Back to Message Passing

Eventual and Strong Consistency
Paxos

INF346, 2014

So far…

Read-write registers cannot solve:

- Wait-free consensus
- Wait-free set agreement
- 1-resilient consensus
  - ✓Can be generalized to k-resilient k-set agreement
- Consensus is universal

Message-passing

- Consider a network where every two processes are connected via a reliable channel
  - ✓no losses, no creation, no duplication
- Which shared-memory results translate into message-passing?
- Implementing a distributed service

Implementing message-passing

Theorem 1 A reliable message-passing channel between two processes can be implemented using two 1W1R registers

Corollary 1 Consensus is impossible to solve in an asynchronous message-passing system if at least one process may crash

Implementing shared memory

Theorem 2 A 1W1R regular register can be implemented in a (reliable) message-passing model where a majority of processes are correct

Implementing a 1W1R register

Upon write(v)
  t++
  send [v,t] to all
  wait until received [ack,t] from a majority
  return ok

Upon read()
  r++
  send [?,r] to all
  wait until received {(t' ,v' ,r)} from a majority
  return v' with the highest t'

Implementing a 1W1R register, contd.

Upon receive [v,t]
  if $t > t_i$ then
    $v_i := v$
    $t_i := t$
    send [ack,t] to the writer

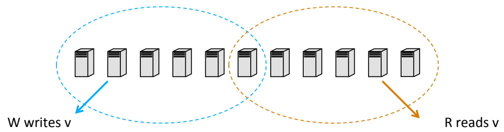
Upon receive [?,r]
  send $[v_i, t_i, r]$ to the reader

What register is it? Regular? Atomic?

A correct majority is necessary

Otherwise, the reader may miss the latest written value

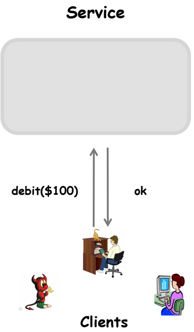
The quorum (set of involved processes) of any write operation must intersect with the quorum of any read operation:



How to build a consistent and reliable system?

Service accepts requests from *clients* and returns responses

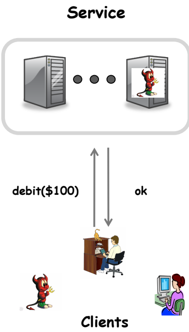
- Liveness: every persistent client receives a response
- Safety: responses constitute a total order w.r.t. a *sequential specification*



How to build a fault-tolerant system?

Replication:

- Service = collection of *servers*
- Some servers may *fail*



CAP theorem [Brewer 2000]

No system can combine:

- Consistency: all servers observe the same evolution of the system state
- Availability: every client's request is eventually served
- Partition-tolerance: the system operates despite a partial failure or loss of communication

Sounds familiar, no?

Strongly consistent replicated state machine

Universal construction in message-passing:

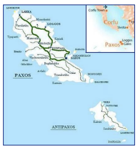
- Clients access the service via a standard interface
- Servers run replicas of the (sequential) service
- (A subset of) faulty servers do not affect consistency and availability

Leslie Lamport: The Part-Time Parliament. ACM Trans. Comput. Syst. 16(2): 133-169 (1998)

Paxos: some history

- Late 80s: a three-phase consensus algorithm
  - ✓A Greek parliament reaching agreement
- 1989: a Paxos-based fault-tolerant distributed database
- 1990: rejected from TOCS

"All three referees said that the paper was mildly interesting, though not very important, but that all the Paxos stuff had to be removed."



This submission was recently discovered behind a filing cabinet in the TOCS editorial office. Despite its age, the editor-in-chief felt that it was worth publishing. Because the author is currently doing field work in the Greek isles and cannot be reached, I was asked to prepare it for publication.

The author appears to be an archeologist with only a passing interest in computer science. This is unfortunate; even though the obscure ancient Paxon civilization he describes is of little interest to most computer scientists, its legislative system is an excellent model for how to implement a distributed computer system in an asynchronous environment.

…

Keith Marzullo
University of California, San Diego
(preface for the TOCS 1998 paper)

Paxos today

- Underlies a large number of practical system when strong consistency is needed
  - ✓Google Megastore, Google Spanner
  - ✓Yahoo Zookeeper
  - ✓Microsoft Azure
  - ✓….
- ACM SIGOPS Hall of Fame Award in 2012
- Turing award 2013

Consensus: recall the definition

A process *proposes* an *input* value in V (|V|≥2) and tries to *decide* on an *output* value in V

- *Agreement:* No two process decide on different values
- *Validity:* Every decided value is a proposed value
- *Termination:* No process takes infinitely many steps without deciding
  (Every correct process decides)

Model

- Asynchronous system
- Reliable communication channels
- Processes fail by crashing
- A majority of correct processes

But we proved that 1-resilient consensus is impossible even with shared memory!

"CAP theorem" is violated!

Where is the trick?

Ω: an oracle

- Eventual leader failure detector
- Produces (at every process) events:
  - ✓‹Ω, leader, p›
  - ✓We also write p=leader()
- Eventually, all correct processes output the same correct process as the leader

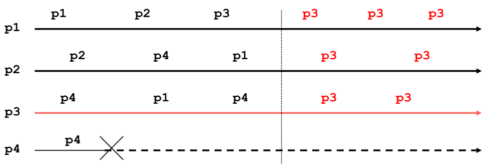
Can be implemented in eventually synchronous system:

- ✓There is a bound on communication delays and processing that holds only eventually
- ✓There is an a priori unknown bound in every run

Leader election Ω: example

There is a time after which the same correct process is considered leader by everyone.

(Sufficient to output a binary flag leader/not leader)



Paxos/Synod algorithm

- Let's try to decouple liveness (termination) from safety (agreement)
- Synod made out of two components:
  - ✓Ω - the eventual leader oracle
  - ✓(ofcons) obstruction-free consensus

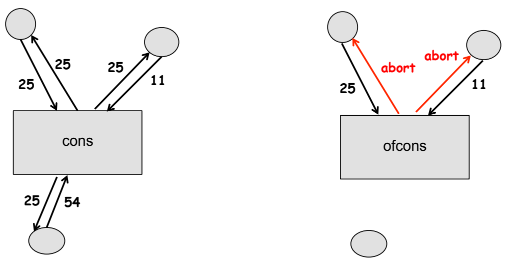
Obstruction-free Consensus (ofcons)

- Very similar to consensus
  - ✓except for Termination
  - ✓ability to abort
- Request:
  - ✓‹ofcons, propose, v›
- Indications:
  - ✓‹ofcons,decide, v'›
  - ✓‹ofcons,abort›

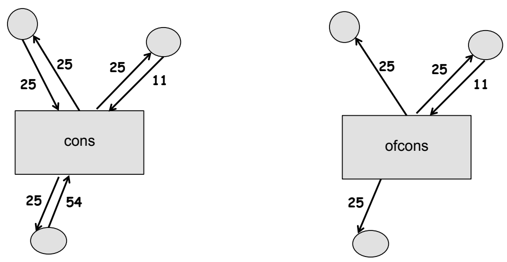
Obstruction-free Consensus

- *C1. Validity*:
  - ✓Any value decided is a value proposed
- *C2. Agreement:*
  - ✓No two correct processes decide differently
- *C3. Obstruction-Free Termination:*
  - ✓If a correct process p proposes, it eventually decides or aborts.
  - ✓If a correct process decides, no correct process aborts infinitely often.
  - ✓If a single correct process proposes a value infinitely many times, p eventually decides.

Consensus vs. OF-Consensus



Consensus vs. OF-Consensus



Consensus using Ω and ofcons

- Straightforward
  - ✓Assume that in cons everybody proposes

```
upon ‹cons, propose, v›
  while not(decided)
      if    self=leader() then
       result = ofcons.propose(v›
       if result=(decide,v') then
           return v'
```

25

Link to Paxos/Synod

- External cons.propose events come in a state machine replication algorithm as requests from clients
  - ✓As in universal construction
- Focus now on implementing OFCons

26

OFCons

- Not subject to FLP impossibility!
- Can be implemented in fully asynchronous system
  - ✓Using the correct-majority assumption
  - ✓Or read-write
- Synod OFCons: a 2-phase algorithm

27

Synod OFCons I

```
Code of every process pi:

Initially:
   ballot:=i-n; proposal:=nil; readballot:=0; imposeballot:=0;
   estimate:= nil; states:=[nil,0]^n

upon ‹ofcons, propose, v›
   proposal := v; ballot:=ballot + n; states:=[nil,0]^n
   send [READ, ballot] to all

upon receive [READ,ballot'] from pj
   if readballot ≥ ballot' or  imposeballot ≥ ballot' then
           send [ABORT, ballot'] to pj
   else
           readballot:=ballot'
           send [GATHER, ballot', imposeballot, estimate] to pj

upon receive [ABORT, ballot] from some process
   return abort
```

28

Synod OFCons II

```
upon receive [GATHER, ballot, estballot, est] from pj
  states[pj]:=[est,estballot]

upon  #states ≥ majority
  if ∃ states[pk]≠[nil,0] then
     select states[pk]=(est,estballot) with highest estballot
     proposal:=est;
  states:=[nil,0]^n
  send [IMPOSE, ballot, proposal] to all

upon receive [IMPOSE,ballot',v] from pj
   if readballot > ballot' or  imposeballot > ballot' then
          send [ABORT, ballot'] to pj
  else
         estimate := v; imposeballot:=ballot'
         send [ACK, ballot'] to pj
```

29

Synod OFCons III

```
upon received [ACK, ballot] from majority
  send [DECIDE, proposal] to all

upon receive [DECIDE, v]
   send [DECIDE, proposal] to all
   return [decide, v]
```

30

Correctness

- Validity
- Agreement (try to do it yourselves)
  - ✓When is the decided value determined?
- OF Termination
  - ✓Show that a correct process that proposes either decides or aborts
  - ✓If a single process keeps going
    - It will eventually propose with a highest ballot number not seen so far
    - This process will not abort with such a ballot number

Original Synod algorithm [Lamport 98]

- Further optimizations
  - ✓Less modular
- Misses explicit aborts of SynodOFC
  - ✓Process simply do not answer to old ballots
- Assumes eventually reliable links
  - ✓Messages are not retransmitted
  - ✓Cannot assume that a majority will be gathered in every ballot

Synod I

```
Code of every process pi

Initially:
   ballot:=i-n; proposal:=nil; readballot:=0; imposeballot:=0;
   estimate:= nil; decided:=false; states:=[nil,0]^n

upon ‹cons, propose, v›
   repeat periodically
         if self=leader() then
            proposal := v; ballot:=ballot + n; states:=[nil,0]^n
            trigger ‹bebBroadcast, [READ, ballot]›
   until decided

upon ‹bebDeliver, pj, [READ,ballot’]›
   if readballot < ballot’ and imposeballot < ballot’ then
         readballot:=ballot’
         send [GATHER, ballot’, imposeballot, estimate] to pj
```


Synod II

```
upon receive [GATHER, ballot, estballot, est] from pj
   states[pj]:=[est,estballot]

upon  #states ≥ majority
   if ∃ states[pk]≠[nil,0] then
         select states[pk]=(est,estballot) with highest
   estballot
         proposal:=est;
   states:=[nil,0]^n
   trigger ‹bebBroadcast, [IMPOSE, ballot, proposal]›

upon ‹bebDeliver, pj, [IMPOSE,ballot’,v]›
    if readballot ≤ ballot’ and imposeballot < ballot’ then
         estimate := v; imposeballot:=ballot’
         send ‹[ACK, ballot’, v]› to ALL
```


34

Synod III

```
upon received [ACK, ballot, v] from majority and not(decided)
   trigger ‹cons, decide, v›   //do not return
   decided:=true
   periodically send ‹[DECIDE, v]› to all
```


35

Time Complexity

- Fault-free time complexity: 4
  - \+ 1 communication step for decision relaible broadcast
- Optimizations
  - ✓Getting rid of the first READ phase
- Allow a single process (presumed leader, say p1) to skip the READ phase in its 1st ballot
  - ✓Reduces fault-free time complexity to 2

36

From Synod to Paxos

- Paxos is a state-machine replication (SMR) protocol
  - ✓i.e., a universal construction given a sequential object
- Implemented as totally-ordered broadcast:
  - ✓Exports one operation toBroadcast(m) and issues toDeliver(m') notifications
  - ✓Every message m (to)broadcast by a correct process pi is eventually (to)delivered by pi
  - ✓Every message m delivered by a process pi is eventually delivered by every correct process
  - ✓No message is delivered unless it was previously broadcast
  - ✓No message is delivered twice
  - ✓The messages are delivered in the same order at all processes

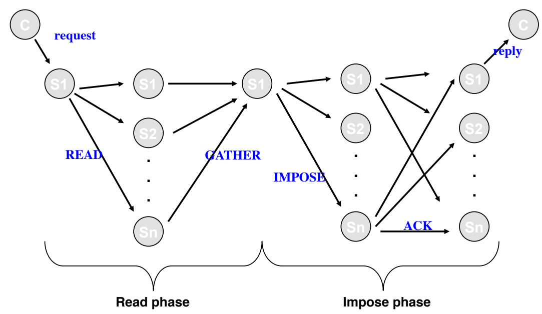
From Synod to Paxos

- But consensus (Synod) is one shot…
  - ✓How to most efficiently transform Synod to toBroadcast (Paxos)?

Paxos SMR

- Clients initiate requests
- Servers run consensus
  - ✓Multiple instances of consensus (Synod)
  - ✓Synod instance 25 to agree on the 25th request to be ordered
- Both clients and servers have the (unreliable) estimate of the current leader (some server)
- Clients send requests to the leader
- The leader replies to the client

Paxos Failure-Free Message Flow



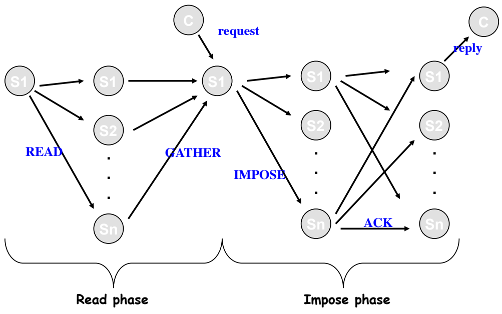
Observation

- READ phase involves no updates/new consensus proposals
  - ✓Makes the leader catch up with what happened before
- Most of the time the leader will remain the same
  - ✓+ nothing happened before (e.g., new requests)

Optimization

- Run READ phase only when the leader changes
  - ✓and for multiple Synod instances simultaneously
- Use the same ballot number for all future Synod instances
  - ✓run only IMPOSE phases in future instances
  - ✓Each message includes ballot number (from the last READ phase) and ReqNum, e.g., ReqNum = 11 when we're trying to agree what the 11th operation should be
- When a process increments a ballot number it also READs
  - ✓e.g., when leader changes

Paxos Failure-Free Message Flow



43 43

Potential Issues?

- Holes/Gaps detected in the READ phase
  - ✓The leader detected a value in READ/GATHER for requests 1-12, 14, and 17
  - ✓but not for 13, 15 and 16
- The leader than runs the IMPOSE phase for instances 13, 15 and 16 with a special proposal
  - ✓A noop value ("do nothing")

44 44

What's next? Handling CAP

- Paxos provides strong consistency
  - ✓All servers (replicas) witness the same state evolution
  - ✓Liveness assuming the eventual leader (or eventual synchrony) may not be sartisfactory
  - ✓Especially for large-scale (geo) replication
- Eventual consistency
  - ✓ Assuming no more updates, all replicas eventually converge to the same state
  - ✓Simple and efficient
  - ✓Amazon's Dynamo
  - ✓Too weak?
- Causal consistency
  - ✓+ Causally related [Lamport 78] events are observed in the same (causal) order
- In real systems:
  - ✓A mixture of all this ☺

 45

Bibliographic project

- Team of two: 10 mins presentation of a research paper + 5 mins discussion
  - ✓What is the problem? What is its motivation?
  - ✓What is the idea of the solution?
  - ✓What is new and what is interesting here?
    - Technical details: unnecessary
- Final grade = 1/3 for the presentation (April 30, May 5 and 6) + 2/3 written exam (May 7)
- The list of papers (with pdfs) and the link to a form to submit your choice:
  - ✓http://perso.telecom-paristech.fr/~kuznetso/INF346/
  - ✓Bid the papers ASAP

 46