Distributed Systems — Introduction

Allan Clark

School of Informatics
University of Edinburgh

http://www.inf.ed.ac.uk/teaching/courses/ds
Autumn Term 2012

Distributed Systems — Definitions

- *"A system in which hardware or software components located at networked computers communicate and coordinate their actions only by message passing." — Coulouris*
- *"A system that consists of a collection of two or more independent computers which coordinate their processing through the exchange of synchronous or asynchronous message passing."*
- *"A distributed system is a collection of independent computers that appear to the users of the system as a single computer." — Tanenbaum*
- *" A distributed system is a collection of autonomous computers linked by a network with software designed to produce an integrated computing facility."*

Computer Networks vs. Distributed Systems

- Computer Network: the autonomous computers are explicitly visible — have to be explicitly addressed
- Distributed System: existence of multiple autonomous computers is transparent
- The study of computer networks is concerned with how to send messages between machines, whilst the study of distributed systems is how to use those networks to get stuff done.
- However,
  - many problems in common,
  - in some sense networks (or parts of them, e.g., name services) are also distributed systems, and
  - normally, every distributed system relies on services provided by a computer network.

Reasons for Distributed Systems

- Inherent distribution stemming from the application domain, e.g.
  - cash register and inventory systems for chain-stores
  - computer supported collaborative work
  - multi-player games
- Resource sharing is often a strong motivation
- Load distribution
  - amazon.com is not a single computer
  - these separate computers can be turned on-off for different demand profiles
- Critical failure tolerance, e.g. peer-to-peer networks
  - amazon.com isn't even located on a single site.
  - It is therefore resilient to (to some extent) earthquakes, power outages and more mailicious attacks

Consequences

These may be good, bad or somewhere in between:

- Software - how to design and manage it in a distributed system
- Dependency on the underlying network infrastructure
- Easy access to shared data raises security concerns
- Emergent behaviour, sometimes good, bad, or just fascinating

Consequences

- Distributed systems are concurrent systems
  - This concept will come up again and again
  - Synchronization and coordination by message passing
  - Sharing of resources, as both a positive and a negative
  - Typical problems of concurrent systems
    - Deadlocks and Livelocks
    - Unreliable communication
- Absence of a global clock
  - Due to asynchronous message passing there are limits on the precision with which processes in a distributed system can synchronize their clocks

Consequences — continued

- Absence of a global state
  - In the general case, there is no single process in the distributed system that would have a knowledge of the current global state of the system
    - Due to concurrency and message passing communication
- Specific failure modes
  - Processes run autonomously, in isolation
    - Failures of individual processes may remain undetected
    - Individual processes may be unaware of failures in the system context

Emerged/emerging Distributed Systems

1. Commerce
2. Encyclopedias (more generally knowledge stores)
3. Publishing in general
4. Finance
5. Education
6. Science
7. Healthcare

Examples of Distributed Systems

Web Search

- Google's infrastructure is one of the world's largest installations of a distributed system. It must visit and index a ridiculously large volume of web content in a variety of formats and then index this content for speedy results.
- Any numbers I give would be out of date tomorrow and are in any case unimaginable
- 68 Billion pages, maybe
- Data centres around the world
- A distributed file system designed for very fast access to very large files

Examples of Distributed Systems

(Massively) Multiplayer Games

- A particular need for fast response times
- Propogation of events and maintenance of the universe (or global state).
- The consequences of failure are potentially not as bad for the users (though major loss of revenue for the vendors)
- Most commericial offerings depend upon large infrastructure whether that be centrally managed or more distributed
- But, we are seeing the emergence of peer-to-peer based architectures for online games, with each user contributing some resources
- As such online games can be seen as a testbed for distributed systems (as they have proven in the past)

Examples of Distributed Systems

Online Betting

- Clearly betting has moved from the high street to the Internet
- More importantly there are now examples of distributed "layers" or "bookmakers"
- Examples are `betfair.com` and `intrade.com`
- Traditionally a bookmaker (using a greybeard and mathematics) would "set" or "fix" the odds for each particular bet
- Distributed bookmakers allow anyone to "back" or "lay" any particular bet (or market) at any particular price
- For example I can offer odds that Stoke City will win the EPL this year at odds of 1 in 4
- Sadly it is unlikely that anyone will take up this offer.
- Odds emerge as a market outcome

Examples of Distributed Systems

Financial Markets

- On the forefront of distributed systems development
- Due to a need for real-time information from a multitude of sources
- Have a need to relay events to potentially large numbers of clients
- For this reason they have unsual underlying architectures
- Emergent behaviour can be undesirable here, e.g. flash crash 2:45

Examples of Distributed Systems

Financial Markets

- On the forefront of distributed systems development
- Due to a need for real-time information from a multitude of sources
- Have a need to relay events to potentially large numbers of clients
- For this reason they have unsual underlying architectures
- Emergent behaviour can be undesirable here, e.g. flash crash 2:45
  - Thursday 6th May 2010
  - Dow Jones industrial average plunged approximately 1000 points
  - This was about 9% at the time
  - The largest one day point decline ever
  - The losses were recovered within minutes
  - Nobody knows to this day what happened

Examples of Distributed Systems

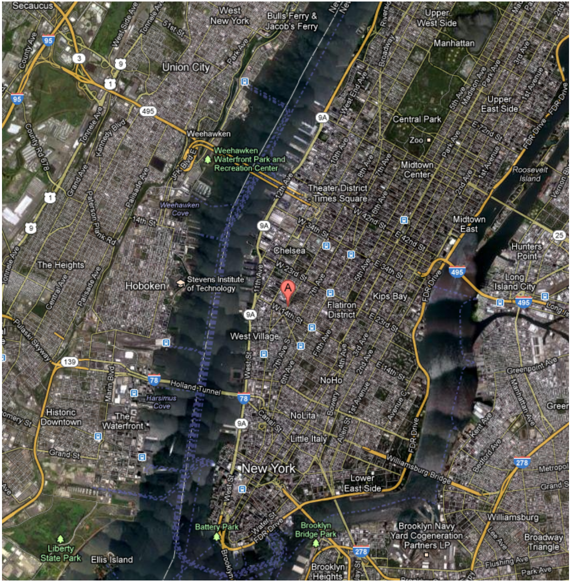
Examples of Distributed Systems

Google bought this place for 1.9 billion dollars

Examples of Distributed Systems

Building	Location	Height	Built	Price (USD)
The Shard	London, UK	310 metres	2012	3.9b
Antilla	Mumbai, India	173 metres	2007/10	2b
Taipei 101	Taipei, Taiwan	509 metres	2004	1.76b

Examples of Distributed Systems



Examples of Distributed Systems

Source Code Control

- Source code control is the endeavour to maintain a full history of changes to a project's source code, often by multiple authors
- Only the original source code and the changes (or diffs) are stored
- Concurrent updates are allowed when different parts of the code are changed, in which case the changes can be "merged"
- Where the same part is changed concurrently there is a "conflict" which must be resolved before operations may continue
- This allows for multiple versions of the source code such as a release and development branch
- Bugs can be tracked down to the change in which the bug was introduced, thereby elminating many possible causes

Examples of Distributed Systems

Source Code Control

- This has always tended to be a distributed system
- In the sense that there are multiple authors
- Traditionally there was a client-server based architecture
- One centralised server with the single repository
- Authors request:
  - New revisions
  - That their revisions be recorded in the global history

Examples of Distributed Systems

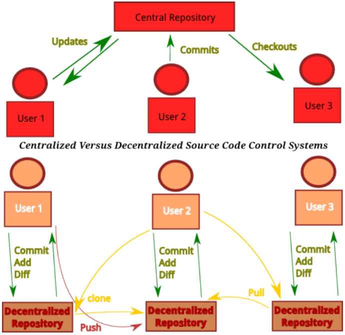
Source Code Control

- Recently (last decade or so) source code control systems have been decentralised or distributed
- Each contributor clones the entire history and has a local repository.
- Revisions can be sent and received between any two repositories
- There is greater fault tolerance, if the original centralised server fails a different one is simply declared the new master
- Merging can occur between smaller groups before commiting to a larger audience (of the master repository)

Examples of Distributed Systems

Source Code Control

- Centralised: cvs, subversion, ClearCase, Vault
- Distributed: git, mercurial, darcs, bazaar, bitkeeper



Challenges in Design of Distributed Systems

- 1. Heterogeneity
  - Hardware, Networks, Operating Systems, Programming Languages
  - Not just heterogeneity of implementation but sometimes of characteristics such as reliability or speed.
  - In a sense this much of this is a networking problem, that is the difficulty of sending messages around heterogeneous networks
  - But it does have implications, such as I have mentioned before for software versioning
  - Approaches generally use abstraction
    - Middleware (e.g., CORBA): transparency of network, hard- and software and programming language heterogeneity
    - Mobile Code (e.g., JAVA): transparency from hard-, software and programming language heterogeneity through virtual machine concept

Challenges in Design of Distributed Systems

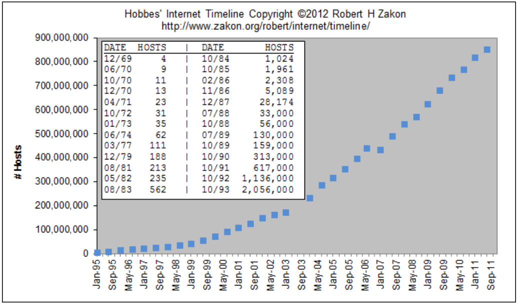
- 2. Openness
  - How open a distributed system is determines whether it can be extended domain, both size and functionality
  - Mostly determined by how well published are the interfaces which are used
  - Many web-services are being turned into mobile applications because they have well defined and published interfaces
  - An open system is less reliant on a particular vendor
- 3. Security,
  - has essentially three main components:
    1. Confidentiality — protection against access by unauthorised individuals
    2. Integrity — protection against alteration or corruption
    3. Availability — protection against loss of access whether circumstantial or a malicious denial of service attack
  - Security forms a later part of this course — but in summary, encryption only gets you part of the way there

Challenges in Design of Distributed Systems

- 4. Scalability
  - Does the system remain effective given expectable growth?
  - Expectable growth of physical resources and
  - Expectable growth of users
  - Avoiding Performance bottlenecks
    - Early Domain Name Lookup consisted of a single centrally hosted file
    - The "hosts.txt" file mapped names to numerical addresses
    - Client computers were required to periodically re-download this file from its known location (at SRI, now SRI International)
    - The "hosts.txt" file still exists on most operating systems today and can be used for much hilarity if you can access your friend's hosts.txt file
    - $ dig www.some-annoying-site.com ⇒ 173.194.67.103
    - 173.194.67.103 www.bbc.co.uk

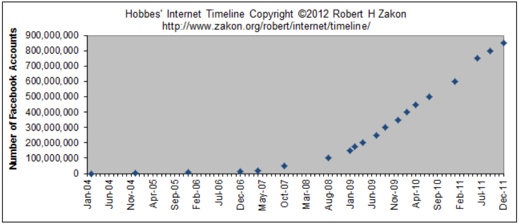
Challenges in Design of Distributed Systems

- 4. Scalability
  - After DNS was developed:
  - Some time in the late 1970s it was decided that 32 bit addresses would be enough, but they are currently running out.
  - IP addresses are in the process of switching from 32 bit addressing to 128 bit addressing but overcompensating could have been a serious performance issue



Challenges in Design of Distributed Systems

- 4. Scalability
  - Expectable growth is often non-obvious



Challenges in Design of Distributed Systems

- 5. Handling of failures
  - Detection (may be impossible)
  - Masking
    - retransmission
    - redundancy of data storage
    - generally not guaranteed in the worst case
  - Tolerance
    - exception handling (e.g., timeouts when waiting for a web resource)
  - Recovery
    - Can be especially tough, the failed process may have left some permanent data in an inconsistent state
  - Redundancy
    - redundant routes in network
    - replication of name tables in multiple domain name servers

Challenges in Design of Distributed Systems

- 6. Concurrency
  - Consistent scheduling of concurrent threads (so that dependencies are preserved, e.g., in concurrent transactions)
  - Avoidance of dead- and livelock problems
  - Actions are concurrent if A may happen before B and B may happen before A
  - Generally you hope for consistent results in either case
  - I will have more to speak about concurrency

Challenges in Design of Distributed Systems

- 7. Transparency: concealing the heterogeneous and distributed nature of the system so that it appears to the user like one system.
  - Transparency categories (according to ISO's Reference Model for ODP)
    - Access: access local and remote resources using identical operations e.g., network mapped drive
    - Location: access without knowledge of location of a resource e.g., URLs, email addresses
    - Concurrency: allow several processes to operate concurrently using shared resources in a consistent fashion
    - Replication: use replicated resource as if there was just one instance
    - Failure: allow programs to complete their task despite failures e.g., retransmit of email messages
    - Mobility: allow resources to move around
    - Performance: adaption of the system to varying load situations without the user noticing it
    - Scaling: allow system and applications to expand without need to change structure or application algorithms

Any Questions

Any Questions?

Distributed Systems — Fundamental Concepts

Allan Clark

School of Informatics
University of Edinburgh

http://www.inf.ed.ac.uk/teaching/courses/ds
Autumn Term 2012

Fundamental Concepts

- Distributed Systems are first and foremost complex software systems
- Architectural paradigms pertinent to distributed systems:
  - Layers
  - Client-Server

Layers

- The basic idea of a layered approach in general:
  - layer: A group of closely related and highly coherent functionalities
  - service: The functionality provided to the superior layer.

An Example Layer Approach

1. Physical transistors
2. Chip architecture; uses physical transistors and provides a set of (binary encoded) machine instructions for basic operations
3. Assembly code; uses binary codes to provide almost the same instructions in an alphabet incoding.
4. Systems programming language: compiler uses the assembly code to expose a high-level programming language such as C
5. Operating System (kernel): uses the systems programming language to provide a range of services to aid application programming
6. Application programming language: provides servies for the application programmer using the operating system and systems programming language

Layering in Distributed Systems

Typically:

1. Computer and Network
2. Platform: hardware and operating system providing access to network protocols
3. Middleware: Used to achieve transparency of heterogeniety at the platform level
4. Applications and services built on top of the middleware

Client-Server Architecture

- The Client-Server Architecture basic mode:
- Client: A process wishing to access some resource or perform operations on a different computer
- Server: Process which accepts requests from clients and processes those requests eventually providing a response
- The client is often referred to as the “active” player and the server the “passive” since it is the client which initiates communication.
- In common parlance a server is a machine but here it is a process
- In order to satisfy some request the server may become a client and make some request of a different server
- Where this is taken to an extreme we get “Peer Processes” which have largely the same functionality and do not describe a client-server architecture

Client-Server Architecture

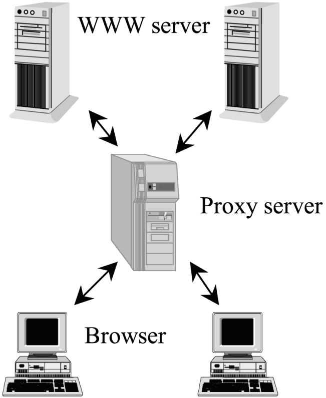
Variants – Multiple Servers

- Service provided by multiple servers
- Many commercial web services implemented by many different physical servers. This is so common now that it is almost the single server that is the variant.
- Motivation:
  - Peformance
  - Reliability
- Servers generally must maintain a replicated or distributed database

Client-Server Architecture

Variants – Proxy Servers

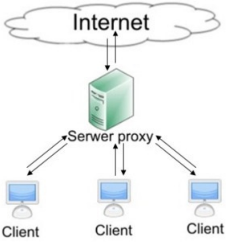
- Proxy server provides transparency of replication/distribution



Client-Server Architecture

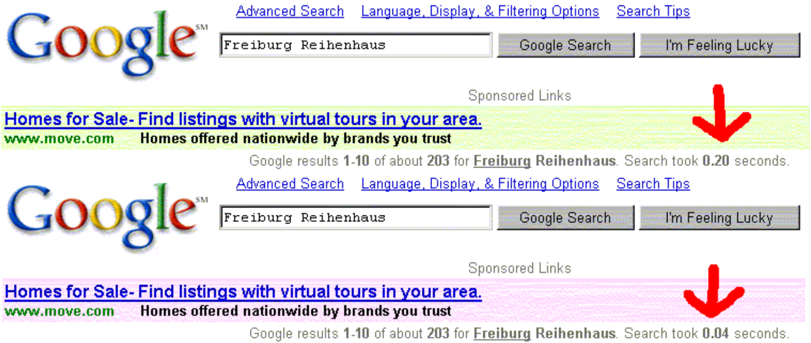
Variants – Proxy Servers

- Proxy server may maintain cache of responses to recent requests
- Requires that identical requests receive identical responses, often this means that the cache store is time bounded
- Frequently used in search engines



Client-Server Architecture

Variants – Proxy Servers



Client-Server Architecture

Further Client-Server Variants

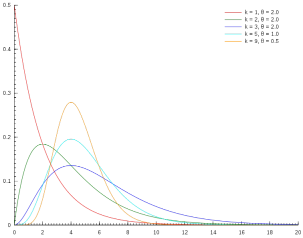
- Mobile Code
  - Code that is sent to the client
  - Java Applets, Flash etc.
- Mobile Agents — really a specific form of mobile code
- Thin Clients
  - Note so much a variant as an extreme example

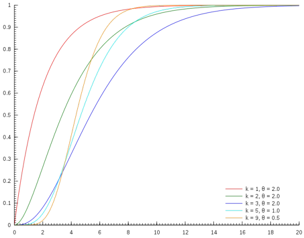
Client-Server Architecture

- Software Implications
  - Use of client-server has impact on the software architecture used
  - What kinds of requests and responses are allowed
  - What are the synchronisation mechanisms between client and server
  - Smaller shorter requests vs. Larger slower requests.

Client-Server Architecture

- Design Challenges
  - Quality of service
    - Performance: Response times





    - Performance: throughput
    - Performance: timeliness
    - Reliability: Server must obviously be generally available
    - Adaptability: For example to high and low demand
    - Dependability: Fault tolerance, not just the server but a client may be faulty (`isup.me`)
    - Security: The server is an obvious point to attack as well as the communication channels of any distributed system

Peer-to-Peer Architecture

- Client-Server approach scales poorly
- As the number of users grows so too do the demands on the centralised resources at the server
- In response Peer-to-Peer architectures arose from the realisation that the resources (computing, data and networking) owned by users of a service could be put to use to support that service
- This has a number of useful consequences but most obviously the shared resources available to users grows with the growth of new users.
- The distributed source code control systems described earlier could be described as peer-to-peer source code control.
- More to say on Peer-to-Peer distributed systems later

Fundamental Interaction Model

- Distributed System
  - Multiple processes
  - Connected by communication channels
- Distributed Algorithm
  - Steps to be taken by each process
  - Defines the communication between processes
  - Does not directly define the sequence of steps globally
- We create models to:
  - Make explicit all relevant assumptions about the distributed system we are modelling/designing
  - Make generalisations about what is possible given those assumptions, for example desirable properties such as no deadlock.
- Model aspects (may or may not be the same model):
  - Interaction model
  - Performance model
  - Failure model
  - Security model

Interaction Model

Synchronous distributed system

- time to execute each step of a computation within a process has known lower and upper bounds
- message delivery times are bound to a known value
- each process has a clock whose drift rate from real time is bounded by a known value

Asynchronous distributed system

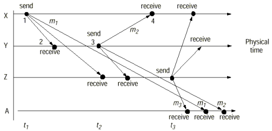
- no bound on process execution times
- no bound on message delivery times
- no bound on clock drift rate

Note

- synchronous distributed systems are easier to handle, but determining realistic bounds can be hard or impossible
- asynchronous distributed systems are more abstract and general: a distributed algorithm executing on one system is likely to also work on another one

Interaction Model

Event Ordering



- As we will see later, in a distributed system it is impossible for any process to have a view on the current global state of the system
- Possible to record timing information locally, and abstract from real time (logical clocks)
- event ordering rules:
  - if e_1 and e_2 happen in the same process and e_1 happens before e_2 then $e_1 \longrightarrow e_2$
  - if e_1 is the sending of a message m and e_2 is the receiving of the same message m then $e_1 \longrightarrow e_2$
  - Hence, $\longrightarrow$ describes a partial ordering relation on the set of events in the distributed system

Performance Model

Performance Characteristics of Communication Channels

- latency delay between sending and receipt of message
  - network access time (e.g. Ethernet transmission delay)
  - time for first bit to travel from sender's network interface to receiver's network interface.
- throughput: number of units (eg packets) delivered per unit of time
- bandwidth: amount of information transmitted per time unit
- delay jitter: variation in delay between different messages of the same type, (e.g., video frames)

Failures

Omission Failures

- process omission failures
  - detection with timeouts
  - crash is fail-stop if other processes can detect with certainty that process has crashed
- communication omission failures: message is not being delivered — dropping of messages
  - possible causes:
    - network transmission error
    - receiver incomming message buffer overflow

Arbitrary Failures

- process: omit intended processing steps or carry out unintended ones
- communication channel: corruption or duplication etc.

Failures

Class of Failure	Affects	Description
Fail-stop	Process	Process halts and remains halted. Other processes may detect this.
Crash	Process	Process halts and remains halted. Other processes may not detect this.
Omission	Channel	A message inserted in one outgoing buffer never arrives at the other end's incoming buffer
Send-omission	Process	A process completes a *send* but the message is never put in its outgoing buffer
Receive-omission	Process	A message is put in a process's incoming buffer but the process never receives it.
Arbitrary (Byzantine)	Process / Channel	Process/channel exhibits arbitrary behaviour: it may send/transmit arbitrary messages at arbitrary times.

Failures

Masking/Hiding Failures

- A service may mask an error by hiding it entirely or,
- converting it into a more acceptable type of error
- A reliable protocol can be built upon an unreliable protocol by requesting retransmission of dropped messages
- Message sequence numbers can be used to ensure no message is delivered twice, particularly when used with a guaranteed delivery protocol.
- Parity bits or checksums can be used to detect an error and thereby turn an arbitrary failure into an omission failure.

Security

- Two related problems:
  - We wish to make sure only the intended recipient(s) can receive a message
  - Additionally messages (for example invocation requests) should be authenticated so that we know from whom they originated
- These can be largely mitigated against with the use of modern cryptographic algorithms
- However their use incurs some cost which we may hope to minimise
- Denial of service
  - generating debilitating network or server load so that services become the equivalent of unavailable
- Mobile Code:
  - requires executability priviledges on target machine
  - code may be malicious

Summary — Fundamental Interaction Model

- We have looked at architectural models: Client-Server and Peer-to-Peer.
- These are complemented by fundamental models to aid in reasoning about behaviour:
  - Interaction model
    - Classifies models as synchronous or asynchronous
    - Identify basic components from which distributed systems are built
  - Performance model — sometimes combined with interaction
    - concerned with the efficiency of completing global tasks
    - can be used to compare approaches
  - Failure model
    - Used to analyse how resilient a distributed system is to failures
    - Can be used to classify what can go wrong and how that affects the system including other peers
  - Security model
    - Allows us to keep the costs associated with security measures to a minimum

Networking — Types of Networks

1. Personal Area Networks — generally wireless e.g. bluetooth
2. Local Area Networks
3. Wide Area Networks
4. Wireless local area networks
5. Wireless Wide Area Networks (3G and now 4G)
6. Internetworks — comprising of potentially many kinds of networks linked together by routers and gateways. The Internet being the most obvious example.

Getting Messages to Destinations — Switching

Broadcasting

- Broadcasting is one way of getting the message to its intended recipient
- Simply send it to everyone and have all the receivers filter their messages to receive only the ones intended for them
- A bit like *spam*
- Local area networks are commonly built on this technology (in particular Ethernet is)
- Wireless networks are necessarily broadcast networks
- Cryptography can be used to force filtering on the receivers
- Broadcasting does not scale well with the number of senders

Getting Messages to Destinations — Switching

Broadcasting

Photo copyright Kwozie flickr user

Getting Messages to Destinations — Switching

Circuit Switching

- Was used for the telephone system

- Very rarely used for computer networks
- Circuit switching does have some advantages including greater efficiency once the circuit has been initiated
- Long distance networks required several switches in-between end-points.
- However it has several disadvantages including:
  - low adaptability to changing traffic
  - low adaptability to loss of communication channel

Getting Messages to Destinations — Switching

Packet Switching or Store and Forward

- When networks were built with computers so came the possibility to do some processing at each node along the path
- Packet switching is an example of what is called a “store and forward” network
- Each packet is treated separately at each node, it is first stored and then a decision is made about how and where to forward it
- The postal system is an example of a store and forward network, using packet switching

Getting Messages to Destinations — Switching

Packet Switching or Store and Forward

- Packet Switching can adapt to changing network conditions
- Including the loss of a communication channel
- They do incur some disadvantages, in particular packages may arrive out of order
- Packet lengths are restricted in order to:
  - Each computer in the network can allocate sufficient storage to hold the largest possible incoming packet
  - Avoid undue delays in waiting for communication channels to become free (essentially the same reason you don't send an unsegmented thesis to the printer)
- "frame relay" is a compromise between circuit and package switching.

Protocols

- Protocols enable communication between computers
- A protocol specifies:
  1. The sequence of messages that must be exchanged e.g. message - acknowledgement
  2. The format of the data in the messages
- A key idea is that of protocol layering
- Software at the sender and receiver is arranged in modules representing each layer
- Conceptually the software at layer N is communicating with the other computer at layer N
- But in reality is invoking and reacting to the layer below
- In particular one can build a reliable communication layer atop an unreliable communication layer.

Routing

- Routing is required in networks larger than a LAN
- Adaptive routing allows for changes in network traffic and connectivity
- A routing algorithm is implemented by a program in the network layer at each node
- It must:
    1. Determine the route taken by each packet as it travels through the network. A circuit switched network will set up a route for all subsequent packets but a packet switched network will perform the same steps for each packet. The routing algorithm in a packet switched network must therefore be simple and efficient.
    2. Dynamically update its knowledge of the network so as to better route subsequent packets/circuits
- Internet routing is essentially path finding in graphs.

Routing — Example Algorithm

Router Information Protocol (RIP)

1. Maintain a routing table:

Dest	Link	Cost
1	local	0
2	2	1
8	2	4

2. Periodically — and whenever the local routing table changes — send table in summary form to all accessible links
3. If a routing table packet is received from a neighbouring router update your own table accordingly:
   - If there is a new destination add that row to your table
   - If there is a lower cost route to an existing node update the appropriate row
   - If the table was received on link N replace all differing rows with N as the link
4. If a link $\mathcal{L}$ becomes unavailable set cost to ∞ for all entries with $\mathcal{L}$. Since the routing table has changed, send it to all accessible links.

Routing — Example Algorithm

Router Information Protocol (RIP)

Dest	Link	Cost
Allan	local	0
Bob	lBob	1
Alice	lAlice	1
Susan	lAlice	5

Routing — Example Algorithm

Router Information Protocol (RIP)

Bob sends me a new table and it has information about a node I hadn't seen before "Harry" at a cost of 8

Dest	Link	Cost
Allan	local	0
Bob	lBob	1
Alice	lAlice	1
Susan	lAlice	5
Harry	lBob	9

Routing — Example Algorithm

Router Information Protocol (RIP)

Susan now sends me an updated table and it contains information about "Harry" that she can get a packet there within 5 hops.

Dest	Link	Cost
Allan	local	0
Bob	lBob	1
Alice	lAlice	1
Susan	lAlice	5
Harry	lAlice	6

Routing — Example Algorithm

Router Information Protocol (RIP)

- Don't forget that after each of these updates I perform a send to all outgoing links.
- In particular Bob could now have received my table linking to Harry in 6 which would mean he would have a new route to Harry through me at a cost of 7 beating his previous 8.
- I now receive a table from Alice with the "Harry" link set to ∞.

Dest	Link	Cost
Allan	local	0
Bob	lBob	1
Alice	lAlice	1
Susan	lAlice	5
Harry	lAlice	∞

Routing — Example Algorithm

Router Information Protocol (RIP)

I then later detect that the link lAlice has been broken

Dest	Link	Cost
Allan	local	0
Bob	lBob	1
Alice	lAlice	∞
Susan	lAlice	∞
Harry	lAlice	∞

Routing — Example Algorithm

Router Information Protocol (RIP)

Bob then later sends his table which still has a link to Harry at cost 8

Dest	Link	Cost
Allan	local	0
Bob	lBob	1
Alice	lAlice	∞
Susan	lAlice	∞
Harry	lBob	9

Routing — Example Algorithm

Router Information Protocol (RIP)

- This algorithm has been shown to eventually converge on the best routes to each destination whenever the network is changed
- This is a simple version of the algorithm and it may be improved in many ways:
  1. The cost metric can take into account bandwidth
  2. Avoid undesirable intermediate states before convergence, such as loops.
     - Optional home exercise: show an example where there is a looping intermediate state
- Note: this is a distributed algorithm: I promised you that "parts of computer networks are distributed systems"

Networking Issues

- Performance — We are of course most interested in the speed with which individual messages can be transferred between two computers.
  - latency delay after a send is initiated before data begins to arrive at the destination
  - data transfer rate this is the bits per second rate that is quoted.
  - Message transmission time = latency + length/data transfer rate
  - Though longer messages may require segmentation into multiple messages
  - Latency affects small frequent message passing which is common for distributed systems

Networking Issues — Performance

- Time required to transmit a short message and receive a reply on a small local network: about half a millisecond (0.0005s)
- Time required to invoke an operation on an object in local memory: sub-microsecond (0.000001s)
- About a thousand times slower on the network
- However, networks can outperform hard-disks.
- So if you have one large server with a very large amount of system memory this may perform better than several machines with small amounts of system memory
- Over the Internet we might be looking at about 5-500 milliseconds
- Some of this is latency (switching delays at routers) and some is data transfer rate (contention for network circuits)

Networking Issues — Reliability

- Physical transmission media is generally pretty reliabile — though wireless less so
- Message losses are often due to software errors
- Many applications are able to recover and/or tolerate transmission errors.
- A guaranteed communication channel is often therefore needless overhead.
- In particular because the software itself may lose the message it must be designed to account for that — it may then as well cope with transmission failure by the communication channel.
- But it depends on the transmission media
- Must try to reduce the amount of incorrect data that is transmitted as well as the amount of checking done on correct data.

Reliability

Left $\longleftrightarrow$ 1 $\longleftrightarrow$ 2 $\longleftrightarrow$ 3 $\longleftrightarrow$ 4 $\longleftrightarrow$ 5 $\longleftrightarrow$ Right

Left $\longleftrightarrow$ 1 $\longleftrightarrow$ 2 $\longleftrightarrow$ 3 $\longleftrightarrow$ 4 $\longleftrightarrow$ 5 $\longleftrightarrow$ Right

Left $\longleftrightarrow$ 1 $\longleftrightarrow$ 2 $\longleftrightarrow$ 3 $\longleftrightarrow$ 4 $\longleftrightarrow$ 5 $\longleftrightarrow$ Right

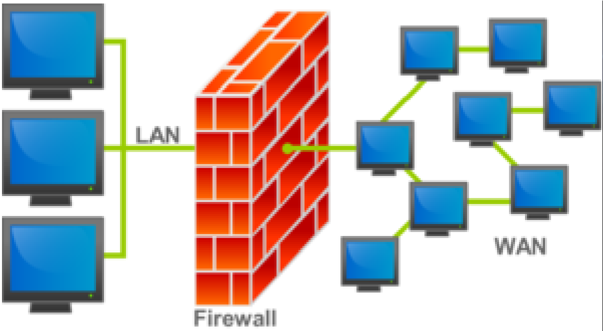
Left $\longleftrightarrow$ 1 $\longleftrightarrow$ 2 $\longleftrightarrow$ 3 $\longleftrightarrow$ 4 $\longleftrightarrow$ 5 $\longleftrightarrow$ Right

Left $\longleftrightarrow$ 1 $\longleftrightarrow$ 2 $\longleftrightarrow$ 3 $\longleftrightarrow$ 4 $\longleftrightarrow$ 5 $\longleftrightarrow$ Right

Red denotes a node at which error detection/correction occurs

- If the probability of a message getting through any channel is 0.5 then completing the trip is $0.5^6 = 0.016$
- Fortunately communication channels are generally more reliable
- $(\frac{9999}{10000})^6 = 0.9994 > \frac{999}{1000}$

Networking Issues — Security



- Security is generally handled more at the application layer
- Generally through cryptographic techniques
- Though the network can provide some level of security
- A firewall is catch all solution with associated inefficiencies
- For some organisations those inefficiencies are deemed appropriate.

Interprocess Communication

Interprocess Communication

UDP and TCP

- Two internet protocols provide two alternative transmission protocols for differing situations with different characteristics
- User Datagram Protocol — UDP
  - Simple and efficient message passing
  - Suffers from possible omission failures
  - Provides error detection but no error correction
- Transmission Control Protocol – TCP
  - Built on top of UDP
  - Provides a guaranteed message delivery service
  - But does so at the cost of additional messages
  - Has a higher latency as a stream must first be set up
  - Provides both error detection and correction

UDP and TCP

- User Datagram Protocol — UDP
  - Is connectionless
  - Used for small requests from possibly large numbers of clients
  - Examples: DNS, RIP and VOIP and online gaming
  - VOIP: The biran prefmros smoe erorr mkasnig for us
  - Sometimes used for larger requests when the application may be able to do its own error correction
- Transmission Control Protocol – TCP
  - Is connection based
  - Used for larger requests
  - Examples: SMTP, HTTP and TELNET

UDP and TCP

Failure Models

- User Datagram Protocol — UDP
  - Sometimes packages are dropped — no guaranteed validity
  - Messages may be delivered out of order — no guaranteed validity
  - Checksums are used to provide near guaranteed integrity
- Transmission Control Protocol – TCP
  - Uses checksums to give near guaranteed integrity
  - Uses sequence numbers, timeouts and retransmissions to provide guaranteed validity
  - If the communication channel is bad enough then timeouts may occur often enough for the connection to be deemed broken
    - Therefore there is no absolute guarantee of reliabile communication
    - Processes cannot distinguish between network failure and failure of the other process
    - Processes cannot be sure that recent messages have succeeded or failed.

Send and Receive

- Communication between to separate hosts is supported by two operations, simply send and receive
- Two modes of communication:
  1. Synchronous
     - communicating processes synchronise on every message
     - Hence both sending and receiving are blocking operations
     - Sort of the "instant messaging" of the data communication world
  2. Asynchronous
     - Only the receive operation is blocking, the send is not
     - The "e-mail" of the data communcation world
     - A form of non-blocking receive can be built, however this is really just a thread which waits and then sends a signal to the parent thread when there is received data to be read

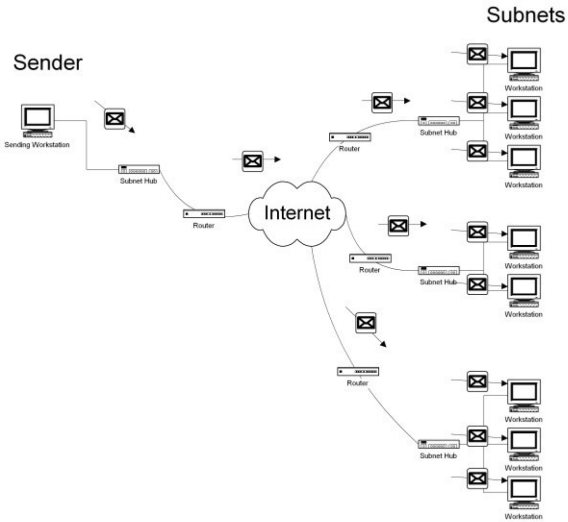
Sockets

- Sockets are a dominant abstraction programmers use for writing synchronous and asynchronous communication
- A process may use the same socket for sending and receiving
- A socket is associated with an Internet Address and a Port number
- Generally servers will advertise on which ports they will receive
- Only a single process can receive on a particular port
- Sockets can be used to send/receive UDP messages
- Sockets can also be used to set up a TCP stream of communication, generally two such streams are initiated to enable two-way communication between the two hosts.

Multicast

- For some applications it is appropriate to send a single message to many recipients
- Multicast is essentially a selective broadcast
  1. unicast
  2. anycast (one of a group)
  3. multicast
  4. broadcast
- The most common reasons for multicast are:
  1. Efficiency
  2. Simplicity/transparency for the sender, in particular the sender need not necessarily know all the recipients
- However there are some issues, in particular we must consider the failure semantics of multi-recipient messages.
- Attempts to provide strict failure semantics for multicast messages unfortunately often negate part or all of these two advantages

Multicast



Subnets
Sender
Sending Workstation
Subnet Hub
Router
Internet
Router
Subnet Hub
Workstation
Workstation
Workstation
Router
Subnet Hub
Workstation
Workstation
Router
Subnet Hub
Workstation
Workstation
Workstation

Multicast

Uses of Multicast

1. Fault tolerance based on replicated services
2. Data replication for increased efficiency
3. Discovery of services in spontaneous networking
4. Propagation of event notifications

Multicast

Plausible Failure Semantics

1. Maybe semantics — The multicast equivalent of UDP, some processes may receive each message some may not. They may receive messages in different orders
2. Either all members receive a message or none do, some may receive a message out of order
3. All members of the group receive every message in the correct order
    - called: *totally ordered multicast*
    - We will see this in more detail in a later part of the course

IP-Multicast

- UDP failure semantics:
- For each message, some members of the group may receive the message some may not
- IP-Multicast is built on top of IP
- The sender is unaware of the identities of the individual recipients of the message
- IP addresses (in IPv4) in the range 224.0.0.0 to 239.255.255.255 are reserved for multicast traffic and are managed globally
- Any socket (that is any port on any computer with an IP address) may join any IP-multicast group
- IGMP (Internet Group Management Protocol) is used both for requesting entry to a group and for communication between adjacent routers

IP-Multicast

- Upon receiving a multicast message a multicast router sends the message on to any links which have members of the group
- To avoid eternally propogating messages, each multicast message has a "Time To Live" variable which is decremented with each propogation
- Groups ownership is not addressed by the IP-multicast protocol
- For small local groups this can be achieved through using a small time to live number
- Over the Internet other solutions are required, for example Multcast Address Allocation Architecture is a client-server based solution, in which the server maintains addresses which are free.

Multicast XCAST Implementation

- An alternative way to implement multicast is to require the sender to attach each recipient address to the message
- This is used by XCAST (Explicit Multi-Unicast), which is implemented on top of IP and places each receiver's address in the IP packet header
- Since IP-packets are limited in size, this places a strict limit on the size of the group
- The group must also be known ahead of time
- However it is appropriate for use when there are a large number of small sessions which have a small number of groups
- Video conferencing for example

External Data and Marshalling

- Ultimately processes/algorithms wish to exchange data
- But messages are restricted to a sequence of bytes
- Hence the communicating processes must agree in advance a suitable format in which the data should be converted to/from a sequence of bytes
- Examples:
  - XML
  - Java serialisation
  - JSON
  - CORBA

External Data and Marshalling

CORBA

- Common Object Request Broker Architecture
- Marshals data for receivers that have prior knowledge of the types of the objects to be communicated
- Type information is defined in an Interface Definition Language (IDL) file
- IDL files can be automatically mapped to programming language type definitions and code to (de)marshall object
- Has the disadvantage that types must be agreed upon in advance
- Has the advantage that there is no overhead in communicating the type

External Data and Marshalling

Java Serialisation

- Includes the full type information in the marshalled data
- Uses reflection in order to obtain that type information
- Is restricted of course to use with the Java programming language (and languages specifically designed to interoperate)
- The .NET framework has a similar approach

XML

- More general than either Java Serialisation or CORBA
- Can be used in both modes, that is either to send type information together with the data or agree on pre-existing types

External Data and Marshalling

JSON

- Javascript Object Notation
- Includes type information, but that type information is basic
- Number, String, Boolean, Array, Null or
- Object — a list of key-value pairs
- Is becoming very popular because it is useful for many languages and requires no parsing by the application programmer
- In particular popular with dynamic languages such as Python

Summary

- UDP provides simple, efficient, connectionless sending of messages with few guarantees
- TCP provides connection-based sending atop UDP with greater guarantees of validity, no omission failures
- Programming APIs built atop these tend to rely on the *Sockets* abstraction to provide synchronous or asynchronous send and receive operations
- Marshalling is used to send complex data structures as one-dimensional sequences of bytes
- Different approaches may require prior agreement as to the types of the marshalled data and may make constraints on programming language used

Any Questions

Any Questions?

Questions

- Question : Are — and if not, why not? — platform layers not generally standardised to reduce/remove the need for middlewares like some kind of distributed POSIX?
- Answer : To some degree, for example the *Sockets* abstraction is widely implemented. Essentially middleware exists either because popular platforms have not agreed upon a common abstraction or because that abstraction more usefully sits outside of the realm of the "platform". Why platform vendors cannot agree upon common abstractions is more of a social and possibly economic question.

Distributed Systems — Questions

Questions

- Question : Proxy servers provide transparency of replication/distribution. Can they be classified as middleware?
- Answer : Middleware is a term used only for software, since a proxy must ultimately be realised in hardware we wouldn't normally say that a proxy is middleware.

Questions

- Question : With regards to synchronous and asynchronous systems, specifically determining realistic bounds, why can one not just specify bounds that covers all possible circumstances? For example, assuming that, say, HTTP GET requests will always return within 10 minutes if the server is available? Obviously this assumption is ridiculous, but at least then the bounds are known.
- Answer : The key point is whether or not one can determine useful bounds. The distinction is more in how we then treat the communication system. All systems can have fairly unreasonable bounds applied — a message may arrive instantaneously or may take 1000 years. Atop which you could attempt to build a reliable communication system using a timeout of 2000 years. Alternatively one could simply assume asynchronicity and build on top of that.

Questions

- Question : Can you give an example of a process omission failure and the difference between process omission and arbitrary failures?
- Answer :
  - A process omission failure is when a message which should have been sent (or received) simply isn't. It might be because the code of the process is erroneous, or it might be some software driver is incorrect. For example perhaps the message was put in the out-going buffer but that buffer was full and the software did not deal with that correct.
  - So for example in the RIP protocol one of the routers may simply fail to send on their updated RIP-table after an update
  - An arbitrary failure is when a message is sent, but the message sent is not the correct message. Generally this is more likely to be incorrect logic in the code of the process itself.

Questions

- Question : *Can you give an example of a process omission failure and the difference between process omission and arbitrary failures?* — continued
- Answer :
  - In the RIP algorithm one process may simply send an incorrect table. Or it may erroneously set all link costs to ∞ and hence — at least temporarily — continuously send incorrect routing tables.
  - The question is, given such an error, how does the distributed algorithm cope with this. Is it detectable? Is the behaviour acceptable even if it is not detected or in the meantime before it is detected?

Introducing PEPA

- PEPA: Performance Evaluation Process Algebra
- Modellers define their model by first describing a set of sequential components and then combining those sequential components together in parallel to form the main system equation.
- Definitions are built using the choice $(+)$, prefix $(.)$ operators.
- The system equation is built using the cooperation operators $\bowtie_L$, $\|$ and hiding $\backslash$.

Service Example

$$
\begin{array}{lcl}
\mathit{Service} & = & (\mathit{request}, \top).\mathit{Service} \\
 & + & (\mathit{service}, r_{\mathit{serve}}).\mathit{Service} \\
 & + & (\mathit{break}, r_{\mathit{break}}).\mathit{Broken} \\
\mathit{Broken} & = & (\mathit{repair}, r_{\mathit{repair}}).\mathit{Service} \\
 & + & (\mathit{request}, \top).\mathit{Broken} \\
 & & \\
\mathit{Client} & = & (\mathit{request}, r_{\mathit{join}}).\mathit{Wait} \\
\mathit{Wait} & = & (\mathit{service}, \top).\mathit{Client} \\
 & & \\
 & & \mathit{Service} \bowtie_{L} \mathit{Client}[\mathit{clients}] \\
\mathit{where}\ L & = & \{\mathit{request}, \mathit{service}\} \\
\mathit{and}\ \mathit{Client}[3] & = & \mathit{Client} \parallel \mathit{Client} \parallel \mathit{Client}
\end{array}
$$

State Specifications Examples

$Broken == 1$	The/a server is in the state *Broken*
$Wait > 3$	More than three clients waiting
$Broken == 1$ && $Wait > 3$	Both the previous are true
$Service < Wait$	Fewer servers ready than clients waiting

Activity Probe Specifications Examples

$a : start, b : stop$	Any state between the a and b actions
$P :: (a : start, b : stop)$	. . . as observed by a single P process
$(a\|b\|c) : start, (x\|y) : stop$	choice to start and end
$(a, a, a)/b : start, b : stop$	As before, without a b interrupting

PEPA

More information at: `www.dcs.ed.ac.uk/pepa`

Concurrent Finite State Machines

- An example of an interaction modelling framework
- Due to Brand and Zafiropoulo
- Consist of a set of finite state machines which can communicate via a set of communication channels
- Every FSM represents a concurrent, communication process
- One pair of channels (C_{ij} and C_{ji}) for each pair of machines
- Every communication channel is:
  - full-duplex
  - error-free
  - has a first-in-first-out strategy
  - had unbounded capacity
  - So this represents a perfect full-duplex channel

Concurrent Finite State Machines

Formalisation

- N: a positive integer
- i, j = 1, . . . , N indexes over processes
- $\langle Q_i \rangle_{i=1}^{N}$ N disjoint finite sets, Q_i denotes the state of process I.
- $\langle A_{ij} \rangle_{i,j=1}^{N}$ disjoint sets where A_{ij} denotes the message alphabet for the channel $i \longrightarrow j$
- $\forall i A_{ii} = \{\}$
- δ: relation determining for each pair (i,j) the following functions
  - $Q_i \times A_{ij} \rightarrow Q_i$: send from i to j
  - $Q_i \times A_{ji} \rightarrow Q_i$: receive from j at i
- $\langle q_i^0 \rangle$ the initial states such that $\forall i (q_i^0 \in Q_i)$
- AND SO: we call $(\langle Q_i \rangle, \langle q_i^0 \rangle, \langle A_{ij} \rangle, \delta)$ a protocol

Concurrent Finite State Machines

Notation

- `si` $\in Q_i$: state of process i
- `xij` $\in A_{ij}$: a message
  - `?xij` reception of a message
  - `!yji` sending of a message
- `f((s1, ..  sn)) = (f(s1), .., f(sn))`
- `x,y`: message
- `X,Y`: sequence of messages
- `x, xy, xY, xXY` : concatenated sequences of messages

Concurrent Finite State Machines

Alternating Bit Protocol

- Simple protocol securing unreliable message channels
- Sender sends message `msgn` with $n \in 0, 1$ a sequence number
- Receiver acknowledges with `ackn`
- Sender sets new sequence number at $1 + \mathrm{n} \bmod 2$
- Retransmission of current message when wrong sequence number receieved

Concurrent Finite State Machines

- Semantics of a protocol:
  - The set of admissable state sequences
- State of a protocol:
  - sum of:
    1. Local state of each of the processes
    2. state of all channels (which is the sequence of all messages along it which have been sent but not received)
  - We call this the global system state.

Concurrent Finite State Machines

Obtaining All Computations:

1. Initially all processes in their initial states and all channels empty
2. System is in a current global system state s
3. State transition triggered by send and receive events
   - send event:
     - add a message to the tail of the appropriate channel
     - update the local state of the sending process
   - receive event:
     - take the message from the head of the message queue
     - update the local state of the receiving process
4. Leads to a new global system state

Concurrent Finite State Machines

State Transition Relation

- Let P be a protocol and G be the set of all global system states (S,C)
- The $\vdash (G \longrightarrow G)$ is defined as follows: $(S, C) \vdash (S', C')$ iff $\exists i, k, xij$ such that (S, C) and (S', C') are identical other than either
  - si' $= \delta$(si, !xij) and cij' $=$ cij xij
  - or
  - si' $= \delta$(si, ?xij) and cji $=$ xji cji'

Concurrent Finite State Machines

Reachable Global System State

- Let:
  - G_0 be the initial global system state
  - G a global system state of the same protocol
  - $\vdash$ the state transition relation of the same protocol
  - $\vdash^*$ the transitive closure of $\vdash$
- We say that G is reachable if:
  - $G \vdash^* G$

Paths and the language accepted by a protocol can be defined via $\vdash^*$ as would be done for NFAs.

Concurrent Finite State Machines

Expressiveness

- Theorem: CFSMs are Turing complete
  - Many proofs possible (including proof by claiming it is obvious)
  - One such idea:
    - three processes P1, P2, P3
    - Simulate the control of the Turing Machine in P2
    - Use P1 and the channels c12 and c21 to simulate the left of the tape
    - Use P3 and the channels c23 and c32 to simulate the right of the tape
    - since all cij have unbounded capacity we have an inifite tape

Consequences

- global state space has unbounded size
- undecidable problems include:
  - termination
  - will some communication event ever be executed?
  - is some system state reachable?
  - is the protocol deadlock free?

Distributed Systems — Time and Global State

Allan Clark

School of Informatics
University of Edinburgh

http://www.inf.ed.ac.uk/teaching/courses/ds
Autumn Term 2012

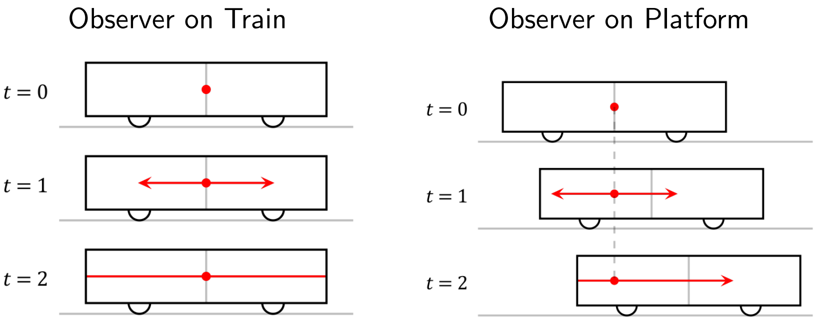
Introduction In this part of the course we will cover:

- Why time is such an issue for distributed computing
- The problem of maintaining a global state
- Consequences of these two main ideas
- Methods to get around these problems

Global Notion of Time

- Einstein  showed that the speed of light is constant for all observers regardless of their own velocity
- He (and others) have shown that this forced several other (sometimes counter-intuitive) properties including:
  1. length contraction
  2. time dilation
  3. relativity of simultaneity
     - Contradicting the classical notion that the duration of the time interval between two events is equal for all observers
     - It is impossible to say whether two events occur at the same time, if those two events are separated by space
     - A drum beat in Japan and a car crash in Brazil
     - However, if the two events are causally connected — if A causes B — the RoS preserves the causal order

Global Notion of Time



- However, if the two events are causally connected — if A causes B — the relativity of simultaneity preserves the causal order
- In this case, the flash of light happens before the light reaches either end of the carriage for all observers

Global Notion of Time

In Our World

- We operate as if this were not true, that is, as if there were some global notion of time
- People may tell you that this is because:
- On the scale of the differences in our frames of references, the effect of relativity is negligible

Global Notion of Time

In Our World

- We operate as if this were not true, that is, as if there were some global notion of time
- People may tell you that this is because:
- On the scale of the differences in our frames of references, the effect of relativity is negligible
- It's true that on our scales the effects of relativity are negligible
- But that's not really why we operate as if there was a global notion of time
- Even if our theortical clocks are well synchronised, or mechanical ones are not
- We just accept this inherent inaccuracy build that into our (social) protocols

Global Notion of Time

Physical Clocks

- Computer clocks tend to rely on the oscillations occuring in a crystal
- The difference between the instantaneous readings of two separate clocks is termed their *"skew"*
- The *"drift"* between any two clocks is the difference in the rates at which they are progressing. The rate of change of the skew
- The *drift* rate of a given clock is the drift from a nominal "perfect" clock, for quartz crystal clocks this is about 10^{-6}
- Meaning it will drift from a perfect clock by about 1 second every 1 million seconds — 11 and a half days.

Global Notion of Time

Coordinated Universal Time and French

- The most accurate clocks are based on atomic oscillators
- Atomic clocks are used as the basis for the international standard *International Atomic Time*
- Abbreviated to TAI from the French Temps Atomique International
- Since 1967 a standard second is defined as 9,192,631,770 periods of transition between the two hyperfine levels of the ground state of Caesium-133 (Cs^{133}).
- Time was originally bound to astronomical time, but astronomical and atomic time tend to get out of step
- Coordinated Universal Time — basically the same as TAI but with *leap seconds* inserted
- Abbreviated to UTC again from the French Temps Universel Coordonné

Global Notion of Time

Correctness of Clocks

- What does it mean for a clock to be correct?
- The operating system reads the node's hardware clock value, $H(t)$, scales it and adds an offset so as to produce a software clock $C(t) = \alpha H(t) + \beta$ which measures real, physical time t
- Suppose we have two *real* times t and t' such that $t < t'$
- A physical clock, H, is correct with respect to a given bound 'p' if:
- $(1 - p)(t' - t) \leq H(t') - H(t) \leq (1 + p)(t' - t)$
-
  - $(t' - t)$ — The true length of the interval
  - The measured length of the interval
  - The smallest acceptable length of the interval
  - The largest acceptable length of the interval

Global Notion of Time

Correctness of Clocks

$(1 - p)(t' - t) \leq H(t') - H(t) \leq (1 + p)(t' - t)$

- An important feature of this definition is that it is monotonic
- Meaning that:
- If $t < t'$ then $H(t) < H(t')$
- Assuming that $t < t'$ with respect to the precision of the hardware clock

Global Notion of Time

Monotonicity

- What happens when a clock is determined to be running fast?
- We could just set the clock back:
- but that would break monotonicity
- Instead, we retain monotonicity:
  - $C_i(t) = \alpha H(t) + \beta$
  - decreasing β such that $C_i(t) \leq C_i(t')$ for all $t < t'$

Global Notion of Time

External vs Internal Synchronisation

- Intuitively, multiple clocks may be synchronised with respect to each other, or with respect to an external source.
- Formally, for a synchronisation bound $D > 0$ and external source S:
  - Internal Syncronisation
    - $\mid C_i(t) - C_j(t) \mid < D$
    - No two clocks disagree by D or more
  - External Syncronisation
    - $\mid C_i(t) - S(t) \mid < D$
    - No clock disagrees with external source S by D or more
- Internally synchronised clocks may not be very accurate at all with respect to some external source
- Clocks which are externally synchronised to a bound of D though are automatically internally synchronised to a bound of $2 \times D$.

Synchronising Clocks

Synchronising in a synchronous system

- Imagine trying to synchronise watches using text messaging
- Except that you have bounds for how long a text message will take
- How would you do this?
  1. Mario sends the time t on his watch to Luigi in a message m
  2. Luigi should set his watch to $t + T_{trans}$ where T_{trans} is the time taken to transmit and receive the message m
  3. Unfortunately T_{trans} is only bound, it is not known
  4. We do know that $min \leq T_{trans} \leq max$
  5. We can therefore acheive a bound of $u = max - min$ if the Luigi sets his watch to $t + min$ or $t + max$
  6. We can do a bit better an achieve a bound of $u = \frac{max - min}{2}$ if Luigi sets his watch to $t + \frac{max + min}{2}$
  7. More generally if there are N clocks (Mario, Luigi, Peach, Toad, ...) we can achieve a bound of $(max - min)(1 - \frac{1}{N})$
  8. Or more simply we make Mario an external source and the bound is then $max - min$ (or $2 \times \frac{max - min}{2}$)

Synchronising Clocks

Cristian's Method

- The previous method does not work where we have no upper bound on message delivery time, i.e. in an asynchronous system
- Cristian's method is a method to synchronise clocks to an external source.
- This could be used to provide external or internal synchronisation as before, depending on whether the source is itself externally synchronised or not.
- The key idea is that while we might not have an upper bound on how long a single message takes, we can have an upper bound on how long a round-trip took.
- However it requires that the round-trip time is sufficiently short as compared to the required accuracy.

Synchronising Clocks

Cristian's Method

- Luigi sends Mario (our source/server) a message m_r requesting the current time, and records the time T_{sent} at which m_r was sent according to Luigi's current clock
- Upon receiving Luigi's request message m_r Mario responds with the current time according to his clock in the message m_t.
- When Luigi receives Mario's time t in message m_t, at time T_{rec} according to his own clock the round trip took $T_{round} = T_{rec} - T_{sent}$
- Luigi then sets his clock to $t + \frac{T_{round}}{2}$
- Which assumes that the elapsed time was split evenly between the exchange of the two messages.

Synchronising Clocks

Cristian's Method

- How accurate is this?
- We often don't have accurate upper bounds for message delivery times but frequently we can at least guess conservative lower bounds
- Assume that messages take at least *min* time to be delivered
- The earliest time at which Mario could have placed his time into the response message m_t is *min* after Luigi sent his request message m_r.
- The latest time at which Mario could have done this was *min* before Luigi receives the response message m_t.
- The time on Mario's watch when Luigi receives the response m_t is:
  - At least $t + min$
  - At most $t + T_{round} - min$
  - Hence the width is $T_{round} - (2 \times min)$
- The accuracy is therefore $\frac{T_{round}}{2} - min$

Synchronising Clocks

The Berkley Algorithm

- Like Cristian's algorithm this provides either external synchronisation to a known server, or internal synchronisation via choosing one of the players to be the master
- Unlike Cristian's algorithm though, the master in this case does not wait for requests from the other clocks to be synchronised, rather it periodically polls the other clocks.
- The other's then reply with a message containing their current time.
- The master, estimates the slaves current times using the round trip time in a similar way to Cristian's algorithm
- It then averages those clock readings together with its own to determine what should be the current time.
- It then replies to each of the other players with the amount by which they should adjust their clocks

Synchronising Clocks

The Berkley Algorithm

- If a straight forward average is taken a faulty clock could shift this average by a large amount, and therefore a *fault tolerant average* is taken
- This is exactly as it sounds, it averages all the clocks that do not differ by a chosen maximum amount.

Network Time Protocol

Pairwise synchronisation

- Similar to Cristian's method however:
- Four times are recorded as measured by the clock of the process at which the event occurs:
  1. T_{i-3} — Time of sending of the request message m_r
  2. T_{i-2} — Time of receiving of the request message m_r
  3. T_{i-1} — Time of sending of the response message m_t
  4. T_i — Time of receiving of the response message m_t
- So if Luigi is requesting the time from Mario, then T_{i-3} and T_i are recorded by Luigi and T_{i-2} and T_{i-1} are recorded by Mario
- Note that because Mario records the time at which the request message was received and the time at which the response message is sent, there can be a non-neglible delay between both
- In particular then messages may be dropped

Network Time Protocol

Pairwise synchronisation

- If we assume that the true offset between the two clocks is O_{true}:
- And that the actual transmission times for the messages m_r and m_t are t and t' respectively then:
- $T_{i-2} = T_{i-3} + t + O_{true}$ and
- $T_i = T_{i-1} + t' - O_{true}$
- $T_{round} = (t + t') = (T_i - T_{i-3}) - (T_{i-1} - T_{i-2})$
- $O_{guess} = \frac{(T_{i-2} - T_{i-3}) + (T_{i-1} - Ti)}{2}$

Network Time Protocol

Pairwise synchronisation

- This is the non-trivial line:
- $O_{guess} = \frac{(T_{i-2} - T_{i-3}) + (T_{i-1} - T_i)}{2}$
- $$\begin{array}{lcl} T_{i-2} - T_{i-3} & = & t + O_{true} \\ T_{i-1} - T_i & = & O_{true} - t' \\ \hline & = & (t - t') + (2 \times O_{true}) \end{array}$$
- $O_{guess} = \frac{t - t'}{2} + O_{true}$
- $O_{true} = O_{guess} + \frac{(t - t')}{2}$

Since we know that $T_{round} > \mid t - t' \mid$:

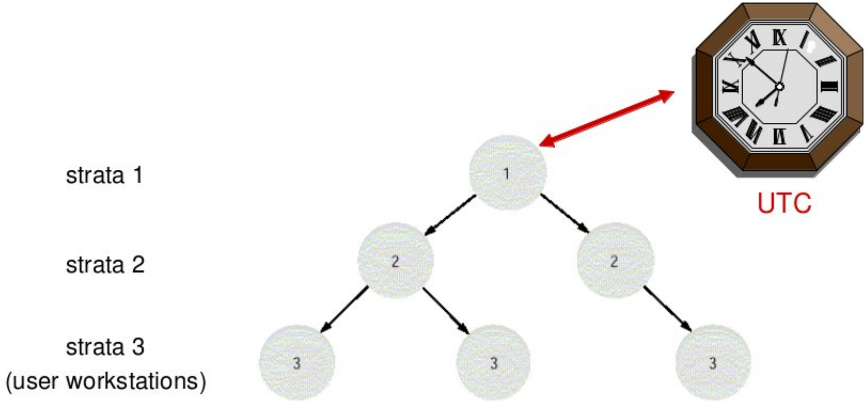
- $O_{guess} - \frac{T_{round}}{2} \leq O_{true} \leq O_{guess} + \frac{T_{round}}{2}$
- O_{guess} is the guess as to the offset
- T_{round} is the measure of how accurate it is which is essentially based on how long the messages were in transit

Synchronising Clocks

Network Time Protocol

- Network Time Protocol (actually abbreviated was NTP) is designed to allow clients to synchronise with UTC over the Internet.
- NTP is provided by a network of servers located across the Internet.
- Primary servers are connected directly to a time source such as a radio clock receiving UTC.
- Other servers are connected in a tree, with their *strata* determined by how many branches are between them and a primary server
- Strata N servers synchronise with Strata N - 1 servers
- Eventually a server is within a user's workstation
- Errors may be introduced at each level of synchronisation and they are cumulative, so the higher the strata number the less accurate is the server

Network Time Protocol



Note: Arrows denote synchronization control, numbers denote strata.

Note: this picture does not show synchronisation between servers at the same strata, but this does occur

Synchronising Clocks

Network Time Protocol

- NTP servers synchronise in one of three ways:
  1. Multicast mode
     - Not considered very accurate
     - Intended for use on a high-speed LAN
     - Can be accurate enough nonetheless for some purposes
  2. Procedure call mode
     - Similar to Cristian's method
     - Servers respond to requests from higher-strata servers
     - Who use round-trip times to calculate the current time to some degree of accuracy
     - Used for example in network file servers which wish to keep as accurate as possible file access times
  3. Symmetric mode
     - Used where the highest accuracies are required
     - In particular between servers nearest the primary sources, that is the lower strata servers
     - Essentially similar to procedure-call mode except that the communicating servers retain timing information to improve their accuracy over time

Network Time Protocol

Overview

- In all three modes messages are delivered using the standard UDP protocol
- Hence message delivery is unreliable
- At the higher strata servers can synchronise to high degree of accuracy over time
- But in general NTP is useful for synchronising accurately to UTC, whereby accurate is at the human level of accuracy
- Wall clocks, clocks at stations etc
- In summary: we can synchronise clocks to a bounded level of accuracy, but for many applications the bound is simply not tight enough

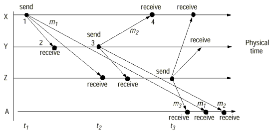
Logical Clocks

Asynchronous Orderings

- So we can achieve some measure of synchronisation between physical clocks located at different sites
- Ultimately though we will never be able to synchronise clocks to arbitrary precision
- For some applications low precision is enough, for others it is not.
- Where we cannot guarantee a high enough order of precision for synchronisation, we are forced to operate in the asynchronous world
- Despite this we can still provide a *logical* ordering on events, which may useful for certain applications

Logical Clocks

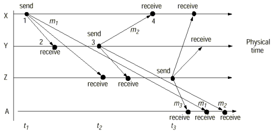
Logical Ordering



- Logical orderings attempt to give an order to events similar to physical causal ordering of reality but applied to distributed processes
- Logical clocks are based on the simple principles:
- Any process can order the events which it observes/executes
- Any message must be sent before it is received

Logical Clocks

Logical Ordering — Happened Before



- More formally we define the *happened-before* relation $\rightarrow$ by the three rules:
  1. If e_1 and e_2 are two events that happen in a single process and e_1 proceeds e_2 then $e_1 \rightarrow e_2$
  2. If e_1 is the sending of message m and e_2 is the receiving of the same message m then $e_1 \rightarrow e_2$
  3. If $e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$ then $e_1 \rightarrow e_3$

Logical Clocks

Logical Ordering — A Logical Clock

- Lamport designed an algorithm whereby events in a logical order can be given a numerical value
- This is a *logical clock*, similar to a program counter except that there is no backward jumping, and so it is monotonically increasing
- Each process P_i maintains its internal logical clock L_i
- So in order to record the logical ordering of events, each process does the following:
  - L_i is incremented immediately before each event is issued at P_i
  - When the process P_i sends a messsage m it attaches the value of its logical clock $t = L_i(m)$.
  - Upon receiving a message (m, t) process P_j computes the new value of L_j as $max(L_j, t)$

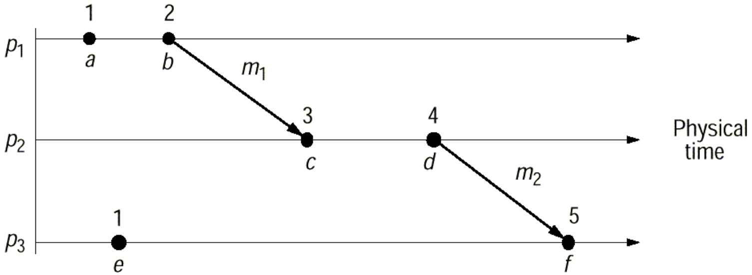
Logical Clocks

Properties

- Key point: using induction we can show that:
- $e_1 \rightarrow e_2$ implies that $L(e_1) < L(e_2)$
- However, the converse is not true, that is:
- $L(e_1) < L(e_2)$ does not imply that $e_1 \rightarrow e_2$
- It is easy to see why, consider two processes, P_1 and P_2 which each perform two steps prior to any communication.
- The two steps on the first process P_1 are concurrent with both of the two steps on process P_2.
- In particular $P_1(e_2)$ is concurrent with $P_2(e_1)$ but $L(P_1(e_2)) = 2$ and $L(P_2(e_1)) = 1$

Logical Clocks

Lamport Clocks — No reverse implication



- Here event $L(e) < L(b) < L(c) < L(d) < L(f)$
- but only $e \rightarrow f$
- e is concurrent with b, c and d.

Logical Clocks

Total Ordering

- Just as the happened-before relation is a partial ordering
- So to are the numerical Lamport stamps attached to each event
- That is, some events have the same number attached.
- However we can make it a total ordering by considering the process identifier at which the event took place
- In this case $L_i(e_1) < L_j(e_2)$ if either:
  1. $L_i(e_1) < L_j(e_2)$ OR
  2. $L_i(e_1) = L_j(e_2)$ AND $i < j$
- This has no physical meaning but can sometimes be useful

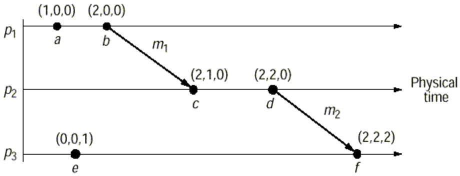
Vector Clocks

Vector Clocks augment Logical Clocks

- Vector clocks were developed (by Mattern and Fidge) to overcome the problem of the lack of a reversed implication
- That is: $L(e_1) < L(e_2)$ does not imply $e_1 \rightarrow e_2$
- Each process keeps it own vector clock V_i (an array of Lamport clocks, one for every process)
- The vector clocks are updated according to the following rules:
  1. Initially $V_i[j] = 0$
  2. As with Lamport clocks before each event at process P_i it updates its own Lamport clock within its own vector clock: $V_i[i] = V_i[i] + 1$
  3. Every message P_i sends includes its entire vector clock $t = V_i$
  4. When P_i receives a timestamp V_x then it updates all of its vector clocks with: $V_i[j] = max(V_i[j], V_x[j])$

Vector Clocks

Vector Clocks augment Logical Clocks



- Vector clocks (or timestamps) are compared as follows:
  1. $V_x = V_y$ *iff* $V_x[i] = V_y[i] \quad \forall i, 1 \ldots N$
  2. $V_x \leq V_y$ *iff* $V_x[i] \leq V_y[i] \quad \forall i, 1 \ldots N$
  3. $V_x < V_y$ *iff* $V_x[i] < V_y[i] \quad \forall i, 1 \ldots N$
- As with logical clocks: $e_1 \rightarrow e_2$ implies $V(e_1) < V(e_2)$
- In contrast with logical clocks the reverse is also true: $V(e_1) < V(e_2)$ implies $e_1 \rightarrow e_2$

Vector Clocks

Vector Clocks augment Logical Clocks

- Of course vector clocks achieve this at the cost of larger time stamps attached to each message
- In particular the size of the timestamps grows proportionally with the number of communicating processes

Summary of Logical Clocks

- Since we cannot achieve arbitrary precision of synchronisation between remote clocks via message passing
- We are forced to accept that some events are concurrent, meaning that we have no way to determine which occured first
- Despite this we can still achieve a logical ordering of events that is useful for many applications

Global State

- Correctness of distributed systems frequently hinges upon satisfying some global system invariant
- Even for applications in which you do not expect your algorithm to be correct at all times, it may still be desirable that it is "good enough" at all times
- For example our distributed algorithm maybe maintaining a record of all transactions
  - In this case it might be okay if some processes are behind other processes and thus do not know about the most recent transactions
  - But we would never want it to be the case that some process is in an inconsistent state, say applying a single transaction twice.

Global State

- Motivating examples:
  1. Distributed garbage collection
  2. Distributed deadlock detection
  3. Distributed termination detection
  4. Distributed debugging

Global State — Absence of a Global Time

You may have thought you got away from global time discussions

- Consider what happens to each of our distributed problems should we have a global time
- Distributed Garbage Collection
  - Agree a global time for each process to check whether a reference exists to a given object
  - This leaves the problem that a reference may be in transit between processes
  - But each process can say which references they have sent before the agreed time and compare that to the references received at the agreed time

Global State — Absence of a Global Time

You may have thought you got away from global time discussions

- Consider what happens to each of our distributed problems should we have a global time
- Distributed Deadlock Detection
  - Somewhat depends upon the problem in question, however:
  - At an agreed time all processes send to some master process the processes or resources for which they are waiting
  - The master process then simply checks for a loop in the resulting graph

Global State — Absence of a Global Time

You may have thought you got away from global time discussions

- Consider what happens to each of our distributed problems should we have a global time
- Distributed Termination Detection
  - At an agreed time each process sends whether or not they have completed to a master process
  - Again this leaves the problem that a message may be in transit at that time
  - Again though, we should be able to work out which messages are still in transit

Global State — Absence of a Global Time

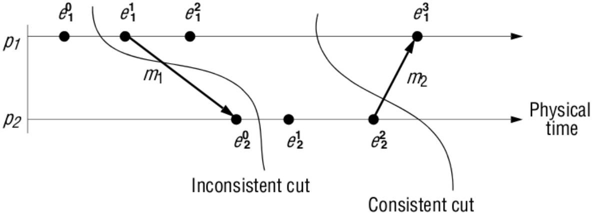
You may have thought you got away from global time discussions

- Consider what happens to each of our distributed problems should we have a global time
- Distributed Debugging
  - At each point in time we can reconstruct the global state
  - We can also record the entire history of events in the exact order in which they ocurred.
  - Allowing us to replay them and inspect the global state to see where things have gone wrong as with traditional debugging

Global State — Consistent Cuts

- So, if we had synchronised clocks, we could agree on a time for each process to record its state
- The combination of local states and the states of the communication channels would be an actual global state
- Since we cannot do that we attempt to find a "*cut*"
- A cut is a partition of events into those occurring before the cut and those occurring after the cut
- The goal is to assemble a meaningful global state from the the local states of processes recorded at different times

Global State — Consistent Cuts



- A consistent cut is one which does not violate the happens before relation $\rightarrow$
- If $e_1 \rightarrow e_2$ then either:
  - both e_1 and e_2 are before the cut or
  - both e_1 and e_2 are after the cut or
  - e_1 is before the cut and e_2 is after the cut
  - but not
  - e_1 is after the cut and e_2 is before the cut

Global State — Consistent Cuts

Runs and Linearisations

- A *consistent global state* is one which corresponds to a consistent cut
- A "*run*" is a total ordering of all events in a global history which is consistent with the local history of each process
- A "*linearisation*" is a total ordering of all events in the global history which is consistent with the happens-before relation $\rightarrow$
- So all linearisations are also runs
- Not all runs pass through consistent global states but all linearisations pass only through consistent global states

Global State — Safety and Liveness

- When we attempt to examine the global state, we are often concerned with whether or not a property holds
- Some properties, B, are properties we hope never hold and some properties, G, are properties we hope always hold
- *Safety* is the property that a bad property B does not hold for any reachable state
- *Liveness* is the property that a good property G holds for all reachable states

Global State — Stable and Unstable properties

- Some properties we wish to establish are *stable* properties
- Such properties may never become true, but once they do they remain true
- Our four example properties:
  - Garbage is *stable*: once an object has no valid references (at a process or in transit) will never have any valid references
  - Deadlock is *stable*: once a set of processes are deadlocked they will always be deadlocked without external intervention
  - Termination is *stable*: once a set of processes have terminated they will remain terminated without external intervention
  - Debugging is not really a property but the properties we may look for whilst debugging are likely *non-stable*

Global State — Snapshot

Chandy and Lamport

- The goal is to record a snapshot, or global state, of a set of processes
- The algorithm is such that the combination of recorded states may never have occured simultaneously
- However the computed global state is always a consistent one
- The state is recorded locally at each process
- The algorithm also does not address the issue of *gathering* the recorded global state.
- Though generally the locally recorded state can then be sent to some pre-agreed master process.

Global State — Chandy and Lamport

Assumptions

- There is a path between any two pairs of processes, in both directions
- Any process may initiate a global snapshot at any time
- The processes may continue their execution and send/receive normal messages whilst the snapshot takes place

Global State — Chandy and Lamport

Assumptions

- There is a path between any two pairs of processes, in both directions
- Any process may initiate a global snapshot at any time
- The processes may continue their execution and send/receive normal messages whilst the snapshot takes place
- Neither channels nor processes fail
- Communication is reliable such that every message that is sent arrives at its destination exactly once
- Channels are unidirectional and provide FIFO-ordered message delivery.

Global State — Chandy and Lamport

Algorithm — Receiver

Receiving rule for process p_i

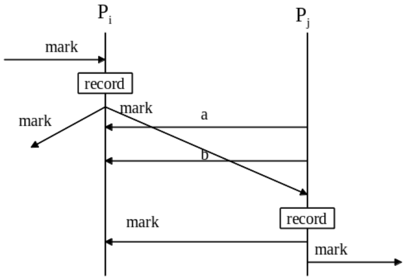
1. On receipt of a Marker message over channel c:
2. if p_i has not yet recorded state:
3. record process state now
4. record the state of c as the empty set
5. turn on recording of messages arriving on all other channels
6. else
7. records the state of c as the set of messages it has recorded since p_i first recorded its state

Global State — Chandy and Lamport

Algorithm — Sender

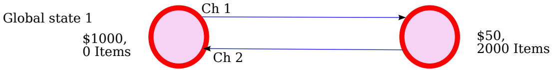
Sending rule for process p_i

1. After p_i has recorded its state:
2. p_i sends a marker message for each outgoing channel c
3. before it sends any other messages over c



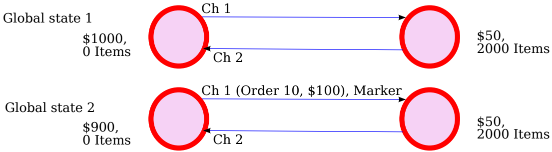
Global State — Chandy and Lamport Example

We begin in this global state, where both channels are empty, the states of the processes are as shown, but we say nothing about what has gone before.



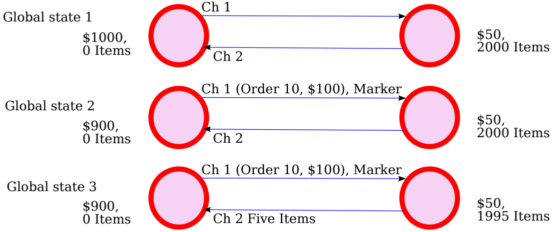
Global State — Chandy and Lamport Example

The left process decides to begin the snapshot algorithm and sends a Marker message over channel 1 to the left process. It then decides to send a request for 10 items at $10 each.



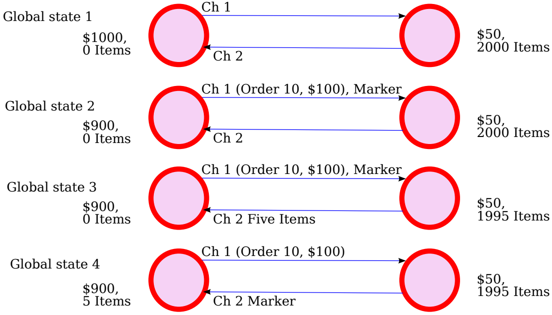
Global State — Chandy and Lamport Example

Meanwhile, the right process responds to an earlier request and sends 5 items to the left process over channel 2.

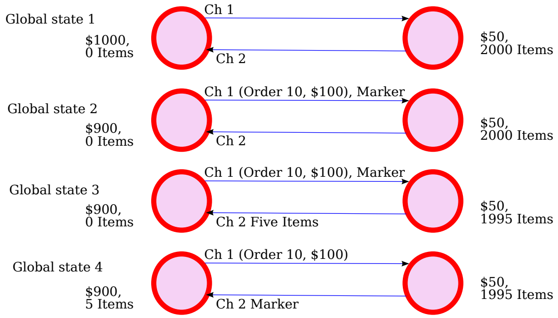


Global State — Chandy and Lamport Example

Finally the right process receives the Marker message, and in doing so records its state and sends the left process a Marker message over channel 2. When the left process receives this Marker message it records the state of channel two as containing the 5 items it has received since recording its own state.



Global State — Chandy and Lamport Example



The final recorded state is:

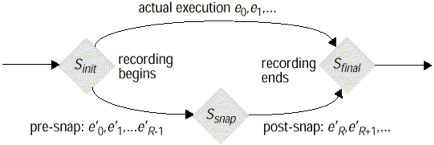
Left Process	$1000, 0
Right Process	$50, 1995
Channel 1	empty
Channel 2	Five Items

Global State — Chandy and Lamport

Reachability

- The cut found by the Chandy and Lamport algorithm is always a consistent cut
- This means that the global state which is characterised by the algorithm is a consistent global state
- Though it may not be one that ever occurred
- We can though define a reachability relation:
  - This is defined via the initial, observed and final global states when the algorithm is run
  - Assume that the events globally occurred in an order $Sys = e_1, e_2 \ldots$
  - Let S_{init} be the global state immediately before the algorithm commences and S_{final} be the global state immediately after it terminates. Finally S_{snap} is the recorded global state
  - We can find a permutation of Sys called Sys' which:
    - contains all three states: S_{init}, S_{snap} and S_{final}
    - Does not break the happens-before relationship on the events in Sys

Global State — Chandy and Lamport — Reachability



- It may be that there are two events in *Sys*, e_n and e_{n+1} such that e_n is a post-snap event and e_{n+1} is a pre-snap event
- However we can swap the order of e_n and e_{n+1} since it cannot be that $e_n \rightarrow e_{n+1}$
- We continue to swap adjacent pairs of events until all pre-snap events are ordered before all post-snap events. This gives us the the linearisation Sys'
- The reachability property of the snapshot algorithm is useful for recording stable properties
- However any non-stable predicate which is True in the snapshot may or may not be true in any other state
- Since the snapshot may not have actually occured

Global State — Chandy and Lamport

Use Cases

- No work which depends upon the global state is done until the snapshot has been gathered
- They are therefore useful for:
  1. Evaluating after the kind of change that happens infrequently
  2. Stable changes, since the property that you detect to have been true “when” the snapshot was taken will still be true once the snapshot has been gathered
  3. The kind of property that has a correct or an incorrect answer rather than a range of increasingly appropriate answers: Routing vs Garbage Collection
  4. Properties that need not be detected and acted upon immediately, for example garbage collection.

Distributed Debugging

- Distributed debugging was the application of our four example applications that stood out for being concerned with unstable properties
- This is a problem for our global snap-shot technique since its main usefulness is derived from our reachability relation which in turn means little for a non-stable property
- Distributed debugging is in a sense a combination of logical/vector clocks and global snapshots

Example Non-Stable Condition

- Suppose we are implementing an online poker game
- There is a process representing each player and one representing the pot in the centre of the table
- Players can "send chips" to the pot, and once winners have been decided the pot may send chips back to some of the players.
- We wish to make sure that the total amount of chips in the game never exceeds the initial amount
- It may be less than the initial amount since some chips may be in transit between a player and the centre pot.
- But it cannot be more than the initial amount.

Distributed Debugging

- Suppose that we have a history H of events $e_1, \ldots, e_n$
- $H(e_1, \ldots e_n)$ is therefore the true order of events as they actually occurred in our system
- Recall then that a *run* is any ordering of those events in which each event occurs exactly once
- But a *linearisation* is a consistent run
  - A consistent run is one in which the "happens-before" relation is satisfied for all pairs of events e_i, e_j
  - If $e_i \rightarrow e_j$ then any linearisation (or consistent run) will order e_i before e_j.
  - Importantly then, all linearisations only pass through consistent states

Distributed Debugging

The possibly relation

- Any linearisation *Lin* of our history of events *H* must therefore pass through only consistent states
- A property P that is true in any state through which *Lin* passes, was conceivably true at some global state through which *H* passed
- If this is the case for some property *p* and some linearisation we say $possibly(p)$
- Note: suppose we had taken a global snapshot during the set of events *H* to determine if the property *p* was true and determined that it was: $Snap(p)$ evaluates to true.
- This would imply that *p* was possible.
- However the reverse is not true, so:
  - $Snap(p) \implies possibly(p)$
  - $possibly(p) \not\Longrightarrow Snap(p)$

Distributed Debugging

The definitely relation

- The sister relation to the *possibly* relation is the *definitely* relation
- This states that for any linearisation *Lin* of *H*, *Lin* must pass through some consistent global state *S* for which the candidate property is true
- Since *H* is a linearisation of itself, then the candidate property was certainly true at some point in the history of events.

More formally:

- The statement *possibly*(p) means that there is a consistent global state S through which at least one linearisation of H passes such that $S(p)$ is true.
- The statement *definitely*(p) means that for all linearisations L of H, there is a consistent global state S through which L passes such that $S(p)$ is True

Distributed Debugging

Possibly vs Definitely

- You may think that the possibly relation is useless
- Since I knew before we started that some predicate was potentially true at some point.
- However, $\neg(\mathit{possibly}(p)) \implies \mathit{definitely}(\neg p)$
- But, from $\mathit{definitely}(\neg p)$ we cannot conclude $\neg(\mathit{possibly}(p))$.
- $\mathit{definitely}(\neg p)$ means that there is at least one state in all linearisations of H such that p is not true, but not all states.
- $\neg(\mathit{possibly}(p))$ however would require that $\neg(p)$ was true in all states in all linearisations
- Another way to put this is that $\mathit{definitely}(p)$ and $\mathit{definitely}(\neg p)$ may be true simultaneously but $\mathit{possibly}(p)$ and $\neg(\mathit{possibly}(p))$ cannot.

Distributed Debugging

Basic Outline

- The processes must all send messages recording their local state to a master process
- The master process collates these and extracts the consistent global states
- From this information the *possibly*(p) and *definitely*(p) relations may be computed.

Distributed Debugging

Collecting The Global States

- Each process sends their initial state to the master process in a state message and thereafter periodically send their local state.
- The preparing and sending of these state messages may delay the normal operation of the distributed system but does not otherwise affect it: so debugging may be turned on and off.
- "Periodically" is better defined in terms of the predicate for which we are debugging.
- So we do not send a state message to the master process other than, initially and whenever our local state changes.
- The local state need only change with respect to the predicate in question. We can concurrently check for separate predicates as well by marking our state messages appropriately.
- Additionally even if the local state changes we need only send a state message if that update could have altered the value of the predicate.

Distributed Debugging

State Message Stamps

- In order that the master process can assemble the set of consistent states from the set of state messages the individual processes send it ..
- Each state message is stamped with the Vector clock value at the local process sending the state message: $\{s_i, V(s_i)\}$
- If $S = \{s_1, \ldots s_n\}$ is a set of state messages received by the master process, and $V(s_i)$ be the vector time stamp of the particular local state s_i
- Then it is known that S is a consistent global state *iff*:
- $V_i[i] >= V_j[i] \quad \forall i, j1, ...N$

State Message Stamps

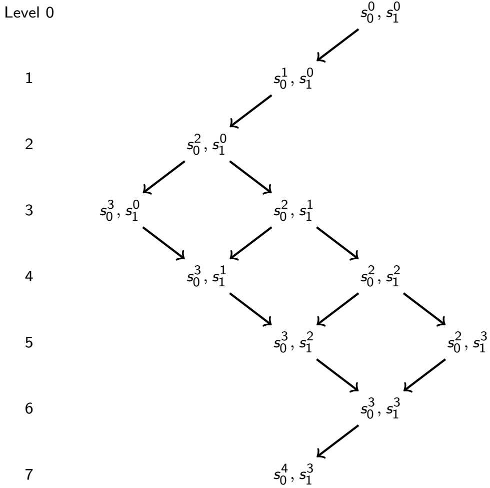
Assembled Consistent Global States

- S is a consistent global state *iff*:
- $V_i[i] >= V_j[i] \quad \forall i, j 1, ...N$
- This says that the number of p_i's events known at p_j when it sent s_j is no more than the number of events that had occurred at p_i when it sent s_i.
- In other words, if the state of one process depends upon another (according to happened-before ordering), then the global state also encompasses the state upon which it depends.

Assembling Consistent Global States

- Imagine the simplest case of 2 communicating processes.
- A plausible global state is $S(s_0^x, s_1^y)$
- The subscripts, 0 and 1, refer to the process index
- The superscripts x and y refer to the number of events which have occurred at the particular process.
- The "level" of a given state is $x + y$, which is number of events which have occurred globally to give rise to the particular global state S.

Assembling Consistent Global States



Evaluating Possibly and Definitely

1. A state $S' = \{s_0^{x_0'}, \ldots s_N^{x_N'}\}$ is reachable from a state $S = \{s_0^{x_0}, \ldots s_N^{x_N}\}$
2. If
   - S' is a consistent state
   - The level of S' is 1 plus the level of S and:
   - $x_{i'} = x_i$ or $x_{i'} = 1 + x_i \quad \forall 0 \leq i \leq N$

Evaluating Possibly

1. Level = 0
2. States = $\{(s_0^0, \ldots s_N^0)\}$
3. while (States is not empty)
   - Level = Level + 1
   - Reachable = {}
   - for S' where level(S') = Level
     - if S' is reachable from some state in States
     - then if p(S') then output possibly(p) is True and quit
     - else place S' in Reachable
   - States = Reachable
4. output possibly(p) is false

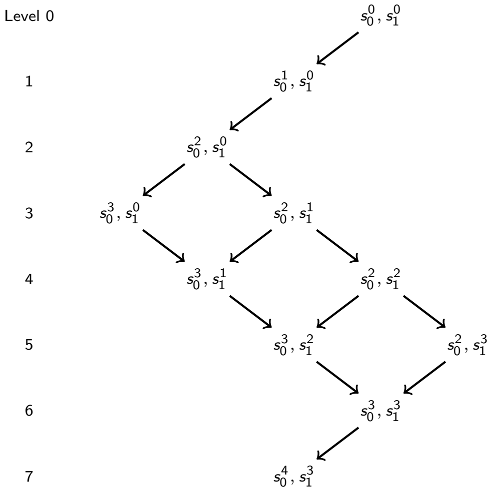
Evaluating Definitely

1. Level = 0
2. States = $\{(s_0^0, \ldots s_N^0)\}$
3. while (States is not empty)
   - Level = Level + 1
   - Reachable = {}
   - for S' where level(S') = Level
     - if S' is reachable from some state in States
     - then if $\neg(p(S'))$ then place S' in Reachable
   - States = Reachable
4. if Level is the maximum level recorded
5. then output definitely(p) is false
6. else output definitely(p) is true

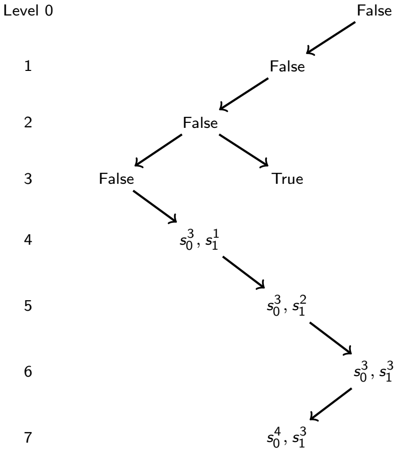
Note: Should also check if it is true in the initial state

Evaluating Definitely

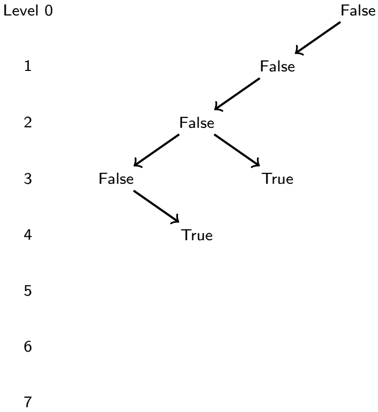
Recall:



Evaluating Definitely



Evaluating Definitely



Definitely(p) is True

Evaluating Possibly and Definitely

- Note that the number of states that must be evaluated is potentially huge
- In the worse case, there is no communication between processes, and the property is False for all states
- We must evaluate all permutations of states in which each local history is preserved
- This system therefore works better if there is a lot of communication and few local updates (which affect the predicate under investigation)

Distributed Debuggin

In a synchronous system

- We have so far considered debugging within an asynchronous system
- Our notion of a consistent global state is one which could potentially have occurred
- In a synchronous system we have a little more information to make that judgement
- Suppose each process has a clock internally synchronised with the each other to a bound of D.
- With each state message, each process additionally time stamps the message with their local time at which the state was observed.
- For a single process with two state messages (s_i^x, V_i, t_i) and (s_i^{x+1}, V_i', t_i') we know that the local state s_i^x was valid between the time interval:
- $t_i - D$ to $t_i' + D$

Distributed Debugging

In a synchronous system

- Recall our condition for a consistent global state:
- $V_i[i] >= V_j[i] \quad \forall i, j 1, ... N$
- We can add to that:
- $t_i - D \leq t_j \leq t_i' + D$ and vice versa forall i,j
- Note, this makes use of the bounds imposed in a synchronous system but speaks nothing of the time taken for a message to be delivered
- Therefore obtaining useful bounds is rather plausible
- But if there is a lot of communication then we may not prune the number of states which must be checked

Distributed Debugging

Summary

- Each process sends to a monitor process state update messages whenever a significant event occurs.
- From this the monitor can build up a set of consistent global states which may have occurred in the true history of events
- This can be used to evaluate whether some predicate was possibly true at some point, or definitely true at some point

Summary

- We noted that even in the real world there is no global notion of time
- We extended this to computer systems noting that the clocks associated with separate machines are subject to differences between them known as the *skew* and the *drift*.
- We nevertheless described algorithms for attempting the synchronisation between remote computers
  - Cristian's method
  - The Berkely Algorithm
  - Pairwise synchronisation in NTP
- Despite these algorithms to synchronise clocks it is still impossible to determine for two arbitrary events which occurred before the other.
- We therefore looked at ways in which we can impose a meaningful order on remote events and this took us to logical orderings

Summary

- Lamport and Vector clocks were introduced:
  - Lamport clocks are relatively lightweight provide us with the following $e_1 \rightarrow e_2 \implies L(e_1) < L(e_2)$
  - Vector clocks improve on this by additionally providing the reverse implication $V(e_1) < V(e_2) \implies e_1 \rightarrow e_2$
  - Meaning we can entirely determine whether $e_1 \rightarrow e_2$ or $e_2 \rightarrow e_1$ or the two events are concurrent.
  - But do so at the cost of message length and scalability
- The concept of a true history of events as opposed to *runs* and *linearisations* was introduced
- We looked at Chandy and Lamport's algorithm for recording a global snapshot of the system
- Crucially we defined a notion of reachability such that the snapshot algorithm could be usefully deployed in ascerting whether some stable property has become true.

Time and Global State

Summary

- Finally the use of consistent cuts and linearisations was used in Marzullo and Neiger's algorithm
- Used in the debugging of distributed systems it allows us to ascertain whether some transient property was possibly true at some point or definitely true at some point.
- We compare these asynchronous techniques with the obvious synchronous techniques
- We observe that while the synchronous techniques would be more accurate often, they will occasionally be wrong
- The asynchronous techniques are frequently conservative in that they may be imprecise but never wrong
- For example two events may be deemed concurrent meaning that we do not know which occurred first, but we will never erroneously ascertain that e_1 occurred before e_2

Any Questions

Any Questions?

Distributed Systems — Coordination and Agreement

Allan Clark

School of Informatics
University of Edinburgh

http://www.inf.ed.ac.uk/teaching/courses/ds
Autumn Term 2012

Coordination and Agreement

Overview

- In this part of the course we will examine how distributed processes can agree on particular values
- It is generally important that the processes within a distributed system have some sort of agreement
- Agreement may be as simple as the goal of the distributed system
  - Has the general task been aborted?
  - Should the main aim be changed?
- This is made more complicated than it sounds, since all the processes must, not only agree, but be confident that their peers agree.
- We will look at:
  - mutual exclusion to coordinate access to shared resources
  - The conditions necessary in general to guarantee that a global consensus is reached
  - Perhaps more importantly the conditions which prevent this

Coordination and Agreement

No Fixed Master

- We will also look at dynamic agreement of a master or leader process i.e. an election. Generally after the current master has failed.
- We saw in the Time and Global State section that some algorithms required a global master/nominee, but there was no requirement for that master/nominee process to be fixed
- With a fixed master process agreement is made much simpler
- However it then introduces a single point of failure
- So here we are generally assuming no fixed master process

Synchronous vs Asynchronous

- Again with the synchronous and asynchronous
- It is an important distinction here, synchronous systems allow us to determine important bounds on message transmission delays
- This allows us to use timeouts to detect message failure in a way that cannot be done for asynchronous systems.

Coping with Failures

- In this part we will consider the presence of failures, recall from our Fundamentals part three decreasingly benign failure models:
  1. Assume no failures occur
  2. Assume omission failures may occur; both process and message delivery omission failures.
  3. Assume that arbitrary failures may occur both at a process or through message corruption whilst in transit.

A Brief Aside

Failure Detectors

- Here I am talking about the detection of a crashed process
- Not one that has started responding erroneously
- Detecting such failures is a major obstracle in designing algorithms which can cope with them
- A failure detector is a process which responds to requests querying whether a particular process has failed or not
- The key point is that a failure detector is not necessarily accurate.
- One can implement a “reliable failure detector”
- One which responds with: “Unsuspected” or “Failed”

Failure Detectors

Unreliable Failure Detectors

- An "unreliable failure detector" will respond with either: "Suspected" or "Unsuspected"
- Such a failure detector is termed an "unreliable failure detector"

A simple algorithm

- If we assume that all messages are delivered within some bound, say D seconds.
- Then we can implement a simple failure detector as:
- Every process p sends a "p is still alive" message to all failure detector processes, periodically, once every T seconds
- If a failure detector process does not receive a message from process q within $T + D$ seconds of the previous one then it marks q as "Suspected"

Failure Detectors

Reliable and Unreliable

- If we choose our bound D too high then often a failed process will be marked as “Unsuspected”
- A synchronous system has a known bound on the message delivery time and the clock drift and hence can implement a reliable failure detector
- An asynchronous system could give one of three answers: “Unsuspected”, “Suspected” or “Failed” choosing two different values of D
- In fact we could instead respond to queries about process p with the probability that p has failed, if we have a known distribution of message transmission times
- e.g., if you know that 90% of messages arrive within 2 seconds and it has been two seconds since your last expected message you can conclude there is a:

Failure Detectors

Reliable and Unreliable

- NOT a 90% chance that the process p has failed.
- We do not know how long the previous message was delayed
- Even if so, Bayes theorem tells that, in order to calculate the probability that p has failed given that we have not received a message we would also require the probability that p fails within the given time increment without prior knowledge.
- Bayes: $P(a|b) = \frac{P(b|a) \times P(a)}{P(b)}$
- here $a = p$ has failed and $b =$ the message has failed to be delivered
- Further the question arises what would the process receiving that probability information do with it?
- 1. if $(p > 90)$ ...
  2. else ...

Coordination and Agreement

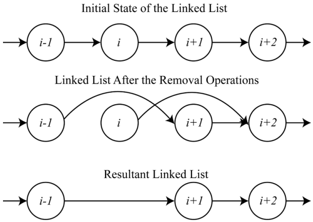
Mutual Exclusion

- Ensuring mutual exclusion to shared resources is a common task
- For example, processes A and B both wish to add a value to a shared variable ‘a’.
- To do so they must store the temporary result of the current value for the shared variable ‘a’ and the value to be added.

Time	Process A	Process B	
1	t = a + 10		A stores temporary
2		t’ = a + 20	B stores temporary
3		a = t’	(a now equals 25)
4	a = t		(a now equal 15)

- The intended increment for a is 30 but B’s increment is nullified

Mutual Exclusion




```
new-next = i.next
(i-1).next = new-next
```

```
new-next = (i+1).next
i.next = new-next
```

Shamelessly stolen from Wikipedia

- A higher-level example is the concurrent editing of a file on a shared directory
- Another good reason for using a source code control system

Distributed Mutual Exclusion

- On a local system mutual exclusion is usually a service offered by the operating system's kernel.
- But for a distributed system we require a solution that operates only via message passing
- In some cases the server that provides access to the shared resource can also be used to ensure mutual exclusion
- But here we will consider the case that this is for some reason inappropriate, the resource itself may be distributed for example

Distributed Mutual Exclusion

Generic Algorithms for Mutual Exclusion

- We will look at the following algorithms which provide mutual exclusion to a shared resource:
    1. The central-server algorithm
    2. The ring-based algorithm
    3. Ricart and Agrawala — based on multicast and logical clocks
    4. Maekawas voting algorithm
- We will compare these algorithms with respect to:
    1. Their ability to satisfy three desired properties
    2. Their performance characteristics
    3. How fault tolerant they are

Generic Algorithms for Mutual Exclusion

Assumptions and Scenario

- Before we can describe these algorithms we must make explicit our assumptions and the task that we wish to achieve
- Assumptions:
  1. The system is asynchronous
  2. Processes do not fail
  3. Message delivery is reliable: all messages are eventually delivered exactly once.
- Scenario:
  - Assume that the application performs the following sequence:
    1. Request access to shared resource, blocking if necessary
    2. Use the shared resource exclusively — called the *critical section*
    3. Relinquish the shared resource
- Requirements:
  1. Safety: At most one process may execute the critical section at any one time
  2. Liveness: Requests to enter and exit the critical section eventually succeed.

Generic Algorithms for Mutual Exclusion

Assumptions and Scenario

- The *Liveness* property assures that we are free from both deadlock and starvation — starvation is the indefinite postponement of the request to enter the critical section from a given process
- Freedom from starvation is referred to as a "*fairness*" property
- Another fairness property is the order in which processes are granted access to the critical section
- Given that we cannot ascertain which event of a set occured first we instead appeal to the "*happened-before*" logical ordering of events
- We define the *Fairness* property as: If e_1 and e_2 are requests to enter the critical section and $e_1 \rightarrow e_2$, then the requests should be granted in that order.
- Note: our assumption of request-enter-exit means that process will not request a second access until after the first is granted

Generic Algorithms for Mutual Exclusion

Assumptions and Scenario

- Here we assume that when a process requests entry to the critical section, then until the access is granted it is blocked only from entering the critical section
- In particular it may do other useful work and send/receive messages
- If we were to assume that a process is blocked entirely then the *Fairness* property is trivially satisfied

Generic Algorithms for Mutual Exclusion

Assumptions and Scenario

- Here we are considering mutual exclusion of a single critical section
- We assume that if there are multiple resources then either:
  - Access to a single critical section suffices for all the shared resources, meaning that one process may be blocked from using one resource because another process is currently using a different resource or
  - A process cannot request access to more than one critical section concurrently or
  - Deadlock arising from two (or more) processes holding each of a set of mutually desired resources is avoided/detected using some other means
  - We also assume that a process granted access to the critical section will eventually relinquish that access

Generic Algorithms for Mutual Exclusion

Desirable Properties — Recap

- We wish our mutual exclusion algorithms to have the three properties:
  1. Safety — No two processes have concurrent access to the critical section
  2. Liveness — All requests to enter/exit the critical section eventually succeed.
  3. Fairness — Requests are granted in the logical order in which they were submitted

Distributed Mutual Exclusion Algorithms

Central Server Algorithm

- The simplest way to ensure mutual exclusion is through the use of a centralised server
- This is analogous to the operating system acting as an arbiter
- There is a conceptual token, processes must be in possesion of the token in order to execute the critical section
- The centralised server maintains ownership of the token
- To request the token; a process sends a request to the server
  - If the server currently has the token it immediately responds with a message, granting the token to the requesting process
  - When the process completes the critical section it sends a message back to the server, relinquishing the token
  - If the server doesn't have the token, some other process is "currently" in the critical section
  - In this case the server queues the incoming request for the token and responds only when the token is returned by the process directly ahead of the requesting process in the queue (which may be the process currently using the token)

Distributed Mutual Exclusion Algorithms

Central Server Algorithm

- Given our assumptions that no failures occur it is straight forward to see that the central server algorithm satisfies the Safety and Liveness properties
- The Fairness property though is not

Distributed Mutual Exclusion Algorithms

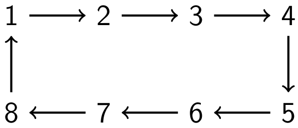
Central Server Algorithm

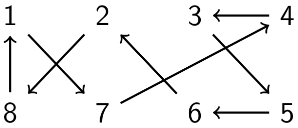
- Given our assumptions that no failures occur it is straight forward to see that the central server algorithm satisfies the Safety and Liveness properties
- The Fairness property though is not
- Consider two processes P_1 and P_2 and the following sequence of events:
    1. P_1 sends a request r_1 to enter the critical section
    2. P_1 then sends a message m to process P_2
    3. P_2 receives message m and then
    4. P_2 sends a request r_2 to enter the critical section
    5. The server process receives request r_2
    6. The server process grants entry to the critical section to process P_2
    7. The server process receives request r_1 and queues it
- Despite $r_1 \rightarrow r_2$ the r_2 request was granted first.

Distributed Mutual Exclusion Algorithms

Ring-based Algorithm

- A simple way to arrange for mutual exclusion without the need for a master process, is to arrange the processes in a logical ring.
- The ring may of course bear little resemblance to the physical network or even the direct links between processes.





Distributed Mutual Exclusion Algorithms

Ring-based Algorithm

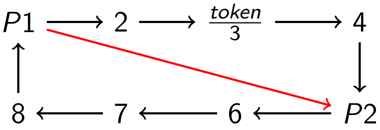
- The token passes around the ring continuously.
- When a process receives the token from its neighbour:
  - If it does not require access to the critical section it immediately forwards on the token to the next neighbour in the ring
  - If it requires access to the critical section, the process:
    1. retains the token
    2. performs the critical section and then:
    3. to relinquish access to the critical section
    4. forwards the token on to the next neighbour in the ring

Distributed Mutual Exclusion Algorithms

Ring-based Algorithm

- Once again it is straight forward to determine that this algorithm satisfies the Safety and Liveness properties.
- However once again we fail to satisfy the Fairness property

Ring-based Algorithm



- Recall that processes may send messages to one another independently of the token
- Suppose again we have two processes P_1 and P_2 consider the following events
  1. Process P_1 wishes to enter the critical section but must wait for the token to reach it.
  2. Process P_1 sends a message m to process P_2.
  3. The token is currently between process P_1 and P_2 within the ring, but the message m reaches process P_2 before the token.
  4. Process P_2 after receiving message m wishes to enter the critical section
  5. The token reaches process P_2 which uses it to enter the critical section before process P_1

Distributed Mutual Exclusion Algorithms

Multicast and Logical Clocks

- Ricart and Agrawala developed an algorithm for mutual exclusion based upon mulitcast and logical clocks
- The idea is that a process which requires access to the critical section first broadcasts this request to all processes within the group
- It may then only actually enter the critical section once each of the other processes have granted their approval
- Of course the other processes do not just grant their approval indiscriminantly
- Instead their approval is based upon whether or not they consider their own request to have been made first

Distributed Mutual Exclusion Algorithms

Multicast and Logical Clocks

- Each process maintains its own Lamport clock
- Recall that Lamport clocks provide a partial ordering of events but that this can be made a total ordering by considering the process identifier of the process observing the event
- Requests to enter the critical section are multicast to the group of processes and have the form $\{T, p_i\}$
- T is the Lamport time stamp of the request and p_i is the process identifier
- This provides us with a total ordering of the sending of a request message $\{T_1, p_i\} < \{T_2, p_j\}$ if:
  - $T_1 < T_2$ or
  - $T_1 = T_2$ and $p_i < p_j$

Multicast and Logical Clocks

Requesting Entry

- Each process retains a variable indicating its state, it can be:
  1. "Released" — Not in or requiring entry to the critical section
  2. "Wanted" — Requiring entry to the critical section
  3. "Held" — Acquired entry to the critical section and has not yet relinquished that access.
- When a process requires entry to the critical section it updates its state to "Wanted" and multicasts a request to enter the critical section to all other processes. It stores the request message $\{T_i, p_i\}$
- Only once it has received a "permission granted" message from all other processes does it change its state to "Held" and use the critical section

Multicast and Logical Clocks

Responding to requests

- Upon receiving such a request a process:
  - Currently in the "Released" state can immediately respond with a permission granted message
  - A process currently in the "Held" state:
    1. Queues the request and continues to use the critical section
    2. Once finished using the critical section responds to all such queued requests with a permission granted message
    3. changes its state back to "Released"
  - A process currently in the "Wanted" state:
    1. Compares the incoming request message $\{T_j, p_j\}$ with its own stored request message $\{T_i, p_i\}$ which it broadcasted
    2. If $\{T_i, p_i\} < \{T_j, p_j\}$ then the incoming request is queued as if the current process was already in the "Held" state
    3. If $\{T_i, p_i\} > \{T_j, p_j\}$ then the incoming request is responded to with a permission granted message as if the current process was in the "Released" state

Multicast and Logical Clocks

Safety, Liveness and Fairness

- Safety — If two or more processes request entry concurrently then whichever request bares the lowest (totally ordered) timestamp will be the first process to enter the critical section
- All others will not receive a permission granted message from (at least) that process until it has exited the critical section
- Liveness — Since the request message timestamps are a total ordering, and all requests are either responded to immediately or queued and eventually responded to, all requests to enter the critical section are eventually granted
- Fairness — Since lamport clocks assure us that $e_1 \rightarrow e_2$ implies $L(e_1) < L(e_2)$:
- for any two requests r_1, r_2 if $r_1 \rightarrow r_2$ then the timestamp for r_1 will be less than the timestamp for r_2
- Hence the process that multicast r_1 will not respond to r_2 until after it has used the critical section
- Therefore this algorithm satisfies all three desired properties

Distributed Mutual Exclusion Algorithms

Maekawas voting algorithm

- Maekawa's voting algorithm improves upon the multicast/logical clock algorithm with the observation that not all the peers of a process need grant it access
- A process only requires permission from a subset of all the peers, provided that the subsets associated with any pair of processes overlap
- The main idea is that processes vote for which of a group of processes vying for the critical section can be given access
- The processes that are within the intersection of two competing processes can ensure that the Safety property is observed

Distributed Mutual Exclusion Algorithms

Maekawas voting algorithm

- Each process p_i is associated with a *voting set* V_i of processes
- The set V_i for the process p_i is chosen such that:
  1. $p_i \in V_i$ — A process is in its own voting set
  2. $V_i \cap V_j \neq \{\}$ — There is at least one process in the overlap between any two voting sets
  3. $|V_i| = |V_j|$ — All voting sets are the same size
  4. Each process p_i is contained within M voting sets

Distributed Mutual Exclusion Algorithms

Maekawas voting algorithm

- The main idea in contrast to the previous algorithm is that each process may only grant access to one process at a time
- A process which has already granted access to another process cannot do the same for a subsequent request. In this sense it has already voted
- Those subsequent requests are queued
- Once a process has used the critical section it sends a release message to its voting set
- Once a process in the voting set has received a release message it may once again vote, and does so immediately for the head of the queue of requests if there is one

Maekawas voting algorithm

The state of a process

- As before each process maintains a state variable which can be one of the following:
  1. "Released" — Does not have access to the critical section and does not require it
  2. "Wanted" — Does not have access to the critical section but does require it
  3. "Held" — Currently has access to the critical section
- In addition each process maintains a boolean variable indicating whether or not the process has "voted"
- Of course voting is not a one-time action. This variable really indicates whether some process within the voting set has access to the critical section and has yet to release it
- To begin with, these variables are set to "Released" and False respectively

Maekawas voting algorithm

Requesting Permission

- To request permission to access the critical section a process p_i:
  1. Updates its state variable to "Wanted"
  2. Multicasts a request to all processes in the associated voting set V_i
  3. When the process has received a "permission granted" response from all processes in the voting set V_i: update state to "Held" and use the critical section
  4. Once the process is finished using the critical section, it updates its state again to "Released" and multicasts a "release" message to all members of its voting set V_i

Maekawas voting algorithm

Granting Permission/Voting

- When a process p_j receives a request message from a process p_i:
  - If its state variable is "Held" or its voted variable is True:
    1. Queue the request from p_i without replying
  - otherwise:
    1. send a "permission granted" message to p_i
    2. set the voted variable to True
- When a process p_j receives a "release" message:
  - If there are no queued requests:
    1. set the voted variable to False
  - otherwise:
    1. Remove the head of the queue, p_q:
    2. send a "permission granted" message to p_q
    3. The voted variable remains as True

Maekawas voting algorithm

Deadlock

- The algorithm as described does not respect the Liveness property
- Consider three processes p_1, p_2 and p_3
- Their voting sets: $V_1 = \{p_1, p_2\}$, $V_2 = \{p_2, p_3\}$ and $V_3 = \{p_3, p_1\}$
- Suppose that all three processes concurrently request permission to access the critical section
- All three processes immediately respond to their own requests
- All three processes have their "voted" variables set to True
- Hence, p_1 queues the subsequently received request from p_3
- Likewise, p_2 queues the subsequently received request from p_1
- Finally, p_3 queues the subsequently received request from p_2
- $\ddot{\frown}$

Maekawas voting algorithm

Safety, Liveness and Fairness

- Safety — Safety is achieved by ensuring that the intersection between any two voting sets is non-empty.
  - A process can only vote (or grant permission) once between each successive “release” message
  - But for any two processes to have concurrent access to the critical section, the non-empty intersection between their voting sets would have to have voted for both processes
- Liveness — As described the protocol does not respect the Liveness property
  - It can however be adapted to use Lamport clocks similar to the previous algorithm
- Fairness — Similarly the Lamport clocks extension to the algorithm allows it to satisfy the Fairness property

Performance Evaluation

- We have four algorithms: central server, ring based, Ricart and Agrawala's and Maekawa's voting algorithm
- We have three logical properties with which to compare them, we can also compare them with respect to performance:
- For performance we are interested in:
  1. The number of messages sent in order to *enter* and *exit* the critical section
  2. The *client delay* incurred at each *entry* and *exit* operation
  3. The *synchronisation delay*, this is delay between one process exiting the critical section and a waiting process entering
- Note: which of these is (more) important depends upon the application domain, and in particular how often critical section access is required

Mutual Exclusion Performance Evaluation

Central Server Algorithm

- Entering the critical section:
  - requires two messages, the request and the reply — even when no other process currently occupies it
  - The client-delay is the time taken for this round-trip
- Exiting the critical section:
  - requires only the sending of the "release" message
  - Incurs no delay for the client, assuming asynchronous message passing.
- The synchronisation-delay is also a round-trip time, the time taken for the "release" message to be sent from client to server and the time taken for the server to send the "grant" message to the next process in the queue.

Mutual Exclusion Performance Evaluation

Ring-based Algorithm

- Entering the critical section:
  - Requires between 0 and N messages
  - Delay, these messages are serialised so the delay is between 0 and N
- Exiting the critical section:
  - Simply requires that the holding process sends the token forward through the ring
- The synchronisation-delay is between 1 and N-1 messages

Mutual Exclusion Performance Evaluation

Ricart and Agrawala

- Entering the critical section:
  - This requires 2(N - 1) messages, assuming that multicast is implemented simply as duplicated message, it requires N-1 requests and N-1 replies.
  - Bandwidth-wise this may be bad, but since these messages are sent and received concurrently the time taken is comparable to the round-trip time of the previous two algorithms
- Exiting the critical section:
  - Zero if no other process has requested entry
  - Must send up to N-1 responses to queued requests, but again if this is asynchronous there is no waiting for a reply
- The synchronisation-delay is only one message, the holder simply responds to the queued request

Mutual Exclusion Performance Evaluation

Maekawa's Voting algorithm

- Entering the critical section:
  - This requires $2 \times \sqrt{N}$ messages
  - As before though, the delay is comparable to a round-trip time
- Exiting the critical section:
  - This requires $\sqrt{N}$ messages
  - The delay though is comparable to a single message
  - The total for entry/exit is thus $3 \times \sqrt{N}$ which compares favourably to Ricart and Agrawala's total of $2(N-1)$ where $N > 4$.
- The synchronisation-delay is a round-trip time as it requires the holding process to multi-cast to its voting set the "release" message and then intersecting processes must send a permission granted message to the requesting process

Mutual Exclusion Performance Evaluation

Further Considerations

- The ring-based algorithm continuously consumes bandwidth as the token is passed around the ring even when no process requires entry
- Ricart and Agrawala — the process that last used the critical section can simply re-use it if no other requests have been received in the meantime

Mutual Exclusion Algorithms

Fault Tolerance

- None of the algorithms described above tolerate loss of messages

Mutual Exclusion Algorithms

Fault Tolerance

- None of the algorithms described above tolerate loss of messages
- The token based algorithms lose the token if such a message is lost meaning no further accesses will be possible

Mutual Exclusion Algorithms

Fault Tolerance

- None of the algorithms described above tolerate loss of messages
- The token based algorithms lose the token if such a message is lost meaning no further accesses will be possible
- Ricart and Agrawala's method will mean that the requesting process will indefinitely wait for (N - 1) "permission granted" messages that will never come because one or more of them have been lost

Fault Tolerance

- None of the algorithms described above tolerate loss of messages
- The token based algorithms lose the token if such a message is lost meaning no further accesses will be possible
- Ricart and Agrawala's method will mean that the requesting process will indefinitely wait for (N - 1) "permission granted" messages that will never come because one or more of them have been lost
- Maekawa's algorithm cannot tolerate message loss without it affecting the system, but parts of the system may be able to proceed unhindered

Fault Tolerance

Process Crashes

- What happens when a process crashes?
  1. Central server, provided the process which crashes is not the central server, does not hold the token and has not requested the token, everything else may proceed unhindered

Fault Tolerance

Process Crashes

- What happens when a process crashes?
  1. Central server, provided the process which crashes is not the central server, does not hold the token and has not requested the token, everything else may proceed unhindered
  2. Ring-based algorithm — complete meltdown, but we may get through up to N-1 critical section accesses in the meantime

Fault Tolerance

Process Crashes

- What happens when a process crashes?
  1. Central server, provided the process which crashes is not the central server, does not hold the token and has not requested the token, everything else may proceed unhindered
  2. Ring-based algorithm — complete meltdown, but we may get through up to N-1 critical section accesses in the meantime
  3. Ricart and Agrawala — complete meltdown, we might get through additional critical section accesses if the failed process has already responded to them. But no subsequent requests will be granted

Fault Tolerance

Process Crashes

- What happens when a process crashes?
  1. Central server, provided the process which crashes is not the central server, does not hold the token and has not requested the token, everything else may proceed unhindered
  2. Ring-based algorithm — complete meltdown, but we may get through up to N-1 critical section accesses in the meantime
  3. Ricart and Agrawala — complete meltdown, we might get through additional critical section accesses if the failed process has already responded to them. But no subsequent requests will be granted
  4. Maekawa's voting algorithm — This can tolerate some process crashes, provided the crashed process is not within the voting set of a process requesting critical section access

Mutual Exclusion Algorithms

Fault Tolerance

- All of these algorithms may be adapted to recover from process failures
- Given a failure detector(s)
- Note, however, that this problem is non-trivial
- In particular because for all of these algorithms a failed process looks much like one which is currently using the critical section
- The key point is that the failure may occur *at any point*
- A synchronous system may be sure that a process has failed and take appropriate action
- An asynchronous system cannot be sure and hence may steal the token from a process currently using the critical section
  - Thus violating the *Safety* property

Mutual Exclusion Fault Tolerance

Considerations

- Central server
  - care must be taken to decide whether the server or the failed process held the token at the time of the failure
  - If the server itself fails a new one must be elected, and any queued requests must be re-made.
- Ring-based algorithm
  - The ring can generally be easily fixed to circumvent the failed process
  - The failed process may have held or blocked the progress of the token
- Ricart and Agrawala
  - Each requesting process should record *which* processes have granted permission rather than simply how many
  - The failed process can simply be removed from the list of those required
- Maekawa's voting algorithm
  - Trickier, the failed process may have been in the intersection between two voting sets

Elections

- Several algorithms which we have visited until now required a master or nominee process, including:
  1. Berkley algorithm for clock synchronisation
  2. Distributed Debugging
  3. The central server algorithm for mutual exclusion
- Even other algorithms may need a nominee to actually report the results of the algorithm
- For example Chandy and Lamport's snap shot algorithm described how to record the local state at each process in such a way that a consistent global state could be assembled from the local states recorded at different times
- To actually be useful these local states must be gathered together, a simple way to do this is for each local process to send their locally recorded state to a nominee process

Elections

No Fixed Master/Nominee

- A simple way to provide a master process, is to simply name one
- However if the named process fails there should be a recovery plan
- A recovery plan requires that we dynamically decide who should become the new master/nominee
- Even with a fixed order this is non-trivial, in particular as all participants must agree that the current master as failed
- A more dynamic election process can allow for greater flexibility of a running system

Elections

Assumptions and Scenario

- We will assume that any of the N processes may call for an election of a nominee process at any time
- We will assume that no process calls more than one such election concurrently
- But that all N processes may separately call for an election concurrently

Elections

Requirements

- We require that the result of the election should be unique
- (no hung-parliaments or coalitions)
- Even if multiple processes call for an election concurrently
- We will say that the elected process should be the best choice:
  - For our purposes we will have a simple identifier for each process, and the process with the highest identifier should "win" the election
  - In reality the identifier could be any useful property, such as available bandwidth
  - The identifiers should be unique and consist of a total ordering
  - In practice this can be done much like equal Lamport time stamps can be given an artificial ordering using a process identifier/address
  - However care would have to be taken in the case that several properties were used together such as uptime, available bandwidth and geographical location

Elections

Assumptions and Scenario

- Each process at any point in time is either a *participant* or a *non-participant* corresponding to whether the process itself believes it is participating in an election
- Each process p_i has a variable $elected_i$ which contains the identifier of the elected process
- When the process p_i first becomes a participant, the $elected_i$ variable is set to the special value $\perp$
- This means that the process does not yet know the result of the election

Elections

Requirements

- Safety A participant process p_i has $elected_i = \bot$ or $elected_i = P$, where P is chosen as the non-crashed process at the end of the run with the largest identifier
- Liveness All processes participate and eventually either crash or have $elected_i \neq \bot$
- Note that there may be some process p_j which is not yet a participant which has $elected_j = Q$ for some process which is not the eventual winner of the election
- An additional property then could be specified as, no two processes concurrently have $elected_i$ set to two different processes
  - Either one may be set to a process and the other to $\bot$
  - But if they are both set to a process it should be the same one
  - We'll call this property Total Safety

Elections

Election/Nominee Algorithms

- We will look at two distributed election algorithms
  1. A ring-based election algorithm similar to the ring-based mutual-exclusion algorithm
  2. The bully election algorithm
- We will evaluate these algorithms with respect to their performance characteristics, in particular:
  - The total number of messages sent during an election — this is a measure of the bandwidth used
  - The turn-around time, measured by the number of serialised messages sent:
    - Recall Ricart and Agrawala's algorithm for mutual exclusion that required $2(N-1)$ messsages to enter the critical section, but that that time only amounted to a turn-around time, since the only serialisation was that each response message followed a request message.

Elections

Ring-based Election Algorithm

- As with the ring-based mutual exclusion algorithm the ring-based election algorithm requires that the processes are arranged within a logical ring
- Once again this ring is logical and may bear no resemblance to any physical or geographical structure
- As before all messages are sent clockwise around the ring
- We will assume that there are no failures after the algorithm is initiated
- It may have been initiated because of an earlier process failure, but we assume that the ring has been reconstructed following any such loss
- It is also possible that the election is merely due to high computational load on the currently elected process

Ring-based Election Algorithm

Initiating an election

- Initially all processes are marked as “non-participant”
- Any process may begin an election at any time
- To do so, a process p_i:
  1. marks itself as a “participant”
  2. sets the $elected_i$ variable to $\perp$
  3. Creates an election message and places its own identifier within the election message
  4. Sends the election message to its nearest clockwise neighbour in the ring

Ring-based Election Algorithm

Receiving an election message

- When a process p_i receives an election message:
  1. Compares the identifier in the election message with its own
  2. if its own identifier is the lower:
     - It marks itself as a participant
     - sets its $elected_i$ variable to $\perp$
     - forwards the message on to the next clockwise peer in the ring
  3. if its own identifier is higher:
     - It marks itself as a participant
     - sets its $elected_i$ variable to $\perp$
     - Substitutes its own identifier into the election message and forwards it on to the next clockwise peer in the ring

Ring-based Election Algorithm

Receiving an election message

- When a process p_i receives an election message:
  1. Compares the identifier in the election message with its own
  2. if its own identifier is the lower:
     - It marks itself as a participant
     - sets its $elected_i$ variable to $\perp$
     - forwards the message on to the next clockwise peer in the ring
  3. if its own identifier is higher:
     - It marks itself as a participant
     - sets its $elected_i$ variable to $\perp$
     - Substitutes its own identifier into the election message and forwards it on to the next clockwise peer in the ring
  4. if its own identifier is in the received election message:
     - Then it has won the election
     - It marks itself as non-participant
     - sets its $elected_i$ variable to its own identifier
     - and sends an "elected" message with its own identifier to the next clockwise peer in the ring

Ring-based Election Algorithm

Receiving an elected message

- When a process p_i receives an elected message:
  1. marks itself as a non-particpant
  2. sets its $elected_i$ variable to the identifier contained within the elected message
  3. if it is not the winner of the election:
     - forward the *elected* message on to the next clockwise peer in the ring
  4. otherwise The election is over and all peers should have their $elected_i$ variable set to the identifier of the agreed upon elected process

Ring-based Election Algorithm

Required Properties

- Safety:
  - A process must receive its own identifier back before sending an *elected* message
  - Therefore the *election* message containing that identifier must have travelled the entire ring
  - And must therefore have been compared with all process identifiers
  - Since no process updates its $elected_i$ variable until it wins the election or receives an *elected* message no participating process will have its $elected_i$ variable set to anything other than $\perp$
- Liveness:
  - Since there are no failures the liveness property follows from the guaranteed traversals of the ring.

Ring-based Election Algorithm

Performance

- If only a single process starts the election
- Once the process with the highest identifier sends its *election* message (either initiating or because it received one), then the election will consume two full traversals of the ring.
- In the best case, the process with the highest identifier initiated the election, it will take $2 \times N$ messages
- The worst case is when the process with the highest identifier is the nearest anti-clockwise peer from the initiating process In which case it is $(N - 1) + 2 \times N$ messages
- Or $3N - 1$ messages
- The turn-around time is also $3N - 1$ since all the messages are serialised

Elections

The Bully Election Algorithm

- Developed to allow processes to fail/crash during an election
- Important since the current nominee crashing is a common cause for initiating an election
- Big assumption, we assume that all processes know ahead of time, all processes with higher process identifiers
- This can therefore not be used *alone* to elect based on some dynamic property
- There are three kinds of messages in the Bully algorithm
  1. *election* — sent to announce an election
  2. *answer* — sent in response to an election message
  3. *coordinator* — sent to announce the identity of the elected process

The Bully Election Algorithm

Failure Detector

- We are assuming a synchronous system here and so we can build a reliable failure detector
- We assume that message delivery times are bound by T_{trans}
- Further that message processing time is bound by $T_{process}$
- Hence a failure detector can send a process $p_{suspect}$ a message and expect a response within time $T = 2 \times T_{trans} + T_{process}$
- If a response does not occur within that time, the local failure detector can report that the process $p_{suspect}$ has failed

The Bully Election Algorithm

A simple election

- If the process with the highest identifier is still available
- It knows that it is the process with the highest identifier
- It can therefore elect itself by simply sending a *coordinator* message

The Bully Election Algorithm

A simple election

- If the process with the highest identifier is still available
- It knows that it is the process with the highest identifier
- It can therefore elect itself by simply sending a *coordinator* message
- You may wonder why it would ever need to do this
- Imagine a process which can be initiated by any process, but requires some coordinator
  - For example global garbage collection
  - For which we run a global snapshot algorithm
  - And then require a coordinator to:
    1. collect the global state
    2. figure out which objects may be deleted
    3. alert the processes which own those objects to delete them
- The initiator process cannot be sure that the previous coordinator has not failed since the previous run.
- Hence an election is run each time

The Bully Election Algorithm

An actual election

- A process which does not have the highest identifier:
- Begins an election by sending an *election* message to all processes with a higher identifier
- It then awaits the *answer* message from at least one of those processes
- If none arrive within our time bound $T = 2 \times T_{trans} + T_{process}$
  - Our initiator process assumes itself to be the process with the highest identifier who is still alive
  - And therefore sends a *coordinator* message indicating itself to be the newly elected coordinator
- otherwise The process assumes that a *coordinator* message will follow. It may set a timeout for this *coordinator* message to arrive.
- If the timeout is reached before the *coordinator* message arrives the process can begin a new election

The Bully Election Algorithm

Receiving Messages

- *coordinator* If a process receives a *coordinator* message it sets the $elected_i$ variable to the named winner
- *election* If a process receives an *election* message it sends back an *answer* message and begins another election (unless it has already begun one).

The Bully Election Algorithm

Starting a process

- When a process fails a new process may be started to replace it
- When a new process is started it calls for a new election
- If it is the process with the highest identifier this will be a simple election in which it simply sends a *coordinator* message to elect itself
- This is the origin of the name: Bully

The Bully Election Algorithm

Properties

- The *Liveness* property is satisfied.
  - Some processes may only participate in the sense that they receive a *coordinator* message
  - But all non-crashed processes will have set $elected_i$ to something other than $\perp$.
- The *Safety* property is also satisfied if we assume that any process which has crashed, either before or during the election, is not replaced with another process with the same identifier during the election.
- *Total Safety* is not satisfied

The Bully Election Algorithm

Properties

- Unfortunately the *Safety* property is not met if processes may be replaced during a run of the election
  - One process, say p_1, with the highest identifier may be started just as another process p_2 has determined that it is currently the process with the highest identifier
  - In this case both these processes p_1 and p_2 will concurrently send *coordinator* messages announcing themselves as the new coordinator
  - Since there is no guarantee as to the delivery order of messages two other processes may receive these in a different order
  - such that say: p_3 believes the coordinator is p_2 whilst p_4 believes the coordinator is p_1.
- Of course things can also go wrong if the assumption of a synchronous system is incorrect

The Bully Election Algorithm

Performance Evaluation

- In the best case the process with the current highest identifier calls the election
  - It requires (N - 1) *coordinator* messages
  - These are concurrent though so the turnaround time is 1 message
- In the worst case though we require $\mathcal{O}(N^2)$ messages
  - This is the case if the process with the lowest identifier calls for the election
  - In this case $N - 1$ processes all begin elections with processes with higher identifiers
- The turn around time is best if the process with the highest identifier is still alive. In which case it is comparable to a round-trip time.
- Otherwise the turn around time depends on the time bounds for message delivery and processing

Election Algorithms Comparision

Ring-based vs Bully

	Ring Based	Bully
Asynchronous	Yes	No
Allows processes to crash	No	Yes
Satisfies Safety	Yes	Yes/No
Dynamic process identifiers	Yes	No
Dynamic configuration of processes	Maybe	Maybe
Best case performance	$2 \times N$	$N - 1$
Worst case performance	$3 \times N - 1$	$\mathcal{O}(N^2)$

Global Agreement

MultiCast

- Previously we encountered group multicast
- IP multicast and Xcast both delivered "Maybe" semantics
- That is, perhaps some of the recipients of a multicast message receive it and perhaps not
- Here we look at ways in which we can ensure that all members of a group have received a message
- And also that multiples of such messages are received in the correct order
- This is a form of global consensus

Global Agreement

Assumptions and Scenario

- We will assume a known group of individual processes
- Communication between processes is
  - message based
  - one-to-one
  - reliable
- Processes may fail, but only by crashing
  - That is, we suffer from process omission errors but not process arbitrary errors
- Our goal is to implement a $multicast(g, m)$ operation
- Where m is a message and g is the group of processes which should receive the message m

Global Agreement

deliver and receive

- We will use the operation *deliver*(m)
- This delivers the multicast message m to the application layer of the calling process
- This is to distinguish it from the *receive* operation
- In order to implement some failure semantics not all multicast messages received at process p are delivered to the application layer

Global Agreement

Reliable Multicast

- Reliable multicast, with respect to a multicast operation $multicast(g, m)$, has three properties:
  1. Integrity — A correct process $p \in g$ delivers a message m at most once and m was multicast by some correct process
  2. Validity — If a correct process multicasts message m then some correct process in g will eventually deliver m
  3. Agreement — If a correct process delivers m then all other correct processes in group g will deliver m
- Validity and Agreement together give the property that if a correct process which multicasts a message it will eventually be delivered at all correct processes

Global Agreement

Basic Multicast

- Suppose we have a reliable one-to-one $send(p, m)$ operation
- We can implement a *Basic Multicast*: $Bmulticast(g, m)$ with a corresponding *Bdeliver* operation as:
  1. $Bmulticast(g, m) =$ for each process p in g:
     - $send(p, m)$
  2. On $receive(m)$: $Bdeliver(m)$
- This works because we can be sure that all messages will eventually receive the multicast message since $send(p, m)$ is reliable
- It does however depend upon the multicasting process not crashing
- Therefore *Bmulticast* does not have the *Agreement* property

Global Agreement

Reliable Multicast

- We will now implement reliable multicast on top of basic multicast
- This is a good example of protocol layering
- We will implement the operations:
- $Rmulticast(g, m)$ and $Rdeliver(m)$
- which are analogous to their $Bmulticast(g, m)$ and $Bdeliver(m)$ counterparts but have additionally the Agreement property

Global Agreement

Reliable Multicast — Using Basic Multicast

- On initialisation: $Received = \{\}$
- Process p to $Rmulticast(g, m)$:
  - $Bmulticast(g \cup p, m)$
- On $Bdeliver(m)$ at process q:
  - If $m \notin Received$
    - $Received = Received \cup \{m\}$
    - If $p \neq q : Bmulticast(g, m)$
    - $Rdeliver(m)$

Global Agreement

Reliable Multicast

- Note that we insist that the sending process is in the receiving group, hence:
- *Validity* — is satisfied since the sending process p will deliver to itself
- *Integrity* — is guaranteed because of the integrity of the underlying *Bmulticast* operation in addition to the rule that m is only added to *Received* at most once
- *Agreement* — follows from the fact that every correct process that $Bdelivers(m)$ then performs a $Bmulticast(g, m)$ before it $Rdelivers(m)$.
- However it is somewhat inefficient since each message is sent to each process $\mid g \mid$ times.

Global Agreement

Reliable Multicast Over IP

- So far our multicast (and indeed most of our algorithms) have been described in a vacuum devoid of other communication
- In a real system of course there is other communication going on
- So a reasonable method of implementing reliable multicast is to piggy-back acknowledgements on the back of other messages
- Additionally the concept of a "negative acknowledgement" is used
- A negative acknowledgement is a response indicating that we believe a message has been missed/dropped

Global Agreement

$\frac{\textit{Reliable Multicast}}{\textit{IP}}$

- We assume that groups are closed — not something assumed for the previous algorithm
- When a process p performs an $Rmulticast(g, m)$ it includes in the message:
  - a sequence number S_g^p
  - acknowledgements of the form $\{q, R_g^q\}$
- An acknowledgement $\{q, R_g^q\}$ included in message from process p indicates the latest message multicast from process q that p has delivered.
- So each process p maintains a sequence number R_g^q for every other process q in the group g indicating the messages received from q
- Having performed the multicast of a message with an S_g^p value and any acknowledgements attached, process p then increments its own stored value of S_g^p
- In other words: S_g^p is a sequence number

Global Agreement

- The sequence numbers S_g^p attached to each multicast message, allows the recipients to learn about messages which they have missed
- A process q can $Rdeliver(m)$ only if the sequence number $S_g^p = R_g^p + 1$.
- Immediately following $Rdeliver(m)$ the value R_g^p is incremented
- If an arriving message has a number $S \leq R_g^p$ then process q knows that it has already performed *Rdeliver* on that message and can safely discard it
- If $S > R_g^p$ then the receiving process q knows that it has missed some message from p destined for the group g
- In this case the receiving process q puts the message in a *hold-back* queue and sends a negative acknowledgement to the sending process p requesting the missing message(s)

Global Agreement

Properties

- The hold-back queue is not strictly necessary but it simplifies things since then a simple number can represent all messages that have been delivered
- We assume that IP-multicast can detect message corruption (for which it uses checksums)
- Integrity is therefore satisfied since we can detect duplicates and delete them without delivery
- Validity property holds again because the sending process is in the group and so at least that will deliver the message
- Agreement only holds if messages amongst the group are sent indefinitely and if sent messages are retained (for re-sending) until all groups have acknowledged receipt of it
- Therefore as it stands *Agreement* does not formally hold, though in practice the simple protocol can be modified to give acceptable guarantees of *Agreement*

Global Agreement

Uniform Agreement

- Our Agreement property specifies that if any correct process delivers a message *m* then *all* correct processes deliver the message *m*
- It says nothing about what happens to a failed process
- We can strengthen the condition to Uniform Agreement
- Uniform Agreement states that if a process, whether it then fails or not, delivers a message *m*, then all correct processes also deliver *m*.
- A moment's reflection shows how useful this is, if a process could take some action that put it in an inconsistent state and then fail, recovery would be difficult
- For example applying an update that not all other processes receive

Global Agreement

Ordering

- There are several different ordering schemes for multicast
- The three main distinctions are:
  1. FIFO — If a correct process performs $mulitcast(g, m)$ and then $multicast(g, m')$ then every correct process which delivers m' will deliver m before m'
  2. Causal — If $mulitcast(g, m) \rightarrow multicast(g, m')$ then every process which delivers m' delivers m before m'
  3. Total — If a correct process delivers m before it delivers m' then every correct process which delivers m' delivers m before m'

Global Agreement

Ordering

- There are several different ordering schemes for multicast
- The three main distinctions are:
  1. FIFO — If a correct process performs $mulitcast(g, m)$ and then $multicast(g, m')$ then every correct process which delivers m' will deliver m before m'
  2. Causal — If $mulitcast(g, m) \rightarrow multicast(g, m')$ then every process which delivers m' delivers m before m'
  3. Total — If a correct process delivers m before it delivers m' then every correct process which delivers m' delivers m before m'
- Note that Causal ordering implies FIFO ordering
- None of these require or imply *reliable* multicast

Global Agreement

Total Ordering

- As we saw Causal ordering implies FIFO ordering
- But Total ordering is an orthogonal requirement
- Total ordering only requires an ordering on the delivery order, but that ordering says nothing of the order in which messages were sent
- Hence Total ordering can be combined with FIFO and Causal ordering
- FIFO-Total ordering or Causal-Total ordering

Multicast Ordering

Implementing FIFO Ordering

- Our previous algorithm for reliable multicasting
- More generally sequence numbers are used to ensure FIFO ordering

Multicast Ordering

Implementing Causal Ordering

- To implement Causal ordering on top of Basic Multicast (*bmulticast*)
- Each process maintains a vector clock
- To send a Causal Ordered multicast a process first uses a *bmulticast*
- When a process p_i performs a *bdeliver*(m) that was multicast by a process p_j it places it in the holding queue until:
  - It has delivered any earlier message sent by p_j
  - and
  - It has delivered any message that had been delivered at p_j before p_j multicast m
- Both of these conditions can be determined by examining the vector timestamps

Global Agreement

Implementing Total Ordering

- There are two techniques to implementing Total Ordering:
  1. Using a sequencer process
  2. Using *bmulticast* to illicit proposed sequence numbers from all receivers

Implementing Total Ordering

Using a sequencer

- Using a sequencer process is straight forward
- To total-ordered multicast a message *m* a process *p* first sends the message to the sequencer
- The sequencer can determine message sequence numbers based purely on the order in which they arrive at the sequencer
  - Though it could also use process sequence numbers or Lamport timestamps should we wish to, for example, provide FIFO-Total or Causal-Total ordering
- Once determined, the sequencer can either *bmulticast* the message itself
- Or, to reduce the load on the sequencer, it may just respond to process *p* with the sequence number which then itself performs the *bmulticast*

Implementing Total Ordering

Using Collective Agreement

- To total-order multicast a message, the process *p* first performs a *bmulticast* to the group
- Each process then responds with a proposal for the agreed sequence number
  - And puts the message in its hold-back queue with the suggested sequence number provisionally in place
- Once the process *p* receives all such responses it selects the largest proposed sequence number and replies to each process (or uses *bmulticast*) with the agreed upon value
- Each receiving process then uses this agreed sequence number to deliver (that is TO-deliver) the message at the correct point

Ordered Multicast

Overlapping Groups

- So far we have been happy to assume that each receiving process belongs to exactly one multicast group
- Or that for overlapping groups the order is unimportant
- For some applications this is insufficient and our orderings can be updated to account for overlapping groups

Ordered Multicast

Overlapping Groups

- Global FIFO Ordering If a correct process issues $multicast(g, m)$ and then $multicast(g', m')$ then every correct process in $g \cap g'$ that delivers m' delivers m before m'
- Global Causal Ordering If $multicast(g, m) \rightarrow multicast(g', m')$ then every correct process in $g \cap g'$ that delivers m' delivers m before m'
- Pairwise Total Ordering If a correct process delivers message m sent to g before it delivers m' sent to g' then every correct process in $g \cap g'$ which delivers m' delivers m before m'

Ordered Multicast

Overlapping Groups

- Global FIFO Ordering If a correct process issues $multicast(g, m)$ and then $multicast(g', m')$ then every correct process in $g \cap g'$ that delivers m' delivers m before m'
- Global Causal Ordering If $multicast(g, m) \rightarrow multicast(g', m')$ then every correct process in $g \cap g'$ that delivers m' delivers m before m'
- Pairwise Total Ordering If a correct process delivers message m sent to g before it delivers m' sent to g' then every correct process in $g \cap g'$ which delivers m' delivers m before m'
- A simple, but inefficient way, to do this is force all multicasts to be to the group $g \cup g'$, receiving processes then simply ignore the multicast messages not intended for them.
- e.g. process $p \in g - g'$ ignore multicast messages sent to g'

Summary

Further Thoughts

- These algorithms to perform mutual exclusion, nominee election and agreed multicast suffer many drawbacks
- Many are subject to some assumptions which may be unreasonable
- Particularly when the network used is not a Local Area Network
- These problems can be, and are, overcome
- But for each individual application the designer should consider whether the assumptions are a problem
- It may be that coming up with a solution which is less optimal but does not rely on, say, a reliable communication network, may be the best approach
- For example, Routing Information Protocol

Consensus

Three Kinds

- The problems of mutual exclusion, electing a nominee and multicast are all instances of the more general problem of consensus.
- Consensus problems more generally then are described as one of three kinds:
  1. Consensus
  2. Byzantine Generals
  3. Interactive Consensus

Global Agreement

Consensus

- A set of processes $\{p_1, p_2, \ldots p_n\}$ each begins in the *undecided* state
- Each proposes a single value v_i
- The processes then communicate, exchanging values
- To conclude, each process must set their decision variable d_i to one value and thus enter the *decided* state
- Three desired properties:
  - Termination: each process sets its $decision_i$ variable
  - Agreement: If p_i and p_j are correct processes and have both entered the *decided* state, then $d_i = d_j$
  - Integrity: If the correct processes all proposed the same value v, then any correct process p_i in the *decided* state has $d_i = v$

Global Agreement

Byzantine Generals

- Imagine three or more generals are to decide whether or not to attack
- We assume that there is a commander who issues the order
- The others must decide whether or not to attack
- Either the lieutenants or the commander can be faulty and thus send incorrect values
- Three desired properties:
  - Termination: each process sets its $decision_i$ variable
  - Agreement: If p_i and p_j are correct processes and have both entered the *decided* state, then $d_i = d_j$
  - Integrity: If the commander is correct then all correct processes decide on the value proposed by the commander
- When the commander is correct, *Integrity* implies *Agreement*, but the commander may not be correct

Global Agreement

Interactive Consensus

- Each process proposes its own value and the goal is for each process to agree on a vector of values
- Similar to consensus other than that each process contributes only a part of the final answer which we call the *decision vector*
- Three desired properties:
  - Termination: each process sets its $decision_i$ variable
  - Agreement: The final decision vector of all processes is the same
  - Integrity: If p_i is correct and proposes v_i then all correct processes decide on v_i as the ith component of the decision vector

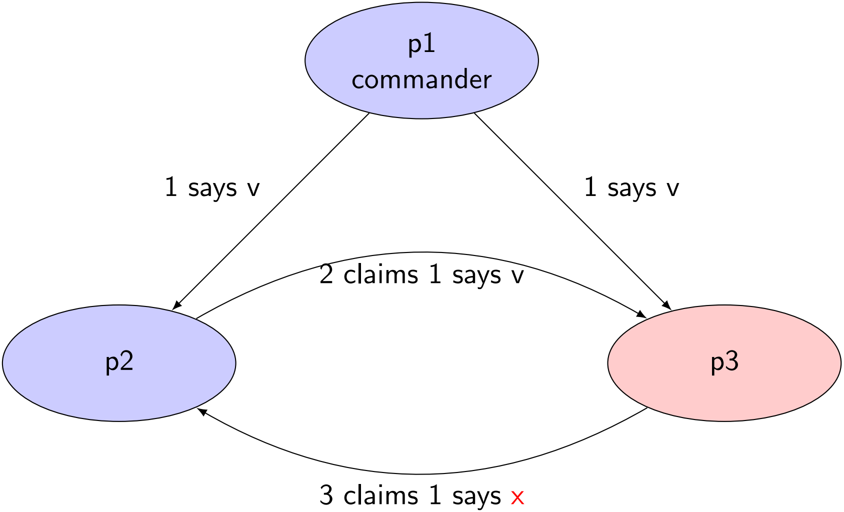
Global Agreement

Relating the three

- Assuming we had a solution to any of the three problems we could construct a solution to the other two
- For example, if we have a solution to Interactive Consensus, then we have a solution to Consensus, all we require is some way consistent function for choosing a single component of the decision vector
  - We might choose a majority function, maximum, minimum or some other function depending on the application
  - It only requires that the function is context independent
- If we have a solution to the Byzantine Generals then we can construct a solution to Interactive Consensus
  - To do so we simply run the Byzantine Generals solution N times, once for each process
- The point is not necessarily that this would be the way to implement such as solution (it may not be efficient)
  - However if we can determine an impossibility result for one of these problems we know that we also have the same result for the others

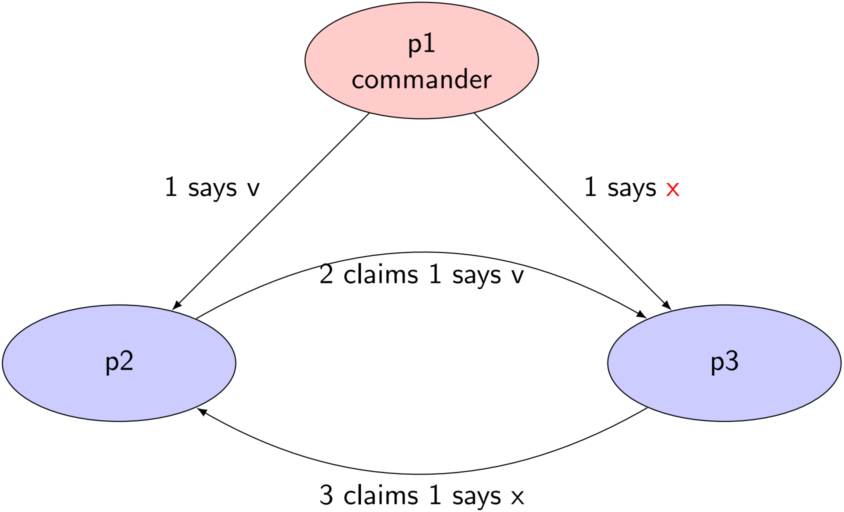
Global Agreement

Byzantine Generals in a Synchronous System



Global Agreement

Byzantine Generals in a Synchronous System



Global Agreement

Impossible

- Recall:
  - Agreement: If p_i and p_j are correct processes and have both entered the *decided* state, then $d_i = d_j$
  - Integrity: If the commander is correct then all correct processes decide on the value proposed by the commander
- In both scenarios, process p_2 receives different values from the commander p_1 and the other process p_3
- It can therefore know that one process is faulty but cannot know which one
- By the *Integrity* property then it is bound to choose the value given by the commander
- By symmetry the process p_3 is in the same situation when the commander is faulty.
- Hence when the commander is faulty there is no way to satisfy the *Agreement* property, so no solution exists for three processes

Global Agreement

$N \leq 3 \times f$

- In the above case we had three processes and at most one incorrect process, hence $N = 3$ and $f = 1$
- It has been shown, by Pease *et al* that more generally no solution can exist whenever $N \leq 3 \times f$
- However there can exist a solution whenever $N > 3 \times f$
- Such algorithms consist of *rounds* of messages
- It is known that such algorithms require at least $f + 1$ message rounds
- The complexity and cost of such algorithms suggest that they are only applicable where the threat is great
- That means either the threat of an incorrect or malicious process is great
- and/or the cost of failing due to inability to reach consensus is large

Global Agreement

Consensus in an Asynchronous System

- Fisher *et al* have shown that it is impossible to design an algorithm which is guaranteed to reach consensus in an asynchronous system, under the following condition:

Global Agreement

Consensus in an Asynchronous System

- Fisher *et al* have shown that it is impossible to design an algorithm which is guaranteed to reach consensus in an asynchronous system, under the following condition:
  - We allow a single process crash failure
- Even if we have 1000s of processes, and the failure is a crash rather than an arbitrary failure of just a single process, any consensus algorithm is not guaranteed to reach consensus
- Clearly this is a pretty benign set of circumstances
- We therefore know that there is no solution in an asynchronous system to either:
  1. Byzantine generals (and hence consensus or interactive consensus)
  2. Totally order and reliable multicast

Consensus in an Asynchronous System

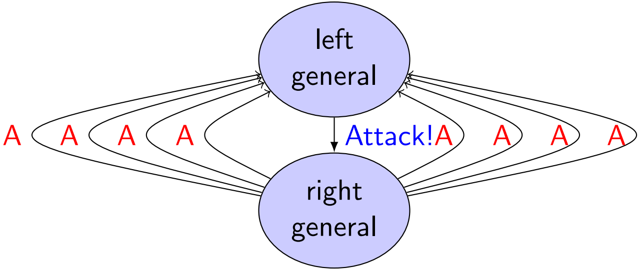
So what to do?

- The important word in the previous impossibility result is: guarantee
- There is no algorithm which is guaranteed to reach consensus
- Consensus has been reached in asynchronous systems for years
- Some techniques for getting around the impossibility result:
  - Masking process failures, for example using persistant storage such that a crashed process can be replaced by one in effectively the same state
    - Thus meaning some operations appear to take a long time, but all operations do eventually complete
  - Employ failure detectors:
    - Although in an asynchronous system we cannot achieve a reliable failure detector
    - We can use one which is "perfect by design"
    - Once a process is deemed to have failed, any subsequent messages that it does send (showing that it had not failed) are ignored
    - To do this the other processes must agree that a given process has failed

Consensus in an Asynchronous System

Back to the pair of attacking generals

A = Acknowledge Attack!



- If the probability of any one message being dropped is 0.5
- Then the probability that two acknowledgements fail to be returned is 0.25
- For 3 it is 0.125 etc, for 8 it is $\frac{1}{256} = 0.0039$
- In reality we have to consider the probability that the message is not dropped but not received by some time out value t
- This complicates the calculation but not the general idea

Summary

- We looked at the problem of Mutual Exclusion in a distributed system
  - Giving four algorithms:
    1. Central server algorithm
    2. Ring-based algorithm
    3. Ricart and Agrawala's algorithm
    4. Maekawa's voting algorithm
  - Each had different characteristics for:
    1. Performance, in terms of bandwidth and time
    2. Guarantees, largely the difficulty of providing the *Fairness* property
    3. Tolerance to process crashes
- We then looked at two algorithms for electing a master or nominee process
- Then we looked at providing multicast with a variety of guarantees in terms of delivery and delivery order

Summary

- We then noted that these were all specialised versions of the more general case of obtaining consensus
- We defined three general cases for consensus which could be used for the above three problems
- We noted that a synchronous system can make some guarantee about reaching consensus in the existance of a limited number of process failures
- But that even a single process failure limits our ability to guarantee reaching consensus in an asynchronous system
- In reality we live with this impossibility and try to figure out ways to minimise the damage

Any Questions

Any Questions?

Distributed Systems — Distribution and Operating Systems

Allan Clark

School of Informatics
University of Edinburgh

http://www.inf.ed.ac.uk/teaching/courses/ds
Autumn Term 2012

Distribution and Operating Systems

Overview

- This part of the course will be chiefly concerned with the components of a modern operating system which allow for distributed systems
- We will examine the design of an operating system within the context that we expect it to be used as part of a network of communicating peers, even if only as a client
- In particular we will look at providing concurrency of individual processes all running on the same machine
- Concurrency is important because messages take time to send and the machine can do useful work in between messages which may arrive at any time
- An important point is that in general we hope to provide transparency of concurrency, that is each process believes that it has sole use of the machine
- Recent client machines such as smartphones, have, to some extent, shunned this idea

Distribution and Operating Systems

Operating Systems

- An Operating System is a single process which has direct access to the hardware of the machine upon which it is run
- The operating system must therefore provide and manage access to:
  - The processor
  - System memory
  - Storage media
  - Networks
  - Other devices, printers, scanners, coffee machines etc

http://fotis.home.cern.ch/fotis/Coffee.html

Distribution and Operating Systems

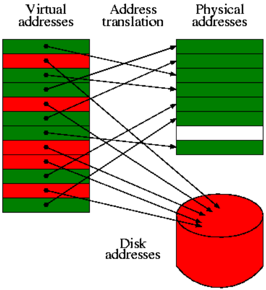
Operating Systems

- As a provider of access to physical resources we are interested in the operating system providing:
  - Encapsulation: Not only should the operating system provide access to physical resources but also hide their low-level details behind a useful abstraction that applications can use to get work done
  - Concurrent Processing: Applications may access these physcial resources (including the processor) concurrently, and the process manager is responsible for achieving concurrency transparency
  - Protection: Physical resources should only be accessed by processes with the correct permissions and then only in safe ways. Files for example can only be accessed by applications started by users with the correct permissions.

Distribution and Operating Systems

Encapsulation

- For example application programmers work with "files" and "sockets" rather than "disk blocks" and "raw network access"
- Application programmers work as though the system memory was limitless (though not costless) and the operating system provides the concept of virtual memory to emulate the existance of more memory



Distribution and Operating Systems

Concurrent Processing

- Through encapsulation applications operate as though they had full use of the computer's hardware
- It is the task of the operating system not only to maintain this pretence but also fully utilise the machine's hardware
- In general Input/Output requests take a relatively long time to process, for example saving to persistent storage
- When a particular program makes such a request it is placed in the "*BLOCKED*" state and another process is given use of the machine's CPU
- In this way the machine's CPU should never be idle whilst some process wishes to do some useful processesing
- The operating system also must provide ways for separate processes to communicate with one another

Protection

- The aim of protection within an operating system is to make sure that a single process cannot unduly disrupt the running of other processes or the physical resources that they share
- The process from which we require protection may be either faulty or deliberately malicious
- There are two kinds of operations from which the operating system can protect the physical resources
  1. Unauthorised access
     - As an example using the file system, the operating system does not allow a process to update (write to) a file for which the owner (a user) of the process does not have write access to
  2. Invalid operations
     - An example again using the file system would be that a process is not allowed to arbitrarily set the file pointer to some arbitrary value

Kernel Mode

- Most processors have two modes of operation: *Kernel mode* and *User mode*, also known as: *priviledged mode* and *unpriviledged mode*
- Generally operating system writers try to write code so that as little as possible is run in *Kernel mode*
- Even other parts of the operating system itself may be run in *User Mode*, thus providing protection even from parts of the operating system
- Although there is sometimes a performance penalty for operating in *User Mode* as there is a penalty for a so-called *system call*
- There have been some attempts to avoid this, such as Typed Assembly Language, in which such code is type-safe and hence can be trusted (more) to run in *Kernel mode*.

Distribution and Operating Systems

Operating System Components

- Process Manager: Takes care of the creation of processes. Including the scheduling of each process to physical resources (such as the CPU)
- Thread Manager: Thread, creation, synchronisation and scheduling.
- Communication Manager: Manages the communication between separate processes (or threads attached to separate processes).
- Memory Management: Management of physical and virtual memory. Note this is *not* the same as automatic memory management (or garbage collection) provided by the runtime for some high-level languages such as Java.
- Supervisor: The controller for interrupts, system call traps and other kinds of exceptions (though not, generally, language level exceptions).

Monolithic vs Microkernel

- A monolithic kernel provides all of the above services via a single image, that is a single program initialised when the computer boots
- A microkernel instead implements only the absolute minimum: Basic virtual memory, Basic scheduling and Inter-process communication
- All other services such as device drivers, the file system, networking etc are implemented as user-level server processes that communicate with each other and the kernel via IPC
- `www.dina.dk/~abraham/Linus_vs_Tanenbaum.html` Historical spat between Andrew Tanenbaum and Linus Torvalds (and others) on the merits of *Minix* (a microkernel) and *Linux* (a monolithic kernel)
- *Linux* and *Minix* are both examples of a Network Operating System. Also mentioned in the above is *Amoeba*, an example of a Distributed Operating System

Monolithic vs Microkernel

The Microkernel Approach

- The major advantages of the microkernel approach include:
  - Extensibility — major functionality can be added without modifying the core kernel of the operating system
  - Modularity — the different functions of the operating system can be forced into modularity behind memory protection barriers. A monolithic kernel must use programming language features or code conventions to *attempt* to ensure this
  - Robustness — relatively small kernel might be likely to contain fewer bugs than a larger program, however, this point is rather contentious
  - Portability — since only a small portion of the operating system, its smaller kernel, relies on the particulars of a given machine it is easier to port to a new machine architecture
  - Not just an architecture, a different purpose, such as mainframe server or a smartphone

Distribution and Operating Systems

The Monolithic Approach

- The major advantage of the monolithic approach is the relative efficiency with which operations may be invoked
- Since services share an address space with the core of the kernel they need not make system calls to access core-kernel functionality
- Most operating systems in use today are a kind of hybrid solution
- *Linux* is a monolithic kernel, but modules may be dynamically loaded and unloaded at run time.
- *Mac OS X* and *iOS* are built around the *Darwin* core, which is based upon the *XNU* hybrid kernel that includes the *Mach* micro-kernel.

Distribution and Operating Systems

Network vs Distributed Operating Systems

- Network Operating Systems:
  - There is an operating system image at each node
  - Each node therefore has control over which processes run at that physcial location
  - A user may invoke a process on another node, for example via `ssh`, but the operating system at the user's node has no control over the processes running at the remote node
- Distributed Operating Systems:
  - Provides the view of a single system image maintaining all processes running at every node
  - A process, when invoked, or during its run, may be moved to a different node in the network
  - Generally the reason for this is that the current node is more computationally loaded than the target node
  - It could also be that the target node is physically closer to some physical resource required by the process
  - The idea is to maximise the configuration of processes to nodes in a way which is completely transparent to the user

Network vs Distributed Operating Systems

- Today there are no distributed operating systems in general use
- Part of this may be down mostly to momentum
  - In a similar way to CISC vs RISC processors back in the 90s
- Part of it though is likely due to users simply preferring to maintain some control over their own resources
- In particular everyone believes their applications to be of higher priority than their neighbours'
- In contrast the Network Operating System provides a good balance as stand-alone applications can be run on the users' own machine whilst the network services allow them to explicitly take advantage of other machines when appropriate

Processes

- A process within a computer system is a separate entity which may be scheduled to be run on a CPU by the operating system
- It has attached to it an execution environment consisting of: its own code, its own memory state and higher-level resources such as open files and windows
- Each time the kernel performs a context-switch, allowing a different process to run on the CPU, the old execution environment is switched out and is replaced with the new one
- Several processes, or execution environments, may reside in memory simultaneously.
- However each process believes it has sole use of memory and hence accesses to memory go through a mapping, which maps the accessed address to the address at which it currently, physically resides
- In this way the OS can move execution environments about in memory and even out to disk

Distribution and Operating Systems

Processes and Threads

- Traditionally processes were used by computers to perform separate tasks
- Even a single application could be split into several related processes that communicated amongst each other
- However, for many purposes these separate processes meant that sharing between related activities was awkward and expensive
- For example a server application might have a separate process to handle each incoming request (possibly setting up a connection)
- But each such process was running the same code and possibly using the same resources to handle the incoming requests (such as a set of static web-pages for example)

Distribution and Operating Systems

Threads

- Hence separate processes were inappropriate for such tasks
- An early work-around was for the application to write its own basic 'sub-process scheduler'
- For example allowing a request object time to run before 'switching' to the next request object
- But this was throwing out a lot of the advantages of operating system level separate processes
- So threads were introduced as a lightweight - operating system provided, alternative
- Now a process consists of its address-space, and a set of threads attached to that process
- The operating system can perform less expensive context switches between threads attached to the same process
- And threads attached to the same process can access the same memory etc, such that communication/synchronisation can be much cheaper and less awkward

Processes and Threads

Shared Memory

- A server application generally consists of:
  - A single thread, the receiver-thread which receives all the requests, places them in a queue and dispatches those requests to be dealt with by the
  - worker-threads
- The worker-thread which deals with the request may be a thread in the same process or it may be a thread in another process
- There must be a portion of shared memory though, for the queue resides in memory owned by the receiver-thread
- A thread in the same process automatically has access to the same part of memory
- If separate processes are used then there must be a portion of shared memory such that the worker-thread can access any request which the receiver-thread has dispatched to it

Threads

A server utilising threads

- Imagine a server application, suppose that the receiver-thread places all incoming requests in a queue accessible by the worker-thread(s)
- Let us suppose that each request takes 2ms of processing and 8ms of Input/Output
- If we have a single worker thread then the maximum throughput of serviced requests is 100 per-second, since each request takes $2ms + 8ms = 10ms$

Threads

A server utilising threads

- Now consider what happens if there are two threads:
  - The second thread can process a second request whilst the first is blocked waiting for Input/Output
  - Under the best conditions each thread may perform its $2ms$ of processing whilst the other thread is blocked waiting for Input/Output
  - In calculating throughput then we can assume that the $2ms$ of processing occurs concurrently with the proceeding request
  - Hence on average each request takes $8ms$ meaning the maximum throughput is $1000/8 = 125$ requests per-second

Threads

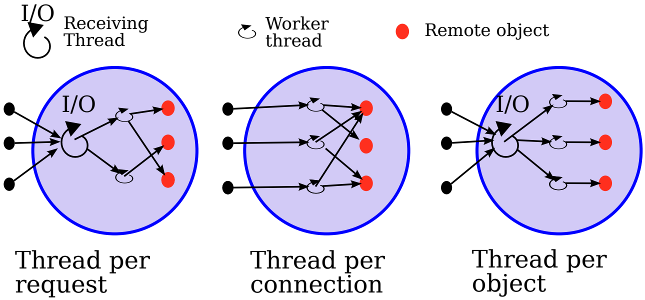
Threading and the Cache

- The cache of the processor is a small piece of hardware which stores recently accessed elements of memory
- Separate processes have separate memory address spaces
  - Hence when a process switch occurs the cache is flushed
- Separate threads belonging to the same process however share the same execution environment
  - Hence when switching between threads belonging to the same process no flush of the cache is performed
  - It's possible then that using threads can reduce the processing time for each individual request, since any access to memory may result in a cache hit even if the current request hasn't accessed the same part of memory

Server Threads

Possible Strategies

- There are three general threading strategies in use for servers
  1. A thread per request
  2. A thread per connection
  3. A thread per server object
- Which one is used depends on the application and in particular whether connections are long-lived and “busy” or not



Thread strategies

- In the thread per-request many threads are created and destroyed, meaning that there is a large amount of thread maintenance overhead
- This can be overcome to some extent by re-using a thread once it has completely finished with a request rather than killing it and starting a new one.
- In the thread per-connection and thread per-object strategies the thread maintenance over-head is lower
- However, the risk is that there may be low utilisation of the CPU, because a particular thread has several waiting requests, whilst other threads have nothing to do
- That one thread with many requests may require to wait for some I/O to be completed, whilst the remaining threads sit idle because they have no waiting requests.
- If you have many concurrent connections (or objects) this may not be a concern

Threads vs Processes

Main Arguments for Threads

- Creating a new thread within an existing process is cheaper than creating a new process
- Switching to a new thread within the same process is cheaper than switching to a thread within a different process
- Threads within the same process can share data and other resources more efficiently and conveniently than threads within separate processes

Main Arguments for Processes

- Threads within the same process are not protected from each other
- In particular they share memory and therefore may modify/delete an object still in use by another thread

Rebuttal

- However modern type-safe languages can provide similar safety guarantees

Threads Implementation

Operating Systems Support vs User Library

- Most major operating systems today support multi-threaded processes allowing the operating system to schedule threads
- Alternatively the OS knows only of separate processes and threading is implemented as a user-level library
- Such an implementation suffers from the following drawbacks:

1. The threads within a process cannot take advantage of a multi-processor
2. When a thread makes a blocking system call (e.g., to access input/output), the entire process is blocked, thus the threaded application cannot take advantage of time spent waiting for I/O to complete
3. Although this can be mitigated by using kernel level non-blocking I/O, other blocks such as a page-fault will still block the entire process
4. Relative prioritisation between processes and their associated threads becomes more awkward

Threads Implementation

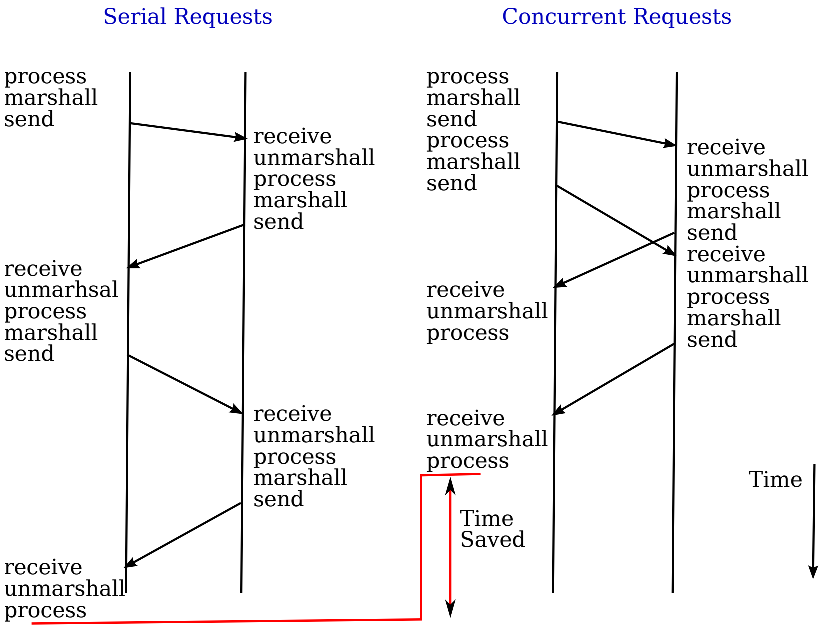
Operating Systems Support vs User Library

- In contrast the thread implementation as a user-level library has the following advantages:
  1. Some operations are faster, for example switching between threads does not automatically require a system call
  2. The thread-scheduling module can be customised for the particular application
  3. Many more user-level threads can be supported than can be by the kernel

Threads in the Client

- Threads are clearly useful for the server what about the client?
- Imagine a web-browser which visits a particular page, the first request is returned with the HTML for the page in question
- Within that HTML may be a number of image tags
- `<img src="smiley.gif" height="42" width="42">`
- The client must then make a further request for each image (some images might not even be hosted at the same server — *hotlinking*)
- But it doesn't particularly matter in which order these requests are made, or, crucially, in which order they are received
- Hence the web-browser can spawn a thread for each image and request them concurrently

Threads in the Client



Distribution and Operating Systems

Communication Primitives

- Some operating systems provide kernel level support for high-level communication primitives such as remote procedure-call, remote method invocation and group communication
- Although this can increase efficiency due a decrease in the required number of systems calls, such communication abstractions are usually left to the middleware
- Operating systems tend to provide the well known sockets abstraction for connection-based communication using TCP and connectionless communication using UDP
- Middleware provides the higher-level communication abstractions since it is then more flexible, different implementations and protocols can be updated more readily than for an entire operating system

Remote Invocation — Performance

- A null invocation is an invocation to a remote procedure which takes zero arguments, executes a null procedure and returns no values
- The time taken for a null invocation between user processes connected by a LAN is of the order of a tenth of a millisecond
- By comparison, using the same sort of computer, a local procedure call takes a small fraction of μ-second — let's say at most 0.0001 milliseconds
- Hence, over the LAN it is around 1000 times slower
- For the null invocation we need to transfer a total of around 100 bytes — over Ethernet it is estimated that the total network time for this is around 0.01 milliseconds

Remote Invocation — Performance

- The observed delay then is $0.0001 + 0.01 + x = 0.1$ where x is the delay accounted for by the operating system and user-level remote procedure-call code
- $x = 0.0899$ — or 89% of the delay
- This was a rough calculation but clearly the operating system and RPC protocol code is responsible for much of the delay
- The cost of a remote invocation increases if we add arguments and return values, but the null invocation provides a measure of the *latency*
- The *latency* can be important since it is often large in comparison to the remainder of the delay
- In particular we frequently wish to know if we should make one remote invocation with large arguments/results or many smaller remote invocations

Distribution and Operating Systems

Latency

- $\mathit{Message\ transmission\ time} = \mathit{latency} + \frac{\mathit{length}}{\mathit{data\ transfer\ rate}}$
- Though longer messages may require segmentation into multiple messages
- Latency affects small frequent message passing which is common for distributed systems

Distribution and Operating Systems

Virtualisation

- The goal of system virtualisation is to provide multiple virtual machines running on top of the actual physical machine architecture
- Each virtual machine has its own instance of an operating system
- The operating system on each virtual machine need not be the same
- In a similar way in which each operating system schedules the the individual processes the *virtualisation system* manages the allocation of physical resources to the virtual machines which are running atop it

Virtualisation

Why?

- The system of user processes already provides some level of protection for each user against the actions of another user
- System virtualisation offers benefits in terms of increased security and backup
- A user can be charged for the time that their virtual machine is run on the actual physical machine
- It's a good way of running a co-location service, since the user can essentially pay for the virtual machine performance that is required/used rather than a single physical machine
- Sharing a machine is difficult, in particular the upgrade of common libraries and other utilities, but system virtualisation allows each user's machine/process to exist in a microcosm separate to any other user's processes

Virtualisation Use Cases

Server Farms

- An organisation offering several services can assign a single virtual machine to each service
- Virtual machines can then be dynamically assigned to physical servers
- Including the ability to migrate a virtual machine to a different physical server — something not quite so easy to do for a process
- This allows the organisation to reduce the cost of investment in physical servers
- And can help reduce energy requirements as fewer physical servers need be operating in times of low-demand

Virtualisation Use Cases

Cloud Computing

- More and more computing is now being done "in the cloud"
- This is both in terms of "platform as a service" and "software as a service"
- The first can be directly offered via virtualisation as the user can be provided with one or more virtual machines
- Interesting blog post of a developer who ditched his macbook for an ipad and a Linode instance
- `http://yieldthought.com/post/12239282034/swapped-my-macbook-for-an-ipad`

Virtualisation Use Cases

Dynamic Resource Demand

- Developers of distributed applications may require the efficient dynamic allocation of resources
- Virtual machines can be easily created and destroyed with little overhead
- For example online multiplayer games, may require additional servers when the number of hosted games increases

Testing Platforms

- A completely separate use is a single desktop developer of a multiplatform application
- Such a developer can easily run instances of popular operating systems on the same machine and easily switch between them

Virtualisation

Is it my turn to run?

- It is interesting now to note that there are several hierarchical layers of scheduling
- The virtualisation layer decides which virtual machine to run
- The operating system then decides the execution environment of which process to load
- The operating system then decides which thread within the loaded execution environment to run
- If user-level threads are implemented on top of this then the user-level thread library decides which thread object to run

Distribution and Operating Systems

Summary

- Distributed Operating Systems are an ideal allowing processes to be migrated to the physical machine more suitable to run it
- However, Network Operating Systems are the dominant approach, possibly more due to human tendancies than technical merit
- We looked at microkernels and monolithic kernels and noted that despite several advantages true microkernels were not in much use
- This was mostly due to the performance overheads of communication between operating system services and the kernel
- Hence a hybrid approach was common

Distribution and Operating Systems

Summary

- We looked at processes and how they provide concurrency, in particular because such an application requires concurrency because messages can be received at any time and requests take time to complete, time that is best spent doing something useful
- but noted that separate processes were frequently ill-suited for an application communicating within a distributed system
- Hence threads became the mode of concurrency offering lightweight concurrency.
- Multiple threads in the same process share an execution environment and can therefore communicate more efficiently and the operating system can switch between them more efficiently

Summary

- We also looked at the costs of operating system services on remote invocation
- Noting that it is a large factor and any design of a distributed system must take that into account — in particular the choice of protocol is crucial to alleviate as much overhead as possible
- Finally we looked at system virtualisation and noted that it is becoming the common-place approach to providing cloud-based services
- Virtualisation also offers some of the advantages of a microkernel including increased protection from other users' processes

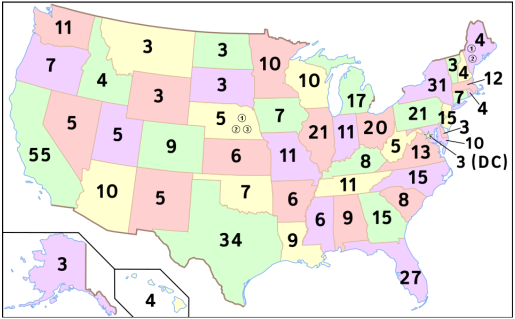
Any Questions

Any Questions?

US Presidential Election

As a distributed system

- For those of you that don't know, the US presidential election is tomorrow November the 6th
- Each state has allocated to it a number of "electoral college" votes based on the size of the population of the state
- Each state then votes and allocates all of the state's electoral college votes to the party with the highest vote share in the state



US Presidential Election

Popular Vote

- I am not arguing that this is a good system
- Why not just take the popular vote?
- That is, count up all the votes in the entire election and the party/candidate with the most votes wins the election?
- Mostly historical reasons, arguably accuracy reasons
-

Candidate	George W. Bush	Al Gore
EC Votes	271	266
Popular Vote	50,456,002	50,999,897
Percentage	47.9	48.4

US Presidential Election

Efficiency

Candidate	George W. Bush	Al Gore
Alaska	167,398	79,004
New York	2,403,374	4,107,697
New Mexico	286,417	286,783
Florida	2,912,790	2,912,253
- In highly partisan states counting need not be accurate
- In highly contested states, maybe we better have a recount
- Note that this means the popular vote may be incorrect, whilst the electoral college vote less likely so
- A statewide vote may order a recount if a candidate wins by less than 1000 votes
- Nationally we might require a margin of at least 100, 000 votes to forego a recount
- A national recount is more expensive than a statewide recount

US Presidential Election

Robustness

- We term each state as either Democrat or Republican
- But as the previous table shows most states are split quite closely
- New Hampshire — `fivethirtyeight.com` projections:

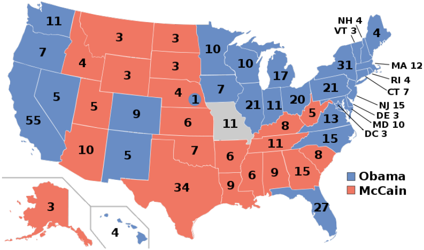
	DEM	REP	MARGIN
Polling average	48.9	46.3	Obama +2.6
Adjusted polling average	49.0	46.2	Obama +2.8
State fundamentals	50.4	44.4	Obama +6.0
Now-cast	49.1	46.0	Obama +3.1
Projected vote share ±3.7	51.2	48.0	Obama +3.2
Chance of winning	80%	20%	

- With the electoral college votes each state's influence is known and limited
- Hence a corrupted state can have only a known and limited effect on the final outcome

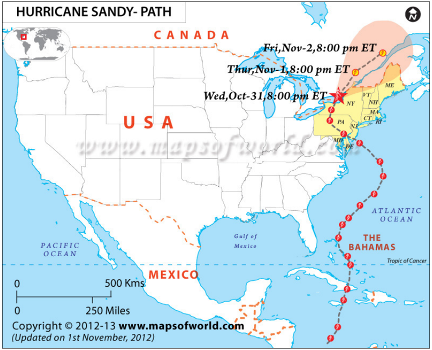
US Presidential Election

Robustness

- This year may see another robustness result come significantly in to play
- Hurricane Sandy has devastated parts of the north east coast



2008 Electoral College Results Map



US Presidential Election

Robustness

- Suppose we had three states, each with a single EC vote
- Each has a population of 1000 voters:
-

State	Dem Votes	Rep Votes
Left Carolina	700	300
North Fencia	550	450
Right Carolina	300	700
Total Pop Vote	1550	1450
Total EC	2	1

US Presidential Election

Robustness

- Now suppose Left Carolina is hit by a hurricane the week before the election, and only 500 people vote
State	Dem Votes	Rep Votes
Left Carolina	350	150
North Fencia	550	450
Right Carolina	300	700
Total Pop Vote	1200	1300
Total EC	2	1

US Presidential Election

Robustness

- Now suppose Left Carolina is hit by a hurricane the week before the election, and only 500 people vote
-

State	Dem Votes	Rep Votes
Left Carolina	350	150
North Fencia	550	450
Right Carolina	300	700
Total Pop Vote	1200	1300
Total EC	2	1

- I'm not arguing that this is a good electoral system
- Just that it has some redeeming qualities
- and that those qualities could be put to use in some distributed algorithm for an application in which the final result need not necessarily be exactly correct, but not horribly wrong

Distributed Systems — Peer-to-Peer

Allan Clark

School of Informatics
University of Edinburgh

http://www.inf.ed.ac.uk/teaching/courses/ds
Autumn Term 2012

Peer-to-Peer Systems

Overview

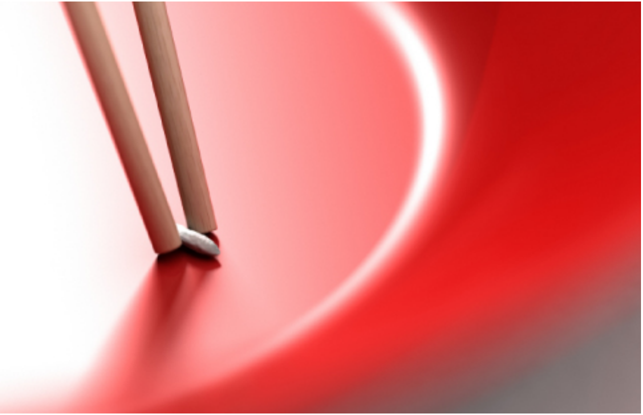
- This section of the course will discuss peer-to-peer systems
- We will look at the motivations for a such a system
- The limitations of a such a system
- Characterstics of such systems and hence the suitable types of applications for peer-to-peer systems
- As well as how to provide middleware frameworks for creating peer-to-peer applications which have the additional difficulty that they must be application agnostic

Google's Daily Processing of Bytes

- Apparently Google (as of around 2009) processes around 24 petabytes of data every day
- This is quite a lot
- How much?

Rice Bytes

- Let's imagine that a single byte is represented by a single grain of rice



Peer-to-Peer Systems

Rice Bytes

- A kilobyte, 1K or 1024 bytes then is a 1024 grains of rice, or about a bowl

Peer-to-Peer Systems

Rice Bytes

- A megabyte then, represented as rice, is a sack of rice:

Peer-to-Peer Systems

Rice Bytes

- Next up is 1024 megabytes, commonly referred to as a gigabyte
- This is represented as two large shipping containers full of rice
- 1 shipping unit = 1 TEU (twenty-foot equivalent unit)
- We could feed everyone in Edinburgh two bowls of rice


Rice Bytes

- So what is a 1024 gigabytes?
- Less well known, but it is a terabyte
- With this many grains of rice we would require 2048 shipping containers
- It is also enough rice to feed a meal to everyone in the European Union (about 500 million people), twice

This particular ship has a capacity of 1618 TEU

Rice Bytes

- The largest container ships are the Mærsk fleet
- Each can carry 15,500 TEU (containers)
- A petabyte is equivalent to 2097152 containers
- Hence we would need 135 of the largest ever container ship.
- Enough to feed everyone on the planet 146 bowls of rice or cover New York City with about a metre of rice

Peer-to-Peer Systems

Rice Bytes

- The largest container ships are the Mærsk fleet
- Each can carry 15,500 TEU (containers)
- A petabyte is equivalent to 2097152 containers
- Hence we would need 135 of the largest ever container ship.
- Enough to feed everyone on the planet 146 bowls of rice or cover New York City with about a metre of rice

Peer-to-Peer Systems

- That's one petabyte, Google gets through 24 or so a day
- Or 1920 bowls of rice for every one of the 7 billion people on the planet today
- Or covering New York City to a depth of 24 metres in rice

thanks to: `http://noiseinmyhead.wordpress.com/2008/06/26/visualizing-huge-amounts-of-data/`

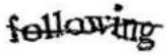
Peer-to-Peer Systems

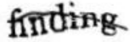
Centralised Servers

- Providing a service via a single centralised named server is an obvious architecture
- It simplifies much of the design
- But it has an obvious flaw, as the number of clients grows so too does the work done by the centralised server
- Even if we had more computer capacity, we may be limited by the available physical bandwidth to that particular site

Peer-to-Peer Systems

A Plausible Solution

- Peer-to-peer systems arose from the realisation that users could contribute some of their own resources to the growing system
- Meaning that as the number of users grows, so too does the number of available resources
- Clay Shirky termed this: exploiting the resources "on the edge of the Internet"
- These resources can be:
  - storage
  - compute cycles
  - bandwidth
  - content
  - human presence
    - 



Google Down

- In a very timely fashion Google was unreachable for around 3-5% of the Internet on Monday evening PST.
- Recall the Routing Information Protocol, it is essentially a trust based protocol
- If a particular router claims to be able to route packets to a particular network which it cannot, some other routers may believe
- If so they start sending packets to a network which will be unable to deliver them
- Hence some hosts, will find the target network unreachable

Border Gateway Protocol

- The RIP is a highly simplified version of what is used throughout the Internet
- Often referred to as BGP or Border Gateway Protocol
- Being more complex allows it to be more robust, but at the same time "route leakage" can occur
- This is when the faulty route is leaked out, such that gateways and routers further afield start to route via the faulty route
- In this case, California couldn't reach Google (located in California) because of a faulty route originating from an ISP in Indonesia
- This was likely due to a "fat fingered" address than a malicious attempt to subvert Google

Peer-to-Peer Systems

Common Features:

1. Their design ensures that each user contributes resources to the system
2. Although their resources may differ, all nodes have the same functionality, capabilities and responsibilities
3. Their correct operation does not depend on the existence of any centrally administered systems
4. They can be designed to offer a limited degree of anonymity to the providers and users of resources
5. A key issue for their efficient operation is the choice of an algorithm for the placement of data (resources) across many hosts and subsequent access to it in a manner that balances the workload and ensures availability without adding undue overheads

Unreliability of Providers

- The owners of the computers sharing resources in a peer-to-peer system may be a variety of individuals and organisations
- None of them provide any level of service guarantee, in particular nodes join and leave the system *at will*
- Leading to unpredictable availability of any particular process/node
- Meaning that the provision of any particular resource should not depend upon the continued availability of any particular node
- Preparing for this requires redundancy in a way which may help against malicious attack or unpredicted outages
- The required redundancy may even help with performance
- As a last resort, we may simply have to put up with unavailability of certain resources

Peer-to-Peer Systems

Popular Uses

- These features mean that a corporations hoping to collect revenue from a service have shied from such systems
- It is difficult to make any kind of service level guarantees
- However peer-to-peer have been very popular for file-sharing systems mostly because such systems do not pretend to offer any particular level of service, they operate a strictly “maybe” policy
- In addition a relatively large level of service can be obtained from very little outlay
- Academics have therefore also been somewhat drawn to peer-to-peer systems

Peer-to-Peer Systems

Distributed Computation

- Peer-to-peer systems are generally associated with the sharing of data resources and the bandwidth required to access those shared data resources, but we noted other resources
- The famous *SETI@Home* project aims to use individuals' spare computing cycles to perform part of the larger computation of analysing received radio signals for intelligent communication
- *SETI@Home* is an interesting example as it does not require communication between individual nodes
- That is, each segment may be analysed in isolation
- A brand of computation that is termed "*embarrasingly parallelisable*"
- Utilising the Internet's vast array of computers for a broader range of tasks will depend upon the development of a distributed platform which supports communication between participating nodes

Peer-to-Peer Systems

Distributed Computation

- There is a further threat to the platform of distributed computing
- Climate Change
- When distributed computing first became popular it was seen as a very green use of otherwise idle (but switched on) computers
- Computers at the time used roughly the same amount of energy to remain switched and idle as when doing some calculation
- Hence using those idle computers to do anything remotely useful was seen as a great re-use of resources
- Today though, computers use much less energy when idle and hence running them at full power to perform a large computation is seen as a waste of energy unless that computation is somewhat important

Peer-to-Peer Systems

Three generations

- Although peer-to-peer systems have existed since at least the 1980s, they first really became popular when always-on broadband became generally available (start of this century)
- We can identify three generations of peer-to-peer systems:
  1. Napster music exchange — relied in part on a central server
  2. File sharing systems — with greater fault tolerance and no reliance on a central server, examples include:
     - Gnutella
     - DirectConnect
     - Kazaa
     - Emule
     - Bittorrent
     - FreeNet
  3. The emergence of middleware layers for peer-to-peer systems — making possible the application independent provision of resources

Peer-to-Peer Systems

Napster

- Napster was an early offering in peer-to-peer style systems
- Offering the ability for users to share data files it quickly became popular with those sharing music files
- However Napster was shut down as a result of:
  - People sharing copyrighted music
  - This lead the owners of the copyrighted material to instigate legal proceedings against the Napster service operators
  - This in turn caused the Napster service to be shut down

Napster

Napster's Modus Operandi

- Napster relied upon a central index of files available for download
- Each new peer that joined the network, communicated to the central service a list of all available files
- When a user had a request for a particular file the following steps where executed:
  1. A file location request is made by a user to the centrally managed Napster index
  2. The Napster server responds to the request with a list of peers who have the requested file available
  3. The user then requests that file from one of the list of peers
  4. The peer from which the file is requested then delivers the file directly to the requesting user, without central server intervention
  5. Finally, once the requested file is received by the user it informs the centrally managed Napster server such that the index of files may be updated
     - That is, the requesting user now has the particular file

Napster

Key point

- The indexing system was not distributed (though it was replicated)
- The distributed resources were both the available files
  - In terms of the fact that they are stored on peer computers and not any centrally managed machines
  - Additionally in that they originated from the users themselves
  - And finally the bandwidth available at each peer, since files are delivered straight from peer to peer without going via a central server

Legal Proceedings

- Napster argued that they were not liable for the copyright infringement because they were not part of the copying process
- The argument ultimately failed as the index servers were viewed as an essential part of the copying process
- The index servers were at known network addresses, meaning that their owners could not retain anonymity
- Hence they could be targeted by lawsuits

Napster Lessons and legacy

- Napster performed load balancing, directing user requests to users closer (in terms of network hops) to the requesting user
- Thus avoiding all users requesting a file from the same user
- Napster used a replicated, unified index of all available music files, this didn't represent a huge limitation since there was little requirement for the replicated indexes to be consistent
- But it could be a limitation for another application
- Napster also took advantage of the fact that music files are immutable data resources, they do not get updated
- No guarantees were made about the availability of any particular file. A user made a request which may or may not be satisfied

Napster Lessons and legacy

- Napster then was ultimately shutdown
- But many derivative file-sharing networks live on
- Independence from any centrally managed server makes legal action far harder to pursue and ultimately less potent
- Whatever your views on the sharing of © material it is not particularly difficult to imagine "legitimate" uses
- Many people around the world are opressed in particular without right to the freedom of expression
- Many countries for example do not allow access to Facebook or Twitter
- During the "Arab Spring" the use of sites such as Twitter and Facebook are well known to have been crucial
- Both were blocked by several governments in an attempt to quash an uprising

Peer-to-Peer Middleware

- With the third generation of peer-to-peer systems came about the development of middleware on top of which peer-to-peer systems could be built
- Developing middleware is more problematic than a single application because we cannot take advantage of any application specific assumptions
- Such as the file sharing assumption that there need be no guarantee of the availability of any particular file

Indexing

- Restricting ourselves for the moment to providing access to data resources, a key problem is the indexing of available files to hosts at which those files are available
- Napster, used a central server with a known address
- Gnutella and other second generation peer-to-peer file-sharing systems use a partioned and distributed index
- Both systems made the assumption that different users could have different results when requesting access to a specific resource

Peer-to-Peer Middleware

Functional Requirements

- The aim of peer-to-peer middleware is to simplify the construction of services implemented over widely distributed hosts
- Any node must therefore be able to locate and communicate with any individual resource which is made available
- The system must be able to cope with the arbitrary addition or removal of resources and hosts
- As with all middleware, peer-to-peer middleware (if it is to be widely adopted) must offer a simple/appropriate programming interface

Peer-to-Peer Middleware

Non-functional Requirements

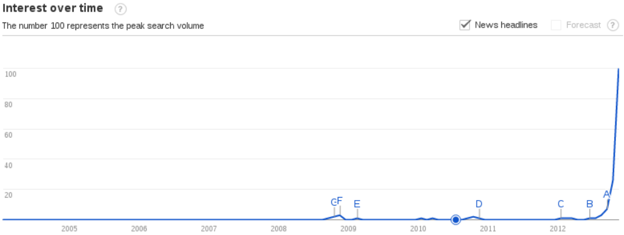
- Global Scalability — the very idea of peer-to-peer systems is to both cope with and exploit large numbers of users. Peer-to-peer systems must therefore be able to support applications that access millions of objects on hundreds of thousands of hosts
- A peer-to-peer system should be able to take advantage of the ability for service provision to grow dynamically as the number of users increase
- In the previous part of the course we saw how system virtualisation can aid a central service in dynamically adjusting service provision but for a peer-to-peer system this should not be necessary

Peer-to-Peer Middleware

Non-functional Requirements

- Load Balancing — The performance of any system exploiting large numbers of hosts, even if those hosts were co-located, depends upon being able to distribute the load across those hosts evenly.
- This can be achieved to some extent by randomly placing resources and replicating heavily used resources

Web searches for “Nate Silver”



Peer-to-Peer Middleware

Non-functional Requirements

- Optimisations for local interactions — The "network distance" between peers has a large impact on the latency of individual interactions. Additionally network traffic is highly impacted if there are many distant interactions
  - We saw an example of this for Napster, that attempted to return to a requesting user, provider hosts which were "network near" to the requesting host

Peer-to-Peer Middleware

Non-functional Requirements

- Accomodating highly adaptable host availability — Most peer-to-peer systems are constructed such that hosts are free to join or leave at any time. Some studies of peer-to-peer networks have shown large turnover in participating hosts. Re-distribution of load when hosts join and leave is a major technical challenge
  - Note that it may even be that all members interested in a particular resource leave, but that that resource should not disapear
  - Consider a peer-to-peer social network, say a peer-to-peer Facebook
  - A single user's profile must be retained even when not only that user has left but also all the friends of that particular user

Peer-to-Peer Middleware

Non-functional Requirements

- Security of Data — Particularly in an environment of heterogeneous trust.
  - File sharing systems do not by their very nature require much of security of data, the whole point is that data is shared
  - Consider again the peer-to-peer version of Facebook
  - A single user's profile must be stored on several machines, but should only be available to a group of authorised users (that user's friends)

Non-functional Requirements

- Anonymity and Deniability — Anonymity is a legitimate concern for many applications
  - In particular situations demanding a resistance to censorship.
  - "whistleblowing" on a company or group
- A related requirement is that hosts demand a root to deniability if they are to be used to store/forward data originating from other users. Otherwise the risk in involving oneself in a peer-to-peer network is high. Here the use of a large number of hosts can actually be an advantage. The key phrase is "plausible deniability"
- Key disclosure laws — some countries inforce that the user supply a key to law enforcement/government representatives for any encrypted data (or enforce mandatory decryption)
- In the UK at least three people have been prosecuted and convicted for refusing to supply decryption keys
- The defence is to "prove" that one does not possess the encryption key or that the data is random

Obvious Solution

- Recall that we want a service such that: Any node is able to locate and communicate with any individual resource which is made available
- The obvious solution is to maintain a database at each node of all resource (objects) of interest
- This isn't going to work though for several reasons:
  1. It does not scale
  2. It involves a heavy amount of traffic to relay all updates to all nodes
  3. Not all nodes are always available, hence re-joining the network would have a heavy cost associated with it
- Knowledge of the locations of all objects must be partitioned and distributed throughout the network
- A high degree of replication is required to counteract the intermittent availability of hosts

Peer-to-Peer Systems

Telephone Trees

- Not so common now since we have convenient broadcast of messages via text or email
- The goal is to broadcast some message to a group of people,
  - generally these were the parents of a group of children
  - the message related to say the ETA back from some group excursion
- Each parent knew the phone numbers of up to four others
- When they received a call giving information, it was then their duty to inform the "branches" of which they knew
- This was, in a sense, a routing overlay, built upon the routing mechanism already in place for the telephone system
- Although of course in this case it was used for broadcasting rather than locating a resource

Peer-to-Peer Systems

GUIDs

- Peer-to-Peer systems usually store multiple copies of any given resource object as a redundancy guard against unavailability of a single copy
- Each object is associated with a GUID (globally unique identifier)
- Each person in the phone-tree did not need to know the names, addresses, or anything about those individuals to which they should forward the call
- They only required to know their GUID, which was in this case their phone number
- GUIDs should be opaque, that is, they reveal nothing about the object to which it refers or its location (see later)
  - In this sense they are nothing like a postal address
  - More *like* your mobile phone number

GUIDS — small aside

- The Open Software Foundation recommends an algorithm for generating GUIDs
- V1 of this algorithm used, as a part of the GUID, the network card *MAC* address
- Meaning that the creator of a GUID (and hence a document to which it is attached) could be determined from the GUID alone
- This fact was used to David L. Smith the person who released the *Melissa* virus into the wild
- He was sentenced to 10 years (serving 20 months) and fined $5000
- V4 of the algorithm does not do this

Peer-to-Peer Systems

Routing Overlays

- A distributed algorithm known as a routing overlay takes responsibility for routing requests to some node which holds the object
- The object of interest may be placed at, and subsequently relocated at any node in the network
- It is termed an overlay since it implements in the client a routing algorithm that is quite separate from the routing of individual IP packets
- The routing overlay ensures that any node can access any object through a sequence of nodes, by exploiting the knowledge at each of them to locate the destination object

Routing Overlays

Main Tasks of the Routing Overlay

- Routing of Requests to Objects
  - A client wishing to perform some act upon a particular object must send that that request, with the GUID attached, through the routing overlay
- Insertion of Objects
  - A node wishing to insert a new object, must compute a new GUID for that object and announce it to that routing overlay such that that object is available to all nodes
- Deletion of Objects
  - When an object is deleted the routing overlay must make it unavailable for other clients
- Node addition and removal
  - Nodes may join and leave the service at will. The routing overlay must organise for new nodes to take over some of the responsibilities of other (hopefully nearby) nodes
  - When a node leaves, the routing overlay must distribute its responsibility to remaining nodes

Overlay Routing

Distributed Hash Tables

- A distributed hash table has three operations:
  1. *put*(*GUID*, *data*): stores data at all nodes responsible for the object identified by GUID
  2. *remove*(*GUID*): deletes all references to GUID and the associated data
  3. *get*(*GUID*) : retrieves the data associated with GUID from one of the nodes responsible for it
- Note then that operations may be subject to mutual-exclusion style race conditions
- A count of something for example involves first retrieving the current count and storing the incremented count. These two operations could clearly be interleaved by two concurrent processes

Distributed Object Location and Routing

- DOLR has the following operations:
  1. *publish*(*GUID*): Makes the node performing the *publish* the host for the object corresponding to *GUID*. The *GUID* should be computed from the object (or a part of it).
  2. *unpublish*(*GUID*): Makes the object corresponding to *GUID* unavailable.
  3. *sendToObj*(*msg*, *GUID*, [*n*]): Sends a message to the target object. This could be a request to update the object, or more likely, a request to open a connection in order to transfer the data associated with the object.
     - [*n*] is optional and specifies the number of replicas that the delivery of the same message should reach

Overlay Routing

Replication

- In order that an object remains available across node addition and removal, storage of an object must occur at more than one node
- For a Distributed Hash Table, some replication factor *r* is chosen (an appropriate choice gives a very high probability of continuous availability)
  - The object is then replicated at *r* nodes which are the *r* nodes numerically closest to the host node
- For the Distributed Object Location and Routing protocol, locations for the replicas of data objects are decided outwith the routing layer.
  - The DOLR layer is notified of these *host address* of each replica using the *publish* operation

Readable Object Identifiers

- GUIDs, nice though they are, are not human readable
- Client applications must therefore obtain the GUIDs for resources using some human-readable name or search request
- Ideally, such a lookup is also stored in a peer-to-peer manner
- This avoids a centralised service a la Napster and the associated disadvantages of such a centralised service
- Bittorrent is an interesting example, it uses individual web pages to publish “stub” files
    - The stub file includes the object’s GUID and:
    - The URL of a *tracker* which holds an up-to-date list of network addresses willing to provide the requested object
    - Note that it essentially uses existing search engines as the search facilities
    - Websites with particular object “stub” files may be “attacked”, but:
        - There may be many of them
        - Each web site may only host a small number, perhaps only a single, stub file

Pastry

- Implements a Distributed Hash Table
- Can be used for any application for which objects are stored and retrieved
- generally more useful if the objects are immutable or updated rarely
- Squirrel is an application built upon Pastry, it is a peer-to-peer web-cache system
- This works well as although the objects may be updated it is not crucial that all replicas are consistent

Pastry

Pastry Routing Overlay

- In Pastry each object and node is given an opaque 128-bit GUID
- In a network with N participating nodes the Pastry routing algorithm will deliver a message to an object or node within $\mathcal{O}(logN)$ steps
- If the GUID identifies a currently inactive node then the message is delivered to the node with a GUID numerically closest to the target GUID
- Each step along the route involves the use of an underlying transport protocol, usually UDP.
- Each such step, transfers the message from the current node to a node which is numerically closer to its destination
- Closer here though is in the entirely artificial GUID space and may in fact involve routing the message geographically more distant to the target node than the current node

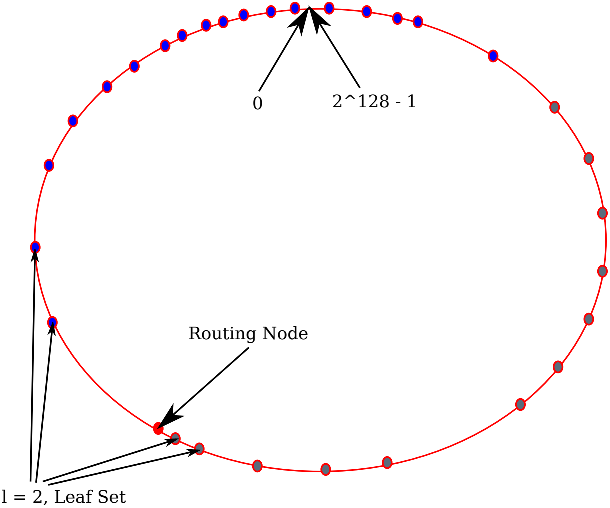
Peer-to-Peer Systems

Routing Overlay — Ring based

- Each node stores a vector L of size $2 \times l$ containing the GUIDs and IP addresses of nodes whose GUIDs are numerically closest on each side: l nodes above and l nodes below
- The vector L is known as the leaf set, and leaf sets are updated when nodes join or leave the network
- The GUID space is treated as circular, so GUID 0's lower neighbour is $2^{128} - 1$ and vice versa
- Any node with GUID D upon receiving a message for D':
  - If D' is in L then M can be directly forwarded to the target node
  - Otherwise M is forwarded to the GUID in L numerically closest to D'. Which will be either the left most or the right most node in L
    - It is the right most, if $D' > D$ and $D' - D < \frac{2^{128}-1}{2}$ or $D > D'$ and $D - D' > \frac{2^{128}-1}{2}$
    - And the left most node otherwise
- To deliver a message we require at most $\frac{N}{2 \times l}$ hops

Routing Overlay

Ring-Based



Routing Overlay — Using Routing Tables

- Each node maintains a tree-structured routing table giving GUIDs and IP addresses for a set of nodes spread throughout the entire range $0 \ldots 2^{128} - 1$
- However the routing table for node with GUID D will have an increased density of coverage for GUIDs which are numerically close to D

GUID Routing Table

The routing table for a node with GUID 90B . . .

0	0	1	2	3	4	5	6	7	8	9	A	B	C
	n_0	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8		n_A	n_B	n_C
1	90	91	92	93	94	95	96	97	98	99	9A	9B	9C
		n_{91}	n_{92}	n_{93}	n_{94}	n_{95}	n_{96}	n_{97}	n_{98}	n_{99}	n_{9A}	n_{9B}	n_{9C}
2	900	901	902	903	904	905	906	907	908	909	90A	90B	90C
	n_{900}	n_{901}	n_{902}	n_{903}	n_{904}	n_{905}	n_{906}	n_{907}	n_{908}	n_{909}	n_{90A}		n_{90C}

. . . the table has as many rows as there are hexadecimal digits in a 128 bit number, 32

- 128 bit GUIDs are examined as a string of 32 hexadecimal digits
- Each row has 15 entries (curtailed here for space)
- One for each value that does not match the current node's prefix
- The entry in each cell is the IP address of a node with a GUID with the prefix corresponding to the row and column

Pastry

The Pastry Routing Algorithm

- To handle a message M addressed to GUID D at node A, where $R[p, i]$ is the element at column i, row p of the routing table at node A:

  If$(L_{-l} < D < L_l)$
      Forward M to the element L_i of the leaf set with the GUID closest to D or the current node A
  else
      Find p, the length of the longest common prefix of D and A, and i, the $(p+1)^{th}$ hexadecimal digit of D
      If $(R[p, i] \neq null)$
          Forward M to $R[p, i]$
      else
          Forward M to any node in L (or R) with a common prefix of length p but a GUID that is numerically closer

- The lines in grey implement the previous ring-based algorithm, hence we can be sure that the algorithm will succeed in routing each message

Pastry Routing Algorithm

- Each table must have the property that:
  - The GUID of the node addressed in $R[p, i]$ has a common prefix with the target GUID D of length at least $p + 1$
  - Provided of course that $D[p] = i$
  - Another way of saying this is, should we have the cell: $| \overset{16A2C}{n} |$
  - Then n addresses a node with a GUID with the prefix $16A2C$
  - Note that we would not have such a cell if the *current* node had the prefix $16A2C$
  - Hence each time a message is forwarded it is forwarded to a node with a GUID that has longer matching prefix than the current node, so eventually it must be forwarded to the correct node

0	0 n_0	1 n_1	2 n_2	3 n_3	4 n_4	5 n_5	6 n_6	7 n_7	8 n_8	9	A n_A	B n_B	C n_C
1	90	91 n_{91}	92 n_{92}	93 n_{93}	94 n_{94}	95 n_{95}	96 n_{96}	97 n_{97}	98 n_{98}	99 n_{99}	9A n_{9A}	9B n_{9B}	9C n_{9C}
2	900 n_{900}	901 n_{901}	902 n_{902}	903 n_{903}	904 n_{904}	905 n_{905}	906 n_{906}	907 n_{907}	908 n_{908}	909 n_{909}	90A n_{90A}	90B	90C n_{90C}

. . . the table has as many rows as there are hexadecimal digits in a 128 bit number, 32

Pastry Overlay Routing

Host Integration

- When a host joins the network it must follow a specific protocol to obtain its table and leaf nodes as well as updating others
- The new node first computes a suitable GUID for itself
- The joining node should have the address of at least one existing node, it contacts this (or finds a nearer neighbour, where nearer is in reference to actual network distance)
- Suppose our new node has GUID X and its first contact has GUID A. The node sends a *join* request message to A giving X as its destination GUID
- The node A then forwards this request message to the node with the numerically closest GUID to X, let's call it Z
- Of course A does not in general know what that node is, it simply forwards on the *join* message as though routing to node X

Host Integration

Building the Routing table for X

- The key point is that a node (Z) must be able to tell that it is the currently closest (numerically) GUID to X
- It can know this due to its own leaf set
- As the *join* message is forwarded (ultimately to Z), the forwarding nodes help build up the routing table of X
- Note that the first row of X does not really depend upon the GUID X, so it can simply copy the first row of A.
  - It must update it slightly since X and A do not necessarily share the same first digit
  - In place $R_A[0, i]$ where $i = A[0]$ is the first digit of A, there will be no address, so in slot for X we can simply place the address of A
  - Additionally $R_X[0, j]$ where $j = X[0]$ is the first digit of X can be left empty even though $R_A[0, j]$ may not be

Host Integration

Concrete example of X and A

- Suppose the GUID of X is number $1(0000 \ldots 1)$
- The GUID of A is number $2^{127}(1000 \ldots 0)$
- The first row of A in positions $2 \ldots F$ are perfectly valid
- The value that A has for prefix 0 is worthless to X, but that's okay because in that position X will have no address (it's the red entry in the first row for X because it is the prefix of X)
- A has no entry in column 1 (it's A's red entry in the first row), but that's okay because we know a good address to fill in that column in X's first row, the value is the address of A.

Host Integration

Routing of the Join message

- The second row of A's table though is probably not relevant
- During its travels to the node numerically closest to X, (Z), the join message passes through some nodes $B, C \ldots Y$
- Each node $B, C \ldots Y$ through which the join message passes, transmit relevant parts of their routing tables and leaf sets to D
- Because of the routing algorithm the second row of $B's$ table will be relevant for X, so it simply sends X its second row, and also forwards the message on to C
- Now the third row of C should be applicable for X since it shares the same prefix of at least length 2.
- In fact C may have been in row n of B's table and hence can send X rows $2 \ldots n$
- When the message finally arrives at Z, we should have built up most of X's new routing table, and all we require is a good leaf set

Host Integration

Routing of the Join message

- Z is the numerically closest GUID to X
- Suppose $X > Z$:
  - The left hand side of X's leaf set is the left hand side of Z's but Z itself
  - The right hand side is exactly the right hand side of Z's original leaf set
  - Z however should update the right hand side of its leaf set to include X as the closest and optionally remove the right most node from the leaf set.
- Finally then once X has received and built up its own routing table and leaf nodes, it sends this information to all the nodes in its leaf node set and routing table such that they may update their accounts appropriately
- Incorporating this new node into the network requires the transmission of $\mathcal{O}(logN)$ messages

Pastry Overlay Routing

Host Removal

- A host may fail or leave at any time
- When this happens we must repair the routing tables and leaf sets so as not to contain the departed node
- We will assume that neighbouring nodes can detect a failed node via periodic polling and consider mostly the case of a node which departs intentionally
- Assume either way that a node D detects, or is alerted by the departing node itself, that node X has left the network
- Node D, looks for a close node L' in its own leaf set and requests a copy of the leaf set of L'
- The leaf set which L' sends D should overlap that of D and in particular contain a node suitable to replace that of X
- Other neighbouring nodes are then informed of the failure and they perform a similar procedure

Pastry Overlay Routing

Fault Tolerance

- Nodes may gracefully leave the system, but they may also fail, in a peer-to-peer system this could represent the user switching off or killing the process
- Failed nodes are detected through a system of “heartbeat” messages sent by non-failed nodes to their leaf sets
- However, failed node notification will not propagate through the network quick enough to eliminate routing failures
- If the application in question requires reliable delivery of messages then a reliable protocol must be built upon the routing overlay
- Recall at the start of the course we discussed reliable (TCP) communication and unreliable (UDP) communication
- One reason to use unreliable communication is that the application built ontop of the communication may be required to perform its own ommission/error detection/correction
- This is one such example

Pastry Overlay Routing

Fault Tolerance

- Where such a *re-try* mechanism is used it should allow the *Pastry* routing overlay time to adapt to an error
- However, as it stands this may not overcome all errors and certainly will not help in the presence of a malicious node
- To overcome this, an element of randomness is introduced into the routing algorithm
- To forward a message a node P, might choose not to immediately send it it to the node in P's routing table with the longest matching prefix, but instead, with a small probability, send to a node higher up the routing table.

0	0 n_0	1 n_1	2 n_2	3 n_3	4 n_4	5 n_5	6 n_6	7 n_7	8 n_8	9	A n_A	B n_B	C n_C
1	90	91 n_{91}	92 n_{92}	93 n_{93}	94 n_{94}	95 n_{95}	96 n_{96}	97 n_{97}	98 n_{98}	99 n_{99}	9A n_{9A}	9B n_{9B}	9C n_{9C}
2	900 n_{900}	901 n_{901}	902 n_{902}	903 n_{903}	904 n_{904}	905 n_{905}	906 n_{906}	907 n_{907}	908 n_{908}	909 n_{909}	90A n_{90A}	90B	90C n_{90C}

. . . the table has as many rows as there are hexadecimal digits in a 128 bit number, 32

Pastry Overlay Routing

Locality

- The routing table has an address associated for each possible digit in the i^{th} position which does not match the current node's i^{th} digit
- Each such address has a GUID with a prefix of length $i - 1$ which matches the current node's
- In a well populated overlay, and in particular in the early rows of the table, there will be many such choices
- Each choice is made based on a metric which measures network locality
- Usually IP hops, or round-trip time
- This cannot guarantee optimal routings but has been shown in simulations to produce routes that are only 30-50% longer
- It also helps route around failed nodes which have large round-trip times

Tapestry

- *Tapestry* is similar in goals to *Pastry*
- The *Tapestry* infrastructure uses a distributed hash table routing mechanism similar to the one described for *Pastry*
- However, the exposed *Tapestry* API is that of a DOLR (Distributed Object Location and Routing) interface
- Recall: DOLR has the following operations:
  1. *publish*(*GUID*): Makes the node performing the *publish* the host for the object corresponding to *GUID*. The *GUID* should be computed from the object (or a part of it).
  2. *unpublish*(*GUID*): Makes the object corresponding to *GUID* unavailable.
  3. *sendToObj*(*msg*, *GUID*, [*n*]): Sends a message to the target object. This could be a request to update the object, a request to open a connection in order to transfer the data associated with the object.
     - [*n*] is optional and specifies the number of replicas that the delivery of the same message should reach

Tapestry

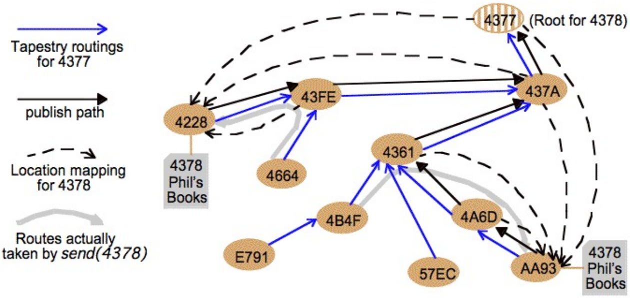
- Because replication is handled by the application rather than *Tapestry* itself, this gives applications additional flexibility in how to handle replication
- For example a file-sharing system may not need to explicitly handle replication since it done implicitly whenever a user copies an existing resource
- It is possible that absolutely no replication (of at least some resources) is necessary or desired
  - For example an online game could operate with each player hosting their own current state
  - When the player leaves, the state need not persist
  - Though the player's account may persist, this would be an example where some, but not all of resources are replicated

Tapestry

- Each object and routing node has a 160-bit identifier (GUID) associated with it
- In addition each (published) object is associated with exactly one "*root node*"
- The root node maintains a table mapping object GUIDs to the addresses of all replicas
- The root node will be the node with a GUID numerically closest to the GUID associated with the object
- When a node invokes $publish(GUID)$ the message is routed to the object's associated root node
- When a $sendToObj(GUID, msg, [n])$ message is invoked that too is routed to the root node of the object
  - The root node may then choose how many and which replicas to send that message to
  - The decision obviously being application dependent

Tapestry Routing

Figure 10.10: Tapestry routing From [Zhao et al. 2004]



Replicas of the file *Phil's Books* (G=4378) are hosted at nodes 4228 and AA93. Node 4377 is the root node for object 4378. The Tapestry routings shown are some of the entries in routing tables. The publish paths show routes followed by the publish messages laying down cached location mappings for object 4378. The location mappings are subsequently used to route messages sent to 4378.

Structured or Unstructured

- Structured peer-to-peer networks have a specific distributed data structure maintaining the routing overlay
- The structure imposed means that the peer-to-peer networks are efficient, offering some bound on, say, the number of messages required to route a message to an object
- Pastry for example relied upon the logical ring of GUID ids, and the routing tables made up distributed 'trees'
- However this is paid for in the cost of maintaining the distributed data structure underneath the peer-to-peer network
- An alternative is an *unstructured* peer-to-peer network

Peer-to-Peer Systems

Unstructured

- An unstructured peer-to-peer network does not rely on any distributed data structure
- Instead it relies upon an ad-hoc system of adding peers as they become available
- Each node joins the network by following some simple, local rules.
- A joining node must establish connectivity with a set of ‘neighbours’
- It knows that the neighbours will also be connected to their own neighbours and so on
- Connectivity to everyone follows from a ‘Kevin Bacon’ style arrangement, except that there is no special node

Unstructured Peer-to-Peer

Locating an object

- In an unstructured peer-to-peer network then, it is straightforward and inexpensive to join and leave a network
- However locating an object must be done by searching the resulting "mish-mash" of connections
- This approach then cannot guarantee to locate any specific object
- It is also possible that excessive amounts of traffic are used in locating and using objects
- Still, unstructured peer-to-peer networks have been shown to work
- In fact they are the dominant paradigm used in the Internet today

Unstructured dominance

-
  - Gnutella
  - Limewire
  - Freenet
  - Bittorrent
- All examples of unstructured peer-to-peer networks
- Many studies have estimated the overall proportion of Internet traffic which is peer-to-peer
- They vary widely in their estimates from some 20% to over 70%
- Safe to say it is a significant proportion, it's hard to say what is taken up with unnecessary transfer of data

Unstructured peer-to-peer systems

Unnecessary Data Transfer

- A variety of reasons, including inefficiency of the peer-to-peer system in question which may not be satisfying requests, dropping messages, or simply not pairing up providers with consumers in a network-efficient manner
- We may also get a lot of dropped connections because peers may leave at any time — file splitting is used to mitigate this
- Broken files, incorrectly labelled files etc
- Due to the uncertainty of availability many users "download now, consider later"
- Content may not be offered in the size desired, eg a whole album as opposed to single song which is desired

Structured vs Unstructured

Comparison

- Structured
  - Advantages
    1. Guaranteed to locate (existing) objects
    2. Relatively low message overhead
  - Disdvantages
    1. Need to maintain complex overlay structures
    2. Slow to adapt to highly dynamic networks
    3. Software is difficult to upgrade if it updates the distributed data structures used
- Unstructured
  - Advantages
    1. Self-organising and naturally resilient to node failure
    2. Different versions of software can often interoperate with little engineering effort
  - Disdvantages
    1. Offers no guarantees on locating objects even if they exist
    2. Can generate large amounts of messaging overhead

Unstructured Peer-to-Peer

Searching

- When file-sharing, a major problem is the location of desirable files
- We will stick to the problem of file-sharing but the same problem exists for many other similar applications
- Whether we are using a structured or unstructured peer-to-peer network we may still require to do some search to find an appropriate GUID
- The search strategies we look at now are applicable in a number of places, but we will specialise the case to search for a file in an unstructured peer-to-peer network

Peer-to-Peer Systems

File Searching

- The problem of searching for a particular file (or one of a set which is appropriate) becomes the problem of searching the entire network
- Naïvely done this could flood the network with many search requests
- A simple strategy is that a search request is sent to the nearest neighbours, each of whom respond with success or forward the search on to their neighbours
- Similar to IP multicast, each such search request has a time-to-live variable which is decremented each time the request is forwarded
- The approach though does not scale well

Improvements

- Expanded Ring Search
  - If there is an effective replication strategy in place, many searches may complete successfully locally
  - This is particularly true of file-sharing networks where the most popular files are those which are searched for the most often
  - Expanded ring search does the same as the naïve version but starts with a very small time-to-live variable
  - If that search fails, it tries again with a larger time-to-live variable
  - and so on, up to some limit

Peer-to-Peer Systems

Improvements

- Random Walks
  - A search agent can be set off in search of the desired file
  - The agent is of course not an actual agent but simply a message
  - When the message arrives at a node, the successfully found file can be sent directly back to the originator of the random walk agent
  - If not, it is forwarded to one other peer, the choice of peer is made randomly
  - A peer wishing to search may set off several random "agents" concurrently
  - Again they are generally equipped with a time-to-live counter

Improvements

- Gossiping
  - A node sends a request to a neighbour with a given probability
  - Hence a request spreads probabilistically through the network
  - The *Gossiping* name alludes to the way in which a search spreads through the network as a rumour spreads through social networks
  - Sometimes these are called *epidemic protocols*, because the (in the case) search spreads through the network like a virus

Improvements

- Ultra-Peers
  - In a pure peer-to-peer network all peers are treated equally
  - An ultra-peers system makes the observation that we may treat peers as equals but that does not reflect reality
  - A few selected peers are designated *ultra-peers*, generally because they have extra resources and some commitment to extended availability within the peer-to-peer system
  - These ultra-peers are heavily connected with each other, and ordinary peers connect themselves to one or more ultra-peers
  - This can offer dramatic improvements in terms of the number of hops required for exhaustive search
  - The ultra-peers are the Kevin Bacons of the peer-to-peer system

Query Routing Protocol

- In this system peers exchange information about the files/resources they have available
- For example each peer may gather together a set of words in the file names of their available files.
- These words are then sent to the associated ultra-peer
- The ultra-peer collates all these into a single table of available 'words' and exchanges this information with its neighbouring ultra-peers
- So when a (text based) search query is made each ultra-peer knows which search paths are likely to obtain positive results

Peerson

- Peerson (`www.peerson.net`) is a distributed peer-to-peer social network akin to Facebook
- Encryption is utilised heavily in order to provide security of user data
- This is in contrast to centralised servers which may encrypt stored data, but then the keys are stored in the same place
- In Peerson encryption keys are required to access any files (parts of a user's profile)
- The user has control over who may obtain those keys

Summary

- We began with looking at the motivations behind the development of peer-to-peer systems
  - Break the reliance of the system on a central server which may be vulnerable from attack, both technical and bureaucratic
  - Utilising the resources of those using the server such that capacity grows with the number of users
  - Providing anonymity to content providers
- The now defunct pioneering system *Napster*
  - Napster relied on a central server, but that server hosted no content, bandwidth to the central server was limited as well because no content was therefore downloaded from the central server
  - Instead the central server was merely used by remote peers to locate content and setup independent connections between peers
  - Ultimately though the reliance on a central server proved enough fodder for the entertainment industry's lawyers and Napster was shutdown

Summary

- Napster however proved the feasibility of the concept and several services grew into the space left behind by Napster
- Such services do not rely on any single central server and have so far proved resilient to legal attacks
- However we focused our attention on efforts to provide a generic framework for building peer-to-peer applications
- Such frameworks currently focus on providing a distributed hash table, storing objects and replicas at multiple peers for later retrieval
- Distributed Object Location and Routing systems are an extension providing a more convenient API, in particular for objects which may be updated

Summary

- In general though peer-to-peer systems have and continue to be used mostly for file sharing
- In particular the sharing of immutable files such as music files and video files
- Objects tend to be visited exactly once by a user and hence unstructured networks flourish as the additional structure provided by a distributed data structure cannot be put to great use
- The low cost and dynamic service provision mean that they are continued ot be offered by those with small budgets
- Large corporations such as Microsoft, Apple, Google, Facebook and Twitter are yet to embrace peer-to-peer applications

Any Questions

Any Questions?

Distributed Systems — Security

Allan Clark

School of Informatics
University of Edinburgh

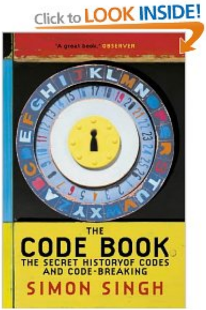
http://www.inf.ed.ac.uk/teaching/courses/ds
Autumn Term 2012

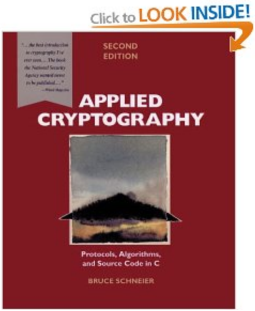
Security

Overview

- In this part of the course we will look at security in distributed systems
- Cryptography will provide the basis of secrecy and integrity
  - That is, making sure that no unauthorised entity may read any particular message
  - No unintended message is delivered, including a duplication of an intended message
- We will examine private-key techniques as well as public-key techniques and digital signatures
- We will look at cryptographic algorithms

Books

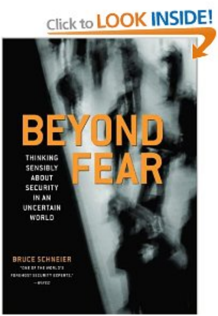

Click to LOOK INSIDE!
"A great book." OBSERVER
THE
CODE BOOK
THE SECRET HISTORY OF CODES AND CODE-BREAKING
SIMON SINGH


Click to LOOK INSIDE!
SECOND EDITION
APPLIED CRYPTOGRAPHY
Protocols, Algorithms, and Source Code in C
BRUCE SCHNEIER

Security

Books

- We will focus on threats to distributed systems caused by the inavoidable exposure of their communication channels
- The largest threat is generally human error
- Bruch Schneier also has a newsletter each month called "cryptogram" which talks about many security related topics including cryptography and physical/human related policies



Security

Cryptography

- Although computer security and computer cryptography are separate subjects, digital cryptography provides the basis for most of the mechanisms that we use in computer security
- It is only in recent years (the 1990s) that cryptographic techniques have been wrestled from the domain of the military into the domain of public knowledge and use
- When Bruce Schneier first published his book "*Applied Cryptography*" in 1994 the legal status of including cryptographic algorithms and techniques was in doubt.

Security

Pre-1999 US Munitions Control

- RSA crypto-algorithms, were, until 1999, classified by the US State Department as munitions
- Meaning they were classified in the same category as: chemical and biological weapons, tanks, heavy artillery, and military aircraft
- Additionally this meant that it was illegal to export such cryptographic algorithms, with penalties including $1m fines and long prison sentences
- This was obvious buffoonery:
  - It is impossible to enforce
  - The technology is widely available throughout the world
  - Algorithms published in international journals
  - Some cryptographic algorithms were developed outside the US

Security

Pre-1999 US Munitions Control

- Popular email programs such as *Netscape Communicator* had to have separate downloads for US based downloaders and external downloaders
- When it went open-source and became *mozilla* this was more nonsense since very quickly the external versions were patched to include full 160-bit encryption
- People took to methods of highlighting how ridiculous such an export ban was, one such effort demonstrated that RSA crypto algorithms can be written in a fairly short amount of Perl code

```
#!/bin/perl -sp0777i<X+d*lMLa^*lN%0]dsXx++lMlN/dsM0<j]dsj
$/=unpack('H*',$_);$_=`echo 16dio\U$k"SK$/SM$n\EsN0p[lN*1
lK[d2%Sa2/d0$^Ixp"|dc`;s/\W//g;$_=pack('H*',/((..)*)$/)
```

Security

Pre-1999 US Munitions Control

- So to highlight how ludicrous it was people started attaching it to emails
- Technically if said emails were sent outwith the US such people could have been prosecuted

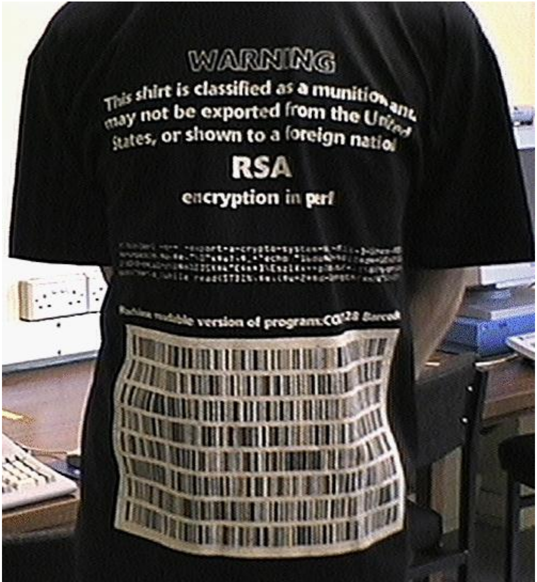
```
--

The following is classified as munitions by
 the US state department:

#!/bin/perl -sp0777i<X+d*lMLa^*lN%0]dsXx++lMlN/dsM0<j]dsj
$/=unpack('H*',$_);$_=`echo 16dio\U$k"SK$/SM$n\EsN0p[lN*1
lK[d2%Sa2/d0$^Ixp"|dc`;s/\W//g;$_=pack('H*',/((..)*)$/)
```

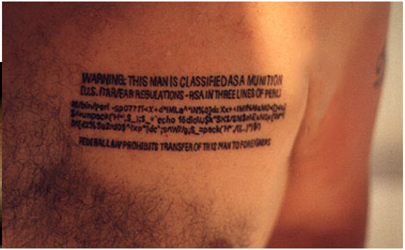
Security

T-Shirt



Security

Tattoos



Security Model

We will assume

- Wherever you are in the world you have access to cryptographic protocols and algorithms
- There are a set of nodes which share resources
  - Resources may be physical or data/programming objects
- Communication is via message passing only, and hence access to shared resources occurs via message passing
- The nodes are connected via a network which may be accessed by any enemy
- An enemy may copy or read any message transmitted through the network
- They may also inject arbitrary messages, to any destination purporting to come from any source

Security

Policies and Mechanisms

- There is a distinction between a security policy and a security mechanism
- Security policies are independent of the security mechanisms used with that policy
- A system cannot be secured using only security mechanisms
- For example, the door to your accommodation is likely secured using a lock and key, that is the security mechanism
- But it is near useless without the accompanying policy:
  - The last person to leave the building should lock the door

Security

Threats and Attacks

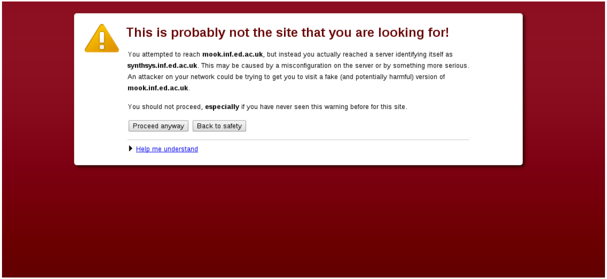
- For most types of network, and certainly wireless networks, it is generally obvious that an attacker wishing to obtain private information can simply listen in on all messages
- Doing so means that it is relatively simple to construct a computer that would simply log all messages between communicating computers
- Depending on the application simply knowing the contents of some messages may be enough, otherwise the attacker may need information about the distributed algorithm in question in order to construct information from the data in the messages that were recorded

Security

Threats and Attacks

- A slightly more elaborate attack is to construct a server in between the client and the intended server
- If the client does not authenticate the server, then it may send private information to what it believes to be the intended server
- Often the fake server will then log the information sent to it, but then also forward it on the real server in question
- Thus the attack is non-trivial to detect.
- This is a common technique for obtaining web-passwords

Security



- Third party "*Certificate Authorities*" issue digital certificates containing encryption keys to verify the identity of secure websites

Security

Threats and Attacks

- Threats and attacks fall into three broad categories:
  1. Leakage
     - The acquisition of data by unauthorised entities
  2. Tampering
     - The alteration of data by an unauthorised entity
  3. Vandalism
     - Distruption to the service in question without gain to the perpetrators

Security

Threats and Attacks

- We can further distinguish attacks in a distributed system by the way in which communication channels are misused:
  1. *Eavesdropping*
     - Obtaining copies of messages without authority
  2. *Masquerading*
     - Sending or receiving messages using the identity of another process/entity without their authority
  3. *Message Tampering*
     - Intercepting messages and altering them before forwarding them on to their intended recipient
  4. *Replaying*
     - Storing intercepted messages and sending them at a later date. This attack can be effective even when used against authenticated and encrypted messages (think of the two generals problem)
  5. *Denial of Service*
     - Flooding a service with requests such that it cannot handle legitimate requests

Security

Information Existence

- Regardless of how strong your encryption may be, the detection of a message transmitted between two processes may leak information
- The mere existence of such a message may be the source of information.
- For example a flood of messages to a dealer of a particular set of stocks may indicate a high-level of trading for a particular stock
- One possible defence is to regularly send nonsense/ignorable messages

Security

Trade-offs

- Ultimately all security measures involve trade-offs
- A cost is incurred in terms of computational work and network usage for use of cryptography and other protocols
- Where a security measure is not correctly specified it may limit the availability of the service for legitimate users/uses
- These costs must be stacked up against the threat or cost of failure to maintain security
- Generally we wish to avoid disaster and minimise mishaps

Security

Assume the worst

- Interfaces are exposed distributed systems are designed such that processes offer a set of services, or an interface. These interfaces must be open to allow for new clients. Attackers therefore are able to send an arbitrary message to any interface
- Networks are insecure An attacker can send a message and falisfy the origin address so as to masquerade as another user. Host addresses may be spoofed so that an attacker may receive a message intended for another
- Algorithms and program code is available to attackers Messages sent may be intercepted but that may not be useful since to make sense of the message an attacker may need to know the purpose/protocol within which the message is sent. Assume that that may be the case

Security

Assume the worst

- Attackers may have access to large resources Do not therefore rely on the fact that you may compute something faster than an attacker, or that an attacker has a limited timeframe in which their attack may be valid/dangerous/worthwhile
- Assume all code may have flaws the part of your software responsible for security must be trusted. Often called the *trusted computing base*. It should be minimised, for example application programmers should not be trusted to protect data from their users

Security

Cryptography

- Modern Cryptography relies on the use of algorithms which distort a message and reverse that distortion using a secrets called *keys*
- A simple substitution cyper is an example of this:
- In this case the key is the mapping of characters:
  - $a \mapsto f, b \mapsto x, c \mapsto j, \ldots$
- Today's encryption techniques are believed to have the property that the decryption key cannot be feasibly guessed using the cypertext (the encrypted message)

Security

Cryptography

- There are two main algorithms in use:
  1. shared secret keys
     - both parties must share knowledge of the secret key and it must not be shared with any other party
  2. public/private key pairs
     - The sender uses the receiver's *public* key to encrypt the message.
     - The encryption cannot be reversed by the *public* key and can only be reversed by the receiver's *private* key
     - The sender needs to know the receiver's *public* key but need not know the receiver's *private* key
     - Anyone may know the receiver's *public* key but the *private* key must be known only to the receiver
- Both kinds of algorithms are very useful and widely used
- public/private key algorithms require 100/1000 times more processing power
- The lack of need for initial secure transfer of the private key often outweighs the disadvantage

Security

Some Notation and Characters

- Alice and Bob are participants in security protocols
  - Alice has the secret key K_A and Bob the secret key K_B
  - They have a shared secret key K_{AB}
- Alice has a private key K_{Apriv} and a public key K_{Apub}
- $\{M\}_K$ is a message encrypted with key K
- $[M]_K$ is a message signed with key K
- Carol and Dave are extra participants for 3,4 party protocols
- Eve is an eavesdropper
- Mallory is a malicious attacker
- Sara is a server

Security

Scenario 1. Secure communication

- Cryptography can be used to enable secure communication
- In this instance each message is encrypted and can only be decrypted with the correct secret key
- So long as that secret key is not compromised then secrecy can be maintained
- Integrity is generally maintained using some redundant information within the encrypted message, such as a checksum

Security

Scenario 1. Secure communication

- Alice wishes to send some secret information to Bob
- If they share the secret key K_{AB} then:
- Alice uses the key and an agreed encryption algorithm $E(K_{AB}, M)$ to encrypt and send any number of messages $\{M_i\}_{K_{AB}}$
- Bob decrypts the messages using the corresponding decryption algorithm $D(K_{AB}, M)$
- Two problems:
  1. How can Alice initiate this communication by sending the secret key K_{AB} to Bob securely?
  2. How does Bob know that a message $\{M_i\}$ isn't a copy of an earlier encrypted message sent by Alice but intercepted by Mallory?

Security

Scenario 2. Authentication

- Cryptography can be used to authenticate communication between a pair of participants
- If there is a shared secret key known only to two parties, then a successful decryption of a received message requires that the message was originally encrypted using the appropriate key
- If only one (other) party knows of that secret key then we can deduce from whom the message originated

Security

Scenario 2. Authentication

- Alice wishes to communicate with Bob
- Sara is a securely managed authentication server
- Sara stores a secret key for each user, each user knows (or can generate from a password) their own secret key.
- Sara may generate a ticket which consists of a new shared key together with the identity of the participant to whom the ticket is issued

Scenario 2. Authentication

- Steps to secure communication:
  1. Alice sends a request to Sara stating who she is and requesting a ticket for secure communication with Bob.
  2. Sara creates a new secret key K_{AB} to be shared between Alice and Bob. Sara encrypts the ticket using Bob's secret key and sends that together with the secret key all encrypted with Alice's secret key $\{(\{ticket\}_{K_B}, K_{AB})\}_{K_A}$
  3. Alice decrypts this message and obtains the shared secret key and a message containing the ticket encrypted using Bob's secret key. Alice cannot decrypt this ticket message
  4. Alice sends the ticket together with her identity and a request for shared communication to Bob
  5. Bob decrypts the ticket: $\{(K_{AB}, Alice)\}_{K_B}$, confirms that the ticket was issued to the sender (Alice). Alice and Bob can then communicate securely using the (now) shared secret key K_{AB}. Generally the key is used for a limited amount of time before a new one is requested from Sara.

Scenario 2. Authentication

- This is a simplified version of Needham and Schroeder algorithm which is used in Kerberos system (developed at MIT and used here)
- The simplified version does not protect against a replay attack, where old authentication messages are replayed
- It is used within organisations since the individual private keys, K_A, K_B etc, must be shared between the authentication server and the participants in some secure way
- It is therefore inappropriate for use with wide area applications such as eCommerce
- An important breakthrough was the realisation that the user's password need not be sent through the network each time authentication is required. Instead "challenges" are used
- When the server sends Alice the ticket and new shared private key it encrypts it with Alice's own private key. An attacker pretending to be Alice would be defeated at this point

Scenario 3. Authenticated Communcation with Public Keys

- Assuming that Bob has generated his own public/private key pair K_{Bpub}, K_{Bpriv} then Alice and Bob can securely set up a shared private key K_{AB}
- We also assume that there is some public-key certificate system such that Alice can obtain Bob's public key in a way that she is confident that it is indeed Bob's public key
  1. Alice obtain's Bob's public key K_{Bpub}
  2. Alice creates a new shared key K_{AB} and encrypts it using K_{Bpub} using a public-key algorithm. This she sends to Bob $\{K_{AB}\}_{K_{Bpub}}$
  3. Bob decrypts this using the appropriate private key to obtain the shared private key K_{AB}. Shared communication can now take place

Scenario 3. Authenticated Communcation with Public Keys

- This is a hybrid cryptographic protocol and is widely used as it exploits useful features of both public-key and secret-key encryption algorithms
- The slower public-key algorithm is used to set up the speedier secret-key communication
- Problem:
  - The distribution of public keys. Mallory may intercept Alice's initial request to obtain Bob's public key and simply send Alice their own public key.
  - Mallory then intercepts the sending of the shared key which they copy and then re-encrypt using Bob's real public key and forward it to Bob.
  - Mallory can then intercept all subsequent messages since they have the shared secret key. They may need to in order to forward the messages on to Bob and Alice depending on the delivery mechanism.

Digital Signatures

- Cryptography can be used to implement digital signatures
- Alice can encrypt a message using Bob's public key such that only Bob can decrypt the message
- Alice can also encrypt the message using her own secret key
- Anyone can decrypt the message so long as they know Alice's public key
- Provided we can be sure that the public key in question really is that of Alice's we now know that the message must have originated from Alice, since only Alice knows Alice's secret key
- Rather than encrypt the entire message Alice can compute a digest of the message, where a digest is similar to a checksum except that two distinct messages are very unlikely to have the same digest value
- This digest is encrypted and attached to the message, the receiver can then check that the unencrypted digest matches the (receiver computed) digest of the contents of the message

Security

Scenario 4. Digital Signatures

- Alice wishes to sign a document M so that any subsequent receiver can be sure that it originated from Alice
  1. Alice computes a fixed length digest of the document $Digest(M)$
  2. Alice encrypts the digest with her private key and attaches the result to the message $M, \{Digest(M)\}_{K_{Apriv}}$
  3. Alice makes the document with signature available
  4. Bob obtains the signed document, extracts M and computes $d = Digest(M)$
  5. Bob decrypts $\{Digest(M)\}_{K_{Apriv}}$ using K_{Apub} and compares the result to d, if they match the signature is valid.

Scenario 4. Digital Signatures

- We have three requirements of digital signatures
  1. Authentic It convinces the recipient that the signer deliberately signed the document and it has not been altered by anyone else
  2. Unforgeable It provides proof that no one else deliberately signed the document. In particular the signature cannot be copied and placed on another document
  3. Non-repudiable The signer cannot credibly deny that the document was signed by them
- Note that encryption of the entire document, or its digest, gives good evidence for the signature as unforgeable
- Non-repudiable is the most difficult to achieve for digital signatures. A signer may simply deliberate disclose their secret key to others and then claim that anyone could have signed it
- This can be solved through engineering but is generally solved through social contract "If you give away your secret key you are liable"

Security

5. Certificates
   - Suppose Alice would like to shop with Carol
   - Carol would like to be sure that Alice has some form of bank account
   - Alice has a bank account at Bob's bank
   - Bob's bank provides Alice with a *certificate* stating that Alice does indeed have an account with Bob.
   - Such a certificate is digitally signed with Bob's private key K_{Bpriv} and can be checked using Bob's public key K_{Bpub}

Security

5. Certificates
   - Now suppose Alice wished to carry out an attack such that she convinced Carol that someone else's account was owned by herself
   - This is quite simple, Alice only requires to generate a new public-private key pair $K_{BprivFake}, K_{BpubFake}$
   - She then creates a certificate falsely claiming that Alice is the owner of some account and signs it using $K_{BprivFake}$
   - If she can convince Carol that $K_{BpubFake}$ is the true public key of Bob's bank, then this attack should work no problem

5. Certificates
   - The solution is for Carol to require a certificate from a trusted fourth party, Dave from the Bankers' Federation, whose role it is to certify the public keys of banks
   - Dave issues a public-key certificate for Bob's public key K_{Bpub}. This is signed using Dave's private key K_{Dpriv} and can be verified using Dave's public key K_{Dpub}
   - Of course now we have a recursive problem, since now we need to authenticate that K_{Dpub} is the legitimate public key of Dave from the Bankers' federation.
   - We break the recursion by insisting that at some point Carol must trust one person, say Dave, and to do so may require to meet them in person.
   - Note that Carol only has to trust Dave in order to verify bank account certificates from a variety of banks

Security

5. Certificates
   - To make certificates useful, we require:
     1. A standard format such that certificate issuers and users can construct and interpret them successfully.
     2. Agreement on the way in which chains of certificates are constructed and in particular the notion of a trusted authority
   - In addition, we may wish to revoke a certificate, for example if someone closes their account
   - This is problematic since once the certificate is given it can be copied and stored etc
   - The usual solution is for the certificate to have an expiration date, meaning that the holder of the certificate must periodically renew it (in the say way that one renews a passport)

Cryptographic Algorithms

- Until now we have just assumed there is some method of encrypting the plaintext into the corresponding cyphertext using a particular key
- Additionally that there is some inverse operation to decrypt the cyphertext back into the original plaintext, using the same or corresponding decryption key
- The encryption depends on two things, the method E and the key K
- A message M has an encrypted version $\{M\}_K$ if:
  - $\{M\}_K = E(K, M)$
- The mathematically minded can think of an encryption algorithm as describing a (large) family of encryption functions from which one is selected by any given key
- Decyption of course gives the original message when used with the correct key
  - $M = D(K', \{M\}_K)$

Security

Symmetric Algorithms

- Shared secret key, or symmetric algorithms use the same key for decryption as for encryption, such that:
- $M = D(K, E(K, M))$ or $M = D(K, \{M\}_K)$
- It should be the case that the inverse function $M = E^{-1}(\{M\}_K)$ is so hard to compute as to be infeasible
- However both $E(K, M)$ and $D(K, \{M\}_K)$ should be relatively easy to compute
- Such functions are known as *one-way* functions

Security

Defence — symmetric algorithms

- Whilst a strong *one-way* function defends against an attack which attempts to discover M given $\{M\}_K$
- It does not necessarily defend against an attack which seeks to discover K given M and $\{M\}_K$ (and crucially E)
- This has been an important attack and was used heavily during World War II to break the Nazi *enigma* encryption scheme
- The simplest and highly effective attack is a *brute-force* attack in which all keys are attempted, computing $E(K, M)$ to see if it matches $\{M\}_K$
- The number of possible keys depends on the length of K, if it has N bits then there are 2^N possibilities (though you need only try 2^{N-1} on average.

Security

Block Ciphers

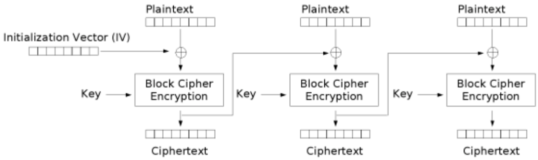
- Most algorithms operate on a fixed size of block
- For larger messages we split it up into a number of blocks and encrypt each one in serial, independently
- Hence the first block is available for transmission as soon as it is encrypted
- However this is a slight weakness, since the attacker can recognise repeated patterns and infer the relationship to the plaintext

Security

Cipher Block Chaining

- In Cipher Block Chaining each block is combined with the precedeing block.
- Note that this still means the previous block may be transmitted as soon as it is ready
- Generally *XOR* is used, if we have block N of plaintext and $\{M^{N-1}\}_K$ of cipherext, then block N is encrypted as: $\{M^N\}_K = E(K, N \oplus \{M^{N-1}\}_K)$
- Upon decryption, each block is xor'ed with the preceding block, this works since xor is its own inverse
- This is intended to prevent identical portions of the plaintext from encrypting to identical portions of ciphertext
- But there is a slight weakness at the start of each stream of of blocks
- To prevent this we insert a different piece of plaintext in front of each message, known as the *initialisation vector*.

Cipher Block Chaining



Cipher Block Chaining (CBC) mode encryption

Security

Cryptographic Algorithms

- There are many well designed cryptographic algorithms such that $E(K, M) = \{M\}_K$ such that the value of M is concealed and computing K requires a brute-force attack
- Confusion Non-destructive operations such as *xor* and circular shifting are used to combine each block of plaintext with the key
  - This confuses the relationship between M and $\{M\}_K$
  - If the blocks are larger than a few characters then this defeats attempts at cryptanalysis based on character frequencies
- Diffusion There is usually repetition and redundancy in the plaintext. Diffusion is used to dissipate regular patterns that result by transposing portions of each plaintext block.

Security

TEA — Secret Key Algorithm

- k is the key of length four (64-bit integers)
- text is originally the plaintext to be encrypted, two 64-bit integers

1. delta = 0x9e3779b9, sum = 0
2. y = text[0], z = text[1]
3. for (n= 0; n < 32; n++)
4. sum += delta
5. y += ((z << 4) + k[0]) $\oplus$ (z+sum) $\oplus$ ((z >> 5) + k[1])
6. z += ((y << 4) + k[2]) $\oplus$ (y+sum) $\oplus$ ((y >> 5) + k[3])
7. text[0] = y; text[1] = z;

Security

TEA — Tiny Encryption Algorithm

- On each of the 32 rounds the two halves of the text are repeatedly combined with shifted portions of the key and each other
- The *xor* and shifted portions of the text provide *confusion*
- Shifting and swapping of the two portions of the text provide *diffusion*
- The non-repeating constant *delta* is combined with each portion of the text on each cycle to obscure the key in case it might be revealed by a section of the text which does not vary

Security

TEA — Decryption

1. delta $=$ 0x9e3779b9, sum $=$ delta $<<$ 5
2. y $=$ text[0], z $=$ text[1]
3. for (n$=$ 0; n $<$ 32; n$++$)
4. z -$=$ ((y $<<$ 4) $+$ k[2]) $\oplus$ (y $+$ sum) $\oplus$ ((y $>>$ 5) $+$ k[3])
5. y -$=$ ((z $<<$ 4) $+$ k[0]) $\oplus$ (z $+$ sum) $\oplus$ ((z $>>$ 5) $+$ k[1])
6. sum -$=$ delta;
7. text[0] $=$ y; text[1] $=$ z;

Security

DES

- The Data Encryption Standard
- Is mostly of historical importance now since its keys are 56-bits long
- Too short to resist brute-force attack using modern hardward
- Maps a 64-bit plaintext into a 64-bit ciphertext using a 56-bit key
- The algorithm has 16 dependent stages known as *rounds*
- Algorithm was developed in 1977 and was slow on machines of the time when written in software
- However the algorithm could be implemented in hardware and was incorporated into network interface chips

Security

DES — Cracked

- In June 1997 it was succesfully cracked in a brute-force attack
- The attack was performed as part of a competition to illustrate the need for 128-bit long keys
- About 14,000 computers took part in a distributed computation to crack the 56-bit key
- The program was aimed at cracking a known plaintext/ciphertext pair, to obtain the unknown key (and then use that to decrypt new ciphertext)
- Later, the EEF developed a machine that could successfully crack 56-bit keys in around three days

Security

Triple DES

- One solution to the weakness of 56-bit keys is to simply apply the algorithm more than once with more than one key
- $E_{3DES}(K_1, K_2, M) = E_{DES}(K1, D_{DES}(K_2, E_{DES}(K_1, M)))$
- This is equivalent to the strength of a single key with a length of around 112-bits
- But it is slow since it must be applied three times
- And DES is already considered slow by modern standards

Security

IDEA

- International Data Encryption Algorithm
- Uses 128-bit keys
- A successor to DES, its algorithm is based on the algebra of groups, and has 8 rounds of *xor*, addition modulo 2^{16} and multiplication
- Like DES uses the same function for encrytion as for decryption, which is useful if it is to be implemented in hardware.
- IDEA has been analysed extensively, and no major weaknesses have been found. It is also around three times faster than the speed of DES (and hence 9 times faster than triple-DES)

Security

AES

- US National Institute for Standards and Technology invited proposals for AES (advanced encryption standard)
- The winner, the Rijndael algorithm, was selected from 21 algorithms submitted by cryptographers in 11 countries
- The cipher has variable block and key lengths, with specifications for keys with lengths 128, 192 or 256 bits to encrypt blocks with the same lengths
- The number of rounds varies from 9 to 13
- The algorithm can be implemented efficiently on a wide range of processors and in hardware

Security

Public-key Algorithms

- There are relatively few practical public-key algorithms
- They depend on the trap-door functions of large numbers to produce keys
- The keys K_e and K_d are a pair of very large numbers
- The encryption performs an operation such as exponentiation on one of them
- Decryption is a similar function using the other key.
- If the exponention uses modulo arithmetic it can be shown that the result is the same as the original value of M, so:
- $D(K_d, E(K_e, M)) = M$
- RSA is the most widely known public-key algorithm

Security

RSA

- Rivest, Shamir and Adelman, based on the product of two very large prime numbers
- Again despite extensive attempts and investigations, no flaws have been found

Security

RSA, to find a key pair K_e, K_d

1. We need to find three numbers e, d and N, the keys will be $K_e = e, N$ and $K_d = d, N$
2. Choose two large prime number P and Q both larger than 10^{100} (a googol)
   - $N = P \times Q$
   - $Z = (P - 1) \times (Q - 1)$
3. For d choose any number that is relatively prime with Z $(gcd(d, Z) = 1)$
4. To find e, solve: $e \times d = 1 \ mod \ Z$
   - So $e \times d$ is the smallest element in the series $Z + 1, 2Z + 1, 3Z + 1, \ldots$ which is divisible by d

Security

RSA

- So the function to encrypt a single block of plaintext M is
- $E'(e, N, M) = M^e \ mod \ N$
- So the largest length of M is $log_2(N)$ bits
- And to decrypt a block of text is:
- $D'(d, N, c) = c^d \ mod \ N$
- Rivest, Shamir and Adelman proved that E' and D' are mutual inverses, so $E'(D'(x)) = D'(E'(x)) = x$ for all values of P in the range $0 \leq P \leq N$
- Note that encryption requires e and N so $K_e = e, N$
- And decryption requires d and N so $K_d = d, N$

RSA — Concrete Example

1. Choose P and Q as very large prime numbers
   - $P = 5$ and $Q = 11$
2. $N = P \times Q$ and $Z = (P - 1) \times (Q - 1)$
   - $N = 55$ and $Z = 40$
3. For d choose any number that is a relative prime of Z
   - $d = 7$
4. To find e solve $e \times d = 1 \ mod \ Z$
   - $41, 81, 121, 161, \ldots$
   - $e \times 7 = 161, e = 23$
5. The numerical value of a block must be less than N, so the length of a block k must be such that $2^k < N$ here we will be forced to choose $k = 5$

Security

RSA — Concrete Example

- So to encrypt the block M with numerical value 24 using the $K_e = 23, 55$
  1. $E'(e, N, M) = M^e \ mod \ N$
  2. $E'(e, N, M) = 24^{23} \ mod \ N$
  3. $E'(e, N, M) = 5557232403542850518537839470182\,4 \ mod \ 55$
  4. $E'(e, N, M) = 19$
- To decrypt with $K_d = 7, 55$
- $D'(d, N, c) = c^d \ mod \ N$
- $D'(d, N, c) = 19^7 \ mod \ N$
- $D'(d, N, c) = 893871739 \ mod \ 55$
- $D'(d, N, c) = 24$

I tried first with $M = 21$ but $21^{23} \ mod \ 55 = 21$

Security

RSA — Cracking

- Given that the public key K_e contains N, to figure out e and d (and hence K_d) an attacker requires to factorise N
- In our example the prime factorisation of 55 is relatively easy to figure out $5, 11$
- The attacker would therefore know Z, they wouldn't know the choice of d but could brute-force try all possibilities
- In practice of course P and Q are chosen to be $> 10^{100}$ so $N > 10^{200}$, hence factorisation of N is extremely computationally expensive
- Factorisation of a number as large as 10^{200} would take 4 billion years using the best known algorithm on a computer that performs 1 million instructions per second.
- Intel Core i7 Extreme Edition 3960X (Hex core) $=$ 177,730 MIPS
- $(4000000000 \times 31556900)/177730000000 = 710221$
- So 710000 years

Security

RSA Challenges

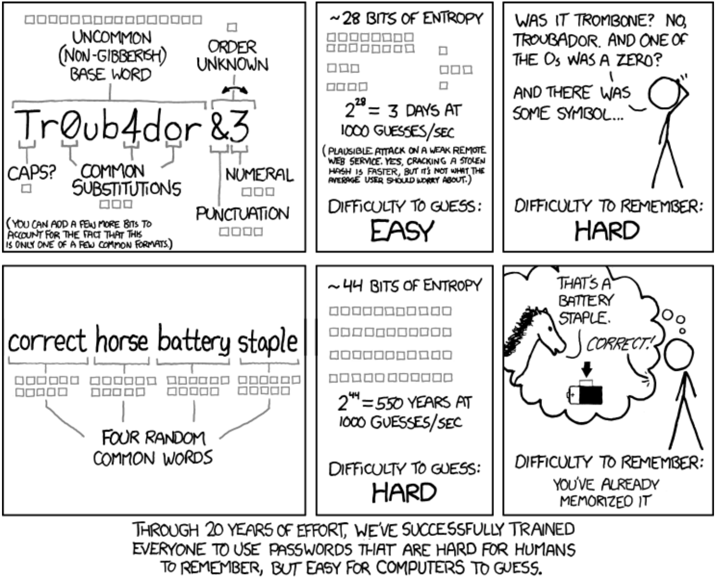
- The RSA Corporation issued a challenge to factor numbers of more than 100 decimal digits
- Numbers up to 232 decimal digits (768 binary digits) have been successfully factored
- Though there is still a 212 decimal digit (704 binary digits) number which remains unfactored
- Keys as large as 2048 bits are used in some applications
- All of this security somewhat depends upon the currently known best factoring algorithms not being improved (either because it is impossible or simply because no-one figures out how)

Security

Public-key algorithms

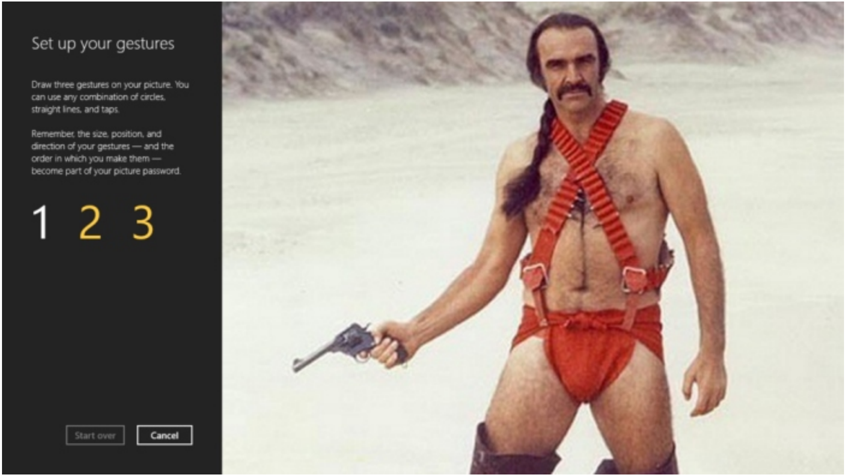
- It is worth noting a problem for all public-key cryto-algorithms
- An attacker as an unlimited supply of ciphertexts with known plaintexts
- Since the encryption is done using the public key and the attacker has access to the public key they can simply create as many plaintext/ciphertext pairs as they require
- They may even do so with any given text, for example the zero plaintext
- Additionally if the *unknown* encrypted message was really short, one could simply brute-force try all messages of the same length encrypting them to see if they match the encrypted message
  - This is obviously defeated by making sure that each message is at least as long as the key such that this form of brute-force attack is less feasible than a brute-force attack on the key

XKCD


UNCOMMON (NON-GIBBERISH) BASE WORD
ORDER UNKNOWN
Tr0ub4dor &3
CAPS?
COMMON SUBSTITUTIONS
NUMERAL
PUNCTUATION
(YOU CAN ADD A FEW MORE BITS TO ACCOUNT FOR THE FACT THAT THIS IS ONLY ONE OF A FEW COMMON FORMATS.)
~28 BITS OF ENTROPY
2^28 = 3 DAYS AT 1000 GUESSES/SEC
(PLAUSIBLE ATTACK ON A WEAK REMOTE WEB SERVICE. YES, CRACKING A STOLEN HASH IS FASTER, BUT IT'S NOT WHAT THE AVERAGE USER SHOULD WORRY ABOUT.)
DIFFICULTY TO GUESS: EASY
WAS IT TROMBONE? NO, TROUBADOR. AND ONE OF THE Os WAS A ZERO?
AND THERE WAS SOME SYMBOL...
DIFFICULTY TO REMEMBER: HARD
correct horse battery staple
FOUR RANDOM COMMON WORDS
~44 BITS OF ENTROPY
2^44 = 550 YEARS AT 1000 GUESSES/SEC
DIFFICULTY TO GUESS: HARD
THAT'S A BATTERY STAPLE.
CORRECT!
DIFFICULTY TO REMEMBER: YOU'VE ALREADY MEMORIZED IT
THROUGH 20 YEARS OF EFFORT, WE'VE SUCCESSFULLY TRAINED EVERYONE TO USE PASSWORDS THAT ARE HARD FOR HUMANS TO REMEMBER, BUT EASY FOR COMPUTERS TO GUESS.

Security

Zardoz Jeff Atwood @CodingHorror recently blogged about his Surface RT authentication:



Any Questions

Any Questions?

Distributed Systems — Summary

Allan Clark

School of Informatics
University of Edinburgh

http://www.inf.ed.ac.uk/teaching/courses/ds
Autumn Term 2012

Review — Introduction

Definition of Distributed Systems

- We debated over the definition of a distributed system and decided that the distinguishing features were:
  - Independent computers
  - Coordination achieved only through message passing
- There is also the notion of transparency of distribution, that is that the distributed system should appear to the users as a single computer
  - We decided that this was a nice feature but not an essential one
- We distinguished the study of distributed systems from the study computer networks by noting:
  - Computer networking is the study of how to send messages between remote computers
  - Distributed systems is the study of how to use that capability to get work done

Review — Introduction

Challenges

- We identified several challenges involved in designing and building distributed systems:
  1. Heterogeniety
  2. Openness
  3. Security
  4. Scalability
  5. Handling of failures
  6. Concurrency
  7. Transparency

Review — Fundamental Concepts

- A distributed algorithm was defined as the steps to be taken at each process — in particular the sequence of steps taken globally is not defined
- We defined a synchronous system as one in which we have known upper and lower bounds for:
  - the time taken for each process to execute each step in the computation
  - Time taken for message delivery
  - The clock drift rate from real time for each process
- An asynchronous system has no such bounds
- We noted that all asynchronous systems could be made synchronous by assuming very large bounds
- The defining feature of a synchronous system is that the bounds were useful
- Synchronous systems allow for simpler algorithms but determining useful bounds is often difficult

Review — Fundamental Concepts

Models

- We create models of our distributed systems in order to make explicit all relevant assumptions
- Make generalisations about what is possible given those assumptions
  - The interaction model allows us to determine logical properties of the algorithm, such as termination, correctness and other properties more dependent on the application
  - The performance model allows us to improve on the abstract performance available from the interaction model by combining with performance data of the underlying machines and network mediums
  - The failure model allows us to permit (or exclude) classes of errors and reason about the effect of those errors on the operation or performance of our algorithm
  - The security model is used to assess risk of information leakage/distortion even given the use of cryptography

Review — Fundamental Concepts

Network Issues

- Latency is defined as the time it takes for data to first arrive at the destination after the send is initiated
- data transfer rate is how much data per unit of time may be transfered
- message delivery time $= latency + \frac{\text{message length}}{\text{data transfer rate}}$
- latency affects small frequent messages which are common for distributed systems

Reliability

$$\text{Left} \longleftrightarrow 1 \longleftrightarrow 2 \longleftrightarrow 3 \longleftrightarrow 4 \longleftrightarrow 5 \longleftrightarrow \text{Right}$$

$$\text{Left} \longleftrightarrow 1 \longleftrightarrow 2 \longleftrightarrow 3 \longleftrightarrow 4 \longleftrightarrow 5 \longleftrightarrow \text{Right}$$

$$\text{Left} \longleftrightarrow 1 \longleftrightarrow 2 \longleftrightarrow 3 \longleftrightarrow 4 \longleftrightarrow 5 \longleftrightarrow \text{Right}$$

$$\text{Left} \longleftrightarrow 1 \longleftrightarrow 2 \longleftrightarrow 3 \longleftrightarrow 4 \longleftrightarrow 5 \longleftrightarrow \text{Right}$$

$$\text{Left} \longleftrightarrow 1 \longleftrightarrow 2 \longleftrightarrow 3 \longleftrightarrow 4 \longleftrightarrow 5 \longleftrightarrow \text{Right}$$

Red denotes a node at which error detection/correction occurs

- If the probability of a message getting through any channel is 0.5 then completing the trip is $0.5^6 = 0.016$
- Fortunately communication channels are generally more reliable
- $(\frac{9999}{10000})^6 = 0.9994 > \frac{999}{1000}$

UDP and TCP

- Two internet protocols provide two alternative transmission protocols for differing situations with different characteristics
- User Datagram Protocol — UDP
  - Simple and efficient message passing
  - Suffers from possible omission failures
  - Provides error detection but no error correction
- Transmission Control Protocol – TCP
  - Built on top of UDP
  - Provides a guaranteed message delivery service
  - But does so at the cost of additional messages
  - Has a higher latency as a stream must first be set up
  - Provides both error detection and correction
- IP Multicast has UDP-like failure semantics (maybe)

Review — Fundamental Concepts

Marshalling

- Marshalling is the process of flattening out a complex data structure into a series of bytes which can be sent in a message
- CORBA, Java Serialisation, XML and JSON
- Some come with instructions to the receiver on how to re-construct the flattened, others require pre-agreement on the types of the communicated data structures
- XML more general, JSON becoming popular because programmers are “fed-up” of parsing

Review — Time and Global State

Synchronising Clocks

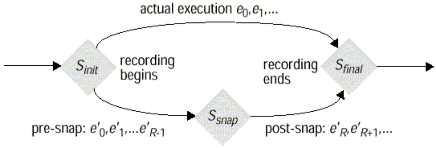
- We noted that even in the real world there is no global notion of time
- We thought that perhaps that didn't matter because we are all travelling at slow speeds relative to each other
- However our clocks are not true clocks they are mechanical and as such are subject to drift and skew
- We nevertheless described algorithms for attempting the synchronisation between remote computers
  - Cristian's method
  - The Berkely Algorithm
  - Pairwise synchronisation in NTP
- Despite these algorithms to synchronise clocks it is still impossible to determine for two arbitrary events which occurred before the other.
- We therefore looked at methods to impose a meaningful order on remote events and this took us to logical orderings

Review — Time and Global State

Logical Orderings and States

- Logical orderings based on the intuitive and simple idea of the “happens-before” relation:
  - $e_1 \rightarrow e_2$ if e_1 and e_2 occur at the same process and e_1 occurs before e_2, or:
  - e_1 is the sending of some message and e_2 is the receiving of that same message, or:
  - There exists some event $e_{1.5}$ such that: $e_1 \rightarrow e_{1.5} \rightarrow e_2$
- Lamport and Vector clocks were introduced:
  - Lamport clocks are relatively lightweight provide us with the following $e_1 \rightarrow e_2 \implies L(e_1) < L(e_2)$
  - Vector clocks improve on this by additionally providing the reverse implication $V(e_1) < V(e_2) \implies e_1 \rightarrow e_2$
  - Meaning we can entirely determine whether $e_1 \rightarrow e_2$ or $e_2 \rightarrow e_1$ or the two events are concurrent.
  - But do so at the cost of message length and scalability
- The concept of a true history of events as opposed to *runs* and *linearisations* was introduced

Global State — Chandy and Lamport — Reachability



- We looked at Chandy and Lamport's algorithm for recording a global snapshot of the system
- Crucially we defined a notion of reachability such that the snapshot algorithm could be usefully deployed in ascerting whether some stable property has become true.

Review — Time and Global State

Distributed Debugging

- Finally the use of consistent cuts and linearisations was used in Marzullo and Neiger's algorithm
- Used in the debugging of distributed systems it allows us to ascertain whether some transient property was possibly true at some point or definitely true at some point.
- Suppose we have a monitor M and two processes P_1 and P_2
  - We start with $P_1(x = 100)$ and $P_2(y = 50)$
  - M receives a message from $P_1, x = 50$
  - M receives a message from $P_2, y = 100$
  - The monitor then records four global states:
    1. $x = 100, y = 50$
    2. $x = 50, y = 50$
    3. $x = 100, y = 100$
    4. $x = 50, y = 100$
  - Both states 2 and 3 could not have occurred, but we do not know which occurred
  - If we wish to know whether the sum was ever 200 then we say "possibly" but not "definitely"

Review — Time and Global State

Distributed Debugging

- The purpose of a snapshot algorithm was to record a global state that is logically consistent with some state which was actually experienced, but there is no attempt to record a state which was "actually experienced"
- Distributed debugging on the other hand hopes to record such "actually experienced" states, and is conservative in the sense that it considers more states than may actually have occurred

Review — Coordination and Agreement

Mutual Exclusion and Election

- We looked at the problem of Mutual Exclusion in a distributed system
  - Giving four algorithms:
    1. Central server algorithm
    2. Ring-based algorithm
    3. Ricart and Agrawala's algorithm
    4. Maekawa's voting algorithm
  - Each had different characteristics for:
    1. Performance, in terms of bandwidth and time
    2. Guarantees, largely the difficulty of providing the *Fairness* property
    3. Tolerance to process crashes
- We then looked at two algorithms for electing a master or nominee process; ring-based and bully algorithms
- Then we looked at providing multicast with a variety of guarantees in terms of delivery and delivery order

Review — Coordination and Agreement

General Consensus

- We then noted that these were all specialised versions of the more general case of obtaining consensus
- We defined three general cases for consensus which could be used for the above three problems
- We noted that a synchronous system can make some guarantee about reaching consensus in the existance of a limited number of process failures
- But that even a single process failure limits our ability to guarantee reaching consensus in an asynchronous system
- In reality we live with this impossibility and try to figure out ways to minimise the damage

Review — Distribution and Operating Systems

Operating System Characterisations

- Distributed Operating Systems are an ideal allowing processes to be migrated to the physical machine more suitable to run it
- However, Network Operating Systems are the dominant approach, possibly more due to human tendancies than technical merit
- We looked at microkernels and monolithic kernels and noted that despite several advantages true microkernels were not in much use
- This was mostly due to the performance overheads of communication between operating system services and the kernel
- Hence a hybrid approach was common

Review —- Distribution and Operating Systems

Concurrency, processes and threads

- We looked at processes and how they provide concurrency, in particular because such an application requires concurrency because messages can be received at any time and requests take time to complete, time that is best spent doing something useful
- but noted that separate processes were frequently ill-suited for an application communicating within a distributed system
- Hence threads became the mode of concurrency offering lightweight concurrency.
- Multiple threads in the same process share an execution environment and can therefore communicate more efficiently and the operating system can switch between them more efficiently

Review — Distribution and Operating Systems

Operating System Costs and Virtualisation

- We also looked at the costs of operating system services on remote invocation
- Noting that it is a large factor and any design of a distributed system must take that into account — in particular the choice of protocol is crucial to alleviate as much overhead as possible
- Finally we looked at system virtualisation and noted that it is becoming the common-place approach to providing cloud-based services
- Virtualisation also offers some of the advantages of a microkernel including increased protection from other users' processes

Review — Peer-to-Peer Systems

Motivations and Napster

- We began with looking at the motivations behind the development of peer-to-peer systems
  - Break the reliance of the system on a central server which may be vulnerable from attack, both technical and bureaucratic
  - Utilising the resources of those using the service such that capacity grows with the number of users
  - Providing anonymity to content providers
- The now defunct pioneering system *Napster*
  - Napster relied on a central server, but that server hosted no content, bandwidth to the central server was limited as well because no content was therefore downloaded from the central server
  - Instead the central server was merely used by remote peers to locate content and setup independent connections between peers
  - Ultimately though the reliance on a central server proved enough fodder for the entertainment industry's lawyers and Napster was shutdown

Review — Peer-to-Peer Systems

Peer-to-Peer Frameworks

- Napster however proved the feasibility of the concept and several services grew into the space left behind by Napster
- Such services do not rely on any single central server and have so far proved resilient to legal attacks
- However we focused our attention on efforts to provide a generic framework for building peer-to-peer applications
- Such frameworks currently focus on providing a distributed hash table, storing objects and replicas at multiple peers for later retrieval
- Distributed Object Location and Routing systems are an extension providing a more convenient API, in particular for objects which may be updated

Review — Peer-to-Peer Systems

Structured vs Unstructured

- Two related problems for the use of a peer-to-peer system:
  1. initially finding the resource you are interested in and thus obtaining its logical address (GUID)
  2. Routing to the logical address (GUID) once it is known
- Analogous to internet addresses which first translate the text url into an integer address and then route to that address
- For internet addresses it is efficient to do this in two stages, because once you have the integer address you can access the resource more efficiently and you may do this many times
- For file-sharing, once the file is found, that likely constitutes the one and only time that that particular user will access that particular resource
- Hence relying on unstructured search is reasonable
- Even the search is less structured than domain name lookup since domain names are an exact one-to-one mapping, file searches are not

Review — Security

- Although we noted that human error is a large cause of security breaches our concern here was technical security, which was mostly achieved through the use of cryptography
- Our assumption is that the network, atop which our distributed system is constructed, is insecure. Messages may be, deleted, read, duplicated, modified and inserted
- A man-in-the-middle attack is one in which the attacker makes independent connections with two victims and relays the messages between them.
- You can apply this to beat a master in a blind game of Chess (or Go, etc)
  - Set up two games against two masters making sure you are black in one and white in the other
  - Mirror each players moves to the opposite board
  - You will win one game and lose the other
- Usually though the man-in-the-middle masquerades as each of the two

Review — Security

Cryptography

- Modern cryptography makes use of algorithms which distort a message such that it is difficult/infeasible to recover the original message without knowledge of the key
- Shared secret-key algorithms are symmetric and make use of the same key to both encrypt and decrypt the message
- A message is secure provided that no one else knows/discovers the shared secret key
- Such that: $D(K, E(K, M)) = M$
- Public/private key algorithms are not symmetric. One key is used to encrypt the message whilst a corresponding key is used to decrypt the encrypted message:
- If K_e and K_d are a key-pair: $D(K_d, E(K_e, M)) = M$
- Generally a person publishes their public key and anyone can send a secure message to them by encrypting it with the public key

Review — Security

Hybrid Protocols

- If Alice sends a message to bob $\{M\}_{K_{Bpub}} = E(K_{Bpub}, M)$
- Bob can decrypt this $M = D(K_{Bpriv}, \{M\}_{K_{Bpub}}$, but to reply with M' Bob must use Alice's public key: $\{M'\}_{K_{Apub}} = E(K_{Apub}, M')$
- Alice can decrypt this with her private key: $M' = D(K_{Apriv}, \{M'\}_{K_{Apub}})$
- This has the attractive advantage that no pre-agreed shared secret is required. Alice and Bob can be on opposite sides of the world and still communicate in secret without risk that the sharing of a shared secret key was eavesdropped
- However public key encryption algorithms are 100/1000 times slower than shared secret key encryption, if Alice and Bob are to have a prolonged communication this is slow
- Alternatively M could have been a new shared secret key
- This uses public-key crytography to set up shared secret-key cryptography giving the benefits of both kinds

Security — Review

Digital Signature

- Rather than encrypt a message with Bob's public key Alice can instead encrypt a message with her own private key
- Doing so means that to decrypt the message requires Alice's public key which is generally available, so the message is insecure
- However, since a message decrypted with Alice's public key must have been encrypted with Alice's secret key we know for sure that Alice must have encrypted (and sent) the message
- So as to avoid encrypting an entire document, Alice may compute a digest (similar to a checksum) of the document and encrypt the digest and attach it to the document
- Digital signatures fulfil well the properties of Authentication and Unforgeable but can fail to be Non-repudiable

Review — Security

Certificates

- All public-key cryptography, including digital signatures, suffer from the problem of authenticating a public key
- That is, being sure that the public key, or the entity providing the public key, really is that of the entity advertised
  - I can claim to be Microsoft
  - Just as easily I can give you a public key, claim that that key is Microsoft's public key, and that I am therefore Microsoft
- Certificates are essentially digital signatures attached to public keys (or other digital signatures)
- Of course certificates may also be falsely claimed in the same way however one "*certification authority*" may certify many public-keys/digital signatures, such that the receiver need only trust the *certification authority* to enable trust of many others
  - For example `https` signatures

Review — Security

Key Iteration

- If the attacker knows a plaintext/ciphertext pair (M/M_e), it can simply try all possible keys K_{poss} until $M_e = E(K_{poss}, M)$
- This represents a problem for public key cryptography since the attacker can generate as many plaintext/ciphertext pairs as they require
- It means that keys must be long enough
- In addition the message must be long enough, suppose the message was just two bytes long, there are only 65,536 possible messages, and the attack knows the encryption key (ie. the public key)
  - The attacker can therefore simply iterate over all possible messages, encrypting them with public key to see if they get the same ciphertext
  - Suppose an encrypted message M_e is heard by the attacker, which they know was encrypted with the public key K_p
  - The attacker simply tries all possible M' until $E(K_p, M') = M_e$

Any Questions

End of the Course!
Thank you for your Attention!
Good luck with your Exams!
Any Questions?