Design Techniques for Building Fast Servers

Hector M. Briceño
MIT Laboratory for Computer Science
hbriceno@lcs.mit.edu

1 Introduction

Servers are becoming crucial in today's work environments. Increasing demands have driven the need for high-performance servers. Many of these servers run on top of general-purpose operating systems (OSs). This approach allows for running many services at once over a generalized hardware abstraction. These abstractions generally come at the price of much lower performance than if the servers have access to low level hardware resources. In order to achieve high performance, some users have shifted to using a totally specialized and integrated operating system designed for a single server configuration. Since this approach does not multiplex resources across server applications and traffic is bursty, this can lead to underutilization of resources and to a big price tag.

We plan to take advantage of a current shift in OS design towards giving applications greater control over resource management (e.g. the exokernel project [1]), to build faster servers which both take full advantage of the hardware resources and allow multiple servers to be run on a single machine. We do this using three main techniques:

- Specialization to shorten the common path and to reduce complexity by ruling out unnecessary cases.
- Module Integration (e.g. file systems and network) to increase performance when information available on one module can be used to hint or to act upon another module.
- Event-driven organization to better match the request-response nature of clients' queries.

Currently we are investigating the value of these techniques by building a Network File System (NFS) server and a web server. [2]

2 Network File System (NFS) Server

NFS clients use Remote Procedure Calls (RPCs) to perform file system primitives like read, write, create, and lookup on files stored by NFS servers. RPCs work by encoding arguments in a portable format, sending a request and waiting for a reply which is also encoded in a portable format; it retransmits on timeout. We use a network stack that is specialized for this task, which blends IP, UDP and RPC into just one layer. In NFS, lookup and attribute operations make up 50% of all requests. We specialize the file system to be able to quickly process these requests by placing related file information close together. Additionally, we integrate part of the file system into the network module, so that information is available to the network stack can be used to hint future accesses to the file system. Finally, to simplify programming and improve performance we use a single event loop that catches network and file system events. We discuss the main techniques in more detail:

Specialization. Specialization allows us to make interfaces to the hardware match their exact usage. In our NFS server, we specialize the network stack for RPC calls, reducing the overhead to construct the RPC headers and allowing us to differentiate network packets very early. This blending of multiple layers is commonly known in the literature as Integrated Layer Processing (ILP). Specialization is also used in the file system design, which is targeted for NFS usage. Attribute fetching and name lookup operations comprise on average nearly 50% of all NFS requests. These operations need to retrieve file attributes depending on the name of a file in a directory. On current systems, directories contain for a given name, a pointer to this information. To retrieve this information one must perform two I/O operations, one to read read the pointer and another to get the actual information. In our NFS server, we can avoid the extra operation if we include the file attribute inside the directory itself. Furthermore, this structure can be in a format that requires little or no processing to send back to the client.

Module Integration. Our NFS server integrates the network stack with the file system. For example, the first fragment of a multi-fragment NFS write request contains all information necessary to compute the disk blocks that need to be written. We can use this information to hint the file system on the blocks we are going to write. This could allow the file system to place the disk head over the track we are going to use. On a 8 Kilobyte request over a 10Mbits/s Ethernet network, it will take at least 6.5 ms between the arrival of the first fragment and the last fragment of the request. This approach would allow us to overlap the network travel time with the seek time, greatly reducing the time from when the last fragment arrives to when the write request completes.

Event-driven Organization. Traditionally, to process multiple requests concurrently, many "daemons" or servers are run for a given service. This is necessary to allow multiple outstanding requests, because most file system operations are synchronous and do not have an asynchronous interface. This has the problem of synchronizing access to shared state and degraded performance due to context switching. We can achieve

Hector M. Briceño is advised by Frans Kaashoek and is supported in part by ARPA contract N00014-94-1-0985, and by an NSF National Young Investigator Award to Prof. Kaashoek. Address: Room 521c, 545 Technology Square, Cambridge MA 02139. Web: http://www.pdos.lcs.mit.edu/

same level of concurrency by using a single event-loop and asynchronous I/O calls. Whenever a procedure issues a disk operation, it resumes back to the event-loop and, at the completion of the operation we continue the procedure where we left off. This event-loop allows many RPC operations to occur concurrently without the need to block on the disk. This has two advantages over the daemon approach: there is no need to do context switching between daemons and no need to synchronize access to shared data. There are two event types of interest: network events and disk events.

Network events are generated by incoming packets. On these events, we dispatch the packet to the appropriate RPC procedure handler. If the RPC procedure has to access the disk, it issues a request and returns to the event loop; otherwise, it immediately sends the results back to the client.

Disk events result from the completion of disk tasks that can operate without CPU intervention. On these events, we look at which procedure was responsible for initiating the event, and resume the procedure. Using the information returned in the disk request, the procedure computes the result and sends it back to the client. It is possible that the procedure might have to issue another disk request, in which case it does so and returns to the event loop.

Additionally, we can reduce the response latency of these events by using *Application-specific Safe Handlers* (ASHs) [1]. ASHs are procedures that are downloaded into the kernel and executed immediately upon the occurrence of an event regardless of which process is scheduled. For example, we can use them to quickly respond to disk requests that require little processing like NFS write procedures. Such requests only need an acknowledgement and the new file attributes to be sent back. We set up an ASH to send back to the client the new file attributes on the completion of the disk write. We can also use an ASH to check the transaction cache on incoming RPC requests. If we have previously answered such a request, we take the answer from the cache and respond immediately from the ASH, otherwise we propagate the event to the event-loop.

3 Web Server

We also have a design and partial implementation of a Web Server that uses these techniques of specialization, integration and event-driven organization to improve its performance. Preliminary measurements indicate performance increases of more than an order of magnitude over NCSA/1.4.2 running on Ultrix for small web pages.

Specialization. We specialize two tasks: connection setup and file allocation. A great portion of the latency of a HTTP query stems from connection setup time. We can specialize our TCP stack by precomputing most information that will be necessary at connect time. Additionally, when a client fetches a HTML file, the server knows that the client will most likely fetch any other files that are included this HTML file. Therefore by placing the HTML file and the files that are included in it near each other on the disk, we can reduce the fetch time of future requests.

TCP uses a checksum to detect data corruption. The computation of this checksum for given data is always the same. We take advantage of this fact to pre-compute and save the checksum of the files that reside in our web server. By storing the checksum with the data, we avoid having to compute it on every transmission of the file. This would be hard to do in current systems, since they do not give the user access to such low levels of the TCP stack. This is an improvement, since most of these files don't change often and are read many times.

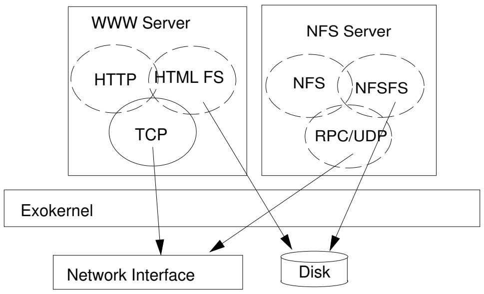

Figure 1. Multiple specialized servers time-sharing a single hardware platform using an exokernel.

Module Integration. Since TCP is a reliable protocol, it retransmits data when it does not receive an acknowledgement from the client. To do this, most TCP implementations keep data in retransmission buffers until an acknowledgement is received from the client. This can consume a lot of resources. By integrating the TCP stack with the disk buffer cache, we can eliminate the need for TCP retransmission buffers, since the data can be recreated cheaply or taken from the disk buffer cache. This allows us to consume less resources and have a larger disk buffer cache and avoid expensive data copy operations.

Event-driven Organization. Our web server uses an event-driven organization for the same reasons mentioned in the previous example with the addition of another type of event: script completion events. Web servers can run scripts to generate pages on the fly and add extra functionality. We can have a method by which data is passed back and forth between the server process and a script via Inter Process Communication (IPC). This can make script startup look instantaneous since the scripts can already be running. Via IPC, we pass in the arguments to the script, when the script returns, our event loop will detect an script completion event, and send back the results to the client.

References

[1] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-specif ic resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[2] M. Frans Kaashoek et al. Server operating systems. In *7th ACM SIGOPS European Workshop*, 1996.