Gorums: New Abstractions for Implementing Quorum-based Systems

Tormod Erevik Lea, Leander Jehl and **Hein Meling**

University of
Stavanger

hein.meling@uis.no

August 5th, 2019

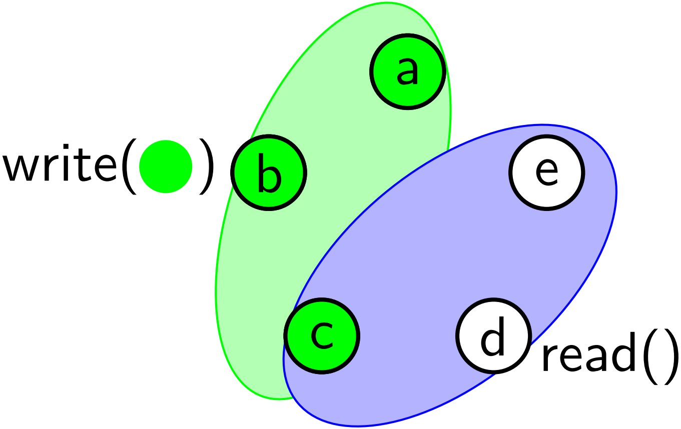
Outline

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

Gorums framework

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

Majority Quorum Example



Other Types of Quorum Systems

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

How can we build a quorum system?

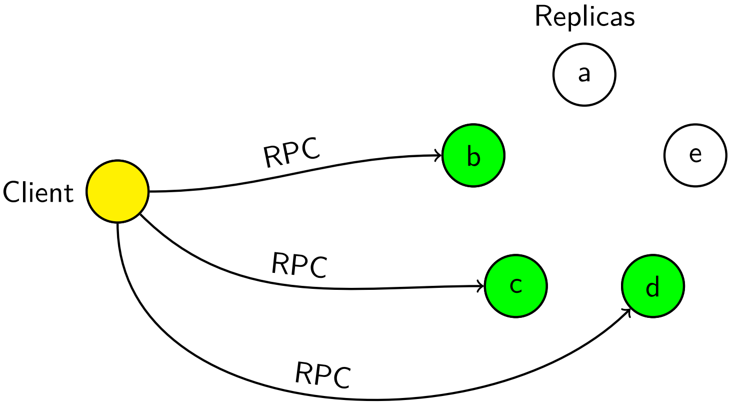
Invoke a Quorum of Replicas

- Access state stored at each replica

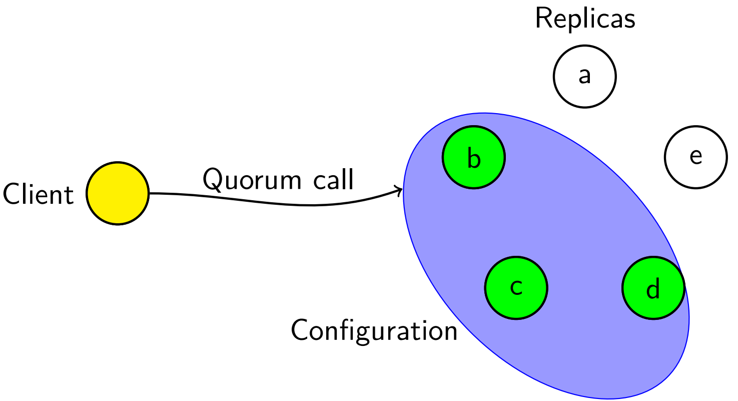
Invoke a Quorum of Replicas

- 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

```
service Storage {
  rpc Read(ReadRequest) returns (State) {

  }
  rpc Write(State) returns (WriteReply) {

  }
}

message State {
  string Value  = 1;
  int64 Timestamp = 2;
}

message WriteReply {
  bool New = 1;
}

message ReadRequest {}
```

A Quorum-based Read/Write Storage

```
service Storage {
  rpc Read(ReadRequest) returns (State) {
    option (gorums.qc) = true;
  }
  rpc Write(State) returns (WriteReply) {
    option (gorums.qc) = true;
  }
}

message State {
  string Value  = 1;
  int64 Timestamp = 2;
}

message WriteReply {
  bool New = 1;
}

message ReadRequest {}
```

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

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

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

Configuration and Quorum Specification

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

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

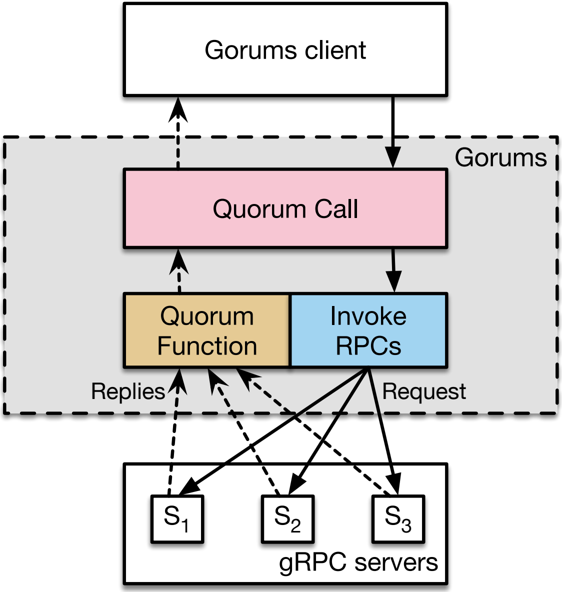
```go
func (c *Configuration) Read(ctx Context, a *ReadRequest) (*State, error)
    ...
    replyChan := make(chan internalValue, c.n)
    for _, n := range c.nodes {
        go callGRPCRead(ctx, a, n, replyChan)
    }
    ...
}
```

Abstraction #2: Quorum Call

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

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

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

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

- Gorums runtime calls this quorum function for each reply received

Quorum Specification Interface

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

Quorum Function #1 (simple majority)

Algorithm 1 Simple quorum function

```
func (qs QuorumSpec) ReadQF(replies []State)
    if $len(replies) \geq qs.\mathrm{QSize}$ then        ▷ check quorum size
        return replies[0], true                          ▷ quorum, return reply
    return nil, false                                    ▷ no quorum yet
```

Quorum Function #2 (basic reply checking)

Algorithm 2 Paxos first phase quorum function

```
func ($qs$ QuorumSpec) PaxosPrepareQF($replies$ []Promise)
    if len($replies$) $<$ $qs$.majQSize then                ▷ majority quorum size
        return $nil$, $false$                               ▷ no quorum yet, await more replies
    $reply$ := new(Promise)                                 ▷ initialize reply with nil/0 fields
    for $r$ := range $replies$ do
        if $r.ballot > reply.ballot$ then
            $reply.ballot := r.ballot$
        if $r.vballot \geq reply.vballot$ then
            $reply.vballot := r.vballot$
            $reply.value := r.value$
    return $reply$, $true$                                  ▷ quorum found
```

Quorum Function #3 (complex)

Algorithm 3 EPaxos PreAccept quorum function

```
func (qs QuorumSpec) PreAcceptQF(replies []PreAccReply)
    if replies[len(replies)$-$1].Type $=$ Abort then
        return replies[len(replies) $-$ 1], true                       ▷ single Abort
    if len(replies) $<$ qs.SlowQSize then
        return nil, false                                              ▷ no quorum yet
    reply := new(PreAccReply)                        ▷ initialize reply with nil/0 fields
    for $r$ := range replies do
        if $r$.Type $=$ Conflict then
            reply.Type := Conflict
            reply.Conflicts := reply.Conflicts $\cup$ $r$.Conflicts
    if reply.Type $=$ Ok $\wedge$ len(replies) $<$ qs.FastQSize then
        reply.Type := Conflict
        return reply, false
    return reply, true
```

Quorum Function Template

Template for quorum functions

```
func $(qs\ \textsc{QuorumSpec})\ \mathrm{MethodQF}(replies\ [\,]\textsc{MethodReply})$
    if $qs.\mathrm{Abort}(replies[\mathrm{len}(replies) - 1])$ then          ▷ check last reply
        return $replies[\mathrm{len}(replies) - 1], true$                  ▷ abort call
    if $\mathrm{len}(replies) < qs.\mathrm{QSize}$ then                     ▷ check quorum size
        return $nil, false$                                                  ▷ no quorum yet, await more replies
    if $\neg qs.\mathrm{IsQuorum}(replies)$ then                             ▷ check content for quorum
        return $nil, false$                                                  ▷ no quorum yet, await more replies
    $reply := qs.\mathrm{Combine}(replies)$                                  ▷ combine replies into single reply
    if $qs.\mathrm{WaitForMore}(replies)$ then
        return $reply, false$                                                ▷ return possible but prefer waiting
    return $reply, true$                                                     ▷ terminate call and return
```

Quorum Call Semantics

#	**QFunc return**	**Quorum call action**
	Reply, Bool	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:

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

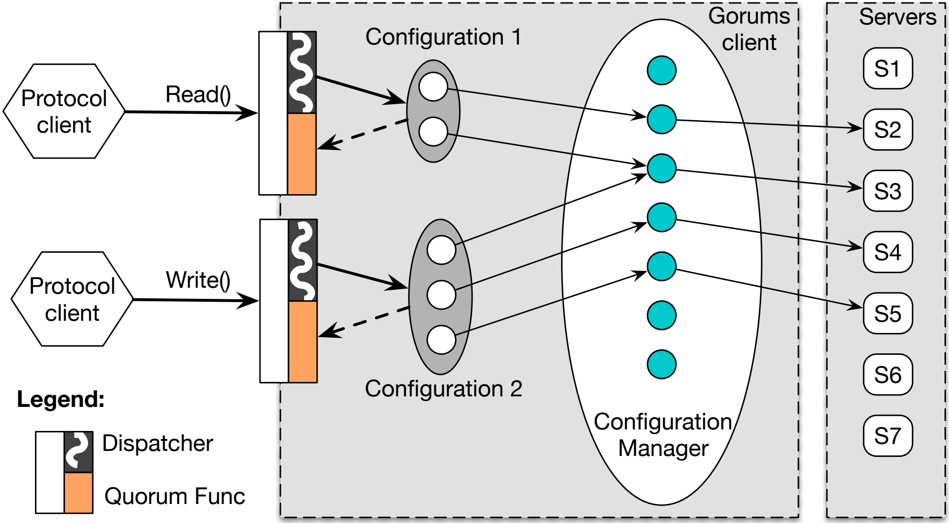
User code:

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

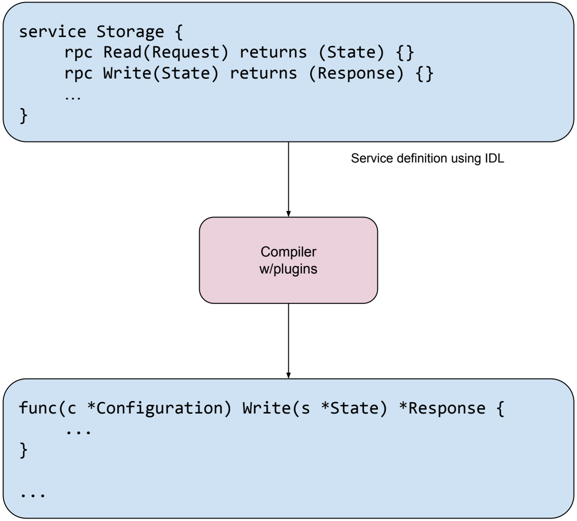
Implementation

- 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



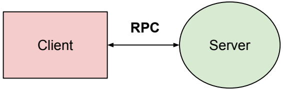
```
service Storage {
     rpc Read(Request) returns (State) {}
     rpc Write(State) returns (Response) {}
     …
}
```

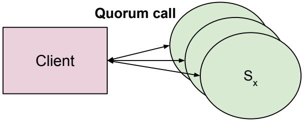
Service definition using IDL

```
func(c *Configuration) Write(s *State) *Response {
     ...
}

...
```

Generated source/library





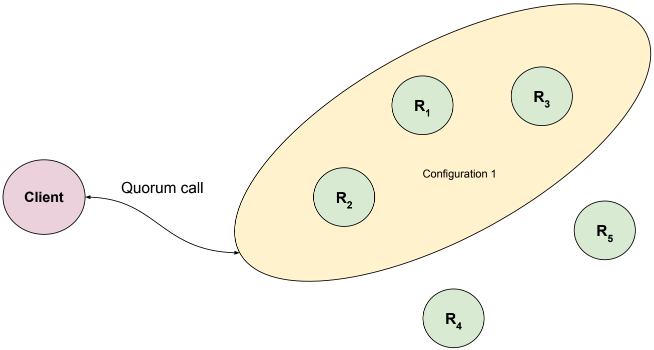
Case Studies

- **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

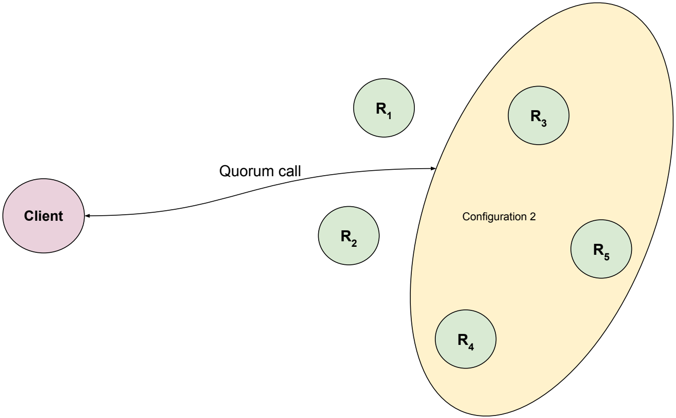
Initial Motivation: Reconfiguration

- 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

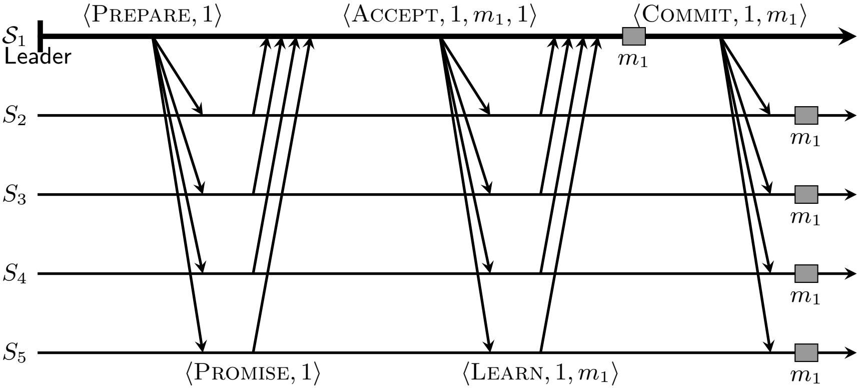


Initial Motivation: Reconfiguration



Simple Majority Quorums

Single-decree Paxos: Non-faulty Execution


$\langle \text{Prepare}, 1 \rangle$
$\langle \text{Accept}, 1, m_1, 1 \rangle$
$\langle \text{Commit}, 1, m_1 \rangle$
S_1
Leader
m_1
S_2
m_1
S_3
m_1
S_4
m_1
S_5
$\langle \text{Promise}, 1 \rangle$
$\langle \text{Learn}, 1, m_1 \rangle$
m_1

Single-decree Paxos: Proto File

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

Single-decree Paxos: Proto File 2

```
message PrepareMsg {
  uint32 rnd  = 1;
}

message PromiseMsg {
  uint32 rnd  = 1;
  uint32 vrnd = 2;
  Value vval  = 3;
}

message AcceptMsg {
  uint32 rnd  = 1;
  Value val   = 2;
}

message LearnMsg {
  uint32 rnd  = 1;
  Value val   = 2;
}
```

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

```go
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
    accMsg := &AcceptMsg{Rnd: p.crnd, Val: p.cval}
    lrnMsg, err := p.config.Accept(accMsg)
```

Single-decree Paxos: Protocol Phases

```go
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
    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

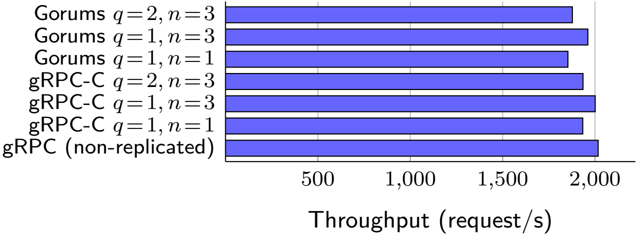
Algorithm 5 EPaxos PreAccept quorum function

```
func (qs QuorumSpec) PreAcceptQF(replies []PreAccReply)
    if replies[len(replies)−1].Type = Abort then
        return replies[len(replies) − 1], true                      ▷ single Abort
    if len(replies) $<$ qs.SlowQSize then
        return nil, false                                            ▷ no quorum yet
    reply := new(PreAccReply)                ▷ initialize reply with nil/0 fields
    for $r$ := range replies do
        if $r$.Type = Conflict then
            reply.Type := Conflict
            reply.Conflicts := reply.Conflicts $\cup$ $r$.Conflicts
    if reply.Type = Ok $\wedge$ len(replies) $<$ qs.FastQSize then
        reply.Type := Conflict
        return reply, false
    return reply, true
```

Experimental Evaluation

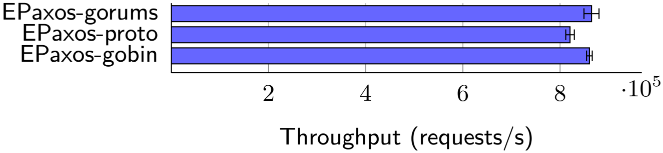
- The cost of abstraction
- Two sets of benchmarks:
  - Micro-benchmarks
  - EPaxos system benchmarks
- Original EPaxos modified to use Gorums

Gorums Micro-benchmarks

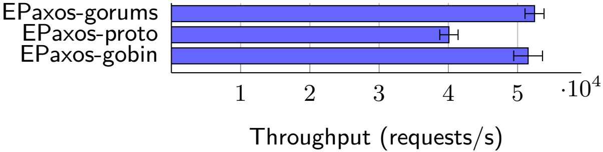


EPaxos Benchmarks

16 B request size



1 kB request size



Byzantine Storage

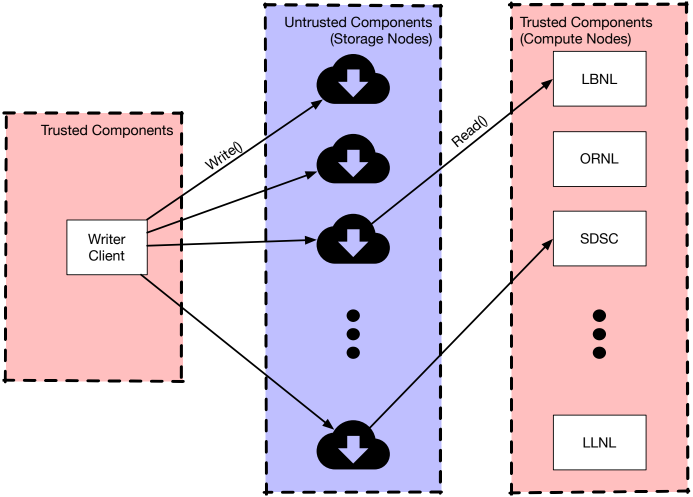
Byzantine Storage: Overview

- 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


Trusted Components
Writer Client
Write()
Untrusted Components (Storage Nodes)
Read()
Trusted Components (Compute Nodes)
LBNL
ORNL
SDSC
LLNL

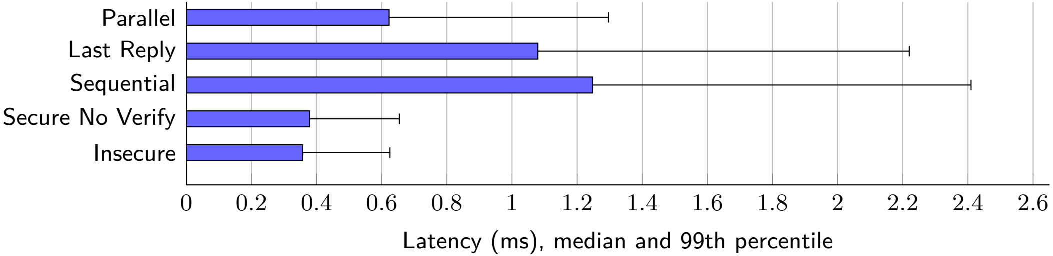
Byzantine Storage: Quorum Specification

```go
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



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`