A Pattern Language For Improving the Capacity of Layered Client/Server Systems with Multi-Threaded Servers

Dorina Petriu
Dept. of Systems and Computer Engineering
Carleton University, Ottawa, K1S 5B6, Canada
email: petriu@sce.carleton.ca

Gurudas Somadder
Dept. of Systems and Computer Engineering
Carleton University, Ottawa, K1S 5B6, Canada
email: gurudas@sce.carleton.ca

ABSTRACT

The paper describes a set of patterns that extend the pattern language proposed in [Meszaros96] for improving the capacity of reactive systems. The intent of these patterns is to identify some specific causes that limit the efficiency of a distributed layered client-server system with multi-threaded servers, and to find appropriate corrective measures. The type of systems considered here is a subclass of the larger category of reactive systems, and the new patterns are dealing with their specific performance characteristics. The effects of the patterns are illustrated with performance measurements conducted on a layered client-server system.

INTRODUCTION

Problem Domain

Many distributed applications are based on the client-sever paradigm and use various kinds of software servers (as for example, name servers, databases, network file servers, web servers, etc.) The performance of such systems depends strongly not only on the contention and queueing delays for hardware devices (such as processors, I/O devices, communication networks, etc.) but also on the contention for software servers. In order to satisfy the requests of its clients, a software server needs to access the services of one or more subservient servers, which may be either hardware or software. For example, a web server executing a client requests runs on its own processor and makes alternate requests to a network file server and a database server, each of which, in turn, needs the services of a processor and of one or more I/O devices (see Figure 1). Under high load, the system capacity may be limited either by one of its hardware resources or by one of its software servers. A typical (but not the only) class of such systems is known as a three-tier client/server architecture, used particularly for large business applications [Aarsten+96], [Hirschfeld96]. The system functionality is distributed into three tiers: front-end clients, middle application servers and back-end database server.

For performance analysis purposes, it is useful to represent a system with software servers as a *layered client/server model* (see Figure 1), in the form of a directed acyclic graph whose nodes represent clients and servers, and whose arcs denote service requests. The software entities are represented as parallelograms, and the

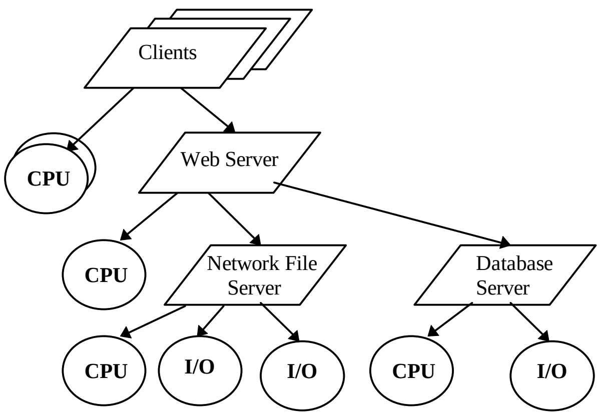


Figure 1. Example of a layered client/server system

hardware devices as circles. The nodes with outgoing and no incoming arcs are the clients, the intermediate nodes with both incoming and outgoing arcs are usually software servers, and the leaf nodes are hardware servers. It is worth to mention that a layered system does not imply a strict layering of servers (for example, servers in the same layer can call each other or can skip over layers).

The layered client/server model has been developed in previous work as a performance modelling approach that extends the well-known Queueing Network model in order to capture and solve analytically the performance characteristics of systems with software servers [Woodside+95], [Rolia+95], [Franks+96].

Set of Proposed Patterns

The layered client/server systems, as all reactive systems, have to meet performance requirements at a reasonable cost under high loads. The set of patterns proposed in the paper extend the pattern language for improving the capacity of reactive systems introduced in [Meszaros96] by considering cases where the limiting capacity factor is a software server. The new patterns are also complementary to another pattern language regarding the distribution of functionality in three-tier client/server systems presented in [Aarsten+96], [Hirschfeld96].

Figure 2 shows the patterns described in the paper and their relationships with some of the patterns from [Meszaros96] (shown in boxes) . The patterns in plain boxes are described in detail in the reference, those in dashed boxes only mentioned briefly. In order to limit the size of the diagram, only those patterns from [Meszaros96] which have a direct relationship with the newly proposed patterns are shown here.

Capacity Bottleneck pattern [Meszaros96] deals with identifying the limiting factor in a system's capacity, leading to a number of patterns related to the type of resource that is the actual limit:

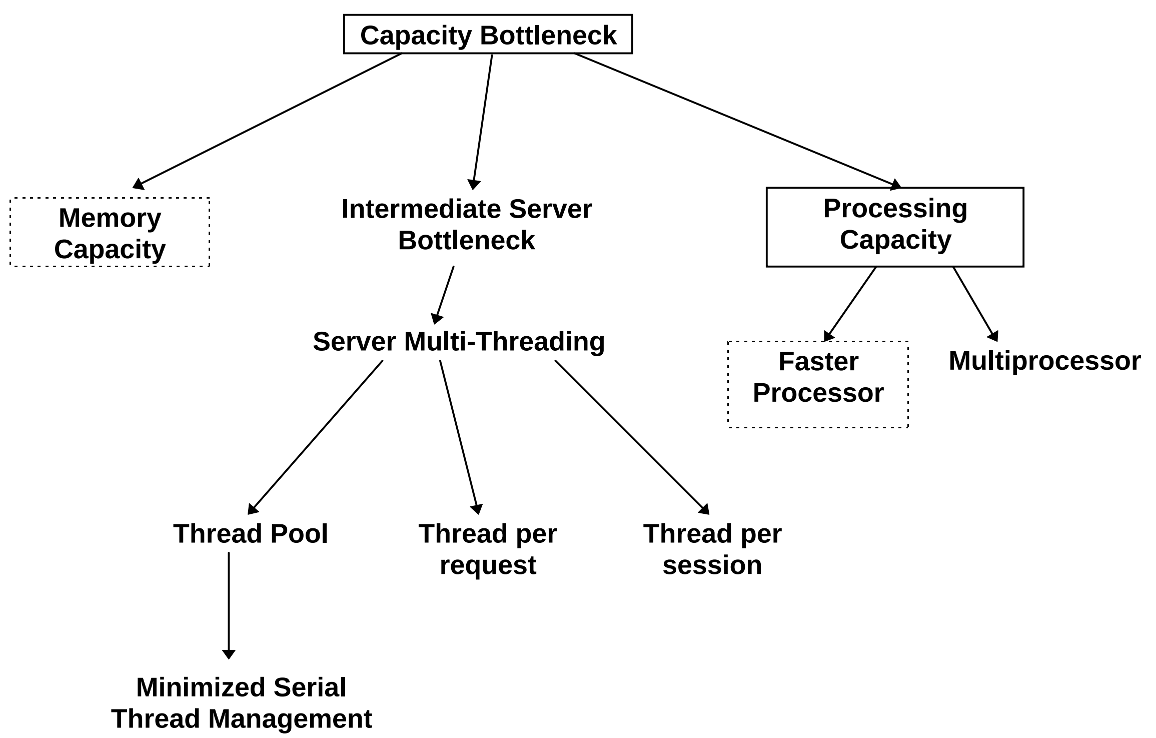


Fig. 2. The relationships between the proposed patterns and the pattern language for improving the capacity of reactive systems introduced in [Meszaros96]

Processing Capacity, Memory Capacity, Messaging Capacity. This paper shows that another type of limiting resource exists -- a software server, as described in the *Intermediate Server Bottleneck* pattern. The so-called software bottleneck phenomenon was described in previous performance analysis work, such as [Neilson+95] and [Franks+96], but it was never expressed in pattern form, nor was it related to existing pattern languages. *Server Multi-threading* pattern shows how to alleviate such a bottleneck by increasing the concurrency level of the server. It is specialized by the next three patterns, *Thread per request, Thread per session* and *Thread pool*, each describing a different solution applied in a different context. *Minimized Serial Thread Management* shows how to further increase the system efficiency when using the thread pool technique. *Multiprocessor* pattern points to a hardware solution for improving the system performance. The patterns are written according to the style and techniques presented in [Meszaros+96].

In this work we have not considered shedding-the-load strategies and the related patterns from [Meszaros96] which are very appropriate for "open" reactive systems, such as telephone switches, where the arrival rate of requests varies widely and may go, at times, beyond the capacity of the system. We have considered here "closed" systems with a limited number of clients which do not give up on their requests, wait patiently to be served and may send a new request only after the previous one has been completed. However, the layered client-server systems may have open arrivals, in which case shedding-the-load patterns will also apply. Moreover, the patterns introduced in this paper are not limited to closed layered server systems, but can be applied to any reactive systems with software servers.

INTERMEDIATE SERVER BOTTLENECK

Problem

A software server receiving requests from multiple clients may be the limiting resource in the system at high request rates, becoming the system bottleneck. Its request input queue builds up faster than any other queues, and the server is the first element to saturate under increasing load by reaching its maximum utilization. An intermediate server bottleneck prevents its subservient hardware resources from being used at their full capacity. How do we recognize when the system bottleneck is a software server and not a hardware one?

Context

We are using a system with one or more software servers, that can be represented as a layered client/server model. A software server may offer a range of service types to its clients, each one with different execution times and resource requirements. The *capacity* of a software server (i.e., the maximum rate at which the server is able to complete requests) can be determined by considering that the server is busy 100% of the time. The capacity depends on the following elements:

- the number of requests *processed* simultaneously by the server (not those waiting in the input queue)
- the average time for a request that is a weighted average of service times for different request types
- the service time for a given request type that is the sum of server's own execution time plus the nested services provided by the subservient servers.

Capacity = Nb_simult_requests / Average_service_time

*Average_service_time = Sum (Request_type_service * Percentage_of_this_request_type)*

Request_type_service = Own_excecution_time + Sum (Nested_service_times)

The own execution time in the last relation includes both the actual CPU time used by the server on the behalf of a given request type, and the waiting delay for the CPU. The nested service times also include actual service plus queueing. It can be seen from these relations that the capacity of a software server cannot be easily estimated with a simple back-of-the-envelope approach, because it depends not only on the server's requirements for resources, but also on the queueing delays in the whole system.

Forces

- We want a software system able to offer a high capacity by using the available hardware resources as efficiently as possible.
- The system bottleneck must be correctly identified, since any attempt to improve the overall performance has to deal with the bottleneck first.
- Increasing the capacity of a resource that is <u>not</u> the bottleneck will have little or no effect on the overall system performance.
- An intermediate server "feeds" work to its subservient servers, and may become the limiting factor in the system if it's unable to keep them busy enough. In other words, an intermediate server is the bottleneck if it becomes saturated while its subservient servers are under-utilized.
- Work propagates top-down, whereas utilization propagates bottom-up. A saturated server tends to saturate its users. The primary limiting factor in such a case is the lower level server.

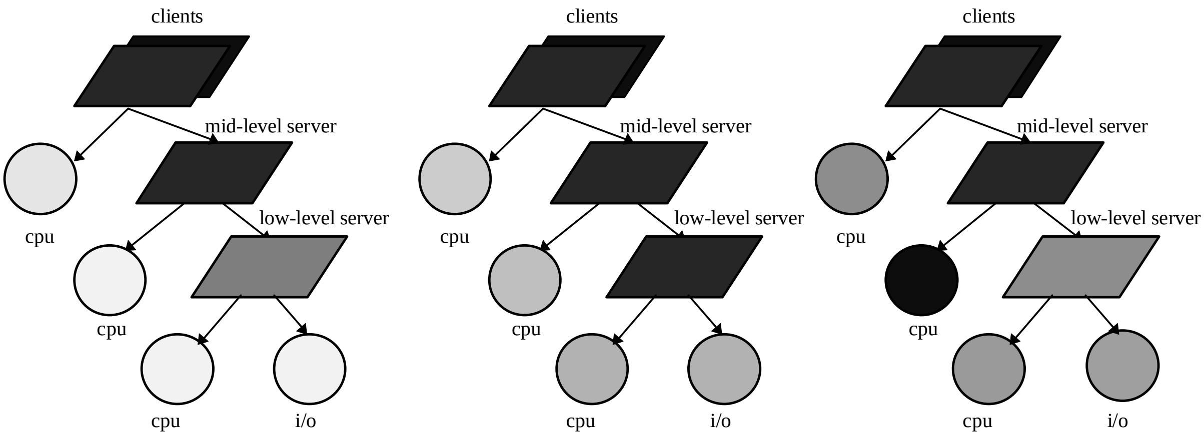
Solution

Understand which element is limiting the capacity of the system by analyzing the utilization of different servers at high load. The limiting element may be either a leaf or an intermediate node.

If all the leaf nodes in the graph are under-utilized while one or more intermediate nodes are highly utilized, then we have an *intermediate server bottleneck*. The bottleneck is a saturated software server whose subservient servers are *all* under-utilized. A larger difference in utilization means a stronger bottleneck. Figures 3.a and 3.b illustrate two cases of software bottleneck in a simple layered client/server system with a mid-level and a low-level software server. Different degrees of shading represent different utilization levels, the darkest corresponding to components that are busy all the time. Note that a bottleneck located at a higher-level software server will "feed" less work to the lower level servers, enforcing stronger limitations on the system capacity. For example, the case shown in Figure 3.a is more inefficient than the one in Figure 3.b. In order to alleviate a software bottleneck, apply the rest of the patterns presented in the paper.

If a leaf node has a very high utilization, then it represents a *hardware bottleneck*, which should be treated with one of the capacity bottleneck patterns presented in [Meszaros96]. In general, a system with a hardware bottleneck is more efficient than a similar one with a software bottleneck, since the hardware resources are better utilized.

Note that a highly utilized software server with at least one saturated subservient server is not the bottleneck -- the subservient server is. A special case is that of a software server using a single subservient server (as for example, a CPU-bound software server running on its own CPU). Both servers will have the same utilization; if both are saturated, the lower server is the bottleneck.



a) Software Bottleneck: mid-level server **b) Software Bottleneck: low-level server** **c) Hardware Bottleneck: CPU of mid-level server**

Fig. 3 Example of software and hardware bottleneck (degrees of shading represent utilization levels, the darkest one corresponding to saturation)

Example

Consider the layered system from Fig.3 implemented and measured as described in the appendix. The mid-level server is single-threaded and can process only one request at a time. The measurements for the utilization of different servers and the system throughput are shown in Fig.4. (The utilization represents the percentage of time a server is busy serving requests, including the waiting for nested services). The system reaches saturation at about 11 clients, beyond which the throughput holds steady because the system has reached its maximum capacity. The limiting factor can be identified from the utilization of different servers at high load: the mid-level server is 100% busy, while its CPUs and the low-level server are terribly under-utilized at below 20% (the utilization curves for the CPU and the low-level server are overlapped in the figure). The mid-level server is the bottleneck.

Related Patterns

Intermediate Server Bottleneck is the entry point to a number of patterns described below, whose intent is to increase the capacity of software servers and to alleviate cases of software bottleneck, improving therefore the overall system performance.

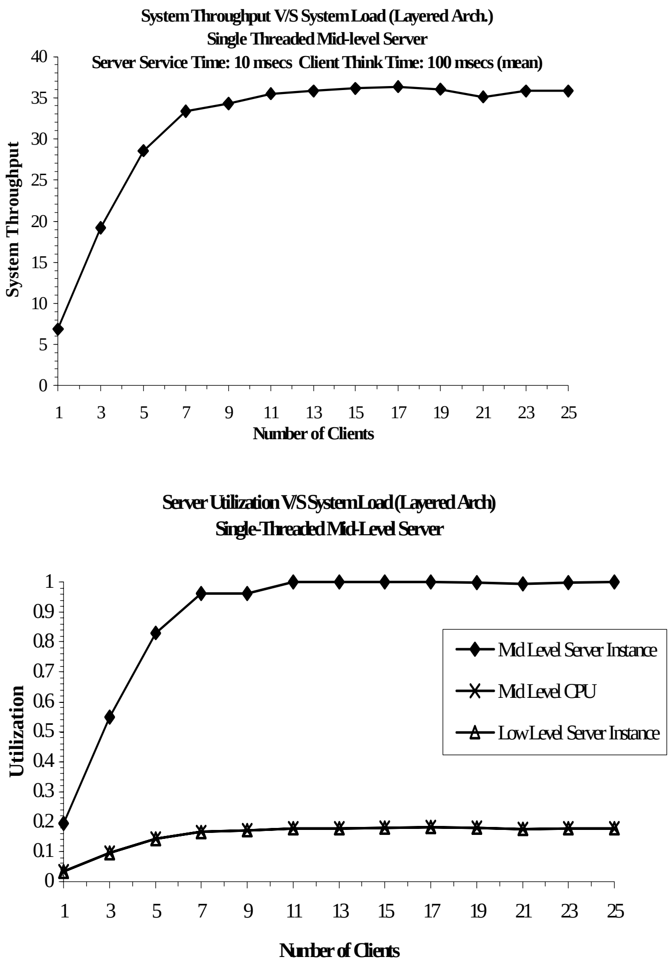


Fig. 4 Example of intermediate server bottleneck

SERVER MULTI-THREADING

Problem

The capacity of a layered system is limited by one of its intermediate servers, which prevents the hardware resources from being used efficiently. How can we relieve this type of bottleneck?

Context

Apply this pattern when a software server with *more than one* subservient server cannot "feed" enough work to its subservient servers to keep them busy. Such a case occurs, for example, when a server was programmed in a "sequential" style, using blocking I/O and communication primitives. Its serial behaviour prevents the subservient servers from working in parallel on different requests.
If an intermediate server requires *only* a single CPU (without any I/O or other subservient server) then there is nothing to gain by multi-threading, because there is no potential for actual parallel execution of different requests.

Forces

- Using concurrent servers that process requests in parallel leads to more efficient systems.
- Designing and debugging a reactive concurrent server that uses a single thread of control to process more than one request at the same time is rather difficult, especially if the services are

complex. Such a server may never block, so it cannot use blocking or stream I/O, which are more user-friendly than non-blocking I/O. Concurrent events must be demultiplexed and dispatched to non-blocking event handlers, the context of requests to subservient servers must be saved, etc.

- Using multiple threads allows several requests to be processed in parallel, while each thread will process a request at a time, being programmed in a sequential style. A thread can block waiting for I/O, network communication or services from other servers, and can easily store the context of the current request.
- If the parallel threads need to use shared resources such as data objects, additional complexity is introduced in the form of critical section problems (and require mechanisms such as mutex and read/write locks.)

Solution

Increase the concurrency level of a software server by multi-threading**.** There are several approaches to build a multi-threaded server which can be applied in different situations (as discussed in the next three patterns). Some solutions provide better performance in certain conditions, by paying a higher price in term of system resources. These solutions have been discussed in detail in [Smith+96].

Multi-threading is efficient only if the given software server uses more than one subservient server, and thus the concurrent requests can be processed in parallel.

The efficiency benefits brought by each additional thread tend to diminish when the number of threads increases. This effect is due to the fact that more threads lead to a higher competition for the subservient servers, and thus to longer queueing delays and nested service times.

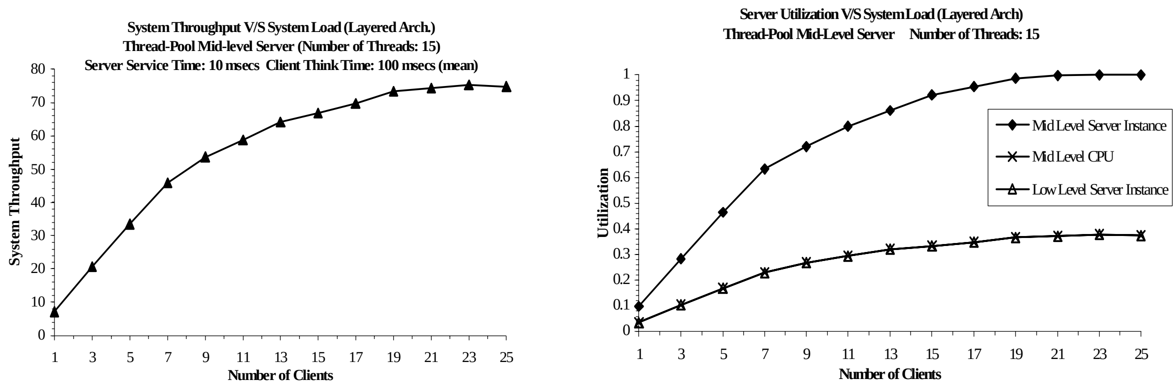


Fig. 5 Alleviate Intermediate Server Bottleneck by Multi-threading (15 threads)

Examples

A first example (see Fig.5) shows how the software bottleneck from Fig.4 is alleviated when the mid-level server has 15 threads instead of one. (The thread pool approach discussed in one of the following patterns has been applied in this example). The system throughput is more than double, system saturation is reached for a higher number of clients, and the utilization of the two subservient servers has also doubled (the two utilization curves are still overlapped). The system bottleneck was alleviated but not completely removed, since the mid-level server still saturates first.

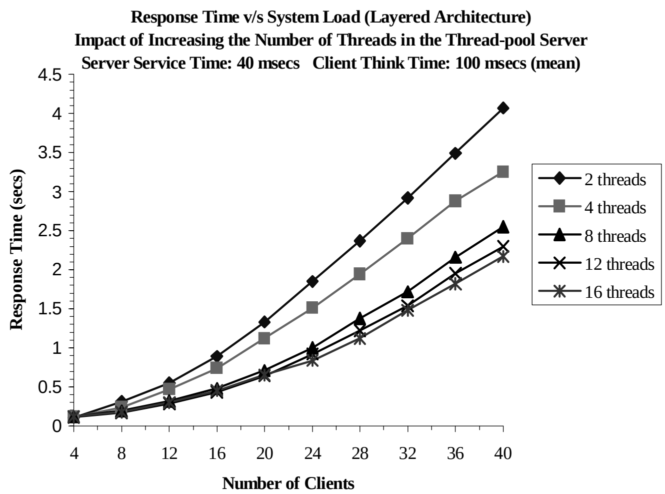


Fig. 6 Diminished returns with the increase in the number of threads

A second example given in Fig.6 shows that a gradual increase in the number of threads in the mid-level server brings diminished returns in terms of efficiency. Practitioners should use this knowledge when choosing the number of threads, such that a better efficiency is obtained without consuming too many system resources.

Related Patterns

The next three patterns, *Thread Per Request*, *Thread Per Session* and *Thread Pool*, specialize the current pattern, by describing three different solutions to multi-threading a server, each applicable in specific circumstances. *Thread Per Request* should be used when the requests are long and complex. *Thread Per Session* is appropriate for cases when clients make frequent requests for short services. *Thread Pool* is especially useful when a bound must be put on the consumption of system resources by the threads.

THREAD PER REQUEST

Problem

We want to apply the *Server Multi-threading* pattern in order to increase the capacity of a software server.

Context

The server must handle complex requests coming from multiple clients with variable frequencies. Each request takes a relatively long time to process.

Forces

- We would like to apply a simple programming solution, and avoid a lot of thread management. Although each thread will take care of a single request (resembling a sequential program) the access to common data shared by all threads requires synchronization primitives, which are an added complexity, can cause contention, overheads and can overwhelm the performance benefits of the concurrent execution (as shown in [McKenney96]).
- Increasing the number of threads leads to undue consumption of system resources (memory, file descriptors, etc.) and to execution overheads (thread creation and management, critical sections to protect shared data, etc.)
- Allocate resources (threads, memory, I/O ports, etc.) only when necessary, and release them as soon as possible.
- Session related data must be available to all requests related to that session.

Solution

The server spawns off a thread for each request when it arrives, and destroys the thread when the request is completed. The programming effort to spawn off the threads is quite limited, and the system resources are only held for the period the request is being served. However, storing session related data over several requests requires special measures (i.e., server data objects that out-live the threads).

Resulting Context

The creational overhead for a thread is incurred for every request, which makes the technique useful only for relatively long services. *Thread Per Session* and *Thread Pool* try to overcome this drawback in different ways. (Note that in new Unix systems, which are optimized for lightweight thread creation and where performance is expressed in process-forks-per-ms, this may not be nearly as much of an issue!). Also, the resource consumption may be too high for a large number of requests.
If subsequent requests need any session-related data, we need to ensure that this data is accessible regardless of the thread used to process a particular request. *Thread Per Session* solves this problem.

THREAD PER SESSION

Problem
We want to apply the *Server Multi-threading* pattern in order to increase the capacity of a software server.

Context
The clients carry on long-duration sessions with the server, sending multiple requests per session.

Forces
- We would like to maintain program simplicity and low thread management.
- Amortize the overhead of thread creation/destruction over the length of a session.
- Session related data must be available to all requests related to that session.
- A large number of threads leads to undue consumption of system resources.

Solution
The server spawns off a thread for each session started by a client, which is exclusively associated with it for the entire period of the session. This amortizes the cost of spawning a thread across multiple requests. Also, the problem of session-related data is easily solved: such data is local to the thread used exclusively by every session.
This approach is the most expensive in terms of resource consumption (especially for large numbers of on-going sessions), but given sufficient resources can achieve the highest throughput. Resources can be held without being used if some sessions submit infrequent requests.

Resulting Context
Spawning a thread for each requests will lead to excessive resource consumption when a high number of sessions are opened simultaneously. We know from the *Server-Multithreading* pattern that it is not worthwhile to increase the number of threads beyond a certain limit: the efficiency gains become smaller and smaller, whereas the resource consumption grows proportionally with the number of threads. This is a “no win” situation which must be avoided if possible. The problem is addressed by the *Thread Pool* pattern.

THREAD POOL

Problem
We want to apply the *Server Multi-threading* pattern in order to increase the capacity of a software server.

Context
There are too many simultaneous requests and/or sessions, and we must put a bound on system resources consumption for multi-threading.

Forces
- Enforcing bounds on resource consumption complicates the thread management.
- Amortize the overhead of thread creation/destruction over a longer period of time.
- Request may arrive when the server is unable to start the execution of new requests due to lack of resources. Measures should be taken to store the requests for a later time.
- Session related data must be available to all requests related to that session.

Solution
The server pre-spawns a pool of threads, whose number may be fixed, or changed dynamically at a low rate. The resource consumption for threads is bounded. The requests arriving when all the threads

are busy must be queued for later processing, which is an added complexity and overhead. This approach requires the most programming effort, due to thread pool management.

Resulting Context

Thread pool is really a cost-reduced version of the previous two approaches, *Thread Per Request* and *Thread Per Session*, addressing some of the consequences described in their respective Resulting Contexts (namely, bounding the consumption of system resources and amortizing the high cost of thread creation). However, a new overhead is introduced in this approach due to the allocation of a worker thread to every request. More exactly, the following activities are done serially in the main thread for all the incoming requests: event demultiplexing, receiving the messages from the clients, queueing the messages internally and dispatching the worker threads. The next pattern addresses the problem of minimizing the serial thread management.

MINIMIZED SERIAL THREAD MANAGEMENT

Context

The thread pool approach is one of the most attractive solution to multi-threading for systems with many clients. However, the thread pool management operations use shared data objects and are done serially in the main thread. Most of these activities are short, except for the receiving of messages when the messages are large.

Problem

Can the thread pool management be further improved by moving some of the serial thread management work to the parallel worker threads?

Forces

- Parallelizing the workload enhances the potential for performance gains, especially when longer, autonomous activities can be moved to parallel threads.
- Any work done serially on behalf of each and every request limits those gains.
- Shared data objects (as those used for the thread pool) must be used in a critical section, to protect their consistency.
- We gain nothing in term of efficiency by moving to parallel threads activities which must be performed in a critical section.

Solution

Move the actual message reception for large message sizes from the main thread to the worker threads.

Example

Figure 7 shows the effect on the system throughput of moving the reception of client messages from the main thread to the worker threads in a software server using a thread pool with 10 threads. It is easy to see that this parallelization has a positive effect that becomes more important with the size of the messages.

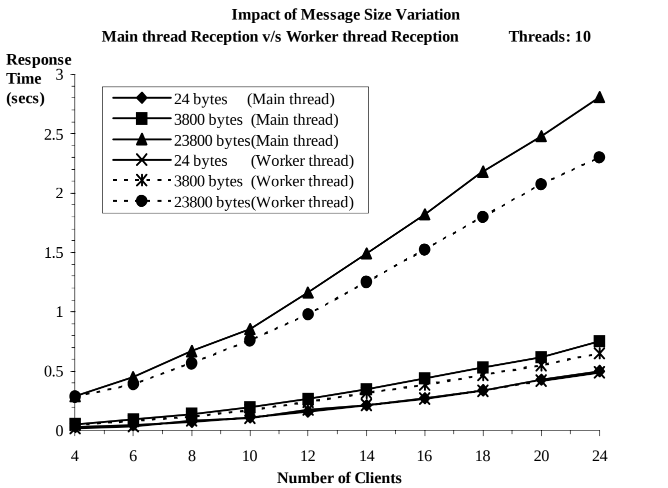


Fig. 7 Message reception in the Main Thread (serial reception) Vs. Worker thread (parallel reception)

MULTIPROCESSOR PATTERN

Context

After multi-threading the servers and moving all the significant work to parallel threads, the potential for concurrency in the server went up, and the workload was "pushed down" to the subservient servers. It is possible that the effect of such measures is to move the limiting capacity factor from the intermediate server to one of the subservient servers. This pattern applies if the new bottleneck is the processor on which the server is running.

Problem

How can we further improve the system capacity ?

Forces

- The efficiency of a multi-threaded server benefits from having more subservient servers (in this case, more CPUs).
- Additional hardware may be expensive.
- We would like to limit the programming effort (i.e., software costs) to change the application code to make it suitable for a multiprocessor.
- Multi-processing primitives often change the application interface.
- Some operating systems offer the same application interface if running on a single-processor or multiprocessor. (Such an example is Solaris 2.5).

Solution

Run the multi-threaded server on a multiprocessor instead of a single processor, on top of an operating system that does not require application interface changes for going from a single processor to a multiprocessor. Take advantage of executing the concurrent threads in parallel. The increase in capacity will be considerable, but still not linear with the number of processors (due to the serial portion of the workload and to contention for common data).

Related patterns

Another alternative to increasing the processing power is the *Faster Processor* pattern from [Meszaros96]. Another pattern from the same language, *Share the load*, also adds new processors to the system, but involves code changes since it selects the functions to be moved, clearly partitioned from the ones to stay.

The *Multiprocessor* pattern presented here takes advantage of the fact that a multi-threaded server is already parallelized, so it does not require any further changes to the application code.

ACKNOWLEDGEMENTS

Special thanks are due to Gerard Meszaros, who was the EuroPLoP'97 shepherd for this paper. Gerard has patiently guided our first attempts and helped us gain insights into the intricate approaches and techniques for describing a pattern language.

This work was supported in part by grants from the Telecommunication Research Institute of Ontario and the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[Aarsten+96] A.Aarsten, D. Brugali, G. Menga, "Patterns for Three-Tier Client/Sever Applications", Proc. of PLop'96.

[Fraks+96] G.Franks, S.Majumdar, J.Neilson, D.Petriu, J.Rolia, M.Woodside, ``Performance Analysis of Distributed Server Systems", Proc. of The 6-th International Conference on Software Quality, Ottawa, pp.15-26, October 1996.

[Hirschfeld96] R. Hirschfeld, "Three-Tier Distribution Architecture", Proc. of PLop'96.

[Mesazaros96] G.Meszaros, "A Pattern Language for Improving the Efficiency of Reactive Systems", In *Pattern Languages of Program Design, Vol.2*, (J.Vlissides, J.Coplien and N.Kerth eds.), pp.575-591, Reading, MA: Addison-Wesley, 1996.

[Mesazaros+96] G.Meszaros, J. Doble, "A Pattern Language for Pattern Writing", Proc. of PLoP'96.

[Neilson+95] J.E. Neilson, C.M.Woodside, D.C. Petriu, S.Majumdar, ``Software Bottlenecking in Client-Server Systems and Rendezvous Networks", *IEEE Trans. on Software Eng.*, Vol.21, No.9, pp.776-782, Sept. 1995.

[McKenney96], P.E.McKenney, "Selecting Locking Designs for Parallel Programs", ", In *Pattern Languages of Program Design, Vol.2*, (J.Vlissides, J.Coplien and N.Kerth eds.), pp.501-531, Reading, MA: Addison-Wesley, 1996.

[Rolia+95] J.A.Rolia, K.C.Sevcik, "The Method of Layers", *IEEE Trans. on Software Eng.*, Vol. 21, No. 8, pp. 689-700, August 1995.

[Somadder+97] G. Somadder, D.C. Petriu, "Performance Measurements of Multi-Threaded Servers in a Distributed Environment", (accepted) The Joint Int. Conference on Open Distrib. Processing and Distributed Platforms (ICODP'97), Toronto, Canada, May 27-30, 1997.

[Schmidt94] D.C.Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications", in Proc. of the 6th USENIX C++ Technical Conference, Cambridge Mass., April 1994.

[Schmidt+96] Schmidt, D.C. and Vinoski, S. "Object Interconnections: Comparing Alternative Programming Techniques for Multi-threaded Servers" (Column 5, 6 & 7), in *SIGS C++ Report Magazine,* Feb., Apr & July, 1996.

[Woodside+95] C.M. Woodside, J.E. Neilson, D.C. Petriu, S. Majumdar, ``The Stochastic RendezVous Network Model for Performance of Synchronous Client-Server-like Distributed Software," *IEEE Trans. on Computers*, Vol.44, No.1, pp. 20-34, January 1995.

APPENDIX

In order to illustrate the effect of the proposed patterns, we have implemented and measured a layered server system with an architecture as in Figure 8. The system was designed and implemented by using the Adaptive Communication Environment (ACE) toolkit [Schmidt94]. Several version of software servers with different threading models were implemented and measured [Somadder+97].

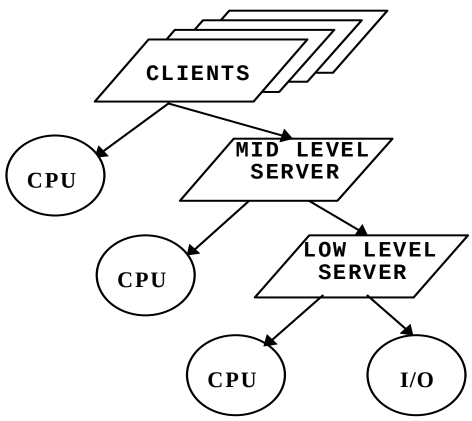


Fig. 8 Layered Client/ Server System

The measurements were performed for different workload intensities by varying the number of clients. Each point on the performance graphs was obtained by taking the average results of 10 similar experiments, each experiment having a duration of 300 request cycles of a tagged client. The repetition of the measurements was necessary to account for performance variations due to transient loads on the communication network and hosts. 95% of the results were in a confidence interval of plus/minus 2% around the mean. Even with these precautions, some curves look noisy due to external loads out of our control.