Simple-minded Clock Sync

ICMP Protocol (types 13-14): Send a message with your local time and a unique message id. The receiver responds with the unique id, your time, his time, and a new id. You respond with your time & his time. And continue forever.

Fields:
Originate timestamp (last time sender touched the message)
Receive timestamp (first time receiver touched the message)
Transmit timestamp (last time echoer touched it)
Identifier – like a session id
Sequence number – incremented with each time message

Clock algorithm: Estimate the network delay, using the unique id (each message must be paired with its response to get an accurate estimate of the delay).

Problems:

1) Processing time – at least at link layer
2) Dropped messages
3) Variances in delay
4) Faulty servers
5) Malicious servers

Telecommunications Synchronization

T1 lines?
DS-k lines
SONET: Start of message is determined by clock + semaphore that is used to determine clock drift.
Master clocks: hierarchy and strat

Stratum 0: atomic clocks or other highly accurate clocks
Stratum 1: computers serving time from stratum 0 devices
Stratum 2: computers getting time from stratum 1 computers
Etc.

NTP can actually bump the stratum up if the computer's time estimates are enough worse than its peers

Distributed time synchronization

Page 6:
Multiple servers/peers provide redundancy and diversity.
Clock filters select best from a window of eight time offset samples.
Intersection and clustering algorithms pick best truechimersand discard falsetickers.
Combining algorithm computes weighted average of time offsets.

Filtering

The best offset samples should occur with the lowest delays. "The clock selection algorithm determines from among all associations a suitable subset of truechimers capable of providing the most accurate and trustworthy time using principles similar to [DOL95].

Intersection and Clustering

Intersect confidence intervals – or get the smallest interval consistent with the largest number of sources

Examples:
[3,7], [6,8], [5.5,6.5] gives [6,6.5]
[3,7],[6,9.5],[9,11] could use [6,7] or [9,9.5] – the latter is smaller
Naive method: use center of interval

Synchronization

Dolev 95 - Multiple clocks all try to act as leader sending synchronous signal JACM 95

Processors have duration timers and adjustment registers. When a processor starts, its duration timer is initialized (to 0?). The adjustment registers keep track of an adjustment used to determine the processor's approximate "real time" and the goal of synchronization is to adjust it intermittently.

Assumptions:

1) A correct duration timer is a monotone increasing function of real time with bounded drift.
2) Enough links are in the "up" condition enough of the time for all processors to communicate
3) A processor can "know" who sent each message (using authentication if necessary)
4) At most f faulty processors
5) Successive synchronizations don't overlap

3 and 4 are not necessary unless processors can have Byzantine failures.

Requirements:

1) Logical clocks stay "close together"
2) Logical clocks stay within a linear envelope of the duration timers.

Algorithm 1 Idea:
Choose a time interval PER such that synchronization messages are sent by all the correct processors every PER time units. As a result, the adjustment register may be increased by a small amount.
A time value (and signatures) is sent in each synch message. It is a multiple of PER. The expected synch value is stored in ET – this jumps by PER when it changes.

The participants flood the network with the sync message, ignoring all but the first copy. The more signatures that the message has, the earlier it is allowed to be.

```
automaton Sync(PER:Nat, E:Real, process:Nat, epsilon:Real)
signature
      input receive(t:Real, signers:Set[Nat])          % MSG
      output send(t:Real, signers:Set[Nat])            % SEND

states
      DT:Real := 0;
      ET:Real := PER;
      A:Real := -DT;
      lastSigners:Set[Nat] := {};

      %C derived is DT+A

transitions
      output send(t:Real, signers:Set[Nat])
      pre
            ( (DT + A) >= ET
            /\ t = ET
            /\ signers = { process }
            ) \/
            (lastSigners ~= {}
            /\ signers = insert(process, lastSigners)
            /\ t=ET-PER
            );
      eff
            ET := ET+PER;


      input receive(t:Real, signers:Set[Nat])
      eff
            if t = ET /\ ET - size(signers) * E < DT + A then
                  lastSigners := signers;
                  A := ET - DT;
                  ET := ET + PER;
            fi;

trajectories
trajdef localclock
stop when DT+A = ET
evolve d(DT) = 1+epsilon;
```

To distinguish between different clocks, the algorithm uses assumptions about message delays and clock drift, and rejects “unreasonable” values.