This and upcoming lectures?

- We'll focus on concepts relating to ***time***
  - Time as it can be "used" in systems
  - Systems that present behaviors best understood in terms of temporal models (notably the transactional model)
  - Event ordering used to ensure consistency in distributed systems (multicasts that update replicated data or program state)

What time is it?

- In distributed system we need practical ways to deal with time
  - E.g. we may need to agree that update A occurred before update B
  - Or offer a “lease” on a resource that expires at time 10:10.0150
  - Or *guarantee* that a time critical event will reach all interested parties within 100ms

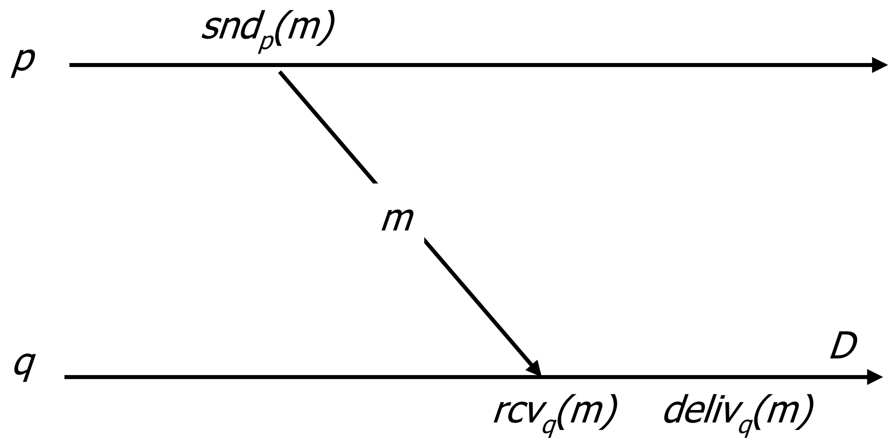
But what does time “mean”?

- Time on a global clock?
  - E.g. with GPS receiver
- … or on a machine’s local clock
  - But was it set accurately?
  - And could it drift, e.g. run fast or slow?
  - What about faults, like stuck bits?
- … or could try to agree on time

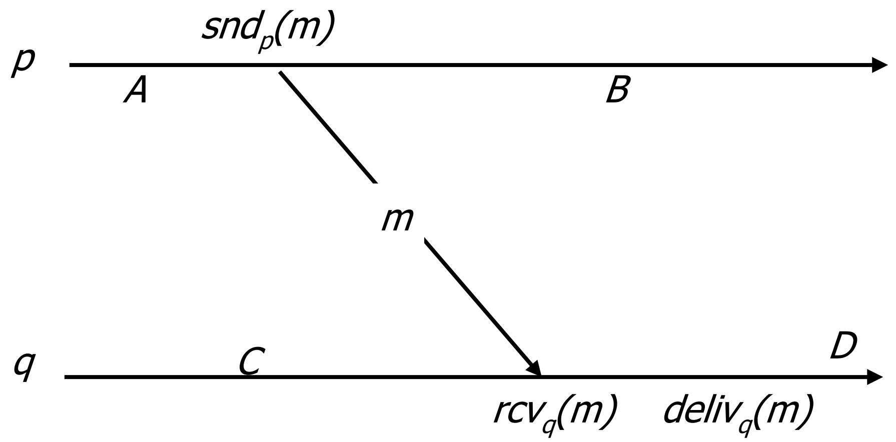
Lamport’s approach

- Leslie Lamport suggested that we should reduce time to its basics
  - Time lets a system ask “Which came first: event A or event B?”
  - In effect: time is a means of labeling events so that…
    - If A happened before B, TIME(A) < TIME(B)
    - If TIME(A) < TIME(B), A happened before B

Drawing time-line pictures:

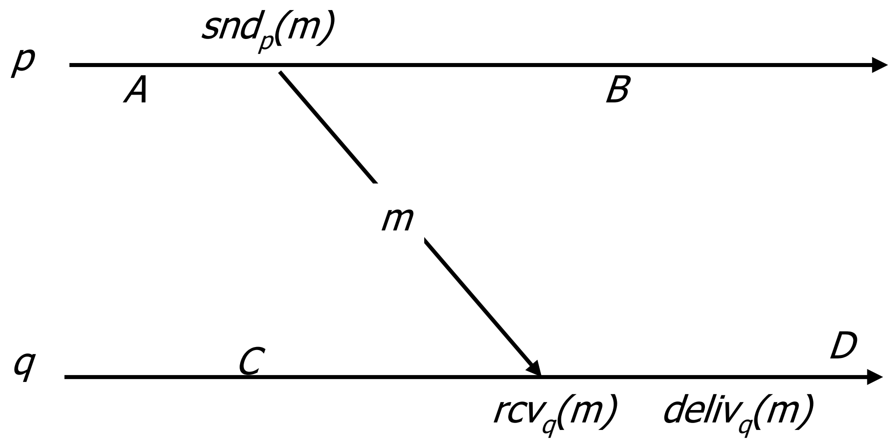


Drawing time-line pictures:



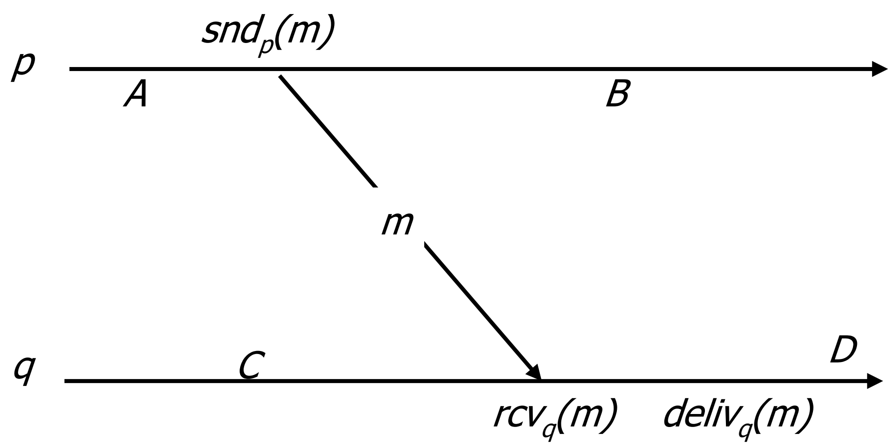
- A, B, C and D are “events”.
  - Could be anything meaningful to the application
  - So are snd(m) and rcv(m) and deliv(m)
- What ordering claims are meaningful?

Drawing time-line pictures:



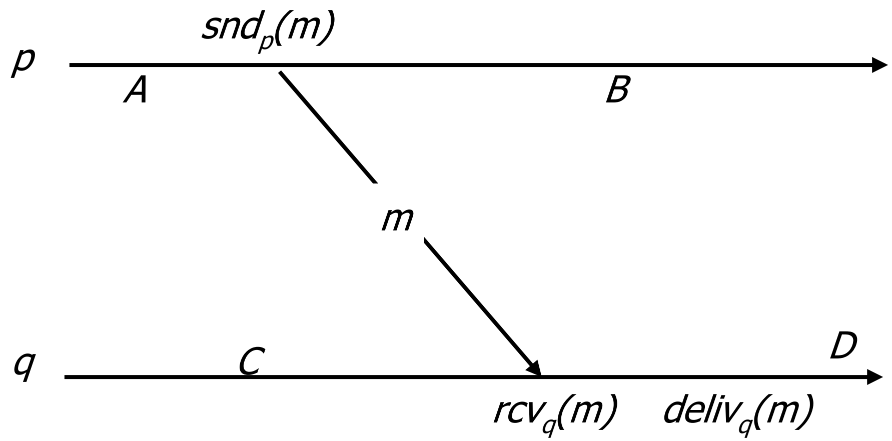
- A happens before B, and C before D
  - "Local ordering" at a single process
  - Write $A \xrightarrow{p} B$ and $C \xrightarrow{q} D$

Drawing time-line pictures:



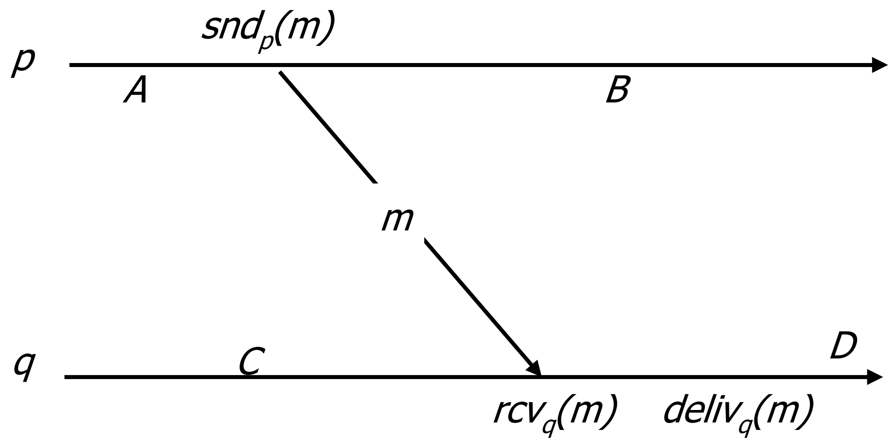
- $snd_p(m)$ also happens before $rcv_q(m)$
  - "Distributed ordering" introduced by a message
  - Write $snd_p(m) \xrightarrow{M} rcv_q(m)$

Drawing time-line pictures:



- A happens before D
  - Transitivity: A happens before $snd_p(m)$, which happens before $rcv_q(m)$, which happens before D

Drawing time-line pictures:



- B and D are concurrent
  - Looks like B happens first, but D has no way to know. No information flowed…

Happens before "relation"

- We'll say that "A happens before B", written $A \rightarrow B$, if
  1. $A \rightarrow^{P} B$ according to the local ordering, or
  2. A is a *snd* and B is a *rcv* and $A \rightarrow^{M} B$, *or*
  3. A and B are related under the transitive closure of rules (1) and (2)
- So far, this is just a mathematical notation, not a "systems tool"

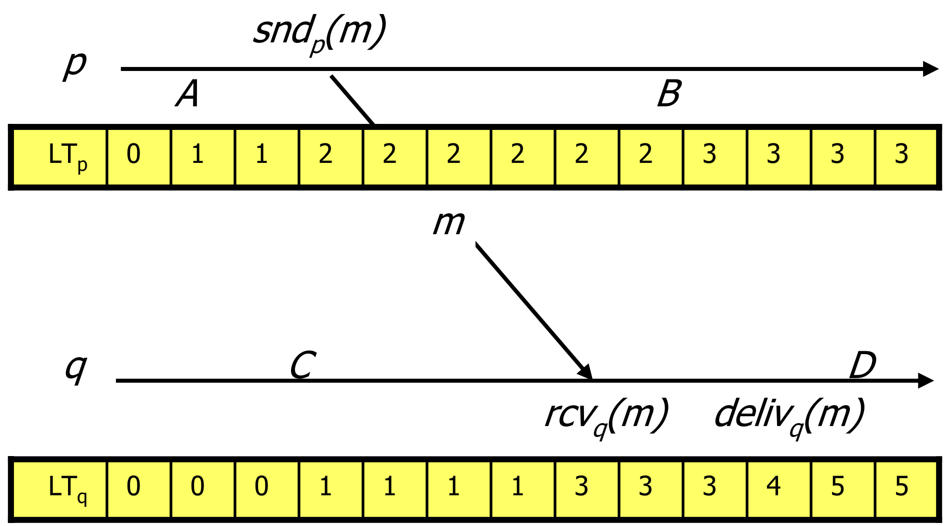
Logical clocks

- A simple tool that can capture parts of the happens before relation
- First version: uses just a single integer
  - Designed for big (64-bit or more) counters
  - Each process *p* maintains LT_p, a local counter
  - A message *m* will carry LT_m

Rules for managing logical clocks

- When an event happens at a process *p* it increments LT_p.
  - Any event that matters to *p*
  - Normally, also *snd* and *rcv* events (since we want receive to occur "after" the matching send)
- When p sends *m*, set
  - $LT_m = LT_p$
- When q receives *m*, set
  - $LT_q = \max(LT_q, LT_m)+1$

Time-line with LT annotations



- LT(A) = 1, LT(snd_p(m)) = 2, LT(m) = 2
- LT(rcv_q(m))=max(1,2)+1=3, etc…

Logical clocks

- If A happens before B, $A \rightarrow B$, then $LT(A) < LT(B)$
- But converse might not be true:
  - If $LT(A) < LT(B)$ can't be sure that $A \rightarrow B$
  - This is because processes that don't communicate still assign timestamps and hence events will "seem" to have an order

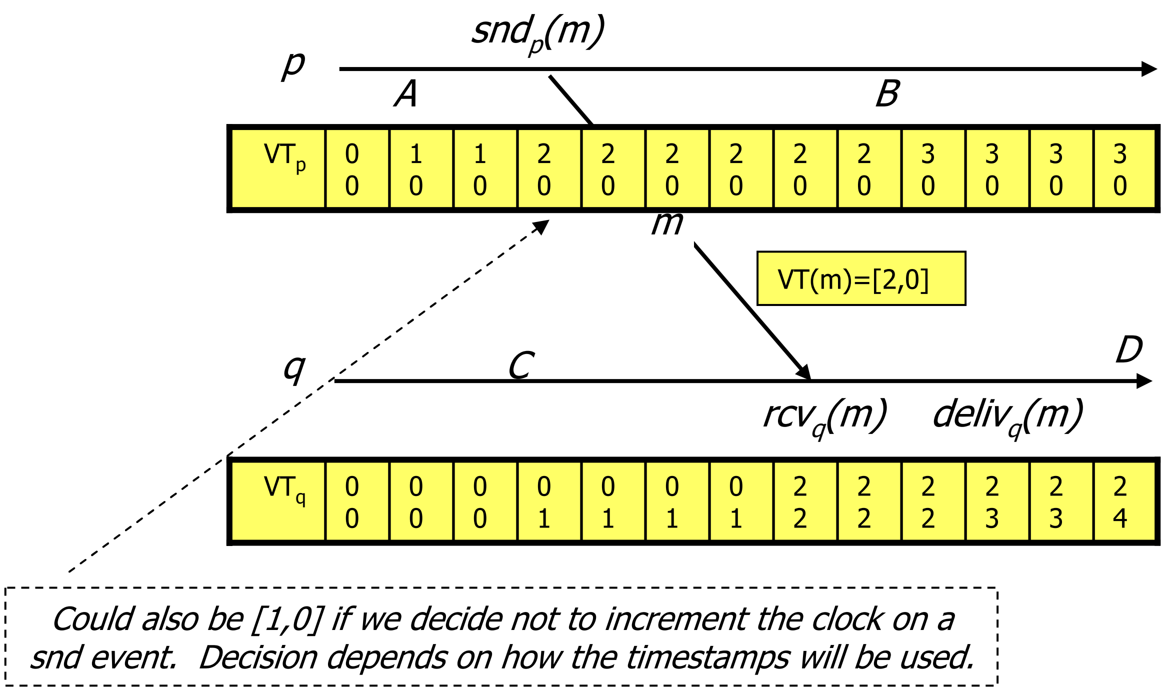
Can we do better?

- One option is to use *vector* clocks
- Here we treat timestamps as a list
  - One counter for each process
- Rules for managing vector times differ from what did with logical clocks

Vector clocks

- Clock is a vector: e.g. VT(A)=[1, 0]
  - We'll just assign p index 0 and q index 1
  - Vector clocks require either agreement on the numbering, or that the actual process id's be included with the vector
- Rules for managing vector clock
  - When event happens at p, increment $VT_p[index_p]$
    - Normally, also increment for snd and rcv events
  - When sending a message, set $VT(m)=VT_p$
  - When receiving, set $VT_q=max(VT_q, VT(m))$

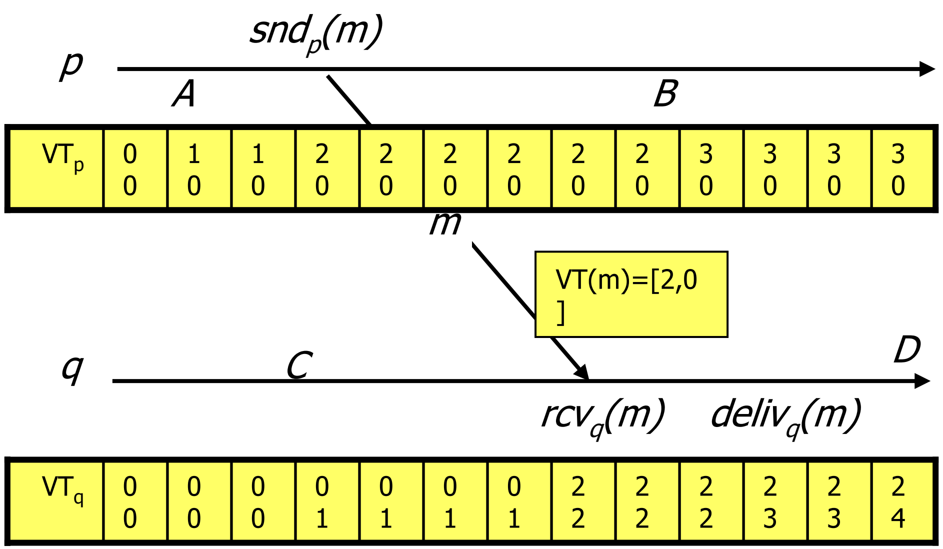
Time-line with VT annotations



Rules for comparison of VTs

- We'll say that $VT_A \leq VT_B$ if
  - $\forall_I$, $VT_A[i] \leq VT_B[i]$
- And we'll say that $VT_A < VT_B$ if
  - $VT_A \leq VT_B$ but $VT_A \neq VT_B$
  - That is, for some i, $VT_A[i] < VT_B[i]$
- Examples?
  - [2,4] ≤ [2,4]
  - [1,3] < [7,3]
  - [1,3] is "incomparable" to [3,1]

Time-line with VT annotations



- VT(A)=[1,0]. VT(D)=[2,4]. So VT(A)<VT(D)
- VT(B)=[3,0]. So VT(B) and VT(D) are incomparable

Vector time and happens before

- If $A \rightarrow B$, then $VT(A) < VT(B)$
  - Write a chain of events from A to B
  - Step by step the vector clocks get larger
- If $VT(A) < VT(B)$ then $A \rightarrow B$
  - Two cases: if A and B both happen at same process p, trivial
  - If A happens at p and B at q, can trace the path back by which q "learned" $VT_A[p]$
- Otherwise A and B happened concurrently

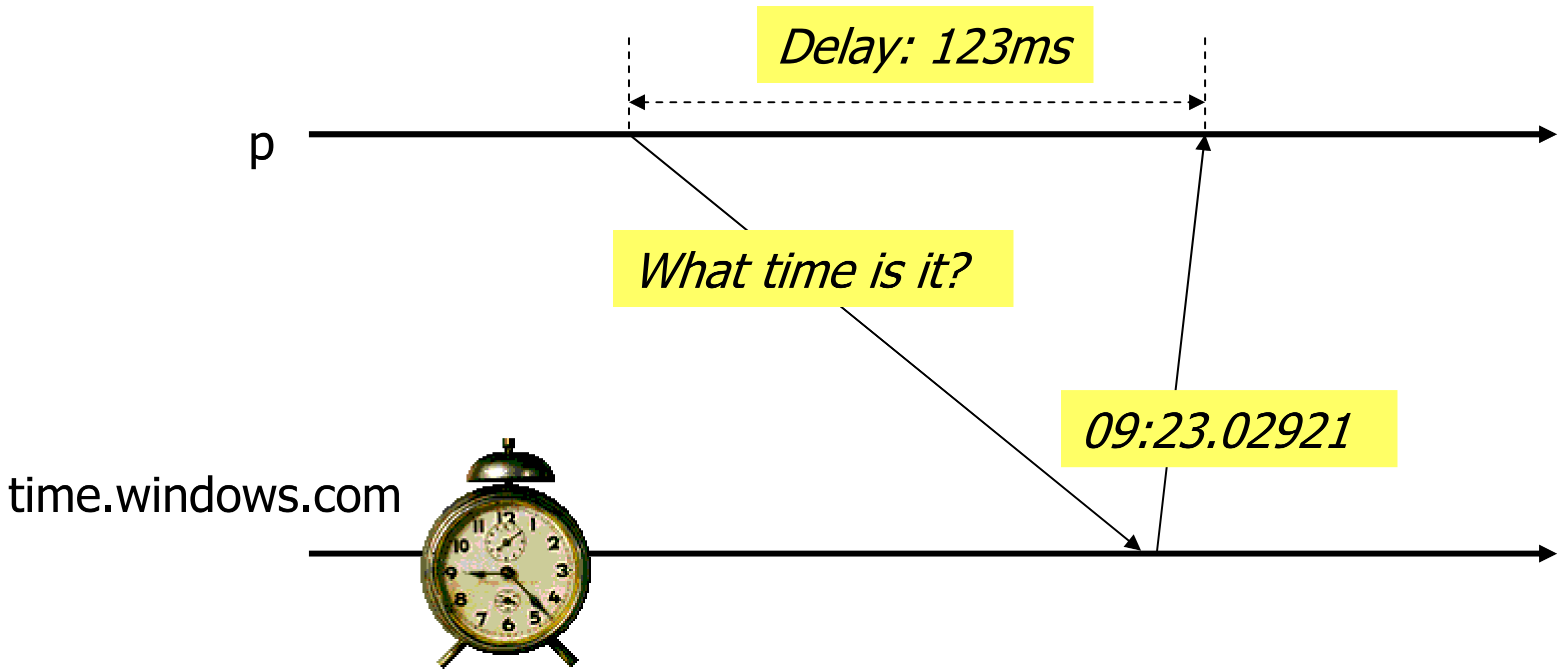
Introducing “wall clock time”

- There are several options
  - “Extend” a logical clock or vector clock with the clock time and use it to break ties
    - Makes meaningful statements like “B and D were concurrent, although B occurred first”
    - But unless clocks are closely synchronized such statements could be erroneous!
  - We use a clock synchronization algorithm to reconcile differences between clocks on various computers in the network

Synchronizing clocks

- Without help, clocks will often differ by many milliseconds
  - Problem is that when a machine downloads time from a network clock it can't be sure what the delay was
  - This is because the "uplink" and "downlink" delays are often very different in a network
- Outright failures of clocks are rare…

Synchronizing clocks



- *Suppose p synchronizes with time.windows.com and notes that 123 ms elapsed while the protocol was running… what time is it now?*

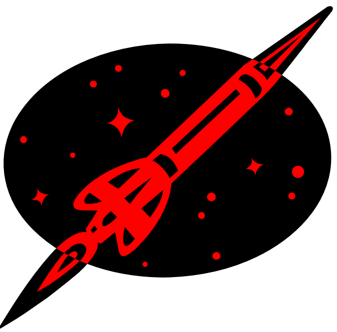
Synchronizing clocks

- Options?
  - P could guess that the delay was evenly split, but this is rarely the case in WAN settings (downlink speeds are higher)
  - P could ignore the delay
  - P could factor in only “certain” delay, e.g. if we *know* that the link takes at least 5ms in each direction. Works best with GPS time sources!
- In general can’t do better than uncertainty in the link delay from the time source down to p

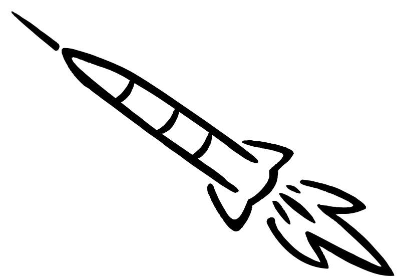
Consequences?

- In a network of processes, we must assume that clocks are
  - Not perfectly synchronized. Even GPS has uncertainty, although small
    - We say that clocks are "inaccurate"
  - And clocks can drift during periods between synchronizations
    - Relative drift between clocks is their "precision"

Thought question



- We are building an anti-missile system
- Radar tells the interceptor where it should be and what time to get there
- Do we want the radar and interceptor to be as accurate as possible, or as precise as possible?



Thought question

- We want them to agree on the time but it isn't important whether they are accurate with respect to "true" time
  - "Precision" matters more than "accuracy"
  - Although for this, a GPS time source would be the way to go
    - Might achieve higher precision than we can with an "internal" synchronization protocol!

Real systems?

- Typically, some “master clock” owner periodically broadcasts the time
- Processes then update their clocks
  - But they can drift between updates
  - Hence we generally treat time as having fairly low accuracy
  - Often precision will be poor compared to message round-trip times

Clock synchronization

- To optimize for precision we can
  - Set all clocks from a GPS source or some other time "broadcast" source
    - Limited by uncertainty in downlink times
  - Or run a protocol between the machines
    - Many have been reported in the literature
    - Precision limited by uncertainty in message delays
    - Some can even overcome arbitrary failures in a subset of the machines!

For next time

- Read the introduction to Chapter 14 to be sure you are comfortable with notions of time and with notation
- Chapter 23 looks at clock synchronization