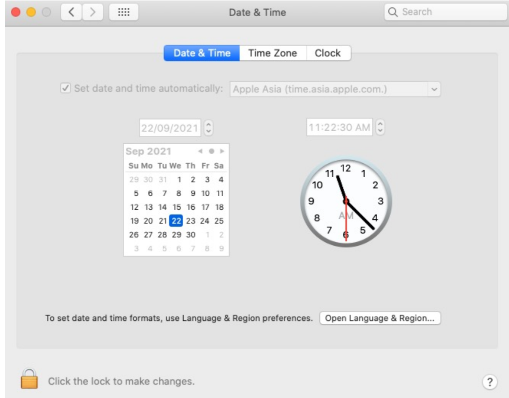


CS 582: Distributed Systems

Time in Distributed Systems

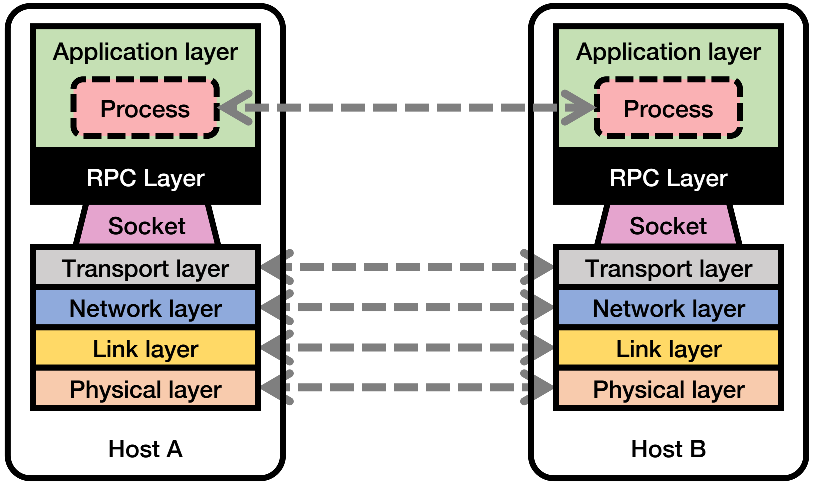
Dr. Zafar Ayyub Qazi

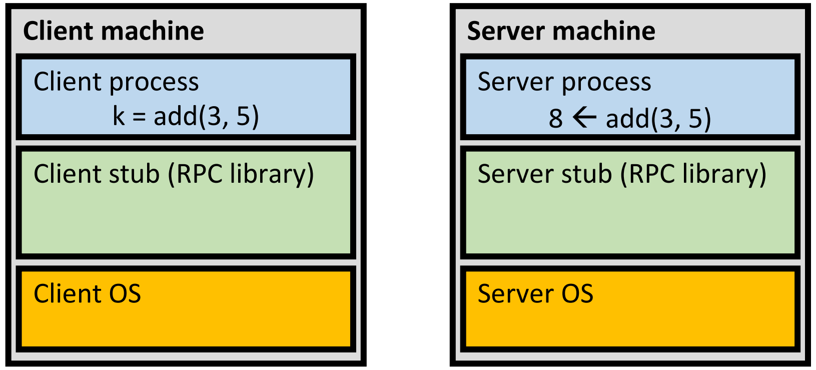
Fall 2024

Acknowledgements: Slides use some material from Indranil Gupta (UIUC), Radhika Mittal (UIUC), and Martin Kleppmann (University of Cambridge)

Recap: RPCs

- RPCs key building block, used commonly
- Make network comm. appear like a local procedure call
- Issues surrounding machine heterogeneity
- Subtle issues around failures
  - Need retransmissions to deal with failures
  - At-most-once w/ duplicate filtering
    - Discard server state w/ cumulative acks





Recap: RPC Semantics

- Exactly-Once
  - Remote procedure will be executed exactly once on the server
- At-Least-Once
  - Remote procedure may execute more than once
  - Possible Side effects
- At-Most-Once:
  - Remote procedure will not execute more than once

Go's net/rpc is at-most-once

- Opens a TCP connection and writes the request
  - TCP may retransmit but server's TCP receiver will filter out duplicates internally, with sequence numbers
  - No retry in Go RPC code (i.e., will not create a second TCP connection)
- However: Go RPC returns an error if it doesn't get a reply
  - After a TCP connection timeout
  - Perhaps server didn't see request
  - Perhaps server processed request but server/net failed before reply came back

Exactly-Once RPC semantics?

- Need retransmission
- Plus the duplicate filtering of at most once scheme
  - To survive client crashes, client also needs to make sure it has the same unique id after restart OR
  - The client should record pending RPCs on disk
    - So it can replay them with the same unique identifier
- Plus story for making server reliable
  - To survive server crashes, server should log to disk results of completed RPCs (to suppress duplicates)
  - Even if server fails, the system needs to continue with full state
    - Have replicated and consistent server-side

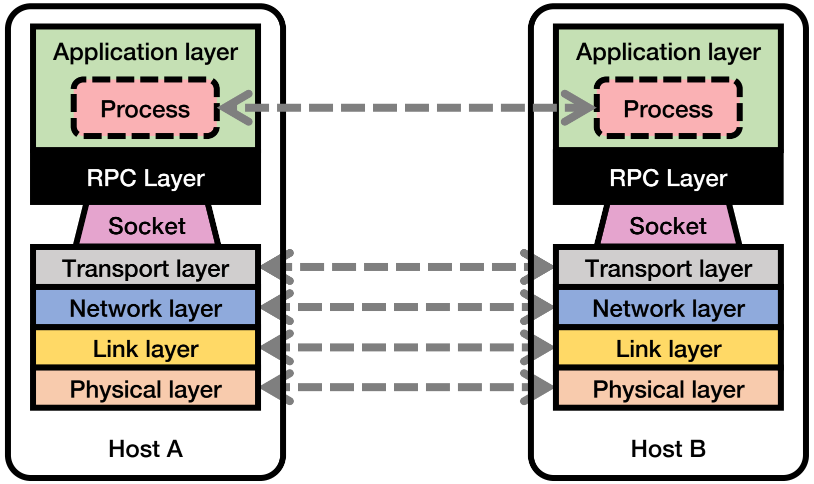
Synchronous and asynchronus RPCs

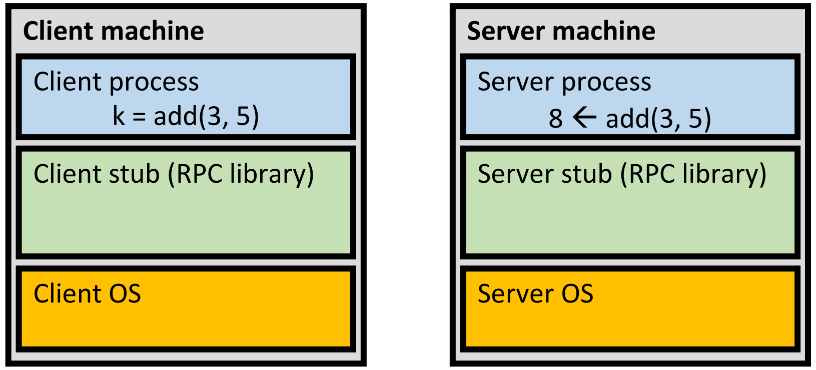
- Synchronous remote procedure call is a blocking call
  - I.e., when a client has made a request to the server, the client will wait until it receives a response from the server

- Asynchronous RPC: Client does not wait for a response
  - This is useful when the RPC call is a long-running computation on the server, meanwhile, client can continue execution.

Summary of RPCs

- RPCs key building block, used commonly
- Make network comm. appear like a local procedure call
- Issues surrounding machine heterogeneity
- Subtle issues around failures
  - Need retransmissions to deal with failures
  - At-most-once w/ duplicate filtering
    - Discard server state w/ cumulative acks
  - Exactly-once with:
    - retransmissions + at-most-once
      fault tolerance (of servers)





Questions

Concept Check

- Consider an RPC system that provides At-Least-Once semantics. Evaluate the safety of using such an RPC system in the following scenarios:
  1. An RPC server receiving and executing multiple times the same read operation: get(key1)
  2. An RPC server receiving and executing multiple times the same write operation: put(key2)
  3. An RPC server receiving and executing multiple times an increment operation, x = x++

Rest of the lecture

- Notion of time in distributed systems
- Need for time synchronization
- Cristian's algorithm
- Berkeley algorithm

Learning Outcomes

By the end of today's lecture, you should be able to:

- ❑ Explain how time is measured on computers
- ❑ Discuss the necessity of time synchronization in distributed systems
- ❑ Analyze the challenges associated with time synchronization in distributed environments
- ❑ Calculate and interpret clock skew and clock drift
- ❑ Explain how time synchronization can be done in a synchronous network
- ❑ Explain Cristian's algorithm and apply it to solve time synchronization problems
- ❑ Evaluate the accuracy of Cristian's algorithm in different scenarios
- ❑ Explain Berkeley algorithm and apply it to solve time synchronization problems

Clocks and time in Distributed Systems

- Distributed systems often need to measure time, e.g.,
  - Failure detector timeouts, retry timers, for scheduling tasks
  - Log files & databases: record when an event occurred
  - Performance measurements, statistics, profiling
  - Data with time-limited validity (e.g., cache entries)
  - **Determining order of events across several nodes**

- We distinguish two types of clock:
  - **Physical clocks:** count number of seconds elapsed
  - **Logical clocks:** count events, e.g., messages sent

Physical clocks in computers

- Nearly all computers have a circuit for keeping track of time
- Typically based on vibrating quartz crystals
  - These vibrating quartz crystals can oscillate at well-defined frequencies
  - Associated with a crystal are two registers, a counter and holding register
  - Each oscillation decrements the counter by one. When the counter gets to zero, an interrupt is generated.
  - It is possible to generate a certain number of interrupts every second
- These quartz clocks are cheap but not totally accurate
  - Due to manufacturing imperfections, some clocks run slightly faster than others
  - Moreover, the oscillation frequency varies with the temperature

Clock skew and clock drift rate

- Clock skew: the relative difference between two clock values
    - Like the distance between two vehicles on a road
- Clock drift rate: change in skew from a reference clock per unit time (measured by the reference clock)
    - Like the difference in speed of two vehicles on the road

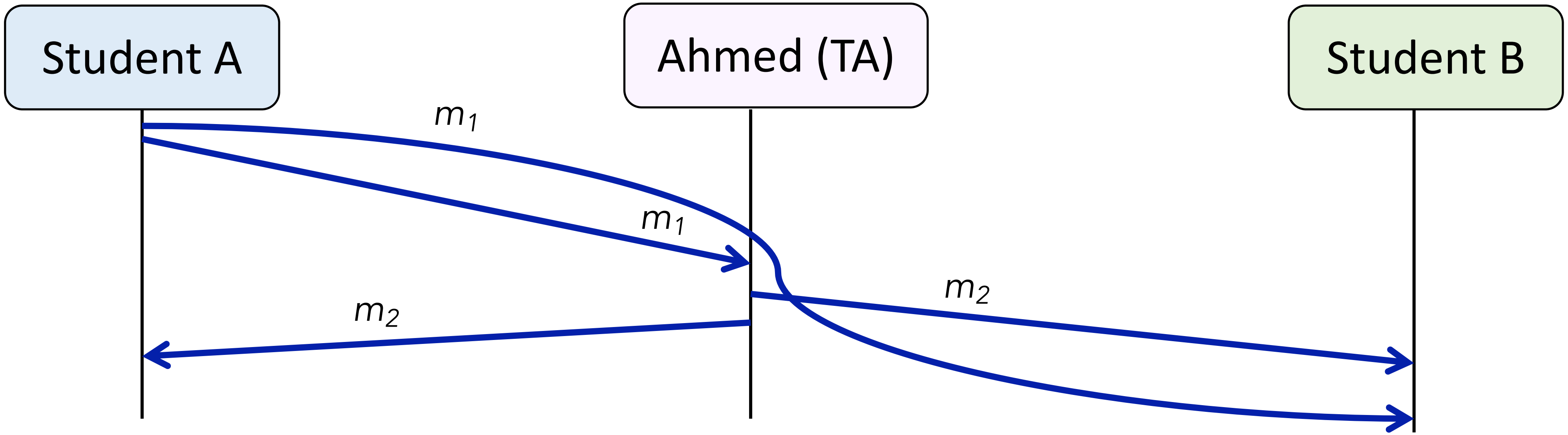
Ordinary and authoritative clocks

- Ordinary quartz crystal clocks (cheap, laptops/desktops/phones):
  - Drift rate is about 10^{-6} seconds/second
  - Drift by 1 second every 11.6 days.
  - Skew of about 30 minutes after 60 years.
- High-precision atomic clocks (expensive ~$25K)
  - Drift rate is about 10^{-13} seconds/second
  - Skew of about 0.18ms after 60 years
  - Used as standard for real time
  - Universal Coordinated Time (UTC)* obtained from such clocks

* Refer to Chapter 6.1 to understand how UTC is measured

Why time synchronization?

Example: Slack like Application



m_1: "Do we have a quiz on Mon?"
m_2: "No"

Student B sees m_2 first, m_1 second, even though logically m_1 happened before m_2

Ordering events

More generally, use timestamps to order events in a distributed system

- Requires the system clocks to be synchronized with one another

Why challenging?

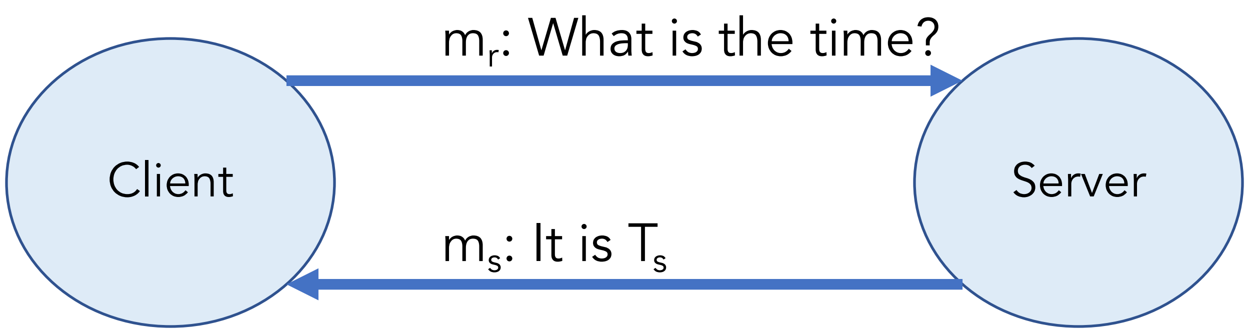
- Computers track physical time with internal clocks
    - These clocks can disagree – can drift apart over time

- Solution: Periodically get the current time from a server that has a more accurate time source (e.g., atomic clock or GPS receiver)

- Challenges
    - Message/processing delays vary and may be unbounded

Synchronization in synchronous systems

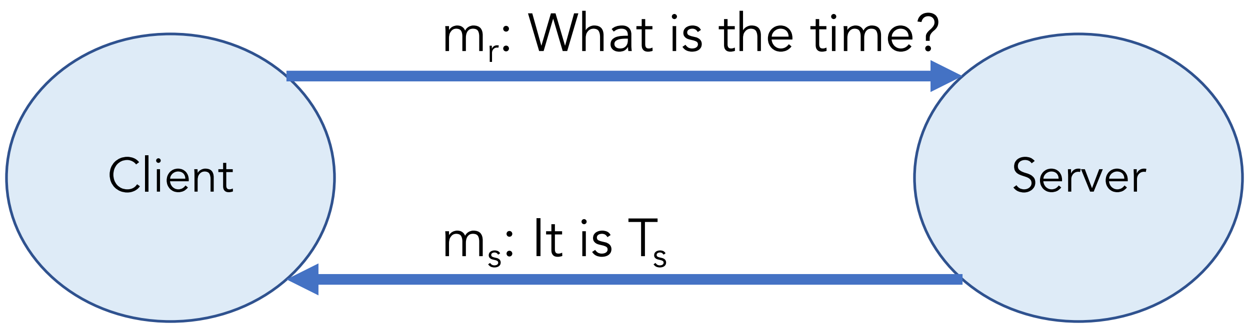
Synchronization in synchronous systems



Let max and min be the maximum and minimum one-way network delay

Can we guarantee perfect time synchronization?

Synchronization in synchronous systems



Ignore the processing delays at the client/server

What time T_c should client adjust its local clock to after receiving m_s?

Let max and min be the maximum and minimum one-way network delay

How should we set the value of T_c s.t. we have the lowest upper bound for skew?

Synchronization in asynchronous systems

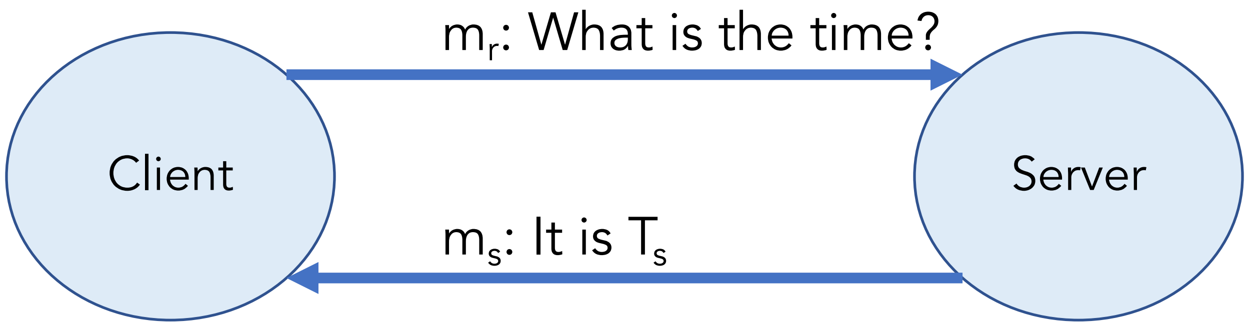
- Cristian's Algorithm
- Berkeley Algorithm
- Network Time Protocol

Challenge in asynchronous systems

- Latencies can be unabounded in an asynchronous system

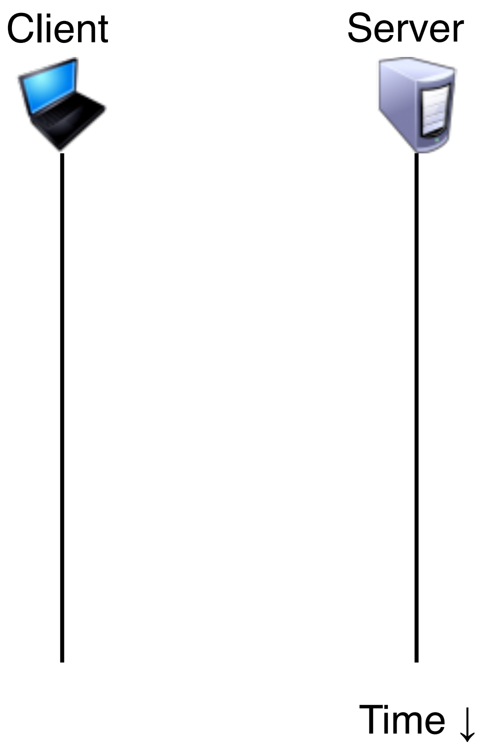
Cristian's algorithm

Cristian's Algorithm



- Client measures the round trip time (T_{round})
- $T_c = T_s + (T_{round} / 2)$

Cristian's Algorithm



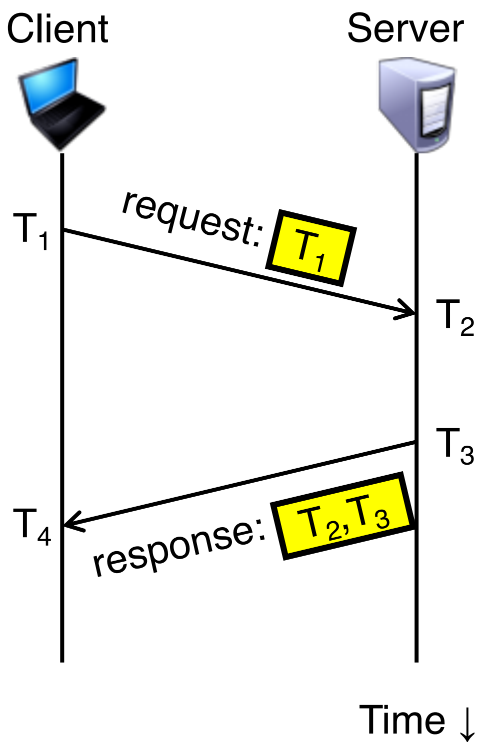
Cristian's Algorithm

1. Client sends a request packet, timestamped with its local clock T_1

2. Server timestamps its receipt of the request T_2 with its local clock

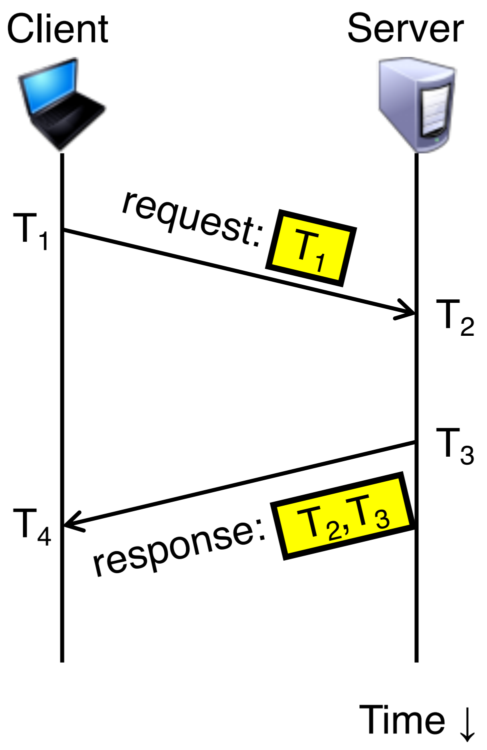
3. Server sends a response packet with its local clock T_3 and T_2

4. Client locally timestamps its receipt of the server's response T_4



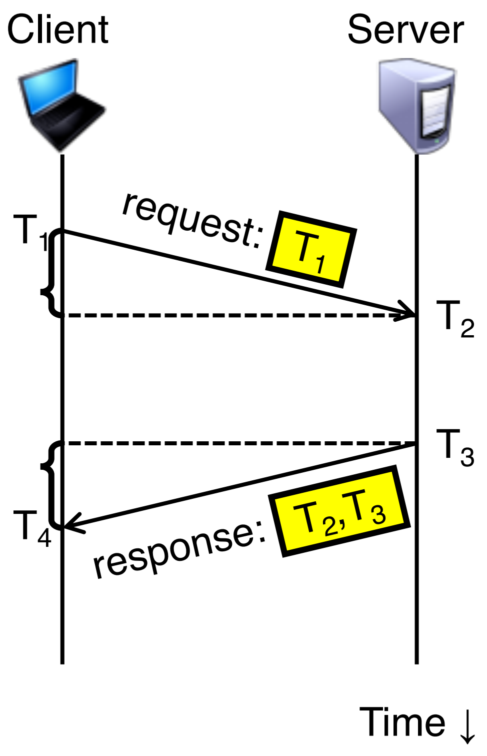
Cristian's Algorithm: Considerations

- We can't find the exact one-way network delay from the server to the client
- Hence, assume network delays are symmetric
  - i.e., the network delay experienced by the request message = network delay of response message



Cristian's Algorithm: Offset calculation

- Round-trip network delay, $\alpha = (T_4 - T_1) - (T_3 - T_2)$
- Offset = $\alpha/2$
- Estimated server time when client receives response = $T_3 +$ Offset
- $T_c = T_3 +$ Offset



Skew(client, server)?

More on Cristian's Algorithm ...

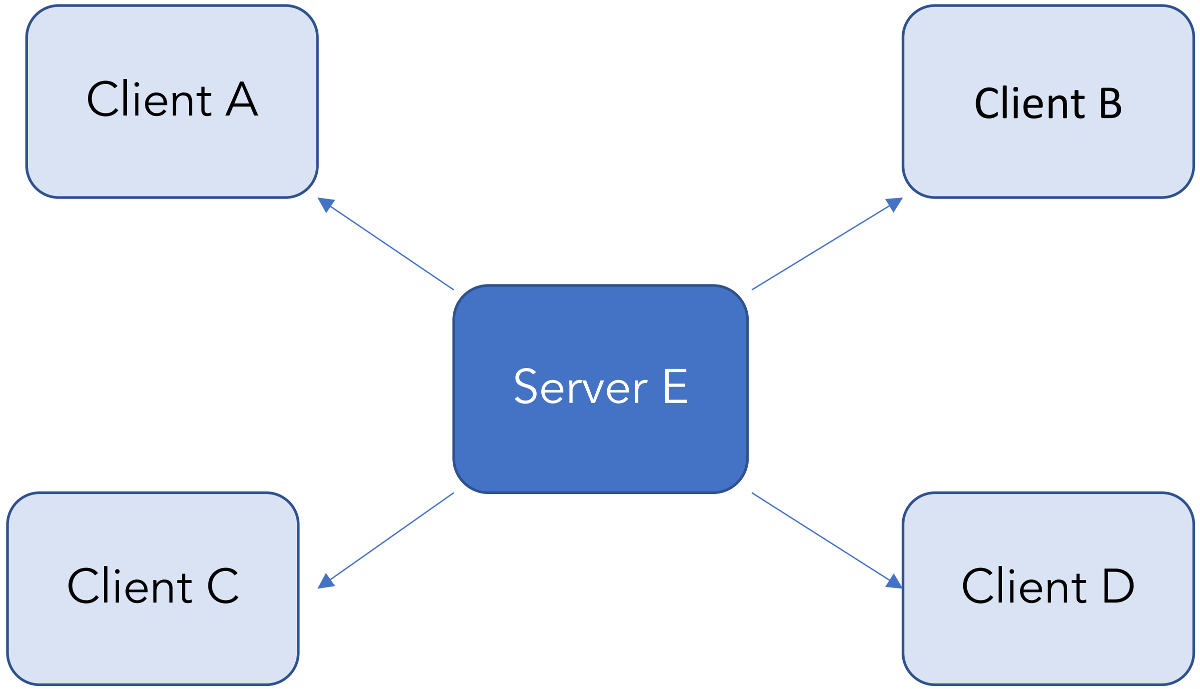
- How can you improve accuracy (reduce max. skew)?
- Cannot handle faulty time servers

Berkeley Algorithm

Berkeley Algorithm

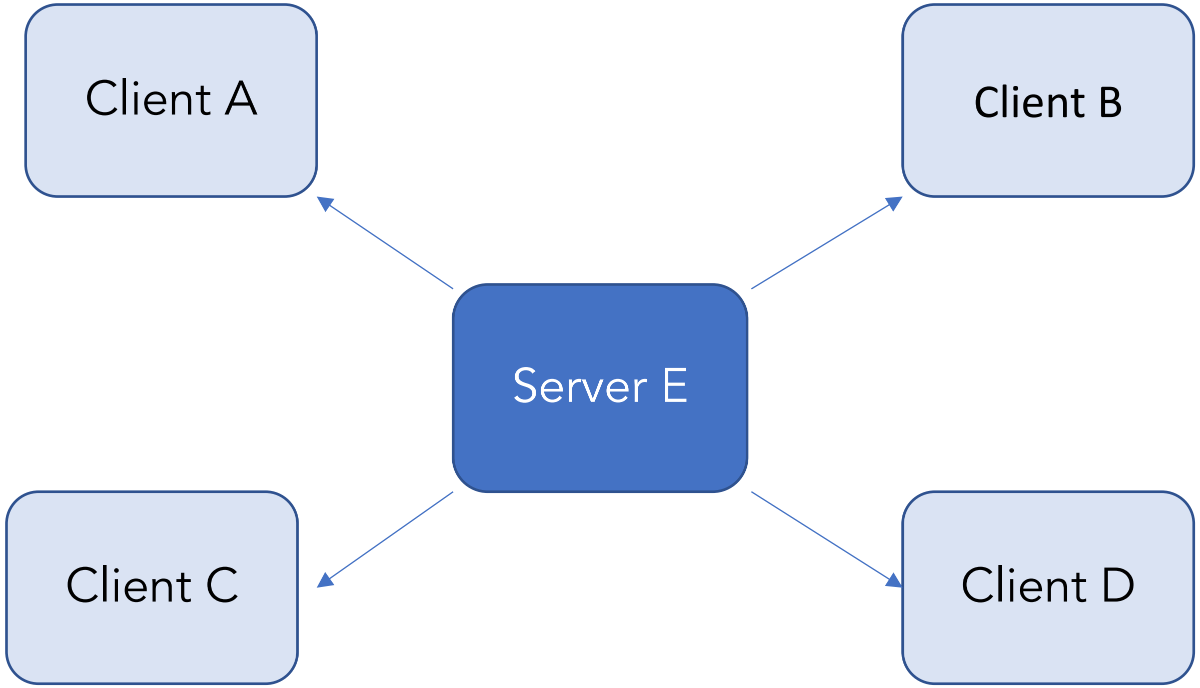
- Assumes no machine has an accurate time source
- Obtains average from participating computers
- Synchronizes all clocks to average

Berkeley Algorithm

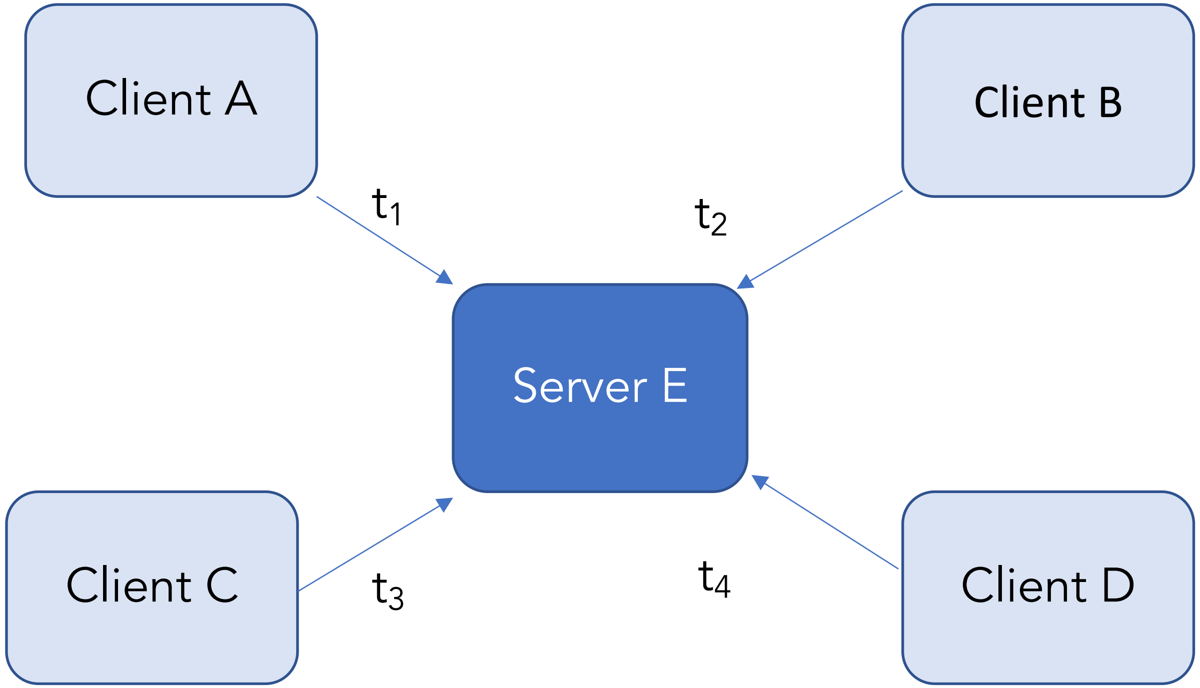


Berkeley Algorithm

1. Server polls clients about their time

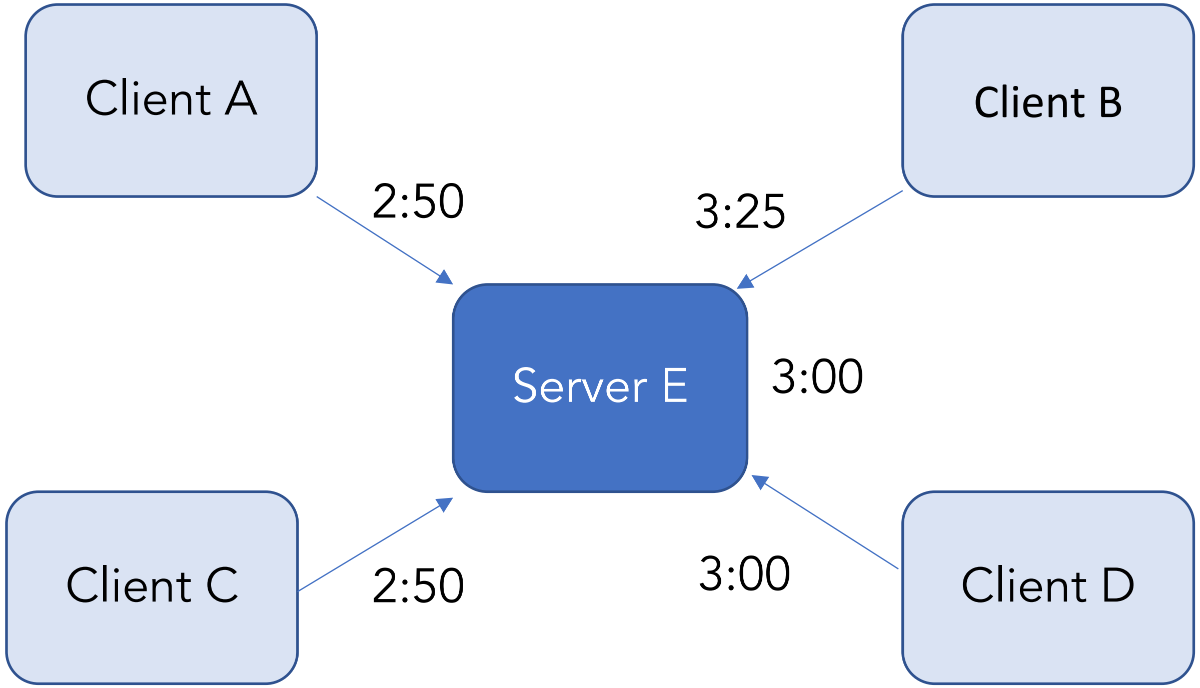


Berkeley Algorithm



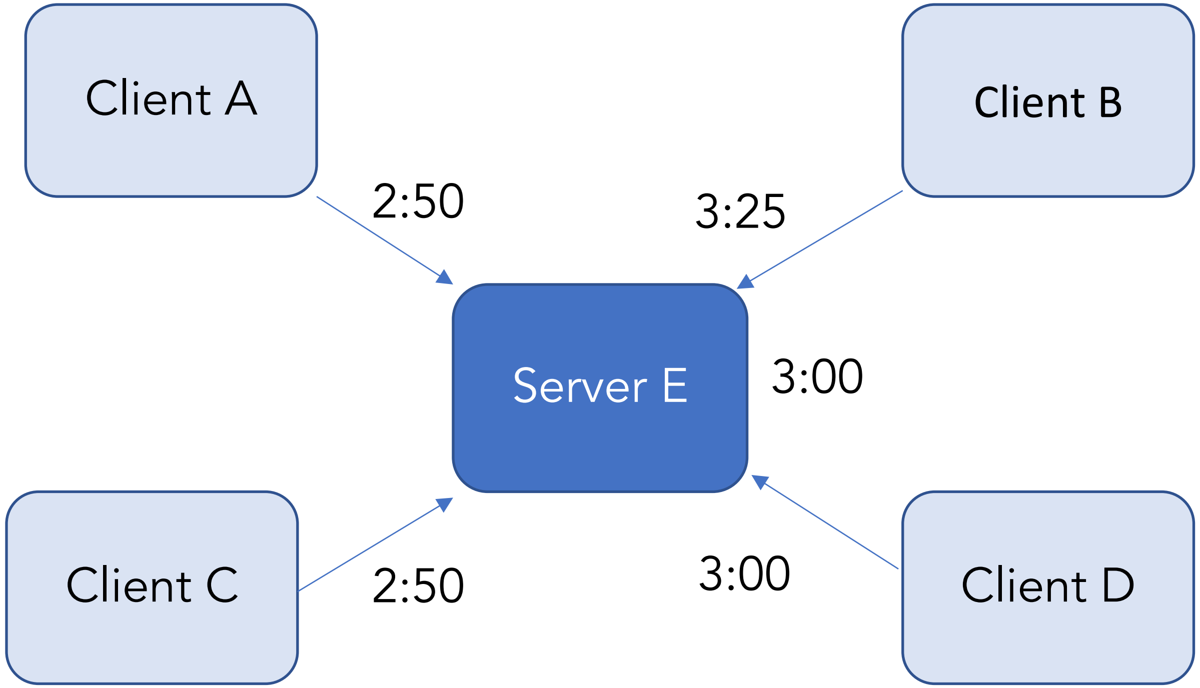
1. Server polls clients about their time
2. Clients reply back with their time

Berkeley Algorithm



1. Server polls clients about their time
2. Clients reply back with their time
3. Server uses Cristian's algorithm to estimate time at each client

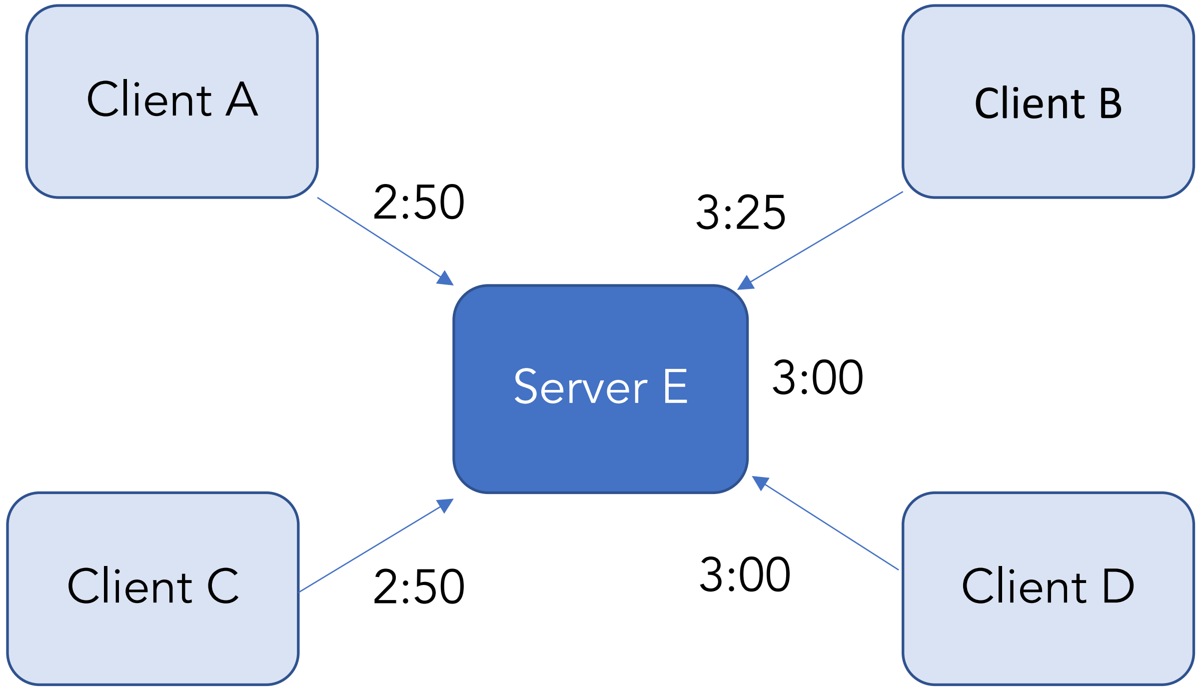
Berkeley Algorithm



1. Server polls clients about their time
2. Clients reply back with their time
3. Server uses Cristian's algorithm to estimate time at each client
4. Average all local times (including its own) – use as updated time

Updated time at server: ?

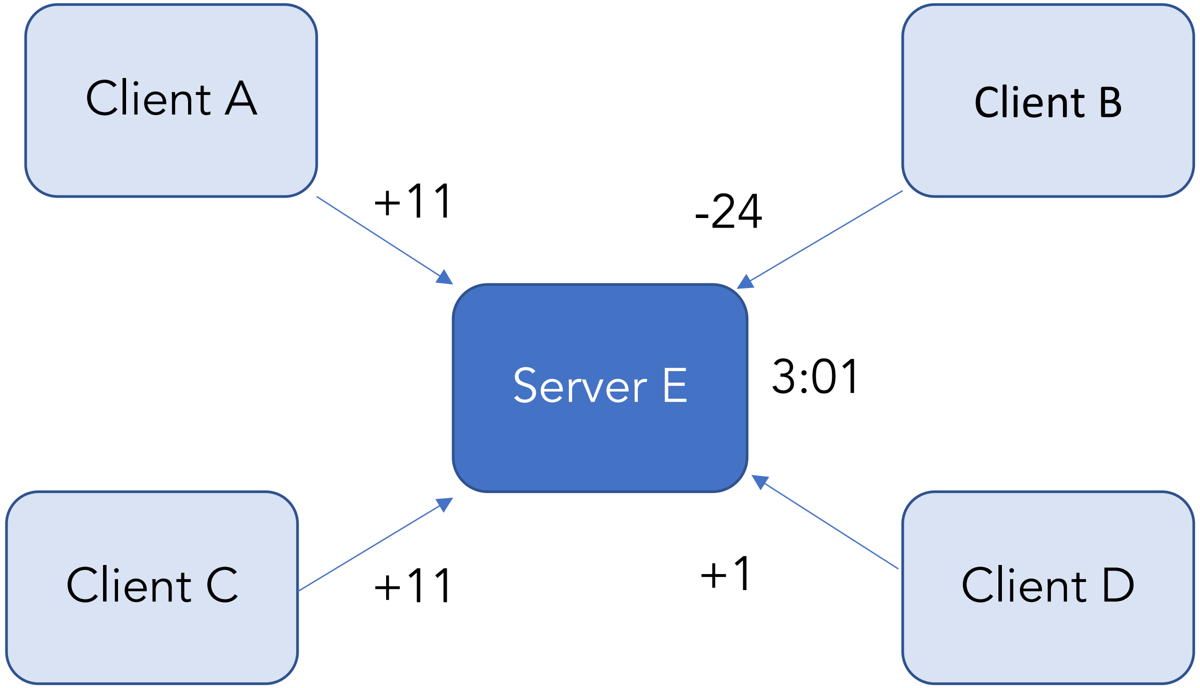
Berkeley Algorithm



1. Server polls clients about their time
2. Clients reply back with their time
3. Server uses Cristian's algorithm to estimate time at each client
4. Average all local times (including its own) – use as updated time

Updated time at server: 3:01

Berkeley Algorithm



1. Server polls clients about their time

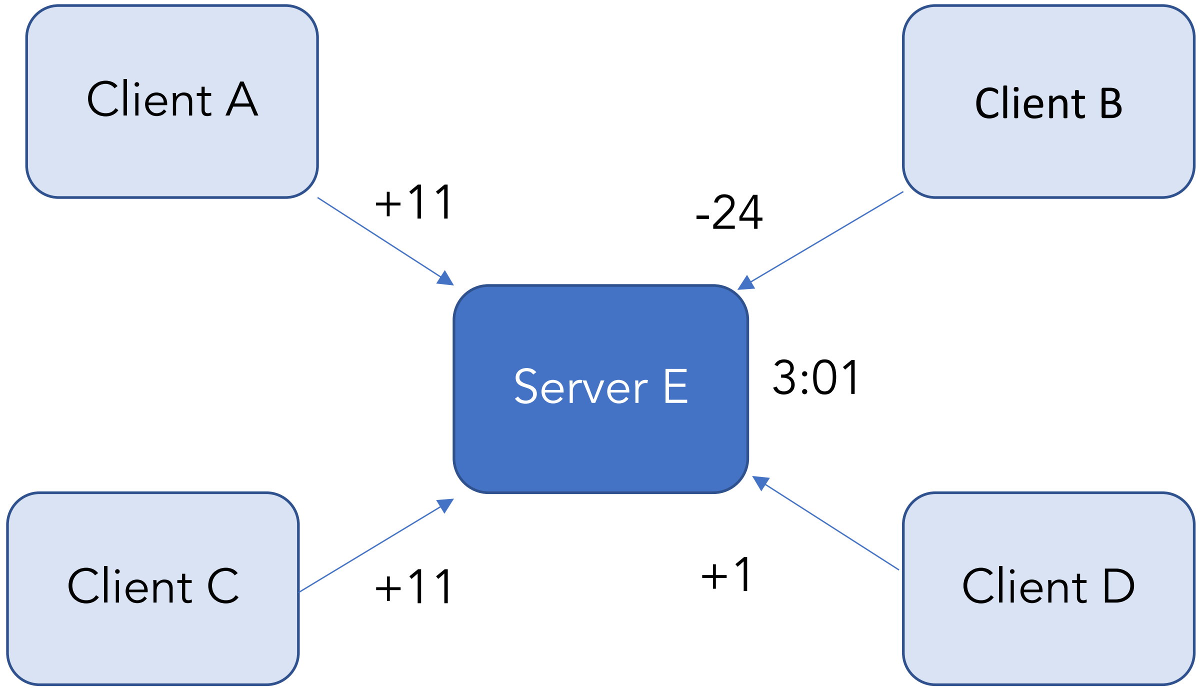
2. Clients reply back with their time

3. Server uses Cristian's algorithm to estimate time at each client

4. Average all local times (including its own) – use as updated time
Updated time at server: 3:01

5. *Send the offset (amount by which each clock needs adjustment).*

Berkeley Algorithm



1. Server polls clients about their time

2. Clients reply back with their time

3. Server uses Cristian's algorithm to estimate time at each client

4. Average all local times (including its own) – use as updated time
Updated time at server: 3:01

5. Send the offset (amount by which each clock needs adjustment).

All clients directly apply the offsets to theirs clocks

Implementing negative offsets

- If we decrease clock value
  - May violate ordering of events within the same process

- Should try to decrease speed of clock but not decrease clock value

Berkeley algorithm: dealing with faulty clocks

- Selects the largest set of clocks that do not differ from each other more than a threshold, α, and averages the differences of these clocks

- Prevents malfunctioning clocks and clocks with abnormally large drift rates from adversely affecting the synchronization process

Summary: Physical Time

- Perfect physical time synchronization is impossible to guarantee
- Cristian's algorithm: to synchronize time with a time server
  - Skew < RTT/2
- Berkeley algorithm: to synchronize time between a group of servers
  - Use Cristian's algorithm to estimate time at each client
  - Average all local times
  - Send offsets
  - Deal carefully with negative offsets

Next Lecture

- Network Time Protocol (NTP)