CacheDOCS: A Dynamic Key-Value Object Caching Service

Julien Gascon-Samson*
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, BC, Canada
*Email: julien.gascon-samson@ece.ubc.ca

Michael Coppinger†, Fan Jin†, Jörg Kienzle‡, Bettina Kemme‡
School of Computer Science
McGill University, Montreal, QC, Canada
†Email: {michael.coppinger,fan.jin}@mail.mcgill.ca
‡Email: {joerg,kemme}@cs.mcgill.ca

***Abstract*—Caching plays an important role in many domains, as it can lead to important performance improvements. A key-value based caching system typically stores the results of popular queries in efficient storage locations. While caching enjoys widespread usage in the context of dynamic web applications, most mainstream caching systems store static binary items, which makes them impractical for many real-world applications that would benefit from storing dynamic items.**

In this paper, we propose CacheDOCS, a dynamic key-value object caching service that allows for caching arbitrary objects. As part of our model, CacheDOCS provides an API that supports the execution of operations against cached objects, and allows for clients to seamlessly subscribe to keep their local copies in sync with cached remote objects. CacheDOCS supports multiple update dissemination strategies in order to optimize performance, and proposes a versioning mechanism to ensure consistency. We implemented a full version of CacheDOCS and we ran several performance-related experiments under three use-case scenarios.

I. Introduction

Caching plays an important role in many domains nowadays, as the use of such techniques can significantly reduce the load on systems that need to process complex operations, by typically storing the results of commonly-requested queries in efficient storage locations. Key-value based caching notably enjoys widespread usage in the world of web applications, as these applications often generate large amounts of complex queries that are processed repeatedly and in parallel. In this context, the results of commonly-executed queries are cached in fast memory such as in RAM [7], [1], [12]. Then, should multiple clients be launching the same complex query over a given time interval, then the query can be launched only once, and the cached result can be reused, which can reduce the load on the database system.

While web applications constitute the *de facto* usage pattern of caching systems, other applications can benefit from caching as well. For instance, a large-scale multiplayer game might benefit from caching the large pool of in-game objects that players frequently access. Alternatively, a collaborative document editing platform might cache the documents that users are currently accessing. An important limitation of mainstream caching systems however is that they are typically *static*, which means that changes cannot be applied to a given cached item; instead, upon changing, the value must be overwritten (more details given at section II-A). A second limitation is that upon a given cached value changing, clients are not notified of the change. In a web application context, these limitations might be acceptable, as the cached result of a given database query might simply overwrite the previous result of the query, and that clients typically perform database requests (or access the cache) on-demand; i.e., in a pull-based manner. However, in other applications, such as the two examples aforementioned, which might benefit more from a push-based paradigm and in which data items are expected to be updated, the use of a static caching system might not be a suitable choice.

In this paper, we propose CacheDOCS, an object-based key-value caching service that extends the principles of mainstream key-value stores in order to capture the dynamism and meet the needs of push-based applications. For instance, in a multiplayer game context, the caching of game-specific objects that can be dynamically retrieved and updated by players could alleviate the load on the game server infrastructure. In a collaborative document editing platform scenario, objects representing the documents that are being edited by several users could be stored in the cache; thus alleviating the load on the service-specific server infrastructure.

As major contributions, CacheDOCS provides three main improvements over the typical key-value store model:

1) Full objects can be cached as opposed to byte sequences.
2) Objects can be updated by allowing for methods to be invoked over cached objects, following the principles of *remote method invocation* (RMI).
3) Interested clients can transparently be notified and their local copy of a given cached object can automatically be updated whenever an update is performed on the cached object, following a publish/subscribe model.

This paper provides the following additional contributions:

- We provide a consistency model that prevents conflicting updates on cached objects, and that maintains the consistency of local copies of cached objects
- We provide several object update dissemination strategies to reduce bandwidth usage and increase performance
- We built a full system implementation in Java, using RMI [10] and using the Dynamoth [8] pub/sub platform
- We ran several experiments under three use cases to analyze our system under various scenarios

This paper is organized as follows: in section II, we present some background concepts on which CacheDOCS is built; in

section III, we present the architecture and the model of our system; in section IV, we describe our implementation and the various experiments that we ran; in section V, we present the related work and finally, in section VI, we present our conclusions and future work.

II. Background

A. Key-Value Stores

Key-value stores such as the well-known Memcached open-source platform maintain a list of $\{k, v\}$ pairs, where k is a key and v is the value corresponding to key. Such systems can be viewed as large-scale hash tables. k and v are typically byte sequences; as a result, storing an object as a value requires serialization. For efficiency purposes, some or all of the $\{k, v\}$ pairs are stored in RAM.

Key-value stores provide $\texttt{get}(k)$ and $\texttt{put}(k, v)$ operations to respectively retrieve an item v from key k, and to store a given $\{k, v\}$ pair. From a developer's standpoint, prior to computing v from k, one typically looks whether k exists in the cache, and uses the corresponding value if it exists. Otherwise, v is computed from k (in a web/database context, a given database query k would be sent to the database engine to be computed to obtain v). Then, the pair $\{k, v\}$ is put in the cache for future read accesses. Note that the update of v requires a new invocation of `put`.

B. Publish-Subscribe

The publish-suscribe (pub/sub) paradigm [6] allows for efficient decoupling of content producers from content consumers by proposing a model where content consumers, referred to as subscribers, express interest in receiving some content, and where content producers, referred to as publishers, generate content that is disseminated to interested subscribers.

Several pub/sub variants have been proposed in the literature; the most common ones being topic-based and content-based publish/subscribe. In topic-based pub/sub, subscriptions are expressed over *topics*, which is typically a string. Publications are also tagged with a *topic*, and are disseminated to subscribers that have previously subscribed to the corresponding topic. In contrast, in content-based pub/sub, subscriptions are expressed over more elaborate predicates, which results in a more complex matching process that matches the *contents* of the publications themselves against the subscription predicates in order to determine to which subscribers they should be issued. Content-based pub/sub is outside the scope of this paper and we will focus our attention on topic-based pub/sub, which bears some similarities with key-value stores, as both make use of a *key* as a decoupling and indexing mechanism.

Conceptually, a topic-based pub/sub service, such as Dynamoth [8] or Redis [1], maintains a list of pairs $\{t, \mathbb{S}\}$, where t is a topic and serves as a key and $\mathbb{S}$ is a set of subscribers for topic t. Topic-based pub/sub systems typically expose three main operations: $\texttt{subscribe}(t)$, which adds the caller (subscriber) to $\mathbb{S}$, $\texttt{unsubscribe}(t)$, which removes the subscriber from $\mathbb{S}$, and $\texttt{publish}(t, m)$, which transmits publication message m to the subscribers of t. Many applications can be built over topic-based pub/sub systems, such as push-based notification systems or distributed event-based systems, in which event consumers can subscribe to events that they are interested in receiving and in which event producers can push relevant events to interested consumers.

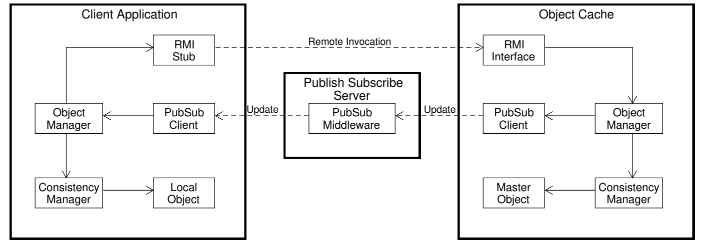


Figure 1: Architecture of CacheDOCS

III. System Model

Our system, CacheDOCS, proposes an augmented key-value based caching service that supports the caching of arbitrary objects across many different applications. In addition to adding and removing objects from the cache, it also supports the execution of arbitrary operations (methods) over cached objects. Finally, it allows clients to subscribe to objects for which they wish to be notified of changes, and it keeps the local copy of cached objects fully synchronized with the cache.

Figure 1 presents the high-level architecture of CacheDOCS. As an implementation choice, our caching service allows for storing Java objects and exposes an API through a Java RMI interface that clients use to interact with the service. To simplify, we take that model as an example to describe our system, but in practice, a different implementation could support arbitrary languages and communication paradigms.

A. Object Caching as a Service

The object cache service, denoted on the right, accepts requests from clients and stores pairs of $\{k, v\}$ in the *object manager*, where v is a Java object that maps to key k. The CacheDOCS API, exposed through a RMI interface, supports the typical key-value store operations $\texttt{get}(k)$ and $\texttt{set}(k, v)$ to respectively retrieve and store a Java object (from now on, we denote a remotely cached object as v_r, to distinguish from a local copy v_l). The invocation of a `get` operation returning remote object v_r first triggers the serialization of v_r, then the transmission of the serialized state to the client, which then deserializes it to reconstruct a local copy v_l of the fetched object. The same process is followed in reverse order to store objects in the cache (`set` operation). Note that as an optimization, upon serializing an object v_r, CacheDOCS caches the serialized sequence of bytes to reduce the performance costs of serialization. Note that the `get` and `set` operations offer a level of features comparable to a typical key-value store, with the addition that they support the storage and retrieval of objects and not only sequence of bytes.

B. Synchronization of Cached Objects

The basic `get` operation mentioned in the previous section does not maintain a *link* with the cached object v_r; i.e., it retrieves a snapshot of the currently cached *version* of v_r. As a consequence, locally performed updates (local

method calls) are not propagated to the cache and remote method calls performed on the cache on are propagated to the caller of `get`. For that reason, CacheDOCS also exposes a `getAndSubscribe`(k) operation, which first allows one to retrieve an object v_r, and then keep in sync the changes carried out on the local copy v_l and on the remote copy v_r. Upon `getAndSubscribe` being invoked for key k, a pub/sub subscription is also established on *topic* k, through the pub/sub middleware on the pub/sub server (which can be collocated with the cache service). Then, on client side (left of figure 1), the pair $\{k, v_l\}$ is registered with the *local* object manager, which, in conjunction with the *remote* object manager, is linked to the pair $\{k, v_r\}$ to provide two way synchronization. The following section describes the process by which operations are invoked on cached objects.

C. Invoking Operations

After retrieving a synchronized copy v_l of a cached object v_r, a given client can invoke operations (methods) on v_l. Methods can be classified into two categories: A- deterministic read-only methods that do not alter the state of the object (i.e., functions which simply return a value such as a *getter*), and B- all other methods which alter the state of the object (or which are read-only and non-deterministic). Note that in our current implementation, a developer relying on the CacheDOCS API must indicate which methods belong to category A, through Java annotations. We are currently looking at code analysis techniques to automate this process.

For methods of category A, there is no need to forward the operation invocation to the remote cache manager, as a given local object v_l contains the same set of attributes as the corresponding remote object v_r. As such methods only access v_l's attributes, and that all attributes are synchronized upon retrieving v_r, then it can be executed locally. However, for methods of class B, the invocation must be forwarded to the remote cache. To do so, the local object manager traps all method calls requiring forwarding. When the invocation of one such corresponding method is requested, the method signature as well as all of it's parameters are serialized. Then, the call is automatically dispatched to the remote object manager through RMI.

Upon receiving the method invocation request, the remote object manager deserializes the method signature along with the parameters, locates the relevant object v_r through it's key k, and invokes the method using *reflection*, which triggers a state change of v_r. In order to make sure that all *subscribers* $\mathbb{S}$ to k obtain the latest version of v_l, CacheDOCS propagates the update to all $S \in \mathbb{S}$ through the pub/sub interface, using one of several possible strategies, which are discussed in the next section.

D. Update Propagation Strategies

CacheDOCS supports four different propagation strategies to disseminate object updates to subscribers. For each strategy, we discuss their usefulness and trade-offs in terms of bandwidth and CPU usage. We consider a key k, a remote object at states v_{r0} and v_{r1}, respectively *before* and *after* the invocation of the requested operation, a corresponding local object v_{l0} (before applying the update) and v_{l1} (after applying the update) for every subscriber $S \in \mathbb{S}$ to k. Note that we assume that $v_{r0} = v_{l0}$ and $v_{r1} = v_{l1}$.

1) Serialized Object Update Strategy: v_{r1} is serialized and sent to all subscribers. As mentioned previously, as an optimization, CacheDOCS will cache v_{r1} after an initial serialization. Then, S performs deserialization, and replaces v_{l0} by v_{l1} in the local cache. This strategy can be efficient for small objects, as the propagation time and the bandwidth should remain low. It can also be efficient for CPU-intensive operations, as they will need to be executed only once (on the remote cache). For large objects, however, it might be inefficient, and bandwidth consumption might be very high.

2) Operation Update Strategy: The method invocation along with the parameters is sent to all subscribers in serialized form. Then, all subscribers invoke the operation over v_{l0} in their local cache to obtain v_{l1}, following the same principle as for clients sending their method invocations to the service, as described at section III-C. This strategy can be efficient for large objects, as the serialization of the operation will most likely be small. It might be less efficient for CPU-intensive operations as the operations will need to be executed both on the remote server and locally.

3) Binary Diff Update Strategy: Remote object states v_{r0} and v_{r1} and converted to binary form through serialization. Then, a binary diff is computed between the binary representations of the the new and old states, and transmitted to subscribers. The subscribers must then serialize v_{l0}, apply the binary diff and deserialize the patched binary representation to obtain the new state v_{l1}. This strategy can be costly in terms of CPU usage, but it can nevertheless be a good choice for CPU-intensive operations executed against a large object, as it can yield similar benefits as the serialized object update strategy with potentially smaller update sizes.

4) Attribute Patch Update Strategy: A list of attributes that were changed in v_{r1} compared to v_{r0} is computed (pairs of $\{\texttt{attr}, \texttt{newVal}\}$) and sent to all subscribers. Using reflection, each subscriber *patches* each attribute that was modified. Note that in our current implementation, we require developers using the CacheDOCS API to tag the set of attributes that are changed by a given method, using Java annotations. We are currently investigating the use of static and dynamic analysis techniques to automate this process. This strategy can be efficient for large objects and CPU-intensive operations, as the performance hit of serialization-deserialization and binary diff generation can be avoided, as well as the performance hit of executing the operation twice (on the remote cache, then on the local copy).

E. Consistency Management

As with any distributed system, consistency issues may arise due to the interleaving of transactions. Considering a local object at state v_{l0} and a corresponding remote object at state v_{r0}. Assuming both subscribers S_1 and S_2 perform an operation simultaneously, the remote object cache could generate states v_{r1} and v_{r2}. This would be incorrect, as both S_1 and S_2 assumed object v to be at state v_{r0} prior to executing the operation, which might be dangerous.

The remote and local consistency manager components are in charge of tracking such inconsistencies. Upon submitting an operation for object v to the remote cache, a given subscriber S piggybacks it's current version number for object v. In the example above, the consistency manager would accept the first operation (let it be from S_1), and would then reject the second. S_2 would be notified of the execution failure, then would be updated to state v_{l1} and could then decide whether to reinvoke the operation at the new local state v_{l1}.

On the other hand, the local consistency manager is in charge of preventing concurrent accesses to a given object v; i.e., to prevent v from being simultaneously accessed and updated to a newer version through the pub/sub interface. Due to space constraints, the exact mechanism cannot be entirely described here.

F. Nested Objects

Our current iteration of CacheDOCS supports the storage of nested Java objects, but with the limitation that we only trap direct method calls over the *root* object, and not method calls invoked over nested sub-objects. We are currently investigating mechanisms that would allows us to trap nested calls using aspect-oriented programming (AspectJ [9]) as well as an extension which would allow our service to support the storage of linked objects; i.e., assuming object v_A contains a reference to v_B, and that both objects are stored in the cache, then an update operation performed on v_A should also impact v_B.

IV. Implementation and Experiments

A. Implementation

As mentioned previously, we built a full implementation of CacheDOCS in Java, and our object caching service caches Java objects. The service offers a RMI-based API that clients can consume to access the cache, retrieve, store and subscribe to Java objects, as well as a client-side library which handles transparent synchronization of updates on tracked local objects. We chose the Dynamoth [8] publish/subscribe service running on top of the Redis middleware to support object update dissemination.

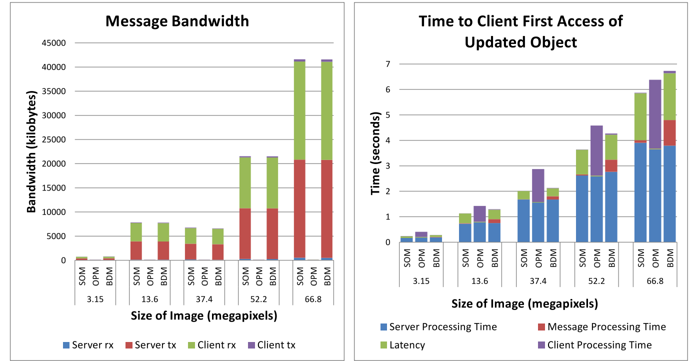
As mentioned previously, an alternate implementation could be made to support other languages and object paradigms, while applying the principles of our model.

B. Experimental Setup

All of our experiments were executed over machines of the McGill's School of Computer Science labs. For all experiments, one machine was used to run the caching service, and one machine was used as a client to the service.

Note that in the context of this paper, we opted to run targeted experiments that model three specific use-cases in which CacheDOCS could be used. For all experiments, we analyze and compare the performance of the different update propagation strategies, under two angles:

1) The incoming and outgoing bandwidth usage incurred by the operation invocation on the client (*client tx* and *client rx* curves) and on the server (*server tx* and *server rx* curves).



(a) Bandwidth Usage (b) Operation Invocation Time

Figure 2: Collaborative Graphics Editing Results

2) The time needed (1) for the server to execute the operation (*server processing time* curve), (2) for the server to generate the update dissemination message to be sent through the pub/sub interface (*message processing time* curve), (3) for the server to disseminate the update back to the client (*latency* curve) and (4) for the client to deserialize the update and apply the update (*client processing time* curve).

In all graphs, the various propagation strategies are denoted as follows: (a) Serialized Object Update is labelled as *SOM*, (b) Operation Update is labelled as *OPM*, (c) Binary Diff Update is labelled as *BDM* and (d) Attribute Patch Update is labelled as *AM*.

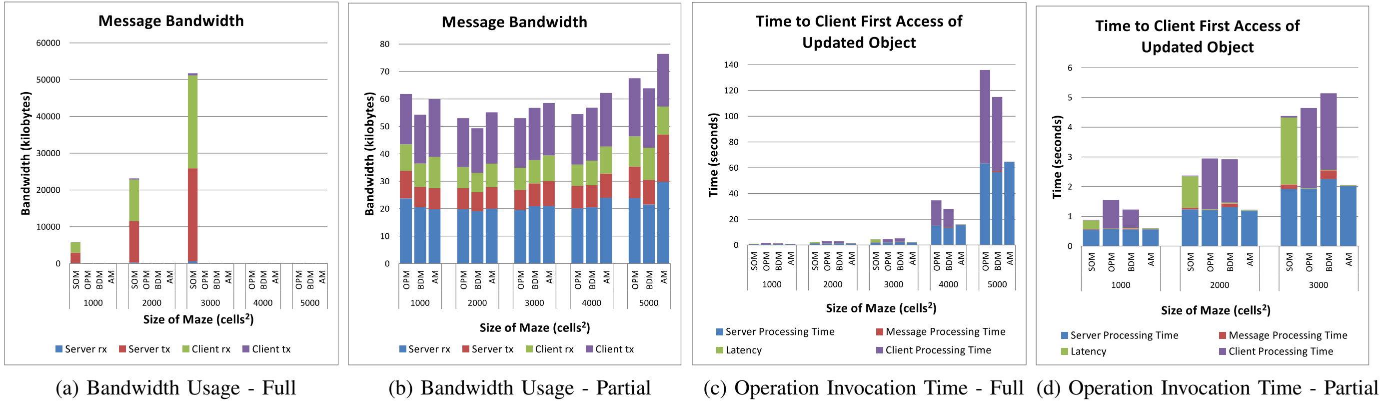
As future work, we will be looking at evaluating the global performance as well as the scalability of a real deployment of CacheDOCS in the cloud with large amounts of clients.

C. Collaborative Graphics Editing

The goal of this experiment is to model clients working on a distributed graphics editing application, in which the images that are being edited are stored as objects in the cache. As a result, the operations are submitted to the cache, which disseminates the updates to the clients using the various strategies. As part of our experiment, we invoked an image rotate method call over an image of various sizes (from 3.15 to 66.8 megapixels). Our results are shown in figure 2.

An image rotation operation becomes fairly expensive over larger images; thus, significant server processing time is required on the server (figure 2b), and on the client for OPM (as the operation itself is forwarded). However, for that same reason, bandwidth usage (figure 2a) is minimal for OPM compared to the other approaches which require sending a large amount of data (the full serialized state for SOM or the binary diff for BDM). Note that since a rotation transforms all pixels, the binary diff approach is very inefficient.

Overall, the SOM approach leads to an overall lower operation invocation time although the difference is not marginal. Therefore, in this context, the OPM approach would most likely be the best one as it performs only slightly worse than SOM, but it uses practically no bandwidth. Note that



(a) Bandwidth Usage - Full (b) Bandwidth Usage - Partial (c) Operation Invocation Time - Full (d) Operation Invocation Time - Partial

Figure 3: Maze Multiplayer Game Pathfinding Results

we omitted AM, as an image object does not really contain attributes, but raw pixels stored in an array.

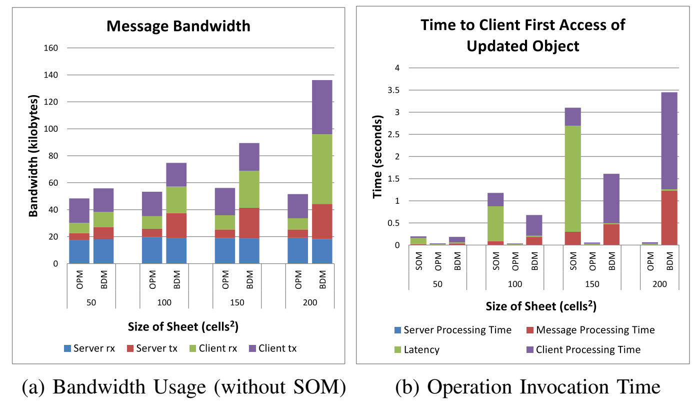
D. Maze Multiplayer Game Pathfinding

The goal of this experiment is to model a maze game application with several players. In the context of this experiment, we store a maze object which contains a 2D array of cells and a list of players. For each cell, there can be four possible walls (left, right, up, bottom), represented by boolean values. Each player contains a cell position. For our experiments, we invoke a *movePlayer* method on the maze object, over mazes of varying sizes (1000x1000 to 5000x5000). This method uses a pathfinding algorithm to determine if a given player is allowed to move from his current cell to another target cell. Note that for the Attribute Patch (AM) strategy, the invocation of this method modifies the *list of players* attribute of the maze class.

Our results are shown in figure 3. In subfigures 3a and 3b, we respectively show the bandwidth usage for all strategies, and for all strategies *except* SOM, as it is higher by order of magnitudes. In figures 3c and 3d, we respectively show the operation invocation time results for all maze sizes, and for maze sizes between 1000x1000 and 3000x3000 only, as the growth is quadratic.

We first observe that the bandwidth usage of SOM becomes extremely high as the quadratic maze size increases (for that reason, we omitted the execution of SOM for maze sizes above 3000x3000). For all other strategies, the bandwidth remains very small and almost constant. This is explained by the fact that OPM only sends the operation to be invoked, that BDM only sends a diff, which is efficient as very few changes are carried out to the large object and that AM only sends the *list of players* attribute, which is marginal compared to the size of the *maze tiles* attribute.

Regarding the operation invocation time, we observe that it grows in a linear-like pattern as the total number of cells increases. In all approaches, as it can be expected, the server processing time is the same, as the server takes the same time to execute the operation. The pathfinding algorithm takes a significant time to be executed over large mazes, as can be expected. With OPM, we practically double that time, as clients also need to execute this same algorithm; therefore, OPM is the worst choice in large mazes. As SOM requires a large amount of bandwidth, the propagation delay (latency) takes a considerable time. For BDM, a significant time is also spent at client-side to apply the binary patch, which is almost comparable to the operation execution time. To a lesser extent, BDM also adds some processing overhead (to generate the diff) server-side. AM is then the most efficient approach in this case, as it is lighter than all other approaches (the only noticeable processing time is attributed to the operation invocation itself server-side).



(a) Bandwidth Usage (without SOM) (b) Operation Invocation Time

Figure 4: Collaborative Spreadsheet Editing Results

E. Collaborative Spreadsheet Editing

In this experiment, we model a multi-user collaborative spreadsheet editing tool *à la* Google Sheets. The cache is used to store *spreadsheet* objects which are then retrieved by clients to perform operations and receive updates. The spreadsheet object contains a list of cells. Each cell can hold a value or a formula. In the case of a formula, the cell might depend on a set of other cells. Thus, upon a user *editing* a cell, a topological sorting of the dependent cells must be performed, and these cells need to be reevaluated as well.

The cells in the first column are initially set to 1 and the remaining cells, except for the last column, are set to the sum of the previous cells in the row. The cells in the last column are set to the sum of all previous cells in the column. We ran experiments over spreadsheets of size 50x50, 100x100, 150x150 and 200x200. In each experiment, we set the value of the first cell $(0,0)$, and we analyze the performance of this invocation. The results are given in figure 4.

The bandwidth results (subfigure 4a) do not include SOM, as these results were larger by orders of magnitude compared to the other approaches: for a size of 50x50, approximately 2000 kb, for 100x100, 16000 kb and for 150x150, 52000 kb. For performance considerations, the SOM strategy was not executed for the 200x200 spreadsheet. Note that results for AM could not be generated, since as mentioned previously, annotations must be declared manually in this version of CacheDOCS and that our spreadsheet model contains an *array* of cells (one large array attribute).

Our SOM results reveal that this strategy exhibits the worse performance, both from a bandwidth usage standpoint; i.e., the whole spreadsheet has to be serialized, and also from a invocation time standpoint, as the serialized spreadsheet is transferred over the network, which causes huge bottlenecks.

On the other end, despite the fact that the OPM strategy involves the execution of the method twice (server-side and client-side), it is nevertheless the most efficient approach in this case as bandwidth usage remains approximately constant irrespective of the spreadsheet size, as only the operation itself is transmitted. As for the BDM strategy, it is also very efficient bandwidth-wise, as the modification of only one cell has very limited impact on the size of the diff. However, a significant amount of time is spent on preparing the diff server-side and patching and deserializing client-side.

V. Related Work

To the best of our knowledge, we found no approaches that were similar to what CacheDOCS provides and that offered an equivalent set of features.

The RMI system itself [10] can be used to build a partial caching service, as *proxies* to objects can be exported and retrieved through the network. Then, method calls can be invoked on proxies of remote objects, which results in a remote execution. While concurrency is permitted, it must be manually handled by the use of appropriate Java constructs. Unlike our proposal, RMI itself does not provide transparent and dynamic update dissemination. As mentioned previously, we make use of RMI in our current implementation, in combination with other mechanisms. [4] proposes such a RMI-based approach and discusses the problem of the commutativity of method invocations over distributed objects. In [5], the authors propose a caching middleware for RMI applications, through the form of a drop-in replacement, which allows for some of the operations on distributed objects to be executed locally, and then applied later or in batch to the remote object, in order to increase performance. In [3], the authors propose an approach for caching object graphs across different clusters, and compare their solution against RMI. In [2], the authors apply load balancing techniques to distribute cached items among multiple servers. Their approach notably exploits the popularity and the inter-item relationships.

As opposed to SQL databases, object databases store data in a manner that allows for direct mapping between programming language objects and the data representation [13], [11]. Nevertheless, we think that CacheDOCS goes further as it allows for the execution of arbitrary operations through the form of method invocations on any arbitrarily cached object.

VI. Conclusion

In this paper, we presented CacheDOCS, a dynamic object caching service that extends the concept of a typical key-value store by allowing for full objects to be cached. CacheDOCS allows for updates to be applied on cached objects, and includes a pub/sub-based notification mechanism to efficiently dispatch updates to relevant clients in a transparent manner. Several update propagation strategies are provided to maximize performance and minimize bandwidth, and consistency is ensured through the use of a versioning-based update protocol.

As future work, we would like to offer better support for object nesting and for inter-object links. We are also planning on optimizing and testing CacheDOCS in a cloud-based, very large-scale setting. Finally, we are aiming at integrating dynamic selection of the best propagation strategy, as well as multi-server support in the cloud.

References

[1] Redis website (2013), http://www.redis.io/
[2] Asad, O., Kemme, B.: Adaptcache: Adaptive data partitioning and migration for distributed object caches. In: Proceedings of the 17th International Middleware Conference. pp. 7:1–7:13. Middleware '16, ACM, New York, NY, USA (2016)
[3] Banditwattanawong, T., Maruyama, K., Hidaka, S., Washizaki, H.: Proxy-and-hook: a java-based distributed object caching. In: INDIN '05. 2005 3rd IEEE International Conference on Industrial Informatics, 2005. pp. 819–824 (Aug 2005)
[4] Eberhard, J., Tripathi, A.: Semantics-based object caching in distributed systems. IEEE Transactions on Parallel and Distributed Systems 21(12), 1750–1764 (Dec 2010)
[5] Eberhard, J., Tripathi, A.: Efficient Object Caching for Distributed Java RMI Applications, pp. 15–35. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
[6] Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Comput. Surv. 35(2), 114–131 (2003)
[7] Fitzpatrick, B.: Distributed caching with memcached. Linux J. 2004(124), 5– (Aug 2004)
[8] Gascon-Samson, J., Garcia, F.P., Kemme, B., Kienzle, J.: Dynamoth: A scalable pub/sub middleware for latency-constrained applications in the cloud. In: ICDCS 2015. pp. 486–496 (June 2015)
[9] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ, pp. 327–354. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
[10] Krishnaswamy, V., Walther, D., Bhola, S., Bommaiah, E., Riley, G., Topol, B., Ahamad, M.: Efficient implementations of java remote method invocation (rmi). In: Proceedings of the 4th Conference on USENIX Conference on Object-Oriented Technologies and Systems - Volume 4. pp. 2–2. COOTS'98, USENIX Association, Berkeley, CA, USA (1998)
[11] Roopak, K.E., Rao, K.S.S., Ritesh, S., Chickerur, S.: Performance comparison of relational database with object database (db4o). In: 2013 5th International Conference and Computational Intelligence and Communication Networks. pp. 512–515 (Sept 2013)
[12] Waddington, D., Colmenares, J., Kuang, J., Song, F.: Kv-cache: A scalable high-performance web-object cache for manycore. In: 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing. pp. 123–130 (Dec 2013)
[13] Wells, D.L., Blakeley, J.A., Thompson, C.W.: Architecture of an open object-oriented database management system. Computer 25(10), 74–82 (Oct 1992)