Where Are Your Files?
Where’s your data?
Distributed Storage

- Scales
- Cost-effective: can even be made up of many cheap, low-reliability storage nodes
- Provides reliability via redundancy

Google’s 1st server
Failures Happen

• Nodes (storage/compute) crash
  – Sometimes recover
• Processes are unresponsive (asynchronous)
  – E.g., due to GC stalls
• Networks delay/drop messages (async., lossy)
  – Buffer overflows, config errors, bad NICs
• Networks go down for periods
  – Routing loops, router failures, net maintenance
Anecdotal Evidence

• Microsoft: 40.8 link failures/day
  – 5 min to one week long
  – Path redundancy reduces loss by 43%
• Google: in cluster’s 1st year
  – 5 racks see 50% packet loss
  – 8 net maintenance/year, 30 min loss in 4
  – 3 router failures/year
• Companies report partition post-mortems
  – Netflix, Github, AWS, ....
  – Resulted in split brain
Asynchrony

• Unresponsive node indistinguishable from crashed one
  – Timeout without making sure it’s dead
• Delays indistinguishable from drops
• Perfect failure detection impossible
  – “false suspicions” inevitable
Fault-Tolerant Distributed Storage Model

∞ clients (all can fail)

f can fail (crash)
asynchronous

n servers

Reliable Distributed Storage, Idit Keidar
Fault-Tolerance 101

• Replication
  – Multiple copies (e.g., 3) of each data item
  – Copies on distinct storage nodes

• Disaster recovery
  – Copies geographically dispersed
Emulating Shared Memory

- Can we provide the illusion of reliable atomic shared-memory in a message-passing system?
- In an asynchronous system?
- Where clients and servers can fail?
Shared Memory Emulation

Clients (processes)

Object’s API

Reliable emulation

\[ f < n \text{ can crash} \]

\[ n \text{ servers} \]

Reliable Distributed Storage, Idit Keidar
Simple Read/Write Emulation

Clients (processes)

Read/Write

Reliable register emulation

f < n can crash

n servers

Reliable Distributed Storage, Idit Keidar
Register

- Holds a value
- Can be read
- Can be written
- Interface:
  - int read(); // returns last written value
  - void write(int v); // returns ack
Atomic (Linearizable) Register

- Each API call should –
  - “Take effect”
    - Effect defined by the sequential specification
  - Instantaneously
    - Take 0 time
  - Between its invocation and response
    - Real-time order
    - A pending call (invocation and no response) can either occur after its invocation or not at all
Example 1

write(0)  
write(1)  
read(0)  

linearizable

time
Example 2

write(0)  read(1)  write(1)  read(0)

time
Example 2

write(1) happened after write(0)

not linearizable
Liveness: Wait-Freedom

• Wait-free
  – Every operation by a correct process $p$ eventually completes
  – In a finite number of $p$’s steps
  – Regardless of steps taken/not taken by other processes
Emulating A Register

• Can we emulate a wait-free atomic shared register?
• In an asynchronous system?
• Where clients and servers can fail?
Take I: Failure-Free Case

(No server failures)

• Each server keeps a local copy of the register
• Let’s try state machine replication
• Using atomic broadcast:
  – broadcast(m)
  – deliver(m)
  – Messages are delivered in the same order at all servers
Emulation with Atomic Broadcast (Failure-Free)

• Upon client request (read/write)
  – Broadcast the request

• Upon deliver write request
  – Write to local copy of register
  – If from local client, return ack to client

• Upon deliver read request
  – If from local client, return local register value to client
What If Processes Can Crash?

• Does the same solution work?

• FLP says: no consensus/state machine replication
  – In asynchronous network
  – With crash failures
  – But consensus with eventual synchrony/failure detectors possible (Paxos, ZooKeeper, Raft)
Take II: 1-Reader 1-Writer (SRSW)

• Single-reader – there is only one process that can read from the register
• Single-writer – there is only one process that can write to the register
• The reader and writer are just 2 processes
  – The other $n-2$ processes are there to help

*For simplicity, we eliminate the distinction between clients and servers for now*
Trivial Solution?

• Writer simply sends message to reader
  – When does it return ack?
  – What about failures?

• We want a *wait-free* solution:
  – If the reader (writer) fails, the writer (reader) should be able to continue writing (reading)
ABD: Fault-Tolerant Emulation
[Attiya, Bar-Noy, Dolev 95]

• Assumes up to $f<n/2$ processes can fail
• Main ideas:
  – Store value at majority of processes before write completes
  – read from majority
  – read intersects write, hence sees latest value
Example: Reliable Storage Emulation

Can’t wait forever

f can fail
Example: Reliable Storage Emulation

Write to n-f, i.e., majority
Example: Reliable Storage Emulation
Example: Reliable Storage Emulation

Read from n-f, i.e., majority
Example: Reliable Storage Emulation

Every two majorities intersect
Example: Reliable Storage Emulation

Write to majority
Example: Reliable Storage Emulation

Read from majority
Example: Reliable Storage Emulation

Use timestamps
Example: Reliable Storage Emulation

Return value with biggest timestamp
SRSW Algorithm: Variables

• At each process:
  – $x$, a copy of the register
  – $t$, initially 0, unique tag associated with latest write
SRSW Algorithm: Write

- write(x,v)
  - choose tag > t
  - set x ← v; t ← tag
  - send (“write”, v, t) to all

- Upon receive (“write”, v, tag)
  - if (tag > t) then set x ← v; t ← tag fi
  - send (“ack”, v, tag) to writer

- When writer receives (“ack”, v, t) from majority (counting an ack from itself too)
  - return ack to client
SRSW Algorithm: Read

- read(x,v)
  - send ("read") to all
- Upon receive ("read")
  - send ("read-ack", x, t) to reader
- When reader receives ("read-ack", v, tag) from majority (including local values of x and t)
  - choose value v associated with largest tag
  - store these values in x,t
  - return x
Does This Work?

• Only possible overlap is between read and write
  – why?
• When a read does not overlap any write –
  – It reads at least one copy that was written by the latest write (why?)
  – This copy has the highest tag (why?)
• What is the linearization order when there is overlap between read and write?
• What if 2 reads overlap the same write?
Example

- finds a copy that was written
- but local copy written by read
- write(1) already happened
- read(1)
- read(?)
- write(1)

linearizable
Wait-Freedom

• Only waiting is for majority of responses
• There is a correct majority
• All correct processes respond to all requests
  – Respond even if the tag is smaller
Take III: n-Reader 1-Writer (MRSW)

• n-reader – all the processes can read
• Does the previous solution work?
• What if 2 reads by different processes overlap the same write?
Example

- finds a copy that was written
- does not find a written copy, returns 0
- write(1) already happened
- not linearizable
MRSW Algorithm
Extending the Read

• When reader receives (“read-ack”, v, tag) from majority
  – choose value v associated with largest tag
  – store these values in x,t
  – send (“propagate”, x, t) to all (except writer)

• Upon receive (“propagate”, v, tag) from process i
  – if (tag > t) then set x ← v; t ← tag fi
  – send (“prop-ack”, x, t) to process i

• When reader receives (“prop-ack”, v, tag) from majority
  (including itself)
  – return x
The Complete Read

Phase 1: Read
(read) ("read") ("read-ack", v, t)
Phases
(propagate", v, t)
(prop-ack")

Phase 2: Write-Back
Multi-reader only
(read-ack", v, t)
(propagate", v, t)
Take IV: n-Reader n-Writer (MRMW)

• n-writer – all the processes can write to the register

• Does the previous solution work?
Playing Tag

• What if two writers use the same tag for writing different values?

• Need to ensure *unique* tags
  – That’s easy: break ties, e.g., by process id

• What if a later write uses a smaller tag than an earlier one?
  – Must be prevented (*why?*)
M RPM W Algorithm
Extending the Write

• To perform write(x,v)
  – send ("query") to all
• Upon receive ("query") from i
  – send ("query-ack", t) to i
• When writer receives ("query-ack", tag) from majority (counting its own tag)
  – choose unique tag > all received tags
  – continue as in 1-writer algorithm
• What if another writer chooses a higher tag before write completes?
Phase 1: Read
Multi-writer only

Phase 2: Write

The Complete Write
Can We Emulate *Every* Atomic Object the Same Way?

• We only emulated a read/write object

• Think of a general object type, with some data members and some methods
  – Queue, stack, counter, ...

• Can we support it the same way?
R/W Registers vs. Consensus

- ABD works even if the system is completely asynchronous
- In consensus (e.g., Paxos), there is no progress when there are multiple leaders
- Here, there is always progress – multiple writers can write concurrently
  – One will prevail (which?)
Disk Paxos
Consensus in Shared Memory

• A shared object supporting a method \( \text{decide}_i(v_i) \) returning a value \( d_i \)
• Satisfying:
  – **Agreement**: for all \( i \) and \( j \) \( d_i = d_j \)
  – **Validity**: \( d_i = v_j \) for some \( j \)
  – **Termination**: decide returns
Solving Consensus in/with Shared Memory

• Assume asynchronous shared memory system with *atomic* R/W registers

• *Can we solve consensus?*
  – Consensus is *not* solvable if even one process can fail (shared-memory version of [FLP])
  – Yes, if no process can fail
  – Yes, with eventual synchrony or failure detectors
Shared Memory (SM) Paxos

• Consensus
  – In asynchronous shared memory
  – Using wait-free regular R/W registers
    • As emulated by ABD
  – And leader-election failure detector $\Omega$

• Wait-free
  – *Any* number of processes may fail ($t < n$)
    • Unlike message-passing model
Leader Election Failure Detector

• $\Omega$ – Leader
  
  – Outputs one trusted process
  
  – *Stable* from some point on:
    
    All correct procs. trust the same correct proc.

• Is the weakest for consensus
  
  [Chandra, Hadzilacos, Toueg 96]
Regular Registers

• SM Paxos can use registers that provide weaker semantics than atomicity

• SWMR *regular register*: a read returns
  – Either a value written by an overlapping write
  or
  – The register’s value before the first write that overlaps the read
Regular versus Atomic

Read(1) can return 0

Read(0)

Write(0)

Write(1)

Write(1) already happened

Regular can return 0

Not linearizable
Variables

• Paxos variables are:
  – BallotNum, AcceptVal, AcceptNum

• SM version uses *shared* regular registers:
  – \( x_i = \langle \text{bal}, \text{val}, \text{num}, \text{decision} \rangle_i \) for each process \( i \)
  – Initially \( \langle \langle 0,0 \rangle, \bot, \langle 0,0 \rangle, \bot \rangle \)
  – Writeable by \( i \), readable by all (SWMR)

• Each process keeps *local* variables \( b, v, n \)
  – Initially \( \langle \langle 0,0 \rangle, \bot, \langle 0,0 \rangle \rangle \)
SM Paxos: Phase I

if leader (by $\Omega$) then

$\textbf{b} \leftarrow$ choose new unique ballot

write $\langle \textbf{b}, \textbf{v}, \textbf{n}, \bot \rangle$ to $x_i$

read all $x_j$’s

if some $x_j$.bal $> b$ then start over

if all read $x_j$.val’$s = \bot$ then

$v \leftarrow$ initial value

else $v \leftarrow$ read val with highest num

Only b changed in this phase

Write is like sending to all

Read instead of waiting for acks

No ack: someone moved on!
Phase I Summary

• Classical Paxos:
  – Leader chooses new ballot, sends to all
  – Others ack if they did not move on to a later ballot
  – If no majority, try again
  – Otherwise, move to Phase 2

• SM Paxos:
  – Leader chooses new ballot, writes its variable
  – Leader reads to check if anyone moved on to a later ballot
  – If any one moved on, try again
  – Otherwise, move to Phase 2
SM Paxos: Phase II
Leader Cont’d

\[ n \leftarrow b \]
write \( \langle b, v, n, \perp \rangle \) to \( x_i \)
read all \( x_j \)’s
if some \( x_i.\text{bal} > b \) then start over
write \( \langle b, v, n, v \rangle \) to \( x_i \)
return \( v \)

Like sending “accept” to all
Read to see if all would have accepted this proposal

v, n changed in this phase

When don’t they?

Decide
Why Read Twice?

- `write(b') > b`
- `read`
- `write`
- `read`
- `write(b)`
- `read`
- `write`
- `read`

- `write(b')` did not complete
- `read` does not see `b'`

- decides
- doesn't decide
Adding The Non-Leader Code

while (true)
  if leader (by $\Omega$) then
    [ leader code from previous slides ]
  else
    read $x_{ld}$, were ld is leader
    if $x_{ld}.decision \neq \bot$ then
      return $x_{ld}.decision$
Liveness

- The shared memory is reliable
- The non-leaders don’t write
  - They don’t even need to be “around”
- The leader only fails if there is contention
  - Another leader competes with it
  - By $\Omega$, eventually only one leader will compete
  - In shared memory systems, $\Omega$ is called a contention manager
Validity

• By induction
• Leader always proposes its own value or one previously proposed by an earlier leader
  – Regular registers suffice
Agreement “By Example”

- write(b) → read
- write(v) → read
- write decision

- no write(b) for b > b completed
- read does not see any b > b

- write(b > b) → read → write
- read sees b, v
- writes v
Agreement Proof Idea

• Look at lowest ballot, b, in which some process decides v
• By uniqueness of b, no \( v' \neq v \) is decided with \( b \)
• Prove by induction that every decision with \( b' > b \) is \( v \)
Termination

• When one correct leader exists
  – It eventually chooses a higher b than all those written before
  – No other process writes a higher ballot
  – So it does not start over, and hence decides

• Any number of processes can fail
Optimization

• The first write (of b) does not write consensus values
• A leader running multiple consensus instances can perform the first write once and for all and then perform only the second write for each consensus instance
Leases

• We need eventually accurate leader (Ω)
  – But what does this mean in shared memory?
• We would like to have mutual exclusion
  – Not fault-tolerant!
• Lease: fault-tolerant, time-based mutual exclusion
  – Live but not safe in eventual synchrony model
Using Leases

• A client that has something to write tries to obtain the lease
  – Lease holder = leader
  – May fail...

• Example implementation:
  – Upon failure, backoff period

• Leases have limited duration, expire
# Lock versus Lease

<table>
<thead>
<tr>
<th>Lock</th>
<th>Lease</th>
</tr>
</thead>
</table>
| ✅ **Blocking**  
  • Using locks is not wait-free  
  • If lock holder fails, we’re in trouble | ✅ **Non-blocking**  
  • Expires regardless whether holder fails |
| ✅ **Always safe**  
  • Never two lock-holders | ✅ **Unsafe**  
  • Two lease-holders possible due to asynchrony  
  • OK for algorithms like Paxos |
Disk Paxos

• Consensus using $n \geq 2t+1$ fault-prone disks
  – Disks can incur crash failures

• Solution combines:
  – $m$-process shared memory Paxos and
  – ABD-like emulation of shared registers from fault-prone ones
Disk Paxos Data Structures

Process i can write block[i][j] in each disk j, can read all blocks
Read Emulation

• In order to read $x_i$
  – Issue read block[i][j] for each disk j
  – Wait for majority of disks to respond
  – Choose block with largest $b,n$

• Is this enough?

• How did ABD’s read emulation work?
One Read Round Enough for Regular

finds a copy that was written

returning 0 is OK for regular

write(0) read(0) read(1) write(1)

time

write(0)

read(0)
Write Emulation

• In order to write $x_i$
  – Issue write block[i][j], for each disk j
  – Wait for majority of disks to respond

• Is this enough?
Summary

• ABD: Emulate reliable shared memory
  – In asynchronous system
  – Using fault-prone storage nodes (minority)

• SM Paxos: Solve consensus
  – In asynchronous reliable shared memory
  – Using leader-election failure detector
  – Tolerate any number of client failures

• Disk Paxos: Combine the two
Additional Challenges & New Results

Reconfiguration

Codes to Mitigate Storage Blow Up
Reconfiguration

• Limited availability: always need C, D, and E
• After removing A, B, need two of {C,D,E}
The Challenge
The Challenge

- Majority of \{A,B,C\}
- Majority of \{A,B,C,D,E\}

Split Brain!

A

Remove D

E
Dynamic Reconfiguration: Abstraction and Optimal Asynchronous Solution

A. Spiegelman, I. Keidar, and D. Malkhi, DISC 2017
Dynamic Reliable Objects

Object’s API

Reliable emulation

Extend API with reconfig operation

ChangeConfig(-$S_1$,$+S_4$)

$S_4$ $S_3$ $S_2$ $S_1$
What Now?

Object’s API

Reliable emulation

ChangeConfig(-S_1,+S_4)

When can I shut S1 down?

S_4  S_3  S_2  S_1

NEW
Contributions

• Clean model for dynamic objects
  – API, failure condition, complexity metrics

• General abstraction for reconfiguration

• Optimal asynchronous register emulation
  – see paper
Storage Blow Up
k-of-n Erasure Codes

D \{ \}

encode \{ D/k \}
n
D/k \{ \}

decode \{ k \}
Why Codes?

To tolerate one failure

• With replication

• With erasure codes
Reliable Storage Example

• $n = 2f + k$

$f = 1$

$n = 4$

$k = 2$
Write
Write

Generate timestamp

encode

Reliable Distributed Storage, Idit Keidar
Write

Can’t wait forever
Write

Wait for \( n-f \) replies
Read

Wait for $n-f$ replies
Read

decode
What About Concurrency?
Write
Write
Write

Overwrite?

Reliable Distributed Storage, Idit Keidar
Write

Overwrite?
Suppose yes, if timestamp is bigger
Write
No written value can be restored!
Write

Read

Forever!

Reliable Distributed Storage, Idit Keidar
What About Replication?
Write

No problem!

Read
Back to Coding ...
Overwrite?
No!

Reliable Distributed Storage, Idit Keidar
Overwrite Green?
Suppose yes
Read

cannot be restored!
What can be overwritten? Nothing!
Distributed Storage: Space Bounds

• Spiegelman et al. PODC 2016: $\Omega(D \cdot \min(f,c))$
  – Lock-free multi-writer
  – $f$ failures,
  – $c$ concurrent writes
  – Value size $D$

• Berger et al. DISC 2018: $\Omega(k \cdot \min(2^D, R))$
  – $k$-out-of-$n$ coding
  – $R$ visible readers; $R$ infinite with invisible readers
  – Value size