P2P Logging and Timestamping for Reconciliation

Mounir Tlili, W. Kokou Dedzoe, Esther Pacitti, Patrick Valduriez
Atlas team, INRIA and LINA, University of Nantes, France
{FirstName.LastName@univ-nantes.fr, Patrick.Valduriez@inria.fr}

Reza Akbarinia
University of Waterloo, Canada
rakbarin@cs.uwaterloo.ca

Pascal Molli, Gérôme Canals
ECOO team, LORIA-INRIA
{FirstName.LastName}@loria.fr

Stéphane Laurière
XWiki, France
slauriere@xwiki.com

ABSTRACT

In this paper, we address data reconciliation in peer-to-peer (P2P) collaborative applications. We propose P2P-LTR (Logging and Timestamping for Reconciliation) which provides P2P logging and timestamping services for P2P reconciliation over a distributed hash table (DHT). While updating at collaborating peers, updates are timestamped and stored in a highly available P2P log. During reconciliation, these updates are retrieved in total order to enforce eventual consistency. In this paper, we first give an overview of P2P-LTR with its model and its main procedures. We then present our prototype used to validate P2P-LTR. To demonstrate P2P-LTR, we propose several scenarios that test our solutions and measure performance. In particular, we demonstrate how P2P-LTR handles the dynamic behavior of peers with respect to the DHT.

1. INTRODUCTION

Collaborative applications are getting common as a result of rapid progress in distributed technologies (grid, P2P, and mobile computing). Constructing these applications on top of P2P networks has many advantages which stem from P2P properties: decentralization, self-organization, scalability and fault-tolerance. As an example of such application, consider a second generation wiki such as XWiki [12, 13] that works over a P2P network and enables users to edit, add, and delete Web documents.

In a collaborative application, many users frequently need to access and update information even if they are disconnected from the network, e.g. in a train or another environment that does not provide good network connection. This requires that users hold local replicas of shared documents. However, a collaborative application requires optimistic multi-master replication to assure data availability at anytime.

Optimistic replication is largely used as a solution to provide data availability for these applications. It allows asynchronous updating of replicas so that applications can progress even though some nodes are disconnected or are under failed. This enables asynchronous collaboration among users. However, concurrent updates cause replica divergence and conflicts, which should be reconciled. In most existing solutions [7, 9], timestamp reconciliations are not well adapted to peers' dynamicity (peers may join and leave the network at anytime). Some semantic reconciliation engines are implemented in a single node (reconciler node), which may introduce bottlenecks [8, 4] and single point of failures. Thus, we choose to explore P2P reconciliation. We focus on timestamped P2P reconciliation. The challenge consists of providing a distributed (P2P) highly available structure supporting multi-master reconciliation and eventual consistency in the presence of dynamicity and concurrent updates on the same document, which is a typical case in P2P collaborative applications.

In this paper we present P2P-LTR, a fully distributed P2P structure over a DHT that provides the following services: a timestamp service based on KTS [1], a highly available log service (P2P-Log) storing timestamped updates, and a retrieval algorithm getting the timestamped updates in total order. Our main goal is to provide eventual consistency in the presence of dynamicity and failures. This approach is generic and could be used by any reconciliation engine. In this paper, we consider a general P2P text edition context such as XWiki.

To validate P2P-LTR we implemented it using OpenChord [11, 6]. Next, we implemented a prototype to create specific scenarios to test and validate P2P-LTR. For instance, we may specify the number of peers or network latencies, or may provoke failures. We use our prototype to check the correctness and response times of P2P-LTR. In our demonstration, we show how P2P-LTR generates timestamps in a fully P2P and continuous manner, managing concurrent updates. Then, we demonstrate how the P2P-Log works to provide high availability of updates in the DHT. Next, we demonstrate the retrieval algorithm that gets timestamped updates from the P2P-Log in total order. We issue several simultaneous updates coming from different peers and show that P2P-LTR manages concurrency correctly, and provides eventual consistency. Finally, we show how P2P-LTR deals with peer s' dynamicity and failures.

The rest of this paper is organized as follows: In Section 2, we describe our general P2P-LTR model and the main concepts. In Section 3, we present P2P-LTR's main procedures and summarize P2P-LTR functionalities. In Section 4, we describe our prototype. We present the main demonstration features in Section 5. Finally, Section 6 concludes.

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-306-8/08/08

2. P2P-LTR MODEL

In this section, we present our P2P-LTR model and the main concepts of our approach. In our model, we consider five types of peers (see Figure 1):

User Peer: implements the user application (denoted by *u*) that holds primary copies (in our case, documents). Tentative update actions performed by users on primary copies are captured after each document save operation (see Figure 2). These updates are wrapped together in the form of a *patch* (a sequence of updates). A *tentative patch* is afterwards timsetamped in continuous timestamp order by interacting with its corresponding Master-key. Based on this, each patch is executed in the timestamp order at each involved user peer (masters of the same document), to assure eventual consistency. To assure total order, continuous timestamped patches are stored at the P2P Log and users may retrieve them at specific Log-Peers for reconciliation.

Dynamic Master-key Peer: responsible for generating *continuous timestamps* for a document: each new patch of a document in the DHT has a timestamp (denoted by *ts*), which is exactly one unit greater than the timestamp, say *ts'*, of the previous patch on the same document, i.e. $ts = ts' + 1$. Each document is identified by a *key* value. Using this key, the user peer locates the Master-key, by hashing the name of the document using a specific hash function h_t. When a new timestamp is generated, the Master-key *publishes* the timestamped patch in the P2P-Log at specific Log-Peers. For this, the Master-key peers must first have a set of pairwise independent hash functions $H_r = \{h_1, h_2, \ldots, h_n\}$ which we call *replication hash functions*, used for implementing patch replication in the DHT.

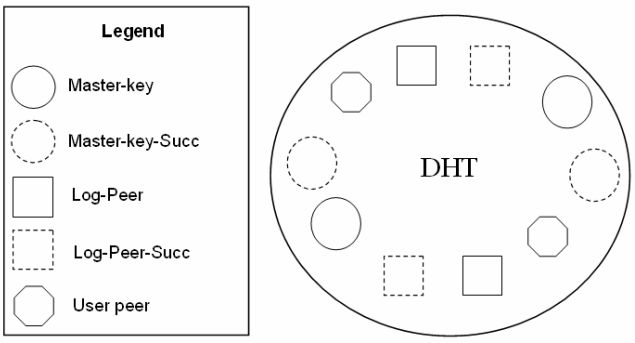

Figure 1. Components peers of P2P-LTR

For a given key, the Master-key peer assumes the responsibility of sustaining the last timestamp value (denoted by *last-ts*) value and mediating between concurrent updates.

Master-key Succ: replaces the Master-Key in case of crashes.

Log-Peer: peer that is responsible for holding a timestamped patch done on a replica (document). A patch is replicated by a Master-key peer by performing: $Put(h_1(key+ts),Patch)$, $Put(h_2(key+ts),Patch) \ldots Put(h_n(key+ts),Patch)$.

Log-Peers-Succ: replaces the Log-Peers in case of crashes.

Our network model is semi-synchronous, similar to the ones proposed in [3, 1, 4].

3. P2P-LTR PROCEDURES

In this section, we summarize the main procedures of P2P-LTR: patch timestamp validation, patch replication and patch retrieval.

In our model, each user peer (e.g. running locally the XWiki application) has a local primary copy of the document (e.g. XWiki document, see Figure 2). Thus a user u_1 may work asynchronously. When she modifies a specific document *d*, the generated patch is considered as a tentative patch because its timestamp number is still not validated. The validation procedure consists of providing a *continuous timestamp* value to the new patch considering concurrent updates on the same document *d*, performed by other users (master of the same document). Recall that since patch generation may be done concurrently, it may happen that an user generates new tentative patche without knowing that previous validated patches on the same document *d* are available at the P2P-Log. The *patch timestamp validation procedure* is done by contacting the Master-key of *d.*

To handle validation, at each peer, each document has an associated local timestamp value (denoted by *ts*). Recall that the Master-key holds the last timestamp (denoted by *last-ts*) provided for any peer of the same document. Thus, for a given document, the user peer u_1 first contacts the corresponding Master-key and asks it to publish the patch with the timestamp value *ts* by invoking $put(h_t(key), patch+ts)$, where h_t is the *timestamp hash function* used to locate Master-key peers wrt. to a specific key (i.e. document). If the Master-key local timestamp value *(last-ts)* is equal to *ts*, then the Master-key increments by one *last-ts* value by using *gen_ts(key)*, and confirms the user peer u_1 that it will trigger the *patch replication procedure*. Next, the Master-key replicates the patch in the P2P-Log (at the Log-Peers) by invoking *sendToPublish(key, last-ts, patch)* and acknowledges u_1, with a message containing the validated timestamp value.

If the Master-key local timestamp value (*last-ts*) is greater than *ts*, that means that there are previous validated patches available in the P2P-Log, generated by other users, that must be integrated in u_1's document *d* before (e.g. for instance by using So6 [10] reconciliation engine which is based on operational transformation [5]). To accomplish this, u_1 must perform the retrieval procedure to get all missing patches in continuous timestamp order, by using $get(h_i(key+ts))$, where h_i is one of the replication hash functions. Afterwards, u_1 restarts the *timestamp validation* procedure again until *last-ts* value is equal to *ts* value.

To manage concurrent patch timestamp validation on the same document, the corresponding Master-key serves each user peer sequentially. That is, a new timestamp *ts* value for a given document *d* is provided after the replication of the previous timestamped *(ts-1)* patch on *d*.

Each Master-key Peer provides three main operations for patch management:

- ***gen_ts(key):*** given a key, generates an integrer number as a timestamp for key with two main properties: the timestamps generated by the Master-key peer have the *monotonicity* and *continuous timestamping* property, i.e. two timestamps generated for the same key are monotonically increasing and the difference between the timestamps of any two consecutive updates is one.

- ***last_ts(key):*** given a key, returns the last timestamp generated for key. The *last_ts* operation can be implemented like *gen_ts* except that *last_ts* is simpler: it only returns the value of timestamps and does not need to increase its value.
- ***sendToPublish(key, last-ts, patch):*** for each h in H_r it *puts* (replicates) the patch by using: $Put(h_1(key+ts),Patch),Put(h_2(key+ts),Patch)\ldots Put(h_n(key+ts),Patch)$ at the Log-Peers that are *rsp(key,h)*. In addition, it replicates the *last-ts* at the Master-Succ Peer.

To summarize, P2P-LTR is composed of the following three main procedures:

1. Edit a page locally (produces a tentative patch)
2. Validate the tentative patch timestamp value (considering other updaters) and retrieve patches if necessary.
3. After timestamp validation, replicate the new patch at the P2P-Log.

Figure 2. XWiki Document example in editing mode

4. PROTOTYPE

In this section, we describe the prototype used to validate P2P-LTR main procedures.

The current implementation of the prototype is based on Open Chord which is an open source implementation of the Chord protocol. Open Chord is distributed under the GNU General Public License (GPL). It provides all DHT functionalities which are needed for implementing P2P-LTR, e.g. *lookup*, *get* and *put* functions. We implemented our own *successor management* and *stabilization* protocols on top of Open Chord to handle peers dynamicity and failures wrt. to P2P-LTR, since the ones proposed by Open chord are not suited to P2P-LTR.

In our prototype, peers are implemented as Java objects. They can be deployed over a single machine or several machines connected together via a network. Each object contains the code which is needed for implementing P2P-LTR services. To communicate between peers, we use Java RMI [2] which allows an object to invoke a method on a remote object.

The prototype provides a GUI that enables the user to manage the DHT network (e.g. create the DHT, add/remove peers to/from the system, etc.), store/retrieve data in/from the DHT, monitor the data stored at each peer, the keys for which the peer has generated a timestamp, etc. (see Figure 3).

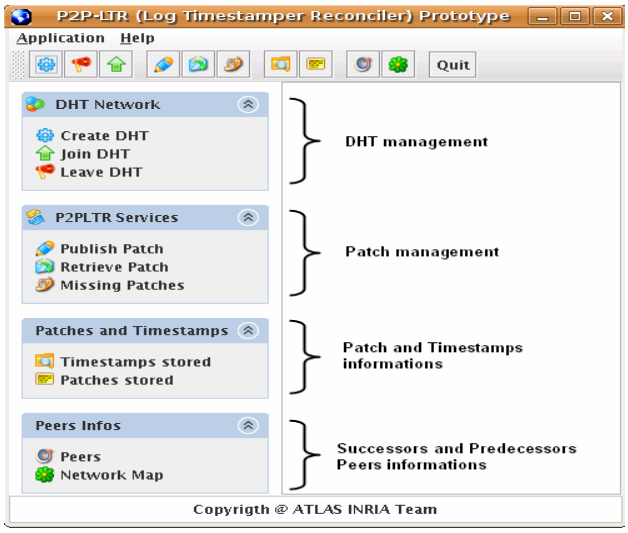

Figure 3. P2P-LTR Main Interface

5. DEMONSTRATION SCENARIOS

The key features of P2P-LTR, are demonstrated through the following scenarios:

Timestamp generation. This scenario is used to show that the responsibility for the continuous timestamp generation is distributed over all peers of the DHT, i.e. each Master-key peer is responsible for timestamping a subset of the documents, (see Figure 4).

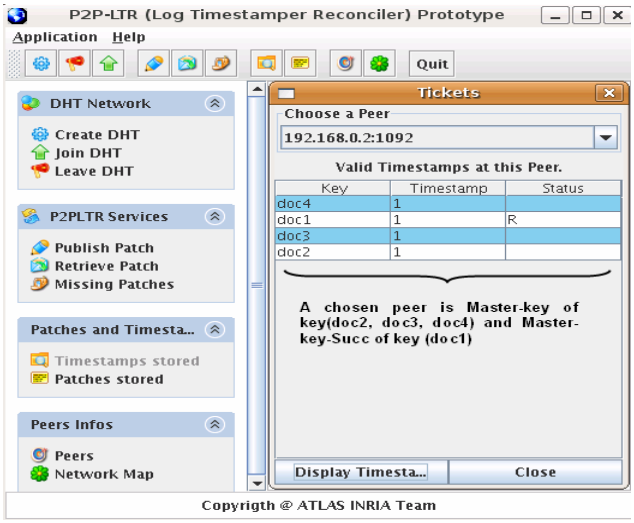

Figure 4. Set valid timestamps at a chosen Master Peer

Concurrent patch publishing. This scenario is used to show that P2P-LTR manages correctly concurrent patch publishing on a same document. For this, we submit concurrent patches for a document coming from different users and show that eventual consistency is assured. Figure 5 shows that when a peer performs

the retrieval procedure in the presence of other updaters, it retrieves continuous timestamp patches.

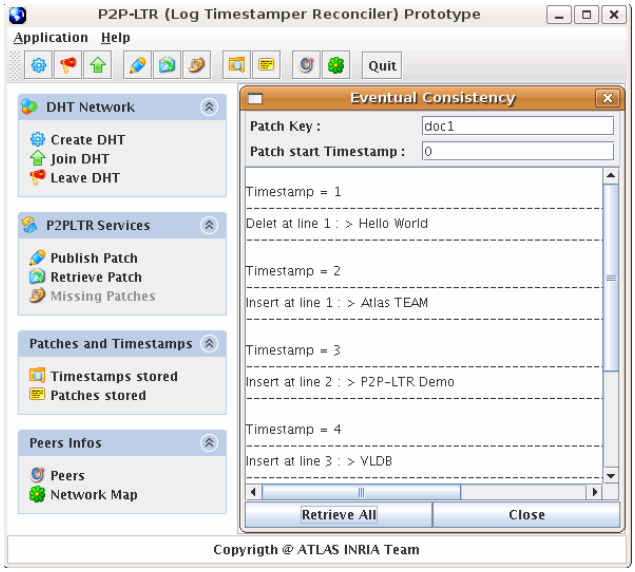

Figure 5. Missing patches retrieval on total order

Master-key peer departures. In this scenario, we focus on the cases where a Master-key peer leaves the system normally or as a result of a failure. In this case the leaving peer triggers DHT destabilization which yields P2P-LTR to manage stabilization in order to assure correctness. We first demonstrate that when a Master-key peer leaves the system normally P2P-LTR transfers its key and timestamps to its Master-Succ peer. To do this, we show that a new pair Master-key and Master-key-succ is established correctly. Using our prototype, we show that the set of keys and timestamp values related to the Master-key that left the DHT are correctly inserted into its successor peer. We also demonstrate the cases where the Master-key peer fails. We show that P2P-LTR assures that its successor takes over correctly, assuring continuous timestamps for the key.

New Master-key peer joining. This scenario focuses on the cases where a new peer joins the system and becomes a Master-key peer for certain keys. In this case, the joining peer triggers DHT destabilization. P2P-LTR assures that the old responsible transfers its keys and timestamps to the new Master-key, without violating eventual consistency.

6. CONCLUSION

In this paper, we presented P2P-LTR which provides P2P logging and timestamping services for P2P reconciliation over a distributed hash table (DHT). To validate P2P-LTR, we developed a prototype and several scenarios that test our solutions and measure performance. In addition, we demonstrate our implementation solutions over a DHT to manage some challenging scenarios related to peers' dynamicity and failures. Through our prototype, we show that P2P-LTR behaves correctly and assures eventual consistency despite peers' dynamicity and failures. We are currently integrating P2P-LTR with XWiki [12, 13] using a So6 variant as text reconciliation engine.

7. ACKNOWLEDGEMENTS

This work is partially funded by French ANR XWiki Concerto project and European STREP Grid4All project.

8. REFERENCES

[1] R. Akbarinia, E. Pacitti, P. Valduriez. Data Currency in Replicated DHTs. *ACM SIGMOD Int. Conf. on Management of Data*, 211-222, 2007.

[2] Java RMI. http://java.sun.com.

[3] P. Linga et. Al.: Guaranteeing Correctness and Availability *in P2P Range Indices, ACM SIGMOD Int. Conf. on Management of Data*, USA, 323-334, 2005.

[4] V. Martins and E. Pacitti. Dynamic and distributed reconciliation in P2P-DHT networks. *European Conf. on Parallel Computing (Euro-Par)*, pages 337-349, 2006.

[5] P. Molli, G. Oster, H. Skaf-Molli, A. Imine. Using the transformational approach to build a safe and generic data synchronizer. *ACM SIGGROUP Conference on Supporting Group Work, (GROUP)*, 212-220, 2003.

[6] Open Chord version 1.0.2 User's Manual. http://www.uni-bamberg.de

[7] E. Pacitti, C. Coulon, P. Valduriez, T. Özsu . Preventive Replication in a Database Cluster. *Distributed and Parallel Databases, Vol. 18, No. 3, 2005, 223-251.*

[8] N. Preguiça, M. Shapiro, and C. Matheson. Semantics-based reconciliation for collaborative and mobile environments. In *Proc. of the Int. Conf. on Cooperative Information Systems (CoopIS)*, 38-55, 2003.

[9] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42-81, 2005.

[10] So6: http://dev.libresource.org

[11] I. Stoica, R. Morris, D.R. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup service for internet applications. In *Proc. of the ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 149-160, 2001.

[12] XWiki Concerto: http://concerto.xwiki.com

[13] XWiki: http://www.xwiki.org