Master 2 IFI, CSSR + Ubinet

Distributed Algorithmic

Françoise Baude

Université de Nice Sophia-Antipolis

UFR Sciences – UFR Polytech

baude@unice.fr

web site : deptinfo.unice.fr/~baude/AlgoDist

Oct. 2013

Chapter 2: Time, cuts & consistent snapshots

Course 2: plan

1. Time in an asynchronous distributed system
   - Motivation: time
   - Physical clocks synchronization
   - Ordering of events: Causality relation
   - Logical clocks
     - Integer
     - Vector
     - Others
2. Consistent snapshots and cuts
   - Motivation
   - Consistency
   - Snapshot Distributed algo. assuming FIFO channels

1. Use of time: motivation

- Date events happening in a distributed system (logging, tracing, visualizations, debugging, …)
  - E.g.: give a precise occurrence date to electronic business transactions impacting several sites (merchant, bank, etc)
  - Be able to replay the distributed application execution:
    - Messages must be sent, received and treated in the same order
    - => goal is to obtain the same timed graph, not mandatorily the same exact real time occurrence for all events
- Problem: date all events correctly, specially when they are correlated
  - But, can not rely on a single observer (="god")
  - An observer only sees the happening of some events, and their relative orders. Global order can not be inferred from that

Physical clocks

- A unique physical clock would be perfect !
  - But, doesn't exist
- A few official synchronized sources of unique time on earth
  - Atomic-based and very precise clocks, that provide the International Atomic Time
    - 1sec=9192631770 transitions of Celsium133 atom
  - Coordinated Universal Time (UTC): several government agencies radio-broadcast this official Time from all over the world
    - Eg Greenwich in Europe
    - Correct the IAT, according to the UTC standard, since 1/1/1972
  - GPS satellites are also a reliable source of time, as they embed atomic clocks
- Computer clocks: one per distributed computer, acting as receiver
  - Not natively synchronized
    - Use of the Network Time Protocol to resynchronize w.r.t. to UTC
      - Several levels of NTP servers, level 1 being the closest to UTC sources
    - Always the problem that reading a time is done remotely: uncertainty introduced due to the non instantaneous propagation delay. Nature of the source thus impacts the precision of the clock
  - Clock drifts still there: need to periodically re-fix them mutually

Physical clocks synchronization

- Goal: synchronize 2+ clocks, with a given accuracy,
  - timestamp distributed events with the clock reading at their occurrence site
  - It should be possible to know
    - in which order distributed events occurred
    - And, the time difference between them
- External vs. internal clock synchronisation
  - Synchro. w.r.t. official external UTC time
    - Use NTP for instance
  - Synchronize internally, so to use the same referential of time on the distributed system, and upper bound clock drift, in the range of the expected accuracy
    - For all real t, all i, j, $|Ci(t) - Cj(t)| < D$, D=clocks agreement bound
  - Choice: depend on the public or private (open vs confined) status of the distributed system

5

Cristian's method for clock external synchronization

- General method for clock synchronization in an asynchronous messaging system
  - whenever impossible to bound the message transmission delay (we only know that delay is finite)
- P requests time to the source S, at time ts
- S replies with a message m(t) (t is the time on S)
- P receives m(t) at time ts+round-trip delay
- P sets its clock to t + (round-trip/2)
  - Round-trip / 2 is a not so bad approximation of a one-way transmission delay
    - Possible to repeat the method, and keep the minimal Round-trip
- Accuracy is ±(Round-trip / 2 – *min*) whenever we know *min*, the minimum delay for a one-way communication between P and S

6

Berkeley algorithm for internal clock synchronization

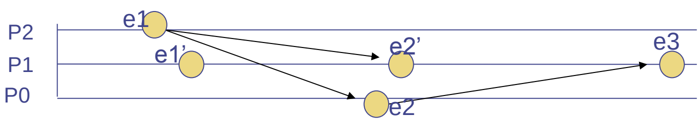
- Used in Berkeley Unix 4.3 BSD
  - Goal is that between any two machines, the clock difference never exceed a δ value
1. Regularly a master fetches the time from all participants clocks,
   - Estimating clock values considering an average round-trip delay
2. compute a “fault-tolerant” average of these,
   - Including its own (“more correct” = no subject to transport delay) value
   - Considering only values that are within a skew of δ thus eliminating clock values that are too much different
3. report back to each participant the needed + or - adjustment.

Rem: Solutions for tolerating more faults, like master failure, exist.

7

Happened-before relation: →

- When 2 events e1, e2,
  - Are local to a process Pi, e1 → e2
  - e1: message send on Pi, e2: corresponding message reception on Pj, e1 → e2
- Several events, e1, e2, e3
  - If e1 → e2, and e2 → e3, then, e1 → e3
- Not all events are mandatorily related along →
  - Incomparable, independent, concurrent: $\not\to$ also ||
    - Non transitivity of | |
- Happened-before relation: also named <u>Causality</u>



e1 → e2
e1 → e2'
e2 → e3
e1 → e3
e1' → e2'
e2' → e3
e1' → e3

e1 $\not\to$ e1'
e1' $\not\to$ e2
e2 $\not\to$ e2'

e1' $\not\to$ e1
e2 $\not\to$ e1'
e2' $\not\to$ e2

8

Logical clocks: motivation

- A cheap alternative when events have not to be stamped with real time values, but only the happened-before relation matters
- All events happening on one site are always correctly ordered along the happened-before relation.
  - Their associated clock-based date are coherent with the relation
- The problem is when the sites are different:
  - How to make sure "if event1 happened before event2, Clock(event1 on site A) < Clock(event2 on site B)" ?
    - Clocks on sites A and B may not be correctly synchronized, may skew
    - Event1 and Event2 may be unrelated even if they occurred in the real time along a specific order
      - Logical instead of Physical clocks can suffice

Logical clock general definition

- Timestamp events with a date, gained from a logical clock L, so that
  - If e1 → e2, then L(e1) < L(e2)
  - And, one of the two features below for L:
    - If e1 || e2, then either L(e1) < L(e2), or it can be that L(e2)<L(e1)
    - If e1 || e2, then L(e1) and L(e2) can not be ordered with <
      - More powerful, because then L(e1) < L(e2) implies e1 →e2
- Devise necessary logical clock management rule, i.e. how L should 'tick'
  - L must increase in accordance with the distributed → relation
  - Distributed clocks should synchronize along →

Lamport's integer logical clock

- On each Pi
  - Its clock L, is set =0 initially
  - Before each (local) event, L is increased by 1.
    - So, for 2 successive local events e1, e2, L(e1) < L(e2)
  - On sending of a message m to Pj, m is stamped with current value of L
    - e1 is local to Pi, e2 is a message send to Pj, L(e1)<L(e2)
  - On reception of a message m, with timestamp l,
    1. Fix Pi'L relatively to Pj'L: L = max (L, l)
    2. Increase L by 1, in order to correctly date the event corresponding to the reception of m
    - L(reception of m) > L(sending of m) for any m

Properties of Lamport' clock

- Correct w.r.t happened-before relation
- But, L(e1)<L(e2), does not imply e1 →e2
- < is not a total-order relation
  - Possible that L(e1) = L(e2) when e1 and e2 happens on 2 sites
  - If total order required, add e.g. site identifier
    - L(e1)=8 on site A, L(e2)=8 on site B, assume A "smaller" than B, then L(e1) < L(e2)
    - Can be necessary when 2 requests to do something have the same value, but, there is a need to order them in an non-ambiguous and same order on all sites.

- Exo: play with → , and apply Lamport clock

Vector clocks [Fidge/Mattern]

- For a N process system,
  - N-size vector clocks
  - Not a scalable solution… ☹
- On each Pi
  - Initially, V[j]=0, for all j=1..N
  - Just before Pi timestamps an event, V[i]=V[i]+1
  - Pi timestamps each message it sends with V
  - When Pi receives a timestamp t in a message, it sets V[j]=max(V[j],t[j]) for all j. This is a *merge*
- *Properties:* For Pi vector clock V
  - V[i] is the number of events that Pi has timestamped
  - V[j] for j ≠i is the number of events that occurred on Pj that Pi has potentially been affected by.

Vector timestamps comparison

- For two vector timestamps V1, V2
  - V1 = V2 iff V1[j] = V2[j], for all j
  - V1 ≤ V2 iff V1[j] ≤ V2[j], for all j
  - V1 < V2 iff V1 ≤ V2 and V1≠ V2
- It is obvious: for any e1 → e2, it implies V(e1) < V(e2)
  - e1 and e2 local to Pi, obvious
  - e1 send on Pj, e2 reception on Pi=> V(e2) contains max V(e1)[] for all j, including for i, where V(e2)[i]>V(e1)[i]
  - Still true by transitivity
- Most interesting:
  - V(e1) < V(e2) also implies e1 → e2
  - V(e1) ≠ V(e2) iff e1 || e2
- Exo: on vector clocks

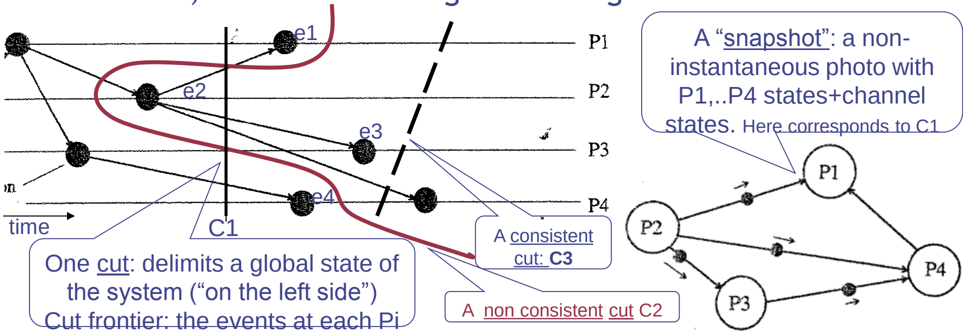
2. Consistent Distributed Snapshots

- Required in order to get a ± "instantaneous", global, but correct view of an asynchronous and distributed system
  - View = constituted by the states of each process, and each channel in the system => this gives us a Global state
    - Some particular global stable states: deadlock, terminated
    - Where a global state of the system is needed?: garbage collection, debugging, fault-recovery from stored/checkpointed global state
  - Without relying on a single observer
  - Without relying on a single clock to record all process states at the same time
    - And how to record communication channel states ?
  - Synchronized physical clocks could be used, only if clock skew not too high
    - Still costly; so better not relying on them

15

Consistency, Snapshot, Cut: definitions

- Consistency= respect of the causality relation
- When recording the state of all processes, all causally past events must be integrated
  - Forbid to include an event e2, caused by an event e1, and not including e1 in the global state



General algorithm to take a “snapshot”

- Impossible to visit all sites and all channels at the same instant…
  - whereas, it is easy to record successive states for any individual process
  - Must include which messages have been sent and, which have been received from each neighbor
    - Solution to “photography” channel states is needed
      - A message sent that is not shown to be received at its destination is <u>in transit</u>, in the corresponding channel
- We aim for a distributed and asynch. message-passing algorithm
  - Recording the history of events on all processes
  - Able to delimit a consistent cut (with a frontier) in the history= a set of events, with the property that for each event, all those belonging to its past are part of the snapshot
    - {e1,e2,e3,e4} is the frontier of the consistent cut C3 on previous slide
    - Exo: a consistent cut “c”can be labeled with a vector clock Vc as the max of all vector clocks associated to the events of the frontier
  - A naïve and incorrect solution would be:
    - PA records its state, and asks all its neighbors (PB, PC) to do the same.
    - PB records its state before sending a message M to PC. PB state does not include this ‘message send’ event
    - Assume PA-> PC communications take longer: PC records its state after it received the message M. PC state includes the message reception event
    - => the union of PA, PB, PC states is not a consistent snapshot

Distributed Snapshot algo for FIFO channels [Chandy-Lamport]

- Channels are FIFO. Messages are not lost. No failure
- Snapshot algo. executes concurrently with the application
- Special “control” message
  - When receiving it for the 1st time through a channel:
    - Pi records its state, and channel state = empty
    - Pi forwards control message to all its outgoing neighbors
  - Messages received through the other incoming channels after a 1st received “control” msg are logged
  - When not the 1st time:
    - Pi adds to its state all logged msgs that came from this channel so far
- Any process may initiate the algo. at any time (triggers one control msg for itself), but concurrent algo. execs must be distinguishable
- Terminated: all Pi received control msg from all incoming channels
- Logged msgs on P->Q, logged by Q are “msgs sent by P to Q while P and Q already logged their state, and Q waited the control msg from P” (m3 in the Ex.)