Consistency

Distribute
- partition data and computation across multiple machines
  - Java applets, DNS, WWW
Replicate
- make copies of data available at different machines
  - mirrored websites, replicated files, replicated databases
Cache
- allow client processes to access local copies
  - Web caches, file caching

The Model

- Shared data is kept in a data store
  - a register, a file system, a database...
- Clients access the data store through read and write operations
- Consistency Semantics: a contract between the data store and its clients that specifies the results that clients can expect to obtain when accessing the data store

Lamport's Registers

- Safe:
  A read not concurrent with any write returns the most recently written value
- Regular:
  Safe + a read overlapping with a write obtains either the old or the new value
- Atomic:
  Reads and writes are totally ordered so that values returned by reads are the same as if the operations had been performed with no overlapping
Lamport’s Registers

**Safe:**
A read not concurrent with any write returns the most recently written value

**Regular:**
Safe + a read overlapping with a write obtains either the old or the new value

**Atomic:**
Reads and writes are totally ordered so that values returned by reads are the same as if the operations had been performed with no overlapping

---

Lamport’s Registers

**Safe:**
A read not concurrent with any write returns the most recently written value

**Regular:**
Safe + a read overlapping with a write obtains either the old or the new value

**Atomic:**
Reads and writes are totally ordered so that values returned by reads are the same as if the operations had been performed with no overlapping

---

**Lamport’s Registers**

**Safe:**
A read not concurrent with any write returns the most recently written value

**Regular:**
Safe + a read overlapping with a write obtains either the old or the new value

**Atomic:**
Reads and writes are totally ordered so that values returned by reads are the same as if the operations had been performed with no overlapping
Lamport's Registers

**Safe:**
A read not concurrent with any write returns the most recently written value.

**Regular:**
Safe + a read overlapping with a write obtains either the old or the new value.

**Atomic:**
Reads and writes are totally ordered so that values returned by reads are the same as if the operations had been performed with no overlapping.

Atomic semantics is equivalent to Linearizability (Herlihy and Wing, 1991)

Sequential Consistency

"The result of any execution is the same as if the operations of all the processes were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by the program" (Lamport, 1979)

Sequential Consistency

"The result of any execution is the same as if the operations of all the processes were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by the program" (Lamport, 1979)

Is this data store sequentially consistent?
Weakening Sequential Consistency: Causal Consistency

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. (Hutto and Ahamad, 1990)

Is this data store causally consistent? Yes
Sequentially consistent? No

Is this data store causally consistent?

Is this data store causally consistent?
More Weakening: FIFO Consistency

"Writes done 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" (PRAM consistency, Lipton and Sandberg 1988)

\[
\begin{align*}
\text{Process } p_1: & \\
x := 1 \\
\text{if } (y = 0) \text{ then} & \\
\quad \text{kill}(p_2) \\
\text{Process } p_2: & \\
y := 1 \\
\text{if } (x = 0) \text{ then} & \\
\quad \text{kill}(p_1)
\end{align*}
\]

What are the possible outcomes?

Lessons

- The stronger the consistency model, the easier to program in it (sequential, atomic)
- The weaker the consistency model, the least expensive it is to enforce it (FIFO)
- Can we have our cake and eat it too?
Towards Weak Consistency

Consider a process in a critical section, updating records of a distributed database:
- Many updates may affect the same record.
- No other process can access the records while they are being updated.
- Should all updates be propagated to all replicas?
- Better ideas?

Weak Consistency (Dubois et al, 1988)

- Uses an explicit synchronization variable.
- `synchronize(S)` works like a memory fence: operations on one side of the fence cannot move to the other side.
- Weak consistency requires that:
  - Accesses to synchronization variables be sequentially consistent.
  - Writes to a variable by one process and access (read or write) of that variable by another process be separated by two synchronization operations:
    - One executed after the write by the writer.
    - One executed before the access by the second process.

Weak Consistency

(Weak Consistency (Dubois et al, 1988))

- Uses an explicit synchronization variable.
- `synchronize(S)` works like a memory fence:
  - Accesses to a synchronization variable are sequentially consistent.
  - No access to a synchronization variable is allowed until all previous writes have completed everywhere.
  - No data access (read or write) is allowed until all previous accesses to synchronization variables have been performed.
- A write to a variable followed by an access to the same variable by a different process are to be separated by two synchronization operations:
  - One executed after the write by the writer.
  - One executed before the access by the second process.

Release Consistency (Gharachorloo et al. 1990)

- The synchronize operation does not allow to specify whether the invoker is about to:
  - Finish a series of writes.
  - Start a series of reads.
- Release consistency distinguishes between two synchronization operations:
  - `ACQUIRE`: All local copies of protected data are brought up to date with remote ones.
  - `RELEASE`: Protected data that have been changed are propagated out to other local copies.
- Release consistency variables can just be FIFO consistent (not necessarily sequentially consistent).

Following these rules, the result of an execution will be no different than with a sequentially consistent data store.
Entry Consistency
(Bershad et al. 1991)

- To increase concurrency, each shared data item is explicitly associated with a synchronization variable.
- Synchronization variables may be accessed in exclusive (for writes) and non-exclusive (read-only) mode.

\[ P_1: \text{Acq}(L_x) \rightarrow (\text{W}(x) \text{Acq}(L_y) \text{R}(y) \text{Rel}(L_x) \text{Rel}(L_y)) \]

\[ P_2: \text{Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the guard synchronization variable, not even in nonexclusive mode.} \]

\[ P_3: \text{After an exclusive mode access to a synchronization variable has been performed, any other process next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner (i.e. the process that last acquired it).} \]

Bayou
(http://www2.parc.com/csl/projects/bayou/)

System Model

- Many machines: your laptop, your iPhone, your iPod...
- Each storing a copy of a Database (say, a calendar)
- Write any, read any
- Any ideas?

System Model

- Many machines: your laptop, your iPhone, your iPod...
- Each storing a copy of a Database (say, a calendar)
- Write any, read any
- No continuous network connectivity (partitions)
- Heterogeneous network
- Clients can update the database when disconnected
Design Goals

- Arbitrary communication topologies
- Operation over low-bandwidth networks
- Incremental progress, even through disconnections
- Eventual consistency: if no updates, all replicas eventually converge
- Efficient storage management
- Propagation through transportable media (your key)
- Lightweight management of dynamic replica set
- Allow application to manage inconsistencies

Anti-entropy

- Entropy = Disorder
  - Anti-entropy brings two replicas up-to-date with each other
- Three design choices:
  - it is a one-way operation between pairs of servers:
    - no need for all servers, or even a quorum of them, to be available!
    - eventual consistency through epidemic spread of updates
  - it occurs through the propagation of write operations
    - no need to swap databases!
  - write propagation is constrained by the accept-order

Basic Setup

- Each replica/server maintains
  - database
  - write log
- Replica maintains all local and remote updates in a log
- Replica applies all updates to the database when it receives them, potentially rolling back to manage a conflict
- Invariant: Entry sorted by accept order
- Goal: if all replicas have the same ordered log, then their databases are identical

Accept Order

- When a replica receives a write from a client application, it assigns to it an accept stamp \((\text{acceptTime}, \text{replicaId})\)
- \text{acceptTime} defines:
  - a total order over all writes accepted by a replica
  - a partial order (accept order) over all writes in the system
- With replicaId, accept stamps define a total order over all writes
- We want an update mechanism that maintains the following prefix property over the set of writes known by a server:

  If the log of replica \(R_i\) contains a write \(w\) first accepted by replica \(R_j\), then the log also contains all the writes accepted by \(R_j\) prior to \(w\)
Version Vectors

- Prefix property enables compact representation of a replica's position.
- Each replica \( R_i \) maintains a version vector \( R_i.V[] \) such that 
  \( R_i.V[j] \) is the largest accept stamp of any write accepted 
  by \( R_i \) and known to \( R_i \). ....sounds familiar?
- Replicas use version vectors to bring each other up-to-date.

Basic Anti-Entropy (run at replica \( S' \))

- \( \text{anti-entropy}(S,R) \)
- Get \( R_i.V[] \) from receiving replica \( R \)
- /* send all the writes unknown to \( R \) */
- \( w := \text{first write in } S-writeLog \)
- while(\( w \)) do
-   if \( R_i.V(w.replicaId) < w.acceptTime \) then
-     /* \( w \) is new for \( R \) */
-     send \( w \) to \( R \)
- \( w := \text{next write in } S-writeLog \)
- Supports
  - multiple communication topologies
  - different policies for when and with whom to reconcile
  - incremental updates(!)
  - can be used with transportable media
  - it slices! it dices!
- But does it work?

Two issues

- The prefix property could be violated if the anti-entropy protocol is run over channels that are not FIFO
  □ easy to fix
- The protocol requires the log to grow without bound
  □ when can a replica garbage collect a write?
  1. wait for the write to have propagated to all replicas
  2. In Bayou, as soon as it knows that the write has become stable

Stable writes

- A write is stable if its position in the write log will not change
- We saw how to implement stability in our implementation of causal delivery:
  □ use gap detection property of vector clocks!
  □ happy?
Stability

- A write is stable if its positions in the write log will not change.
- We saw how to implement stability in our implementation of causal delivery:
  - Use gap detection property of vector clocks!
  - Happy?
- Partitions can prevent garbage collection!
- In Bayou, a write is stable as soon as a primary replica says so.

Write Stability in Bayou

- A write is stable when a primary replica declares it so by committing the write.
- The primary commits the write by assigning to it a unique, monotonically increasing commit sequence number (CSN).
- New total order of writes enforced by the ordered triple 
  \((CSN, acceptTime, replicaId)\).

Propagating committed writes (run at replica \(S\))

```verbatim
anti-entropy(S, R)
Get R.V[1] and R.CSN from receiving replica R
/* first send all the committed writes unknown to R */
if R.CSN < S.CSN then
  w := first committed write unknown to R
while (w) do
  if w.acceptTime ≤ R.V(replicaID) then
    /* R has the write, but does not know it is committed */
    send (COMMIT, R, w.acceptTime, w.replicaId, w.CSN) to R
  else send w to R.
  w := next committed write in S.writeLog
w := first tentative write
/* then send all tentative writes */
while (w) do
  if R.V(w.replicaID) < w.acceptTime then
    send w to R
  w := next write in S.writeLog
```

Write log truncation

- Replica \(S\) maintains a version vector \(S.OSN\) representing truncated prefix of its (committed) write log.
- \(S.OSN\) is the CSN for \(S\).
- If \(S.OSN > R.CSN\), then \(S\) has discarded committed writes that \(R\) is missing.
- What to do?
Propagating committed writes (run at replica $S$)

anti-entropy ($S, R$)

Get $R.V[\cdot]$ and $R.CSN$ from receiving replica $R$
/* check if $R.writeLog$ s.t. not enough to send tentative and committed writes */
if ($S.CSN < R.CSN$) then
   /* execute a full database transfer */
   Roll back $S.DB$ to the state corresponding to $S.O$
   send $S.DB$ to $R$ /* this will be the new $R.DB$ */
   send $S.O$ to $R$ /* this will be the new $R.O$ */
   /* now same as before */
if ($R.CSN < S.CSN$) then
   $w$ := first committed write unknown to $R$
while (w) do
   if $w.acceptTime \leq R.V(replicaId)$ then
      send (COMMIT, $w.acceptTime$, $w.$replicaId, $w.CSN$) to $R$
   else send $w$ to $R$.
   $w$ := next committed write in $S.writeLog$
while (w) do
   if $R.V(w.$replicaId) < $w.acceptTime$ then send $w$ to $R$
   $w$ := next write in $S.writeLog$

Guaranteeing Consistency

- Causally consistent prefix at any time
  - replicas maintain a logical clock that advances when new writes are received by a client or through anti-entropy
- Total order enforced by primary
  - eventual consistency
- Causal prefix property ensures several session guarantees

Consistency

- Suppose you want to access a database but don't want to keep a replica
- Contact one server, start processing
- If the server fails, switch to a different one
- What if the new one is inconsistent with the old one?

Session guarantees in Bayou

- Monotonic Reads
  - If a client reads the value of $x$, any successive read operation on $x$ by that client returns the same, or a more recent value
- Monotonic Writes
  - A write operation by a client on $x$ is completed before any successive write operation on $x$ by the same client
- Read Your Writes
  - The effect of a write operation by a client on $x$ will always be seen by a successive read operation on $x$ by the same client
- Writes Follow Reads
  - A write on $x$ by a client following a previous read on $x$ by the same client is guaranteed to take place on the same or a more recent value of $x$ that the one that was read
Replica Management

- Need a mechanism to
  - assign unique id to a replica
  - determine when a replica is created or retired

Announce creation/retirement using a write!
- new replica \( R_i \) "creates itself" by sending creation write to \( R_j \)
- Creation write is entered in \( R_i \)'s writeLog with accept stamp \((\infty, \text{acceptTime}_{R_i}, R_j)\)
- \( R_i \)'s replicaId := \((\text{acceptTime}_{R_i}, R_j)\)
- \( R_i \)'s acceptTime := \( \text{acceptTime}_{R_i} + 1 \)

The Case of the Missing VV entry

- What if \( S \), during an anti-entropy session with \( R_i \), finds that \( R_i \) does not know of server \( R_j \)?
  - it could be because \( R_i \) has not heard of \( R_j \) yet...
  - ...or, because \( R_i \) has already heard of \( R_j \)'s retirement

Suppose \( R_j \) is missing the entry for \( R_j = (TS_{k,j}, R_k) \)
- if \( R_i.V(R_k) \geq TS_{k,j} \) then \( R_i \) knows of \( R_j \)'s creation
  - \( R_i \) has seen \( R_j \) retire. \( S \) needs not forward writes accepted by \( R_j \)
- if \( R_i.V(R_k) < TS_{k,j} \) then \( R_j \) is news to \( R_i \)
  - \( S \) sends to \( R_i \) all the writes \( S \) knows \( R_j \) accepted