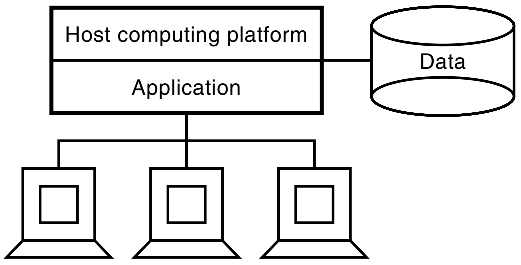

Figure 1. Host-based processing.

BUSINESS DATA PROCESSING

Fast-moving markets, frequent technological turnover, and rising customer expectations are examples of the many factors that continually change the basis of competition for modern businesses. Faced with such challenges, an enterprise often seeks to improve its competitive position through the innovative use of data processing technology. Information technology (IT) has evolved from rudimentary methods of data gathering in the 1970s to more sophisticated technologies for managing resources, sensing and monitoring market developments, and organizing information and human knowledge.

The stunning increase in processing speed of microprocessors during the last few decades, coupled with the rapid growth of networking, lead to even richer technological possibilities for modern businesses. Recent IT advances such as Web-based transaction processing and component-enabled software give firms new means of executing core processes, reducing costs, and creating new value for customers. Modern business data processing incorporates diverse technologies for networking, database management, software engineering, communications, and organizational coordination, and requires a keen architectural sensibility that seeks to integrate software and hardware components into systems that perform reliably in increasingly complex business environments.

CLIENT-SERVER ARCHITECTURE

Client-server architecture is a fundamental element in the design and deployment of business software systems. It relies on a simple model of cooperative processing: a client submits computing requests to a server, which processes the request and returns the results to the client. Client and server do not denote hard-coded functions. Rather, they are *roles* that application resources play as they interact in a computing environment.

To understand client-server as an architectural concept, it is useful first to identify predecessors of the client-server approach. *Host-based application processing* is an approach wherein all functional and data components of an application reside and execute on a single, centralized computing platform (Fig. 1). Remote users can execute the application from "dumb" terminals connected to the host across a network. From an architectural perspective, host-based processing is totally nondistributed.

Another common approach is the *single-user platform* model. Here all functional and data components reside on a single computing platform dedicated to a single user. The model applies to the use of personal computers and laptops for applications such as word processing, spreadsheets, desktop publishing, and personal data-base applications. Like host-based processing, the single-user platform model is totally nondistributed. It is possible to apply *local-area network* (LAN) technology (see LOCAL AREA NETWORKS) to give single-user applications simultaneous, shared access to data dispersed across a network. While this gives the appearance of creating a distributed environment, in fact it only chains desktop applications together so that they seem to reside together on a shared platform. Applications get access to remotely distributed data, but they are still designed to be executed on single platforms by single users.

Approach to Client-Server Design

Client-server takes a different approach to business application design. It seeks to partition functional and data components so that they can be executed on different computing platforms that share access to services such as printing or resources such as data repositories. Client-server evolved from a model of shared device processing, where single computing platforms such as those depicted in Fig. 2 began sharing access to a common resource, typically a file on the hard disk of a single computer or a printer connected to the LAN. For example, Novell's NetWare enabled networked computers to share access to a dedicated print service located at one of the nodes of the LAN.

As LANs grew in size and more powerful workstations became connected to networks, systems developers recognized the utility of distributing more than just dedicated file and print services across the network. They realized that *applications* themselves could be designed in a way that separated and distributed core functionalities and resources. Application processing in such a model becomes a game of coordination and communication: clients request services from

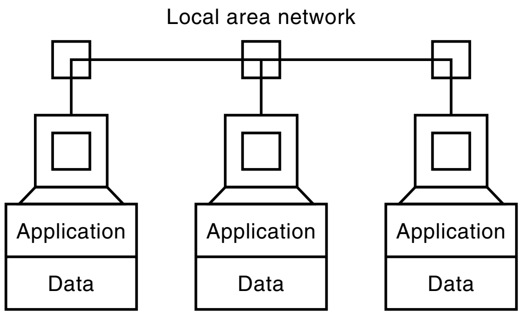

Figure 2. Single-user platforms connected across a local-area network.

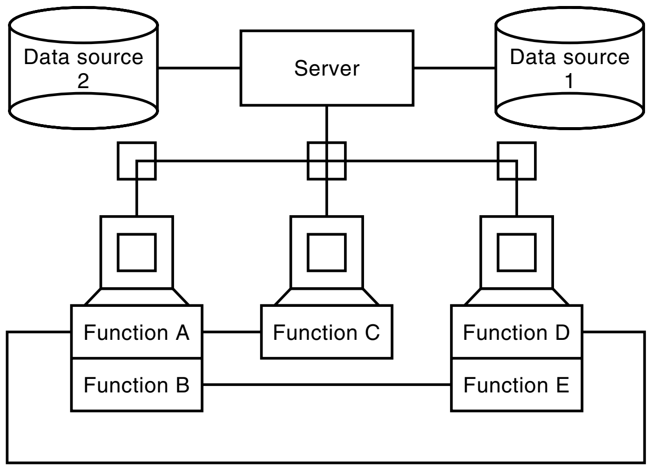

Figure 3. An example of a client-server environment.

servers; servers service those requests. Application design with client-server changes fundamentally: software developers must now carefully distribute resources and functions

- To leverage existing computer capabilities inside the enterprise
- To ensure that resources (e.g., data, microprocessor cycles) required for efficient application processing are easily accessible to the clients who need them
- To exploit the decentralization of client-server to create more powerful application experiences for the user

Client-server environments in which all the processing devices can request and provide services on an equal basis are examples of what is called *peer-to-peer,* or *end-to-end,* processing.

Benefits of the Client-Server Approach

From an engineering management perspective, client-server architecture yields the following advantages over host-based application processing and single-user platforms.

More Flexible Response to Increasing Workloads. In host-based application processing, response time for remote users running an application that resides on a mainframe computer remains relatively fixed until the system's capacity limit approaches. When the host reaches capacity, however, expensive upgrades are needed to accommodate the increased computing load. In single-platformed architectures, a user running an expensive application at one computer cannot exploit slack computing cycles available on another computer. By distributing applications, client-server effectively distributes application workloads so that the net response time for the average user decreases.

Increased Scalability. Scalability refers to the ease with which a distributed computing system can be scaled upward to handle new computing needs, either in the form of response times, new application requirements, or even new business directives. Enterprises worry about scalability because they want to protect and exploit their existing investments in IT, and they want their IT to grow to meet constantly changing business needs. Clients and servers running together across heterogeneous combinations of hardware and software require what is frequently called an *open systems* approach: they adhere to common protocols for interdevice and interapplication communication. To scale up existing IT structures in a captive market setting, enterprises must return to the provider from whom they purchased their original system. If that vendor does not provide solutions to match the enterprise's current needs, then the enterprise must pay huge switching costs to start from scratch, essentially, with another vendor's products. The client-server model encourages the creation of products and solutions based on shared standards for networking, system management, application programming, system services, and interfaces between applications and system services. Standards such as X/Open and standards bodies such as ISO (International Organization for Standardization) and OMG (Object Management Group) help provide mechanisms for software interoperability, which is really the key benefit of the client-server model.

Technological Innovation. Open systems allow independent hardware and software providers to create products that adhere to the broad boundaries laid out in industry-standard protocols. When coupled with the natural dynamics of market competition, open systems allow customers to experience the benefits of rapid technological innovation and product transitions. Rather than waiting for the technological advances created within the walls of a single, proprietary vendor, customers get access to a much greater number of technological advances created by multiple vendors participating in an open systems market.

OBJECT-ORIENTED METHOD FOR BUSINESS SYSTEMS DEVELOPMENT

Object orientation (OO) is an expansive model for expressing the context, requirements, and behavior of software systems. OO evolved in response to shortcomings and difficulties systems developers encountered in previous methodologies for software development. The systems developed using older methods frequently suffer from the following shortcomings:

- *Long Development Cycles.* Because of the long development cycles associated with conventional methods of systems development, too often the requirements of the business environment have changed drastically by the time systems developed according to earlier requirement assessments are finally deployed.
- *No Modifiability.* Once they are designed, coded, and debugged, those systems run until they are completely replaced.
- *High Maintenance Costs and Risks.* It is difficult for new programmers to understand the purpose and function of what are often referred to as *legacy systems,* programs based on older and possibly obsolete assessments of business need.
- *Lack of Scalability.* Older methods do not scale with the increasing complexity of large software systems for distributed transaction processing (e.g., airline reservation systems, banking systems) and sprawling data manage-

ment activities (e.g., decentralized global companies with data on bookings, backlog, shipments, and billings among distributed geographical locations). The reliability of these systems suffers as a result.

OO promises to reduce development time, increase flexibility and adaptability of software, and thereby improve overall software quality.

The primary end of OO is more modular, and thus more extensible, software composed of what are called *objects.* Objects can be embodied in programming code that is ultimately translatable into bits and bytes; but objects can also be expressed in purely abstract terms. The abstractions used to specify an object seem at first strange to programmers trained in procedural languages (e.g., Pascal, Fortran, and C) because they require software designers to separate the *definition* of software from its *implementation.* Such a separation is useful because it allows developers to identify objects from an application domain, and *then* to decide how to fit procedures and functional behavior around those objects. OO models can be used to communicate with application experts, to model complex enterprises, to prepare documentation, and to design programs and databases. In this way OO confronts one of the fundamental challenges of IT systems development: developers and users typically lack expertise in each other's domain, and therefore lack the vocabulary needed to specify system requirements and constraints unambiguously. By providing a unified framework for business *and* software engineering, OO bridges the gap between the formulation of an enterprise solution and its technical implementation.

OO Fundamentals and Their Implications for Business Systems Development

Objects are entities defined in terms of the following simple elements. (See OBJECT-ORIENTED PROGRAMMING for a more detailed treatment.)

- *Method / Procedure / Operation / Behavior.* Action that the object can take on data that reside inside the object. Some treatments of OO draw a distinction between an operation, considered to be an abstract process or service, and a method, considered to be the specification of an operation.
- *Properties / Attributes.* The types of data that an object records and manipulates, sometimes simply referred to as the object's data.
- *Message.* A request that an object sends to another object to invoke one of the receiving object's methods. The set of all such requests defines the receiving object's *interface.* An object satisfies an interface if it can be specified as the target object in each potential request described by the interface.

Two key principles shape the construction of object systems. *Encapsulation* refers to the packaging of data and methods together inside an object. For Object A to access Object B's data, A must send B a message; B processes A's request if the request is part of B's interface specification. Object A cannot access B's data except through the passing of messages. In the old procedural style of software specification, procedures accessed data that was typically grouped together at the beginning of a program. Ambiguity about which procedures were using which pieces of data made it difficult to debug and modify those old software systems. Encapsulation protects an object's data from arbitrary or unintended uses while hiding implementation details from the requestors of an object's services. For this reason encapsulation is often referred to as *data hiding.* It enforces a clear separation between *function* and *implementation,* allowing programmers to modify software systems by changing object implementations locally without having to modify an application that uses those objects.

To understand how encapsulation can be useful in a business context, let us consider a brief example. Suppose we have an object `Customer Order` consisting of attributes `Request Date` and `Receipt Date`, with an associated method called `Schedule` which computes a schedule date for the order using `Request Date` and `Receipt Date`. Suppose now that shifting business conditions require a change in the scheduling technique used to assign schedule dates to customer orders; the goal is now to schedule orders according to a function that includes `Leadtime Target` as well. Encapsulation allows us to localize this change inside the implementation of `Customer Order` without having to rewrite interfaces to other objects or functions of other objects that interact with `Customer Order`. When `Customer Order` receives a `Schedule` request, the requesting object does not need to know about the new formula for assigning schedule dates to customer orders—all it knows (or needs to know) is that the order will be scheduled.

Inheritance is a way of organizing objects into groupings that allow them to share attributes and methods. An object *type* specifies a family of objects. A *class* specifies the possible methods and attributes that apply to the objects of a fixed type. For example, since an Employee is a type of Person, an Employee object might inherit the attributes of a Person such Name, Age, and Address. It might also make sense to give the Employee object access to the same methods for updating, manipulating, and displaying this data. Employee defines a subclass that *inherits* the methods and attributes of the superclass Person. It might also be necessary to give Employee attributes and methods that are not inherited from Person, what are usually called *private* or *native* methods and attributes. For example, Employees need to get Paychecks, and the Employee object probably needs to have methods for displaying Paychecks. In this case, the attribute Paycheck and the method for displaying it are native to the object Employee.

A *class library* is a repository of object classes that can be used to provide commonly used functionalities. A *framework* is a class library that provides a software developer with a set of building blocks with which he can create new applications. For example, a framework for graphical interfaces allows a programmer to build graphical applications by invoking basic operations such as creating, bending, stretching, connecting, and deleting graphical icons, without requiring the programmer to implement those operations from scratch. Class libraries or frameworks can significantly reduce the time it takes to develop complex software systems. The success of new languages such as Java depends critically on the availability of frameworks and class libraries to help business programmers develop new applications efficiently.

Construction of Enterprise Software Systems Using OO

The OO paradigm has given rise to new mindsets and methodologies for constructing large software systems. OO languages such as C++, Smalltalk, and, more recently, Java have been used to build software applications for executing business transactions and for storing and manipulating large data sets (see OBJECT-ORIENTED PROGRAMMING). OO is useful as a method for conceptualizing software design and as a language that software developers and business users can use to communicate the context and function of software systems. To date OO has been especially useful in the engineering of software systems that make intensive use of graphical user interfaces (GUIs), but it has achieved less implementation success in software systems that make use of more complex, data-driven business rules. For example, the object model is fundamentally at odds with the relational model that has become the standard for database management [see RELATIONAL DATABASES and OBJECT-ORIENTED DATABASES). Technologies for translating between relational-database and object models are in the nascent stages of deployment, and show promise for connecting object-oriented tools to the essential data processing challenges most organizations face.

Various methods for OO analysis, such as the Booch, Rumbaugh, or Jacobson methods, can be used to design and deploy complex OO systems. Such methods adopt different perspectives on object design—for example, some use objects to model organizational processes such as order fulfillment, while others are better suited to the creation of class libraries for specific types of applications. At root, however, they all help a system developer

- Identify software objects that correspond to concepts or entities in the external business environment (e.g., customers, orders, bookings, policies)
- Specify *attributes* and *behaviors* for each object
- Determine hierarchical, functional, and data-sharing *relationships* among the objects of the system and *events* that trigger the invocation of methods among them

An older model of software development, where phases such as Requirements Analysis, Design, Coding, Testing, Debugging, and Validation proceed in a stepwise, linear fashion, is rendered obsolete in the OO paradigm. *Rapid prototyping* also known as *rapid application development* (RAD), is a centerpiece of object-oriented systems development: developers create a series of trial versions for a software system and continually test and refine those versions until they converge on the desired functionality.

DISTRIBUTED OBJECT COMPUTING

One key promise of object technology is *reusability,* the ability to create, exchange, and repeatedly use software *components* to build new software systems. According to such a scheme, software development evolves from a slow, expensive, sometimes arcane process into a nimbler, more design-oriented exercise wherein objects with clearly exposed interfaces are knit together to create complex systems. Some developers and architects envision the creation of object *foundries* and *factories* which produce, manage, and distribute components to support such a development approach. The immediate challenge is to converge on standards for the creation, linking, and execution of software components.

Three-Tier Models

The emergence of component-based software is evidence of a general movement away from traditional client-server computing to newer models of *distributed object computing.* In the first generation client-server model, applications were developed according to a *two-tier approach:* data for the application typically originated at the server, while user interface and computational tasks were handled at the client.

The two-tier model created several problems.

1. It did not allow optimal load-balancing among distributed clients because each client was solely responsible for performing business calculations required by the application.
2. Upgrading applications became difficult because upgrades typically consist of key changes to the business logic of an application, unique copies of which had proliferated to each client on the network.
3. Tying interface and implementation together at the client led to rigid constraints on what clients could do—no more and no less than what was programmed into the current application release.

In light of these shortcomings, software system developers began to see the advantages of decoupling interface from implementation, particularly by making application upgrades in decentralized, multi-user environments easier and also by making it possible for clients to request and fetch code chunks from a network on an as-needed basis.

The separation of interface from implementation has led to a new *three-tier model* (see Fig. 4), which provides the functional basis for components. By collecting implementation into an isolated middle layer and by specifying standards for describing the behaviors, properties, and events that can activate objects in that middle layer, developers can create and access generic software components that are fully interoperable in complex IT environments with multiple operating systems, and that can be stored in local clients or downloaded from a network to create richer application experiences for human users.

A *distributed object* obeys the regular properties of an object such as inheritance and classing, but is packaged as a

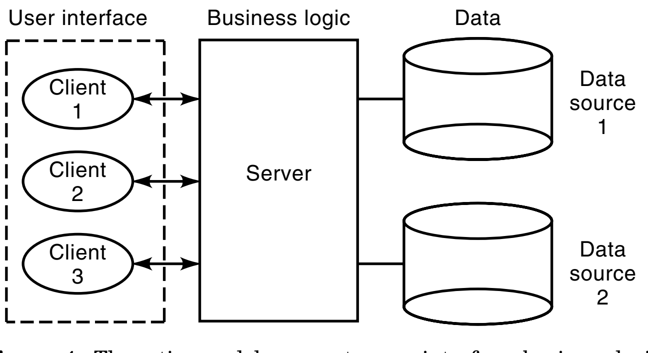

Figure 4. Three-tier models separate user interface, business logic, and data into distinct layers, which can be distributed among multiple servers and clients.

binary (or executable) software module accessible to remote clients by means of a method invocation. In essence the OO distinction between implementation and definition is taken a step further: the client need not know which language or compiler built a particular object, or even where the object physically resides. It only needs to know the name of the object and the interface it publishes.

Frameworks for Distributed Objects

A leading framework for distributed objects is CORBA (Core Object Request Broker Architecture), put forth by OMG, a non-profit consortium, in 1989. CORBA is a peer-to-peer computing framework wherein all applications are objects as previously defined. The central element of CORBA is what is called an ORB, or object request broker. It is easiest to think of an ORB as an *object interconnection bus:* it provides the means for objects to locate and activate other objects on a network, regardless of the processor or programming language used to develop those objects. ORBs also perform tasks such as managing connections among objects and delivering data. ORBs perform all of the intermediating functions that enable objects to interoperate on a fast, flexible basis.

To communicate object functions independently of their implementations, distributed applications use metadata such as IDL, Interface Definition Language. IDL is an abstract, declarative syntax for object encapsulations. An IDL compiler generates what are called *stub* and *skeleton* programs for each object's interface. A stub allows a client to access an object with a local function call. Transparently, it provides an interface to the ORB, which marshals the parameters of an object's method into formats suitable for transmission. The skeleton, meanwhile, provides the server-side implementation of an object's IDL interface. When an ORB receives a request, the skeleton provides a callback to a server-supplied function implementation.

When Object A requests services from Object B, the ORB locates B's implementation, transmits the parameters of A's request, and transfers control to B. When B is finished processing A's request, it passes control back to the ORB along with the results. Figure 5 depicts the approach.

Other commercial frameworks such as ActiveX and JavaBeans apply somewhat different approaches to the problem of modularizing and linking software components. While it is impossible to describe all of these frameworks in detail, it is useful to identify the following general features of any framework for distributed object computing:

1. *Coordinated Messaging.* Distributed software execution depends critically on a mechanism for coordinating messages among distinct software objects. Tools such as ORBs provide simple, efficient means of brokering transactions among distributed software components. They also introduce new efficiencies in distributed software design and performance by reducing the number of unique interfaces that need to be managed among distributed objects. As Fig. 6 shows, the bus or hub topologies characteristic of a message broker architectures such as CORBA reduce the number of custom interfaces among N software components from a number on the order of $N \times N$ to a number on the order of N.
2. *Network-Oriented Software.* Frameworks for distributed object computing allow applications to exploit resources that reside on a local file system *or* on a network. In this sense distributed object computing renders the distinction between client and server less relevant: objects, components, and data can be accessed dynamically on an as-needed basis; applications interact on a peer-to-peer basis. Enterprise software developers can use this capability to bridge data and application resources that typically reside in separate systems, such as shipping, manufacturing, finance, and sales.
3. *Metadata for Extensibility and Interoperability.* Tools such as IDL allow objects to expose their services and data to all the other objects on a network. This allows developers to join objects together without hard-coding calls to particular objects on particular servers. Services can be added to a software system and discovered at runtime. If metadata are defined consistently and ubiquitously across many services and applications, the raw amount of software needed to create software systems solutions should decrease because software chunks from previous applications or software developed using a different microprocessor or operating system architecture can be used as building blocks for new systems.

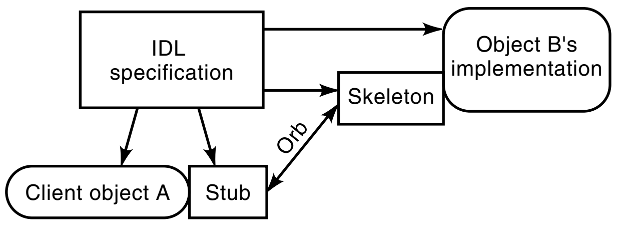

Figure 5. An ORB intermediating between stubs and skeletons.

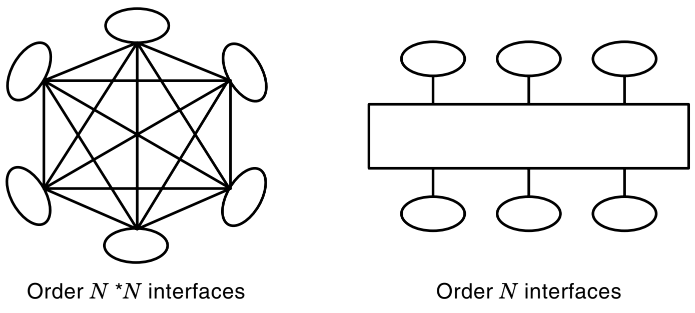

Figure 6. Custom interfaces versus message broker architectures.

INTERNET/INTRANET

In the early 1990s business organizations began to recognize the *Internet,* a decentralized system for linking computers first developed by the Defense Advanced Research Projects Agency, as a powerful means of disseminating information and knowledge. Though the Internet had previously found use primarily among academic institutions, around 1993 businesses and organizations began to serve World-Wide Web pages describing their products and services, and joined their employees together via Internet services such as electronic mail. The Internet and its enterprise counterpart, the *intra-*

net—so named to connote the linking of services and people *inside* an organization—provide powerful new tools for businesses to communicate, collaborate, and transact.

The Internet is a global, heterogeneous network of computers joined together through the shared use of a computer networking protocol called TCP/IP (see COMPUTER NETWORKS). *HTML* (HypterText mark-up language) is a simple scripting standard for the graphical presentation of information. HTTP (HyperText transfer protocol) is a networking standard based on TCP/IP that allows electronic files, software, graphics, and smaller programs called *applets* to be located, accessed, and transferred over a network. Together, HTML and HTTP provide the core technical foundation for browsing the *World-Wide Web.* Other related protocols such as FTP (file-transfer protocol), NFS (network file system), MIME (multipurpose Internet mail extensions), and SMTP (simple mail transfer protocol) also provide services for distributing and sharing information in a distributed network environment. All of these technical standards are deployed in both the Internet and Intranets. What distinguishes the Internet from an intranet is use and content, not underlying technology.

Uses and Benefits of Internet/Intranet

The Internet allows an organization to

- Make information about it products and services available to a wide base of potential customers
- Gather potentially valuable information (using freely available search engines, for example) about customer needs and competitor movements
- Execute business transactions, such as inventory replenishment and suppliers and invoicing with customers

An intranet, on the other hand, is an internally focused tool that records, facilitates, and enhances workflow. For example, it allows an organization to

- Collaborate on projects with participants from remote geographies and different groups
- Record institutional rationale in the form of presentations, e-mail conversations
- Share information on processes, projects, practices, and policies

Table 1 outlines some key differences between Internet and intranet.

Intranets provide many benefits to an organization. They allow *simplified information management and streamlined internal communication* using the browser paradigm. Browsers such as Mosaic, Netscape Navigator, and Internet Explorer all speak the same language of information exchange, primarily HTML. The ability to transfer Web pages using HTTP allows information on distributed servers to flow freely through an enterprise. Web navigation and search engines enable organizational participants to find the information they need to make better decisions or to interact more effectively with peers.

Challenges in Deployment of Intranet Technology

Before an organization can exploit intranet technology to improve business processes, however, it must analyze and solve a few architectural issues. MIS managers must configure all the desktop systems in the enterprise so that they have TCP/IP capability and a browser. They must also make decisions about what search engine to deploy. Perhaps most important, they must establish processes and policies for publishing, accessing, and exchanging information on the intranet.

As with the Internet, the most salient issue for intranet deployment at the time of this writing is security. Not all information can be placed on-line—for example, payroll, engineering prototypes, and shipment information are probably much too sensitive for all members of an organization to access. The main device for insuring security for the Internet or an intranet is a *firewall,* a software mechanism that filters individual packets of data as they pass into and out of specified servers and clients and screens them on the basis of source, destination, and service types (e.g., http, e-mail, and FTP).

Table 1. Key Differences between Internet and Intranet

Parameter	Internet	Intranet
Primary Uses	Branding and electronic commerce	Workflow
Interaction	Pull model wherein user identifies and downloads desired information in the form of Web pages. E-mail allows one-to-one or one-to-many communication. Also provides forum for linking users with shared interests in virtual communities.	Cross-functional, decentralized collaboration and communication, primarily via e-mail.
Transaction	Primarily electronic commerce, supported by technologies for security (e.g., user authentication, validation of network transmissions).	Support and execute internal processes such as human resources fulfillment, manufacturing specification, inventory control, employee expense reporting, etc.
Access	For the average consumer, access is typically over 14.4 kbs/s or 28.8 kbs/s phone lines.	For the average corporate user, access is typically over T1 or T3 lines that enable data transfer rates of at least 1.5 Mbs/s.

Management Policies

Policies for intranet management must address the following issues:

- *Content.* What kind of information can and cannot go on the intranet? Who is accountable for the information that gets published?
- *Administration.* How much usage of the intranet is allowed? Who is responsible for updating specific Web sites? What tools get deployed for monitoring usage and for converting documents and images to formats that

allow easy access and efficient downloading (e.g., Postscript vs. Acrobat)?

- *Design.* What is the uniform look and feel of sites on an intranet?
- *Security.* What tools should be deployed to protect against viruses, particularly those that can enter an intranet through an Internet access point? How is confidential organizational information protected?

First- and Second-Generation Intranets

Web browsing has been the primary mechanism by which organizations and individuals share and exchange information over the Internet or the intranet. The first generation of enterprise intranets have concentrated primarily on publishing information. Second-generation intranets concentrate on collaborating, interacting, and transacting, drawing on technologies and standards that go beyond browsers, HTTP, and HTML.

One of the technologies that has been used extensively for moving from first-generation intranets to second-generation intranets is *CGI* (*common gateway interface*), a standard for interfacing applications with information servers. CGI allows a basic level of realtime interactivity between a user and a Web-based application. Typically a user fills out what is frequently called a web form; the CGI application processes the information in the form by storing it to a remote database server or by comparing it to information accessed from a remote HTTP server; and finally the CGI application completes the transaction by outputing an answer back to the user's browser. Commercial database applications allow easy access to relational databases from the World-Wide Web and dynamic generation of HTML pages. Second-generation intranet applications for collaboration and transaction depend critically on environments for distributed object computing and component-enabled software.

BIBLIOGRAPHY

G. Booch, *Object-Oriented Analysis and Design with Applications,* 2nd ed., Reading, MA: Addison-Wesley, 1994.

A. Goldberg, *Smalltalk-80: The Interactive Programming Environment,* Reading, MA: Addison-Wesley, 1983.

C. Hall, *Technical Foundations of Client/Server Systems,* New York: Wiley, 1994.

T. Mowbray and R. Zahavi, *The Essential CORBA: Systems Integration Using Distributed Objects,* New York: Wiley, 1995.

R. Orfali and D. Harkey, *Client/Server Programming with Java and CORBA,* New York: Wiley, 1997.

P. Renaud, *Introduction to Client/Server Systems: A Practical Guide for Systems Profesionals,* 2nd ed., New York: Wiley, 1996.

D. A. Taylor, *Object-Oriented Technology, A Manager's Guide,* 2nd ed., Reading, MA: Addison-Wesley, 1998.

D. Taylor and P. Harmon, *Objects in Action: Commercial Applications of Object-Oriented Technologies,* Reading, MA: Addison-Wesley, 1993.

D. A. Taylor, *Business Engineering with Object Technology,* New York: Wiley, 1995.

TOM CHÁVEZ
Rapt Technologies Corporation