Distributed Systems - Theoretical Foundations (pt. 2) Agreement Protocols

CS239 Advanced Operating Systems April 14, 1997 Dr. Richard Guy

Lecture Source Materials

Singhal, Ch. 8: **Agreement Protocols**

Lamport, Shostak, & Pease, **The Byzantine Generals Problem,** ACM TOPLAS, July 1982.

Outline

- Theoretical Foundations (part 2 of 2)
  - Vector clocks
  - Causal ordering of messages
  - Global state
- Agreement Protocols

Theoretical Foundations

- Motivation
- Model
- Inherent limitations of distributed system
- Virtual clocks
- Causal ordering of messages
- Global state

Key question

- Is it possible to obtain a coherent global state of a distributed system, given the absence of a global clock, no shared memory, and the existence of arbitrary delay communication channels?

"Virtual clock" solutions

- Lamport's logical clocks
  - limitations of Lamport's clocks
- Fidge/Mattern vector clocks

Logical clocks

- Model and definitions
- System of clocks

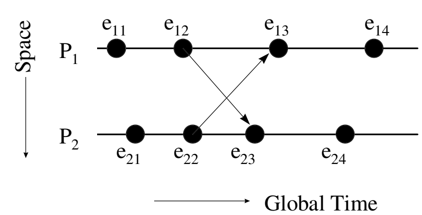
Model

- Process execution is sequence of “events”
- Event is arbitrary granularity
  - procedure call
  - single instruction
- Sending a message is single event
- Receiving a message is single event

“Happened before” relation

- captures causal dependencies between events
- relation $\rightarrow$ is defined as follows:
  - $a \rightarrow b$ if a and b are events in the same process and a occurred before b
  - $a \rightarrow b$, if a is the event of sending a message m in a process and b is the event of receiving m in another process
  - if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$ (transitive)

Space-time diagram



Practical realization of $\rightarrow$

- System of logical clocks, C_i
  - one clock C_i at each process P_i in system
  - C_i assigns number $C_i(a)$ to any event a at P_i
    - timestamp of event a
  - values of C_i have no relation to physical time
  - values of C_i are monotonically increasing
  - implementable by simple counters

Limitations of Lamport’s clock

- If $a \rightarrow b$, then $C(a) < C(b)$
- Is the reverse true?
  - if $C(a) < C(b)$, is $a \rightarrow b$ always true? NO!
- Lamport’s clock cannot distinguish between clock advancement from local vs remote events
- hard to reason based on event timestamps

Vector clocks

- Similar to logical clocks, but with vector rather than scalar values.
- Vector is length n, n is number of processes
  - $C_i[i]$, the ith entry of C_i, corresponds to P_i's logical time
  - $C_i[j], j \neq i$ is P_i's best guess of P_j's logical time
    - $C_i[j]$ is time of last event at P_j which "happened before" the current time at P_i.

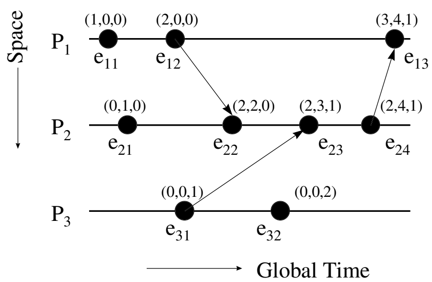
Implementation rules

- **[IR1]** Clock C_i is incremented between any two successive events in process P_i

  $C_i[i] := C_i[i] + d \qquad (d > 0)$
- **[IR2]** If a is the event of sending a message m by process P_i , then m is assigned a vector timestamp $t_m = C_i(a)$ (after applying IR1). When m is received by process P_j , then C_j is

  $\forall k \neq j, C_j[k] := \max(C_j[k], t_m[k]); C_j[j] += d$

Space-time diagram

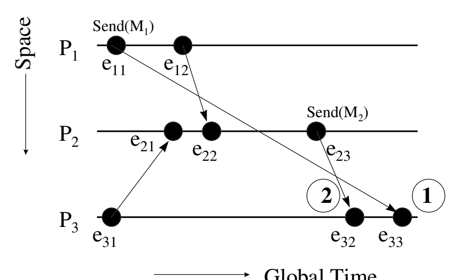


Vector Timestamps

- are "pairwise" comparable to determine
  - (not) equal
  - (not) less than or equal to
  - (not) less than
  - concurrent
  - "less than or equal to" is a partial order
  - concurrent is not transitive

- Events a and b are causally related if
  - $t^a < t^b$, or $t^b < t^a$
  - $a \rightarrow b$ iff $t^a < t^b$
- Can look at timestamps to determine causal relationships, which was not possible with Lamport's logical clocks
- Scalable?

Causal ordering of messages



Causal ordering of messages

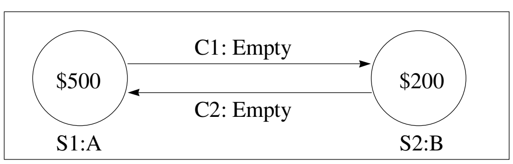
- Matching receive order with send order
  - typically, the desired semantics
- Protocol key concepts
  - delay (buffer) delivery until preceeding messge is known to have been delivered
  - use message-event-specific timestamp, with increment-by-one clock, to identify messages
  - Birman, et al, assumes broadcast messages

Global state

- Task: to record and collect a coherent global state in a distributed system
- Key problem: correctly recording "state" of communication channels

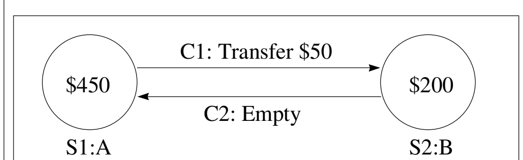
Global state problem

Global state: 1



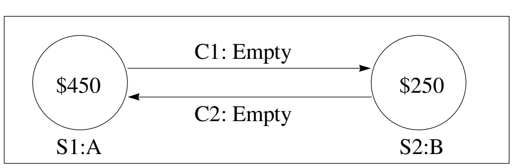
Global state problem

Global state: 2

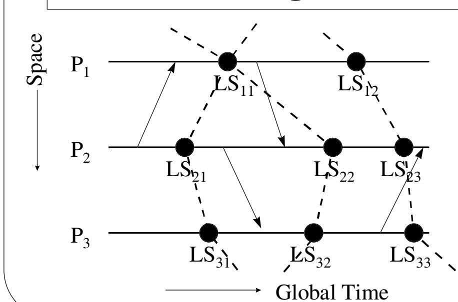


Global state problem

Global state: 3



(In)consistent global state



Chandy-Lamport Algorithm

- Key idea: use special computation-independent marker to delimit/flush channel
- processes record local state and channel state, upon receiving marker(s)
- after receiving markers on all incoming channels, send recorded state to initiator

Algorithm details, part 1

- Marker sending/receiving rules
- *Marker sending rule for process P*
  - *P* records its local state
  - *P* sends marker along all outgoing channels, if it hasn't yet done so

Algorithm details, part 2

- *Marker receiving rule for process Q*
  - If *Q* has not yet recorded its state
    - record state of *C* as an empty sequence
    - follow the marker sending rule
  - else
    - record state of *C* as the sequence of messages received on *C* after *Q*'s state was recorded (earlier) and before *Q* received the marker along *C*.

Algorithm comments

\+ Multiple initiations allowed
- single process
- multiple processes

\+ Particularly useful for stable state detection
- deadlock detection
- computation termination

– Assumes FIFO channels

Summary

- "No global clock" is important because...
- "No shared memory" is important because...
- The virtual clock approach helps because...
- Causal message ordering is useful because...
- Global state determination is used in…

Skipping material in...

- Ch. 6: Distributed Mutual Exclusion
- Ch. 7: Distribued Deadlock Detection

Agreement Protocols Outline

- Motivation
- Definition
- System Model
- Problem Classification
- Byzantine Agreement
- Applications

Motivation

- Cooperation or competition
  - distributed commit
  - unique resource reservation/usage
- Easy, if failure-free
- Harder (impossible?) with failures
  - fail-stop would be nice, but...
  - malicious process prevent agreement?

Approach

- processors exchange messages
- faulty process may
  - fail to respond/send as expected
  - send malicious messages
- nonfaulty don't know who is faulty
- all (nonfaulty) execute same protocol

System Model

- *n* processors, of which at most *m* are faulty
- system is logically fully connected
  - message passing directly between all processors
- receiver always knows identity of sender
- links don't break; only processors may fail
- agreement is binary: 0 or 1 are final values

Additional model parameters

- Synchronous vs asynchronous computation
  - rounds
  - asynchronous proven not solvable!!!
- Failure model (crash, omission, malicious)
- Authenticated vs non-authenticated messages
- Computational complexity
  - time, message traffic, storage overhead

Problem Classification

	Byzantine agreement	Consensus	Interactive consistency
Who initiates the value?	One	All	All
Final agreement	Single	Single	Vector

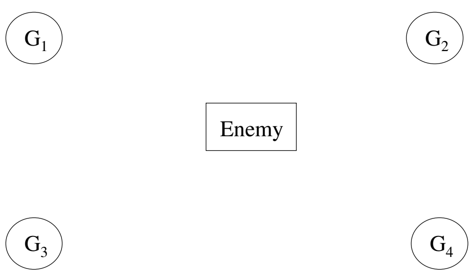
Classification comments

- All three are closely related
- B.A. solution can be basis for other two
  - not necessarily most efficient...

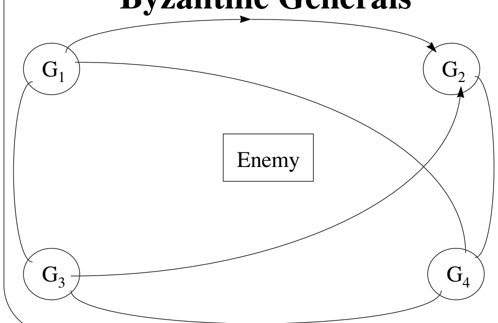
Byzantine Agreement

- Byzantine Generals
- Upper bound on |Faulty processors|
- Impossibility result
- Lamport-Shostak-Pease Algorithm

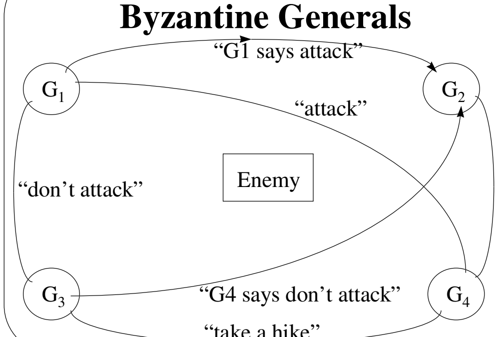
Byzantine Generals



Byzantine Generals



Byzantine Generals



Upper bound on |Faulty processors|

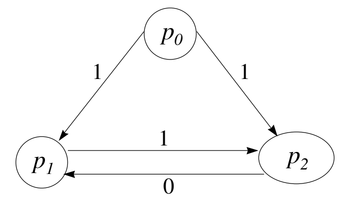
- Too many faulty processors can prevent nonfaulty processors from agreement
- How many is too many?
- If fully connected, non-authenticated

  $m \leq \lfloor (n-1)/3 \rfloor$
- If authenticated, $m < n$ for agreement

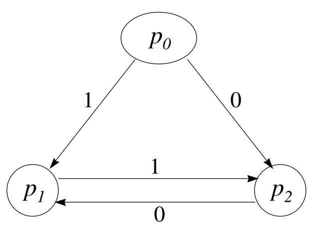
Impossibility result

- Agreement impossible if n=3, m=1
- Consider n=3, m=1, values are 0 and 1, processor p_0 initiates value
- two cases (indistinguishable)
  – p_0 is not faulty
  – p_0 is faulty

p_0 is not faulty



p_0 is faulty



Lamport-Shostak-Pease Algorithm

- “Oral message” (non-authenticated) approach
- recursively defined(!)

Algorithm OM(0)

- source processor sends value to every processor
- each processor uses received value
  – default value is 0, if faulty source doesn’t send

Algorithm OM(m), m>0

- source processor sends value to every processor
- foreach i,let v_i := value process i gets from source

  Processor i initiates **Algorithm OM(m-1)** using v_i
  – processor i acts as new source
  – sends to n-2 (or n-3, etc.) other processors

- foreach i and each $j(\neq i)$,let v_j := value process i received from processor j in Step 2 using Algorithm OM(m-1).

  Processor i uses the value *majority*($v_1, v_2, ..., v_{n-1}$)

Algorithm OM() comments

- message complexity is O(n^m)
- requires m+1 rounds
- Dolev solution is polynomial, but 2m+3 rounds

Applications of Agreement

- clock synchronization
  - multiple initial values
- atomic commit

Summary

- Agreement protocols are needed to…
- Important parameters are…
- It is impossible to reach agreement when …
- Byz. agreement solutions are valuable …
- Typical applications of agreement are ...