CacheDOCS: A Dynamic Key-Value Object Caching Service

ICDCS-PED 2017

Julien Gascon-Samson, Michael Coppinger, Fan Jin, Jörg Kienzle, Bettina Kemme

Post-Doctoral Fellow @ Univerity of British Columbia
Current Advisor: Dr Karthik Pattabiraman
Department of Electrical and Computer Engineering
Vancouver, Canada

Work completed under the advisement of:
Dr Jörg Kienzle and Dr Bettina Kemme
School of Computer Science, McGill University
Montreal, Canada

Electrical and Computer Engineering

Monday June 5th, 2017

Key-Value Store

Key-Value Store

Stores pairs of $\{k, v\}$:

k_1	v_1
k_2	v_2
k_3	v_3
k_4	v_4
k_5	v_5

Key-Value Store

Key-Value Store

Stores pairs of $\{k, v\}$:

k_1	v_1
k_2	v_2
k_3	v_3
k_4	v_4
k_5	v_5

Operations:

- get(k): obtain value v for key k

Key-Value Store

Key-Value Store

Stores pairs of $\{k, v\}$:

k_1	v_1
k_2	v_2
k_3	v_3
k_4	v_4
k_5	v_5

Operations:

- $\text{get}(k)$: obtain value v for key k
- $\text{put}(k, v)$: put pair $\{k, v\}$

Key-Value Store

Key-Value Store

Stores pairs of $\{k, v\}$:

k_1	v_1
k_2	v_2
k_3	v_3
k_4	v_4
k_5	v_5

Operations:

- get(k): obtain value v for key k
- put(k, v): put pair $\{k, v\}$

Example - Database Context

```
String sql =
     "SELECT␣*␣FROM␣DATA";
if (cache.contains(sql)) {
     return cache.get(sql);
}
Object result = db.query(sql);
cache.put(sql, result);
return result;
```

Key-Value Store

Key-Value Store

Stores pairs of $\{k, v\}$:

k_1	v_1
k_2	v_2
k_3	v_3
k_4	v_4
k_5	v_5

Operations:

- get(k): obtain value v for key k
- put(k, v): put pair $\{k, v\}$

Example - Database Context

```
String sql =
     "SELECT␣*␣FROM␣DATA";
if (cache.contains(sql)) {
     return cache.get(sql);
}
Object result = db.query(sql);
cache.put(sql, result);
return result;
```

- Caching can be deployed as part of a system
- Or offered as a service in the cloud

Limitations of key-value stores

- Storage of “static” pairs of key-value

Limitations of key-value stores

- Storage of “static” pairs of key-value
- Pull-based interface

Limitations of key-value stores

- Storage of “static” pairs of key-value
- Pull-based interface

Storing objects in a key-value store:

Player.java

```
public class Player {
    String name;
    int x;
    int y;
}
```

Limitations of key-value stores

- Storage of “static” pairs of key-value
- Pull-based interface

Storing objects in a key-value store:

Player.java

```
public class Player {
    String name;
    int x;
    int y;
}
```

- Assuming player P_1, we must serialize the object to store it
    - put("*Julien*", *serialize*(P_1))

Limitations of key-value stores

- Storage of “static” pairs of key-value
- Pull-based interface

Storing objects in a key-value store:

Player.java

```
public class Player {
    String name;
    int x;
    int y;
}
```

- Assuming player P_1, we must serialize the object to store it
  - put(" *Julien*", *serialize*(P_1))
- Upon P_1 changing (i.e., moving), then we must reserialize P_1:
  - put(" *Julien*", *serialize*(P_1))

Retrieving an Object

Player.java

```java
public class Player {
    String name;
    int x;
    int y;
}
```

Retrieving an Object

Player.java

```
public class Player {
    String name;
    int x;
    int y;
}
```

Assuming a game with many players, which are cached:

Retrieving an Object

Player.java

```
public class Player {
    String name;
    int x;
    int y;
}
```

Assuming a game with many players, which are cached:

Retrieving a cached player

```
Player p;
while(true) {
    p = unserialize(cache.get("Julien"));
    sleep(100);
}
```

Complex Object

Designing a Collaborative Drawing Application

```
public class Drawing {
    Color[][] pixels;

    void drawLine(int x1, int y1, int x2, int y2, Color color);
    void drawRectangle(int x1, int y1,
        int x2, int y2, Color color);
    void fill(int color);
    void getPixel(int x, int y) { return pixels(x,y); };
} //...
```

CacheDOCS

CacheDOCS - Object Caching as a Service:

Large-Scale Games

- Many players & in-game objects
- Offloading the centralized game infrastructure
- Increasing scalability

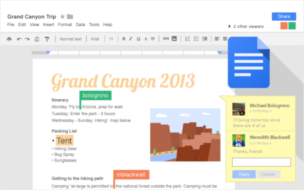
CacheDOCS

CacheDOCS - Object Caching as a Service:

Large-Scale Games

- Many players & in-game objects
- Offloading the centralized game infrastructure
- Increasing scalability

Collaborative Document Editing



Grand Canyon 2013

Itinerary
Monday: Fly to Arizona, prep for walk
Tuesday: Enter the park - 3 hours
Wednesday - Sunday: Hiking! map below

Packing List
- Tent
- Hiking Gear
- Bug Spray
- Sunglasses

Getting to the hiking path
Camping 'at-large is permitted in the national forest outside the park. Camping must be

Michael Bolognino
5:19 PM May 9
I'll bring mine too since there are 4 of us

Meredith Blackwell
5:21 PM May 9
Thanks, friend!

- Hosting many live documents (several millions)
- Many users can collaborate on the same document
- Increasing scalability

High-Level Architecture

Object Caching as a Service

CacheDOCS API

- get(k): obtain value v for key k
- put(k, v): put pair $\{k, v\}$

Object Caching as a Service

CacheDOCS API

- `get`(k): obtain value v for key k
- `put`(k, v): put pair $\{k, v\}$
- `getAddSubscribe`(k):
    - obtain value v for key k,
    - subscribe to be notified of changes to v: pub/sub

Object Caching as a Service

CacheDOCS API

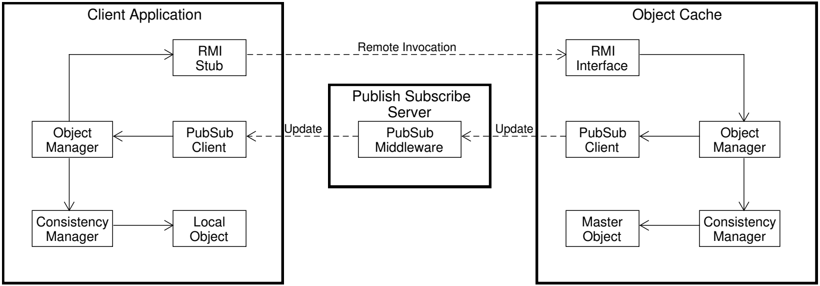
- get(k): obtain value v for key k
- put(k, v): put pair $\{k, v\}$
- `getAddSubscribe`(k):
    - obtain value v for key k,
    - subscribe to be notified of changes to v: pub/sub

Publish/Subscribe (Topic-Based)

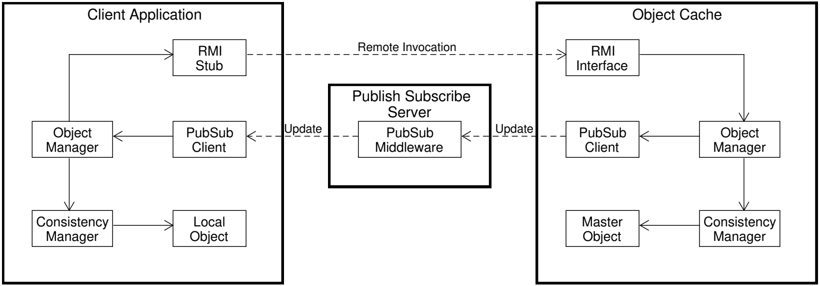
Use pub/sub to disseminate updates of cached objects to interested clients:

- `getAddSubscribe`(k) ⇒ `subscribe`(k)
- v is modified ⇒ `publish`(k, "v")

Architecture



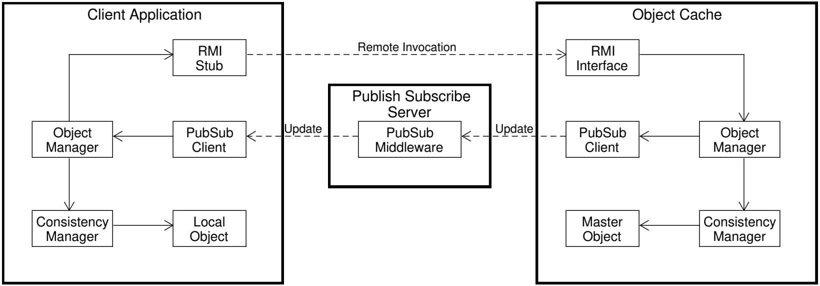
Architecture



Object Cache

- Caches Java Objects ($\{k, v_r\}$ pairs)
- Holds the *masters* (v_r)
- Exposed through RMI
- Pub/Sub Interface
    - Push changes to objects

Architecture



Client Application

- `get`(k), `put`(k, v), `getAddSubscribe`(k)
- Maintains local copies v_l of cached objects
- Synchronizes changes with master objects (v_r)

Object Cache

- Caches Java Objects ($\{k, v_r\}$ pairs)
- Holds the *masters* (v_r)
- Exposed through RMI
- Pub/Sub Interface
  - Push changes to objects

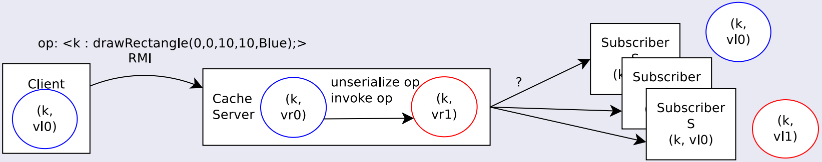
Update Propagation

1. Motivation
2. High-Level Architecture
3. Update Propagation
4. Implementation & Evaluation
5. Conclusion

Propagating Updates

Method invocation sent to master.
Upon the state of a cached object $\{k, v_r\}$ changing:

Update Propagation



op: <k : drawRectangle(0,0,10,10,Blue);>
RMI

Client
(k, vl0)

Cache Server
(k, vr0)
unserialize op
invoke op
(k, vr1)

?

Subscriber S (k
Subscriber S
Subscriber S (k, vl0)

(k, vl0)

(k, vl1)

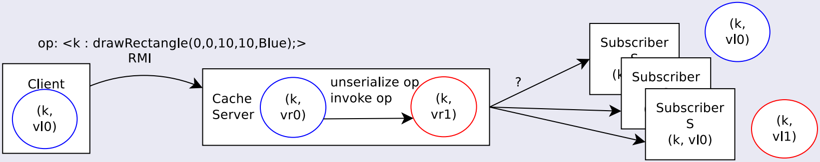
Propagating Updates

Method invocation sent to master.

Upon the state of a cached object $\{k, v_r\}$ changing:

- Updates are propagated to all subscribers $S \in \mathbb{S}$

Update Propagation



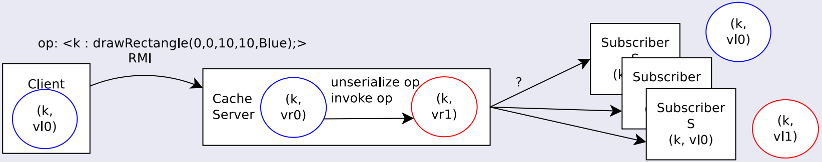
Propagating Updates

Method invocation sent to master.

Upon the state of a cached object $\{k, v_r\}$ changing:

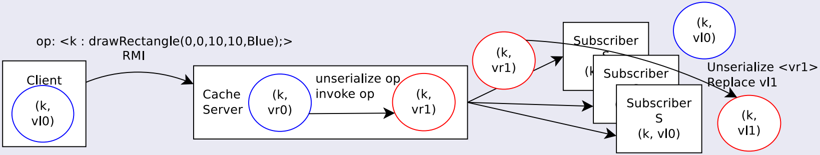
- Updates are propagated to all subscribers $S \in \mathbb{S}$
- Four different propagation strategies

Update Propagation



1- Serialized Update Strategy

Principle - Serialized Update Strategy



op: <k : drawRectangle(0,0,10,10,Blue);>
RMI

Client
(k, vl0)

Cache Server
(k, vr0)
unserialize op
invoke op
(k, vr1)

(k, vr1)

Subscriber
Subscriber
Subscriber
S
(k, vl0)

(k, vl0)

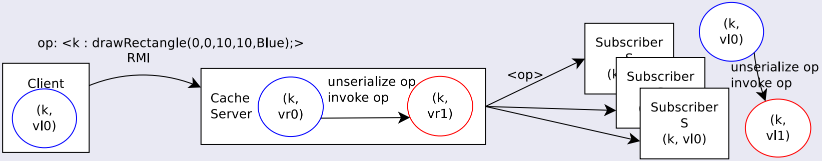
Unserialize <vr1>
Replace vl1

(k, vl1)

1. v_{r1} serialized and sent to all $S \in \mathbb{S}$
2. All S deserialize v_{r1}, and replace v_{l0} by v_{r1} in the local cache

2- Operation Update Strategy

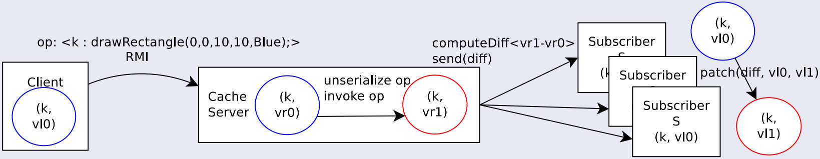
Principle - Operation Update Strategy



1. Operation serialized and sent to all $S \in \mathbb{S}$
2. All S unserialize and execute the operation on v_{l0} to obtain v_{l1}

3- Binary Diff Update Strategy

Principle - Binary Diff Update Strategy



op: <k : drawRectangle(0,0,10,10,Blue);>
RMI

Client
(k, vl0)

Cache Server
(k, vr0)
unserialize op
invoke op
(k, vr1)

computeDiff<vr1-vr0>
send(diff)

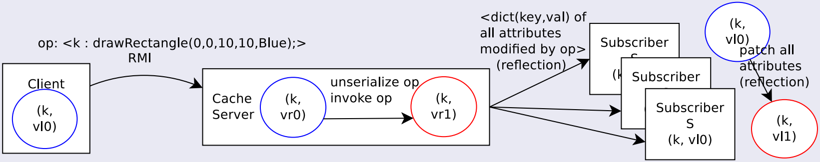
Subscriber
Subscriber
Subscriber
S
(k, vl0)

(k, vl0)
patch(diff, vl0, vl1)
(k, vl1)

1. Cache serializes v_{r0} and v_{r1} computes the binary diff $v_{r0} \rightarrow v_{r1}$
2. Binary diff sent to all $S \in \mathbb{S}$
3. All S patch the serialized v_{l0} with the diff, and deserialize to obtain v_{r1}

4- Attribute Patch Update Strategy

Principle - Attribute Patch Update Strategy



1. Cache sends dict of modified attributes ($v_{r0} \rightarrow v_{r1}$) to $S \in \mathbb{S}$
2. All S patch v_{l0} with the new attributes to obtain v_{l1}

Handling Concurrency

- S_1: Local object at state v_{l0}
- S_2: Local object at state v_{l0}
- Cache: Object at state v_{r0}

Handling Concurrency

- S_1: Local object at state v_{l0}
- S_2: Local object at state v_{l0}
- Cache: Object at state v_{r0}

Incorrect Behavior

- S_1 performs an operation: $v_{r0} \rightarrow v_{r1}$

Handling Concurrency

- S_1: Local object at state v_{l0}
- S_2: Local object at state v_{l0}
- Cache: Object at state v_{r0}

Incorrect Behavior

- S_1 performs an operation: $v_{r0} \rightarrow v_{r1}$
- S_2 performs an operation: $v_{r1} \rightarrow v_{r2}$

Handling Concurrency

- S_1: Local object at state v_{l0}
- S_2: Local object at state v_{l0}
- Cache: Object at state v_{r0}

Incorrect Behavior

- S_1 performs an operation: $v_{r0} \rightarrow v_{r1}$
- S_2 performs an operation: $v_{r1} \rightarrow v_{r2}$
  - Incorrect as S_2 assumes state v_{r0}

Handling Concurrency

- S_1: Local object at state v_{l0}
- S_2: Local object at state v_{l0}
- Cache: Object at state v_{r0}

Incorrect Behavior

- S_1 performs an operation: $v_{r0} \rightarrow v_{r1}$
- S_2 performs an operation: $v_{r1} \rightarrow v_{r2}$
  - Incorrect as S_2 assumes state v_{r0}

Correct Behavior

- S_1 performs an operation, sends “version” 0: $v_{r0} \rightarrow v_{r1}$

Handling Concurrency

- S_1: Local object at state v_{l0}
- S_2: Local object at state v_{l0}
- Cache: Object at state v_{r0}

Incorrect Behavior

- S_1 performs an operation: $v_{r0} \rightarrow v_{r1}$
- S_2 performs an operation: $v_{r1} \rightarrow v_{r2}$
    - Incorrect as S_2 assumes state v_{r0}

Correct Behavior

- S_1 performs an operation, sends “version” 0: $v_{r0} \rightarrow v_{r1}$
- S_2 performs an operation, sends “version” 0:

Handling Concurrency

- S_1: Local object at state v_{l0}
- S_2: Local object at state v_{l0}
- Cache: Object at state v_{r0}

Incorrect Behavior

- S_1 performs an operation: $v_{r0} \rightarrow v_{r1}$
- S_2 performs an operation: $v_{r1} \rightarrow v_{r2}$
  - Incorrect as S_2 assumes state v_{r0}

Correct Behavior

- S_1 performs an operation, sends "version" 0: $v_{r0} \rightarrow v_{r1}$
- S_2 performs an operation, sends "version" 0:
  - Rejected as cached version is now 1 (v_{r1})

Handling Concurrency

- S_1: Local object at state v_{l0}
- S_2: Local object at state v_{l0}
- Cache: Object at state v_{r0}

Incorrect Behavior

- S_1 performs an operation: $v_{r0} \rightarrow v_{r1}$
- S_2 performs an operation: $v_{r1} \rightarrow v_{r2}$
  - Incorrect as S_2 assumes state v_{r0}

Correct Behavior

- S_1 performs an operation, sends "version" 0: $v_{r0} \rightarrow v_{r1}$
- S_2 performs an operation, sends "version" 0:
  - Rejected as cached version is now 1 (v_{r1})
- S_2 receives update and patches to version 1 (v_{l1})

Handling Concurrency

- S_1: Local object at state v_{l0}
- S_2: Local object at state v_{l0}
- Cache: Object at state v_{r0}

Incorrect Behavior

- S_1 performs an operation: $v_{r0} \rightarrow v_{r1}$
- S_2 performs an operation: $v_{r1} \rightarrow v_{r2}$
  - Incorrect as S_2 assumes state v_{r0}

Correct Behavior

- S_1 performs an operation, sends "version" 0: $v_{r0} \rightarrow v_{r1}$
- S_2 performs an operation, sends "version" 0:
  - Rejected as cached version is now 1 (v_{r1})
- S_2 receives update and patches to version 1 (v_{l1})
- S_2 can retry the operation

Implementation & Evaluation

Implementation

- Implementation in Java
- Supports the caching of Java objects
- RMI-based API and client-side library
- Dynamoth Pub/Sub service over Redis for update propagation

Experimental Setup

- All experiments run over McGill School of Computer Science labs

Experimental Setup

- All experiments run over McGill School of Computer Science labs
- One machine for the caching server, one machine for the client

Experimental Setup

- All experiments run over McGill School of Computer Science labs
- One machine for the caching server, one machine for the client
- Targeted experiments that model 3 specific use-cases of CacheDOCS

Experimental Setup

- All experiments run over McGill School of Computer Science labs
- One machine for the caching server, one machine for the client
- Targeted experiments that model 3 specific use-cases of CacheDOCS
  1. Collaborative Graphics Editing

Experimental Setup

- All experiments run over McGill School of Computer Science labs
- One machine for the caching server, one machine for the client
- Targeted experiments that model 3 specific use-cases of CacheDOCS
    1. Collaborative Graphics Editing
    2. Maze Multiplayer Game Pathfinding

Experimental Setup

- All experiments run over McGill School of Computer Science labs
- One machine for the caching server, one machine for the client
- Targeted experiments that model 3 specific use-cases of CacheDOCS
    1. Collaborative Graphics Editing
    2. Maze Multiplayer Game Pathfinding
    3. Collaborative Spreadsheet Editing

Experimental Setup

- All experiments run over McGill School of Computer Science labs
- One machine for the caching server, one machine for the client
- Targeted experiments that model 3 specific use-cases of CacheDOCS
    1. Collaborative Graphics Editing
    2. Maze Multiplayer Game Pathfinding
    3. Collaborative Spreadsheet Editing

    - Compare the different propagation strategies

Experimental Setup

- All experiments run over McGill School of Computer Science labs
- One machine for the caching server, one machine for the client
- Targeted experiments that model 3 specific use-cases of CacheDOCS
  1. Collaborative Graphics Editing
  2. Maze Multiplayer Game Pathfinding
  3. Collaborative Spreadsheet Editing

  - Compare the different propagation strategies

Future Work

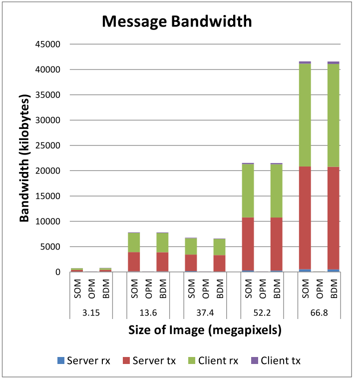
Global performance & scalability of CacheDOCS with many clients in the cloud

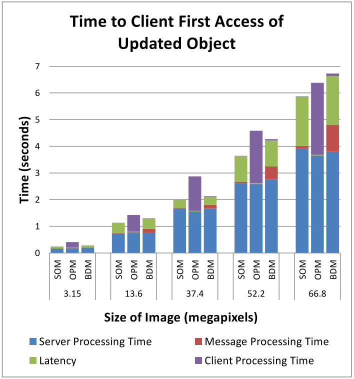
Experiment 1 - Collaborative Graphics Editing

Cache: stores *drawings* | Client: performs a “rotate” operation

Experiment 1 - Collaborative Graphics Editing

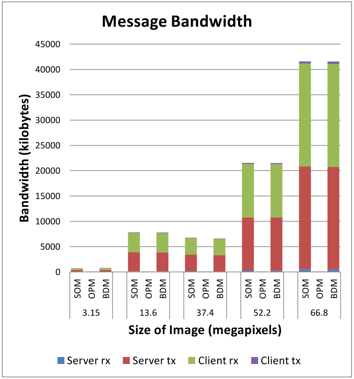
Cache: stores *drawings* | Client: performs a "rotate" operation

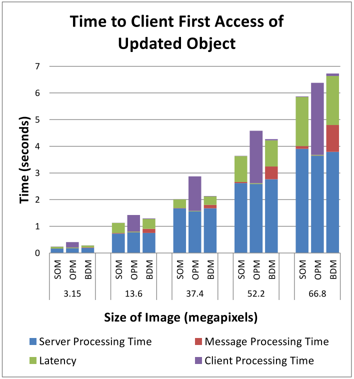




Experiment 1 - Collaborative Graphics Editing

Cache: stores *drawings* | Client: performs a “rotate” operation

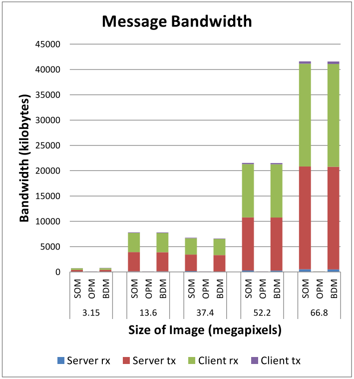


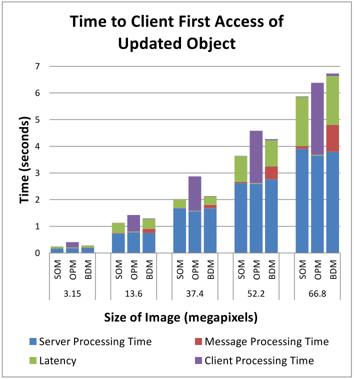


1) Sending Serialized Object High bandwidth

Experiment 1 - Collaborative Graphics Editing

Cache: stores *drawings* | Client: performs a "rotate" operation



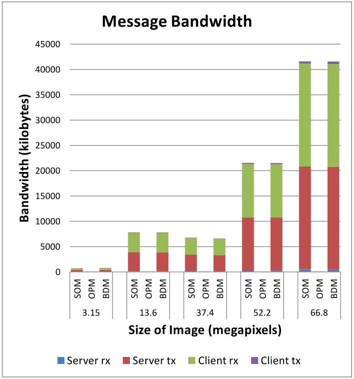


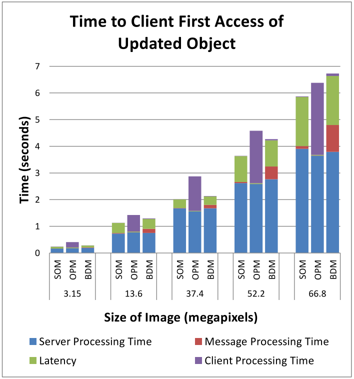
1) Sending Serialized Object High bandwidth

2) Sending Operation <u>Low bandwidth, slightly higher time</u>

Experiment 1 - Collaborative Graphics Editing

Cache: stores *drawings* | Client: performs a “rotate” operation





1) Sending Serialized Object High bandwidth

2) Sending Operation Low bandwidth, slightly higher time

3) Sending Binary Diff High bandwidth (rotation affects all pixels)

Conclusion

Conclusion and Future Work

Contributions

- Caching of full dynamic (Java) objects
- Invokation of remote operations over local copies of cached objects
- Forwarded to the master cached copy
- Push-based dissemination of updates: several strategies
- Consistency management
- Experiments with 3 different use cases

Conclusion and Future Work

Contributions

- Caching of full dynamic (Java) objects
- Invokation of remote operations over local copies of cached objects
- Forwarded to the master cached copy
- Push-based dissemination of updates: several strategies
- Consistency management
- Experiments with 3 different use cases

Future Work

- Better support for object nesting & inter-object links
- Testing & optimizing CacheDOCS in the cloud (large-scale)
- Dynamic selection of best propagation strategy

Invoking Methods

Invoked over synchronized copy v_l of cached object v_r with key k.

Types of methods

Invoking Methods

Invoked over synchronized copy v_l of cached object v_r with key k.

Types of methods

1. A- Deterministic, read-only methods (i.e., getter)
   - Operation executed locally

Invoking Methods

Invoked over synchronized copy v_l of cached object v_r with key k.

Types of methods

1. A- Deterministic, read-only methods (i.e., getter)
    - Operation executed locally
2. B- Non-deterministic methods or state-altering
    - The "method call" (signature + params) is serialized, sent to the cache and executed

Invoking Methods

Invoked over synchronized copy v_l of cached object v_r with key k.

Types of methods

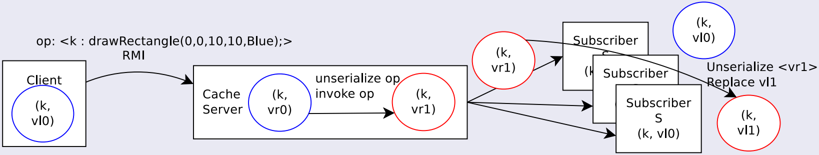
1. A- Deterministic, read-only methods (i.e., getter)
    - Operation executed locally
2. B- Non-deterministic methods or state-altering
    - The "method call" (signature + params) is serialized, sent to the cache and executed

Drawing Class Example

```
void drawLine(int x1, int y1, int x2, int y2, Color color); // Type B
void drawRectangle(int x1, int y1,
    int x2, int y2, Color color); // Type B
void fill(int color); // Type B
void getPixel(int x, int y) { return pixels(x,y); }; // Type A
```

1- Serialized Update Strategy

Principle - Serialized Update Strategy



op: <k : drawRectangle(0,0,10,10,Blue);>
RMI

Client
(k, vl0)

Cache Server
(k, vr0)
unserialize op
invoke op
(k, vr1)

(k, vr1)

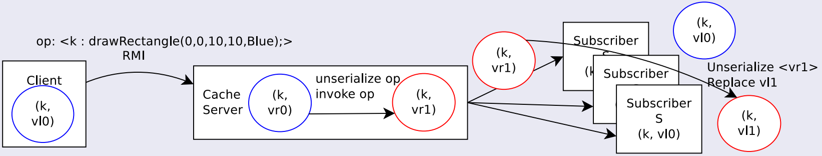
Subscriber
Subscriber
Subscriber S
(k, vl0)

(k, vl0)
Unserialize <vr1>
Replace vl1
(k, vl1)

1. v_{r1} serialized and sent to all $S \in \mathbb{S}$
2. All S deserialize v_{r1}, and replace v_{l0} by v_{r1} in the local cache

1- Serialized Update Strategy

Principle - Serialized Update Strategy



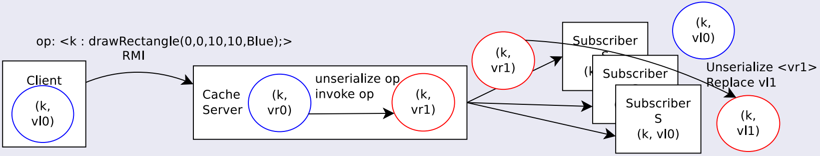
1. v_{r1} serialized and sent to all $S \in \mathbb{S}$
2. All S deserialize v_{r1}, and replace v_{l0} by v_{r1} in the local cache

Pros

- Executed only once
  - CPU Intensive operations
- For small objects:
  - Low propagation time
  - Low bandwidth

1- Serialized Update Strategy

Principle - Serialized Update Strategy



1. v_{r1} serialized and sent to all $S \in \mathbb{S}$
2. All S deserialize v_{r1}, and replace v_{l0} by v_{r1} in the local cache

Pros

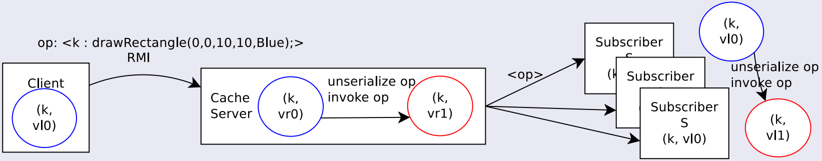
- Executed only once
  - CPU Intensive operations
- For small objects:
  - Low propagation time
  - Low bandwidth

Cons

- Large objects:
  - Long serialization time
  - Long propagation time
  - High bandwidth

2- Operation Update Strategy

Principle - Operation Update Strategy



op: <k : drawRectangle(0,0,10,10,Blue);>
RMI

Client
(k, vl0)

Cache Server
(k, vr0)
unserialize op
invoke op
(k, vr1)

<op>

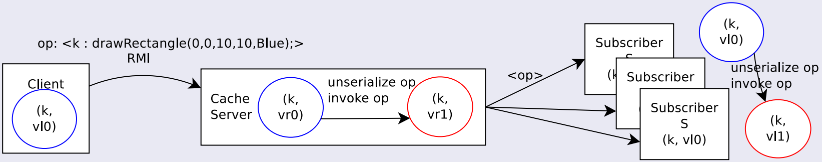
Subscriber
Subscriber
Subscriber S
(k, vl0)

(k, vl0)
unserialize op
invoke op
(k, vl1)

1. Operation serialized and sent to all $S \in \mathbb{S}$
2. All S unserialize and execute the operation on v_{l0} to obtain v_{l1}

2- Operation Update Strategy

Principle - Operation Update Strategy



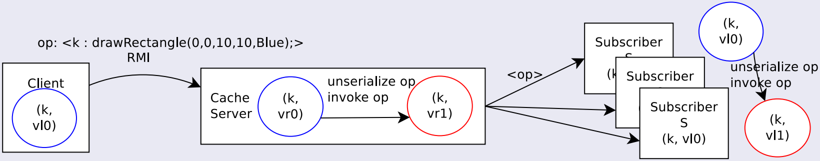
1. Operation serialized and sent to all $S \in \mathbb{S}$
2. All S unserialize and execute the operation on v_{l0} to obtain v_{l1}

Pros

- Great for large objects
  - They don't have to be serialized
  - Example: *Drawing* class

2- Operation Update Strategy

Principle - Operation Update Strategy



1. Operation serialized and sent to all $S \in \mathbb{S}$
2. All S unserialize and execute the operation on v_{l0} to obtain v_{l1}

Pros

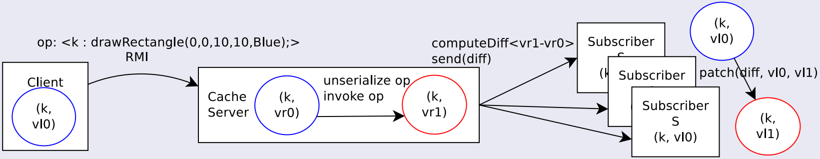
- Great for large objects
  - They don't have to be serialized
  - Example: *Drawing* class

Cons

- If the operation takes a long time to execute
  - Execution will occur twice (cache & client-side)

3- Binary Diff Update Strategy

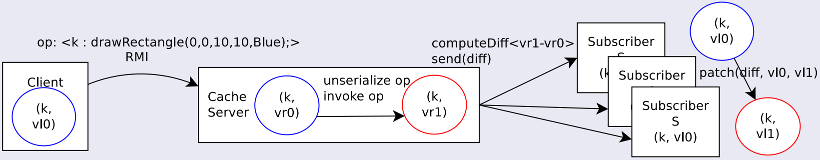
Principle - Binary Diff Update Strategy



1. Cache serializes v_{r0} and v_{r1} computes the binary diff $v_{r0} \rightarrow v_{r1}$
2. Binary diff sent to all $S \in \mathbb{S}$
3. All S patch the serialized v_{l0} with the diff, and deserialize to obtain v_{r1}

3- Binary Diff Update Strategy

Principle - Binary Diff Update Strategy



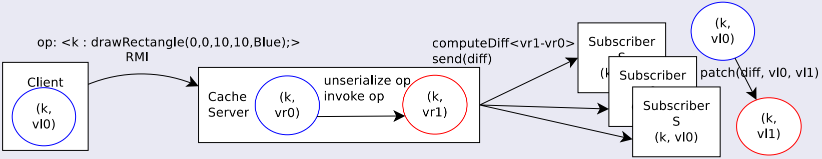
1. Cache serializes v_{r0} and v_{r1} computes the binary diff $v_{r0} \rightarrow v_{r1}$
2. Binary diff sent to all $S \in \mathbb{S}$
3. All S patch the serialized v_{l0} with the diff, and deserialize to obtain v_{r1}

Pros

- Large objects with CPU-intensive operations
    - Smaller “patch” size?

3- Binary Diff Update Strategy

Principle - Binary Diff Update Strategy


op: <k : drawRectangle(0,0,10,10,Blue);>
RMI
Client
(k, vl0)
Cache Server
(k, vr0)
unserialize op
invoke op
(k, vr1)
computeDiff<vr1-vr0>
send(diff)
Subscriber
Subscriber
Subscriber S
(k, vl0)
(k, vl0)
patch(diff, vl0, vl1)
(k, vl1)

1. Cache serializes v_{r0} and v_{r1} computes the binary diff $v_{r0} \rightarrow v_{r1}$
2. Binary diff sent to all $S \in \mathbb{S}$
3. All S patch the serialized v_{l0} with the diff, and deserialize to obtain v_{r1}

Pros

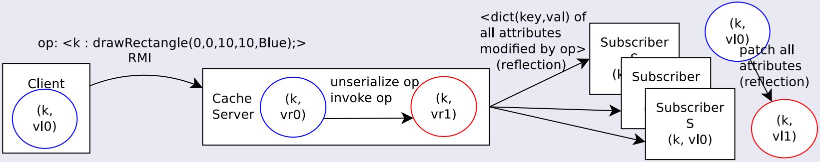
- Large objects with CPU-intensive operations
  - Smaller "patch" size?

Cons

- Can be CPU-costly and thus can take time

4- Attribute Patch Update Strategy

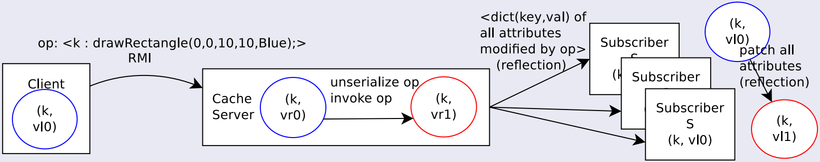
Principle - Attribute Patch Update Strategy



1. Cache sends dict of modified attributes ($v_{r0} \rightarrow v_{r1}$) to $S \in \mathbb{S}$
2. All S patch v_{l0} with the new attributes to obtain v_{l1}

4- Attribute Patch Update Strategy

Principle - Attribute Patch Update Strategy



op: <k : drawRectangle(0,0,10,10,Blue);>
RMI

Client
(k, vl0)

Cache Server
(k, vr0)
unserialize op
invoke op
(k, vr1)

<dict(key,val) of all attributes modified by op>
(reflection)

Subscriber
Subscriber
Subscriber S
(k, vl0)

(k, vl0)
patch all attributes
(reflection)
(k, vl1)

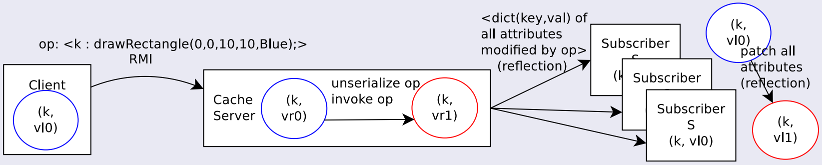
1. Cache sends dict of modified attributes ($v_{r0} \rightarrow v_{r1}$) to $S \in \mathbb{S}$
2. All S patch v_{l0} with the new attributes to obtain v_{l1}

Pros

- Large-objects and CPU-intensive operations
  - Serialization-deserialization + diff avoided
  - Only one execution

4- Attribute Patch Update Strategy

Principle - Attribute Patch Update Strategy



1. Cache sends dict of modified attributes ($v_{r0} \rightarrow v_{r1}$) to $S \in \mathbb{S}$
2. All S patch v_{l0} with the new attributes to obtain v_{l1}

Pros

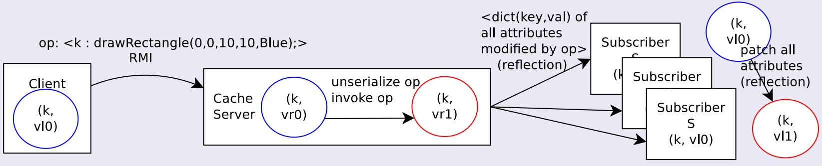
- Large-objects and CPU-intensive operations
  - Serialization-deserialization + diff avoided
  - Only one execution

Cons

- Requires developpers to tag the set of changed attributes for each method

4- Attribute Patch Update Strategy

Principle - Attribute Patch Update Strategy



op: <k : drawRectangle(0,0,10,10,Blue);>
RMI

Client
(k, vl0)

Cache Server
(k, vr0)
unserialize op
invoke op
(k, vr1)

<dict(key,val) of all attributes modified by op>
(reflection)

Subscriber
Subscriber
Subscriber S
(k, vl0)

(k, vl0)
patch all attributes
(reflection)
(k, vl1)

1. Cache sends dict of modified attributes ($v_{r0} \rightarrow v_{r1}$) to $S \in \mathbb{S}$
2. All S patch v_{l0} with the new attributes to obtain v_{l1}

Pros

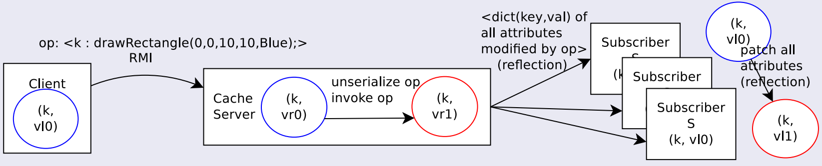
- Large-objects and CPU-intensive operations
  - Serialization-deserialization + diff avoided
  - Only one execution

Cons

- Requires developpers to tag the set of changed attributes for each method

4- Attribute Patch Update Strategy

Principle - Attribute Patch Update Strategy



1. Cache sends dict of modified attributes ($v_{r0} \rightarrow v_{r1}$) to $S \in \mathbb{S}$
2. All S patch v_{l0} with the new attributes to obtain v_{l1}

Pros

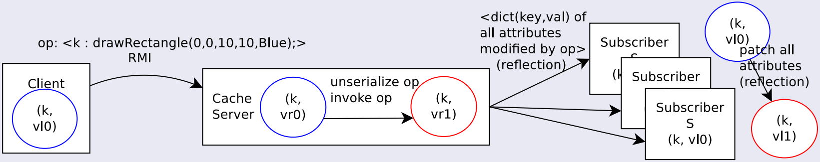
- Large-objects and CPU-intensive operations
  - Serialization-deserialization + diff avoided
  - Only one execution

Cons

- Requires developpers to tag the set of changed attributes for each method

4- Attribute Patch Update Strategy

Principle - Attribute Patch Update Strategy



op: <k : drawRectangle(0,0,10,10,Blue);>
RMI

Client
(k, vl0)

Cache Server
(k, vr0)
unserialize op
invoke op
(k, vr1)

<dict(key,val) of all attributes modified by op>
(reflection)

Subscriber
Subscriber
Subscriber S
(k, vl0)

(k, vl0)
patch all attributes (reflection)
(k, vl1)

1. Cache sends dict of modified attributes $(v_{r0} \rightarrow v_{r1})$ to $S \in \mathbb{S}$
2. All S patch v_{l0} with the new attributes to obtain v_{l1}

Pros

- Large-objects and CPU-intensive operations
  - Serialization-deserialization + diff avoided
  - Only one execution

Cons

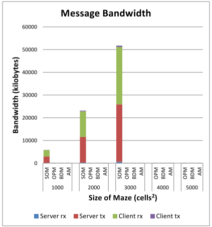
- Requires developpers to tag the set of changed attributes for each method

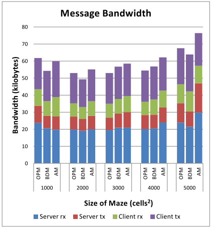
Experiment 2 - Maze Multiplayer Game Pathfinding

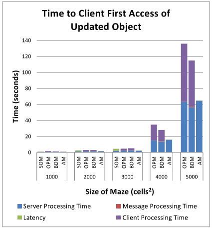
- Cache: stores *mazes* which contain cells and players
- Client: performs a “movePlayer” operation ⇒ pathfinding to check if move legal. If yes, player position changed. Attributes: {list of all players}.

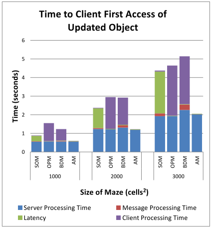
Experiment 2 - Maze Multiplayer Game Pathfinding

- Cache: stores *mazes* which contain cells and players
- Client: performs a "movePlayer" operation ⇒ pathfinding to check if move legal. If yes, player position changed. Attributes: {list of all players}.


Message Bandwidth

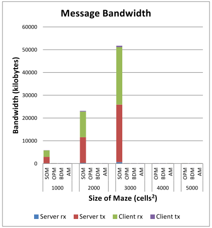

Message Bandwidth

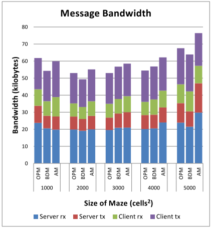

Time to Client First Access of Updated Object

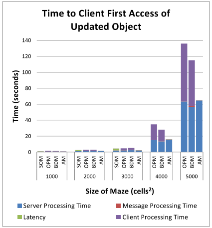

Time to Client First Access of Updated Object

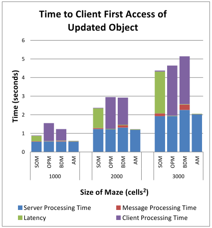
Experiment 2 - Maze Multiplayer Game Pathfinding

- Cache: stores *mazes* which contain cells and players
- Client: performs a "movePlayer" operation ⇒ pathfinding to check if move legal. If yes, player position changed. Attributes: {list of all players}.





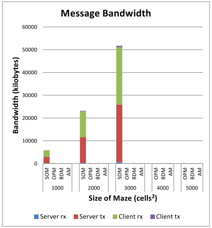


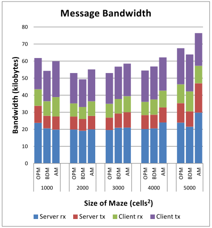


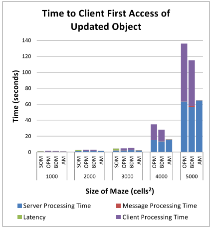
1) Sending Serialized Object Very high bandwidth (n^2 #cells)

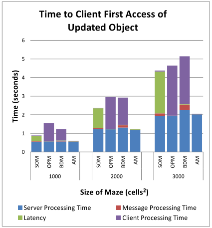
Experiment 2 - Maze Multiplayer Game Pathfinding

- Cache: stores *mazes* which contain cells and players
- Client: performs a “movePlayer” operation ⇒ pathfinding to check if move legal. If yes, player position changed. Attributes: {list of all players}.







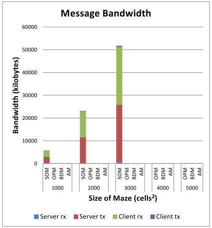


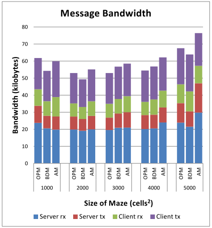
1) Sending Serialized Object Very high bandwidth (n^2 #cells)

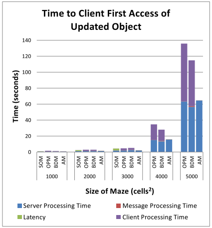
2) Sending Operation Very low bandwidth, high execution time

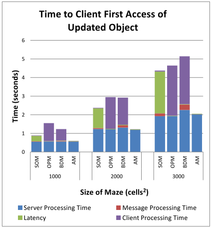
Experiment 2 - Maze Multiplayer Game Pathfinding

- Cache: stores *mazes* which contain cells and players
- Client: performs a "movePlayer" operation ⇒ pathfinding to check if move legal. If yes, player position changed. Attributes: {list of all players}.









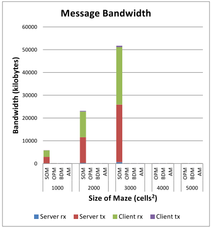
1) Sending Serialized Object Very high bandwidth (n^2 #cells)

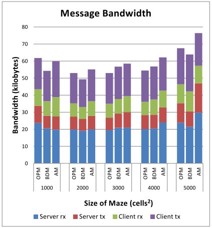
2) Sending Operation Very low bandwidth, high execution time

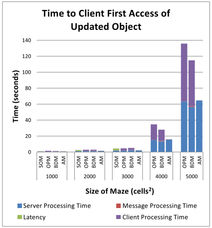
3) Sending Binary Diff Very low bandwidth, high execution time

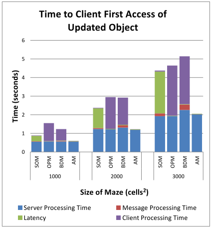
Experiment 2 - Maze Multiplayer Game Pathfinding

- Cache: stores *mazes* which contain cells and players
- Client: performs a "movePlayer" operation ⇒ pathfinding to check if move legal. If yes, player position changed. Attributes: {list of all players}.









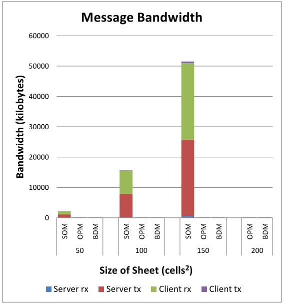
1) Sending Serialized Object Very high bandwidth (n^2 #cells)
2) Sending Operation Very low bandwidth, high execution time
3) Sending Binary Diff Very low bandwidth, high execution time
4) Sending Attributes Very low bandwidth, low execution time

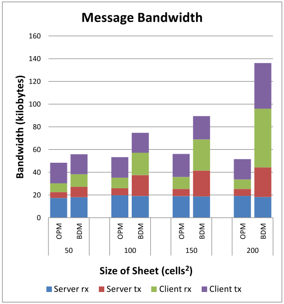
Experiment 3 - Collaborative Spreadsheet Editing

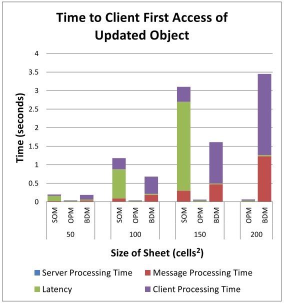
- Cache: stores *spreadsheets*: list of cells with values or formulas.
- Client: alter the value of a cell, which might impact the values of other cells (topological sort).

Experiment 3 - Collaborative Spreadsheet Editing

- Cache: stores *spreadsheets*: list of cells with values or formulas.
- Client: alter the value of a cell, which might impact the values of other cells (topological sort).

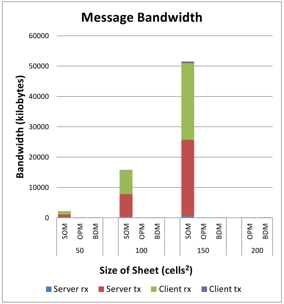


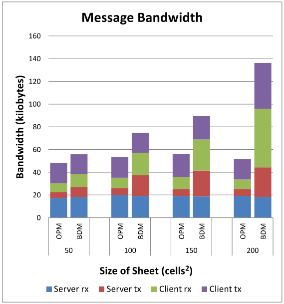


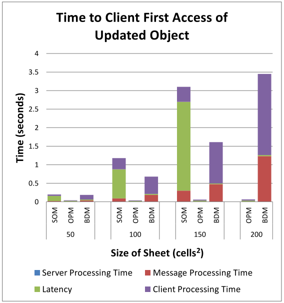


Experiment 3 - Collaborative Spreadsheet Editing

- Cache: stores *spreadsheets*: list of cells with values or formulas.
- Client: alter the value of a cell, which might impact the values of other cells (topological sort).



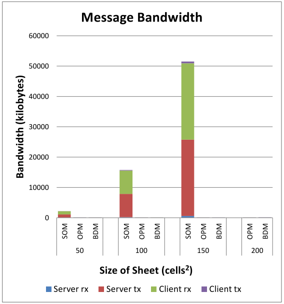


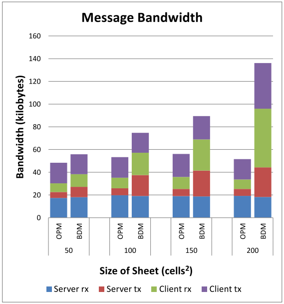


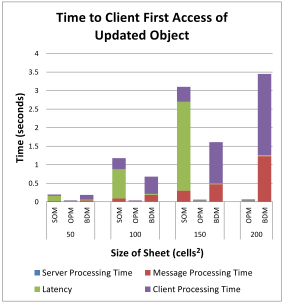
1) Sending Serialized Object Huge bandwidth (n^2) and time

Experiment 3 - Collaborative Spreadsheet Editing

- Cache: stores *spreadsheets*: list of cells with values or formulas.
- Client: alter the value of a cell, which might impact the values of other cells (topological sort).





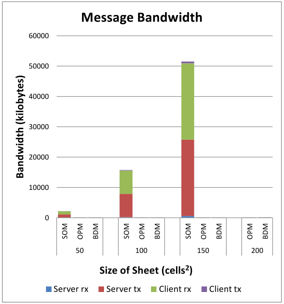


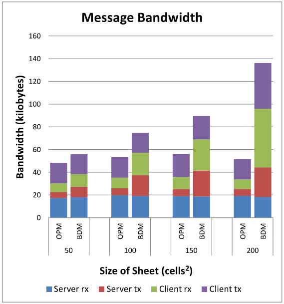
1) Sending Serialized Object Huge bandwidth (n^2) and time

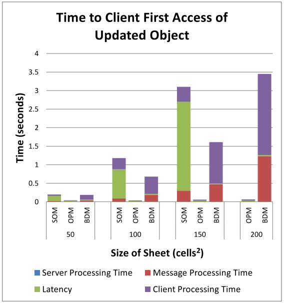
2) Sending Operation Low bandwidth, very low execution time

Experiment 3 - Collaborative Spreadsheet Editing

- Cache: stores *spreadsheets*: list of cells with values or formulas.
- Client: alter the value of a cell, which might impact the values of other cells (topological sort).







1) Sending Serialized Object Huge bandwidth (n^2) and time

2) Sending Operation Low bandwidth, very low execution time

3) Sending Binary Diff Low bandwidth, high execution time (small diff but long to compute)

Sources for images

- http://learningworksforkids.com/wp-content/uploads/WoW-screen-2.jpg
- https://lh4.googleusercontent.com/yUAOWyj2Gt_oVbLjJpPQnhX1
  IrwBvPAmPx6pnaihOxg9VRHZt7p28g5qUUbO013b6If-Mw=s640-h400-e365