Stout
An Adaptive Interface to Scalable Cloud Storage

John C. McCullough John Dunagan[†]
Alec Wolman[†] Alex C. Snoeren

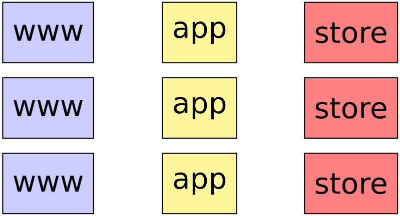
UC San Diego

Microsoft Research[†]

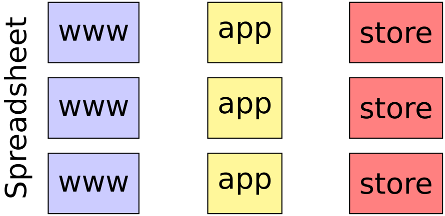
June 23, 2010

Scalable Multi-tiered Services

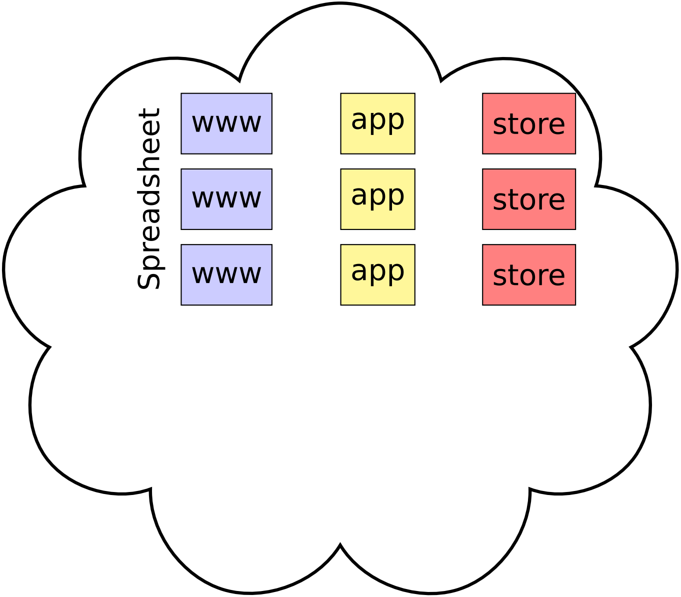
Scalable Multi-tiered Services



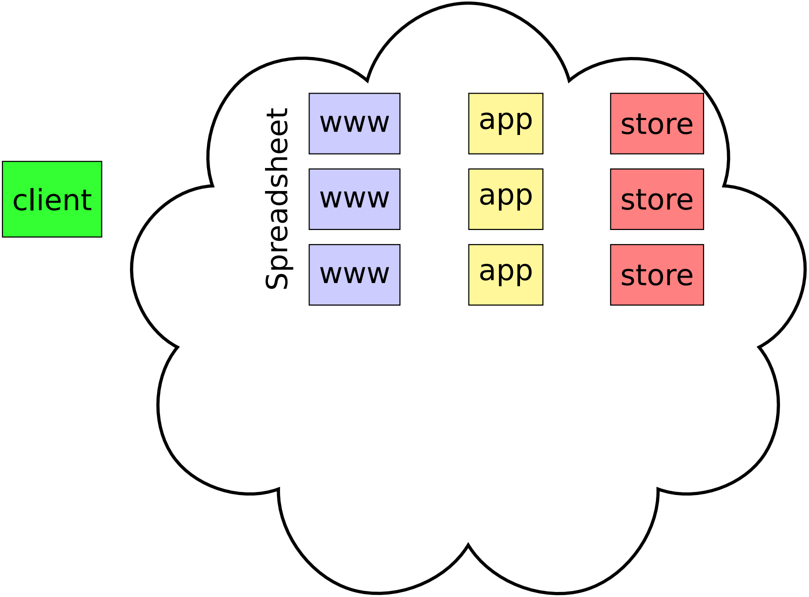
Scalable Multi-tiered Services



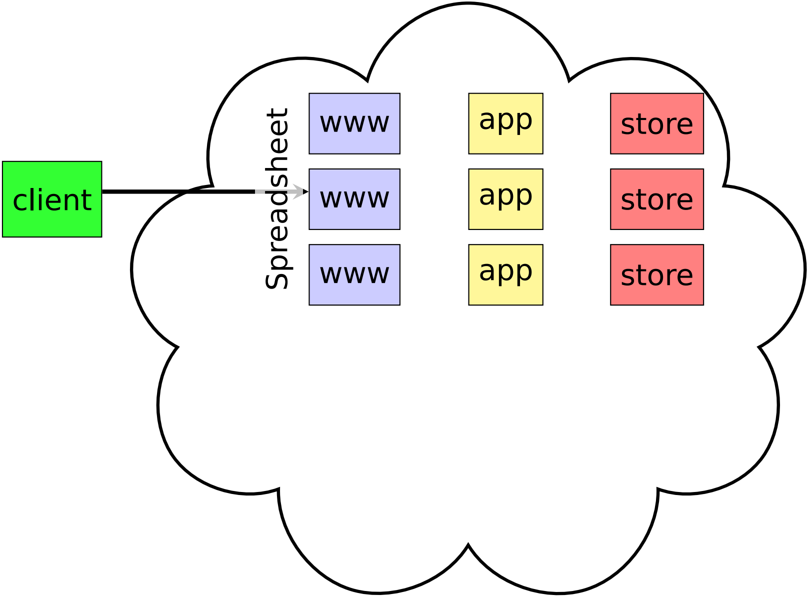
Scalable Multi-tiered Services



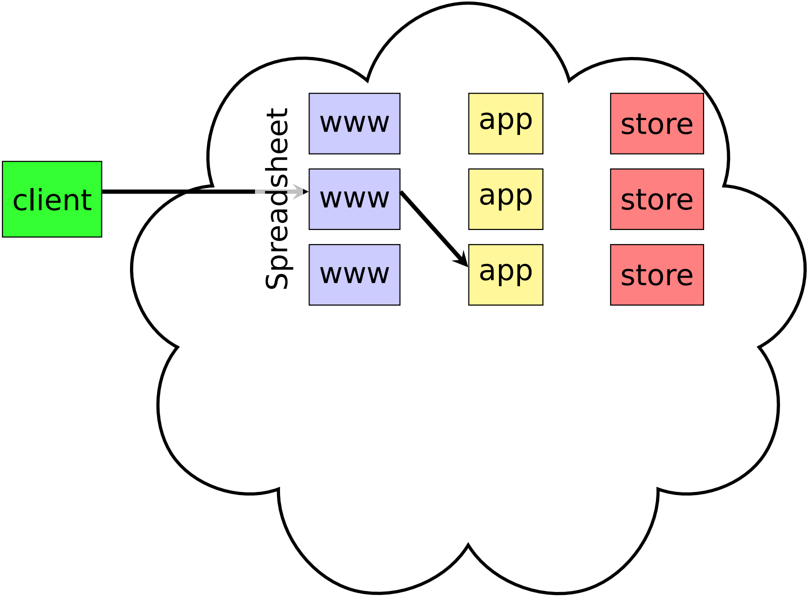
Scalable Multi-tiered Services



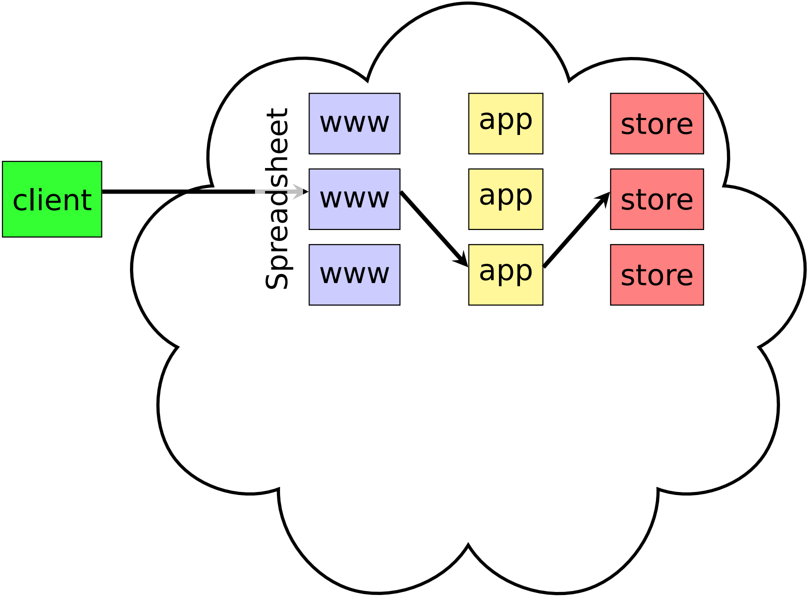
Scalable Multi-tiered Services



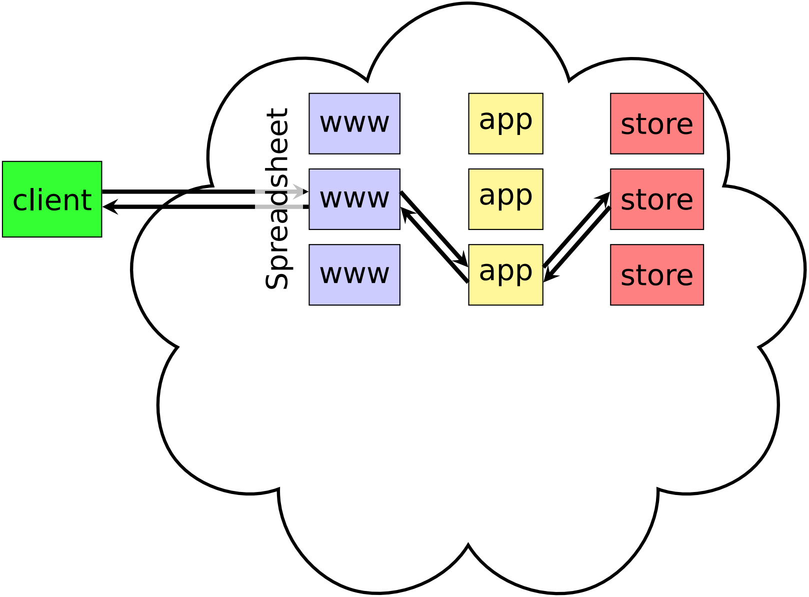
Scalable Multi-tiered Services



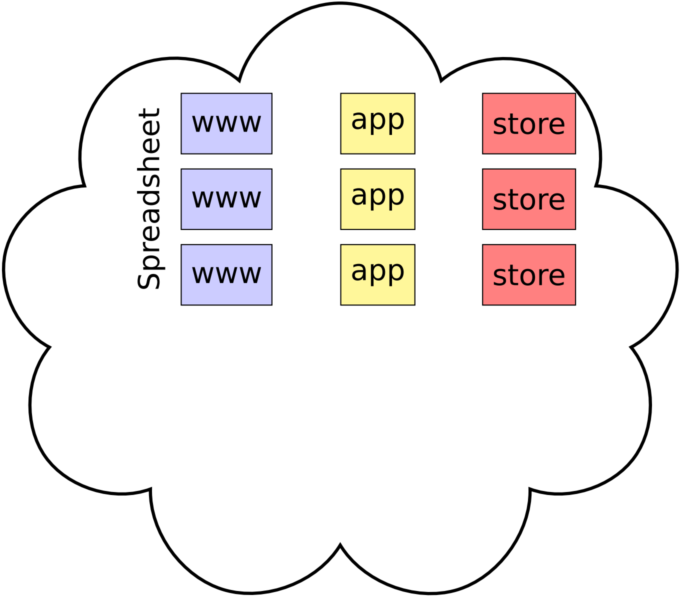
Scalable Multi-tiered Services



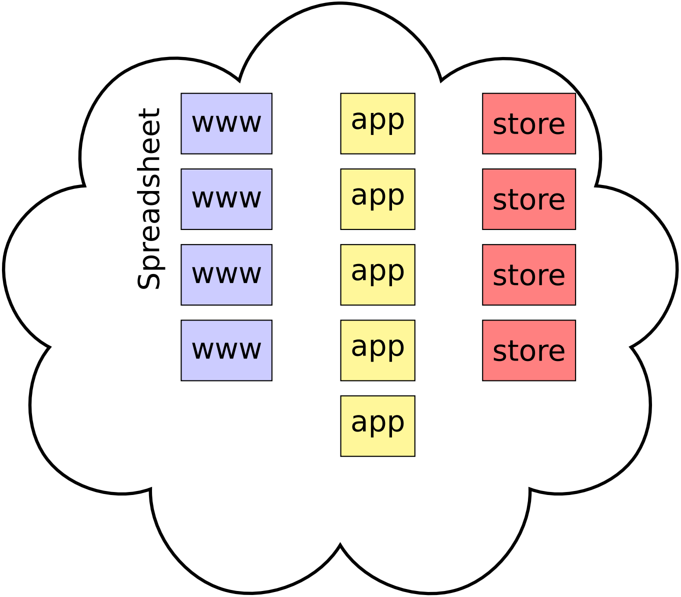
Scalable Multi-tiered Services



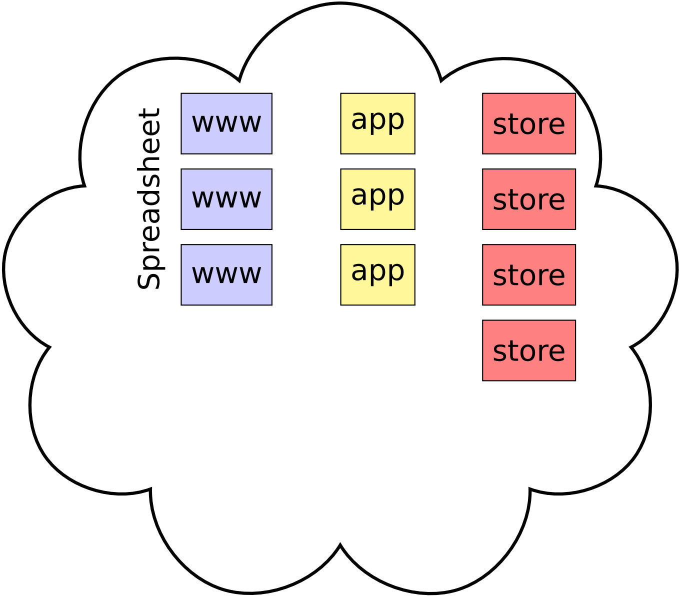
Scalable Multi-tiered Services



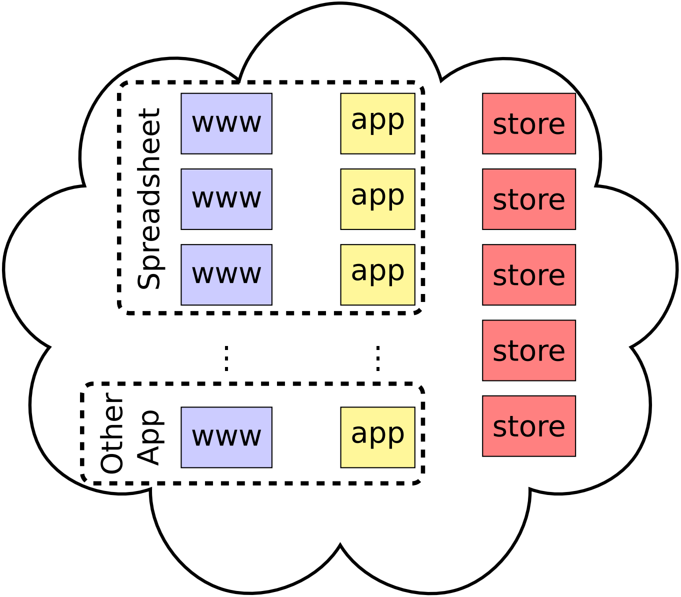
Scalable Multi-tiered Services



Scalable Multi-tiered Services



Scalable Multi-tiered Services



Scalable Multi-tiered Services

Key-Value Storage

app → store

- Simple interface
  - read(key) → value
  - write(key, value)
- Natural to send requests right away

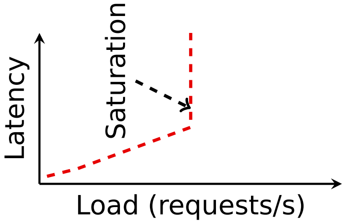
Key-Value Storage

- Simple interface
  - read(key) → value
  - write(key, value)
- Natural to send requests right away
- Block for response to survive failures

Key-Value Storage

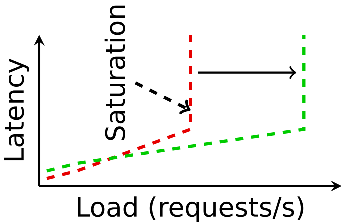
app → store

- Simple interface
  - read(key) → value
  - write(key, value)
- Natural to send requests right away
- Block for response to survive failures
- Performance characteristics:

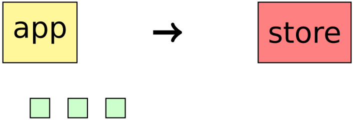


Key-Value Storage

- Simple interface
  - read(key) → value
  - write(key, value)
- Natural to send requests right away
- Block for response to survive failures
- Performance characteristics:

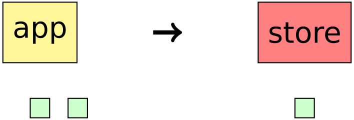


Improving Performance Under Load



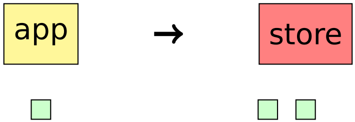
- Application server handles requests for many clients
- Storage request overheads
  - Networking delay
  - Protocol-processing
  - Disk seeks
  - etc.

Improving Performance Under Load



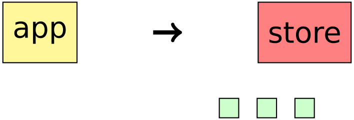
- Application server handles requests for many clients
- Storage request overheads
  - Networking delay
  - Protocol-processing
  - Disk seeks
  - etc.

Improving Performance Under Load



- Application server handles requests for many clients
- Storage request overheads
  - Networking delay
  - Protocol-processing
  - Disk seeks
  - etc.

Improving Performance Under Load



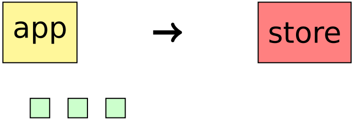
- Application server handles requests for many clients
- Storage request overheads
  - Networking delay
  - Protocol-processing
  - Disk seeks
  - etc.

Improving Performance Under Load

app → store

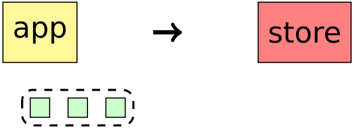
- Application server handles requests for many clients
- Storage request overheads
  - Networking delay
  - Protocol-processing
  - Disk seeks
  - etc.

Improving Performance Under Load



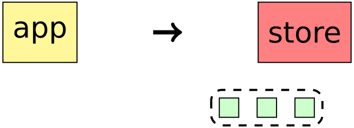
- Application server handles requests for many clients
- Storage request overheads
  - Networking delay
  - Protocol-processing
  - Disk seeks
  - etc.

Improving Performance Under Load



- Application server handles requests for many clients
- Storage request overheads
  - Networking delay
  - Protocol-processing
  - Disk seeks
  - etc.
- Batch to amortize overheads

Improving Performance Under Load



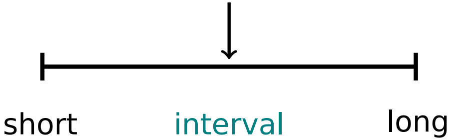
- Application server handles requests for many clients
- Storage request overheads
  - Networking delay
  - Protocol-processing
  - Disk seeks
  - etc.
- Batch to amortize overheads

Selecting a Batching Interval

- Most apps use a fixed batching interval

Selecting a Batching Interval

- Most apps use a fixed batching interval



Selecting a Batching Interval

- Most apps use a fixed batching interval
- Latency/throughput tradeoff

better throughput ⟶

⟵ better latency

short interval long

Selecting a Batching Interval

- Most apps use a fixed batching interval
- Latency/throughput tradeoff

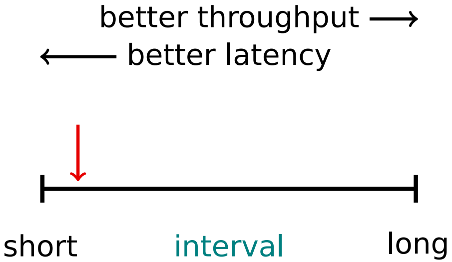
better throughput ⟶

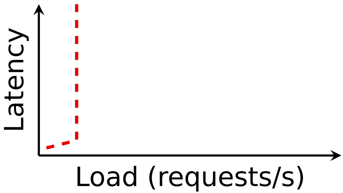
⟵ better latency

short interval long

Selecting a Batching Interval

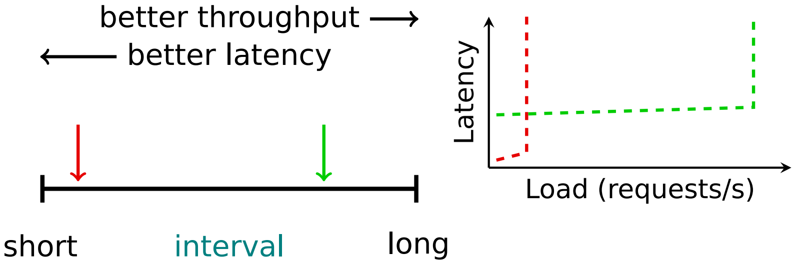
- Most apps use a fixed batching interval
- Latency/throughput tradeoff





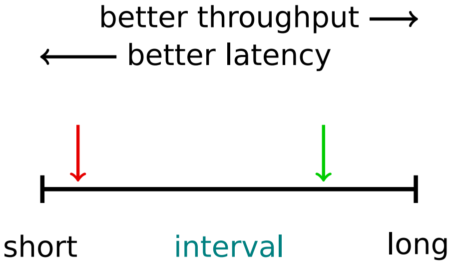
Selecting a Batching Interval

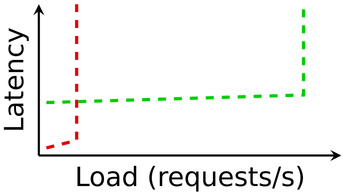
- Most apps use a fixed batching interval
- Latency/throughput tradeoff



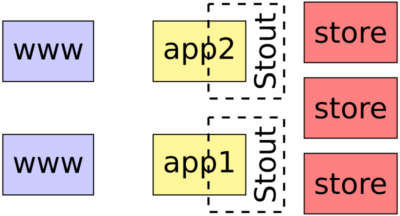
Selecting a Batching Interval

- Most apps use a fixed batching interval
- Latency/throughput tradeoff
- Want flexible batching interval
  - Short when lightly loaded
  - Long when heavily loaded





Solution: Stout

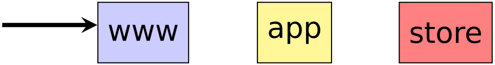


- Stout is a storage interposition library
- Our contribution is a technique for independently adjusting the batching interval

Outline

1. Introduction
2. Application Structure
3. Adaptive Batching
4. Evaluation

Overlapped Request Processing



Overlapped Request Processing

Overlapped Request Processing

ProcessRequest(req):

Overlapped Request Processing


```
ProcessRequest(req):
   key = Parse(req)
```

Overlapped Request Processing


```
ProcessRequest(req):
   key = Parse(req)
   Process(key,req)
```

Overlapped Request Processing


```
ProcessRequest(req):
   key = Parse(req)
   Process(key,req)
   PersistState(key)
```

Overlapped Request Processing


```
ProcessRequest(req):
   key = Parse(req)
   Process(key,req)
   PersistState(key)
```

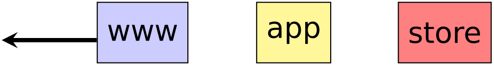
Overlapped Request Processing


```
ProcessRequest(req):
   key = Parse(req)
   Process(key,req)
   PersistState(key)
   reply = MakeReply(req)
```

Overlapped Request Processing


```
ProcessRequest(req):
   key = Parse(req)
   Process(key,req)
   PersistState(key)
   reply = MakeReply(req)
   SendReply(reply)
```

Overlapped Request Processing

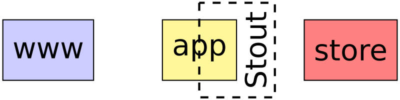



```
ProcessRequest(req):
   key = Parse(req)
   Process(key,req)
   PersistState(key)
   reply = MakeReply(req)
   SendReply(reply)
```

Overlapped Request Processing

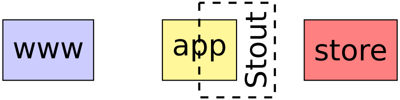

```
ProcessRequest(req):
   key = Parse(req)
   Process(key,req)
   PersistState(key)
   reply = MakeReply(req)
   SendReply(reply)
```

Overlapped Request Processing



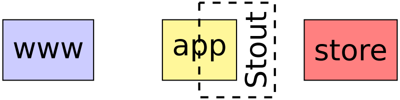

```
ProcessRequest(req):
   key = Parse(req)
   Process(key,req)
   PersistState(key)
   reply = MakeReply(req)
   SendReply(reply)
```

Overlapped Request Processing



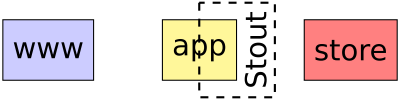

```
ProcessRequest(req):
   key = Parse(req)
   Process(key,req)
   MarkDirty(key)
   reply = MakeReply(req)
   SendReply(reply)
```

Overlapped Request Processing




```
ProcessRequest(req):
   key = Parse(req)
   Process(key,req)
   MarkDirty(key)
   reply = MakeReply(req)
   SafeReply(key,reply)
```

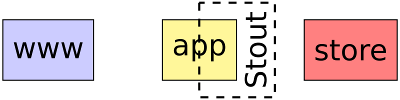
Overlapped Request Processing




```
ProcessRequest(req):
   key = Parse(req)
   Process(key,req)
   MarkDirty(key)
   reply = MakeReply(req)
   SafeReply(key,reply)
```

```
BatchingLoop:
   keys = DirtyKeys()
   replies = Depends(keys)
   AsyncWrite(keys, replies)
   Sleep(interval)
```

Overlapped Request Processing

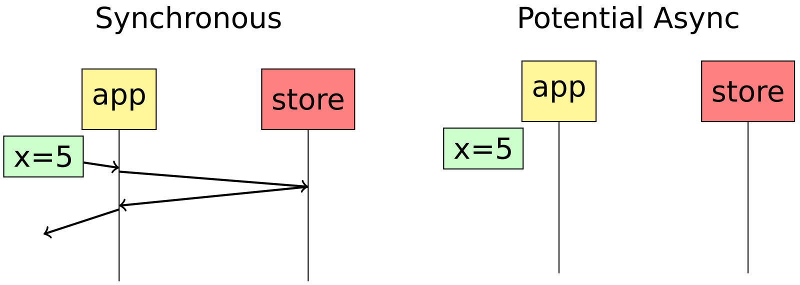



```
ProcessRequest(req):
   key = Parse(req)
   Process(key,req)
   MarkDirty(key)
   reply = MakeReply(req)
   SafeReply(key,reply)  ←
```

```
BatchingLoop:
   keys = DirtyKeys()
   replies = Depends(keys)
   AsyncWrite(keys, replies)
   Sleep(interval)  ←
```

Staying Safe: Consistency

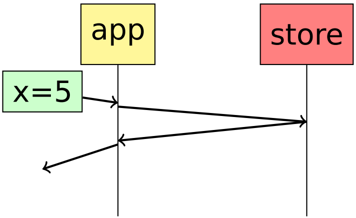
- Don’t reveal uncomitted state



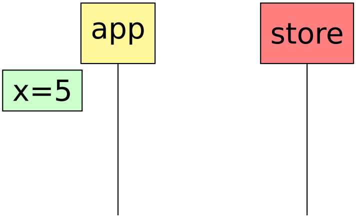
Staying Safe: Consistency

- Don't reveal uncomitted state
- Potential async: Inconsistency on failure

Synchronous

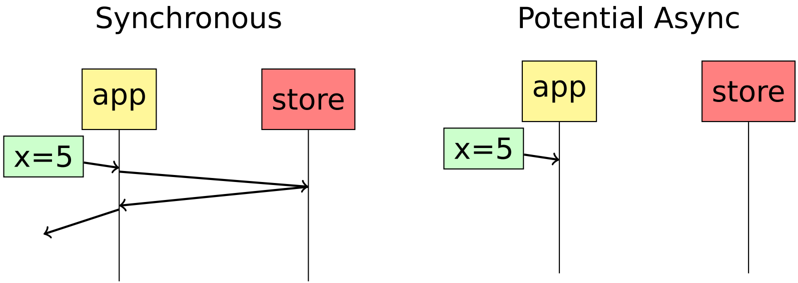


Potential Async



Staying Safe: Consistency

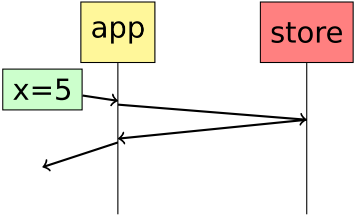
- Don’t reveal uncomitted state
- Potential async: Inconsistency on failure



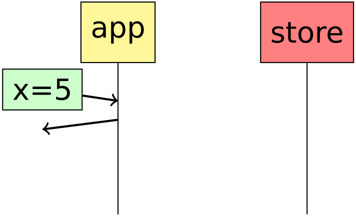
Staying Safe: Consistency

- Don’t reveal uncomitted state
- Potential async: Inconsistency on failure

Synchronous

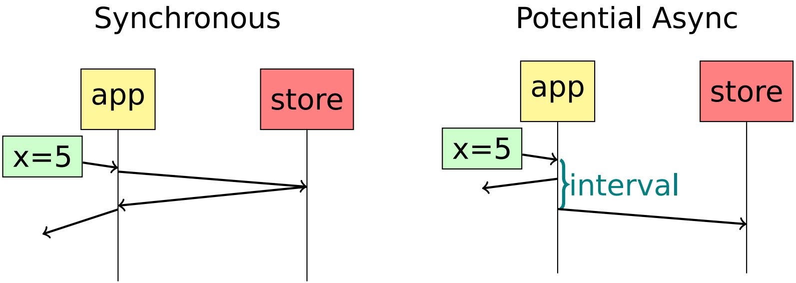


Potential Async



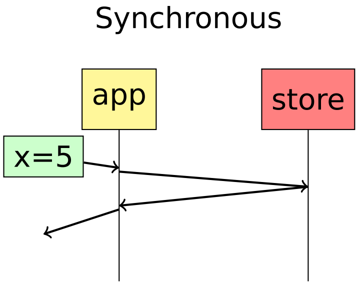
Staying Safe: Consistency

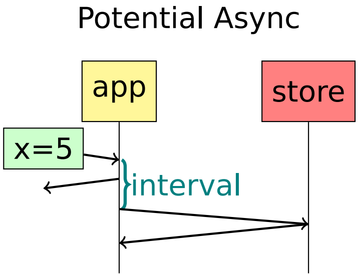
- Don't reveal uncomitted state
- Potential async: Inconsistency on failure



Staying Safe: Consistency

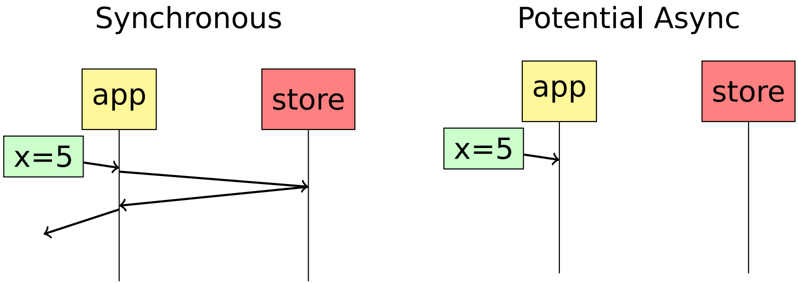
- Don’t reveal uncomitted state
- Potential async: Inconsistency on failure





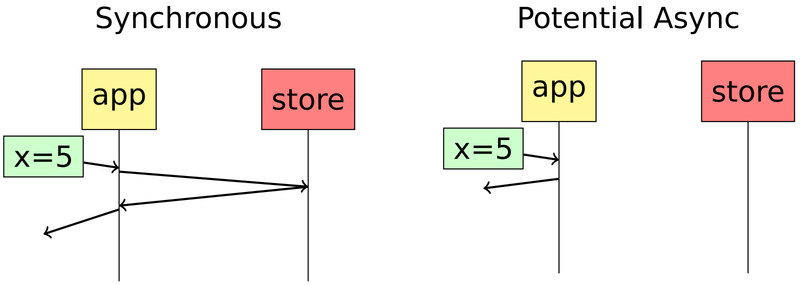
Staying Safe: Consistency

- Don't reveal uncomitted state
- Potential async: Inconsistency on failure



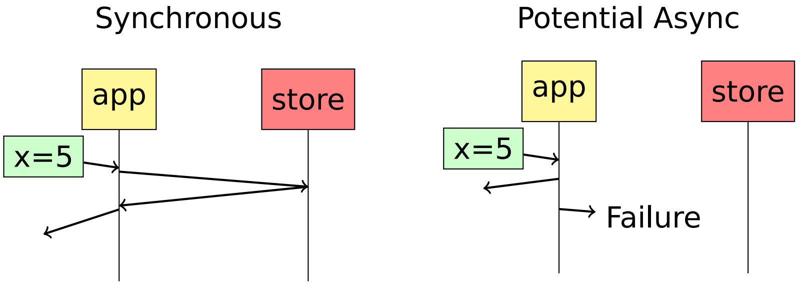
Staying Safe: Consistency

- Don’t reveal uncomitted state
- Potential async: Inconsistency on failure



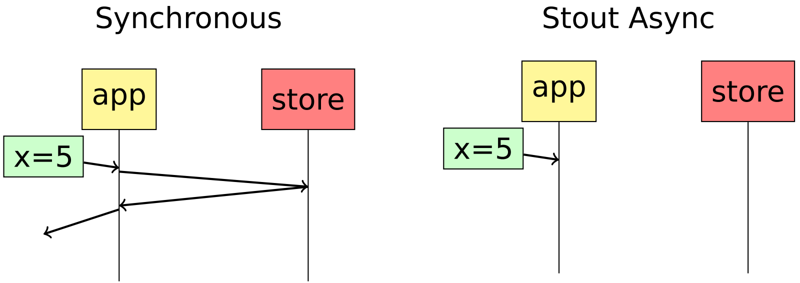
Staying Safe: Consistency

- Don't reveal uncomitted state
- Potential async: Inconsistency on failure



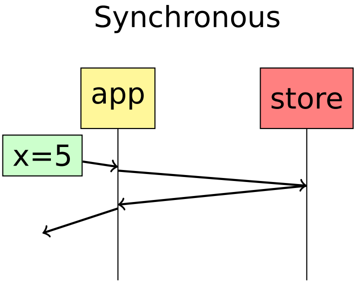
Staying Safe: Consistency

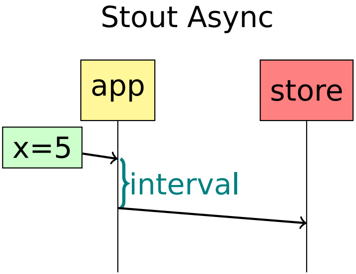
- Don't reveal uncomitted state
- Potential async: Inconsistency on failure
- Stout provides serialized update semantics



Staying Safe: Consistency

- Don't reveal uncomitted state
- Potential async: Inconsistency on failure
- Stout provides serialized update semantics

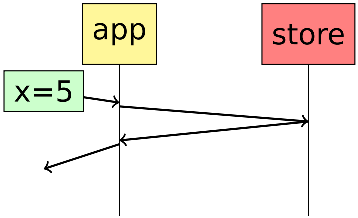




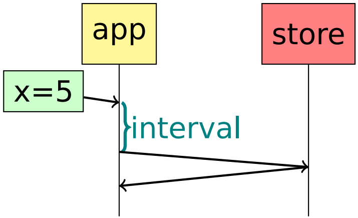
Staying Safe: Consistency

- Don’t reveal uncomitted state
- Potential async: Inconsistency on failure
- Stout provides serialized update semantics

Synchronous

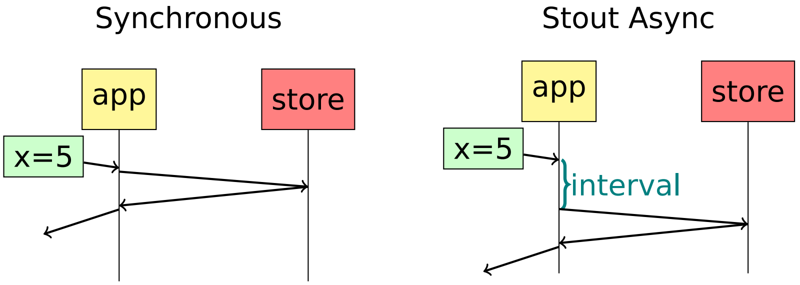


Stout Async



Staying Safe: Consistency

- Don't reveal uncomitted state
- Potential async: Inconsistency on failure
- Stout provides serialized update semantics



Benefit: Write Collapsing

- Batched commits enable further optimization
- Can write most recent version only
- Reduces load at the store

Benefit: Write Collapsing

- Batched commits enable further optimization
- Can write most recent version only
- Reduces load at the store

x=5

Benefit: Write Collapsing

- Batched commits enable further optimization
- Can write most recent version only
- Reduces load at the store

x=5

x=6

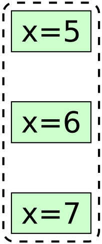
Benefit: Write Collapsing

- Batched commits enable further optimization
- Can write most recent version only
- Reduces load at the store

x=5
x=6
x=7

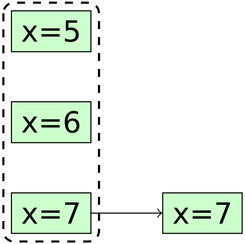
Benefit: Write Collapsing

- Batched commits enable further optimization
- Can write most recent version only
- Reduces load at the store



Benefit: Write Collapsing

- Batched commits enable further optimization
- Can write most recent version only
- Reduces load at the store

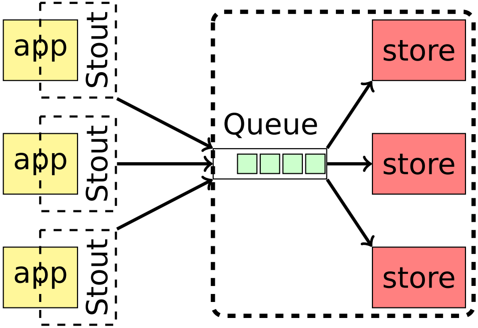


Outline

1. Introduction
2. Application Structure
3. Adaptive Batching
4. Evaluation

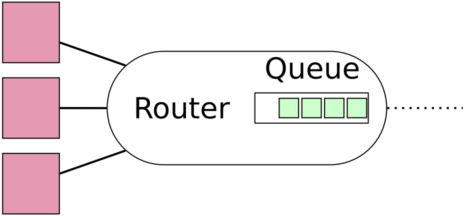
Adapting to Shared Storage

- Storage system is a shared medium
- Independently reach efficient fair share
- Delay as congestion indicator
  - Rather than modifying storage for explicit notification



Delay-based Congestion Control

- Unknown bottleneck capacity
- Traditional TCP signaled via packet loss
- Delay-based congestion control triggered by latency changes



Applications to Storage

	Networking	Storage
Mechanism	Change Rate	Change Size
ACCELERATE	Send Faster	Batch Less
BACK-OFF	Send Slower	Batch More

Algorithm

if perf < recent perf

    BACK-OFF

else

    ACCELERATE

Algorithm: Estimating Storage Performance

if perf < recent perf

  BACK-OFF

else

  ACCELERATE

$$\frac{\text{batch size}}{\mathit{latency} + \text{interval}}$$

Algorithm: Estimating Storage Capacity

if perf < recent perf

BACK-OFF

else

ACCELERATE

if backed-off

$$\frac{EWMA(\text{batch size}_i)}{EWMA(lat_i) + EWMA(\text{interval}_i)}$$

else // accelerated

$$MAX_i\left(\frac{\text{batch size}_i}{lat_i + \text{interval}_i}\right)$$

Algorithm: Achieving Fair Share

if perf < recent perf

  BACK-OFF

else

  ACCELERATE

Algorithm: Achieving Fair Share

if perf < recent perf

BACK-OFF $\longrightarrow (1+\alpha) * \text{interval}_i$

else

ACCELERATE

Algorithm: Achieving Fair Share

if perf < recent perf

BACK-OFF $\longrightarrow (1 + \alpha) * \text{interval}_i$

else

ACCELERATE $\longrightarrow (1 - \beta) * \text{interval}_i + \beta * \sqrt{\text{interval}_i}$

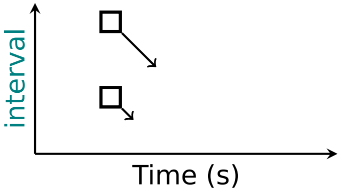
Algorithm: Achieving Fair Share

if perf < recent perf

BACK-OFF $\longrightarrow (1+\alpha) * \text{interval}_i$

else

ACCELERATE $\longrightarrow (1-\beta) * \text{interval}_i + \beta * \sqrt{\text{interval}_i}$



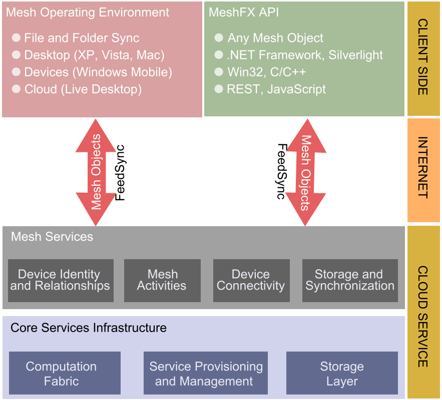
Outline

1. Introduction
2. Application Structure
3. Adaptive Batching
4. Evaluation

Evaluation

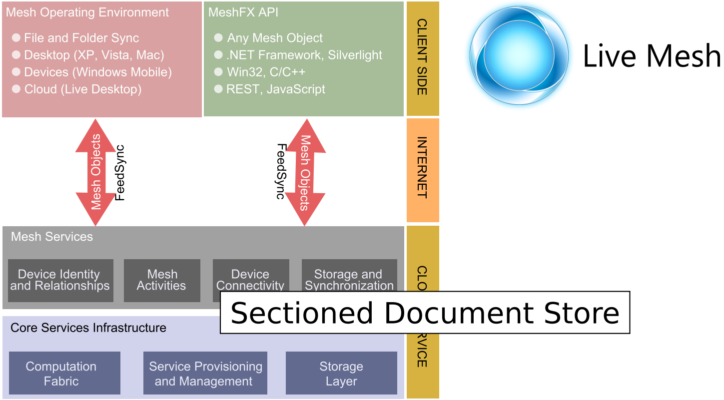
- Baseline Storage System Performance
  - Benefits of batching
  - Benefits of write-collapsing
- Stout
  - Versus fixed batching intervals
  - Workload variation

Evaluation



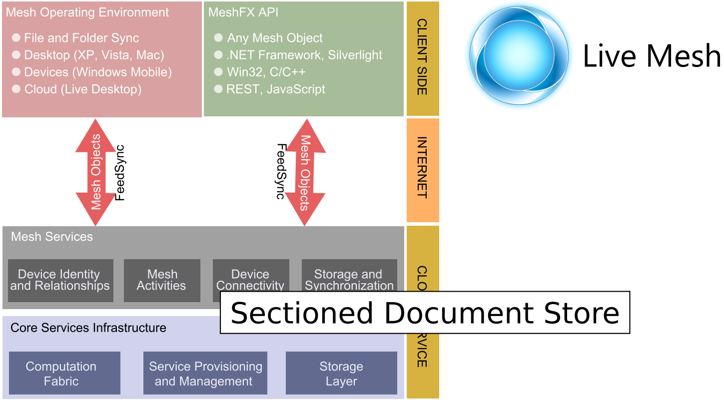
Live Mesh

Evaluation



Mesh Operating Environment
File and Folder Sync
Desktop (XP, Vista, Mac)
Devices (Windows Mobile)
Cloud (Live Desktop)
MeshFX API
Any Mesh Object
.NET Framework, Silverlight
Win32, C/C++
REST, JavaScript
CLIENT SIDE
Mesh Objects
FeedSync
FeedSync
Mesh Objects
INTERNET
Mesh Services
Device Identity and Relationships
Mesh Activities
Device Connectivity
Storage and Synchronization
Core Services Infrastructure
Computation Fabric
Service Provisioning and Management
Storage Layer
Sectioned Document Store
Live Mesh

Evaluation



Our Workload

- 256-byte documents: IOPS dominated
- 50% read, 50% write

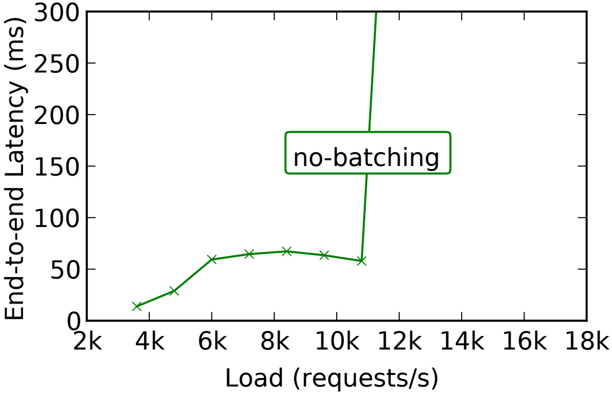
Evaluation: Configuration

Evaluation Platform

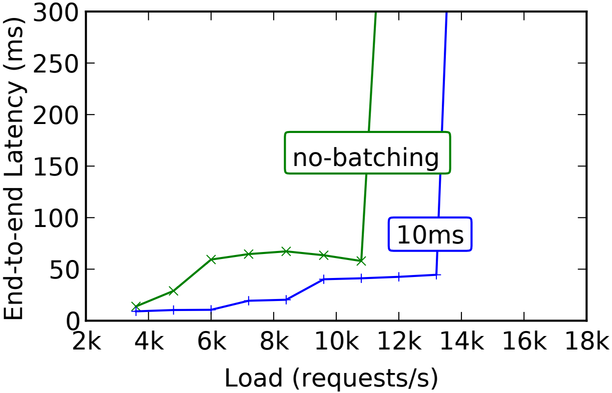
- 50 machines
  - 1 Experiment Controller
  - 1 Lease Manager
  - 12 Frontends
  - 32 Middle Tiers
  - 4 Storage (Partitioned Key-Value w/MSSQL as storage)

12× www 32× app 4× store

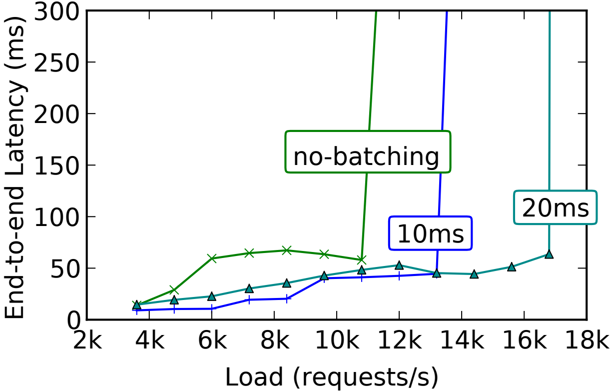
Baseline: Importance of Batching



Baseline: Importance of Batching

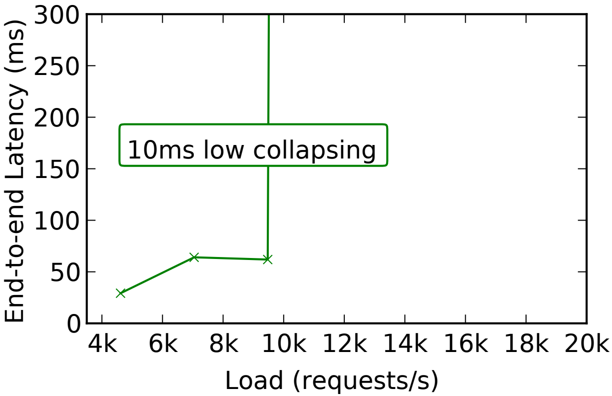


Baseline: Importance of Batching



- Batching improves performance

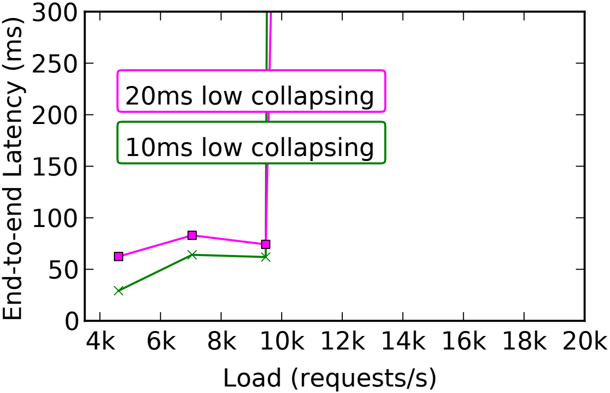
Baseline: Importance of Write-Collapsing



Low collapsing 10k Documents

High collapsing 100 Documents

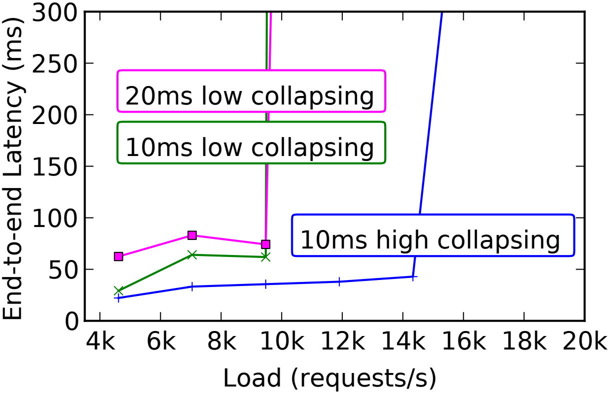
Baseline: Importance of Write-Collapsing



Low collapsing 10k Documents

High collapsing 100 Documents

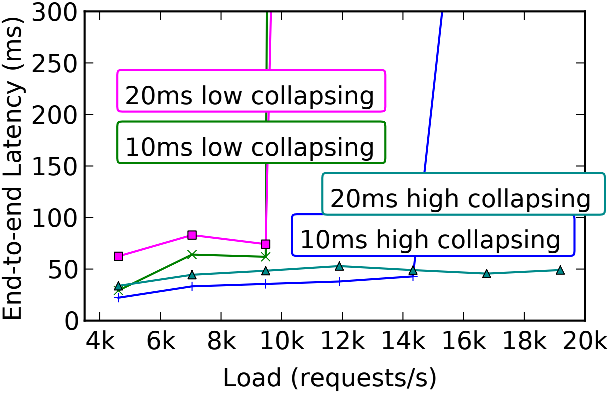
Baseline: Importance of Write-Collapsing



Low collapsing 10k Documents

High collapsing 100 Documents

Baseline: Importance of Write-Collapsing

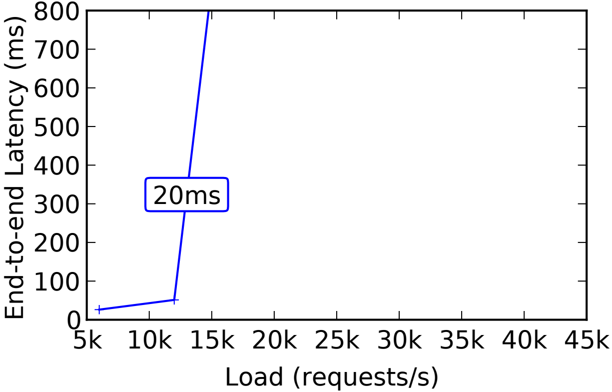


Low collapsing 10k Documents

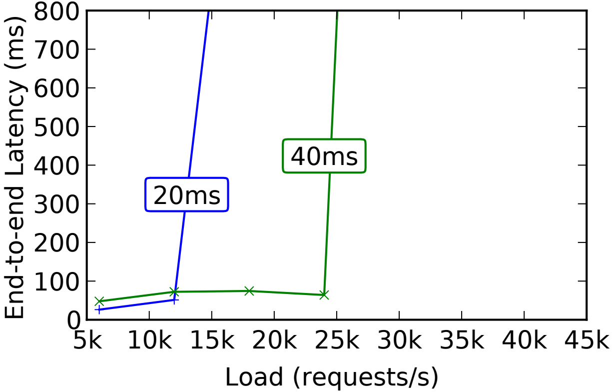
High collapsing 100 Documents

- Improvement dependent on workload

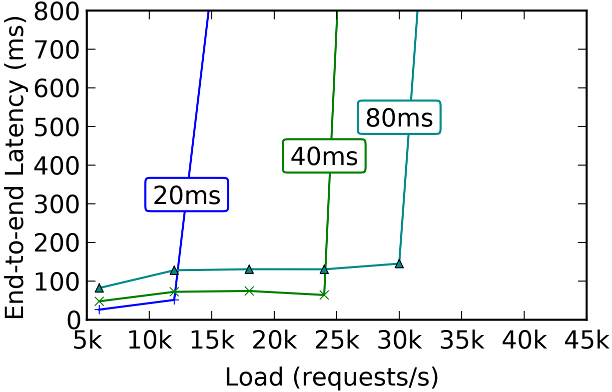
Evaluation: Stout vs. Fixed Intervals



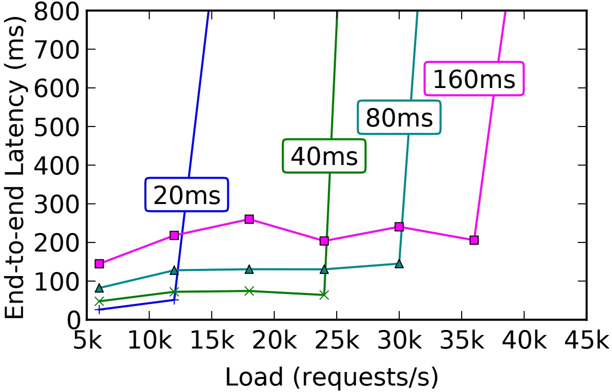
Evaluation: Stout vs. Fixed Intervals



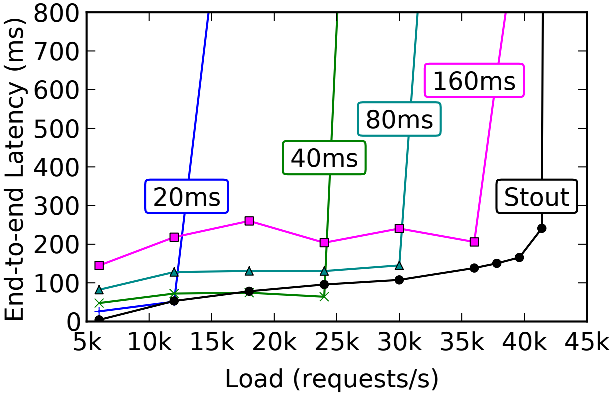
Evaluation: Stout vs. Fixed Intervals



Evaluation: Stout vs. Fixed Intervals

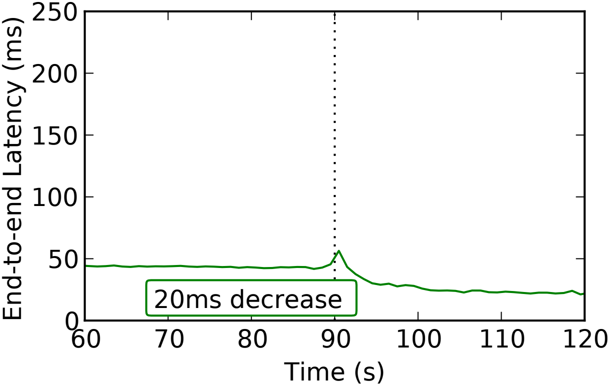


Evaluation: Stout vs. Fixed Intervals



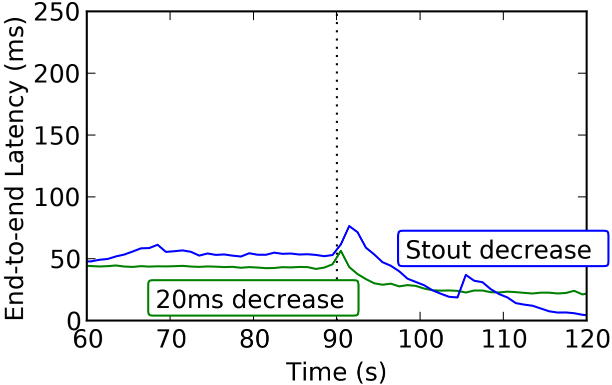
- Stout better than any fixed interval across wide range of workloads

Evaluation: Workload Variation



Decrease	12k requests/s → 8k requests/s
Increase	12k requests/s → 18k requests/s

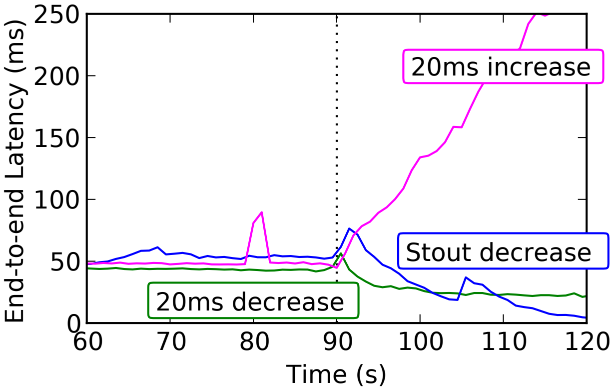
Evaluation: Workload Variation



Decrease 12k requests/s → 8k requests/s

Increase 12k requests/s → 18k requests/s

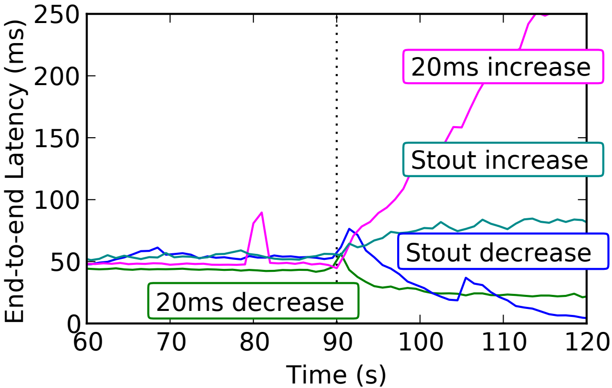
Evaluation: Workload Variation



Decrease 12k requests/s → 8k requests/s

Increase 12k requests/s → 18k requests/s

Evaluation: Workload Variation



Decrease 12k requests/s → 8k requests/s

Increase 12k requests/s → 18k requests/s

Additional Evaluation

- Fairness (Jain's Fairness index of 0.96)
- Stout achieves similar performance with:
  - PacificA
  - SQL Data Services

Conclusion

- Batching improves storage performance
- Current practice is fixed latency/throughput tradeoff
- Stout introduces distributed adaptation technique
- Achieve 3× higher throughput over low-latency fixed interval for modified Live Mesh service

Questions?