Isis and Virtual Synchrony

In the next few slides we look at the sorts of group arrangements supported by ISIS, an implementation of virtual synchrony.
In a peer group, processes cooperate to get something done. Though they all are kept up to date on one another’s state, each process might be doing different things. However, if one fails, any of the others can take over for it.
This is the model used in a previous version of PuddleStore: a peer group of servers receives requests from a potentially large number of clients.
In diffusion groups, servers multicast messages to each other and to a set of clients. The clients are strictly passive and perhaps serve to display data.
Hierarchical groups facilitate large groups of servers. Sub groups are organized in a tree. The organization essentially makes up for shortcomings of the multicast protocol.
The concerns here are actually orthogonal to those of reliable multicasts. (Though for synchronous group communication to work, multicasts must be reliable.)
Virtually Synchronous Group Communication

• Each multicast is “sort of” atomic
  – all parties receive it, or:
    - if the sender fails, then either all parties receive the multicast or none do
    – the sending and receiving of a multicast are separate events (subject to causal ordering constraints)
    – no process may join a group between the sending and the receiving of a multicast
  • Multicasts may be specified to be totally and causally ordered, or just causally ordered
    – it’s up to the sender
CBCAST is the name given to the multicast protocol used for virtual synchrony with just causal ordering. Here each process is modifying a variable that’s only updated in that particular process (though replicated in all processes). These events are clearly concurrent and thus CBCAST can be safely used.
ABCAST is the name given to the multicast protocol used for virtual synchrony with total and causal ordering. Here each process in the group is inserting an item into a queue (that’s replicated in all processes). If these operations were just causally (and not totally) ordered, each process might have a different idea of the order of items in the queue. Thus ABCAST would be used in this case.
VC(a) is process a’s vector clock. VC(m) is the vector-clock timestamp associated with message m.
Each message is assigned (and carries) a unique ID so that it can be identified in a sets-order message.
Implementing ABCAST (2)

– receivers of sets-order message queue it in causal order (in their delay queues)
– when all causally preceding messages have been delivered
  - receivers deliver ABCAST messages in order given by sets-order message
Multicast Stability

• A multicast is stable if it's been received by all intended recipients
• Assuming reliable transport, sender knows when multicast is stable
• Would like all other participants to know as well
• \(stable(p_a) = \text{largest } n \text{ such that: all } m \text{ from } p_a \text{ with } VC(m)[a] \leq n \text{ are stable}\)
• Each process includes current stable value with each message sent
We first look at processor failures, euphemistically known as “view changes.”
Formally ...

- $G = \text{set of processes potentially in multicast group}$
- $\text{view}_a \subseteq G$, for $a = 1, 2, 3, \ldots$
- $\text{view}_a \cap \text{view}_{a+1} = \text{view}_a \text{ or } \text{view}_{a+1}$
- $|\text{view}_a| - |\text{view}_{a+1}| = 1 \text{ or } -1$
  - group memberships change by just one process at a time
- Assume (for now) that an oracle notifies all remaining participants that the view has changed
Consequences of View Change 1

- A multicast might be disrupted
  - process sends message to some, but crashes before it can send message to the rest
- Solution
  - on notification of view change:
    - for all non-stable multicasts from non-extant processes
      - forward message to all others
        - (processes must ignore duplicates)
In this example, P1 starts a multicast, but crashes after sending the message to P2; thus the message is not sent to the other processes in the group. On receiving notification of the new view, P2 realizes it has a non-stable multicast from the previous view, sent by a now non-extant process. So it forwards it to the others. In the meantime, P3 does another multicast.
Virtual synchrony makes the previous slide have the effect shown here.
Consequences of View Change 2

- Vector clocks get messed up
  - what about components corresponding to non-extant processes?
  - what about adding new components?
- Solution
  - must restart vector-clock time, with new (shorter or longer) vectors
    - must first deal with all messages with old time stamps
    - (messages must be tagged with which view they’re from)
Flush messages are used to make certain that all processing from the previous view has been completed before processing starts for the new view. This very much depends on the communication channels’ being reliable and FIFO.
In this example, P1 crashes in the middle of a multicast. The remaining processes are notified of the new view. P4 sends flush messages to the others. P2 notices that it has received a non-stable multicast from the previous view, so it forwards it on to the others before sending its own flush messages. Process 3 receives the forwarded cbcast1 as well as flush messages from the others. After sending its own flush, it can “install” view 2 and thus be able to send a new cbcast.
Example: Virtual
Consequences of View Change 3

- Some flush messages don’t get sent
  - another process has crashed: another view change

- Solution
  - confirmation of view a+2 installs both a+1 and a+2
    - non-stable messages of both views are forwarded
    - inhibit_send counter is decremented twice

- Extended solution
  - Flush messages may be after successive view changes
This example is similar to the previous one, except that P2 crashes before it can complete the forwarding of cbcast1. P3 can't send a new multicast within view 2, since it hasn't received flushes from all its members (and thus can't install view 2). When view 3 is announced, P3 notices that it has an unstable multicast in its cache, so it forwards it to the other remaining group member. Once P3 receives (and sends) a flush, it can do its multicast.
Example: Virtual

P1

View 1

cbcast 1

P2

View 2

P3

View 3

cbcast 2
How do we deal with processes that are members of multiple groups? Each process must have one vector clock per group so as to keep track of how many events have taken place with respect to each group. Otherwise, when a message arrives carrying a vector time stamp, the receiving process would not be able to determine whether the counts of events in the time stamp refer just to events of process groups the receiver belongs or to other process groups as well.

So, it makes sense for processes to have one vector clock for each group they are in, maintaining them independently. When they multicast a message, the message carries the sender’s vector timestamp for the group being multicast to.

However, suppose P1 multicasts a message to group A, and then multicasts a message to group B. P2, a member of both groups, receives both multicasts. Though each multicast message contains a vector time stamp for the group it was sent to, P2 will not have enough information to determine which was sent first. Since both were sent by P1, there is a clear causal ordering, but it is undeterminable by P2. So, for processes such as P2 to determine the correct orderings, messages must carry time stamps for all process groups.
Coping

- $VC_r(p)[i] = i^{th}$ component of process $p$'s vector clock from group $r$
- Each message carries time stamps of all groups
  - process $b$ receiving $m$ from $a$, sent in group $r$:
    - delays delivery of $m$ until
      - $VC_r(m)[a] = VC_r(b)[a]+1$
      - $(\forall$ processes $p \in r \land p \neq a)$:
        $VC_r(m)[p] \leq VC_r(b)[p]$
      - $(\forall$ groups $r \in G_b): VC_r(m) \leq VC_r(b)$
        - $G_b$ is the set of groups process $b$ is in
Successful Failure

- How are views maintained?
- How can it be determined that a process has failed?
  - NFS approach: when it restarts
  - ostracism
    - come to agreement that a process is down
    - even if it isn’t, it must reapply for membership
- What about network partitions?
  - must have quorum of potential group membership to continue functioning
In the Swiss Exchange (SWX, left), the core servers are configured using a primary-backup scheme, as are the hub servers. All updates to the exchange state are fully replicated by reliably multicasting each event to the full set of client systems. In the New York Stock Exchange (NYSE, right), the core servers back one-another up, but each handles a share of the load, and similarly for hub servers. In the NYSE, a given data item is forwarded only to those display systems which subscribe to the item, normally a very small subset of the total number of displays. A consequence is that the SWX is best understood as a tree of process groups, while the NYSE is best understood as a set of three large process group (in fact, both systems are somewhat more complex, but we omit details for brevity). In the SWX, reliable multicast is used to deliver quotes directly to the leaf nodes. In the NYSE, quotes flow over a point-to-point forwarding architecture, and reliable multicast is employed mostly to track the topology of the tree, reconfigure after failure, and to track interests of the leaf nodes (which change over time).

The slide is Figure 1 from “A Review of Experiences with Reliable Multicast” by K. Birman, Software Practice and Experience, Vol. 29, Issue 9, 1999. The quoted text is the caption of this figure.
“Virtual synchrony in PHIDEAS. The leader is responsible for connections to the database. Communication includes multicasts of events affecting the cluster state and state transfer to a joining (recovering) machine. Membership changes are reported by the reliable multicast technology to the application layer. As processes join, the state of the system is transferred to them.”

The slide is Figure 3 from “A Review of Experiences with Reliable Multicast” by K. Birman, Software Practice and Experience, Vol. 29, Issue 9, 1999. The quoted text is the caption of this figure. Replication is done for each “ATC sector.”