Atomic Broadcast

- Broadcast
  - A message sent to all processes/nodes.
- Multicast
  - A message sent to all in a group, a multicast group.
- **All** or **none** nodes in the group should get the message.
- Two broadcast messages sent from the same or different nodes should be delivered in the same order to all the nodes (in the group).

Atomic Broadcast implementation

- Two main principles for implementation:
  - asynchronous
    - can be expensive (many messages) and/or slow.
  - synchronous
    - fast
    - requires synchronized clocks within the nodes.

Asynchronous algorithms

- Virtual Ring algorithms
  - Chang-Maxemchuck protocol
- ISIS system
  - ABCAST
  - CBCAST
  - GBCAST

Chang-Maxemchuck Protocol

Chang, Maxemchuck 1984

- uses a virtual ring and a *token* message to achieve atomic broadcast.
- The node that holds the token message can give its broadcast message a unique timestamp that then can be added to the token message before it is sent further on the ring.
- When the other nodes gets the message they acknowledge it in the token message.
- When the sender gets the token message back it can check that it is acknowledged by all others.
  if so it puts a confirmation for it in the token message, otherwise a reject.
- When the other nodes get the token message again and sees a confirmed message information the corresponding message can be delivered.
- To deliver a broadcast message the sender must wait for the token message, then let it pass two rounds on the ring before the message is delivered.
  - This makes the algorithm slow.
  - but it uses little network resources compared to other algorithms.

ABCAST

Birman & Joseph 1987

- Does **not** require FIFO!
- but the network must be safe, i.e. no messages are lost.
- Each node must have a unique name.
- Each node has a queue for broadcasts waiting to be delivered.
  - These messages are marked as pending or ready.
  - Each broadcast message will get a
    - unique identifier, *id*, as defined below, totally ordered.
    - unique global timestamp, *gt*, as defined below.

ABCAST — Sending node

- Sending node:

1. When a process wants to send a broadcast message, the message is sent to all receivers.
   it is given a timestamp that is higher than all (by the sender) known timestamps.
   This timestamp together with the sender's node's unique name will be the unique identifier, *id*, for the message.
   The message is inserted into the local message queue of not delivered messages in timestamp order (this means that it must be put at the end!). The message is marked pending.

2. When the node has got an acknowledgement with a global timestamp proposal for a given message from all the other nodes the maximum of those is chosen as global timestamp for the message.
   In the local queue the message's timestamp is updated to the chosen timestamp, the message is marked ready and the queue is sorted in timestamp order. If two messages have the same timestamp they will be ordered according to the senders identifier.
   A confirmation message giving information on the global timestamp is sent to all other nodes.

ABCAST — Receiving node

- Receiving node:

3. When receiving a broadcast message it is given a timestamp that is higher than any in the nodes message queue and not lower than the timestamp given by the sender.
   Then it is put last in the local broadcast message queue marked pending.
   Then an acknowledgement message with the proposed timestamp is sent to the originator of the message.

4. When receiving a confirmation message giving information on the global timestamp the corresponding message is given that timestamp and be marked ready.
   Then the queue is sorted.

ABCAST — All participating nodes

5. When the first message in the queue is marked ready it should be delivered to the receiving process and then erased from the queue.

6. If the first message in the queue is marked pending, no messages should be delivered even if they are marked ready.

CBCAST

- "cheaper" protocol
- Guarantees that all nodes gets two (or more) *broadcast* messages in the same order if their initiating can be ordered according to the partial order "$\rightarrow$".
  i.e. if a broadcast message can have any influence on another.
- Avoids sorting broadcast messages that are independent of each other.
- Implementation:
  - Add a list to the broadcast message of which events (broadcast messages) that it might be dependant on.

GBCAST

- Protocol for deciding which nodes belong to a multicast group at each time.
- ABCAST and CBCAST (as well as other atomic broadcast protocols) must know which nodes that participate in the group.
  - If one node crashes or become isolated from the others it will not be possible to deliver broadcasts.
  - If the nodes recognize this, e.g. using time-out, the failing node can be excluded from the group.
  - Then the nodes can continue to send atomic broadcasts.
- It is important to know exactly between which two broadcasts the group changed in order to know which nodes get which broadcasts, i.e. to keep a consistent system.
- When a node wants to join the group again it must also be done between to identified broadcasts.
- This protocol is important to achieve fault-tolerance to the system.
  - Otherwise the use of atomic broadcasts can lead to low availability.

Fault tolerant Atomic Broadcast Synchronous Algorithm

Cristian 1989

- Designed for the North American Air Control System commissioned by IBM.
- Uses the Client-Server clock synchronization algorithm.
- G is a network with n nodes and m links.
  - Each node has a physical clock.
  - Node p's clock is denoted by C_p
  - $C_p(t)$ denotes the clock value at the real time t.
- Assumptions:
  1. The nodes have a unique names that are totally ordered.
  2. F is the set of nodes with links that got failures during the atomic broadcast.

     G-F is the sub net of G where links and nodes work correctly.
     Assume that G-F is connected,
     i.e. the protocol does not allow a network partition.

3. Each clock is going forward.
   Two read operations on the same clock should give different values.
   i.e. a clock is not allowed to stop or to be to slow.

   Also the clocks should be synchronized in such a way that there is a small value ε such that for every real point of time t it should hold:
   $\forall\, p,q \in G\text{-}F: |\, C_p(t) - C_q(t)\,| < \varepsilon$.
4. There is a maximal time for sending and treating the messages that the protocol uses, δ.

   Assume
   p and q are two correctly working neighbor nodes connected with a correct link
   r is an arbitrary node.
   p sends a message at the real time u
   q receives the same message at the real time v

   Then it should hold that

   $$0 < C_r(v) - C_r(u) < \delta$$

We also assume that the operating system offers the operation:

`schedule(A,T,p)`, which implies that `A` will be executed at time `T` with parameters `p`.

Protocol that survives "omission failure"

- The protocol uses flooding.
  - The flooding is interrupted at each node when a message arrives a second time. (the first technique we described for flooding termination)
- The message format is: `(T,s,σ)`
  - *T* is a timestamp that gives the initiating time,
  - s is the senders unique name,
  - σ is the information that should be delivered (the broadcast message).
- The messages will get a unique identity: (*T*,s)

- The received messages are placed in a log *H* (queue of broadcast messages waiting to be delivered)
  - When there is no faults in the network a message will be received within the time $d\delta$ after it was sent from the originator
    d is the network diameter
    $d\delta$ will then be the maximal message jump times the maximal handling and transmission time.
  - The messages are time stamped with the sending time *T*.
    When the clock in the receiver is $T + d\delta + \varepsilon$ there can not arrive a message from any nod with a lower timestamp than *T*.
    ε is the maximal error among the local clocks.
  - For *omission failure* the time limit has to be extended to $T + \pi\delta + d\delta + \varepsilon$.
    Failures may increase the network diameter to $\pi + d$.

$\Delta \equiv \pi\delta + d\delta + \varepsilon$ is the protocol terminating time


```
task Start;
 const Δ = (π + d)δ + ε;
 const myid:node_Name = .....;
 var T:Time; σ:Text;
  loop
   take(σ);
   T ← clock
   send_all(T,myid,σ);
   H ← H ⊕ (T,myid,σ);
   schedule(End,T+Δ,T);
  end loop;
```

```
task Relay;
 const Δ = (π + d)δ + ε;
 var U,T:Time; σ:Text; s:node_Name;
  loop
   receive((T,s,σ),l);
   U ← clock;
   [U≥ T+Δ: "late message" iterate];
   [T∈ dom(H) & s ∈ dom(H(T)): "deja vu" iterate];
   send_all_but(l,(T,s,σ));
   H ← H ⊕ (T,s,σ);
   schedule(End,T+Δ,T);
  end loop;
```



```
task End(T:Time);
 var p:node_Name; val:node_Name → Text;
  val ← H(T);
  while dom(val) ≠ {}
     p ← min(dom(val));
     deliver(val(p));
     val ← val\p;
  end while;
  H ← H\T;
```