

CS290i - Lecture 12 Virtual Synchrony & JavaGroups

Scalable Internet Services and Systems, Spring 2001

Thorsten von Eicken
Department of Computer Science
University of California at Santa Barbara

(Many illustrations borrowed from Ken Birman)

Cache Coherence

- † **Problem (from computer architecture):**
 - † Large memories are slow, fast memories are small
 - † Fast machines require memory hierarchies (caches)
 - † Regs, L1 cache, L2 cache, DRAM, disk
 - † Multiprocessors require L1/L2 caches per processor
 - † Threads of the same process (address space) can execute simultaneously on multiple procesors
 - † How can the caches be kept coherent?

2

Invalidation-based proto.

- † **Idea:**
 - † Invalidate all other copies before changing local one
- † **Processor A writes location L**
 - † Broadcast invalidation to all other processors
 - † Wait for ACKs
 - † Write local copy of L
 - † Write-through to next level of memory hierarchy
- † **Performance issues?**
 - † Number of broadcasts
 - † Number of invalidations
- † **Improvements**
 - † Keep track of sharing state of each location

3

Inval.-based proto. (cont.)

- † **Many protocol alternatives**
- † **Example: ISM**
 - † States
 - † I = invalid: "empty cache line"
 - † S = shared: others may have copy: read-only
 - † M = modified: locally modified (exclusive): read-write
 - † Note:
 - † Assumes that all caches see reads and can transition M->S

4

Update-based protocols

† Idea:

- † Update other copies when changing local one
 - † Don't just broadcast invalidation, bcast actual data

† Enhancements

- † Keep track of states
 - † Shared, Owned, Modified
- † Keep track of who has a copy
 - † Only send updates to caches having a copy
 - † Keep track of 0, 1, 2, or ALL copies

5

Relationship to objects

† Example 1: Web session

- † Given a cookie, names user and current "state"
- † Kept only on all web servers, not DB
- † Web server receives HTTP request
 - † Broadcasts to "read & invalidate" session object
 - † Updates local copy
 - † Sends response
- † Problem: fault-tolerance... How can we do better?

† Key differences:

- † HW assumes broadcast bus (except in large scale multiprocessors)
- † HW assumes "next level of memory" (except in COMA)
- † HW does not deal with fault-tolerance
- † HW has tight complexity limits

6

Relationship to objects

† Example 2: Account holdings

- † Holding object has list of stocks held in account
- † Persistent store in DB
- † Web server updates holdings
 - † Invalidates holding obj
 - † Reads from DB, performs update, write DB
 - † Keeps object as "owned"
- † Problems:
 - † How to avoid simultaneous updates
 - † How to avoid reading from DB (read from other node)

7

Group communication

† Extends 1-1 communication to 1-n

- † N servers communicate with each other
- † Servers know who is in the group, i.e. who is "up"

† 1-1:

- † Connection: bidirectional byte stream between two endpoints
- † Socket: handle onto an endpoint
- † Operations: connect, close, send, & receive

† 1-n:

- † Group: multidirectional byte stream between N endpoints
- † Channel: handle onto an endpoint
- † Operations: join, leave, unicast, multicast, & receive

8

Virtual Synchrony

† Problem: ordering of messages

- † Members A and B cast messages a & b “simultaneously”
 - † Member C sees order a , then b , D sees b then a
 - † Member E joins at the same time, sees only a

† Solution

- † Ken Birman, Cornell University
- † Provide message ordering guarantees
 - † E.g. “all members see all messages in the same order”
- † What matters is “causality”
 - † E.g. if b depends on a , then b should arrive after a
 - † Real-time (e.g. a was sent before b) is largely irrelevant

† Problems

- † Performance
- † Definition of causality

9

Logical clocks

† Proposed by Lamport to represent causal order

† Write:

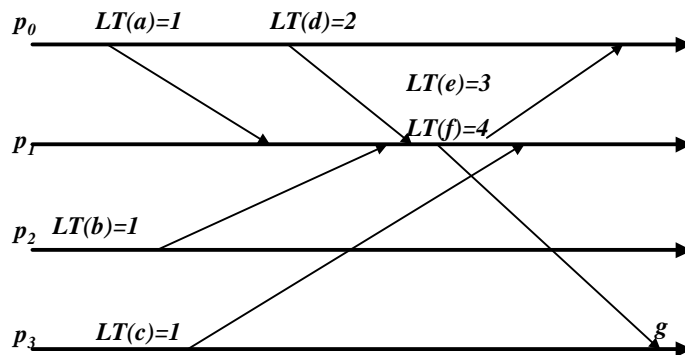
- † $LT(e)$ to denote logical timestamp of an event e
- † $LT(m)$ for a timestamp on a message
- † $LT(p)$ for the timestamp associated with process p

† Algorithm

- † Ensures that if $a \rightarrow b$, then $LT(a) < LT(b)$
- † Each process maintains a counter, $LT(p)$
- † For each event other than message delivery:
set $LT(p) = LT(p) + 1$
- † When sending message m , $LT(m) = LT(p)$
- † When delivering message m to process q :
set $LT(q) = \max(LT(m), LT(q)) + 1$

10

Illustration



11

Concurrent events

† If a, b are concurrent

- † $LT(a)$ and $LT(b)$ may have arbitrary values!

† Thus

- † logical time lets us determine that a potentially happened before b , but not that a definitely did so!

† Example

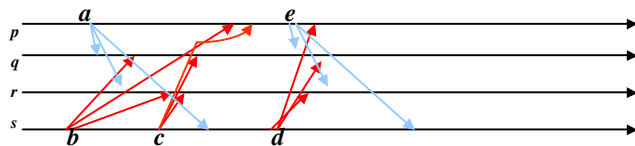
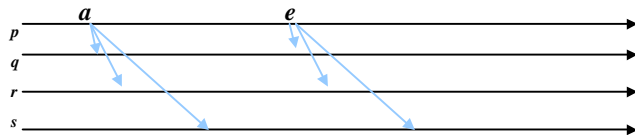
- † Processes p and q never communicate
- † Both will have events 1, 2, ...
- † But even if $LT(e) < LT(e')$ e may not have happened before e'

12

Ordering properties: FIFO

† Fifo or sender ordered multicast: fbcast

- † Messages are delivered in the order they were sent (by any single sender)



delivery of c to p is delayed until after b is delivered

13

Implementing FIFO order

† Basic reliable multicast algorithm has this property

- † Without failures all we need is to run it on FIFO channels (like TCP)
- † With failures need to be careful about the order in which things are done but problem is simple

† Multithreaded applications

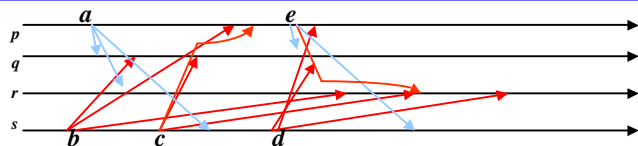
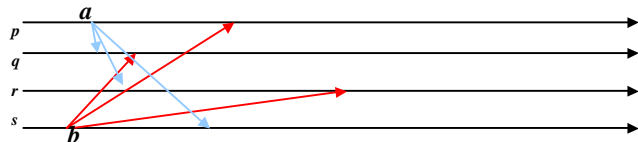
- † must carefully use locking or order can be lost as soon as delivery occurs!

14

Ordering props: Causal

† Causal or happens-before ordering: cbcast

- † If $\text{send}(a) \rightarrow \text{send}(b)$ then $\text{deliver}(a)$ occurs before $\text{deliver}(b)$ at common destinations



*delivery of c to p is delayed until after b is delivered
E causally depends on c
delivery of e to r is delayed until after b&c are delivered*

15

Insights about c/fbcast

† These two primitives are asynchronous:

- † Sender doesn't get blocked and can deliver a copy to itself without "stopping" to learn a safe delivery order
- † If used this way, the multicast can seem to sit in the output buffers a long time, leading to surprising behavior
- † But this also gives the system a chance to concatenate multiple small messages into one larger one.

16

Distrib. State Machines

† Idea:

- † Replicated object or service that advances through a series of “state machine transitions”
- † All replicas keep the “current state”
- † Events causing state transitions are multicast to all members
- † All members “apply” the event to the local state machine copy

† Requirement

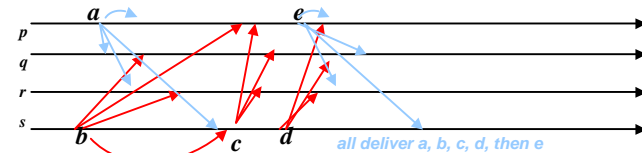
- † Clearly all copies need to make the same transitions
- † Leads to a need for totally ordered multicast

17

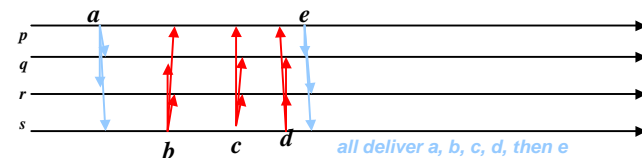
Ordering properties: Total

† Total or locally total multicast: abcast

- † Messages are delivered in same order to all recipients (including the sender)



- † Can be visualized as:



18

Implementing Total Order

† Many ways have been proposed

- † Just have a token that moves around
 - † Token has a sequence number
 - † When you hold the token you can send the next burst of multicasts
- † Extends to a cbcast-based version
 - † Use this when there are multiple concurrent threads sending messages
- † Send all messages via serializing coordinator
 - † Requires extra hop and doesn't scale well

19

How to think about order?

† Usually, we think in terms of state machines and total order

† But all the total order approaches are costly

- † There is always someone who may need to wait for a token or may need to delay delivery
- † Loses benefit of asynchronous execution
- † Could be several orders of magnitude slower!

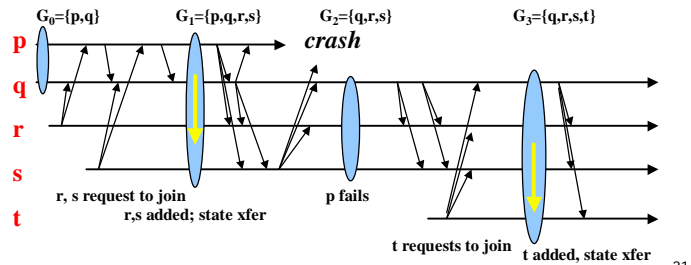
† So often we prefer to find sneaky ways to use fbcast or cbcast instead

20

Membership changes

† Virtual synchrony

- † Synchronizes membership change with multicasts
- † Idea: between any pair of successive group membership views, the same set of multicasts are delivered to all members
- † Makes algorithms much simpler



21

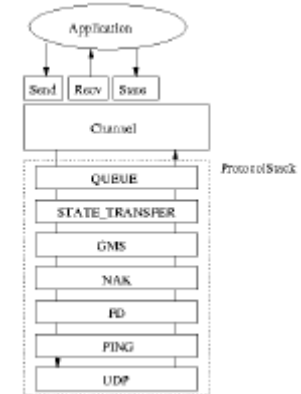
JavaGroups

† Goals

- † Implement group communication in Java
- † Implement virtual synchrony
- † Implement common design patterns

† Architecture:

- † Groups, Channels
- † Messages, Views
- † Protocol stacks



22

Channel class

```

† public abstract class Channel implements Transport {
    public void Connect(Object group_address) throws ChannelClosed;
    public void Disconnect();
    public void Close();
    public void Send(Message msg)
        throws ChannelNotConnected, ChannelClosed;
    public Object Receive(long timeout)
        throws ChannelNotConnected, ChannelClosed, Timeout;
    public Object Peek(long timeout)
        throws ChannelNotConnected, ChannelClosed, Timeout;

    public View GetView();
    public Object GetLocalAddress();
    public Object GetGroupAddress();
    public void SetOpt(int option, Object value);
    public Object GetOpt(int option);
    public void BlockOk();
    public boolean GetState(long timeout);
    public boolean GetAllStates(long timeout);
    public void ReturnState(Serializable state) {}
}
    
```

23

Receive method

† Return value: Object

- † Message
 - † Data sent by other group member (or by self)
- † View
 - † New membership view
 - † View orders all members consistently globally
- † SuspectEvent
 - † Some member seems to be dead
- † BlockEvent
 - † Stop sending so new view can be installed
- † GetStateEvent, SetStateEvent
 - † Used for state transfer

24

Some Design Patterns

† RequestCorrelator

- † Assign ID to requests
- † Match responses to requests
 - † Multiple outstanding requests
 - † Multiple responses to each request

† Multicast RPC

- † One-way, get first response, get N/all responses

† Distributed Hashtable

- † Local changes to hashtable get reflected everywhere
- † Total ordering of changes

25

JavaGroups

† Open source

- † <http://Javagroups.sourceforge.net>
- † Use JChannel implementation

26