Advanced Blockchain Engineering (ASCI a27)

Chapter 5. State Machine Replication

- Paxos
- PBFL
- Zyzzyva

Dick H.J. Epema
Distributed Systems Group

April 2018
State Machine Replication (1/2)

- **Fault-free** centralized operation
  - a single server maintains a **state machine** (e.g., a data store)
  - clients issue **requests** to the server (e.g., reading and writing)
  - server **serializes** and executes commands

- In the **face of faults** or **poor performance**
  - replicate the server: **State Machine Replication** (SMR)
  - have the replicas execute **the same client requests in the same order**: servers have to **achieve consensus** on the log of client requests (linearizability)
State Machine Replication (2/2)

• Potential types of faults:
  – stopping / pausing
  – malicious (due to explicit attacks or software errors)

• Models are usually assumed to be **asynchronous**
  – sometimes weaker timing assumptions
  – may lead to livelock

• Three algorithms:
  – Paxos (stopping failures)
  – Practical Byzantine Fault Tolerance (Byzantine failures)
  – Zyzzyva (Byzantine failures)
Paxos: system properties

- Processes
  - may **stop** (crash) and **restart** (amounts to pausing)
  - processes need some **stable storage** in order to rejoin
  - but are **not malicious** (not Byzantine)
- Messages may be
  - lost
  - duplicated
  - received out of order
  - but **not corrupted**
- Processes **may refrain from sending** any message prescribed by the algorithm
Single-value Paxos: problem statement

• Processes can propose a value
• Processes have to achieve consensus:
  o agreement: they choose the same value
  o validity: they choose a value that has been proposed
  o termination: impossible (FLP result)
• Validity trivially satisfied in algorithm
• To still achieve progress/termination:
  o use election and/or some form of timing assumption
• Agreement (consistency) is the big problem
The Paxos metaphor (1/2)

- Introduced by Leslie Lamport
- The parliament of Paxos has to agree on a sequence of decrees
- Members of parliament ("priests") can propose a ballot on a decree
- Potentially multiple ballots needed to reach a decision on a single decree
- Members can vote in a ballot, or abstain (not vote)
- Members of parliament communicate via messengers
- Messengers are reliable, but may take an arbitrary time (possibly forever) to deliver a message
The Paxos metaphor (2/2)

- Members of parliament can walk into and out of the Chamber at any time (they can “pause”)
- While in parliament, members are cooperative
- Each member of parliament has a ledger in which he writes the accepted decrees he is aware of
- The synod algorithm: The members have to agree on a single decree
- Distributed system interpretation: distributed, fault-tolerant implementation of a database
The synod algorithm (1/3): ballots

- A series of numbered ballots is executed
- Priests can vote in favor or abstain (but not vote against)
- Possibly multiple ballots for the same decree
- A ballot $B$ consists of:
  - a decree $B_d$
  - a quorum $B_q$
  - the set $B_v$ of priests who vote for the decree
  - a ballot number $B_n$

  - A ballot is **successful** if $B_q$ is a subset of $B_v$ (whole quorum votes in favor)
The synod (2/3): correctness conditions

- A set of ballots guarantees consistency and progress if:
  1. every ballot has a **unique number**
  2. the quorums of any two ballots **overlap**
  3. for every ballot, if any priest in its quorum voted in an earlier ballot, **then the decree of the ballot is equal to the decree of the latest of those earlier ballots**

- For condition 1, use lexicographic ordering on increasing (integer, priest) id pairs
- For condition 2, use strict majorities
- The problem now is to find a protocol that makes the set of ballots satisfy condition 3
The synod (3/3): example

<table>
<thead>
<tr>
<th>number</th>
<th>decree</th>
<th>quorum and voters</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>α</td>
<td>A      B    C    D</td>
</tr>
<tr>
<td>5</td>
<td>β</td>
<td>A      B    C    E</td>
</tr>
<tr>
<td>14</td>
<td>α</td>
<td>B      D    E</td>
</tr>
<tr>
<td>27</td>
<td>β</td>
<td>A      C    D</td>
</tr>
<tr>
<td>29</td>
<td>β</td>
<td>B      C    D</td>
</tr>
</tbody>
</table>

- For ballot 29:
  - B voted in ballot 14
  - C voted in ballots 5 and 27
  - D voted in ballots 2 and 27
  - Latest of these ballots: 27
  - Condition 3: Ballots 27 and 29 have the same decree
Single-value Paxos: types of processes

- **Proposers**: the processes that propose values (e.g., the clients)
- **Acceptors**: the processes that choose a value (the servers)
- **Learners**: the processes that learn the result (clients and servers)
- Processes may play multiple roles
- Alternatively, have coordinators or leaders collecting values from proposers and proposing on their behalf
Single-value Paxos: main idea 1

- The algorithm proceeds in **rounds**
  - identified by an integer
  - round numbers are **pre-assigned to proposers** in which only they propose (e.g., round number modulo N = process id)
  - multiple rounds may be executed **simultaneously, out of order, and may be skipped** altogether
  - all messages are identified by a round number
  - a process **ignores messages** that carry a round number smaller than the largest it knows about
Single-value Paxos: main idea 2

- In each round, the algorithm proceeds in **two phases** with send-reply from proposers to acceptors and back

  - **Phase 1:**
    - a proposer tries to build a **majority of acceptors** without proposing an actual value by asking them to participate
    - the acceptors reply with **the last values they voted for and the corresponding round numbers**

  - **Phase 2:**
    - a proposer proposes to a majority of the acceptors **the last value voted for among a majority of acceptors** *(condition 3!!)*
    - the acceptors **vote positively for this value or abstain**
Data structures

- Data structures in a **proposer**:
  - prnd the highest-numbered round it has started
  - pval the value it proposes in that round

- Data structures in an **acceptor**:
  - rnd highest-numbered round in which it has participated
  - vrnd highest-numbered round in which it has voted
  - vval the value for which it voted in round vrnd
The algorithm: phase 1 of a round (1/2)

I. A proposer does a request_to_participate:

i := new round number
prnd := i
pval := nil  /* only later pick a value to propose */
send(request_to_participate,i) to a set of acceptors
   /* start building a majority */
The algorithm: phase 1 of a round (2/2)

II. An acceptor receives a request_to_participate

upon receipt of (request_to_participate,i) do
  if (i>rnd) then /* only react to new round numbers */
    rnd := i
    send(participate,i,vrnd,vval) to proposer /* send latest vote */
    /* acceptor promises to do only later votes */
The algorithm: phase 2 of a round (1/2)

III. A proposer does a request_to_vote

upon receipt of (participate,i,vrnd,vval) from majority Q do
    round = max{vrnd} in messages from Q /* condition 3 */
    if (round=0) then pval := any value /* no previous votes */
    else pval := vval in message from Q with vrnd=round
send(request_to_vote,i,pval) to acceptors in Q
The algorithm: phase 2 of a round (2/2)

IV. An acceptor receives a request_vote message

upon receipt of (request_to_vote,i,v) do
  if ( (i ≥ rnd) and (vrnd ≠ i) ) then
    rnd, vrnd := i
    vval := v
    send(vote,i,v) to all learners
/* learner learns value if it has received a vote from a majority */
Consistency (1/2)

• **To prove:**
  
  *When a proposer proposes value v in round i, in no previous round another value than v has been or still can be decided*

• **Proof:**
  
  o suppose pval=v and v was last voted for in round k<i
  o consider any round number j, with j<i
  o **Case 1:** k<j<i  
    Because a majority Q of acceptors told the proposer that their latest vote was in round k and that they would only do votes later than round i in the future, no value can have been or will be decided in round j.
Consistency (2/2)

- **Proof (ct’d):**
  - **Case 2:** j=k
    - Trivial: in any round, only a single value can be up for decision because a majority of acceptors votes for it
  - **Case 3:** j<k
    - Use induction

![Diagram showing Q', Q, and Q with j, k, and i points]
Livelock

• **Problem:** livelock is possible

• **Exercise:** create an example of livelock

• **Solution:** run an election algorithm to select a coordinator/leader that does the proposals

• Only when re-elections are in progress, potential delay/confusion
Multi-value Paxos

• Run an election algorithm to select a single leader for all slots in the sequence to be decided
• This leader can start out with creating a majority of acceptors for all slots
• For all but a finite number, no previous votes from acceptors, so the leader can propose any value from the proposers
• After a new leader has been elected, there may be gaps in the sequence that has been decided so far
• For those gaps, propose an “empty” proposal (“the sky is blue”) in order to avoid confusion
Criticism of Paxos

- Difficult to understand (especially multi-decree Paxos)
- Single-decree vs multidecree decomposition is not the right one
- Other proposals for decomposition:
  - leader election
  - log replication
- Difficult to implement in real systems (GFS, HDFS, etc)
PBFT and Zyzzyva (1/5): assumptions

- Handle **independent** Byzantine node failures of replicas
- **Adversary** cannot break collision-resistant hashes, encryption, signatures
- Also clients may exhibit failures (use admission control)
- Use message digests and signatures
- Provide **safety**: linearizability (does not depend on synchrony)
- Provide **liveness**: assume weak synchronicity: message delays grow at most linearly with time/system is synchronous for periods of time
PBFT and Zyzzyva (2/5): views and data

• At every moment, there is a view
  – with a primary
  – with the other nodes as backups
  – view number $v$ has primary $p = v \mod n$

• Replica data structures
  – state machine
  – view number
  – message log
  – checkpoints
PBFT and Zyzzyva (3/5): similarities

• **Algorithm structure**
  – agreement protocol
  – checkpoint protocol
  – view-change protocol

• **Checkpoints**
  – maintain history
  – stable checkpoints: truncate history
  – limit range of request numbers considered
PBFT and Zyzzyva (4/5): differences

• **PBFT:**
  – achieves consensus on request order with a 3-phase protocol among replicas
  – “a correct server only emits replies that are stable”

• **Zyzzyva:**
  – faster speculative execution with larger burden on the clients
  – builds support among backups for view changes
  – “a correct client only acts on replies that are stable”
PBFT and Zyzzyva (5/5): number of faulty nodes

- Required: $n > 3f$
- When waiting for reply from all replicas, proceed after $n-f$ replies
- These replies may happen to contain replies from all $f$ failing replicas
- Replies from non-faulty ones should out-number replies from faulty ones: $n-2f > f$

- Replicas numbered 0,1, …, $n-1$
- Assume $n = 3f+1$
PBFT: overview of normal operation

1. **Client sends request** to the primary (with logical time stamp)
2. **Primary assigns sequence number and multicasts request** to backups
3. **Replicas** execute the request and **reply** to the client
4. **Client waits for f+1 replies** with the same result
PBFT: overview of view change

- If a client **does not receive** \( f+1 \) identical replies soon enough, it sends its request to all replicas.

- A replica then
  - re-sends its reply to the client, if it has already processed the request
  - otherwise it sends the request to the primary

- If the primary then does not multicast the request to the replicas, it is **suspected of failure**

- Replicas then **initiate a view change**

- **The new view is announced** by its primary
PBFT: normal operation

- Three-phase protocol (three types of messages):
  - pre-prepare + prepare phases: totally order requests in the same view
  - prepare + commit phases: totally order requests across views
- All three types of messages contain a view and a request number
PBFT: accepting a pre-prepare

• A backup accepts a pre-prepare message if:
  – it is in the same view
  – it has not accepted a pre-prepare with the same view and sequence number
• It then enters the prepare phase and broadcasts a prepare message
• The predicate \(\text{prepared}(m,v,n,i)\) is true if replica \(i\) has entered into its message log:
  – the request
  – the corresponding pre-prepare message
  – \(2f\) corresponding prepare message from other replicas
• **Assertion:** if \(\text{prepared}(m,v,n,i)\) is true for a correct replica \(i\), then \(\text{prepared}(m',v,n,j)\) is false for any \(m \neq m'\) and any correct \(j\)
PBFT: commit

- When prepared(...,i) is true, replica i broadcasts a commit message
- Predicate committed(m,v,n) is true if prepared(m,v,n,i) is true in at least \( f + 1 \) correct replicas
- Predicate committed-local(m,v,n,i) is true if prepared(m,v,n,i) is true and replica i has accepted \( 2f+1 \) commit messages (then execute request)
- **Assertion:** if committed-local(...,i) is true in some correct replica i, then committed(…) is true
- **Consequences:**
  - correct replicas agree on the sequence numbers of requests even if they commit locally in different views
  - a request that commits locally at a correct replica, does so in at least \( f+1 \) correct replicas
PBFT: checkpoints

- **Checkpoint:**
  - state after the execution of a multiple of K requests

- **Stable checkpoint:**
  - a checkpoint with a “proof”

- Replicas broadcast *checkpoint messages* with *sequence number* the last request represented in the checkpoint

- **Proof of correctness of a checkpoint:**
  - 2f+1 matching checkpoint messages

- Upon a checkpoint becoming stable:
  - discard previous checkpoints
  - discard all messages related to earlier requests
PBFT: view change

- When in view $v$ the timer of a backup expires, it broadcasts a view-change message with parameters:
  - $v+1$
  - the sequence number $n$ of the last stable checkpoint $s$ it knows
  - a set of $2f+1$ checkpoint messages proving the correctness of $s$
  - for every request $m$ that prepared at $i$ with request number higher than $n$, the corresponding pre-prepare message and $2f$ prepare messages ("the message log after the last stable checkpoint")
PBFT: new view

• When the primary of view v+1 receives 2f view-change messages, it broadcasts a **new-view message with parameters**:
  – v+1
  – the set of view-change messages received by the primary
  – a set of pre-prepare messages derived from the view-change messages received to cause requests that may be missing at some replicas to be executed
• The primary then enters view v+1
• When a backup receives a **new-view message**
  – it derives from the pre-prepare messages in it and from its own message log on which of these messages it still has to act
  – it may have to retrieve requests or checkpoints from other replicas
Zyzzyva (1/3): fast case

- A client sends its request to the primary
- The primary assigns a sequence number and forwards the request to the backups
- The client receives $3f+1$ matching responses from the replicas
Zyzzyva (2/3): two-phase case

- The client receives between $3f$ and $2f+1$ matching responses from the replicas.
- The client sends a **commit certificate** to all replicas.
- The client waits for $2f+1$ acks of the certificate from the replicas.
Zyzzyva (3/3): view-change case

- The client receives fewer than $2f+1$ matching responses from the replicas
- The client sends its request to all replicas
- If the primary still reacts, fine
- Otherwise the backups will invoke a view change
Zyzzyva: view change

- **A replica initiates a view change** by sending an **I-hate-the-primary** message to all replicas.
- When a replica receives **f+1** such messages, it
  - commits to the view change
  - sends the i-hate-t-p messages plus a **view-change** message to all replicas.
- When **the new primary receives 2f+1** view-change messages, it
  - broadcasts a **new-view** message,
  - along with the supporting view-change messages.
- When **a replica receives 2f+1** view-change messages, it
  - waits for a new-view message
  - if that takes too long, it initiates a change to view v+2