Distributed Software Development

Distributed Computing Architectures

Chris Brooks

Department of Computer Science
University of San Francisco
Challenges in designing a distributed system include:

- How to place the components
- What is the relationship between the components
- What level of abstraction exists for the user
- What external requirements are placed on the system,

Different design requirements lead to different architectural models.
6-1: Design Issues

- Transparency
- Flexibility
- Dependability
- Performance
- Scalability
6 Architectural models describe how a system is constructed.

- Placement of components
- Placement of data
- Functional roles
- Communication patterns
Software architecture originally referred to layers providing services to higher-level or lower-level layers.
   ▶ TCP/IP is an example of this.

Layers provide varying levels of abstraction.

Platform layer
   ▶ Provides OS-level services to upper layers.
   ▶ Provides an implementation-dependent interface into system.
Middleware is a layer that sits between the platform and the application.

- **Purpose**: mask heterogeneity, provide a consistent programming model for application developers.

- Provides abstractions such as remote method invocation, communication between processes on different machines, replication, real-time data transmission.

- Examples: CORBA, RMI, DCOM
Middleware has made developing distributed systems much easier.

- Provides a generic, reusable set of services.
- Abstracts away from platform-specific issues.

Not a silver bullet; some applications may still need to be aware of the fact that they are in a distributed environment.
6-6: Message-passing paradigms

The simplest approach to communication between processes is via message passing.
- Request is sent, reply received.

- Advantages: simple, easy to implement, well-understood.

- Disadvantages: lack of transparency and flexibility
  - Design is centered around I/O
  - Typically, one component must explicitly know the location of other components.
The client-server model is probably the best-known distributed computing paradigm.

Two (or more) processes take on asymmetric roles
- The client sends requests for service to the server.
- the server responds to the request.

Event synchronization is easy: Server listens for requests, processes them in order, and returns responses.

HTTP, FTP, DHCP, SMTP, DNS, finger, etc etc.
6-8: Client-server model

6 Servers can also act as clients of other processes.
   △ Web servers request services of file servers, DNS

6 The terms ’client’ and ’server’ describe the roles that components in a system play.
6-9: Multiple Servers

6 An issue with the client-server model is the single point of failure.
   ▲ Lack of scalability
   ▲ Lack of reliability

6 A workaround for this is to provide multiple physical servers that all provide the same logical service.
Servers may partition the data or problem and each work on part of it.

- The Web, distributed file systems are examples of this.
- This provides scalability, some reliability.

Alternatively, several servers may provide identical services via replication.

- For example, www.google.com is not a single machine, but a server farm.
- This provides scalability and reliable access to a single centralized resource.
6-11: Proxies and Caches

A weakness of distributed systems is the lack of data locality. This can lead to lags in performance as data is transmitted across the network.

This can be addressed through the use of caches.

- Browsers cache recently-used pages or objects.
- Networks cache recently accessed Web data within a system.
- Within the Internet backbone, companies such as Akamai cache recently-requested data.
6-12: Example: Squid

6 Squid is an example of a *proxy server*

6 Squid caches recent Internet traffic
   △ DNS lookups, FTP, HTTP traffic.

6 Squid looks up requested objects in its cache
   △ On a hit, the object is returned from the cache.
   △ On a miss, squid fetches the data either from the host or from a *sibling cache*.
   △ Squid caches are organized hierarchically; cache requests propagate up toward the Internet backbone.
In a *peer-to-peer* system, all processes play similar roles. 
- Equivalent capabilities and responsibilities.

All peers may interact with each other.

In theory, removing a server can improve efficiency by removing bottlenecks.

Downside: potentially more communication needed.

Also: discovery process necessary.
6-14: Peer-to-peer systems: Jabber

- Jabber is an open instant-messaging protocol.
- Allows any Jabber process to talk with any other (indirectly).
  - Messages are routed through a Jabber server.
  - This solves the discovery problem.
- Is this true P2P?
Jabber is an open instant-messaging protocol.

Allows any Jabber process to talk with any other (indirectly).
- Messages are routed through a Jabber server.
- This solves the discovery problem.

Is this true P2P?
- Maybe.

This sort of architecture is sometimes called *hierarchical P2P*.
So far, the systems we’ve looked at focus on exchanging data.
An alternative approach is to instead distribute code.
  △ Applets, ActiveX are well-known examples
Why might mobile code be a good thing?
Why might mobile code be a good thing?
- Better interactivity
- Faster for large data sets.
- Data security.
- Ability to provide customized, extensible services.

Downside: security risks.
The logical extension of mobile code are *mobile agents*

- Processes that can move to a machine, perform a computation, save state, and move to another machine.

Could be used to:
- Process large or secure data sets
- Reduce bandwidth usage
- Install or reconfigure software on remote machines.
- Useful in cases where network connectivity is intermittent.
6-19: Mobile agents

6 Serious security concerns remain.

6 This was a big research area a few years ago (Telescript, Aglets, Agent Tcl).

6 Lack of a 'killer app' and persistent security and interoperability issues have kept it from being widely used.

6 Most people have moved back towards data transfer.
A network computer is a low-end machine that maintains a minimal OS.

- No disk, or a small disk

Files and applications are delivered on-demand from a central server.

When the computer boots up, it requests a filesystem and applications.

Reduces maintenance, provides a single updating point.

Allows institutions to buy smaller, cheaper components. (in theory)
6-21: Network Computers

6 This was Sun and Oracle’s vision in the mid-90s.
6 Java would be used as the language to deliver applications on demand to clients.
6 To date, this hasn’t worked out.
   △ Cheap hardware
   △ Software issues
   △ Bandwidth problems
6 Interestingly, Google may be resurrecting this idea.
   △ This was a part of Netscape’s original vision.
   △ Programming at the browser level.
   △ Key apps (mail, photos, desktop search) available via the Web.
6-22: Thin Clients

6 A *thin client* is similar to a network computer.

6 Process runs remotely, but is displayed locally.

6 X-windows, VNC are well-known versions.
   △ Advantage: clients are cheap and low-cost
   △ Disadvantage: latency in rendering, scalability.

6 Cheap component costs have made thin clients less attractive.
This is probably the new 'killer app' of distributed computing.

Wireless devices, such as PDAs, phones, digital cameras, wearable computers, laptops all spontaneously discovering and communicating with one another.

Wide variety of ranges
- GSM: many kilometers, Bluetooth: a few meters

And bandwidths
- GSM: 13Kbps, 802.11g: 6.7Mbps.
Key features:
- Easy entry into a local network.
- Discovery of local services.

Issues:
- Users may have limited connectivity.
- Security and privacy.
There are lots of challenges in dealing with mobile devices:

- Discovery services
  - Registration of services and lookup of services.
- Routing of messages
- Appropriate delivery of content
- Dealing with connection outages to a component.
A weakness of the client-server and peer-to-peer models is that they focus explicitly on sending messages between processes.

Users typically know that they are dealing with a system that spans multiple machines.

In constructing software, a more programmatic interface may be easier to work with.

This leads to the *remote procedure call* paradigm.

A middleware layer manages all communication between client and server; from the application’s point of view, computation appears to happen locally.
In a remote procedure call, the requesting process *marshals* the parameters of the procedure call.

This data is sent to the server, where the data is unpacked and the procedure evaluated.

The return value (if any) is marshaled and returned to the requester.
The natural extension to RPC is an object-oriented approach.

Objects provide access to their methods via a network interface.

This can make the system more dynamic

- New objects, methods or interfaces can be added as required.
- As opposed to statically-allocated servers.
One well-known implementation of distributed objects is Java’s Remote Method Invocation mechanism.

An RMI server registers objects with an object registry.

The RMI client then retrieves an object reference via the registry.

Methods are evaluated exactly as if the object was stored locally.

Objects can either be:
- Remote: object remains on the server and is referenced remotely.
- Serializable: object is transferred to the client and manipulated locally.
An alternative approach to RPC is to use an object request broker (ORB)

- The ORB acts as an intermediary and passes method calls on to an object that services them.

RMI achieves this through the registry.

Prevents clients from having to know the location of the server.

CORBA is a language-independent implementation of ORBs.

COM, DCOM and JavaBeans provide similar effects.
An extension of distributed objects is network (or web) services.

- Service providers register their services with a registry.
  - For example, UDDI
- Requesting processes look up services using the registry.
- SOAP and .NET provide support for Web Services.
6-32: Groupware

6 Groupware is designed for *computer-supported collaborative work*

6 Multiple users want to participate in a collaborative session.
   ▲ Whiteboards, video conferencing, NetMeeting

6 Challenges: provide each user with a consistent picture of how the shared objects change.
So how does one choose the appropriate architecture?

- Abstraction vs Overhead.
  - Client-server has low abstraction, low overhead.
  - Distributed objects, groupware have high abstraction, high overhead.

- More overhead means more resources, more running time.

- More abstraction makes complex applications easier to build.
Scalability

- This can refer to system performance as the number of components increases.
- Also to the level of programmer complexity as the number of components increases.
- For example, RMI and CORBA can manage large number of objects transparently to the user.
Cross-platform support

- RMI only works with Java, COM only works with Windows.
- CORBA, SOAP are cross-platform
- Most message-passing systems are platform independent.
Other issues:
- Stability
- Maturity
- Reliability
- Fault tolerance
- Availability of developer tools
6-37: Summary

- Message-passing paradigms
  - Client-server
  - P2P
  - Mobile code/mobile agents
  - Mobile devices

- Procedure-based paradigms
  - RPC
  - Distributed objects
  - ORBs
  - Web services