Topics – Clock Synchronization

• Physical Clocks
• Clock Synchronization Algorithms
  – Network Time Protocol
  – Berkeley Algorithm
  – Clock Synchronization in Wireless Networks
• Logical Clocks
  – Lamport’s Logical Clocks
  – Vector Clocks
Clock Synchronization

• In a centralized system, time is **unambiguous**
  — Processes obtain the time, by issuing a system call to the kernel
    • Process 1 issues R1 to obtain time – Kernel return T1
    • Process 2 issues R2 to obtain time – Kernel returns T2
    • It is guaranteed that T2 ≥ T1

• In a distributed system, achieving agreement on time is **not** trivial
  — It is impossible to guarantee that physical clocks run at the same frequency

Physical clocks

• Computer Clocks are actually Timers, a precisely machined quartz crystal
  — When kept under tension, crystals oscillate at a well-defined frequency

• Associated with the crystal are two registers, a counter and holding register
  — At each oscillation, the counter decrements by 1
  — When counter reaches zero, an interrupt is generated
    • Possible to program a timer to generate an interrupt 60 times a second – Each interrupt corresponds to a clock tick

• Within a singly computer, a clock being slightly off is tolerable, with minimal impact

• In multi-computers systems, clock skew is problematic
Dealing with drift

- Set computer to true time
  - Not good idea to set clock back – Illusion of time moving backwards can confuse message ordering and software development environments

- Gradual clock correction
  - If fast then make clock run slower until it synchronizes
  - If slow then make clock run faster until it synchronizes

![Compensating for a fast clock](chart.png)

\[ \frac{dc}{dt} > 1 \]

UTC time, \( t \)

Computer's time, \( C \)

Clock Synchronized

Linear Compensating Function Applied

Skew
**NTP Basic Operation**

- **NTP is set up pairwise**
  - A probes B for its current time and B also probes A
- **Compute the offset, ϑ, and the delay estimation, δ**
  - \( δ = \frac{(T_2 - T_1 + (T_4 - T_3))}{2} \)
- **Buffer 8 pairs of (ϑ, δ), and select the minimal value for δ as the best delay estimation between the two servers**
  - The associated ϑ with the minimal delay δ is the most reliable

![Diagram of NTP Basic Operation](image)

**The Berkeley Algorithm – Request**

- The time daemon asks all the other machines for their clock values.
- Daemon computes the average of the received answers
- The time daemon tells everyone how to adjust their clock.

![Diagram of The Berkeley Algorithm](image)
Logical Clocks

• For many DS algorithms, associating an event to an absolute “real time” is not essential
  – What’s important is that the processes in the Distributed System agree on the ordering in which certain events occur
  – “relative time” may not relate to the “real time”.
• Such “clocks” are referred to as Logical Clocks
  – Lamport’s timestamps
  – Vector timestamps

Lamport’s Logical Clocks

• To synchronize logical clocks, Lamport defines a relation called “happens-before”
• “happens before” relation is denoted as →
• The relation can be observed directly in two situations:
  – If a and b are events in the same process, and a occurs before b, then a → b is true.
  – If a is the event of a message being sent by one process, and b is the event of the message being received by another process, then a → b
Concurrent Events

- If two events happen without any message, then we can't say anything about their relative occurrence in time.
- We can say that $a \rightarrow b \rightarrow c \rightarrow d \rightarrow f$, but we can say little about $e$ other than $e \rightarrow f$.

Vector Clocks (2)

Vector clocks are constructed by letting each process $P_i$ maintain a vector $VC_i$ with the following two properties:

1. $VC_i[i]$ is the number of events that have occurred so far at $P_i$. In other words, $VC_i[i]$ is the local logical clock at process $P_i$.
2. If $VC_i[j] = k$ then $P_i$ knows that $k$ events have occurred at $P_j$. It is thus $P_i$'s knowledge of the local time at $P_j$. 
Vector Clocks – Messages

• **Message Passing**
  – When \( P_i \) sends a message to \( P_j \), the message has timestamp \( t[] = VT_i[] \)

  – When \( P_j \) receives the message, it sets \( VT_j[k] \) to \( \max (VT_j[k], t[k]) \), for \( k = 1, 2, ..., N \)

  * For example, \( P_2 \) receives a message with timestamp \( (3,2,4) \) and \( P_2 \)'s timestamp is \( (3,4,3) \), then \( P_2 \) adjusts its timestamp to \( (3,4,4) \)

Vector Clocks Analysis

• **Claim –** \( e \rightarrow e' \) iff \( e.VT < e'.VT \)
Process Synchronization

- Mutual Exclusion Algorithms
  - Permission Based
    - Centralized
    - DeCentralized
    - Distributed
  - Token Based
    - Token Ring

- Election Algorithms
  - Bully Algorithm
  - Ring Algorithm
  - Election in Wireless Networks
  - Election in Large Scale Systems

Mutual Exclusion – Classification
Centralized Algorithm

- Centralized Algorithm achieves DME by closely “mimicking” ME in single processor systems
  - One process, C, is the Coordinator – Coordinates access to resources
  - Other processes issue requests to access resource
    1. Request Resource
    2. *Wait for Response*
    3. Receive Grant
    4. Access Resource
    5. Release Resource

DME Decentralized Voting Algorithm

- Unlike in the centralized scheme, if the resource cannot be granted, a decentralized DME coordinator informs the requesting process that the resource is unavailable
- To gain access to a resource, a process my secure a majority vote \( m \geq \), where \( m \) is the number of coordinators
- A failed coordinator resets at arbitrary moments, not having to remember any vote it gave before the crash
  - Ignoring previously granted permissions, may lead a recovered coordinator to grant permission again to another process
A Distributed ME Algorithm

- **Ricart and Agrawala Algorithm** assumes there is a mechanism for "**totally ordering of all events**" in the system and a **reliable** message system
  - **Lamport’s algorithm** can be used for total ordering
- A process wanting to enter it CS sends a message with (CS name, process id, current time) to all processes, including itself
- When a process receives a CS request from another process, it reacts based on its current state with respect to the CS requested.
  - **Three** possible cases must be considered

RA Algorithm – Example

- **P₀** and **P₂** request access to the same resource, nearly at the same time
  - Both send requests with timestamps, 8 and 12, respectively
- **P₀** has the lowest timestamp, 8
  - **P₀** wins access to the resource
- Upon completion of its CS, **P₀** sends OK message to **P₂**
  - **P₂** can enter its own CS
A Token Ring Algorithm

- An unordered group of processes on a network.
- A logical ring constructed in software
  - Token circulates at high speed on the network
  - A process must have a token to enter its CS

Election Algorithms

- Many distributed algorithms such as mutual exclusion and deadlock detection require a coordinator process.
- When the coordinator process fails, the distributed group of processes must execute an election algorithm to determine a new coordinator process.
  - Different criteria can be used to select the new coordinator
The Bully Algorithm

When any process, P, notices that the coordinator is no longer responding it initiates an election:

1. P sends an election message to all processes with higher id numbers.
2. If no one responds, P wins the election and becomes coordinator.
3. If a higher process responds, it takes over.
   - Process P’s job is done.

Bully Algorithm Example

- 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.
Ring Algorithm – Basic Operation

- RA assumes that the processes are logically ordered in a ring *implies a successor pointer and an active process list* that is unidirectional.
- When any process, P, notices that the coordinator is no longer responding it initiates an election:

  1. P sends message containing **P’s process ID** to the **next available** successor.

  2. At each active process, the receiving process adds its process number to the list of processes in the message and forwards it to its successor.

     - Eventually, **the message gets back to the sender**.

  3. The initial sender sends out a second message letting everyone know who the coordinator is **the process with the highest number** and indicating the current members of the active list of processes.
Consistency Models

• Background – Replication Motivation
• Data-Centric Consistency Models
  – Continuous Consistency
  – Consistent Ordering of Operations
• Client-Centric Consistency Models
  – Eventual Consistency
  – Monotonic Reads
  – Monotonic Writes
  – Read Your Writes
  – Writes Follow Reads

Performance and Scalability

• Main issue – To keep replicas consistent, we generally need to ensure that all conflicting operations are done in the same order, across all servers
• Conflicting Operations – Concurrent Transactions
  – Read–Write Conflict: a read operation and a write operation act concurrently
  – Write–Write Conflict: two concurrent write operations
• Guaranteeing global ordering on conflicting operations may be a costly operation, with impact on scalability
• Potential Solution – Weaken consistency requirements to avoid global synchronization, when possible
What is a Consistency Model?

• A “consistency model” is a contract between a distributed data-store and its processes.
  – If the processes agree to the rules, the data-store will perform correctly and as advertised.

Data-Centric Consistency Models – Strong and Weak Models

• **Strong consistency models:** Operations on shared data are synchronized – Do not require synchronization operations
  – **Strict consistency** – Related to absolute global time
  – **Sequential Consistency** – Also known as serializability
  – **Linearizability** – To achieve atomicity
  – **Causal Consistency** – To maintain only causal relations
  – **FIFO Consistency** – To maintain only individual ordering

• **Weak consistency models:** Synchronization occurs only when shared data is locked and unlocked – Rely on synchronization operations
  – **Weak Consistency**
  – **Release Consistency**
  – **Entry Consistency**

• **Note** – Weaker the consistency model are more scalable
Sequential Consistency – Definition

Sequential consistency – The result of any execution is the same as if the read and write operations by all processes were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program.

- Any valid interleaving of read and write operations is OK, but all processes must see the same interleaving.
  - The events observed by each process must globally occur in the same order, or it is not sequentially consistent.
    - It doesn’t actually matter if the events don’t really agree with clock time, as long as they are consistent.

Linearizability

- In sequential consistency, absolute time is somewhat irrelevant – The order of events is most important.
- Linearizability – Sequential + Operations are ordered according to a global time
- A data store is said to be linearizable when each operation is timestamped and the following condition holds:
  - “The result of any execution is the same as if the operations by all processes on the data store were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program.
  - In addition, if \( ts_{Op_1}(x) < ts_{Op_2}(y) \), then \( Op_1(x) \) should precede \( Op_2(y) \) in this sequence.”
Causal Consistency

• **Necessary condition** – Writes that are potentially causally related must be seen by all processes in the same order.
  – Concurrent writes may be seen in a different order on different machines.

• If event A is a direct or indirect result of another prior event B, then all processes should observe event A before observing event B.
  
  \[
  A = A + 1; // \text{First two events are causally related,}
  
  B = A * 5; // \text{because } B \text{ reads } A \text{ after } A \text{ was written.}
  
  C = C * 3; // \text{This is a concurrent statement.}
  \]

FIFO Consistency

• Necessary Condition – Writes performed by a single process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.

<table>
<thead>
<tr>
<th>P1: W(x)a</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2: R(x)a  W(x)b  W(x)c</td>
</tr>
<tr>
<td>P3: R(x)b  R(x)a  R(x)c</td>
</tr>
<tr>
<td>P4: R(x)a  R(x)b  R(x)c</td>
</tr>
</tbody>
</table>

• A valid sequence of events of FIFO consistency.
  – P₂’s writes are seen in the correct order.

• FIFO consistency is easy to implement.
Weak Consistency - Semantics

- The weak consistency models enforce consistency on a group of operations, as opposed to individual reads and writes as is the case for strict, sequential, causal and FIFO consistency models.
- A synchronize(S) operation by P, causes all writes by P to be propagated to all other replicas and all external writes are propagated to P.
  - Process P forces the just written value out to all the other replicas.
  - Process P can be sure it’s getting the most recently written value before it reads.

Summary of Consistency Models

<table>
<thead>
<tr>
<th>Consistency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linearizability</td>
<td>All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp.</td>
</tr>
<tr>
<td>Sequential</td>
<td>All processes see all shared accesses in the same order. Accesses are not ordered in time.</td>
</tr>
<tr>
<td>Causal</td>
<td>All processes see causally-related shared accesses in the same order.</td>
</tr>
<tr>
<td>FIFO</td>
<td>All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Consistency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weak</td>
<td>Shared data can be counted on to be consistent only after a synchronization is done.</td>
</tr>
<tr>
<td>Release</td>
<td>Shared data are made consistent when a critical region is exited.</td>
</tr>
<tr>
<td>Entry</td>
<td>Shared data pertaining to a critical region are made consistent when a critical region is entered.</td>
</tr>
</tbody>
</table>

(a) Consistency models not using synchronization operations.
(b) Models with synchronization operations.
Client-Centric Consistency Models

- Data-centric consistency models aim at providing the system-wide view on a data store.
- Client-centric consistency models are generally used for applications that lack simultaneous updates
  - Most operations involve reading data.
- Weak, **client-centric** consistency models
  - Eventual consistency
  - Monotonic reads
  - Monotonic writes
  - Read your writes
  - Writes follow reads

Eventual Consistency

- Systems such as DNS and WWW can be viewed as applications of large scale distributed and replicated databases that tolerate a relatively high degree of inconsistency
- **Common Assumption** — if no updates take place for a long time, all replicas will gradually and eventually become consistent
- This form of consistency is called **eventual consistency**
- Eventual consistency requires only that updates are guaranteed to propagate to all replicas
- Eventual consistent data stores work fine as long as clients always access the same replica — **what happens when different replicas are accessed?**
Consistency for Mobile Users – Inconsistencies

• Various inconsistencies may occur as a mobile user moves from location A to location B
  – Updates at A may not have yet been propagated to B
  – May be reading newer entries than the ones available at A
  – Updates at B may eventually conflict with those at A

• The consistency model must ensure that the entries updated and/or read at by the mobile user at A, are in B the way you left them in A.
  – This requirement ensures that the database will appear to be consistent to you.

Client-Centric Consistency – Guarantees

• Guarantees are the responsibility of “session manager”, not servers
• Two sets are maintained
  – Read-set – set of writes that are relevant to session reads
  – Write-set – set of writes performed in session
• Update dependencies captured in read sets and write sets
• Four different client-central consistency models
  – Monotonic reads
  – Monotonic writes
  – Read your writes
  – Writes follow reads
Fundamentals of Functional Programming Languages

• The basic process of computation is fundamentally different in a FPL than in an imperative language
  – In an imperative language, operations are done and the results are stored in variables for later use
  – Management of variables is a constant concern and source of complexity for imperative programming

• FPL takes a mathematical approach to the concept of a variable
  – Variables are bound to values, not memory locations
  – A variable's value cannot change, which eliminates assignment as a possible operation

FPL and IPL – Example

• Summing the integers 1 to 10 in IPL – The computation method is variable assignment

```plaintext
total = 0;
for (i = 1; i ≤ 10; ++i)
    total = total+i;
```

• Summing the integers 1 to 10 in FPL – The computation method is function application

```plaintext
sum [1..10]
```
Lambda Calculus

• The lambda calculus is a formal mathematical system to investigate functions, function application and recursion.
• A lambda expression specifies the parameter(s) and the mapping of a function in the following form
  \( \lambda x \cdot x \times x \times x \) – for the function cube \( (x) = x \times x \times x \)
• Lambda expressions describe nameless functions
• Lambda expressions are applied to parameter(s) by placing the parameter(s) after the expression
  \( (\lambda x \cdot x \times x \times x) 3 \Rightarrow 3 \times 3 \times 3 \Rightarrow 27 \)
  \( (\lambda x,y \cdot (x-y) \times (y-x)) (3,5) \Rightarrow (3-5) \times (5-3) \Rightarrow -4 \)

FPL Map and Fold

• “map” and “fold” – FPL higher-order functions
• (map f list1 [list2 list3 …])
  \( \Rightarrow (\text{map square} \ (1 \ 2 \ 3 \ 4)) \Rightarrow (1 \ 4 \ 9 \ 16) \)
• (fold f list […])
  \( \Rightarrow (\text{fold} + \ (1 \ 4 \ 9 \ 16)) \Rightarrow 30 \)
• (fold + (map square (map – list1 list2))))
MapReduce

- Programmers specify two functions:
  - `map (k, v) → <k', v'>*`
  - `reduce (k', v') → <k', v'>*`
  - All values with the same key are reduced together
- The execution framework handles everything else!
- **Not quite ...**
  - `partition (k', number of partitions) → partition for k'`
  - Often a simple hash of the key, e.g., `hash(k') mod N`
  - Divides up key space for parallel reduce operations
  - `combine (k', v') → <k', v'>*`
  - Mini-reducers that run in memory after the map phase
  - Used as an optimization to reduce network traffic

Roots in Functional Programming
MapReduce Data Flow

- A MapReduce job is a unit of work to be performed
  - Job consists of the MapReduce Program, the Input data and the Configuration Information
- The MapReduce job is divided into two types of tasks – map tasks and reduce tasks
  - It is not uncommon for MapReduce jobs to have thousands of individual tasks to be assigned to cluster nodes
- The Input data is divided into fixed-size pieces called splits
  - One map task is created for each split
  - The user-defined map function is run on each split
- Configuration information indicates where the input lies, and the output is stored

Jobtracker – Scheduling and Coordination

- In large jobs, the total number of tasks may exceed the number of tasks that can be run on the cluster concurrently,
  - The Jobtracker must maintain a task queue and assign nodes to waiting tasks as the nodes become available.
- Another aspect of Jobtracker’s responsibilities involves coordination among tasks belonging to different jobs
  - Jobs from different users, for example
- Designing a large-scale, shared resource to support several users simultaneously in a predictable, transparent and policy-driven fashion is challenging!
MapReduce Stragglers

- The speed of a MapReduce job is sensitive to the stragglers’ performance – tasks that take an unusually long time to complete
  - The map phase of a job is only as fast as the slowest map task.
  - The running time of the slowest reduce task determines the completion time of a job
- Stragglers may result from unreliable hardware
  - A machine recovering from frequent hardware errors may become significantly slower
- The barrier between the map and reduce tasks further compounds the problem

MapReduce – Combiners

- Combiner functions can be used to minimize the data transferred between map and reduce tasks
  - Combiners are particularly useful when MapReduce jobs are limited by the bandwidth available on the cluster
    - Combiners are user-specified functions
- Combiner functions run on the map output
  - The combiner’s is then fed to the reduce function
- Since it is an optimization function, there is no guarantee how many times combiners are called for a particular map output record, if at all
  - Calling the combiner zero, one, or many times should produce the same output from the reducer.
Combiner Property

• The combiner function calls can be expressed as follows:
  – Max(0, 20, 10, 25, 15) = Max(Max(0, 20, 10), Max(25, 15)) = Max(20, 25) = 25
  – Max() is commonly referred to as **distributive**

• Not all function exhibit distributive property
  – Mean(0, 20, 10, 25, 15) = (0+20+10+25+15)/5 = 14
  – Mean(Mean(0, 20, 10), Mean(25, 15)) = Mean (10, 20) = 15

• Combiners do not replace producers
  – Producers are still needed to process recorders with the same key from different maps