Today: Classical Problems in Distributed Systems

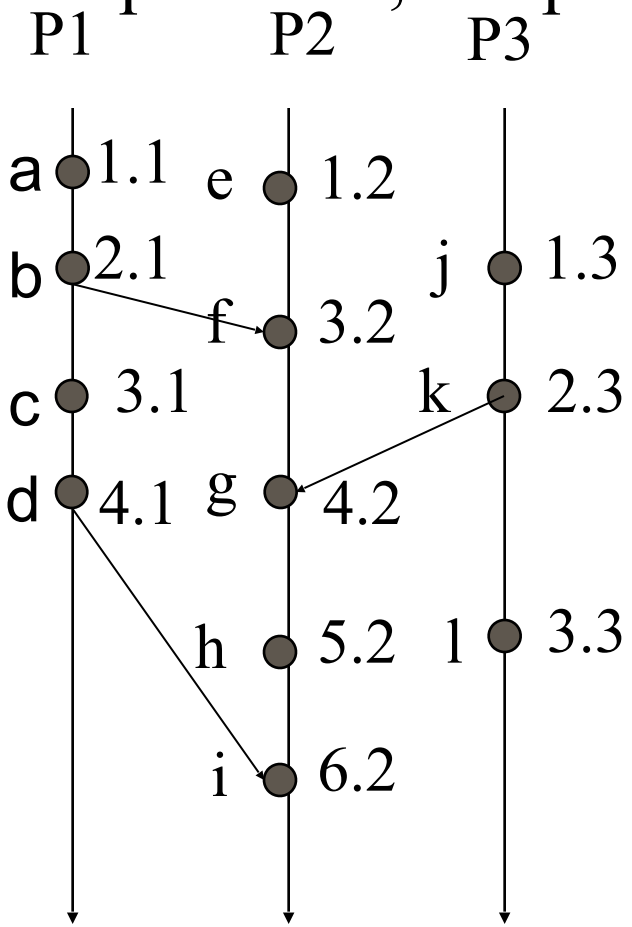
- More logical time

Next few classes:

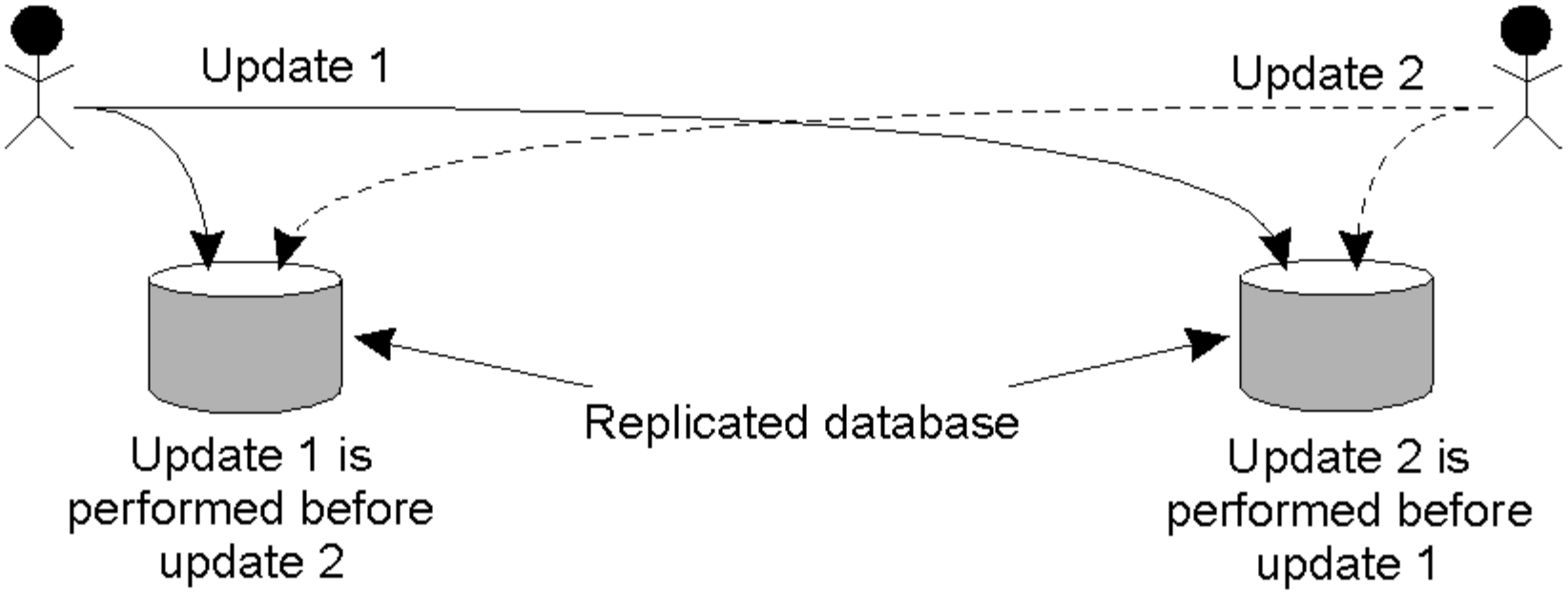
- Leader election
- Mutual exclusion
- Distributed transactions
- Deadlock detection
- CAP Theorem

Total Order

- Create total order by attaching process number to an event. If time stamps match, use process # to order



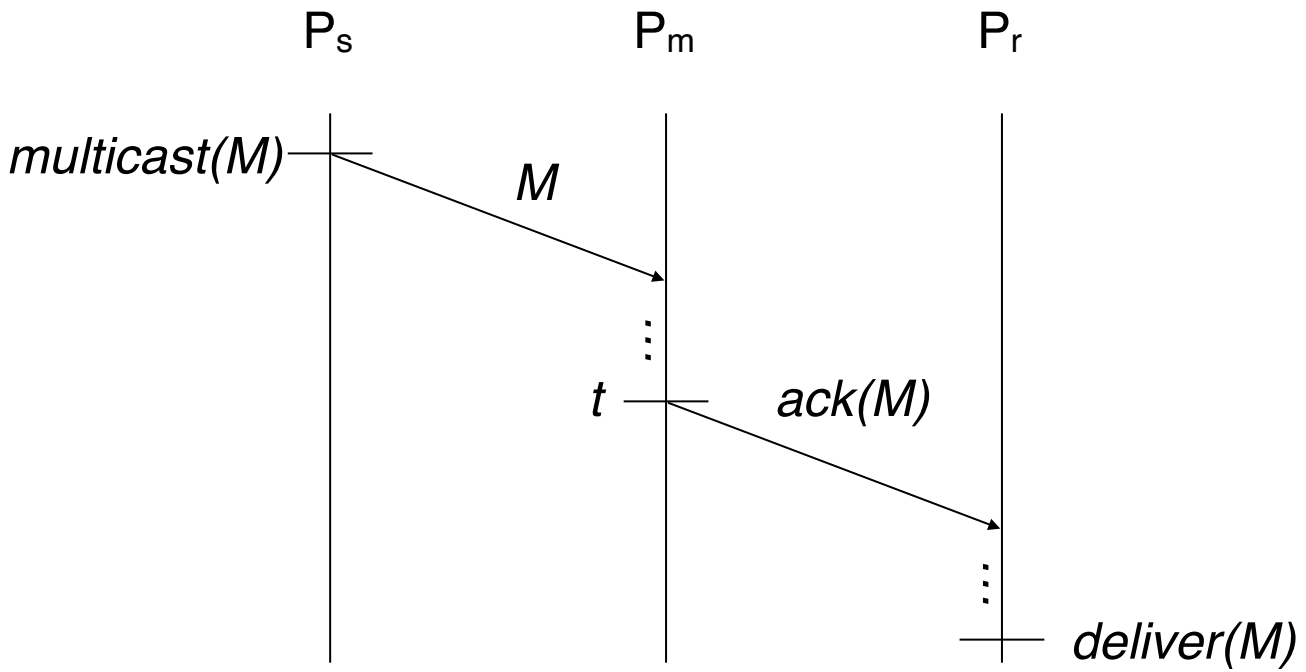
Example: Totally-Ordered Multicasting



Totally-Ordered Multicast Algorithm

- Assumptions
  - No failures
  - FIFO channels
- Algorithm
  - Multicast messages have timestamp equal to LC at sender
  - When receive a msg, put it in a queue sorted by its timestamp and send an ack to all
  - If head message in queue is acked by everyone, deliver

Why Does It Work?



- When P_r delivers M, it must be sure that no other message should be delivered before M
- P_r knows because for each process P_m
  - P_r has already received all msgs sent by P_m before t (FIFO)
  - P_r knows that all msgs P_m sends after t have timestamp > $C(M)$

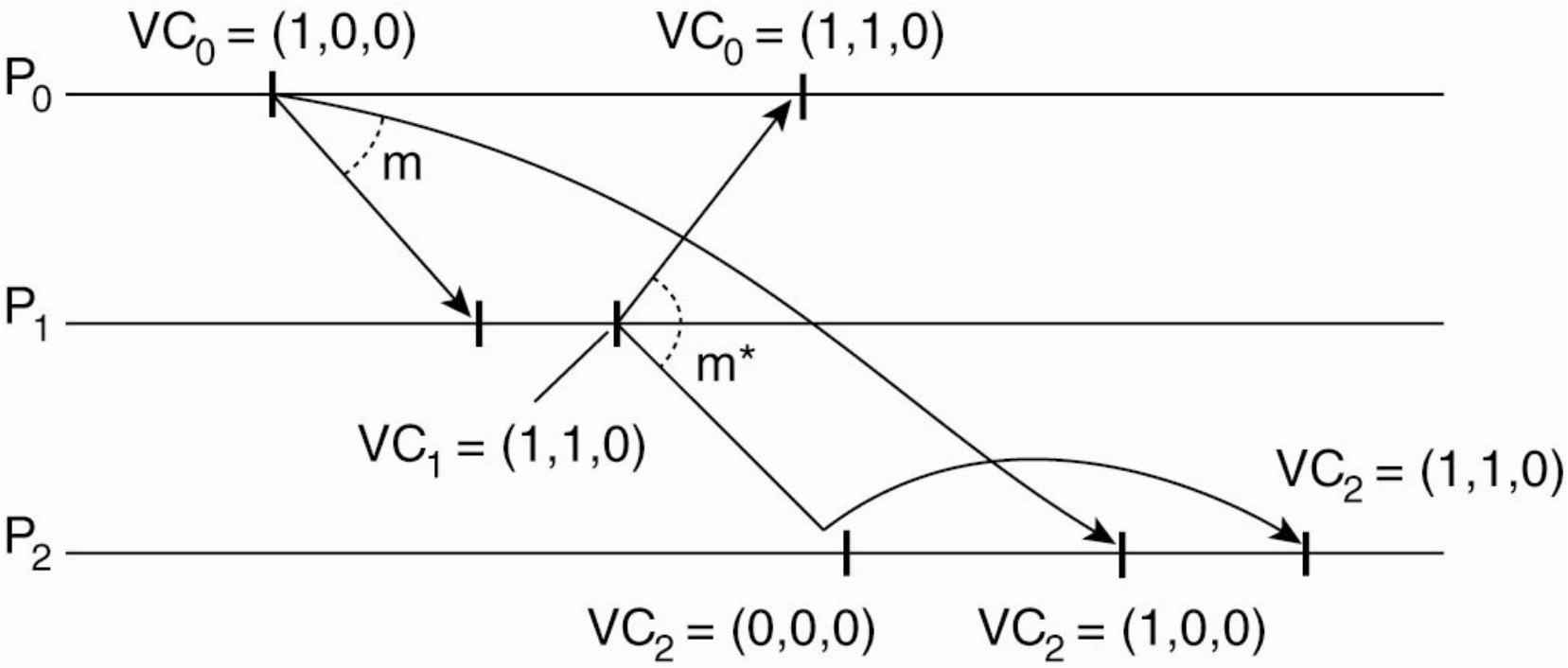
Causality

- Lamport's logical clocks
  - If *A* -> *B* then *C(A)* < *C(B)*
  - Reverse is not true!!
    - Nothing can be said about events by comparing time-stamps!
    - If *C(A)* < *C(B)*, then ??
- Need to maintain *causality*
  - If a -> b then a is casually related to b
  - *Causal delivery*:If send(m) -> send(n) => deliver(m) -> deliver(n)
  - Capture causal relationships between groups of processes
  - Need a time-stamping mechanism such that:
    - If *T(A)* < *T(B)* then *A* should have causally preceded *B*

Vector Clocks

- Each process *i* maintains a vector V_i
  - $V_i[i]$: number of events that have occurred at *i*
  - $V_i[j]$: number of events *i* knows have occurred at process *j*
- Update vector clocks as follows
  - Local event: increment $V_i[i]$
  - Send a message: piggyback entire vector V
  - *j* receives a message from *i*: $V_j[k] = \max(V_j[k], V_i[k])$ for all *k*
    - *j* is told how many events at each other process happened before the receipt event
    - *j* assigns $V_j[j] = V_j[j]+1$ to the receipt event
- *Exercise:* prove that if $V(A)<V(B)$, then *A* causally precedes *B* and the other way around.

Enforcing Causal Communication

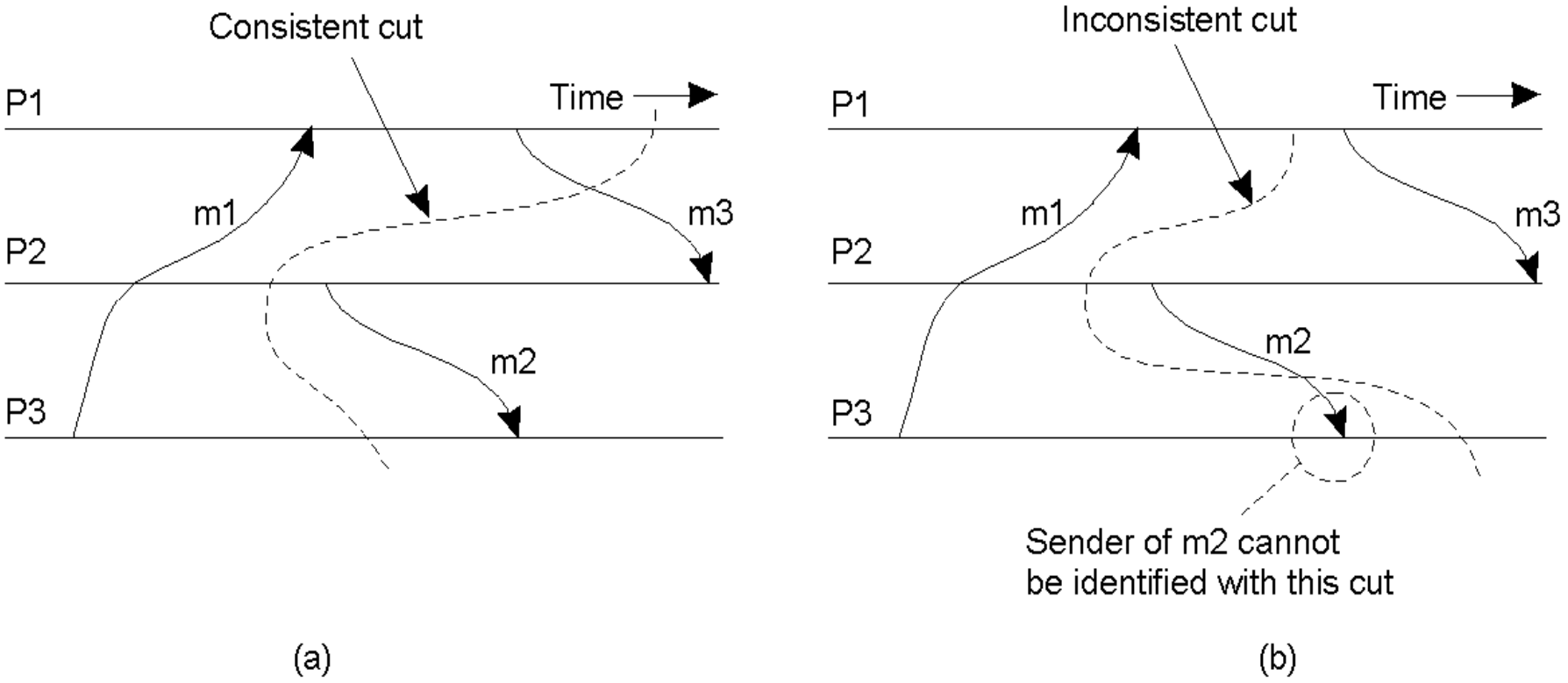


- Conditions for delivery on message sent from Pi to Pj
  - ts(m)[i] = VCj[i] + 1 (this is the next message sent from Pi)
  - ts(m)[k] <= VCj[k] for all other k (Pj has received full context)

Global State

- Global state of a distributed system
  - Local state of each process
  - Messages sent but not received (state of the queues)
- Many applications need to know the state of the system
  - Failure recovery, distributed deadlock detection
- Problem: how can you figure out the state of a distributed system?
  - Each process is independent
  - No global clock or synchronization
- Distributed snapshot: a consistent global state

Global State (1)



a) A consistent cut

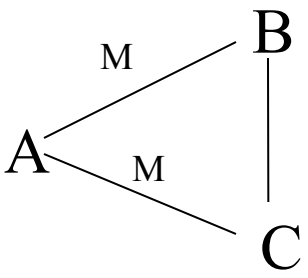
b) An inconsistent cut

Distributed Snapshot Algorithm

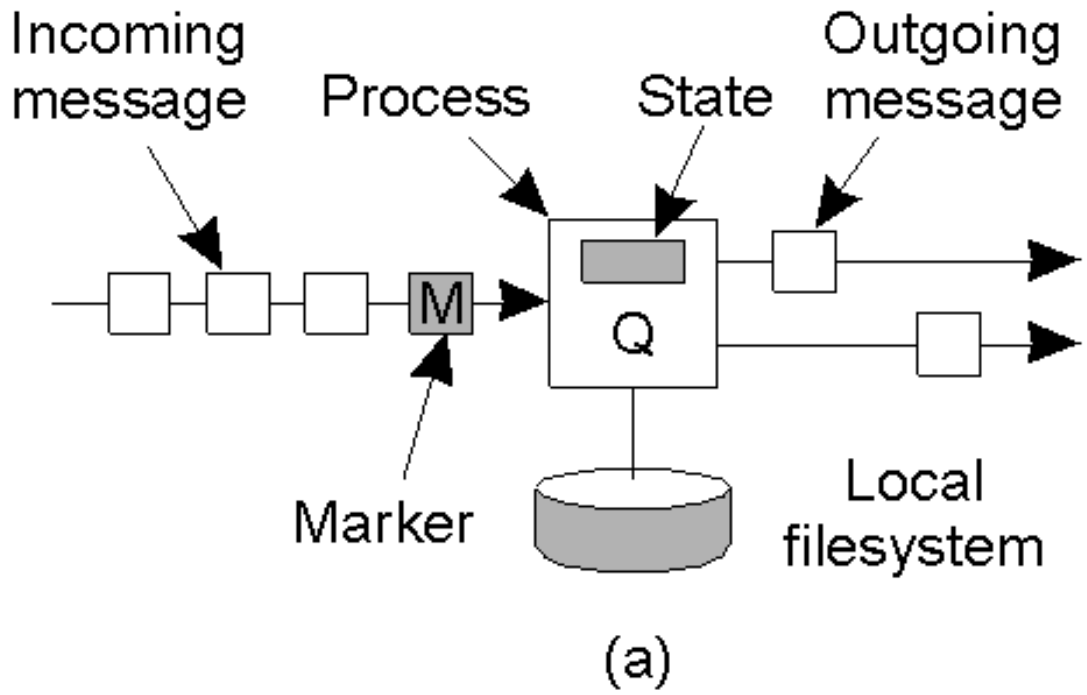
- Assume each process communicates with another process using unidirectional point-to-point channels (e.g, TCP connections)
- Any process can initiate the algorithm
  - Checkpoint local state
  - Send marker on every outgoing channel
- On receiving a marker
  - Checkpoint state if first marker and send marker on outgoing channels, save messages on all other channels until:
  - Subsequent marker on a channel: stop saving state for that channel

Distributed Snapshot

- A process finishes when
  - It receives a marker on each incoming channel and processes them all
  - State: local state plus state of all channels
  - Send state to initiator
- Any process can initiate snapshot
  - Multiple snapshots may be in progress
    - Each is separate, and each is distinguished by tagging the marker with the initiator ID (and sequence number)

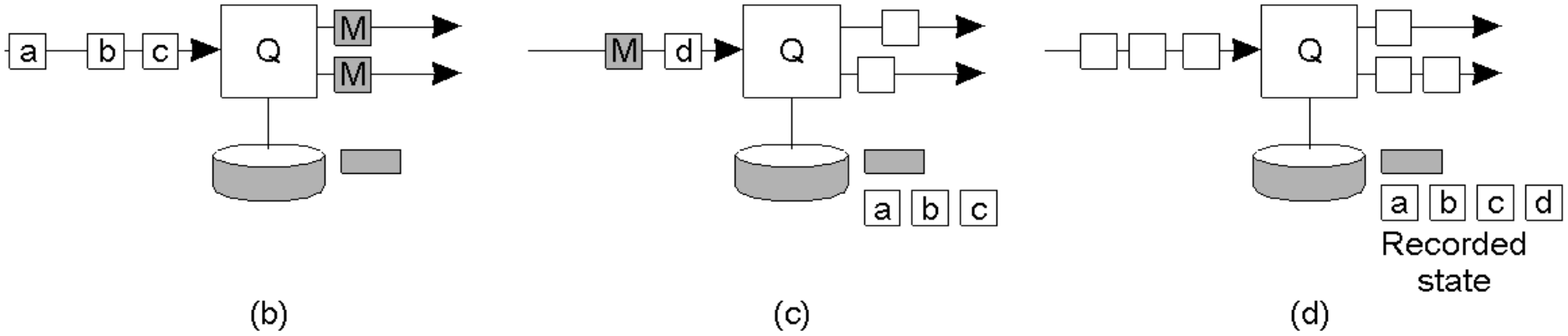


Snapshot Algorithm Example



a) Organization of a process and channels for a distributed snapshot

Snapshot Algorithm Example



b) Process Q receives a marker for the first time and records its local state

c) Q records all incoming message

d) *Q* receives a marker for its incoming channel and finishes recording the state of the incoming channel