Proseminar Distributed Systems
Summer Semester 2016

Paxos algorithm

Stefan Resmerita
stefan.resmerita@cs.uni-salzburg.at

The Paxos algorithm

- Family of protocols for reaching consensus among distributed agents
- Agents may fail (and later recover)
- Messages may be delayed or lost

- Employed to achieve fault-tolerant state machine replication

The Paxos algorithm (contd.)

- A voting protocol by which a set of communicating agents (processes) agree on a value
- Messages are broadcast, they may be delayed or may be lost, but not corrupted
- Any agent may fail; if an agent fails, it must be able to re-start from a saved protocol state
- Values may be proposed by different agents at different times during the voting

Agent Roles

- Proposer
  - Initiates a voting round
  - Proposes a value to be agreed upon
- Acceptor
  - Acknowledges a proposer
  - Votes for a proposed value (accepts the value) or against (doesn‘t answer or disagrees)
- Learner
  - Detects when a value has been accepted by a majority of acceptors

Basic Paxos Algorithm – Phase 1

- A proposer *P* sends a *prepare* request with a proposal number *n* to a majority of acceptors
- An acceptor A checks if the requested n is greater than the maximum proposal number m whose value A has already **accepted** so far.
  - If so, then it sends *to P* an *ACK* message (A, m, v_m)
    - v_m is the accepted value of proposal number m
    - if A accepted no value so far then it responds with null values
  - Otherwise, it does not recognize ***P***'s request (no response or response with a *NACK* message)

Basic Paxos Algorithm – Phase 2a

- Proposer *P* receives a set **M** of *ACK* messages as response to its proposal number *n*. If these come from a majority of acceptors, then
  - It takes $m_{\max} = \max_{(A,m,v_m)\in\mathbf{M}} m$
  - It sets $v_n = \begin{cases} v_{m_{\max}}, & \text{if } m_{\max} \neq NULL \\ a, & \text{otherwise} \end{cases}$

    where a is some value chosen by P
  - It sends to each acceptor A, with $(A,m,v_m)\in\mathbf{M}$, an *accept* request for proposal number *n* with value v_n

Basic Paxos Algorithm – Phase 2b

- Upon receiving an *accept* request for proposal number *n*, an acceptor accepts v_n iff it has not already acknowledged a *prepare* request with number greater than *n*
- The acceptance message is sent to a set of learners
- When a learner receives acceptance messages from a majority of acceptors, it commits the accepted value

Basic Paxos Algorithm – Progress

- Proposer P completes phase 1 for proposal number n_1
- Then another proposer Q completes phase 1 for a proposal number $n_2 > n_1$
- Phase 2 of P fails
- P starts another phase 1 with number $n_3 > n_2$
- Phase 2 of Q fails
- And so on...

Basic Paxos Algorithm – Progress

- A distinguished proposer must be established
- Only the distinguished proposer may issue proposals
- The leader may fail, in which case a new leader is chosen
- Further reading about the Paxos algorithm
  - Links on the DS lecture website
  - Additional links on the PS website!

Multi-Paxos

- To agree on a *sequence* of values
- Each process runs multiple Paxos instances concurrently
- A message between processes includes, in addition to proposal number and value, the index of the Paxos instance to which the above data refers

Proseminar Distributed Systems
Summer Semester 2016

Distributed Clock Synchronization

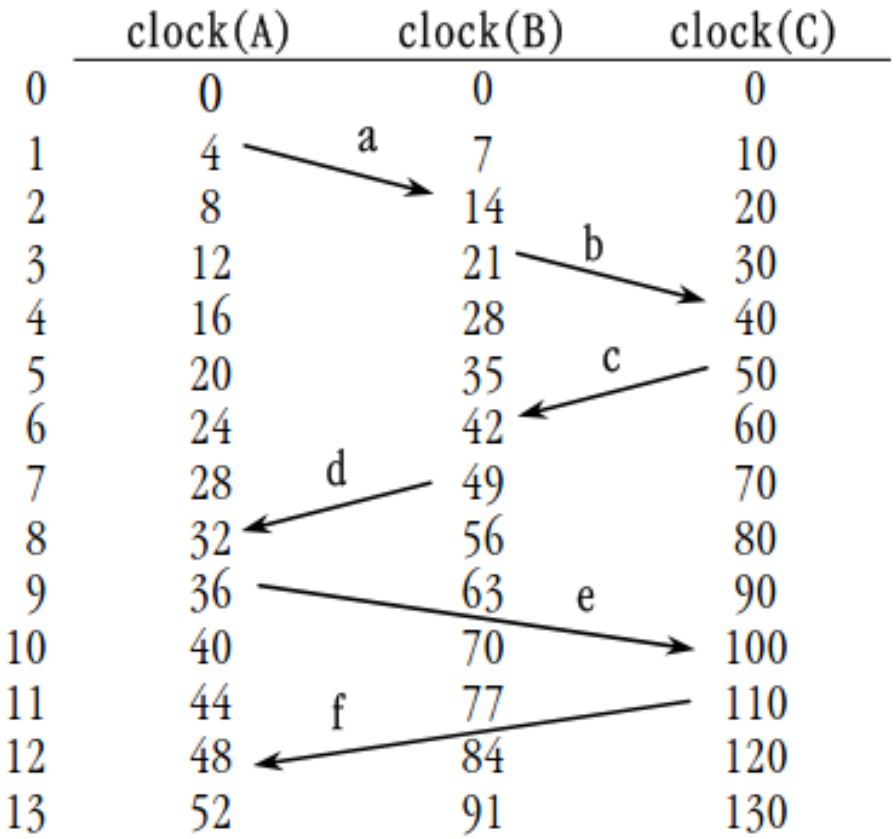
Stefan Resmerita
stefan.resmerita@cs.uni-salzburg.at

Logical clocks

- Interconnected DE nodes
  - Events are ordered using timestamps at each node
  - Timestamps are generated locally
  - The „time“ at a node is the timestamp of the last processed event
  - Nodes communicate via messages

- Global ordering of events must be ensured

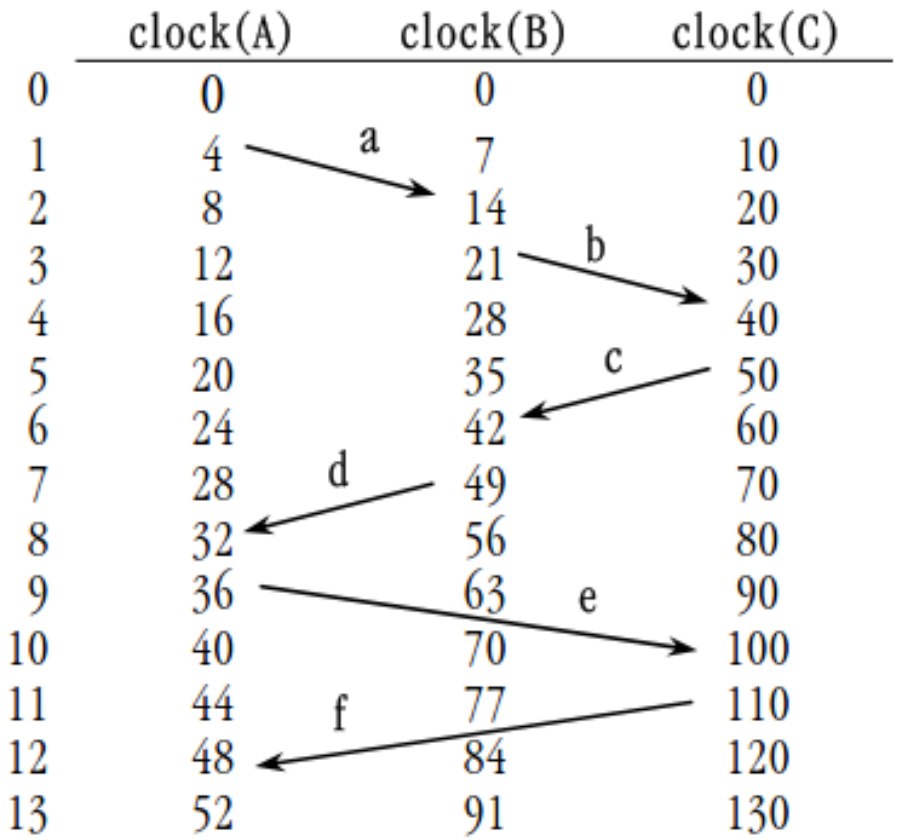
Logical clocks example

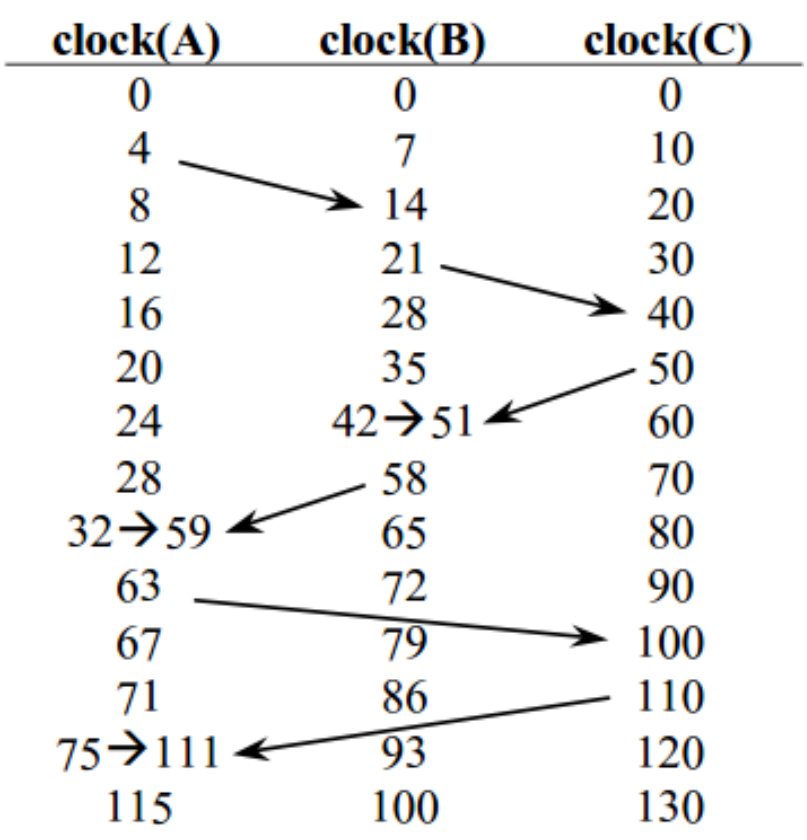


Lamport‘s algorithm

- Each message includes the sender‘s time *t*
- If the receiver‘s time is less than *t*, then the clock of the receiver is advanced to *t+1*
  - Otherwise, do nothing

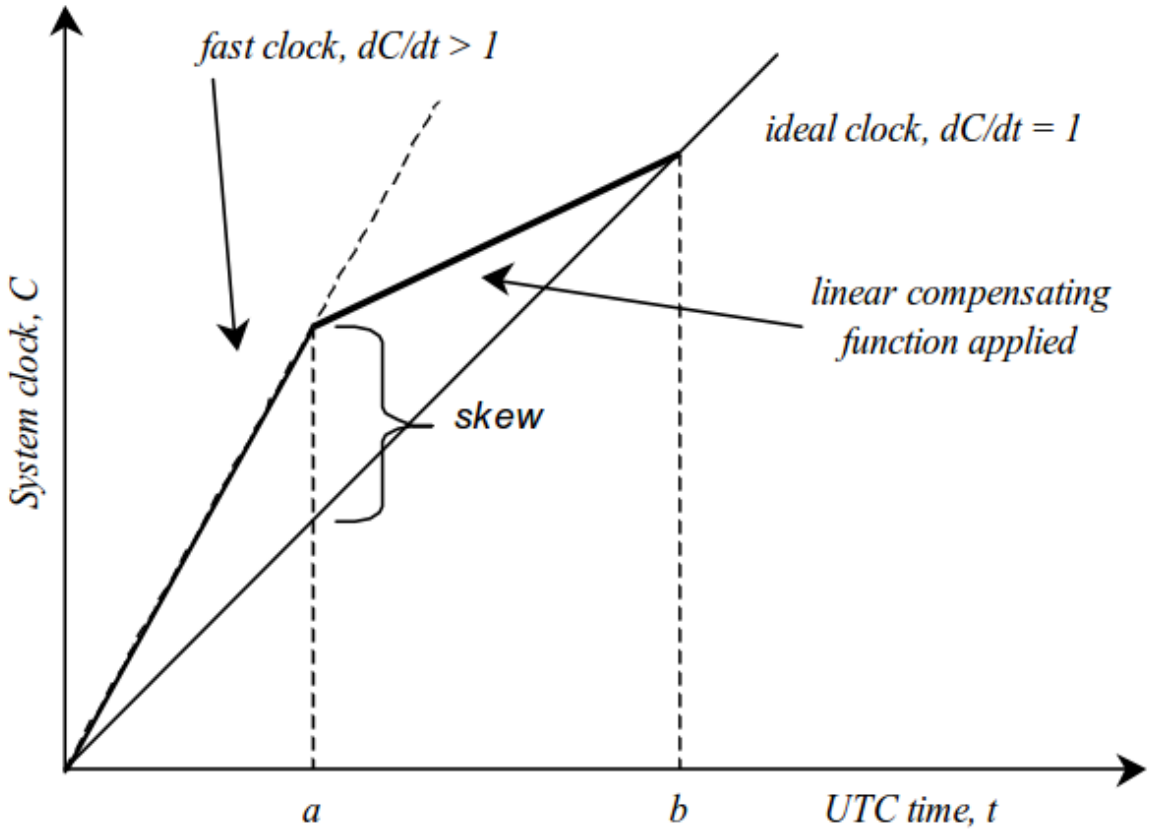
Lamport‘s algorithm example





Physical clocks

- Drift compensation



Christian‘s algorithm

- Reference time provided by server
- T_0: local time when client issues time request to server
- T_1: local time when server‘s reply arrives at client
- T_{server}: time value specified by server

$$T_{client} = T_{server} + (T_1 - T_0)/2$$

Berkeley algorithm

- Participating nodes synchronize to average time
- Master node:
  - periodically asks the others for their times
  - computes the average
  - sends back the offset for each slave node
  - adjusts its own time as well
- Example: