Distributed application

- application that executes a collection of protocols to coordinate the actions of multiple processes on a network, such that all components cooperate together to perform a single or small set of related tasks.
Advantages

• lots of advantages including the ability to connect remote users with remote resources in an open and scalable way.

• When we say open, we mean each component is continually open to interaction with other components.

• When we say scalable, we mean the system can easily be altered to accommodate changes in the number of users, resources and computing entities.
Challenges

• Probably the most difficult challenge is a distributed system must be able to continue operating correctly even when components fail

• When you design distributed systems, you have to say, "Failure happens all the time." So when you design, you design for failure. It is your number one concern.
Partial Failure

• If I send a message to you and then a network failure occurs, there are two possible outcomes.
  
  • One is that the message got to you, and then the network broke, and I just didn't get the response.
  
  • The other is the message never got to you because the network broke before it arrived.
  
• So if I never receive a response, how do I know which of those two results happened? I cannot determine that without eventually finding you. The network has to be repaired or you have to come up, because maybe what happened was not a network failure but you died.
Reliable Distributed Systems

- Fault-Tolerant
- Highly Available
- Recoverable
- Consistent
- Scalable
- Predictable Performance
- Secure
Failure Modes

- Halting failures
- Fail-stop
- Omission failures
- Network failures
- Network partition failure
- Timing failures
- Byzantine failures
8 fallacies

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.
Questions:
- How should I send/receive the request?
- How should I program it?
- How do I deal with requests failing?
Sending/receiving: Network layers

- Application layer: HTTP, SMTP, IM, FTP, NTP
- Transport layer: TCP, UDP, Multicast
- Network layer: IP (Internet protocol)
- Link layer: Physical transmission (e.g., Ethernet)
UDP vs. TCP vs. Multicast

- **UDP**: think of it as a unreliable postcard
  - sends a single packet, no notification if it gets lost
- **TCP**: sets up a reliable, two way communication
- **Multicast**: A broadcast of packets
  - generally UDP style packets
  - Generally over local networks
  - Wider more general uses not widely deployed
SMTP

-bash-3.00$ telnet localhost 25
Trying 127.0.0.1...
Connected to localhost.cs.umd.edu (127.0.0.1).
Escape character is '^[].'
220 weevil.cs.umd.edu ESMTP We are all Kosh.
HELO whitehouse.gov
250 weevil.cs.umd.edu Hello localhost [127.0.0.1], pleased to meet you
MAIL FROM: <obama@whitehouse.gov>
250 2.1.0 <obama@whitehouse.gov>... Sender ok
RCPT TO: <pugh@cs.umd.edu>
250 2.1.5 <pugh@cs.umd.edu>... Recipient ok
DATA
354 Enter mail, end with "." on a line by itself
Subject: Keep up the good work
From: obama@whitehouse.gov
To: pugh@cs.umd.edu

Great job Bill, keep up the good work.

0.
.
250 2.0.0 lAF1iaqa018678 Message accepted for delivery
QUIT
221 2.0.0 weevil.cs.umd.edu closing connection
Connection closed by foreign host.
Remote Procedure Calls

- We don’t have to program at a low-level; the language and libraries can help. In particular: RPC.
- A client can directly “invoke” a method that is executed on a different machine or process.
- Various protocols hide/abstract the issues involved with communicating the argument and results.
- Various incarnations, dating back to at least 1976.
Remote procedure call (RPC) is a technology that allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction. That is, the programmer would write essentially the same code whether the subroutine is local to the executing program, or remote. When the software in question is written using object-oriented principles, RPC may be referred to as remote invocation or remote method invocation.
An important difference between remote procedure calls and local calls is that remote calls can fail because of unpredictable network problems. Also, callers generally must deal with such failures without knowing whether the remote procedure was actually invoked.
• Java RMI (often used within multi-tier apps)
• Erlang RPC (we’ll see details later)
• Sun RPC (implements NFS)
• SOAP (often used for web apps)
• REST (implements “state” as URI)
Failures
Cluster supercomputers

- Many of the early cluster supercomputers were built from high end machines, all co-located in the same equipment room (perhaps even the same rack), using a special high end network switch.
- Many used a framework called MPI (Message Passing Interface) for sending messages.
- A hardware failure frequently killed the entire computation.
Modern distributed systems

- Built using COTS machines and networks
- may be distributed over several locations
  - want to make servers available near customers
  - but needs to be one unified distributed system, because the system needs a unified/consistent view of the world
- So many machines that you expect some to be failing all the time
Failures happen

- RPC can be seductively dangerous
  - program as though the service you are calling is local
- But isn’t, and there are many differences
- Although failure can happen in a local call, it is uniform
  - It effects both the caller and the callee
Carnegie Hall

- Suppose a client process requests a ticket-selling server to check for a seat in the orchestra section of Carnegie Hall.
- If it's available, the server records the request and the sale.
- But the request fails by timing out.
- Was the seat available and the sale recorded? Even if there is a backup server to which the request can be re-issued, there is a risk that the client will be sold two tickets, which is an expensive mistake in Carnegie Hall.
RPC mechanisms have some way of noting failure

• Using Java RMI, any method that can be invoked remotely must be declared to throw RemoteException
  • forces you to decide what you are going to do if some kind of remote failure occurs
  • catching and discarding the exception is probably not a good idea.
Handling failure

• Almost all of the frameworks developed in the 80’s, 90’s, and early 2000’s don’t tolerate failure well

• With MPI, job gets checkpointed at various points
  • checkpoint saves entire state of computation
  • Can run on 1200 machines at night, 500 machines during the day
  • Handles failures, interruptions and configuration changes
Hiding failure

- Hiding failure is better than handling it, if you can do it.
- Hiding is easier in a functional programming language
  - Repeating a computation has no impact
  - Can reexecute any computation that fails or is late
    - Efficiency is an issue
- Avoids Heisenbugs (except those in the non-functional implementation)
Details: RPC models
SOAP

- Simple Object Access Protocol
- Successor to XML-RPC
  - SOAP can be used for situations other than RPC, but RPC is the most common
- Sends XML messages over HTTP/HTTPS
  - tunnels through firewalls that allow HTTP but not other traffic
SOAP 1.1 request to search for "shrdlu winograd maclisp teletype"

<?xml version='1.0' encoding='UTF-8'?><soap11:Envelope xmlns="urn:GoogleSearch" xmlns:soap11="http://schemas.xmlsoap.org/soap/envelope/"><soap11:Body><doGoogleSearch><key>00000000000000000000000000000000</key><q>shrdlu winograd maclisp teletype</q><start>0</start><maxResults>10</maxResults><filter>true</filter><restrict/></doGoogleSearch><soap11:Body></soap11:Envelope>
<soap11:Envelope
   xmlns="urn:GoogleSearch"
   xmlns:google="urn:GoogleSearch"
   xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:soap11="http://schemas.xmlsoap.org/soap/envelope/">
  <soap11:Body>
    <doGoogleSearchResponse>
      <return>
        <documentFiltering>false</documentFiltering>
        <estimatedTotalResultsCount>3</estimatedTotalResultsCount>
        <directoryCategories soapenc:arrayType="google:DirectoryCategory[0]"></directoryCategories>
        <searchTime>0.194871</searchTime>
        <resultElements soapenc:arrayType="google:ResultElement[3]">
          <item>
            <cachedSize>12k</cachedSize>
            <hostName></hostName>
            <snippet> <b>... </b> on a simple dialog (via <b>teletype</b>) with a user, about a <b>... </b> <br> http://hci.stanford.edu/<b>winograd</b>/<b>shrdlu</b><br> . It is written in <b>MacLisp</b>, vintage <br> 1970, and to <b>... </b> <br> <directoryCategory>
              <specialEncoding></specialEncoding>
              <fullViewableName></fullViewableName>
            </directoryCategory>
            <relatedInformationPresent>true</relatedInformationPresent>
            <directoryTitle></directoryTitle>
            <summary></summary>
            <URL>http://hci.stanford.edu/cs147/examples/shrdlu/</URL>
            <title><b>SHRDLU</b></title>
          </item>
          <item>
            <cachedSize>12k</cachedSize>
            <hostName></hostName>
            <snippet> <b>... </b> on a simple dialog (via <b>teletype</b>) with a user, about a <b>... </b> <br> http://hci.stanford.edu/<b>winograd</b>/<b>shrdlu</b><br> . It is written in <b>MacLisp</b>, vintage <br> 1970, and to <b>... </b> <br> <directoryCategory>
              <specialEncoding></specialEncoding>
              <fullViewableName></fullViewableName>
            </directoryCategory>
            <relatedInformationPresent>true</relatedInformationPresent>
            <directoryTitle></directoryTitle>
            <summary></summary>
            <URL>http://hci.stanford.edu/cs147/examples/shrdlu/</URL>
            <title><b>SHRDLU</b></title>
          </item>
        </resultElements>
      </return>
    </doGoogleSearchResponse>
  </soap11:Body>
</soap11:Envelope>
REST: Representational State Transfer

- resources (sources of specific information)
- each of which can be referred to using a global identifier (a URI)
- applications communicate via a standardized interface (e.g. HTTP) and exchange representations of these resources
- a protocol that is
  - Client/Server
  - Stateless
  - Cacheable
  - Layered
REST uses HTTP methods

- **GET**: get a value, idempotent, cachable
- **DELETE**: remove resource
- **PUT**: overwrite resource
- **POST**: sometimes the same as PUT, sometimes means add after
Sample RPC application

- Operations:
  - getUser()
  - addUser()
  - removeUser()
  - updateUser()
  - getLocation()
  - addLocation()
  - removeLocation()
  - updateLocation()
  - listUsers()
  - listLocations()
  - findLocation()
  - findUser()

- Client code to delete user 001
  - exampleAppObject = new ExampleApp('example.com:1234')
  - exampleAppObject.removeUser('001')
Sample Rest Application

- Defines resources
  - http://example.com/users/
  - http://example.com/users/{user} (one for each user)
  - http://example.com/findUserForm
  - http://example.com/locations/
  - http://example.com/locations/{location} (one for each location)
  - http://example.com/findLocationForm
- Client code to access this application may look something like this:
  - userResource = new Resource('http://example.com/users/001')
  - userResource.delete()
Details: Distributed Programming using Java
Sending/receiving
Streams

- You should all know InputStream, OutputStream, Reader, Writer
- DataOutputStream adds writeInt, writeLong, writeDouble, writeUTF
- DataInputStream provides matching read methods
Object Streams

• ObjectOutputStream and ObjectInputStream allow objects to be sent
  – any serializable object

• Any object that implements Serializable can be serialized
  – if all of its fields are serializable
Object graphs

- Writing to an object output stream writes a graph of objects
  - writing a map sends all the elements
  - writing one node of a graph sends all connected nodes
- Reconstructed into an isomorphic graph
potential problems

• You write a class that receiver doesn’t have
  – error
• You write a different version of a class than the receiver has
  – maybe OK
Java Remote Method Invocation

- Provides remote proxies for objects
  - invoke method on local proxy/stub
  - method is invoked over the network, using object serialization
  - wait for response
  - response is also sent back via object serialization
Remote Objects and proxies

- Some objects are Remote objects
- can be referenced remotely
- Other JVM’s that might reference the remote object have proxies for the remote object
- Implement the interfaces that are defined as being accessible remotely
- not all methods of a remote object might be available remotely
Object marshaling

- RMI uses an extension of Object serialization
- Uses Object marshaling to send arguments to method call, and receive return value (or thrown exception)
- If Serializable/Externalizable object seen, send binary form of object
  - wrap it with a code base where the code can be downloaded from
- If Remote object is passed, send proxy for the object
Bootstrapping RMI

• You can use a Naming registry to bind a name to a remote object
• A naming registry listens to a particular port on a particular IP address
• Other JVM’s can perform lookups
  • give an IP address, a port, and a name
  • get back a remote proxy
remote code base

- RMI provides an extension to object serialization
- allows you to provide a URL from which your classes can be downloaded in received doesn’t have them
Hiding the details
The details are hard, so hide the details

- Lots of attempts to let people write code that doesn’t include any explicit parallelism, synchronization or communication
- the compiler/system just works out the details
Vectoring compilers

- Early supercomputers had vector units
  - could efficiently perform operations such as performing a point-wise addition to two vectors
- Lots of work to automatically vectorize FORTRAN code
  - very successful
  - but successful only because programmers learned to write code that could be vectorized
Parallelizing compilers

- Take a program and compile it for execution on a parallel computer
- perhaps a NUMA SMP
- or, much harder, a cluster
- Automatically extract course parallelism, optimize for locality, handle communication
- not as successful, particularly for clusters
Languages with array constructs

- Parallelizing compilers had a hard time because they had to reverse engineer code to figure out what it was doing;
- Sometimes, the code was pretty gnarly
- First improvements: explicit array constructs

Languages for parallel computing

• Add language constructs to describe how data should be distributed across machines, which loops should be run in parallel

• High Performance Fortran (HPF): academically groundbreaking, a commercial and practical failure

• Unified Parallel C (UPC): some success, very popular at NSA
Types of computations

• Most of these frameworks are designed for scientific/numeric computations

• weather forecasting, airflow simulation, nuclear bomb explosion simulation

• Blue Gene/L has 65,536 processors, running at 280 teraflops (280 trillion floating point operations per second)
Languages used in practice

- In practice, most of the very large scientific applications are programmed in MPI
- explicit message sends and receives
- When you are spending $100 million on a computer, you can chain a bunch of programmers to their terminals down in the basement
Various efforts to develop next generation languages

- Effects underway to develop next generation parallel programming languages for scientific applications
  - Fortress: from Sun Microsystems
  - X10: from IBM
  - Chapel: from Cray