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

Diagram:
- Client
- www
- www
- www
- app
- app
- app
- store
- store
- store

Flow:
Client → www → www → www → app → app → app → store → store → store
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

- 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

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

![Graph showing performance characteristics](image-url)
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:

![Graph showing performance characteristics: Load (requests/s) vs Latency and Saturation](image)

Load (requests/s)  
Latency  
Saturation
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.

Diagram:

```
  app   →   store
```
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

- 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

<table>
<thead>
<tr>
<th>Load (requests/s)</th>
<th>Latency</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short</td>
<td>Better</td>
<td>Long</td>
</tr>
<tr>
<td>Long</td>
<td>Better</td>
<td>Short</td>
</tr>
</tbody>
</table>
Selecting a Batching Interval

- Most apps use a fixed batching interval

<table>
<thead>
<tr>
<th>short</th>
<th>interval</th>
<th>long</th>
</tr>
</thead>
</table>

Load (requests/s) | Latency | Throughput | Latency/throughput tradeoff |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>Better latency and throughput</td>
</tr>
</tbody>
</table>
Selecting a Batching Interval

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

![Diagram showing the tradeoff between latency and throughput with short and long intervals]

- Better throughput →
- Better latency ←

- Short interval
- Long interval
Selecting a Batching Interval

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

![Diagram showing latency and throughput tradeoff between short and long intervals.](image)

- Better throughput → better latency
- Short interval to long interval
Selecting a Batching Interval

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

![Graph showing latency and throughput tradeoff](image)

- Short interval: better latency
- Long interval: better throughput

Load (requests/s) vs. Latency
Selecting a Batching Interval

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

![Graph showing tradeoff between latency and throughput with varying interval lengths](image)
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

```
<table>
<thead>
<tr>
<th>Load (requests/s)</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>short interval</td>
<td>better latency → better throughput</td>
</tr>
<tr>
<td>long</td>
<td></td>
</tr>
</tbody>
</table>
```
Solution: Stout

- Stout is a storage interposition library
- Our contribution is a technique for independently adjusting the batching interval
1. Introduction
2. Application Structure
3. Adaptive Batching
4. Evaluation
Overlapped Request Processing

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

BatchingLoop:
  keys = DirtyKeys()
  replies = Depends(keys)
  AsyncWrite(keys, replies)
  Sleep(interval)
Overlapped Request Processing

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

batching_loop:
  keys = DirtyKeys()
  replies = Depends(keys)
  AsyncWrite(keys, replies)
  Sleep(interval)
Overlapped Request Processing

ProcessRequest(req):

www → app → store
Overlapped Request Processing

ProcessRequest(req):
key = Parse(req)
Overlapped Request Processing

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

www → app → store
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 uncommitted state
Staying Safe: Consistency

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

Synchronous

![Diagram showing synchronous state update]

Potential Async

![Diagram showing potential async state update during failure]
Staying Safe: Consistency

- Don’t reveal uncommitted state
- Potential async: Inconsistency on failure
Staying Safe: Consistency

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

Synchronous

Potential Async
Staying Safe: Consistency

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

**Synchronous**

- app
  - x=5

**Potential Async**

- app
  - x=5
- store
  - interval
Staying Safe: Consistency

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

Diagram:

Synchronous:
- App
- Store
  - x = 5

Potential Async:
- App
- Store
  - x = 5
  - Interval
Staying Safe: Consistency

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

![Diagram showing synchronous and potential async scenarios]
Staying Safe: Consistency

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

```
Synchronous

app
store

x=5

Potential Async

app
store

x=5
```
Staying Safe: Consistency

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

Synchronous

<table>
<thead>
<tr>
<th>app</th>
<th>store</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=5</td>
<td></td>
</tr>
</tbody>
</table>

Potential Async

<table>
<thead>
<tr>
<th>app</th>
<th>store</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=5</td>
<td></td>
</tr>
</tbody>
</table>

Failure
Staying Safe: Consistency

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

Synchronous

![Synchronous diagram]

Stout Async

![Stout Async diagram]
Staying Safe: Consistency

- Don’t reveal uncommitted state
- Potential async: Inconsistency on failure
- Stout provides serialized update semantics
Staying Safe: Consistency

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

**Synchronous**

```
app
x=5
```

```
store
```

**Stout Async**

```
app
x=5
```

```
store
```

interval
Staying Safe: Consistency

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

Synchronous

- app
  - x=5
- store

Stout Async

- app
  - x=5
- store
  - interval
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

$\text{x=5}$
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
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
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

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Networking</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCELERATE</td>
<td>Change Rate</td>
<td>Change Size</td>
</tr>
<tr>
<td>BACK-OFF</td>
<td>Send Faster</td>
<td>Batch Less</td>
</tr>
<tr>
<td></td>
<td>Send Slower</td>
<td>Batch More</td>
</tr>
</tbody>
</table>
Algorithm

\[ \text{if } \text{perf} < \text{recent perf} \]
\[ \text{BACK-OFF} \]
\[ \text{else} \]
\[ \text{ACCELERATE} \]
Algorithm: Estimating Storage Performance

if \( \text{perf} < \text{recent perf} \)

BACK-OFF

else

ACCELERATE

\[
\text{batch size} = \frac{\text{latency} + \text{interval}}{16}
\]
Algorithm: Estimating Storage Capacity

if \( \text{perf} < \text{recent perf} \)
   BACK-OFF
else
   ACCELERATE

if backed-off
   \[
   \frac{\text{EWMA}(\text{batch size}_i)}{\text{EWMA}(	ext{lat}_i) + \text{EWMA}(\text{interval}_i)}
   \]
else // accelerated
   \[
   \text{MAX}_i\left(\frac{\text{batch size}_i}{\text{lat}_i + \text{interval}_i}\right)
   \]
Algorithm: Achieving Fair Share

if perf < recent perf
    BACK-OFF
else
    ACCELERATE
Algorithm: Achieving Fair Share

\[ \text{if } \text{perf} < \text{recent perf} \]
\[ \text{BACK-OFF} \rightarrow (1 + \alpha) \times \text{interval}_i \]
\[ \text{else} \]
\[ \text{ACCELERATE} \]
Algorithm: Achieving Fair Share

\[
\text{if perf} \ < \ \text{recent perf} \\
\text{BACK-OFF} \rightarrow (1 + \alpha) \ast \text{interval}_i \\
\text{else} \\
\text{ACCELERATE} \rightarrow (1 - \beta) \ast \text{interval}_i + \beta \ast \sqrt{\text{interval}_i}
\]
Algorithm: Achieving Fair Share

\[
\text{if } \text{perf} < \text{recent perf} \\
\quad \text{BACK-OFF } \rightarrow (1 + \alpha) \times \text{interval}_i \\
\text{else} \\
\quad \text{ACCELERATE } \rightarrow (1 - \beta) \times \text{interval}_i + \beta \times \sqrt{\text{interval}_i}
\]
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

Our Workload

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

Sectioned Document Store

Live Mesh
Evaluation

Sectioned Document Store

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

![Graph showing the comparison between no-batching and batching in terms of end-to-end latency for different load levels. The graph indicates that batching significantly improves performance.]
Baseline: Importance of Batching

Batching improves performance.
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

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

![Graph showing end-to-end latency (ms) vs. load (requests/s). The graph indicates that at 20ms, Stout outperforms fixed intervals across various load levels.](image-url)
Evaluation: Stout vs. Fixed Intervals

The diagram illustrates the comparison between Stout and fixed intervals across a wide range of workloads. It shows that Stout consistently outperforms fixed intervals, with end-to-end latency ranging from 20 ms to 40 ms, whereas fixed intervals exhibit higher latency.

Key observations:
- **20ms**: Stout's latency remains stable across the load range.
- **40ms**: Fixed intervals show higher latency, particularly at higher loads.

Overall, the evaluation suggests that Stout is a better choice for performance under various workloads compared to fixed intervals.
Evaluation: Stout vs. Fixed Intervals

Stout better than any fixed interval across a wide range of workloads.
Evaluation: Stout vs. Fixed Intervals

Stout better than any fixed interval across wide range of workloads.
Evaluation: Stout vs. Fixed Intervals

- Stout better than any fixed interval across wide range of workloads
Evaluation: Workload Variation

Decrease 12k requests/s $\rightarrow$ 8k requests/s

Increase 12k requests/s $\rightarrow$ 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?