

CLIENT-SERVER COMPUTING

For articles on related subjects see DISTRIBUTED SYSTEMS; ELECTRONIC COMMERCE; OPERATING SYSTEMS; CONTEMPORARY ISSUES; and TCP/IP.

Introduction

Client-server computing is a distributed computing model in which *client* applications request services from *server* processes. Clients and servers typically run on different computers interconnected by a computer network. Any use of the Internet (*q.v.*), such as information retrieval (*q.v.*) from the World Wide Web (*q.v.*), is an example of client-server computing. However, the term is generally applied to systems in which an organization runs programs with multiple components distributed among computers in a network. The concept is frequently associated with *enterprise computing*, which makes the computing resources of an organization available to every part of its operation.

A *client* application is a process or program that sends messages to a server via the network. Those messages request the server to perform a specific task, such as looking up a customer record in a database or returning a portion of a file on the server's hard disk. The client manages local resources such as a display, keyboard, local disks, and other peripherals.

The *server* process or program listens for client requests that are transmitted via the network. Servers receive those requests and perform actions such as database queries and reading files. Server processes typically run on powerful PCs, workstations (*q.v.*), or mainframe (*q.v.*) computers.

An example of a client-server system is a banking application that allows a clerk to access account information on a central database server. All access is done via a PC client that provides a graphical user interface (GUI). An account number can be entered into the GUI along with how much money is to be withdrawn or deposited, respectively. The PC client validates the data provided by the clerk, transmits the data to the database server, and displays the results that are returned by the server.

The client-server model is an extension of the *object-based* (or *modular*) programming model, where large pieces of software are structured into smaller components that have well defined interfaces. This decentralized approach helps to make complex programs maintainable and extensible. Components interact by exchanging messages or by *Remote Procedure Calling* (RPC—see DISTRIBUTED SYSTEMS). The calling component becomes the client and the called component the server.

A client-server environment may use a variety of operating systems and hardware from multiple vendors; standard network protocols like TCP/IP provide compatibility. Vendor independence and freedom of choice are further advantages of the model. Inexpensive PC equipment can be interconnected with mainframe servers, for example.

Client-server systems can be scaled up in size more readily than centralized solutions since server functions can be distributed across more and more server computers as the number of clients increases. Server processes can thus run in parallel, each process serving its own set of clients. However, when there are multiple servers that update information, there must be some coordination mechanism to avoid inconsistencies.

The drawbacks of the client-server model are that security is more difficult to ensure in a distributed environment than it is in a centralized one, that the administration of distributed equipment can be much more expensive than the maintenance of a centralized system, that data distributed across servers needs to be kept consistent, and that the failure of one server can render a large client-server system unavailable. If a server fails, none of its clients can make further progress, unless the system is designed to be fault-tolerant (see FAULT-TOLERANT COMPUTING).

The computer network can also become a performance or reliability bottleneck: if the network fails, all servers become unreachable. If one client produces high network traffic then all clients may suffer from long response times.

Design Considerations

An important design consideration for large client-server systems is whether a client talks directly to the server, or whether an *intermediary process* is introduced between the client and the server. The former is a two-tier architecture, the latter is a three-tier architecture.

The *two-tier architecture* is easier to implement and is typically used in small environments (one or two servers with one or two dozens of clients). However, a two-tier architecture is less scalable than a three-tier architecture.

In the *three-tier architecture*, the intermediate process is used for decoupling clients and servers. The intermediary can *cache* frequently used server data to ensure better performance and scalability (see CACHE MEMORY). Performance can be further increased by having the intermediate process distribute client requests to several servers so that requests execute in parallel.

The intermediary can also act as a translation service by converting requests and replies from one format to another or as a security service that grants server access only to trusted clients.

Other important design considerations are:

- ◆ **Fat vs. thin client:** A client may implement anything from a simple data entry form to a complex business application. An important design consideration is how to partition application logic into client and server components. This has an impact on the scalability and maintainability of a client-server system. A “thin” client receives information in its final form from the server and does little or no data processing. A “fat” client does more processing, thereby lightening the load on the server.
- ◆ **Stateful vs. stateless:** Another design consideration is whether a server should be stateful or stateless. A *stateless* server retains no information about the data that clients are using. Client requests are fully self-contained and do not depend on the internal state of the server. The advantage of the stateless model is that it is easier to implement and that the failure of a server or client is easier to handle, as no state information about active clients is maintained. However, applications where clients need to acquire and release locks on the records stored at a database server usually require a *stateful* model, because locking information is maintained by the server for each individual client (see DATABASE CONCURRENCY CONTROL).
- ◆ **Authentication:** For security purposes servers must also address the problem of authentication (*q.v.*). In a networked environment, an unauthorized client may attempt to access sensitive data stored on a server. Authentication of clients is handled by using cryptographic techniques such as *public key encryption* (see CRYPTOGRAPHY, COMPUTERS IN) or special *authentication* (*q.v.*) servers such as in the OSF DCE system described below.

In public key encryption, the client application “signs” requests with its private cryptographic key (see DIGITAL SIGNATURE), and encrypts the data in the request with a secret session key known only to the server and to the client. On receipt of the request, the server validates the signature of the client and decrypts the request only if the client is authorized to access the server.

- ◆ **Fault tolerance:** Applications such as flight-reservation systems and real-time market data feeds must be fault-tolerant. This means that important services remain available in spite of the failure of part of the computer system on which the servers are running (high availability), and that no information

is lost or corrupted when a failure occurs (consistency). For the sake of high availability, critical servers can be replicated, which means they are provided redundantly on multiple computers. If one replica fails then the other replicas still remain accessible by the clients. To ensure consistent modification of database records stored on multiple servers, a transaction processing (TP—*q.v.*) monitor can be installed. TP monitors are intermediate processes that specialize in managing client requests across multiple servers. The TP monitor ensures that such requests happen in an “all-or-nothing” fashion and that all servers involved in such requests are left in a consistent state, in spite of failures.

Distributed Object Computing

Distributed object computing (DOC) is a generalization of the client-server model. Object-oriented modeling and programming are applied to the development of client-server systems. Objects are pieces of software that encapsulate an internal state and make it accessible through a well-defined *interface*. In DOC, the interface consists of object operations and attributes that are remotely accessible. Client applications may connect to a remote instance of the interface with the help of a naming service. Finally, the clients invoke the operations on the remote object. The remote object thus acts as a server.

This use of objects naturally accommodates heterogeneity and autonomy. It supports heterogeneity since requests sent to server objects depend only on their interfaces and not on their internals. It permits autonomy because object implementations can change transparently, provided they maintain their interfaces.

If complex client-server systems are to be assembled out of objects, then objects must be compatible. Client-server objects have to interact with each other even if they are written in different programming languages and run on different hardware and operating system platforms.

Standards are required for objects to interoperate in heterogeneous environments. One of the widely adopted vendor-independent DOC standards is the OMG (Object Management Group) CORBA (Common Object Request Broker Architecture) specification. CORBA consists of the following building blocks:

- ◆ **Interface Definition Language:** Object interfaces are described in a language called IDL (Interface Definition Language). IDL is a purely declarative language resembling C++. It provides the notion of interfaces (similar to classes), of interface inheritance, of operations with input and output arguments, and of data types (*q.v.*) that can be

passed along with an operation. IDL serves for declaring remotely accessible server objects in a platform- and programming language-neutral manner, but not for implementing those objects. CORBA objects are implemented in widely used languages such as C++, C, Java, and Smalltalk.

- ◆ **Object Request Broker:** The purpose of the ORB (Object Request Broker) is to find the server object for a client request, to prepare the object to receive the request, to transmit the request from the client to the server object, and to return output arguments back to the client application. The ORB mainly provides an object-oriented RPC facility.
- ◆ **Basic Object Adapter:** The BOA (Basic Object Adapter) is the primary interface used by a server object to gain access to ORB functions. The BOA exports operations to create object references, to register and activate server objects, and to authenticate requests. An object reference is a data structure that denotes a server object in a network. A server installs its reference in a *name server* so that a client application can retrieve the reference and invoke the server. The object reference provides the same interface as the server object that it represents. Details of the underlying communication infrastructure are hidden from the client.
- ◆ **Dynamic Invocation Interface:** The DII (Dynamic Invocation Interface) defines functions for creating request messages and for delivering them to server objects. The DII is a low-level equivalent of the communication stubs (message-passing interfaces) that are generated from an IDL declaration.
- ◆ **Internet Inter-ORB Protocol:** The Internet Inter-ORB Protocol (IIOP) allows CORBA ORBs from different vendors to interoperate via a TCP/IP connection. IIOP is a simplified RPC protocol used to invoke server objects via the Internet in a portable and efficient manner.
- ◆ **Interface and Implementation Repository:** The CORBA Interface Repository is a database containing type information (interface names, interface operations, and argument types) for the interfaces available in a CORBA system. This information is used for dynamic invocation via the DII, for revision control, and so forth. The Implementation Repository provides information allowing an ORB to locate and launch server objects.

Client-Server Toolkits

A wide range of software toolkits for building client-server software is available on the market today. Client-server toolkits are also referred to as *middle-*

ware. CORBA implementations are an example of well-known client-server middleware. Other examples are OSF DCE, DCOM, message-oriented middleware, and transaction processing monitors.

- ◆ **OSF DCE:** The Open Software Foundation (OSF) Distributed Computing Environment (DCE) is a *de facto* standard for multivendor client-server systems. DCE is a collection of tools and services that help programmers in developing heterogeneous client-server applications. DCE is a large and complex software package; it mainly includes a remote procedure call facility, a naming service, a clock synchronization service, a client-server security infrastructure, and a *threads* package (see MULTITASKING).
- ◆ **DCOM:** Distributed Component Object Model (DCOM) is Microsoft's object protocol that enables *ActiveX* components to communicate with each other across a computer network. An *ActiveX* component is a remote accessible object that has a well-defined interface and is self-contained. *ActiveX* components can be embedded into Web documents, so that they download to the client automatically to execute in the client's Web browser (see WORLD WIDE WEB). DCOM provides a remote instantiation facility allowing clients to create remote server objects. It also provides a security model to let programmers restrict who may create a server object and who may invoke it. Finally, an Interface Definition Language (IDL) is provided for defining remotely accessible object interfaces and composing remote procedure calls.
- ◆ **MOM:** Message-Oriented Middleware (MOM) allows the components of a client-server system to interoperate by exchanging general purpose messages. A client application communicates with a server by placing messages into a *message queue*. The client is relieved of the tasks involved in transmitting the messages to the server reliably. After the client has placed a message into a message queue, it continues other work until the MOM informs the client that the server's reply has arrived. This kind of communication is called *asynchronous messaging*, since client and server are decoupled by message queues. MOM functions much like electronic mail, storing and forwarding messages on behalf of client and server applications. Messages may be submitted even when the receiver happens to be temporarily unavailable, and are thus inherently more flexible and fault-tolerant than RPC. Examples of MOM are IBM's MQSeries product and the OMG Event Service. Web *push technologies* such as Marimba's *Castanet* also fall into the category of message-oriented middleware.

◆ **Transaction Processing (TP) Monitors:** Transaction processing (*q.v.*) monitors allow a client application to perform a series of requests on multiple remote servers while preserving consistency among the servers. Such a series of requests is called a *transaction*. The TP monitor ensures that either all requests that are part of a transaction succeed, or that the servers are *rolled back* to the state they had before the unsuccessful transaction was started. A transaction fails when one of the involved computers or applications goes down, or when any of the applications decides to *abort* the transaction. TP monitors are part of client–server products such as Novell’s Tuxedo and Transarc’s Encina.

A TP monitor can be used within a banking system when funds are withdrawn from an account on one database server and deposited in an account on another database server. The monitor makes sure that the transaction occurs in an “all or nothing” fashion. If any of the servers fails during the transfer then the transaction is rolled back such that both accounts are in the state they were before transaction was started.

Bibliography

1995. Mowbray, T. J., and Zahavi, R. *The Essential CORBA*. New York: John Wiley.
1996. Andrade, J. M. (ed.), Dwyer, T., Felts, S., and Carges, M. *The Tuxedo System: Software for Constructing and Managing Distributed Business Applications*. Reading, MA: Addison-Wesley.
1997. Shan, Y.-P., Earle, R. H., and Lenzi, M. A. *Enterprise Computing With Objects: From Client/Server Environments to the Internet*. Reading, MA: Addison-Wesley.
1998. Orfali, R., and Harkey, D. *Client/Server Programming with Java and CORBA*, 2nd Ed. New York: John Wiley.

Websites

Client–server frequently asked questions URLs: <http://www.abs.net/~lloyd/csfaq.txt>.

OMG CORBA documentation URL: <http://www.omg.org>.

OSF DCE documentation URL: <http://www.rdg.opengroup.org/public/pubs/catalog/dz.htm>.

Microsoft ActiveX and related technology URL: <http://www.microsoft.com/com>.

Silvano Maffei

CLUSTER COMPUTING

For articles on related subjects see CLIENT–SERVER COMPUTING; COOPERATIVE COMPUTING; DATABASE MANAGEMENT SYSTEM; DISTRIBUTED SYSTEMS; MULTIPROCESSING; NETWORKS, COMPUTER; PARALLEL PROCESSING; and SUPERCOMPUTERS.

Introduction

A cluster of computers, or simply a *cluster*, is a collection of computers that are connected together and

used as a single computing resource. Clusters have been used from the dawn of electronic computing as a straightforward way to obtain greater capacity and higher reliability than a single computer can provide. Clusters can be an informal, if not anarchic, computer organization. Often they have not been built by computer manufacturers but rather assembled by customers on an *ad hoc* basis to solve a problem at hand.

The first cluster probably appeared in the late 1950s or early 1960s when some company’s finance officer, realizing that payroll checks wouldn’t get printed if the computer broke down, purchased a spare. Software tools for managing groups of computers and submitting batch jobs to them, such as IBM’s Remote Job Entry (RJE) System, became commercially available in the mid-1970s. By the late 1970s, Tandem Computers began selling highly reliable systems that were clusters, with software to make them appear to access a single database system. However, it was not until the early 1980s that DEC (Digital Equipment Corporation—*q.v.*) coined the term *cluster* for a collection of software and hardware that made several VAX minicomputers (*q.v.*) appear to be a single time-sharing (*q.v.*) system called the VAXcluster.

With the appearance of very high performance personal workstations (*q.v.*) in the early 1990s, technical computer users began replacing expensive supercomputers with clusters of those workstations which they assembled themselves. Computer manufacturers responded with prepackaged workstation clusters, which became the standard form of supercomputers by the mid-1990s; a system of this type with special-purpose added hardware achieved the milestone of defeating the reigning human chess champion, Garry Kasparov (*see* COMPUTER CHESS). By 1998, even those systems were being challenged by user-constructed clusters of increasingly powerful personal computers. A very large, highly diffuse and informal cluster—using spare time on approximately 22,000 personal computers owned by volunteers, connected only occasionally though the Internet—succeeded in February 1998 in decoding a “challenge” message encrypted using the Data Encryption Standard system with a 56-bit key (*see* CRYPTOGRAPHY, COMPUTERS IN). The answer was found by simply trying one after another of the 63 quadrillion possible keys; success came after taking only 39 days to examine 85% of the keys. Appropriately, the decoded message read “Many hands make light work.”

Individual spectacular feats such as this are not, however, the reason that computer industry analysts estimated that half of all high performance server computer systems would be clusters by the turn of the century. Clusters provide a practical means of increasing