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

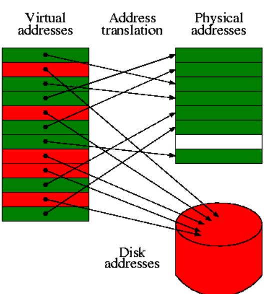
Operating Systems

- 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

Distribution and Operating Systems

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

Distribution and Operating Systems

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

Distribution and Operating Systems

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)

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

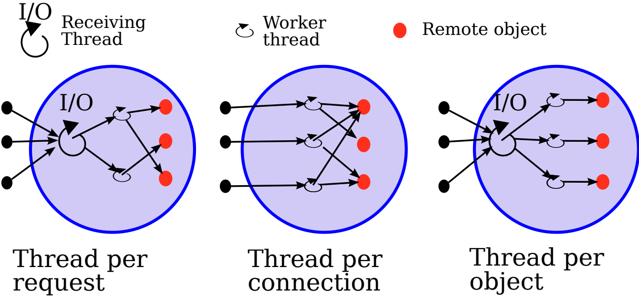
Threading and the Cache

- 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

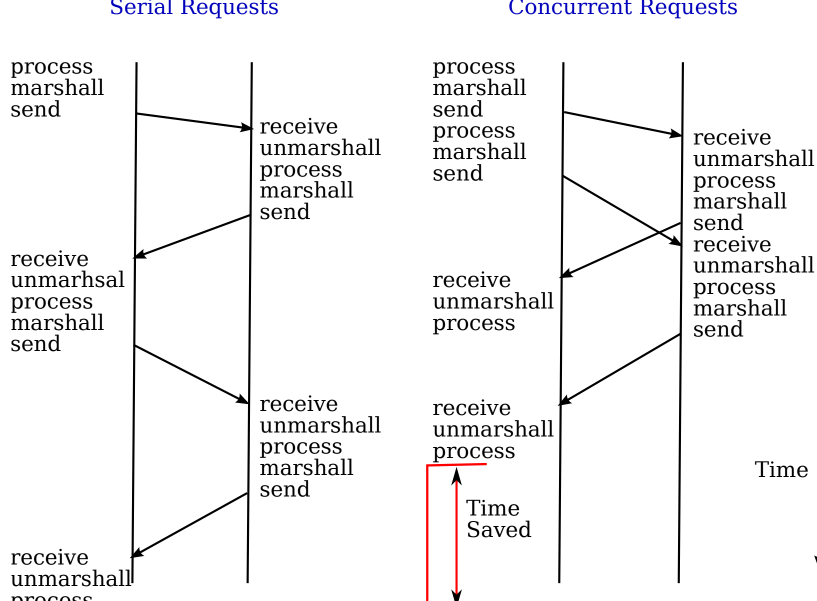
Operating Systems Support vs User Library

- 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

Distribution and Operating Systems

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

- $\textit{Message transmission time} = \textit{latency} + \frac{\textit{length}}{\textit{data transfer rate}}$
- 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

Distribution and Operating Systems

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

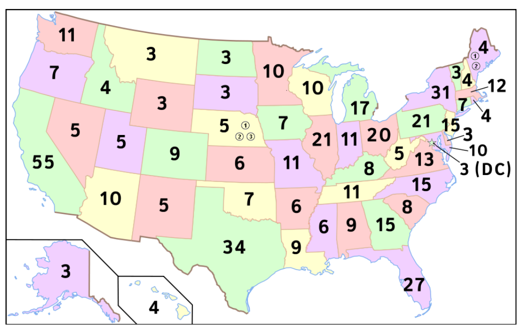
Any Questions

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:

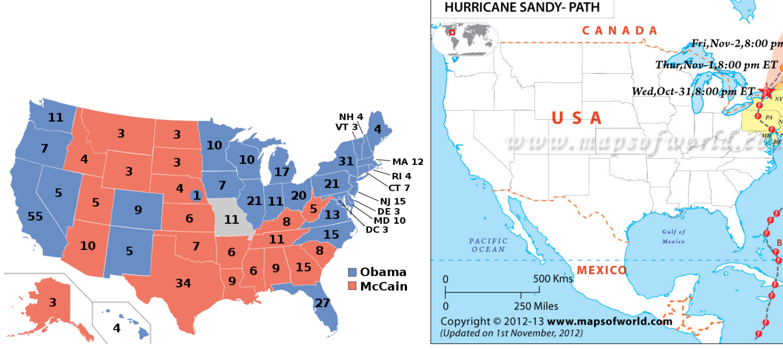
	DEM	REP	MARGIN
Polling average	48.9	46.3	Obama +2.6
Adjusted polling average	49.0	46.2	Obama +2.8
State fundamentals	50.4	44.4	Obama +6.0
Now-cast	49.1	46.0	Obama +3.1
Projected vote share ± 3.7	51.2	48.0	Obama +3.2
Chance of winning	80%	20%	

- 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