CompSci131

Parallel and Distributed Systems

Prof. A. Veidenbaum

Today's topics

- Gossip-based coordination
- Consistency and replication
- Consistency models

- Reading assignment:
  - Today: 6.7, 7.1-7.2
  - Next time: 7.2

Last Lecture Covered

- Elections at scale, with multiple leaders
- Epidemic and gossip protocols

Gossip based coordination

- **Examples of using epidemic protocols**
  - Aggregation
  - Large-scale peer sampling
  - Overlay construction
- **Aggregation: collect or aggregate info from all nodes**
  - Let's consider it in terms of numerical values
  - A node Pi has a number v_i, contacts Pj which has v_j
  - Both update $v_i, v_j = (v_i + v_j) / 2$
  - Eventually all nodes will have the same value – the average
- **A variation - P_1 starts with 1, every other node with 0**
  - The result is 1/N, where n is the number of nodes
  - Now all nodes know N

Gossip based coordination (2)

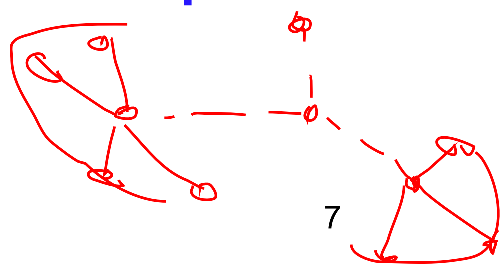
- **How does a node choose a peer randomly?**
  - Need to know all nodes in the network
    - » Works for small networks only

- **Solution: a peer sampling service (PSS)**
  - Fully decentralized
  - Can be done using epidemic protocols
  - Using *partial views*

- **A partial view is a list of *c* neighbors a node maintains**
  - Ideally each node is a randomly chosen, *live* node
    - » Nodes regularly exchange entries from their partial view
    - » Each entry has an age associated with it

PSS (2)

- The following operations are used
  - selectPeer: Randomly select a neighbor from the local partial view
  - selectToSend: Select some other entries from the partial view,
    - » add to the list intended for a selected neighbor
      - Includes its own entry
    - » List size is c/2 + 1
  - selectToKeep: Add received entries to partial view
    - » remove repeated items and shrink the view to c items.
  - ReceiveFromAny
- Two ways to construct new views upon receipt
  - Discard the entries they sent to each other, i.e. swap them
  - Discard as many old entries as possible
    - » Still maintaining a total of c



PSS (3)

```
selectPeer(&Q);
selectToSend(&bufs);
sendTo(Q, bufs);

receiveFrom(Q, &bufr);
selectToKeep(p_view, bufr);
```

```


receiveFromAny(&P, &bufr);
selectToSend(&bufs);
sendTo(P, bufs);
selectToKeep(p_view, bufr);
```

- **As long as peers regularly exchange partial views, selecting from a partial view is indistinguishable from randomly selecting from the entire network**

Consistency

- **Data replication creates multiple copies**
  - **For performance, fault tolerance**
- **It is multiple copies that cause the consistency problem**
  - **A modification of one copy requires update of all copies**
  - **How can it be done?**
    - » **Must prevent old (inconsistent) values from being used**
- **Can try the following: send update to all replicas**
  - **while doing a local update, and wait for all to ACK**
  - **Problems?**
- **Will also consider consistency for shared memory**
  - **The issue here is what to expect when reading/updating data**

- **Data consistency is hard to implement efficiently at large scale**
  - **This is because all copies need to be updated synchronously**
  - **Sometimes, a *simpler* model will suffice**

- **The consistency problem has no general solution**
  - **It can only be solved by loosening some requirements!**
  - **Or for special cases, like shared memory systems**

- **We will look at**
  1. **Keeping replicas consistent**
  2. **Content distribution to replicas**

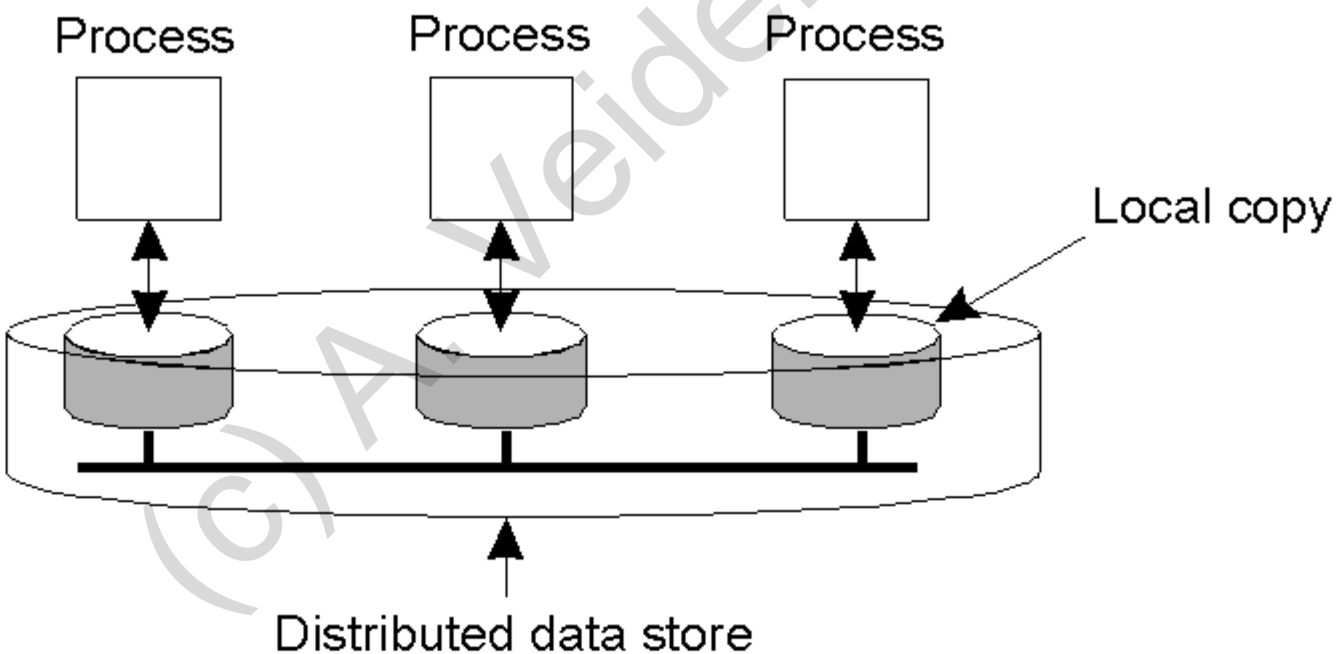
(Replicated) Data Consistency

- **An intuitive definition: data is consistent when *ALL* copies *ALWAYS* have *the same* value**
  - **Hard/expensive to achieve**
    - » **Recall totally ordered multicast**
  - **Requires an atomic update: update all copies “at once”**
    - » **May not be scalable**
      - **Replication was supposed to solve scalability problems!**

- **Can we loosen the consistency constraints?**
  - **For instance, do not require the atomicity of update**
  - **BUT:**
    - » **require that a DS still has “*correct*” execution/results**

Data-Centric Consistency

- A logical data store - shared memory, file, database
  - physically distributed and replicated across multiple nodes
    - » Each node can access a local (or nearby) copy
- *A write* is an operation that changes data
- Otherwise, it's a *Read*



A Consistency Model

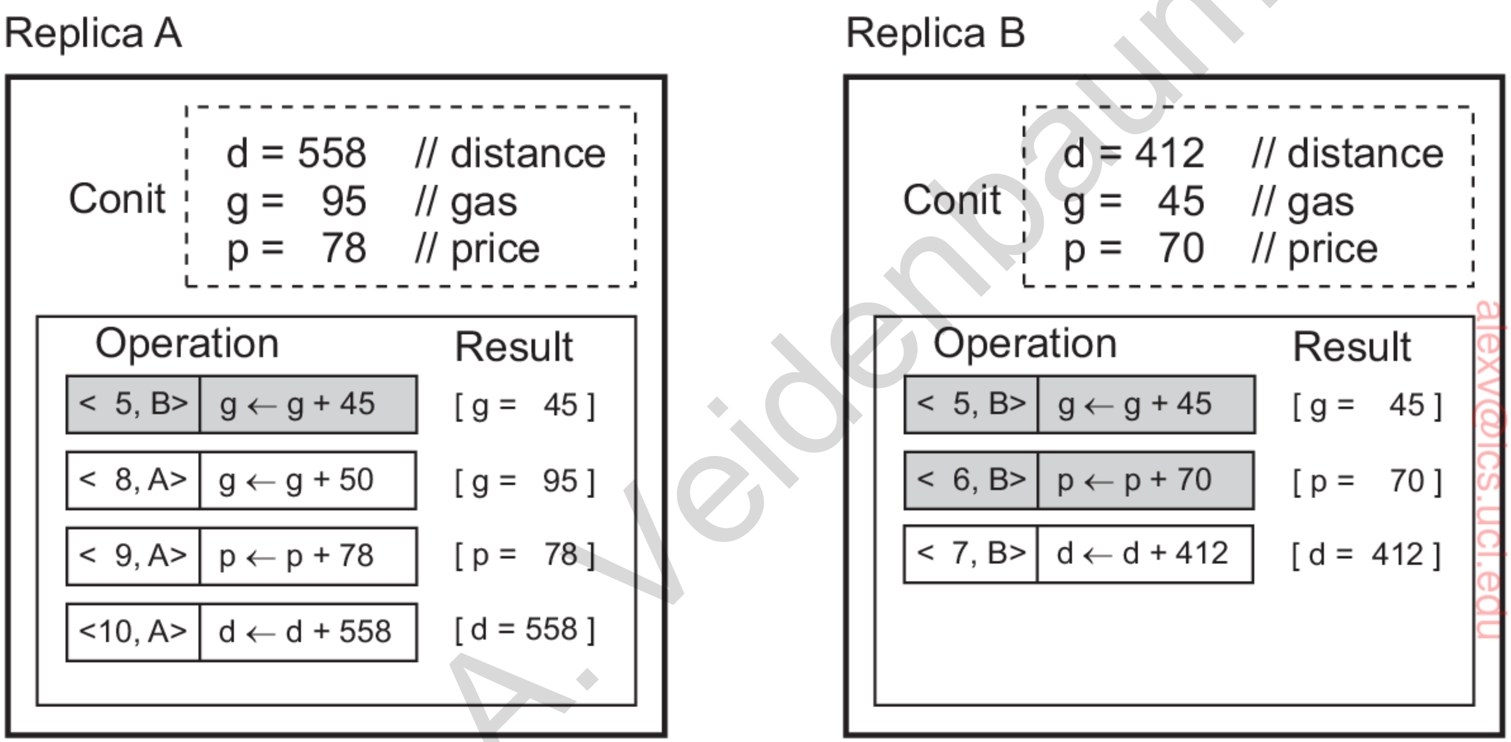
- A contract between processes and the data store
  - If processes obey certain rules the store will work “correctly”
- Intuitively, correctly means a read gets the most recently written data value
  - That of the “last” write
    - Or a well-defined approximation
- What is the last write?
  - Hard to say without a global clock
- A consistency model defines/restricts which (write) values a read can return
- Will look at such models next

Continuous Consistency

- An application defines consistency it can tolerate
- Three types of *inconsistency* can be distinguished
  - A numerical value deviation
    - » Define a tolerance range if data behavior is known
  - "Staleness" between replicas
    - » Replica is several updates behind
      - It may be acceptable to have a slightly older data…
  - Deviation in update ordering
    - » Among different replicas
      - Typically in a local update, while waiting for global agreement
      - Often used with roll-back
- A consistency unit (conit) is a data unit on which consistency is defined – an update unit
  - a byte, a word, a cache line, a page, etc

Example

- **Tracking a fleet of cars**
  - **3-variable conit, initially all values = 0**



Replica A		Replica B	
Vector clock A	= (11, 5)	Vector clock B	= (0, 8)
Order deviation	= 3	Order deviation	= 1
Numerical deviation	= (2, 482)	Numerical deviation	= (3, 686)

- **Shaded values are committed**

Example (2)

- **Order deviation is how many entries are not committed at a node**
  - **Can specify maximum allowable deviation**
- **Can also define a numerical limit on deviation**
  - **What update values have not been seen yet…**
- **How to know the deviation?**
  - **Basically need separate communication**
    - » **This assumes that this is cheaper than synchronizing replicas!**