore, we model the manufacturing workload only at the level of work orders. We define a single request class *WorkOrder*, which represents a request for processing a work order. This keeps our model simple, and as will be seen later, it is enough to provide us with sufficient information about the behavior of the manufacturing application.

Altogether, we end up with five request classes: NewOrder, ChangeOrder, OrderStatus, CustStatus, and WorkOrder. The following resources are used during their processing:

- The CPU of a WebLogic server (WLS-CPU),
- The local area network (LAN),
- The CPUs of the database server (DBS-CPU), and
- The disk drives of the database server (DBS-I/O).

In order to determine the service demands at these resources, we conducted a separate experiment for each of the five request classes. In each case, we deployed the benchmark in a configuration with a single WebLogic server and then injected requests of the respective class into the system. During the experiment, we monitored the system resources and measured the time requests spent at each resource during their processing. For the database server, we used the Oracle 9i Intelligent Agent, which provides exhaustive information about CPU consumption and I/O wait times. For the application server, we monitored the CPU utilization using operating system tools; we then used the *service demand law* ($D = U/X$) to derive the CPU service demand (see Section 7.2.3).

We decided we could safely ignore network service demands, since all communication was taking place over a 100 MBit LAN, and communication times were negligible. Figure 7.9 reports the service demand measurements for the five request classes in our workload model.

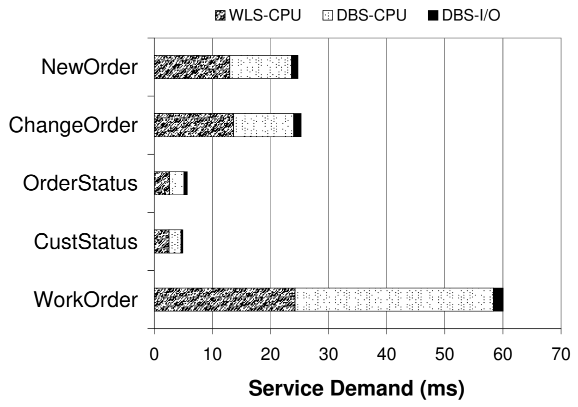


Fig. 7.9: Workload service demands

Database I/O service demands are much lower than CPU service demands. This stems from the fact that data is cached in the database buffer, and disks are usually accessed only when updating or inserting new data. However, even in this case, the I/O overhead is minimal, since the only thing that is done is to flush the database log buffer, which is performed with sequential I/O accesses. Here we would like to point out that the benchmark uses relatively small data volumes for the workload intensities generated. This results in data contention (Kounev and Buchmann, 2002), and as we will see later, it causes some difficulties in predicting transaction response times. Once we know the service demands of the different request classes, we proceed with the last step in workload characterization, which aims to quantify the workload intensity. For each request class, we must specify the rates at which requests arrive. We should also be able to vary these rates, so that we can consider different scenarios. To this end, we modified the SPECjAppServer driver to allow more flexibility in configuring the intensity of the workload generated. Specifically, the new driver allows us to set the number of concurrent order entry clients simulated as well as their average *think time*, that is, the time they "think" after receiving a response from the system, before they send the next request. In addition to this, we can specify the number of planned production lines run in the manufacturing domain and the time they wait after processing a work order before starting a new one. In this way, we can precisely define the workload intensity and transaction mix. We will later study in detail several scenarios under different transaction mixes and workload intensities.

7.2.6.2 Building a Performance Model

We now build a queueing network model of our SPECjAppServer deployment environment. We first define the model in a general fashion and then customize it to our concrete workload scenarios. We use a closed model, which means that for each instance of the model, the number of concurrent clients sending requests to the system is fixed. Figure 7.10 shows a high-level view of our queueing network model.

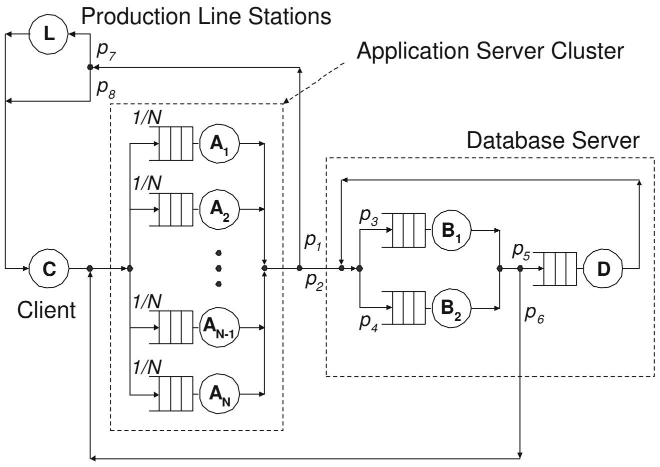


Fig. 7.10: Queueing network model of the system

In the following, we briefly describe the queues used:

C :	Infinite-Server (IS) queue (delay resource) used to model the client machine, which runs the SPECjAppServer driver and emulates virtual clients sending requests to the system. The service time of order entry requests at this queue is equal to the average client think time; the service time of WorkOrder requests is equal to the average time a production line waits after processing a work order before starting a new one. Note that times spent on this queue are not part of system response times.
$A_1..A_N$:	Processor-Sharing (PS) queues used to model the CPUs of the N WebLogic servers.
B_1, B_2 :	Processor-Sharing (PS) queues used to model the two CPUs of the database server.

D : First-Come-First-Served (FCFS) queue used to model the disk subsystem (made up of a single 100 GB disk drive) of the database server.

L : Infinite-Server (IS) queue (delay resource) used to model the virtual production line stations in the manufacturing domain. Only WorkOrder requests visit this queue. Their service time at the queue corresponds to the average delay at the production line stations simulated by the manufacturing application during work order processing.

The model is a closed queueing network model with the five classes of requests (jobs) defined in the previous section. The behavior of requests in the model is defined by specifying their respective routing probabilities p_i and service demands at each queue they visit. We discussed the service demands in the previous section. To set the routing probabilities, we examine the life cycle of client requests in the queueing network. Every request is initially at the client queue C, where it waits for a user-specified think time. After the think time elapses, the request is routed to a randomly chosen queue A_i, where it queues to receive service at a WebLogic server CPU.

We assume that requests are evenly distributed among the N WebLogic servers; that is, each server is chosen with probability $1/N$. Processing at the CPU may be interrupted multiple times if the request requires some database accesses. Each time this happens, the request is routed to the database server, where it queues for service at one of the two CPU queues B_1 or B_2 (each chosen equally likely, so that $p_3 = p_4 = 0.5$). Processing at the database CPUs may be interrupted in case I/O accesses are needed. For each I/O access, the request is sent to the disk subsystem queue D; after receiving service there, it is routed back to the database CPUs. This may be repeated multiple times, depending on the routing probabilities p_5 and p_6.

Having completed their service at the database server, requests are sent back to the application server. Requests may visit the database server multiple times during their processing, depending on the routing probabilities p_1 and p_2. After completing service at the application server, requests are sent back to the client queue C. Order entry requests are sent directly to the client queue (for them, $p_8 = 1$ and $p_7 = 0$), while WorkOrder requests are routed through queue L (for them, $p_8 = 0$ and $p_7 = 1$), where they are additionally delayed for 1 s. This delay corresponds to the 1 s delay at the three production line stations imposed by the manufacturing application during work order processing.

In order to set routing probabilities p_1, p_2, p_5, and p_6, we need to know how many times a request visits the database server during its processing and, for each visit, how many times, I/O access is needed. Since we know only the total service demands over all visits to the database, we assume that requests visit the database just once and need a single I/O access during this visit. This allows us to drop routing probabilities p_1, p_2, p_5, and p_6 and leads us to the simplified model depicted in Figure 7.11.

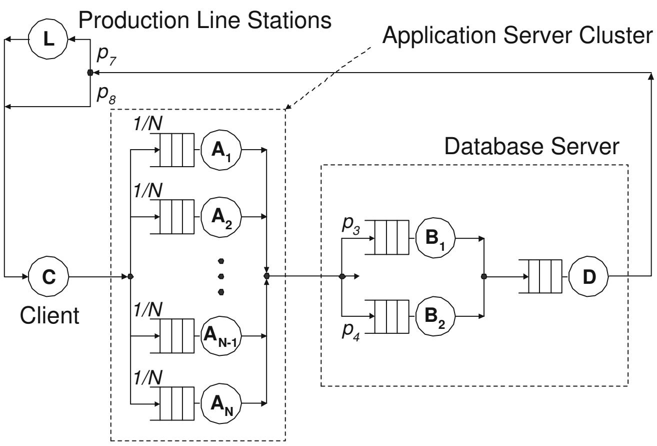


Fig. 7.11: Simplified QN model of the system

The following input parameters need to be supplied before the model can be analyzed:

- Number of order entry clients (NewOrder, ChangeOrder, OrderStatus, and CustStatus),
- Average think time of order entry clients—*Customer Think Time*,
- Number of planned production lines generating WorkOrder requests,
- Average time production lines wait after processing a work order, before starting a new one—*Manufacturing (Mfg) Think Time*, and
- Service demands of the five request classes at queues A_i, B_j, and D.

In our study, we consider two types of deployment scenarios. In the first one, large order lines in the manufacturing domain are turned off. In the second one, they are running as defined in the benchmark workload. The reason for this separation is that large order lines introduce some asynchronous processing, which is generally hard to model using queueing networks. We start with the simpler case where we do not have such processing, and we then show how large order lines can be integrated into the model.

7.2.6.3 Model Analysis and Validation

We now proceed to analyze several different instances of the model, and we then validate them by comparing results from the analysis with measured data. We first consider the case without large order lines and study the system in three scenarios representing low, moderate, and heavy load, respectively. In each case, we examine deployments with different number of application servers—from one to nine. Table 7.5 summarizes the input parameters for the three scenarios we consider.

Table 7.5: Model input parameters for the three scenarios

Parameter	Low	Moderate	Heavy
NewOrder clients	30	50	100
ChangeOrder clients	10	40	50
OrderStatus clients	50	100	150
CustStatus clients	40	70	50
Planned lines	50	100	200
Customer think time	2 s	2 s	3 s
Mfg think time	3 s	3 s	5 s

We employed the *PEPSY-QNS* tool (Bolch and Kirschnick, 1994), which supports a wide range of solution methods (over 30) for product-form and non-product-form queueing networks. Both exact and approximate methods are provided, which are applicable to models of considerable size and complexity. For the most part, we have applied the *multisum method* (Bolch, 1989) for solution of the queueing network models in this case study. However, to ensure plausibility of the results, we cross verified them with results obtained from other methods such as *bol_aky* and *num_app* (Bolch and Kirschnick, 1994). In all cases, the difference was negligible.

Low Load Scenario Table 7.6 summarizes the results we obtained for our first scenario. We studied two different configurations—the first one with one application server and the second one with two application servers. The table reports throughput (X) and response time (R) for the five request classes as well as CPU utilization (U) of the application server and the database server. Results obtained from the model analysis are compared against results obtained through measurements, and the modeling error is reported.

As we can see from the table, while throughput and utilization results are extremely accurate, this does not hold to this extent for response time results. This is because when we run a transaction mix, as opposed to a single transaction, some additional delays are incurred that are not captured by the model. For example, delays result from contention for data access (database locks, latches), processes, threads, database connections, and so on. The latter is often referred to as *software contention*, in contrast to *hardware contention* (contention for CPU time, disk access, and other hardware resources). Our model captures the hardware contention aspects of system behavior and does not represent software contention aspects. While software contention may not always have a big impact on transaction throughput and CPU utilization, it usually does have a direct impact on transaction response time; therefore, the measured response times are higher than the ones obtained from the model. In Kounev (2006), some techniques were presented for integrating both hardware and software contention aspects into the same model.

Table 7.6: Analysis results for the first scenario—low load

	One application server			Two application servers		
Metric	Model	Measured	Error	Model	Measured	Error
NewOrder throughput	14.59	14.37	1.5%	14.72	14.49	1.6%
ChangeOrder throughput	4.85	4.76	1.9%	4.90	4.82	1.7%
OrderStatus throughput	24.84	24.76	0.3%	24.89	24.88	0.0%
CustStatus throughput	19.89	19.85	0.2%	19.92	19.99	0.4%
WorkOrder throughput	12.11	12.19	0.7%	12.20	12.02	1.5%
NewOrder response time	56 ms	68 ms	17.6%	37 ms	47 ms	21.3%
ChangeOrder resp. time	58 ms	67 ms	13.4%	38 ms	46 ms	17.4%
OrderStatus response time	12 ms	16 ms	25.0%	8 ms	10 ms	20.0%
CustStatus response time	11 ms	17 ms	35.2%	7 ms	10 ms	30.0%
WorkOrder response time	1,127 ms	1,141 ms	1.2%	1,092 ms	1,103 ms	1.0%
WebLogic server CPU util.	66%	70%	5.7%	33%	37%	10.8%
Database server CPU util.	36%	40%	10%	36%	38%	5.2%

From Table 7.6, we see that the response time error for requests with very low service demands (e.g., OrderStatus and CustStatus) is much higher than the average error. This is because the processing times for such requests are very low (around 10 ms) and the additional delays from software contention, while not that high as absolute values, are high relative to the overall response times. The results show that the higher the service demand for a request type, the lower the response time error. Indeed, the requests with the highest service demand (WorkOrder) always have the lowest response time error.

Moderate Load Scenario In this scenario, we have 260 concurrent clients interacting with the system and 100 planned production lines running in the manufacturing domain. This is twice as much compared to the previous scenario. We study two deployments—the first with three application servers and the second with six. Table 7.7 summarizes the results from the model analysis. Again, we obtain very accurate results for throughput and utilization, and we also obtain accurate results for response time. The response time error does not exceed 35%, which is considered acceptable in most capacity planning studies (Menascé et al., 2004).

Heavy Load Scenario In this scenario, we have 350 concurrent clients and 200 planned production lines in total. We consider three configurations—with four, six, and nine application servers, respectively. However, we slightly increase the think times in order to make sure that our single machine database server is able to handle the load. Table 7.8 summarizes the results for this scenario. For models of this size, the available algorithms do not produce reliable results for response time, and therefore, we only consider throughput and utilization in this scenario.

Table 7.7: Analysis results for the second scenario—moderate load

	Three application servers			Six application servers		
Metric	Model	Measured	Error	Model	Measured	Error
NewOrder throughput	24.21	24.08	0.5%	24.29	24.01	1.2%
ChangeOrder throughput	19.36	18.77	3.1%	19.43	19.32	0.6%
OrderStatus throughput	49.63	49.48	0.3%	49.66	49.01	1.3%
CustStatus throughput	34.77	34.24	1.5%	34.80	34.58	0.6%
WorkOrder throughput	23.95	23.99	0.2%	24.02	24.03	0.0%
NewOrder response time	65 ms	75 ms	13.3%	58 ms	68 ms	14.7%
ChangeOrder resp. time	66 ms	73 ms	9.6%	58 ms	70 ms	17.1%
OrderStatus response time	15 ms	20 ms	25.0%	13 ms	18 ms	27.8%
CustStatus response time	13 ms	20 ms	35.0%	11 ms	17 ms	35.3%
WorkOrder response time	1,175 ms	1,164 ms	0.9%	1,163 ms	1,162 ms	0.0%
WebLogic server CPU util.	46%	49%	6.1%	23%	25%	8.0%
Database server CPU util.	74%	76%	2.6%	73%	78%	6.4%

Table 7.8: Analysis results for the third scenario—heavy load

	Four app. servers			Six app. servers			Nine app. servers		
Metric	Model	Msrd.	Error	Model	Msrd.	Error	Model	Msrd.	Error
NewOrder throughput	32.19	32.29	0.3%	32.22	32.66	1.3%	32.24	32.48	0.7%
ChangeOrder throughput	16.10	15.96	0.9%	16.11	16.19	0.5%	16.12	16.18	0.4%
OrderStatus throughput	49.59	48.92	1.4%	49.60	49.21	0.8%	49.61	49.28	0.7%
CustStatus throughput	16.55	16.25	1.8%	16.55	16.24	1.9%	16.55	16.46	0.5%
WorkOrder throughput	31.69	31.64	0.2%	31.72	32.08	1.1%	31.73	32.30	1.8%
WebLogic server CPU util.	40%	42%	4.8%	26%	29%	10.3%	18%	20%	10.0%
Database server CPU util.	87%	89%	2.2%	88%	91%	3.3%	88%	91%	3.3%

Large Order Lines Scenario We now consider the case when large order lines in the manufacturing domain are enabled. The latter are activated upon arrival of large orders in the customer domain. Each large order generates a separate work order, which is processed asynchronously at one of the large order lines. As already mentioned, this poses a difficulty since queueing networks provide limited possibilities for modeling this type of asynchronous processing. As shown in Kounev (2006), other state-space-based models such as *queueing Petri nets (QPNs)* are much more powerful in such situations.

Since large order lines are always triggered by NewOrder transactions (for large orders), we can add the load they produce to the service demands of NewOrder requests. To this end, we rerun the NewOrder experiments with the large order lines turned on. The additional load leads to higher utilization of system resources, and it impacts the measured NewOrder service demands (WLS-CPU: 23.49 ms, DBS-CPU: 21.61 ms, DBS-I/O: 1.87 ms). While this incorporates the large order line activity into our model, it changes the semantics of NewOrder jobs. In addition to the NewOrder transaction load, they now also include the load caused by large order lines. Thus, performance metrics (throughput, response time) for NewOrder requests no longer correspond to the respective metrics of the NewOrder transaction. Therefore, we can no longer quantify the performance of the NewOrder transaction on itself. Nevertheless, we can still analyze the performance of other transactions and gain a picture of the overall system behavior. Table 7.9 summarizes the results for the three scenarios with large order lines enabled. For lack of space, this time we look only at one configuration per scenario—the first one with one application server, the second one with three, and the third one with nine.

Table 7.9: Analysis results for the scenario with large order lines

	Low/1-AS		Moderate/3-AS		Heavy/9-AS	
Metric	Model	Error	Model	Error	Model	Error
ChangeOrder throughput	4.79	6.4%	19.09	3.5%	15.31	4.5%
OrderStatus throughput	24.77	2.9%	49.46	2.3%	48.96	3.1%
CustStatus throughput	19.83	2.4%	34.67	2.1%	16.37	1.9%
WorkOrder throughput	11.96	5.7%	23.43	2.6%	29.19	1.2%
WebLogic server CPU util.	80%	0.0%	53%	1.9%	20%	0.0%
Database server CPU util.	43%	2.4%	84%	2.4%	96%	1.0%

7.2.7 Conclusions from the Analysis

We used a queueing network model to predict the system performance in several different configurations, varying the workload intensity and the number of application servers available. The results enable us to give answers to the initial capacity planning questions. For each configuration, we obtained approximations for the average request throughput, the response time, and the server utilization. Depending on the Service-Level Agreements (SLAs) and the expected workload intensity, we

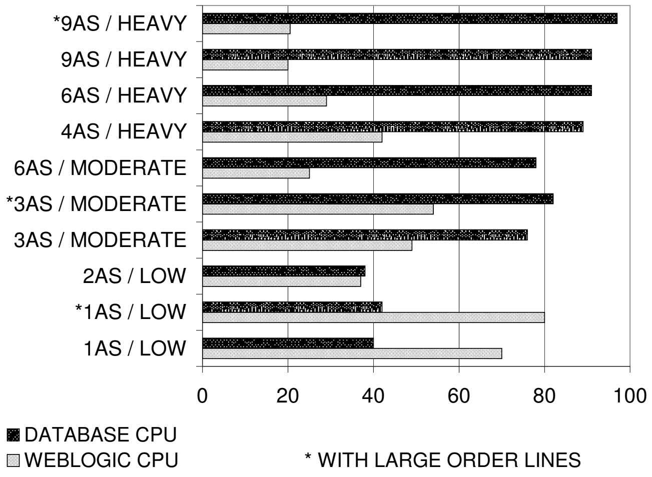


Fig. 7.12: Server utilization in different scenarios

can now determine how many application servers we need in order to guarantee adequate performance. We can also see, for each configuration, which component is mostly utilized and thus could become a potential bottleneck (see Figure 7.12). In the first scenario, we saw that by using a single application server, the latter could easily turn into a bottleneck, since its utilization would be twice as high as that of the database server. The problem is solved by adding an extra application server. In the second and third scenarios, we saw that with more than three application servers, as we increase the load, the database CPU utilization approaches 90%, while the application servers remain less than 50% utilized. This clearly indicates that, in this case, our database server is the bottleneck.

7.3 Concluding Remarks

In this chapter, we introduced some basic quantitative relationships between the most common performance metrics. We showed how these relationships, referred to as operational laws, can be applied to evaluate a system's performance based on measured or known data. This approach, known as operational analysis, can be seen as part of queueing theory, which provides general methods to analyze the queueing behavior at one or more service stations. Having looked at operational analysis, we provided a brief introduction to the basic notation and principles of queueing theory. While queueing theory is used in many different domains, from manufacturing to logistics, in this chapter, we focused on using queueing theory for performance evaluation of computer systems. Nevertheless, the introduced concepts and mathematical models are relevant for any processing system where the assumptions discussed in the beginning of the chapter hold. The chapter was wrapped up with a case study, showing how to build a queueing model of a distributed software system and use it to predict the system performance for different workload and configuration scenarios.

References

Balsamo, S. (2000). “Product Form Queueing Networks”. In: *Performance Evaluation: Origins and Directions*. Ed. by G. Haring, C. Lindemann, and M. Reiser. Vol. 1769. Lecture Notes in Computer Science. Springer-Verlag: Berlin, Heidelberg, pp. 377–401 (cited on p. 166).

Baskett, F., Chandy, K. M., Muntz, R. R., and Palacios, F. G. (1975). “Open, Closed, and Mixed Networks of Queues with Different Classes of Customers”. *Journal of the ACM*, 22(2). ACM: New York City, NY, USA, pp. 248–260 (cited on pp. 166, 167).

Bertoli, M., Casale, G., and Serazzi, G. (2009). “JMT: Performance Engineering Tools for System Modeling”. *SIGMETRICS Performance Evaluation Review*, 36(4). ACM: New York, NY, USA, pp. 10–15 (cited on p. 169).

Bolch, G. (1989). *Performance Evaluation of Computer Systems with the Help of Analytical Queueing Network Models*. Teubner Verlag: Leipzig, Germany (cited on p. 177).

Bolch, G. and Kirschnick, M. (1994). *The Performance Evaluation and Prediction SYstem for Queueing NetworkS—PEPSY-QNS*. Tech. rep. TR-I4-94-18. Germany: University of Erlangen-Nuremberg (cited on p. 177).

Bolch, G., Greiner, S., Meer, H. de, and Trivedi, K. S. (2006). *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*. Second Edition. John Wiley & Sons: Hoboken, New Jersey, USA (cited on pp. 160, 163, 166, 168, 169).

Cox, D. R. (1955). “A Use of Complex Probabilities in the Theory of Stochastic Processes”. *Mathematical Proceedings of the Cambridge Philosophical Society*, 51(2). Cambridge University Press: Cambridge, UK, pp. 313–319 (cited on p. 167).

Denning, P. J. and Buzen, J. P. (1978). “The Operational Analysis of Queueing Network Models”. *ACM Computing Surveys*, 10(3). ACM: New York, NY, USA, pp. 225–261 (cited on p. 150).

Field, T. (2006). *JINQS: An Extensible Library for Simulating Multiclass Queueing Networks V1.0 User Guide*. Imperial College London. London, UK (cited on p. 169).

Franks, R. G. (2000). “Performance Analysis of Distributed Server Systems”. PhD thesis. Ottawa, Canada: Carlton University (cited on p. 168).

Gagniuc, P. A. (2017). *Markov Chains: From Theory to Implementation and Experimentation*. John Wiley & Sons: Hoboken, New Jersey, USA (cited on p. 163).

Harchol-Balter, M. (2013). *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press: Cambridge, UK (cited on pp. 160, 163).

Hirel, C., Sahner, R. A., Zang, X., and Trivedi, K. S. (2000). "Reliability and Performability Modeling Using SHARPE 2000". In: *Proceedings of the 11th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS 2000)*. (Schaumburg, IL, USA). Lecture Notes in Computer Science. Springer-Verlag: Berlin, Heidelberg, pp. 345–349 (cited on p. 169).

Jordon, D. (2014). *Queueing-tool: A network simulator*. https://github.com/djordon/queueing-tool. Accessed: 2019-09-18 (cited on p. 169).

Kelly, F. P. (1975). "Networks of Queues with Customers of Different Types". *Journal of Applied Probability*, 12(3). Applied Probability Trust, pp. 542–554 (cited on p. 166).

– (1976). "Networks of Queues". *Advances in Applied Probability*, 8(2). Applied Probability Trust, pp. 416–432 (cited on p. 166).

Kendall, D. G. (1953). "Stochastic Processes Occurring in the Theory of Queues and their Analysis by the Method of the Imbedded Markov Chain". *The Annals of Mathematical Statistics*, 24(3). The Institute of Mathematical Statistics, pp. 338–354 (cited on p. 162).

Kounev, S. (2005). *Performance Engineering of Distributed Component-Based Systems—Benchmarking, Modeling and Performance Prediction*. Ph.D. Thesis, Technische Universität Darmstadt, Germany. Shaker Verlag: Herzogenrath, Germany (cited on p. 169).

– (2006). "Performance Modeling and Evaluation of Distributed Component-Based Systems using Queueing Petri Nets". *IEEE Transactions on Software Engineering*, 32(7). IEEE Computer Society: Washington, DC, USA, pp. 486–502 (cited on pp. 177, 180).

Kounev, S. and Buchmann, A. (2002). "Improving Data Access of J2EE Applications by Exploiting Asynchronous Processing and Caching Services". In: *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB 2002)*. (Hong Kong, China). VLDB Endowment, pp. 574–585 (cited on p. 173).

– (2003). "Performance Modeling and Evaluation of Large-Scale J2EE Applications". In: *Proceedings of the 29th International Conference of the Computer Measurement Group on Resource Management and Performance Evaluation of Enterprise Computing Systems (CMG 2003)*. (Dallas, TX, USA), pp. 273–283 (cited on p. 169).

Lazowska, E. D., Zahorjan, J., Graham, G. S., and Sevcik, K. C. (1984). *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall: Upper Saddle River, NJ, USA (cited on p. 160).

Little, J. D. C. (1961). "A Proof for the Queuing Formula: $L = \lambda W$". *Operations Research*, 9(3). Institute for Operations Research and the Management Sciences (INFORMS): Linthicum, Maryland, USA, pp. 383–387 (cited on p. 154).

Menascé, D. A. and Almeida, V. A. (1998). *Capacity Planning for Web Performance: Metrics, Models, and Methods*. Prentice Hall: Upper Saddle River, NJ, USA (cited on p. 171).

Menascé, D. A., Almeida, V. A., and Dowdy, L. W. (1994). *Capacity Planning and Performance Modeling: From Mainframes to Client-Server Systems*. Prentice Hall: Upper Saddle River, NJ, USA (cited on p. 159).

– (2004). *Performance by Design: Computer Capacity Planning By Example*. Prentice Hall: Upper Saddle River, NJ, USA (cited on pp. 150, 151, 154, 155, 157, 158, 178).

Sahner, R. A. and Trivedi, K. S. (1987). "Reliability Modeling Using SHARPE". *IEEE Transactions on Reliability*, 36(2). IEEE: Piscataway, New Jersey, USA, pp. 186–193 (cited on p. 169).

Smith, C. U. and Williams, L. G. (1997). "Performance Engineering Evaluation of Object-Oriented Systems with SPE*ED". In: *Proceedings from the International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS 1997)*. Ed. by R. Marie, B. Plateau, M. Calzarossa, and G. Rubino. Vol. 1245. Lecture Notes in Computer Science. Springer-Verlag: Berlin, Heidelberg, pp. 135–154 (cited on p. 169).