2 On Distributed Systems

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

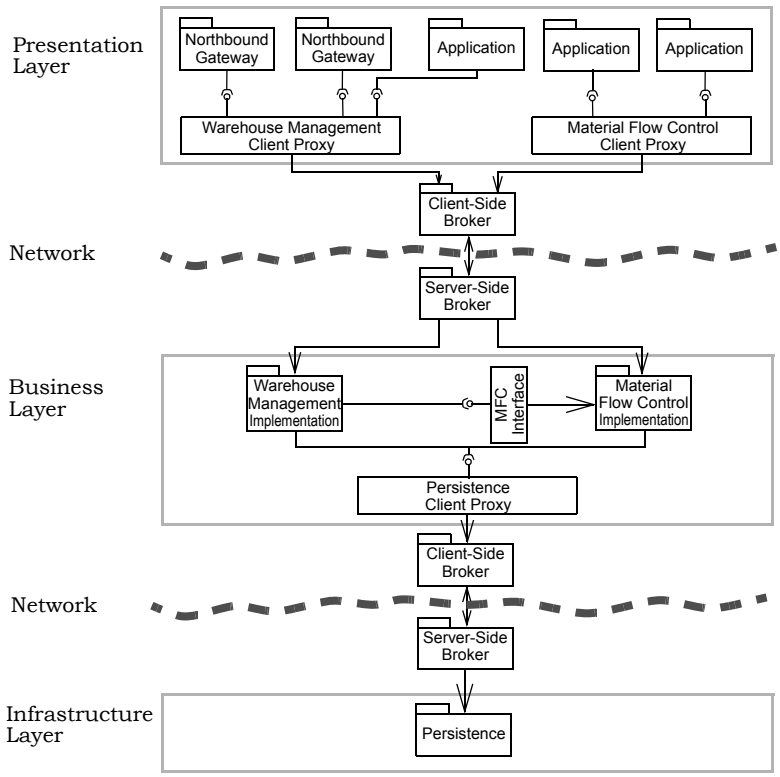
Leslie Lamport

A distributed system is a computing system in which a number of components cooperate by communicating over a network. The explosive growth of the Internet and the World Wide Web in the mid-1990's moved distributed systems beyond their traditional application areas, such as industrial automation, defense, and telecommunication, and into nearly all domains, including e-commerce, financial services, health care, government, and entertainment. This chapter describes the key characteristics and challenges of developing distributed systems and presents several key software technologies that have emerged to resolve these challenges.

2.1 Benefits of Distribution

Most computer software traditionally ran in *stand-alone systems*, where the user interface, application 'business' processing, and persistent data resided in one computer, with peripherals attached to it by buses or cables. Few interesting systems, however, are still designed this way. Instead, most computer software today runs in *distributed systems,* where the interactive presentation, application business processing, and data resources reside in loosely-coupled computing nodes and service tiers connected together by networks.

The following diagram illustrates a three-tier distribution architecture for a warehouse management process control system, whose pattern-based design we discuss in depth in Part II, *A Story*, of this book. The three tiers in this example are connected by a BROKER architecture (237).



The following properties of distributed systems make them increasingly essential as the foundation of information and control systems [Tan92]:

- *Collaboration and connectivity*. An important motivation for distributed systems is their ability to connect us to vast quantities of geographically distributed information and services, such as maps, e-commerce sites, multimedia content, and encyclopedias. The popularity of instant messaging and chat rooms on the Internet highlights another motivation for distributed systems: keeping in touch with family, friends, co-workers, and customers.
- *Economics*. Computer networks that incorporate PDAs, laptops, PCs, and servers often offer a better price/performance ratio than centralized mainframe computers. For example, they support decentralized and modular applications that can share expensive peripherals, such as high-capacity file servers and high-resolution printers. Similarly, selected application components and services can be delegated to run on nodes with specialized processing attributes, such as high-performance disk controllers, large amounts of memory, or enhanced floating-point performance. Conversely, small simple applications can run on inexpensive commodity hardware.
- *Performance and scalability*. Successful software typically collects more users and requirements over time, so it is essential that the performance of distributed systems can scale up to handle the increased load and capabilities. Significant performance increases can be gained by using the combined computing power of networked computing nodes. In addition—at least in theory—multiprocessors and networks can scale easily. For example, multiple computation and communication service processing tasks can be run in parallel on different nodes in a server farm.
- *Failure tolerance*. A key goal of distributed computing is to tolerate partial system failures. For example, although all the nodes in a network may be live, the network itself may fail. Likewise, an endsystem in a network or a CPU in a multiprocessor system may crash. Such failures should be handled gracefully without affecting all—or unrelated—parts of the system. A common way to

implement fault tolerance is to replicate services across multiple nodes and/or networks. Replication helps minimize single points of failure, which can improve system reliability in the face of partial failures.

- *Inherent distribution.* Some applications are inherently distributed, including telecommunication management network (TMN) systems, enterprise business systems that span multiple divisions in different regions of the world, peer-to-peer (P2P) content sharing systems, and business-to-business (B2B) supply chain management systems. Distribution is not optional in these types of systems—it is essential to meet customer needs.

2.2 Challenges of Distribution

Despite the increasing ubiquity and importance of distributed systems, developers of software for distributed systems face a number of hard challenges [POSA2], including:

- *Inherent complexities*, which arise from fundamental domain challenges: For example, components of a distributed system often reside in separate address spaces on separate nodes, so inter-node communication needs different mechanisms, policies, and protocols than those used for intra-node communication in a stand-alone systems. Likewise, synchronization and coordination is more complicated in a distributed system since components may run in parallel and network communication can be asynchronous and non-deterministic. The networks that connect components in distributed systems introduce additional forces, such as latency, jitter, transient failures, and overload, with corresponding impact on system efficiency, predictability, and availability [VKZ04].
- *Accidental complexities*, which arise from limitations with software tools and development techniques, such as non-portable programming APIs and poor distributed debuggers. Ironically, many accidental complexities stem from deliberate choices made by developers who favor low-level languages and platforms, such as C and C-based operating system APIs and libraries, that scale up

poorly when applied to distributed systems. As the complexity of application requirements increases, moreover, new layers of distributed infrastructure are conceived and released, not all of which are equally mature or capable, which complicates development, integration, and evolution of working systems.

- *Inadequate methods and techniques*. Popular software analysis methods and design techniques [Fow03b] [DWT04] [SDL05] have focused on constructing single-process, single-threaded applications with 'best-effort' QoS requirements. The development of high-quality distributed systems—particularly those with stringent performance requirements, such as video-conferencing or air traffic control systems—has been left to the expertise of skilled software architects and engineers. Moreover, it has been hard to gain experience with software techniques for distributed systems without spending much time wrestling with platform-specific details and fixing mistakes by costly trial and error.
- *Continuous re-invention and re-discovery* of core concepts and techniques. The software industry has a long history of recreating incompatible solutions to problems that have already been solved. There are dozens of general-purpose and real-time operating systems that manage the same hardware resources. Similarly, there are dozens of incompatible operating system encapsulation libraries, virtual machines, and middleware that provide slightly different APIs that implement essentially the same features and services. If effort had instead been focused on enhancing a smaller number of solutions, developers of distributed system software would be able to innovate more rapidly by reusing common tools and standard platforms and components.

2.3 Technologies for Supporting Distribution

To address the challenge described above, therefore, three levels of support for distributed computing were developed: *ad hoc network programming*, *structured communication*, and *middleware* [Lea02]. At the *ad hoc network programming* level reside interprocess communication (IPC) mechanisms, such as shared memory, pipes, and sockets [StRa05], that allow distributed components to connect and exchange information. These IPC mechanisms help address a key challenge of distributed computing: enabling components from different address spaces to cooperate with one another.

Certain drawbacks arise, however, when developing distributed systems only using ad hoc network programming support. For instance, using sockets directly within application code tightly couples this code to the socket API. Porting this code to another IPC mechanism or redeploying components to different nodes in a network thus becomes a costly manual programming effort. Even porting this code to another version of the same operating system can require code changes if each platform has slightly different APIs for the IPC mechanisms [POSA2] [SH02]. Programming directly to an IPC mechanism can also cause a paradigm mismatch, for example, local communication uses object-oriented classes and method invocations, whereas remote communication uses the function-oriented socket API and message passing.

Some applications and their developers can tolerate the deficiencies of ad hoc network programming. For example, traditional embedded systems, such as controllers for automobile engines or power grids, run in a homogeneous distributed environment whose initial functional requirements, component configuration, and choice of IPC mechanism rarely changes. Most other types of applications cannot tolerate these deficiencies, however, since they apply run in a heterogeneous computing environment and/or face continuous requirement changes.

The next level of support for distributed computing is *structured communication*, which overcomes limitations with ad hoc network programming by not coupling application code to low-level IPC mechanisms, but instead offering higher-level communication

mechanisms to distributed systems. Structured communication encapsulates machine-level details, such as bits and bytes and binary reads and writes. Application developers are therefore presented with a programming model that embodies types and a communication style closer to their application domain.

Historically significant examples of structured communication are Remote Procedure Call (RPC) platforms, such as Sun RPC [Sun88] and the Distributed Computing Environment (DCE) [RKF92]. RPC platforms allow distributed applications to cooperate with one another much like they would in a local environment: they invoke functions on each other, pass parameters along with each invocation, and receive results from the functions they called. The RPC platform shields them from details of specific IPC mechanisms and low-level operating system APIs. Other examples of structured communication include PROFInet [WK01], which provides a run-time model for industrial automation that defines several message-oriented communication protocols, and ACE [SH02] [SH03], which provides reusable C++ wrapper facades and frameworks that perform common structured communication tasks across a range of OS platforms.

Despite its improvements over ad hoc network programming, structured communication does not fully resolve the challenges described above. In particular, components in a distributed system that communicate via structured communication are still aware of their peers' remoteness—and sometimes even their location in the network. While location awareness may suffice for certain types of distributed systems, such as statically configured embedded systems whose component deployment rarely changes, structured communication does not fulfill the following the properties needed for more complex distributed systems:

- *Location-independence of components*. Ideally, clients in a distributed system should communicate with collocated or remote services using the same programming model. Providing this degree of location-independence requires the separation of code that deals with remoting or location-specific details from client and service application code. Even then, of course, distributed systems have failure modes that local systems do not have [WWWK96].

- *Flexible component (re)deployment*. The original deployment of an application's services to network nodes could become suboptimal as hardware is upgraded, new nodes are incorporated, and/or new requirements are added. A redeployment of distributed system services may therefore be needed, ideally without breaking code and or shutting down the entire system.
- *Integration of legacy code*. Few complex distributed systems are developed from scratch. Instead, they are constructed from existing elements or applications that may not have originally been designed to integrate into a distributed environment—in fact, the source code may not even be available. Reasons for integrating legacy code include leveraging existing software components, minimizing software certification costs, or reducing time-to-market.
- *Heterogeneous components*. Distributed system integrators are increasingly faced with the task of combining heterogeneous enterprise distributed systems built using different off-the-shelf technologies, rather than just integrating proprietary software developed in-house. Moreover, with the advent of enterprise application integration (EAI) [HoWo03] it has become necessary to integrate components and applications written in different programming languages into a single, coherent distributed system. Once integrated, these heterogeneous components should perform a common set of tasks properly.

Mastering these challenges requires more than structured communication support for distributed systems. Instead it requires dedicated *middleware* [ScSc02], which is distribution infrastructure software that resides between an application and the operating system, network, or database underneath it. Middleware provides the properties described above so that application developers can focus on their primary responsibility: implementing their domain-specific functionality.

Realizing the need for middleware has motivated companies, such as Microsoft, IBM, and Sun, and consortia, such as the Object Management Group (OMG) and the World Wide Web Consortium (W3C), to develop technologies for distributed computing. Below, we describe a number of popular middleware technologies, including distributed object computing, component middleware, publish/ subscribe middleware, and service-oriented architectures and Web Services [Vin04a].

Distributed Object Computing Middleware

A key contribution to distributed system development was the emergence of *distributed object computing (DOC) middleware* in the late 1980s and early 1990s. DOC middleware represented the confluence of two major information technologies: RPC-based distributed computing systems and object-oriented design and programming. Techniques for developing RPC-based distributed systems, such as DCE [OG94], focused on integrating multiple computers to act as a unified scalable computational resource. Likewise, techniques for developing object-oriented systems focused on reducing complexity by creating reusable frameworks and components that reify successful patterns and software architectures. DOC middleware therefore used object-oriented techniques to distribute reusable services and applications efficiently, flexibly, and robustly over multiple, often heterogeneous, computing and networking elements.

CORBA 2.x [OMG03a] [OMG04a] and Java RMI [Sun04c] are examples of DOC middleware technologies for building applications for distributed systems. These technologies focus on interfaces, which are contracts between clients and servers that define a location-independent means for clients to view and access object services provided by a server. Standard DOC middleware technologies like CORBA also define communication protocols and object information models to enable interoperability between heterogeneous applications written in various languages running on various platforms [HV99].

Despite its maturity, performance, and advanced capabilities, however, DOC middleware had various limitations, including:

- *Lack of functional boundaries*. The CORBA 2.x and Java RMI object models treat all interfaces as client/server contracts. These object models do not, however, provide standard assembly mechanisms to decouple dependencies among collaborating object implementations. For example, objects whose implementations depend on other objects need to discover and connect to those objects explicitly. To build complex distributed applications, therefore, application developers must explicitly program the connections among interdependent services and object interfaces, which is extra work that can yield brittle and non-reusable implementations.
- *Lack of software deployment and configuratoin standards*. There is no standard way to distribute and start up object implementations remotely in DOC middleware. Application administrators must therefore resort to in-house scripts and procedures to deliver software implementations to target machines, configure the target machine and software implementations for execution, and then instantiate software implementations to make them ready for clients. Moreover, software implementations are often modified to accommodate such *ad hoc* deployment mechanisms. The need of most reusable software implementations to interact with other software implementations and services further aggravates the problem. The lack of higher-level software management standards results in systems that are harder to maintain and software component implementations that are much harder to reuse.

Component Middleware

Starting in the mid to late 1990s, *component middleware* evolved to address the limitations of DOC middleware described above. In particular, to address the lack of functional boundaries, component middleware allows a group of cohesive component objects to interact with each other through multiple provided and required interfaces and defines standard runtime mechanisms needed to execute these component objects in generic applications servers. To address the lack of standard deployment and configuration mechanisms,

component middleware also often specifies the infrastructure to package, customize, assemble, and disseminate components throughout a distributed system.

Enterprise JavaBeans [Sun03] [Sun04a] and the CORBA Component Model (CCM) [OMG02] [OMG04b] are examples of component middleware that define the following general roles and relationships:

- A *component* is an implementation entity that exposes a set of named interfaces and connection points that components use to collaborate with each other. Named interfaces service method invocations that other components call synchronously. Connection points are joined with named interfaces provided by other components to associate clients with their servers. Some component models also offer event sources and event sinks, which can be joined together to support asynchronous message passing.
- A *container* provides the server runtime environment for component implementations. It contains various pre-defined hooks and operations that give components access to strategies and services, such as persistence, event notification, transaction, replication, load balancing, and security. Each container defines a collection of runtime strategies and policies, such as transaction, persistence, security, and event delivery strategies, and is responsible for initializing and providing runtime contexts for the managed components. Component implementations often have associated metadata written in XML that specify the required container strategies and policies [OMG03b].

In addition to the building blocks outlined above, component middleware also typically automates aspects of various stages in the application development lifecycle, notably component implementation, packaging, assembly, and deployment, where each stage of the lifecycle adds information pertaining to these aspects via declarative metadata [DBOSG05]. These capabilities enable component middleware to create applications more rapidly and robustly than their DOC middleware predecessors.

There are well-defined relationships between components and objects in a component architecture [Szy02]. In general, components are created at build time, may be loaded at runtime, and define the implementation detail for runtime behavior. Likewise, objects are

created at runtime, their type is packaged within a component, and their runtime actions are what drives program behavior. Thus, components get written, built and loaded, whereas objects get created and interact.

Publish/Subscribe and Message-Oriented Middleware

RPC platforms, DOC middleware, and component middleware are all based on a request/response communication model, where requests flow from client to server and responses flow back from server to client. However, certain types of distributed applications, particularly those that react to external stimui and events, such as control systems and online stock trading systems, are not well-suited certain aspects of the request/response communication model. These aspects include *synchronous communication* between the client and server, which can underutilize the parallelism available in the network and endsystems, *designated communication*, where the client must know the identity of the server, which tightly couples it to a particular recipient, and *point-to-point communication*, where a client talks with just one server at a time, which can limit its ability to convey its information to all interested recipients.

An alternative approach to structuring communication in certain types of distributed systems is therefore to use *message-oriented middleware*, which is supported by IBM's MQ Series [IBM99], BEA's MessageQ [BEA06] and TIBCO's Rendezvous, or *publish/subscribe* middleware, which is supported by the Java Messaging Service (JMS) [Sun04b], the Data Distribution Service (DDS) [OMG05b], and WS-NOTIFICATION [OASIS06c] [OASIS06c]. The main benefits of message-oriented middleware include its support for *asynchronous communication*, where senders transmit data to receivers without blocking to wait for a response. Many message-oriented middleware platforms provide transactional properites, where messages are reliably queued and/or persisted until consumers can pick them up. Publish/subscribe middleware augments this capability with *anonymous communication*, where publishers and subscribers are loosely coupled and thus do not know about each other existence since the address of the receiver is not conveyed along with the event data, and *group communication*, where there can be multiple subscribers who receive events sent by a publisher.

Publish/subscribe middleware typically allows applications to run on separate nodes and write/read events to/from a global data space in a distributed system. Applications can share information with others by using this global data space to declare their intent to produce events, which is often categorized into one or more topics of interest to participants. Applications that want to access topics of interest—or simply handle all messages on a particular queue—can declare their intent to consume the events.

The elements of publish/subscribe middleware are separated into the following roles:

- *Publishers* are sources of events, that is, they produce events on certain topics that are then propagated through the system. Depending on architecture implementation, publishers may need to describe the type of events they generate *a priori*.
- *Subscribers* are the event sinks of the system, that is, they consume data on topics of interest to them. Some architecture implementations require subscribers to declare filtering information for the events they require.
- *Event channels* are components in the system that propagate events from publishers to subscribers. These channels can propagate events across distribution domains to remote subscribers. Event channels can perform various services, such as filtering and routing, QoS enforcement, and fault management.

The events passed from publishers to consumers can be represented in various ways, ranging from simple text messages to richly-typed data structures. Likewise, the interfaces used to publish and subscribe the events can be generic, such as `send` and `recv` methods that exchange arbitrary dynamically typed XML messages in WS-NOTIFICATION, or specialized, such as a data writer and data readers that exchange statically typed event data in DDS.

Service-Oriented Architectures and Web Services

Service-Oriented Architecture (SOA) is a style of organizing and utilizing distributed capabilities that may be controlled by different organizations or owners. It therefore provides a uniform means to offer, discover, interact with and use capabilities of loosely coupled [Kaye03] and interoperable software services to support the requirements of the business processes and application users [OASIS06a]. The term 'SOA' was originally coined in the mid-1990's [SN96] as a generalization of the interoperability middleware standards available at the time, including RPC-, ORB-, and messaging-based platforms.

The ubiquity of the World Wide Web (WWW) and the lessons learned from earlier forms of middleware were leveraged to form the initial version of SOAP [W3C03]. SOAP is a protocol for exchanging XML-based [W3C06b] messages over a computer network, normally using HTTP [FGMFB97]. Initially SOAP was intended as a platform-agnostic protocol that could be used over the Web to allow interoperability with various types of middleware, including CORBA, EJB, JMS, and proprietary message-oriented middleware systems, such as IBM's MQ Series and TIBCO Rendezvous.

The introduction of SOAP spawned a popular new variant of SOA called *Web Services* that is being standardized by the World Wide Web Consortium (W3C). Web Services allow developers to package application logic into services whose interfaces are described with the Web Service Description Language (WSDL) [W3C06a]. WSDL-based services are often accessed using standard higher-level Internet protocols, such as SOAP over HTTP. Web Services can be used to build an Enterprise Service Bus (ESB), which is a distributed computing architecture that simplifies interworking between disparate systems. Mule [Mule06] and Celtix [Celtix06] are open-source examples of the ESB approach to melding groups of heterogeneous systems into a unified distributed application.

Despite some highly publicized drawbacks [Bell06] [Vin04b], Web Services have established themselves as the technology of choice for most enterprise business applications. This does not mean, however, that Web Services will completely displace earlier middleware technologies, such as EJB and CORBA. Rather, Web Services

complements these earlier successful middleware technologies and provides standard mechanisms for interoperability. For example, the Microsoft Windows Communication Foundation (WCF) platform [MMW06] and the Service Component Architecture (SCA) [SCA05] being defined by IBM, BEA, IONA, and others combine aspects of component-based development and Web technologies. Like components, WCF and SCA platforms provide black-box functionality that can be described and reused without concern for how a service is implemented. Unlike traditional component technologies, however, WCF and SCA are not accessed using the object model-specific protocols defined by DCOM [Box97] [Thai99], Java RMI, or CORBA. Instead, Web services are accessed using Web protocols and data formats, such as HTTP and XML, respectively.

Since initial Web Services developments provided an RPC model that exchanged XML messages over HTTP they were touted as replacements for more complicated EJB components or CORBA objects. When used for fine-grained distributed resource access, however, the performance of Web Services is often several orders of magnitude slower than DOC middleware due to its their use of plain-text protocols, such as XML over HTTP [EPL02]. As a result, the use of Web Services for performance-critical applications, such as distributed real-time and embedded systems in aerospace, military, financial services, and process control domains, is now considered much less significant than using them for loosely-coupled document-oriented applications, such as supply-chain management.

Rather than trying to replace older approaches, today's Web Services technologies are instead focusing on middleware integration, thereby adding value to existing middleware platforms. WSDL allows developers to abstractly describe Web Service interfaces while also defining concrete bindings, such as the protocols and transports required at runtime to access the services. By providing these common communication mechanisms between diverse middleware platforms, Web Services allow component reuse across an organization's entire application set, regardless of their implementation technologies. For example, projects such as the Apache Web Services Invocation Framework (WSIF) [Apache06], Mule, and CeltiXfire, aim to allow applications to access Web Services transparently via EJB, JMS, or the SCA. This move towards

integration allows services implemented in these different technologies to be integrated into an ESB and made available to a variety of client applications. Middleware integration is thus a key focus of Web Services applications for the foreseeable future [Vin03]. By focusing on integration, Web Services increases reuse and reduces middleware lock-in, so developers can use the right middleware to meet their needs without precluding interoperability with existing systems.

2.4 Limitations with Middleware

Despite the many benefits of middleware described in this chapter, it is not a panacea for distributed systems. All the middleware technologies described above are primarily just 'messengers' between elements in distributed applications, and sometimes the messages just cannot be delivered despite heroic efforts from the middleware. As a result, distributed applications must be prepared to handle network failures and server crashes. Likewise, middleware cannot magically solve problems resulting from poor deployment decisions, which can significantly degrade system stability, predictability, and scalability.

In other words, middleware is an important part of a distributed system, but it cannot handle responsibilities that are application-specific and thus beyond its scope. Distributed systems must therefore be designed and validated carefully, even when middleware allows them to be independent of the concrete location of other components.