Server Design Principles

Iterative versus Concurrent Servers

Iterative and Concurrent Servers

Two common designs for network servers using sockets are the following:

- *Iterative*: The server handles one client at a time, processing that client's request(s) completely, before proceeding to the next client.
- *Concurrent*: The server is designed to handle multiple clients simultaneously.

Iterative servers relevant when requests can be handled quickly.

Concurrent servers are suitable when significant time is required for processing a request.

A UDP Echo Server

```
#include <syslog.h>
#include "id_echo.h"
#include "become_daemon.h"

int
main(int argc, char *argv[])
{
    int sfd;
    ssize_t numRead;
    socklen_t addrlen, len;
    struct sockaddr_storage claddr;
    char buf[BUF_SIZE];
    char addrStr[IS_ADDR_STR_LEN];

    if (becomeDaemon(0) == -1)
        errExit("becomeDaemon");

    sfd = inetBind(SERVICE, SOCK_DGRAM, &addrlen);
    if (sfd == -1) {
        syslog(LOG_ERR, "Could not create server socket (%s)", strerror(errno));
        exit(EXIT_FAILURE);
    }

    /* Receive datagrams and return copies to senders */

    for (;;) {
        len = sizeof(struct sockaddr_storage);
        numRead = recvfrom(sfd, buf, BUF_SIZE, 0,
                           (struct sockaddr *) &claddr, &len);
        if (numRead == -1)
            errExit("recvfrom");

        if (sendto(sfd, buf, numRead, 0, (struct sockaddr *) &claddr, len)
                != numRead)
            syslog(LOG_WARNING, "Error echoing response to %s (%s)",
                    inetAddressStr((struct sockaddr *) &claddr, len,
                                   addrStr, IS_ADDR_STR_LEN),
                    strerror(errno));
    }
}
```

UDP Echo client

```
#include "id_echo.h"

int
main(int argc, char *argv[])
{
    int sfd, j;
    size_t len;
    ssize_t numRead;
    char buf[BUF_SIZE];

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s: host msg...\n", argv[0]);

    /* Construct server address from first command-line argument */

    sfd = inetConnect(argv[1], SERVICE, SOCK_DGRAM);
    if (sfd == -1)
        fatal("Could not connect to server socket");

    /* Send remaining command-line arguments to server as separate datagrams */

    for (j = 2; j < argc; j++) {
        len = strlen(argv[j]);
        if (write(sfd, argv[j], len) != len)
            fatal("partial/failed write");

        numRead = read(sfd, buf, BUF_SIZE);
        if (numRead == -1)
            errExit("read");

        printf("[%ld bytes] %.*s\n", (long) numRead, (int) numRead, buf);
    }

    exit(EXIT_SUCCESS);
}
```

TCP Echo Server

```
#include <signal.h>
#include <syslog.h>
#include <sys/wait.h>
#include "become_daemon.h"
#include "inet_sockets.h"       /* Declarations of inet*() socket functions */
#include "tlpi_hdr.h"

#define SERVICE "echo"          /* Name of TCP service */
#define BUF_SIZE 4096

static void             /* SIGCHLD handler to reap dead child processes */
grimReaper(int sig)
{
    int savedErrno;             /* Save 'errno' in case changed here */

    savedErrno = errno;
    while (waitpid(-1, NULL, WNOHANG) > 0)
        continue;
    errno = savedErrno;
}

/* Handle a client request: copy socket input back to socket */
```

```
static void
handleRequest(int cfd)
{
    char buf[BUF_SIZE];
    ssize_t numRead;

    while ((numRead = read(cfd, buf, BUF_SIZE)) > 0) {
        if (write(cfd, buf, numRead) != numRead) {
            syslog(LOG_ERR, "write() failed: %s", strerror(errno));
            exit(EXIT_FAILURE);
        }
    }

    if (numRead == -1) {
        syslog(LOG_ERR, "Error from read(): %s", strerror(errno));
        exit(EXIT_FAILURE);
    }
}

int
main(int argc, char *argv[])
{
    int lfd, cfd;               /* Listening and connected sockets */
    struct sigaction sa;

    if (becomeDaemon(0) == -1)
        errExit("becomeDaemon");
```

```
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    sa.sa_handler = grimReaper;
    if (sigaction(SIGCHLD, &sa, NULL) == -1) {
        syslog(LOG_ERR, "Error from sigaction(): %s", strerror(errno));
        exit(EXIT_FAILURE);
    }

    lfd = inetListen(SERVICE, 10, NULL);
    if (lfd == -1) {
        syslog(LOG_ERR, "Could not create server socket (%s)", strerror(errno));
        exit(EXIT_FAILURE);
    }

    for (;;) {
        cfd = accept(lfd, NULL, NULL);  /* Wait for connection */
        if (cfd == -1) {
            syslog(LOG_ERR, "Failure in accept(): %s", strerror(errno));
            exit(EXIT_FAILURE);
        }

        /* Handle each client request in a new child process */

        switch (fork()) {
        case -1:
            syslog(LOG_ERR, "Can't create child (%s)", strerror(errno));
            close(cfd);                 /* Give up on this client */
            break;                      /* May be temporary; try next client */

        case 0:                         /* Child */
            close(lfd);                 /* Unneeded copy of listening socket */
            handleRequest(cfd);
            _exit(EXIT_SUCCESS);

        default:                        /* Parent */
            close(cfd);                 /* Unneeded copy of connected socket */
            break;                      /* Loop to accept next connection */
        }
    }
}
```

Preforked Server Designs

- Instead of creating a new child process (or thread) for each client, the server precreates a fixed number of child processes (or threads) immediately on startup (i.e., before any client requests are even received). These children constitute a so-called *server pool*.
- Each child in the server pool handles one client at a time, but instead of terminating after handling the client, the child fetches the next client to be serviced and services it, and so on.

Some interesting problems.

Incoming client connection needs to routed to a process or thread.

All processes can wait on the accept call and only one process will be let go on the arrival of a connection.

With multiple threads, we could have the parent process doing the accept and handing over the connection to a thread.