Optimistic Scheduling with Geographically Replicated Services in the Cloud Environment (COLOR)

Wenbo Zhu[1], Murray Woodside
Systems and Computer Engineering
Carleton University
Ottawa, Canada
{wenboz, murray.woodside}@sce.carleton.ca

***Abstract*—This paper proposes a system model that unifies different optimistic algorithms designed for deploying geographically replicated services in a cloud environment. The proposed model thereby enables a generalized solution (COLOR) by which well-specified safety and timeliness guarantees are achievable in conjunction with tunable performance requirements. The proposed solution explicitly takes advantage of the unique client-cloud interface in specifying how the level of consistency violation may be bounded, for instance using probabilistic rollbacks or restarts as parameters. The solution differs from traditional Eventual Consistency models in that inconsistency is solved concurrently with online client-cloud interactions over strongly connected networks. We believe that such an approach will bring clarity to the role and limitations of the ever-popular Eventual Consistency model in cloud services.**

Keywords-optimistic algorithms; replication; cloud service

I. INTRODUCTION

Geographically replicated services provide reliability and availability in face of network or cluster level failures. When designed correctly they also allow the service to scale better with improved client-to-cloud communication and load balancing for serving internet-scale clients. However, it is known that performance and consistency are antagonistic requirements in such an environment. As a result, the so-called Eventual Consistency (EC) model [1] (which accepts reduced consistency) is being widely accepted, e.g. in light of the well-known CAP theorem [2].

In contrast to such an all-or-nothing approach, this research considers a model under which inconsistency is largely bounded and is resolved in an online fashion with a generalized client-cloud interface. The proposed solution is called COLOR (Client-Oriented Layered Optimistic Replication), and the key contributions lie in its optimistic algorithms that 1) actively detect and 2) concurrently resolve inconsistency as clients interact with the cloud. The effectiveness of COLOR is demonstrated by its novel capabilities to support tunable performance under both the steady-state and fail-over events.

In COLOR, a set of tunable parameters are supported, including choice of different weak algorithms, their run-time parameters such as timeout or quorum, and most importantly partial ordering in the form of state partitioning. These parameters allow detailed analytic or simulation models to be created to evaluate the trade-off between performance and consistency, while the latter may be measured as perceived system failures by the client.

This paper is organized as follows. Section 2 surveys related work, and Section 3 specifies the system model. Section 4 describes several optimistic algorithms in the context of COLOR system model and explains how the level of consistency may be quantified and tuned, along with different application-level concerns in adopting the COLOR based solution. Section 5 examines the tunable trade-off with analytic models that involve two basic total-ordering algorithms. Section 6 discusses the full scope of this work and remaining challenges, and Section 7 closes with a discussion of contributions.

II. RELATED WORK

One of the most notable works on tunable consistency can be found in [3], which defines three dimensions for quantifying the level of inconsistency: 1) message ordering; 2) staleness; and 3) numeric gap. However, it does not offer strict consistency to applications and the overall solution represents an offline EC model which reconciles diverged replica states only via out-of-band mechanisms. In contrast, [4] presents a solution that leverages the client-server semantics, and inconsistency is detected and solved directly as it happens. However, their solution aims to hide inconsistency from clients which then have to participate directly in some blocking consensus algorithm. Since inconsistency is restricted to the server and strict consistency is enforced externally, the performance issue remains. Similar solutions can also be found in [8]. None of these solutions is practical in a cloud environment due to its complicated client-side logic and one-to-many connectivity requirement.

Many optimistic multicast or replication algorithms have been proposed which aim to either improve performance at the algorithm level by taking advantage of specific communication patterns [9] or to give up strong consistency to trade for better performance. We are not particularly interested in the first group of optimistic algorithms due to their limited performance gains in a WAN environment. COLOR leverages those multicast algorithms that don't guarantee strong consistency. However COLOR doesn't treat them just as binary parameters as in traditional optimistic

[1]Current Affiliation: Google Inc.

replication solutions such as [10]. Rather COLOR identifies different run-time parameters, such as the delay interval in committing tentative messages in order to tune the effect of resulting inconsistency in the view of applications.

Several multicast or replication algorithms have exploited the so-called partial ordering ([6] [7]) in order to improve the performance. These algorithms are capable of delivering messages in parallel that do not mutually conflict with each other. Limitations of these algorithms are mostly practical ones because ordering relation between two messages has to be identified in the run-time and before messages are delivered. This is not always possible as conflicts of two messages may only be identifiable at the commit time. In contrast, COLOR has adopted a very different approach in which partial ordering is identified statically in the format of state partitioning which is done optimistically. In other words, partial ordering in COLOR is only optimistic because many-to-many relations may introduce conflicting changes across two partitions.

III. SYSTEM MODEL

The COLOR system model represents a typical cloud environment where state is replicated geographically and clients access the replicated service via some loosely coupled protocol and interface such as HTTP and REST [5]. The geographically distributed replicas of data centers will just be called "replicas". Each client often has a preferred replica at any point of time, called its "local replica". While this model may also be applicable to other application environments, we think that the cloud environment, as specified in the following, is important enough to warrant its own system model specification.

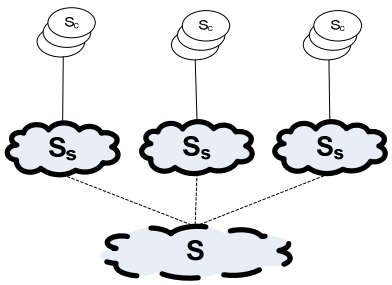

Figure 1. State Representation in the System Model.

A. State Representation

As shown in Fig. 1, the first element of the system model is the replicated global state, S, which is stored on every local replica (e.g. within each data center) and is fully replicated across multiple such replicas. S is deterministically created and replicated via a totally-ordered history of messages, denoted as *hist*(S). Each replica has a local version of S, denoted as S_s, for which a prefix of *hist*(S_s) is globally-committed and made consistent across replicas; with the remaining part being tentatively committed only on the local replica. The committed prefix on a local replica is denoted as *pre*(S), while uncommitted history is denoted as *post*(S_s). Each client that interacts with the replicated service also possesses a local, client-view of S, denoted as S_c, which is created on the client via a history of interactions with one or multiple replicas. Interactions between the client and replicas include 1) queries of S; 2) notifications of changes made to S; and 3) client-initiated updates to S. Only update messages from 3) are included in *hist*(S) when they are committed globally.

Further based on the above definition we note that the local state representation of S_s and S_c respectively serve also as the interface through which a client interacts with the cloud. The idea behind REST is a common property seen in today's cloud applications [5]. Most importantly by modeling the client-to-cloud interface against a common state representation it becomes possible to enable fine-grained state invalidation and recovery as discussed in the following sections.

B. Consistency Requirements

The second element of the system model is the consistency requirements, which start with the following two properties: 1) a local replica has a knowledge of some current version of the globally committed *hist*(S), i.e. *pre*(S) and any uncommitted *hist*(S_s) constitutes its local post(S_s); 2) a client may request specific update messages to be committed globally but otherwise its local S_c represents a tentative version of uncommitted *hist*(S_c), i.e. *post*(S_c).

Invalid state may be created on a replica when its local state S_s is updated from any uncommitted *post*(S_s), which may be the result of different optimistic algorithms. Consequently S_c may become invalid when the client receives messages from a replica with any uncommitted *post*(S_s). Any message created from an invalid S_s or S_c is by definition invalid and will invalidate the local state of the recipient if delivered.

Our definition of consistency requirements differs from traditional state-machine based replication [11] in several ways:

1) Messages are a logical concept and the only source of truth is the global State S and its change history.

2) Inconsistency is always the result of direct violation of causality on the client-side due to invalid state representation being received or sent.

3) Local *hist*(S_c) or *hist*(S_s) serves merely as change versions of S. As such, the overall solution is not subject to indeterminism issues seen in traditional state-machine based replication.

C. Client Messages

As discussed in the previous section, it's important to note that the role of a client is only a logical concept. More specifically client messages may or may not be generated directly on a physical client. How a client message is identified is purely decided by whether or not the message makes up *hist*(S).

Situations when a client message may instead be generated on the local replica include: 1) stateless or stateful server-side "stored procedures"; 2) non-deterministic local scheduling. Typical cases for 1) also include messages generated from server-side sessions. Local in-memory

[1]Current Affiliation: Google Inc.

transaction scheduling is one typical case for 2). In the extreme case when a centralized RDBMS is used for scheduling messages, the centralized RDBMS server effectively becomes the client (in the view of the storage layer where S is generated and stored).

Nevertheless, in all the above cases it's always the physical client that owns S_c. Furthermore, client messages in *hist*(S) are always originated from and by the physical client.

Under the above abstraction, multi-tiered servers are not a direct concern to COLOR because client-messages and state updates are now interchangeable. This requirement also closely follows the general concept of REST and therefore any RPC style of client-messages that invokes remote business logic on a local replica is not directly involved in this system model.

We also note that direct client-to-client communication is not considered. This is to say that inconsistency as a result of out-of-band communication is not addressed in COLOR. When such a requirement exists, any client initiated messages will have to be explicitly committed to the global S, and only messages generated out of a valid S is allowed to be sent back to the client. As discussed next, inconsistency in the COLOR is to be resolved in real-time and the assumption is that transient inconsistency on a client is tolerable as long as 1) inconsistency is bounded both in time and space; and 2) applications are aware of tentative *post*(S_c).

D. Extended Semantics

Depending on different application and system requirements, session state may also be persisted and replicated as part of the global state S (but with its own partition). Otherwise we simply treat any local session state as the extension of client state S_c. The fact that session state may only be accessed by a single client is not particularly interesting to this model.

It's worth noting that the so-called session consistency [12] is trivially enforced under the COLOR model as the starting position of any *post*(S_c) should be monolithically increasing as a client switches from one local replica to another. The same requirement applies to the FIFO programming order, which has to be enforced on a local replica for as long as a client stays connected to this replica.

Next, we note that queries are to be executed locally, i.e. only on the local replica, in contrast to concurrent update messages which have to replicated and executed on every replica. Local concurrency control (CC) is required although updates are to be serialized globally. Explicit client-server CC state should be treated as part of S, e.g. change versions, locking state.

Because this model doesn't assume the server state is valid at all times, it significantly differs from traditional offline-client models, in which causality is simply guaranteed by always assuming valid server state while conflicts are resolved locally with optimistic CC.

IV. OPTIMISTIC SCHEDULING

Following the above definition of the COLOR system model, the key to optimistic scheduling is the scheduling algorithm that decides the order in which client messages, whether they are reads or writes, are applied to S against each other. Message ordering is by nature a blocking operation and the underlying scheduling delay decides not only the client-perceived performance but also the overall scalability of the replicated service. Optimistic scheduling aims to improve performance by allowing local or non-uniform agreements [11] when a global uniform agreement cannot be made timely.

A. Online Algorithms

An optimistic algorithm is considered an "online" algorithm if conflicts are resolved concurrently at the same time when clients are interacting with the replicated service and when new messages are being applied to the replicated global S. On the contrary, an "offline" algorithm will resolve conflicts out-of-band of client-cloud interactions by having clients operate against an always valid snapshot of S and by merging *post*(S_c) as a single update message to the global S.

1) Optimistic Ordering Agreement

The first category of online algorithms allows for relaxed global total-ordering. Uniform Total-Order has inherent overhead from running required consensus algorithms. Its performance is also subject to communication instability especially when the number of replicas increases. By allowing non-uniform Total-Ordering, messages will be delivered with no more than one server-to-server unicast delay under the steady-state. Moreover, in the case when total ordering is decided on a single master replica that most clients interact with, even the unicast delay is avoided and the whole ordering overhead is reduced to FIFO ordering.

In the event of a master failover when a remote replica responsible for global ordering becomes non-responsive, the ordering algorithm in question may also be reduced to FIFO ordering which can be decided entirely locally. This level of optimistic scheduling has rarely been studied before and it bounds the transient delay of a master fail-over or membership change, which may become blocking or unstable in the event of network partitioning.

2) Optimistic Membership Termination

The second category of online algorithms is focused on fail-over events. While consensus algorithms such as Paxos [6] ensure a certain degree of liveness, their performance degrades in face of any unstable leader election, especially when network communication is unstable or partitioned. In such a case, an optimistic fail-over algorithm will employ best-effort heuristics to decide on a temporary leader among replicas that are well connected. Such an algorithm allows for faster termination of concurrent leader election and keeps the underlying scheduling algorithm progressive even in face of cascading fail-over events. Network connectivity issues represent one of the most common failure types for which the well-known EC model is usually adopted.

Early termination of membership agreement is not well studied as an optimistic algorithm. COLOR focuses on this special type of optimistic scheduling due to the following considerations:

- It matches the goal of timely message delivering in COLOR.

[1]Current Affiliation: Google Inc.

- It introduces no special consistency requirement, and the solution to any transient inconsistency as a result of optimistic membership agreement can be solved similar to the first category.

3) Optimistic Partial Ordering

The third category of optimistic scheduling in COLOR employs state partitioning as an optimistic way of executing partial ordering. Given the sheer scope of the global S for most cloud-based applications, it is mandatory to break S into multiple partitions so that updates are totally ordered only in the scope of each individual partition. Partitioning offers a significant performance gain due to increased parallelism and reduced scheduling delay, especially when the set of replicas for each partition may be dynamically allocated. However, partitioning isn't always possible when partitions of S involve any many-to-many relations. Optimistic partitioning is then introduced with the assumption that conflicts involving relations are rare if S is partitioned properly (for the purpose of optimistic replication).

As noted before, partitioning can also be seen as a form of partial ordering. Compared to Generic Broadcast [7] or Generalized Paxos [6], partitioning based partial-ordering are much more practical to implement since the ordering knowledge is static and available a priori. On the other hand, violation of ordering relation has to be detected instead in the run-time after updates have been committed globally.

Optimistic partition offers an ideal tunable space to COLOR since the granularity of partitions can be changed in the runtime based on conflicting rate or other run-time stats. Moreover its state based scheme fits extremely well with the first two categories and the REST based client-to-replica interface.

Complexities of optimistic scheduling lie in mostly how to deduce invalid *post*(S_c) and how to minimize the algorithm overhead now that messages from a different partition may be able to invalid the state of the current partition. Garbage collection of *hist*(S) is a challenge too. Our current solution in prototype involves a hierarchical partition model to manage the above complexity as well as to offer a pre-specified tunable parameter to the COLOR system.

B. Outline of an Algorithm with Bounded Inconsistency

Each of the above algorithms involves some local *hist*(S_s) at a replica, which includes two parts: 1) *pre*(S) represents the committed message history which may not be entirely known to the local replica; and 2) *post*(S_s) represents the uncommitted message history generated by the local replica.

On the client side, every new message from the local replica carries a local version of the *hist*(S_c) signature which includes 1) a tuple of a *post*(S_c) hash and the last message-id of *post*(S_c); and 2) the last message-id of *pre*(S). Since clients may interact with different replicas and read-only messages are only executed on the local replica, the client is expected to keep multiple copies of *hist*(S_c), each from a different replica. There is no computation required on the client and the local *hist*(S_c) copies will be piggybacked (e.g. as HTTP cookies) with each client-initiated message.

When one or more of the following exception conditions are detected by a local replica, the client will be signaled to invalidate and reload S_c as if the system had restarted:

1) When a list of hist(S_c) is received on a local replica, re-hash each pre(S) up to the last message-id of post(S_s). Any mismatched hist(S_c) invalidates S_c.

2) Each message-id contains a global sequence-id and a local message id. If the last message-id of post(S_c) cannot be found in any pre(S_s), also invalidate S_c.

It is possible that the message-id of a newly received *post*(S_c) is earlier than what's known to the local replica. The local replica has the choice to reject such a message (e.g. for the sake of session consistency) as the current replica may lag in the global *pre*(S). The client will then fail-over to a different replica. The above algorithm also has to deal with retransmissions, concurrent *hist*(S_s), while keeping the client-side algorithm mostly passive and transparent to applications.

The algorithm outline here is much simplified, merely to illustrate the following design principles that are keys to the effectiveness of COLOR:

- The client-side logic should be kept to minimum or almost zero.
- The notation of message or message-id is completely transparent to the client run-time, which only knows the identity of a partition (in preparation of invalidation).
- Clients are not allowed to choose local replicas and client-side fail-over is common although messages from a single client should be routed to the same replica at any single point of time, as an implementation-level optimization.

C. Commit Protocols

The choice of commit protocols is completely out of the scope of COLOR. Any commit protocol could be supported to generate the global S and the corresponding *hist*(S). For example, a uniform consensus algorithm could be executed in conjunction with optimistic ordering. Or a centralized RDMBS may be used as the commit layer, which may involve two-phase commit should any data be replicated.

How a commit protocol communicates the *hist*(S) to the COLOR system is implementation dependent. However we note that the logical state representation of the global S may not match the storage model of S as the latter often involves a different normalized schema. Such an "O-R" mapping is very common in client-server based applications, and cloud-based applications are not immune to this problem however we devise the client-to-cloud abstraction.

D. Application Concerns

As described in the system model, it's conceivable that applications may choose to rely on server-side sessions to handle the above algorithm as well as to manage related client-side state. In such a case the role of a physical client will be limited to purely user-interface (UI) on which S_c is rendered and user inputs are collected. Actual client messages are generated from server-side sessions too.

[1]Current Affiliation: Google Inc.

Overall, applications are free to choose between 1) a purely REST based design that requires very limited intelligence from the client, and 2) a rich-client design that involves explicit client-side cache management. Currently we are in the process to standardize 1) as HTTP extensions.

V. TUNABLE PERFORMANCE

When running one or multiple optimistic algorithms, performance may be tuned via different timing or ordering parameters. The online algorithms have defined three different categories. For category 1), the total ordering delay may be bounded with a timeout which will trigger local FIFO delivery along with a new round of master election as discussed in category 2). Similarly the master election may be terminated with a timeout before majority is reached. For category 3), the degree of partitioning may be tuned to meet different performance goals.

In the following we provide a much simplified analytic solution to show how the tradeoff between consistency and performance may be tuned using an example under the first category. The following formulas specify how inconsistency measured as probability of client restarts *Exp* may be expressed as a function of different performance parameters.

$$\mathrm{Exp}\{n\} = P_c \lambda_c d_{dis} = \lambda_c d^3_{\mathrm{dis}} / (r + d_{dis})(d_c + d_{dis}) \quad (1)$$

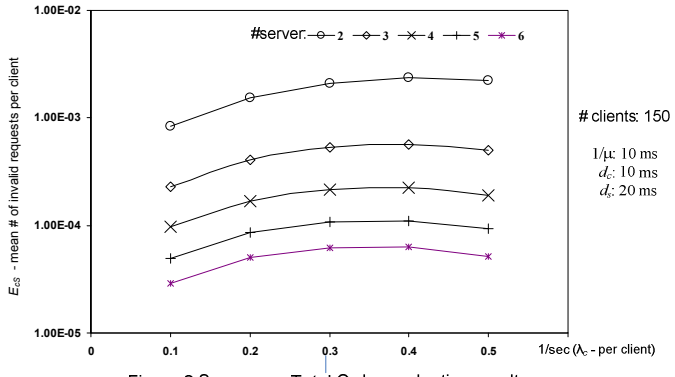

Figure 2 Sequencer-Total-Order evaluation result.

$$\mathrm{Exp}\{n\} = P_c[a/(n_c/n_s)] = (\lambda_c d^2_{dis}\ d_s\ n_s) / (r_T + d_{dis})(d_c + d_{dis}) \quad (2)$$

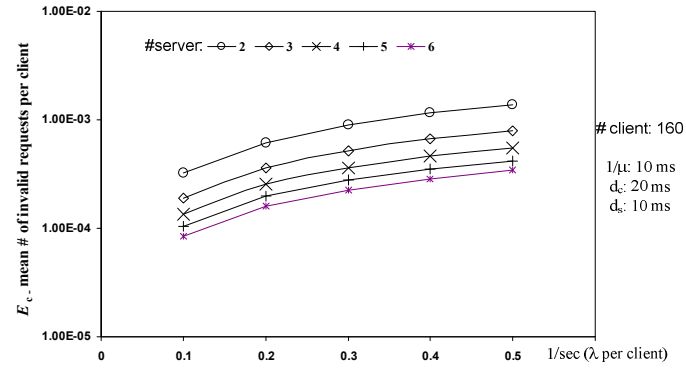

Figure 3. Token-Total-Order evaluation result.

In the above, optimistic delivery is enabled via either sequencer *(1)* or token *(2)* based non-uniform total ordering respectively, with *d* representing different communication delays, *r* the message processing delay, *n* the number of replicas and λ the arrival rate, and *c* representing the client, *s* the server, *dis* the client-server interaction, and in case of (2), *a* representing the mean batch size of a token.

The above example highlights several tunable parameters ranging from the delay of message delivery to the size of batch to the distribution of clients. The rate of inconsistency is measured as client restarts as only one partition is involved in this example.

In contrast, evaluation of partitioning based schemes are much more involved as the performance is decided by overlapped concurrent executions of optimistic algorithms. A canonical example is an online document model in which documents may be linked to each other. Concurrent edits will then cause links to break, which serves as an important consistency metrics.

Evaluation of early membership termination will have to rely on a state based approach such as Petri Net. The QoS guarantee will become a more important metrics rather than steady-state latency.

VI. REMAINING WORK

Category 3) algorithms need be more strictly defined. The current algorithms involve a rather complicated scheme to detect inconsistency and to garbage collect obsolete message histories. The algorithm overhead, especially the message size overhead, needs be minimized as applications may deal with 100's partitions at the same time.

While the evaluation work is well under way, more accurate performance models need be created, especially for the partitioning model (category 3) and the optimistic membership model (category 2). In both cases, simulation will be used as the primary means to evaluate the effectiveness of the solution.

The COLOR system model assumes a pluggable commit layer which is responsible for generating the global *hist*(S) prefix. The interface between local replicas and the commit layer needs be standardized to allow off-the-shelf database or storage solutions to be used as the commit layer.

Lastly there are several standard client-side caching frameworks that will enable COLOR to adopt a more fine-controlled recovery solution. For instance, a rich client may be able to track all the valid snapshot S states as well as its own local operation log. In a failure event, the client may be able to recover to a valid state by simply reloading a most recent version of S and reapplying its local operation log. In doing so transient inconsistency will not result in any action that is visible to end-users.

VII. CONTRIBUTIONS

This paper proposes a generalized solution to the widely known CAP and EC problems with a system model that may be easily adapted to different system architectures. The overall approach enables tunable performance with online conflict resolution and bounded inconsistency. A

[1]Current Affiliation: Google Inc.

standardized client-cloud interface is specified in order to enable quantifying the trade-off requirements between performance and consistency.

While many different optimistic algorithms have been studied before, COLOR represents a novel solution in the following areas:

- Transient fail-over scheduling is covered under the same system model that drives steady-state scheduling. Both scheduling are optimistic and are targeting the same real-time and QoS guarantees with mainly timer based parameters.
- Optimistic partial ordering is proposed as one of the main tunable parameters, and it offers a true, contiguous tunable space that applications or system operators can finally control and measure.
- The choice of a state representation based system model decouples state-machine based replication semantics from the consistency requirements. This allows any existing replication or multicast algorithms to be integrated into the same framework.

The final goal of COLOR is to create a proven solution that actually works in a real cloud environment. To achieve this goal, the COLOR system model has taken into account requirements unique to the client-to-cloud computing environment while retaining generality at the same time.

REFERENCES

[1] Terry, D., Theimer, M., Petersen, K., Demers, A., Spreitzer, M., Hauser, C.: Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. P. ACM SIGOPS Operating Systems Review vol. 29, iss. 5 (1995)

[2] Lynch, N., Gilbert S.: Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. P. ACM SIGACT News vol. 33, iss. 2 (2002)

[3] Yu, H., Vahdat, V.: Design and Evaluation of a Conit-Based Continuous Consistency Model for Replicated Services. J. ACM Transactions on Computer Systems vol. 20, iss. 3 (2002)

[4] Felber, P., Schiper, A.: Optimistic Active Replication. In: ICDCS-21 Proceedings of the 21st International Conference on Distributed Computing Systems, pp. 333-341 (2001)

[5] Fielding, R., Taylor, R.: Principled Design of the Modern Web Architecture. J. ACM Transactions on Internet Technology (TOIT) vol. 2-2, pp. 115–150 (2002)

[6] Lamport, L.: Generalized Consensus and Paxos. In: Microsoft Research Technical Report MSR-TR-2005-33 (2005)

[7] Pedone, P., Schiper, A.: Generic Broadcast. In: DISC-99 13th. Intl. Symposium on Distributed Computing, pp. 94-108 (1999)

[8] Karamanolis, C., Magee, C.: Client Access Protocols for Replicated Service. J. IEEE Transaction on Software Engineering vol. 25-1, pp. 3-22 (1999)

[9] Pedone, F., Schiper, A.: Optimistic Atomic Broadcast. J. Theoretical Computer Science - Special issue: Distributed Computing Archive vol. 291, iss. 1 (2003)

[10] Krishnamurthy, S., Sanders, W., Cukier, M.: Performance Evaluation of a QoS-Aware Framework for Providing Tunable Consistency and Timeliness. In: Proceedings of the International Workship on Quality of Service (2002)

[11] Defago, X., Schiper, A., Urban P.: Totally ordered broadcast and multicast algorithm: A comprehensive survey. In: ACM Computing Surveys vol. 36, iss. 4 (2004)

[12] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's Highly Available Key-value Store. In: Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (2007)

[1]Current Affiliation: Google Inc.