What Are Client/Server Software Architectures?

Client/server computing is the logical extension of modular programming.

Modular programming has as its fundamental assumption that separation of a large piece of software into its constituent parts ("modules") creates the possibility for easier development and better maintainability.

Client/server computing takes this a step farther by recognizing that those modules need not all be executed within the same memory space.

With this architecture, the calling module becomes the *client* (that which requests a service), and the called module becomes the *server* (that which provides the service).

The logical extension of this is to have clients and servers running on the appropriate hardware and software platforms for their functions.

For example, database management system servers running on platforms specially designed and configured to perform queries, or file servers running on platforms with special elements for managing files.

ssh@vnet.ibm.com

For a long time it was widely-held *myth* that client/server computing had something to do with PCs or Unix machines. Cloud-computing, and mobile-computing, are contemporary examples demonstrating that the choice of hardware and operating system platforms has become quite irrelevant, and that *interoperability* through standards is all-important for success.

What Does A Client Process Do?

The client is a process that sends a message to a server process, requesting that the server perform a service.

Client programs usually manage the user-interface portion of the application, validate data entered by the user, dispatch requests to server programs, and sometimes execute business logic. The client-based process is the front-end of the application that the user sees and interacts with.

The client process often manages the local resources that the user interacts with such as the monitor, keyboard, and peripherals.

One of the key elements of a *client workstation* is the graphical user interface (GUI). Normally a part of operating system, i.e. the window manager detects user actions, manages the windows on the display, and displays the data in the windows.

What Does A Server Process Do?

Server programs generally receive requests from client programs, execute database retrieval and updates, manage data integrity and dispatch responses to client requests. The server-based process *may* run on another machine on the network. This server could be the host operating system or network file server, providing file system services and application services.

The server process often manages shared resources such as databases, printers, communication links, or high powered-processors. The server process performs the *back-end* tasks that are common to similar applications.

kalakota@uhura.cc.rochester.edu

What is a Two-Tier Architecture?

A two-tier architecture is one where a client talks directly to a server, with no intervening server. It is typically used in small environments (fewer than 50 simultaneous clients).

A common error in client/server development is to prototype an application in a small, two-tier environment, and then attempt to scale up by simply adding more client connections to the server.

This approach will usually result in an ineffective system, as the server becomes overwhelmed. To properly scale to hundreds or thousands of users, it is usually necessary to move to a three-tier architecture.

What is a Three-Tier Architecture?

A three-tier architecture introduces (another) server (or an *agent*) between the client(s) and the traditional server.

The role of the agent is manyfold. It can provide translation services (as in adapting a legacy application on a mainframe to a client/server environment), metering services (as in acting as a transaction monitor to limit the number of simultaneous requests to a given server), or intelligent agent services (as in mapping a request to a number of different servers, collating the results, and returning a single response to the client.

Lloyd.Taylor@jhuapl.edu

What is an 'Intranet'?

The explosion of the World Wide Web is due to the world-wide acceptance of a common transport (TCP/IP), server standard (HTTP), and markup language (HTML). Many corporations have discovered that these same technologies can be used for internal client/server applications with the same ease that they are used on the Internet.

Thus was born the concept of the *intranet* - the use of Internet technologies for implementing internal client/server applications.

One key advantage of Web-based intranets is that the problem of managing code on the client is greatly reduced. Assuming a standard browser on the desktop, all changes to user interface and functionality can be done by changing code on the HTTP server. Compare this with the cost of updating client code on 2,000 desktops.

A second advantage is that if the corporation is already using the Internet, no additional code needs to be licensed or installed on client desktops. To the user, the internal and external information servers appear integrated.

A rapidly-disappearing disadvantage is that there is limited ability to provide custom coding on the client. In the early days of the Web, there were limited ways of interacting with the client. The Web was essentially "read-only", with protocols such as *gopher* and *WAIS*. With the release of code tools such as Java and JavaScript, this limitation is no longer a major issue.

Lloyd.Taylor@jhuapl.edu

Characteristics Of Client/Server Architectures

1. A combination of a client or front-end portion that interacts with the user, and a server or back-end portion that interacts with the shared resource.

   The client process contains solution-specific logic and provides the interface between the user and the rest of the application system. The server process acts as a software engine that manages shared resources such as databases, printers, modems, or high powered processors.

2. The front-end task and back-end task have fundamentally different requirements for computing resources such as processor speeds, memory, disk speeds and capacities, and input/output devices.

3. The environment is typically heterogeneous and multivendor. The hardware platform and operating system of client and server are not usually the same. Client and server processes communicate through a well-defined set of standard application program interfaces (APIs), RPCs, and RMIs.

4. An important characteristic of client-server systems is scalability:

   - *Horizontal scaling* means adding or removing client workstations with only a slight performance impact.

   - *Vertical scaling* means migrating to a larger and faster server machine, or to multiservers.

kalakota@uhura.cc.rochester.edu

Partitioning Client/Server Responsibilities

In moving a single, monolithic application to a separated client/server configuration, we must address a number of issues:

1. Is there a *functional* partition at all?

   Are there separate responsibilities that *can* be performed by separate tasks?
   Should they be separated?

2. Is there a *data-driven* partition?

   Can different sections of the data be centralized, or split between multiple tasks?
   Can these multiple tasks execute on separate hardware, with separate address-spaces?
   Should distinct data partitions be replicated?

3. Is there an extensive use of global variables?

   Is there significant state information that controls the execution of the application?
   Is it possible, and desirable, to centralize this information anyway?

4. Are there any hidden intra-application communication mechanisms (such as variables, exceptions, or signals)?

   Is there unusual, possibly asynchronous, control flow in the application (e.g. the use of global goto's, or asynchronous signals)?

Concurrency (and hence speed) in Servers

The primary motivation for providing concurrency in servers is, of course, speed.

Concurrency is derived from using a 'non-queuing' model of execution, either by using a new (copy of the) server to support each client, or to provide faster, 'time-sliced' response to each client.

If no concurrency is available in the server, pending requests from new and existing clients are either blocked or refused (c.f. the `listen()` socket API call).

In general, clients leave their concurrency to the operating system, unless the application is sufficiently large, or time-critical, that it is the only process on a CPU and it performs its own internal scheduling.

Increased concurrency, and hence speed, is required (demanded) when:

- forming responses requires significant I/O.
- processing time is proportional to the type of request.
- application-specific, high-performance, hardware is available.

Definitions:

- *iterative servers* - single request at a time.
- *concurrent servers* - multiple 'simultaneous' requests.

Iterative Servers - managing a single connection

Firstly, consider a *iterative server* - each connection, or *session*, with a distinct client is handled until its completion.

Our server process will fully service each client until its completion, and continue to service subsequent clients until told to terminate.

Case 1 - makes subsequent clients wait:

```
//  Open a socket, listen, bind an address
    ....

    bool keep_going = true;

    while(keep_going) {
        int new_client   = accept(mysocket, ....);

        if(new_client == -1) {
            perror("accept");
            keep_going = false;
        }
        else {
     extern bool service(int sd);

            keep_going = service(new_client);

     shutdown(new_client, SHUT_RDWR);
            close(new_client);
        }
    }

    shutdown(mysocket, SHUT_RDWR);
    close(mysocket);
```

If the time to service each client is lengthy, or of indeterminable duration, new clients may be kept waiting for their initial connection - perhaps being terminated with an error of `ECONNREFUSED`.

Iterative Servers - managing one process per client

In this second model, a separate operating system process is spawned to handle each new client. There is thus no waiting required if any clients are slow or long-running, but (significant?) additional load is added to an operating system.

Case 2 - wll service subsequent clients quicker, but cannot scale indefinitely:

```
//  Open a socket, listen, bind an address
    ....
    bool keep_going = true;

    while(keep_going) {
        int new_client   = accept(mysocket, ....);

        if(new_client == -1) {
            perror("accept");
            keep_going = false;
        }
        else {
            switch ( fork() ) {
            case -1:
                keep_going = false;
                perror("fork");
                break;

            case  0: {
                extern bool service(int sd);

                close(mysocket);
                (void)service(new_client);

                shutdown(new_client, SHUT_RDWR);
                close(new_client);
                exit(EXIT_SUCCESS);
                break;
            }
            default:
                shutdown(new_client, SHUT_RDWR);
                close(new_client);
                break;
            }
        }
    }
    shutdown(mysocket, SHUT_RDWR);
    close(mysocket);
```

Note that while this example will work, that it is not complete. In particular, the parent process will need to (eventually) wait for the completion of its child processes, else many "zombie" processes will persist.

Concurrent Servers Using *select()*

With a concurrent server, a single server handles many clients within the same process. This obviates the need for interprocess communication between multiple servers (such as file locking).

We use a new network supporting system call, *select()*, to inform our process which descriptors are *ready for I/O*. The descriptors may be open to files, devices, sockets or pipes.

select deals with sets of descriptors (implemented as an array or bitmap in C or C++) and provides functions for their manipulation.

Here we examine descriptors open for *reading*, timing out each 10 seconds:

```
#include <sys/select.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

    bool  keep_going = true;
    ....
    while(keep_going) {
        fd_set readset;

        FD_CLR(&readset);
        FD_SET(sd, &readset);

        foreach other descriptor open for reading {
            FD_SET(desc, &readset);
        }

        struct timeval timeout;

        timeout.tv_sec  = 10;              // wait up to 10 seconds
        timeout.tv_usec =  0;

        if(select(MAXDESC, &readset, NULL, NULL, &timeout) >= 0) {
            if(FD_ISSET(sd, &readset)) {
                client = accept(sd, ...); // yet another new client
                ......
            }
            foreach other descriptor open for reading {
                if(FD_ISSET(desc, &readset)) {
                    service(desc);
                    .......
                }
            }
        }
    }
```

Note that this example is typical in employing only one set of file descriptors, *readset*. A service much more concerned about I/O speeds, particularly disk blocking, would employ another set of file descriptors, *writeset*, or perform asynchronous file I/O.

The Internet Supervisor Daemon - *inetd*

One problem with having many internetworking services supported, is that each operating system host requires many (idle) daemons just waiting for incoming connections on their reserved port - each consumes some memory and a process slot.

The common solution is a single 'super' daemon, listening for incoming connections on *many* ports. When a connection is made on, say, `telnet`'s port (=23) the 'true' `telnet` daemon is invoked to service the connection.

The `inetd` daemon listens (`accepts()`) on many ports simultaneously (using `select()`), and then either handles the connection itself or spawns a new process.

`inetd` reads its configuration information from `/etc/inetd.conf` :

```
telnet stream tcp nowait root   /usr/sbin/tcpd  in.telnetd
#
finger stream tcp nowait nobody /usr/sbin/tcpd  in.fingerd
#
ftp    stream tcp nowait root   /usr/sbin/tcpd  in.ftpd -l -a
#
time   stream tcp nowait root  internal
time   dgram  udp wait   root  internal
echo   stream tcp nowait root  internal
echo   dgram  udp wait   root  internal
```

A conceptually similar *multi-protocol server* implementation in Java may be found in Flanagan's excellent *Java Examples in a Nutshell*, Chapter 5.

All code examples from this book are also available.

Addressing Between Heterogeneous Machines

The Internet's addressing mechanisms define end-points of communication using 32-bit host addresses and 16-bit ports. To provide fixed length headers, addresses and ports are integers (not strings). The Internet protocols define a *network standard ordering* - 'fields are described left to right, with the most significant octet on the left and the least significant octet on the right.'

Big-Endian	Motorola 680x0, Sun SPARCs.
Little-Endian	Intel x86, DEC-Alpha, Apple Silicon M1/M2.
Bi-Endian	SGI MIPS, IBM/Motorola PowerPC.

Applications must convert incoming addresses and ports, arriving in *network order*, to their *host order*, and convert outgoing integers to *network order*:

```
#include  <stdint.h>

#if __BYTE_ORDER == __BIG_ENDIAN
#define ntohl(x)         (x)
#define ntohs(x)         (x)
 .....
#else
extern uint32_t ntohl(uint32_t __netlong);
extern uint16_t ntohs(uint16_t __netshort);
 .....
#endif

    uint32_t   ipaddr;
    uint16_t   tcp_port;

    ipaddr              = ntohl(connection.ipaddr);
    tcp_port            = ntohs(connection.port);

    connection.ipaddr   = htonl(ipaddr);
    connection.port     = htons(tcp_port);
```

Jonathan Swift's Gulliver's Travels, published in 1726, provided the earliest literary reference to computers, in which a machine would write books. This early attempt at artificial intelligence was characteristically marked by its inventor's call for public funding and the employment of student operators. Gulliver's diagram of the machine actually contained errors, these being either an attempt to protect his invention or the first computer hardware glitch.

The term endian is used because of an analogy with the story Gulliver's Travels, in which Swift imagined a never-ending fight between the kingdoms of the Big-Endians and the Little-Endians (whether you were Lilliputian or Brobdignagian), whose only difference is in where they crack open a hard-boiled egg.

Accessing Protocol and Service Information

There is a significant amount of 'common' knowledge about protocols, services, addressing and hosts available in *many* files.

Fortunately, most of this is accessible through support libraries, often provided as part of an Application Programming Interface (API).

Because, most of this information itself will be frequently requested by applications (and WWW applications here are *not* helping), the information is often provided by network servers.

Examples include mappings between:

- protocol names and protocol numbers - see *getprotoent()*.
- host names, host addresses and host aliases - see *gethostent()*.
- service names and service ports - see *getservent()*.

Because most of these functions cache their information, most return pointers to static information that must be copied by the application.

The growth of multi-threaded applications is driving the need for *re-entrant* versions of these functions, in which the caller provides an address into which the result is copied.

Typical API Library Routines

Access to lists of networking structures and services is generally provided by library routines.

For example, the traditional Unix API provides routines to discover hosts, networks, protocols, application services, and the necessary data conversion operations.

Consider the following code to iterate through (many) hostnames at UWA:

```
#include  <stdint.h>

int main(int argc, char *argv[])
{
    struct  netent *ne;
    struct hostent *he;

    uint32_t  hostorder, netorder;

    setnetent(1);            // open the network name database

    while(ne = getnetent()) {   // foreach network entry
 printf("NETWORK %s :\n", ne->n_name);
 hostorder = (ne->n_net << 8);

 sethostent(1);       // open the host name database

 for(int i=1 ; i<=254 ; i++) {
     netorder = htonl(hostorder);
     if(he = gethostbyaddr((char *)&netorder, sizeof(netorder), AF_INET))
                printf("%36s : %s\n", he->h_name, inet_ntoa(he->h_addr));
                ++hostorder;
            }
        }
        endhostent();        // close the host name database
    }
    endnetent();             // close the network name database
    return 0;
}
```