Chapter 06: Coordination

Version: February 25, 2017
Physical clocks

Problem
Sometimes we simply need the exact time, not just an ordering.

Solution: Universal Coordinated Time (UTC)
- Based on the number of transitions per second of the cesium 133 atom (pretty accurate).
- At present, the real time is taken as the average of some 50 cesium clocks around the world.
- Introduces a leap second from time to time to compensate that days are getting longer.

Note
UTC is broadcast through short-wave radio and satellite. Satellites can give an accuracy of about ±0.5 ms.
Clock synchronization

**Precision**

The goal is to keep the deviation between two clocks on any two machines within a specified bound, known as the precision $\pi$:

$$\forall t, \forall p, q : |C_p(t) - C_q(t)| \leq \pi$$

with $C_p(t)$ the computed clock time of machine $p$ at UTC time $t$.

**Accuracy**

In the case of accuracy, we aim to keep the clock bound to a value $\alpha$:

$$\forall t, \forall p : |C_p(t) - t| \leq \alpha$$

**Synchronization**

- **Internal synchronization**: keep clocks precise
- **External synchronization**: keep clocks accurate
Clock specifications

- A clock comes specified with its **maximum clock drift rate** $\rho$.
- $F(t)$ denotes oscillator frequency of the hardware clock at time $t$.
- $F$ is the clock’s ideal (constant) frequency $\Rightarrow$ living up to specifications:

$$\forall t : (1 - \rho) \leq \frac{F(t)}{F} \leq (1 + \rho)$$

Observation

By using hardware interrupts we couple a software clock to the hardware clock, and thus also its clock drift rate:

$$C_p(t) = \frac{1}{F} \int_0^t F(t) dt \Rightarrow \frac{dC_p(t)}{dt} = \frac{F(t)}{F}$$

$$\Rightarrow \forall t : 1 - \rho \leq \frac{dC_p(t)}{dt} \leq 1 + \rho$$

Fast, perfect, slow clocks

Clock time, $C$

- **Fast clock**: $\frac{dC_p(t)}{dt} > 1$
- **Perfect clock**: $\frac{dC_p(t)}{dt} = 1$
- **Slow clock**: $\frac{dC_p(t)}{dt} < 1$

UTC, $t$
Detecting and adjusting incorrect times

Getting the current time from a time server

Computing the relative offset $\theta$ and delay $\delta$

**Assumption:** $\delta T_{req} = T_2 - T_1 \approx T_4 - T_3 = \delta T_{res}$

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

$$\delta = \frac{(T_4 - T_1) - (T_3 - T_2)}{2}$$
Detecting and adjusting incorrect times

Getting the current time from a time server

Computing the relative offset $\theta$ and delay $\delta$

Assumption: $\delta T_{req} = T_2 - T_1 \approx T_4 - T_3 = \delta T_{res}$

$$\theta = T_3 + ((T_2 - T_1) + (T_4 - T_3))/2 - T_4 = ((T_2 - T_1) + (T_3 - T_4))/2$$

$$\delta = ((T_4 - T_1) - (T_3 - T_2))/2$$

Network Time Protocol

Collect eight $(\theta, \delta)$ pairs and choose $\theta$ for which associated delay $\delta$ was minimal.
Keeping time without UTC

Principle

Let the time server scan all machines periodically, calculate an average, and inform each machine how it should adjust its time relative to its present time.

Using a time server

The Berkeley algorithm
Keeping time without UTC

Principle

Let the time server scan all machines periodically, calculate an average, and inform each machine how it should adjust its time relative to its present time.

Using a time server

You’ll have to take into account that setting the time back is never allowed ⇒ smooth adjustments (i.e., run faster or slower).
Reference broadcast synchronization

Essence

- A node broadcasts a reference message \( m \) ⇒ each receiving node \( p \) records the time \( T_{p,m} \) that it received \( m \).
- **Note:** \( T_{p,m} \) is read from \( p \)'s local clock.

Problem: averaging will not capture drift ⇒ use linear regression

NO: \( Offset[p,q](t) = \frac{\sum_{k=1}^{M}(T_{p,k} - T_{q,k})}{M} \)

YES: \( Offset[p,q](t) = \alpha t + \beta \)

RBS minimizes critical path

- Message preparation
- Time spent in NIC
- Delivery time to app.
- Critical path RBS
- Usual critical path
The Happened-before relationship

Issue
What usually matters is not that all processes agree on exactly what time it is, but that they agree on the order in which events occur. Requires a notion of ordering.
The Happened-before relationship

**Issue**

What usually matters is not that all processes agree on exactly what time it is, but that they agree on the order in which events occur. Requires a notion of ordering.

**The happened-before relation**

- If $a$ and $b$ are two events in the same process, and $a$ comes before $b$, then $a \rightarrow b$.
- If $a$ is the sending of a message, and $b$ is the receipt of that message, then $a \rightarrow b$.
- If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

**Note**

This introduces a partial ordering of events in a system with concurrently operating processes.
Logical clocks

Problem

How do we maintain a global view on the system’s behavior that is consistent with the happened-before relation?

Attach a timestamp $C(e)$ to each event $e$, satisfying the following properties:

$P1$ If $a$ and $b$ are two events in the same process, and $a \rightarrow b$, then we demand that $C(a) < C(b)$.

$P2$ If $a$ corresponds to sending a message $m$, and $b$ to the receipt of that message, then also $C(a) < C(b)$.

Problem

How to attach a timestamp to an event when there's no global clock $\Rightarrow$ maintain a consistent set of logical clocks, one per process.
Logical clocks

Problem

How do we maintain a global view on the system’s behavior that is consistent with the happened-before relation?

Attach a timestamp $C(e)$ to each event $e$, satisfying the following properties:

P1 If $a$ and $b$ are two events in the same process, and $a \rightarrow b$, then we demand that $C(a) < C(b)$.

P2 If $a$ corresponds to sending a message $m$, and $b$ to the receipt of that message, then also $C(a) < C(b)$.
Logical clocks

Problem
How do we maintain a global view on the system’s behavior that is consistent with the happened-before relation?

Attach a timestamp $C(e)$ to each event $e$, satisfying the following properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>If $a$ and $b$ are two events in the same process, and $a \rightarrow b$, then we demand that $C(a) &lt; C(b)$.</td>
</tr>
<tr>
<td>P2</td>
<td>If $a$ corresponds to sending a message $m$, and $b$ to the receipt of that message, then also $C(a) &lt; C(b)$.</td>
</tr>
</tbody>
</table>

Problem
How to attach a timestamp to an event when there’s no global clock ⇒ maintain a consistent set of logical clocks, one per process.
Logical clocks: solution

Each process $P_i$ maintains a local counter $C_i$ and adjusts this counter

1. For each new event that takes place within $P_i$, $C_i$ is incremented by 1.
2. Each time a message $m$ is sent by process $P_i$, the message receives a timestamp $ts(m) = C_i$.
3. Whenever a message $m$ is received by a process $P_j$, $P_j$ adjusts its local counter $C_j$ to $\max\{C_j, ts(m)\}$; then executes step 1 before passing $m$ to the application.

Notes

- Property P1 is satisfied by (1); Property P2 by (2) and (3).
- It can still occur that two events happen at the same time. Avoid this by breaking ties through process IDs.
Logical clocks: example

Consider three processes with event counters operating at different rates

<table>
<thead>
<tr>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>10</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>16</td>
<td>20</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>18</td>
<td>24</td>
<td>30</td>
<td>18</td>
<td>24</td>
<td>30</td>
</tr>
<tr>
<td>24</td>
<td>32</td>
<td>40</td>
<td>24</td>
<td>32</td>
<td>40</td>
</tr>
<tr>
<td>30</td>
<td>40</td>
<td>50</td>
<td>30</td>
<td>40</td>
<td>50</td>
</tr>
<tr>
<td>36</td>
<td>48</td>
<td>60</td>
<td>36</td>
<td>48</td>
<td>60</td>
</tr>
<tr>
<td>42</td>
<td>56</td>
<td>70</td>
<td>42</td>
<td>56</td>
<td>70</td>
</tr>
<tr>
<td>48</td>
<td>64</td>
<td>80</td>
<td>48</td>
<td>64</td>
<td>80</td>
</tr>
<tr>
<td>54</td>
<td>72</td>
<td>90</td>
<td>54</td>
<td>72</td>
<td>90</td>
</tr>
<tr>
<td>60</td>
<td>80</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

m₁ m₂ m₃ m₄

P₁ adjusts its clock
P₂ adjusts its clock
Logical clocks: where implemented

Adjustments implemented in middleware

Application layer
- Application sends message
- Adjust local clock and timestamp message

Middleware layer
- Middleware sends message
- Adjust local clock

Network layer
- Message is received
- Message is delivered to application
Example: Total-ordered multicast

Concurrent updates on a replicated database are seen in the same order everywhere

- $P_1$ adds $100 to an account (initial value: $1000$)
- $P_2$ increments account by 1%
- There are two replicas

Result

In absence of proper synchronization: replica #1 ← $1111$, while replica #2 ← $1110$. 
Example: Total-ordered multicast

Solution

- Process $P_i$ sends timestamped message $m_i$ to all others. The message itself is put in a local queue $queue_i$.
- Any incoming message at $P_j$ is queued in $queue_j$, according to its timestamp, and acknowledged to every other process.
Example: Total-ordered multicast

Solution

- Process $P_i$ sends timestamped message $m_i$ to all others. The message itself is put in a local queue $queue_i$.
- Any incoming message at $P_j$ is queued in $queue_j$, according to its timestamp, and acknowledged to every other process.

$P_j$ passes a message $m_i$ to its application if:

1. $m_i$ is at the head of $queue_j$
2. for each process $P_k$, there is a message $m_k$ in $queue_j$ with a larger timestamp.
Example: Total-ordered multicast

Solution

- Process $P_i$ sends timestamped message $m_i$ to all others. The message itself is put in a local queue $queue_i$.
- Any incoming message at $P_j$ is queued in $queue_j$, according to its timestamp, and acknowledged to every other process.

$P_j$ passes a message $m_i$ to its application if:

1. $m_i$ is at the head of $queue_j$
2. for each process $P_k$, there is a message $m_k$ in $queue_j$ with a larger timestamp.

Note

We are assuming that communication is reliable and FIFO ordered.
Lamport’s clocks for mutual exclusion

```python
class Process:
    def __init__(self, chan):
        self.queue = []  # The request queue
        self.clock = 0   # The current logical clock

    def requestToEnter(self):
        self.clock = self.clock + 1  # Increment clock value
        self.queue.append((self.clock, self.procID, ENTER))  # Append request to q
        self.cleanupQ()  # Sort the queue
        self.chan.sendTo(self.otherProcs, (self.clock, self.procID, ENTER))  # Send request

    def allowToEnter(self, requester):
        self.clock = self.clock + 1  # Increment clock value
        self.chan.sendTo([requester], (self.clock, self.procID, ALLOW))  # Permit other

    def release(self):
        tmp = [r for r in self.queue[1:] if r[2] == ENTER]  # Remove all ALLOWs
        self.queue = tmp
        self.clock = self.clock + 1  # Increment clock value
        self.chan.sendTo(self.otherProcs, (self.clock, self.procID, RELEASE))  # Release

    def allowedToEnter(self):
        commProcs = set([req[1] for req in self.queue[1:]])  # See who has sent a message
        return (self.queue[0][1] == self.procID and len(self.otherProcs) == len(commProcs))
```

Example: Total-ordered multicasting
Lamport’s clocks for mutual exclusion

```python
def receive(self):
    msg = self.chan.recvFrom(self.otherProcs)[1]
    self.clock = max(self.clock, msg[0])
    self.clock = self.clock + 1
    if msg[2] == ENTER:
        self.queue.append(msg)
        self.allowToEnter(msg[1])
    elif msg[2] == ALLOW:
        self.queue.append(msg)
    elif msg[2] == RELEASE:
        del self.queue[0]
    self.cleanupQ()
```

# Pick up any message
# Adjust clock value...
# ...and increment
# Append an ENTER request
# and unconditionally allow
# Append an ALLOW
# Just remove first message
# And sort and cleanup

Example: Total-ordered multicasting
Analogy with total-ordered multicast

- With total-ordered multicast, all processes build identical queues, delivering messages in the same order.
- Mutual exclusion is about agreeing in which order processes are allowed to enter a critical section.
Observation

Lamport’s clocks do not guarantee that if $C(a) < C(b)$ that $a$ causally preceded $b$.

Concurrent message transmission using logical clocks

Event $a$: $m_1$ is received at $T = 16$;
Event $b$: $m_2$ is sent at $T = 20$. 

Observation

We cannot conclude that $a$ causally precedes $b$. 

Note
Vector clocks

Observation

Lamport’s clocks do not guarantee that if $C(a) < C(b)$ that $a$ causally preceded $b$.

Concurrent message transmission using logical clocks

<table>
<thead>
<tr>
<th></th>
<th>$P_1$</th>
<th></th>
<th>$P_2$</th>
<th></th>
<th>$P_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>20</td>
<td>40</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>30</td>
<td>50</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
<td>24</td>
<td>30</td>
<td>40</td>
<td>60</td>
</tr>
<tr>
<td>24</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>50</td>
<td>70</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>40</td>
<td>40</td>
<td>60</td>
<td>80</td>
</tr>
<tr>
<td>36</td>
<td>36</td>
<td>48</td>
<td>50</td>
<td>70</td>
<td>90</td>
</tr>
<tr>
<td>42</td>
<td>42</td>
<td>48</td>
<td>60</td>
<td>80</td>
<td>100</td>
</tr>
<tr>
<td>48</td>
<td>48</td>
<td>61</td>
<td>70</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>70</td>
<td>69</td>
<td>80</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>76</td>
<td>77</td>
<td>85</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Event $a$: $m_1$ is received at $T = 16$;
Event $b$: $m_2$ is sent at $T = 20$.

Note

We cannot conclude that $a$ causally precedes $b$. 
Causal dependency

Definition

We say that $b$ may causally depend on $a$ if $ts(a) < ts(b)$, with:

- for all $k$, $ts(a)[k] \leq ts(b)[k]$ and
- there exists at least one index $k'$ for which $ts(a)[k'] < ts(b)[k']$

Precedence vs. dependency

- We say that $a$ causally precedes $b$.
- $b$ may causally depend on $a$, as there may be information from $a$ that is propagated into $b$. 
Capturing causality

Solution: each $P_i$ maintains a vector $VC_i$

- $VC_i[i]$ is the local logical clock at process $P_i$.
- If $VC_i[j] = k$ then $P_i$ knows that $k$ events have occurred at $P_j$.

Maintaining vector clocks

1. Before executing an event $P_i$ executes $VC_i[i] \leftarrow VC_i[i] + 1$.
2. When process $P_i$ sends a message $m$ to $P_j$, it sets $m$’s (vector) timestamp $ts(m)$ equal to $VC_i$ after having executed step 1.
3. Upon the receipt of a message $m$, process $P_j$ sets $VC_j[k] \leftarrow \max\{VC_j[k], ts(m)[k]\}$ for each $k$, after which it executes step 1 and then delivers the message to the application.
Vector clocks: Example

Capturing potential causality when exchanging messages

![Diagram showing vector clocks example]

Analysis

<table>
<thead>
<tr>
<th>Situation</th>
<th>ts(m2)</th>
<th>ts(m4)</th>
<th>ts(m2) &lt; ts(m4)</th>
<th>ts(m2) &gt; ts(m4)</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>(2,1,0)</td>
<td>(4,3,0)</td>
<td>Yes</td>
<td>No</td>
<td>m2 may causally precede m4</td>
</tr>
<tr>
<td>(b)</td>
<td>(4,1,0)</td>
<td>(2,3,0)</td>
<td>No</td>
<td>No</td>
<td>m2 and m4 may conflict</td>
</tr>
</tbody>
</table>
Causally ordered multicasting

Observation

We can now ensure that a message is delivered only if all causally preceding messages have already been delivered.

Adjustment

\( P_i \) increments \( VC_i[i] \) only when sending a message, and \( P_j \) “adjusts” \( VC_j \) when receiving a message (i.e., effectively does not change \( VC_j[j] \)).
Causally ordered multicasting

Observation

We can now ensure that a message is delivered only if all causally preceding messages have already been delivered.

Adjustment

$P_i$ increments $VC_i[i]$ only when sending a message, and $P_j$ “adjusts” $VC_j$ when receiving a message (i.e., effectively does not change $VC_j[j]$).

$P_j$ postpones delivery of $m$ until:

1. $ts(m)[i] = VC_j[i] + 1$
2. $ts(m)[k] \leq VC_j[k]$ for all $k \neq i$
Causally ordered multicasting

Enforcing causal communication

Example: Take VC \(3 = [0, 2, 2]\), ts(\(m\)) = \([1, 3, 0]\) from \(P_1\). What information does \(P_3\) have, and what will it do when receiving \(m\) (from \(P_1\))?
Causally ordered multicasting

Enforcing causal communication

Example

Take $VC_3 = [0, 2, 2]$, $ts(m) = [1, 3, 0]$ from $P_1$. What information does $P_3$ have, and what will it do when receiving $m$ (from $P_1$)?
Mutual exclusion

Problem
A number of processes in a distributed system want exclusive access to some resource.

Basic solutions
Permission-based: A process wanting to enter its critical section, or access a resource, needs permission from other processes.

Token-based: A token is passed between processes. The one who has the token may proceed in its critical section, or pass it on when not interested.
Permission-based, centralized

Simply use a coordinator

(a) Process $P_1$ asks the coordinator for permission to access a shared resource. Permission is granted.

(b) Process $P_2$ then asks permission to access the same resource. The coordinator does not reply.

(c) When $P_1$ releases the resource, it tells the coordinator, which then replies to $P_2$. 
Mutual exclusion Ricart & Agrawala

The same as Lamport except that acknowledgments are not sent

Return a response to a request only when:

- The receiving process has no interest in the shared resource; or
- The receiving process is waiting for the resource, but has lower priority (known through comparison of timestamps).

In all other cases, reply is deferred, implying some more local administration.
Example with three processes

(a) Two processes want to access a shared resource at the same moment.
(b) $P_0$ has the lowest timestamp, so it wins.
(c) When process $P_0$ is done, it sends an $OK$ also, so $P_2$ can now go ahead.
Mutual exclusion: Token ring algorithm

Essence

Organize processes in a **logical** ring, and let a token be passed between them. The one that holds the token is allowed to enter the critical region (if it wants to).

An overlay network constructed as a logical ring with a circulating token

![Diagram of a token-ring network](image)
Decentralized mutual exclusion

**Principle**
Assume every resource is replicated $N$ times, with each replica having its own coordinator $\Rightarrow$ access requires a majority vote from $m > N/2$ coordinators. A coordinator always responds immediately to a request.

**Assumption**
When a coordinator crashes, it will recover quickly, but will have forgotten about permissions it had granted.
Decentralized mutual exclusion

How robust is this system?

Let \( p = \frac{\Delta t}{T} \) be the probability that a coordinator resets during a time interval \( \Delta t \), while having a lifetime of \( T \).
Decentralized mutual exclusion

How robust is this system?

- Let \( p = \Delta t / T \) be the probability that a coordinator resets during a time interval \( \Delta t \), while having a lifetime of \( T \).

- The probability \( P[k] \) that \( k \) out of \( m \) coordinators reset during the same interval is

\[
P[k] = \binom{m}{k} p^k (1 - p)^{m-k}
\]
Decentralized mutual exclusion

How robust is this system?

- Let $p = \Delta t / T$ be the probability that a coordinator resets during a time interval $\Delta t$, while having a lifetime of $T$.

- The probability $\mathbb{P}[k]$ that $k$ out of $m$ coordinators reset during the same interval is

$$
\mathbb{P}[k] = \binom{m}{k} p^k (1 - p)^{m-k}
$$

- $f$ coordinators reset $\Rightarrow$ correctness is violated when there is only a minority of nonfaulty coordinators: when $m - f \leq N/2$, or, $f \geq m - N/2$. 
Decentralized mutual exclusion

How robust is this system?

Let $p = \Delta t / T$ be the probability that a coordinator resets during a time interval $\Delta t$, while having a lifetime of $T$.

The probability $P[k]$ that $k$ out of $m$ coordinators reset during the same interval is

$$P[k] = \binom{m}{k} p^k (1 - p)^{m-k}$$

$f$ coordinators reset $\Rightarrow$ correctness is violated when there is only a minority of nonfaulty coordinators: when $m - f \leq N/2$, or, $f \geq m - N/2$.

The probability of a violation is $\sum_{k=m-N/2}^{N} P[k]$. 
Decentralized mutual exclusion

Violation probabilities for various parameter values

<table>
<thead>
<tr>
<th>N</th>
<th>m</th>
<th>p</th>
<th>Violation</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>5</td>
<td>3 sec/hour</td>
<td>$&lt; 10^{-15}$</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>3 sec/hour</td>
<td>$&lt; 10^{-18}$</td>
</tr>
<tr>
<td>16</td>
<td>9</td>
<td>3 sec/hour</td>
<td>$&lt; 10^{-27}$</td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td>3 sec/hour</td>
<td>$&lt; 10^{-36}$</td>
</tr>
<tr>
<td>32</td>
<td>17</td>
<td>3 sec/hour</td>
<td>$&lt; 10^{-52}$</td>
</tr>
<tr>
<td>32</td>
<td>24</td>
<td>3 sec/hour</td>
<td>$&lt; 10^{-73}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>m</th>
<th>p</th>
<th>Violation</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>5</td>
<td>30 sec/hour</td>
<td>$&lt; 10^{-10}$</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>30 sec/hour</td>
<td>$&lt; 10^{-11}$</td>
</tr>
<tr>
<td>16</td>
<td>9</td>
<td>30 sec/hour</td>
<td>$&lt; 10^{-18}$</td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td>30 sec/hour</td>
<td>$&lt; 10^{-24}$</td>
</tr>
<tr>
<td>32</td>
<td>17</td>
<td>30 sec/hour</td>
<td>$&lt; 10^{-35}$</td>
</tr>
<tr>
<td>32</td>
<td>24</td>
<td>30 sec/hour</td>
<td>$&lt; 10^{-49}$</td>
</tr>
</tbody>
</table>
## Decentralized mutual exclusion

### Violation probabilities for various parameter values

<table>
<thead>
<tr>
<th>N</th>
<th>m</th>
<th>p</th>
<th>Violation</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>5</td>
<td>3 sec/hour</td>
<td>$&lt; 10^{-15}$</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>3 sec/hour</td>
<td>$&lt; 10^{-18}$</td>
</tr>
<tr>
<td>16</td>
<td>9</td>
<td>3 sec/hour</td>
<td>$&lt; 10^{-27}$</td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td>3 sec/hour</td>
<td>$&lt; 10^{-36}$</td>
</tr>
<tr>
<td>32</td>
<td>17</td>
<td>3 sec/hour</td>
<td>$&lt; 10^{-52}$</td>
</tr>
<tr>
<td>32</td>
<td>24</td>
<td>3 sec/hour</td>
<td>$&lt; 10^{-73}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>m</th>
<th>p</th>
<th>Violation</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>5</td>
<td>30 sec/hour</td>
<td>$&lt; 10^{-10}$</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>30 sec/hour</td>
<td>$&lt; 10^{-11}$</td>
</tr>
<tr>
<td>16</td>
<td>9</td>
<td>30 sec/hour</td>
<td>$&lt; 10^{-18}$</td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td>30 sec/hour</td>
<td>$&lt; 10^{-24}$</td>
</tr>
<tr>
<td>32</td>
<td>17</td>
<td>30 sec/hour</td>
<td>$&lt; 10^{-35}$</td>
</tr>
<tr>
<td>32</td>
<td>24</td>
<td>30 sec/hour</td>
<td>$&lt; 10^{-49}$</td>
</tr>
</tbody>
</table>

So....

What can we conclude?
## Mutual exclusion: comparison

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Messages per entry/exit</th>
<th>Delay before entry (in message times)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Distributed</td>
<td>$2 \cdot (N - 1)$</td>
<td>$2 \cdot (N - 1)$</td>
</tr>
<tr>
<td>Token ring</td>
<td>$1, \ldots, \infty$</td>
<td>$0, \ldots, N - 1$</td>
</tr>
<tr>
<td>Decentralized</td>
<td>$2 \cdot m \cdot k + m, k = 1, 2, \ldots$</td>
<td>$2 \cdot m \cdot k$</td>
</tr>
</tbody>
</table>
Election algorithms

Principle
An algorithm requires that some process acts as a coordinator. The question is how to select this special process *dynamically*.

Note
In many systems the coordinator is chosen by hand (e.g. file servers). This leads to centralized solutions ⇒ single point of failure.
Election algorithms

Principle
An algorithm requires that some process acts as a coordinator. The question is how to select this special process dynamically.

Note
In many systems the coordinator is chosen by hand (e.g. file servers). This leads to centralized solutions ⇒ single point of failure.

Teasers
1. If a coordinator is chosen dynamically, to what extent can we speak about a centralized or distributed solution?
2. Is a fully distributed solution, i.e. one without a coordinator, always more robust than any centralized/coordinated solution?
Basic assumptions

- All processes have unique id’s
- All processes know id’s of all processes in the system (but not if they are up or down)
- Election means identifying the process with the highest id that is up
Election by bullying

Principle

Consider $N$ processes $\{P_0, \ldots, P_{N-1}\}$ and let $id(P_k) = k$. When a process $P_k$ notices that the coordinator is no longer responding to requests, it initiates an election:

1. $P_k$ sends an $ELECTION$ message to all processes with higher identifiers: $P_{k+1}, P_{k+2}, \ldots, P_{N-1}$.
2. If no one responds, $P_k$ wins the election and becomes coordinator.
3. If one of the higher-ups answers, it takes over and $P_k$’s job is done.
Election by bullying

The bully election algorithm

![Diagram showing the bully election algorithm]
## Election in a ring

### Principle

Process priority is obtained by organizing processes into a (logical) ring. Process with the highest priority should be elected as coordinator.

- Any process can start an election by sending an election message to its successor. If a successor is down, the message is passed on to the next successor.

- If a message is passed on, the sender adds itself to the list. When it gets back to the initiator, everyone had a chance to make its presence known.

- The initiator sends a coordinator message around the ring containing a list of all living processes. The one with the highest priority is elected as coordinator.
Election in a ring

Election algorithm using a ring

- The solid line shows the election messages initiated by $P_6$
- The dashed one the messages by $P_3$
A solution for wireless networks

A sample network

- Nodes: a, b, c, d, e, f, g, h, i, j
- Connections: a-b, b-c, c-d, d-e, e-f, f-g, g-h, h-i, i-j, j-a
- Capacity labels: 4, 6, 3, 1, 4, 5, 8, 2, 2, 4
- Broadcasting node: 4
A solution for wireless networks

A sample network

g receives broadcast from b first

e receives broadcast from g first
A solution for wireless networks

A sample network
In large-scale distributed systems in which nodes are dispersed across a wide-area network, we often need to take some notion of proximity or distance into account ⇒ it starts with determining a (relative) location of a node.
Computing position

Observation
A node $P$ needs $d + 1$ landmarks to compute its own position in a $d$-dimensional space. Consider two-dimensional case.

Computing a position in 2D

Solution
$P$ needs to solve three equations in two unknowns $(x_P, y_P)$:

$$d_i = \sqrt{(x_i - x_P)^2 + (y_i - y_P)^2}$$
Global positioning system

Assuming that the clocks of the satellites are accurate and synchronized
- It takes a while before a signal reaches the receiver
- The receiver’s clock is definitely out of sync with the satellite

Basics

Observation
4 satellites $\Rightarrow$ 4 equations in 4 unknowns (with $\Delta r$ as one of them)
Global positioning system

Assuming that the clocks of the satellites are accurate and synchronized

- It takes a while before a signal reaches the receiver
- The receiver’s clock is definitely out of sync with the satellite

Basics

- $\Delta_r$: unknown deviation of the receiver’s clock.

Observation

4 satellites $\Rightarrow$ 4 equations in 4 unknowns (with $\Delta_r$ as one of them)
Global positioning system

Assuming that the clocks of the satellites are accurate and synchronized

- It takes a while before a signal reaches the receiver
- The receiver’s clock is definitely out of sync with the satellite

Basics

- $\Delta_r$: unknown deviation of the receiver’s clock.
- $x_r, y_r, z_r$: unknown coordinates of the receiver.

Observation

4 satellites $\Rightarrow$ 4 equations in 4 unknowns (with $\Delta_r$ as one of them)
Global positioning system

Assuming that the clocks of the satellites are accurate and synchronized

- It takes a while before a signal reaches the receiver
- The receiver’s clock is definitely out of sync with the satellite

Basics

- $\Delta_r$: unknown deviation of the receiver’s clock.
- $x_r, y_r, z_r$: unknown coordinates of the receiver.
- $T_i$: timestamp on a message from satellite $i$

Observation

4 satellites $\Rightarrow$ 4 equations in 4 unknowns (with $\Delta_r$ as one of them)
Global positioning system

Assuming that the clocks of the satellites are accurate and synchronized
- It takes a while before a signal reaches the receiver
- The receiver’s clock is definitely out of sync with the satellite

Basics
- \( \Delta_r \): unknown deviation of the receiver’s clock.
- \( x_r, y_r, z_r \): unknown coordinates of the receiver.
- \( T_i \): timestamp on a message from satellite \( i \)
- \( \Delta_i = (T_{\text{now}} - T_i) + \Delta_r \): measured delay of the message sent by satellite \( i \).

Observation
4 satellites \( \Rightarrow \) 4 equations in 4 unknowns (with \( \Delta_r \) as one of them)
Global positioning system

Assuming that the clocks of the satellites are accurate and synchronized
- It takes a while before a signal reaches the receiver
- The receiver’s clock is definitely out of sync with the satellite

Basics
- $\Delta_r$: unknown deviation of the receiver’s clock.
- $x_r, y_r, z_r$: unknown coordinates of the receiver.
- $T_i$: timestamp on a message from satellite $i$
- $\Delta_i = (T_{now} - T_i) + \Delta_r$: measured delay of the message sent by satellite $i$.
- Measured distance to satellite $i$: $c \times \Delta_i$ ($c$ is speed of light)

Observation
- 4 satellites $\Rightarrow$ 4 equations in 4 unknowns (with $\Delta_r$ as one of them)
Assuming that the clocks of the satellites are accurate and synchronized

- It takes a while before a signal reaches the receiver
- The receiver’s clock is definitely out of sync with the satellite

**Basics**

- \( \Delta_r \): unknown deviation of the receiver’s clock.
- \( x_r, y_r, z_r \): unknown coordinates of the receiver.
- \( T_i \): timestamp on a message from satellite \( i \)
- \( \Delta_i = (T_{now} - T_i) + \Delta_r \): measured delay of the message sent by satellite \( i \).
- Measured distance to satellite \( i \): \( c \times \Delta_i \) (\( c \) is speed of light)
- Real distance: \( d_i = c\Delta_i - c\Delta_r = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2} \)

**Observation**

4 satellites \( \Rightarrow \) 4 equations in 4 unknowns (with \( \Delta_r \) as one of them)
WiFi-based location services

Basic idea

- Assume we have a database of known access points (APs) with coordinates
- Assume we can estimate distance to an AP
- Then: with 3 detected access points, we can compute a position.

War driving: locating access points

- Use a WiFi-enabled device along with a GPS receiver, and move through an area while recording observed access points.
- Compute the centroid: assume an access point $AP$ has been detected at $N$ different locations $\{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_N\}$, with known GPS location.
- Compute location of $AP$ as $\vec{x}_{AP} = \frac{\sum_{i=1}^{N} \vec{x}_i}{N}$.

Problems

- Limited accuracy of each GPS detection point $\vec{x}_i$
- An access point has a nonuniform transmission range
- Number of sampled detection points $N$ may be too low.
Computing position

Problems

- Measured latencies to landmarks fluctuate
- Computed distances will not even be consistent

Inconsistent distances in 1D space

Solution: minimize errors

- Use $N$ special landmark nodes $L_1, \ldots, L_N$.
- Landmarks measure their pairwise latencies $\tilde{d}(L_i, L_j)$
- A central node computes the coordinates for each landmark, minimizing:

$$
\sum_{i=1}^{N} \sum_{j=i+1}^{N} \left( \frac{\tilde{d}(L_i, L_j) - \hat{d}(L_i, L_j)}{\tilde{d}(L_i, L_j)} \right)^2
$$

where $\hat{d}(L_i, L_j)$ is distance after nodes $L_i$ and $L_j$ have been positioned.
Computing position

Choosing the dimension $m$

The hidden parameter is the dimension $m$ with $N > m$. A node $P$ measures its distance to each of the $N$ landmarks and computes its coordinates by minimizing

$$\sum_{i=1}^{N} \left( \frac{\tilde{d}(L_i, P) - \hat{d}(L_i, P)}{\tilde{d}(L_i, P)} \right)^2$$

Observation

Practice shows that $m$ can be as small as 6 or 7 to achieve latency estimations within a factor 2 of the actual value.
Vivaldi

Principle: network of springs exerting forces

Consider a collection of $N$ nodes $P_1, \ldots, P_N$, each $P_i$ having coordinates $\vec{x}_i$. Two nodes exert a mutual force:

$$\vec{F}_{ij} = (\tilde{d}(P_i, P_j) - \hat{d}(P_i, P_j)) \times u(\vec{x}_i - \vec{x}_j)$$

with $u(\vec{x}_i - \vec{x}_j)$ is the unit vector in the direction of $\vec{x}_i - \vec{x}_j$

Node $P_i$ repeatedly executes steps

1. Measure the latency $\tilde{d}_{ij}$ to node $P_j$, and also receive $P_j$’s coordinates $\vec{x}_j$.
2. Compute the error $e = \tilde{d}(P_i, P_j) - \hat{d}(P_i, P_j)$
3. Compute the direction $\vec{u} = u(\vec{x}_i - \vec{x}_j)$.
4. Compute the force vector $F_{ij} = e \cdot \vec{u}$
5. Adjust own position by moving along the force vector: $\vec{x}_i \leftarrow \vec{x}_i + \delta \cdot \vec{u}$.
Data dissemination: Perhaps the most important one. Note that there are many variants of dissemination.

Aggregation: Let every node $P_i$ maintain a variable $v_i$. When two nodes gossip, they each reset their variable to

$$v_i, v_j \leftarrow (v_i + v_j)/2$$

Result: in the end each node will have computed the average $\bar{v} = \sum_i v_i/N$. 
Example applications

Typical apps

- **Data dissemination**: Perhaps the most important one. Note that there are many variants of dissemination.

- **Aggregation**: Let every node $P_i$ maintain a variable $v_i$. When two nodes gossip, they each reset their variable to

$$v_i, v_j \leftarrow (v_i + v_j)/2$$

Result: in the end each node will have computed the average $\bar{v} = \sum_i v_i / N$.

- What happens in the case that initially $v_i = 1$ and $v_j = 0, j \neq i$?