Distributed Systems
DAT520 - Spring 2024
Introduction to Gorums

Hein Meling

Outline

- Introduction
- Background: Quorums and Applications
- Gorums' Abstractions
- Several Case Studies and Some Experimental Evaluation
- Conclusions and Feedback

Introduction

- Computers and networks are unreliable and fail

Introduction

- Computers and networks are unreliable and fail
- **Want:** Fault Tolerance and Availability

Introduction

- Computers and networks are unreliable and fail
- **Want:** Fault Tolerance and Availability
- **Solution:** Replicate state across a set of nodes

Introduction

- Computers and networks are unreliable and fail
- **Want:** Fault Tolerance and Availability
- **Solution:** Replicate state across a set of nodes
- **Challenge:** *(strong) consistency*

Introduction

- Computers and networks are unreliable and fail
- **Want:** Fault Tolerance and Availability
- **Solution:** Replicate state across a set of nodes
- **Challenge:** *(strong) consistency*
- Requires complex protocols

Gorums framework

Simplify design and implementation of fault-tolerant quorum-based protocols

Quorum

"A quorum is the minimum number of members of a deliberative assembly necessary to conduct the business of that group." (Wikipedia)

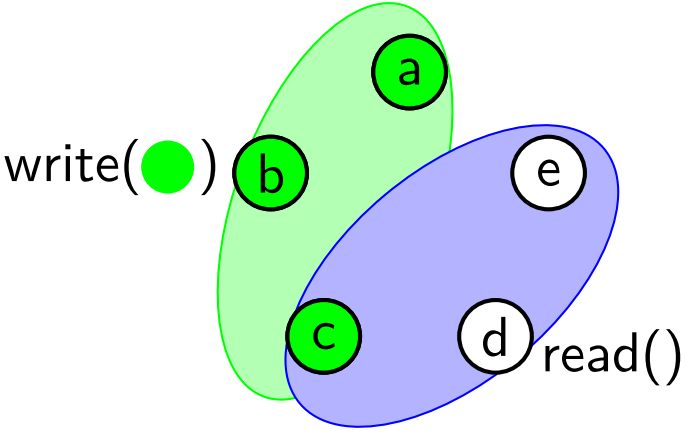
Background: Quorum Systems

- **Quorum System:** Collection of subsets of replicas, where every pair of subsets intersect.

Background: Quorum Systems

- **Quorum System:** Collection of subsets of replicas, where every pair of subsets intersect.
- Example: Set of majorities
  - Set of replicas:
    $\{a, b, c\}$
  - Subsets that intersect:
    $\{\{a, b\} \cap \{a, c\} = a\}$,
    $\{\{a, b\} \cap \{b, c\} = b\}$,
    $\{\{a, c\} \cap \{b, c\} = c\}$

Majority Quorum Example



Other Types of Quorum Systems

- Read/write quorums
- Weighted quorums
- Grid quorums
- Byzantine quorums

Applications: Where do we need quorums?

- Consensus and replicated state machines (RSMs)
- Distributed atomic storage
- Group communication systems
- Replicated databases
- Blockchains

How can we build a quorum system?

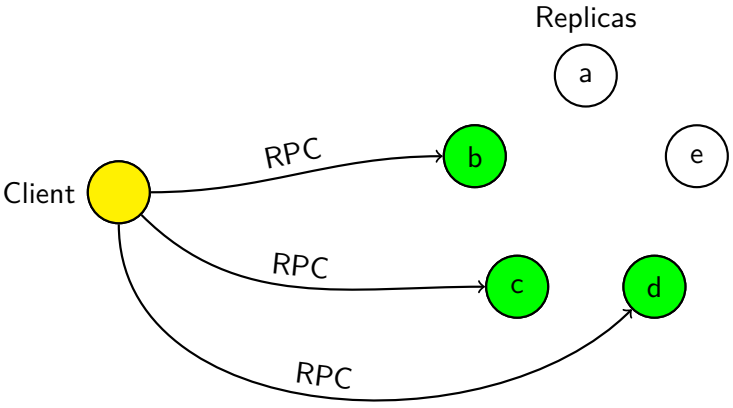
Invoke a Quorum of Replicas

- Access state stored at each replica

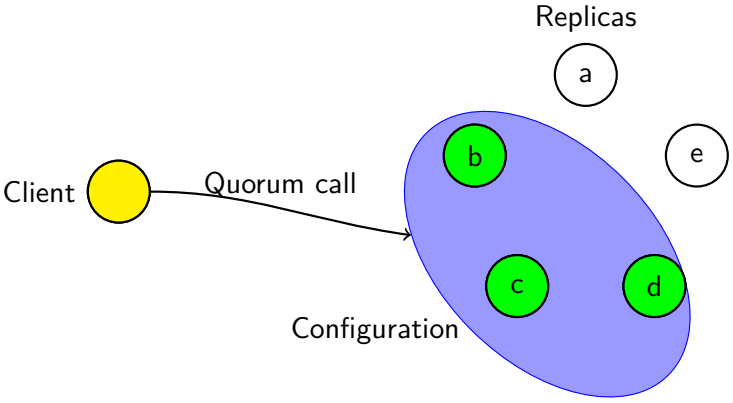
Invoke a Quorum of Replicas

- Access state stored at each replica
- To contact a quorum:
  - Must collect and associate replies from individual replicas
  - Not difficult in general, but adds complexity

Invoke a Quorum using RPCs



Invoke a Quorum using a Quorum Call



A Single-server Read/Write Storage

```
1  service Storage {
2    rpc Read(ReadRequest) returns (State) {
3
4    }
5    rpc Write(State) returns (WriteReply) {
6
7    }
8  }
9
10 message State {
11   string Value  = 1;
12   int64  Timestamp = 2;
13 }
14
15 message WriteReply {
16   bool New = 1;
17 }
18
19 message ReadRequest {}
```

A Quorum-based Read/Write Storage

```
1  service Storage {
2    rpc Read(ReadRequest) returns (State) {
3      option (gorums.quorumcall) = true;
4    }
5    rpc Write(State) returns (WriteReply) {
6      option (gorums.quorumcall) = true;
7    }
8  }
9
10 message State {
11   string Value  = 1;
12   int64  Timestamp = 2;
13 }
14
15 message WriteReply {
16   bool New = 1;
17 }
18
19 message ReadRequest {}
```

Storage Interfaces

```
type StorageClient interface {
    Read(context.Context, *ReadRequest, ...grpc.CallOption)
        (*State, error)
    Write(context.Context, *State, ...grpc.CallOption)
        (*WriteReply, error)
}

type StorageServer interface {
    Read(context.Context, *ReadRequest) (*State, error)
    Write(context.Context, *State) (*WriteReply, error)
}
```

Gorums Abstractions

- Configurations
- Quorum Call
- Quorum Functions

Abstraction #1: Configurations

- Replicas grouped into **configurations**
- Configuration implements the `StorageClient` interface

Abstraction #1: Configurations

- Replicas grouped into **configurations**
- Configuration implements the `StorageClient` interface
- **Quorum specification object**:
  - Specifies a quorum system for the configuration
  - Simple examples only need quorum size parameter

Configuration and Quorum Specification

```
type Configuration struct {
    id     uint32
    nodes  []*Node
    n      int
    mgr    *Manager
    qspec  QuorumSpec
}
```

```
type MajorityQSpec struct {
    quorumSize int
}
```

Configuration and Quorum Specification

```
type Configuration struct {
    id     uint32
    nodes  []*Node
    n      int
    mgr    *Manager
    qspec  QuorumSpec
}
```

```
type MajorityQSpec struct {
    quorumSize int
}
```

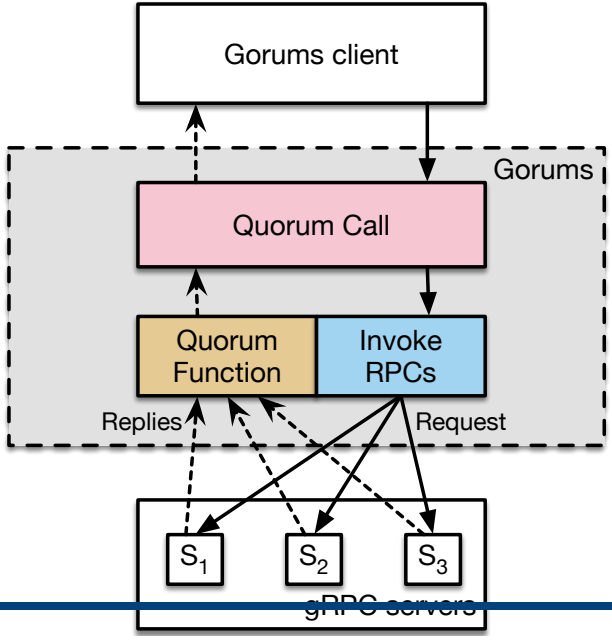
```
unc (c *Configuration) Read(ctx Context, a *ReadRequest) (*State, error) {
    ...
    replyChan := make(chan internalValue, c.n)
    for _, n := range c.nodes {
        go callGRPCRead(ctx, a, n, replyChan)
    }
    ...
```

Abstraction #2: Quorum Call

- Invoke quorum call on a configuration
- Wait for responses from a quorum

$$\textsc{WriteReply},\ \textsc{Error} := \mathit{config}.\mathrm{Write}(\mathit{state}\ \textsc{State})$$

Quorum Call Illustrated



Quorum Logic

- A quorum call needs to determine
  - If a quorum of responses have been received
  - What kind of response to return
- **Quorum logic:** rules for verifying a quorum from individual replies

Motivation: Separation of Concerns

- Quorum logic is often intertwined with protocol logic
- **Our goal:** separate quorum logic from the main control flow of a protocol's operation

Abstraction #3: Quorum Functions

- Gorums uses **quorum functions** to specify quorum logic
- Each service method has a developer-defined quorum function

$$\textsc{WriteReply},\ \textsc{Bool} \ := \ qs.\mathrm{WriteQF}(replies\ [\ ]_{\textsc{WriteReply}})$$

- Gorums runtime calls this quorum function for each reply received

Quorum Specification Interface

```
type QuorumSpec interface {
    ReadQF(replies []*State) (*State, bool)
    WriteQF(replies []*WriteReply) (*WriteReply, bool)
}
```

Quorum Function #1 (simple majority)

Algorithm 1 Simple quorum function

```
1: func (qs QuorumSpec) ReadQF(replies []State)
2:     if len(replies) ≥ qs.QSize then          ▷ check quorum size
3:         return replies[0], true              ▷ quorum, return reply
4:     return nil, false                        ▷ no quorum yet
```

Quorum Function #2 (basic reply checking)

Algorithm 2 Paxos first phase quorum function

```
1: func (qs QuorumSpec) PaxosPrepareQF(replies []Promise)
2:     if len(replies) < qs.majQSize then                    ▷ majority quorum size
3:         return nil, false                                 ▷ no quorum yet, await more replies
4:     reply := new(Promise)                                 ▷ initialize reply with nil/0 fields
5:     for r := range replies do
6:         if r.ballot > reply.ballot then
7:             reply.ballot := r.ballot
8:         if r.vballot ≥ reply.vballot then
9:             reply.vballot := r.vballot
10:            reply.value := r.value
11:    return reply, true                                    ▷ quorum found
```

Quorum Function #3 (complex)

Algorithm 3 EPaxos PreAccept quorum function

```
1:  func (qs QuorumSpec) PreAcceptQF(replies []PreAccReply)
2:      if replies[len(replies)−1].Type = Abort then
3:          return replies[len(replies) − 1], true                ▷ single Abort
4:      if len(replies) < qs.SlowQSize then
5:          return nil, false                                     ▷ no quorum yet
6:      reply := new(PreAccReply)                                 ▷ initialize reply with nil/0 fields
7:      for r := range replies do
8:          if r.Type = Conflict then
9:              reply.Type := Conflict
10:             reply.Conflicts := reply.Conflicts ∪ r.Conflicts
11:     if reply.Type = Ok ∧ len(replies) < qs.FastQSize then
12:         reply.Type := Conflict
13:         return reply, false
14:     return reply, true
```

Quorum Function Template

Template for quorum functions

```
0: func (qs QuorumSpec) MethodQF(replies []MethodReply)
1:     if qs.Abort(replies[len(replies) − 1]) then                ▷ check last reply
           return replies[len(replies) − 1], true                 ▷ abort call
2:     if len(replies) < qs.QSize then                            ▷ check quorum size
           return nil, false                                      ▷ no quorum yet, await more replies
3:     if ¬qs.IsQuorum(replies) then                              ▷ check content for quorum
           return nil, false                                      ▷ no quorum yet, await more replies
4:     reply := qs.Combine(replies)                               ▷ combine replies into single reply
5:     if qs.WaitForMore(replies) then
           return reply, false                                    ▷ return possible but prefer waiting
6:     return reply, true                                         ▷ terminate call and return
```

Quorum Call Semantics

#	**QFunc return** Reply, Bool	**Quorum call action** return Reply, Error
1	*retval*, *true*	return *retval*, *nil* and terminate call
2	*retval*, *false*	if possible: wait for further replies else: return *retval*, *IncompleteError*

Invoked by Gorums:

$$\textsc{WriteReply},\ \textsc{Bool} := \mathit{qs}.\text{WriteQF}(\mathit{replies}\ [\ ]\textsc{WriteReply})$$

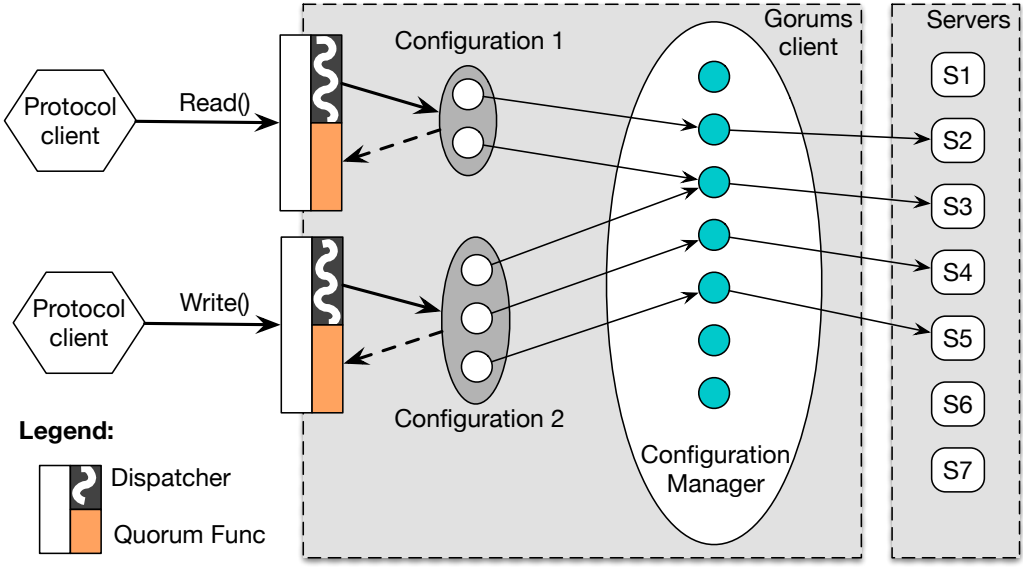
User code:

$$\textsc{WriteReply},\ \textsc{Error} := \mathit{config}.\text{Write}(\mathit{state}\ \textsc{State})$$

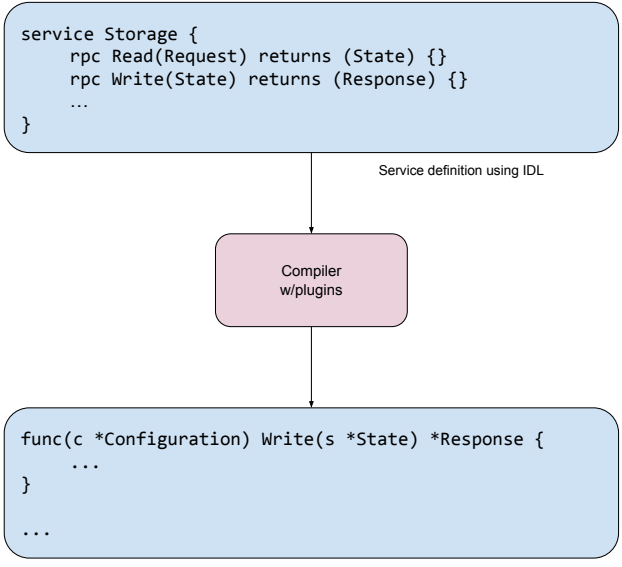
Implementation

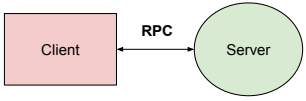
- Gorums is implemented as a library in Go
- Code generation from service definition:
  - Creates a library for clients (and servers)
  - Enabling invocation of quorum calls on configurations
- Builds on established toolchain:
  - Protocol Buffers and gRPC

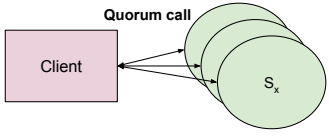
Implementation Overview



Code Generation







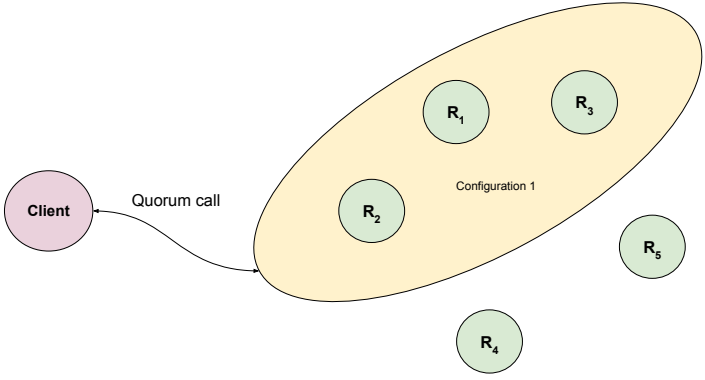
Case Studies

- **Reconfigurable Atomic Storage**
  - SmartMerge [DISC'15], DynaStore and Rambo
  - Evaluating different reconfiguration algorithms [OPODIS'16]
- **Simple Majority Quorums**
  - Consensus: Single-decree Paxos
  - State machine replication: Raft
- **Latency-efficient Quorums**
  - State machine replication: EPaxos [SOSP'13]
  - Evaluate complex quorum logic [IDCDS'17]
- **Byzantine Quorums: Byzantine Storage**
  - Requires verifying digital signatures in the quorum function
  - Evaluate different quorum functions
- **Erasure Coded Distributed Storage**
  - Requires encoding/decoding of data and parity shards
- **Asynchronous Quorum Call**
  - Futures, Correctables, and Streaming replies

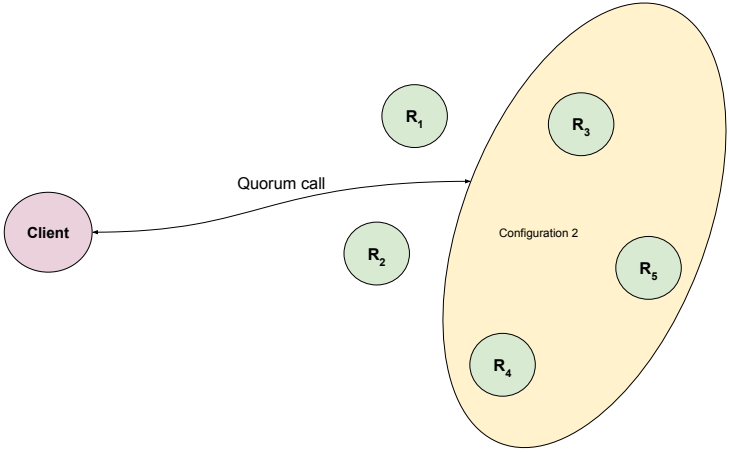
Initial Motivation: Reconfiguration

- Reconfiguration: dynamically changing the replica set
- Difficult to implement correctly
- Previous work: implemented a Paxos-based RSM with support for several reconfigurations protocols

Initial Motivation: Reconfiguration

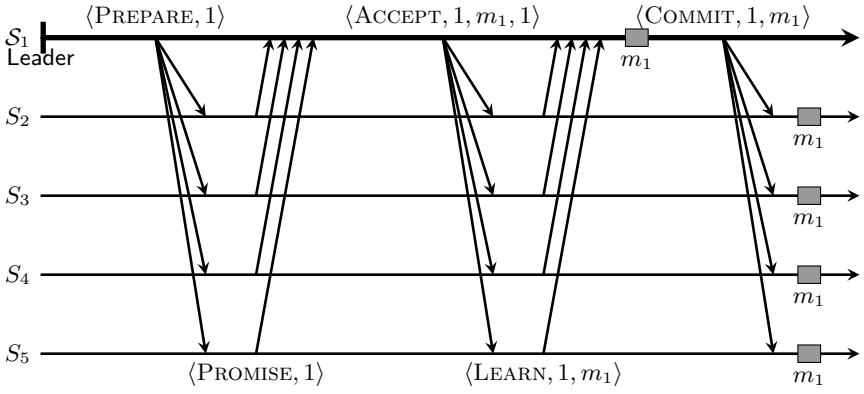


Initial Motivation: Reconfiguration



Simple Majority Quorums

Single-decree Paxos: Non-faulty Execution



Single-decree Paxos: Proto File

```
service SinglePaxos {
  rpc Prepare(PrepareMsg) returns (PromiseMsg) {
    option (gorums.quorumcall) = true;
  }
  rpc Accept(AcceptMsg) returns (LearnMsg) {
    option (gorums.quorumcall) = true;
  }
  rpc Commit(LearnMsg) returns (Empty) {
    option (gorums.quorumcall) = true;
  }
}
```

Single-decree Paxos: Proto File 2

```
1  message PrepareMsg {
2    uint32  rnd  = 1;
3  }
4
5  message PromiseMsg {
6    uint32  rnd  = 1;
7    uint32  vrnd = 2;
8    Value vval  = 3;
9  }
10
11 message AcceptMsg {
12   uint32  rnd  = 1;
13   Value val   = 2;
14 }
15
16 message LearnMsg {
17   uint32  rnd  = 1;
18   Value val   = 2;
19 }
```

Single-decree Paxos: Protocol Phases

```
func (p *Proposer) runPaxosPhases() error {
    // PHASE ONE: send Prepare to obtain quorum of Promises
    preMsg := &PrepareMsg{Rnd: p.crnd}
    prmMsg, err := p.config.Prepare(preMsg)
```

Single-decree Paxos: Protocol Phases

```
func (p *Proposer) runPaxosPhases() error {
    // PHASE ONE: send Prepare to obtain quorum of Promises
    preMsg := &PrepareMsg{Rnd: p.crnd}
    prmMsg, err := p.config.Prepare(preMsg)

    // PHASE TWO: send Accept to obtain quorum of Learns
    if prmMsg.GetVrnd() != Ignore {
        // promise msg has a locked-in value; update proposer state
        p.cval = prmMsg.GetVval()
    }
    // use local proposer's cval or locked-in value from promise msg, if any
    accMsg := &AcceptMsg{Rnd: p.crnd, Val: p.cval}
    lrnMsg, err := p.config.Accept(accMsg)
```

Single-decree Paxos: Protocol Phases

```
func (p *Proposer) runPaxosPhases() error {
    // PHASE ONE: send Prepare to obtain quorum of Promises
    preMsg := &PrepareMsg{Rnd: p.crnd}
    prmMsg, err := p.config.Prepare(preMsg)

    // PHASE TWO: send Accept to obtain quorum of Learns
    if prmMsg.GetVrnd() != Ignore {
        // promise msg has a locked-in value; update proposer state
        p.cval = prmMsg.GetVval()
    }
    // use local proposer's cval or locked-in value from promise msg, if any
    accMsg := &AcceptMsg{Rnd: p.crnd, Val: p.cval}
    lrnMsg, err := p.config.Accept(accMsg)

    // PHASE THREE: send Commit to obtain a quorum of Acks
    ackMsg, err := p.config.Commit(lrnMsg)
}
```

Latency-efficient Quorums

Latency-efficient Quorums

- EPaxos: State machine replication protocol
- Complex quorum logic
  - Majority and fast quorums

Latency-efficient Quorums: Quorum Function

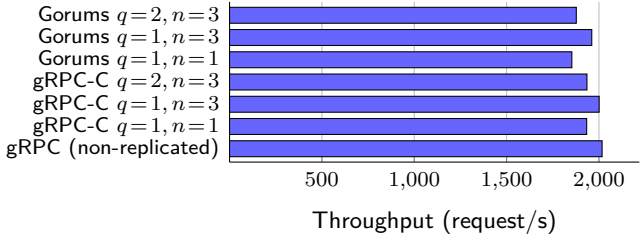
Algorithm 5 EPaxos PreAccept quorum function

```
1:  func (qs QuorumSpec) PreAcceptQF(replies []PreAccReply)
2:      if replies[len(replies)−1].Type = Abort then
3:          return replies[len(replies) − 1], true                ▷ single Abort
4:      if len(replies) < qs.SlowQSize then
5:          return nil, false                                     ▷ no quorum yet
6:      reply := new(PreAccReply)                                 ▷ initialize reply with nil/0 fields
7:      for r := range replies do
8:          if r.Type = Conflict then
9:              reply.Type := Conflict
10:             reply.Conflicts := reply.Conflicts ∪ r.Conflicts
11:     if reply.Type = Ok ∧ len(replies) < qs.FastQSize then
12:         reply.Type := Conflict
13:         return reply, false
14:     return reply, true
```

Experimental Evaluation

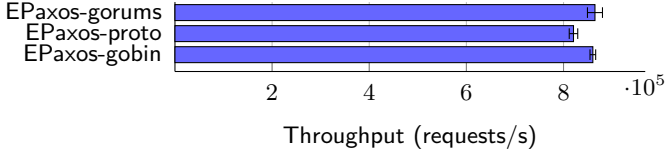
- The cost of abstraction
- Two sets of benchmarks:
  - Micro-benchmarks
  - EPaxos system benchmarks
- Original EPaxos modified to use Gorums

Gorums Micro-benchmarks

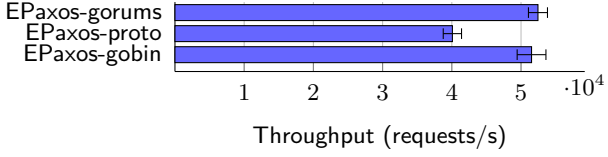


EPaxos Benchmarks

16 B request size



1 kB request size



Byzantine Storage

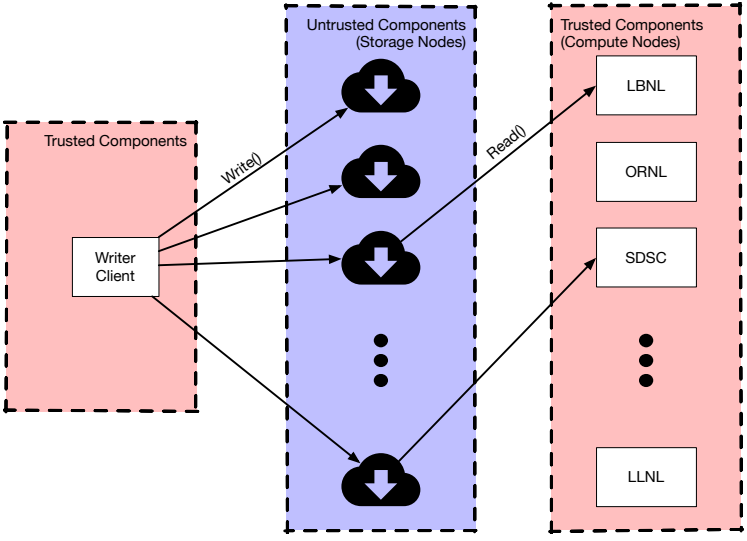
Byzantine Storage: Overview

- Authenticated-Data Byzantine Quorum
  - Textbook algorithm [RSDP]
  - Single Writer: digitally signs and updates storage servers
  - Multiple Readers: read latest version from storage servers and verify the writer's signature

Byzantine Storage: Overview

- Authenticated-Data Byzantine Quorum
  - Textbook algorithm [RSDP]
  - Single Writer: digitally signs and updates storage servers
  - Multiple Readers: read latest version from storage servers and verify the writer's signature
- Assumptions:
  - Servers may be Byzantine faulty
  - Readers and the writer are non-Byzantine
  - Algorithm need $n = 3f + 1$ servers to tolerate f faulty servers
  - Thus, $(n + f)/2$ valid replies form a quorum

Byzantine Storage: Architecture



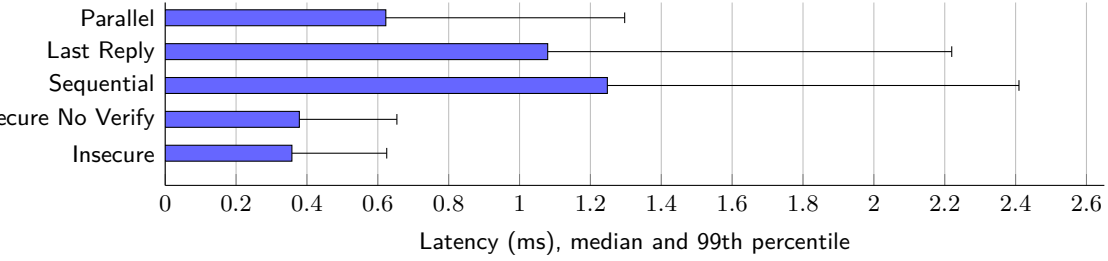
Byzantine Storage: Quorum Specification

```
type AuthDataQ struct {
    n    int                // size of system
    f    int                // tolerable number of failures
    q    int                // quorum size q=(n+f)/2
    pub  *ecdsa.PublicKey   // public key of the writer
}

func (aq *AuthDataQ) ReadQF(replies []*Value) (*Value, bool) {
    if len(replies) <= aq.q {
        return nil, false // not enough replies
    }
    for _, reply := range replies {
        if aq.verify(reply) {
            if reply.Timestamp <= highest.Timestamp {
                continue
            }
            highest = reply
        }
    }
    return highest, true
}
```

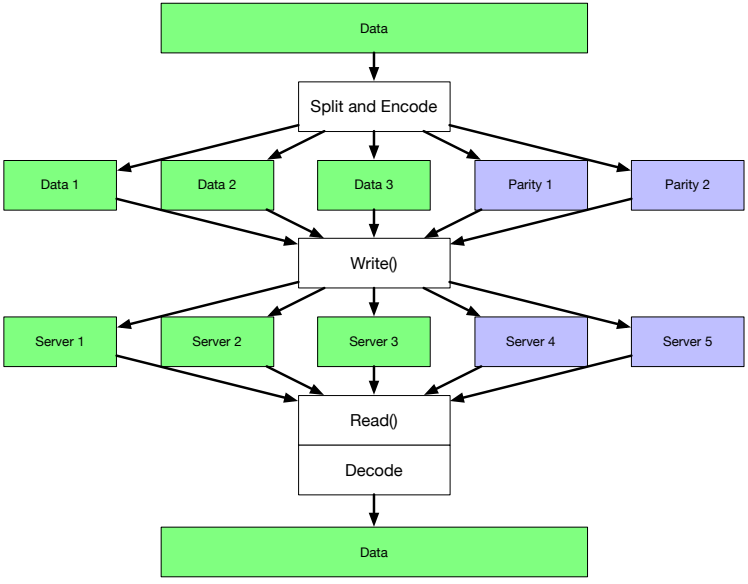
Performance Evaluation of Quorum Functions

- Easy to test and compare different quorum functions for same protocol



Erasure Coded Distributed Storage

Erasure Coded Storage



Erasure Coded Storage: Proto File

```
service Storage {
  rpc Read(Key) returns (Value) {
    option (gorums.quorumcall) = true;
  }
  rpc Write(Value) returns (WriteReply) {
    option (gorums.quorumcall)           = true;
    option (gorums.per_node_arg) = true;
  }
}

message Value {
  string key       = 1; // key used for lookup
   int64 timestamp = 2; // timestamp for storage
   int32 index     = 3; // index of this shard (on server)
   int64 datasize  = 4; // size of the value when reconstructed
  bytes value      = 5; // value for the given key
}
```

Erasure Coded Storage: Writer Client

- Split and encode value to store into data and parity shards
- Uses a per-node translation function

```
shards := qspec.Encode(client.Value)
assignShard := func(v Value, node *Node) *Value {
    index := node.Index()
    v.Index = index
    v.Value = shards[index]
    return &v
}
val := &Value{
    Key:       client.Key,
    Timestamp: client.Timestamp,
}
ack, err := config.Write(val, assignShard)
```

Erasure Coded Storage: Reader Client

- Simple RPC read call

```
val, err := config.Read(client.Key)
```

- Reconstruction takes place in the quorum function

Erasure Coded Storage: Quorum Spec

```
type ECSpec struct {
    reedsolomon.Encoder
    DataShards   int // Number of data shards
    ParityShards int // Number of parity shards
    Shards       int // Total number of shards.
}

func NewEC(n, k int) (*ECSpec, error) {
    enc, err := reedsolomon.New(n, k)
    if err != nil {
        return nil, err
    }
    return &ECSpec{enc, n, k, n + k}, nil
}
```

Erasure Coded Storage: Quorum Function

```
// Returns a single reply given a quorum of erasure coded replies.
func (ec *ECSpec) ReadQF(replies []*Value) (*Value, bool) {
    if len(replies) < ec.DataShards {
        // not enough replies to reconstruct; await more replies
        return nil, false
    }
```

Erasure Coded Storage: Quorum Function

```
// Returns a single reply given a quorum of erasure coded replies.
func (ec *ECSpec) ReadQF(replies []*Value) (*Value, bool) {
    if len(replies) < ec.DataShards {
        // not enough replies to reconstruct; await more replies
        return nil, false
    }

    // rearrange replies to their position in the shards slice
    shards := make([][]byte, ec.Shards)
    for _, r := range replies {
        shards[r.Index] = r.Value
    }
```

Erasure Coded Storage: Quorum Function

```
// Returns a single reply given a quorum of erasure coded replies.
func (ec *ECSpec) ReadQF(replies []*Value) (*Value, bool) {
    if len(replies) < ec.DataShards {
        // not enough replies to reconstruct; await more replies
        return nil, false
    }

    // rearrange replies to their position in the shards slice
    shards := make([][]byte, ec.Shards)
    for _, r := range replies {
        shards[r.Index] = r.Value
    }

    val, err := ec.decode(shards)
    if err != nil {
        return nil, true // failed to decode: abort
    }
    // return a single value with bytes from the data shards
    return val, true
}
```

Asynchronous Quorum Calls

Asynchronous Quorum Calls

- One-way calls – multicast
  - no replies collected
- Asynchronous quorum calls
  - replies collected by a *future* object
  - API: `Done()` and `Get()`: blocking
- Correctable quorum calls
  - replies collected by a *correctable* object
  - each server, one reply
  - API: `Get()`: non-blocking with levels of completeness
- Correctable streams quorum calls
  - each server, a stream of replies

Correctables: Proto File

```
service Storage {
  rpc ReadCorrectable(ReadRequest) returns (State) {
    option (gorums.correctable) = true;
  }

  rpc ReadCorrStream(ReadRequest) returns (stream State) {
    option (gorums.correctable_stream) = true;
  }
}
```

Ongoing and Future Work

- Model-based Testing techniques to improve correctness
- Meta configurations
- Pre-call adaptation
  - Sign outgoing messages
  - Split and encode outgoing messages
- All-to-all communication between servers
  - Useful in many Byzantine fault tolerant protocols
- More protocol examples

Conclusions

- Gorums' Abstractions
  - force separation of protocol logic and quorum logic
  - seems to work well for a diverse set of protocols
- Easy to test quorum functions without running full protocol
- Throughput and latency overhead is mostly negligible

Thank you!
Questions?

`http://www.github.com/relab/gorums`