Frequently asked questions from the previous class survey

- Do RMI URLs have to be unique? Format?
- CORBA marshalling/unmarshalling

Topics covered in this lecture

- Replication
- Consistency Models
- Data centric consistency model
  - Continuous consistency models
  - Sequential consistency

Let's look at a simple Class

```java
public class Person implements Serializable {
    private String name;
    private String place;
    private int year;

    public Person(String aName, String aPlace, int aYear) {
        name = aName;
        place = aPlace;
        year = aYear;
    }

    // Methods for accessing instance variables follow...
}
```

The Java Serialized form

```
<table>
<thead>
<tr>
<th>Field</th>
<th>Class name, version number</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset</td>
<td>1</td>
<td>length</td>
</tr>
<tr>
<td>offset</td>
<td>0</td>
<td>numbers</td>
</tr>
<tr>
<td>offset</td>
<td>64</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>

Values of instance variables
```
Encoding class information

- The information about a class consists of the:
  - Class name
  - Version
    - Should change with major changes to the class
- Version assignment
  - Set by programmer
    - Calculated as a hash of the name of the class and its instance variables, methods, and interfaces
- Deserialization checks to see if it has correct version of the class

Java objects contain references to other objects

- When an object is serialized, all the objects that it references are serialized
  - Ensures when an object is reconstructed all its references can be fulfilled at the destination
- References are serialized as handles
  - Handle is a reference to an object within the serialized form
  - 1:1 correspondence between object references and handles
  - Each object is written exactly once
    - Subsequent occurrences of an object: The handle is written instead of the object

Serializing an object

- Class information is written out
  - Each class is given a handle and no class is written more than once
  - Handles are written where necessary
- This is followed by (types and names of) instance variables
- Recursive procedure continues until
  - Class information, types, and names of instance variables of all classes are written out

How are things written out?

- Contents of instance variables that are of primitive types (e.g., int, char, boolean, etc.)
  - Written in portable binary format using methods of the ObjectOutputStream class
- Strings and characters are written using the writeUTF method
- Universal Transfer Format (UTF)
  - ASCII characters are represented unchanged (1 byte), Unicode characters with multiple bytes

Controlling serialization

- Programmers can modify the effects of serialization
- Declare variables that should not be serialized as transient
  - E.g., references to local resources such as files and sockets
What we will look at in our discussions:
- Replication
- Consistency
  - Models
  - Client models
  - Protocols
- Eventual Consistency
- Brewer’s CAP Theorem

Why are these inter-related topics important?
- Performance
- Correctness
- Failure to account for interactions between these issues?
  - Poor performance
  - Inaccurate results
  - The holy grail of demonstrable incompetency in systems development!

Rationale for replication:
- Reliability
- Availability
- Performance

Rationale for replication: Reliability
- Replication as a safeguard against failures
- Protection against data corruptions

File System example:
- 3 copies
- If one fails, process can choose from the other two
- Read/write performed on each copy
  - At least 2 of the reads must concure
  - Protect against a failing write

Rationale for replication: Increased Availability
- Users require services to be highly available
- Proportion of time when service is accessible with reasonable response times should be close to 100%
- Factors relevant to high-availability
  - Delays due to pessimistic concurrency control
  - Server failures
  - Network partitions and disconnected operations
Replication maintains availability despite server failures [1/2]

- Data is replicated at failure independent servers
- Client software should be able to access data at an alternative server if default server fails

---

Replication maintains availability despite server failures [2/2]

- If each of the $n$ servers has an independent probability $p$ of failing or becoming unreachable
- The availability of an object stored at each of these servers:
  - $1$ - probability(all servers fail or are unreachable)
  - $1 - p^n$

---

Replication maintains availability despite server failures: Example

- There is a 5% probability of independent server failures?
- There are two servers
  - Availability is $1 - p^n$
  - $1 - (0.05)^2 = 1 - 0.0025 = 99.75$

---

Rationale for replication: Performance

- Ability to scale with numbers
  - Processes access data managed by a server
  - Replicate server; distribute work
- Ability to scale with geographical area
  - Place copy of data in proximity of processes using it
  - Time to access service decreases
  - Perceived performance improves

---

But replication exacts a price ...

- A client may perceive better performance but ...
  - More network bandwidth needed
    - To keep replicas in sync
  - Consistency problems
    - When a copy is modified, it becomes different
    - Modifications have to be made on all copies

---

Replication Costs: When and how modifications must be made to copies

- Fetching a page from a remote Web server
  - **Objective:** Improving access times
  - Web browsers locally cache a web page
    - If user requests the same page
      - Returned from cache
      - User is happy with the load times
      - What if user always wants the latest copy?
Simple solutions to the stale copy problem

1. **Don't cache** web page
   - If there is no nearby replica, performance is poor
   - Also, what if the page does not change that often?

2. **Let server invalidate/update** caches
   - Server must track all caches
   - Degrades server performance

---

Replication as a scaling technique

- Placing data copies close to processes
  - Improves access times
  - Distributes work
  - Potential problems ...

Replication for scaling:
Network bandwidth

- Process $P$ accesses a replica $N$ times per second
- Replica is itself updated $M$ times per second
- If $N << M$?
  - Several updated versions of replica never accessed
  - Network traffic to install those versions: wasted!
  - Perhaps installing a replica was not a good idea?

Replication for scaling:
Consistency issues

- Consistency might itself be subject to scaling problems
- Collection of copies is consistent when **all** copies are the same
  - Read on any copy returns the same result
  - Updates propagated to all copies before the next operation?
  - Tight consistency

Consistency issues in replication

- Update performed at all copies as an **atomic operation**
  - Transaction
- Implementing atomicity with large number of replicas is difficult
  - May be dispersed on a WAN
  - Operations cannot complete quickly

Other things that replicas need to agree on ...

- Replicas must agree on **when** operation must be performed locally
- Replicas need to decide on **ordering**
  - Lamport timestamps
  - Coordinator assigned order
The Replication Dilemma

- Alleviating scalability issues
  - Replication and caching: improves performance

- Keeping copies consistent?
  - Requires global synchronization
  - Costly in terms of performance
    - Time
    - Network bandwidth

Data centric consistency models

- Consistency is in the context of read/write operation on distributed, shared data
  - Memory
  - Database
  - File systems

- The broader term data store is more commonly used

Consistency model

- **Contract** between processes and the data store
  - If processes agree to obey certain rules
    - Data store works correctly

Consistency that we intuitively expect

- Process performing a read on a data item
  - Expects value to show results of last write operation on that item

- Without a global clock?
  - Difficult to define which write was the last one
We thus need to provide other definitions …

- Each model restricts values that a read operation on a data item can return.
- Models with the greatest restrictions:
  - Easiest to use
- Models with minor restrictions:
  - Difficult to use
- Easy-to-use models do not perform as well as difficult ones.

Loosening of consistency

- Needed for efficiency and performance
- No general rules
- Tolerance depends on the application

Continuous consistency

- Three axes for defining inconsistencies
- Deviations between replicas in terms of:
  - Numerical values
  - Staleness between replicas
  - Ordering of update operations
- Deviations form continuous consistency ranges

Example of using continuous consistency models: Stock prices

- Two copies of a stock should not deviate by more than 2 cents.
  - Absolute numerical deviation
- Two copies do not deviate by more than 0.5%
  - Relative numerical deviation
- If stock goes up and one replica is updated
  - If change does not violate specified deviations:
    - Replicas are considered consistent

Numerical and Staleness deviations

- Numerical deviation can also be expressed in terms of number of updates
  - Applied at a replica, but not seen by other replicas
- Staleness deviations
  - Last time a replica was updated
  - Replica can provide old data as long as it is not too old
  - Weather reports
Ordering of updates may also be allowed to be different:
- Within a certain bound
- Updates applied tentatively at local copy
  - Need global agreement with all replicas
- Before an update becomes permanent
  - Might be rolled back
  - Applied in a different order

### Consistency Unit (cont)
- Specifies unit over which consistency is to be measured
- Examples
  - Record representing a stock
  - Weather report

Looking at the conit a little closer:
Example with 2 replicas
- Each replica maintains a 2D vector clock
- Operation carried out by replica \( i \) at (its) logical time \( t \): \( <t, i> \)
- Example conit contains data items \( x \) and \( y \)

Tracking consistency deviations:
Conit items \( x \) and \( y \) are initialized to 0

Vector Clocks at each replica

Replica A: Vector Clock A = (15, 8)
Replica B: Vector Clock B = (8, 11)

Committed Operation: Tentative Operation

<table>
<thead>
<tr>
<th>Replica A</th>
<th>Replica B</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = x + 2 )</td>
<td>( y = y + 5 )</td>
</tr>
<tr>
<td>( x = y \times 2 )</td>
<td>( x = y \times 2 )</td>
</tr>
<tr>
<td>( y = y + 5 )</td>
<td>( y = y + 3 )</td>
</tr>
<tr>
<td>( y = y + 2 )</td>
<td>( y = y + 1 )</td>
</tr>
<tr>
<td>( y = y + 1 )</td>
<td>( y = y + 1 )</td>
</tr>
</tbody>
</table>
Order deviations are the number of tentative operations at each replica

<table>
<thead>
<tr>
<th>Replica A</th>
<th>Replica B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation Result</td>
<td>Operation Result</td>
</tr>
<tr>
<td>x = x+2</td>
<td>y = y+2</td>
</tr>
<tr>
<td>x = x+3</td>
<td>y = y+5</td>
</tr>
</tbody>
</table>

Order Deviation = 3

Numerical deviations in our example

- Numerical deviation here is the number of unseen updates from the other replica
- Weight of this deviation at replica A is the maximum difference between:
  - Committed values of conit at A
  - Result from operations at B not seen by A

Conit:
- x=6, y=3
- x=2, y=5

Replica A: Unseen Updates = 1

Replica B: Unseen Updates = 3

Weight = max(diff((2,2)), diff((0,5))) = 5

Quantifying the numerical deviations at each replica

<table>
<thead>
<tr>
<th>Replica A</th>
<th>Replica B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation Result</td>
<td>Operation Result</td>
</tr>
<tr>
<td>x = x+2</td>
<td>y = y+2</td>
</tr>
<tr>
<td>x = x+3</td>
<td>y = y+5</td>
</tr>
<tr>
<td>x = x+6</td>
<td>y = y+6</td>
</tr>
</tbody>
</table>

Unseen Updates = 5

Unseen Updates = 3

Weight = max(diff((2,2)), diff((0,5))) = 5

Tradeoffs between fine grained and coarse grained conits

- If conit represents a lot of data
  - Updates aggregated for all data in conit
  - Replicas become inconsistent sooner
- If conit is smaller
  - Fewer updates needed
  - Total number of conits to be managed goes up

Before we put conits to practical use two things need to happen

- Protocols to enforce consistency
- Developers specify consistency requirements
  - Difficult

Conits are declared alongside updates

AffectsConit(ConitQ, 1, 1)
append message m to queue Q

Appending message m to queue Q belongs to a conit named ConitQ
Conits are declared alongside reads

```
DependsOnConit([Conit], 4, 0, 60)
read message m from the head of queue Q
```

- Numerical deviation: 4
  - At most 4 unseen updates at other replicas
- Ordering deviation: 0
  - No tentative local updates
- Staleness deviation: 60 seconds
  - Check Q for staleness periodically

The contents of this slide-set are based on the following references