Isis and Virtual Synchrony
Reliability through Process Groups

- Processes (on different machines) form a group
- Keep in sync with one another
  - communicate via multicast
Peer Groups
Client/Server Groups

S1, S2, S3, S4

C1, C2, C3
Diffusion Groups

S1 → C1 → C2 → C3 → C4 → C5

S2 → C1 → C2 → C3 → C4 → C5
Synchronous Group Communication

• Each multicast is atomic
  – all parties receive it, or none of them do
  – the sending and receiving of a multicast is a single event: no events happen in between
  – a process’s leaving or joining a group is an event
    - process failure = leaving the group

• Multicasts are totally and causally ordered
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
Example: CBCAST

\[
\begin{align*}
w++ & \\
x++ & \\
y++ & \\
z++ & 
\end{align*}
\]
Example: ABCAST

ins_queue(w)
ins_queue(x)
ins_queue(y)
ins_queue(z)
Implementing CBCAST

• Vector clocks are used
  – process a sending message \( m \):
    - ++VC(a)[a]; VC(m) = VC(a)
      • each message contains its sender’s updated vector timestamp
  – process b receiving m from a:
    - delays delivery of m until
      • for all processes \( k \)
        – VC(m)[k] = VC(b)[k]+1 if \( k == a \)
        – VC(m)[k] \( \leq \) VC(b)[k] otherwise
    - after delivery
      • VC(b) is updated using VC(m)
Implementing ABCAST (1)

• One process-group member is elected the leader

• Process i sending message m
  – m is multicast using CBCAST, but message is marked “do not deliver”
    - receivers hold it in their delay queues
  – receivers deliver all causally preceding messages
  – leader delivers m when causally ok
    - keeps track of the order in which such messages are delivered
    - when convenient, CBCASTs “sets-order” message containing this order
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 multicasts $m$ from $p_a$ with $VC(m)[a] \leq n$
  are stable
• Each process includes current stable value with each message sent
Change of View

Server

Server

Server
Formally ...

- $G =$ set of processes potentially in multicast group
- $\text{view}_a \subseteq G$, for $a = 1, 2, 3, \ldots$
- $\text{view}_a \cap \text{view}_{a+1} = \text{view}_a$ or $\text{view}_{a+1}$
- $|\text{view}_a| - |\text{view}_{a+1}| = 1$ 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)
Example: Real

- cbcast1
- P1
- V
- e
- w
- View
- 1
- forward cbcast1
- View
- 2
- P2
- P3
- P4
- cbcast2
Example: Virtual

P1 -> View 1
P2
P3
P4

View 1 -> View 2

cbcast1

View 2

cbcast2
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)
Flushed Messages

- Process is currently in view a
- On notification of view change (to a+1)
  - increments $inhibit_{send}$ counter
    - may not send new messages till counter is back to 0
  - process forwards all unstable messages from non-extant processes of view a to all processes in view a+1
    - marks such messages as stable
  - sends $flush$ a+1 messages to all processes in view a+1
  - on receipt of $flush$ a+1 messages from all others in view a+1
    - “installs” view a+1
    - decrements $inhibit_{send}$ counter
Example: Real

P1

cbcast1

P2

forward
cbcast1

P3

P4

V iew 1

V iew 2

-- flush message --
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
Example: Real

P1: cbcast1

P2: View 1 → View 2 (forward cbcast1)

P3: View 2 → View 3 (forward cbcast1)

P4: cbcast2

flush message
Example: Virtual

P1

V
i
e
w
1

V
i
e
w
2

V
i
e
w
3

P2

P3

P4

cbcast1

cbcasct2
Coping

• \( VC_r(p)[i] = \text{ith component of process } p \text{'s vector clock from group } r \)

• Each message carries time stamps of all groups
  – process \( b \) receiving \( m \) from \( a \), sent in group \( r \):
    - delays delivery of \( m \) until
      • \( VC_r(m)[a] = VC_r(b)[a]+1 \)
      • \((\forall \text{ processes } p \in r \land p \neq a)\):
        \( VC_r(m)[p] \leq VC_r(b)[p] \)
      • \((\forall \text{ groups } r \in G_b):VC_r(m) \leq VC_r(b) \)
        • \( G_b \) 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
ISIS Success Stories

- Swiss Exchange (SWX)
- New York Stock Exchange (NYSE)
Another Success Story

• French air traffic control (ATC)
  – PHIDEAS project