High-performance network programming, Part 1: Squeeze maximum usage out of your network resources

Girish Venkatachalam

October 02, 2007
(First published September 25, 2007)

If you have UNIX®-based programming experience, then you've probably worried at some point about enhancing your network throughput. In this article, learn some useful techniques to squeeze the most out of your bandwidth, and get a big performance boost with some of the methods described here.

View more content in this series

Introduction

Anyone with UNIX® systems programming experience has probably worried about enhancing the network throughput and, in some cases, disk I/O. This article discusses advanced programming techniques employed by protocol implementers for squeezing the most out of your existing bandwidth. (This article is not about tuning your operating system (OS), configuring the kernel, or system tweaking.)

Though bandwidth available to a particular protocol is limited by Shannon's law and other factors, such as network usage patterns, most of the time it is shoddy programming or naive coding that causes suboptimal utilization of network resources.

Performance enhancement is also an art as much as it is a science. To get the best end-to-end throughput, you have to employ various tools to measure performance, identify bottlenecks, and eliminate them or minimize their impact. You can quickly get a huge performance boost by simple and straightforward scientific methods.

Pipelining and persistent connections

Pipelining is a well-known concept employed by CPUs for reducing the latency involved in the fetch-decode-execute cycle. Fetching each instruction involves a certain latency that can be avoided by prefetching instructions and storing them for later execution. The same concept applies to networking, as described in this section.
Instead of the server processing requests from clients in lockstep, the server can be told in advance what the client is interested in after processing the current request. The server maintains a queue of pending requests, executes them one after another instead of executing one, reads the next request from the wire, and so on. This enhances the response of interactive applications and reduces latency much more than any other technique.

But, it is not always possible to do this. Even when it works, sometimes the server queue runs out. Such cases are rare, and most of the time pipelining works very well. All the common protocols, including HTTP 1.1, NNTP, X11 and so on, employ pipelining in one form or other.

**Persistent Transmission Control Protocol (TCP) connections**

The above technique would obviously fail if you were to start a different TCP connection for every request or transaction. This is not the only reason you should reuse an existing connection. A TCP handshake while establishing and tearing down connections can be a huge overhead that is best avoided if possible.

Improper closure of TCP connections can quickly give you various kinds of headaches on UNIX systems. Another factor is the protocol overhead in new connections. You want to make sure that the network is used as much as possible for real data, instead of headers and other control information exchanges. **Figure 1** shows pipelining with ordinary processing.

**Figure 1. Pipelining and ordinary processing**

![Diagram showing pipelining and ordinary processing](image-url)
How does this translate to programming? Using existing TCP connections is easy, but implementing pipelining is not trivial. The protocol design has to take care of tracking pending requests and identifying which request the response corresponds to.

The next section explains other mechanisms that help in this endeavor.

**Non-blocking I/O, select(2), and poll(2)**

There are two programming models you should be familiar with: synchronous and asynchronous processing. The pipelining technique described above is an example of using asynchronous processing to enhance performance. Synchronous programming leads to simple design, simpler code, and poor performance at times, which is too bad. To get around the problem, you have to employ other tricks, such as non-blocking I/O.

Blocking and non-blocking sockets roughly correspond to synchronous and asynchronous processing, but not at the network level—it is at the OS level. In a typical `socket write(2)` or `send(2)` over a blocking socket, the user process waits for the system call to return. The kernel takes care of moving the process to sleeping state, waiting for the socket to be ready for writing, reading the TCP status code, and so on. Finally, control is returned to the application.

In the non-blocking case, the onus is on the programmer to ensure that the socket is writable, and to ensure that all the data gets written properly. This obviously leads to certain programming inconveniences and learning a new idiom but, once mastered, it is a powerful tool for getting good performance out of all networking code.

It is not enough to just use `read(2)` and `write(2)`, or `recv(2)` or `send(2)`, the moment your socket becomes non-blocking. You have to take the help of additional system calls, such as `poll(2)` or `select(2)`, for figuring out when you can write to the socket or read from the network.

One option is to use `poll(2)` for identifying writeability (because `select(2)` cannot do it), and use `select(2)` for identifying when data arrives on your socket on the other side. **Listing 1** shows an example of non-blocking I/O in detail.

**Listing 1. An example of non-blocking I/O**

```c
/******************************************
* Non blocking socket read with poll(2) *
*                                        *
******************************************/
void poll_wait(int fd, int events)
{
    int n;
    struct pollfd pollfds[1];
    memset((char *) &pollfds, 0, sizeof(pollfds));

    pollfds[0].fd = fd;
    pollfds[0].events = events;

    n = poll(pollfds, 1, -1);
    if (n < 0) {
        perror("poll()");
        errx(1, "Poll failed");
    }
```
size_t readall(int sock, char *buf, size_t n) {
    size_t pos = 0;
    ssize_t res;
    while (n > pos) {
        res = read (sock, buf + pos, n - pos);
        switch ((int)res) {
            case -1:
                if (errno == EINTR || errno == EAGAIN)
                    continue;
                return 0;
            case 0:
                errno = EPIPE;
                return pos;
            default:
                pos += (size_t)res;
        }
        return (pos);
    }
    return (pos);
}

size_t readmore(int sock, char *buf, size_t n) {
    fd_set rfds;
    int ret, bytes;

    poll_wait(sock, POLLERR | POLLIN);
    bytes = readall(sock, buf, n);

    if (0 == bytes) {
        perror("Connection closed");
        errx(1, "Readmore Connection closure");
        /* NOT REACHED */
    }
    return bytes;
}

/******************************************
* Non blocking socket write with poll(2) *
*                                        *
******************************************/

void poll_wait(int fd, int events) {
    int n;
    struct pollfd pollfds[1];
    memset((char *) &pollfds, 0, sizeof(pollfds));
    pollfds[0].fd = fd;
    pollfds[0].events = events;
    n = poll(pollfds, 1, -1);
    if (n < 0) {
        perror("poll()");
        errx(1, "Poll problem");
    }
}
size_t
written(int fd, char *buf, size_t n)
{
    size_t pos = 0;
    ssize_t res;
    while (n > pos) {
        poll_wait(fd, POLLOUT | POLLERR);
        res = write(fd, buf + pos, n - pos);
        switch ((int)res) {
            case -1:
                if (errno == EINTR || errno == EAGAIN)
                    continue;
                return 0;
            case 0:
                errno = EPIPE;
                return pos;
            default:
                pos += (size_t)res;
        }
    }
    return (pos);
}

Internet protocol (IP) fragmentation and other random network influences

Sendfile(2) is a technique that avoids buffer copy overhead, and it directly pushes bits from the file system onto the network. This system call unfortunately has portability problems across modern UNIX systems; however, and it is not even available on OpenBSD. For the sake of portability, avoid using this facility directly.

Sendfile(2) can reduce latency due to redundant memcpy(2). Like non-blocking I/O, this is a technique you can use to enhance performance at an OS level for networking code.

However, there are influences that occur deep down at a network level. You've probably heard of IP fragmentation. Mainly, it hurts performance. Fragmentation and reassembly are a costly affair and, though this is performed only by intermediate routers, it has severe impact on throughput.

There is a Path Maximum Transfer Unit (PMTU) discovery technique to protect you from needing to fragment an IP packet. Using this method, you can at least know (or guess, rather) what TCP segment sizes are likely to be passed through without getting fragmented at the IP level. The OS TCP layer fortunately takes care of splitting the protocol data into TCP segments that avoid IP fragmentation. With TCP being a byte stream without any message boundaries, this works fine and dandy. But watch out for User Datagram Protocol (UDP)—this no longer holds true there.

You also need to ensure your network is not abused by useless and harmful packets (isolate Windoze boxes to a separate virtual LAN). In the UNIX world, useful tools are tcpdump, iftop, and bandwidth monitors such as wmnet.
Summary

This article provided some ideas for getting the most of your bandwidth. Using a combination of different tools can enhance performance.

Stay tuned for the second part of this series, which will have more tricks for maximizing network utilization.

© Copyright IBM Corporation 2007
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)