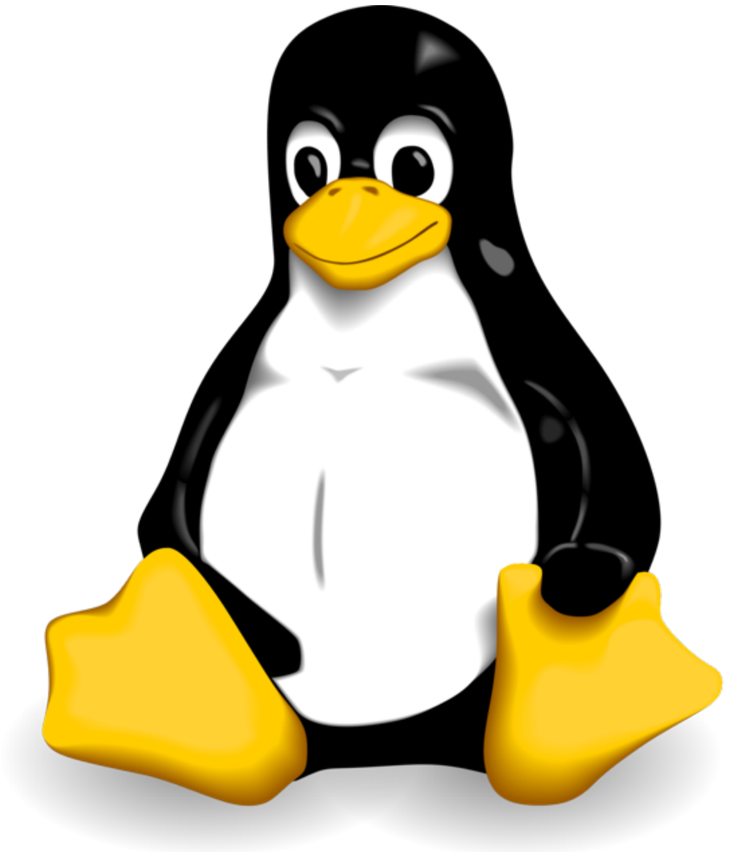
High performance git infrastructure with Go

David Calavera
Code Climate


git

2005

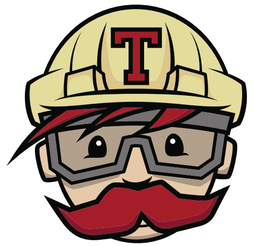


(ノ °□°)ノ ⌒ ┻━━┻

2008

2010...

CODE CLIMATE



Travis CI

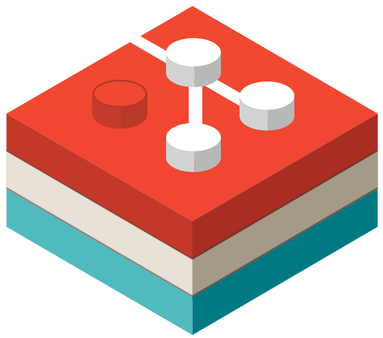
Sourcegraph

How do we make git faster?

* Ruby for brevity 🙈

```
system(“git log …”)*
```

```
git fetch origin
```



libgit2

Disclaimer: examples do not handle errors 😱

github.com/libgit2/git2go

```
// Load with side effects.
// Initialize libgit2's TLS:
//
// func init() {
//     C.git_libgit2_init()
// }
//
// Import package "git", which is
// not very goimports friendly.
import "github.com/libgit2/git2go"
```

```
// Create a new repository.
// Do not use a working directory.
path := “/var/git/repository”
bare := true
r, _ := git.InitRepository(path, bare)

// Clone a repository.
url  := “git://github.com/golang/go”
opts := git.CloneOptions{Bare: bare}
r, _ := git.Clone(url, path, &opts)
```

```
// Create a new remote ref.
name  := “my-fork”
url   := “git://github.com/wadus/go”
rm, _ := r.CreateRemote(name, url)

// Fetch all refs from a remote.
var refspecs []string
rm.Fetch(refspecs, nil, nil)
```

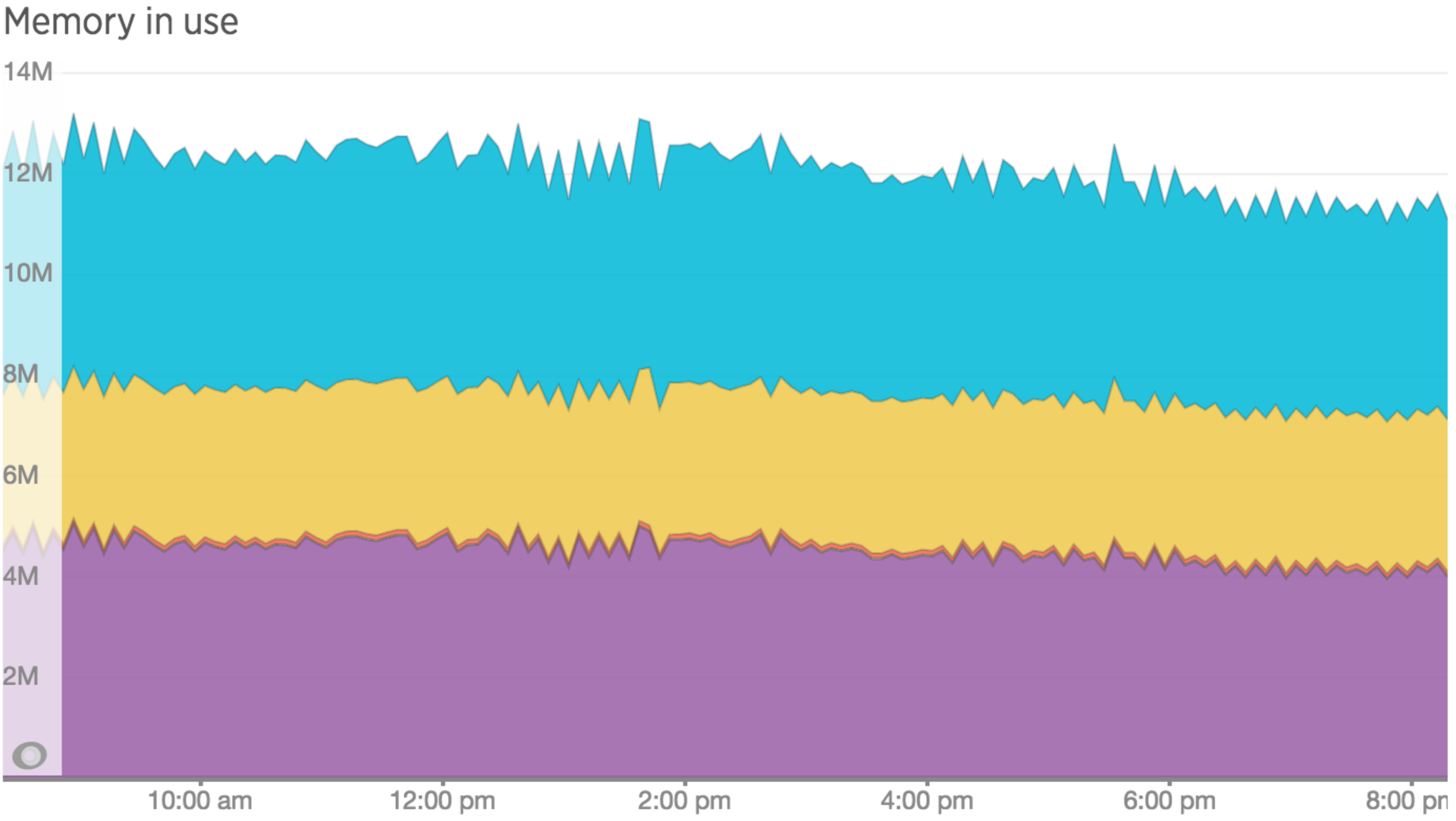
```
// Search for objects.
sha := “4c279186e24f7b3a59aa682a870747df6eaca013”
oid := git.NewOid(sha)

c, _ := r.LookupCommit(oid)
b, _ := r.LookupBlob(oid)
t, _ := r.LookupTree(oid)

o, _ := r.Lookup(oid)
fmt.Printf(“💃 %v\n”, o.Type())
```

```
// Read commit data.
sha  := “4c279186e24f7b3a59aa682a870747df6eaca013”
oid  := git.NewOid(sha)
path := “src/os/exec.go”

c, _ := r.LookupCommit(oid)
t, _ := c.Tree()
e, _ := t.EntryByPath(path)

b, _ := r.LookupBlob(e.Id())
fmt.Printf(“🎉 %q\n”, b.Contents())
```

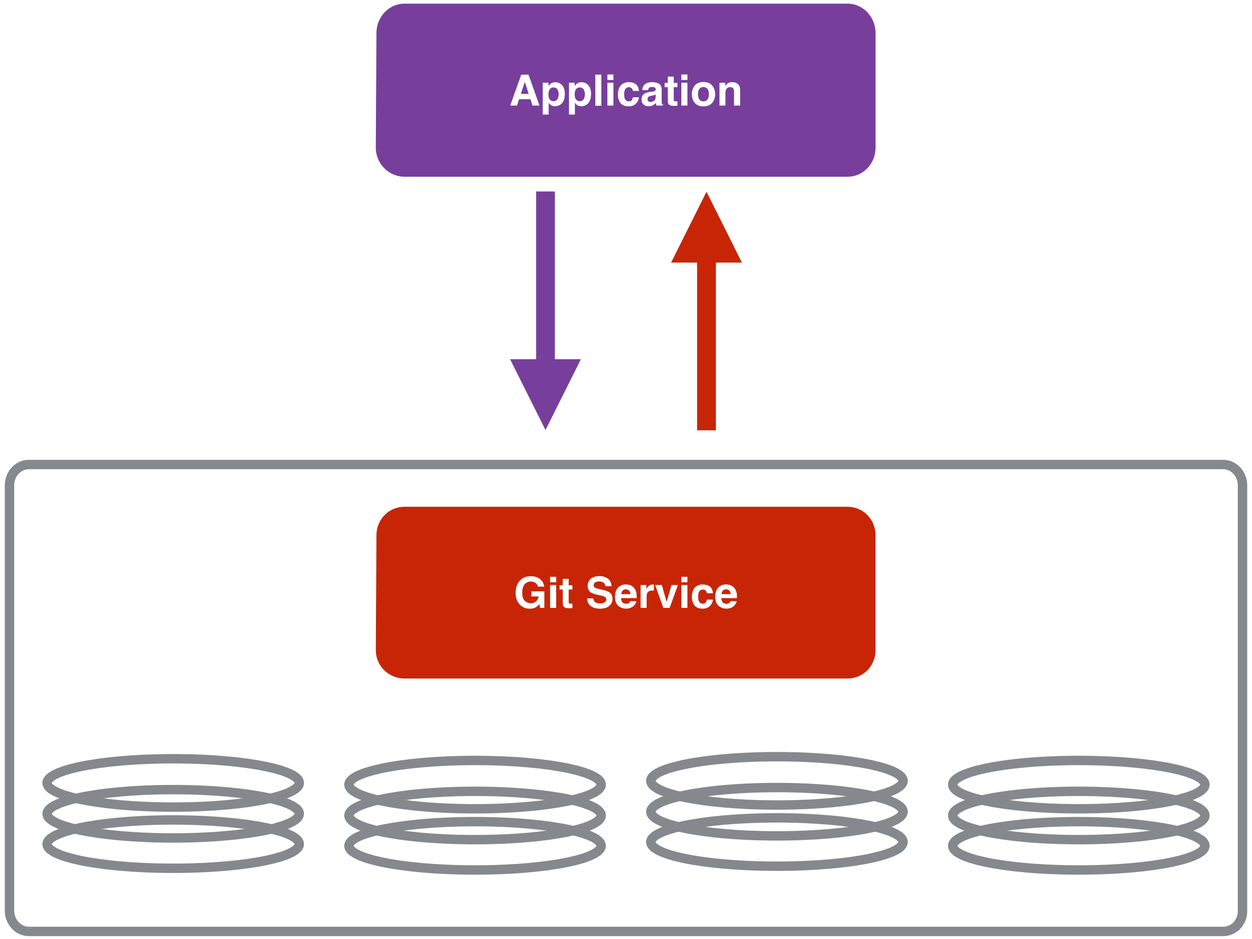
```
// Commit new changes.
idx, _ := r.Index()
idx.AddByPath(“src/os/exec.go”)
t, _   := idx.WriteTree()
idx.Write()

h, _ := r.Head()
c, _ := r.LookupCommit(h)

s := &git.Signature{“me”, “me@me.com”, time.Now()}
m := “Add moar changes”
r.CreateCommit(“”, s, s, m, t, c)
```

The average latency was also lower with [redacted]:

I'm removing the experiment and merging these changes.


Memory in use
14M
12M
10M
8M
6M
4M
2M
10:00 am
12:00 pm
2:00 pm
4:00 pm
6:00 pm
8:00 pm

Designing a distributed git storage


Application
Git Service

Constraint your data model

```
// protocol buffers schema.
message Branch {
  required string name = 1;
}

message Repository {
  optional string name = 1;
   repeated Branch branches = 2;
}
```

```
// Read branches.
var branches []*pb.Branch

f := func(b *git.Branch, t git.BranchType) error {
  n, _ := b.Name()
  p := &pb.Branch{
    Name: &n,
  }
  branches = append(branches, p)
  return nil
}

b, _ := r.NewBranchIterator(git.BranchRemote)
b.ForEach(f)
```

```
// Read branches via http.
h := func(w http.ResponseWriter, r *http.Request) {
  pbBranches := readBranches(r)
  pbRepo     := &pb.Repository{
    Branches: pbBranches,
    }

  data, _    := proto.Marshal(pbRepo)
  w.Write(data)
}

http.HandleFunc(“/r/foo/branches”, h)
```

Design from first principles

*A shared-data system can have at most two of the three following properties: **C**onsistency, **A**vailability, and tolerance to network **P**artitions*

Dr. Eric Brewer

You Can't Sacrifice Partition Tolerance

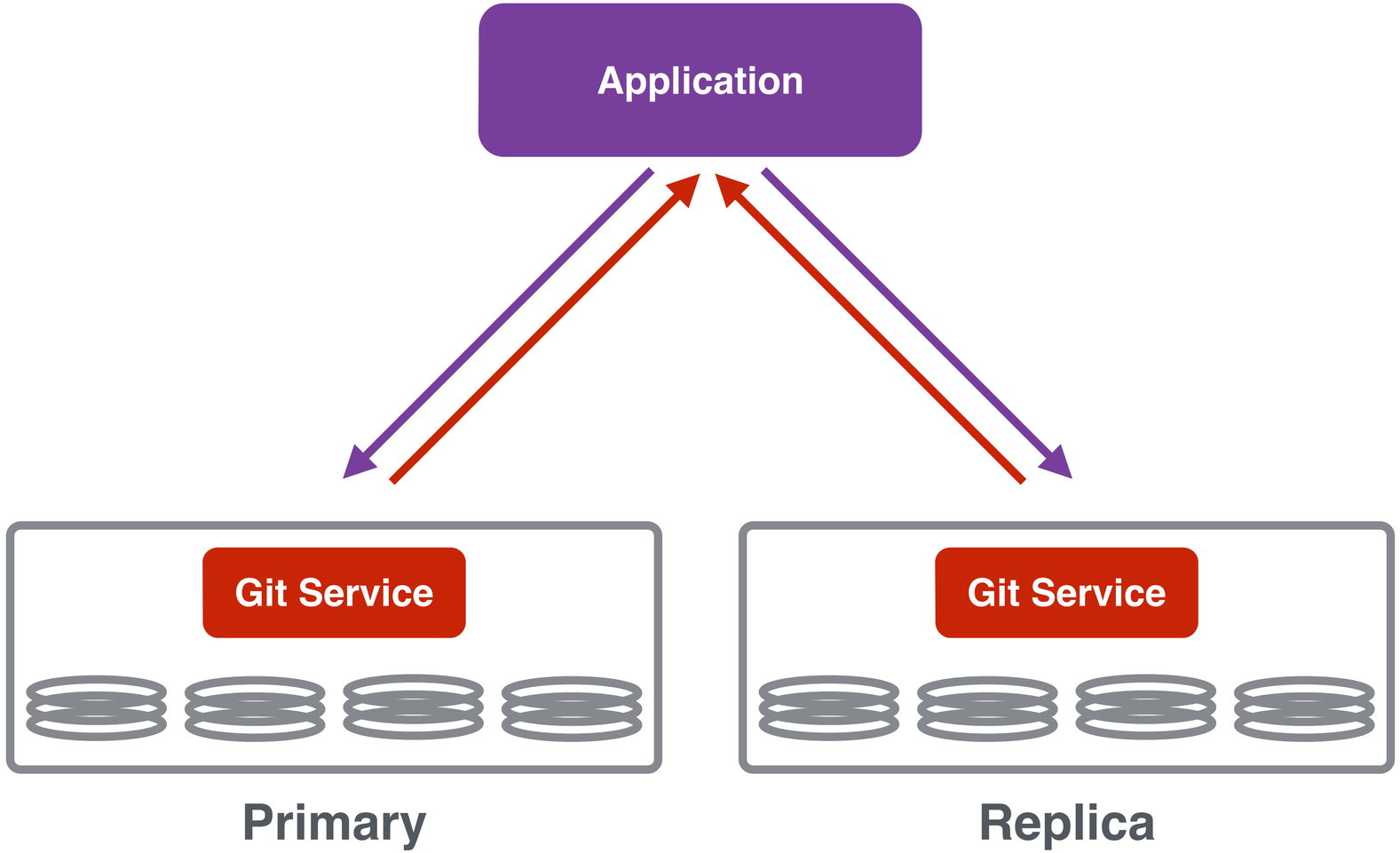
Coda Hale

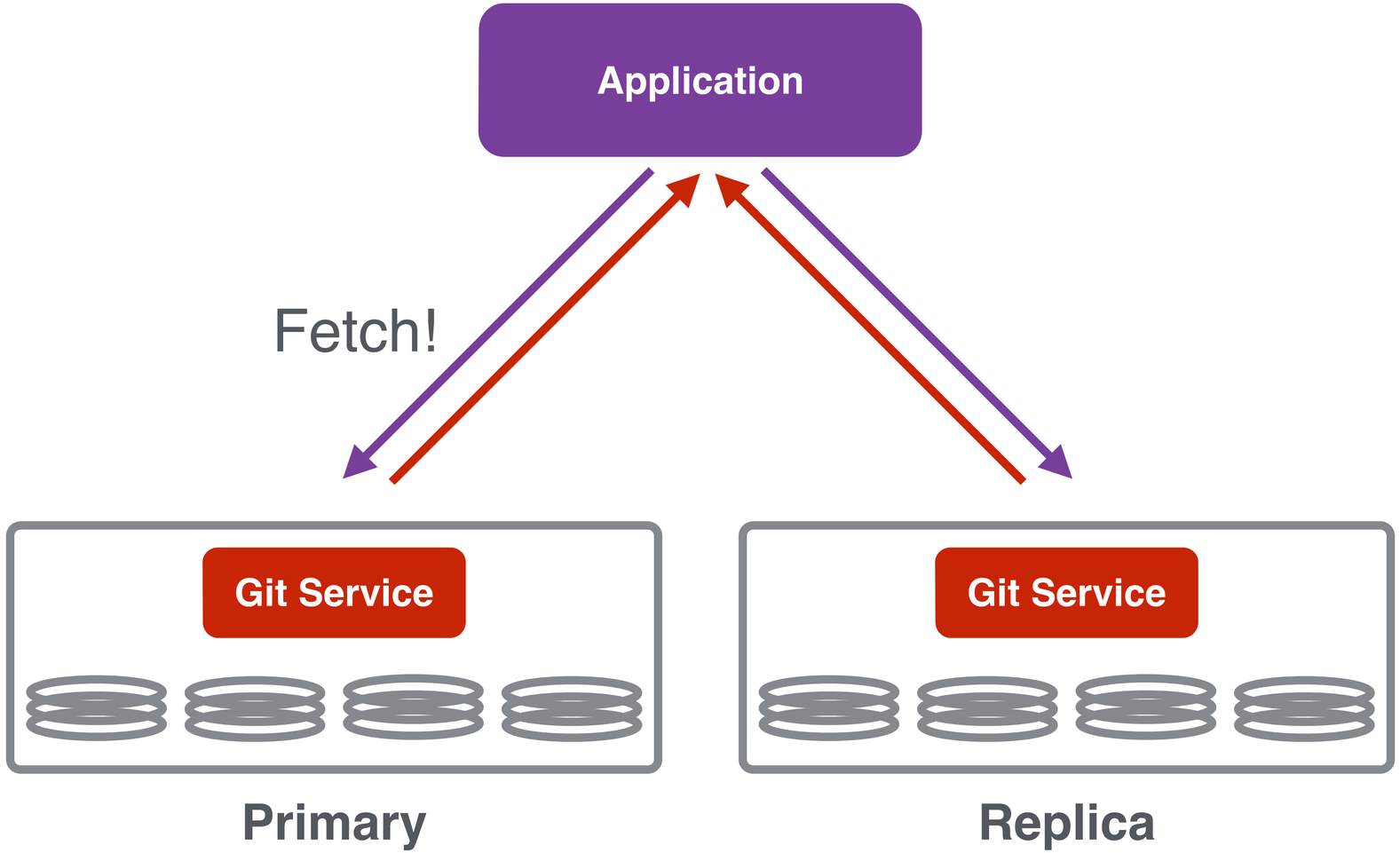
github.com/afex/hystrix-go
github.com/rubyist/circuitbreaker
github.com/eapache/go-resiliency/braker

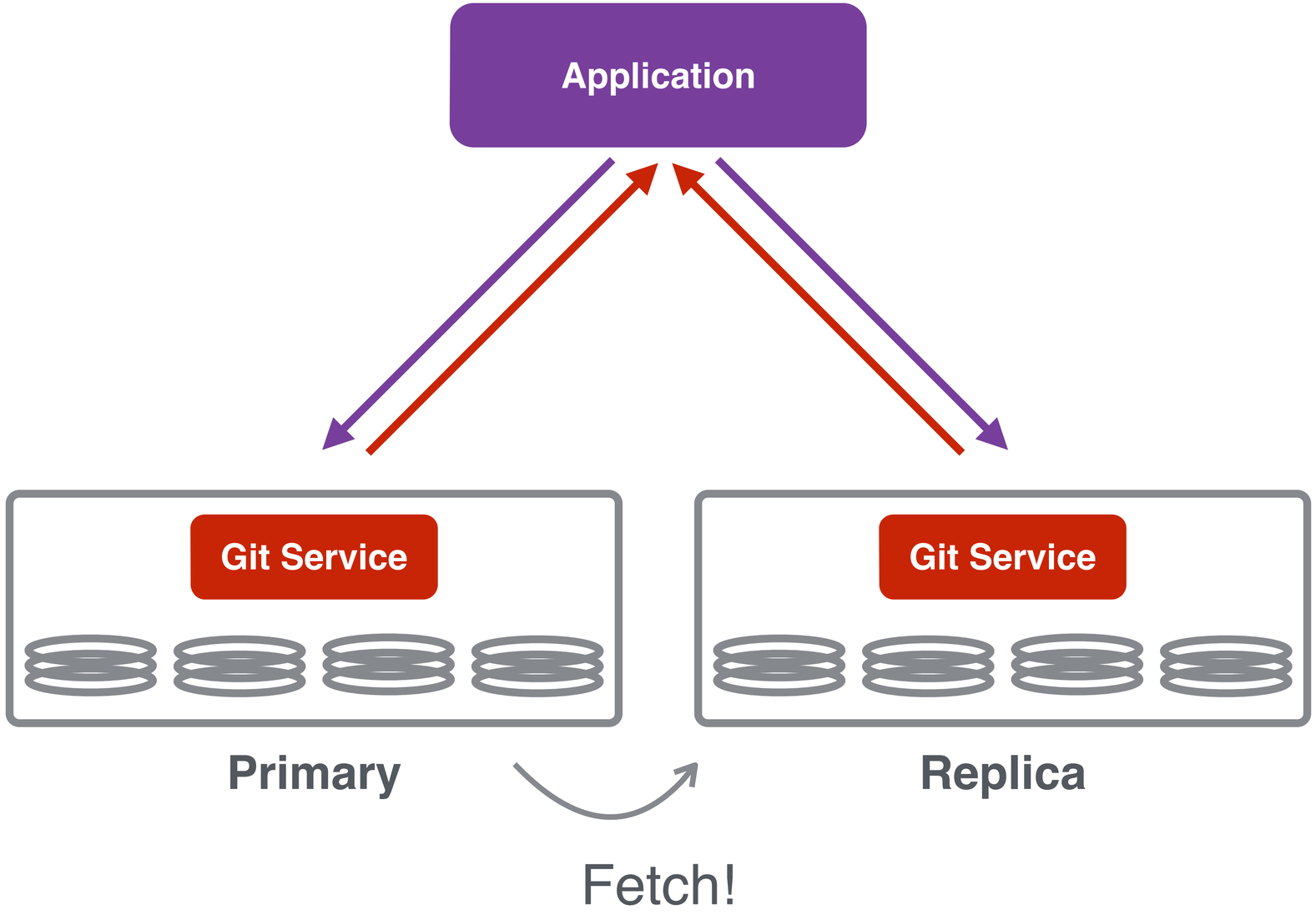
```
// Read branches via http.
import “github.com/rubyist/circuitbreaker”

out := 5 * time.Second
url := “http://git-server/r/foo/branches”
c   := circuit.NewHTTPClient(out, 10, nil)

c.BreakerTripped = func() {
  // Handle partition error response.
}

resp, _ := c.Get(url)
```

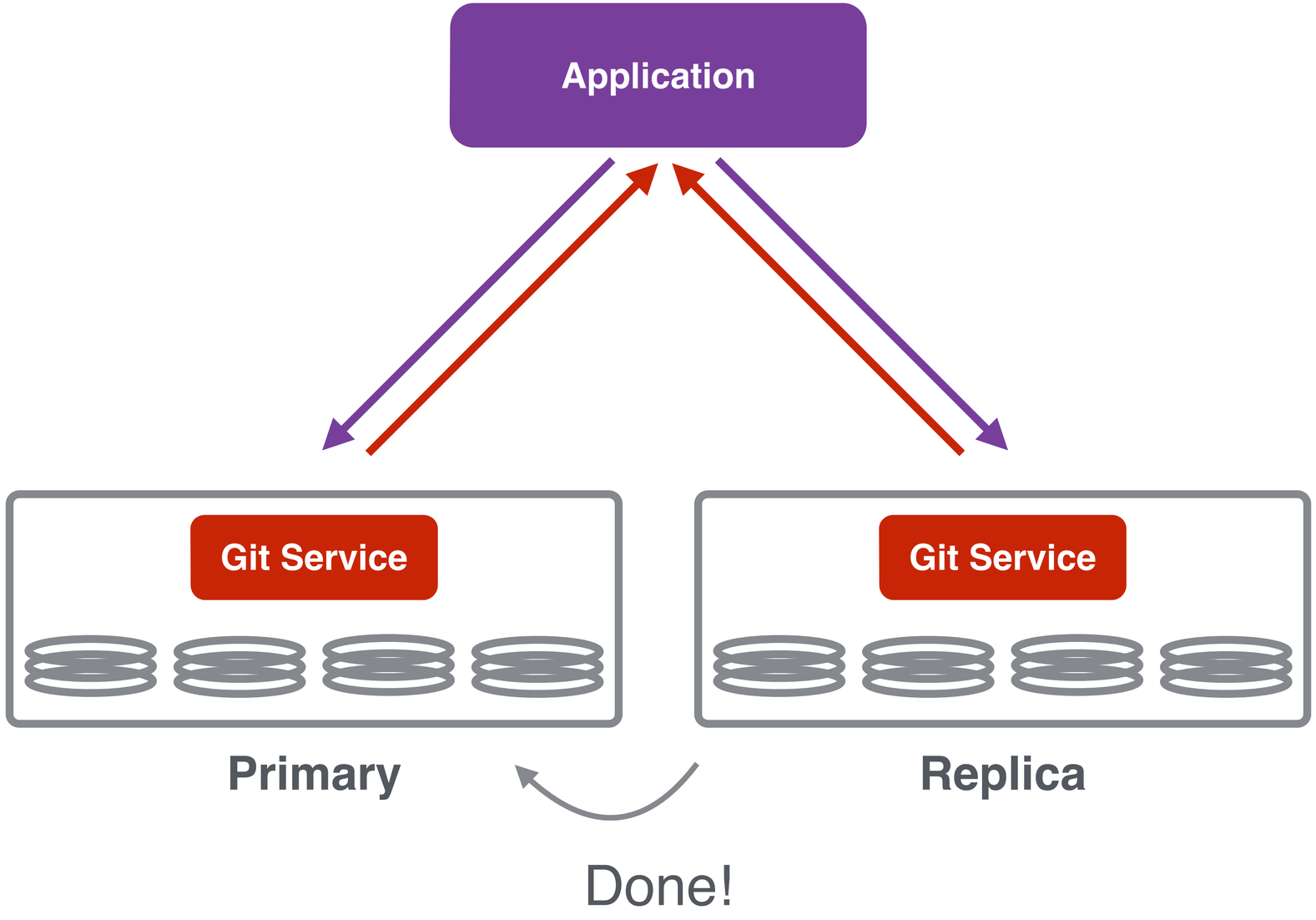
Replication

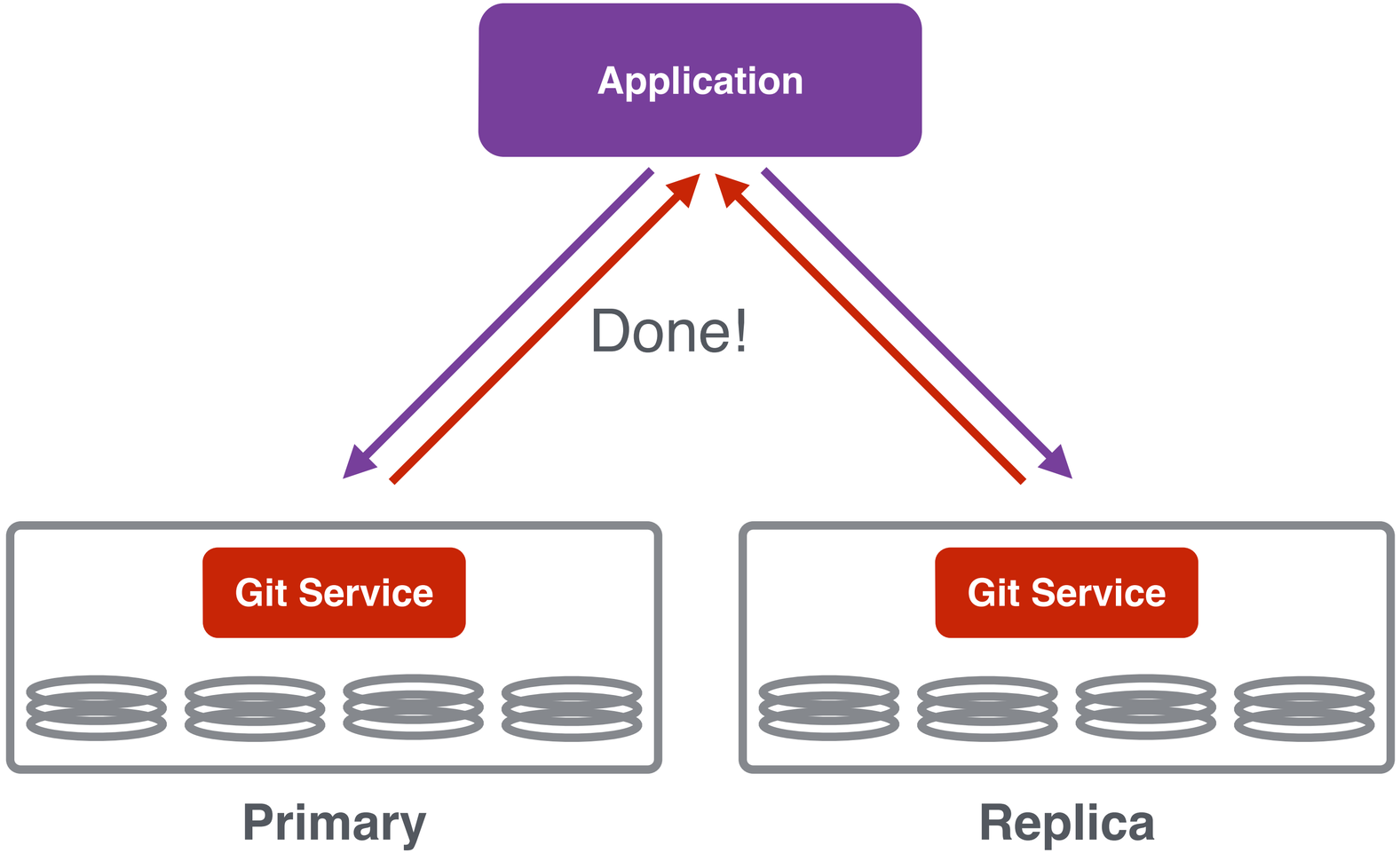
Consistency Vs Availability


Application
Git Service
Git Service
Primary
Replica


Application
Fetch!
Git Service
Git Service
Primary
Replica


Application
Git Service
Git Service
Primary
Replica
Fetch!


Application
Git Service
Git Service
Primary
Replica
Done!


Application
Done!
Git Service
Git Service
Primary
Replica

```
// Handle fetch requests.
h := func(w http.ResponseWriter, r *http.Request) {
  peerChannel := replicateRequest(r)

  rm, _ := repo.LoadRemote(“origin”)
  var refspecs []string
  rm.Fetch(refspecs, nil, nil)

  if peersChannel != nil {
    waitForPeers(peerChannel)
  }

  w.WriteHeader(201)
}

http.HandleFunc(“/r/foo/fetch”, h)
```

```
// Replicate request.
func replicateRequest(r *http.Request) chan int {
  if req.Header.Get(“X-GIT-REPLICATE”) != “” {
    return nil
  }
  peerChannel := make(chan int)

  replicaURL, err := url.Parse(replicaHost)
  replicaURL.Path = r.Path
  replicaURL.Header.Set(“X-GIT-REPLICATE”, “true")
  req, _ := http.NewRequest(“POST”, replicaURL.String(), nil)

  go func() {
    resp, _ := httpClient.Do(req)
    peerChannel <- resp.StatusCode
  }()

  return peerChannel
}
```

```
// Wait for replica response.
func waitForPeers(channel chan int) error {
  replicaStatus := <- channel

  switch replicaStatus {
    case 201:
      // 🎊 👏 🎊
    default:
      // ☔ 😔 ☔
  }

  return nil
}
```

git architectures are fun

Thank you!

@calavera