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

What is a Distributed System?

- A distributed system is a computing system in which a number of *components* cooperate by communicating over a network.
- Most "large" software systems are distributed.
- Traditional application areas include industrial automation, defense, and telecommunications.
- Since the mid 1990's the growth of the World Wide Web has expanded distributed systems into many other areas including e-commerce, financial services, health care, government, and entertainment.

What are Software Components?

- Components are executable units of independent production, acquisition, and deployment that can be composed into a functioning system.
- Most large software systems are comprised of distributed components and some "glue" code.
- Standard "off the shelf" components (COTS) are often used.

Benefits of Distribution

- Collaboration and connectivity.
  - Ability to connect to vast quantities of geographically distributed information and services such as maps, e-commerce sites, multimedia content, and encyclopedias.
  - Ability to keep in touch with family, friends, co-workers, and customers through instant messaging, chat rooms, social networking sites, etc.

Benefits of Distribution (2)

- Economics
  - Computer networks that incorporate PDAs, laptops, PCs, and servers often offer a better price/performance ratio than centralized mainframe computers.
  - Decentralized and modular applications can share expensive peripherals such as high-capacity file servers and high-resolution printers.
  - Selected application components 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.
  - Simple components can run on inexpensive commodity hardware.

Benefits of Distribution (3)

- 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 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 or in different virtual machines on the same server.

Benefits of Distribution (4)

- 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.
  - Similarly, an end-system 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.

Benefits of Distribution (5)

- Inherent distribution.
  - Some applications are inherently distributed, including telecommunication management network (TMN) systems, enterprise business systems that span multiple company 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.

Challenges of Distribution

- Inherent complexities.
  - 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 stand-alone systems.
  - Similarly, synchronization and coordination is more complicated in a distributed system, as 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.

Challenges of Distribution (2)

- Accidental complexities.
  - Arise from limitations of 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 favoring low-level languages and platforms such as C and C-based operating system APIs and libraries, which scale up poorly when applied to distributed systems.
  - As the complexity of application requirements increases, new layers of distributed infrastructure are conceived and released, not all of which are equally mature or capable, which complicates the development, integration, and evolution of working systems.

Challenges of Distribution (3)

- Inadequate methods and techniques.
  - Popular software analysis methods and design techniques 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 a lot of time wrestling with platform-specific details and fixing mistakes by costly trial and error.

Challenges of Distribution (4)

- 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.

Technologies for Supporting Distribution

- To address the challenges of distributed computing, three levels of support for distributed computing were developed:
  - Ad hoc network programming
  - Structured communication
  - Middleware

Ad Hoc Network Programming

- Interprocess communication (IPC) mechanisms such as shared memory, pipes, and sockets allow distributed components to connect and exchange information.
- These IPC mechanisms help address a key challenge of distributed computing: enabling components in different address spaces to cooperate with one another.

Ad Hoc Network Programming (2)

- Using IPC mechanisms (e.g. sockets) directly within application code tightly couples the 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 the code to another version of the same operating system can require code changes if each platform has slightly different APIs for the IPC mechanisms.
- 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.

Ad Hoc Network Programming (3)

- 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, because they run in a heterogeneous computing environment and/or face continuous requirement changes.

Structured Communication

- Overcomes limitations with ad hoc network programming by not coupling application code to low-level IPC mechanisms.
- Instead offers 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 data types and a communication style closer to their application domain.

Structured Communication (2)

- Examples of structured communication are Remote Procedure Call (RPC) platforms, such as Sun RPC and the Distributed Computing Environment (DCE).
- RPC platforms allow distributed applications to cooperate with one another much as 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 call.
  - The RPC platform shields them from the details of specific IPC mechanisms and low-level operating system APIs.

Structured Communication (3)

- 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 certain properties needed for more complex distributed systems.

Desired Properties

- 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.

Desired Properties (2)

- 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, or new requirements are added.
  - A redeployment of distributed system services may therefore be needed, ideally without breaking code or shutting down the entire system.

Desired Properties (3)

- Integration of existing elements.
  - Few complex distributed systems are developed from scratch.
  - They are constructed from COTS components or legacy elements or applications that may or may not have been designed to integrate into a distributed environment.
  - Often the source code is not available.
  - Reasons for integrating existing components include minimizing development costs, minimizing software certification costs, or reducing time-to-market.

Desired Properties (4)

- Heterogeneous components.
  - Distributed system integrators are faced increasingly 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), 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.

Middleware

- Middleware 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.
- Popular middleware technologies include:
  - Distributed object computing
  - Component middleware
  - Publish/subscribe middleware
  - Service-oriented architectures
  - Web services

Distributed Object Computing (DOC) Middleware

- DOC middleware uses object-oriented techniques to distribute reusable services and applications efficiently, flexibly, and robustly over multiple, often heterogeneous, computing and networking elements.
- CORBA and Java RMI are examples of DOC middleware technologies.
- 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.
- DOC middleware technologies like CORBA also define communication protocols and object information models, to enable interoperability between heterogeneous applications written in various languages and running on various platforms.

Limitations of DOC Middleware

- 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 developers must program the connections among interdependent services and object interfaces explicitly, which is extra work that can yield brittle and non-reusable implementations.

Limitations of DOC Middleware (2)

- Lack of software deployment and configuration standards.
  - There is no standard way to distribute and start up object implementations remotely in DOC middleware.
  - Application administrators must create scripts and procedures to deploy server components. Moreover, component implementations are often modified to accommodate such ad hoc deployment mechanisms.
  - 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

- Enterprise JavaBeans (EJB) and the CORBA Component Model (CCM) are examples of component middleware.
- To address the lack of functional boundaries, component middleware allows a group of cohesive component objects to interact with each other through multiple interfaces, and defines the standard runtime mechanisms needed to execute these component objects in generic application servers.
- To address the lack of standard deployment and configuration mechanisms, component middleware often also specifies the infrastructure to package, customize, assemble, and disseminate components throughout a distributed system.

Component Middleware (2)

- A *component* is an implementation entity that exposes a set of named interfaces and connection points that components can use to collaborate with each other.
- A *container* provides the server runtime environment for component implementations.
  - It contains various predefined hooks and operations that give components access to strategies and services, such as persistence, event notification, transaction, replication, load balancing, and security.
  - Component implementations often have associated metadata written in XML that specifies the required container policies regarding transactions, persistence, security, etc.

Component Middleware (3)

- Component middleware also typically automates aspects of various stages in the application development lifecycle including component implementation, packaging, assembly, and deployment.
  - These capabilities enable component middleware to create applications more rapidly and robustly than their DOC middleware predecessors.
- Well defined relationships exist between components and objects in a component architecture.
  - Components are created at build time, loaded at runtime, and define the implementation details for runtime behavior.
  - Objects are created at runtime, their type is packaged within a component, and their runtime actions are what drives program behavior.

RPC, DOC, and Component Middleware Problems

- RPC platforms, DOC middleware, and component middleware are all based on a request/response communication model, in which requests flow from client to server and responses flow back from server to client.
- This model is not well-suited to certain kinds of distributed applications, particularly those that react to external stimuli and events (e.g. control systems and online stock trading systems).

RPC, DOC, and Component Middleware Problems (2)

- *Synchronous communication* between the client and server can underutilize the parallelism available in the network and end-systems.
- *Designated communication* requires the client to know the identity of the server, which tightly couples it to a particular recipient.
- *Point-to-point communication* limits a client to communicating with just one server at a time, which can restrict its ability to convey information to all interested recipients.

Message-Oriented Middleware

- Message-oriented middleware supports asynchronous communication, in which senders transmit data to receivers without blocking to wait for a response.
- Many message-oriented middleware platforms provide transactional properties, in which messages are reliably queued and/or persisted until consumers can pick them up.
- *Publish/subscribe* middleware augments these capabilities with anonymous communication:
  - Publishers and subscribers do not need to know about each other's existence.
  - There can be multiple subscribers that receive events sent by a publisher.

Publish/Subscribe Middleware

- Publish/subscribe middleware typically allows components to run on separate nodes and write/read events to/from a global data space in a distributed system.
- Components 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.

Publish/Subscribe Middleware (2)

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

Service Oriented Architectures (SOA)

- Service-Oriented Architecture (SOA) is a set of design principles used during systems development and integration.
- It provides a uniform means to offer, discover, interact with and use the capabilities of loosely coupled and interoperable software services that can be used within multiple business domains.
- The term 'SOA' was originally coined in the mid-1990s as a generalization of the interoperability middleware standards available at the time, including RPC, ORB, and messaging-based platforms.

SOAP and Web Services

- SOAP is a protocol for exchanging XML-based messages over a network, normally using HTTP.
- Initially SOAP was intended as a platform independent protocol that could be used over the Web to allow interoperability with various types of middleware, including CORBA, EJB, JMS, etc.
- The introduction of SOAP led to a popular new type 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). WSDL-based services are often accessed using standard higher-level protocols, such as SOAP over HTTP.

Web Services (2)

- The simplicity of Web Services complements earlier, more complex, middleware technologies such as EJB and CORBA.
- When used for fine-grained distributed resource access, the performance of Web Services is often several orders of magnitude slower than DOC middleware due to its use of plain-text protocols such as XML over HTTP.
- 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.

Web Services (3)

- 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 describe Web Service interfaces abstractly, 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.

Limitations of Middleware

- Despite the many benefits of middleware, all of these technologies are primarily just "messengers" between elements in distributed applications.
  - Sometimes the messages just cannot be delivered despite heroic efforts from the middleware.
  - Distributed applications must still 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.
- Middleware 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 location of other components.