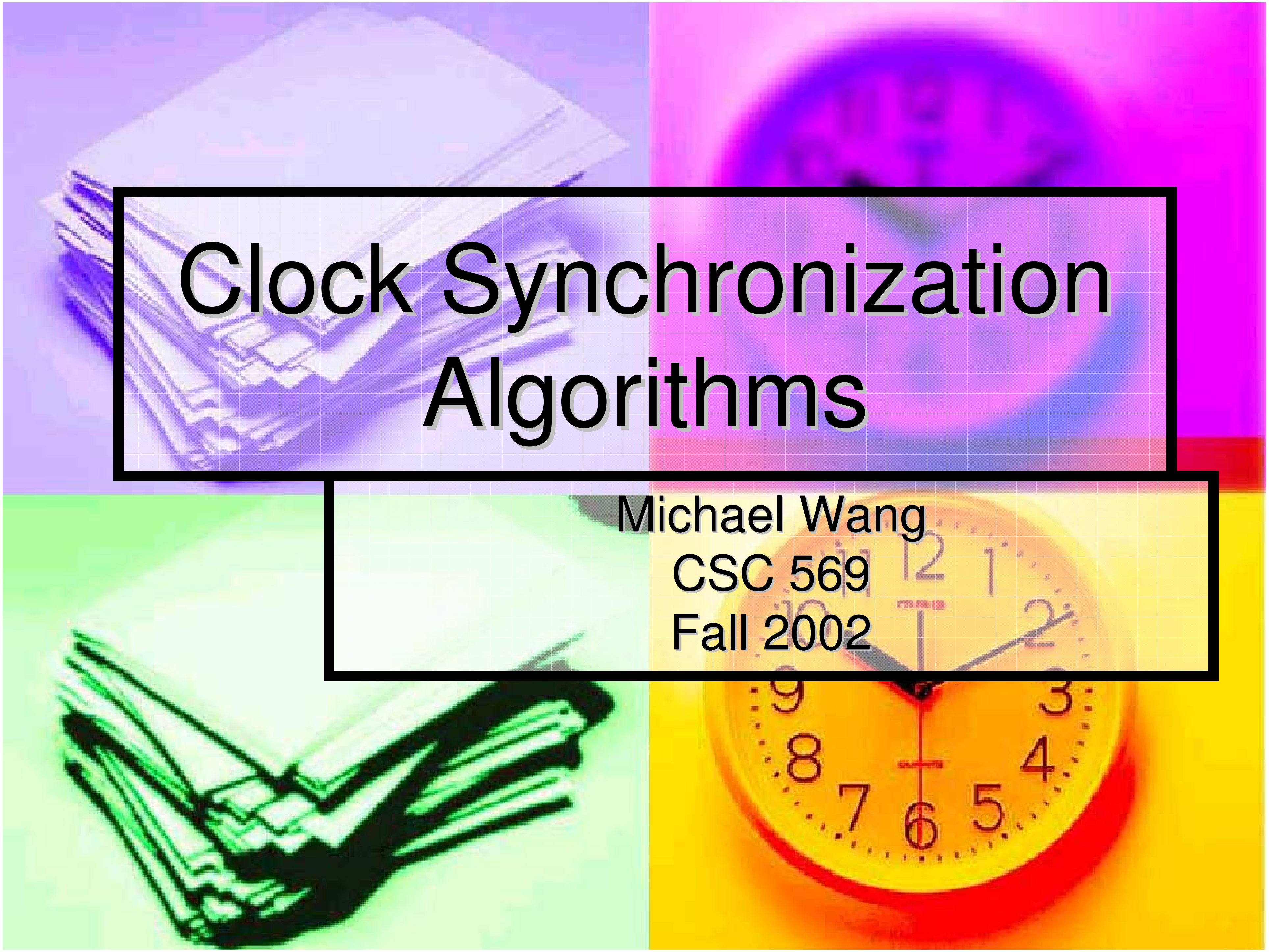

Clock Synchronization Algorithms

Michael Wang
CSC 569
Fall 2002

Presentation Outline

- Background
- Deterministic Algorithms
- Probabilistic Algorithms
- New Implementation
- References

Why Clock Synchronization?

- Independent clocks
- Different times
- Drift
- Logical clocks don’t work
- Need real time

Applications of Clock Synchronizations

- Email servers
- Measure duration of two events on two nodes
- Buying and selling stocks

Problems

- Different times
- Different speeds
- Delay in communication
- Faulty clocks

Example of Not Being Synchronized

- Surveyed 5,722 hosts and gateways
- 60% were off by > 1 minute
- 10% were off by > 13 minutes
- A few were off by > 2 years

Types of Synchronization

Internal

- Processor clocks are close to each other
- For measuring duration within system
- Not externally synchronized

External

- Processor clocks are close to real time
- For real-time systems
- Also internally synchronized

Problem Definition

- n number of clocks
- $C_i(t)$ – reading of a clock C_i at real time t
- $c_i(T)$ – real time when the i-th clock reaches time T
- Drift rate bound: $|d(C_i/dt) - 1| < \rho$
- Sync bound: $|C_i(t) - C_j(t)| <= \beta$
- Clock can only change small amount at each resynchronization

Problem Definition

- Basic requirements
  - S1. At any time the value of all the nonfaultly processors' clocks must be approximately equal. That is

$$|C_i(t) - C_j(t)| <= \beta$$

  - S2. There is a small bound ∑ on the amount by which a nonfaultly processor's clock is changed during each resynchronization.

Types of Algorithms

Deterministic

- Require assumptions about message delays
- Synchronization and bounds are guaranteed

Probabilistic

- No assumptions about message delays
- Guarantees precision with probability

Deterministic Clock Synchronization Assumptions

- *max* = maximum message delay
- *min* = minimum message delay
- *n* = # of clocks
- Closeness of synchronization
  = (*max* – *min*)(1 – 1 / *n*)

Welch-Lynch Algorithm Assumptions

- # of faulty clocks < n/3, or n=3f+1
- Hardware clock is never changed:
  $C(t) = H(t) + CORR(t)$
- Initially all clocks are synchronized:
  $|c_i(T_0) – c_j(T_0)| < \beta$
- Message delay:
  $[\delta - \varepsilon, \delta + \varepsilon]$

Algorithm Overview

- Executes in rounds
- Collects arrival times of messages from other processes until its local clock reaches T_i
- Broadcast its local clock value
- Continues to receive messages until a maximum time
- Changes the CORR function
- Starts another round

Pseudo Code

```
set-timer(T0)

do forever
   // loop until timer Ti breaks
   while receive(m, k) do
        ARR[k] = NOW

   T := NOW
   broadcast(T)
   set-timer( T + (1 + ρ) * (β + δ + ε))

   // loop until timer breaks
   while receive(m, k) do
        ARR[k] = NOW

   AV := mid( reduce(ARR) )
   ADJ := T + δ - AV
   CORR := CORR + ADJ
   set-timer( T + ΔT)
enddo
```

Jalote, P. *Fault Tolerance in Distributed Systems*, p. 95, 1994.

Wait Time

set-timer(T + (1 + ρ) * (β + δ + ε))

- T = now
- β = bound on times on other processes
- δ + ε = maximum message delay
- ρ = local drift rate
- If a message is not received from a process within this time, then that process is faulty.

Changing the CORR function

AV := mid(reduce(ARR))

- Simple averaging function doesn't work because of faulty clocks
- Since there are at most n/3 faulty clocks, the reduce function gets rid of the top 1/3 and bottom 1/3 of the values.
- Take the middle value.

This Works!

- Process i sent message at T_i
- Process j received message at AV
- The average message delay is δ
- Difference between clock at i and clock at j is: $(T_i + \delta) - AV$
- $T_i == T$

Parameters

- ρ, δ, and ε are fixed
- β and ΔT can be configured
- Smaller β means clocks closer to each other
- Smaller ΔT means clocks resynchronize more often
- Also means more messages

Bounds

- $\Delta T <= \beta / 4\rho - \varepsilon / \rho - \rho(\beta + \delta + \varepsilon)$
  $- 2\beta - \delta - 2\varepsilon$
- If ΔT is fixed: $\beta \approx 4\varepsilon + 4\rho\Delta T$
- $ADJ <= (1 + \rho)(\beta + \varepsilon) + \rho\delta$

Optimization to the Welch-Lynch Algorithm

```
while receive(m, k) do
        ARR[k] = NOW
AV := mid( reduce(ARR) )
```

- The reduce function needs to sort the array, and the cut off the top 1/3 and bottom 1/3 of the array.
- Wastes at least log n number of computations

Optimization to the Welch-Lynch Algorithm

```
for i := 0; receive(m, k); i := i + 1
        ARR[i] = NOW
AV := ARR[i / 2]
```

- No need to sort
- No need to waste time

New Pseudo Code

```
set-timer(T0)

do forever
    // loop until timer breaks
    for i := 0; receive(m, k); i := i + 1
          ARR[i] = NOW

    T := NOW
    broadcast(T)
    set-timer( T + (1 + ρ) * (β + δ + ε))

    // loop until timer breaks
    for i := i; receive(m, k); i := i + 1
          ARR[i] = NOW

    AV := ARR[i / 2]
    ADJ := T + δ - AV
    CORR := CORR + ADJ
    set-timer( T + ΔT)
enddo
```

Probabilistic Clock Synchronization Algorithms

- Does not guarantee synchronization
- Uses probability
- Does not require assumptions about message delay
- Can achieve closer synchronization between clocks

Cristian's Algorithm Overview

- Server-client model
- Server is synchronized with external real time
- Clients query server for time to synchronize.

Cristian's Algorithm Overview

- *min* = minimum time to prepare, transmit, and receive a message
- When a process j, wants to know the clock at another process i, it sends a query to i.
- Process i replies to the query.
- Process j times the round trip delay.
- Process j uses round trip delay, *min*, and clock value reported by i to synchronize.

Server Pseudo Code

```
// run forever because it's a daemon
do forever

    // receive a query from process k
    receive(q, k)

    // reply k with the local current time
    send(NOW, k)

enddo
```

Client Pseudo Code

```
// send query to process s (server)
send(q, s)

// get start time
T1 := NOW

// wait for process s to reply
receive(T, s)

// get end time
T2 := NOW

// set new time
D := (T2 – T1) / 2
C := T + D(1 + 2ρ) – min*ρ
```

This Works!

- At receiving reply, the minimum real time >= $T + \min(1 - \rho)$
- The maximum real time <= $T + 2D(1 + 2\rho) - \min(1 + \rho)$ (Demonstrate)
- Average is $T + D(1 + 2\rho) - \min * \rho$

How is it Probabilistic?

- If a process wants a specific precision ε, then it must discard all messages with round-trip delay greater than 2U:
  $U = (1 - 2\rho)(\varepsilon + min)$
- It is probabilistic because it can keep requesting but never get one that has round-trip delay less than or equal to 2U.

Work-around

- Get the minimum of several tries
- Get the average of several tries

Performance

- Minimum error is 3 * ρ * min
- 2 messages for each synchronization
- 2n messages for n processes

Welch-Lynch vs. Cristian

- Peer-to-peer
- Need assumptions
- Max error = $4\varepsilon + 4\rho\Delta T$
- Guarantees
- n^2 messages

- Server-client
- No need for assumptions
- Min error = $3 * \rho * min$
- Probability
- 2n messages

References

- Jalote, P., *Fault Tolerance in Distributed Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1994.
- Mills, D.L., “On the Accuracy and Stability of Clock Synchronized by the Network Time Protocol in the Internet System”.
- Cristian, F., H. Aghili, R. Strong, “*Clock Synchronization in the Presence of Omission and Performance Failures, and Processor Joins*”, 1986.
- Dutertre, B., “*The Welch-Lynch Clock Synchronization Algorithm*”, 1998.