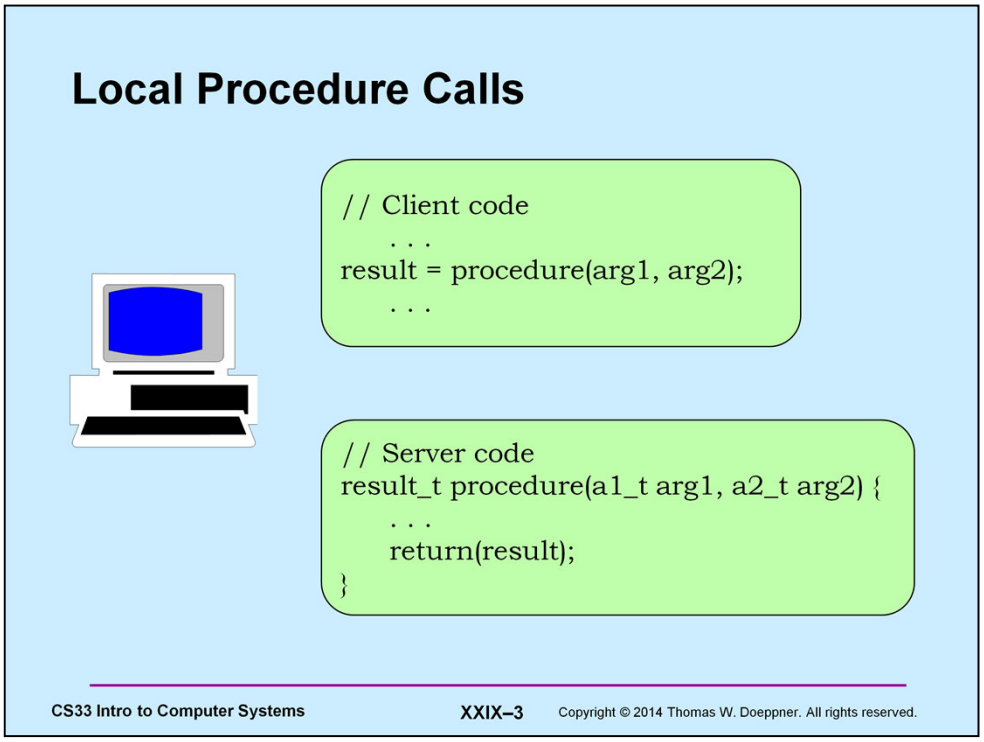
CS 33

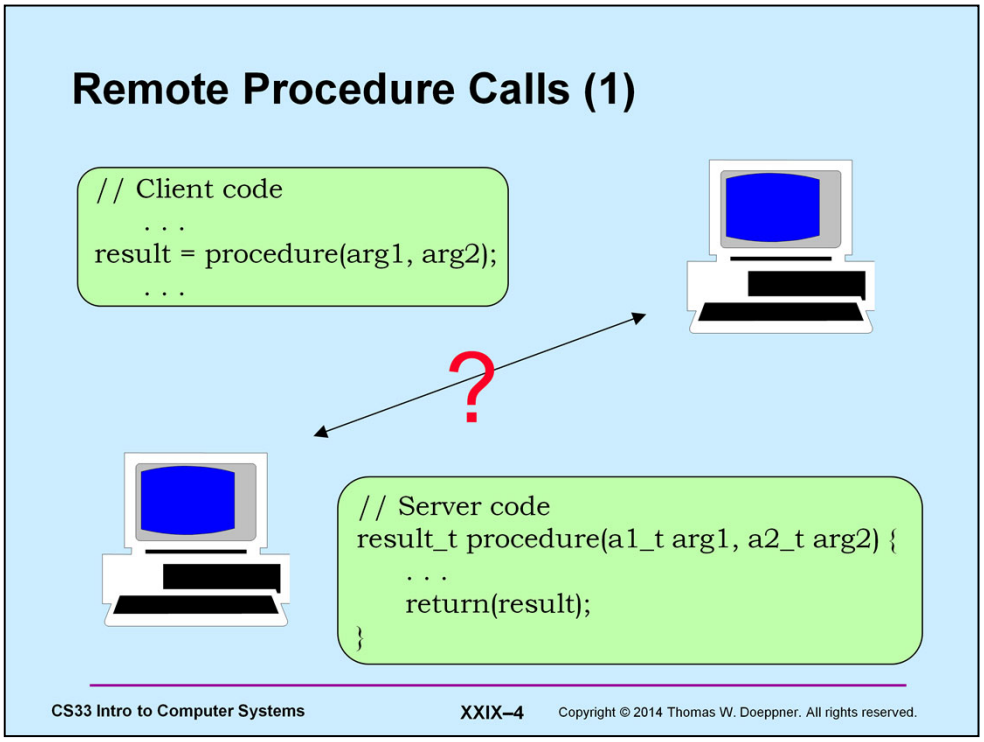
Intro to Distributed Systems

Topics

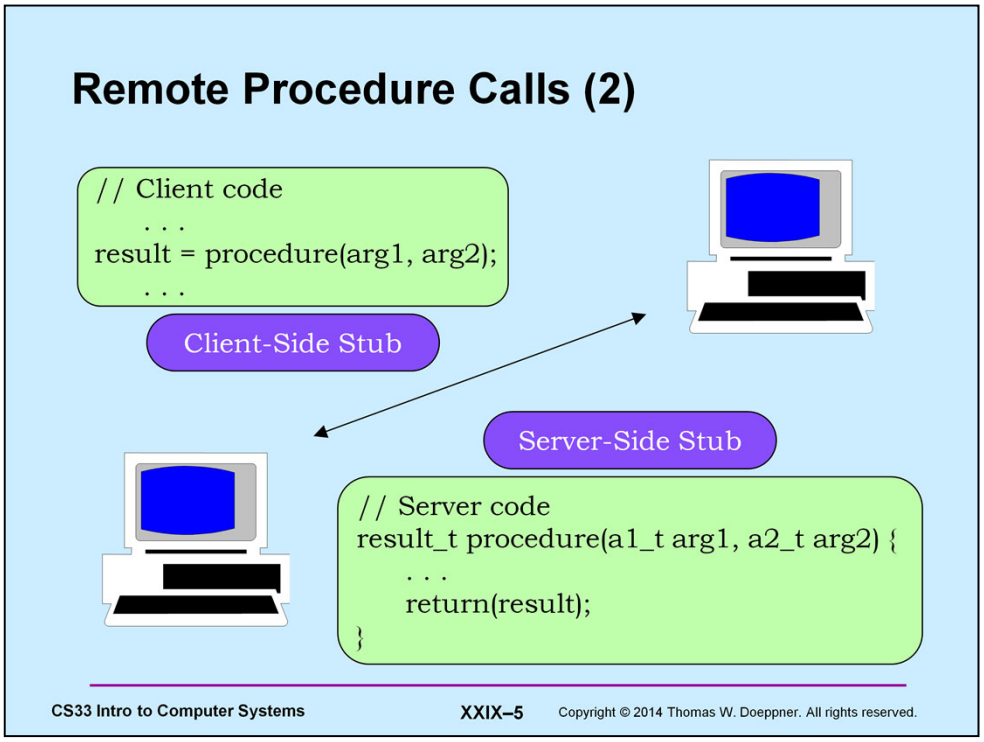
- Remote Procedure Call (RPC)
- Distributed File Systems
- Synchronizing Time



The basic theory of operation of RPC is pretty straightforward. But, to understand *remote* procedure calls, let's first make sure that we understand *local* procedure calls. The client (or caller) supplies some number of arguments and invokes the procedure. The server (or callee) receives the invocation and gets a (shallow) copy of the arguments. In the usual implementation, the callee's copy of the arguments have been placed on the runtime stack by the caller — the callee code knows exactly where to find them. When the call completes, a return value may be supplied by the callee to the caller. Some of the arguments might be out arguments — changes to their value are reflected back to the caller. This is handled in C indirectly — the actual argument, passed by copying, is a pointer to some value. The callee follows the pointer and modifies the value.

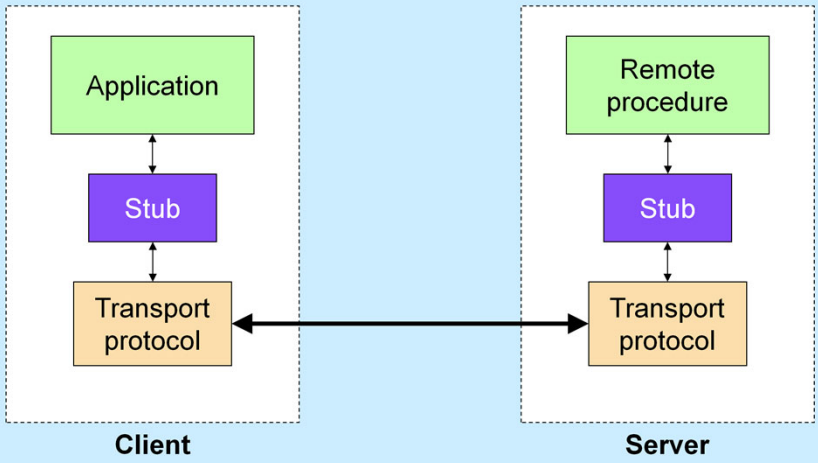


Now suppose that the client and server are on separate machines. As much as possible, we would like remote procedure calling to look and behave like local procedure calling. Furthermore, we would like to use the same languages and compilers for the remote case as in the local case. But how do we make this work? A remote call is very different from a local call. For example, in the local call, the caller simply puts the arguments on the runtime stack and expects the callee to find them there. In C, the callee returns data through out arguments by following a pointer into the space of the caller. These techniques simply don't work in the remote case.



The solution is to use *stub procedures*: the client places a call to something that has the name of the desired procedure, but is actually a proxy for it, known as the *client-side stub*. This proxy gathers together all of the arguments (actually, just the in and in-out arguments) and packages them into a message that it sends to the server. The server has a corresponding *server-side stub* that receives the invocation message, pulls out the arguments, and calls the actual (remote) procedure. When this procedure returns, returned data is packaged by the server-side stub into another message, which is transmitted back to the client-side stub, which pulls out the data and returns it to the original caller. From the points of view of the caller and callee procedure, the entire process appears to be a local procedure call—they behave no differently for the remote case.

Block Diagram



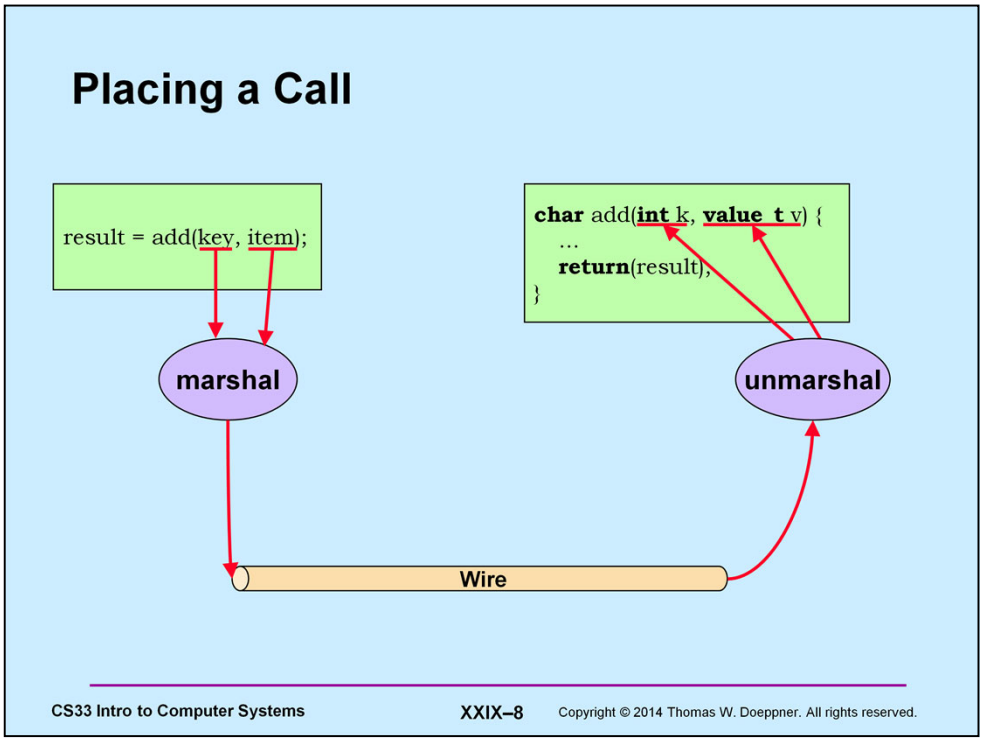
Example

```
typedef struct {
   int      comp1;
   long     comp2;
   float    comp3[6];
   char     *annotation;
} value_t;

typedef struct {
   value_t  element;
   struct list    *next;
} list_t;

char add(int key, value_t value);
char remove(int key, value_t value);
list_t query(int key);
```


Here's an example of a C declaration for a collection of simple database procedures.



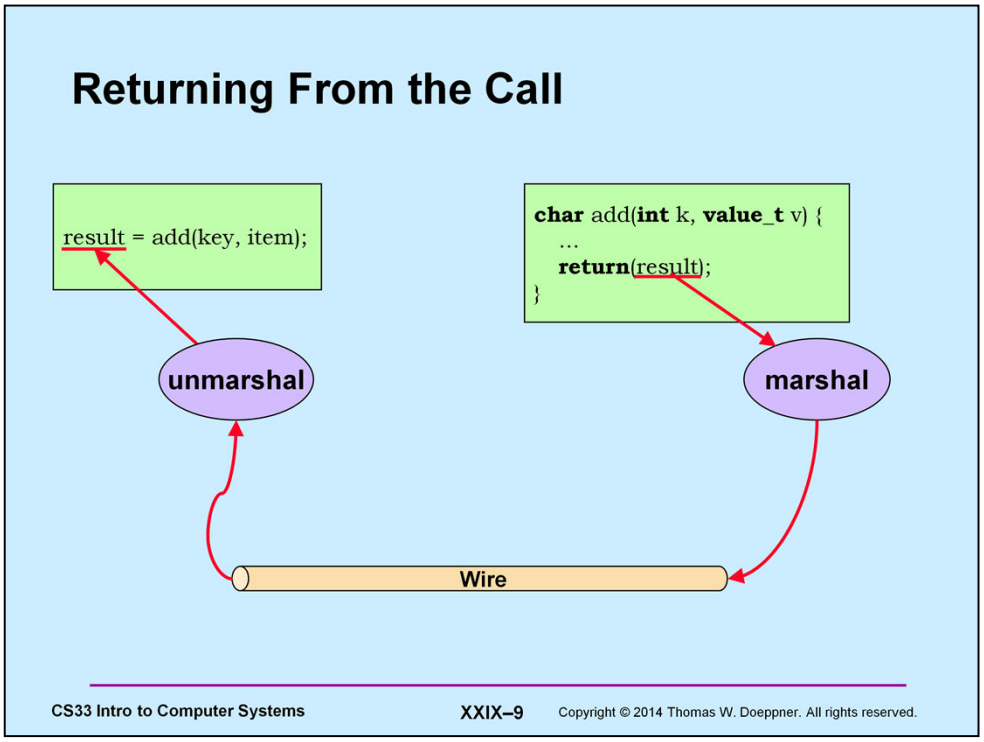
Placing a call involves gathering up the arguments and transmitting them. The gathering work is done by the stub, which *marshals* the arguments, that is, it puts them into a form suitable for transmission on the "wire." The data are sent via the transport protocol, usually either TCP/IP or UDP/IP, to the server, where they are *unmarshaled* by the server-side stub, which passes them to the remote procedure.

One of the issues that marshalling must deal with is the representation of the data. For example, the sending machine might represent integers using a little-endian byte ordering, while the receiving machine uses big-endian. Despite this, the caller and callee must be operating on the same value, even if coded differently.

One approach for handling this problem is for all parties to agree upon a common representation of each data type. Then, if its representation differs from the common one, a sender converts outgoing data to the common representation. The receiver converts if its representation differs from the common one. A disadvantage of this scheme is that conversion is required for communication between identical machines that happen to represent data differently from the standard.

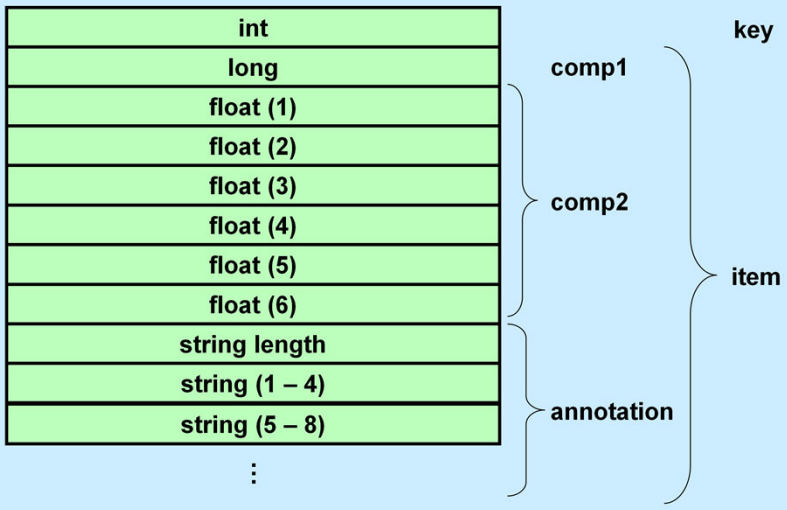
An alternative approach is for the sender transmits data in its native format, along with a code indicating what that format is. The receiver then converts only if necessary. The disadvantage here is that all machines must know about all data representations (though there aren't that many data representations).

Both approaches are used in practice. The RPC package underlying the NFS distributed file system uses the first approach.



When the call completes on the server, the return value is marshalled by the server-side stub, which then transmits them on the wire to the client, whose client-side stub unmarshals them and returns them to the original caller.

Marshalled Arguments



Marshalled Linked List

		array length
0:	value_t	next: 1
1:	value_t	next: 2
2:	value_t	next: 3
3:	value_t	next: 4
4:	value_t	next: 5
5:	value_t	next: 6
6:	value_t	next: -1

Distributed File Systes

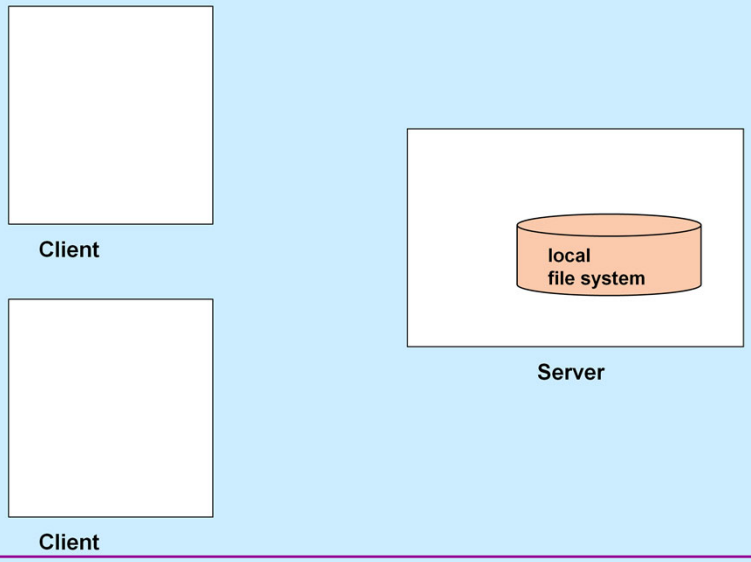
- Network File System (NFS)
  - used by Unix systems
- Common Internet File System (CIFS)
  - used by Windows systems

Guiding Principle

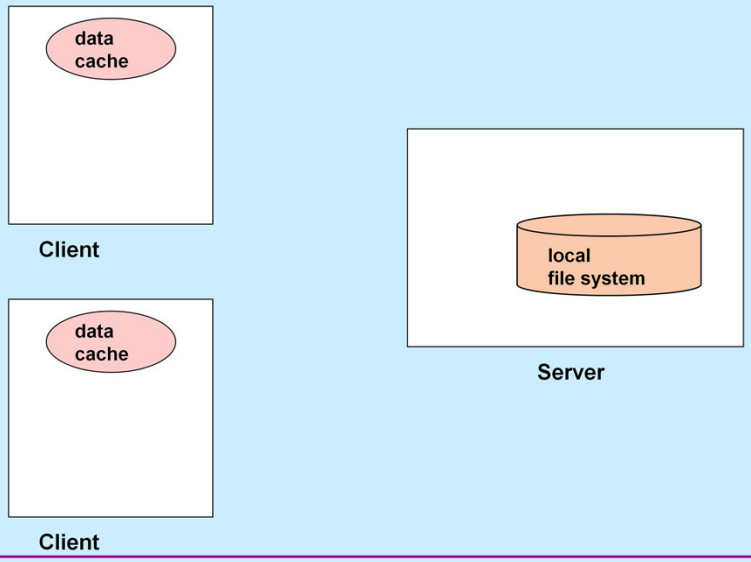
Principle of least astonishment (PLA)

- people don't like surprises, particularly when they come from file systems

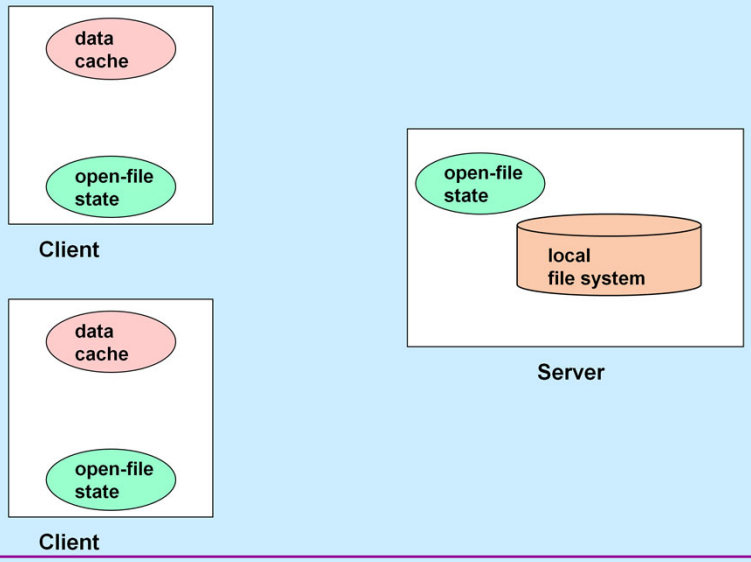
Distributed File Systems



Distributed File Systems



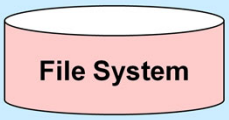
Distributed File Systems

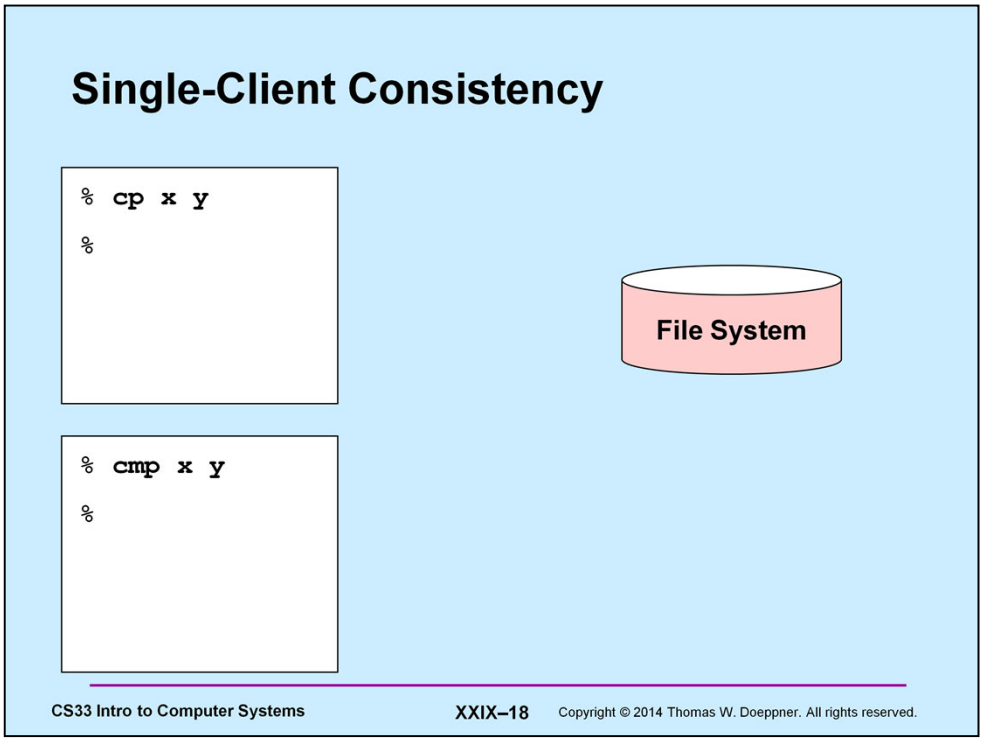


Single-Thread Consistency

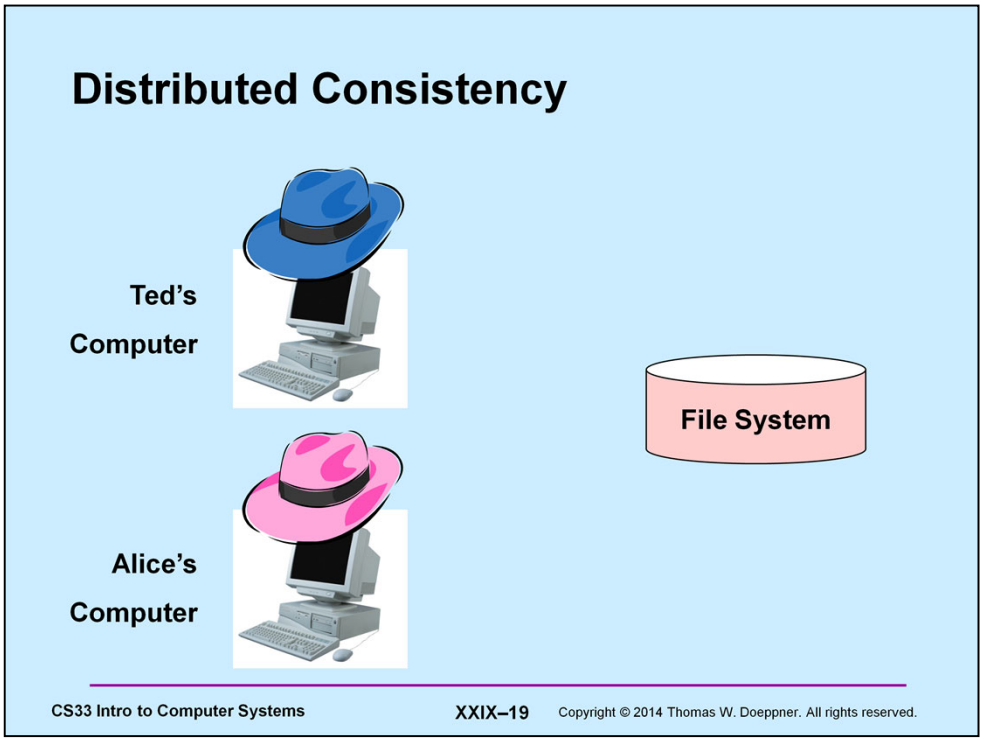
```
write(fd, buf1, size1);
read(fd, buf2, size2);

// no surprises if
// single-thread consistent
// Operations are time-ordered
```





Here the operations are done on separate processes on the same computer. The “cp” takes place just before the “cmp”.

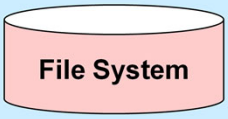


Ted and Alice are using different computers that both access a file system on a server.

Strict Consistency

Ted's Computer

```
write(fd1, "A", 2);
write(fd2, "B", 2);
```



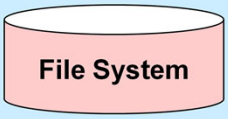
Alice's Computer

```
// an instant later …
read(fd1, buf1, 2);
read(fd2, buf2, 2);
// buf1 contains "A"
// buf2 contains "B"
```


Weak Consistency

Ted's Computer

```
write(fd1, "A", 2);
write(fd2, "B", 2);
```



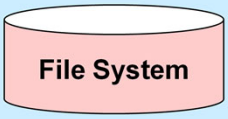
Alice's Computer

```
// a while later …
read(fd1, buf1, 2);
read(fd2, buf2, 2);
// maybe buf1 contains "A"
// maybe buf2 contains "B"
```


Sequential Consistency

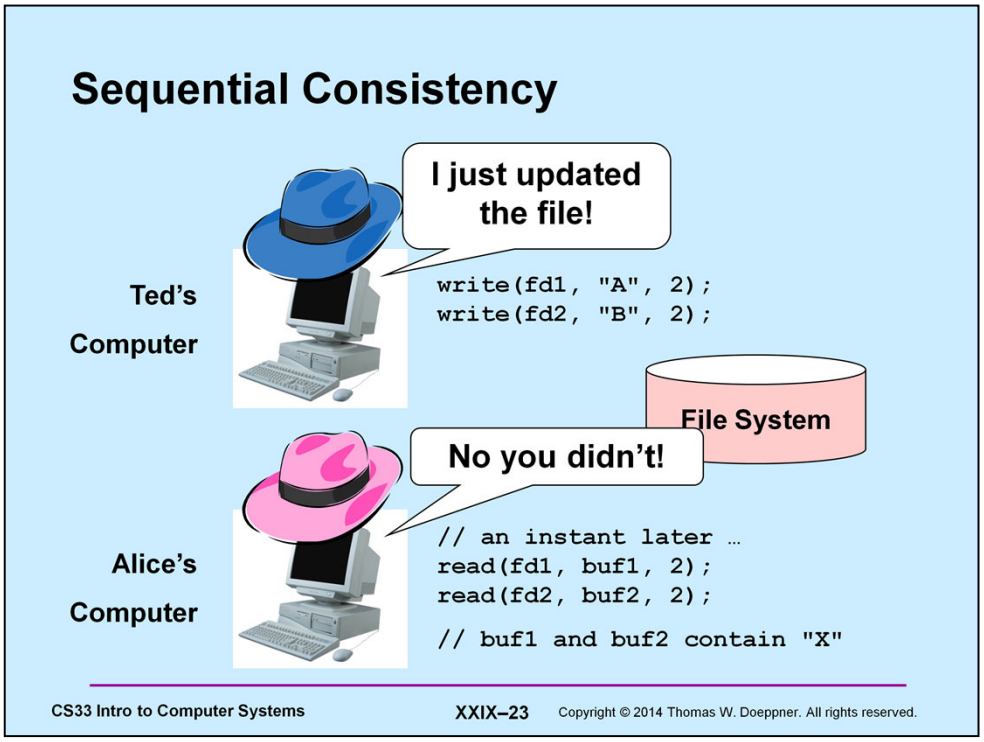
Ted's Computer

```
write(fd1, "A", 2);
write(fd2, "B", 2);
```



Alice's Computer

```
// an instant later …
read(fd1, buf1, 2);
read(fd2, buf2, 2);
// if buf2 contains "B"
// then buf1 contains "A"
```

Assume that both files initially contain “X”.

Entry Consistency

Ted's Computer

```
writelock(fd);
write(fd, "B", 2);
unlock(fd);
```

File System

Alice's Computer

```
// an instant later …
readlock(fd);
read(fd, buf, 2);
unlock(fd);
// buf now contains "B"
```

NFS vs. CIFS …

- NFS
  - single-client consistent
  - entry-consistent if all participants use locks
  - otherwise weakly consistent
- CIFS
  - strictly consistent

Thursday morning, November 17th
At 7:00 a.m.

Maytag, the department's central file server, will be taken down to kick off a filesystem consistency check.

Linux machines will hang.

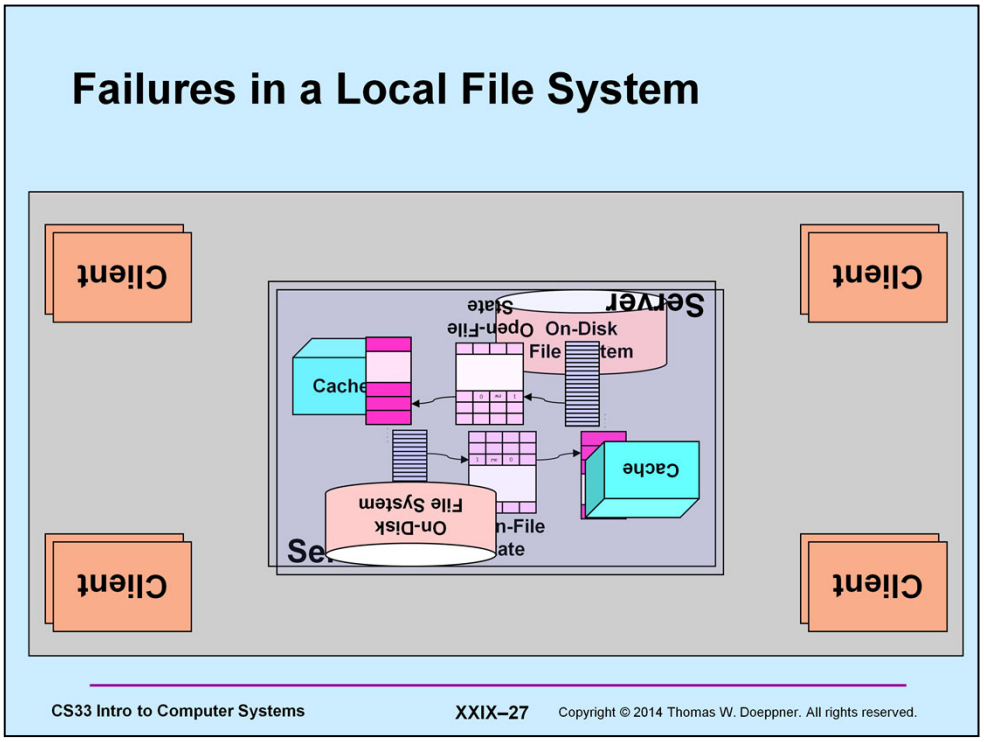
All Windows users should log off.

Normal operation will resume by 8:30 a.m. if all goes well.

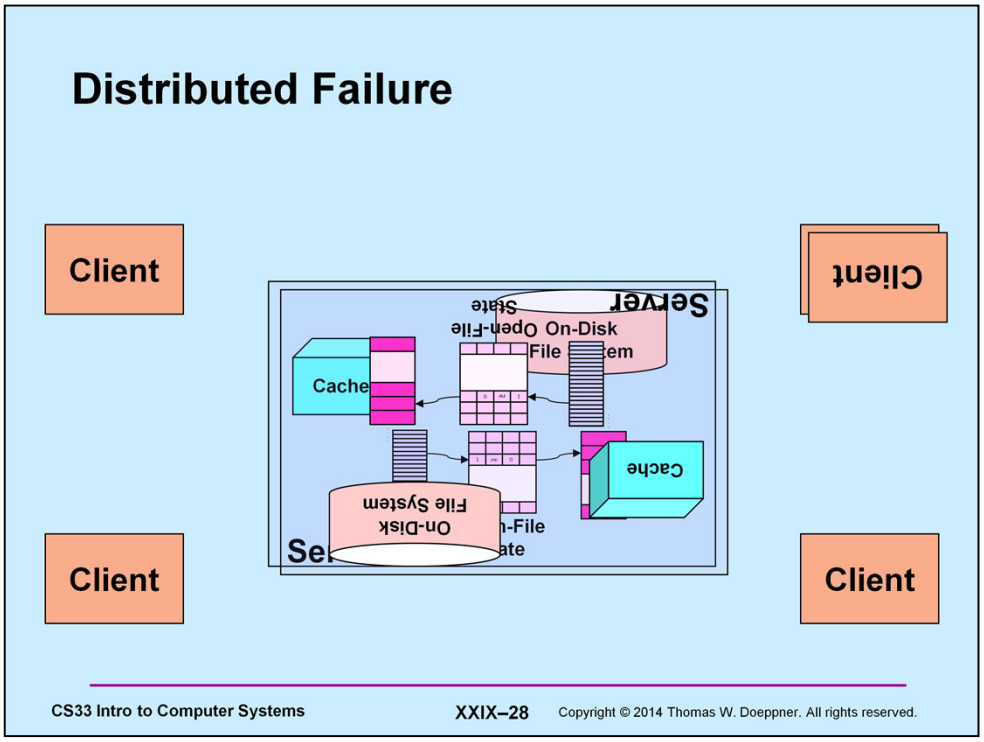
All windows users should log off before this time.

Questions/concerns to problem@cs.brown.edu

(Note that the November 17 in question was in 2005.)

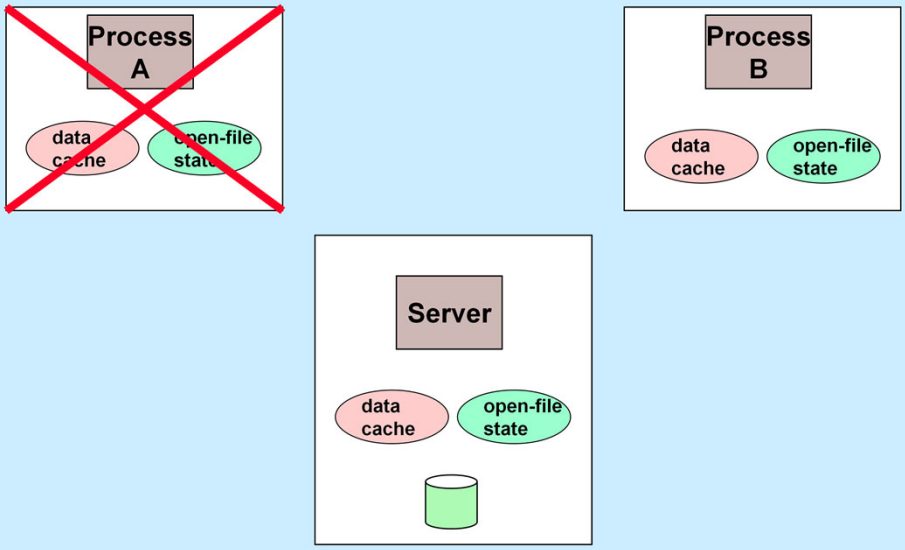


What we're accustomed to with local file systems is that, in the event of a crash, everything goes down. This is simple to deal with.

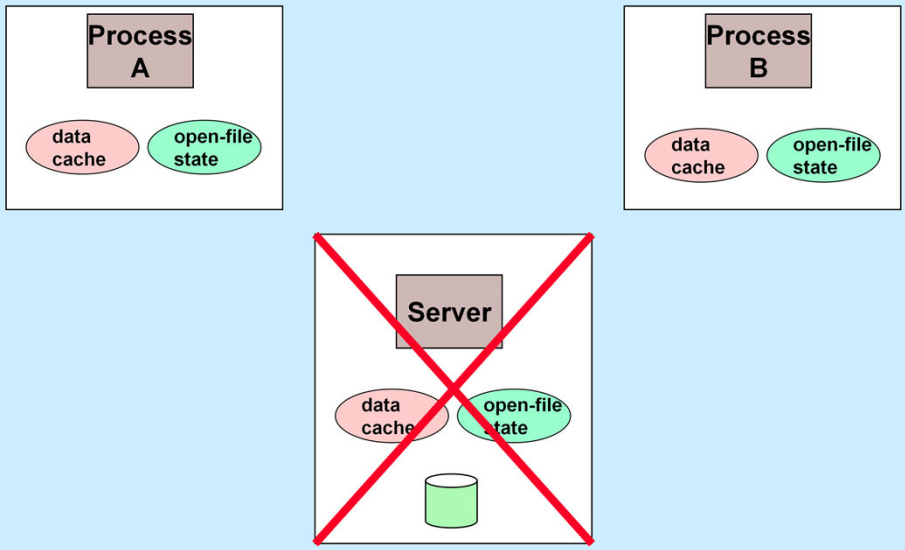


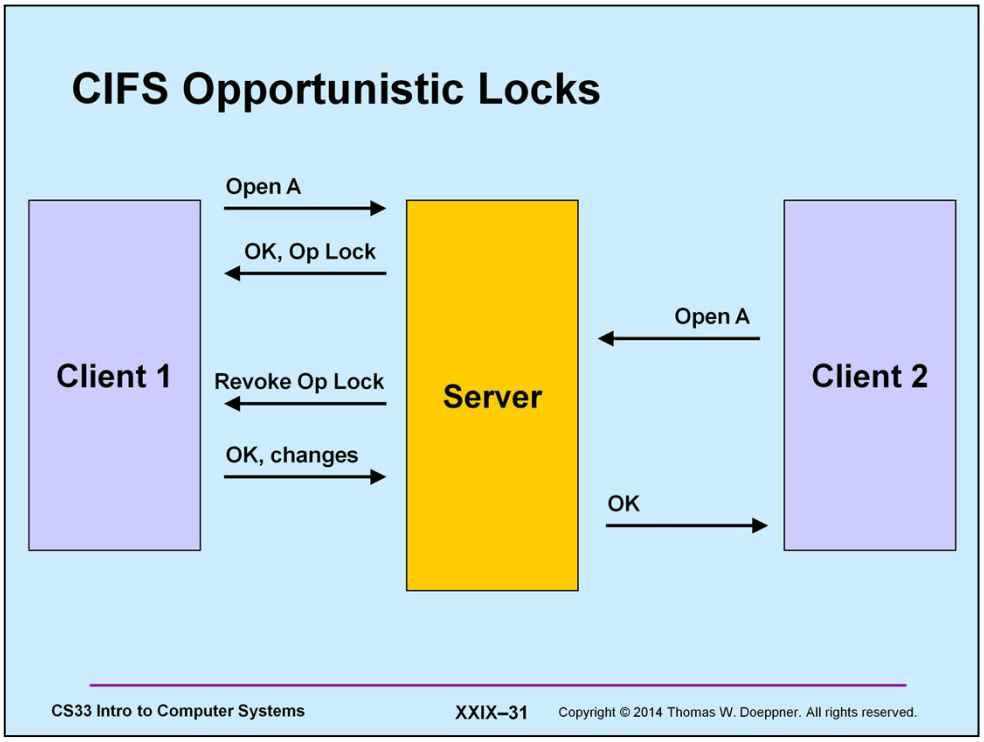
In a distributed system, if the server crashes, there is no inherent reason for clients to crash as well. Assuming there was no damage done to the on-disk file system, client processes might experience a delay while the server is down, but should be able to continue execution once the server comes back up, as if nothing had happened. The crash of a client computer is bad news for the processes running on that computer, but should have no adverse effect on the server or on other client computers. We'd like the effect to be as if the client processes on the crashed computer had suddenly closed all their files and terminated.

NFS Client Crash Recovery



NFS Server Crash Recovery





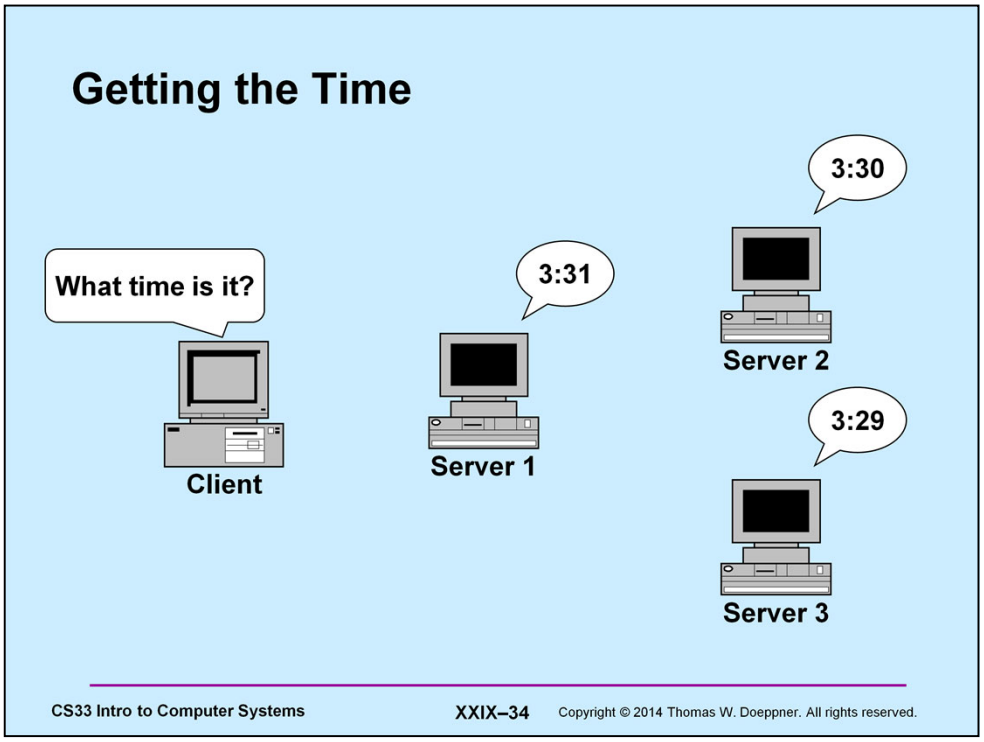
Opportunistic locks are used by CIFS to optimize mandatory locking. However, it's intolerant of server failures.

NFS vs. CIFS …

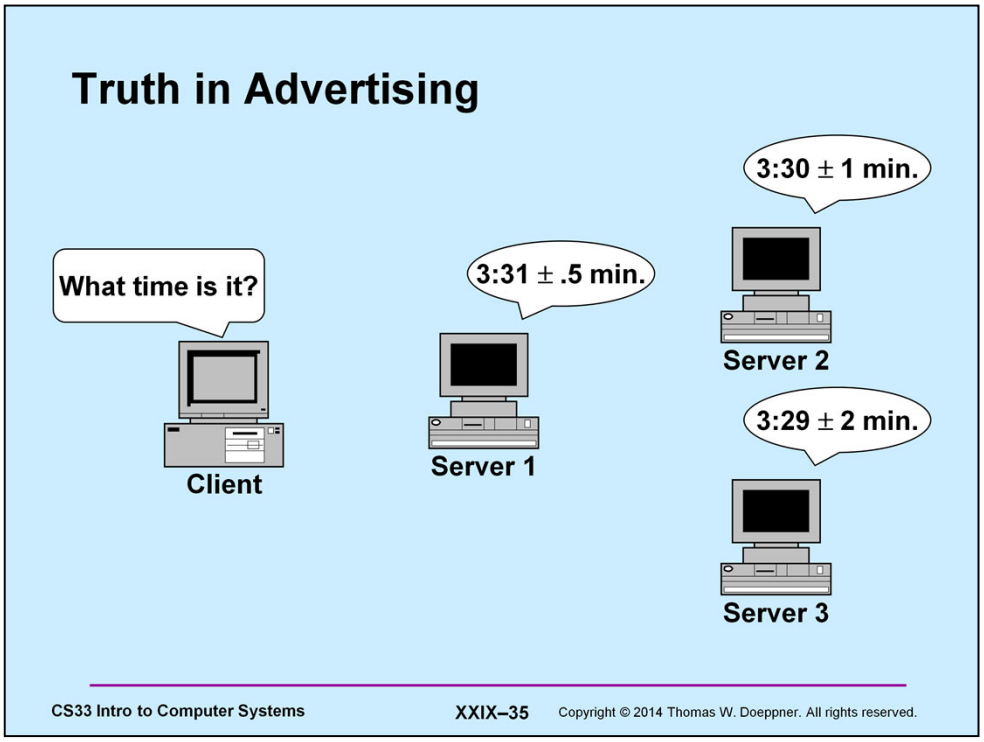
- NFS
  - fairly tolerant of crashes
  - but doesn't support mandatory locking
- CIFS
  - fairly intolerant of crashes
  - but supports mandatory locking

By "NFS", we mean NFSv3, which is used in the CS department. NFSv4 tolerates crashes very well and handles mandatory locking, with some caveats that are ignored by CIFS.

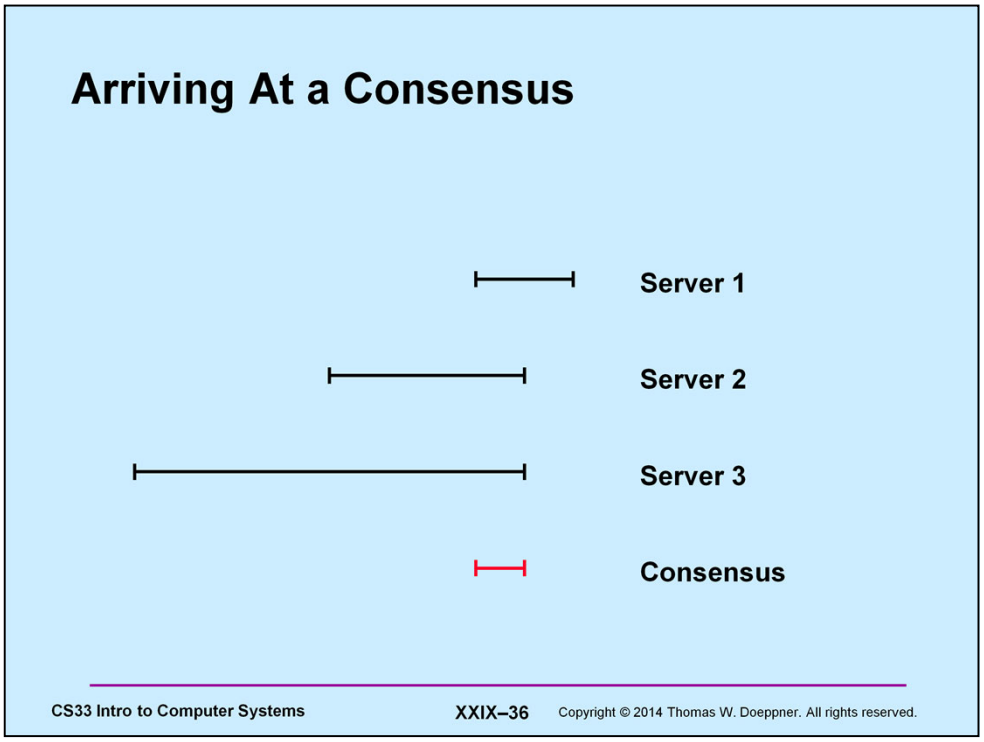
Time



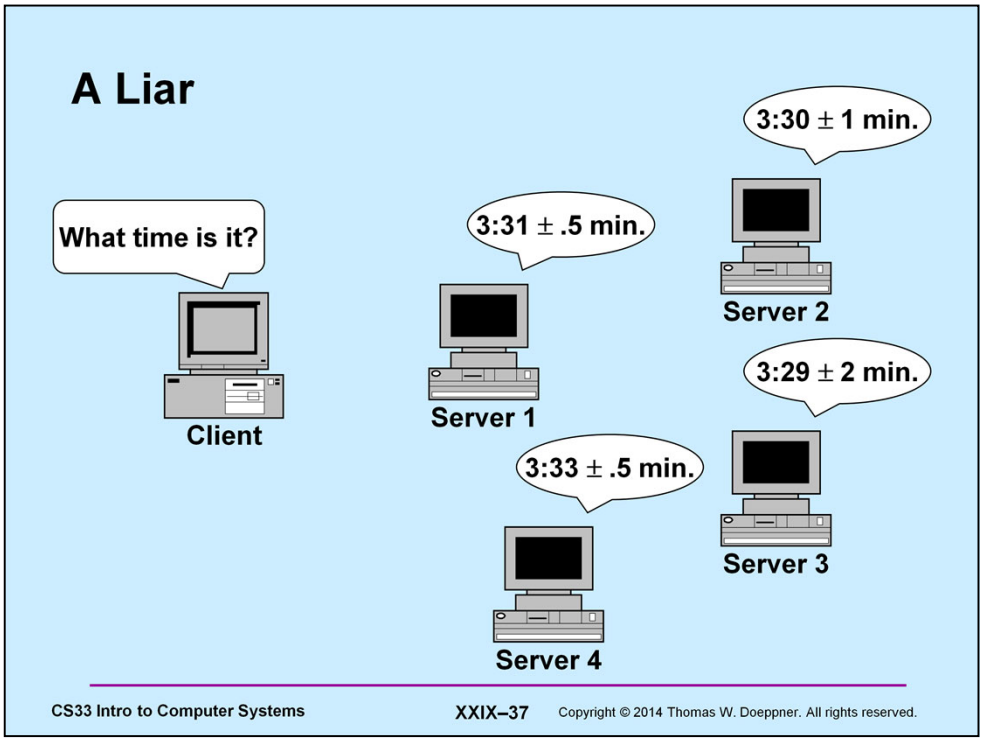
A client might decide to improve its chances of getting the correct time by consulting a number of servers and then somehow "averaging" the results. However, if each server gives it a different result (and each server claims to be correct), there is no basis by which averaging can be applied.



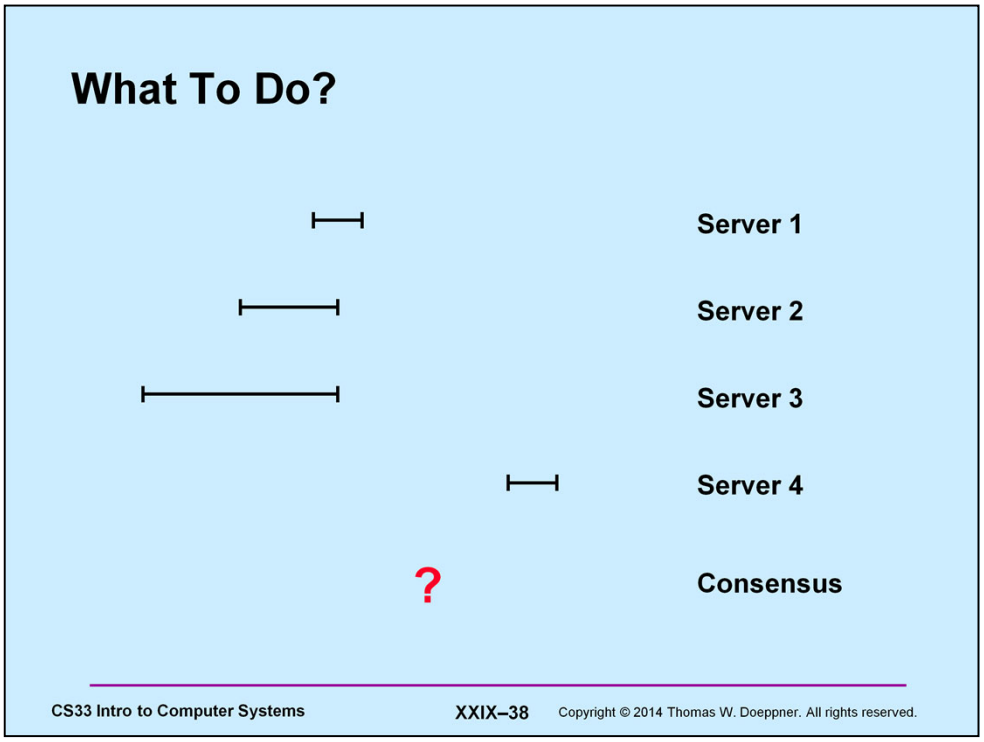
A responsible server will reply not only with what it considers to be the correct time but also with a bound on the accuracy of this time estimate. Thus the client obtains a set of intervals from which it can determine a current time interval that is tighter than any interval obtained from individual servers.



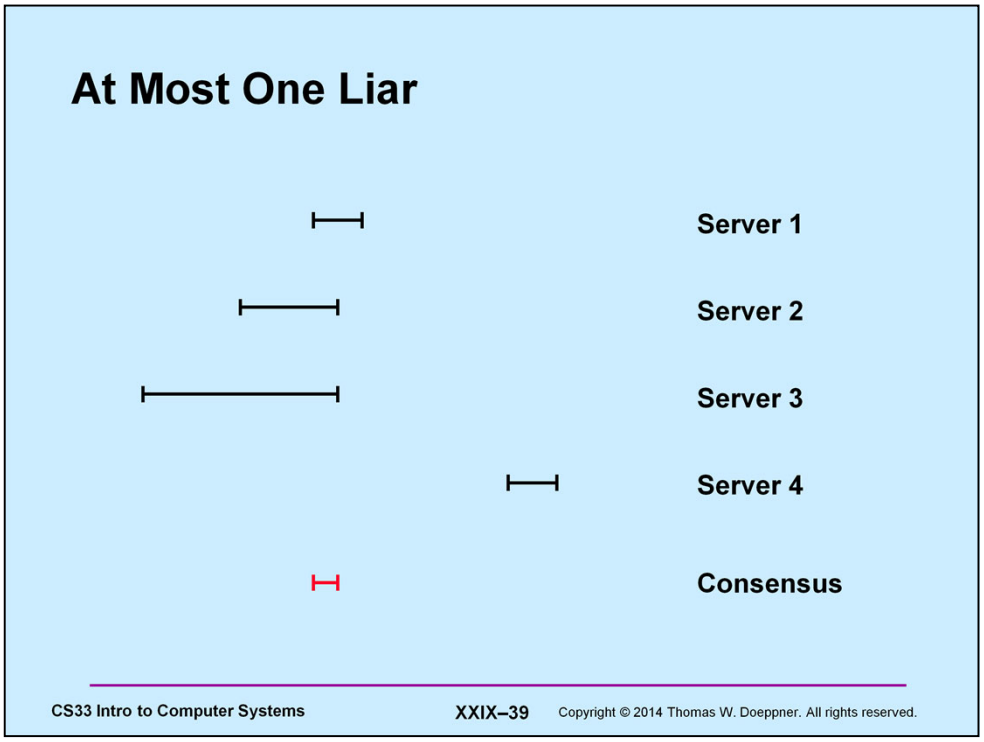
If each server has supplied a correct interval representing the current time, then the current time must lie in the intersection of all of the servers' intervals. Though we don't know exactly where this point is, the client can use the intersection interval as its current time interval (since the intersection interval contains the correct time).



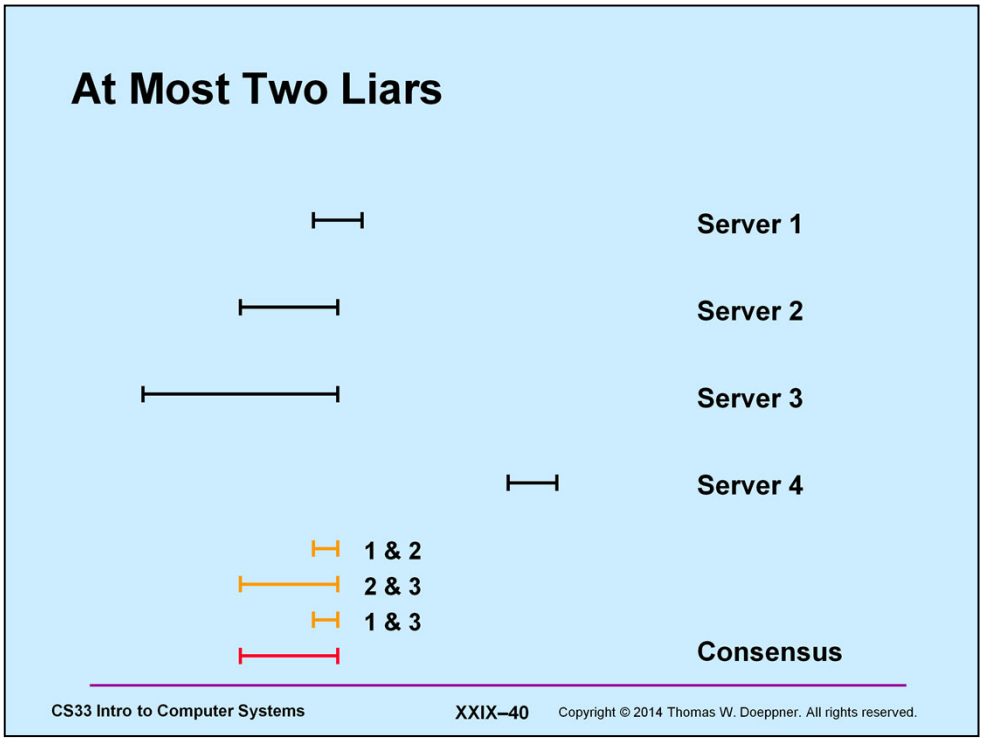
One or more of the servers might be totally wrong about the time of day.



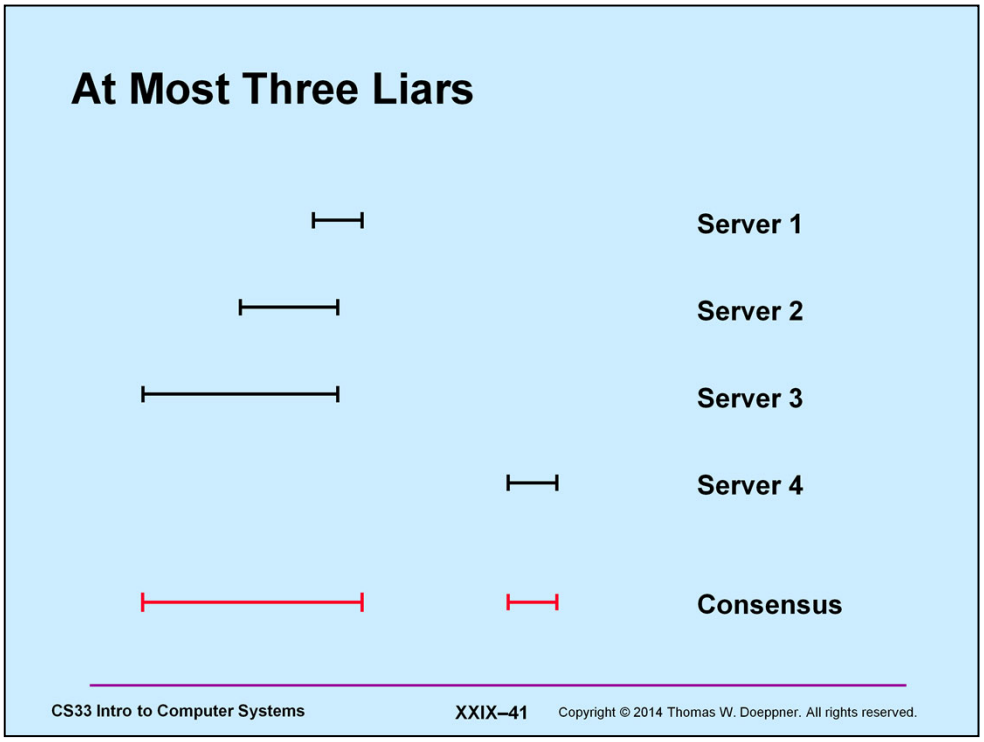
The intersection of these intervals is null. If we assume that there is at most one faulty server, then any point contained in at least three of the intervals can represent the correct time. In general, if we have m servers and at most f of them are faulty, then any point contained in at least $m - f$ of the time intervals can be correct.



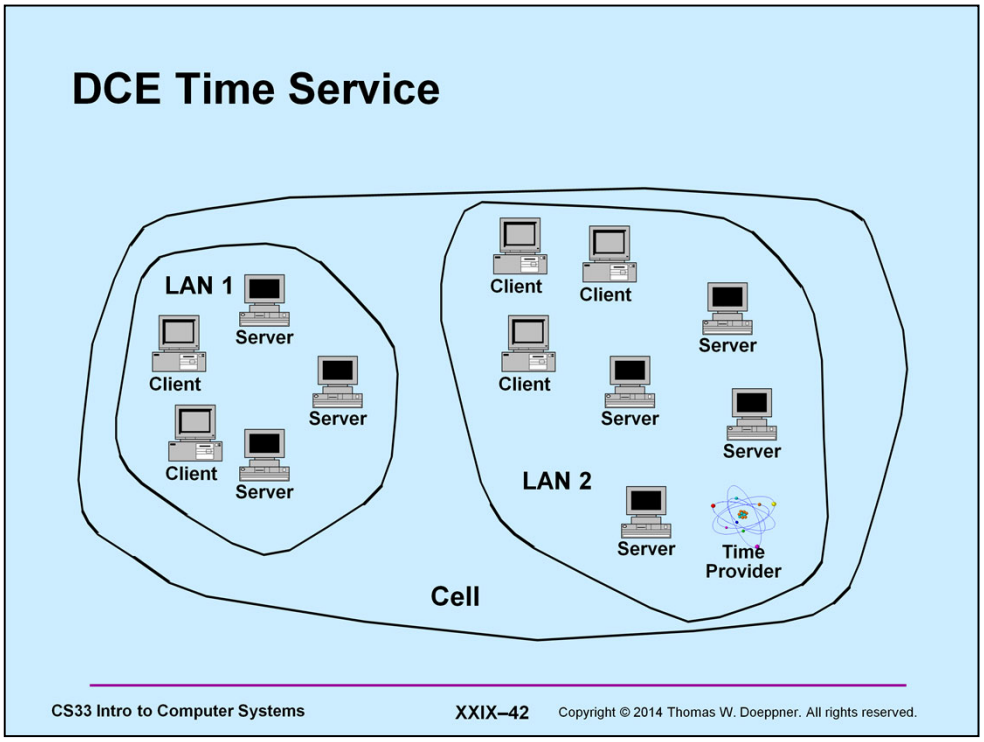
If there is at most one liar, then at least three of the servers must be telling the truth. In this case, the truth-sayers must be the first three servers and the consensus is as before.



However, if all we can determine is that there are at most two liars, then we must look at possible pairs of truth-sayers. Server 4 is still ruled out, but the two truth-sayers might be 1 and 2, or 2 and 3, or 1 and 3. Thus the correct time lies in the union of their intersections.



If all we can say is that there are at most three liars, then any one of the servers might be telling the truth. In this case, the correct time lies in the union of all four intervals, resulting in a disjoint interval.

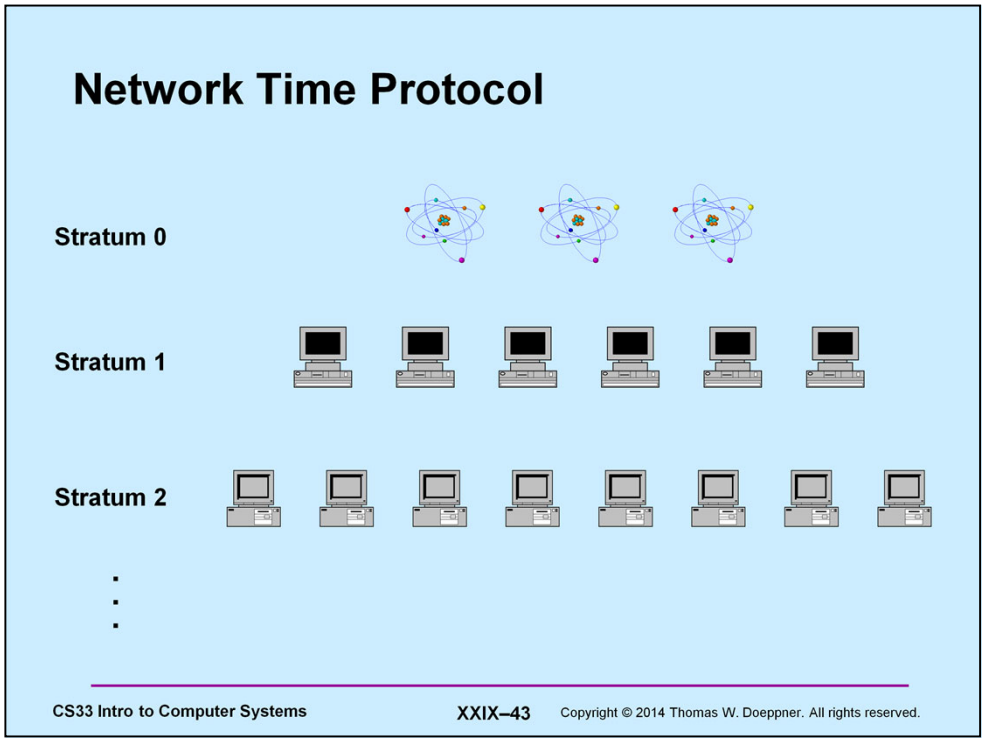


The basic approach to clock synchronization that we just discussed is due to Keith Marzullo: K. A. Marzullo. *Maintaining the Time in a Distributed System: An Example of a Loosely-Coupled Distributed Service*. Ph.D. dissertation, Stanford University, Department of Electrical Engineering, February 1984. The DCE Distributed Time Service (DTS), based on the Digital Time Synchronization Service (DTSS) specification, is provided by a collection of clerks and time servers. Clerks reside on individual computers and are responsible for maintaining the time for their local computers. They contact the time servers, acting as clients, to obtain the information necessary to keep their clocks reasonably accurate.

Time servers fall into two categories: local time servers and global time servers. A collection of local time servers is responsible for keeping the time on an individual LAN. (LAN, in this context, probably means local area network, but really means a collection of computers which are relatively close to one another (in terms of communication delays).) These servers periodically synchronize their clocks with one another's. On a more frequent basis, clerks synchronize with (at least) a subset of the local time servers' clocks and, possibly, with some global servers.

Global time servers are time servers in other LANs (within the cell) that clerks and local time servers may contact if additional sources of time information are needed. The idea is that local time servers are, with respect to communication time, near one another and the clerks, while global time servers are somewhat farther away.

The ultimate source of time is the time provider (TP). This provides the time along with a (presumably tiny) inaccuracy. It might be an atomic clock, or some other time service such as NTP (Network Time Protocol) of the Internet. A time server synchronizes with a TP if one is available (otherwise it synchronizes with other time servers).



The Internet's Network Time Protocol (NTP) also uses the Marzullo approach, though things are organized a bit differently. See http://www.eecis.udel.edu/~mills/ntp/html/index.html#docs for details. Time servers are organized into strata, depending on their "distance" from the ultimate time source. Stratum 0 is the collection of time sources themselves, stratum 1 is the collection of servers connected directly to time sources, etc. Servers of one stratum connect to a number of servers/devices at the next lower stratum, rule out liars, and average the time of apparent truth-sayers. (In NTP terminology, liars are known as "falsetickers" and truthful servers are "truechimers.") Windows uses a simplified version of NTP known as SNTP (simple NTP) — clients contact just one server. Linux and most other Unix implementations use the full-blown NTP protocol, even for clients. Brown University provides a stratum-2 time server at ntp.brown.edu.

What's Time?

- GMT
  - determined by astronomical observations at Greenwich
  - improved versions, accounting for irregularities of earth's rotation and orbit, are UT0, UT1, UT2
- International Atomic time (TAI)
  - based on transitions of energy levels of cesium atom
  - synchronized with earth time in 1958
  - earth has been running slow since then
    - » UTC: leap seconds added as necessary to adjust

UTC is pronounced *coordinated universal time* in English and *temps universel coordonne* in French. "UTC" was chosen apparently so as not to show favoritism to either language. To find the official US time, go to http://www.time.gov/timezone.cgi?Eastern/d/-5/java.