DRAFT DRAFT DRAFT DRAFT Performance-Oriented Patterns in Software Design (A multi-level service approach)

C.M. Woodside

Dept. of Systems and Computer engineering

Carleton University, Ottawa K1S 5B6

October 18, 2001, draft produced for classroom use

Chapter 1. Software Design and Performance (D)

1.1. Performance is Important and Difficult (D.1)

Adequate performance, in the sense of delay and throughput, is essential to the useful functioning of most software products. However it is often the last aspect of a program to be analyzed, only after integration and during final testing. At such a late stage the necessary changes may be difficult and expensive. Projects are delayed, go over cost targets, or fail altogether because of performance problems. This problem has been described by Smith as the “fix-it-later” approach, and she believes many programmers are averse to even considering performance in their designs.

The situation is made worse by the rush to market, and by the greater use of distributed systems such as client-server systems, communicating over the Internet or similar internal networks. Distribution and concurrency make these systems much harder to understand.

If one wants to estimate performance earlier, for example when designing the architecture of software, there is a frustrating lack of machinery to help. The standard approaches to design have limited places for performance requirements or information. It isn’t clear what information is needed, and it is easy to get into a maze of detail, with no guidance as to what is important. Design notations and CASE tools give relatively assis-

tance. In sequential programming it is not so bad, for experience and a few well-known guidelines are often enough to find the good design options. In network-based concurrent systems, and in parallel programming, there have been many disappointments where systems just do not deliver the expected performance. There are too many factors, and designer intuition cannot pull them all together without help.

The known approach to performance is through measurement on a nearly complete product. The techniques are sophisticated and difficult to apply correctly, but great detail can be obtained with adequate instrumentation. Measurement is essential and nothing written here is intended to downplay it, but earlier analysis is essential and this work attacks it via models. These models are constructed from the designs, combined with experience, and are used to calculate estimates to guide the early design, and to give insight into what kinds of situations are likely to arise.

This report assembles some thinking tools suitable for analyzing concurrent and distributed systems, using models. Using the tools, some recurring patterns of design are identified and their performance issues are analyzed in a general way. The main tool is a "layered model" notation that can describe the most important features of computer systems, from the viewpoint of performance. The features include both the hardware and the software design, and the software may include many concurrent processes, and processors and paths executing in parallel.The notation is used to describe and analyze a number of "performance-oriented patterns" or POPs, which seem to occur often in computer systems. These patterns occur at different levels of detail, from high-level architectural relationships between interacting processes (or between the processes and the hardware), down to detailed sequences of actions within a single process. However the main focus is at the level of concurrency design and architecture.

1.2. Patterns (D.2)

Performance-oriented patterns (POPs) are architectural or structural design elements which often recur in software designs. They define relationships between entities and activities. They have something in common with software design patterns, which are partial solutions to certain design problems in a form that can be reused, but POPs are more generic since they do not address function. One POP can represent many different design patterns.

Examples of three different kinds of POP are shown in Figures DC, DD and DE. These are very simple recognizable structural patterns in three different software points-of-view.

Figure DC is instantly recognizable as showing a sequence of activities with a parallel operation, with forking and joining of the flow. The boxes are activities and the arrows represent predecessor relationships. We will term this kind of model an "activity graph", but it is also known as a "task graph" in the literature. Figure DD shows a module interface, just a calling interface between two procedures in a sequential program. The boxes in this figure represent procedures, or some other kind of structural module (object, Ada package). Figure DE shows a similar call but now occurring between two processes with a client-server relationship. The parallelograms represent parallel tasks that execute concurrent and the "call" is a remote procedure call. The different processors for the two

tasks are also shown as ellipses. These are very small patterns that are only building blocks in creating an architecture; our goal is to create a repertoire of useful patterns that wen can recognize in any design, and that can guide our understanding of its performance problems.

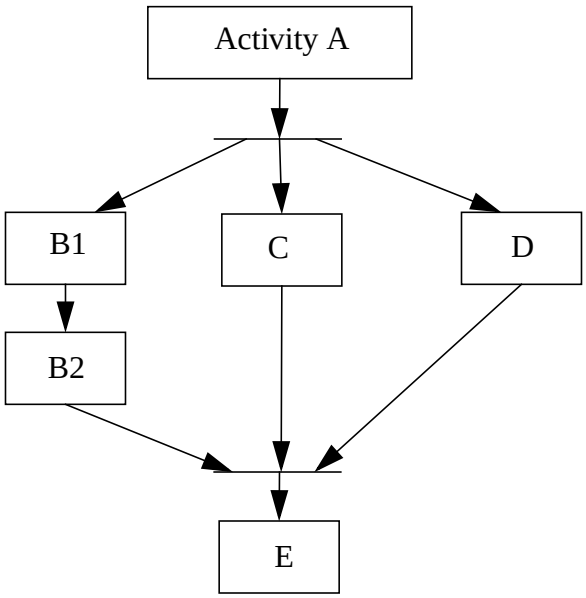


Figure 1.1. An Activity Diagram with a Fork-Join Pattern
(Figure DC)

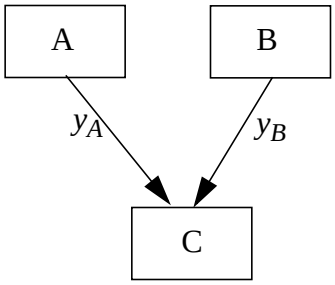


Figure 1.2. A Module Pattern, where one Module Calls Another (Figure DD)

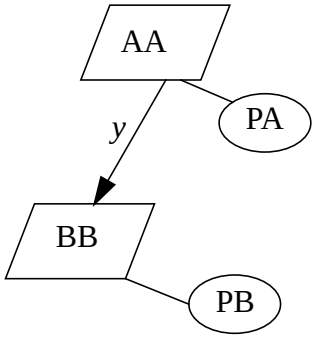


Figure 1.3. A Client-Server Pattern, where one Concurrent Task Requests Service from Another (Figure DE)

In the activity-graph view expressed by Figure DC we can analyze the possibilities and the effectiveness of parallel execution. How long is each of the three subpaths? If two are quite short compared to the third, parallelism offers little improvement. What program parameters affect the subpath lengths? How much of subpath time is overhead to set up the parallel execution on a separate processor, perhaps with communication over a network?

In the view expressed in Figure DD we can examine the effect of modularization on performance. Crossing a module boundary introduces some overhead to pass parameters, which may be greater than the "useful work" in a very small procedure. Aggregating or "inlining" module B into module A may introduce savings, whose value depends on the frequency of calling B.

More significant issues become visible in concurrent software with communicating processes as shown in Figure DE. For one thing the overhead of an interprocess "call" (which may be a remote procedure call or RPC) is commonly much higher. Thus there may be an advantage in combining many requests into one list of requests to reduce overhead (batching of requests).

Quite complex patterns occur between processes, and it is the particular purpose of this report to examine this class of patterns, at the level of concurrency architecture. However in doing so we will sometimes want to use activity patterns and module patterns as well.

1.3. Using Patterns to Improve a Design (D.3)

There are two ways that patterns can help in improving the performance of a design. First, there are some standard *pattern substitutions* that can be used, such as

- inlining a procedure,
- parallelizing a section of code

• introducing an optimistic algorithm

which may reduce the overall work, or cause a response to complete earlier.

Second, when a pattern is detected, one could take advantage of standard rules for optimizing the pattern by modifying the software within it (if such rules existed). We can call this *optimization within the pattern*. One purpose of this report is to find some of these rules, by examining some of the patterns from this point of view and trying to understand how they can be improved. Given a nudge from such a generic indication of the improvement, it is still up to the designer to find a way to achieve the indicated change. For example to maximize the capacity of a pipeline the balance of effort should be shifted towards the earlier stages, if there is significant variance in the execution delay of the stages. How to achieve this is up to the designer.

Even these few examples show that to exploit patterns one has to have some parameters (such as frequencies of calls) and some performance results (such as variations in delays). The numbers could come from measurements, from simulations, or from an analytic model. The report uses a notational framework for the parameters, whatever their source. There may appear to be a bias towards analytic layered modelling, but it is not basic to the ideas. Analytic models have been used to study issues in the report, because they make a self-contained story, but the ideas are general.

1.4. An Office Workflow System (D.4)

A basic but rather typical example will motivate the analysis approach developed later. It is a small office system for storing and processing documents, with a workflow manager that sequences and tracks the operations. Each user has a client workstation (a desktop PC) running the office applications for entering and processing documents. Storage and workflow processing are managed by a single Workflow server task running on a server workstation and accessed over a local area networks (LAN). The storage itself is on a single disk attached to the server. For simplicity we consider the server to be a single task, and we will ignore the LAN delay in considering the system performance.

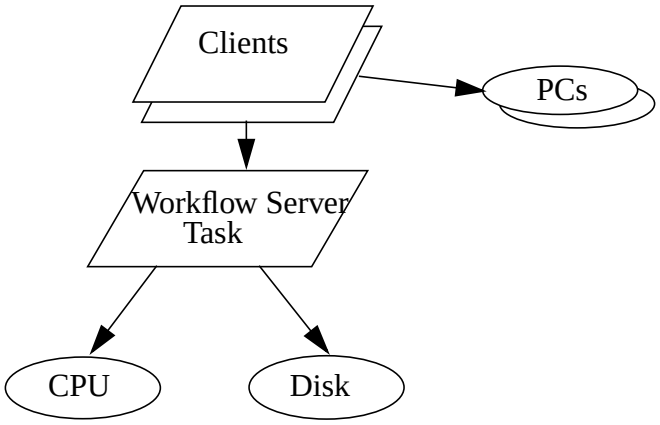


Figure 1.4. Office Workflow System (Figure DG)

This vertical system architecture is a simple case of a common pattern we will later call a “tower”, and all towers share a set of performance characteristics, problems and cures. We will study the pattern in order to be ready for it in any form it takes, and to find its limits.

Some design issues for the software that may affect performance are, how much of the work to do on the clients (the question of “thick” clients versus “thin” clients) and whether the server design must be multi-threaded.

The performance measure we most want to know is the mean response time R-Client for those worksteps that access the server, for different numbers of users N-Client. Other measures that may help us understand the performance are the throughput of the server for different numbers of the users, and the utilizations of the server CPU, the Workflow task and the disk (that is, the fraction of time each one is busy). These utilizations are named U-Server, U-Wflow, and U-Disk.

The software design influences the performance by the way it causes the program to execute operations that take time, and sequences of operations. This is termed the “workload” of the program, and in this case the workload has the following aggregate workload parameters, with values we shall suppose we know:

- the client CPU time per user interaction (CPU-Client),
- the time the users think between entering commands in the client software (Z-Client),
- the mean number of requests to the Workflow task, per user interaction (Y-Client-Wflow),
- the CPU demand of the Workflow task, per request to it (CPU-Wflow),
- the mean number of disk operations by the Workflow task, per request to it (Y-Wflow-Disk),
- the mean disk operation time (S-Disk).

There are four resources which may bottleneck this system and limit its performance: the disk, the server CPU, the server memory, and the Workflow task itself, if it is single-threaded. (A single-threaded task in this work is one which can only serve one request at a time, and cannot serve the next request until the previous one is completed.) Consider the workflow task first:

Bottleneck at the Workflow Task (Single-threaded): if this is the only task running on the Server, then its time to run for one request is the sum of the CPU time and the disk time. Call this sum S-Wflow, for the service time of the Workflow task. When it is computing, the disk is idle and when it is doing I/O the CPU is idle, so neither of these can be saturated. With enough users however the task can be saturated, so a queue of user requests builds up. In this condition the throughput f-Wflow of the Workflow task is limited to

f-Wflow = 1 / S-Wflow = 1 / (CPU-Wflow + Y-Wflow-Disk * S-Disk)

and the throughput of the clients is limited (because they have to wait for Workflow service) to a rate determined by the number of requests it makes to the server:

f-Client = f-Wflow / Y-Client-Wflow

We will see that the mean response time R-Client then is given by a simple equation

R-Client = (N-Client / f-Client) - (Z-Client + CPU-Client)

which can be re-arranged into the form, for large (many client users) N-Client

R-Client = N-Client * Y-Client-Wflow * (CPU-Wflow + Y-Wflow-Disk * S-Disk)
- (Z-Client + CPU-Client)

Figure DM plots the equations for R-Client and f-Client. These curves are a familiar feature of performance analysis. From the equations one can see the software design affects this response time through three of the parameters:

- CPU-Wflow, the CPU demand of one response of the Workflow task, which could be reduced by tuning the code;
- Y-Client-Wflow, the frequency of demands to the Server, which might be reduced by reorganizing the interface between the user applications and the Workflow task or by migrating more functionality to the Clients;
- Y-Wflow-Disk, the frequency of disk operations, which might be reduced by re-organizing the data storage on disk.

Software design also has an effect via the single threaded design of the Workflow task. The task is always busy, either using the CPU or waiting for the Disk. Neither device is saturated. However a multithreaded design is a lot more complex, and has additional CPU overhead costs. In turn this complexity is less serious if one uses a standard multithreaded design pattern.

Balanced Execution: If CPU-Wflow and S-Disk were about equal, then the throughput limit could in principle be doubled by using two threads in the Workflow task. The response time R-Client would be reduced by at least a factor of 2, and maybe by quite a bit more, depending on the other parameters. (The first term in R-Client would be cut in half, before subtracting the final term.) If there were a lot of inter-thread overhead, this would increase CPU-Wflow and reduce the potential gains somewhat, but as a general rule Tower patterns demand multi-threading for best performance.

To get the same effect by tuning the CPU time, it would have to be cut to zero, which is clearly impossible. A 10% reduction in CPU-Wflow would give a 5% increase in the saturation throughput, and a reduction in R-Client that would depend on other parameters as well, but would be at least 5%.

***Bottleneck at the Disk*:** Supposing that there are lots of threads, a bottleneck may appear at the disk. The symptom would be that U-Disk is 1.0 and other utilizations are lower. Disk saturation occurs at the disk throughput value

f-Disk = 1/S-Disk

which corresponds to a Client throughput of

f-Client = 1/(Y-Client-Wflow * Y-Wflow-Disk * S-Disk)

Using the analysis derived in the next chapter we will find that this gives a client response time of

R-Client = (N-Client * Y-Client-Wflow * Y-Wflow-Disk * S-Disk) - (Z-Client + CPU-Client)

The notable difference from the last situation is that the Client response time is governed entirely by the bottleneck at the disk, in the limit for large enough N-Client. The CPU demand does not have any effect and tuning the code for CPU time would be a waste of effort. What will pay off in this situation is reducing the requests to the disks, somehow.

***Bottleneck at the CPU and Memory*:** A bottleneck at the CPU is similar to the Disk, except that now the CPU time is the dominant factor. This is the case with the greatest payoff for code tuning.

A memory bottleneck can arise because the program code is too large, the data space in main memory is too large, or there are too many threads (each of which gets a copy of the thread data structures). The effect in a system with virtual memory is to cause excessive loading and unloading of memory to the swap space on disk, known as thrashing. Thus this looks like a bottleneck at the disk, and instrumentation must be used which identifies the paging operation traffic separately, to identify this.

A final possibility in a multi-threaded system is a bottleneck due to heavy use of a control such as a critical section that protects a shared global data structure or file. What this does is restrict the effective number of threads.

This simple system with so many possibilities should make a convincing case, that the only way to make viable predictions about performance is through a model. How to do this, is the subject of the chapters to come. Chapter H will explain the hardware contention and bottleneck calculations. Chapter S describes how to model a single module or task from the ground up, and introduces patterns in sequential software. Chapter C goes into combinations of concurrent tasks, and patterns to enhance concurrency. It is all put together for the Tower pattern in Chapter P.

1.5. How to Use this Report (D.5)

The material in this report is intended to be used in making architectural design decisions at any point in time, during early systems analysis when the entire system is only on paper, during high-level design, for diagnosis of performance problems when significant improvements are being sought, and when planning additions to an existing system.

.....

1.6. Structure of the Report

The report starts from two points, the nature of simple performance problems caused by bottlenecks (which motivates a good deal of our thinking), and how to describe a software system in terms of activities, modules and processes with resource limitations. Chapter 2 describes performance problems from the simpler viewpoint of programs

which use one resource at a time, typically some device, and the effect and symptoms of a bottleneck at one of the devices. Chapter 3 describes the execution within a single sequential process in two complementary and connected ways, both of which are needed to understand performance within design (as discussed for instance in [maps and paths]: a path or scenario view called activity graphs and a structural or design view of interacting sequential modules. Patterns in these two viewpoints are also discussed, along with optimizations which are well-known folklore to many developers. The conceptual framework of Chapter 3 is called “Multilayered Service Systems (Sequential)” or MSS(Seq), in which the layers are essentially layers of procedure calls.

Chapter 4 extends this conceptual framework to concurrent processes executing remote procedure calls, so concurrency and multiple resources for concurrent processes is introduced into the mix. Multi-threaded processes with homogeneous server threads are included in this discussion. This framework is designated MSS(Res), and is rich enough to address many distributed systems while retaining much of the flavour and simplicity of the sequential system analysis. However parallelism within a single scenario is limited to something called a “second phase of service” in Section 4.X.

Chapter 5 uses the notations of these earlier chapters to describe and analyze a variety of architectural patterns that contain concurrency and multi-threaded servers. These are patterns at the level of tasks and their interactions. For some of these patterns some preliminary versions of optimization rules are stated, based on experience to date; this aspect of the work is somewhat preliminary, and is included to encourage further similar work. “Completeness” of this kind of result is in any case illusory, as problems and their solutions will progress over time.

Chapter 6 extends the discussion to systems with parallelism internal to a single scenario, described by parallel paths with forking and joining (which could be implemented by heterogeneous threads forked within the process). These systems are undoubtedly important in the future of design. The main examples in present-day systems occur in parallel transaction processing and in parallel scientific computing.

Chapter 7 focuses on finer-grained patterns within a concurrent system. This is actually more difficult in some ways than the task-level patterns considered before, because the pattern may contain fragments of several tasks. In particular different patterns for inter-task communications are considered here. This work is also incomplete, as it begins to address the host of fine-grained design patterns that have been described or proposed in recent years.

Performance-Oriented Patterns in Software Design (A multi-level service approach)

C.M. Woodside

Dept. of Systems and Computer engineering

Carleton University, Ottawa K1S 5B6

(Draft version produced for classroom use, September 1997)

October 18, 2001

Chapter 2. Software, Hardware, and Bottlenecks (H)

The workflow example in Chapter D showed how performance of a program is determined by the workload it generates for the parts of the computer. It is the amount of work that must be done by each device (CPU, disk drives, CD-ROMs, printers, and network devices) that determines the responsiveness of the program. The most heavily loaded device may act as a system bottleneck, and it turns out that in many cases the program's performance is determined almost entirely by its bottleneck. The bottleneck is important because it points to improvements, in the parts of the software that load up the bottleneck device.

Linear Software

This chapter lays down basic performance definitions and shows how to determine a bottleneck and its effect on performance. It is restricted to sequential programs which execute one operation at a time, which we will term *linear* software. Linear software executes one activity at a time, using one resource at a time. There is no program parallelism in linear software; when executing an operation on a peripheral device or a remote server, it does not execute on its own CPU as well. There are no locks or mutexes or other resources that must be "held" while executing. Linear software is a starting point for our study of distributed software intended to run on networks, which is often "non-linear" in the sense that it may use many devices at once and may have different kinds of logical resources as well as hardware resources.

A program puts a certain amount of load on each device in the system, called the resource demand for the device. These demands are the central factor in bottleneck analysis.

This chapter shows how linear software is described in terms of its resource demands, and how the demand numbers can be obtained. It summarizes the analysis of bottlenecks by *asymptotic bounds*, which are valuable for a quick approximate understanding of the relationship between software, hardware and performance, and it briefly explains more detailed analysis of

device contention effects by queueing models, which are extensively covered in other works. Finally it returns to the software design problem and introduces the design improvement techniques which are dealt with in later chapters.

2.1. Performance Terminology: Directory Server Example

Response to a System Request

Performance is always defined in relation to the completion of some unit of work by a program or a group of programs, and a single completion is called a response. The time from when a request made to perform the work (by clicking on a button, or entering a command) until it is completed is the response time, and the time from completing one request to completing the next is the response cycle time. The rate of completing responses is the throughput of that class of response.

One type of request will be called a *class* of request, and often the user is associated with the class of request he or she is making (so there might be e-mail class users, word-processing users, and compiling users, for instance). If there is just one type of request in the program, or if a set of request types or even a set of programs are being lumped together in the analysis, then there is just one class of request and of user. We can define the average performance measures:

- mean *system* response time *R(c)* sec. for class *c*,
- mean request-creation delay, or "think time" *Z(c)* sec., between the end of one request and the beginning of the next.
- mean *system* response cycle time *C(c)* sec. for class *c*; *C(c) = Z(c) + R(c)*.
- throughput of *f(c)* requests/sec., on average, for class *c*

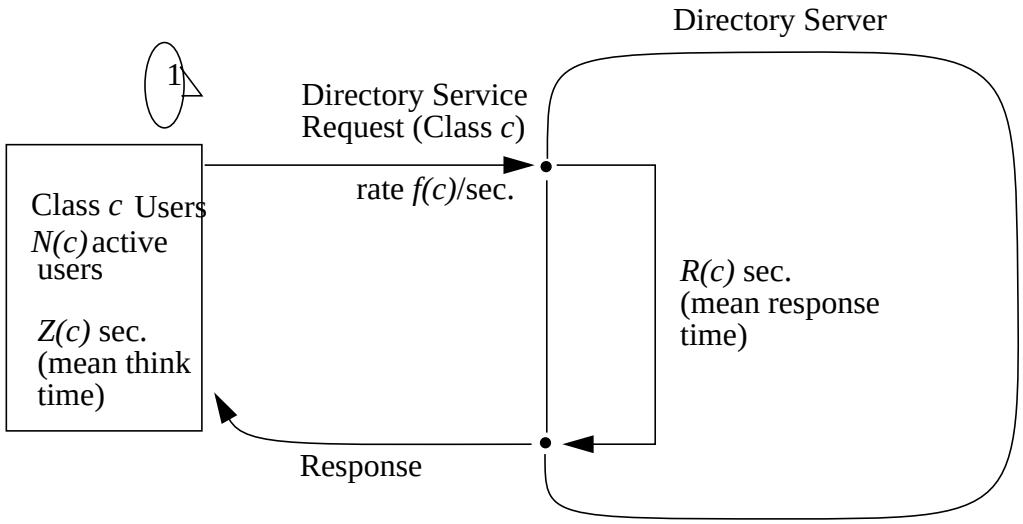


Figure 2.5. Performance Terminology at the User/System Interface (Name Server). (Figure HB)

Figure HB shows an example representing a directory server which receives requests across a network, from users. Each request gives a logical name, such as the name of a printer, and the server retrieves from storage (in main memory or on disk) a physical network address that corresponds to it, to which the user can send print requests directly. In the telephone system there are name servers like this that translate 1-800 numbers into customer phone numbers; in the Network File System (NFS) there is a yellow-pages server to provide addresses for services like printers, and in the internet there are many name servers to determine the routing of messages.

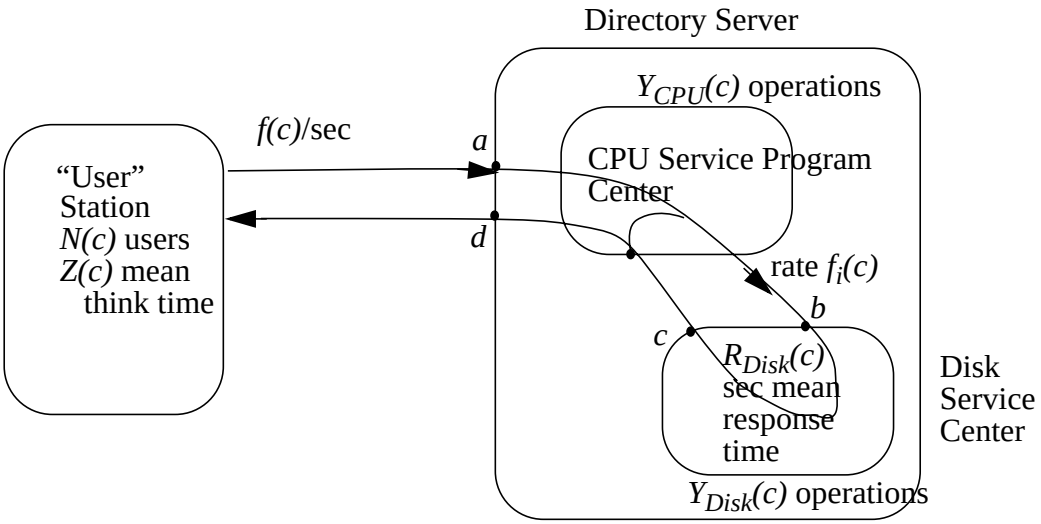


Figure 2.6. Stations in an Abstract Performance Model (Name Server). (Figure HC)

Figure HC shows the system components as abstract stations. The user is replaced by a module which may represent a person making requests, or a program making them. The computer system is opened up to show the devices, and the response is traced through the system by a track showing the sequence of operations in the CPU and the disk. The program code is indicated by a parallelogram inside the symbol for the CPU. We can see that in one execution of the program, a number of $Y_{Disk}(c)$ requests are made to the Disk, and the *disk response time* $R_{Disk}(c)$ is defined as the time from point *b* to *c*.

$R_{Disk}(c)$ = mean response time to one request to device *i*, within a class-*c* operation.

In general for device *i* we define

$Y_i(c)$ = mean number of requests to device *i*, per response of class *c*

The start and end of a response must be exactly defined. In Figure HC a response is defined to begin at point *a* when the request reaches the name server node from the network, and to end at point *d* when the reply message is sent. This is a response from the viewpoint of the designer of the name server, but a user sitting at a remote node would see a different definition,

beginning when he/she originates the request and ending when the result is returned. In yet another case, it might not be a human user but a program at the remote node which originates the request and receives the reply.

If we compare this model to the Workflow Server example in Figure DG we see that the Users correspond to the Clients, the Workflow Server corresponds to the Directory Server, and the CPU and Disk have the same roles as in the earlier examples.

One Device: Response and Demand

The view of performance at a single device is much like that for the program as a whole. For example the disk device shown in more detail in Figure HD serves requests to read and write blocks, and it has a response time of its own. We will assume that a device has only one operation

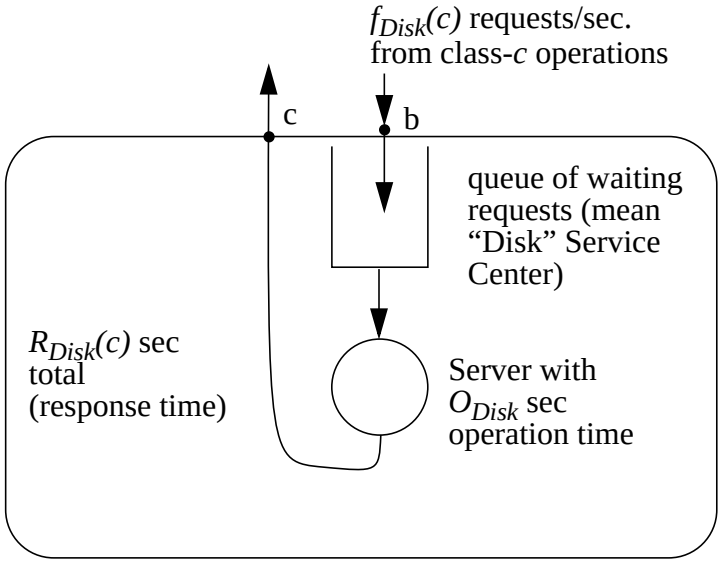


Figure 2.7. In a Model, a Device is a Queue and a Server. (Figure HD)

(this can easily be relaxed). The properties of a service by a device will be identified by subscripts, so device i has a throughput f_i, a response time R_i, etc. In Figure HC or HD the disk response time is the time to traverse from point b to point c. Also a device has a characteristic time to execute an operation, and a utilization level.

- $f_i(c)$ = rate of requests to device i by class-c jobs
- O_i = mean operation time for an operation by device i, in sec., in a class-c operation
- $U_i(c)$ = mean utilization of device i by class-c jobs, $U_i(c) = f_i(c)O_i(c)$. This is the mean fraction of time the device is busy.
- U_i = mean total utilization of device i. $U_i = f_i\, O_i$, a fraction $U_i = \Sigma_c U_i(c)$
- ($U_i \leq 1.0$, if device i is a single device; $U_i \leq m_i$ if it includes m_i identical subdevices, for example a multiprocessor with m_i CPUs.)
- $n_i(c)$ = mean number of class-c jobs using and waiting for device i.

The *demand* by workload class *c* on one device is the average total execution time needed to complete a class-*c* system response. Using $Y_i(c)$ and $O_i(c)$, the average resource demands per response are:

• average service demand $D_i(c) = Y_i(c)\ O_i(c)$ sec. per response of class *c*.

The resource demands describe the workload of class *c* on the hardware, and from them a great deal can be determined about *R(c), C(c)* and $U_i(c)$ for each device.

To illustrate the calculation of demands, consider the directory server example with one class (*c*=1), with a *CPU* of operation time 100 nanoseconds and a disk of operation time 18 milliseconds.

$O_{CPU}(1) = 10^{-7}$ sec; $O_{Disk} = 0.018$ sec

Also suppose the program is such that it demands half a million operations on the *CPU* and an average of 1.7 operations by the disk. Then

$Y_{CPU}(1) = 500{,}000$; $Y_{Disk} = 1.7$ requests/response.

$D_{CPU}(1) = 0.05$ sec/response; $D_{Disk} = 0.0306$ sec/response.

This is a very aggregated, abstract workload description It does not describe the order of operations, or the probability of a single response requiring 1, 2 or even 5 disk operations. However (as we will see) it is enough to *bound* the system capacity and response time, for a system with linear software.

The values for CPU operations are often numerically awkward, with very small operation times and very large numbers of operations. One visit by a job to the CPU queue (the ready-to-run queue) leads to many operations before the job give up the CPU. Sometimes we will use millions of instructions for a CPU workload.

For the directory server example there is just one class of response (class 1). The devices are a CPU and a DISK. The CPU has an operation time of (say) 10^{-7} sec., and the disk has an operation time of 0.18 sec. Suppose there is an average of 1.7 disk operations, the CPU time per response is 500,000 operations. Then

• $Y_{CPU}(1) = 500{,}000$, $D_{CPU}(1) = 0.05$ sec.

• $Y_{Disk}(1) = 1.7$, $D_{Disk}(1) = 0.0306$ sec.

This does not state in what order the operations are carried out, or the probability distribution of either the number of operations or the time per operation. However for each response, it states that on average, a certain number of seconds of work must be done while tracing the path of the response. This is just the operation time of the devices and does not include waiting for a device to become available, so the time to actually do the operation may be greater. In linear software this work must be done in some sequential order.

Users

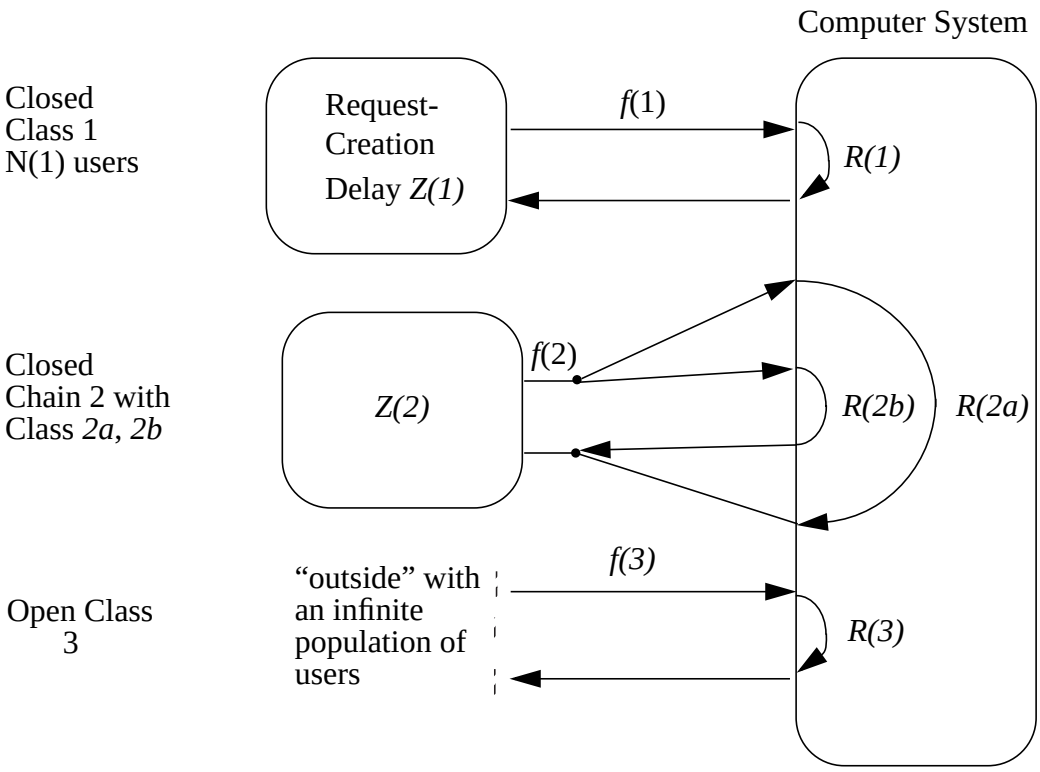


Figure 2.8. Users, Requests, and Open and Closed Classes. (Figure HE)

The users generating class-*c* responses are said to be themselves in class *c*, and there are *N(c)* of them. Often we know *N(c)* and it also is a parameter of the workload for the class. This makes *c* a "closed" class in the analysis. For a closed class it is important that we know the mean request-creation delay or "think time" *Z(c)*, which is the delay of a "User" station in the model. We will use the terms:

- population *N(c)* of a closed class *c*,
- mean number $n_i(c)$ of class-*c* requests are using and waiting for device *i*

When there are several classes there may be one total population *N* which is creating requests of several classes, in certain given proportions; then the group of classes is called a "chain" (we will not have to go into details, but this has the effect of averaging the classes together, and considering the chain itself as a class). Thus sometimes a model may have a "chain *c*" in place of a "class *c*".

Sometimes we only know that *N(c)* is very large, and we prefer to ignore its actual value, and instead assume that we know the rate of creation of requests by the users, *f(c)*. Such a class is called an "open" class, since we are only sure of the requests actually in the system at any time, and they arrive and leave. In theory the mean number of requests being processed could go to infinity! We have, for open systems:

- mean population *n(c)* of class-*c* requests in the system.
- mean number $n_i(c)$ of class-*c* requests are using and waiting for device *i*.

Response Measures

All the above deals with average values of performance measures, $R(c)$, $R_i(c)$, $n_i(c)$. They are the simplest figures to use, but it is sometimes necessary to look at the variance of some measures as well, or the distribution. For example one may need to know the percentiles of the response time. A telephone switch must give dial tone within half a second, in 99% of requests for connection, for instance.

Consider the response time, and let $\tilde{R}(c)$ stand for the actual random "response time" quantity that has average value $R(c)$. The dial tone requirement could be written as "the probability that $\tilde{R} < 0.5$ sec for a "request-for-connection" class system operation must be at least 0.99", or:

$$\text{Prob}\{\tilde{R}\,(\text{request-for-connection}) < 0.5 \text{ sec}\} > 0.99$$

A percentile specification on R can be abbreviated as R_{95} or R_{99}, or more generally as R_α, which is a value of R such that

$$\text{Prob}\{\tilde{R} < R_\alpha\} > \alpha/100$$

Then for dial tone we have $R_{99} = 0.5$ sec. Figure HF illustrates $\tilde{R}$ and R_{95}.

For queue lengths we may also have restrictions, due for instance to finite storage space. In message-based systems, when message buffers overflow it is sometimes necessary to discard messages, and the subsequent recovery operations cause loss of performance. So we may need to know the probability that the queue lengths exceed a limit.

Unfortunately the analysis of distributions is much more complex than the analysis of averages. It almost always requires a detailed simulation or measurement study of the system. The simple performance bounds analysis described below only applies to averages. However, many response and queue distributions in computer and communications systems are approximately exponential. Then the percentiles are roughly proportional to the mean, with the form $R_x = \ln(1-x)R$, which gives

$$R_{95} \cong 3R$$

$$R_{99} \cong 4.6R$$

Using this fact an analysis of the mean gives at least an indication of percentile values, which should be confirmed by more detailed analysis.

2.2. Obtaining Demand Parameters

Demand values for a program which is implemented and running may be obtained from measurements made by the operating system, or by instrumenting the software. Many operating systems record the CPU consumption of each process during a certain interval which might be

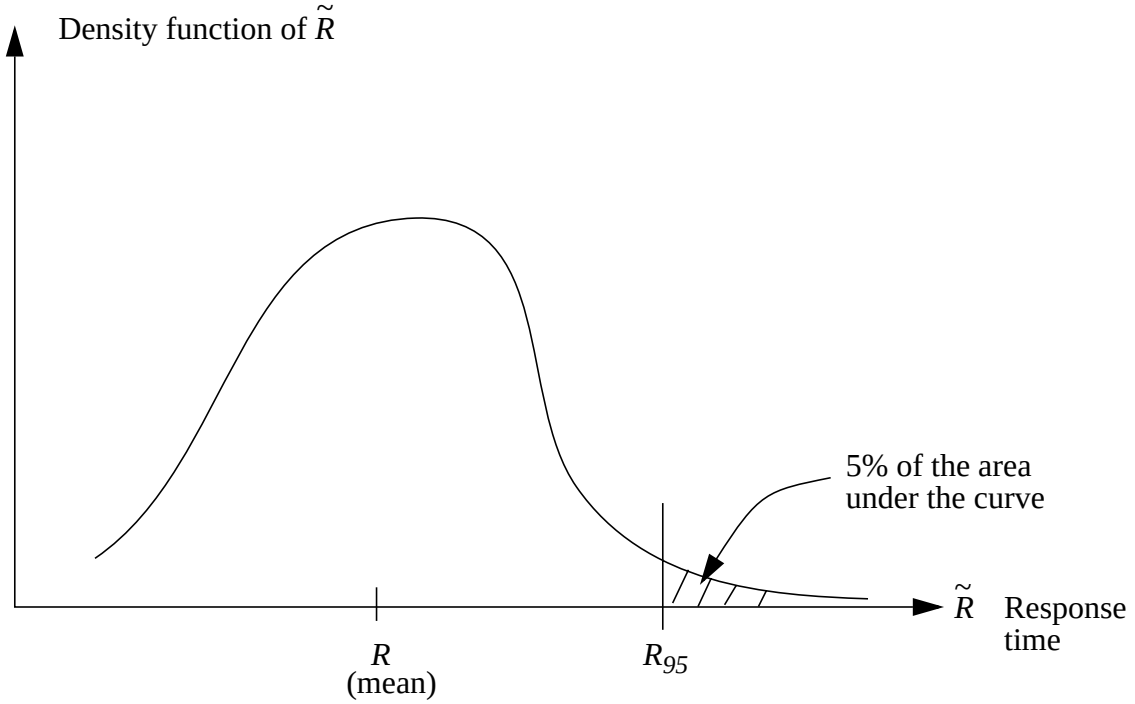


Figure 2.9. Distributions and Percentiles of Response Times.
(Figure HF)

several minutes. If the number of responses for a certain class of users is also known over the same interval, then the CPU demand per response is directly obtainable. The modeller may wish to group into a single class a number of users running different programs, in which case the correct procedure is to add up the CPU consumption for the different programs and users, and divide by the total responses. Care must be taken about responses at the edges of the interval.

Similarly the operating system records the number of I/Os, although special instrumentation may be needed to record which disk they go to. Indeed these days they often go to a file server, which would have to be instrumented also.

A more extensive discussion of this question is found in [jain92], chapters 7 and 8, and in [menasce94], chapter 9.

Notice that demand measurement needed for bottleneck analysis and performance predictions is different from measuring performance values themselves, such as R and f. With measured demands and a model, performance values can be extrapolated for larger numbers of users, faster devices or more devices and so on, while for an experimental performance evaluation the results apply only to the measured configuration.

There are many measurement tools which are intended not to measure demands, but to measure performance values themselves. The approach taken here is intended to *supplement* these measurements by starting early and by using models. It is not intended as a replacement for measurement.

Some metrics are easier to measure than others. Utilizations and throughput counts are relatively easy and are measured by standard operating system tools such as *sar, iostat* and in UNIX.

2.3. Basic Performance Bounds for Linear Software

Given the very simple information defining the resource demands and the number of users, one can obtain useful quick information about the potential of the system for performance, from bounds. The Workflow Server example in Section 1.4 described bounds due to saturation, and now this section derives their equations and gives typical diagrams for single-class systems, and for two classes. These bounds are described in every reference on performance modelling, so only a brief summary will be given here. The bounds are based on two observations:

- no device can be more than 100% utilized: this creates a limit on throughput,
- no response time can be less than the amount of work that must be done: this creates a limit on response time.

Asymptotic bounds are optimistic values of the mean throughput and response times that cannot be actually achieved except in asymptotically large or small systems. It is amazing how many questions can be answered by asymptotic bound calculations.

2.3.1. Saturation Bounds

The throughput of a response cannot be increased beyond a rate that saturates some device, i.e. makes its utilization 100%. This means, if only one class c uses the system, and all devices are single servers, the system throughput is bounded by

$U_i \le f(c)D_i(c) \le 1.0$ for each device i (all single servers)

which limits the throughput to the range

$$f(c) \le f_{max}(c) = min_i 1/D_i(c) = 1/D_{max}(c) \qquad (1a)$$

where $D_{max}(c) = max_i \ D_i(c)$.

A closed system with one class is self-limiting at f_{max} since any attempt to overload it only leads to longer queues and delays at the bottleneck device (the one with largest D_i). An open system will “blow up” for a rate of arrivals larger than f_{max}, and in practice such systems have to limit their arrivals by rejecting some of them.

When there are several classes then saturation arises due to the total load at a device. That is, at device i the throughputs are bounded by

$$U_i = \sum_c f(c)D_i(c) \le 1.0 \qquad (1b)$$

2.3.2. Path Bounds

For a closed system a response also cannot be repeated faster than allowed by the time it takes its user to complete a cycle of user operations (such as thinking and typing) and the computer operations to complete the response. This means that for linear software with only single servers,

$$R(c) \geq D(c) = \sum_i D_i(c)$$

The user think/type delay $Z_{User}(c)$ is not included in $R(c)$, but is a part of the total cycle time:

$$C(c) \geq D(c) + Z_{User}(c) \ .$$

If the response time includes some other pure delay terms that have a constant mean (i.e. no contention) they will also be included in $R(c)$. Let us denote their sum as $Z_{Sys}(c)$ sec., then

$$R(c) \geq D(c) + Z_{Sys}(c)$$
$$C(c) \geq D(c) + Z_{Sys}(c) + Z_{User}(c) = D(c) + Z(c) \qquad (2)$$

We note that delays such as Z_{Sys} do not limit throughput directly, by saturation. Typically, they do not represent device demands, but are nominal figures used to represent delays in some subsystem which is not being analyzed, such as a transport delay through a network.

2.3.3. Little's Formula for Delays and Throughputs

There is a very powerful and simple relationship between flows and delays in all kinds of systems, illustrated in Figure HJ.

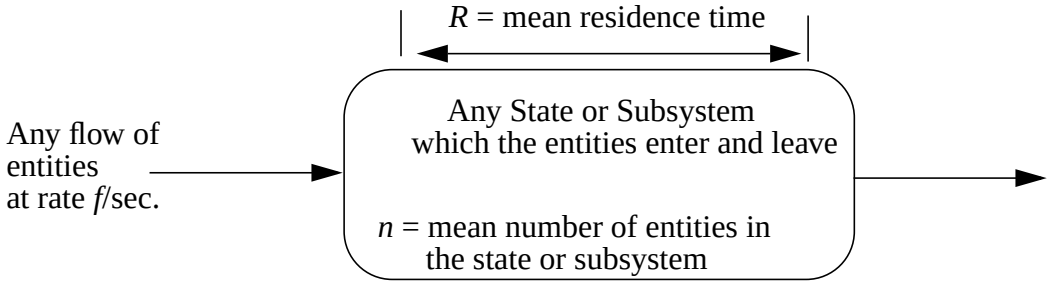


Figure 2.10. Little's Formula for Flows and Delays. (Figure HJ)

If

f = mean flow rate into and out of the subsystem

n = mean number of entities

R = mean residence time in the subsystem

then the result, known as Little's Formula, is

$$n = fR \qquad (3)$$.

The "subsystem" can be chosen arbitrarily: it could be an entire computer system, one device in it, or just one queue of waiting jobs at one device. If we know *f* and *R* we can derive *n*, or if we know *n* and *R* we can derive *f*, and so on.

For our closed computer system model the "subsystem" is an entire response cycle, giving:

$$N(c) = f(c)C(c) \qquad (4)$$

2.3.4. Asymptotic (Optimistic) Bounds: Summary

For a closed system we can use Eq. 4 to put the bounds (Eq. 1) and (Eq.2). Taking Eq.1 and substituting *N(c)/C(c)* for *f(c)* gives

$$N(c)/C(c) \le 1/D_{max}(c)$$
$$C(c) \ge N(c)D_{max}(c)$$
$$R(c) \ge N(c)D_{max}(c) - Z_{User}(c) \qquad (5)$$

and from Eq.2, substituting *C(c) = N(c)/f(c)* (by Little's result)

$$N(c)/f(c) \ge D(c) + Z_{Sys}(c) + Z_{User}(c)$$
$$f(c) \le N(c)/[D(c) + Z_{Sys}(c) + Z_{User}(c)] \qquad (6)$$

Figures HK and HL summarize these bounds for a single class system for the Directory Server example of Figure HC with the parameter values of Table #T. Notice that the two bounds in each diagram intersect at a point *N*= 201.6* representing a user population size at which contention delays become significant. *N** divides cases with negligible contention, from those where contention is an important factor.

Table 1: Directory Server Example: Parameters (Table #T)

	Demand D_i (sec) or Z_i (sec)
CPU	0.05
Disk	0.0306
User Think	10

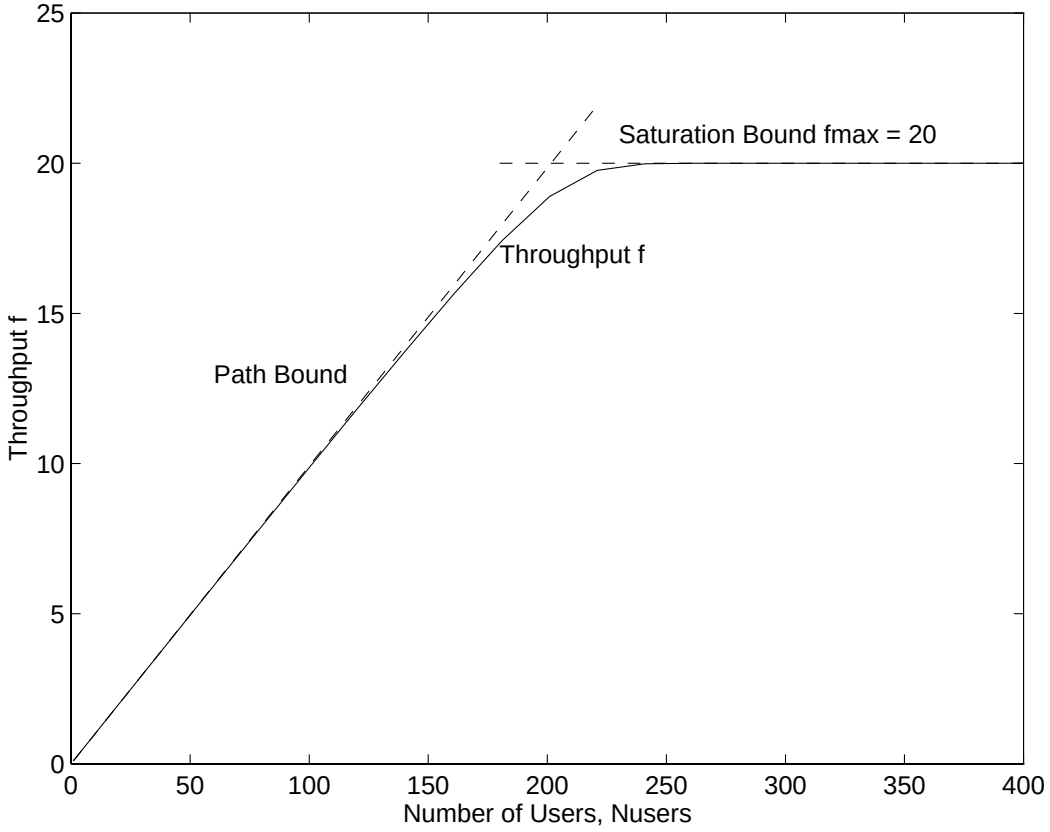


Figure 2.11. Asymptotic Bounds on Throughput, for the Directory Server (Figure HK)

2.3.5. Two-Class Example: Asymptotic Bounds for a Reservations System

A more complex example is described by Figure HM10. It represents a Web Server which handles inquiries for theater and concert tickets. A user makes a series of inquiries to retrieve pages with information about show dates, ticket availability and price, and may also purchase tickets. The server uses two sets of files on two separate disks, one with the reservation information and one for marketing information. A special “reference” station is added to the model to mark the end of a session, and there are eight requests per session. A second class of users, class 2, uses only the CPU and Disk B. Demands are shown per User request for both classes.

New features introduced in this model are, an additional station with a pure delay for Credit Card authorization, a second class of users, and a reference point for throughput which is not a User station.

If a user makes a purchase, the credit-card information is submitted for authorization by the card issuer. The credit card company’s computer system introduces a delay which is included as a simple mean delay Z_{CCReq}, which provides a non-zero value for Z_{Sys} in this example.

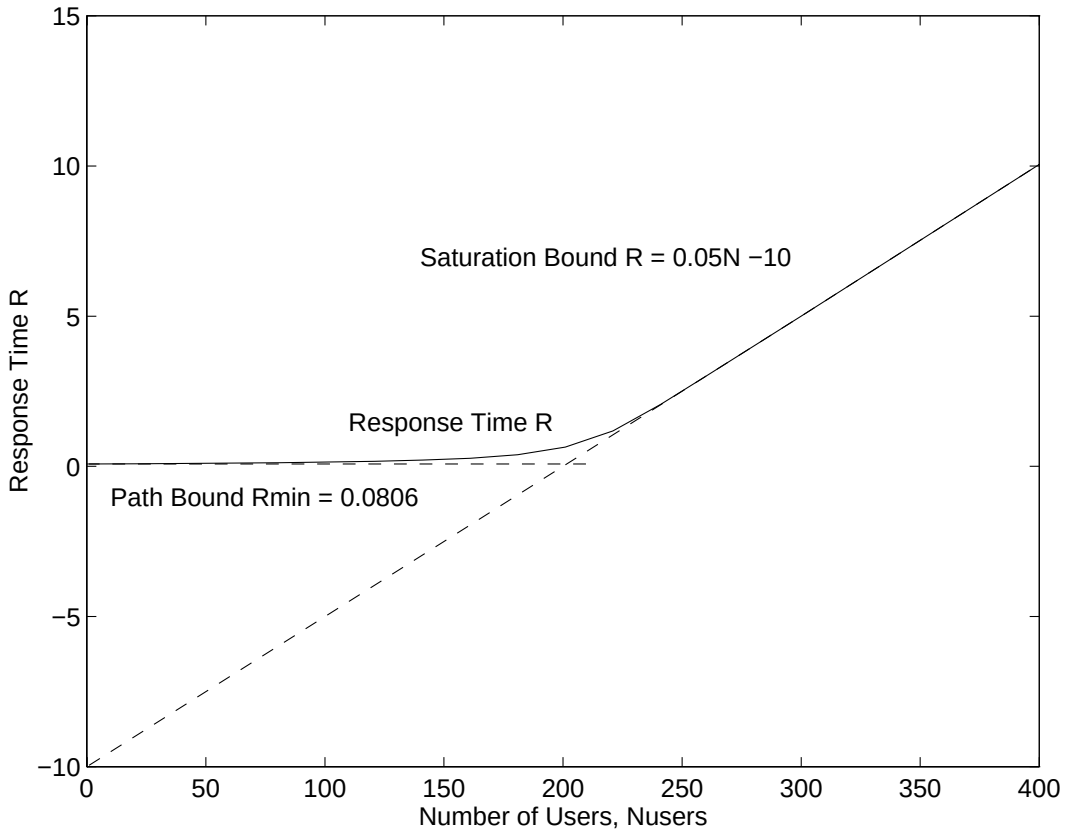


Figure 2.12. Asymptotic Bounds on Response Time, for the Directory Server (Figure HL)

The demands for class 1 in Figure HM10 are worked out in the next chapter, for an entire session with an average of 8 user requests in a session. In the Figure here, the values are divided by 8 to obtain average values per request.

It is normal for a distributed system to be shared by different communities of users that run different programs and impose quite different workloads. Figure HM10 shows these as a separate class called “Other Users”. As long as they all run linear software, their workloads are fully represented by their separate demands, as shown in Figure HM10. Notice how class 2 is different from class 1; it makes heavier CPU demands, and lighter Disk demands, and has a longer “think” time Z_{User}.

The bound values derived in the last section give separate path bounds for each class, and one composite saturation bound for each device. The total demand and total pure delay values are:

D(1) = 0.0124 + 0.132 + 0.174 = 0.318

Z(1) = 7 + 0.225 = 7.225

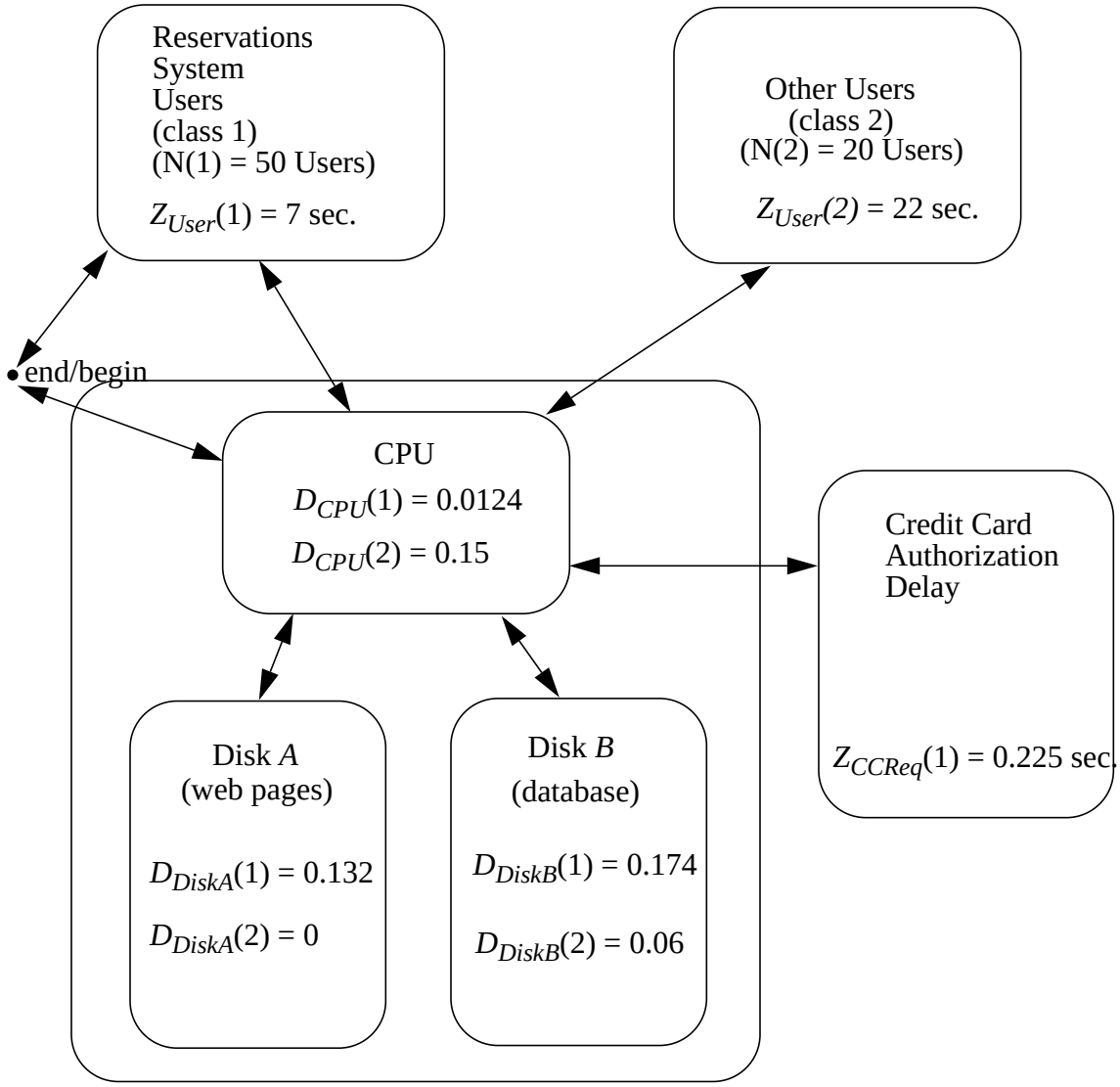


Figure 2.13. Two Classes of Workload. Demands are in sec/response. (Figure HM10)

Class 1 path bounds from Eq. (1a) and (2):

$C(1) \geq 0.318 + 7.225 = 7.543$ sec

$R(1) \geq 7.543 - 7 = 0.543$

$f(1) \leq 50/(0.318 + 7.225) = 6.63$/sec

Class 2 path bounds from Eq. (1a) and (2):

$C(2) \geq 0.21 + 22 = 22.21$ sec

$R(2) \geq 0.21$ sec

$f(2) \leq 20/22.21 = 0.90$/sec

Device composite bounds on throughput, from Eq. (1b):

CPU: $0.0124\, f(1) + 0.15\, f(2) \leq 1.0$

DiskA: $0.132\, f(1) \leq 1.0$

DiskB: $0.174\, f(1) + 0.06\, f(2) \leq 1.0$

The throughput bounds are plotted in Figure HP1.

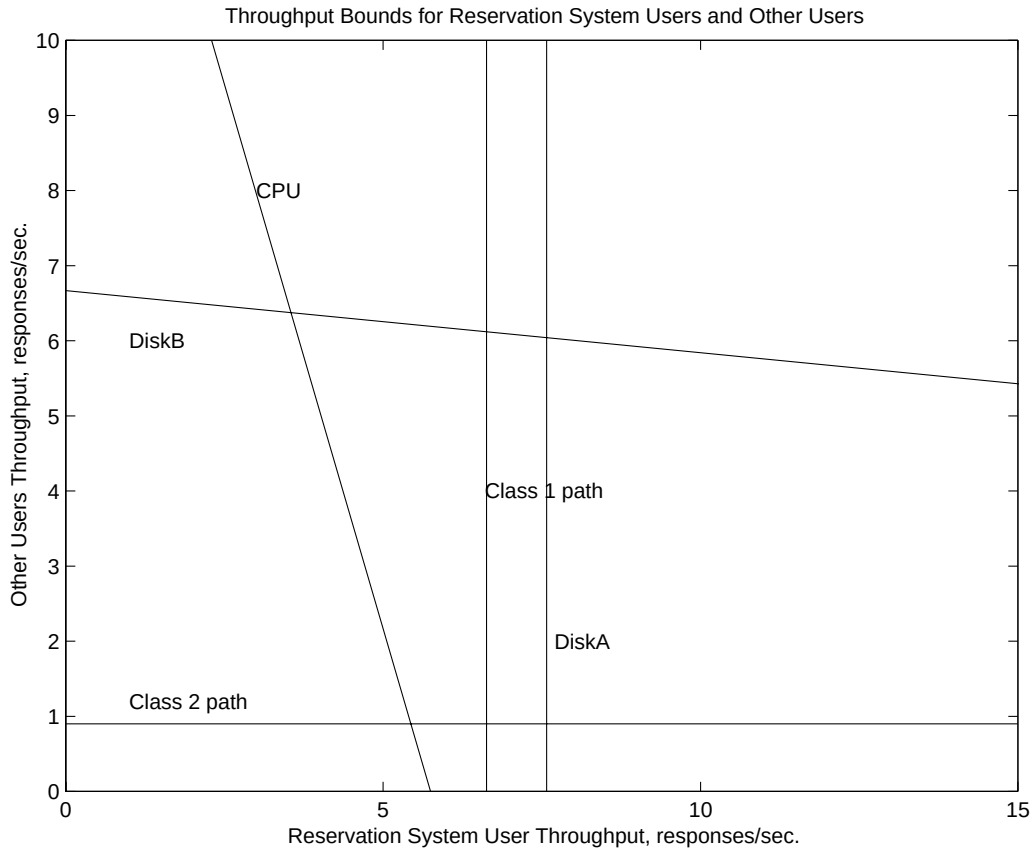


Figure 2.14. Asymptotic Throughput Bounds for Reservations System Users and Other Users. (Fig. HP1)

2.3.6. Other Bounds

Further bounds can be found. For example, because one cannot do worse than wait for every competing job at every station, one obtains pessimistic bounds on delay and throughput:

$$C(c) \le N(c)D(c) + Z_{User} + Z_{Sys}$$
$$f(c) \ge N(c) / [N(c)D(c) + Z_{User} + Z_{Sys}]$$

With stronger mathematical assumptions about the nature of the workload there are also tighter bounds called “Balanced Job Bounds” [BJB82].

2.4. Contention and Queueing Models

The bounds discussed so far do not predict the effect of interference, and this is the role of a queueing model. Queueing models were used to predict the “actual” values represented by solid lines in Figures HK and HL, and the point X in Figure HP1. Given a set of demands, one can compute response time and throughputs by a queueing model. Most solvers make assumptions, however, and the assumptions may not be accurate.

This work is going to sidestep the subject of solving queueing models, as they have been well documented. One does need a model, rather than just bounds. For example in Figure HP1 the bounds only delineate a feasible region, they do not suggest exactly where the throughputs will lie within it. However in this work we will concentrate on deriving models, leaving the mathematics of solving them to one side. To study solution techniques consider Menasce [menasce94], Harrison [Harrison93], Walrand [walrand].

2.5. Software Design Options

The performance engineer has many ways of modifying software to make it perform better. Staying within the bounds of linear software, these all boil down to reducing the demands D in some way. Either the number of operations requested by the program must be reduced, or the average time per operation (which may be affected by cache efficiency, for instance). If it is not necessary to stay with linear software, other mechanisms such as parallel execution may be used.

2.5.1. Reducing the Operation Counts

Some approaches for reducing the number of operations are summarized here. Some these have been summarized by Smith in a number of principles culled from good practice, but I have reorganized and renamed them.

Attack the Bottleneck. Give first and most attention to reducing the number of operations at the bottleneck device. If it is a disk, try to reduce the amount of data stored, or use a more compact format, or store more data in memory to avoid re-reading it later.

Batching is a well-known way to reduce operation counts by grouping operations together. The operations themselves are not reduced but the overhead to transfer data and initialize the algorithm can often be performed just once per group.

Early binding (“fixing principle” of Smith [CUS90]). Late binding is introduced for flexibility and to hide complexity, in the form of pointers, symbolic addresses, inherited functionality, procedures, and many others. Late binding always has overhead to determine the run-time value of the binding, and sometimes the overhead is surprisingly large. In hiding complexity late binding also hides from the designer the knowledge of the run-time costs involved in an operation.

Well-known examples of early binding for performance improvement are

- allocating static memory for storing data, instead of allocating as needed,
- unwinding loops,
- flattening an inheritance hierarchy,
- initializing a fixed buffer pool or task pool, instead of allocating them as needed.

High Runners (“centering” principle of Smith). Concentrate effort on sections of code for which the total demand is high; this is particularly productive when a very small section of code is heavily used. Profiling tools commonly identify these sections as those in which the program spends a high percentage of its time. The actual optimizations do not follow any general principles, but folklore insists that one can always reduce a set of operations by spending effort.

Special Cases or *Fast Path* ([CUS90]). This is a version of high runners, in which a special case can be determined by a test and then processed in an especially simple way. For instance, the special case may need no processing at all, or its result may always be zero or empty.

Locality: store data in some sense “near” the operations on it, to reduce the overhead of accessing and modifying it. The trend to objects which store data and operations together reinforces this advice.

Scrutinize algorithms. Algorithms are often used out of habit, and may have efficiency trade-offs according to the size of data in the application. Look at the alternative algorithms and evaluate them for the particular program being designed.

Optimistic design. This is a variation on Special Cases, in which the test for the special case can only be done after trying to perform it. The special simple processing is done first, on the optimistic assumption that it applies to the case, and the test is done after. If the assumption turns out not to apply, then a more elaborate version of the processing is done after undoing any side effects of the optimistic step. Hash table storage is a widely used example, in which the optimistic assumption is that there will be no collision on the hashed address.

Good ideas for efficient programming are contained in the books *Programming Pearls*, [pearls1] *More Programming Pearls* [pearls2], and *Writing Efficient Programs*, by Jon Bentley [bentley?].

2.5.2. Reducing the Operation Times through Locality

With modern processors and operating systems the times for operations at the programming level is often reduced by caching and virtual memory (memory hierarchy effects). Programs can be modified to reduce their operation times by improving their locality of reference, which in turn increases the efficiency of the memory hierarchy. For instance a repetitive execution of a short section of code will run completely in the cache, whereas if it includes many procedure calls to different locations in the program memory the cache may have to be reloaded for every transfer. Similarly a large number of operations on a small block of data may execute with all the data in cache, where a series of traversals over a large data array causes the data to be reloaded on each pass. Stepping through a data array on elements stored adjacently in memory causes fewer reloads than jumping back and forth in memory.

Anything which applies to processor caches also applies, at a larger granularity, to virtual memory, and to file caches. A program which uses a small working set of pages at a time will make fewer demands for paging operations (which are otherwise invisible to the programmer, but which must be included in our performance models) and will run faster. If it uses a small set of disk pages for data they may be retained in the file system cache of file pages, and they can be read and written much faster than if physical disk operations are needed.

The effect of the memory hierarchy on operation times can be dramatic, on the order of 10 to 1 at both the processor cache and the virtual memory/file cache level. Rules for enhancing locality can be described, and tools such as optimizing compilers are available to help, but in general the results are not predictable by simple models.

This work will assume that a certain level of effort is made in this area, with target values for efficiency which are then incorporated in the demand values for operations. That is, there will be a translation factor between the operations the programmer sees, and the device operation demands, which will account for a baseline level of efficiency in the memory hierarchy. For example, one file system operation will be converted into a certain average number (perhaps less than one) of disk device operations, corresponding to the baseline efficiency of caching.

2.5.3. Improvements by Restructured Software

Beyond simply reducing operation counts and times, restructuring a program can yield performance dividends.

- *Structural aggregation*: Remove the overhead of context changes by bringing modules together. This can include in-lining of code that might otherwise be a procedure call, or placing objects in the same process that might otherwise be pipelined.
- *Distribution*: Move load away from a bottleneck by placing some services on a remote server accessed by a remote procedure call. This move, prompted by resource saturation (important in heavy load), is opposite to the previous which is prompted by reduced total demand (important in light or moderate load).
- *Specialization*: Separate service requests into groups handled by efficient specialized servers. This can combine “Special Cases” for work reduction with “Distribution”. It is the basis for multi-tier Client-Server systems.

2.5.4. Improvements which lead to Non-Linear Software

This chapter has been concerned with linear software, that executes sequentially and uses just one resource at a time. At lower levels in the system hardware and software, parallel execution and simultaneous resources are often used (pipelining, overlapped I/O, mutexes) in ways that the application designer does not see. The visible architecture and design can also use these explicitly.

Evaluating the use of simultaneous resources, concurrency and parallelism is the subject of the remaining chapters. The opportunities for better performance include

- *Parallel processing*: Execute some work in parallel on separate devices. This includes overlapped I/O, delayed writes and commits, as well as parallel subtransactions and multicasts.

- *Multiply Resources*: Provide additional "copies" of a logical resource so that competing tasks can proceed together. This includes multithreaded or asynchronous servers and replicated servers (as in multicopy databases).
- *Resource Restructuring*: Re structure the sequence in which resource are obtained, to combine some resources. For example in run-to-completion systems a table has exclusive control of all resources controlled by the scheduler, which may make a critical section unnecessary.

2.6. Summary of Chapter H

This chapter described the basics of workload modelling at the level of execution on the system as a whole, in terms of classes of users and their demand in devices. It showed how simple optimistic bounds can be obtained from a small amount of workload data. It briefly described alternative ways to reduce the demands made by programs.

All this is meant as an introduction to procedures for breaking down workload module by module, in the next chapter, and for describing and evaluating "non-linear" software, in the chapters after that.

2.7. Related Reading

Modelling by queueing networks is well suited for linear software and is treated in many excellent papers and books. Recent examples are

R. Jain [jain92], which has special strength in analysis of data and results, and experimental design, but also includes a summary of modelling by queueing networks.

Menasce, Almeida and Dowdy [menasce94], which describes models for systems running on networks, and a blend of approaches including queueing.

Allen [allenProb], which is mostly on the theoretical issues underlying the queueing models, but has a chapter on performance modelling.

BIBLIOGRAPHY

[CUS90] C.U. Smith, "Performance Engineering of Software Systems", Addison-Wesley, 1990

[menasce94] D.A. Menasce, V.A.F. Almeida and L.W. Dowdy, "Capacity Planning and Performance Modeling", Prentice Hall PTR, Englewood Cliffs, New Jersey 07632, 1994.

[jain92] R. Jain, "The Art of Computer Systems Performance Analysis", John Wiley & Sons Inc., 1991.

[allenProb] A.O. Allen, "Probability, Statistics, and Queueing Theory with Computer Science Applications", Academic Press, Inc., 1990.

[Harrison93] P.G. Harrison and N.M. Patel, "Performance Modelling of Communication Networks and Computer Architectures", Addison-Wesley, International Computer Science Series, 1993.

[HOPEsem95] C. Cowan and H. Lutfiyya, “Formal Semantics for Expressing Optimism: the Meaning of HOPE”, in Proc. of the 14th Annual ACM Symposium on Principles of Distributed Computing”, Ottawa, Canada, pp. 164-173, August, 1995.

[bubenik] R. Bubenik and W. Zwaenepoel, “Semantics of Optimistic Computation”, in Proc. of the 10th International Conference on Distributed Computing Systems, pp. 20-27, 1990.

[stromyemini] R.E. Strom and S. Yemini, “Optimistic Recovery in Distributed Systems”, ACM Transactions on Computer Systems, pp. 204-226, v3, n3, August, 1985.

[pearls1,] J. Bentley, “Programming Pearls”, Addison-Wesley Publishing Company, 1989.

[pearls2] J. Bentley, “More Programming Pearls”, Addison-Wesley Publishing Company, 1990.

[bentley?] J. Bentley, “Writing Efficient Programs”, publisher, year?

[walrand] J. Walrand, “An Introduction to Queueing Networks”, Prentice Hall, Englewood Cliffs, New Jersey 07632, 1988.

Performance-Oriented Patterns in Software Design (A multi-level service approach)

C.M. Woodside

Dept. of Systems and Computer engineering

Carleton University, Ottawa K1S 5B6

(Draft version produced for classroom use, September 1997)

October 18, 2001

Chapter 1. Tracing Performance to Software Scenarios and Modules (S)

1.1. Introduction

The previous chapter showed how device bottlenecks can be traced back to users executing a certain program, which was represented there as a workload class. The connection is given by performance parameters, mainly the demand of each program for each hardware resource, per response of the program. Each program may have several users. Only linear software (which does one thing at a time) was covered, and only physical resources (devices used one at a time).

This chapter describes the links between the software design and the performance parameters. It focuses on analyzing one particular software module, and the execution of one particular type of response. The designer can think in the software domain and about the particular application, and determine the parameters that will give performance figures; the notation and some simple associated data reductions provide the links. An important feature of the software description that is not evident in the hardware models of the previous chapter is the need to use abstraction to hide detail in the software. An activity may be made up of many other more detailed activities, or a module may be broken down into submodules, and it is essential in considering software of any complexity to be able to do this, and to reason about it at a level that addresses the designer's concerns.

Faced with a set of programs, or designs for programs, how can we extract the essential information for predicting performance? How can we understand the performance issues well enough to improve the software? This chapter uses two descriptions, the first based on behaviour described by scenarios, and the second based on modules. Scenarios will describe the internals and

the flow possibilities of modules, so we will use scenarios to give the details and modules to summarize them. In both cases a queueing model will be derived similar to those seen in the last chapter. These models are lacking various advanced features such as models of critical sections or parallelism, which are addressed in the next chapter.

This chapter also considers scenarios with parallel paths, a first step to modelling nonlinear software.

1.1.1. Scenarios, Use Cases and Activity Graphs

Scenarios describe the processing paths of the system, and are sometimes written down as part of the analysis of a new design or a new feature. An example of a popular scenario capture technique is Jacobson's Use Cases [OOSE], about which Jacobson says:

"Quote..."

Typically one needs several (or many) Use Cases to describe all the functions of a system, and for exceptions. For performance analysis it is permissible to focus on just a few scenarios, which are known to contribute most of the workload, and to ignore those that are seldom performed, or only performed to handle exceptions. Which scenarios to include is a matter of judgement; some kinds of exception handling may occur often enough to demand inclusion.

A Use Case describes a system response in a narrative form that can be easily related to the requirements and the user's view of the system, so it provides an interface between the high-level understanding of a performance problem, and the technical model. A Use Case is a narrative with a sequence of steps, possibly including alternatives that may occur at some steps, and identifying as Actors the participants which are outside the software to be designed.

Theatre Ticket-Reservation System

For example the theatre-ticket reservation system already described as a queueing model has a major Use Case for connecting, interacting and purchasing tickets. It involves Actors which are outside the software to be designed (but may be in the performance model): a User, a database server DB, and a credit-card verification system CCReq. A simple high-level narrative form of the Use Case is:

- a user issues a request to connect to the reservation system, which is processed to set up data tables and connection parameters (detail suppressed here).
- The system presents the user with a menu of choices to
  - *Display*: display program and price information, or ticket availability information,
  - *Reserve*: define a reservation for tickets, to be paid for by a credit card,
  - *Confirm*: confirm a reservation, a step included to allow the user to have second thoughts and cancel the transaction,
  - *Disconnect.*
- The system presents the appropriate operating window for one of the three choices and the user fills it out, possibly returning to the menu for another choice or a more detailed window within the same category.
- If the choice is Display or Reserve, the operation will include read accesses to the ticket database to provide information,

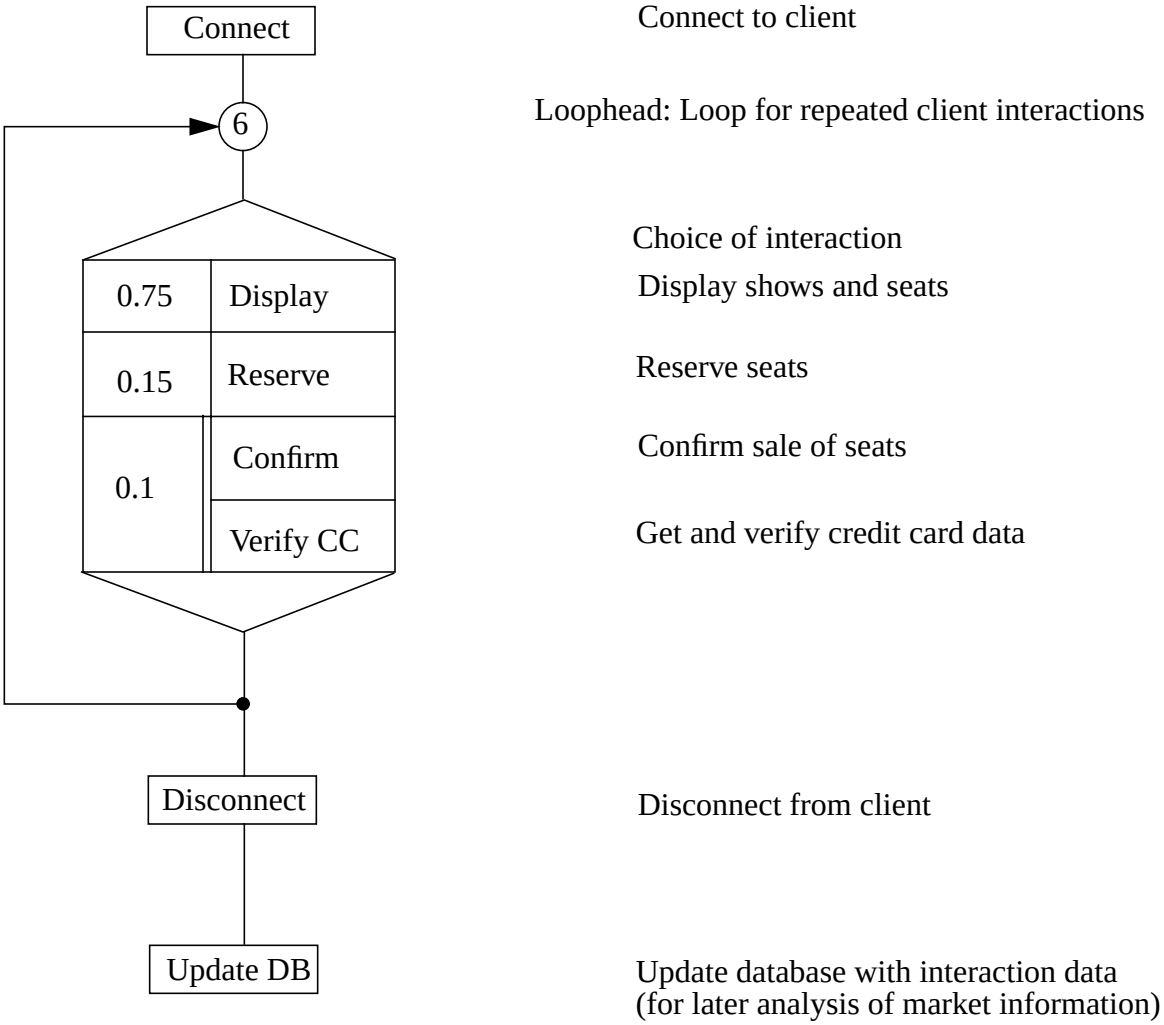


Figure 3.1. Example of an Activity Graph for a Ticket Reservations System. (Figure SA)

- If the choice is Confirm and the user does confirm a purchase and give credit card information, a request to verify the credit card information is sent to the appropriate credit card issuer's system (represented here by a single actor CCReq for simplicity), and the ticket database (actor DB) is updated with the sale,
- when the choice is Disconnect, a disconnection dialog is issued to the user and the network connection is torn down,
- *Update*: following the disconnection, information on the session is added to the marketing database (actor DB again) for the theatre reservations system, including the time of day, the number and type of interactions and the size of the sale.

The ticketing and marketing databases are different but in this analysis they are both assumed to be provided by the same system, identified as the actor DB. This Use Case description will be the basis of the activity graph in Figure SA below, which captures the performance parameters of the activities traced in the Use Case.

Jacobson goes on to develop the software design from a set of Use Cases; we will instead go on to develop a performance model. We do not require that the design be done by Jacobson's method. The value of the Use Case is to provide a bridge between the definitions and actions seen by users, and the internal details. In fact every performance study has to include some kind of explanation of the work being done by the different classes of users in the model, and a Use Case is just a convenient framework for doing it. We will encode each Use Case in an activity graph to capture the performance data, and then reduce the parameters to represent a module or a complete program.

1.1.2. Modules

Modules are central to design, and to a performance model of a design. "Module" is a flexible term that refers to any identifiable piece of software, which might be a language unit such as an object, a procedure, a task, an Ada "package" or a Module "module", or just a section of code. For performance analysis a certain level of module granularity is chosen by the analyst or designer, and performance parameters are determined for one request to the module. Booth and Weicek called such a module description a "performance abstract data type" [booth&weicek], since it hides its other internals for modelling purposes.

If a module M represents an *object*, it implies a need to model different kinds of calls to the *methods* of the object, providing different services and having different workloads. The discussion below uses the term "entry" to designate these different services.

Modules take two complementary roles, as units providing service to programs, and as domains of analysis. As a unit providing service, the module (or rather its "entries") has a given set of execution demands, so its internals need not be examined. When a module is a domain of analysis its entries' internals are described either by scenarios or by module refinement, and execution demands are obtained.

Scenarios for module entries will be represented by activity graphs which provide a detailed description that reflects their requirements (Use Cases) or their design (execution flow). The activity graph description can be *reduced* to a module entry description. But the activity graph can also show the frequency of use of *other* modules, so the activity graph concept is related to the module concept in two ways:

Module M, entry G has execution described by *Activity Graph g*

Activity Graph g, step A invokes *Module m execution*

Figure SAL illustrates these relationships. Invocation of external modules is routinely part of the activity graph definition to be described next, while the Module *M* description is derived from its activity graphs by a *reduction R1* which condenses the graph description into a few demand parameters.

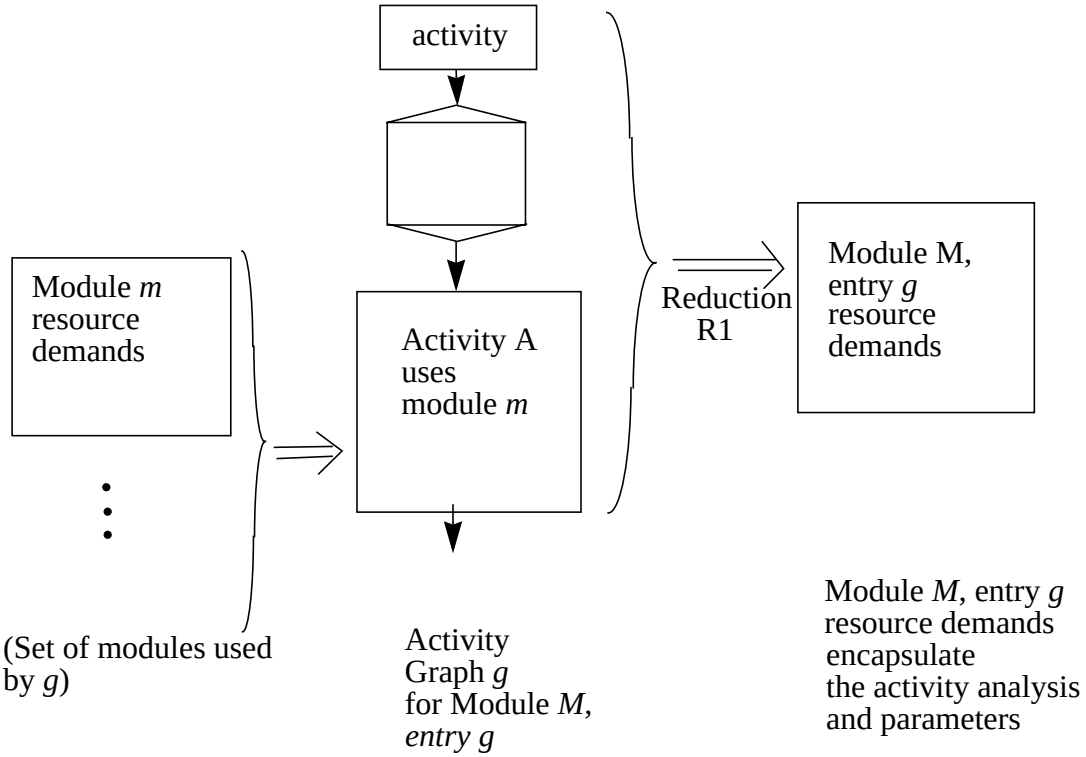


Figure 3.2. Relationships Between Module Descriptions and Scenarios (Activity Graph) Descriptions (Figure SAL)

If we are building a system from existing modules then the modules do not stand in isolation, but one module may call other modules. It is a convenience to be able to hide the further calls by aggregating the other modules into the first one, to get a total workload description for the one interface as indicated in Figure SAM.

The following sections define Activity Graphs which capture the performance parameters of scenarios within modules, and the manipulations that allow parameters captured in a scenario to be used with a module representation, and in a performance model.

We will expect to use the scenario reduction to encapsulate the scenarios in module parameters. We will incorporate other modules via their own demand parameters, suitably aggregated to the level needed by the scenarios. The analysis may be recursive, so low-level modules will be reduced and used by higher level modules or scenarios.

The programs considered in this chapter will mostly still be linear, so we can concentrate on the analysis and combination of modules. This gives the simplest version MSS(Modules) of the MSS framework, for multi-layered service system by modules. Parallelism in activity graphs will be introduced, to be exploited later. In later chapters we will see how the demands generalize to include demands for logical resources and external services, to give the more general framework.

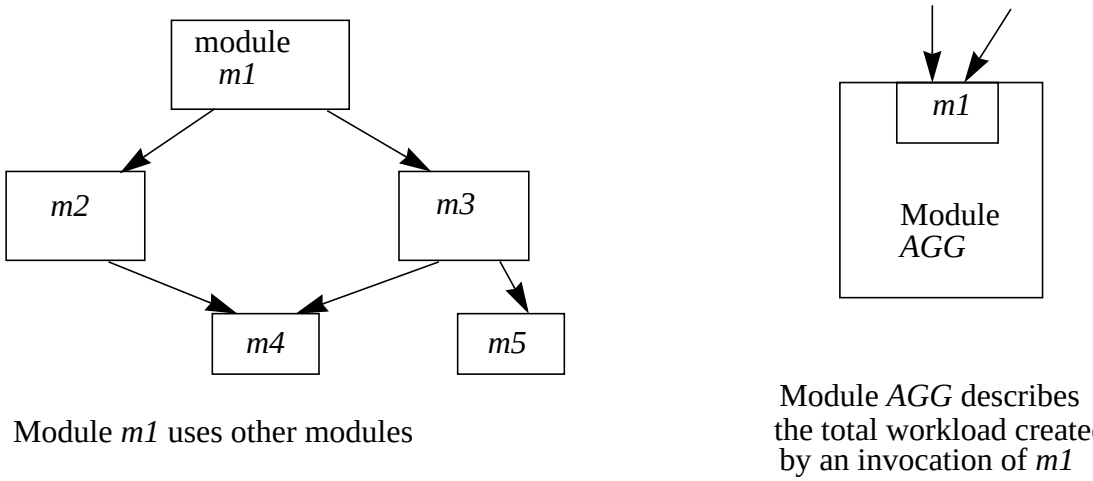


Figure 3.3. Idea of Module Aggregation (Figure SAM)

1.2. Activity Graphs to Capture Scenarios and Workloads

An activity graph expresses a scenario with performance parameters. In the very first planning of a program scenarios are the easiest way to describe the necessary processing, which has been exploited by Jacobson in inventing Use cases. Many other established software design techniques also employ or generate scenarios. If scenarios are not available from the design or requirements analysis, they may be created just for performance analysis. Activity graphs are just a notation we will use here for adding performance information to these scenarios. Other authors have called them “task graphs” or “precedence graphs”, and defined them in much the same way. Smith for example defines “execution graphs” for the same purpose, and describes a practical performance review process to generate parameters from expert expectations [CUS90].

What is new here, compared to Smith’s “execution graphs”, is including calls to modules in describing an activity (Smith only included device operations.) Methods for gathering and exploiting data on existing modules are emphasized. Modules being designed will be treated in the same way as existing modules except that their performance data is derived from expectations. Since a higher level of module re-use will give less dependence on imprecise estimates, and more dependence on known information from the existing modules, it will also give a more accurate performance analysis.

1.2.1. Activity Graph Notation

An activity graph describes the predecessor-successor relationships of software activities, and their workload parameters. An activity may be any portion of processing by a program which can be identified and named without ambiguity. An activity graph has activities as nodes and arcs which show the flow from one activity to another; there are also nodes which provide more complex connections between activities.

Figure SB defines the notation used in this work for picturing activity graphs. An activity is represented by a box, with a sequence of boxes joined by arcs, to define a sequence of activities. A conditional or optional activity is shown within a special choice box that has a triangle above and below, with a place to show its probability. Within one condition, a sequence may be shown, but further nesting of conditions is better represented by nested activity graphs (described later). A set of alternative cases is a collection of alternatives with their probabilities. A loop is identified by a circle at the head, with a mean loop count attached to it. Nested loops may be shown. Parallel operations with forks and joins use a horizontal bar to show the fork or the join. The sequence of processing in each branch of the fork-join may be shown on a separate graph which expands the activity shown for each branch. The same bar for parallelism is used when a message is sent or received in such a way that the flow forks or joins.

Logical resources such as critical sections and locks are important in distributed or concurrent software, and location within the flow of the point of acquisition or release of a resource can be indicated.

An activity graph is drawn vertically down the left side of the page with performance parameters beside it. The notation is defined so that different activities are located on different lines, so the performance information for each activity is written beside the activity in a tabular form.

1.2.2. Performance Parameters and Calculations in an Activity Graph

The performance parameters of an activity graph will be explained first using the example shown in Figure SKD. There are two activity graphs for a module *m* with two high level operations (or entries), called *m.e1* and *m.e2*. The function of this module is not our concern, but it uses file operations and two X-windows functions listed here as Xwin.create, to create a new window, and Xwin.inout, to read and write text in the window. Xwin is a module with two entries, inout and create.

Each activity is described by a “MeanTimes” parameter in the first column (how many times it is executed, for one execution of the graph) and by its use of *logical services* which include logical processor instruction executions, file operations, and execution of other software modules. There is a column for each service, giving its mean execution counts. CPU operations are taken to be machine language instructions, and a unit of one million instructions (one M-In) is used for the CPU demands.

When this information is first obtained it is usually easier and more useful to define values for logical services rather than for hardware operations. Thus, a number of file read operations may be identified in the software and the expected number of these operations within one activity may be recorded. This leaves the task of identifying

- how much is read and written
- how many disk operations or network operations occur for each file operation
- how long each disk operation takes

to a later analysis, when more is known about the application, the operating system, the configuration, and the choice of hardware devices. For instance a file to be read may be on a disk attached to the processor, or accessed over a local network from a network file system like NFS.

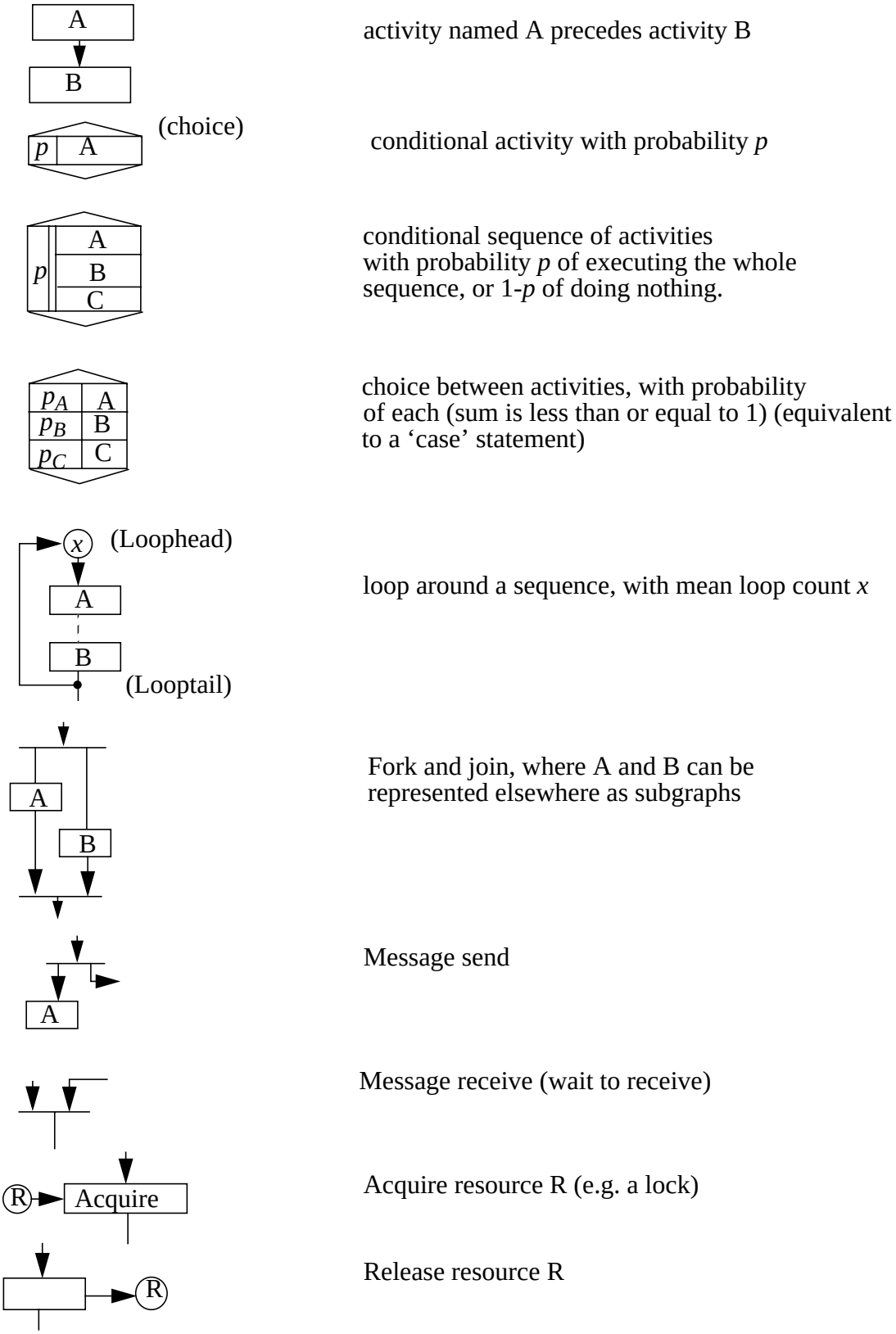


Figure 3.4. Activity Graph Notation. (Figure SB)

The difference arises not in the description of the software but in other decisions. We may in the end build separate models for the two cases, both based on the same software and activity graph, but substituting different file operation sub-models. The case of file operations will be considered again, for example to deal with the size of each operation.

At the bottom of Figure SKD a set of totals is shown, which are the total request counts for service demands when the graph is executed once. The graph can be represented as a single aggregate activity, possibly for use within a graph written at a larger scale, and these are the parameters of that aggregate activity.

1.2.3. Activity Graph Workload Parameters: Alternatives

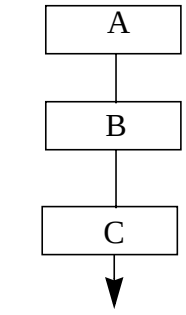
The analyst can decide what will be considered to be internal to each activity and what is considered to be a service obtained by the activity from outside the software modelled by the graph. Making a service external allows it to be analyzed separately, and its parameters to be used in other models, so it is an aid in re-using performance information.

If, in Figure SKD, we wanted to consider the file operations and the X-server operations Xwin.create and Xwin.inout as internal to the activities A to G we would have to substitute in their CPU service and disk operation request counts. Then we obtain the service request counts just for CPU and disk operations as shown in Figure SKDD. In this way all the logical service requests can be eliminated and the device request rate found; it is then only a small step to get a performance model. This will be detailed below as Reduction R1.

1.2.4. A Large Activity Graph Example: Theatre Reservations System

Now the more complex activity graph given earlier in Figure SA for the theater reservation system use case will be analyzed to develop a workload characterization for the system. Looking again at Figure SA, the first activity is to set up a connection with a user that selects the service. There follows a loop representing the repeated interactions with new web pages for information, or for making the reservation. The loop shows a mean loop count of six, and four kinds of interactions in a “choice” box identified by a triangle above and below, with a probability for each choice. Each user interaction displays a page, waits for the user to react and then makes a choice depending on the user’s input. Thus there is one user selection of a hot-link or menu item, for each “choice”. The probabilities given here show that three-quarters of all interactions just display information as the user navigates the possible programmes for which there are tickets for sale. Fifteen percent are interactions for making a reservation, involving selection of seats and prices and filling in details of the desired sale. Somewhat less (10%) are interactions to finally confirm the sale, verify the credit-card transfer with the bank via a network transaction with a credit-card server, and set up the processing of the order. Finally there is a disconnect operation and a database operation to save certain data about the users activities, for later use by marketing.

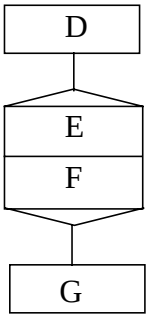
The activity graph is much simpler than a complete program specification, in that it lacks details (such as data definitions and transformations) and it may lack entire functions. Functions that are almost never performed can be left out, particularly in a preliminary analysis. For instance, this graph doesn't specify what is done if the credit-card server doesn't answer.



Activity graph for entry *m.e1*

Service demands per repetition of *m.e1*

	K MeanTimes	CPU (M-In)	File Operations	Xwin.create operations	Xwin.inout operations
A	1	0.1	1.8		
B	1	0.2		1	
C	1	0.1			13
Weighted sum (K x demand)		0.4	1.8	1	13



Activity graph for entry *m.e2*

Service demands per repetition of *m.e2*

	K MeanTimes	CPU (M-In)	File Operations	Xwin.create operations	Xwin.inout operations
D	1	0.15			
	1	0.01			
E	0.9	0.2			3
F	0.1	0.6			6
G	1	0.1	2.5		
Weighted sum (K x demand)		0.50	2.5		3.3

		CPU (M-In)	Disk-op
Device demands for Logical services:	File operation	0.02	1.3
	Xwin.create	0.75	0
	Xwin.inout	0.29	0

Figure 3.5. Activity Graphs for a Module with Two Entries (Figure SKD)

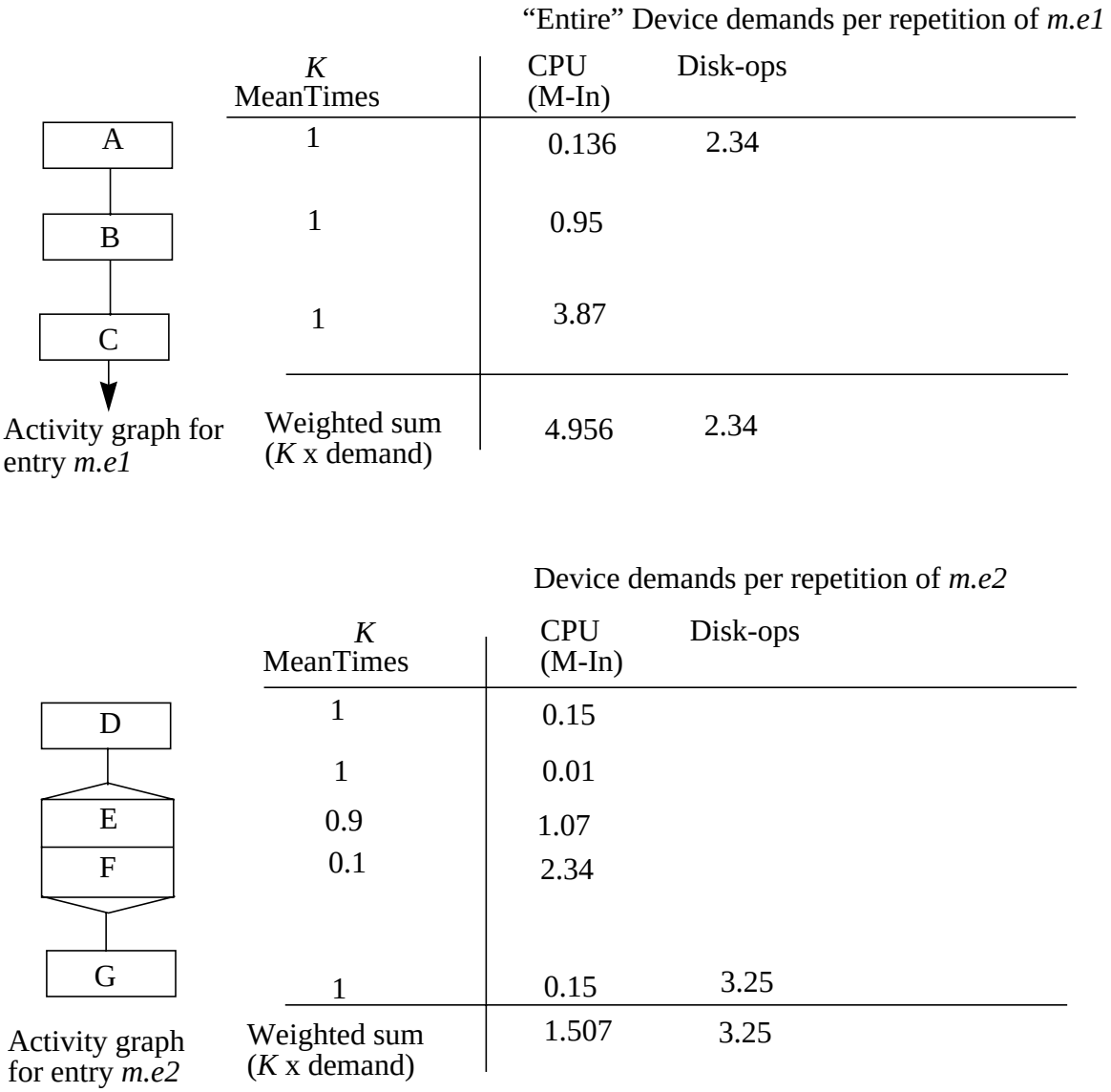
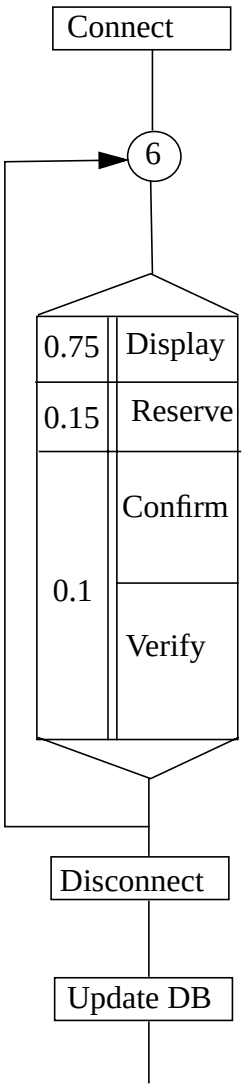


Figure 3.6. Activity Graphs for a Module with Two Entries, with Software Service Demands Resolved into their Device Demands (Figure SKDD)

The value of an activity graph is for understanding and communicating issues to do with performance. It is most useful in the early stages of planning, when there may be no other unified description of program behaviour, or for capturing and understanding a complicated behaviour pattern with important performance effects. It may be produced and reviewed by a development group, and be used to gather expert opinions about expected resource demands of the activities, if (as usual) some data are missing.

The activity graph and its table of the performance parameters is shown in Figure SC. For each line (meaning, for each activity or node) there is a “MeanTimes” figure, which is calculated from the loop counters and the choice probabilities. Then there are mean numbers of requests for services, each time the activity is executed. A service could be:



Graph	Node	MeanTimes	CPU M-Ins	Data Base Server (ops)	Netware Server (ops)	CCReq Server (ops)
Connect	Connect	1	0.010		1	
6	Loophead	6	0.001			
	Choice	6	0.001			
0.75 Display	Display	4.5	0.005	1	1	
0.15 Reserve	Reserve	0.9	0.015	2	1	
0.1 Confirm	Confirm	0.6	0.002		1	
0.1 Verify	Verify CC	0.6	0.004	1		1
Disconnect	Disconnect	1	0.001		1	
Update DB	Update DB	1	0.007	1		

Figure 3.7. Ticket Reservations System: Activity Graph for a Reservation Session, with Parameters. (Figure SC)

- an operation by a device, for instance a CPU instruction or a disk operation,
- a logical service provided by a software module, such as a file operation provided by the file

system, or a logging operation provided by a module called “EventLogger”

- a logical service provided by some remote system, for which the program must wait, such as the Credit Card Server in the Figure. These services will be treated as if they arise from a module which executes on a remote system.

Notice that we delay estimating the CPU time or the other service times at this point; the analysis is more portable if we can first estimate logical operations and services, find the totals, and only at the end insert the device operation times. Then the operation times may be changed easily to consider other devices or models. Similarly the use of services by other modules is simply identified at first; when the other modules are analyzed their parameters can be filled in. This allows one to analyze one module at a time, to “divide and conquer”. Too often a performance analysis is abandoned because the analyst is asked for too much information in one step.

Figure SC shows the mean request counts made to the following devices and services:

- a CPU, in millions of instructions (M-Ins) executed by the activity
- a network information server module, such as a Web server, in operations each of which handles one request from the Web client,
- a database server module, in operations (all queries and updates counted equally).

The parameters in Figure SC show only the direct demands for service made by the activities of the graph. That is, the CPU demand by the activity itself is shown, but not the CPU demand of a module which it calls, even if the module runs on the same processor. The parameters which are shown are called the “local” parameters of the activities. The local parameters are the MeanTimes $K(a)$ for activity a, and the request values $Y_i(a)$ for logical service requests from devices and modules:

$K(a)$ = mean number of times activity a is executed, each time the entire graph is executed
$Y_i(a)$ = mean requests by activity a for logical service i, per execution of a

For example, Y_{CPU}(Display) = 5000 instructions, $Y_{DataBase}$(Display) = 1. For the loop logic, Y_{CPU}(LoopHead) = 1000 instructions is the cost of executing the logic to control the loop, each time through.

To distinguish the CPU demand for the entire activity, including all the services used by the activity, we will call it the “entire” CPU demand. It is found by the first of the two graph reductions described below.

1.2.5. Reduction *R1*, from an Activity Graph to a Device Workload Model (Linear Software Case)

This reduction systematizes the previous discussion about substituting for the parameters of a service, for the particular case where all the logical services are to be substituted successively until only requests to devices are left. These will constitute a performance model at the device level as described in the previous chapter. It is assumed that the activity graph is sequential (no parallel subpaths).

We go from an activity graph to a set of device demands in three steps:

1. Add up the demands for logical services vertically in the graph g, taking into account the MeanTimes factor $K(a)$ of each node. This gives the total local request counts $Y_i(g)$ of

the graph *g* as a whole, both to devices and to module services.

$$Y_i(g) = \sum_a K(a) Y_i(a)$$

2. Eliminate services in modules which are internal to the system described by the graph (the decision as to which modules are internal must be made by the analyst). By successive substitutions reduce the total local request counts for module services down to entire demands for operations from device.

   $Y'_i(g)$ = entire demand, or total service demand from the graph *g*, to service *i*, where *i* must be external to the software. In *R1*, service *i* must be a *device* or an external subsystem which is modelled as a device.

   Each service by a module is assumed to have known average device request counts. Suppose that

   • the graph *g* uses service *j* an average of $Y_j(g)$ times, and

   • service *j* has an external request count of $Y_i'(j)$ for operations of device *i*,

   then service *j* contributes $Y_i'(j)Y_j(g)$ to the entire count for device *i*, and

$$Y'_i(g) = Y_i(g) + \sum_j Y'_i(j) Y_j(g)$$

3. Associate a physical device with each device-service and determine its operation time O_i. This gives the demand $D_i(g)$ for the graph, in seconds of resource-*i* service, per execution of the graph *g*:

$$D_i(g) = Y'_i(g) O_i \ .$$

Figure SE walks through the steps of reduction *R1* for the example shown in Figure SA.

Reduction R1

Step 1: From "Table 0" given in Figure SA, Multiply "MeanTimes" into "Request Counts", to give the following "Table 1", and add up the columns.

Activity	CPU M-Ins	Disk	Data Base Server (ops)	Netware Server (ops)	CCReq Server (ops)	User Input
Connect	0.010			1		
Loophead	0.006					
Choice	0.006					
Display	0.0225		4.5	4.5		
Reserve	0.0135		1.8	0.9		
Confirm	0.012			0.6		
Verify CC	0.024		0.6		0.6	
Disconnect	0.001			1		
Update DB	0.007		1			
Entire Demands=Sum	0.102		7.9	8.0	0.6	

Step 2(a): Module Service Demands to be eliminated

Module Services	CPU	Disk	User Input
Database Server, per op.	0.085	2	
Netware Server, per op.	0.012	1.5	1

Step 2(b): Compute *entire demands* for Ticket Reservations, to give "Table 2":

	CPU	Disk	CCReq.	User input
Local demands (Step 1):	0.102		0.6	
7.9 x Database demands	0.6715	15.8		
8.0 x Netware demands	0.096	12.0		8.0
Total for Ticket Res. = 'Entire' demands Y_i'(Ticket Res)	0.8695	27.8	0.6	8.0

Step 3: Put in Operation Times O_i

	CPU	Disk	CCReq	User input
Device operation time (O_i)(sec)	0.1	0.011	3	7 sec.
Device demands ($D_i = Y_i'O_i$) (sec)	0.08695	0.306	1.8	56 sec.

Figure 3.8. Steps 1, 2 and 3 in Reduction *R1* of the Ticket Reservation System "Reservation Session" Parameters, to Hardware Device Demands per Response. (Figure SE)

For Step 1, the data table beside the activities in Figure SA will be called "Table 0". In each row, the repetition count MeanTimes (i.e. *K(a)* for activity *a*) is multiplied by every entry in the row to give "Table 1." The columns are summed to give the total local demands, that is the request counts made by the activities in the graph. These include logical services in the last four columns. For instance the local CPU requests made by Display is 0.005 Millions of instructions each time, or 0.0225 M-Ins on average each time a client connects to the service (4.5 repetitions x 0.005), and the total over all the activities is 0.102 M-Ins, or 102,000 machine instructions. The interrogations of the user, and a delay for the user to make a selection, are included in the last column.

Step 2 is shown in two parts. Part 2(a) is an auxiliary table with a row for each module used by the activities in "Table 1", and the module requests are broken down into demands for logical device-service requests. There are just two modules, the Database and Netware Servers, which are internal to the graph workload. These will have device demands substituted for them. The other two logical demands columns (CCReq, Server and User) show services which are external to the system being analyzed by the graph reduction, so their request counts are retained at this stage. In step 2(b) these definitions are then substituted into the totals to give the line of "entire" device-service requests, Y'_i, for the graph which will be called "Table 2". For instance the Database Server requires 85,000 CPU instructions and 2 disk-accesses per request (in the appropriate mixture of read and write request types, one assumes -- a point that will be addressed shortly). As there were found to be 7.9 requests to the DataBase server in traversing the graph, this contributes 0.6715 M-Ins (7.9 x 0.085) to the entire CPU demand for the graph. The 8 requests to the Netware server are assumed to each involve one delay for a user interaction. The user is external to the system, and so is treated as if he/she were a device, as is also the external server of the credit-card company.

For Step 3, operation times are defined for the devices, and the device demands D_i are calculated. If the CPU runs at 10 MIPs we obtain a time of 0.1 sec for each M-In, and the demand of 0.8695 M-In gives 0.08695 sec of CPU demand for each response. We could round this to 87 msec. We have an option to give times for the external services CCReq and User. If we can supply times, we can include delays for these services in a queueing model. Here, Y_i, the user input delay is included here as 7 seconds; the total of 8 user delays per response is thus entered as 56 sec. In this table the external CCServ delay is also given a value of 3 sec. Later we will see another option for modelling, which includes CCServ within the model as a module.

The result of this reduction is a set of loadings for the CPU and disk devices and for the Users and CCReq (as external servers) in seconds of service, for each execution of the activity graph. This corresponds to an entire user session with the server, but we will treat it like a "response" (a unit of operation) as described in the previous chapter. Then the loadings D_i are exactly the demand parameters of the hardware queueing models described in the last chapter.

The important values for a hardware queueing model are:

- CPU demand D_{CPU} = 0.087 sec./session,
- disk demand D_{Disk} = 0.306 sec,
- User delay Z_{User} = 56.0 sec., and

• CCReq delay Z_{CCReq} = 1.8 sec.

The servers for User and CCReq are pure delays, or infinite servers in the queueing context, and in the queueing model the two values of delay per response are added to give a total "infinite server" delay of $Z_{User} + Z_{Sys}$ = 57.8 sec./session.

To get the demand values per user response that were analyzed in Section 2.3.5, we divide the demands per session found here, by 8 responses per session.

1.2.6. Summary and directions

This section has shown how the execution scenarios or Use Cases of a system can be developed into activity graphs which capture those aspects of behaviour which are important for performance. It is emphasized that an activity graph can be much less detailed, and less logically complete, than a software specification, since it only has to capture attributes which are executed during performance critical responses, and executed often enough to influence performance. It has also shown how the parameters of an activity graph can be reduced to give total values for the graph, and to give a simple performance model based on device contention alone.

The analysis of hardware-based performance limits is easy to do, and is recommended as a first step in any study. However a deeper analysis may be essential:

- a stochastic queueing model can introduce the effects of random interference, and give performance in unsaturated systems,
- opportunities for parallel subpaths and concurrent operations may be found and analyzed,
- logical resources shared by concurrent processes should be described in the graph, and their effects found.

and these further needs are the object of the MSS framework. To fully investigate them we will need to use the extended properties of activity graphs, such as forks and joins, and it will be a great convenience to capture the idea of a service provided by a module. In fact the role of the activity graphs is to help understand and define the behaviour of modules.

1.3. Patterns in Activity Graphs

Although our main interest is in patterns of concurrent tasks, it is essential to recognize some performance-oriented patterns for sequential activity graphs. POPs are used to express performance problems in the software, and to solve them. In an activity graph a POP is typically a part of the graph (a subgraph), which may be modified or transformed to reduce its level of resource demands. In sequential code expressed by an activity graph or subgraph the only performance "problem" is too large a resource demand, such as too much CPU demand. Less is always better. Demands are expressed through the entire request counts Y_i'(Subgraph). A great deal of analysis has been performed on certain algorithms to determine their demands for certain operations, and to find better algorithms with smaller demands; this may provide solutions to some problems. Works such as Bentley's "Programming Pearls" [pearls1] make suggestions for systematically improving the efficiency of code. C. U. Smith [CUS90] has collected folklore about efficiency and organized it into a number of "performance principles", some of which we will see in our patterns.

In general, demands may be reduced by either

- modifying a demand parameter of an activity, or
- transforming the subgraph to another that has a smaller demand (using a different algorithm)

For instance a CASE block with branches Branch_1,.... and probabilities p_1,... is a simple pattern, and its entire request count for resource i is

$$Y_i'(\text{CaseBlock}) = \sum_j p_j \; Y_i'(\text{Branch}_j)$$

A costly branch (with large Y_i') may be relatively unimportant if its probability is low enough. As a first step towards reducing the overall Y_i'(CaseBlock), attention should be concentrated on branches with a large product $p_i Y_i'$; then the internals of the branch should be examined to find a way to reduce its Y_i' value.

We shall consider two important sequential patterns,

- the “fast path” pattern,
- the “optimistic” pattern and its variations.

These patterns can take many forms. They reappear later in concurrent software, and lie behind many of the mechanisms that give Internet software its scalability.

1.3.1. The Fast Path Pattern

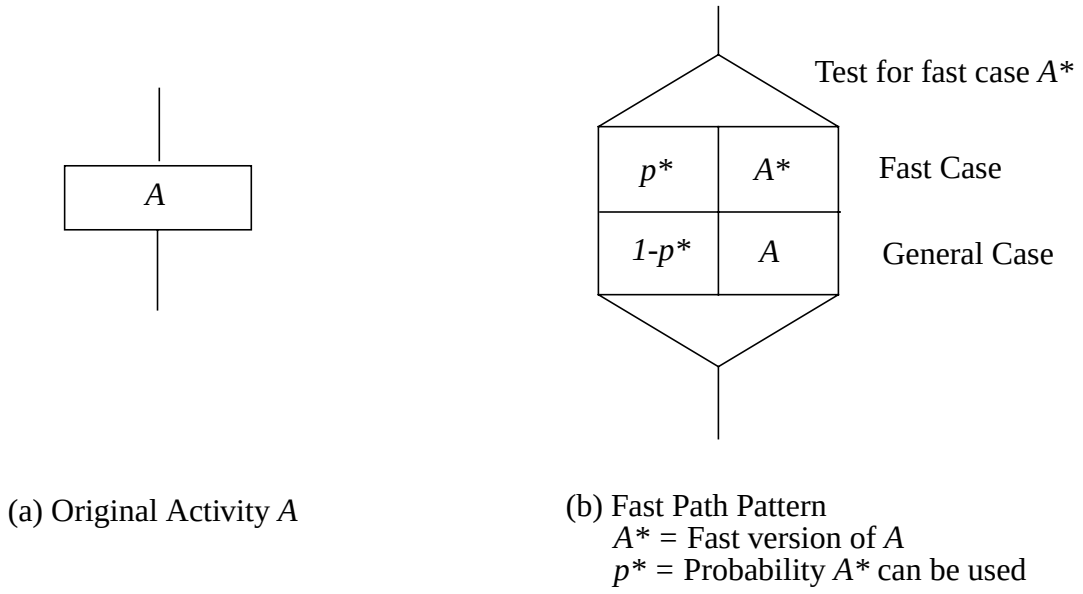
A fast path executes special fast processing that is applicable only to a special case of whatever is being done. Suppose there is an activity A which can process the most general case of the data, then in figure SFG it is replaced by

- a test to determine if the special case holds,
- a CASE block which executes the special case as activity A* if possible, or otherwise the general case A.

A simple example from the earliest days of computer hardware is an arithmetic multiplier that tests if one of the arguments is zero, and if it is, writes zero as the result. The well-known UNIX Make utility tests to see if binaries are “up-to-date” and only recompiles them if the source has changed.

In the Reservations activity graph, a fast path for retrieving a page of theatre program information could first examine the cache of web browser, and if a valid copy of the page is in cache, retrieve and display that page without making the request over the net. Of course to have a page in cache, it is necessary to maintain a cache, which means doing some extra work in other places in the activity graph. Also a page with volatile information like a list of available seats would have to be reloaded every time.

Consider the quantitative advantage of using a particular fast path, ignoring for the moment any extra work in other parts of the program that may be needed to support it. Suppose the probability of the special case is p^* as shown, then the ratio of the resource demands with the fast path, to the demands without it, are



(a) Original Activity *A*

(b) Fast Path Pattern
*A** = Fast version of *A*
*p** = Probability *A** can be used

Figure 3.9. A Fast Path Pattern (Figure SFG)

$$\text{Ratio} = [Y_i'(\text{Test}) + p^* Y_i'(A^*) + (1\text{-}p^*) Y_i'(A)] / Y_i'(A)$$

For cases with an advantage, Ratio < 1.0. If there is extra work outside the fast path, a fraction of it suitable to charge against a single execution of the pattern is added to the numerator.

Clearly there may be more than one special case with its own fast path. The tests may be done all at once, giving an *n*-way CASE block, or one at a time starting with the most likely case, giving recursively nested CASE blocks.

All this discussion has been in terms of a single resource *i*, while implicitly there may be many resources used by the pattern. If all are decreased, then for a sequential program there will be a performance gain. However if some are decreased at the expense of others, the net gain or loss will have to evaluated by a model (a point made repeatedly by Smith). For linear software with no simultaneous resources, a queueing model is sufficient. In general it is more important to reduce the use of resources that are heavily used by the program, and performance may not suffer from additional demands for a lightly used resource. Thus a model solution may guide the tradeoffs made when some resource demands increase and others decrease.

In C. U. Smith's work there is a "Centering Principle" which we may translate roughly to say, "use a fast path when Ratio < 1" [CUS90].

While a fast path may reduce the mean value of the demands if Ratio < 1, it also tends to increase the demand variability expressed as the Coefficient of Variation (CV = variance/ square of the mean). This is because an operation which is a mixture of other operations with different mean demands has a higher demand CV than any of its parts.

1.3.2. The “Optimistic” Pattern

What is called “optimistic design” uses the same idea as a fast path, but in situations where it is impossible to do the test without first executing the special case. This case is executed “optimistically” in the hope that it will succeed. If it does not, then in many cases it is necessary to undo its results in a Recovery step before proceeding to the general case, which is called the “conservative” case since it does not depend on the optimistic assumption. This gives the transformation of activity A as shown in Figure SFK(b). There is another optimistic pattern that can be applied if the conditions that make the test fail may have changed. One may retry the optimistic step again after executing the Recovery.This gives an entirely optimistic transformation as shown in Figure SFK(c).

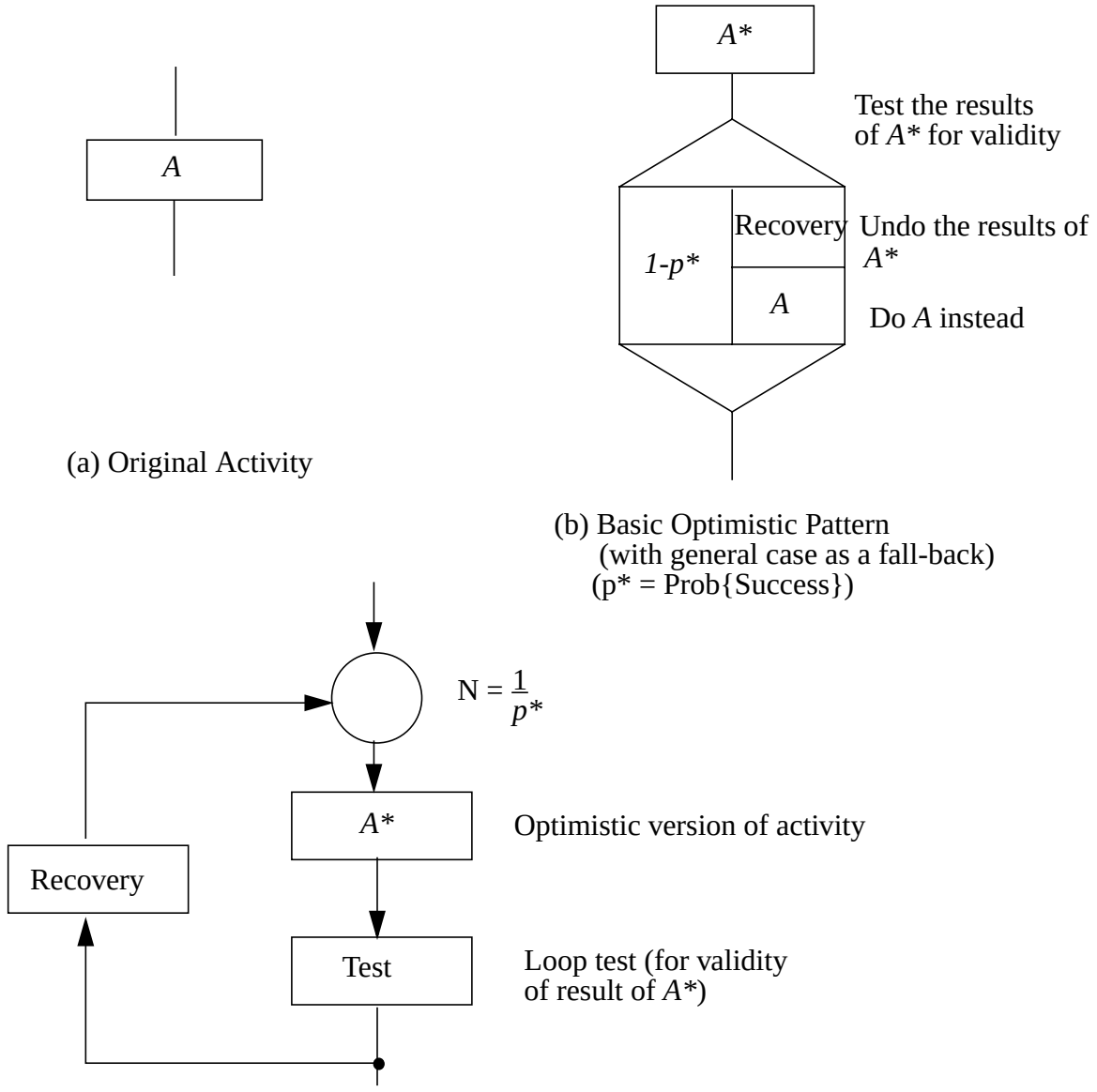


Figure 3.10. Two Optimistic Patterns: (a) Original conservative activity (b) Basic optimistic transformation with conservative fallback to original processing (c) Alternative "entirely optimistic" transformation (Figure SFK)

Examples of optimistic transformations are pervasive, and they have been systematically studied by for example Bubenik, [bubenik] Strom and Yemini [stromyemini] and Cowan [HOPEsem95]. A good example of an entirely optimistic design is the use of optimistic locking of data. The processing of data is carried out without any lock, on the assumption that there is no conflict. The fact the data is being used is recorded, and the results are written in a form that can

easily be annulled. At the end of the operation the possibility of conflict is checked from the data usage records, and if there has been no conflict the results are made permanent ("committed"); if there has been a conflict the results are annulled (rolled back) in a Recovery step and the transaction is restarted, possibly after a period of backoff to reduce the chances of another conflict.

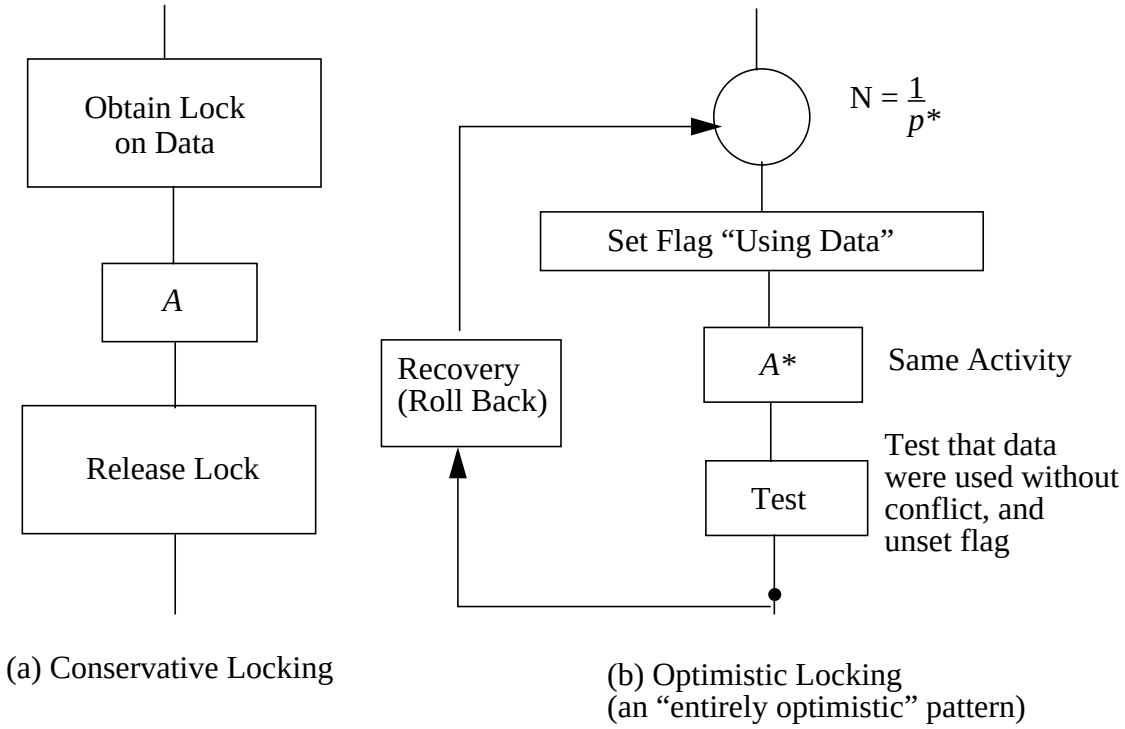


Figure 3.11. Optimistic Locking. (Figure SFN)

An other example of optimism is van Jacobson's optimization of the TCP/IP protocol stack for network file systems using Header Prediction [TCPspeed89], where a packet received by a workstation is written directly to the space it will eventually reach if it is a message to the kernel (e.g. for a remote file request) rather than a message to an application. A large proportion of messages satisfy this, and it saves a copy operation. The recovery, if the header has been read and found to belong to a user message, is to move the packet to a user buffer area. This is also cheaper than the original conservative option of moving it twice. In this case, as in many others, the conservative path is modified in the optimistic design.

The advantage of an optimistic transformation is generally not quite as great as a fast path because the fast operation, the test and perhaps the Recovery have to be carried out, instead of just the test. In Figure SFK(b) the Ratio between the original and the Basic Optimistic demand for resource *i* is seen to be

$$\text{Ratio} = [Y_i'(A^*) + Y_i'(\text{Test}) + (1\text{-}p^*)\,[Y_i'(\text{Recovery}) + Y_i'(A)]] \,/\, Y_i'(A)$$

Figure SFP shows the values of the Ratio for some values of p* and the demands, assuming the following relationship among the demands:

$$Y_i'(A^*) = Y_i'(\text{Test}) = Y_i'(\text{Recovery}) = \alpha\, Y_i'(A).$$

Then:

$$\text{Ratio} = 2\,\alpha + (1 - p^*)\,(1 + \alpha)$$

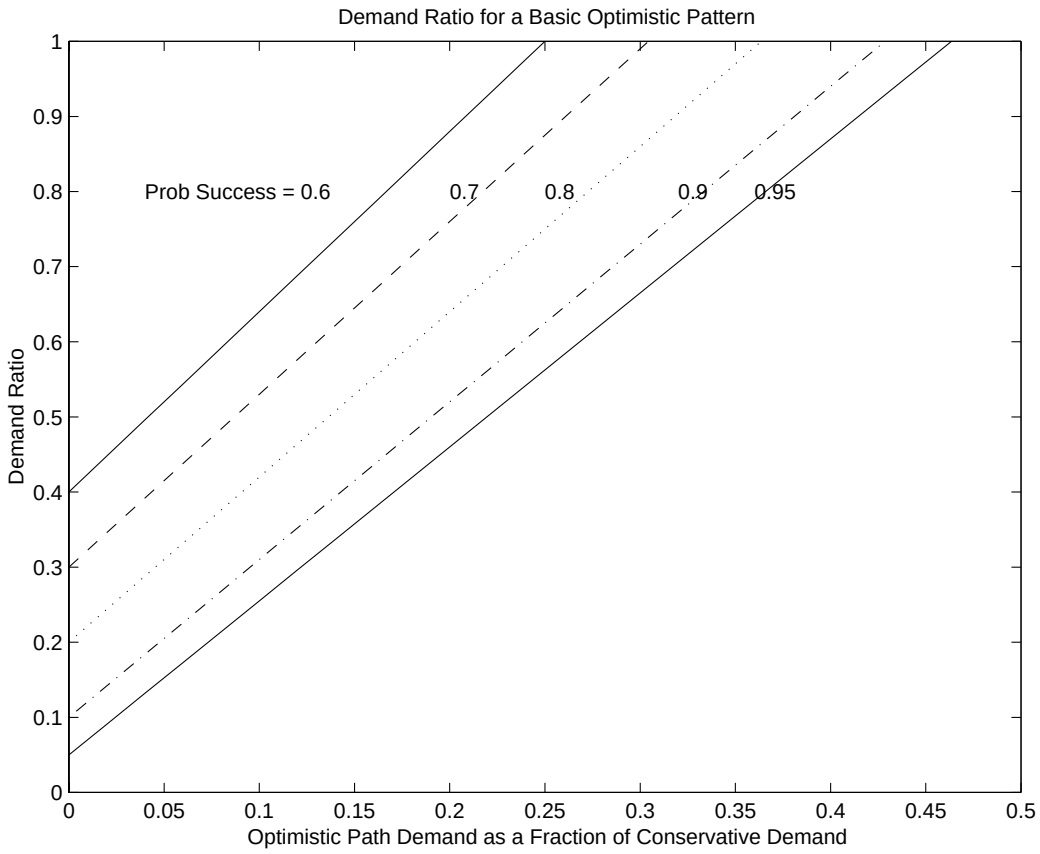


Figure 3.12. Basic Optimistic Pattern: Demand Ratio [Basic Optimistic/Original], for various effort ratios (α) and success ratios (p^*). (Fig. SFP)

In Figure SFK(c) the Entirely Optimistic version has the demand ratio

$$\text{Ratio} = \alpha\,[(3/p^*) - 1]$$

which has the values shown in Figure SFQ for the same assumptions as in Figure SFP. Actual implementations such as optimistic locking may have variations on the pattern such as the extra operations shown for setting flags.A Parallel Section Pattern

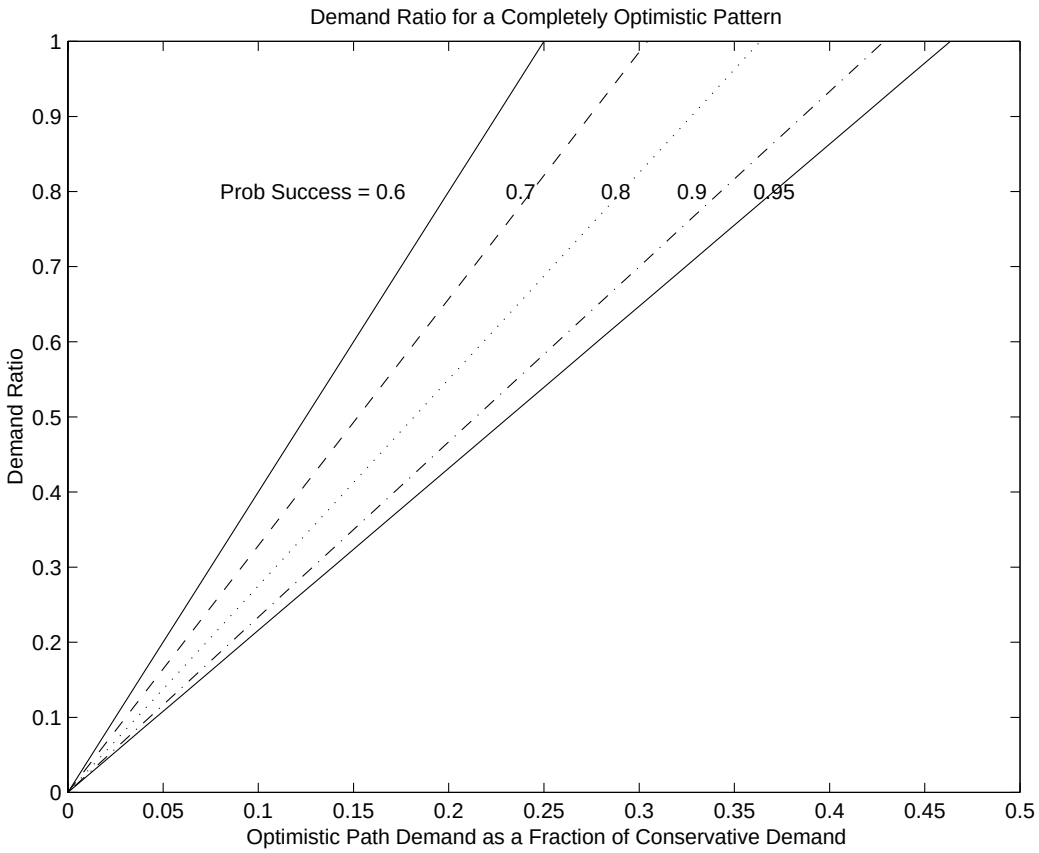


Figure 3.13. Entirely Optimistic Pattern: Demand Ratio [Entirely Optimistic/Original], for various effort ratios (α) and success ratios (p^*). (Fig. SFQ)

1.3.3. A Parallel Section Pattern

This pattern does not really belong in a chapter on sequential software, but it does fit in with activity graphs, so it will be briefly discussed. Figure SFR shows the parallelization of one activity into a set of parallel branches, which we will assume are executed truly in parallel by separate processors. The pattern in Figure SFR(b) shows overhead activities that stand approximately for the effort of sending messages to the other processors and initiating the branches and then later gathering the results together.

Evaluating the benefit of parallelism requires finding the delay along each branch, which absolutely requires a performance prediction.

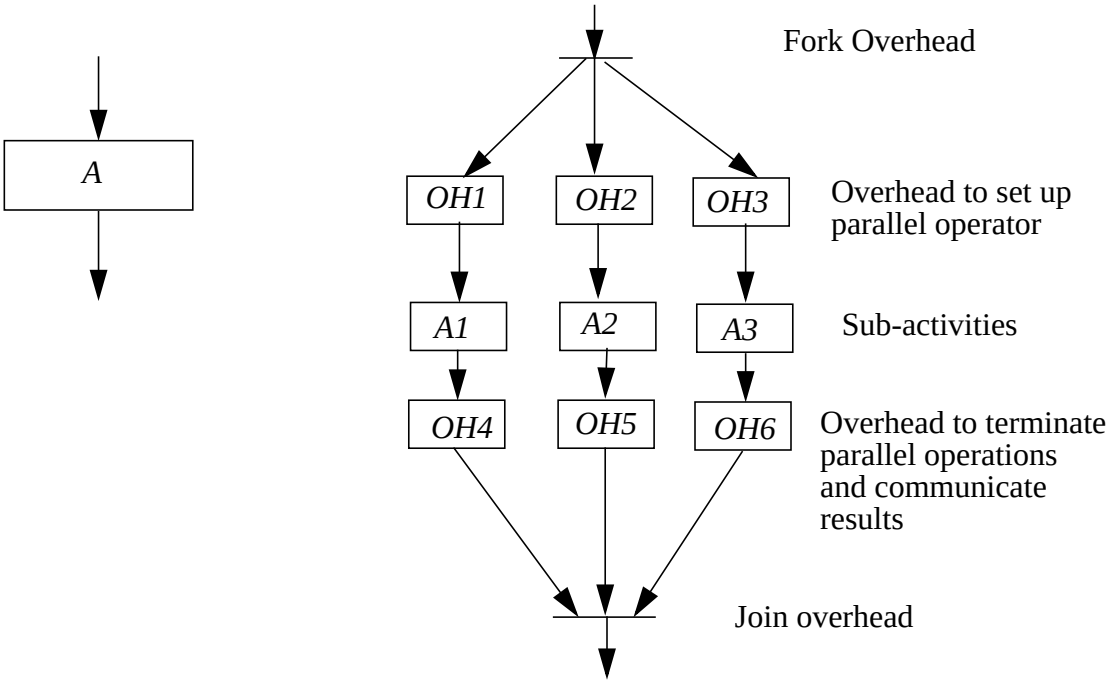


Figure 3.14. Parallel Activity Pattern (Figure SFR)

1.3.4. Conclusions about Patterns in Activity Graphs

Many design patterns are best understood and analyzed within the behaviour sequences they generate, in our activity graphs. The cases examined here are just patterns which reduce total demands within a single process. When we get into concurrent and parallel behaviour we will return to activity graphs to describe useful POPs that traverse multiple processes.

1.4. Module Models

Activity graphs are a step towards a module model. A software designer uses a Use Case or other scenario definition to help arrive at a design in terms of a set of software components and interactions. We have seen how an activity graph captures the performance parameters of a scenario; a *module model* similarly captures the performance parameters of the components and interactions of a design. The modules in the model correspond with the components in the design, at some selected level of granularity, but there may be some points of difference.

The view of a software module adopted for modelling is that of an object (the module) with methods (called *entries*). Each method executes a distinct computation, so it has a separate set of workload parameters. If activity graphs are used to define the module behaviour there is one graph for each entry.

The modules in the performance model may deviate from the software component structure. The analyst may group a set of software components into a single aggregate module in the model, to avoid excessive detail and excessive labour in determining parameters. Alternatively the analyst may subdivide a single “method” into several entries, if its behaviour and workload can

be very different for different values of its arguments. For example instead of having different named entries, a component might have an operation code as one of its arguments, with a different case of execution for each code; the analyst might then model each path as a separate entry. Similarly if two entries with different names have almost the same workload parameters they may be merged in the model. For purposes of performance modelling, entries are services with distinct demand parameters.

1.4.1. Module Notation

The performance parameters of entry *e* of module *m*, denoted as entry *(m.e)*, have exactly the same meaning as the parameters of an activity. That is, they are a set of mean counts for requests for a set of services, for each time the entry is invoked. In place of activity *a* we write the entry name *m.e*, to obtain:

$Y_i(m.e)$ = the local request count for requests by entry *(m.e)* for service *i*, (where *i* may be a service by a device or by a module) each time entry *(m.e)* is invoked.

$Y'_i(m.e)$ = total requests from entry *(m.e)* for operations by service *i*, when entry *(m.e)* is invoked.

A *module model* is a graph with module entries and devices as nodes, and requests from entry to entry, or from entry to device, as arcs, labelled with the mean request count for the requesting entry. The notation for the mean request count from entry *(m.e)* to entry *(k.d)* is $Y_{(k.d)}(m.e)$, while the mean count for device operations on device service *i* is $Y'_i\ (m.e)$. The CPU device which hosts the module is not represented in the graph to avoid a profusion of arcs in larger models; the request count instead is represented as a label on the entry (we will call the CPU the "host" device of the module).

Module models will be assumed to be *acyclic graphs*, that is there are no calling cycles, and therefore no recursive calls. This restriction can be removed, at the cost of more difficult math.

Figure SKE(a) shows the module *m* analyzed earlier in Figure SKD, making demands on a file system and on an X server, Xwin, with the CPU demands in millions of instructions shown in square brackets. Host demands can also be specified in time units, so to be complete it is necessary to specify the units.

If we have the parameters in terms of requests to other modules we can aggregate the modules together to get the entire demands of all the modules. Figure SKE shows this for the module *m* considered above; part (b) shows the file-service module aggregated into module *m*, and part (c) shows all the services aggregated into module *m*.

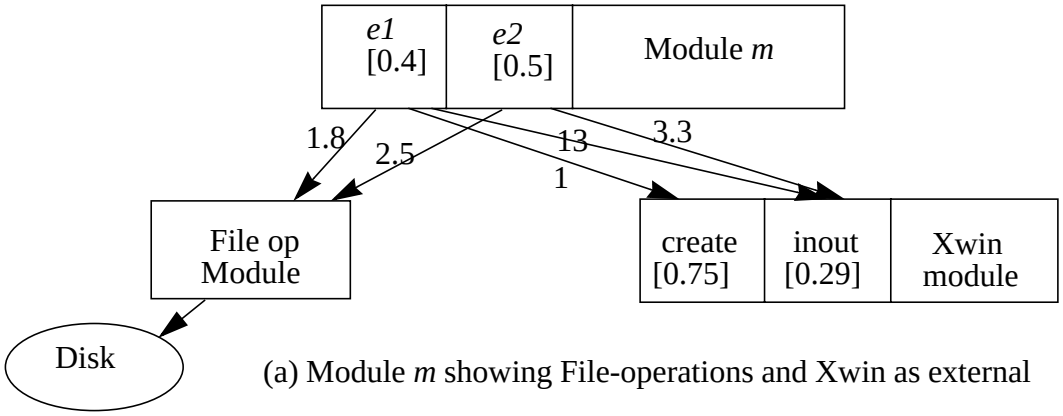
1.4.2. Reduction *R2*, from an Activity Graph to a Module Entry

An entry may be analyzed by creating an activity graph for it, in which case the entry parameters can be obtained from the total parameters of the graph. A reduction similar to R1 is applied:

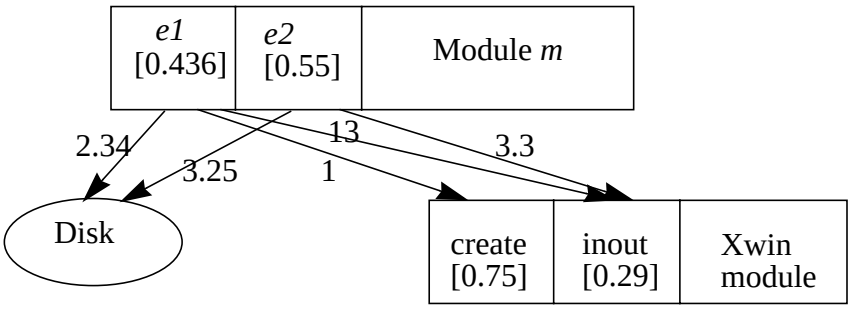
1. Determine which logical services used by module m are internal and which are external.

2. For each entry *(m.e)* in module *m*, there is an activity graph *g*.

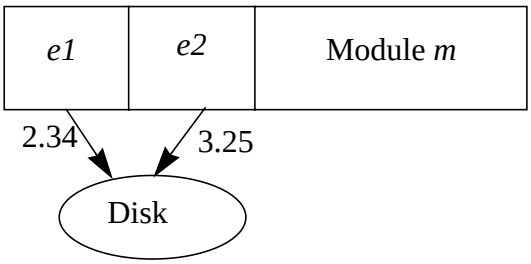
3. Apply steps 1 and 2 of reduction *R1* to graph *g* to find its "entire" demand parameters



(a) Module *m* showing File-operations and Xwin as external



(b) Module *m* showing Xwin as external, File-operations as internal



(c) Module *m* showing all operations as internal

Figure 3.15. Module Reduction *R2*. (Figure SKE)

$Y_i'(g)$.

4. The entire demands of graph *g* are the local demands of entry *(m.e)*:

$$Y_i(m \cdot e) = Y'_i(g)$$

This is how the parameters in Figure SKE were found.

For another example consider the Reservation Session in Figure SA as a module, such that all the logical services (Database, Netware, CCReq, User) are given by external modules. Reservation Session is a kind of pseudo-module representing the behaviour of a user, and the main control program, during a session. It has just one entry. Since there are no internal modules to be reduced the graph reduction only requires step 1 of *R1*. Its local demands Y_i are found in Step 1 of Figure SE to be:

-- to CPU, 0.102 M-In,
-- to Disk, 0
-- to Database Server, 7.9
-- to Netware Server, 8.0,
-- to credit-card server, 0.6

This is sufficient to describe the operation of a session as a module with these demands. Graphical notation is shown in Figure SJ.

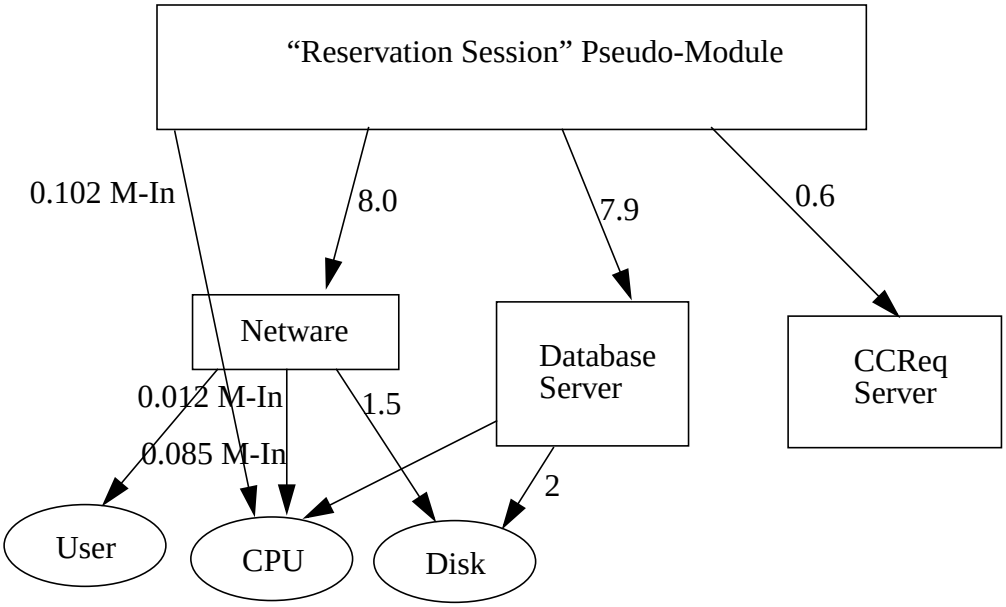


Figure 3.16. Local demands a Ticket "Reservation Session", derived from the activity graph in Step 1 and Step 2(a) of Figure 3.8 (SE). The pseudo-module "Reservation Session" is entered once per user session. (Figure SJ)

On the other hand if the Database and Netware Servers were considered to be internal to the Reservation System module, their demands would be included inside the pseudo-module as in Table 2 of Figure SD, and the average module demands are:

-- to CPU, 0.8695 million instructions
-- to disk, 27.8 disk operations
-- to credit-card server, 0.6 requests for verification
-- to user, 8.6 requests for input.

This is displayed graphically in Figure SK.

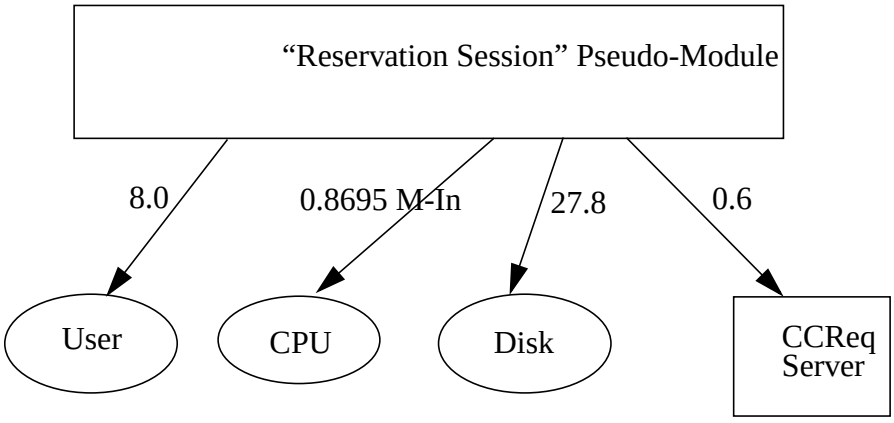


Figure 3.17. Entire demands for a User Session, derived from Step 2(b) of Figure 3.8 (SE) (Figure SK)

Another way to configure the system would have two disks, DiskA for the Netware server and DiskB for the database. Examining Figure SJ we see that the 27.8 disk operations divide into 12 to DiskA and 13.8 to DiskB. In this case, using the operation times in Table SE, the queueing model parameters are

D_{CPU} = 0.087 sec/session if $O_{CPU} = 10^{-7}$ sec, Y'_{CPU} = 0.8695 M-In

D_{DISKA} = 0.132 sec if O_{DISKA} = 0.011 sec, Y'_{DISKA} = 12

D_{DISKB} = 0.174 sec if O_{DISKB} = 0.011 sec, Y'_{DISKB} = 15.8

D_{CCReq} = 1.8 sec if O_{CCReq} = 3 sec, Y'_{CCReq} = 0.6

D_{USER} = 56 sec if O_{USER} = 7 sec, Y'_{USER} = 9

The queueing model in Figure HM is based on these demands, but divided by 8 and stated per response instead of per session.

1.4.3. Reservation System Module TicketRes

The example just completed reduced an activity graph by mechanical steps to a pseudo-module "Reservation Session". If we re-examine the activity graph we can see that it represents, not the behaviour of a software module, but the behaviour of a complete session of the reservation server interacting with the user. If we want to separate the software behaviour from the user behaviour we can identify three kinds of user requests to the reservation system: Connect, Interact

(described by the Choice block in the middle), and Disconnect. We can reorganize the activity graph of Figure SA into the form of Figure SKC, showing each kind of request as a high level activity, with six repetitions of “Interact”. Each high level activity becomes an entry to the new module TicketRes. Figure SKM shows the module TicketRes (with three entries) to represent the software part of the system, and the pseudo-module “Reservation Session” now representing a user session making requests into it.

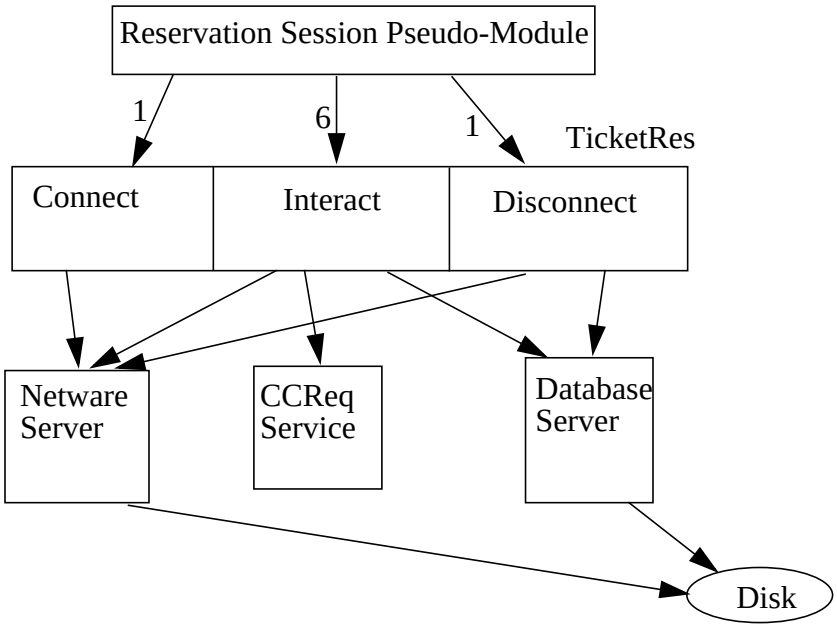


Figure 3.18. TicketRes Module (Represents the Software System separately from the user behaviour). (Figure SKM)

To obtain the parameters for the model in Figure SKM, we must restructure the activity graph in Figure SC with 4 parts:

- one for the user behaviour, making choices;
- one for each of the entries Connect, Interact and Disconnect.

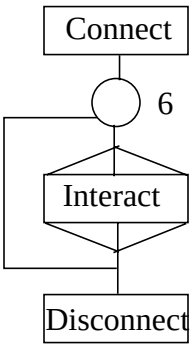


Figure 3.19. (Figure SKC)

For the user behaviour we obtain Figure SKC. Connect is a single activity, sufficiently described in Figure SC. Interact has a separate activity graph corresponding to the middle part of Figure SC, but with the looping cost lumped with the "choice" cost, with the activity data shown in Figure SKD. Disconnect consists of just the two final activities of Figure SC. The entry demands are summarized in Table SKN.

Interact	MeanTimes	CPU (M-In)	DBServer (ops)	Netware Server (ops)	CCReq (ops)
Choice	1	0.002			
Display	0.75	0.005	1	1	
Reserve	0.15	0.015	2	1	
Confirm	0.1	0.002		1	
Verify CC	0.1	0.004	1		1
Local Demands	$\sum K_i Y_i$	0.14	1.15	1	0.1

Figure 3.20. Activity Data for the 'Interact' part of the "Reservation Session" activity graph. (Figure SKD)

Assembling this information we can attach parameters to Figure SKM to obtain Figure SKP. The parameters are still in terms of operations rather than times.

Table 2: Table SKN -- Entry Demands for the TicketRes Module

	Total Local Service Demands				
Entry	CPU (M-In)	Disk	DB Server	Netware Server	CCReq Server
Connect	0.01			1	
Interact	0.014		1.15	1.1	0.1
Disconnect	0.008		1	1	0
DB Server	0.085	2			
Netware	0.012	1.5			

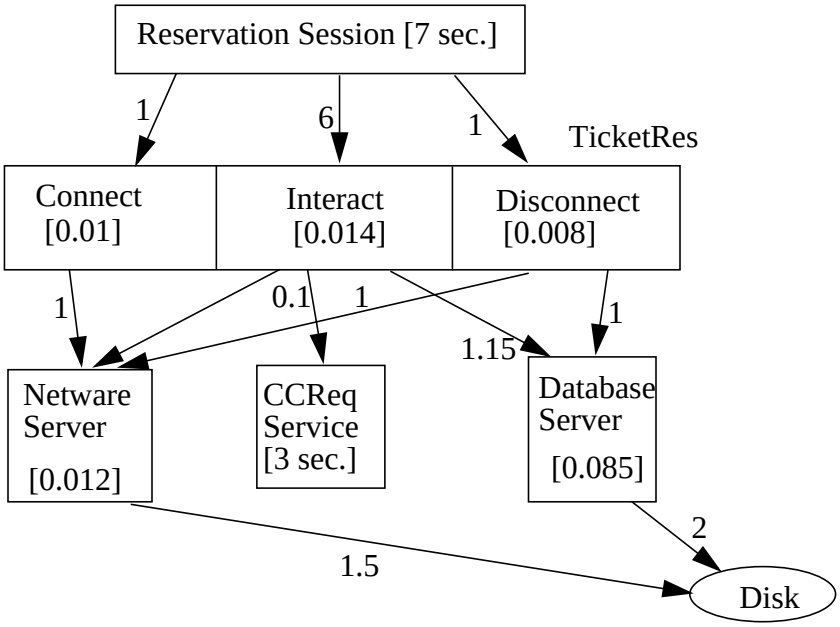


Figure 3.21. TicketRes Module with Logical Device Demand Parameters. (Figure SKP)

1.4.4. Obtaining Module Parameters Directly

We have been considering how to obtain module parameters by recording the plan for execution of each entry as an activity graph, and reducing its parameters to the request counts as given by Reduction *R2* in Section 3.3. On the other hand it may be possible to obtain the entry parameters directly, without the activity graph analysis. This section will describe the steps for direct estimation.

The entry parameters are the mean request counts to services and to the host device or devices. Some of the possibilities for direct estimation are:

- The module may exist, as for instance the file system, and its parameters can be measured. If a prototype exists, request count parameters can (perhaps) be measured from it.
- Data from a similar system or a previous version of the system can be used to estimate the parameters
- A parameter may be a simple property of the operation, such as the need to check the user privileges exactly once, for the print service, even if multiple documents are to be printed. Thus it can be taken from design documentation.
- A parameter may be a property of user behaviour (such as the number of pages in a document) which could change without warning, and which must be estimated as a basis for planning.
- Some parameters may be allocated as a budget to be achieved by the developers. The budget approach is often used with CPU time in developing an application, when a total CPU time

budget is divided among various modules to be developed in parallel.

Measurement of request counts can be done by tracing and by profiling. The UNIX gprof profiling utility counts all calls to a procedure from each other procedure, which is enough to capture counts for procedural-type requests. For requests which are made by system calls, UNIX tracing will record them but only on the basis of calls from within a certain process. Requests made by remote procedure calls or socket-based messages must be captured by software probes.

Execution times can be instrumented similarly. If an entry is represented by a particular procedure, then profiling is adequate. If an entry is associated with a branch in the program then it may be necessary to use software probes and a timing utility to get the processor demand for a given entry separately.

1.5. Multi-level Service Systems by Modules

In complex systems with many components the performance model must use *abstraction* to escape from excessive detail. Modules provide *controlled levels of abstraction* in the model, so that a designer can abstract away some aspects in order to focus on others.

MSS(Modules)

The module-modelling framework MSS(Modules) presented in this section rounds out the concepts and parameters needed to model linear software by modules. The previous section showed how to reduce an activity graph describing flow and behaviour, to a module or a module entry; here we will examine module models at different levels of detail, using an example of a Printing Service that has a rich modular structure.

MSS(Modules) is essentially a procedure call-graph model with workload parameters. Each entry has demand parameters for operations by devices and by other entries. When a module makes a request to a service its own execution is suspended until a return is received from the service. When an entry receives a request it begins to execute, and only terminates when it sends a reply or a return to its requester or caller. This represents procedure calls and also idealized Remote Procedure Calls (RPCs), which are implemented with request and reply messages between concurrent processes. It does not allow concurrent execution within a single response, however, and it does not represent logical resources. These will be introduced in the next chapter, in the extended framework MSS(Resources).

MSS(Modules) is based on long-establigsed (move) notions for modelling workloads based on call rates.

No Software Resources in MSS(Modules)

Because MSS(Modules) does not represent resource limits imposed within the software, the modules in this limited framework must not queue their requests. If two users request the same service at the same time, they both execute it. This implies that they both have the module linked to their own copy of the executable, or the module is fully re-entrant, or, if the module is a distinct process requested by RPCs, it forks a separate thread for each request.

Because the devices are the only resources, device queueing models are adequate for analyzing the system level performance. The value of the module breakdown is to see the detailed source of the workload, as it contributes to the device demands.

1.5.1. An Exploration of MSS(Modules): A Printing Service

To explore the performance semantics of the MSS(Modules) framework, consider an example with deep layering of services, shown in detail in Figure SM. It is a network printing service which is accessed by entry PrintService of a software component called PrintManager. This entry sends print jobs to a PrinterControl module, which in turn uses a PrinterDriver to send data to the printer. The PrinterDriver has two entries, one for control interactions and one for transferring page data, because its behaviour is quite different in these two cases. In the printer itself there is an embedded control program, for instance a postscript interpreter and print engine, controlling the printer hardware. It also has two entries, one for control and one for data. The PrintManager is also shown to store and retrieve data through a MgrInfo module with four entries, one to check that the user has printing privileges, one to store the printfile or files supplied by the user, one to search for a default printer if one was not specified, and then at the end of the print job, one to write the accounting log information to charge for the job. MgrInfo gets this information from the FileSystem, shown as a module offering an interface to the program as part of the operating system. This in turn uses network protocols and a network FileServer program, which accesses a Disk device through a Disk Controller. PrintManager passes the location of each file to print to the PrinterControl, which then accesses the stored printfile itself, one storage page at a time, processes it if necessary and passes it to the PrinterDriver.

The parameters on the arcs express the fact that one user request to PrintManager can pass one printfile, which is passed to the PrinterControl. Each file has *k* storage pages, which affects the number of accesses to the FileSystem, and the requests to pass data to the printer. Notice that PrinterControl passes *k* pages to PrintDriver.Data, and for each of these requests one page is passed to EmbeddedControl.Data

If a module has a single entry (or if only one of its entries is in the model) then it is represented by a node which is both a module and an entry. The idea of a layered service system implies that higher layers make requests of lower layers, which makes the graph acyclic (i.e., there are no loops in the graph). Recursive use of services is not represented in MSS(Modules). This is basically for simplicity, and an extension to represent recursive systems is straightforward.

Each module also uses one or more devices. Real software modules are intended to be loaded and run on one particular device, which we will call its host device. It might be a CPU, a smart terminal, a disk controller or a communications front end. The term “proper module” will be used for a module which makes requests for device operations to just one device, its host. Within this limit a proper module in the model may represent a single software module, or a subsystem that is made up of many software modules that all run on the same host. Figure SM shows proper modules with their host devices as ovals. There is one host device shown attached to each module, but the same host may appear in two places; if two modules are colocated they point to the same device. The Printer and Disk are separate devices (not hosts) represented by circles. FileSystem is a module that runs on WS1 and on WS2.

In Figure SM, if devices are shown then the arc weight from entry (*k.d*) to device *i* is the mean request count for device operations of the entry:

Y_i *(k.d)* = mean number of requests for operations by device *i*, during one invocation of entry (*k.d*)

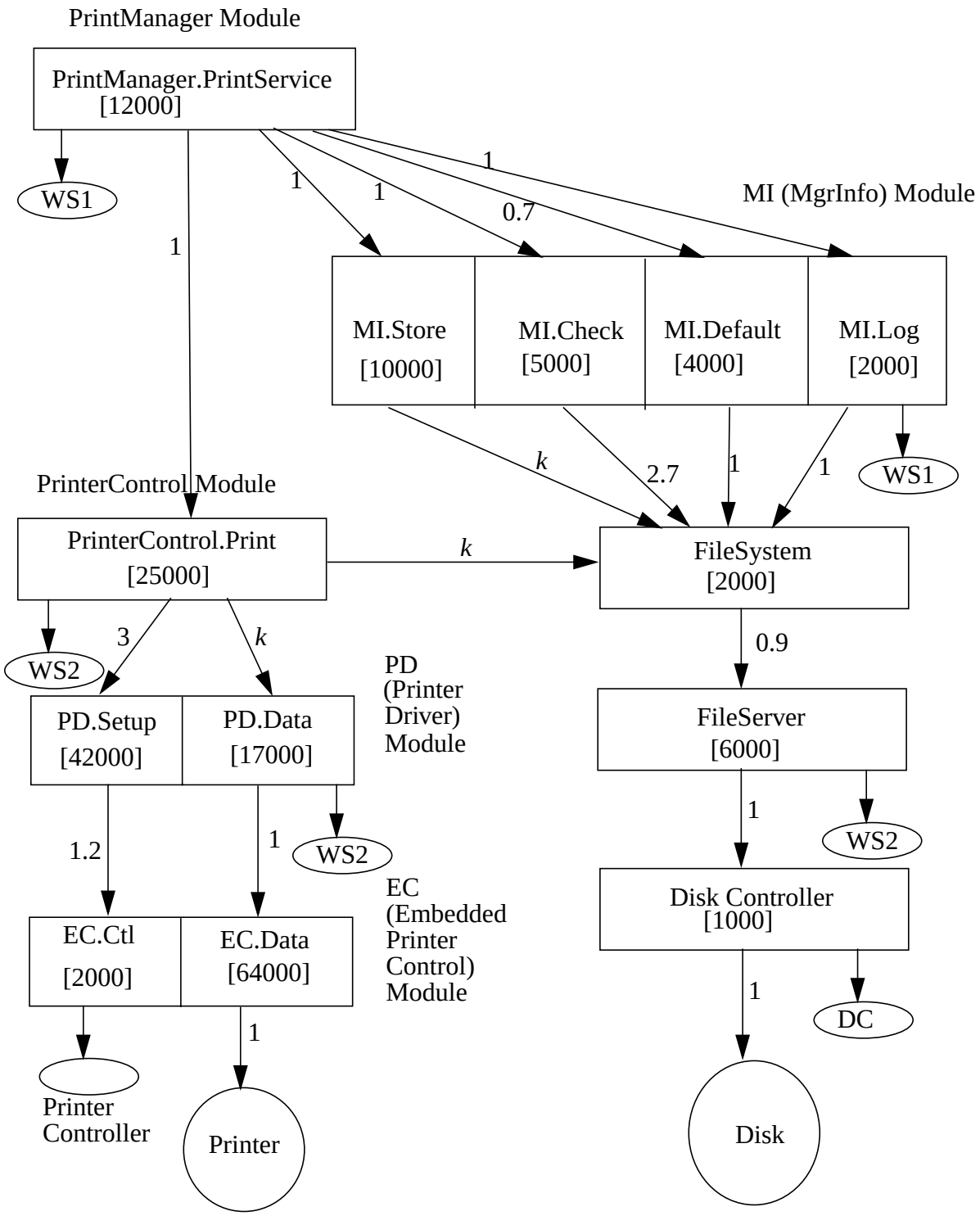


Figure 3.22. Module model of Printing service software. There are *k* file pages in the average job. (Figure SM).

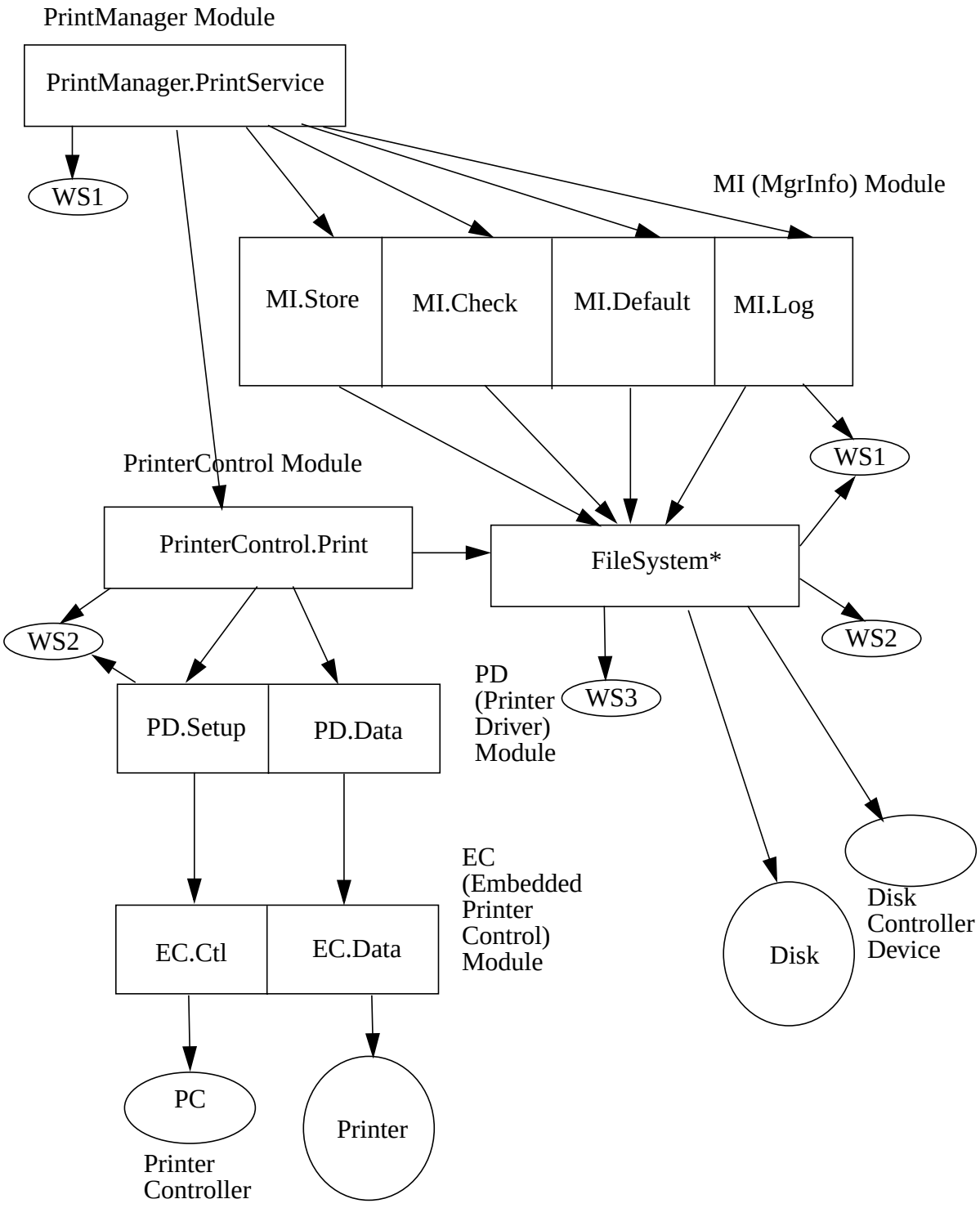


Figure 3.23. Printing service with devices shown. The file system has been aggregated to give a generalized module. (Figure SN)

Notice that a requests to devices and to other modules are shown in the same way, as arrows going to circles (for devices) or to boxes (for modules or module entries), labelled with the mean number of requests. A list of services used by the entry could include both types of descriptors interchangeably.

In a detailed MSS(Modules) model like Figure SM a program is represented entirely in terms of proper modules and their relationships. The detailed model can be aggregated by combining modules, even to the point where the entire program is represented by a single module. In this case it is probably not a proper module, because its submodules may well run on different hosts and the whole program runs on the entire collection of hosts. The term “generalized module” is applied to such an aggregate module that is distributed across several devices. The activities in the activity graphs in Section 3.2 have the attributes of generalized modules. In Figure SN the activities that make up the file system in the previous figure have been aggregated to give a generalized module called FileSystem*, which uses WS1, WS2, WS3, the disk controller, and the disk itself.

1.5.2. Changing the Level of Abstraction: Aggregation

Once a system like the Printing Service is fully understood and the analysis moves on to other issues, it is useful to be able to hide its complexity by aggregating its modules. Any group of modules can be aggregated into a single artificial “supermodule” which exists only in the performance model, where it represents the workload effects of the software in the group of modules. In the limit, if one aggregates all the modules used by a class of users, one arrives at a supermodule representing the total demands per response used in Chapter H. (This is addressed later by Reduction *R4*). However if one aggregates into a moderate number of large modules for separate subsystems, one can study in greater detail the impact and contribution of each subsystem. This in turn can focus design effort where it is needed.

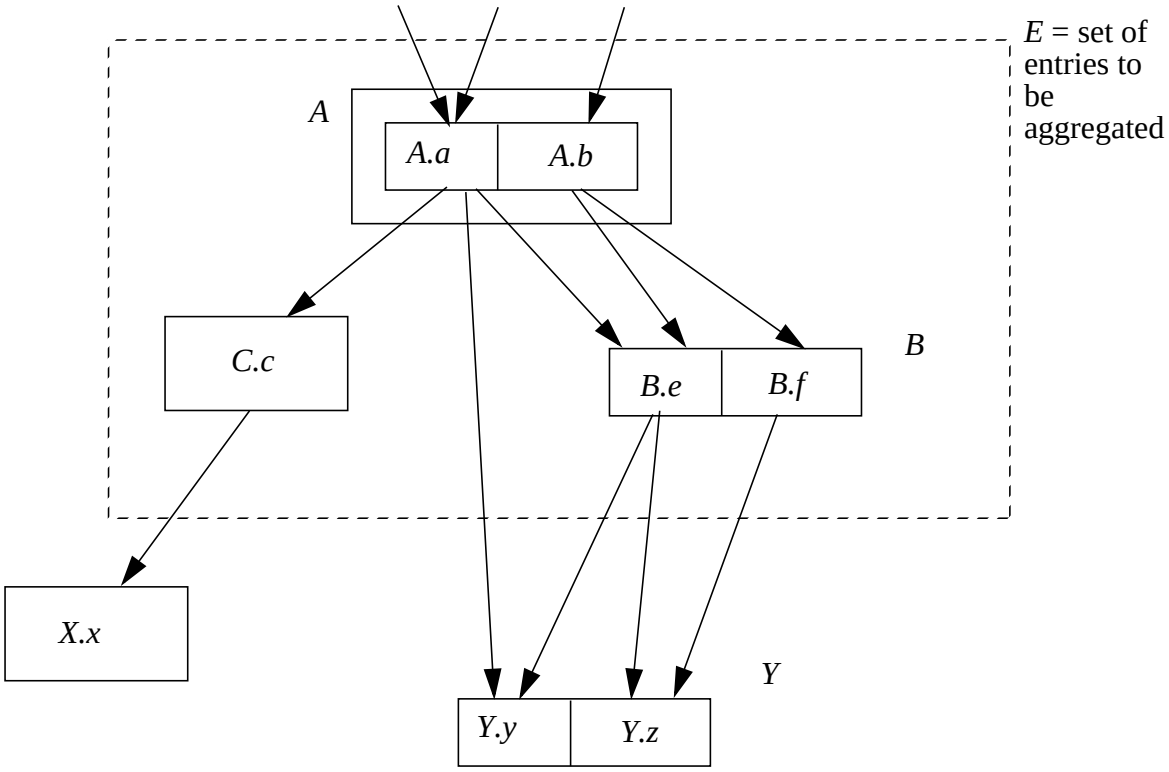
It is essential, in large systems, to be able to move up and down a ladder of different levels of aggregation. Some performance issues require details to be exposed, while some can only be understood at a large perspective. The following Reduction *R3* gives a controlled level of abstraction.

1.5.2.1. Aggregation of Modules (Reduction *R3*)

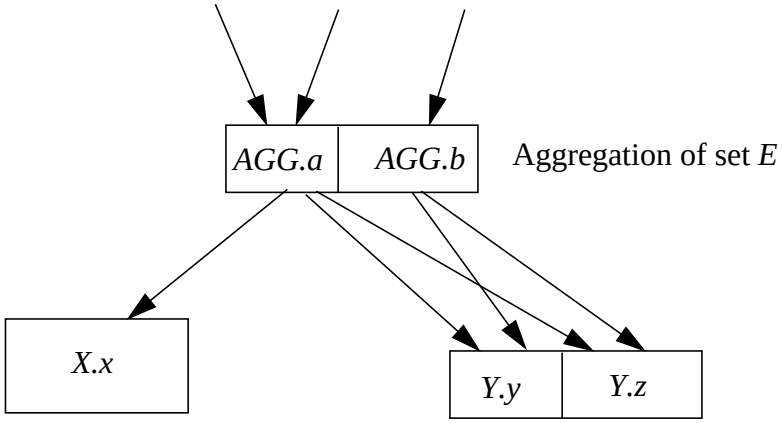
First consider aggregation as an abstract question, as illustrated in Figures SR and ST. There is a group E of modules A,B,C (with various entries *A.a*, *A.b*, etc.) to be aggregated into a single “super module” called *AGG*. What entries will *AGG* have, and how will their parameters be determined?

A special simple case occurs when the interface offered by the group is only through a single “top-level” module, as in Figure SR, however we can equally well have a situation where the group offers external access to two or more of its internal modules, as in Figure ST. Notice that an entry which is accessed externally may also be accessed internally, by calls from modules within the set (provided this does not introduce a cycle into the graph), as shown at module B in Figure ST. The algorithm to be described will handle all these situations.

The goal of aggregation is to find the mean request count for services demanded by the set of modules, for each invocation of a service offered by the set. These sets can be denoted by:



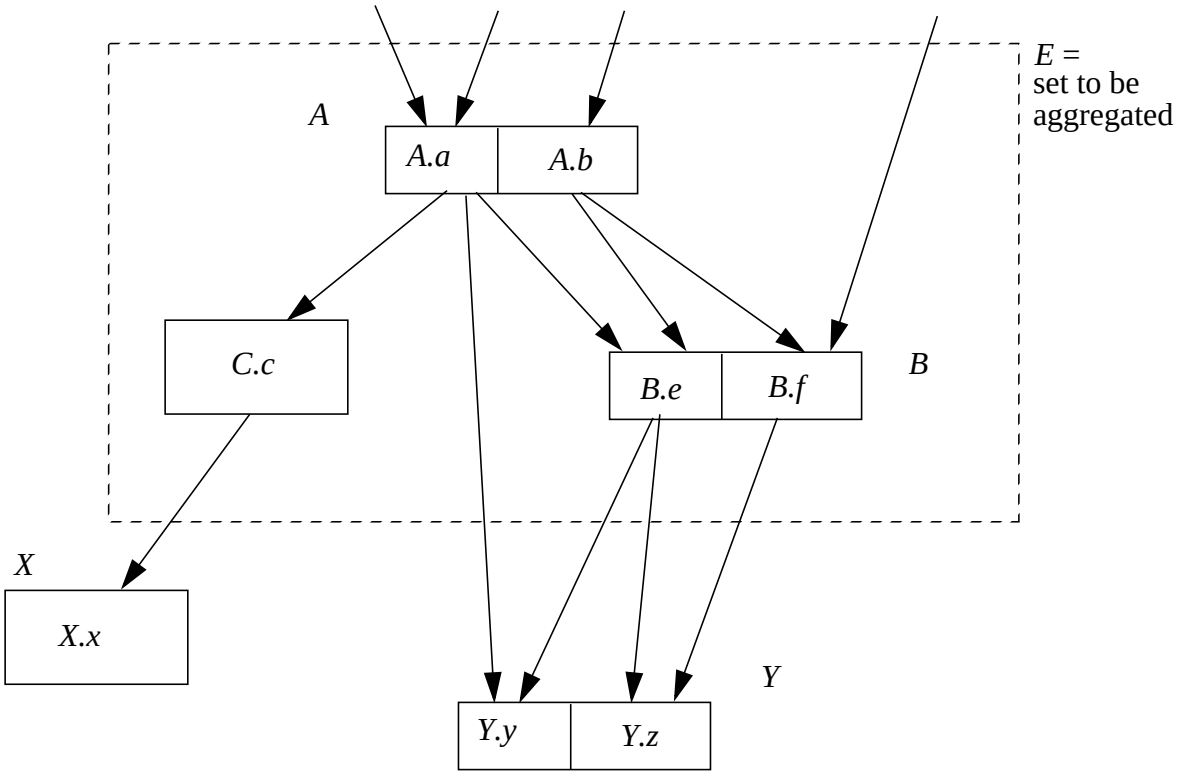
(a) Modules to be Aggregated: Access via only one Module *A*



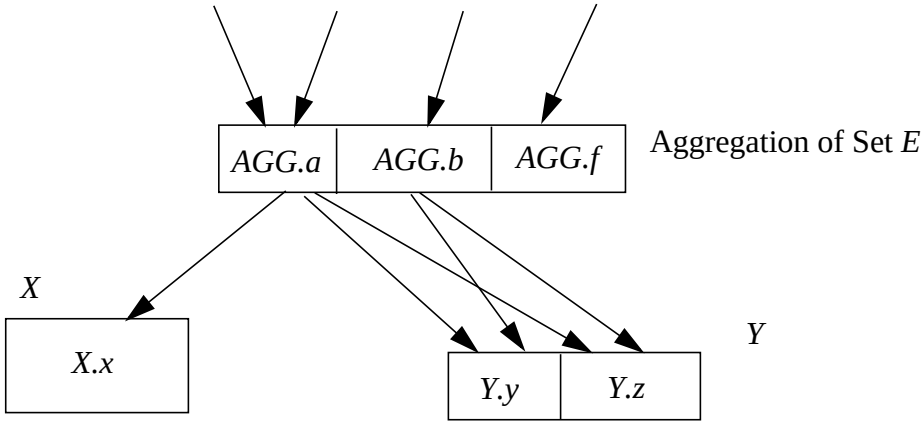
(b) Aggregated Version

Figure 3.24. Aggregation with Single Access (access into the set via one module, Module *A*). (Figure SR)

E = set of entries of the modules to be aggregated, with elements (*m.e*)



(a) Modules to be Aggregated: Access via both Modules *A* and *B*



(b) Aggregated Version, Combining Entries that are Accessed

Figure 3.25. Aggregation with Multiple Access (access into the set via more than one module, Modules *A* and *B*). (Figure ST)

J = set of entries outside of E offering services used by entries in E, with typical element j,
I = set of entries in E that offer services to other modules outside of E,

The algorithm *R3* to aggregate the set of modules into an aggregate module *AGG* is as follows:

1. Order the entries of E in a list such that the first one makes no requests inside E, and later ones make requests only to entries or services that are either not in E, or are earlier in the list. This can always be done because the graph of entries is acyclic.

2. For every entry $m.e$ in E that makes a request to an entry in J, set $Y_j'(m.e) = Y_j(m.e)$. These will become the reduced request counts.

3. Consider the entries in E in order, starting with the first in the list.

4. For each entry $(k.d)$ in E, consider all requests from it to other entries $(m.e)$ in E, with local request count $Y_{(m.e)}(k.d)$. Augment the reduced request count $Y_j'(k.d)$ by the amount:

$$\sum_{(m \cdot e) \in E} Y_{(m \cdot e)}(k \cdot d) Y'_j(m \cdot e)$$

When this is completed every entry in E has a set of reduced request counts to entries in J. Those entries in I, that are visible to the users of the aggregated module, become the entries of the aggregated module, and the rest are hidden. Thus if entry $(m.e)$ in E becomes the entry $(AGG.e)$ of the new aggregated module *AGG*, then the new local request counts for $(AGG.e)$ are the reduced counts of $(m.e)$:

$$Y_j(AGG \cdot e) = Y'_j(m \cdot e) \quad \text{for } j \in J$$

This aggregation has been described as if the demands to other modules are all requests to software services, but it applies equally to requests to hardware services, which are automatically included in the set J. The reduced mean request counts to hardware services are, as before, the requests for logical services offered by the devices.

Figure SU shows an example of aggregation based on the printing service from Figure SM. The parameter k for the number of disk pages in the file to be printed has been set to 3.0. In Figure SU(a), there are two clusters of modules to be aggregated,

- AGGPrintMan containing PrintManager and MgrInfo, and
- AGGPrintCon containing PrintControl and PrinterDriver.

After aggregation, AGGPrintMan will offer entry PrintService, and AGGPrintCon will offer entry Print.

In Figure SU the demands to the host device of each module are also shown. A special convention is used to avoid showing all the host devices in the diagram. A parameter in square brackets is given inside each entry, with the demand value, as: $[Y_{host}(m.e)]$. The units of host demand must be stated with the diagram; here they are logical service demands in instructions, but

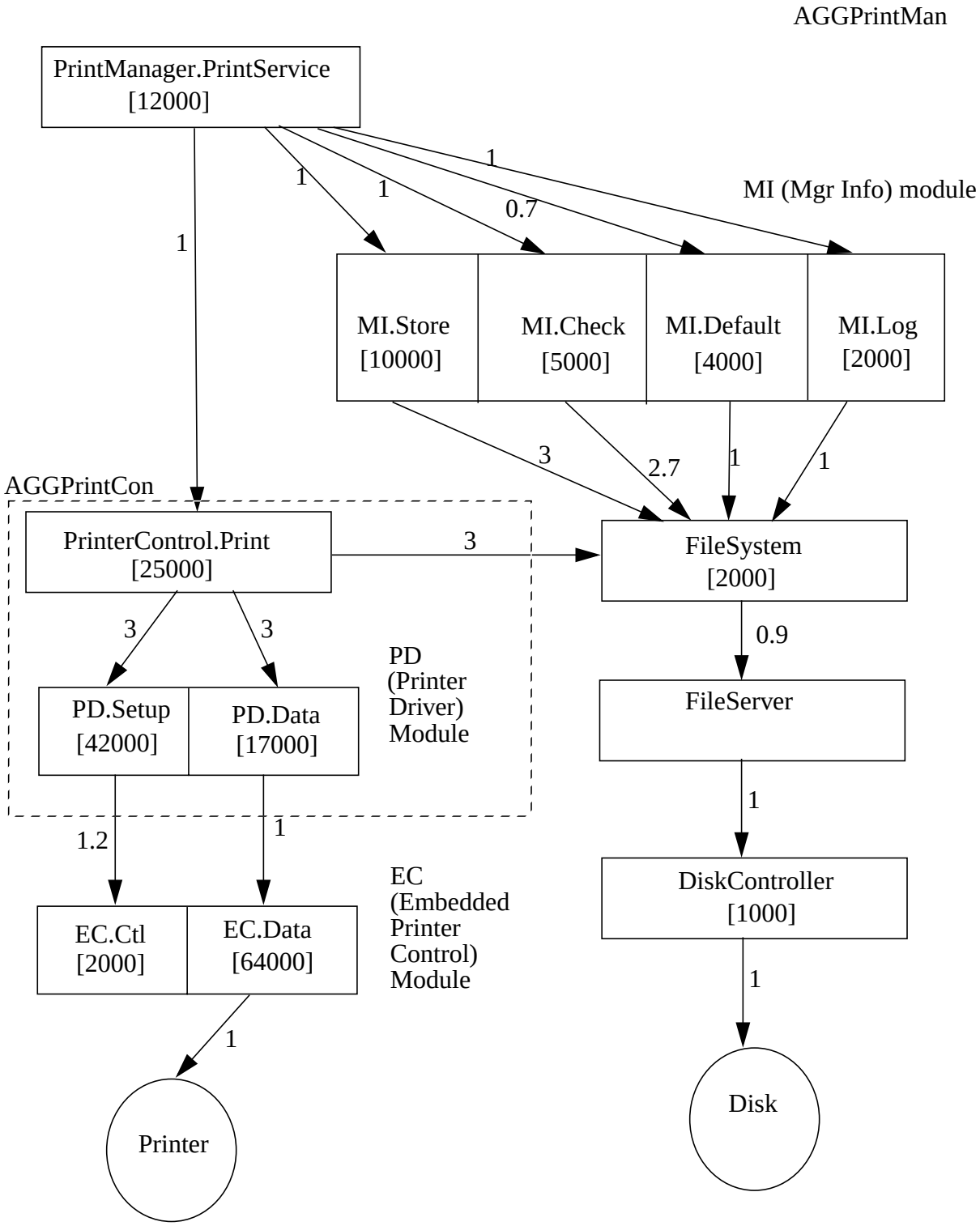


Figure 3.26. (a) Print Service: Identification of Modules for Aggregation, for $k = 3$. (Figure SU(a))

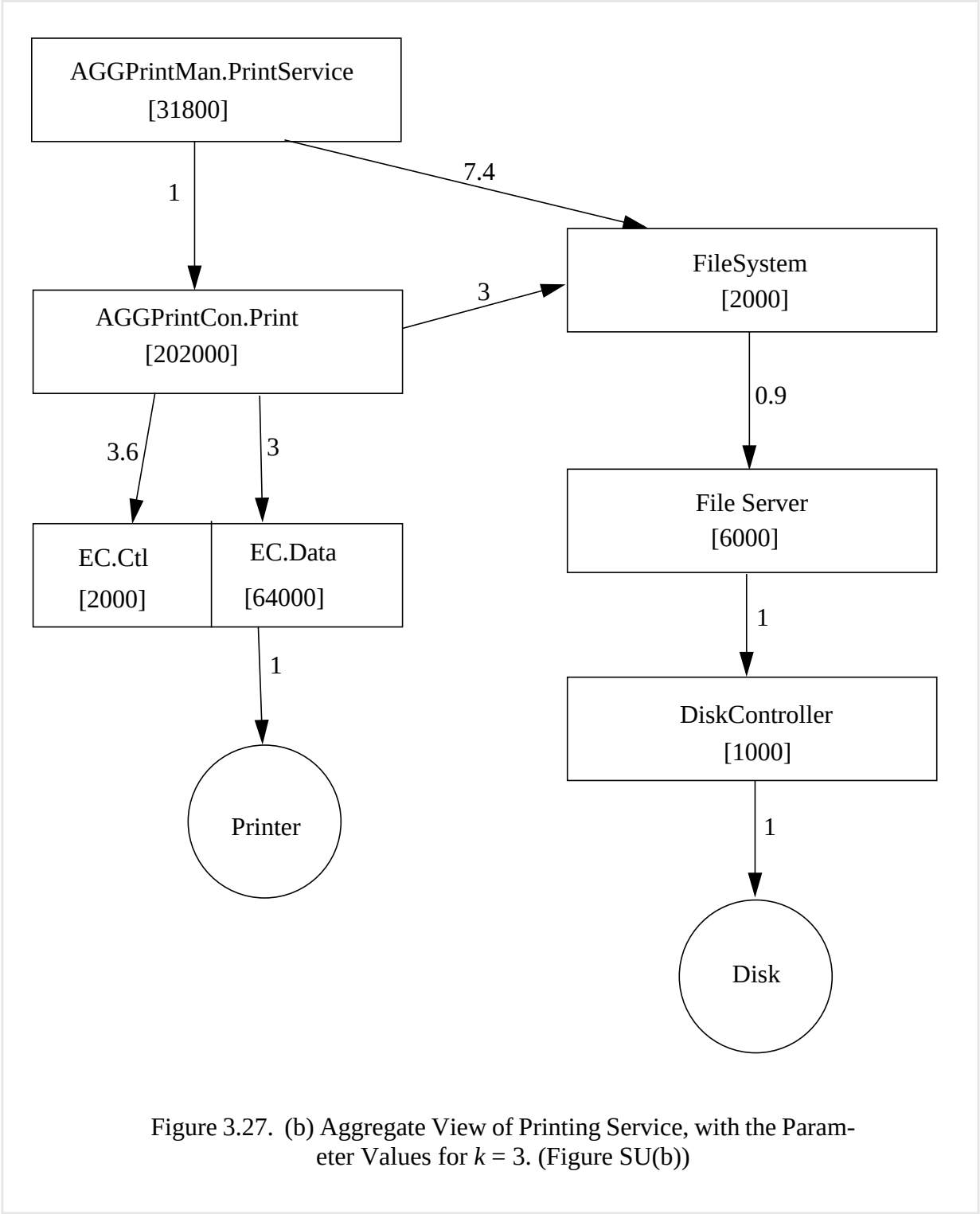


Figure 3.27. (b) Aggregate View of Printing Service, with the Parameter Values for $k = 3$. (Figure SU(b))

in other diagrams they might be given as the time demands, in some suitable unit such as seconds or milliseconds. Figure SV illustrates the request count to a host device with the device shown, and the equivalent diagram with the request parameter in brackets.

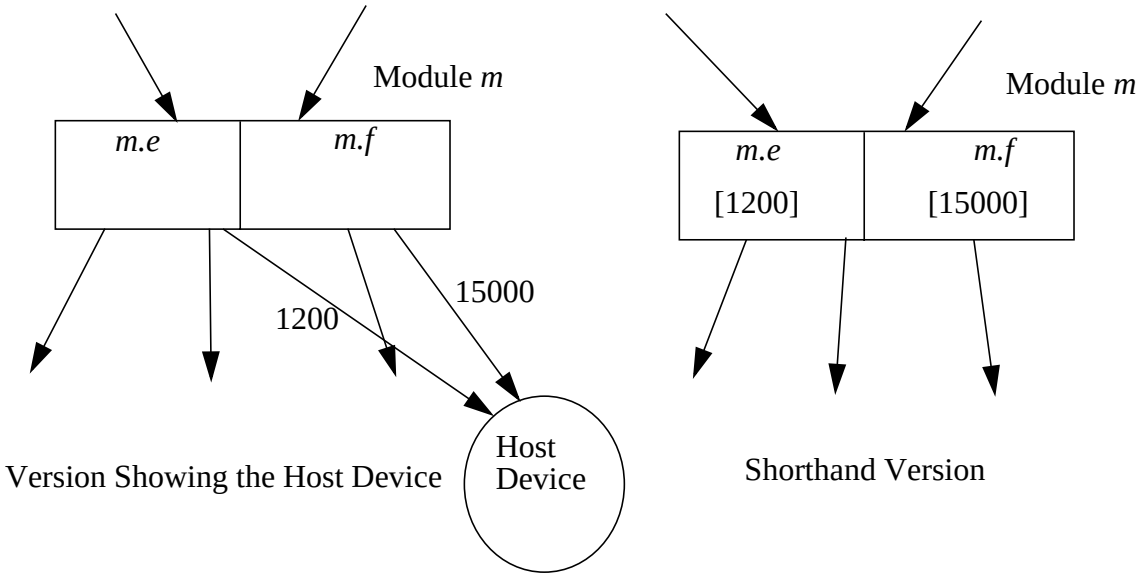


Figure 3.28. The Shorthand for Host Device Request Counts (the parameter in the brackets may also be demand D in units of time). (Figure SV)

Figure SU(b) shows the aggregated module model, found by applying aggregation algorithm R3. To explain just two of the parameters in the reduced model,

$Y_{FileSystem}$(AGGPrintMan.PrintService) = (1 x 3) + (1 x 2.7) + (0.7 x 1) + (1 x 1) = 7.4 = mean requests to the file service on all paths from the entry through the MgrInfo module.

Y_{host}(AGGPrintCon.Print) = 25000 + (3 x 42000) + (3 x 17000) = 202 000 = mean processor instructions per invocation of the Print entry, including the driver code.

These values can be found by applying Reduction R3 step by step.

Note that the fact that the graph of entries is acyclic does not prevent an entry from making a request to another entry of the same module (a well-known feature of object-oriented programming, for instance, when a method uses another method of the same object).

1.5.2.2. Reduction to Obtain Hardware Demands of a Module Entry, and a Complete Program

A single module entry is easily analyzed down to its device demands by Reduction *R4*.

Reduction *R4*: Let the designated entry be entry e, and define the sets

E = all entries called directly or indirectly be e,

J = the set of devices used by e and all entries in E,

I = the entry e alone.

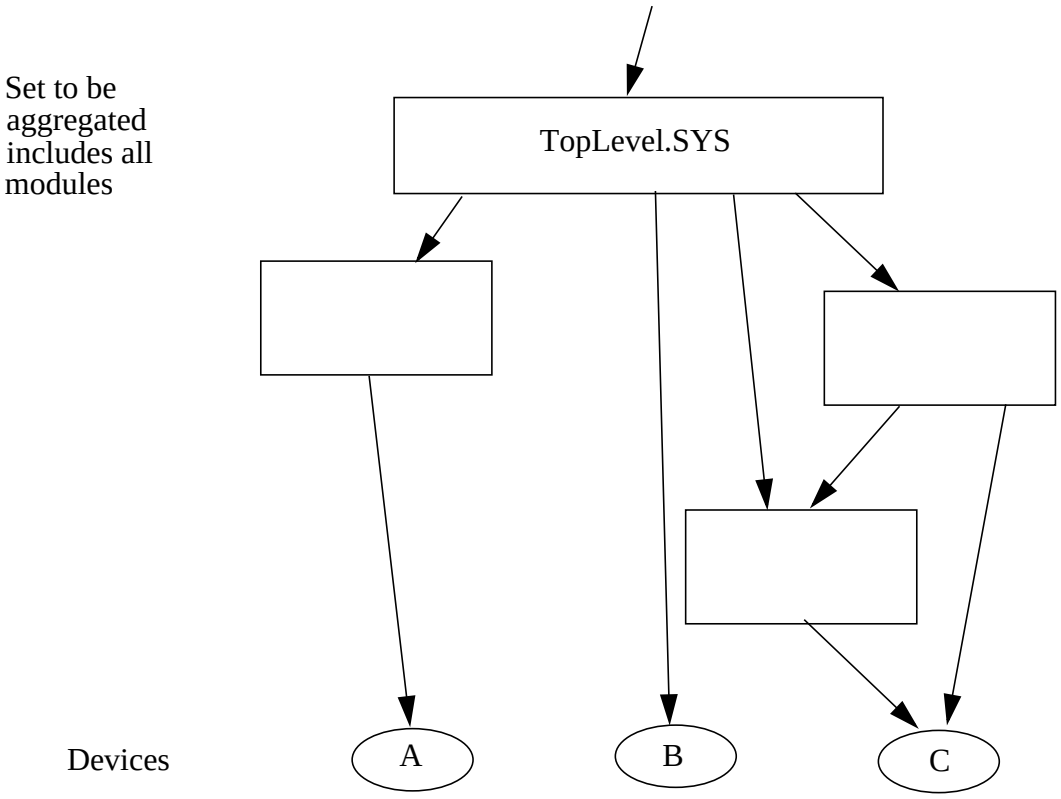
Apply Reduction *R3* to this system. The resulting mean request count $Y_j'(e)$ are the total hardware demands of one invocation of entry e. To obtain the service demand D_j, multiply $D_j(e) = Y_j'(e)O_j$, where O_j is the operation time of device j.

A complete program can be described as a kind of module, such that a user executes the top-level module and the rest of the workload is described by module relationships. The reduction process described for module aggregation can be extended to give only demands for the devices, when the top level module is executed. Suppose the top level module has just one entry (call it entry SYS), and all other modules are accessed via the top level module, as indicated in Figure SX. Then Reduction R4 gives the demands as $Y_j'(SYS)$ requests and $D_j(SYS)$ service demand.

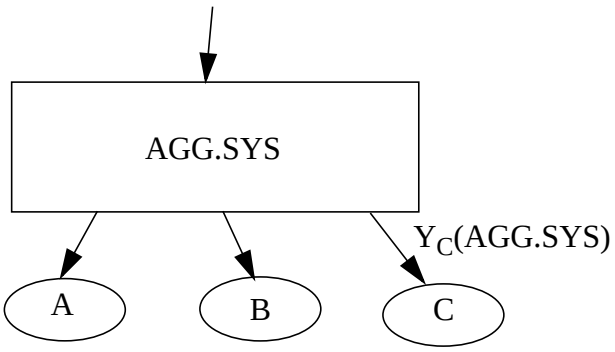
The print service example can be carried this one extra step to illustrate the above points, beginning from the aggregated version in Figure SU(b). First we have to identify actual devices for the host devices; suppose PrintMan runs on workstation WS1, PrintCon runs on WS2, and the FileServer runs on WS3, and all these workstations have operation times of one microsecond. For hardware loadings it is clearer to show the host devices explicitly, as in Figure SY(a). Figure SY(a) reproduces Figure SU(b) except for dropping the prefix AGG in the aggregated module names, and showing the host devices.

One thing that is exposed by showing the host devices is that one module may need to have copies for different hosts, that is the copies run on different workstations. The example here is the FileSystem module, which is the local interface on each workstation to the distributed file service; it must run on both WS1 and WS2, and different request counts go to each copy. In Figure SY the copies are called FS1 and FS2, and the breakdown of requests is shown. The host devices all have operation times of a microsecond, but the disk and printer have longer operation times, which will be shown in the next step.

In Figure SY(a) the dotted box includes all the software modules; applying Reduction R4 gives a notional generalized module and entry we can call PrintJob.PrintService (corresponding to SYS identified in the algorithm for R4), which represents the workload of a single print request. The algorithm gives the reduced model and device request count parameters shown in Figure SY(b). The last step is to multiply the request counts by the device operation times, which are one microsecond for the host devices, 100 millisec. for the printer, and 14 millisec for the disk. This gives the demands D_j shown below each device in Figure SY(b).



(a) Module Model with Devices



(b) Aggregated Module

Figure 3.29. Aggregation to Obtain Hardware Demands from a Module Model. (Figure SX)

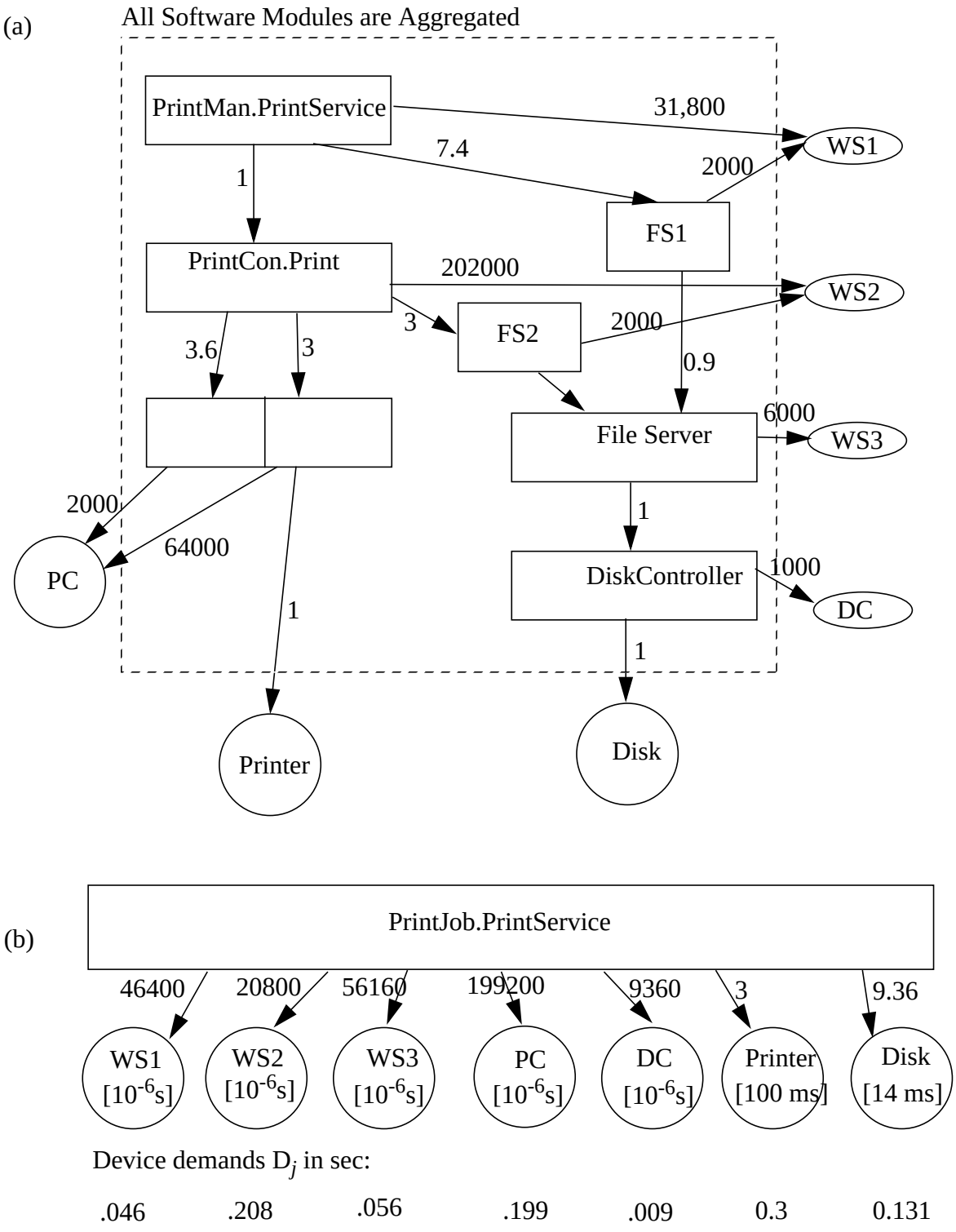


Figure 3.30. Print Service (with Value *k*=3) after Reduction R4, with Host Devices and Service Times Shown. (Figure SY)

From the demands, a queueing model can be constructed at once, following the methods of Chapter H. We will first consider a flow of print requests at a given rate *f*/sec, and no interference from other processing (a rather artificial assumption). The capacity is limited to a rate of 3.33 requests per sec by the largest value of D_j, which is at the Printer device (D = 0.3 sec. per request). The response delay of a print job, as a function of *f*, is shown in Figure SZ, based on a Poisson request flow and ./M/1 servers. Considering the high variability of the size of print jobs and the work that they cause at the heavily-loaded, dominant devices, this may be a reasonable approximation.

Figure 3.31. Response Delay of Print Jobs at a Fixed Request Rate, with No Other Workload. (Figure SZ -- blank, to be done in Matlab)

1.5.3. Introducing Detail

It is also possible to begin the analysis at a coarse level of detail, and refine it later. Coarse-level supermodule parameters can be obtained directly by measurements on the entire subsystem, as described in section 3.4.4, and used for analysis without considering internal details of the subsystem.

To refine the details of a supermodule into finer-grained modules requires a new study of its internals. It is done by making new measurements that distinguish the demands of separate parts, and request counts between them combined with activity graph analysis for behaviour that is planned but not yet measured. One can also refine the entries of a module or supermodule, while keeping the same module level, by estimating the workload demands of different types of requests.

The only “easy” way to move to fine detail is to keep a memory of an initial detailed analysis which was then aggregated, and to return to it.

1.6. A Complete System Model

To use all this analysis to make performance predictions, one has to represent in some way the rest of the workload of the network and the file server. As before, a simple version of the remaining workload is an additional class or classes that execute user application processing on the CPU and generate file server requests. This additional processing would compete with the print requests. A second version would add the “other processing” to the workload of the printing service users themselves.

First consider a complete model with a class of printing users, and a class of other users who do not print. This is similar to the completion of the Reservation System model in Section 2.3.5.

The MSS(Modules) reduction *R4* in the previous section and Figure SY has given the workload demand parameters of class 1 for printing users. The demand parameters of class 2 must be determined. As we are considering a network environment, we may suppose that each other user has a workstation and computer independently. Only when they make a file server request, do they compete with the printing users.

The following reasoning will usually give a satisfactory model for “other users” belonging to class 2:

- their workstation time (thinking, computing, access to local workstation disk) is a pure delay $Z(2)$ sec, between file server requests. $Z(2)$ is the time from a reply to one request, to making the next. It is almost (but not quite) the same as 1/(file service request rate per user);
- the file service demand parameters are found per request. They include operation demands to the file server CPU (*WS3* in our Print Server model), and to the file server disk (or disks).

Another aspect of completing the model is to include demands to other devices, particularly network devices. This is important if the network is a bottleneck. Otherwise it is sufficient to work out network delays for message sending, and add a delay station that causes a delay Z_{Net} to the sender of a message.

A representative set of parameters are given in Table SZA and performance results for the printing service with different class populations are given in Table SZB.

Table 3: Device Demands for Computing and Printing (Table SZA)

Device	Demands for Class 1	Demands for one FileRequest by User in class 2
WS1	0.46	0
WS2	.208	0
WS3	.056	0.015
PC	.199	0
DC	.009	0.002
Printer	.300	0
Disk	.131	0.030
Local WS Delay	0	1.0
Total (D)	0.949	.047
Delay (Z)	0	1.0

Table 4: Performance Results for the Printing Service with N1 Printing Users and N2 other Users (Table SZB)

Paragraph on the Results TO COME.

1.7. Patterns in Module Architectures

Our goal is to recognize POPs and to react to them, however the real opportunities are still to come, with concurrency and software resources. At the level of purely linear software discussed so far the opportunities are limited, and are largely concerned with reducing the demands required for an operation.

A module architecture shows which objects participate in an operation, and how often each intermodule request is made. A pattern may be any subgraph of modules and requests, and patterns are distinguished mainly by their depth and breadth. Again we concentrate on sequential execution within a single process, and on reducing total resource demands by changing patterns. Two types of pattern changes may be used:

- substitution of a pattern that results in lower demands, for a pattern existing in the architecture. As shown in Figure SVD, the substitute must satisfy the same interface but may use different lower level modules, which become part of the new architecture. This is an option when designing with replaceable components.
- aggregation or inlining of a lower module into one that calls it, to eliminate the calling overhead in both modules. This can be important for a very small module which is called very often.

The decision is governed by the module parameter estimates. To evaluate a substitution, compare the entire demands Y_i'(entry) of all the entries of the original pattern to those of the substitute. The entire demands are computed by applying the R3 reduction for each submodel as if one was aggregating all the modules used by the entries of the pattern. If all the resource demands are reduced the evaluation clearly indicates using the substitute; if some are reduced and others increased the factors considered for activity patterns are used here also. A model is constructed and solved to determine if the net gain is positive and large enough to warrant the change.

For inlining or aggregation, the results may be tiny or may be quite large. Large gains can be made if the modules provide tiny fine-grained functions and are called many times. Consider a procedure which makes y calls to a second procedure, such that the overhead in each one is a fraction α of the "useful work"; further suppose that the CPU demand of the useful work is the same in both procedures (call it Y). The entire CPU demands for the original and the inlined case are:

original: $Y(\alpha + 1 + y\,\alpha) + y\,Y\,(\alpha + 1) = Y\,(1 + y + \alpha + 2y\,\alpha)$

inlined: $Y\,(\alpha + 1 + y)$

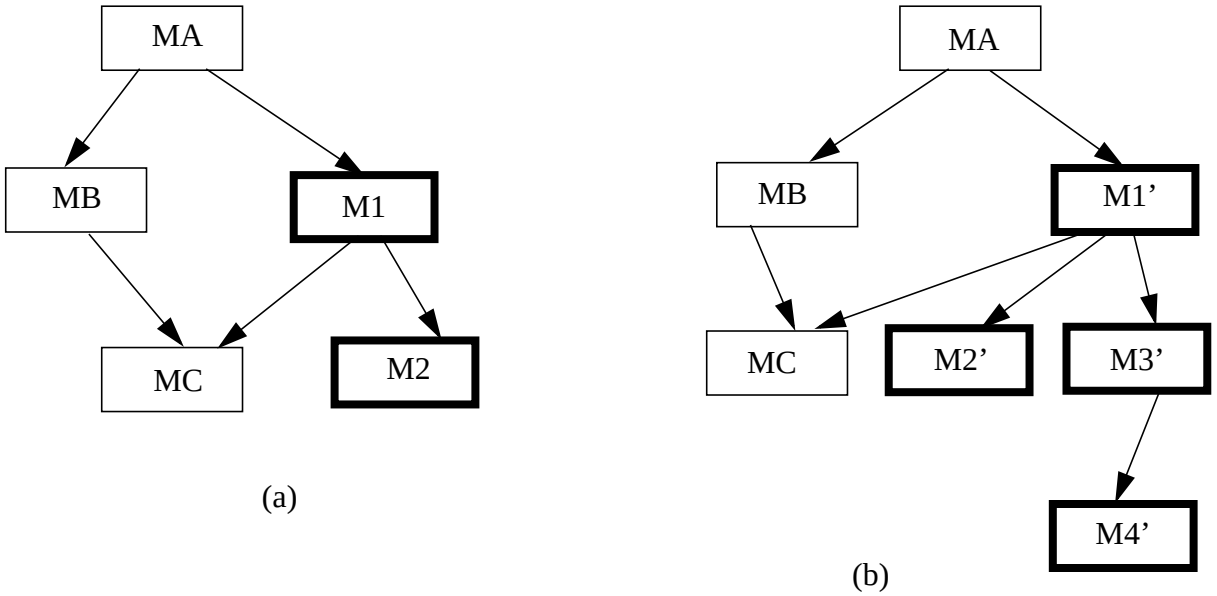


Figure 3.32. Substituting one Module Pattern for Another. (Figure SVD)

Ratio = [α + 1 + *y*] / [1+ *y* + α + *2y* α] ~ 1 / [1 + 2 α], for large *y*

For a deeper calling stack, say of *n* similar procedures,

original: $Y(\alpha + 1 + y\,\alpha)\,(1 + y + ... + y^{n-2}) + y^{n-1}\,Y\,(\alpha + 1)$

inlined: $Y\,(\alpha + 1 + y + y^2 + ... + y^{n-1})$

which gives the same asymptotic value of Ratio = 1 / [1 + 2 α] for large *y*. We conclude that the gain from inlining is dominated by the relationship of calling overhead to useful work, represented by α. The number of calls, controlled by *y*, affects the total size of the term due to this pattern but not the fractional improvement.

Richer and more interesting effects for module patterns are seen when the modules can be concurrent processes, as will be evident in Chapter P.

1.7.1. Controlling “Bloat”

CPU demand “bloat” occurs as software is maintained and evolved over several releases. New features and allowances for new types of devices or services introduce overheads to select or enable the feature or device. For example each time a new feature affects the program flow there may be a test to see if it is enabled. Deeper class hierarchies introduce additional inheritance overhead, following pointer chains to find the code to be executed for each method. Sometimes a new feature is based on a new architectural abstraction that requires on-the-fly translation from existing data structures or command architectures.

Often, in adding features, there are many possible ways to proceed. If the redesign is chosen only for quick programming it may not be the best for controlling bloat. If performance effects are predicted then an acceptable redesign can be found.

As an example, consider a generalized Printing Service that can manage several printers at once, of different types. Each printer has its own driver, possibly in a different host workstation. A decision is made to keep one PrinterControl module which interleaves the operations for all the printers, and sends messages to their drivers. Now this module must keep track of the state of each printer and handle messages from all of the drivers (indicating ready for more data, “error” indications, “done”). A layer of software must be added to PrinterControl.Print to decide, for every message it receives, which printer it is for. The extra decisions, access to the additional state, and the cost of messaging to the Print Driver modules, all contribute to bloat.

Additionally this extended module must also respond to calls from PrintManager, so its interfaces and control are even more complex.

1.8. Software Design Issues within the Linear MSS(Modules) Framework

MSS(Modules) is a performance analysis framework for a broad class of software that we have called *linear*. It includes all classical sequential programs, and also complex systems of concurrent processes communicating by RPCs. The availability of concurrency and parallelism has not led to a tide of non-linear software designs, rather the opposite. System and language designers have attempted to deal with concurrency by supporting linear software that combines many processes, doing one thing at a time.

1.8.1. Potential for Concurrency in Linear Software

Linear software can be structured as a set of several concurrent tasks communicating by messages, either by RPCs in which the sending task suspends until it receives a reply, or by asynchronous “hand-over” messages. There is a cost for this partitioning into concurrent tasks, which is essentially the computing cost of message handling, the additional overhead to schedule additional tasks, and message delays. These costs are often considerable.

Concurrent processes for the parts of an application have a definite role where the parts are in separate places, by necessity (perhaps due to external interfaces in different places, or geographic distribution of information), or where there is no distributed operating system (as in a network of Workstations or in some small simple real-time kernels). These systems are becoming more common as systems migrate onto networks, and local networks evolve into Intranets. The next chapter studies linear versions of these systems as “ideal RPC” systems.

The performance advantages of concurrency within linear software are doubtful. It does break a large execution into smaller parts, which may allow it to fit into a group of smaller computers. It permits pipelining, which may have performance advantages if some stages have specialized requirements. And it promotes flexibility of configurations. Overall however the performance advantages of concurrency lead away from linear software, towards parallelism of various kinds.

There is also a zone of interaction between software design and the capabilities of a distributed operating system. There is no need to statically allocate the concurrent tasks if the operating system can dispatch them to any idle processor of a “symmetric” multiprocessor system.

If static allocation has an advantage it may be in reduced system bus traffic (each processor can have its “own” memory) and better cache efficiency. In a symmetric multiprocessor it may not even be helpful to divide a linear application into concurrent tasks, for if each user or response has just one task this may provide sufficient concurrency.

1.9. Additional Reading

BIBLIOGRAPHY

[CUS90] C.U. Smith, “Performance Engineering of Software Systems”, Addison-Wesley, 1990

[OOSE] I. Jacobson, M. Christerson, P. Jonsson and G. Overgaard, “Object-Oriented Software Engineering: A Use Case Driven Approach”, Addison-Wesley Publishing Co., 1992.

[booth&weicek] Taylor L. Booth and Cheryl A. Wiecek, “Performance Abstract Data Types as a Tool in Software Performance Analysis and Design”, IEEE Transactions on Software Engineering, pp. 138--151, March 1980, v6, n.2.

[pearls1,] J. Bentley, “Programming Pearls”, Addison-Wesley Publishing Company, 1989.

HOPEsem95] C. Cowan and H. Lutfiyya, “Formal Semantics for Expressing Optimism: the Meaning of HOPE”, in Proc. of the 14th Annual ACM Symposium on Principles of Distributed Computing”, Ottawa, Canada, pp. 164-173, August, 1995.

[bubenik] R. Bubenik and W. Zwaenepoel, “Semantics of Optimistic Computation”, in Proc. of the 10th International Conference on Distributed Computing Systems, pp. 20-27, 1990.

[stromyemini] R.E. Strom and S. Yemini, “Optimistic Recovery in Distributed Systems”, ACM Transactions on Computer Systems, pp. 204-226, v3, n3, August, 1985.

[TCPspeed89] D.D. Clark, V. Jacobson, J. Romkey and H. Salwen, “An Analysis of TCP Processing Overhead”, IEEE Communications Magazine, v27, n6, pp. 23-29, June 1989.

DRAFT DRAFT DRAFT DRAFT Performance - Oriented Patterns in Software Design (A multi-level service approach)

C. M. Woodside

Dept. of Systems and Computer Engineering

Carleton University, Ottawa K1S 5B6

Draft of October 18, 2001

Chapter 4. Distributed Linear Software (L)

4.1. Introduction

Much of the software written for distributed systems is linear, like the software described in Chapters 2 and 3, or is nearly linear. It is linear because it is adapted from sequential software written for a single system, or because it is easier to write software with a single thread of control.

The most common computing environment is now a network with one or several servers for the file system, email, web pages, printers and so on. Even a simple application programmed for a single PC will in fact use several other nodes. In these distributed environments there are nodes connected by a network; each node is a distinct computer, with I/O devices (hard disk drive, CD-ROM, video) and processor (possibly more than one). The “environment” that an application designer must consider is the set of nodes that will run the application, and these may be all in one place connected by a single LAN such as an office ethernet, or may be scattered and connected by routers, gateways, and the Internet. In most of these environments every node can send a message to every other node, although the speed and latency may not be the same on all these paths.

Examples of distributed applications that use linear software are in office computing, data base access, transaction processing, data warehousing, decision support, computer aided design, and multimedia conferencing. Some examples match two or more of these categories, as in a decision support system accessing one or more databases.

Compared to a single computer, a distributed environment poses additional performance challenges for linear software: networks introduce additional delays, and device congestion is harder to track and understand. Thus a large network application may be brought to a standstill by one overloaded server, or a single slow network link.

The performance analysis of a given configuration of linear software is exactly the same in a distributed environment, as for a single node. Provided the software satisfies the assumption of linearity, that is there is no significant parallel execution within a single response, no logical resources such as lock that cause significant delays, then as in Chapter H, the key to average performance values is in the total demand on each device, within each class of response. However there is a new factor in finding the demands, because the CPU demands usually depend heavily on the interprocessor communications overhead, and not just on the payload activities captured in the activity graphs and module demands.

This chapter considers the design factors for linear/distributed software, in cases where the target environment is fully defined. That is, the set of nodes, their storage devices and speeds, the existing (competing) workloads, and the communications infrastructure are all determined. The software is to be designed to fit into this system. A combination of a design in modules, an allocation of modules to tasks and tasks to processors, and a defined execution environment will be called a *configuration*; we will mostly consider a design intended for a single configuration. The more general problem of scalable designs intended for a range of configurations is considered in section L.?

We will particularly consider two architectural styles (these are control architectures):

- a *multi-level service* style based on ideal Remote Procedure Calls (RPCs),
- a *piped* style based on ideal data and control handovers.

It will be apparent that the piped style is potentially more efficient, but it is often too inflexible.

In defining architectures the notion of level or layer often describes more than the control aspect considered here. There may be levels, associated with levels of abstraction, as in data communications protocols, but nonetheless data packets may be piped through a series of levels; the control architecture is piped. When the originating level retains a degree of control we find a hybrid architecture we will call “forwarding”.

4.2. Multi-level Linear/Distributed Software

These systems begin from a basic module architecture, and extend it across multiple nodes. Calls and returns to entries that must cross from one node to another require sending a request message and returning a result by a reply message, which we will call an “ideal RPC”. The actual message handling may be via an RPC subsystem or by a pair of asynchronous messages, using any protocol. The linearity assumption implies that the server process is fully re-entrant, so every request is given its own thread or copy of the server.

The value of Remote Procedure Calls (RPCs) is that they conceal the complexity of distributed operation. An RPC service is part of a set of services called midware, which hides the complexity of remote operation, and glues a procedure call at the RPC client together with the remote procedure, which acts as a server. It transforms the arguments of the call into messages across a network, takes care of finding the address of the server, returns the reply message, and

finally returns from the local procedure call at the client. The execution sequence for an RPC is shown as an activity graph in Figure LA. In an ideal RPC the activities labelled PrepareRcv are combined with the following Received activities, removing the (relatively small) amount of parallel activity, and making the thread of execution fully sequential.

RPCs are useful for adapting sequential legacy software to a distributed environment. Perhaps for the same reason they are the basis of a number of attempts to describe and standardize "open distributed systems" in which software from many suppliers would interact. These include DCE (Distributed computing Environment), RM-ODP (Reference Model for Open Distributed Processing), and TINA (Telecommunications Information Network Architecture). In the terminology of Open Distributed Processing or TINA, what we call a processor is a node, a task is a capsule, a module is an object, and an entry is a method. There are some advantages in having a different term for the performance model of an entity, than for the entity itself, so we will continue to use the names task, thread, module and entry in the MSS model. It is sometimes convenient to use "node" for processor, however.

The simplest building-blocks for distributed system behaviour with RPCs are these three elementary behaviour templates, illustrated in Figures LOA and LA:

- a client loop as in Figure LOA(a) is executed by a user interface task or other load-creating task, once per system response. The loop may include a delay for user to think and type (with a delay we will call *Z*), and an activity *DoResponse* which includes all the demands for service that make up a system response. Activity *DoResponse* may be broken down into a more detailed graph to describe the activities of the User task.
- a sequential server loop as in Figure LOA(b) waits to receive a request, executes RPC overhead, executes an activity S, and returns a reply. S includes all the demands for service needed to provide the designated service by the server, and may also be broken down into more detail. This loop describes the simplest form of server, offering just one service (single entry), and with just one thread and one phase.
  - the main activity A contains requests to RPC servers. These are identified with their mean request counts in the activity parameters.
  - a service request generates some resource loading for RPC client overhead, and a request to the server.
- an RPC template as in Figure LA describes a complete blocking request-reply interaction, with the service activity *Serve* provided between the request and the reply.

Together these templates are used to build Client-Server Systems.

Client-Server Systems

Many information systems like the workflow example in Chapter 1 are organized around client tasks running in user workstations on desktops, to run applications, format requests and display results, and server tasks such as central database servers to supply the data. More complex systems with a multi-tiered architecture have additional layers of servers in the middle.

The client and server software have the generic loop templates in Figure LOA, in which the client activity *DoResponse* makes on average two requests to the server, with each request following the ideal RPC template in Figure LA. If we use the methods of the previous chapter to create a module model from these activity graph templates we get the structure shown in

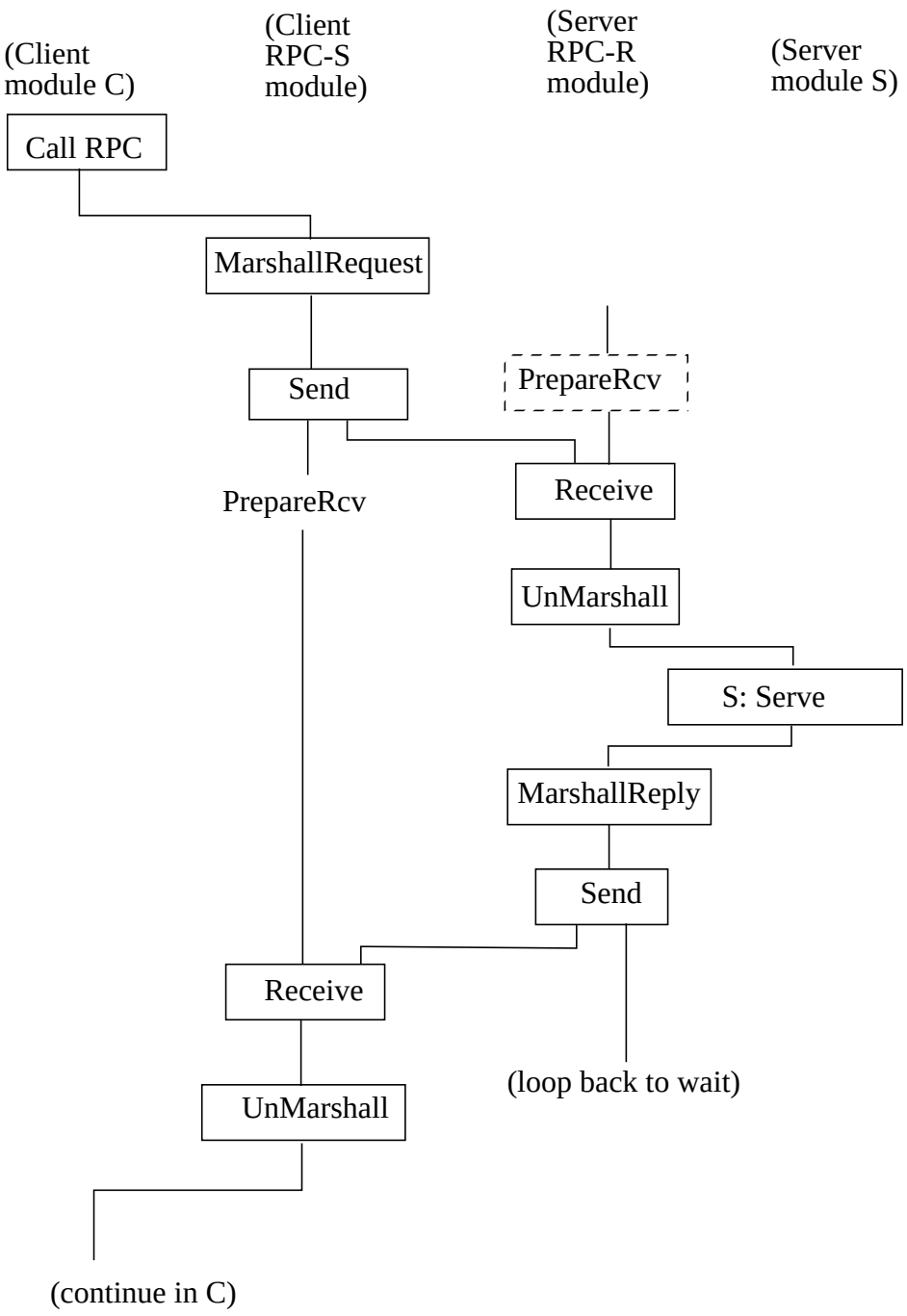
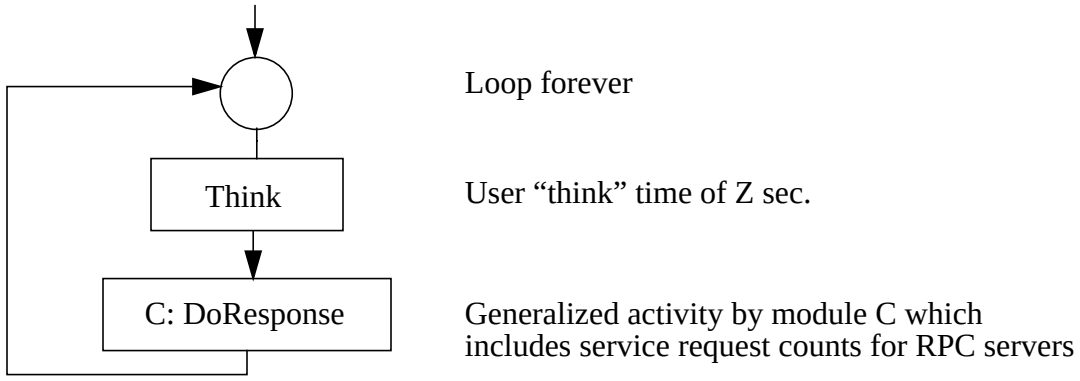
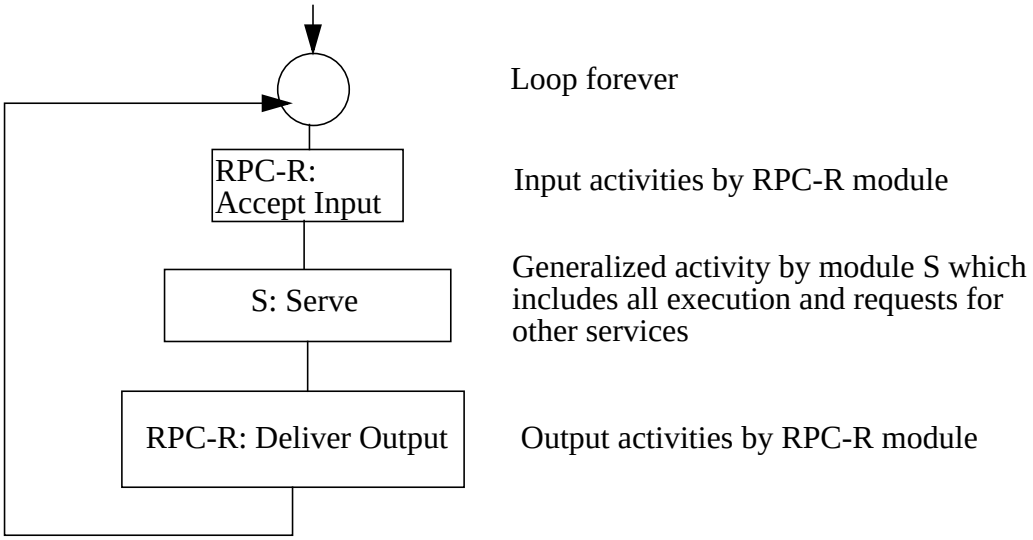


Figure 4.1. The RPC Behaviour Template. The real RPC includes the two PrepareRcv activities as shown, while the Idealized RPC moves their work-load into the following Receive activities. (Figure LA)

Figure LOD, in which workload parameter values have also been inserted. For each module Figure LOD shows the aggregate workload parameters including requests to other modules. RPC overhead modules have also been shown, RPC-S for the sender of the request (the client) and RPC-R for the server. There are dashed boxes around the entire Client and Server modules to indicate the aggregations, which are parallelogram-shaped rather than rectangular because this is the shape we will use to indicate concurrent tasks.



(a) "Client Loop" Behaviour Template (Activity Graph)



(b) "Sequential Server Loop" Behaviour Template (Activity Graph)

Figure 4.2. Figure LOA

Let us consider the effects of distribution on a simple centralized application. In the centralized version the Client Function and Service modules run together and can be aggregated into a single workload. Using the methods of the last section, we obtain demands:

Device	(one CPU)	SDisk	User (think)
Demand D_i(sec/resp)	2.70	0.51	10

With many users the one CPU saturates the system at 1/2.7 = ... responses/sec.

The client-server version shown in Figure LOD places the user interface module in the desktop, and adds RPC overhead functions and costs of 0.1 sec per call to both send and receive. Figure LOE redraws this model showing only the tasks and devices. Now the device demands are:

Device	CProc (Client)	SProc (Server)	SDisk	User (think)
Demand D_i(sec/resp)	0.6	2.5	0.510	10

Because there is one client workstation for each user the CProc demands are equivalent to “think” times. Figure LOF shows the equivalent queueing network model, in which saturation is almost unchanged, because not enough functionality was moved to the client, and SProc is still the bottleneck with demand of 2.5 sec.

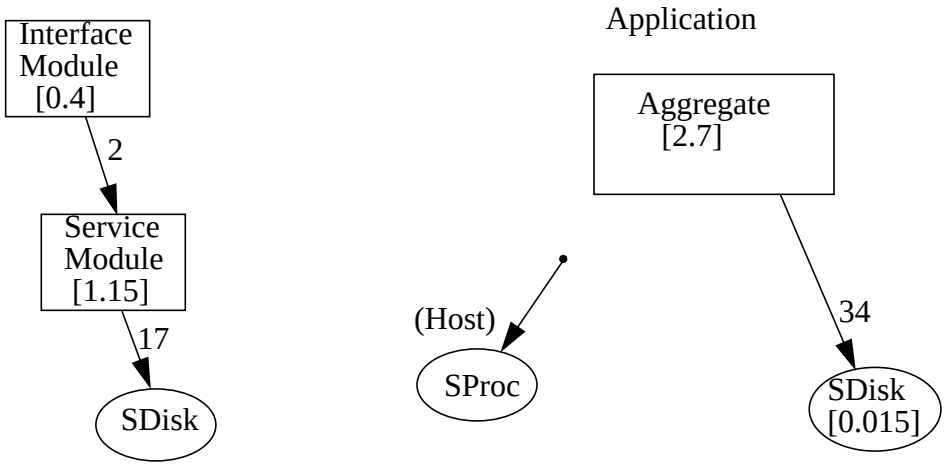


Figure 4.3. Centralized Version of Application. (Figure LOC)

Further improvement could be obtained by moving additional functions to the client workstation, making it a “thick” client. The trick is to move a submodule of MServ that is not too tightly coupled to the rest, to avoid introducing heavy communications costs.

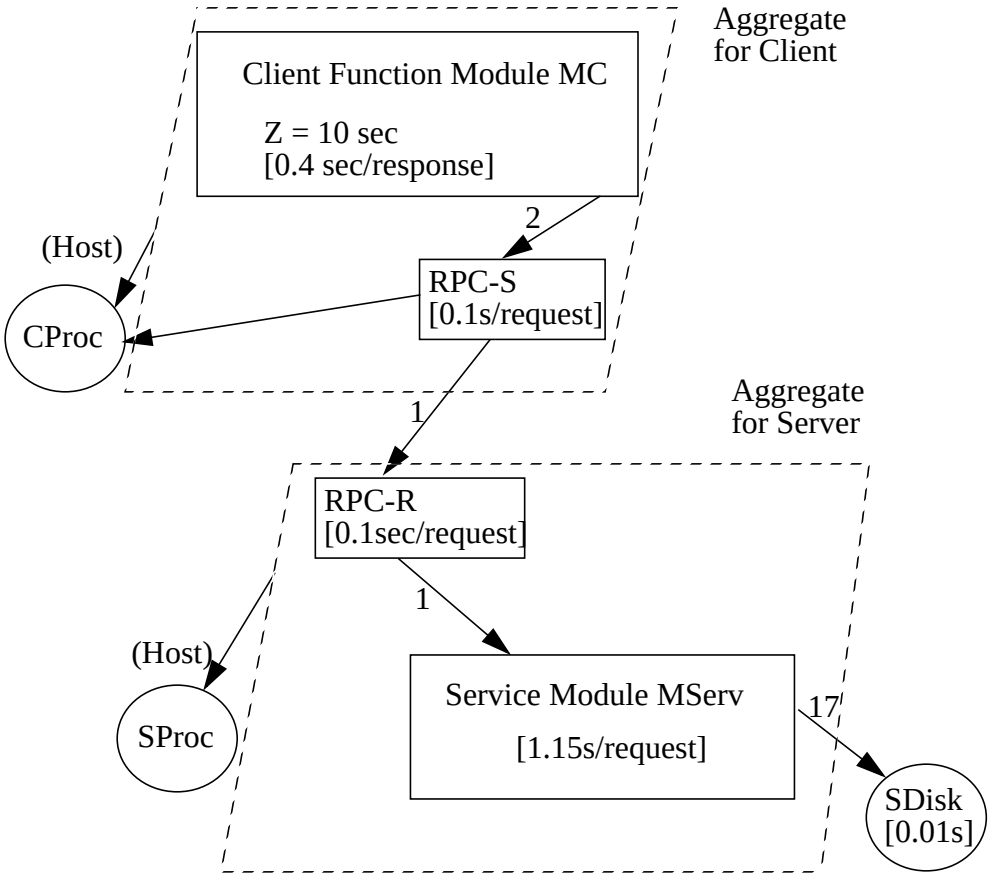


Figure 4.4. Client-Server Version of Application. (Figure LOD)

Performance Predictions

A queueing solution of the original application and the client-server version for 5 clients shows these results, confirming that not much has improved:

Table 5: Performance of Client-Server Version

	Original	Client-Server
User throughput		0.303/sec
User response time		6.5 sec.
SProc utilization		
SDisk utilization		

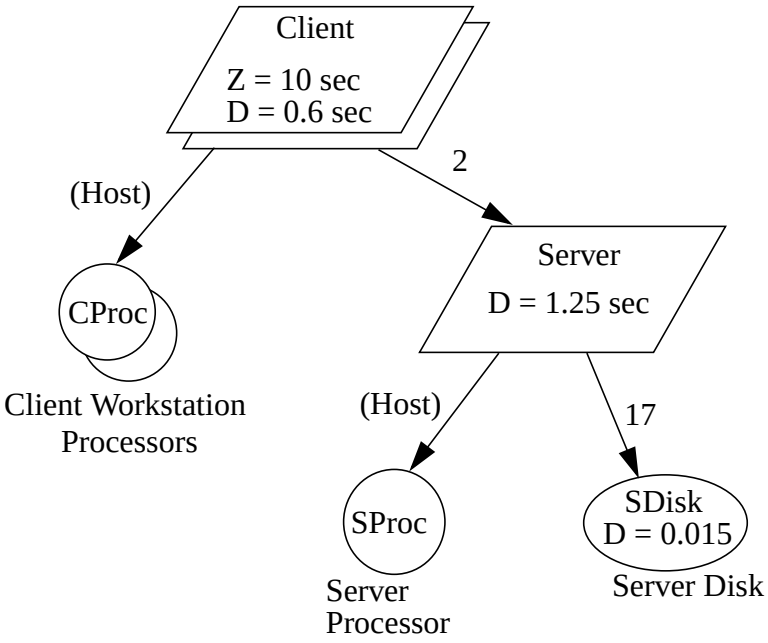


Figure 4.5. Layered Service Model for Client-Server Example, with Parameters Aggregated to the Task Level. (Fig LOE)

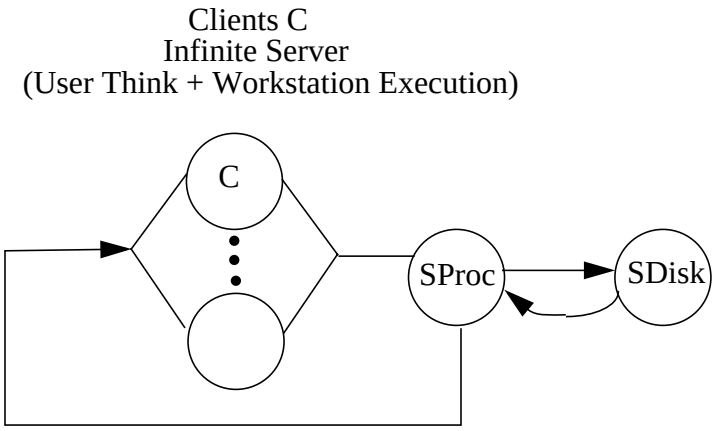


Figure 4.6. Queueing Model for an Idealized Simple Client Example, with an Infinite-threaded Server and Sequential Execution. (Fig. LOF)

4.3. The Layered Modelling Viewpoint (L3)

In layered modelling the software modules or tasks are viewed as servers, and these servers' service time is analyzed as well as the hardware delays. In linear software the software servers are all infinite servers and have no queues of requests waiting to execute, although requests may be queueing for the hardware to execute them. The module and task service times are the sums of the device response times that make up their execution. Nonetheless the shift of viewpoint from hardware to software is a great help to focusing on the connection between software modules and performance.

Layered Model of the Print Service

The print server software introduced in Figure SM will illustrate the uses and advantages of layered modelling in greater depth. In Figure SU the system was aggregated into modules. Figure LBA describes a deployment of these same modules as tasks on four different computers, plus the User nodes. The contents of these tasks are shown again in Figure LB so that RPC overhead modules (RPC-R and RPC-S) and task switching modules (TS) can be added, and a hypothetical module model for the local applications run by the users is included. This gives the layered model and aggregated parameters of Figure LC. In Figure LC the embedded processors PCProc and DCProc have been added for the printer controller and disk controller. The RPC-R and RPC-S modules were allocated 20 ms of host execution per invocation (including the request and the reply), and the TS modules were allocated 5 ms per invocation. The arrows labelled (h) connect each task to its host device. The parameter "Z = 1500" in the User task indicates a think time of 1.5 sec per execution cycle of the User task.

4.3.1. Queueing Model of the Print Server (L3.1)

If we assume ideal RPC interaction as in Figure LA then the execution of one print request is entirely sequential, as it moves from task to task and from node to node. We may imagine an execution token representing work done for the User task, migrating across the network to the remote procedure and returning after.

The demands D_i of the devices in the queue model are exactly the total demand values calculated in Figure SY for the print service, apart from the "User" task which was not represented. The values in Figure SY assumed that all processors had an operation time of 10^{-6} sec., as we will do here.

A queueing network model can be constructed based on the mean demands alone, if suitable assumptions on scheduling disciplines and service times are made (e.g. processor-shared nodes, exponential distributions at printer and disk), and it gives the response times and throughputs shown in Figure LDF.

The weakness of the queueing model is revealed on closer examination:

- It assumes that PCProc and the Printer can be processing separate jobs. Is that possible? Is the printer-controller PCProc really a separate device? Can it even operate while the printer itself is printing, or does it wait for the printer and then process the next page? If it waits then Printer and PCProc should perhaps be modelled as one server with demand 0.499, and a lower maximum throughput! If there is overlap, it is probably only after the first page begins to print.

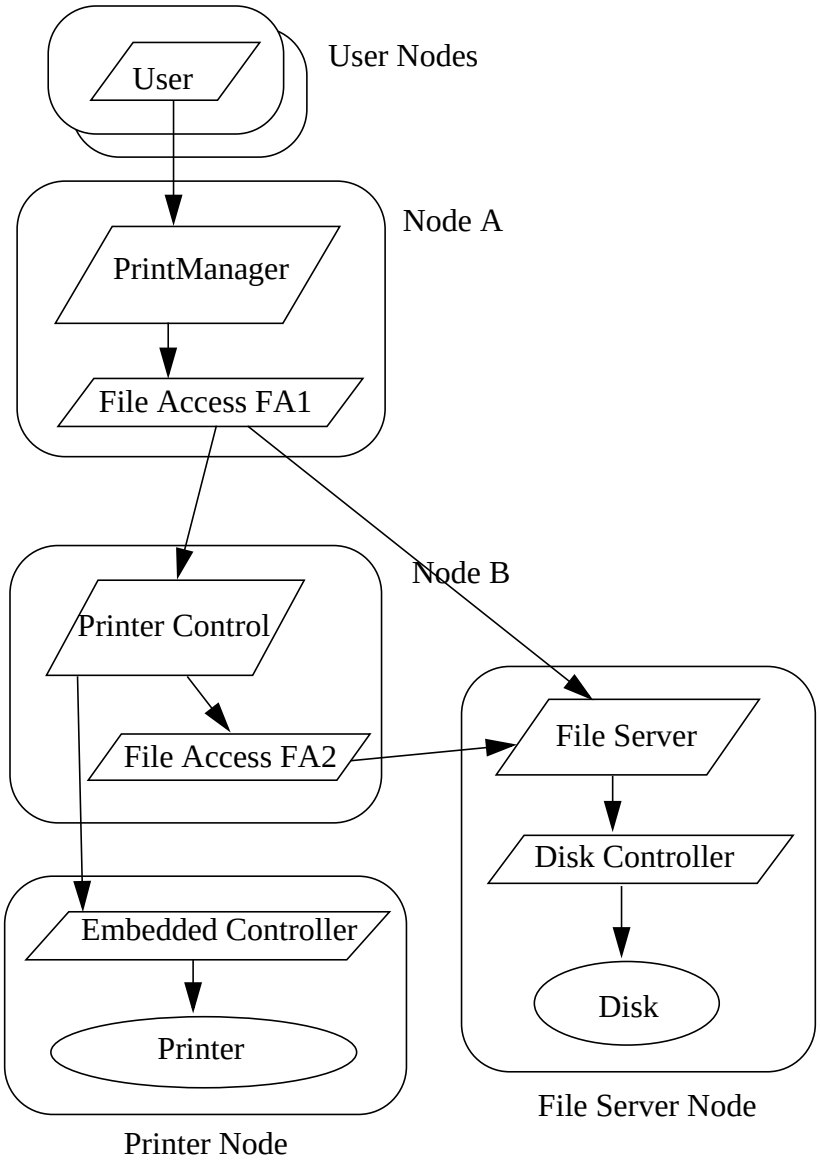


Figure 4.7. Tasks Involved in Print Server Operation (Figure LBA)

- If we wish to expand capacity with an additional printer, can we use one PrintManager sending requests to two PrinterControl processes? Or do we need some kind of duplication in PrintManager also? This could depend on whether PrintManager waits for the end of the print job or not; in the queueing model it is assumed to go on at once to the next job, even while it is in the middle of storing the file on disk in entry MI.Store.

Our first step to understanding some of these questions is to analyze the software module responses within the queueing model.

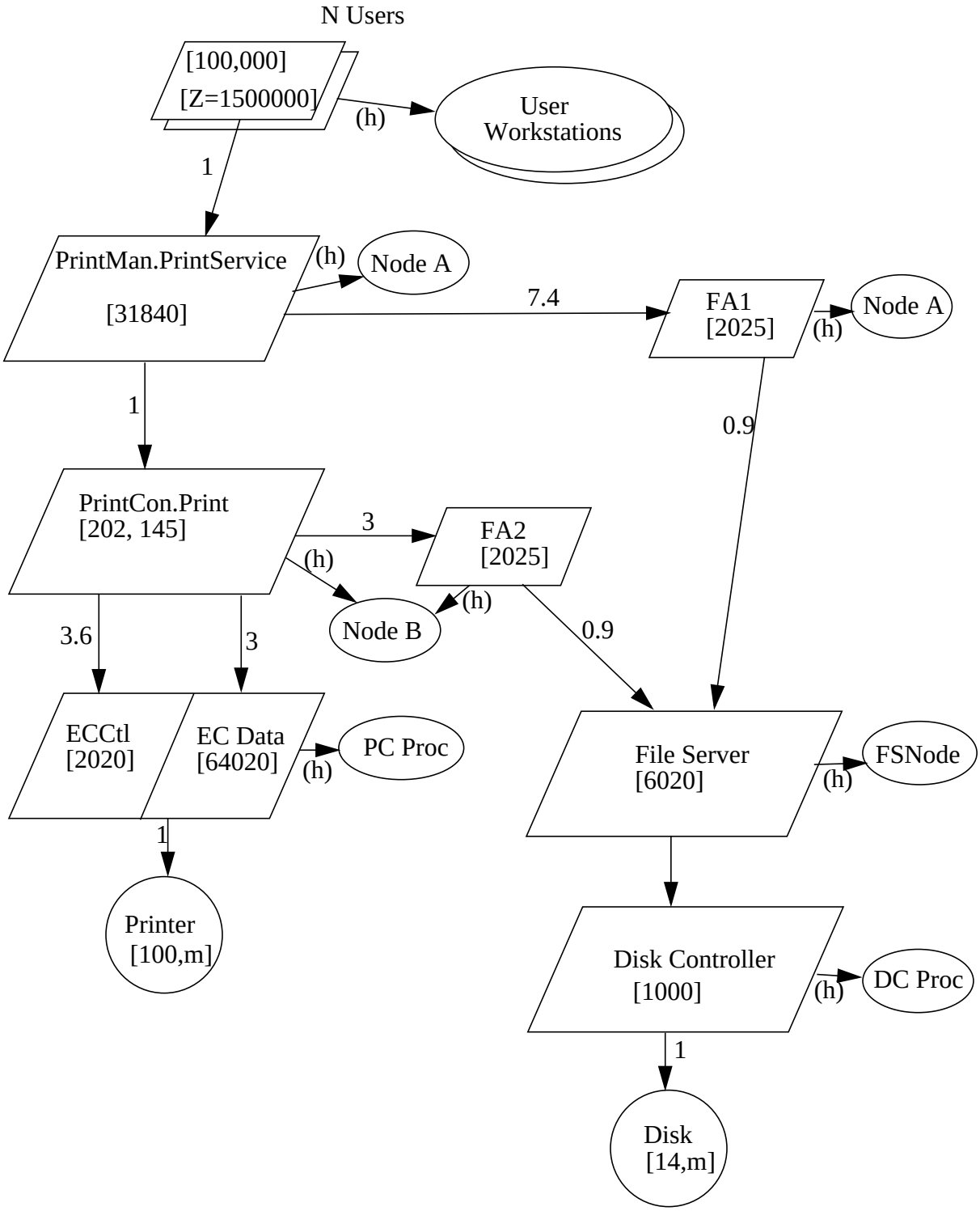


Figure 4.9. Print Server: Concurrent Task Model (including communications overheads) (all times in μsec). (Figure LC)

User Workstations	1.5 + 0.1 = 1.6 sec.
Node A	0.046 s
Node B	0.208
FS Node	0.056
PC Proc	0.199
CD Proc	0.009
Printer	0.3
Disk	0.131

Figure 4.10. Print Server: Queueing Model Demands in sec/response. (Figure LD)

4.3.2. Task Service Times in the Print Server (L.3.2)

To understand the performance aspects of individual software tasks we will consider the service time *X(e)* of entry *e* (the time to execute the entry and all nested operations), and the response time *R(e)* of entry *e*. These quantities are not usually a product of queueing network analysis, since the entry structure is aggregated out in computing total demands. Define:

- *X(e)* = service time of entry *e* (an entry of some task)

From the earlier analysis of an entry, we have defined the parameters:

- $D_{host}(e)$ = mean service demand made to its host device, per invocation of entry *e*, in seconds.
- $Y_d(e)$ = mean request count for service from entry *d*, per invocation of entry *e*, (this is the same definition as in the previous chapter, applied to a task entry instead of a module entry).

When entry *e* makes a request to entry *d*, the blocking delay at entry *e* is $R_d(e)$, read as "response time of *d*, called from *e*":

- $R_d(e)$ = mean total delay for one request from entry *e* to entry *d*.

In the same way the actual time it takes the host to provide $D_{host}(e)$ seconds of service may be longer than *D(e)* because of contention delays, so we will denote it by *R*(*host*|*e*):

- *R(host|e)* = mean total delay for the host device, including queueing, to provide $D_{host}(e)$ seconds of service to entry *e*. This may comprise several separate service times.

A device other than the host is treated as a "task".

Then it is easy to see that the service time of an entry is the sum of the operations it carries out, in two parts: a term for internal delays and operations done directly by the host device, and a sum for delay due to other servers:

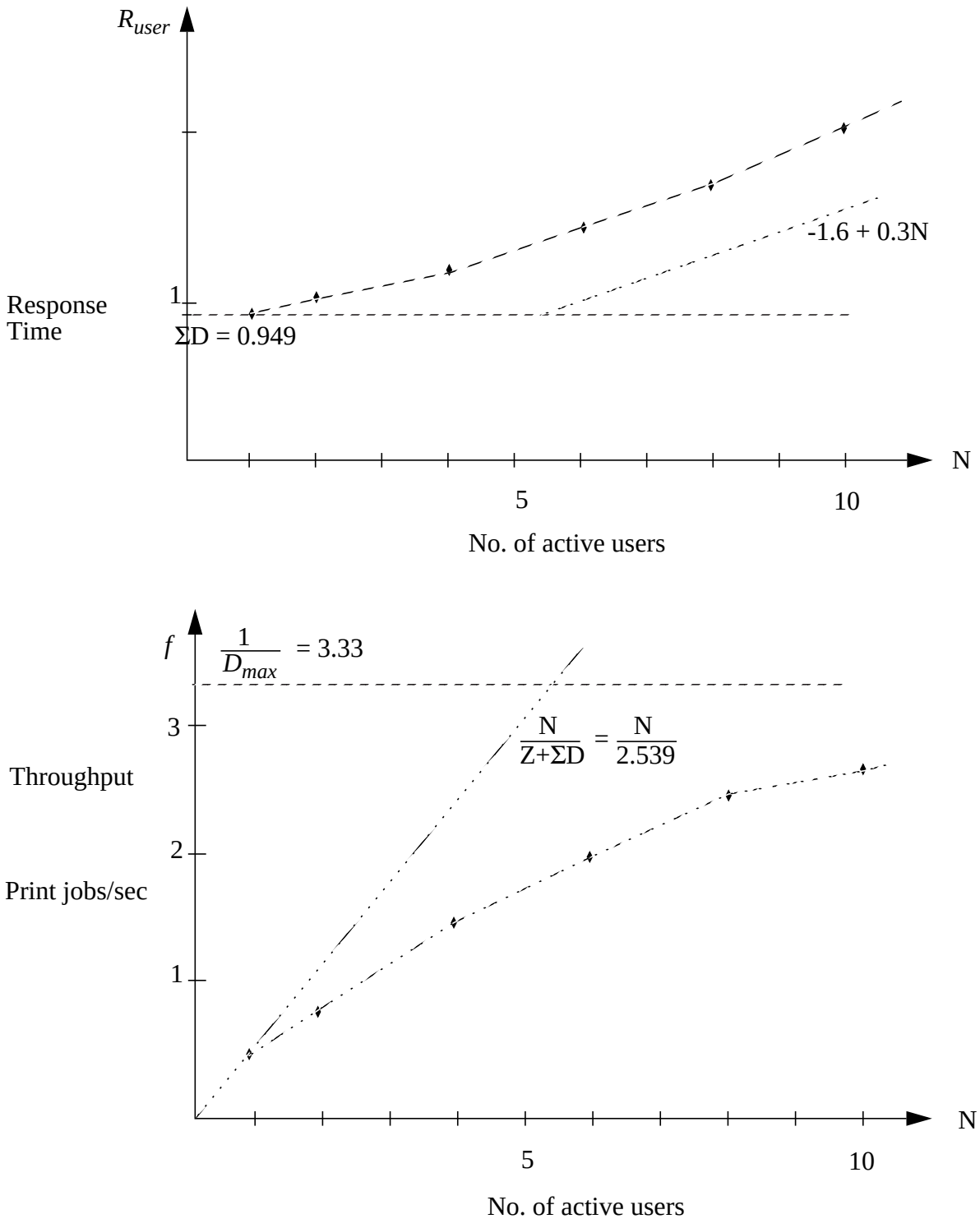


Figure 4.11. Print Server: Queueing Model (Results for Performance seen by Users) (Figure LDF)

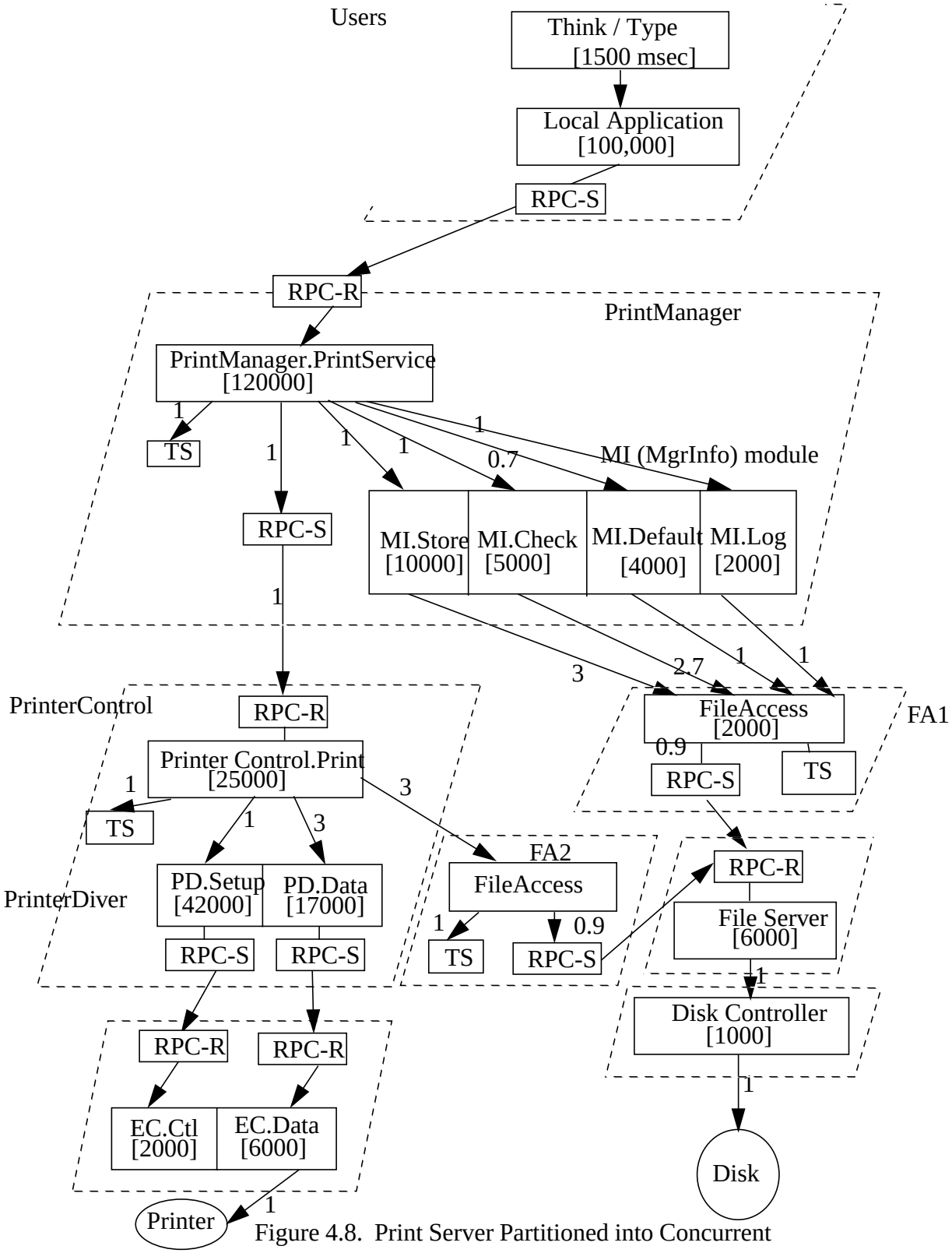


Figure 4.8. Print Server Partitioned into Concurrent Tasks (Case with *k*=3 pages per job). (Figure LB)

$$X(e) = Z(e) + R(host|e) + \sum_d Y_d(e) R_d(e)$$

In an ideal-RPC system with no logical resource limits *R(host|e)* can be found for all *host*, *e* from a queueing model, and then *X(e)* and $R_d(e)$ can be found recursively for all the calls and entries. The queueing model requires some assumptions about the system, notably that either the services times at a given host must be the same for all entries (for FIFO host such as a printer or disk) or the host service discipline must be processor-sharing (often acceptable for a processor) or infinite-server. Then *R(host|e)* is proportional to the demand at *host* from entry *e*, per response (which is $Y'_R e D_{host}(e)$.

The queueing network model in this case has a single chain with many classes, which we will aggregate and treat initially as a single class. It has parameters and performance measures:

Y'_{host} = total operation requests to device *host*, per response

D_{host} = total service demand device *host*, per response

R_{host} = total time a token spends at device *host* per response.

Then:

$$R(host|e) = R_{host} D_{host}(e) Y'_e / D_{host}$$

4.4. Distributing the Functions in Multi-level Systems (L.4)

In multi-level linear software, the placement of functions affects the balance of workload at nodes, the amount of communications overhead at node CPUs, and the network loading. If we regard the module model as a graph with module as nodes and calls as arcs, a placement of functions is a partitioning of the graph into parts with one part for each node. A communications cost is incurred where an arc crosses a partition boundary.

It is also important to determine file placement, and we can include it within function placement by identifying the file access module for the data separately, and placing it on the Mode that also has the file stored. Then the remote file access overhead becomes a communications overhead cost between the module using the data and the node that stores it, if they are not co-located.

Let *m* be an index that runs over the module names in the system, and *n* be an index that runs over the node names, and define:

$H(m_1,m_2)$ = the communications overhead cpu demand added to module m_1 if module m_2 is on a different node,

$\lambda(m_1,m_2)$ = 1 if modules m_1 and m_2 are on different nodes

= 0 otherwise

$a(m,n)$ = 1 if module *m* is placed on node *n*

= 0 otherwise.

Each module occurs just once. Then the problem of maximizing the saturation throughput, or minimizing the saturation response time, boils down to choosing the placement *a* to minimize the largest D_n:

$$\min_{\{a\}} \max_{n} D_n$$

(Stone & Bokhari)

MULTIFIT-COM here)

4.5. Piped Linear/Distributed Software

Piped software does not conform to the module model developed in the last chapter. It is more primitive, and is based directly on activity graph model. The sequence of activities is aggregated into modules, with one module for each stage in the pipeline. At the completion of a module, instead of returning its result, the module sends it in an asynchronous message to the next stage.

org -- quick descrpn

-- simple example *

-- allocation Bokhari

-- expensive handover (files or filters)

-- compare sequence to hierarchical structure, master slave

-- compare to call all way down and return

-- derive forwarding.

Draft: October 18, 2001

Performance - Oriented Patterns in Software Design (A multi-level service approach)

C. M. Woodside

Dept. of Systems and Computer Engineering

Carleton University, Ottawa K1S 5B6

(Draft version produced for classroom use, October 1996)

Chapter 1. The Goals of Performance Engineering

Chapter 2. Performance Delivered by the Hardware

Chapter 3. Tracing Performance to Software Modules and Behaviour

Chapter 4. Concurrent and Resource-Limited Servers: MSS(Resources) (C)

Chapter 5. P: Patterns

Using the small-scale patterns of the last chapter for internal task resources, execution and interaction, we can build and analyze performance models. The MSS(Res) framework of models describes a wide variety of distributed service systems for business computing, industrial automation, communications systems management, etc.

When we begin to analyze model results we find that certain arrangements of concurrent tasks recur and have charateristic performance attributes. These architectural level patterns will be included in our repertoire of "performance-oriented patterns". This chapter considers four architectural patterns which are extremely common in existing and proposed designs, and which are seen in classification of software architectures such as the one by Shaw [??].

- The "Tower" pattern is a layered set of servers showing vertical separation of functions, seen in descriptions of client-server and transaction processing systems. This is a simplified version of Shaw's "Client-Server" architecture.
- The "Lattice" pattern is a set of cross-linked Towers, representing layered service with several servers at each layer. This is a more general version of Shaw's "Client-Server" architecture and could represent a three-tier client-server system or a distributed transaction processing.
- The "Peer-to-Peer" is a model for symmetrical servers which exchange requests with each other. It contains a transformation which produces a special case of the Lattice Patterns.
- The "Flow" pattern represent pipelined processing. This is very common and is one of Shaw's categories. We consider also extended versions of this pattern with servers shared by pipeline tasks.

Further architectural patterns which incorporate fork-join behaviour patterns will be studied in a later chapter.

5.1. The "Tower" Pattern

The name "tower" will be applied to a set of layered servers, which are, as it were, piled one on top of the other to make a tower of tasks. In each of the middle layers there is just one server, while at the top there may be many users, and at the bottom there may be many servers. The basic pattern is shown in Figure PA, with a set of N_1 user tasks, in the top layer, layer 1, making requests to a single server in layer 2, which in turn makes requests to a single server in layer 3, down to where layer *L-1* makes requests to N_L servers in layer L. We have already seen that a database system may have layers like this, with user tasks running on desktops, with a Transaction Manager, a Data Manager, and a File Server, and with a set of disks at the bottom.

The Tower pattern has the usual workload parameters, which are the same for each tasks in a given level. They will each be labelled with subscripts for the level l, counting down from the user tasks at the top:

- there are N_l tasks (assumed symmetrical) at level l, identified as "T_l".
- The task at level l has m_l threads, from $l = 2$ to $L-1$.
- The tasks at the bottom are single threaded.
- Each task at level l has an average host demand of D_l sec, and makes an average of y_l requests to each server at the next lower level.
- As a result each task at level l is invoked on average v_l times per user task cycle, where $v_1 = 1$ and $v_l = y_1 y_2 ... y_{l-1}$, $l = 2, 3.....$
- If f is the total rate of requests from the user level, the invocation rate of each task at layer l is $f_l = v_l f$.
- The N_1 user tasks at the top are single threaded and execute in cycles. A cycle begins with the

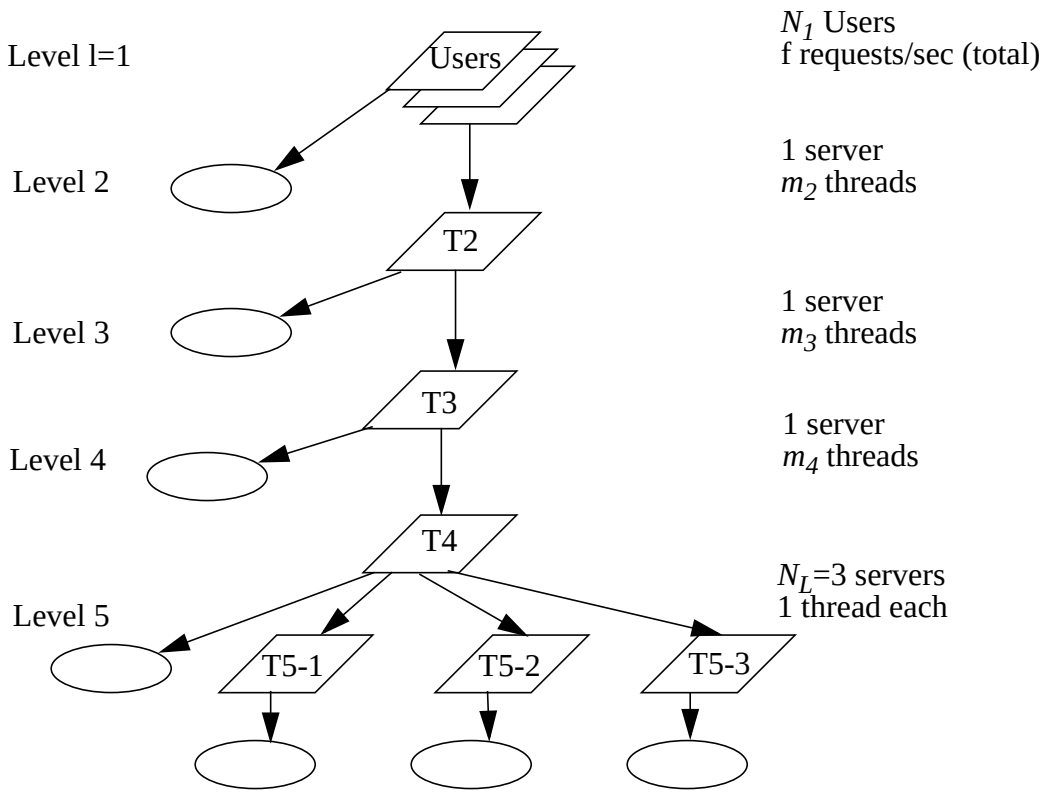


FIGURE 3. Tower Pattern with Five Levels (Figure PA)

user "thinking" for Z seconds, then making a request which the task executes. The end of each cycle of each user task begins the next cycle, with a cycle time of X_1, including the delay Z. The user response time is $X_1 - Z$.

- the total response rate over all user tasks, f responses per second, is given by $f = N_1/X_1$.

A software bottleneck is a serious concern for this pattern. It may occur where a resource-constrained server makes blocking requests to a lower-level service, thread, or resource. The effect is stronger where there are more lower-level servers in the fan-out, but just one lower-level server plus the constrained task's own processor is enough. As well as thread resources, the distribution of host demands and other requests over the levels determines the severity and location of a bottleneck.

Task saturation is indicated by utilization, the fraction of time the task is busy. The task source time, or the read sevice time of $m_l>1$, is X_l and includes blocked time waiting for requests to lower server to complete. At the bottom level L the tasks only have host execution, so $X_L = D_L$. We will consider two kinds of utilization:

- host utilization per task at level l is $HU_l = f_l D_l = f v_l D_l$,
- task utilization at level l is defined per thread (if there is more than one), giving

$U_l = f_l X_l / m_l = f v_l X_l / m_l$, $l \geq 2$. In this case saturation is indicated by U_l approaching m_l.

- task utilization of each user task is $U_1 = f X_1 / N_1$, since the utilization is shared among the N_1 user tasks,
- task utilization of a task at level L is $U_L = f v_L X_L$, since they are single-threaded

Basic Case: Tower1

Let us begin with Tower1 as shown in Figure PA. It has five levels, with 10 user tasks at level 1, then three middle levels with single-threaded servers T2, T3, T4, each with its own processor, and finally three identical bottom-level servers T5_1, T5_2, T5_3. Each server is invoked once per response and has one unit of execution demand ($v_l = 1$, $D_l = 1$). When this system executes, the service time of T4 is four units, one unit for itself and one unit for each of the three bottom-level servers. T3 adds one unit for itself, for a total of five, and T2 takes six units for its task service time. Because the tasks are single-threaded, T2 must finish with one response before it starts the next. The lower level tasks never have more than one request to deal with at a time; T2 effectively sequentializes the entire system. The T2 service time of six units defines the maximum throughput capability as one response every six units, or $f = .166$, approximately. At this maximum throughput, each processor is used at only one-sixth of its capacity. This is an extreme case of a software bottleneck.

What this example will show is that

- when any resource is saturated, all finite higher-level resources are saturated too, while lower-level resources are not,
- bottleneck relief must include the saturated resource, but may have to include other resources too, to get full potential value.
- bottleneck relief will be provided in these examples by multi-threading of tasks, and there is a rule of thumb for how many are needed,
- when there is contention at a resource, its service time increases when throughput goes up, making it difficult to predict the limits from light-traffic measures.

TABLE 3. Performance of Tower1 Shown in Figure PA (Figures are for each task at level *l*) (Total Throughput *f* = 0.166 requests/sec) (Figure PB)

Level *l*	Task Utilization U_l	Task Service Times X_l	Host Utilization HU_l
1 (Users)	1.0	60	0.166
2	1.0	6	0.166
3	0.833	5	0.166
4	0.666	4	0.166
5	0.166	1	0.166

Figure PB shows the task resource saturation at different levels in the system, for different throughputs up to the maximum. At the lower values, the users have a delay between the completion of one request and starting a new one, while at the highest value there is none. The device utilizations are not shown but they are numerically equal to the throughputs. The lowest curve is for a long user "think time" which gives a low total request rate; the second is for a moderate think time giving a moderate rate. Notice how the task utilizations build up at the higher levels, while the device utilizations are the same over the levels. Blocking delays pile up at the higher levels and cause longer task service times, which reflect in higher task utilizations. Because T2 has a single thread, there can only be one active response in the system below it. Since there is no message queueing at levels 3 to 5, the way the delay piles up is very simple. Each task's service time *X* is the sum of its own host demand *D* and the service times of the tasks below it; task utilization is proportional to *X*. Therefore, a server below a single threaded task cannot be fully utilized; some of the time it must be waiting for the next request.

Even though this is a very oversimplified example, it is worth understanding. We can see that:

- the bottleneck at level 2 is due to it being single threaded; for higher throughput we must be able to process several requests at once, which implies multi-threading at levels 2 down to 4. On the other hand, threads in level 5 would do no good as these tasks do not block.
- if we cannot increase the threading levels then the system will continue to be sequential at and below level 2. We could get equivalent or better performance by running all the tasks on a single processor at and below level 2. It would be even better to merge them into a single task, because it would reduce demand for intertask communications overhead! (This overhead has not been separated out here, but is certainly present.)
- the first improvement would be to introduce multiple threads to level 2; then the bottleneck would move down to level 3. Second, multi-thread level 3, and it would move to level 4. Finally with multiple threads at level 4, the bottleneck should move down to level 5 and also to the devices of all levels. Because the heaviest device utilization in Figure PB(d) is 1/6, we can in principle search for a sixfold increase in throughput, when we introduce multiple threads. A sixfold increase would make $f = 1$.
- How many threads to introduce? If one only introduces them in level 2, two threads is enough, because one can be executing while the second one is blocked while T3 executes for it. More would just have to wait for T3 to begin serving them. A similar argument could be made at level 3, but for level 4 (because there are three servers at the next level down), more threads are useful. But then, if one increases the number at level 4 it will pay off to increase them also

at the higher levels.

Thread Rule of Thumb #1

- A simple rule of thumb when (as here) there is no phase-2 work, is that the threads at a level should be the sum of the threads in all the servers at the next level down, plus one, to avoid idle time on a task due to blocking. On the other hand if for any reason a higher level task is constrained in its number of threads, lower level tasks need not have more. For level 1, the N_1 user tasks are treated as threads, so the most threads we expect to be useful is 10.

If the rule of thumb is applied to Tower 1 it says level 5 has 3 tasks of one thread, so level 4 requires 4 threads, level 3 requires 5, and level 2 requires 6. Level 1 has 10 users each of which is constrained to be a single thread. We will focus on the levels 2 to 4, which have (m_2, m_3, m_4) = (6, 5, 4).

Multi-threading

Figure PC shows the results when one introduces multithreading level by level, starting at the top at level 2. The rule of thumb is used to determine m_l to be one more than the sum of values for the next level down. The figure shows the mean number of busy threads in part (a), and the relative saturation of thread resources (mean busy threads over total threads) in part (b), and the mean task service time in part (c). Look first at the values at the right side of the table, as threading is introduced gradually:

- (m_2, m_3, m_4) = (2,1,1) and throughput 0.20/sec;
- (3, 2, 1), and throughput 0.22/sec;
- (6, 5, 4) and throughput 0.47/sec.

The big payoff really comes with threads in level 4, although providing threads only at level 4 would have no effect at all!

The rule of thumb may underestimate the number of threads that can provide a benefit, basically because threads in a server may compete with each other for the next server down the tower (which could not happen when there was only one). It is really necessary to evaluate the effect of threading. Higher numbers of threads give some additional improvement, indicated by the other curves in the Figure:

- (m_2, m_3, m_4) = (7, 6, 5) and f = 0.55/sec,
- (8, 7, 6) and throughput 0.58/sec,
- (9, 8, 7), and throughput 0.62/sec,
- (10, 10, 10) and throughput 0.65/sec.
- (∞,∞,∞) 0.65/sec.

This pattern frequently underperforms, compared to expectations. Why can this example not exceed 0.65 responses/sec even if the users are flooding it with input, there are infinite threads, and each task separately can handle 1.0/sec.? The answer lies in (1) the relatively small number of user tasks, just 10, compared to the amount of work for each response (7 units). Even with more processors, we could never exceed 10/7=1.42 responses/sec. and (2) in random interference of requests, due to the variability in the execution demands. For these evaluations the host demands

were random and exponentially distributed, which is higher variability than is found in some applications, but lower than others. It makes a curious trade-off. With deterministic times there would be no improvement above the rule-of-thumb values (6, 5, 4), at which a full 1.0 response per second is obtained. Random interference throttles back the capacity at (6,5,4) to less than half of that, but then allows additional threads to regain part of the difference.

TABLE 4. Multi-threaded Tasks in Tower 1: Throughput, Task Saturations and Other Measures. (Figure PC)

Threading Level Cases, defined by (m_2, m_3, m_4)								
	(∞,∞,∞)	**(10,10,10)**	**(9,8,7)**	**(8,7,6)**	**(7,6,5)**	**(6,5,4)**	**(3,2,1)**	**(2,1,1)**
Throughput *f*	**0.65**	**0.65**	**0.62**	**0.58**	**0.55**	**0.47496**	**0.223521**	**0.200347**
Level *l*	(a) Mean Busy Threads = Task Utilization (U_l)							
1	10	10	10	10	10	10	10	10
2	9.34	9.35	8.8	7.95	7	5.51	2.90	.96
3	7.8	7.8	7	6.2	5.36	3.89	1.64	1
4	6.24	6.22	5.4	4.7	4	2.75	0.89	0.8
5	0.65	0.65	0.62	0.59	0.55	0.47	0.22	0.2
Level *l*	(b) Task Utilization per Thread (U_l/m_l)							
1		1	1	1	1	1	1	1
2		0.94	0.98	0.993	1	0.92	0.97	0.98
3		0.78	0.88	0.89	0.89	0.78	0.82	1
4		0.62	0.77	0.78	0.8	0.69	0.89	0.8
5		0.65	0.62	0.59	0.55	0.47	0.22	0.2
Level *l*	(c) Task Thread Service Times (X_l)							
1	15.38	15.38	16.13	17.24	18.18	21.05	44.74	49.91
2	14.37	14.38	14.19	13.71	12.73	11.60	12.98	9.77
3	12	12	11.29	10.69	9.75	8.19	7.32	4.99
4	9.6	9.57	8.71	8.10	7.27	5.78	4	3.99
5	1	1	1	1	1	1	1	1

Only a model can predict the balance of these factors. For example,

- if the number of users is increased to 25, and variability is kept the same, the throughput for infinite threads goes up but only to 0.83/sec... (PDM)
- if the variability of execution times is reduced, so its coefficient of variation is 0.5, then f = ... (PDN)

Effect of Variability in Execution Times

High-variance execution and communication behaviour has been observed in networks and in execution statistics and has quite serious performance effects. [refs] High variance or “Self-similarity” was first observed in network traffic when it tends to nullify the advantages of large

scale and multiplexing of traffic. High variance in software execution times increases contention delays and reduces average throughput. However some of that reduction can be gained back by exploiting multi threading.

Consider Tower 1 with rule-of-thumb thread levels of (6, 5, 4), mean service demands of 1.0 units and execution-time standard deviation σ_D of 0 (deterministic), 1.0 (exponential distribution), and 10 (hyper-exponential), and then consider the gain obtained by making the thread units infinite.

TABLE 5. Variability of Execution Demand (Figure PDR)

		Thread Levels		
		(m_1,m_2,m_3) = 6,5,4	**Infinite Threads**	
Execution Demand Std. Dev. σ_D	**Throughput f Resp/sec.**	**Mean Busy Threads ($\overline{m}_1$, $\overline{m}_2$, $\overline{m}_3$)**	**Throughput f Resp/sec.**	**Mean Busy Threads ($\overline{m}_1$, $\overline{m}_2$, $\overline{m}_3$)**
0	1.0	(6, 5, 4)	1.0	
10	0.47	(.55, .39, .27)	0.65	

In these results we see the interesting fact that higher variability at σ_D = 10 reduces throughput dramatically and increased thread levels restore only a fraction - about a third - of the lost throughput. User response time is even more dramatically affected.

Because of the complexity of the interactions even in a relatively trivial architecture like Tower 1, a model is essential for determining the risk posed by variability. Thread levels alone, unfortunately, do not solve the problem and restore the capacity. Another mechanism for achieving improvements is through priority scheculing, essentially by reducing the priority of threads that have executed for a long time. With high-variance jobs a thread that has already had a long time is more likely to need yet more, so they reveal themselves. Unfortunately this kind of dynamic priority scheduling is not yet common on workstations.

Critical Sections Limit Thread Effects

Unfortunately we cannot make all processing multithreaded. Often there is interference between threads because they share resources or data, which requires a critical section in which only one of the threads can execute at a time. It may then be simplest to restrict a task to a single thread and concentrate on making it efficient. Only a critical section which covers local execution only, without any service requests, will have almost no performance effect.

A critical section which covers some but not all service requests is modelled as shown in Figure PE. The pseudo-task CS includes that part of T3 which is within the critical section, and has a single thread. Waiting for CS models the waiting for the critical section. CS includes both execution and some service requests to T4. The execution in T3 and requests from T3 to T4 are divided between the critical section, and non-critical execution of each thread, in a ratio β:(1 - β). With β=1, T3 is effectively single-threaded, while with β = 0 the critical section disappears.

What we see in Figure PE is how the choke-point due to the critical section spreads congestion back up into the system, so that attempts to correct the problem by changes above that point are doomed to failure. The number of threads that are worth providing above T3 depends on how many threads can be used effectively at T3, and this drops as β increases.

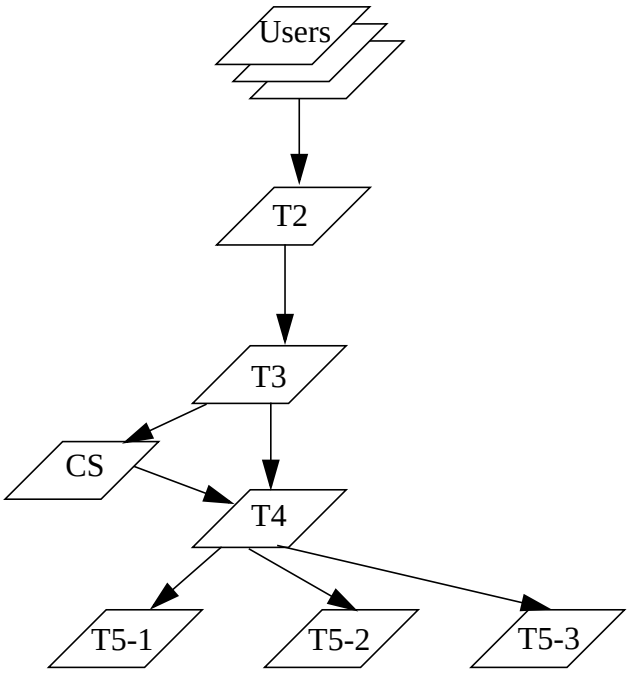


FIGURE 4. Tower1 with a Critical Section Modelled by Pseudo-Task CS. (Figure PE)

How can we estimate the thread resources needed, with such complicated factors at play? A method for estimation is considered next.

TABLE 6. A Critical Section in T3 with Thread Levels of Tower 1 (m_2, m_3, m_4) = (6, 5, 4). (Figure PESIM)

	Critical Section Ration β (β = fraction of T3 execution within the critical section)			
Throughput *f*	**0.0001**	**0.33**	**0.67**	**1.0**
	0.5	**0.36**	**0.26**	**0.2**
Level *l*	(a) Mean Busy Threads			
1	0	10	10	10
2	6	6	6	6
3	4.5	4.6	4.74	4.8
CS	0.14	0.86	0.98	1.0
4	3.21	2.01	1.22	0.8
5	0.5	0.35	0.26	0.2
Level *l*	(b) Task and Critical Section Utilization (per Thread)			
1	1.0	1.0	1.0	1.0
2	1.0	1.0	1.0	1.0

TABLE 6. A Critical Section in T3 with Thread Levels of Tower 1 (m_2, m_3, m_4) = (6, 5, 4). (Figure PESIM)

	Critical Section Ration β (β = fraction of T3 execution within the critical section)			
Throughput *f*	**0.0001**	**0.33**	**0.67**	**1.0**
	0.5	**0.36**	**0.26**	**0.2**
3	0.9	0.92	0.95	0.96
4	0.8	0.50	0.30	0.2
5	0.5	0.35	0.26	0.2
Level *l*	(c) Task Thread Service Times			
1	20	27.8	38.5	50.0
2	12	16.7	23.1	30
3	9	12.8	18.2	24
4	6.42	5.6	4.7	4
5	1	1	1	1

Prior Estimation of Thread Resources

Is there a simple way to estimate the desirable number of threads to provide, that is more accurate than the rule of thumb of one more than the sum over the next level down? An approach which has the advantage that it deals with critical sections, second phases and other potential complexities in the interaction patterns, is to analyze for infinite threads in every task, but including any necessary critical sections. For Tower 1, the diagram in Figure PEM with infinite threads at levels 2 to 4 shows the idea. Levels 1 and 5 are left single threaded because at level 1 threads make no sense, while at level 5 they make no difference. The results show the mean number of active threads:

- $\overline{m}_l = f_l X_l$ = mean active threads per task at level *l*

TABLE 7. Estimating Thread Levels from Models with Infinie Levels ((m_2, m_3, m_4) are infinite) (Figure PEM)

	Critical Section Ratio β (β = fraction of T3 execution within the critical section)			
Throughput *f*	**0.0001**	**0.37**	**0.67**	**1.0**
	0.69	**0.36**	**0.24**	**0.2**
Level *l*	(a) Mean Busy Threads			
1	10	10	10	10
2	9.3	9.64	9.76	9.8
3	7.9	9.13	9.46	9.56
CS	0.29	0.83	0.96	1
4	6.12	2.04	1.19	0.8
5	0.69	0.36	0.24	0.2

In Figure PEM, infinite threads give only a modest increase in throughput over Figure PD, from 0.62 to 0.67, even with no critical section. The mean number of busy threads is about the same. When the critical section fraction increases, moreover, we can see how rapidly the useful number of threads drops. When using these results, it will be useful to make the actual number of threads a little bigger than this; when in doubt pick a configuration and run an analysis.

Most interesting, even with infinite threads and no critical section, the device utilization is only 68%, indicating that 32% of the processing capacity is still not being utilized. This is due to the layered structure, random contention, and the relatively small number of users. However calculations with 20 and 30 users saw device utilizations rise only a little (to 82% and 88% respectively) while the response time skyrocketed from 13 units for 10 users, to 24.3 units for 20, and 34 units for 30.

Delay and a Lightly Loaded Tower Pattern

All of this section has considered only the maximum throughput obtainable from a tower-patterned subsystem, rather than the delay to an input request. It turns out that the changes which favour higher capacity also mostly favour smaller response delays in this case. If delay is the important factor and the system is lightly loaded (eg, there is a longer think time Z between requests), then a satisfactory response time may be obatined with fewer threads. The number of threads required for the servers will be lower, without penalty, just because they would mostly be idle.

Unbalanced Execution Demands

[Figure PEQ and discussion to come]

Summary

Thread resources must be considered in designing for performance, but they must be applied consistently to all tasks, and they cannot overcome other resource constraints such as a critical section between threads.

5.2. Variations on the Tower Pattern

Tower1 is oversimplified in two broad ways. First, real systems are less symmetrical. Their demands are not balanced between levels or tasks, levels may share host processors, the request values are not unity between levels, and there may be second phases of service. Second, they may be interconnected to other tasks, so there may be more services requested at intermediate levels, and requests may come in from other subsystems to various levels. Here we will consider request flows in and out of a tower, and second phase effects.

Fanout:

A vertical sequence of tasks in a larger system may be identified for careful study as a tower pattern, if the links between it and the rest of the system are very weak. If they are not so small as to be completely ignored, requests made to tasks outside the tower can be represented as appended or *fanout* requests. They have the effect of inducing additional delay to threads of the task in the tower making the request. Figure PG shows Tower 1 with task *T3* making y_{3A} fanout requests to the appended task *T3A*, with service time 2 units.

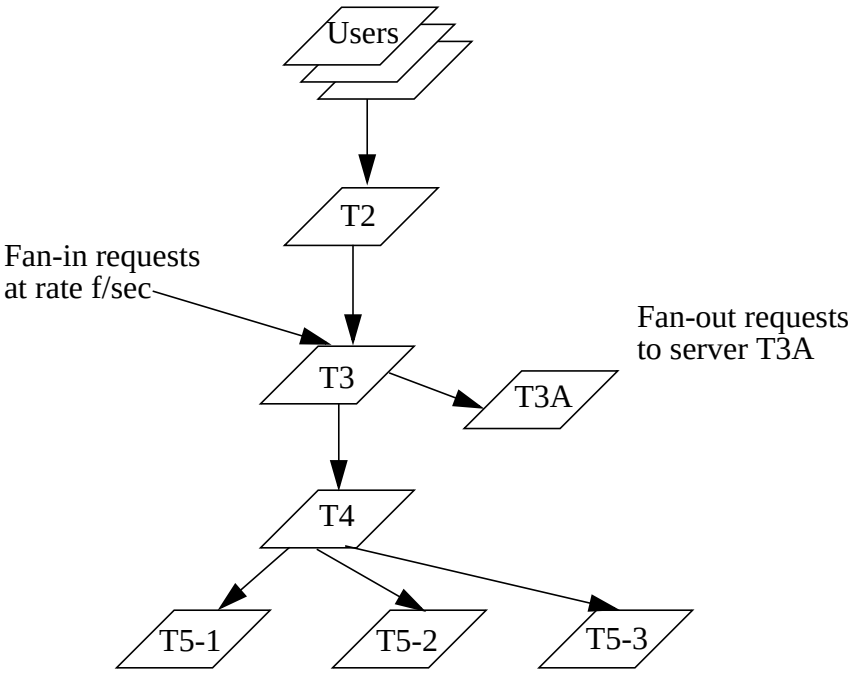


FIGURE 5. Fan-in and Fan-out Requests in Tower1. (Figure PG)

If *T3* is single-threaded it blocks for longer and the tower performance declines; if it can be multi threaded then part of the performance is recovered, as shown in Figure PGF. The rule of thumb suggests that m_3 be one plus the sum of m_4 and the thread count of *T3A*.

Second-phase execution at the appended server changes the picture totally; if it is mostly second-phase there will be almost no effect, as the reply will come back almost at once.

TABLE 8. Effect on User Throughput of Additional Fan-out Requests for *T3* to *T3A*. (Figure PGF)

	User Throughput				
y_{3A} (mean requests to *T3A*)	0	0.5	1.0	1.5	2.0
Single-threaded:	0.167	0.143	0.125	0.111	0.10
(m_2,m_3,m_4) = (6,5,4)	0.465	0.421	0.358	0.288	0.231
Infinite threads	0.67	0.626	0.481	0.331	0.249

Fan-in:

Requests may similarly flow into a tower pattern from other subsystems. Suppose they are represented by an open flow of requests at rate f' arriving at level L', as shown in Figure PGH. Now, as f' increases it tends to saturate that level, and the response time $R_{L'}$ goes up due to queueing delays. The blocking time to the next level up increases, and this makes a higher threading level there more worthwhile.

Second Phases:

"Second phase execution" comprises activities carried out by a server after the reply is sent. In real RPCs there is at least a small amount of second phase in the PrepareRcv activity shown in Figure PC, getting ready for the next message reception. But many servers are designed so the reply is sent as early as possible, with various postponable execution done in the second phase, such as buffer deletion, or writing and closing files. File servers with cached writes are a simple and ubiquitous example of second phase, for the actual write is done after the data is stored in the cache and the client is acnowledged.

The effect of a task's work being in the second phase on a Tower is interesting. In a lightly loaded Tower it results in shorter response times, because a client is blocked less and the server executes in parallel. In a heavily loaded Tower with a bottleneck at that task, however, the effect on the maximum capacity is small. With second phases at a certain level, the level above may require more threads to reach its full capacity, because second phases increase the possibility of queueing; a task may even have to queue when it is the only requester to a server. With increased possibility of queueing additional threads may sit blocked, while others work on new requests.

[Figure PGL to come]

For a Tower with a fan-out, work which is moved into a second phase of the appended task can improve capacity.

Tasks which Share Host Processors:

Where tasks in a Tower share host processors it breaks the pure Tower pattern, for two levels then share a common server. This may reduce the number of threads that can be used with advantage. For instance if two neighboring levels share a host processor, to a first approximation the two tasks could be considered as one in rule of thumb. It should be modified to say that they have the same number of threads, rather than the upper one having one more.

In general, situations have to be considered in detail with a full evaluation.

5.3. Lattice Pattern

A lattice is a set of interconnected, more or less similar Towers, giving a diagram which looks like a lattice-work for climbing plants (e.g., Figure PH). It arises when two or more similar systems are connected together, for example

- a government social services agency has regional offices, each with its own databases of clients, budgets, personnel, etc., but they are linked together for consolidated reporting and to deal with cases which move from one region to another,
- divisions of a company, even though co-located, have their own information systems, but these are connected together for rapid and uniform handling of inter-divisional transactions and joint operations, and for gathering data for upper management.
- a company and its suppliers provide some restricted access to each others information systems to speed up handling of orders and technical arrangements, and tracing of shipments and

payments.

Figure PH shows a totally symmetrical Lattice with three connected Towers of five levels each. Each server makes requests to servers at the next level down, spread across all the towers. A "connectedness" parameter *c* describes the degree to which requests are spread across the other towers:

c_l = (mean requests from a server in level l to servers in each other tower)/(requests in the same tower)

The requests from level 4 to level 5 are not connected between the Towers, but kept local. In Figure PH the request rates y_l and $y_{l\varepsilon l}$ are made to add up to 1, so $y_l = 1/(1+2c_l)$.

A connectedness parameter of unity shows that the requests are equally spread across the towers, while a parameter of zero shows each tower isolated from the others at that level. When the towers are completely symmetrical the connectedness parameter does not affect the load at a server because as many requests flow in from other towers, as flow out to them. this is not true, however if one tower has more users or different execution or request count parameters.

Symmetrical Lattice

In a symmetrical lattice with identical Towers, connectedness does not change the total load on each server, it just moves it around. However it turns out that in a single-threaded lattice the mixing of requests between towers makes a large difference to capacity, because it increases the chances of waiting at a lower server, due to random interference between requests from different towers. Figure PHA shows how the service times increase due to longer blocking delays, and the throughput drops from .167 with *c=0,* to 0.98 with *c=1.0.* This is worth remembering for single-threaded servers.

TABLE 9. Effect of "Connectedness" on Lattice 1 (Figure PHA)

	Throughput, User Responses/sec					
"Connectedness" c	0	0.2	0.4	0.6	0.8	1.0
Thropughput						
(m_2,m_3,m_4) = (1,1,1)	0.167	0.114	0.103	0.100	0.983	0.980
Throughput						
(m_2,m_3,m_4) = (6,5,4)	0.455	0.452	0.450	0.450	0.449	0.449

However multiple threads make a big difference. Figure PHA also shows results for the "rule-of-thumb" levels of threads (m_2, m_3, m_4) = (6,5,4), and the same levels of connectedness. There is still a penalty compared to Figure PC, but it is small. We can effectively carry the one-tower analysis into the Lattice situation. Also, threads give robustness, and make the throughputs insensitive to the connectedness.

Real lattices are only roughly symmetrical. Regional or divisional elements of an organization are not of equal size, and they have specialized needs. We have discovered that a single tower is a good predictor for symmetrical lattive behaviour provided there are enough threads. Is this still true for unsymmetrical cases? To what extent can the towers be analyzed separately? Under what conditions is there a single dominant tower, or bottleneck server? We will consider:

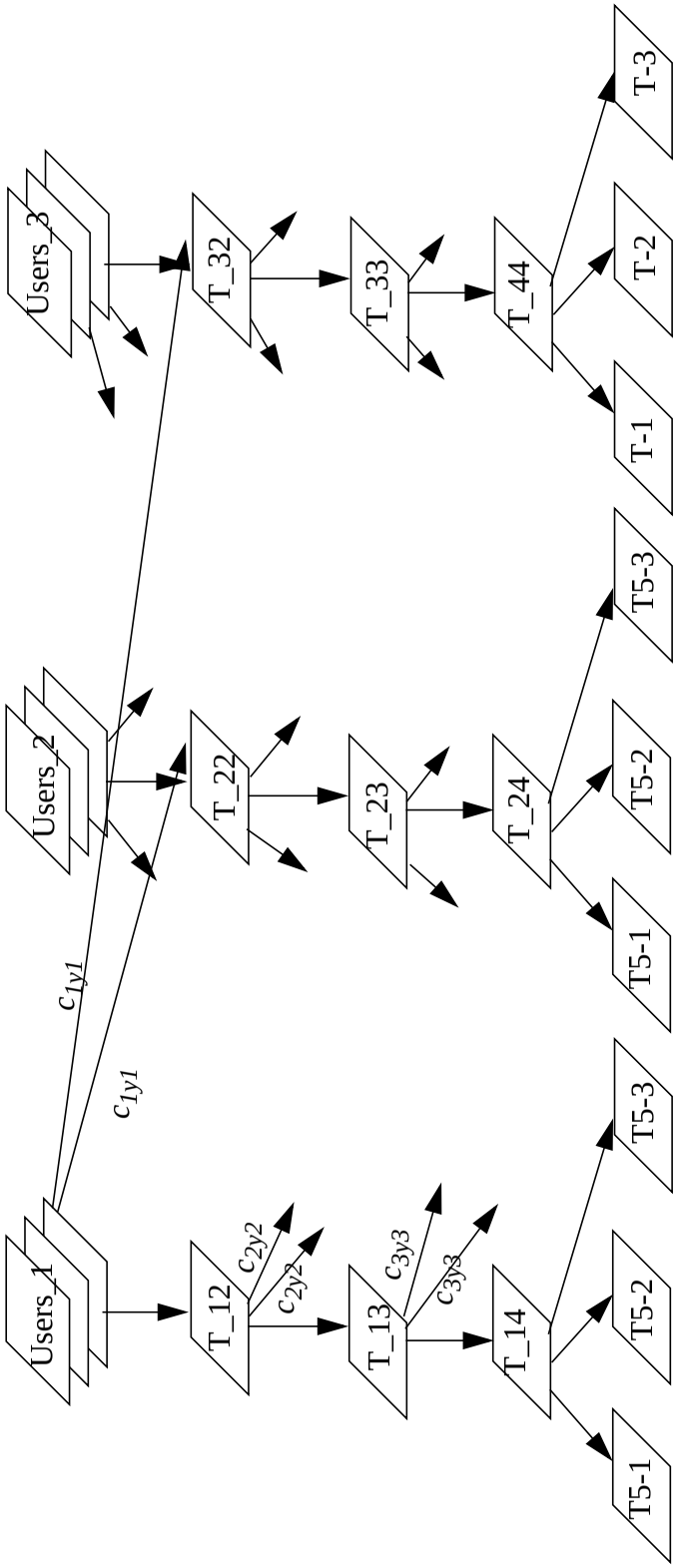


FIGURE 6. Lattice Pattern. (Figure PH)

- providing a higher capacity to the users attached to one tower,
- one server in one tower which makes an increased execution demand,
- the location of a new bottleneck when one is alleviated.

5.4. Peer-to-Peer Pattern

Up to this point all the software considered has been hierarchical with requests descending to servers. What happens in a system with no hierarchy, in which equal peer processes communicate? Such systems are important because of their robustness to failure and their symmetry. Examples arise in distributed databases, and distributed systems to manage facilities or services:

- in an air traffic control system each major airport is a node, and makes requests to neighbouring nodes for state updates, or to hand off aircraft to the next controller. Many of these interactions may be blocking, to ensure correct reception.
- in a distributed factory management system each production center may be a node with its own state, interacting with others to coordinate movement of goods through stages of processing and into the warehouse.

Some analysis is needed before a peer-to-peer system can be modelled with our MSS framework. It is necessary to understand the exact interaction behaviour. For instance, it would be a poor design that used single threaded tasks, and symmetrical blocking interactions as in Figure PLA. When a task makes a blocking request to a peer task, and waits for a reply, a situation is set up which might cause mutual request deadlock, with both tasks waiting for replies and ignoring their request queues. The first step in preventing this is to have multiple threads so a new thread can pickup an incoming request while another thread is blocked waiting for the peer. Even this can deadlock if all threads are waiting (although this is unlikely). A better design (and we suppose that most systems are actually built this way) would recognize that the processing of a request from the peer is different from one generated locally. For one thing it has been partly processed already. For another, the processing can usually now be satisfied at the one site. The Peer/Peer pattern builds these observations into the model, by dividing each task into parts, as shown in Figure PLB. One part Local (for local request handler) handles local requests generated by users, a second part Remote (for remote request handler) handles requests from the peer task(s), and a third part CSect handles critical sections shared by the first two. Local and Remote can be multi-threaded. The LRH and RRH tasks are distinguished to make the model clear; the actual software architecture may not have separate tasks for local and remote requests, but internally the functions associated with them should be recognizable.

The pattern is a special case of a Lattice. In Figure PLB, Task A is modelled by three pseudo tasks. ALocal handles all the requests from AUsers, by invoking AServices for requests which can be satisfied locally, and sending a request to BRemote for requests which can only be satisified at B. If there are more sites the model expands easily by spreading requests out to them also.

There are many design options which affect performance. Commonly TaskA and TaskB will be multi threaded so they can respond to remote requests. There may be common code and a common thread pool for Local and Remote. The difference between Local and Remote requests

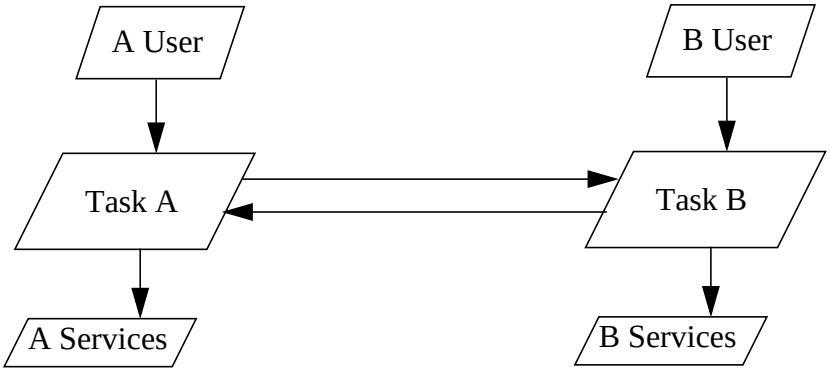


FIGURE 7. Equal, Symmetrical "Peer-to-Peer" Interaction. (Figure PLA)

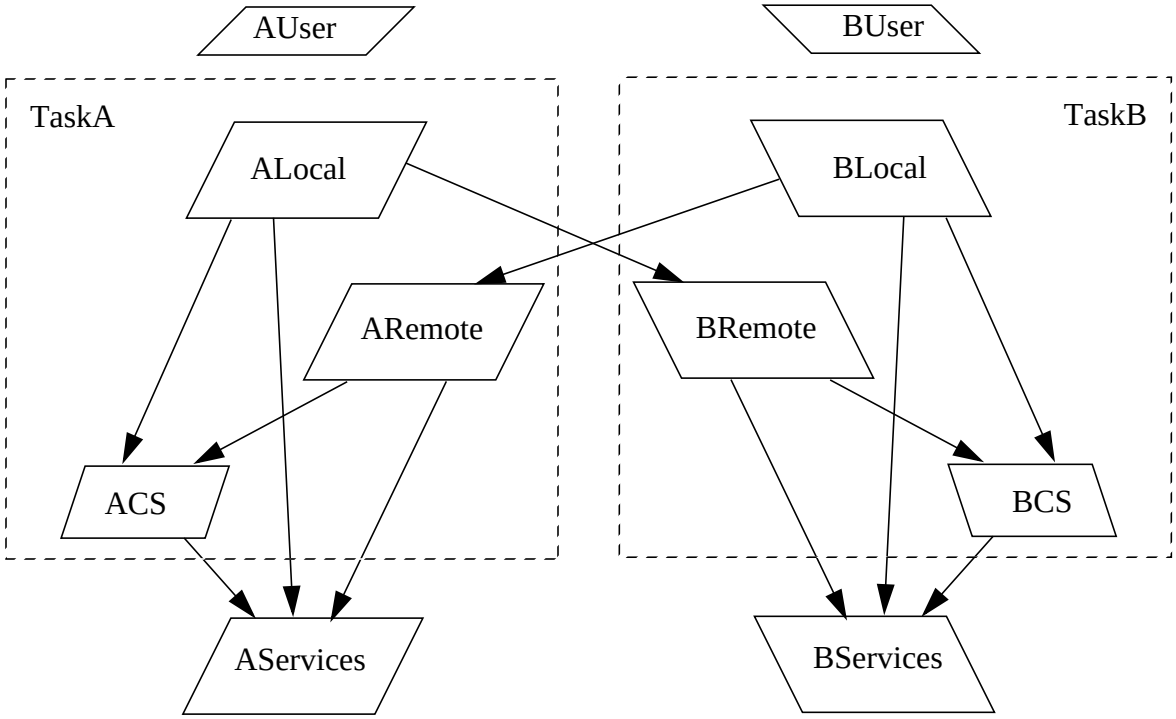


FIGURE 8. A Layered Set of Pseudo-Tasks Represent Task A and Task B. (Figure PLB)

may be only a flag in the request, which causes the path of further processing to follow the paths for Local and Remote psudo tasks in Figure PLB. An important detail may require a further addition to Figure PLB: the Figure translates to a Layered Queueing model with separate thread

pools for the Local and Remote pseudo task. A common thread pool for TaskA can be modelled by an additional speudo task AThread which serves both ALocal and ARemote, as in Figure PLC. Separate “entries” on AThreads are used to keep the strams of requests separate.

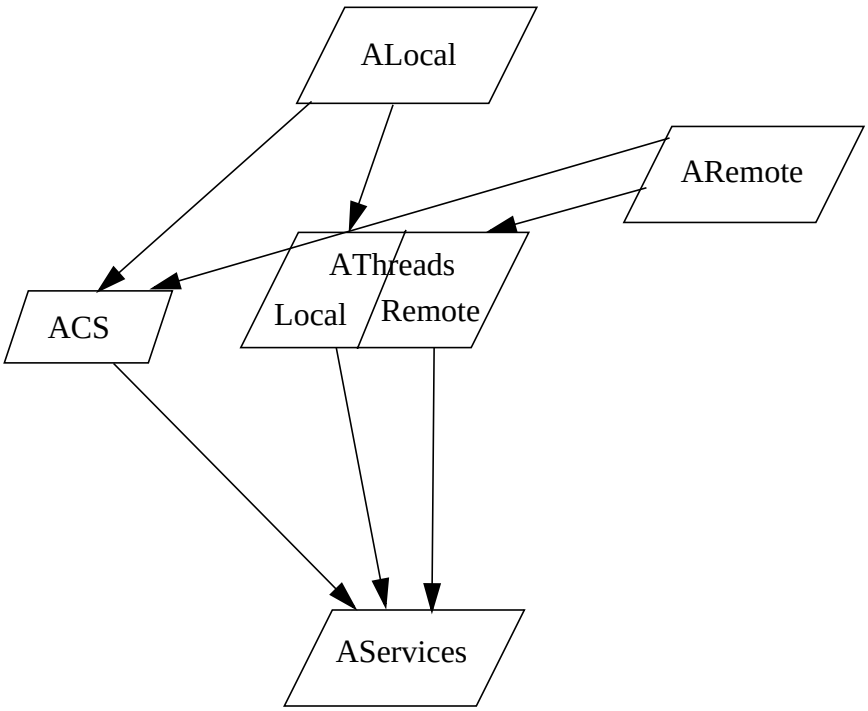


FIGURE 9. A Common Thread-Pool for ALocal and ARemote Modelled by a Pseudo-Task AThreads. (Figure PLC)

5.5. Pipelines with Rendezvous (No buffering)

Our layered model applies directly to pipelines in which the next stage must accept a data token before the previous stage is free to do more work. To apply it we introduce a *third phase* for handing on the data token, so there are three phases as follows:

phase 1: accept a new data token and acknowledge it;

phase 2: operate on it (Processing);

phase 3: send the output token or tokens on, and wait for acknowledgement.

[Remainder to come]

Performance - Oriented Patterns in Software Design (A multi-level service approach)

C. M. Woodside

Dept. of Systems and Computer Engineering

Carleton University, Ottawa K1S 5B6

(Draft version produced for classroom use, October 1996)

Chapter 6. P: Patterns

Using the small-scale patterns of the last chapter for internal task resources, execution and interaction, we can build and analyze performance models. The MSS(Res) framework of models describes a wide variety of distributed service systems for business computing, industrial automation, communications systems management, etc.

When we begin to analyze model results we find that certain arrangements of concurrent tasks recur and have characteristic performance attributes. These architectural level patterns will be included in our repertoire of "performance-oriented patterns". This chapter considers four architectural patterns which are extremely common in existing and proposed designs, and which are seen in classification of software architectures such as the one by Shaw [??].

- The "Tower" pattern is a layered set of servers showing vertical separation of functions, seen in descriptions of client-server and transaction processing systems. This is a simplified version of Shaw's "Client-Server" architecture.
- The "Lattice" pattern is a set of cross-linked Towers, representing layered service with several servers at each layer. This is a more general version of Shaw's "Client-Server" architecture and could represent a three-tier client-server system or a distributed transaction processing.
- The "Peer-to-Peer" is a model for symmetrical servers which exchange requests with each other. It contains a transformation which produces a special case of the Lattice Patterns.
- The "Flow" pattern represent pipelined processing. This is very common and is one of Shaw's categories. We consider also extended versions of this pattern with servers shared by pipeline tasks.

Further architectural patterns which incorporate fork-join behaviour patterns will be

studied in a later chapter.

6.1. The “Tower” Pattern

The name “tower” will be applied to a set of layered servers, which are, as it were, piled one on top of the other to make a tower of tasks. In each of the middle layers there is just one server, while at the top there may be many users, and at the bottom there may be many servers. The basic pattern is shown in Figure PA, with a set of N_1 user tasks, in the top layer, layer 1, making requests to a single server in layer 2, which in turn makes requests to a single server in layer 3, down to where layer *L-1* makes requests to N_L servers in layer *L*. We have already seen that a database system may have layers like this, with user tasks running on desktops, with a Transaction Manager, a Data Manager, and a File Server, and with a set of disks at the bottom.

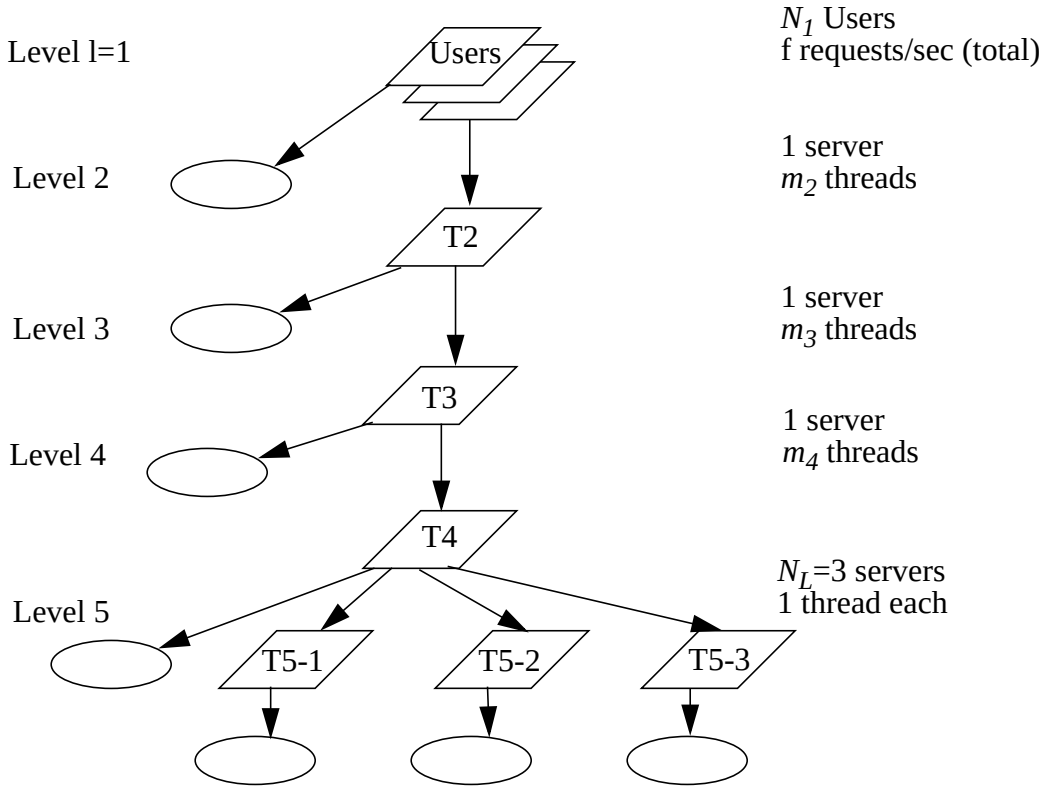


Figure 6.1. Tower Pattern with Five Levels (Figure PA)

The Tower pattern has the usual workload parameters, which are the same for each tasks in a given level. They will each be labelled with subscripts for the level *l*, counting down from the user tasks at the top:

- there are N_l tasks (assumed symmetrical) at level *l*, identified as “T_l”.
- The task at level *l* has m_l threads, from $l = 2$ to *L-1*.
- The tasks at the bottom are single threaded.

- Each task at level l has an average host demand of D_l sec, and makes an average of y_l requests to each server at the next lower level.
- As a result each task at level l is invoked on average v_l times per user task cycle, where $v_1 = 1$ and $v_l = y_1 y_2 ... y_{l-1}$, $l = 2, 3.....$
- If f is the total rate of requests from the user level, the invocation rate of each task at layer l is $f_l = v_l f$.
- The N_1 user tasks at the top are single threaded and execute in cycles. A cycle begins with the user "thinking" for Z seconds, then making a request which the task executes. The end of each cycle of each user task begins the next cycle, with a cycle time of X_1, including the delay Z. The user response time is $X_1 - Z$.
- the total response rate over all user tasks, f responses per second, is given by $f = N_1/X_1$.

A software bottleneck is a serious concern for this pattern. It may occur where a resource-constrained server makes blocking requests to a lower-level service, thread, or resource. The effect is stronger where there are more lower-level servers in the fan-out, but just one lower-level server plus the constrained task's own processor is enough. As well as thread resources, the distribution of host demands and other requests over the levels determines the severity and location of a bottleneck.

Task saturation is indicated by utilization, the fraction of time the task is busy. The task source time, or the read service time of $m_l > 1$, is X_l and includes blocked time waiting for requests to lower server to complete. At the bottom level L the tasks only have host execution, so $X_L = D_L$. We will consider two kinds of utilization:

- host utilization per task at level l is $HU_l = f_l D_l = f v_l D_l$,
- task utilization at level l is defined per thread (if there is more than one), giving

  $U_l = f_l X_l / m_l = f v_l X_l / m_l$, $l \geq 2$. In this case saturation is indicated by U_l approaching m_l.
- task utilization of each user task is $U_1 = f X_1 / N_1$, since the utilization is shared among the N_1 user tasks,
- task utilization of a task at level L is $U_L = f v_L X_L$, since they are single-threaded

Basic Case: Tower1

Let us begin with Tower1 as shown in Figure PA. It has five levels, with 10 user tasks at level 1, then three middle levels with single-threaded servers T2, T3, T4, each with its own processor, and finally three identical bottom-level servers T5_1, T5_2, T5_3. Each server is invoked once per response and has one unit of execution demand ($v_l = 1$, $D_l = 1$). When this system executes, the service time of T4 is four units, one unit for itself and one unit for each of the three bottom-level servers. T3 adds one unit for itself, for a total of five, and T2 takes six units for its task service time. Because the tasks are single-threaded, T2 must finish with one response before it starts the next. The lower level tasks never have more than one request to deal with at a time; T2 effectively sequentializes the entire system. The T2 service time of six units defines the maximum throughput capability as one response every six units, or $f = .166$, approximately. At this maximum throughput, each processor is used at only one-sixth of its capacity. This is an extreme case of a software bottleneck.

What this example will show is that

- when any resource is saturated, all finite higher-level resources are saturated too, while lower-level resources are not,
- bottleneck relief must include the saturated resource, but may have to include other resources too, to get full potential value.
- bottleneck relief will be provided in these examples by multi-threading of tasks, and there is a rule of thumb for how many are needed,
- when there is contention at a resource, its service time increases when throughput goes up, making it difficult to predict the limits from light-traffic measures.

TABLE 6. Performance of Tower1 Shown in Figure PA (Figures are for each task at level *l*) (Total Throughput *f* = 0.166 requests/sec) (Figure PB)

Level *l*	Task Utilization U_l	Task Service Times X_l	Host Utilization HU_l
1 (Users)	1.0	60	0.166
2	1.0	6	0.166
3	0.833	5	0.166
4	0.666	4	0.166
5	0.166	1	0.166

Figure PB shows the task resource saturation at different levels in the system, for different throughputs up to the maximum. At the lower values, the users have a delay between the completion of one request and starting a new one, while at the highest value there is none. The device utilizations are not shown but they are numerically equal to the throughputs. The lowest curve is for a long user "think time" which gives a low total request rate; the second is for a moderate think time giving a moderate rate. Notice how the task utilizations build up at the higher levels, while the device utilizations are the same over the levels. Blocking delays pile up at the higher levels and cause longer task service times, which reflect in higher task utilizations. Because T2 has a single thread, there can only be one active response in the system below it. Since there is no message queueing at levels 3 to 5, the way the delay piles up is very simple. Each task's service time X is the sum of its own host demand D and the service times of the tasks below it; task utilization is proportional to X. Therefore, a server below a single threaded task cannot be fully utilized; some of the time it must be waiting for the next request.

Even though this is a very oversimplified example, it is worth understanding. We can see that:

- the bottleneck at level 2 is due to it being single threaded; for higher throughput we must be able to process several requests at once, which implies multi-threading at levels 2 down to 4. On the other hand, threads in level 5 would do no good as these tasks do not block.
- if we cannot increase the threading levels then the system will continue to be sequential at and below level 2. We could get equivalent or better performance by running all the tasks on a single processor at and below level 2. It would be even better to merge them into a single task, because it would reduce demand for intertask communications overhead! (This overhead has not been separated out here, but is certainly present.)
- the first improvement would be to introduce multiple threads to level 2; then the bottleneck would move down to level 3. Second, multi-thread level 3, and it would move to level 4. Finally with multiple threads at level 4, the bottleneck should move down to level 5 and also to the devices of all levels. Because the heaviest device utilization in Figure PB(d) is 1/6, we can in principle search for a sixfold increase in throughput, when we introduce multiple threads. A sixfold increase would make $f = 1$.
- How many threads to introduce? If one only introduces them in level 2, two threads is enough, because one can be executing while the second one is blocked while T3 executes for it. More would just have to wait for T3 to begin serving them. A similar argument could be made at level 3, but for level 4 (because there are three servers at the next level down), more threads are useful. But then, if one increases the number at level 4 it will pay off to increase them also

at the higher levels.

Thread Rule of Thumb #1

- A simple rule of thumb when (as here) there is no phase-2 work, is that the threads at a level should be the sum of the threads in all the servers at the next level down, plus one, to avoid idle time on a task due to blocking. On the other hand if for any reason a higher level task is constrained in its number of threads, lower level tasks need not have more. For level 1, the N_1 user tasks are treated as threads, so the most threads we expect to be useful is 10.

If the rule of thumb is applied to Tower 1 it says level 5 has 3 tasks of one thread, so level 4 requires 4 threads, level 3 requires 5, and level 2 requires 6. Level 1 has 10 users each of which is constrained to be a single thread. We will focus on the levels 2 to 4, which have (m_2, m_3, m_4) = (6, 5, 4).

Multi-threading

Figure PC shows the results when one introduces multithreading level by level, starting at the top at level 2. The rule of thumb is used to determine m_l to be one more than the sum of values for the next level down. The figure shows the mean number of busy threads in part (a), and the relative saturation of thread resources (mean busy threads over total threads) in part (b), and the mean task service time in part (c). Look first at the values at the right side of the table, as threading is introduced gradually:

- (m_2, m_3, m_4) = (2,1,1) and throughput 0.20/sec;
- (3, 2, 1), and throughput 0.22/sec;
- (6, 5, 4) and throughput 0.47/sec.

The big payoff really comes with threads in level 4, although providing threads only at level 4 would have no effect at all!

The rule of thumb may underestimate the number of threads that can provide a benefit, basically because threads in a server may compete with each other for the next server down the tower (which could not happen when there was only one). It is really necessary to evaluate the effect of threading. Higher numbers of threads give some additional improvement, indicated by the other curves in the Figure:

- (m_2, m_3, m_4) = (7, 6, 5) and f = 0.55/sec,
- (8, 7, 6) and throughput 0.58/sec,
- (9, 8, 7), and throughput 0.62/sec,
- (10, 10, 10) and throughput 0.65/sec.
- (∞,∞,∞) 0.65/sec.

This pattern frequently underperforms, compared to expectations. Why can this example not exceed 0.65 responses/sec even if the users are flooding it with input, there are infinite threads, and each task separately can handle 1.0/sec.? The answer lies in (1) the relatively small number of user tasks, just 10, compared to the amount of work for each response (7 units). Even with more processors, we could never exceed 10/7=1.42 responses/sec. and (2) in random interference of requests, due to the variability in the execution demands. For these evaluations the host demands

were random and exponentially distributed, which is higher variability than is found in some applications, but lower than others. It makes a curious trade-off. With deterministic times there would be no improvement above the rule-of-thumb values (6, 5, 4), at which a full 1.0 response per second is obtained. Random interference throttles back the capacity at (6,5,4) to less than half of that, but then allows additional threads to regain part of the difference.

TABLE 7. Multi-threaded Tasks in Tower 1: Throughput, Task Saturations and Other Measures. (Figure PC)

Threading Level Cases, defined by (m_2, m_3, m_4)								
	(∞,∞,∞)	**(10,10,10)**	**(9,8,7)**	**(8,7,6)**	**(7,6,5)**	**(6,5,4)**	**(3,2,1)**	**(2,1,1)**
Throughput *f*	**0.65**	**0.65**	**0.62**	**0.58**	**0.55**	**0.47496**	**0.223521**	**0.200347**
Level *l*	(a) Mean Busy Threads = Task Utilization (U_l)							
1	10	10	10	10	10	10	10	10
2	9.34	9.35	8.8	7.95	7	5.51	2.90	.96
3	7.8	7.8	7	6.2	5.36	3.89	1.64	1
4	6.24	6.22	5.4	4.7	4	2.75	0.89	0.8
5	0.65	0.65	0.62	0.59	0.55	0.47	0.22	0.2
Level *l*	(b) Task Utilization per Thread (U_l/m_l)							
1		1	1	1	1	1	1	1
2		0.94	0.98	0.993	1	0.92	0.97	0.98
3		0.78	0.88	0.89	0.89	0.78	0.82	1
4		0.62	0.77	0.78	0.8	0.69	0.89	0.8
5		0.65	0.62	0.59	0.55	0.47	0.22	0.2
Level *l*	(c) Task Thread Service Times (X_l)							
1	15.38	15.38	16.13	17.24	18.18	21.05	44.74	49.91
2	14.37	14.38	14.19	13.71	12.73	11.60	12.98	9.77
3	12	12	11.29	10.69	9.75	8.19	7.32	4.99
4	9.6	9.57	8.71	8.10	7.27	5.78	4	3.99
5	1	1	1	1	1	1	1	1

Only a model can predict the balance of these factors. For example,

- if the number of users is increased to 25, and variability is kept the same, the throughput for infinite threads goes up but only to 0.83/sec... (PDM)
- if the variability of execution times is reduced, so its coefficient of variation is 0.5, then f = ... (PDN)

Effect of Variability in Execution Times

High-variance execution and communication behaviour has been observed in networks and in execution statistics and has quite serious performance effects. [refs] High variance or “Self-similarity” was first observed in network traffic when it tends to nullify the advantages of large

scale and multiplexing of traffic. High variance in software execution times increases contention delays and reduces average throughput. However some of that reduction can be gained back by exploiting multi threading.

Consider Tower 1 with rule-of-thumb thread levels of (6, 5, 4), mean service demands of 1.0 units and execution-time standard deviation σ_D of 0 (deterministic), 1.0 (exponential distribution), and 10 (hyper-exponential), and then consider the gain obtained by making the thread units infinite.

TABLE 8. Variability of Execution Demand (Figure PDR)

		Thread Levels		
		(m_1,m_2,m_3) = 6,5,4	**Infinite Threads**	
Execution Demand Std. Dev. σ_D	**Throughput f Resp/sec.**	**Mean Busy Threads ($\overline{m}_1$, $\overline{m}_2$, $\overline{m}_3$)**	**Throughput f Resp/sec.**	**Mean Busy Threads ($\overline{m}_1$, $\overline{m}_2$, $\overline{m}_3$)**
0	1.0	(6, 5, 4)	1.0	
10	0.47	(.55, .39, .27)	0.65	

In these results we see the interesting fact that higher variability at σ_D = 10 reduces throughput dramatically and increased thread levels restore only a fraction - about a third - of the lost throughput. User response time is even more dramatically affected.

Because of the complexity of the interactions even in a relatively trivial architecture like Tower 1, a model is essential for determining the risk posed by variability. Thread levels alone, unfortunately, do not solve the problem and restore the capacity. Another mechanism for achieving improvements is through priority scheduling, essentially by reducing the priority of threads that have executed for a long time. With high-variance jobs a thread that has already had a long time is more likely to need yet more, so they reveal themselves. Unfortunately this kind of dynamic priority scheduling is not yet common on workstations.

Critical Sections Limit Thread Effects

Unfortunately we cannot make all processing multithreaded. Often there is interference between threads because they share resources or data, which requires a critical section in which only one of the threads can execute at a time. It may then be simplest to restrict a task to a single thread and concentrate on making it efficient. Only a critical section which covers local execution only, without any service requests, will have almost no performance effect.

A critical section which covers some but not all service requests is modelled as shown in Figure PE. The pseudo-task CS includes that part of T3 which is within the critical section, and has a single thread. Waiting for CS models the waiting for the critical section. CS includes both execution and some service requests to T4. The execution in T3 and requests from T3 to T4 are divided between the critical section, and non-critical execution of each thread, in a ratio β:(1 - β). With β=1, T3 is effectively single-threaded, while with β = 0 the critical section disappears.

What we see in Figure PE is how the choke-point due to the critical section spreads congestion back up into the system, so that attempts to correct the problem by changes above that point are doomed to failure. The number of threads that are worth providing above T3 depends on how many threads can be used effectively at T3, and this drops as β increases.

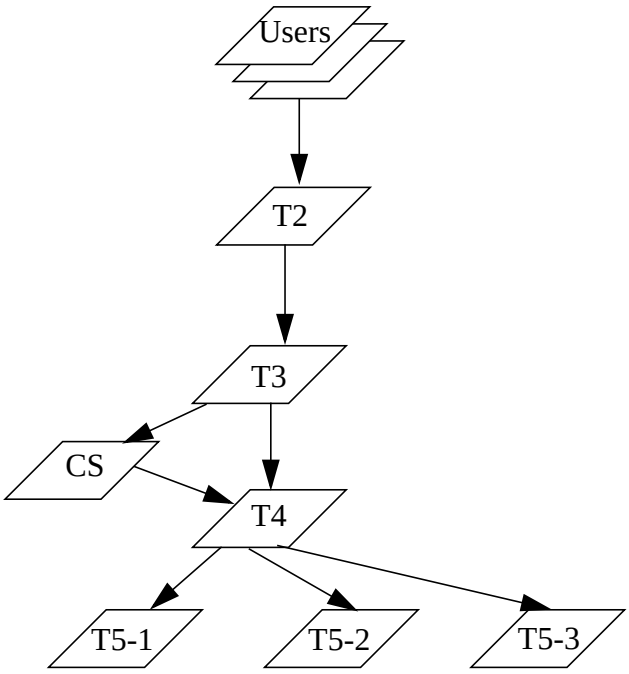


Figure 6.2. Tower1 with a Critical Section Modelled by Pseudo-Task CS. (Figure PE)

How can we estimate the thread resources needed, with such complicated factors at play? A method for estimation is considered next.

TABLE 9. A Critical Section in T3 with Thread Levels of Tower 1 (m_2, m_3, m_4) = (6, 5, 4). (Figure PESIM)

	Critical Section Ration β (β = fraction of T3 execution within the critical section)			
Throughput *f*	**0.0001**	**0.33**	**0.67**	**1.0**
	0.5	**0.36**	**0.26**	**0.2**
Level *l*	(a) Mean Busy Threads			
1	0	10	10	10
2	6	6	6	6
3	4.5	4.6	4.74	4.8
CS	0.14	0.86	0.98	1.0
4	3.21	2.01	1.22	0.8
5	0.5	0.35	0.26	0.2
Level *l*	(b) Task and Critical Section Utilization (per Thread)			
1	1.0	1.0	1.0	1.0
2	1.0	1.0	1.0	1.0

TABLE 9. A Critical Section in T3 with Thread Levels of Tower 1 (m_2, m_3, m_4) = (6, 5, 4). (Figure PESIM)

	Critical Section Ration β (β = fraction of T3 execution within the critical section)			
Throughput *f*	**0.0001**	**0.33**	**0.67**	**1.0**
	0.5	**0.36**	**0.26**	**0.2**
3	0.9	0.92	0.95	0.96
4	0.8	0.50	0.30	0.2
5	0.5	0.35	0.26	0.2
Level *l*	(c) Task Thread Service Times			
1	20	27.8	38.5	50.0
2	12	16.7	23.1	30
3	9	12.8	18.2	24
4	6.42	5.6	4.7	4
5	1	1	1	1

Prior Estimation of Thread Resources

Is there a simple way to estimate the desirable number of threads to provide, that is more accurate than the rule of thumb of one more than the sum over the next level down? An approach which has the advantage that it deals with critical sections, second phases and other potential complexities in the interaction patterns, is to analyze for infinite threads in every task, but including any necessary critical sections. For Tower 1, the diagram in Figure PEM with infinite threads at levels 2 to 4 shows the idea. Levels 1 and 5 are left single threaded because at level 1 threads make no sense, while at level 5 they make no difference. The results show the mean number of active threads:

- $\overline{m_l} = f_l X_l$ = mean active threads per task at level *l*

TABLE 10. Estimating Thread Levels from Models with Infinite Levels ((m_2, m_3, m_4) are infinite) (Figure PEM)

	Critical Section Ratio β (β = fraction of T3 execution within the critical section)			
Throughput *f*	**0.0001**	**0.37**	**0.67**	**1.0**
	0.69	**0.36**	**0.24**	**0.2**
Level *l*	(a) Mean Busy Threads			
1	10	10	10	10
2	9.3	9.64	9.76	9.8
3	7.9	9.13	9.46	9.56
CS	0.29	0.83	0.96	1
4	6.12	2.04	1.19	0.8
5	0.69	0.36	0.24	0.2

In Figure PEM, infinite threads give only a modest increase in throughput over Figure PD, from 0.62 to 0.67, even with no critical section. The mean number of busy threads is about the same. When the critical section fraction increases, moreover, we can see how rapidly the useful

number of threads drops. When using these results, it will be useful to make the actual number of threads a little bigger than this; when in doubt pick a configuration and run an analysis.

Most interesting, even with infinite threads and no critical section, the device utilization is only 68%, indicating that 32% of the processing capacity is still not being utilized. This is due to the layered structure, random contention, and the relatively small number of users. However calculations with 20 and 30 users saw device utilizations rise only a little (to 82% and 88% respectively) while the response time skyrocketed from 13 units for 10 users, to 24.3 units for 20, and 34 units for 30.

CONCLUSION?

Delay and a Lightly Loaded Tower Pattern

All of this section has considered only the maximum throughput obtainable from a tower-patterned subsystem, rather than the delay to an input request. It turns out that the changes which favour higher capacity also mostly favour smaller response delays in this case. If delay is the important factor and the system is lightly loaded (e.g., there is a longer think time Z between requests), then a satisfactory response time may be obtained with fewer threads. The number of threads required for the servers will be lower, without penalty, just because they would mostly be idle.

Unbalanced Execution Demands

Figure PEQ

Summary

Thread resources must be considered in designing for performance, but they must be applied consistently to all tasks, and they cannot overcome other resource constraints such as a critical section between threads.

on unbalance over levels? here or earlier?

6.2. Variations on the Tower Pattern

Tower1 is oversimplified in two broad ways. First, real systems are less symmetrical. Their demands are not balanced between levels or tasks, levels may share host processors, the request values are not unity between levels, and there may be second phases of service. Second, they may be interconnected to other tasks, so there may be more services requested at intermediate levels, and requests may come in from other subsystems to various levels. Here we will consider request flows in and out of a tower, and second phase effects.

Fanout:

A vertical sequence of tasks in a larger system may be identified for careful study as a tower pattern, if the links between it and the rest of the system are very weak. If they are not so small as to be completely ignored, requests made to tasks outside the tower can be represented as appended or *fanout* requests. They have the effect of inducing additional delay to threads of the task in the tower making the request. Figure PG shows Tower 1 with task *T3* making y_{3A} fanout requests to the appended task *T3A*, with service time 2 units.

If *T3* is single-threaded it blocks for longer and the tower performance declines; if it can be

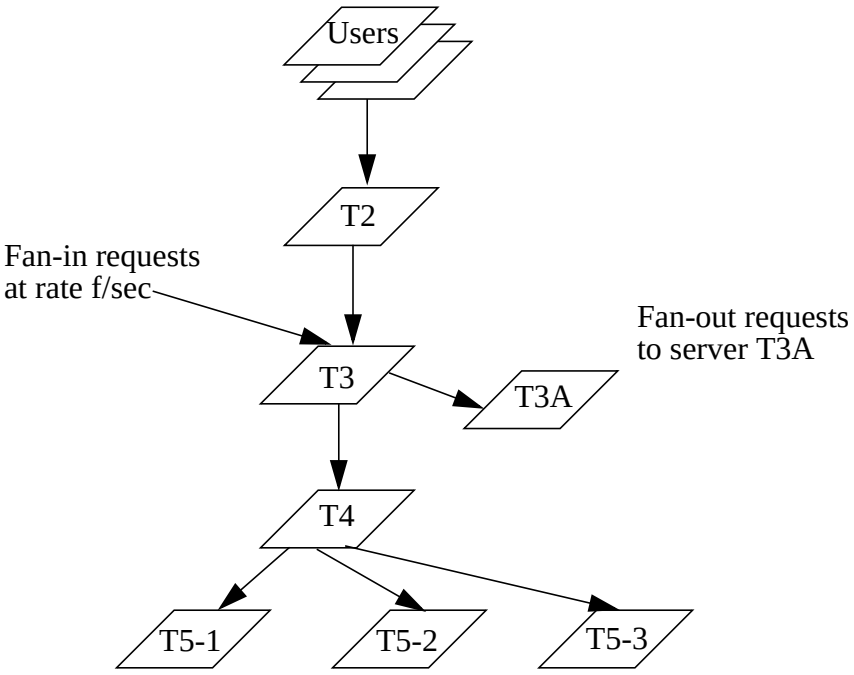


Figure 6.3. Fan-in and Fan-out Requests in Tower1. (Figure PG)

multi threaded then part of the performance is recovered, as shown in Figure PGF. The rule of thumb suggests that m_3 be one plus the sum of m_4 and the thread count of *T3A*.

Second-phase execution at the appended server changes the picture totally; if it is mostly second-phase there will be almost no effect, as the reply will come back almost at once.

TABLE 11. Effect on User Throughput of Additional Fan-out Requests for *T3* to *T3A*. (Figure PGF)

	User Throughput				
y_{3A} (mean requests to *T3A*)	0	0.5	1.0	1.5	2.0
Single-threaded:	0.167	0.143	0.125	0.111	0.10
(m_2,m_3,m_4) = (6,5,4)	0.465	0.421	0.358	0.288	0.231
Infinite threads	0.67	0.626	0.481	0.331	0.249

Fan-in:

Requests may similarly flow into a tower pattern from other subsystems. Suppose they are represented by an open flow of requests at rate f' arriving at level L', as shown in Figure PGH. Now, as f' increases it tends to saturate that level, and the response time $R_{L'}$ goes up due to queueing delays. The blocking time to the next level up increases, and this makes a higher threading level there more worthwhile.

Second Phases:

"Second phase execution" comprises activities carried out by a server after the reply is sent. In real RPCs there is at least a small amount of second phase in the PrepareRcv activity shown in Figure PC, getting ready for the next message reception. But many servers are designed so the reply is sent as early as possible, with various postponable execution done in the second phase, such as buffer deletion, or writing and closing files. File servers with cached writes are a simple and ubiquitous example of second phase, for the actual write is done after the data is stored in the cache and the client is acknowledged.

The effect of a task's work being in the second phase on a Tower is interesting. In a lightly loaded Tower it results in shorter response times, because a client is blocked less and the server executes in parallel. In a heavily loaded Tower with a bottleneck at that task, however, the effect on the maximum capacity is small. With second phases at a certain level, the level above may require more threads to reach its full capacity, because second phases increase the possibility of queueing; a task may even have to queue when it is the only requester to a server. With increased possibility of queueing additional threads may sit blocked, while others work on new requests.

Figure PGL

For a Tower with a fan-out, work which is moved into a second phase of the appended task can improve capacity.

Tasks which Share Host Processors:

Where tasks in a Tower share host processors it breaks the pure Tower pattern, for two levels then share a common server. This may reduce the number of threads that can be used with advantage. For instance if two neighboring levels share a host processor, to a first approximation the two tasks could be considered as one in rule of thumb. It should be modified to say that they have the same number of threads, rather than the upper one having one more.

In general, situations have to be considered in detail with a full evaluation.

6.3. Partitioning and Replication

When a server is saturated and multi-threading has done all it can (or is inappropriate), it can be further scaled up by partitioning it, or replicating it. Either way, the one server is replaced by *n* servers. These solutions are particularly appropriate if there are geographical or functional domains that can be served separately. Partitioning divides either the function, or the data of the original server *A* among the *n* replacement servers A_i*, so that for a particular service (or for particular data) a request must be made to the one server that can provide it; replication provides *n* complete copies of *A* including its data. The partitioned components are in some sense simpler than the original, but the client must now choose between them; the replicated components do not require changes to the clients.

Partitioning and replication are substitution patterns inside a larger architectural pattern such as a Tower. They can be introduced to remove a bottleneck at some server. They are a more expensive change than introducing multi-threading, since they normally imply separate hosting of each rep-

lica or partition. We will compare them to multi-threading, within the Tower of the previous sections, when one level is modified.

Partitioning is the simpler solution to understand. It is a module replacement pattern, as illustrated in Figure PJC. Server *B* is partitioned, along with a further server *C* that it uses. The black dots indicate the interface points of the replacement, where they partition elements B_i and C_i are connected. There is a Router module added to each of the users *A* of the original server *B*, to choose which partition to use, giving the new user *A**. There are the *n* partitioned servers B_1* to B_n*, and for each B_i*, a processor and perhaps a copy or partition C_i* of some server *C*, or some other subsystems associated with *B*. For instance if *B* is a database server then there will usually be a file system attached to it, which will be partitioned into a separate file system C_i* for each B_i*. In data partitioning, each B_i* will have the full set of entries, but their demand parameters may be smaller since they access a smaller volume of data; in function partitioning the B_i* will have different subsets of entries, but the entries will have the same demands. In general the demand parameters of B_i* and their subsystems may be functions of the number *n* of partitions.

In the Tower1 architecture we will suppose level *l* to be partitioned into n symmetrical part-servers, each with all the functions of the original and the same demand parameters. The threads at the level above make *yl+1/n* requests to each part-server.

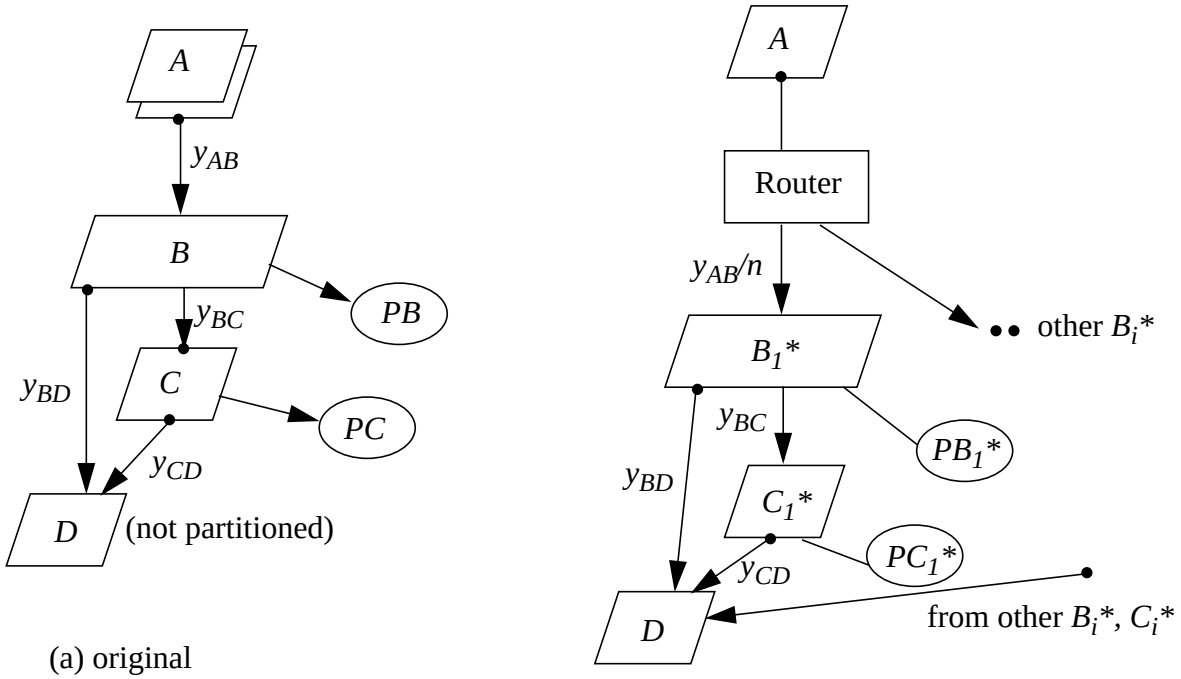


Figure 6.4. A partitioning Pattern for a Server *B* with an Associated Task *C*. (Fig. PJC)

**********Results and discussion.

**********Would it be useful to consider just a client and a replicated server first? The server has a disk, and is already multithreaded

Another Partitioning pattern (considered later) invokes separate partitions in parallel to execute parts of a large operation.

Replication also divides the service requests into n groups, but usually one client always goes to the same replica, since they are all equivalent. It might for instance use the closest one. The pattern, illustrated in Figure PJF, has to allow for an additional aspect not required for partitioning, which is replica coordination operations. For instance any updates made to one replica must be propagated to all of them, and an update transaction must acquire the equivalent of a write lock. Techniques for replica coordination are discussed in [Taylor and Trantafilou], and the figure roughly represents one of their techniques called..... For an update transaction a majority of so-called “primary copies” of the data item must be write-locked, then after the update is done at the one replica it is propagated to all replicas. We will assume all copies are primary copies to avoid the detailed consideration of this factor, which is described in [..]. This generates inter-replica messages which are shown as going to a coordination server at each remote replica site which describes the coordination workload but not the actual lock resource (which is not represented in this pattern, for simplicity).

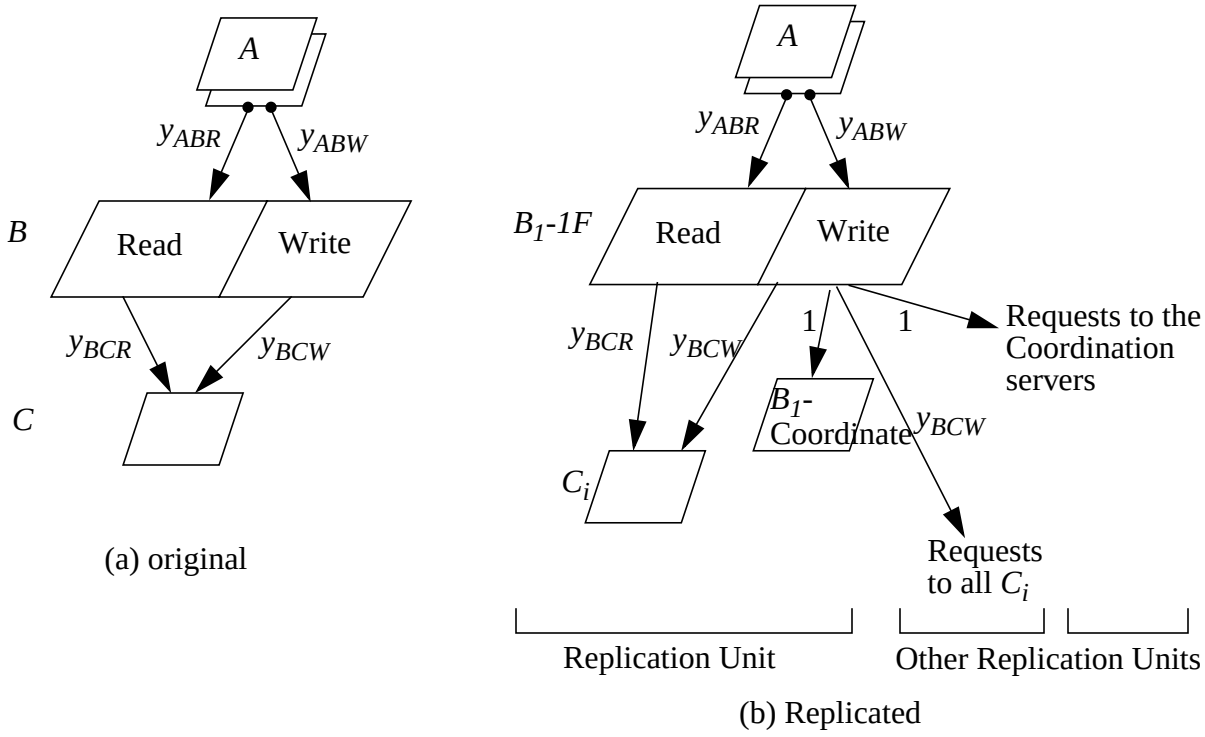


Figure 6.5. A Replication Pattern for a Server B with Reads and Writes. (Figure PJF)

The demands are strongly affected by the update fraction in the request stream. If the read and update requests are mingled, this fraction (call it γ) must be known. The figure assumes that the two fractions have been divided and directed to two separate entries Read and Write, in the original server *A*.

The replica Read entries are unchanged. The replica Write entries include requests to half the other sites' coordination servers (half of the primary copies) for write locks, and to all sites for update propagation. The Write entry of the coordination server is shown with the same demands as a Write in *A*, going directly to all storage servers.

Partitioning puts an additional load on the clients, but a smaller load on the servers because it avoids the coordination workload. Thus if the servers are the bottleneck, partitioning will give higher performance than replication, but replication is siompler to program and offers additional benefits for reliability.

Examples of partitioned subsystems: departmental data bases, RAID disk arrays, segmented memories, multiple buses.

Examples of replicated systems: mirrored web sites

(Examples of replications for reliability, not primarily performance: mirrored disks,..)

It was said earlier that special advantages occurred if the system can be divided along geographic or functional lines. Then the group of users of each replica or partition is defined by the division. There may be savings in communications costs and delays. In the case of partitioned systems each group of users probably is not completely isolated from the other partitions, but has a kind of concentration on its local partition. Say a fraction β of the users' requests go to the local partition, and the rest are equally distributed. In general the performance improves as beta increases; this will be examined in the next section, which describes a partitioned version of a Tower, called a Lattice.

6.4. Lattice Pattern

A lattice is a set of interconnected, more or less similar Towers, giving a diagram which looks like a lattice-work for climbing plants (e.g., Figure PH). It arises when two or more similar systems are connected together, for example

- a government social services agency has regional offices, each with its own databases of clients, budgets, personnel, etc., but they are linked together for consolidated reporting and to deal with cases which move from one region to another,
- divisions of a company, even though co-located, have their own information systems, but these are connected together for rapid and uniform handling of inter-divisional transactions and joint operations, and for gathering data for upper management.
- a company and its suppliers provide some restricted access to each others information systems to speed up handling of orders and technical arrangements, and tracing of shipments and payments.

Figure PH shows a totally symmetrical Lattice with three connected Towers of five levels

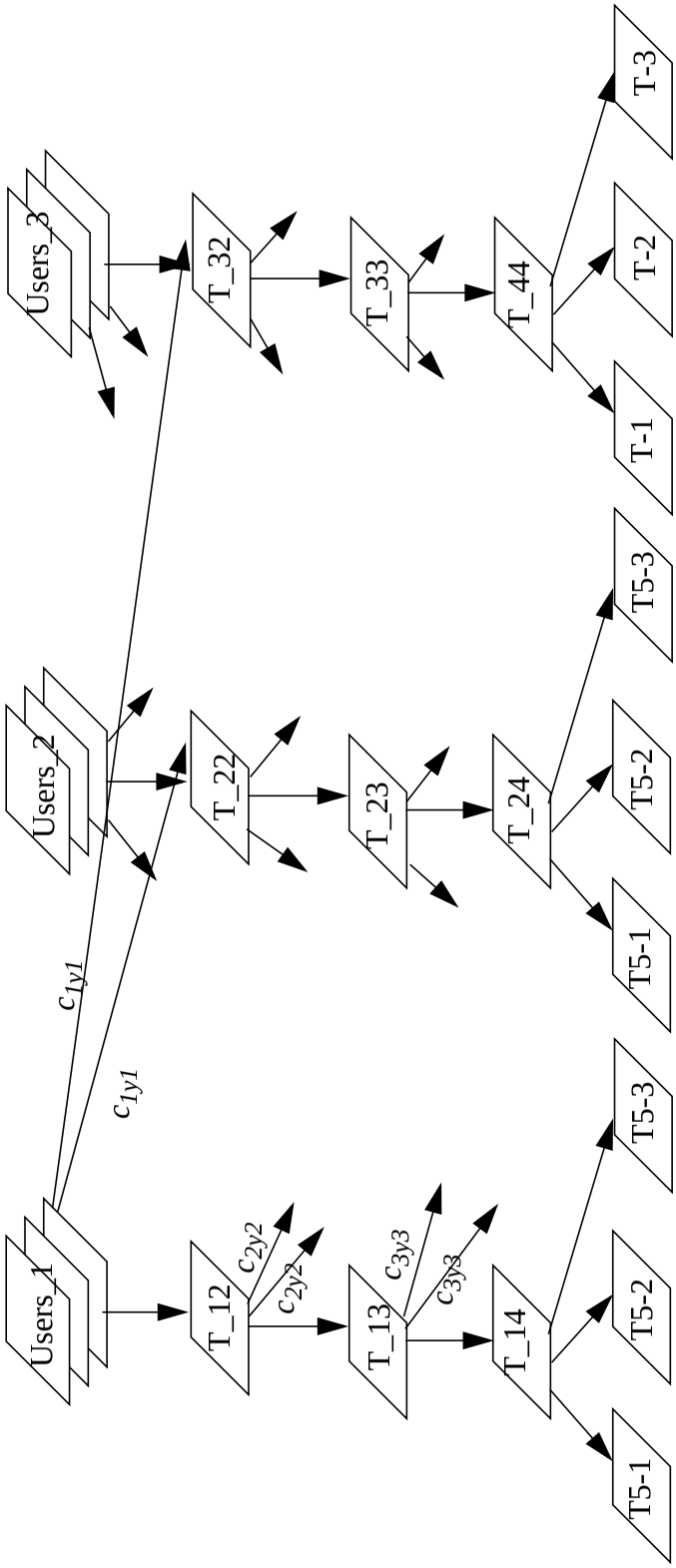


Figure 6.6. Lattice Pattern. (Figure PH)

each. Each server makes requests to servers at the next level down, spread across all the towers. A "connectedness" parameter *c* describes the degree to which requests are spread across the other towers:

c_l = (mean requests from a server in level *l* to servers in each other tower)/(requests in the same tower)

The requests from level 4 to level 5 are not connected between the Towers, but kept local. In Figure PH the request rates y_l and $y_{l\varepsilon l}$ are made to add up to 1, so $y_l = 1/(1+2c_l)$.

A connectedness parameter of unity shows that the requests are equally spread across the towers, while a parameter of zero shows each tower isolated from the others at that level. When the towers are completely symmetrical the connectedness parameter does not affect the load at a server because as many requests flow in from other towers, as flow out to them. this is not true, however if one tower has more users or different execution or request count parameters.

Symmetrical Lattice

In a symmetrical lattice with identical Towers, connectedness does not change the total load on each server, it just moves it around. However it turns out that in a single-threaded lattice the mixing of requests between towers makes a large difference to capacity, because it increases the chances of waiting at a lower server, due to random interference between requests from different towers. Figure PHA shows how the service times increase due to longer blocking delays, and the throughput drops from .167 with *c=0,* to 0.98 with *c=1.0.* This is worth remembering for single-threaded servers.

TABLE 12. Effect of "Connectedness" on Lattice 1 (Figure PHA)

	Throughput, User Responses/sec					
"Connectedness" c	0	0.2	0.4	0.6	0.8	1.0
Throughput						
(m_2,m_3,m_4) = (1,1,1)	0.167	0.114	0.103	0.100	0.983	0.980
Throughput						
(m_2,m_3,m_4) = (6,5,4)	0.455	0.452	0.450	0.450	0.449	0.449

However multiple threads make a big difference. Figure PHA also shows results for the "rule-of-thumb" levels of threads (m_2, m_3, m_4) = (6,5,4), and the same levels of connectedness. There is still a penalty compared to Figure PC, but it is small. We can effectively carry the one-tower analysis into the Lattice situation. Also, threads give robustness, and make the throughputs insensitive to the connectedness.

Real lattices are only roughly symmetrical. Regional or divisional elements of an organization are not of equal size, and they have specialized needs. We have discovered that a single tower is a good predictor for symmetrical lattice behaviour provided there are enough threads. Is this still true for unsymmetrical cases? To what extent can the towers be analyzed separately? Under what conditions is there a single dominant tower, or bottleneck server? We will consider:

- providing a higher capacity to the users attached to one tower,
- one server in one tower which makes an increased execution demand,

• the location of a new bottleneck when one is alleviated.

6.4.1. One High-Capacity Tower in a Lattice

more users, and more demanding in request counts. Increased speed of servers by same factor (say, times five).

vary c, look at how it bogs down and also slows the others, as c increases. For a given c, how much to add to the others?

is there a critical c which is equivalent to “ignorable connectedness?”

software mod: change the slow systems to proactively update the fast one (feed the speed) (design around dominant data paths) (like prefetching -- optimistic?) then c = 0 on high-cap tower, and there are extra calls on the others. Some negative impact on others.

6.4.2. One Server with Heavy Execution

Single bottleneck

delay spreads up and sideways as its D is increased.

effect depends on c

What is the threshold for a bottleneck to appear?

6.4.3. Bottleneck Migration

two heavy servers, fix the biggest one, how sharply does the bneck move and what are the effects on users? Prediction, beginning from an operating state or model.

6.5. The Funnel Pattern

A Funnel may arise as a Lattice in which the number of partitions in a level decreases with depth. Instead of planning the system as similar nearly autonomous systems that are cross-connected, each level is considered separately, with a number of partitions appropriate for its functions and workload. Another origin for this pattern may have independent applications at each level, In client-server systems we often see many different applications available to the users, backed up by smaller numbers of specialized information-access servers, which finally access corporate databases at the bottom level.

Whatever the origin, here we shall consider layered servers with a concentration of work onto fewer servers at the lower levels. The base case in Figure PKD has three levels below the Users. There are four groups of Users at level 1, three servers at level 2, two at level 3 and one at level 4. Each server and User has a separate processor. Initially we shall consider a fully symmetrical system with the parameters:

- Users at level 1: n_1 = 4 groups, N_1 = 25 Users per group, host demand 1 sec to a private processor, a “thinking time” of Z_1 = 5 sec, y_1=1 request to each level-2 server.
- Servers at level 2: n_2 = 3 partitions, single threaded, host demand s_2 = 0.1 sec all in phase 1, y_2 = 1 request to each level-3 server.

- Levels 3 and 4 the same as level 2 except $n_3 = 2$ and $n_4 = 1$ (and level 4 being the bottom makes no requests).

Thus there are 3 requests on average, and 0.3 sec of work to be done at each level, to satisfy one User request. We may wonder if the “funneling in” is appropriate, or is a problem. Perhaps equal partitioning would be better.

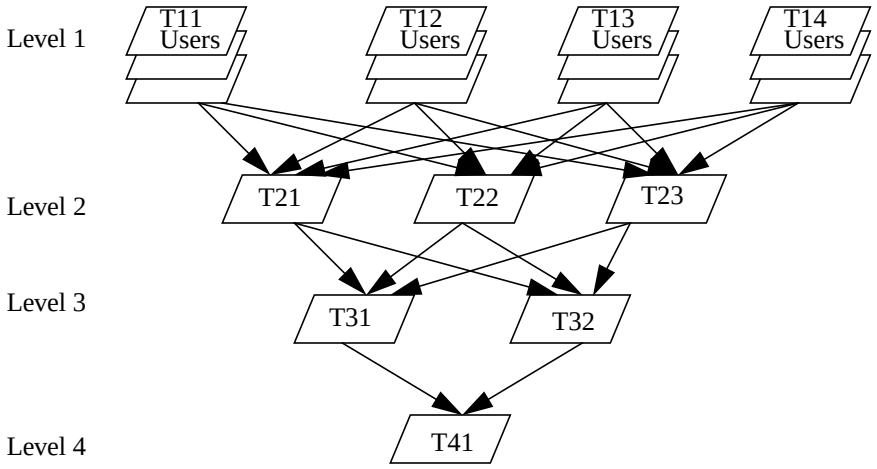


Figure 6.7. Funnel Pattern (Figure PKD)

Some performance evaluations will address these questions:

- as before, what is the impact of multithreading at the different levels? The value of m_l = threads at levels $l = 2$ and 3 will be varied. The result we will find is that, as before, the impact is considerable, and more threads are useful at higher level. However the rule of thumb (which suggests $m_3 = 2$, $m_2 = 5$) overstates the requirements at the higher levels.
- is the smaller number of partitions at the lower levels appropriate, or would more partitions have a useful impact? The values of n_3 and n_4 will be increased. In the results it is not clear that the improvement is sufficient to justify the additional processors.
- how important is the balance of work between levels, and between partitions? Work will be concentrated in one server at each level, and in each level in turn. The results show remarkable insensitivity to the balance of work between the partitions in one level; concentrating work at the bottom level is bad, but at the other levels it has little effect.

Base Case and Multithreading

Here are results for the total User throughput *f* for different threading levels (m_2, m_3) at levels 2 and 3. Response times seen by the user can be calculated as $R = 100/f - 6$ sec (cycle time minus the local execution and the thinking time).

Results table for base case with (m_2, m_3) = (1,1), (1,2), (3,2), (5,2), (10,10), (∞,∞)

Discussion

Width of Lower Levels

Here are results for the total User throughput *f* for different numbers of servers (*n3, n4*) at levels 3 and 4. The service times and total number of requests to the next lower level have been kept the same, but the requests are divided equally among the servers in the lower level. Thus the load at levels 3 and 4 is shared more widely if *n* is increased.

Table of results for (*n3, n4*) = (2,1), (2,2), (3,2), 3,3)

Discussion

Horizontal Balance of Load

Keeping the funnel shape with (*n2, n3, n4*) = (3,2,1), these experiments considered unequal host service demands in the partitions in each level. They modified the host service demands at levels 2 and 3 so the total host demand to be satisfied at each level was still 0.3 sec, but the ratio r_l of the largest demand to the others increased.

Table of results for (r_2, r_3) = (1, 1), (10, 1), (1,10), (10,10)

Discussion.

Vertical Balance of Load

Again keeping the funnel shape and the request frequencies, these experiments considered redistributing the total host demand across the levels by changing the host service times (s_2, s_3, s_4), which are initially (0.1, 0.1, 0.1). There are still 3 requests on average to each level, to satisfy a User request.

Table of results with (s_2, s_3, s_4) =(0.1, 0.1, 0.1), (0.2, 0.025, 0.025), (0.29, 0.005, 0.005), (0.025, 0.2, 0.025), (0.005, 0.29, 0.005), (0.025, 0.025, 0.2), (0.005, 0.005, 0.29)

Discussion

6.5.1. Comments on Funnels and Related Patterns

Unbalanced partitioning, and unbalanced server levels, are a rich optimization problem which cannot be completely characterized in a short space. In specific cases it can be addressed incrementally through software bottleneck analysis.

ref to “shape” measurements paper.

6.6. Peer-to-Peer Pattern

Up to this point all the software considered has been hierarchical with requests descending to servers. What happens in a system with no hierarchy, in which equal peer processes communicate? Such systems are important because of their robustness to failure and their symmetry. Examples arise in distributed databases, and distributed systems to manage facilities or services:

- in an air traffic control system each major airport is a node, and makes requests to neighbouring nodes for state updates, or to hand off aircraft to the next controller. Many of these interactions may be blocking, to ensure correct reception.
- in a distributed factory management system each production center may be a node with its own state, interacting with others to coordinate movement of goods through stages of processing and into the warehouse.

Some analysis is needed before a peer-to-peer system can be modelled with our MSS framework. It is necessary to understand the exact interaction behaviour. For instance, it would be a poor design that used single threaded tasks, and symmetrical blocking interactions as in Figure PLA. When a task makes a blocking request to a peer task, and waits for a reply, a situation is set up which might cause mutual request deadlock, with both tasks waiting for replies and ignoring their request queues. The first step in preventing this is to have multiple threads so a new thread can pickup an incoming request while another thread is blocked waiting for the peer. Even this can deadlock if all threads are waiting (although this is unlikely). A better design (and we suppose that most systems are actually built this way) would recognize that the processing of a request from the peer is different from one generated locally. For one thing it has been partly processed already. For another, the processing can usually now be satisfied at the one site. The Peer/Peer pattern builds these observations into the model, by dividing each task into parts, as shown in Figure PLB. One part Local (for local request handler) handles local requests generated by users, a second part Remote (for remote request handler) handles requests from the peer task(s), and a third part CSect handles critical sections shared by the first two. Local and Remote can be multi-threaded. The LRH and RRH tasks are distinguished to make the model clear; the actual software architecture may not have separate tasks for local and remote requests, but internally the functions associated with them should be recognizable.

The pattern is a special case of a Lattice. In Figure PLB, Task A is modelled by three pseudo tasks. ALocal handles all the requests from AUsers, by invoking AServices for requests which can be satisfied locally, and sending a request to BRemote for requests which can only be satisfied at B. If there are more sites the model expands easily by spreading requests out to them also.

There are many design options which affect performance. Commonly TaskA and TaskB will be multi threaded so they can respond to remote requests. There may be common code and a common thread pool for Local and Remote. The difference between Local and Remote requests may be only a flag in the request, which causes the path of further processing to follow the paths

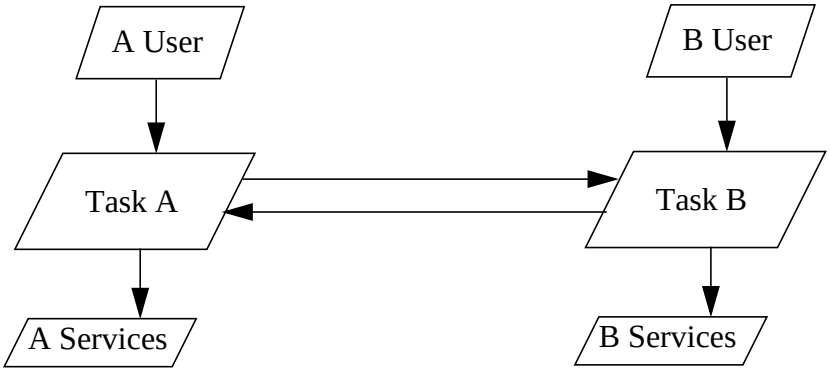


Figure 6.8. Equal, Symmetrical “Peer-to-Peer” Interaction. (Figure PLA)

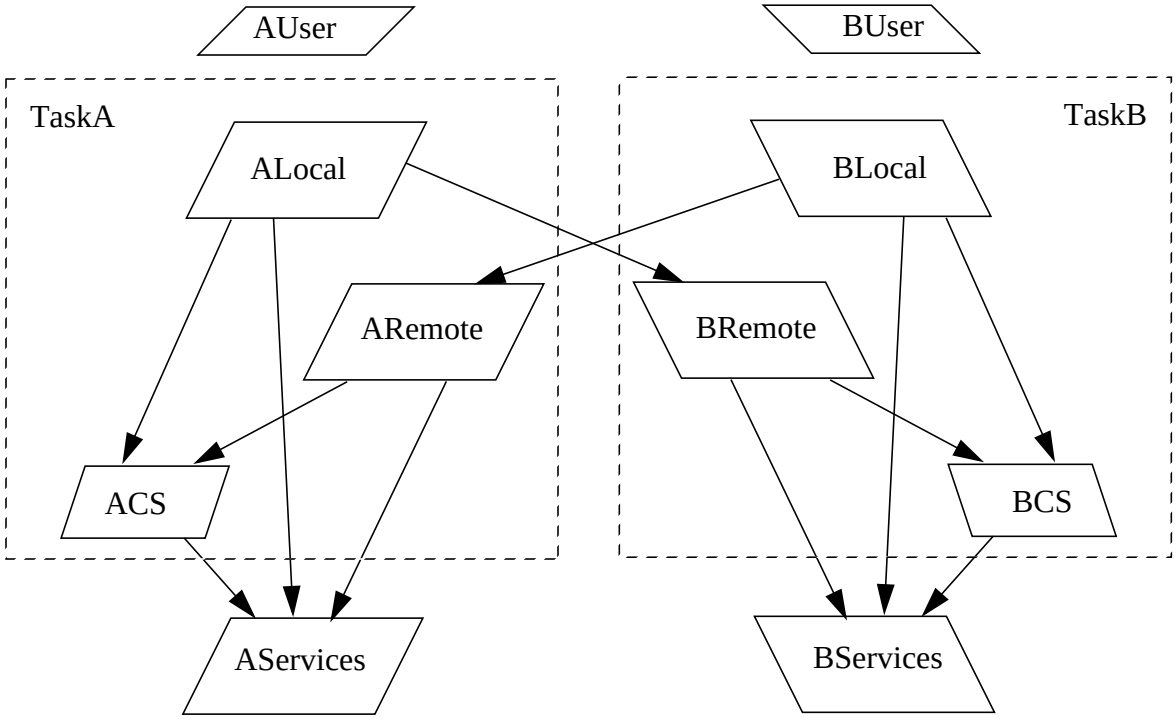


Figure 6.9. A Layered Set of Pseudo-Tasks Represent Task A and Task B. (Figure PLB)

for Local and Remote pseudo tasks in Figure PLB. An important detail may require a further addition to Figure PLB: the Figure translates to a Layered Queueing model with separate thread pools for the Local and Remote pseudo task. A common thread pool for TaskA can be modelled

by an additional pseudo task AThread which serves both ALocal and ARemote, as in Figure PLC. Separate “entries” on AThreads are used to keep the streams of requests separate.

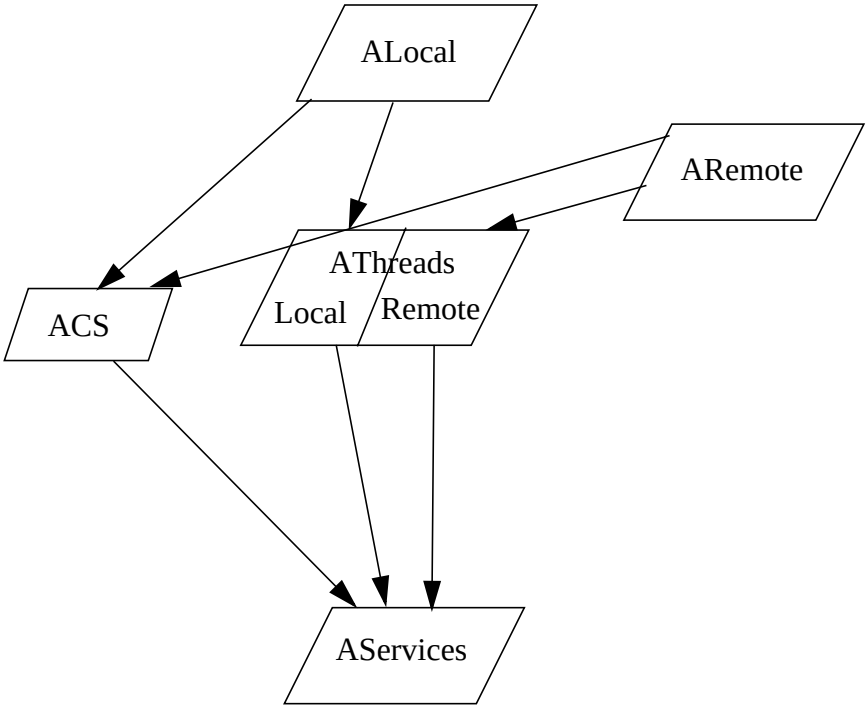


Figure 6.10. A Common Thread-Pool for ALocal and ARemote Modelled by a Pseudo-Task AThreads. (Figure PLC)

6.7. Synchronous Pipelines including Servers

The pipeline is one of the most obvious ways to introduce concurrency into a sequential computation. A pipeline works at the speed of its slowest stage, so to obtain maximum performance the workload of the stages must be suitably balanced. Many familiar pipelines such as UNIX pipes are asynchronous, with some kind of buffer between stages; if a buffer fills up it blocks the stage that is feeding it. If the storage is infinite we can model the pipeline by a sequence of tasks with asynchronous messages, as in Figure PMD(a), or with forwarding headed by a multiserver with multiplicity m = infinity (Figure PMD(b)). In practice space is never infinite and finite buffers often have important effects on performance. There is an extensive literature on “tandem queues with blocking”, which handles this case. We will model it within our MSS(Resources) framework for completeness. A special case of finite buffering is zero buffering, shown in figure PMD(c), which we shall analyze first.

In an unbuffered or synchronous pipeline each task sends its output to its successor and waits for a signal acknowledging receipt of the message; this is a rendezvous in which all the work is done in phase two. After phase two the task sends to the next stage and blocks; it is appropriate to sep-

arate this from the phase-2 “body” of the work, which must all be completed first, into a third phase. Blocking in phase three can be important, since each stage can only take a new input when it has successfully passed on its previous output to its successor. If the service time of each stage is deterministic then the pipe works simply at the rate of the slowest task, but we shall consider some degree of randomness in the execution time.

Table XA shows the throughput of the pipeline in Figure PMD(c) for random execution demands of exponential type (coefficient of variation = 1) and different degrees of imbalance between the workload of the stages. The optimum balance is tapered, lighter at the the beginning and heavier towards the end of the pipe, in a ratio of about 1.5 to 1. *******Other comments.

*****Table XA from pipeline paper in 1988. Bad balance and optimum taper. Exponential service*****

Unfortunately it is not only an imbalance of load that can bottleneck a synchronous pipeline, but also a large-variance service time tends to spread blocking back through the pipe. The effect of variance is considered in Table XB, alone and in combination with imbalance.

*****Table XB... new... variance and balance.*****

The natural response to variability in execution times is to add buffering to absorb some of the variations. In our MSS(Res) framework buffers can be represented as an additional synchronous multiserver stage which simply passes data on to the pipeline stage in its second phase, as shown in Figure PME. Because the previous stage has a rendezvous with the buffer “task” it blocks when no free buffer is available. Since the default queueing of the buffer task threads at the following worker stage is FIFO, this represents FIFO service to the buffers. Table XC shows throughput results for a four-stage pipeline with different numbers of buffers and different variability and balance.

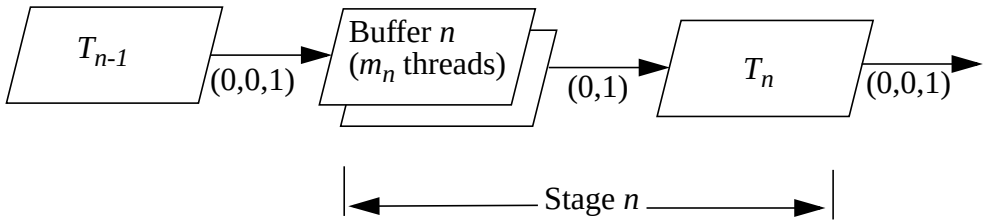


Figure 6.11. A Stage in a Buffered Pipeline with Blocking. (Fig. PME)

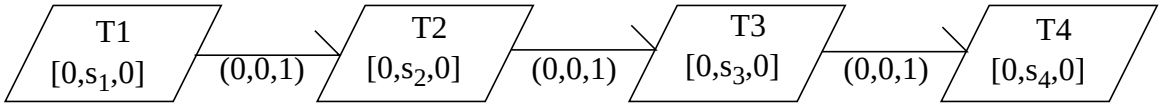
*****Table XC ... new... buffers, balance, variability*****

A feature of pipelines which has been little studied is the possibility that a stage uses a server outside the pipeline, and perhaps shares it with other pipeline stages. Blocking on the server might significantly slow down the overall system, and make it useful to multithread the pipeline stages.

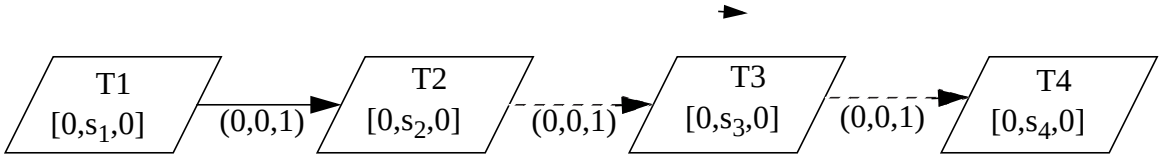
Since in our framework a processor is just another kind of server, this analysis can be applied to pipeline stages which share a processor. One might suppose that if two successive stages share a processor it is as if they are coalesced into a single stage, but what if they are in different parts of the pipe?

*****Example of both situations, and results....*****

More situations can be conveniently modelled as well. One example is window flow control over the entire pipeline, such that internally there are no particular buffer limits but the number of items being processed in total is limited to say mtot. This could be applied when there is a set of shared buffers, shared by the stages, which might be practical in a shared-memory multiprocessor or a virtual-shared-memory system. A conveinet model would use a multiserver with mtot servers as a gateway to limit the entry, and have it forward through the pipeline as in Figure PMD(b).

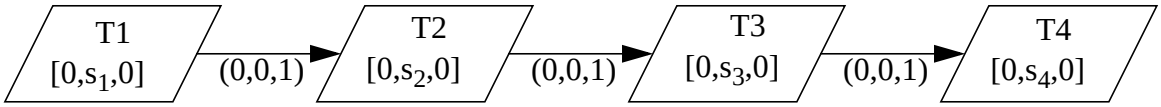


(a) Asynchronous pipeline

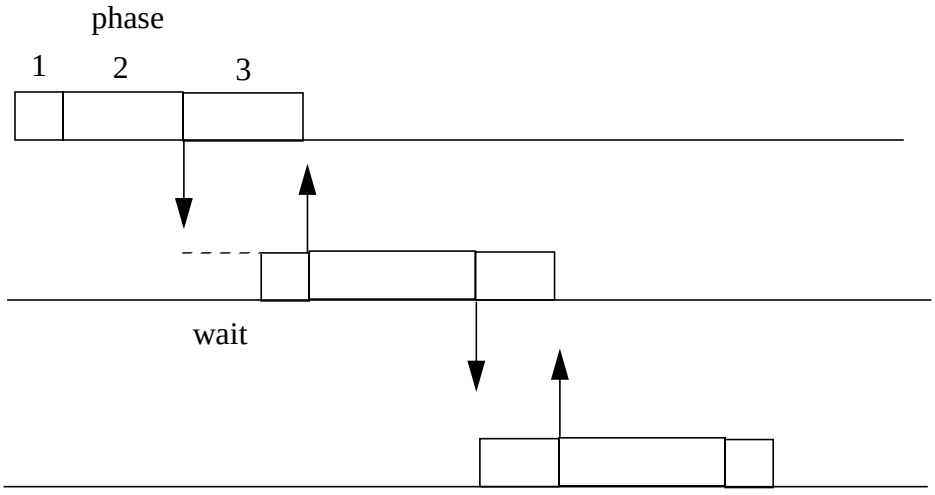


(*m* threads)

(b) Forwarded pipeline (*m* items max.)



(c) Synchronous pipeline



Interactions in a Synchronous Pipeline

Figure 6.12. Pipeline Patterns. (Figure PMD)

(analysis without servers)

with rv unbuffered and buffered, brief. balance.

CV, balance effect

overall window control alternative (shared buffers)

servers

6.8. Pipelines with Rendezvous (No buffering)

Our layered model applies directly to pipelines in which the next stage must accept a data token before the previous stage is free to do more work. To apply it we introduce a *third phase* for handing on the data token, so there are three phases as follows:

phase 1: accept a new data token and acknowledge it;

phase 2: operate on it (Processing);

phase 3: send the output token or tokens on, and wait for acknowledgment.

Performance - Oriented Patterns in Software Design (A multi-level service approach)

C. M. Woodside

Dept. of Systems and Computer Engineering

Carleton University, Ottawa K1S 5B6

(Draft version produced for classroom use, October 1996)

Chapter 7. Patterns with Parallel Paths (J for Join)

7.1 Servers with Internal Parallelism

Parallel execution is one of the main ways to obtain increased performance, although experience says it is easier to imagine than to achieve. In practice a parallel path can be set up in two slightly different ways:

- by sending an asynchronous message to another task, which then proceeds in parallel,
- by forking a distinct thread, which then invokes a service which executes in parallel.

The second approach can lead to a built-in join of the threads, which retains knowledge of the relationship between the sibling parallel paths, and is thus more powerful, although the same effect can be programmed with asynchronous messages (user-managed parallelism) This chapter models both approaches the same way, as if the program sets up distinct, heterogeneous threads which manage parallel paths.

The difference between these threads and the ones in a multithreaded server, is that these are heterogeneous, and they interact with each other (for instance through critical sections).

This chapter will examine four basic architectural patterns that exploit parallelism: Parallel service, deferred RPC communications in various situations, parallelism in pipelines, and parallelism in communications software.

7.2 Parallel Activities and Task Activity Graphs

A basic form of parallel service, seen for instance in parallel subtransactions, has a large job divided into parts and each part is farmed out to a separate server. *n* threads are forked, each thread sends a request to “its” server and blocks, and when all are unblocked the threads join and the large job is completed. Figure JDA shows this pattern in the simple form just described, with activities associated with an entry E of Server.

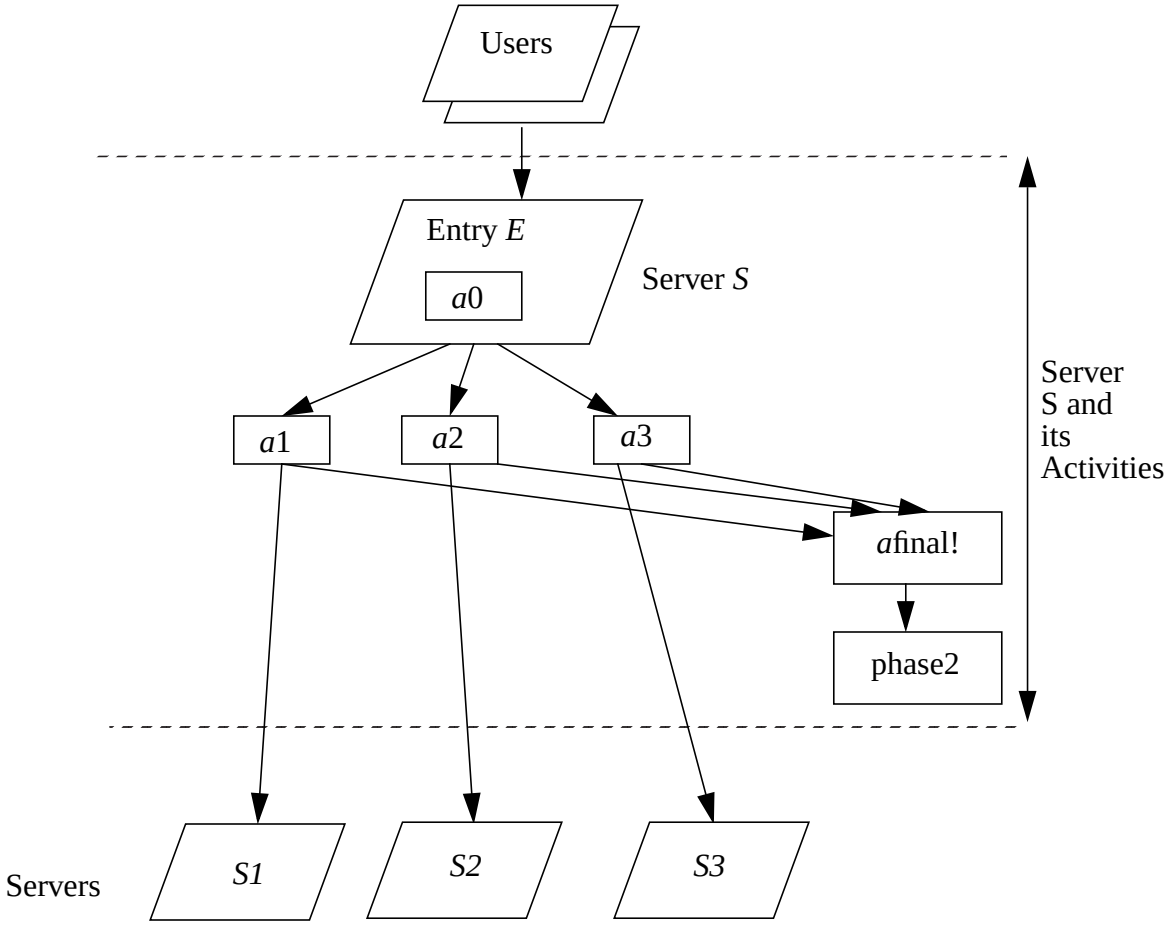


Figure 7.1. Parallel Service and Task Activity Graph. (Fig. JDA)

Parallelism is described in Figure JDA by activities and precedence. The first activity is triggered by the entry, and the others follow in precedence indicated by the arrows. The activity with a “!” after its name generates a reply from the entry to its client when it completes, and represents the end of phase 1. We will call this subgraph, attached to a task, a Task Activity Graph. The phased patterns of execution of an entry which we have been using can be modelled by Task Activity

Graphs as shown in Figure JDC(a) for two phases, and figure JDC(b) for three phases as used in pipelines. An activity in a Task Activity Graph has exactly the same demand parameters as a phase, and a phase is a kind of activity.

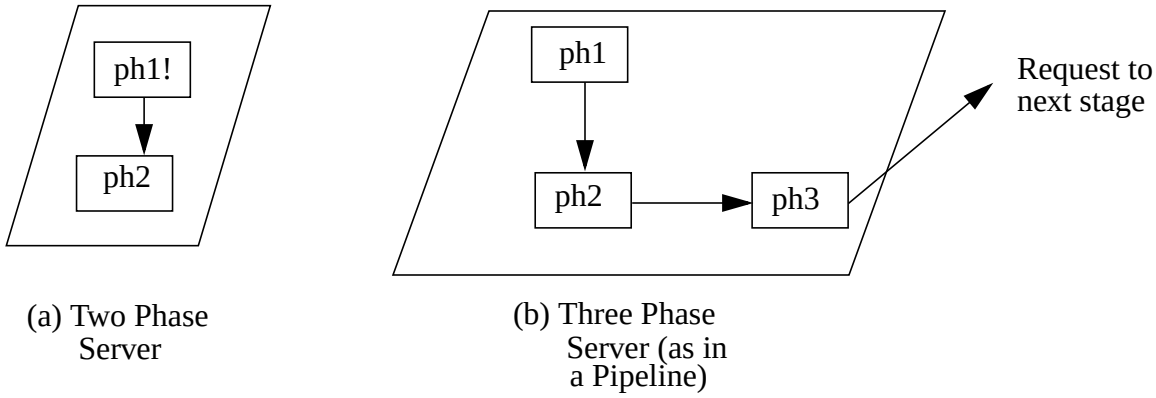


Figure 7.2. Task Activity Graph for Phases of Service. (Figure JDC)

The Task Activity Graph notation in Figure JDA is a reduced version of our earlier Activity Graphs for capturing execution sequences, with all the sequential, case-structured and looping detail reduced into single activities, and all the parallel structure retained.

In the earlier Activity Graph notation in Figure SB, a fork represents a point where two or more separate paths are begun. In a diagram, each parallel path has just one activity which represents another activity graph with the full details. When we reduce the activity graph of an entry to find its demands we reduce this nested activity graph (provided it does not in turn contain parallel paths) separately using reduction R1. In effect, when there is parallelism we do not reduce the activity graph for the entry all the way, but we stop at the level of activities in parallel. At a join, the continuation must wait for all the subpaths to complete before it can continue. We will assume that forks do not have to join later (they can terminate separately), but joins must derive from previous forks. We will also assume at least for now that a fork-join must be enclosed within a single phase, so for example the reply will not be issued by one of a set of parallel activities (this will be relaxed, as it might give a performance advantage to do so).

Referring again to the notation in Figure SB, there is also an asynchronous message send in an activity graph. This will be reduced to a demand parameter giving the number of similar messages generated by the enclosing reduced activity or phase. If the message is a reply to the current request however it is treated differently; it acts to separate phase 1 from phase 2. In a full Task Activity Graph this is represented by a “!” attached to the last activity in the phase, optionally with the entry name.

Finally, in Figure SB there is an asynchronous receive indicated, but we shall not deal with these at this point, unless they are messages that initiate a service. In this case the service begins with phase 2, because there is no reply to an asynchronous request.

7.3 Parallel Service

Figure JDA describes a simple case of a useful pattern for parallel subtransactions in a large data base, or for parallel operations in a scientific program (where the join is called a “barrier synchronization”. The performance is affected by possible contention between the servers which are activated in parallel, which can increase the individual path delays, and by the join delay (which in turn is increased by high variance in the path delays). First consider results for non-conflicting servers with different variability, and for different levels of conflict. Let

n = number of parallel servers

$s_{l,1}$ and $s_{l,2}$= service time in phase 1 and 2 for server l

cv_l = coefficient of variation of service times at server l

Then Table JF shows mean response time results over these parameters,

....

Discussion. Longer with larger cv, longer with more, phase 2 has no effect.

Now consider some of the same cases with a common server S, accessed y_1 times by each of the parallel servers in phase 1, and y_2 times in phase 2.

Longer with y, phase 2 does matter somewhat.

7.4 Asynchronous or Deferred RPCs

An asynchronous or deferred RPC is one in which the sender does not block at once for the reply, but blocks later. Thus the sender has an activity which occurs between sending the request and waiting for the reply. This pattern also describes prefetches, hints sent to a storage server, and any case where a request is sent before the result is needed. It corresponds in a way to phase 2 work, only at the sender. The pattern is easily described as in Figure JGA, with activity a being done before sending the request, and b, before blocking. The throughput results for 10 Users as s_b varies, for a fixed sum $s_a + s_b + s_c = 1$, are shown in Table JGC.

.....

improves with sb up to a point. Rule of Thumb related to second phase of server?

effectiveness in Tower, Lattice, with phase 2

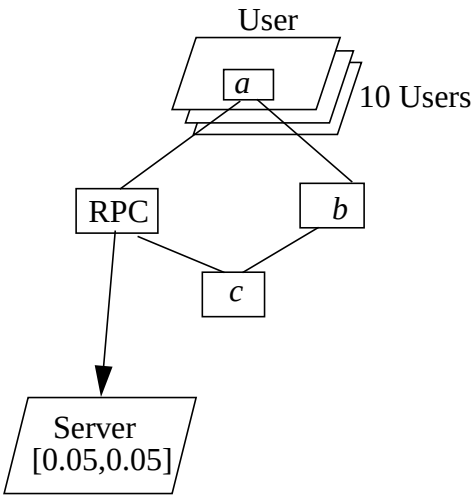


Figure 7.3. The Asynchronous RPC Pattern and Example. (Figure JGA)

7.5 Parallel Pipelines

basic idea

with servers

7.6 Parallelism in Communications

packets:similarity to pipelines; basic comm patterns for one packet, for many

packet parallel protocol stacks

group comm operations (nack approach = optimistic)

**********From here on, old text **********

Introduction

6.7. Parallel Servers

6.8 Asynchronous RPCs

basic pattern

effectiveness in Tower, Lattice, with phase 2

6.9 Parallel Pipelines

basic idea

with servers

6.10 Parallelism in Communications

packets: similarity to pipelines; basic comm patterns for one packet, for many

packet parallel protocol stacks

group comm operations (nack approach = optimistic)