CS4513
Distributed Computer Systems

Synchronization
(Ch 5)

Introduction
• Communication not enough. Need cooperation → Synchronization
• Distributed synchronization needed for:
  - transactions (bank account via ATM)
  - access to shared resource (network printer)
  - ordering of events (network games where players have different ping times)

Outline
• Intro (done)
• Clock Synchronization (next)
• Global Time and State
• Election Algorithms
• Mutual Exclusion
• Distributed Transactions

Clock Synchronization
• When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time
• Consider make
  - Compilation machine compares time stamps
• Same holds when using NFS mount
• Can we set all clocks in a distributed system to have the same time?

Physical Clocks
• "Exact" time was computed by astronomers
  - Take "noon" for two days, divide by 24*60*60
  → Mean solar second
• But...
  - Earth is slowing! (35 days over 300 million years)
  - Short term fluctuations (Magma core, and such)
    → Could take many days for average, but still erroneous
• Physicists take over (Jan 1, 1958)
  - Count transitions of cesium 133 atom
    → 9,192,631,770 == 1 solar second
  - 50 cesium 133 clocks averaged
  → International Atomic Time (TAI)
  - To stop day from "shifting" (remember, earth is slowing) translate TAI into Universal Coordinated Time (UTC)
• UTC is broadcast (shortwave radio pulses)

Clock Synchronization Algorithms
• Not every machine has UTC receiver
  → If one, then keep others synchronized
• Computer timers go off H times/sec, incr counter
  → Ideally, if H=60, 216,000 per hour (dC/dt = 0)
• But typical errors, 10^-5, so 215,998 to 216,002
• Specifications can give you maximum drift rate (p)
  → Every Δt seconds, will be at most 2p/Δt apart
  → If want drift of δ, re-synchronize every δ/2p
  → Various algo (next)
Cristian’s Algorithm

* Every $\delta/2 \rho$, ask server for time
* What are the problems?
  * Major
    - Client clock is fast
    - What to do?
  * Minor
    - Non-zero amount of time to sender
    - What to do?

The Berkeley Algorithm

a) The time daemon asks all the other machines for their clock values
b) The machines answer
c) The time daemon tells everyone how to adjust their clock

Cristian’s and Berkeley’s are centralized. Problems?

Decentralized Algorithms

* Periodically (every $R$ seconds), each machine broadcasts current time
* Collect time samples for some time time ($S$)
  * Take average and set time
  * Can discard m so m faulty clocks don’t hurt
  * Can improve by computing $(T_1 - T_0)/2$
    - Need probes to obtain
  * Used by Network Time Protocol (NTP)
    - Worldwide accuracy of 1-50 msec

Outline

* Intro (done)
* Clock Synchronization (done)
* Global Time and State (next)
* Election Algorithms
* Mutual Exclusion
* Distributed Transactions

Lamport Timestamps

a) Each process with own clock with different rates.
b) Lamport’s algorithm corrects the clocks.
c) Can add machine ID to break ties
Use Example: Totally-Ordered

- San Fran customer adds $100, NY bank adds 1% interest. San Fran will have $1,111 and NY will have $1,110.
- Updating a replicated database and leaving it in an inconsistent state.
- Can use Lamport's to totally order.

Consistent Global State

- Need for state of distributed system, say, for termination detection.

Consistent Global State (2)

- Processes all connected. Can initiate state message (M).
- Organization of a process and channels for a distributed snapshot.

Consistent Global State (3)

- Process Q receives M for the first time and records its local state. Sends M on all outgoing links.
- Q records all incoming messages.
- Q receives M for its incoming channel and finishes recording the state of the incoming channel.
- Can then send state to initiating process.
- System can still proceed normally.

Outline

- Intro (done)
- Clock Synchronization (done)
- Global Time and State (done)
- Election Algorithms (next)
- Mutual Exclusion
- Distributed Transactions

Election Algorithms

- Often need one process as a coordinator.
- All processes in distributed systems may be equal.
  - Assume have some "ID" that is a number.
- Need way to "elect" process with the highest number as leader.
The Bully Algorithm (1)

- Process 4 notices 7 down
- Process 4 holds an election
- Process 5 and 6 respond, telling 4 to stop
- Now 5 and 6 each hold an election

The Bully Algorithm (2)

d) Process 6 tells process 5 to stop
e) Process 6 wins and tells everyone
   * Eventually "biggest" (bully) wins
   * If processes 7 comes up, starts elections again

A Ring Algorithm

- Coordinator down, start ELECTION
  - Send message down ring, add ID
  - Once around, change to COORDINATOR (biggest)
- Even if two ELECTIONS started at once, everyone will pick same leader

Outline

- Intro (done)
- Clock Synchronization (done)
- Global Time and State (done)
- Election Algorithms (done)
- Mutual Exclusion (done)
- Distributed Transactions (next)

Mutual Exclusion: A Centralized Algorithm

a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted.
b) Process 2 then asks permission to enter the same critical region.
   The coordinator does not reply. (Or, can say "denied")
c) When process 1 exits the critical region, it tells the coordinator, then replies to 2.
   * But centralized, single point of failure

A Distributed Algorithm

a) Processes 0 and 2 want to enter the same critical region at the same moment.
b) Process 1 doesn't want to, says "OK". Process 0 has the lowest timestamp, so it wins. Queues up "OK" for 2.
c) When process 0 is done, it sends an OK to 2 so can now enter the critical region.
   * (Again, can modify to say "denied")
**A Token Ring Algorithm**

1. An unordered group of processes on a network.
2. A logical ring constructed in software.
   - Process must have token to enter.
   - If don’t want to enter, pass token along.
   - If host down, recover ring. If token lost, regenerate token. If in critical section long?

**Mutual Exclusion Algorithm**

**Comparison**

- **Centralized** most efficient
- **Token ring** efficient when many want to use critical region

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Messages per entry/exit</th>
<th>Delay before entry (in message times)</th>
<th>Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td>3</td>
<td>2</td>
<td>Coordinator crash</td>
</tr>
<tr>
<td>Distributed</td>
<td>2 (n-1)</td>
<td>2 (n-1)</td>
<td>Process crash</td>
</tr>
<tr>
<td>Token ring</td>
<td>1 to n-1</td>
<td>6 to n-1</td>
<td>Lost token, process crash</td>
</tr>
</tbody>
</table>

- Centralized most efficient
- Token ring efficient when many want to use critical region

**Outline**

- Intro (done)
- Clock Synchronization (done)
- Global Time and State (done)
- Election Algorithms (done)
- Mutual Exclusion (done)
- Distributed Transactions (next)

**The Transaction Model**

- Gives you mutual exclusion plus...
- Consider using PC (Quicken) to:
  - Withdraw $a from account 1
  - Deposit $a to account 2
- If interrupt between 1) and 2), $a gone!
- Multiple items in single, atomic action
  - It all happens, or none
  - If process backs out, as if never started

**Transaction Primitives**

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN_TRANSACTION</td>
<td>Make the start of a transaction</td>
</tr>
<tr>
<td>END_TRANSACTION</td>
<td>Terminate the transaction and try to commit</td>
</tr>
<tr>
<td>ABORT_TRANSACTION</td>
<td>Kill the transaction and restore the old values</td>
</tr>
<tr>
<td>READ</td>
<td>Read data from a file, a table, or otherwise</td>
</tr>
<tr>
<td>WRITE</td>
<td>Write data to a file, a table, or otherwise</td>
</tr>
</tbody>
</table>

* Above may be system calls, libraries or statements in a language (Sequential Query Language or SQL)

**Example: Reserving Flight from White Plains to Nairobi**

(A) BEGIN_TRANSACTION

reserve WP -> JFK;
reserve JFK -> Nairobi:
reserve Nairobi -> Malindi;

END_TRANSACTION

(B) ABORT_TRANSACTION

- Transaction to reserve three flights commits
- Transaction aborts when third flight is unavailable
  - The "all-or-nothing" is one property. Others:
Transaction Properties

1) Atomic -
   • Others don’t see intermediate results, either
2) Consistent
   • System invariants not violated
   • Ex: no money lost after operations
3) Isolated
   • Operations can happen in parallel but as if were done serially
4) Durability
   • Once commits, move forward
   • (Ch 7, won’t cover more)
   • ACID

Classification of Transactions

• Flat Transactions
  - Limited
  - Example: what if want to keep first part of flight reservation? If abort and then restart, those might be gone.
  - Example: what if want to move a Web page. All links pointing to it would need to be updated. It could lock resources for a long time
• Also Distributed and Nested Transactions

Distributed Transactions

* Nested transaction gives you a hierarchy
  - Can distribute (example: WP → JFK, JFK → Nairobi)
  - But may require multiple databases
* Distributed transaction is "flat" but across distributed data (example: JFK and Nairobi database)

Private Workspace (1)

* File system with transaction across multiple files
  - Normally, updates seen + No way to undo
* Private Workspace → Copy files
  - Only update Public Workspace once done
  - If abort transaction, remove private copy.
  - But copy can be expensive!
    - How to fix?

Private Workspace (2)

a) Original file index (descriptor) and disk blocks
b) Copy descriptor only. Copy blocks only when written.
   - Modified block 0 and appended block 3
c) Replace original file (new blocks plus descriptor) after commit

Outline

• Intro (done)
• Clock Synchronization (done)
• Global Time and State (done)
• Election Algorithms (done)
• Mutual Exclusion (done)
• Distributed Transactions
  - Overview (done)
  - Implementation (next)
**Writeahead Log**

- Don’t make copies. Instead, record action plus old and new values.

<table>
<thead>
<tr>
<th>a)</th>
<th>b)</th>
<th>c)</th>
<th>d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 0;</td>
<td>x = 0 / 1</td>
<td>x = 0 / 1</td>
<td>x = 0 / 1</td>
</tr>
<tr>
<td>y = 0;</td>
<td>y = 0</td>
<td>y = 0</td>
<td>y = 0</td>
</tr>
<tr>
<td>BEGIN_TRANSACTION;</td>
<td>[x = 0 / 1]</td>
<td>[x = 0 / 1]</td>
<td>[x = 0 / 1]</td>
</tr>
<tr>
<td>x = x + 1;</td>
<td>y = y + 2</td>
<td>x = x + y;</td>
<td>[x = 0/4]</td>
</tr>
<tr>
<td>y = y + 2;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>END_TRANSACTION;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a) - A transaction
b) - d) log before each statement is executed
  - If transaction commits, nothing to do
  - If transaction is aborted, use log to rollback

**Concurrency Control (1)**

- General organization of managers for handling transactions.

**Concurrency Control (2)**

- General organization of managers for handling distributed transactions.

**Serializability**

- Allow parallel execution, but end result as if serial

Schedule 1:

<table>
<thead>
<tr>
<th>Schedule 1</th>
<th>Schedule 2</th>
<th>Schedule 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN_TRANSACTION</td>
<td>BEGIN_TRANSACTION</td>
<td>BEGIN_TRANSACTION</td>
</tr>
<tr>
<td>x = 0;</td>
<td>x = 0;</td>
<td>x = 0;</td>
</tr>
<tr>
<td>x = x + 1;</td>
<td>x = x + 2;</td>
<td>x = x + 1;</td>
</tr>
<tr>
<td>END_TRANSACTION;</td>
<td>END_TRANSACTION;</td>
<td>END_TRANSACTION;</td>
</tr>
<tr>
<td>x = 0;</td>
<td>x = 0;</td>
<td>x = 0;</td>
</tr>
<tr>
<td>x = x + 2;</td>
<td>x = x + 3;</td>
<td>x = x + 2;</td>
</tr>
<tr>
<td>END_TRANSACTION;</td>
<td>END_TRANSACTION;</td>
<td>END_TRANSACTION;</td>
</tr>
</tbody>
</table>

- If in parallel, only some possible schedules
- 2 is serialized
- Concurrency controller needs to manage

**Two-Phase Locking**

- Can lead to deadlocks (use OS techniques to resolve)
- Can prove: if used by all transactions, then all schedules will be serializable

**Timestamp Ordering**

- Pessimistic
  - Every read and write gets a timestamp (unique, using Lamport’s alg)
  - If conflict, abort sub-operation and re-try
- Optimistic
  - Allow all operations since conflict rate
  - At end, if conflict, roll-back