PIG AND HIVE
TWO HIGHER-LEVEL APPROACHES TO PROCESS DATA WITH MAPREDUCE
MapReduce

• Remember slides on pros and cons of MapReduce, particularly criticism (too low level,...)
• We have seen how to code joins in MR
• How to filter (grep!), group by, ...

• Now: look at higher-level ”tools” on top of MapReduce
• Why? Claim: MapReduce too low level for “normal” users (developers) + large effort for ad hoc queries.
Pig & Pig Latin

- High-level tool for expressing data analysis programs, originated from Yahoo (now at Apache)
- Compiler transforms query into sequence of MapReduce jobs
- **Data Flow** language, **Pig Latin** (not really something like SQL)
- [http://pig.apache.org](http://pig.apache.org)

Pig Latin Commands:

A = LOAD 'input' AS (x, y, z);
B = FILTER A BY x > 5;
STORE B INTO 'output';

Parsing, logical optimization.
Creation of MapReduce jobs + running them.
Example

Input, e.g., using Shell:

```
grunt> ....
```

Commands like:

```plaintext
A = LOAD 'input' AS (x, y, z);
B = FILTER A BY x > 5;
STORE B INTO 'output';
```

Pig operates directly over files (and other sources, if specified by user defined functions (UDFs)).
(Nested) Data Model

• **Atom:**
  – int, double, chararray, etc.
  – E.g., ‘Big Data Analytics’, ‘Behrend’

• **Tuple:**
  – sequence of fields (any types) ( ..., ..., ..., ... )
  – E.g., (‘Big Data Analytics’, 2016, {(1,2,3)})

• **Bag:**
  – collection of tuples (multiset, i.e., can have duplicates)
  – E.g., {('BDA16', 'DBS16')}

• **Map:**
  – Mapping of keys to values
  – E.g., {'Behrend' => {'BDA16'}, 'Brass'=>{'DBS16'}}
Pig Latin: Example: Joins

- A
  - (2, Tie)
  - (4, Coat)
  - (3, Hat)
  - (1, Scarf)

- B
  - (Joe, 2)
  - (Hank, 4)
  - (Ali, 0)
  - (Eve, 3)
  - (Hank, 2)

A = LOAD ......; B = LOAD ......
C = Join A BY $0, B BY $1
Also support for OUTER JOINS
Data with Associated Schema

PARTS = LOAD 'hdfs:///user/hduser/testjoin/parts.txt' as (id: int, name: chararray);

PEOPLE = LOAD 'hdfs:///user/hduser/testjoin/people.txt' as (name: chararray, partsid: int);
Pig Latin: Commands (Subset)

- LOAD, STORE, DUMP
- FILTER
- FLATTEN
- FOR EACH
- GENERATE
- (CO)GROUP
- CROSS
- JOIN
- ORDER BY
- LIMIT

PLUS: Built-in and user-defined functions (UDFs)

http://wiki.apache.org/pig/PigLatin

Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, Andrew Tomkins: Pig latin: a not-so-foreign language for data processing. SIGMOD Conference 2008: 1099-1110
Example: Word Count

//LOAD input file from HDFS
A = LOAD 'hdfs:///user/hduser/gutenberg' AS (line : chararray);

//Parse input lines into words
B = FOREACH A GENERATE FLATTEN(TOKENIZE(line)) as term;

//Remove whitespace-only words
C = FILTER B BY term MATCHES '\\w+';

//Group by term
D = GROUP C BY term;

//and count for each group (i.e., for a term) its occurrences
E = FOREACH D GENERATE group, COUNT($1) as frequency;

//ORDER by frequency of occurrence
F = ORDER E BY frequency ASC;
Example: Word Count (Cont’d)

Output:

.....
....
(which,2475)
(it,2553)
(that,2715)
(a,3813)
(is,4178)
(to,5070)
(in,5236)
(and,7666)
(of,10394)
(the,20592)

2013-05-15 10:02:21,062 [main] INFO
org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.MultiQueryOptimizer - MR plan size after optimization: 3

.....

Counters:
Total records written : 17875
Total bytes written : 178274
...

Job DAG:
job_201305031236_0051 -> job_201305031236_0052,
job_201305031236_0052 -> job_201305031236_0053,

Logically, multiple connected MapReduce jobs form a DAG*

*) DAG = Directed Acyclic Graph
(CO)GROUP Example

• Consider the following data (as CSV input)

owners.csv
adam,cat
adam,dog
alex,fish
alice,cat
steve,dog

• And the following PIG script

owners = LOAD 'owners.csv'
    USING PigStorage(',,')
    AS (owner:chararray,animal:chararray);

grouped = COGROUP owners BY animal;
DUMP grouped;

This returns a list of animals. For each animal, Pig groups the matching rows into bags

<table>
<thead>
<tr>
<th>group</th>
<th>owners</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>{(adam,cat),(alice,cat)}</td>
</tr>
<tr>
<td>dog</td>
<td>{(adam,dog),(steve,dog)}</td>
</tr>
<tr>
<td>fish</td>
<td>{(alex,fish)}</td>
</tr>
</tbody>
</table>

http://joshualande.com/cogroup-in-pig/
(CO)GROUP of Two Tables

owners.csv
adam,cat
adam,dog
alex,fish
alice,cat
steve,dog

pets.csv
nemo,fish
fido,dog
rex,dog
paws,cat
wiskers,cat

owners = LOAD 'owners.csv'
USING PigStorage(',,')
AS (owner:chararray,animal:chararray);

pets = LOAD 'pets.csv'
USING PigStorage(',,')
AS (name:chararray,animal:chararray);

grouped = COGROUP owners BY animal, pets by animal;
DUMP grouped;

What is the difference to a Join?

<table>
<thead>
<tr>
<th>group</th>
<th>owners</th>
<th>pets</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>{(adam,cat),(alice,cat)}</td>
<td>{(paws,cat),(wiskers,cat)}</td>
</tr>
<tr>
<td>dog</td>
<td>{(adam,dog),(steve,dog)}</td>
<td>{(fido,dog),(rex,dog)}</td>
</tr>
<tr>
<td>fish</td>
<td>{(alex,fish)}</td>
<td>{(nemo,fish)}</td>
</tr>
</tbody>
</table>
User Defined Functions in PIG

- Can write your custom UDF in Java and directly use it in PIG
- Here for a simple “eval” function:

```java
REGISTER myudfs.jar;
A = LOAD 'student_data' AS
    (name:chararray, age: int, gpa: float);
B = FOREACH A GENERATE
    myudfs.UPPER(name);
```

```java
public class UPPER extends EvalFunc<String>
{
    public String exec(Tuple input) throws IOException {
        if (input == null || input.size() == 0)
            return null;
        try {
            String str = (String)input.get(0);
            return str.toUpperCase();
        } catch(Exception e){
            throw new IOException("Caught exception processing input row ", e);
        }
    }
}
```

Optimizations

• **Logical Optimization:**
  – Filter as early as possible
  – Eliminate unnecessary information (project)
  – ...

• **Multiple MapReduce jobs** (in general, not only here in Pig) give possibilities to optimize execution order.

• Considering DAG dependencies!

• Reusing stored outputs of previous
Pig vs. Native MapReduce

Two sides of the coin (generally). Statement from Twitter engineer in 2009.

“...typically a Pig script is 5% of the code of native map/reduce written in about 5% of the time. “

“However, queries typically take between 110-150% the time to execute that a native map/reduce job would have taken.”

Pig Latin vs. SQL

• **Pig Latin** is a data flow programming language
  – user specified operation(s) put together to achieve task (imperative)

• **SQL** is declarative
  – user specifies what the result should be, not how it is implemented
Pig vs. RDBMS

• **RDBMS:**
  – tables with predefined schema
  – support of transactions and indices
  – aim at fast response time

• **Pig:**
  – schema at runtime (even optional)
  – any source (by applying user defined functions)
  – no loading/indexing of data as pre-processing: data is loaded at execution time (usually from HDFS)
  – like MapReduce (well, Pig is build on top of MR): aim at throughput, not super fast short queries
Hive

• For structured data
• On top of Hadoop (like Pig) and, hence, HDFS

• “RDBMS for big data”
• Query language is similar to SQL (declarative) (not a data flow language as Pig Latin)

• Originated from Facebook’s effort to analyze their data.
• Now, an Apache Project
Hive QL

```sql
SELECT year, MAX(temperature)
FROM records
WHERE temperature != 9999 AND ..... 
GROUP BY year;
```

No full support of SQL-92 standard.

Note: There are various other projects (specifically at Apache) for big data management for various purposes. Have a closer look if you are interested!
Literature


• Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, Andrew Tomkins: Pig latin: a not-so-foreign language for data processing. SIGMOD Conference 2008: 1099-1110

• http://pig.apache.org
• http://wiki.apache.org/pig/PigLatin
• http://hive.apache.org/
Summary MapReduce

• **Programming paradigm** and **infrastructure** for processing large amounts of data in a batch fashion.

• **Two functions, map and reduce** describe how data is processed and aggregated.

• Have seen multiple **application scenarios and corresponding algorithms**.

• **MR** jobs can be connected to workflows

• **PIG** is one way to automatically translate higher-level instructions to MR jobs
NOSQL

HOW TO WRITE A CV

DO YOU HAVE ANY EXPERTISE IN SQL?

NO

geek & poke

DOESN'T MATTER. WRITE: "EXPERT IN NO SQL"

Leverage the NoSQL boom
Example Key/Value Store: Redis

- [http://try.redis.io/](http://try.redis.io/) <= check this out!

```
SET name "ddm15"
GET name                 # "ddm15"

LPUSH list “a”
LPUSH list “b”
LLENGTH list           # 2
LRANGE list 0 1          # “b”, “a”
```
NoSQL: Wide Spectrum

- Systems come with different properties.
- In-memory vs. disk based.
- ACID vs. BASE
- CRUD, SQL (subset?), or MapReduce support

- **Different systems for specific requirements.** Often triggered by demands inside companies.
- Like Voldemort at Linkedin, BigTable (also Hadoop) at Google, etc.
Overview of Forthcoming Topics

• Fault Tolerance.
• Pessimistic Replication, Optimistic Replication
• Consistency: ACID vs. Base, CAP Theorem
• Placement of data/nodes in network: Consistent hashing.
• Ordering of events: Vector Clocks
• Will look at sample systems, with hands-on experience through exercises.
Wanted Properties

• Data should be always consistent

• Provided service should be always quickly responding to requests

• Data can be (is) distributed across many machines (partitions)

• **Fault Tolerance**: Even if some machines fail, the system should be up and running
FAULT TOLERANCE AND REPLICA MANAGEMENT
Benefits of and Issues with Replication

- Many servers handling replicas of a single object can efficiently serve read/write requests.
- **If one (some) fail, system still is available.**
- The same is true for replicating entire servers (not just Data)
- But....
  - Only if all replicas have the same state (value) reading from one is enough.
  - But how to keep replicas in a **consistent** state?
  - What happens if some replicas are **not reachable**?
  - What if some replicas have **corrupted** data?
  - Need **additional storage** for replicas.
- Not well done replication can mess up performance as well as availability.
Pessimistic and Optimistic Replication

- **Optimistic Replication**: Keep replicated objects with a not enforced synchronization. Thus, replicas can diverge. Need conflict detection and resolution.

- **Pessimistic Replication**: Allow no inconsistencies at all through rigorous synchronization. Single-Copy Consistency.

One-Copy Serializability

• Given a replicated database system.
• A concurrent execution of transactions in a replicated database is one-copy serializable if it is equivalent to a serial execution of these transactions over a single logical copy of the database.

• Strongest correctness criteria for replicated “databases”.
Illustration:

$t_0$ SET $a = 1$

Replica 1

A = 1

A = 2

Replica 2

A = 2

A = 1

t_0$ SET $a = 2$

No
Read one Write All (ROWA)

**Read Operation**

- Client sends read operation.
- This is transformed into a single physical read operation on an arbitrary copy (i.e., read one).

**Write Operation**

- Client sends write operation.
- This is transformed into physical write operations on all copies (i.e., write all).
ROWA Discussion

- Advantages and disadvantages:
  - Straightforward strategy
  - Easy to implement
  - All copies are up-to-date at all times
  - Efficient local read access at each node
  - Update operations depends on the availability of all nodes that have copies
  - Longer runtime for update operations
Primary Copy Strategy (PrimCopy)

• **Principle**
  – Choose one copy as the primary copy
  – All other copies are considered to be derived from the primary copy
  – Performing an update operation requires “locking” the primary copy
  – Read operations at a node can be executed efficiently on the local copy

• **Workflow**
  – Primary copy is locked and updated
  – Primary copy propagates updates to all copies.
Network Partitions

• **Network partitioning**
  – The network is partitioned into two or more sub networks that can no longer communicate with each other

• **Why is this a problem?**
  – Failover to new primary copy or not writing to really all replicas leads to subnetworks
  – These could continue updating data
  => Problems when networks are again joined (re-united): Needs to be consolidated
Aim

• Want approach (system) that
  – consists of multiple nodes (for fault tolerance)
  – no single point of failure
  – single-copy semantics (->no inconsistencies)

• Obviously, nodes can fail, but the system should stay operational.

• Essentially: Nodes have to agree on same actions (same order of commands).
Classes of Failures

• **Byzantine Failures:** The component can exhibit arbitrary and malicious behavior, including perhaps collusion with other faulty components [Lamport et al. ‘82]

• **Fail-stop Failures:** In response to a failure, the component switches to a state that allows other components to detect the failure, and then stops [Schneider ‘84]

Apparently, Byzantine failures can be more disruptive and are, thus, more difficult to handle. Very critical applications should still handle them, but often handling fail-stop failures is enough.
Byzantine Generals Problem

- Imagine several divisions of the Byzantine army camp outside an enemy city.
- Each division has its own general.
- General can communicate via messages.
- Messages can be forged.
Plan of Action and Traitors

• They observe the enemy and **must decide on a common plan of action.**

• However: **Some of the generals might be traitors**, trying to prevent the loyal generals from reaching agreement.

• Idea behind possible solutions:
  – **All loyal generals decide upon the same plan of action.**
  – **A small number of traitors cannot cause the local generals to adopt a bad plan.**
Two Generals’ Problem
aka. two armies problem or the coordinated attack problem

- Variant of problem before
- Armies A1 and A2 want to coordinate attack on army B

- The two generals can communicate through messengers
- Who needs to pass a valley occupied by the enemy.
Two Generals’ Problem (Cont’d)

• The messenger sent by A1 (respectively A2) to reach A2 might get “lost”

• Generals agree on communication, sending time at that they want to attack. E.g., “Attack at dawn”.

• Scenario 1: A1 sends “Attack at dawn”, but messengers gets killed. A2 never gets message. A1 can attack assuming A2 got message?

• Scenario 2: A1 sends “Attack at dawn”, A2 receives messages and replies “Ok”.
Two Generals’ Problem (Cont’d)

• **Scenario 3**: A1 sends “Attack at dawn”, A2 receives message and replies “Ok”. What is A2 doing?
• **Scenario 4**: A1 sends “Attack at dawn”, A2 receives messages and replies “Ok”. A1 receives message and replies “Ok”, .....

http://en.wikipedia.org/wiki/Two_Generals%27_Problem
State Machine

Simple form of a server.

- Set of states
- Set of Inputs
- Set of Outputs
- Transition function (Input x State -> State)
- Output function (Input x State -> Output)
- A special state called Start

Requirements

• Requests from a single client are processed in the order they are created.

• If a request r of a client c is created as a result (consequence) of a request r’ of a client c’, then r’ should be processed before r.
Fault Tolerance

• What does it mean if a system/component is fault tolerant?

• A component is called faulty if its behavior is no longer consistent with its specification. (failure = fail-stop or Byzantine, for instance).
A system consisting of a set of distinct components is $t$ fault-tolerant if it operates as devised provided that not more than $t$ components become faulty.

How does this compare to statistical measures like Mean Time Between Failures (MTBF)?
Fault-Tolerant State Machine

• Can implement fault-tolerant state machine by replicating and running replicas of single state machine.

• Provided that
  – all start in the same state
  – execute the same request in same order
  – all operations are deterministic

they will all do the same thing (-> produce the same output).

• Same order requirement is though part: This requires coordination!
t Fault Tolerance (Cont’d)

• **How many replicas** do we have to keep for the state machine to render it *t fault tolerant*?

• **For fail-stop failures**: $t+1$

• **For Byzantine failures**: $2t+1$
Let’s simply replicate Servers

Things can (and will) get messed up .......

time

Client 1
Server 1
Server 2
Client 2
Naïve Solution

- Add single **coordinator**
- That **serializes all received operations and sends to the replicas**
- **Single point of failure!**
Naïve Solution (Cont’d)

• What if servers crash, come or do not come back to life?

• When is the coordinator acknowledging a write request a client sent?

• Still leaves open problem of having coordinator failing.

• Want: Consensus among nodes, no fixed coordinator.
Consensus Algorithm

- Assume collection of processes that can propose values (e.g., actions, requests)
- A consensus algorithms needs to ensure that only one single value is chosen, and all processes will learn it.

- Basic Question: What operation to execute next?
- So, one application of consensus finding for each operation a node wants to execute.
Replicated State Machine

- Client
- Server
- Consensus module
- Log: x < -3, y < -1, y < -9, ...
- State machine:
  - x: 3
  - y: 9
  - z: 0
Requirements of Consensus

• **Safety**
  – Only a value that has been proposed may be chosen.
  – Only a single value is chosen.
  – A node never learns that a value has been chosen unless it actually has been.

• **Liveness**
  – Some proposed value is eventually chosen.
  – If a value has been chosen, a node can eventually learn the value.
Paxos Consensus Algorithm

• Paxos algorithms by **Leslie Lamport**.

• **Asynchronous, fault-tolerant consensus algorithms.**

• Paper on “Part time parliament” in Greek Island Paxos by Lamport.

• **Paxos is guaranteed to reach agreement with** \(< N/2\) **failing nodes.**

• **But no guarantee on time this agreement takes.**
Assumptions

- Agents keep state in a persistent storage.
- Before sending accept_OK they make the state persistent.
- Agents can act at arbitrary speed, may fail by stopping, and may restart.
- Messages can take arbitrarily long to be delivered, can be duplicated, and can be lost, but can never get corrupted.
Consensus: Roles (Agents)

- The proposers, acceptors, and learners.

- Considered model:
  - Agents **fail by stopping**, and may restart (need logging)
  - Messages can take arbitrarily long to be delivered, can be duplicated, or lost, but they **are not corrupted**.
Setup: Example

\[ w(x) \]

Proposer 1

Proposer 2

Proposer 3

Acceptor 1

Acceptor 2

Acceptor 3

\[ r(y) \]
Naïve Approach: Single Acceptor

• Have only a single acceptor.
• Proposer send proposals (values) to the acceptor.
• The first received proposal is accepted.
• Everyone else agrees on this proposal.

• Not a very good solution in practice. What if the acceptor fails? Single point of failure.
• Obviously, need multiple (all) acceptors.
Majority Required

• Assume now there are 3 acceptors and each one accepts first proposal it receives.

Each proposer tries to get majority of acceptors.

Here, Proposer 1 reaches majority!

Thick arrows indicating proposal reaching acceptor first.
Majority Required (Diff. Case)

• But can also happen that no majority is achieved.

How is it avoided that system is blocking now?

Thick arrows indicating proposal reaching acceptor first.
Sequence Numbers
aka. proposal numbers

• Proposal is sent to all acceptors in a message called “prepare”, together with a sequence number.

• Sequence number is created at proposer, unique (no two proposers use the same), e.g., through clock

• Meaning: Please accept the proposal with this number.

• Sequence number is used to differentiate between older and newer proposals.
Sequence Numbers (Cont’d)

• Because sequence numbers are unique, we for sure can tell which proposal is “newer”
• At the acceptor: Is the incoming sequence number the highest ever seen?
• Then, promise to not accept any older proposals (i.e., promise message)
**Paxos (Made Simple): Phase 1**

**Phase 1.** (a) A proposer selects a proposal number \( n \) and sends a *prepare* request with number \( n \) to a majority of acceptors.

(b) If an acceptor receives a *prepare* request with number \( n \) greater than that of any *prepare* request to which it has already responded, then it responds to the request with a promise not to accept any more proposals numbered less than \( n \) and with the highest-numbered proposal (if any) that it has accepted.

**Paxos (Made Simple): Phase 2**

Phase 2. (a) If the proposer receives a response to its prepare requests (numbered $n$) from a majority of acceptors, then it sends an accept request to each of those acceptors for a proposal numbered $n$ with a value $v$, where $v$ is the value of the highest-numbered proposal among the responses, or is any value if the responses reported no proposals.

(b) If an acceptor receives an accept request for a proposal numbered $n$, it accepts the proposal unless it has already responded to a prepare request having a number greater than $n$. 
Acceptor State

• Each Acceptor keeps

  **Np:** Highest proposal number received so far
  **Na:** Highest proposal number accepted
  **Va:** Proposal value corresponding to Na
Paxos Pseudo Code

Propose(V):
    chose unique N, N>Np
    send Prepare(N) to all nodes
    if Prepare_OK(Na, Va) from majority:
        V* = Va with highest Na, or V if none
        send Accept(N, V*) to all nodes
    if Accept_OK(N) from majority:
        send Decided(V*) to all

Prepare(N):
    if N>Np:
        Np = N
        reply Prepare_OK(Na, Va)

Accept(N,V):
    if N>=Np:
        Na = N, Va = V
        reply Accept_OK(Na, Va)

Acceptor State:
    Np = 
    Na = 
    Va = 

Paxos Example (1)

Proposer 1

Proposer 2

Acceptor 1

Acceptor 2

Acceptor 3

Np =
Na =
Va =

Np =
Na =
Va =

Np =
Na =
Va =
Paxos Example (2)

- Proposer 1
- Proposer 2
- Acceptors 1, 2, 3

Proposer 1 sends prepare(10) to Acceptors 1, 2, 3.
Paxos Example (3)

Acceptors promise to not accept any proposal with number small than

Proposer 1

Proposer 2

Acceptor 1

prepare_ok(10)

Acceptor 2

prepare_ok(10)

Acceptor 3

prepare_ok(10)

Np = 10
Na = Va =

Np = 10
Na = Va =

Np = 10
Na = Va =
Paxos Example (4)

Proposer 1

Proposer 2

REJECT Since 9 < 10

acceptor 1

acceptor 2

acceptor 3

Np = 10
Na = 
Va =

Np = 10
Na =
Va =

Np = 10
Na =
Va =

prepare(9)
Paxos Example (5)

Proposer 1

Proposer 2

Acceptors:
- Acceptor 1
- Acceptor 2
- Acceptor 3

prepare(11)

N_p = 10
N_a = 
V_a = 

N_p = 10
N_a = 
V_a = 

N_p = 10
N_a = 
V_a =
Paxos Example (6)

Proposer 1

Proposer 2

Acceptor 1

prepare_ok(11)

prepare_ok(11)

prepare_ok(11)

Acceptor 2

Acceptor 3

Acceptors promise to not accept any proposal with number small than

Np = 11
Na = Va =
Proposer 1 that got the promise in the first round now tries to get proposal accepted.

Proposer 1

accept(10,X)

Acceptor 1

Np = 11
Na = 
Va = 

Proposer 2

accept(10,X)

Acceptor 2

Np = 11
Na = 
Va = 

Proposer 2

accept(10,X)

Acceptor 3

Np = 11
Na = 
Va = 

Np = 11
Na = 
Va =
The accept message is rejected by the acceptors,
Since in the meantime they gave a promise for sequence number 11.
Paxos Example (9)

Acceptors accept the accept message and update their status plus send back ACK.

Proposer 1

Proposer 2

Acceptor 1
accept(11,Y)

Acceptor 2
accept(11,Y)

Acceptor 3
accept(11,Y)

Np = 11
Na = 11
Va = Y

Np = 11
Na = 11
Va = Y

Np = 11
Na = 11
Va = Y
Proposer 3 gets now active (possibly after crash) and issues a proposal with sequence number 12 and value Z. What happens?
Learning Accepted Values

• Once an acceptor accepts a value it can broadcast it to the “learners” (all nodes).

• (or using some more efficient dissemination plan)

• Learners have to make sure to insists on a majority of acceptors.
Multi-Paxos

• Remember that we wanted to build a replicated state machine (to render it fault tolerant).

• The Paxos algorithm described, is executed to reach a single consensus.

• To realize a distributed state machine, multiple rounds of Paxos are executed to decide on the individual commands.

• There is also a generalized Paxos that allows operations to be accepted in any order (if they are commutative), think: non conflicting operations in DBs, r(x)w(y) ....
Termination of Paxos

• With one proposer it is easy to see that it is not difficult to terminate after some time.

• With two proposers already one can make up a strategy that lets Paxos never terminate.

• What is usually done it to elect a leader “proposer” to assure progress, if multiple rounds of Paxos are executed for the State machine.

• Leader can be elected with Paxos, given, once Acceptors accept a proposal.

• Or, breaking “dueling” proposers by randomized exponential backoff.
Byzantine Paxos

• When not being restricted to fail-stop failures, but Byzantine ones,

• Paxos requires an additional verification round.
Literature

- Leslie Lamport. **Paxos Made Simple.**


