Operating System Principles: Distributed Systems

CS 111

Operating Systems

Peter Reiher

Outline

- Introduction
- Distributed system paradigms
- Remote procedure calls
- Distributed synchronization and consensus
- Distributed system security

Goals of Distributed Systems

- Scalability and performance
  - Apps require more resources than one computer has
  - Grow system capacity /bandwidth to meet demand
- Improved reliability and availability
  - 24x7 service despite disk/computer/software failures
- Ease of use, with reduced operating expenses
  - Centralized management of all services and systems
  - Buy (better) services rather than computer equipment
- Enable new collaboration and business models
  - Collaborations that span system (or national) boundaries
  - A global free market for a wide range of new services

Transparency

- Ideally, a distributed system would be just like a single machine system
- But better
  - More resources
  - More reliable
  - Faster
- *Transparent* distributed systems look as much like single machine systems as possible

Deutsch's "Seven Fallacies of Network Computing"

1. The network is reliable
2. There is no latency (instant response time)
3. The available bandwidth is infinite
4. The network is secure
5. The topology of the network does not change
6. There is one administrator for the whole network
7. The cost of transporting additional data is zero

Bottom Line: true transparency is not achievable

Heterogeneity in Distributed Systems

- Distributed systems aren’t uniform
- Heterogeneous clients
  - Different instruction set architectures
  - Different operating systems and versions
- Heterogeneous servers
  - Different implementations
  - Offered by competing service providers
- Heterogeneous networks
  - Public and private
  - Managed by different orgs in different countries
- Another problem for achieving transparency
  - And sometimes correctness

Fundamental Building Blocks Change

- The old model:
  - Programs run in processes
  - Programs use APIs to access system resources
  - API services implemented by OS and libraries
- The new model:
  - Clients and servers run on nodes
  - Clients use APIs to access services
  - API services are exchanged via protocols
- Local is a (very important) special case

Changing Paradigms

- Network connectivity becomes “a given”
  - New applications assume/exploit connectivity
  - New distributed programming paradigms emerge
  - New functionality depends on network services
- Applications demand new kinds of services:
  - Location independent operations
  - Rendezvous between cooperating processes
  - WAN scale communication, synchronization

Distributed System Paradigms

- Parallel processing
  - Relying on special hardware
- Single system images
  - Make all the nodes look like one big computer
  - Somewhere between hard and impossible
- Loosely coupled systems
  - Work with difficulties as best as you can
  - Typical modern approach to distributed systems
- Cloud computing
  - A recent variant

Loosely Coupled Systems

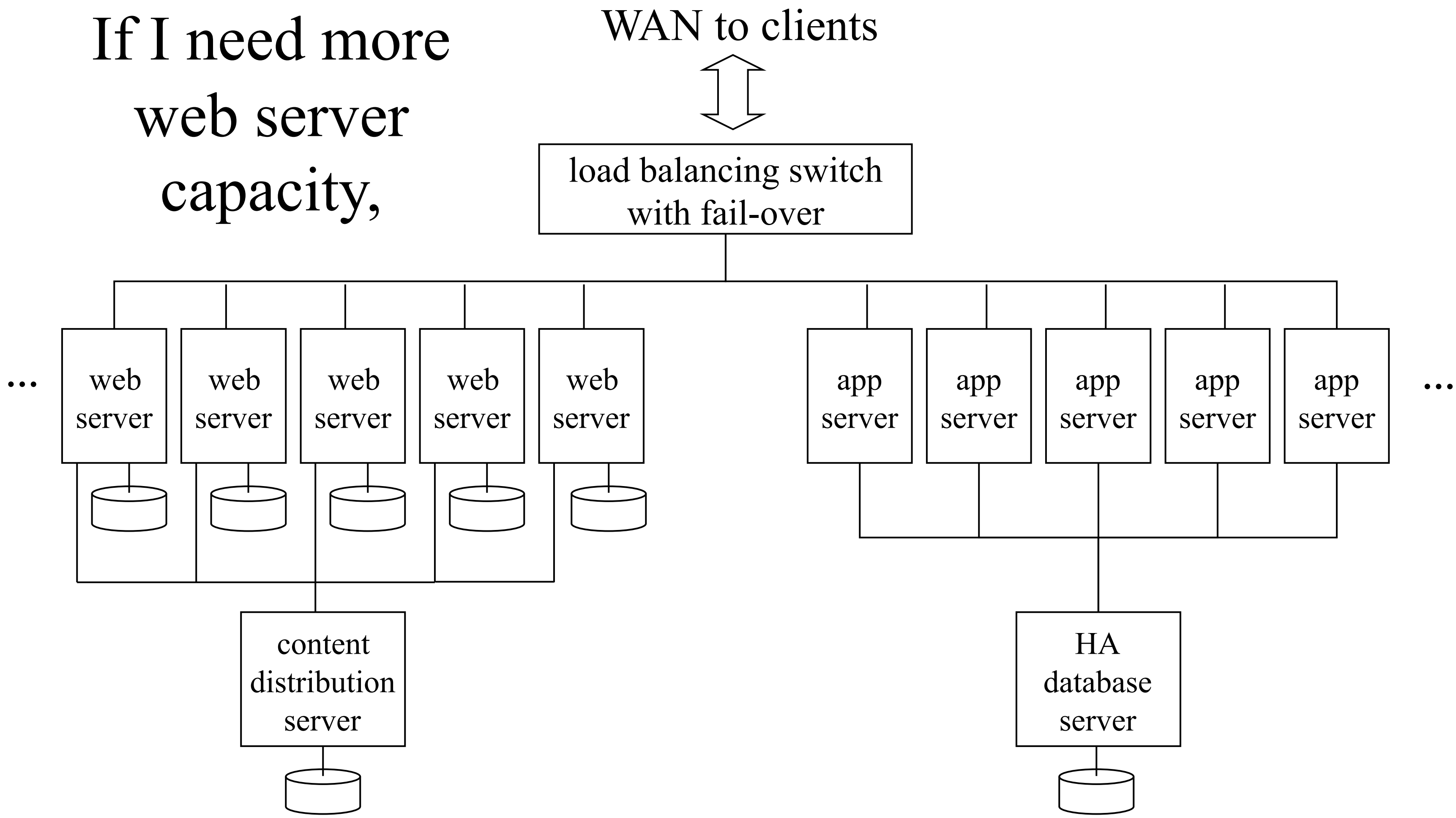
- Characterization:
  - A parallel group of independent computers
  - Serving similar but independent requests
  - Minimal coordination and cooperation required
- Motivation:
  - Scalability and price performance
  - Availability – if protocol permits stateless servers
  - Ease of management, reconfigurable capacity
- Examples:
  - Web servers, app servers, cloud computing

Horizontal Scalability

- Each node largely independent
- So you can add capacity just by adding a node "on the side"
- Scalability can be limited by network, instead of hardware or algorithms
  - Or, perhaps, by a load balancer
- Reliability is high
  - Failure of one of N nodes just reduces capacity

Horizontal Scalability Architecture

If I need more web server capacity,



Elements of Loosely Coupled Architecture

- Farm of independent servers
  - Servers run same software, serve different requests
  - May share a common back-end database
- Front-end switch
  - Distributes incoming requests among available servers
  - Can do both load balancing and fail-over
- Service protocol
  - Stateless servers and idempotent operations
  - Successive requests may be sent to different servers

Horizontally Scaled Performance

- Individual servers are very inexpensive
  - Blade servers may be only $100-$200 each
- Scalability is excellent
  - 100 servers deliver approximately 100x performance
- Service availability is excellent
  - Front-end automatically bypasses failed servers
  - Stateless servers and client retries fail-over easily
- The challenge is managing thousands of servers
  - Automated installation, global configuration services
  - Self monitoring, self-healing systems
  - Scaling limited by management, not HW or algorithms

Cloud Computing

- The most recent twist on distributed computing
- Set up a large number of machines all identically configured
- Connect them to a high speed LAN
  - And to the Internet
- Accept arbitrary jobs from remote users
- Run each job on one or more nodes
- Entire facility probably running mix of single machine and distributed jobs, simultaneously

Distributed Computing and Cloud Computing

- In one sense, these are orthogonal
- Each job submitted might or might not be distributed
- Many of the hard problems of the distributed jobs are the user's problem, not the system's
  - E.g., proper synchronization and locking
- But the cloud facility must make communications easy

What Runs in a Cloud?

- In principle, anything
- But general distributed computing is hard
- So much of the work is run using special tools
- These tools support particular kinds of parallel/ distributed processing
- Either embarrassingly parallel jobs
- Or those using a method like map-reduce
- Things where the user need not be a distributed systems expert

Embarrassingly Parallel Jobs

- Problems where it's really, really easy to parallelize them
- Probably because the data sets are easily divisible
- And exactly the same things are done on each piece
- So you just parcel them out among the nodes and let each go independently
- Everyone finishes at more or less same time

MapReduce

- Perhaps the most common cloud computing software tool/technique
- A method of dividing large problems into compartmentalized pieces
- Each of which can be performed on a separate node
- With an eventual combined set of results

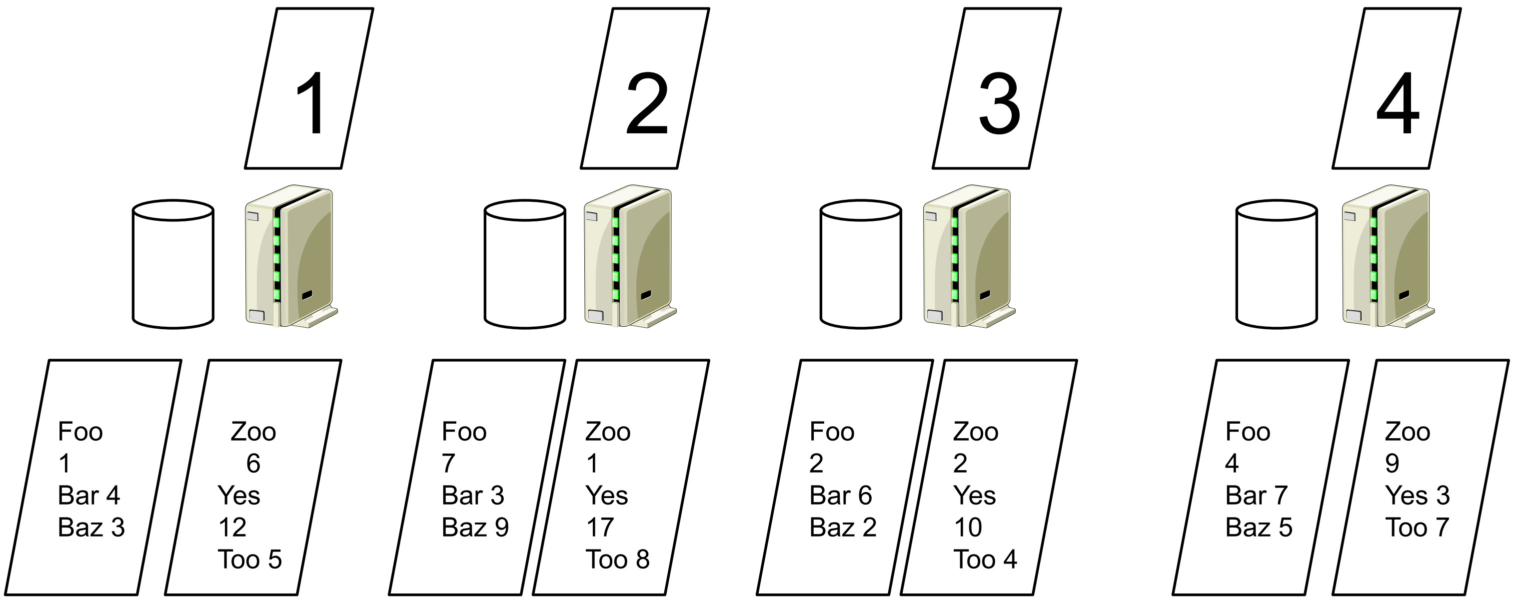
The Idea Behind MapReduce

- There is a single function you want to perform on a lot of data
  - Such as searching it for a string
- Divide the data into disjoint pieces
- Perform the function on each piece on a separate node (*map*)
- Combine the results to obtain output (*reduce*)

An Example

- We have 64 megabytes of text data
- Count how many times each word occurs in the text
- Divide it into 4 chunks of 16 Mbytes
- Assign each chunk to one processor
- Perform the map function of “count words” on each

The Example Continued

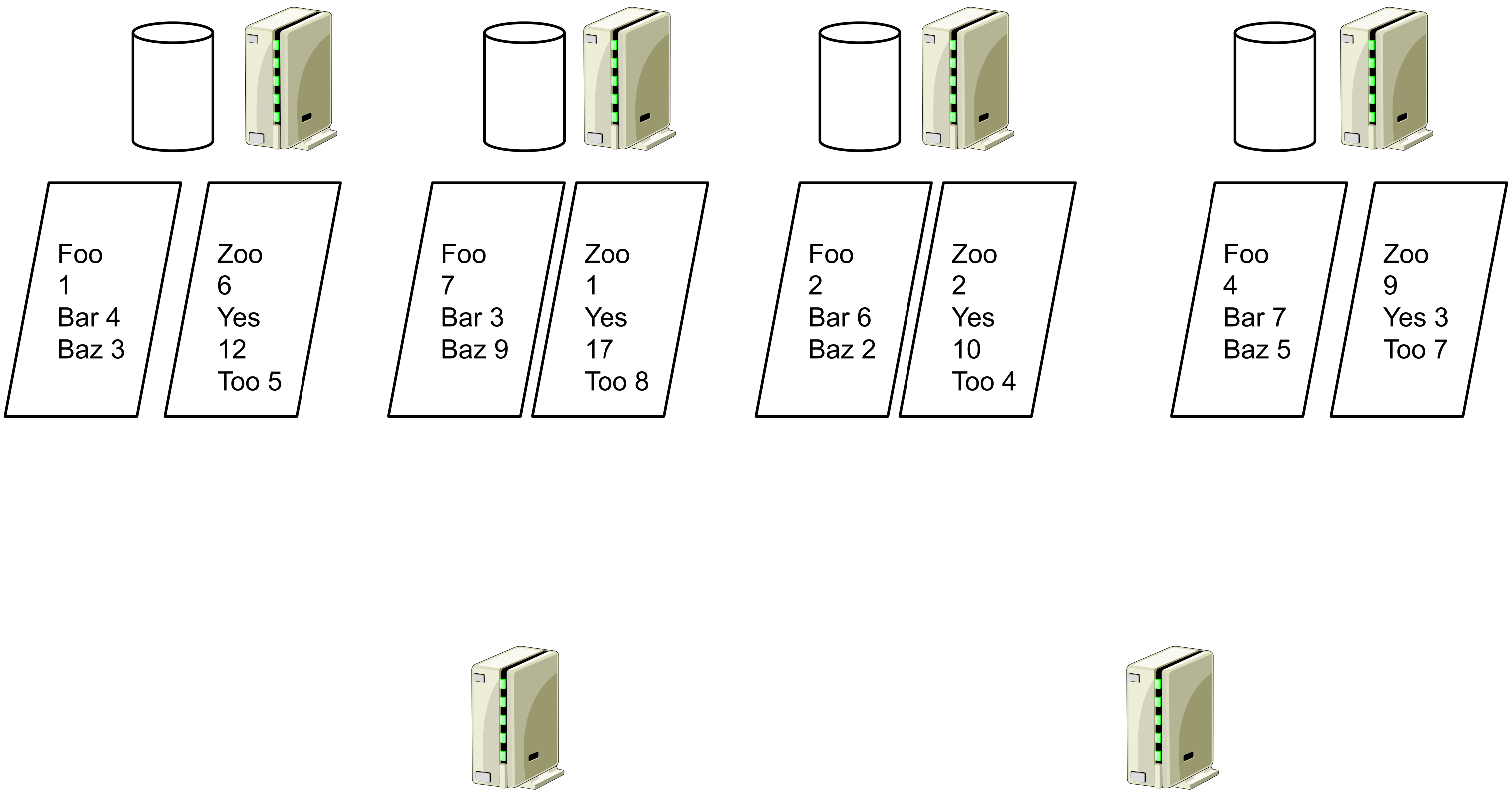


That's the map stage

On To Reduce

- We might have two more nodes assigned to doing the reduce operation
- They will each receive a share of data from a map node
- The reduce node performs a reduce operation to “combine” the shares
- Outputting its own result

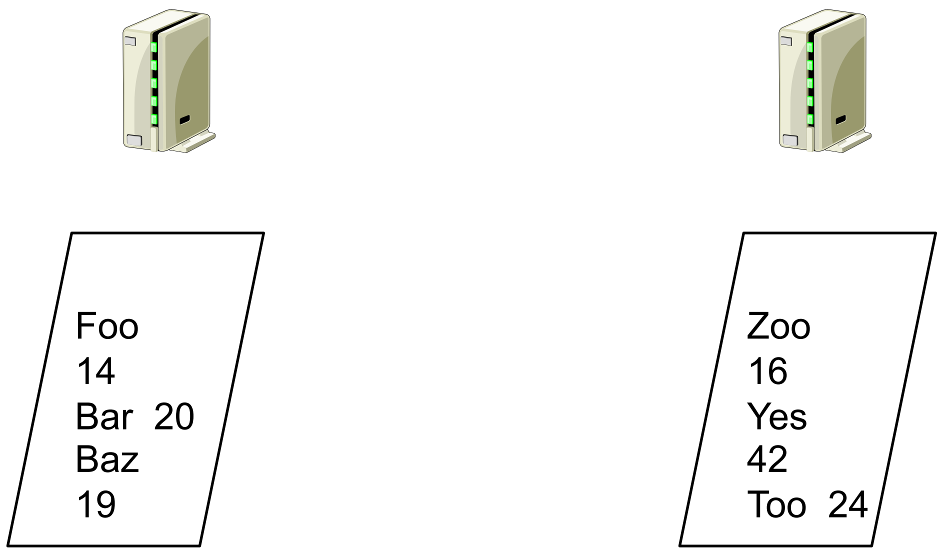
Continuing the Example



The Reduce Nodes Do Their Job

Write out the results to files

And MapReduce is done!



But I Wanted A Combined List

- No problem
- Run another (slightly different) MapReduce on the outputs
- Have one reduce node that combines everything

Synchronization in MapReduce

- Each map node produces an output file for each reduce node
- It is produced atomically
- The reduce node can't work on this data until the whole file is written
- Forcing a synchronization point between the map and reduce phases

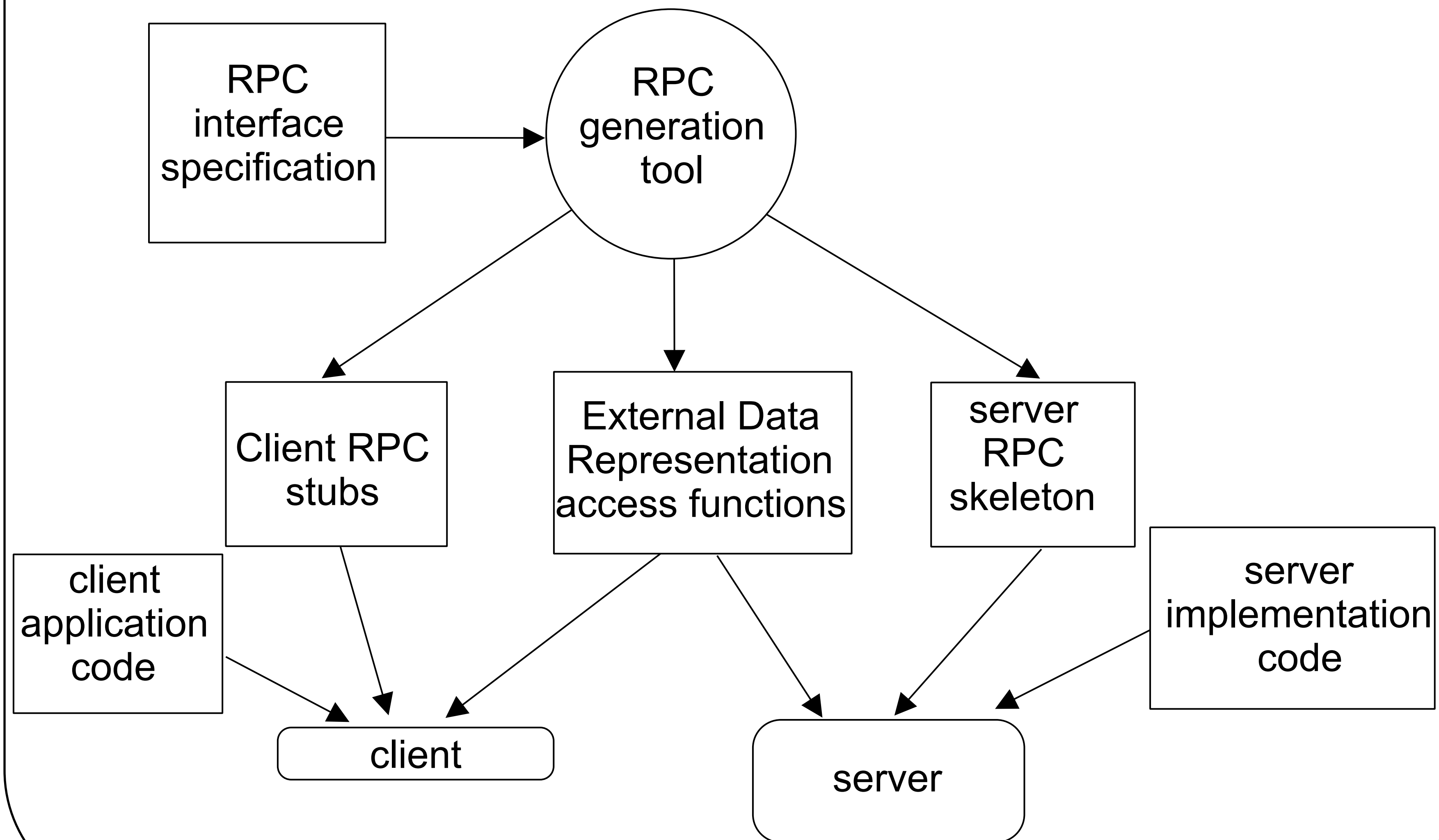
Remote Procedure Calls

- RPC, for short
- One way of building a distributed system
- Procedure calls are a fundamental paradigm
  - Primary unit of computation in most languages
  - Unit of information hiding in most methodologies
  - Primary level of interface specification
- A natural boundary between client and server
  - Turn procedure calls into message send/receives
- A few limitations
  - No implicit parameters/returns (e.g. global variables)
  - No call-by-reference parameters
  - Much slower than procedure calls (TANSTAAFL)

Remote Procedure Call Concepts

- Interface Specification
  - Methods, parameter types, return types
- eXternal Data Representation
  - Machine independent data-type representations
  - May have optimizations for like client/server
- Client stub
  - Client-side proxy for a method in the API
- Server stub (or skeleton)
  - Server-side recipient for API invocations

RPC Tool Chain



Key Features of RPC

- Client application links against local procedures
  - Calls local procedures, gets results
- All RPC implementation inside those procedures
- Client application does not know about RPC
  - Does not know about formats of messages
  - Does not worry about sends, timeouts, resents
  - Does not know about external data representation
- All of this is generated automatically by RPC tools
- The key to the tools is the interface specification

RPC Is Not a Complete Solution

- Requires client/server binding model
  - Expects to be given a live connection
- Threading model implementation
  - A single thread service requests one-at-a-time
  - Numerous one-per-request worker threads
- Limited failure handling
  - Client must arrange for timeout and recovery
- Higher level abstractions improve RPC
  - e.g. Microsoft DCOM, Java RMI, DRb, Pyro

Distributed Synchronization and Consensus

- Why is it hard to synchronize distributed systems?
- What tools do we use to synchronize them?
- How can a group of cooperating nodes agree on something?

What's Hard About Distributed Synchronization?

- Spatial separation
  - Different processes run on different systems
  - No shared memory for (atomic instruction) locks
  - They are controlled by different operating systems
- Temporal separation
  - Can't "totally order" spatially separated events
  - Before/simultaneous/after lose their meaning
- Independent modes of failure
  - One partner can die, while others continue

Leases – More Robust Locks

- Obtained from resource manager
  - Gives client exclusive right to update the file
  - Lease "cookie" must be passed to server on update
  - Lease can be released at end of critical section
- Only valid for a limited period of time
  - After which the lease cookie expires
    - Updates with stale cookies are not permitted
  - After which new leases can be granted
- Handles a wide range of failures
  - Process, client node, server node, network

Lock Breaking and Recovery

- Revoking an expired lease is fairly easy
  - Lease cookie includes a "good until" time
    - Based on server's clock
  - Any operation involving a "stale cookie" fails
- This makes it safe to issue a new lease
  - Old lease-holder can no longer access object
  - Was object left in a "reasonable" state?
- Object must be restored to last "good" state
  - Roll back to state prior to the aborted lease
  - Implement all-or-none transactions

Distributed Consensus

- Achieving simultaneous, unanimous agreement
  - Even in the presence of node & network failures
  - Required: agreement, termination, validity, integrity
  - Desired: bounded time
  - Provably impossible in fully general case
  - But can be done in useful special cases, or if some requirements are relaxed
- Consensus algorithms tend to be complex
  - And may take a long time to converge
- They tend to be used sparingly
  - E.g., use consensus to elect a leader
  - Who makes all subsequent decisions by fiat

Typical Consensus Algorithm

1. Each interested member broadcasts his nomination.
2. All parties evaluate the received proposals according to a fixed and well known rule.
3. After allowing a reasonable time for proposals, each voter acknowledges the best proposal it has seen.
4. If a proposal has a majority of the votes, the proposing member broadcasts a claim that the question has been resolved.
5. Each party that agrees with the winner's claim acknowledges the announced resolution.
6. Election is over when a quorum acknowledges the result.

Security for Distributed Systems

- Security is hard in single machines
- It’s even harder in distributed systems
- Why?

Ensuring Single Machine Security

- All key resources are kept inside of the OS
  - Protected by hardware (mode, memory management)
  - Processes cannot access them directly
- All users are authenticated to the OS
  - By a trusted agent that is (essentially) part of the OS
- All access control decisions are made by the OS
  - The only way to access resources is through the OS
  - We trust the OS to ensure privacy and proper sharing

Distributed Security Is Harder

- Your OS cannot guarantee privacy and integrity
  - Network transactions happen outside of the OS
- Authentication is harder
  - All possible agents may not be in local password file
- The wire connecting the user to the system is insecure
  - Eavesdropping, replays, man-in-the-middle attacks
- Even with honest partners, hard to coordinate distributed security
- The Internet is an open network for all
  - Many sites on the Internet try to serve all comers
  - Core Internet makes no judgments on what's acceptable
  - Even supposedly private systems may be on Internet

Goals of Network Security

- Secure conversations
  - Privacy: only you and your partner know what is said
  - Integrity: nobody can tamper with your messages
- Positive identification of both parties
  - Authentication of the identity of message sender
  - Assurance that a message is not a replay or forgery
  - Non-repudiation: he cannot claim “I didn't say that”
- Availability
  - The network and other nodes must be reachable when they need to be

Elements of Network Security

- Cryptography
  - Symmetric cryptography for protecting bulk transport of data
  - Public key cryptography primarily for authentication
  - Cryptographic hashes to detect message alterations
- Digital signatures and public key certificates
  - Powerful tools to authenticate a message's sender
- Filtering technologies
  - Firewalls and the like
  - To keep bad stuff from reaching our machines

Symmetric Encryption

- Simple fast algorithms
  - Encryption and decryption use the same key
  - Requires sender and receiver to both know the key
  - If you know who shares the key, you also get authentication
- Symmetric encryption provides privacy
  - In order to decrypt the data, you must know the key
- Symmetric encryption provides integrity
  - In order to generate false messages, you must know the key
- Symmetric encryption relies on key secrecy
  - Challenging to achieve in many circumstances
  - Large step between theoretical key secrecy and actual key secrecy in real systems

Tamper Detection: Cryptographic Hashes

- Check-sums often used to detect data corruption
  - Add up all bytes in a block, send sum along with data
  - Recipient adds up all the received bytes
  - If check-sums agree, the data is probably OK
  - Check-sum (parity, CRC, ECC) algorithms are weak
- Cryptographic hashes are very strong check-sums
  - Unique –two messages vanishingly unlikely to produce same hash
    - Particularly hard to find two messages with the same hash
  - One way – cannot infer original input from output
  - Well distributed – any change to input changes output

Using Cryptographic Hashes

- Start with a message you want to protect
- Compute a cryptographic hash for that message
  – E.g., using the Secure Hash Algorithm 3 (SHA-3)
- Transmit the hash securely
- Recipient does same computation on received text
  – If both hash results agree, the message is intact
  – If not, the message has been corrupted/ compromised

Secure Hash Transport

- Why must hash be transmitted securely?
  – Cryptographic hashes aren't keyed, so anyone can produce them (including a bad guy)
- How to transmit hash securely?
  – Typically encrypt it with symmetric cryptography
  – Unless secrecy required, cheaper than encrypting entire message
  – If you have a secure channel, could transmit it that way
    - But if you have secure channel, why not use it for everything?

Public Key Cryptography

- Uses two keys instead of one
- A secret key known only to the owner encrypts
- The public key known to everyone (potentially) decrypts
- Or you can reverse the keys and operations
  - With different effects
- The two keys are related by mathematical properties
  - But must be hard to derive from each other

Practical Use of PK

- Public key cryptography algorithms are computationally expensive
  - 10x to 100x as expensive as symmetric ones
- We use PK only when we can’t use symmetric cryptography
- When is that?
  - Typically to communicate to someone we don’t share a symmetric key with
  - We can share a new symmetric key using PK (*session key*)
  - Not very expensive, since the symmetric key is small

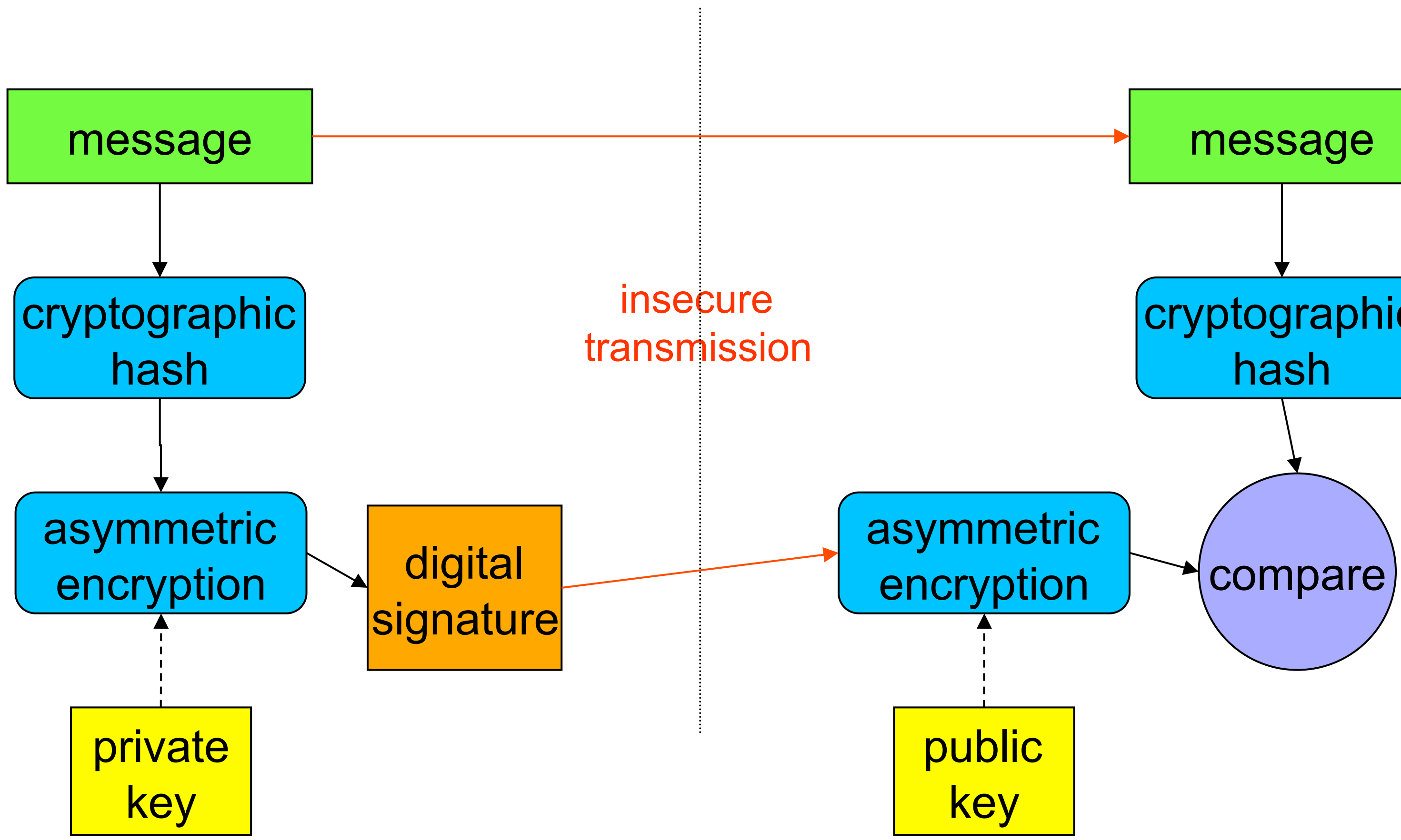
A Principle of Key Use

- Both symmetric and PK cryptography rely on a secret key for their properties
- The more you use one key, the less secure
  - The key stays around in various places longer
  - There are more opportunities for an attacker to get it
  - There is more incentive for attacker to get it
  - Brute force attacks may eventually succeed
- Therefore:
  - Use a given key as little as possible
  - Change them often
  - Within the limits of practicality and required performance

Putting It Together: Secure Socket Layer (SSL)

- A general solution for securing network communication
- Built on top of existing socket IPC
- Establishes secure link between two parties
  - Privacy – nobody can snoop on conversation
  - Integrity – nobody can generate fake messages
- Certificate-based authentication of server
  - Typically, but not necessarily
  - Client knows what server he is talking to
- Optional certificate-based authentication of client
  - If server requires authentication and non-repudiation
- PK used to distribute a symmetric session key
  - New key for each new socket
- Rest of data transport switches to symmetric crypto
  - Giving safety of public key and efficiency of symmetric

Digital Signatures



Digital Signatures

- Encrypting a message with private key signs it
  - Only you could have encrypted it, it must be from you
  - It has not been tampered with since you wrote it
- Encrypting everything with your private key is a bad idea
  - Asymmetric encryption is extremely slow
- If you only care about integrity, you don't need to encrypt it all
  - Compute a cryptographic hash of your message
  - Encrypt the cryptographic hash with your private key
  - Faster than encrypting whole message

Signed Load Modules

- How do we know we can trust a program?
  - Is it really the new update to Windows, or actually evil code that will screw me?
  - Digital signatures can answer this question
- Designate a certification authority
  - Perhaps the OS manufacturer (Microsoft, Apple, ...)
- They verify the reliability of the software
  - By code review, by testing, etc.
  - They sign a certified module with their private key
- We can verify signature with their public key
  - Proves the module was certified by them
  - Proves the module has not been tampered with

An Important Public Key Issue

- If I have a public key
  - I can authenticate received messages
  - I know they were sent by the owner of the private key
- But how can I be sure who that person is?
  - How do I know that this is really my bank's public key?
  - Could some swindler have sent me his key instead?
- I can get Microsoft’s public key when I first buy their OS
  - So I can verify their load modules and updates
  - But how to handle the more general case?
- I would like a certificate of authenticity
  - Guaranteeing who the real owner of a public key is

What Is a PK Certificate?

- Essentially a data structure
- Containing an identity and a matching public key
  - And perhaps other information
- Also containing a digital signature of those items
- Signature usually signed by someone I trust
  - And whose public key I already have

Using Public Key Certificates

- If I know public key of the authority who signed it
  - I can validate the signature is correct
  - I can tell the certificate has not been tampered with
- If I trust the authority who signed the certificate
  - I can trust they authenticated the certificate owner
  - E.g., we trust drivers licenses and passports
- But first I must know and trust signing authority
  - Which really means I know and trust their public key

A Chicken and Egg Problem

- I can learn the public key of a new partner using his certificate
- But to use his certificate, I need the public key of whoever signed it
- So how do I get <u>that</u> public key?
- Ultimately, *out of band*
- Which means through some other means
- Commonly by having the key in a trusted program, like a web browser
- Or hand delivered (as in project 4)

Conclusion

- Distributed systems offer us much greater power than one machine can provide
- They do so at costs of complexity and security risk
- We handle the complexity by using distributed systems in a few carefully defined ways
- We handle the security risk by proper use of cryptography and other tools