

Distributed Software Development

Principles of Distributed Computing

Chris Brooks

Department of Computer Science
University of San Francisco

Department of Computer Science — University of San Francisco — p. 1/??

Introduction

- 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.

Department of Computer Science — University of San Francisco — p. 2/??

Design Requirements

- Some potential design requirements include:
 - Transparency
 - Flexibility
 - Dependability
 - Performance
 - Scalability

Department of Computer Science — University of San Francisco

Architectural Models

- Architectural models describe how a system is constructed.
 - Placement of components
 - Placement of data
 - Functional roles
 - Communication patterns

Department of Computer Science — University of San Francisco — p. 4/??

Software Layers

- 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.

Department of Computer Science — University of San Francisco — p. 5/??

Middleware

- 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

Department of Computer Science — University of San Francisco

Middleware

- 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.

Message-passing paradigms

- One question in designing a distributed system is how components will communicate.
- 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.

Client-server model

- Another question in designing a distributed system is the relationship between 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.

Client-server model

- Servers can also act as clients of other processes.
 - Web servers request services of file servers, DNS
- The terms 'client' and 'server' describe the *roles* that components in a system play.

Multiple Servers

- An issue with the client-server model is the single point of failure.
 - Lack of scalability
 - Lack of reliability
- A workaround for this is to provide multiple *physical* servers that all provide the same *logical* service.

Multiple Servers

- 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.

Proxies and Caches

- A potential 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.

Example: Squid

- Squid is an example of a *proxy server*
- Squid caches recent Internet traffic
 - DNS lookups, FTP, HTTP traffic.
- 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.

Peer-to-peer systems

- An alternative distributed architecture is a *peer-to-peer system*.
- 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.

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?

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?
 - Maybe.
- This sort of architecture is sometimes called *hierarchical P2P*.

Mobile code

- 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?

Mobile code

- 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.

Mobile agents

- 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.

Mobile agents

- Serious security concerns remain.
- This was a big research area a few years ago (Telescript, Aglets, Agent Tcl).
- Lack of a 'killer app' and persistent security and interoperability issues have kept it from being widely used.
- Most people have moved back towards data transfer.

Network Computers

- 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)

Network Computers

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

Thin Clients

- A *thin client* is similar to a network computer.
- Process runs remotely, but is displayed locally.
- X-windows, VNC are well-known versions.
 - Advantage: clients are cheap and low-cost
 - Disadvantage: latency in rendering, scalability.
- Cheap component costs have made thin clients less attractive.

Spontaneous Networks

- 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.

Spontaneous Networks

- Key features:
 - Easy entry into a local network.
 - Discovery of local services.
- Issues:
 - Users may have limited connectivity.
 - Security and privacy.

Mobile Devices and Spontaneous Networks

- 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.

Remote Procedure Call Model

- 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.

RPC

- 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.
- Weaknesses:
 - Little location transparency
 - Not very dynamic - may be hard to compose services at runtime.

Distributed Objects

- 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.

RMI

- 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.

Object Brokers

- 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.

Network Services

- 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.

Groupware

- Yet another distributed computing model is groupware.
- Groupware is designed for *computer-supported collaborative work*
- Multiple users want to participate in a collaborative session.
- All users interact through shared objects.
 - Whiteboards, video conferencing, NetMeeting
- Challenges: provide each user with a consistent picture of how the shared objects change.

Design tradeoffs and requirements

- 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.

Design tradeoffs and requirements

- 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 numbers of objects transparently to the user.

Design tradeoffs and requirements

- 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.

Design tradeoffs and requirements

- Other issues:
 - Stability
 - Maturity
 - Reliability
 - Fault tolerance
 - Availability of developer tools

Summary

- Message-passing paradigms
 - Client-server
 - P2P
 - Mobile code/mobile agents
 - Mobile devices
- Procedure-based paradigms
 - RPC
 - Distributed objects
 - ORBs
 - Web services

Time in Distributed Systems

- In systems with multiple CPUs, the clocks are unlikely to have the exact same time.
 - Variations in manufacturing cause clock skew.
 - Misconfiguration may lead to inconsistencies
- Insight: often, it doesn't matter exactly what time an operation happens, but what order events occur in.
- (exception: hard real-time systems)

Logical clocks

- A logical clock is just a counter(or set of counters)
- What's important is that all processes in the system can use it to produce a consistent ordering.
- This lets all processes in the system construct a global view of the system state.

Global state

- Many interesting problems in distributed computing can be expressed in terms of determining whether some property P is satisfied.
 - Consensus
 - Deadlock
 - Termination
 - Load balancing
- The general version of this problem is called the *Global Predicate Evaluation Problem*

Synchronization

- For example, to detect deadlock, we construct a wait-for graph.
 - Edge from process p_i to p_j if p_i is waiting on a message from p_j .
- If this graph has a cycle, we have deadlock.
- How do we do this while the computation is running?
- We can't pause the computation and collect all the information in one place.

Distributed Computation

- We define a distributed computation to consist of a set of processes p_1, p_2, \dots, p_n .
- Unidirectional channels exist between each process for passing messages.
- We assume these channels are reliable, but not necessarily FIFO.
 - Messages may arrive out of order.
- Assume the communication channels are asynchronous
 - No shared clock
 - No bound on message delay

Events and time

- Most of the time, we don't necessarily care about the exact time when each event happens.
- Instead, we care about the order in which events happen on distributed machines.
- If we do care about time, then the problem becomes one of synchronizing about the global value of a clock.

Distributed Computation

- A process p_i consists of a series of events e_i^1, e_i^2, \dots
- There are three types of events:
 - Local events - no communication with other processes
 - Send events
 - Receive events
- a *local history* is a sequence of events e_i^1, e_i^2, \dots such that order is preserved.

Distributed Computation

- The *initial prefix* of a history containing the first k events is denoted: $h_i^k = e_i^1, e_i^2, \dots, e_i^k$
- $h_i^0 = \langle \rangle$
- The *Global history* of a computation is the union of all local histories.
 - $h_1 \cup h_2 \cup \dots \cup h_n$
- Notice that this doesn't say anything about order of events between processes.
- Since an asynchronous system implies that there is no global time frame between events, we need some other way to order events on different processes.

Cause and Effect

- Cause and effect can be used to produce a partial ordering.
- Local events are ordered by identifier.
- Send and receive events are ordered.
 - If p_1 sends a message m_1 to p_2 , $send(m_1)$ must occur before $receive(m_1)$.
 - Assume that messages are uniquely identified.
- If two events do not influence each other, even indirectly, we won't worry about their order.

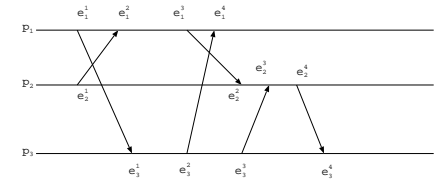
Happens before

- The *happens before* relation is denoted \rightarrow .
- Happens before is defined:
 - If e_i^k, e_i^l and $k < l$, then $e_i^k \rightarrow e_i^l$ (sequentially ordered events in the same process)
 - If $e_i = \text{send}(m)$ and $e_j = \text{receive}(m)$, then $e_i \rightarrow e_j$ (send must come before receive)
 - If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$ (transitivity)
- If $e \not\rightarrow e'$ and $e' \not\rightarrow e$, then we say that e and e' are concurrent. ($e \parallel e'$)
- These events are unrelated, and could occur in either order.

Happens before

- Happens before provides a partial ordering over the global history. (H, \rightarrow)
- We call this a distributed computation.
- A distributed computation can be represented with a space-time diagram.

Space-time diagram



Space-time diagram

- Arrows indicate messages sent between processes.
- Causal relation between events is easy to detect
- Is there a directed path between events?
- $e_1^1 \rightarrow e_3^4$
- $e_1^2 \parallel e_3^1$

Cuts

- A *cut* is a subset of the global history containing an initial prefix for each process.
 - Visually, the cut is a (possibly curved) line through the spacetime diagram.
- The *frontier* of the cut is the last state in each process.
- We'll use a cut to try to specify the global state of a computation at some point in time ...

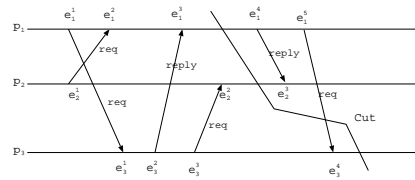
Monitoring a distributed computation

- So how can we use all of this to solve the GPE problem?
- We want to know what the global state of the system is at some point in time ...
- Solution 1: Create p_0 , the *monitor* process.
- The monitor sends each process an 'inquiry' message.
- Each process responds by sending its current local state σ_i
- Once all local states are received, these define the frontier of a cut.
- This cut is the global state. Will this work?

Monitoring a distributed computation

- Recall that the monitor is a part of the computation, and subject to the same communication rules.
- Let's say we want to use this to do deadlock detection.
- Each process can send a request and receive a reply.
- If there is a cycle in the resulting wait-for graph (WFG), we have a deadlock.

Monitoring a distributed computation



Monitoring a distributed computation

- Suppose p_1 receives the monitor message after e_1^3 , p_2 after e_2^2 , and p_3 after e_3^4 .
- The WFG has edges (1,3), (2,1), (3,2).
- The system is not really deadlocked, though; the monitor received an inconsistent picture.
- This is called a ghost deadlock.
- Problem: process p_3 's state reflects receiving a message that (according to the monitor) p_1 never sent.
- Active monitoring isn't going to work.

Consistent cuts

- We need to restrict our monitor to looking at *consistent cuts*
- A cut is consistent if, for all events e and e'
 - $(e \in C \text{ and } e' \rightarrow e) \Rightarrow e' \in C$
- In other words, we retain causal ordering and preserve the 'happens before' relation.
- A consistent cut produces a consistent global state.
- A consistent run is one such that, for all e, e' , if $e \rightarrow e'$, then e appears before e' in the run.
- This produces a series of consistent global states.
- We need an algorithm that produces consistent cuts.

Passive monitor

- Let's alter our monitor p_0 .
 - Rather than sending 'inquiry' messages, it listens.
 - Whenever any other process executes an event, it sends a message to p_0 .
 - p_0 would like to reconstruct a consistent run from these messages.
- How to prevent out-of-order messages?

Synchronous communication

- How could we solve this problem with synchronous communication and a global clock?
- Assume FIFO delivery, delays are bounded by δ
 - $send(i) \rightarrow send(j) \Rightarrow deliver(i) \rightarrow deliver(j)$
 - Receiver must buffer out-of-order messages.
- Each event e is stamped with the global clock: $RC(e)$.
- When a process notifies p_0 of event e , it includes $RC(e)$ as a timestamp.
- At time t , p_0 can process all messages with timestamps up to $t - \delta$ in increasing order.
- No earlier message can arrive after this point.

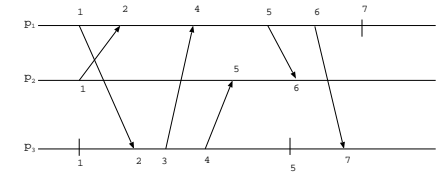
Why does this work?

- If we assume a delay of δ , at time t , all messages sent before $t - \delta$ have arrived.
- By processing them in increasing order, causality is preserved.
- $e \rightarrow e' \Rightarrow RC(e) < RC(e')$
- But we don't *have* a global clock!!

Logical clocks

- Each process maintains a logical clock. (LC).
- Maps events to natural numbers. $(0, 1, 2, 3, \dots)$.
- In the initial state, all LCs are 0.
- Each message m contains a timestamp indicating the logical clock of the sending process.
- After each event, the logical clock for a process is updated as follows:
 - $LC(e) = LC + 1$ if e is a local or send event.
 - $LC(e) = \max(LC, TS(m)) + 1$ if $e = receive(m)$.
- The LC is updated to be greater than both the previous clock and the timestamp.

Logical clock example



Logical clocks

- Notice that logical clock values are increasing with respect to causal precedence.
 - Whenever $e \rightarrow e'$, $LC(e) < LC(e')$
- The monitor can process messages according to their logical clocks to always have a consistent global state.
- Are we done?
 - Not quite: this delivery rule lacks *liveness*.
 - Without a bound on message delay, we can never be sure that we won't have another message with a lower logical clock.
 - We can't detect gaps in the clock sequence.
 - Example: we'll wait forever for the nonexistent message for e_3 from p_1 .

Adding liveness

- We can get liveness by:
 - Delivering messages in FIFO order, buffering out-of-order messages.
 - Deliver messages in increasing timestamp order.
 - Deliver a message m from process p_i after at least one message from all other processes having a greater timestamp has been buffered.

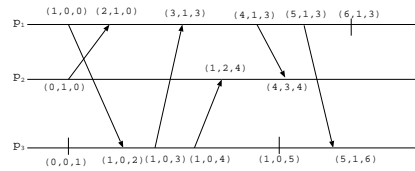
Causal delivery

- Recall that FIFO only refers to messages sent by the same process.
- *causal delivery* is a more general property which says that if $send(m_1) \rightarrow send(m_2)$, then $deliver(m_1) \rightarrow (m_2)$ when different processes are sending m_1 and m_2 .
- Logical clocks aren't enough to give us causal delivery.

Vector clocks

- Solution: keep a “logical clock” for each process.
- these are stored in a vector VC .
 - Assumes number of processes is known and fixed.
- Update rule:
 - $VC(e)[i] = VC[i] + 1$ for send and internal.
 - $VC(e) = \max(VC, TS(m))$ for receive; then $VC(e)[i] = VC[i] + 1$
- On receive, the vector clock takes the max on a component-by-component basis, then updates the local clock.

Vector Clock example



Vector clocks

- The monitor can then process events in order of ascending vector clocks.
- This ensures causality.
- Two clocks are inconsistent if $c_1[i] < c_2[i]$ and $c_1[j] > c_2[j]$
- If a cut contains no inconsistent clocks, it is consistent.
- Vector clocks allow the implementation of causal delivery.

Using cuts

- Cuts are useful if we need to rollback or restore a system.
- Consistent global cuts provide a set of consistent system states.
- Let us answer questions about the global properties of the system.
- Note that we still don't necessarily have the 'true' picture of the system.
 - Concurrent events may appear in an arbitrary order in a consistent run.
- We have enough information to reproduce all relevant ordering.

Summary

- Typically, we don't need exact time, just consistent ordering.
- We want to ensure that events that happen before others are processed first.
- The space-time diagram provides a useful picture of message propagation through a system.
- If we're only concerned about consistent observations, logical clocks work fine.
- Vector clocks are needed when we're concerned about propagation of information through a system. (causal delivery)