Distributed Systems

Clock Synchronization: Physical Clocks

Paul Krzyzanowski

pxk@cs.rutgers.edu

What’s it for?

- Temporal ordering of events produced by concurrent processes
- Synchronization between senders and receivers of messages
- Coordination of joint activity
- Serialization of concurrent access for shared objects

Physical clocks

Logical vs. physical clocks

Logical clock keeps track of event ordering

- among related (causal) events

Physical clocks keep time of day

- Consistent across systems

Quartz clocks

- 1880: Piezoelectric effect
  - Curie brothers
  - Squeeze a quartz crystal & it generates an electric field
  - Apply an electric field and it bends
- 1929: Quartz crystal clock
  - Resonator shaped like tuning fork
  - Laser-trimmed to vibrate at 32,768 Hz
  - Standard resonators accurate to 6 parts per million at 31° C
  - Watch will gain/lose < ½ sec/day
  - Stability > accuracy: stable to 2 sec/month
  - Good resonator <u>can</u> have accuracy of 1 second in 10 years
    - Frequency changes with age, temperature, and acceleration

Atomic clocks

- Second is defined as 9,192,631,770 periods of radiation corresponding to the transition between two hyperfine levels of cesium-133

- Accuracy:
  better than 1 second in six million years

- NIST standard since 1960

UTC

- UT0
  - Mean solar time on Greenwich meridian
  - Obtained from astronomical observation
- UT1
  - UT0 corrected for polar motion
- UT2
  - UT1 corrected for seasonal variations in Earth's rotation
- UTC
  - Civil time measured on an atomic time scale

UTC

- Coordinated Universal Time
- Temps Universel Coordonné
  - Kept within 0.9 seconds of UT1
  - Atomic clocks cannot keep mean time
    - Mean time is a measure of Earth's rotation

Physical clocks in computers

Real-time Clock: CMOS clock (counter) circuit driven by a quartz oscillator

- battery backup to continue measuring time when power is off

OS generally programs a timer circuit to generate an interrupt periodically

- e.g., 60, 100, 250, 1000 interrupts per second
  (Linux 2.6+ adjustable up to 1000 Hz)
- Programmable Interval Timer (PIT) – Intel 8253, 8254
- Interrupt service procedure adds 1 to a counter in memory

Problem

Getting two systems to agree on time

- Two clocks hardly ever agree
- Quartz oscillators oscillate at slightly different frequencies

Clocks tick at different rates

- Create ever-widening gap in perceived time
- Clock Drift

Difference between two clocks at one point in time

- Clock Skew

8:00:00

8:00:00

Sept 18, 2006
8:00:00

8:01:24

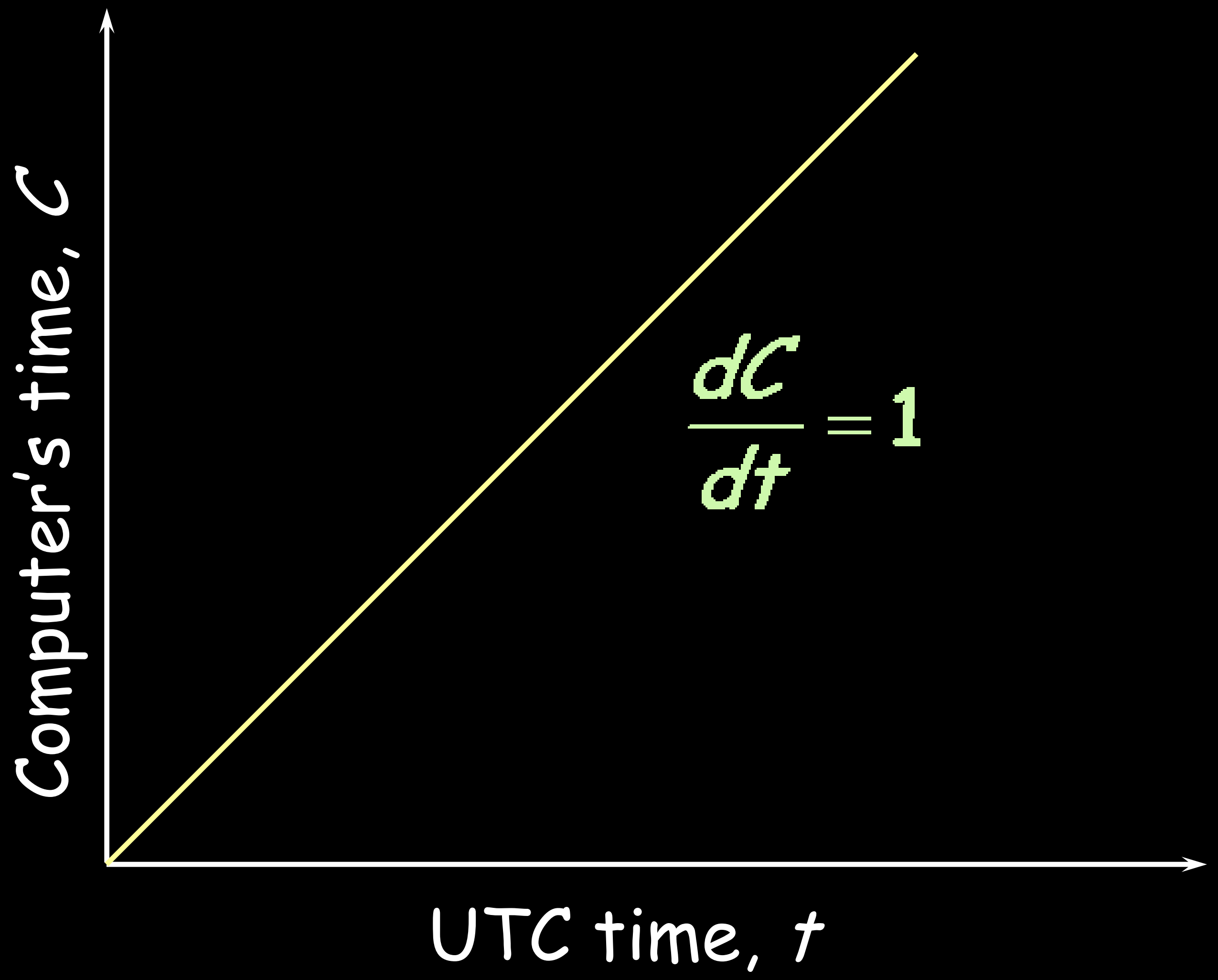
Skew = +84 seconds
+84 seconds/35 days
Drift = +2.4 sec/day

Oct 23, 2006
8:00:00

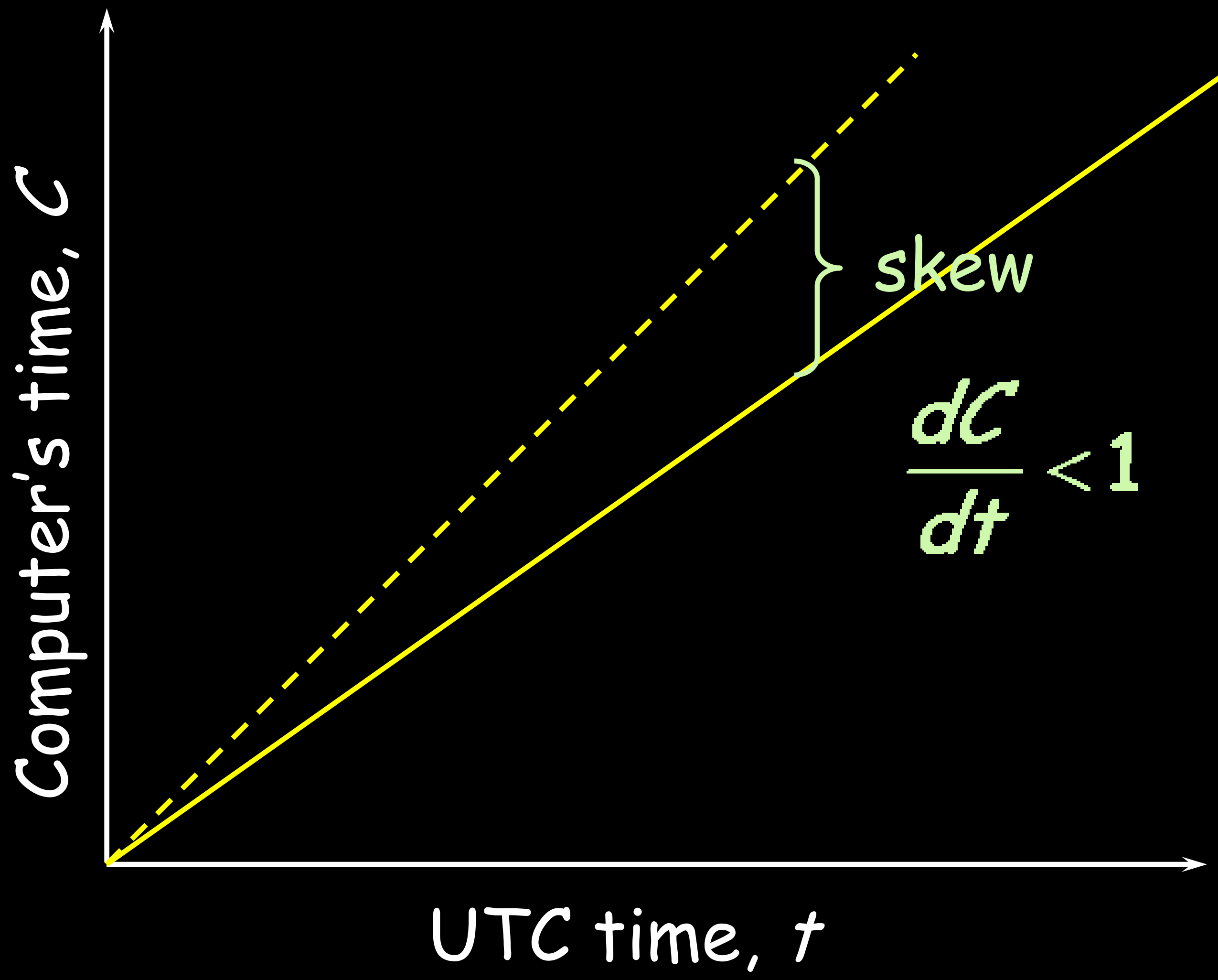
8:01:48

Skew = +108 seconds
+108 seconds/35 days
Drift = +3.1 sec/day

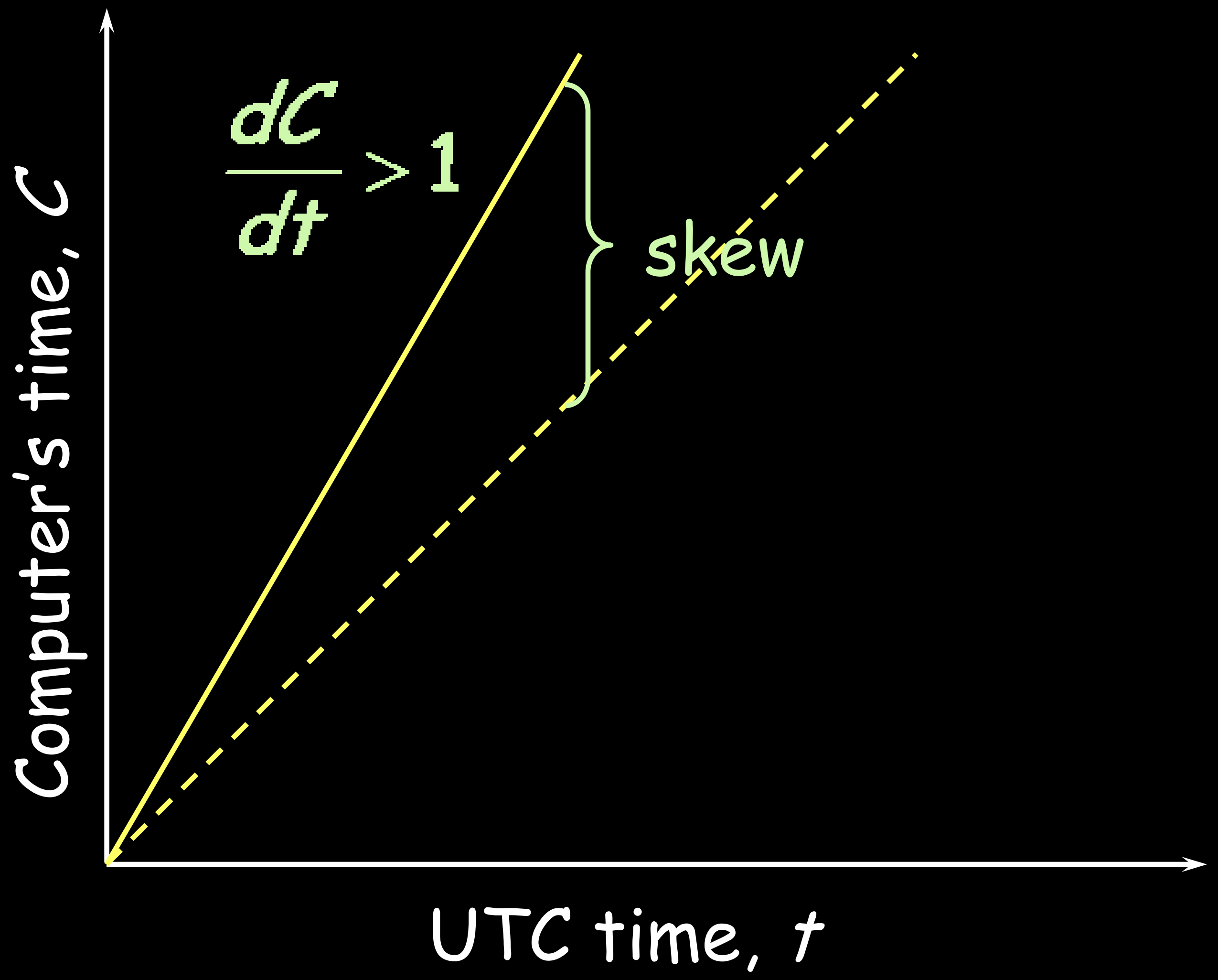
Perfect clock



Drift with slow clock



Drift with fast clock



Dealing with drift

Assume we set computer to true time

Not good idea to set clock back

- Illusion of time moving backwards can confuse message ordering and software development environments

Dealing with drift

Go for *gradual* clock correction

If fast:

Make clock run slower until it synchronizes

If slow:

Make clock run faster until it synchronizes

Dealing with drift

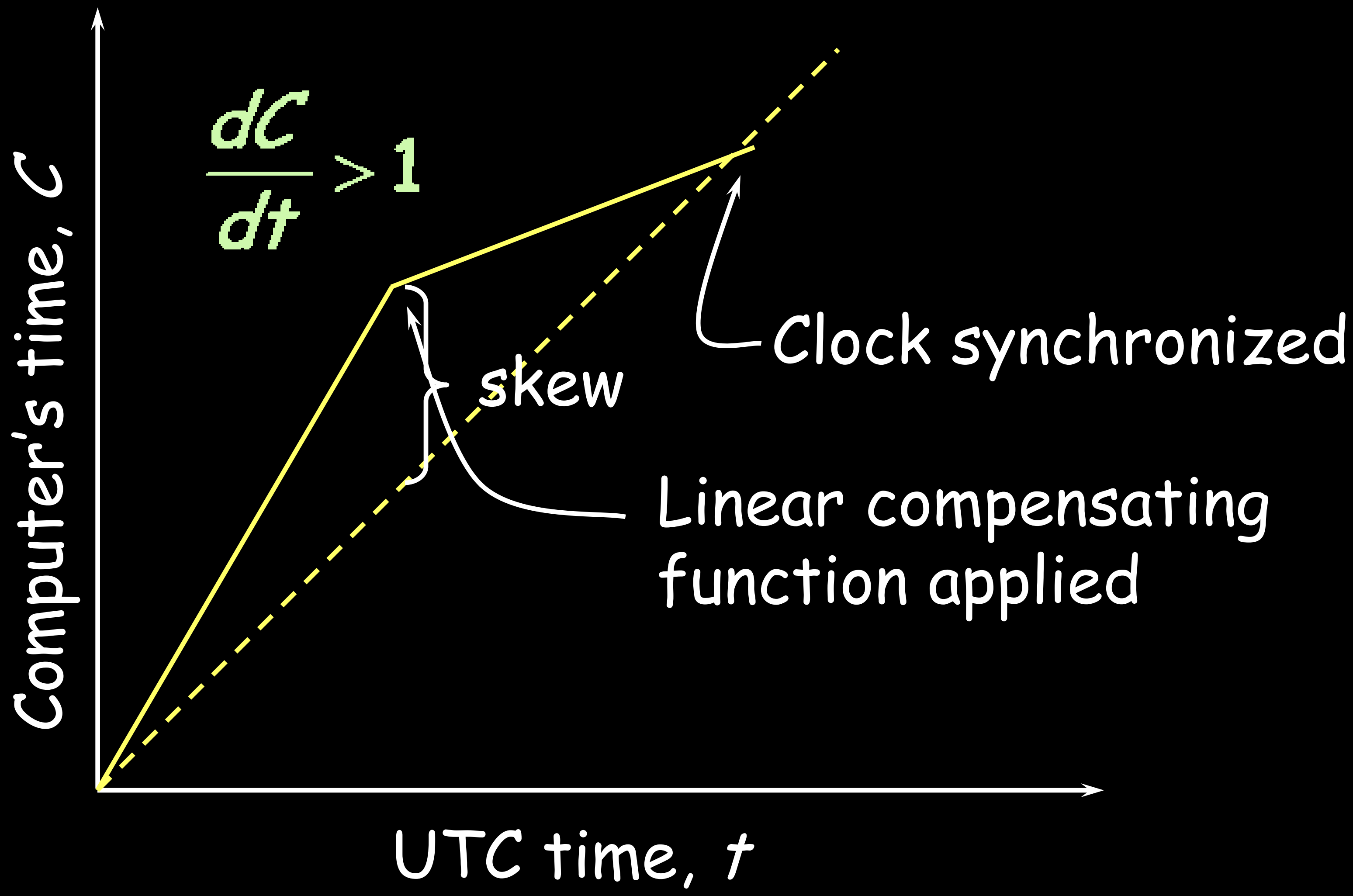
OS can do this:

- Change rate at which it requests interrupts
  - e.g.:
    - if system requests interrupts every 17 msec but clock is too slow:
      - request interrupts at (e.g.) 15 msec
- Or software correction: redefine the interval

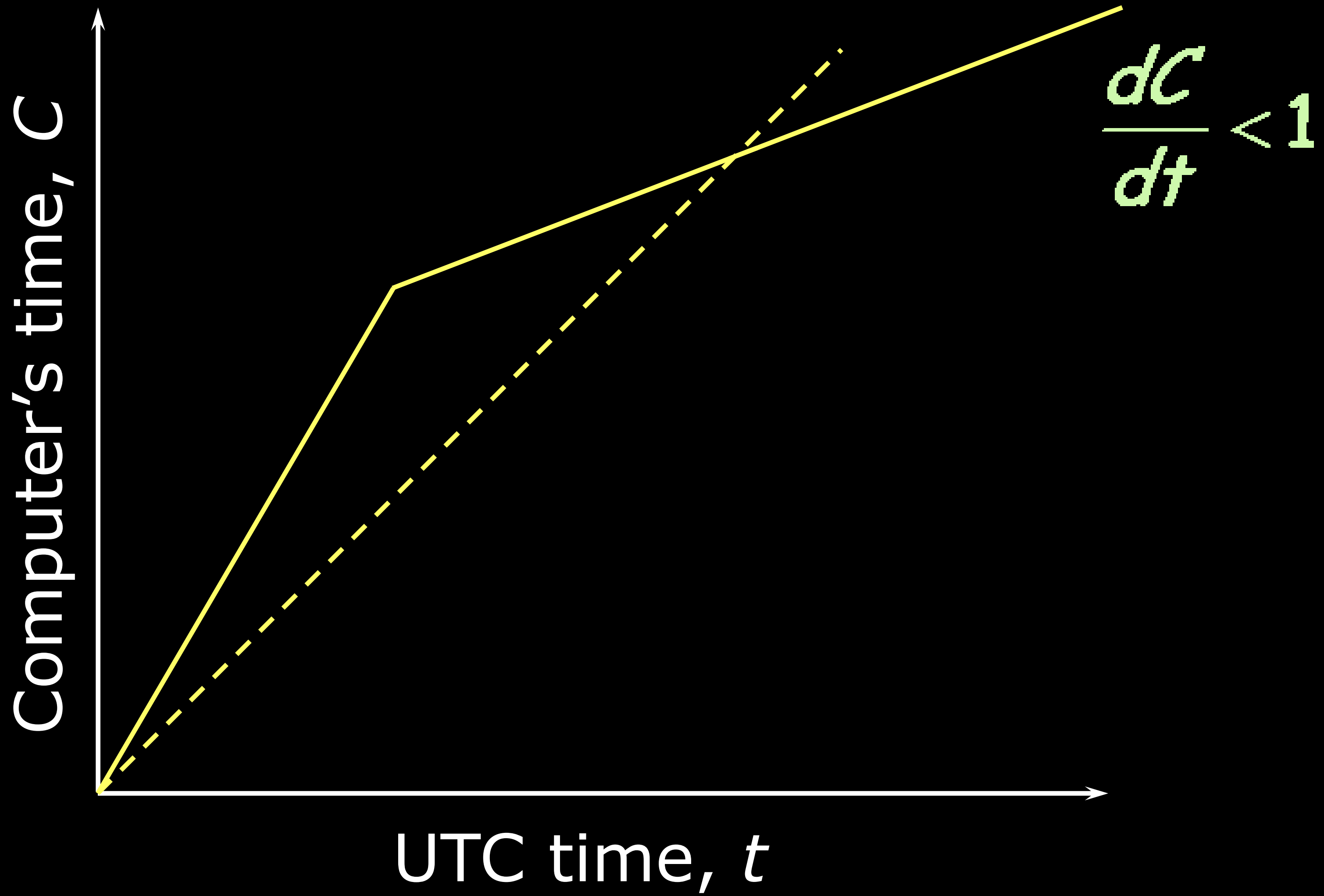
Adjustment changes slope of system time:

- **Linear compensating function**

Compensating for a fast clock



Compensating for a fast clock



Resynchronizing

After synchronization period is reached

- Resynchronize periodically
- Successive application of a second linear compensating function can bring us closer to true slope

Keep track of adjustments and apply continuously

- e.g., UNIX *adjtime* system call

Getting accurate time

- Attach GPS receiver to each computer

  ± 1 msec of UTC

- Attach WWV radio receiver

  Obtain time broadcasts from Boulder or DC

  ± 3 msec of UTC (depending on distance)

- Attach GOES receiver

  ± 0.1 msec of UTC

Not practical solution for every machine

- Cost, size, convenience, environment

Getting accurate time

Synchronize from another machine

- One with a more accurate clock

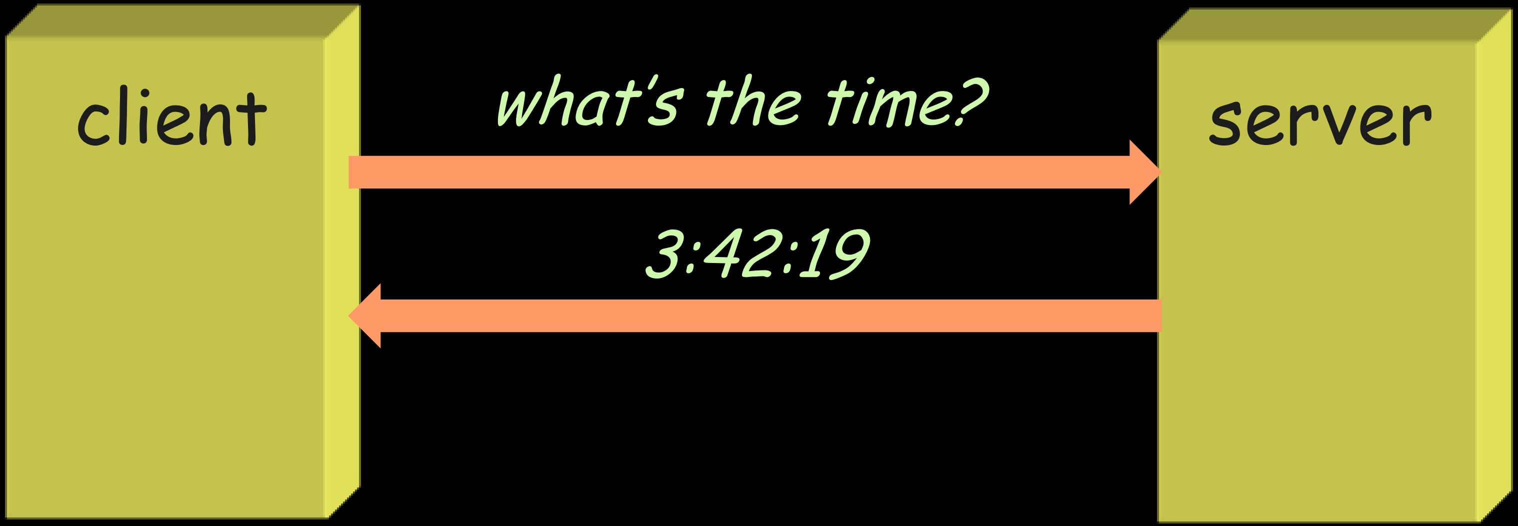
Machine/service that provides time information:

Time server

RPC

Simplest synchronization technique

- Issue RPC to obtain time
- Set time

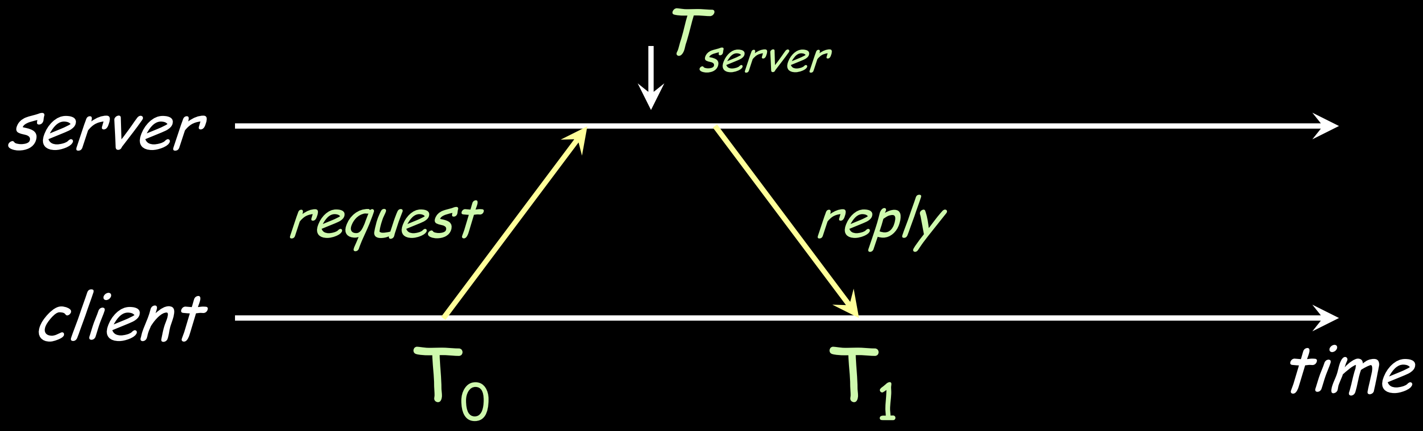


Does not account for network or processing latency

Cristian's algorithm

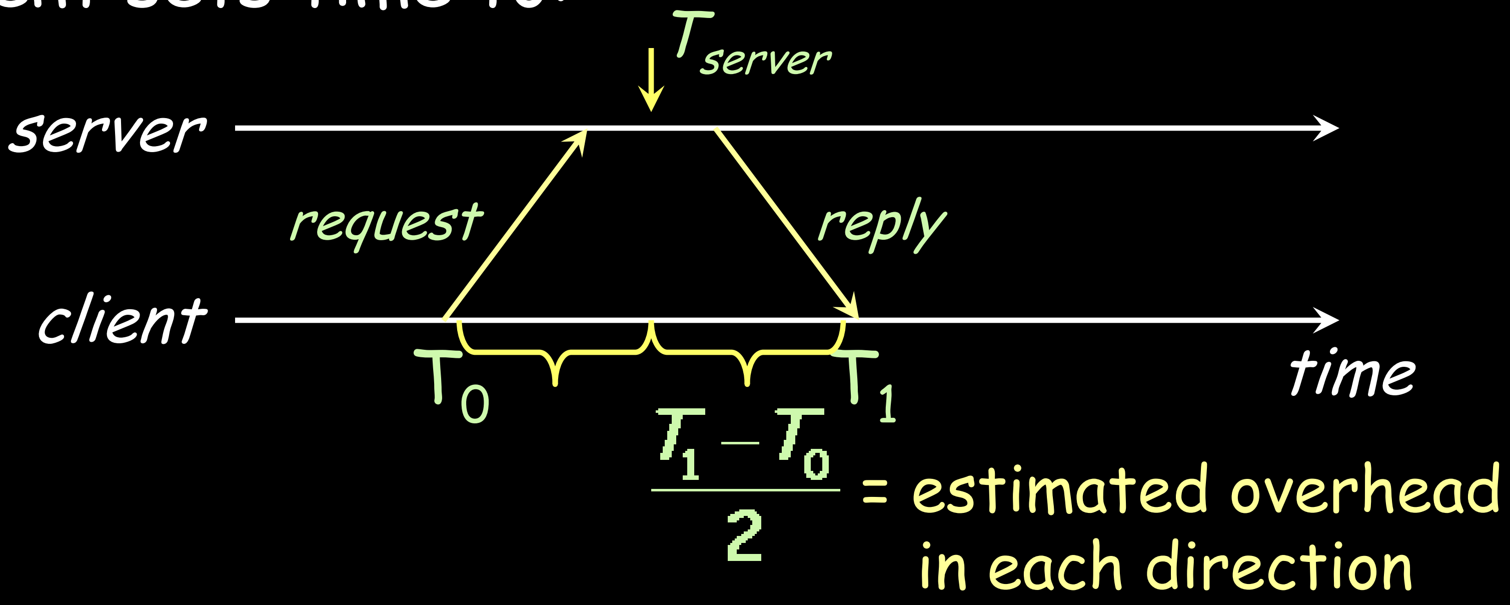
Compensate for delays

- Note times:
  - request sent: T_0
  - reply received: T_1
- Assume network delays are symmetric



Cristian's algorithm

Client sets time to:



$$\frac{T_1 - T_0}{2}$$ = estimated overhead in each direction

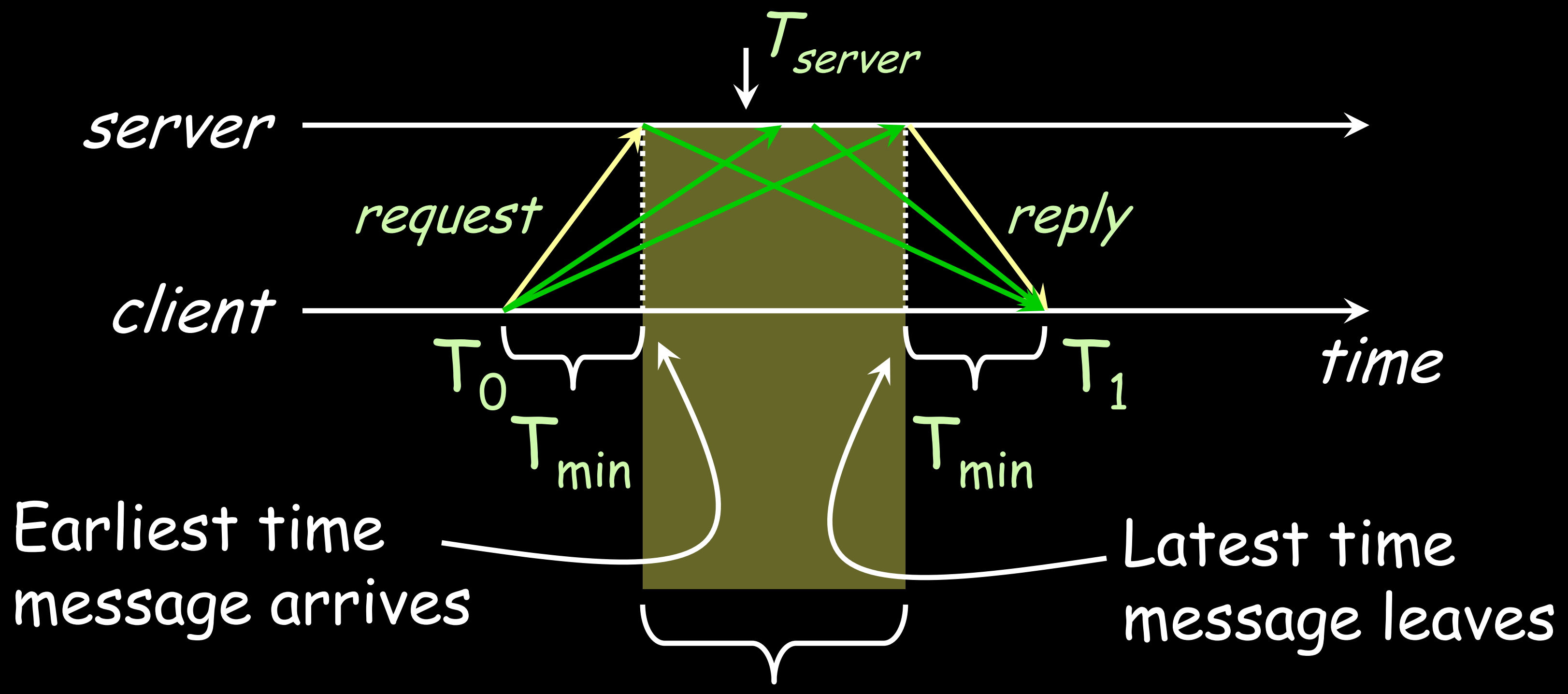
$$\boxed{T_{new} = T_{server} + \frac{T_1 - T_0}{2}}$$

Error bounds

If minimum message transit time (T_{min}) is known:

Place bounds on accuracy of result

Error bounds



range = $T_1 - T_0 - 2T_{min}$

$$\boxed{\textbf{accuracy of result} = \pm \frac{T_1 - T_0}{2} - T_{min}}$$

Cristian's algorithm: example

- Send request at 5:08:15.100 (T_0)
- Receive response at 5:08:15.900 (T_1)
  - Response contains 5:09:25.300 (T_{server})

- Elapsed time is $T_1 - T_0$

  5:08:15.900 - 5:08:15.100 = 800 msec
- Best guess: timestamp was generated

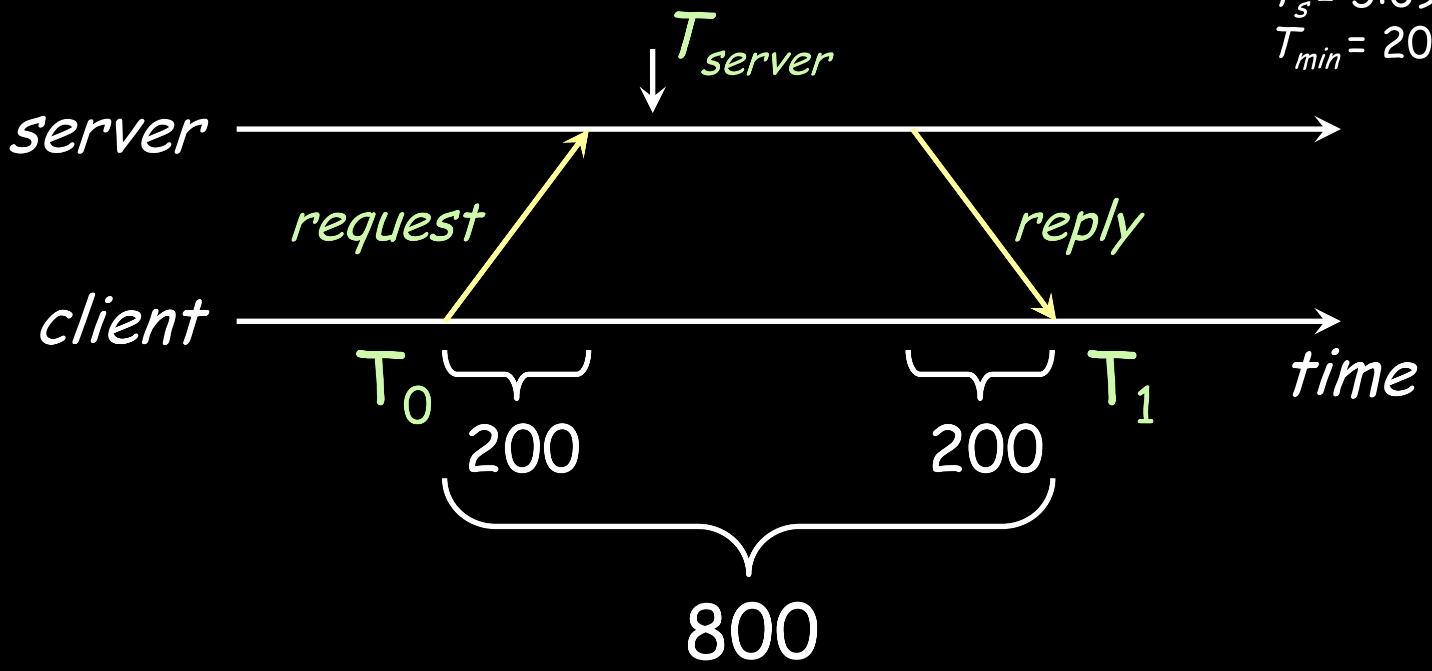
  400 msec ago
- Set time to T_{server}+ *elapsed time*

  5:09:25.300 + 400 = 5:09.25.700

Cristian's algorithm: example

If best-case message time=200 msec

T_0= 5:08:15.100
T_1= 5:08:15.900
T_s= 5:09:25:300
T_{min}= 200msec



$$\text{Error} = \pm\frac{900-100}{2} - 200 = \pm\frac{800}{2} - 200 = \pm 200$$

Berkeley Algorithm

- Gusella & Zatti, 1989

- Assumes no machine has an accurate time source
- Obtains average from participating computers
- Synchronizes all clocks to average

Berkeley Algorithm

- Machines run **time dæmon**
  - Process that implements protocol
- One machine is elected (or designated) as the server (**master**)
  - Others are **slaves**

Berkeley Algorithm

- Master polls each machine periodically
  - Ask each machine for time
    - Can use Cristian's algorithm to compensate for network latency
- When results are in, compute average
  - Including master's time
- *Hope: average cancels out individual clock's tendencies to run fast or slow*
- Send offset by which each clock needs adjustment to each slave
  - Avoids problems with network delays if we send a time stamp

Berkeley Algorithm

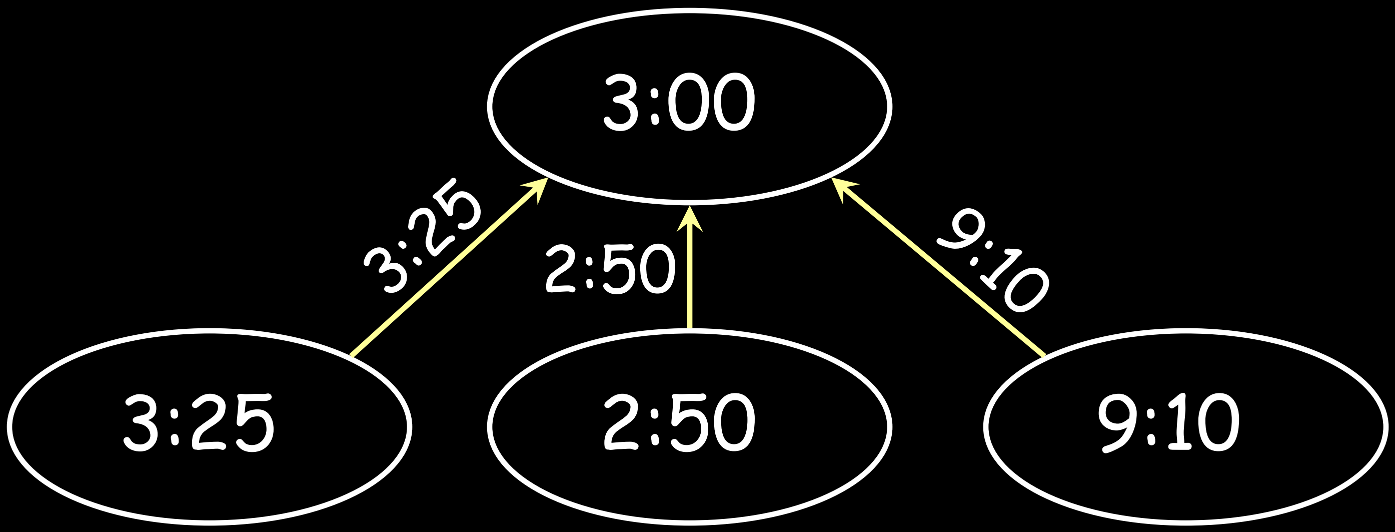
Algorithm has provisions for ignoring readings from clocks whose skew is too great

- Compute a **fault-tolerant average**

If master fails

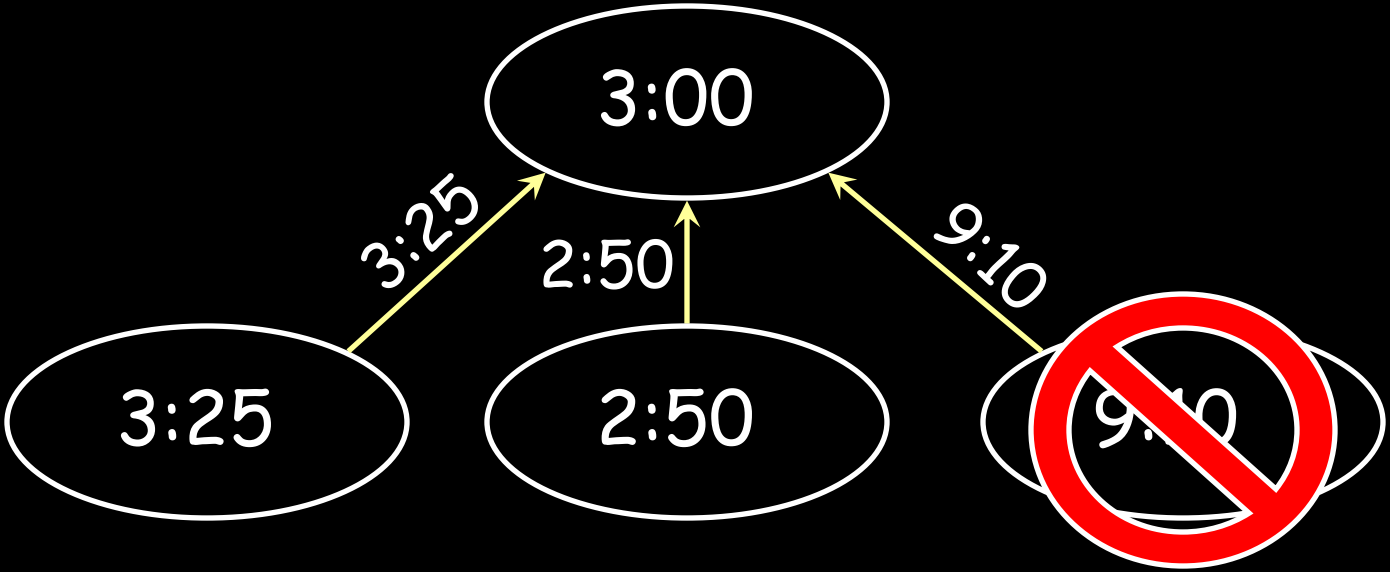
- Any slave can take over

Berkeley Algorithm: example



1. Request timestamps from all slaves

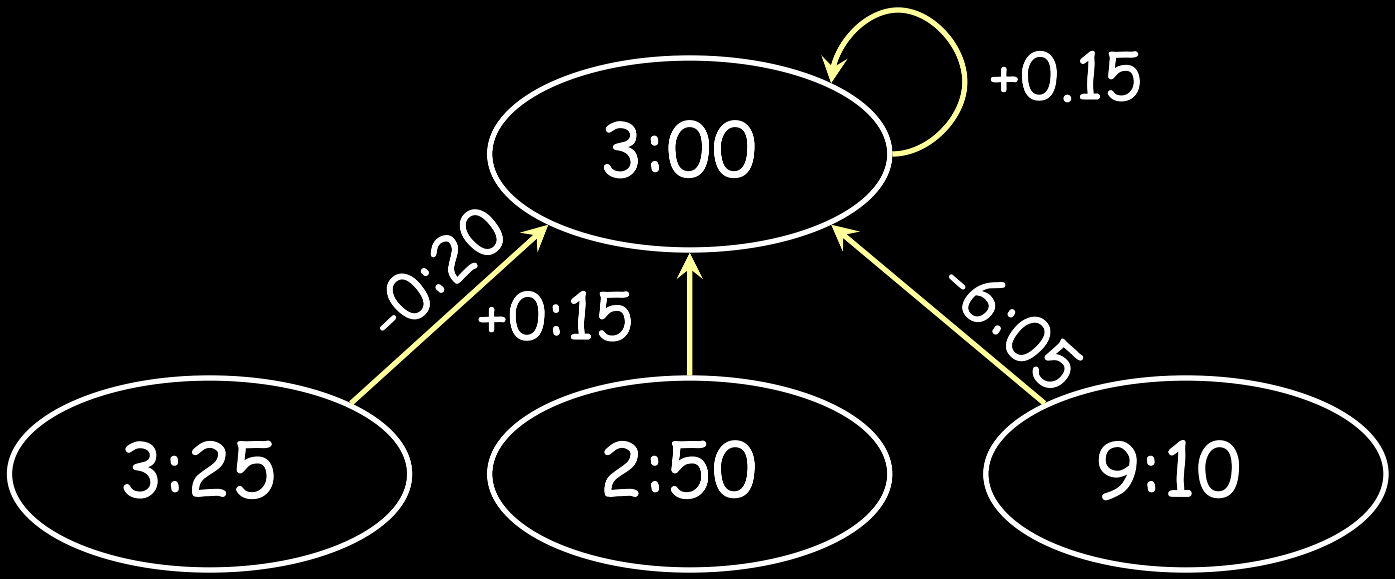
Berkeley Algorithm: example



2. Compute fault-tolerant average:

$$\frac{3{:}25 + 2{:}50 + 3{:}00}{3} = 3{:}05$$

Berkeley Algorithm: example



3. Send offset to each client

Network Time Protocol, NTP

1991, 1992

Internet Standard, version 3: RFC 1305

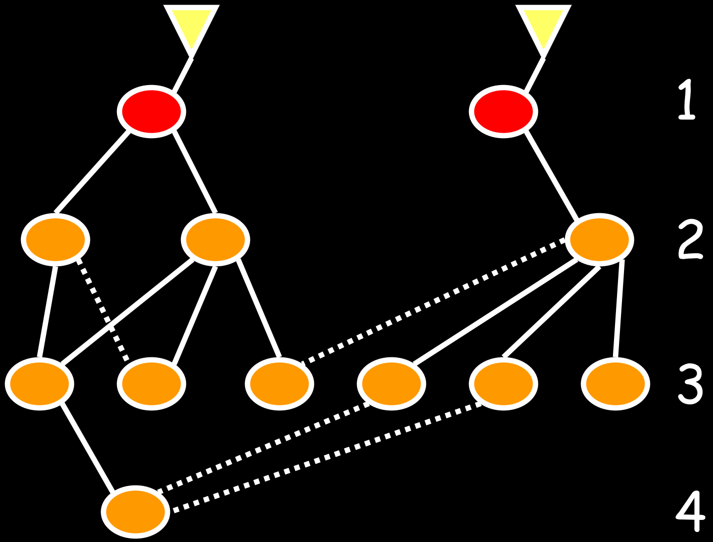
NTP Goals

- Enable clients across Internet to be accurately synchronized to UTC despite message delays
  - Use statistical techniques to filter data and gauge quality of results
- Provide reliable service
  - Survive lengthy losses of connectivity
  - Redundant paths
  - Redundant servers
- Enable clients to synchronize frequently
  - offset effects of clock drift
- Provide protection against interference
  - Authenticate source of data

NTP servers

Arranged in strata

- 1st stratum: machines connected directly to accurate time source
- 2nd stratum: machines synchronized from 1st stratum machines
- …



SYNCHRONIZATION SUBNET

NTP Synchronization Modes

Multicast mode

- for high speed LANS
- Lower accuracy but efficient

Procedure call mode

- Similar to Cristian's algorithm

Symmetric mode

- Intended for master servers
- Pair of servers exchange messages and retain data to improve synchronization over time

All messages delivered unreliably with UDP

NTP messages

- Procedure call and symmetric mode
  - Messages exchanged in pairs
- NTP calculates:
  - **Offset** for each pair of messages
    - Estimate of offset between two clocks
  - **Delay**
    - Transmit time between two messages
  - **Filter Dispersion**
    - Estimate of error – quality of results
    - Based on accuracy of server's clock ***and*** consistency of network transit time
- Use this data to find preferred server:
  - *lower stratum & lowest total dispersion*

NTP message structure

- Leap second indicator
  - Last minute has 59, 60, 61 seconds
- Version number
- Mode (symmetric, unicast, broadcast)
- Stratum (1=primary reference, 2-15)
- Poll interval
  - Maximum interval between 2 successive messages, nearest power of 2
- Precision of local clock
  - Nearest power of 2

NTP message structure

- Root delay
  - Total roundtrip delay to primary source
  - (16 bits seconds, 16 bits decimal)
- Root dispersion
  - Nominal error relative to primary source
- Reference clock ID
  - Atomic, NIST dial-up, radio, LORAN-C navigation system, GOES, GPS, …
- Reference timestamp
  - Time at which clock was last set (64 bit)
- Authenticator (key ID, digest)
  - Signature (ignored in SNTP)

NTP message structure

- T_1: originate timestamp
  - Time request departed client (client's time)
- T_2: receive timestamp
  - Time request arrived at server (server's time)
- T_3: transmit timestamp
  - Time request left server (server's time)

NTP's validation tests

- Timestamp provided ≠ last timestamp received
  - duplicate message?
- Originating timestamp in message consistent with sent data
  - Messages arriving in order?
- Timestamp within range?
- Originating and received timestamps ≠ 0?
- Authentication disabled? Else authenticate
- Peer clock is synchronized?
- Don't sync with clock of higher stratum #
- Reasonable data for delay & dispersion

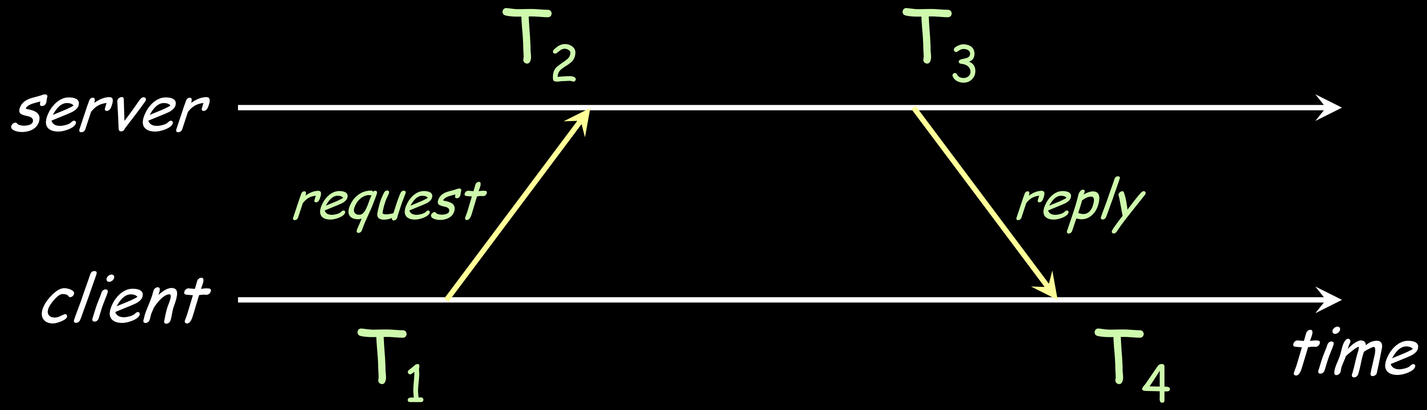
SNTP

Simple Network Time Protocol

- Based on Unicast mode of NTP
- Subset of NTP, not new protocol
- Operates in multicast or procedure call mode
- Recommended for environments where server is root node and client is leaf of synchronization subnet
- Root delay, root dispersion, reference timestamp ignored

RFC 2030, October 1996

SNTP



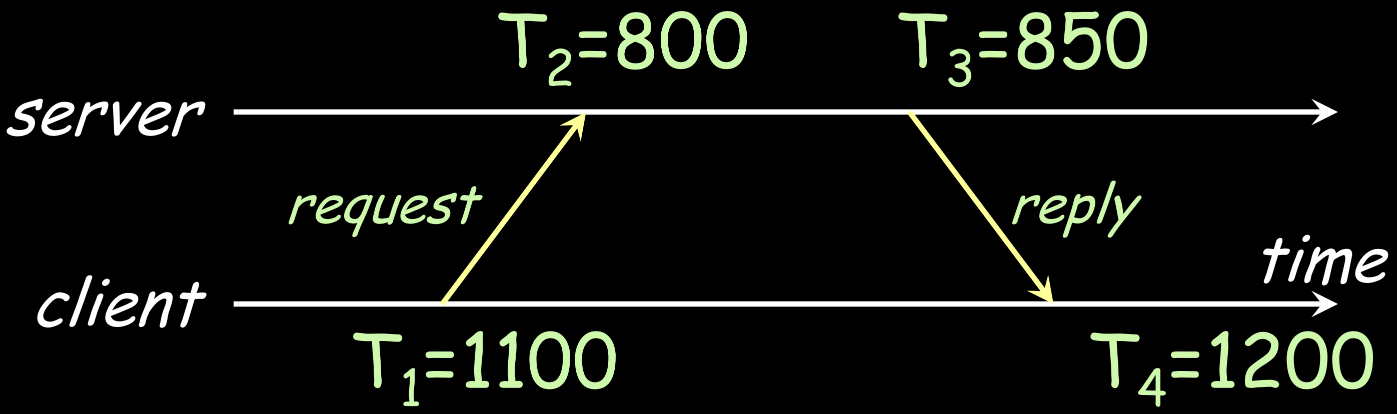
Roundtrip delay:

$$d = (T_4 - T_1) - (T_2 - T_3)$$

Time offset:

$$t = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

SNTP example



Offset =

((800 - 1100) + (850 - 1200))/2

=((-300) + (-350))/2

= -650/2 = **-325**

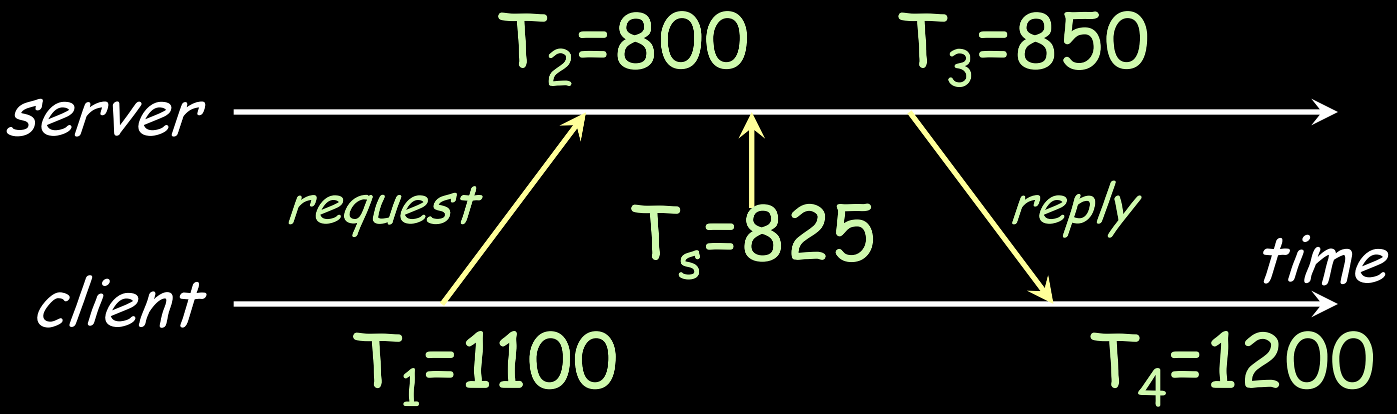
Time offset:

$$t = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

Set time to T_4 + t

= 1200 - 325 = **875**

Cristian's algorithm



Offset = (1200 - 1100)/2 = **50**

Set time to *T_s + offset*
= 825 + 50 = **875**

Key Points: Physical Clocks

- Cristian's algorithm & SNTP
  - Set clock from server
  - But account for network delays
  - Error: uncertainty due to network/processor latency: errors are additive
    ±10 msec and ±20 msec = ±30 msec.
- Adjust for local clock skew
  - Linear compensating function

The end.