Dynamic and Distributed Reconciliation in P2P-DHT Networks *

Vidal Martins[1,2] and Esther Pacitti[1]

[1] ATLAS Group, INRIA and LINA, University of Nantes, France
[2] PPGIA/PUCPR - Pontifical Catholic University of Paraná, Brazil
Firstname.Lastname@univ-nantes.fr

Abstract. Optimistic replication can provide high data availability for collaborative applications in large scale distributed systems (grid, P2P, and mobile systems). However, if data reconciliation is performed by a single node, data availability remains an important issue since the reconciler node can fail. Thus, reconciliation should also be distributed and reconciliation data should be replicated. We have previously proposed the *DSR-cluster* algorithm, a distributed version of the IceCube semantic reconciliation engine designed for cluster networks. However *DSR-cluster* is not suitable for P2P networks, which are usually built on top of the Internet. In this case, network costs must be considered. The main contribution of this paper is the *DSR-P2P* algorithm, a distributed reconciliation algorithm designed for P2P networks. We first propose a P2P-DHT cost model for computing communication costs in a DHT overlay network. Second, taking into account this model, we propose a cost model for computing the cost of each reconciliation step. Third, we propose an algorithm that dynamically selects the best nodes for each reconciliation step. Our algorithm yields high data availability with acceptable performance and limited overhead.

1 Introduction

Large-scale distributed collaborative applications are getting common as a result of rapid progress in distributed technologies (grid, P2P, and mobile computing). Consider a professional community whose members wish to elaborate, improve and maintain an on-line virtual document, e.g. notes on classical literature or common bibliography, supported by a P2P system. They should be able to read and write application data. In addition, user nodes may join and leave the network whenever they wish, thus hurting data availability.

Optimistic replication is largely used as a solution to provide data availability for these applications. It allows asynchronous updating of replicas such that applications can progress even though some nodes are disconnected or have failed. This enables asynchronous collaboration among users. However, concurrent updates may cause replica divergence and conflicts, which should be reconciled. In most existing solutions [11, 13] reconciliation is typically performed by a single node (reconciler node) which may introduce bottlenecks. In addition, if the reconciler node fails, the entire replication system may become unavailable.

* Work partially funded by the ARA Massive Data Project.

In [9], we proposed the *DSR-cluster* algorithm (Distributed Semantic Reconciliation for cluster), a distributed version of the semantic reconciliation engine of IceCube [6, 11] for cluster networks. Tentative *actions*, stored at action logs, are reconciled using *constraints*. Other reconciliation objects, such as *clusters*, are also necessary to produce the global schedule. *DSR-cluster* avoids bottlenecks, speeds up large scale reconciliation, and provides high data availability in case of node failures during reconciliation for cluster networks. In addition, *DSR-cluster* employs a distributed approach for storing reconciliation objects (*actions*, *clusters*, *constraints*, etc.) using a distributed hash table (DHT) [12, 14] in order to provide high data availability.

DSR-cluster proceeds in 5 distributed reconciliation steps. However, it does not take into account network costs during these steps. A fundamental assumption behind *DSR-cluster* is that the communication costs among cluster nodes are negligible. This assumption is not appropriate for P2P systems, which are usually built on top of the Internet. In this case, network costs may vary significantly from node to node and have a strong impact on the performance of reconciliation. Thus, network costs should be considered to perform reconciliation efficiently and to avoid network overload due to the communication with far distant nodes.

In this paper, we propose the *DSR-P2P* algorithm, a distributed reconciliation algorithm designed for P2P networks. The main contributions of this paper are: (1) a DHT cost model for computing communication costs of a P2P network using a DHT overlay network; (2) the DSR-P2P cost model for computing the cost of each reconciliation step based on DHT cost model; (3) the *DSR-P2P* algorithm for selecting the best reconciler nodes based on the DSR-P2P cost model (4); and experimental results that show that our cost-based approach yields high data availability with acceptable performance and limited overhead.

The rest of this paper is organized as follows. Section 2 describes the basis of the *DSR-P2P* semantic reconciliation solution for P2P networks. Section 3 introduces the DHT cost model. Section 4 describes the DSR-P2P cost model and the dynamic allocation algorithm for selecting the best reconciler nodes. Section 5 shows implementation and experimental results. Section 6 compares our work with the most relevant related works. Finally, Section 7 concludes this paper.

2 P2P Distributed Semantic Reconciliation

In this section, we describe the main terms and assumptions we consider for DSR-P2P followed by the main DSR-P2P algorithm itself.

We assume that DSR-P2P is used in the context of a virtual community which requires a high level of collaboration and relies on a reasonable number of nodes (typically hundreds or even thousands of interacting users) [15]. The P2P network we consider consists of a set of nodes which are organized as a distributed hash table (DHT) [12, 14]. A DHT provides a hash table abstraction over multiple computer nodes. Data placement in the DHT is determined by a hash function which maps data identifiers into nodes.

In our solution, a *replica* R is a copy of a collection of objects (e.g. copy of a relational table, or an XML document). A *replica item* is an object belonging to a replica (e.g. a tuple in a relational table, or an element in an XML document). We assume *multi-master* replication, i.e. a replica R is stored in several nodes and all nodes may read or write R. Conflicting updates are expected, but with low frequency.

In order to update replicas, nodes produce *tentative* actions (henceforth actions) that are executed only if they conform to the application semantics. An *action* is defined by the application programmer and represents an application-specific operation (e.g. a write operation on a file or document, or a database transaction). The application semantics is described by means of constraints between actions. A *constraint* is the formal representation of an application invariant (e.g. an update cannot follow a delete).

On the one hand, users and applications can create constraints between actions to make their intents explicit (they are called *user-defined constraints*). On the other hand, the reconciler node identifies conflicting actions, and asks the application if these actions may be executed together in any order (*commutative* actions) or if they are mutually dependent. New constraints are created to represent semantic dependencies between conflicting actions (they are called *system-defined constraints*).

A *cluster* is a set of actions related by constraints, and a *schedule* is a list of ordered actions that do not violate constraints.

With DSR-P2P, data replication proceeds basically as follows. First, nodes execute local actions to update replicas while respecting user-defined constraints. Then, these actions (with the associated constraints) are stored in the DHT using the replica identifier as key. Finally, reconciler nodes retrieve actions and constraints from the DHT and produce a global schedule, by performing conflict resolution in 6 distributed steps based on the application semantics. This schedule is locally executed at every node, thereby assuring eventual consistency [11]. The replicated data is eventually consistent if, when all nodes stop the production of new actions, all nodes will eventually reach the same value in their local replicas.

In order to avoid communication overhead and due to dynamic connections and disconnections, we distinguish *replica nodes*, which are the nodes that hold replicas, from *reconciler nodes*, which is a subset of the replica nodes that participate in distributed reconciliation.

We now present DSR-P2P in more details. First, we introduce the reconciliation objects necessary to DSR-P2P. Then, we present the six steps of the DSR-P2P algorithm.

2.1 Reconciliation Objects

Data managed by DSR-P2P during reconciliation are held by *reconciliation objects* that are stored in the DHT giving the object identifier. To enable the storage and retrieval of reconciliation objects, each reconciliation object has a unique identifier. DSR-P2P uses six reconciliation objects:

- **Communication costs (noted *CC*):** it stores the communication costs to execute each DSR-P2P step, estimated by every replica node, and used to choose reconcilers before starting reconciliation. These costs are computed in terms of latency times.
- **Action log *R* (noted L_R):** it holds all actions that try to update the replica *R*.
- **Action groups of *R* (noted G_R):** actions that manage a common replica item are put together into the same action group in order to enable the parallel checking of semantic conflicts among actions (each action group can be checked independently of the others); every replica *R* may have a set of action groups, which are stored in the *action groups of R* reconciliation object.
- **Clusters set (noted *CS*):** all clusters produced during reconciliation are included in the *clusters set* reconciliation object; a cluster is not associated with a replica.
- **Action summary (noted *AS*):** it comprises constraints and action memberships (an action is a *member* of one or more clusters).
- **Schedule (noted *S*):** it is a list of ordered actions.

The node that holds a reconciliation object is called the *provider node* for that object (e.g. *cost provider* is the node that currently holds *CC*). Provider data are guaranteed to be available using known DHT replication solutions [7]. DSR-P2P's liveness relies on the DHT liveness.

2.2 DSR-P2P Algorithm

DSR-P2P executes reconciliation in 6 distributed steps as showed in Figure 1.

- **Step 1 node allocation:** a subset of connected replica nodes is selected to proceed as reconciler nodes.
- **Step 2 actions grouping:** for each replica *R*, reconcilers put actions that try to update common replica items of *R* into the same group, thereby producing G_R.
- **Step 3 clusters creation:** reconcilers split action groups into clusters of semantically dependent conflicting actions (actions that the application judge safe to execute together, in any order, are semantically independent, even if they update a common replica item); clusters produced in this step are stored in the clusters set, and the associated action memberships are included in the action summary.
- **Step 4 clusters extension:** user-defined constraints are not taken into account in clusters creation; thus, in this step, reconcilers extend clusters by adding to them new conflicting actions, according to user-defined constraints; the associated action memberships are also included in the action summary.
- **Step 5 clusters integration:** clusters extensions lead to clusters overlappings (an overlap occurs when different clusters have common actions, and this is identified by analyzing action memberships); in this step, reconcilers bring together overlapping clusters, thereby producing integrated clusters.

– **Step 6 clusters ordering:** in this step, reconcilers produce the global schedule by ordering actions of integrated clusters; all replica nodes execute this schedule.

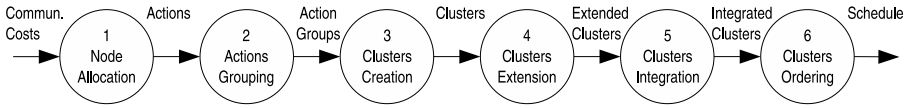


Fig. 1. DSR-P2P Steps

At every step, the DSR-P2P algorithm takes advantage of data parallelism, i.e. several nodes perform simultaneously independent activities on a distinct subset of actions (e.g. ordering of different clusters). No centralized criterion is applied to partition actions. In fact, whenever a set of reconciler nodes request data to a provider, the provider node naively supplies reconcilers with about the same amount of data (the provider node knows the maximal number of reconcilers because it receives this information from the node that launches reconciliation).

3 DHT Cost Model

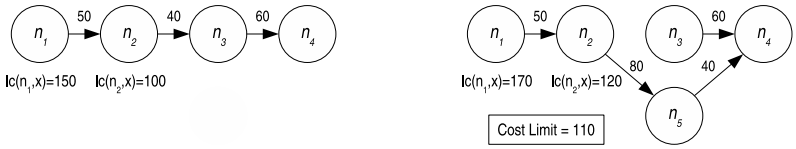
In this section, we propose a basic cost model for computing communication costs in DHTs. On top of it, we can build customized cost models (e.g. in the next section we elaborate a customized cost model for selecting DSR-P2P reconciler nodes).

In our model, we define communication costs (henceforth costs) in terms of latency times. We assume links with variable latencies and constant bandwidths. We intend to consider variable bandwidths in a future work.

Most DHT data access operations consist of a lookup, for finding the address of the node n that holds the requested information, followed by direct communication with n [5]. In the lookup step, several hops may be performed according to nodes' neighborhoods. Therefore, our DHT cost model relies on two metrics: lookup cost and direct cost. The *lookup cost*, noted *lc(n, id)*, is the latency time spent in a lookup operation launched by node n to find the data item identified by *id*. Similarly, *direct cost*, noted $dc(n_i, n_j)$, is the latency time spent by node n_i to directly access n_j.

Node n could easily compute the **lookup cost** *lc(n, id)* by executing the lookup operation and measuring the associated time. However, this approach overloads the node that replies the lookup operation as it receives a lot of lookup messages. Furthermore, the network is overloaded. To avoid these problems, we propose that each node computes its lookup costs by taking advantage of cost information held by its neighbors. We illustrate this solution with an example. In Figure 2a, let n_4 be a node that replies lookup operations searching for $id{=}x$; let arrows indicate the route of a lookup operation (e.g. if n_2 looks for x it makes this route: $n_2 \rightarrow n_3 \rightarrow n_4$); let a number over an arrow be the latency between the associated nodes. In this example, the lookup cost $lc(n_2, x)$ is 100 (i.e. 40 + 60), and $lc(n_1, x)$ is 150 (i.e. 50 + 40 + 60). Instead of executing the lookup

operation to compute *lc(n1, x)*, n_1 can ask n_2 for *lc(n2, x)* and add to this cost the latency between n_1 and n_2 (i.e. *lc(n1, x)* = *lc(n2, x)* + 50). The advantage of this incremental approach is locality and to avoid network overload.



(a) Before joining (b) After joining

Fig. 2. Computing Lookup Costs

Joins and leaves change the neighborhoods of nodes and, accordingly, the routes of lookup messages. As a result, lookup costs must be refreshed. However, we should avoid the refreshment at distant nodes to avoid network overload. To cope with this problem, we introduce two definitions: *cost limit* and *relevant joins and leaves*. *Cost limit* is the maximal acceptable cost for looking up an identifier (it can be a parameter or an adaptively computed value). A join or leave is *relevant* for a node n if it changes the cost for looking up an identifier in which n is interested, such that the old or the new lookup cost does not overtake *cost limit*. Thus, we propose that nodes refresh their lookup costs only in the presence of relevant joins and leaves. We illustrate this approach with an example. In Figure 2b, let cost limit be 110; and consider that n_5 joins the DHT of Figure 2a taking the place of n_3 in the route towards *id=x*. The join of n_5 is relevant only to n_2 as n_2 updates *lc(n2, x)* from 100 (a value that does not overtake cost limit) to 120. In contrast, the join of n_5 is not relevant to n_3 and n_4 since the associated lookup costs remain unchanged. This join is not relevant to n_1 either, because both, the old lookup cost (i.e. 150) and the new one (i.e. 170), overtake cost limit. Thus, n_1, n_3 and n_4 do not participate in the refresh operation.

We now present how we compute **direct cost**. Node n could easily compute the direct cost between n and the provider node for *id* (henceforth *home(id)*) by measuring the latency between n and *home(id)*. However, this approach may overload *home(id)*. To avoid this problem, we propose that nodes locally estimate direct costs. Two equivalent approaches may be used for this estimation: (1) for DHTs that do not rely on nodes' physical location for choosing nodes' neighbors, the latency between a node n and any other node can be estimated based on the latencies between n and its neighbors in the DHT; (2) for location-aware DHTs, where n's neighbors are supposed to be closer to n than other nodes, the same estimation can be made based on the latencies between n and some other nodes randomly selected from a bootstrap list (list of nodes that are likely connected). The advantage of the estimated approach is locality, and its drawback is lack of accuracy. In the performance evaluation we compare the estimated and exact approaches.

The *home(id)* may change due to joins and leaves. Thus, direct costs must also be refreshed. In our solution, *dc(n, home(id))* is refreshed at node n whenever *home(id)* changes and the associated lookup cost (i.e. *lc(n, id)*) is smaller than cost limit. To compute the refreshed value, we use the same strategy employed for computing the initial value. The principle of this approach is to avoid the execution of refreshment operations at far distant nodes, and its advantage is to avoid network overload.

4 DSR-P2P Node Allocation Algorithm

In this section, we present a dynamic distributed algorithm for allocating nodes to DSR-P2P steps using the DHT cost model. We first present the DSR-P2P cost model for each reconciliation step. Next, we describe how the cost provider node selects reconcilers based on DSR-P2P cost model. Finally, we present our approach for managing the dynamic behavior of DSR-P2P costs.

4.1 DSR-P2P Cost Model

The DSR-P2P cost model takes into account each reconciliation step defining a new metric: node step cost. A *node step cost*, noted *cost(i, n)*, is the sum of lookup and direct costs estimated by node n for executing step i of DSR-P2P algorithm. By analyzing the DSR-P2P behavior in terms of lookup and direct access operations at every step, we produced a cost formula for each step of DSR-P2P, which are showed in Table 1. There is no formula associated with step 1 because it is not performed by reconciler nodes.

Table 1. DSR-P2P Cost Model

i	***Cost(i,n)***
2	$lc(n, L_R) + 2dc(n, n_{L_R}) + lc(n, G_R) + dc(n, n_{G_R})$
3	$lc(n, G_R) + 3dc(n, n_{G_R}) + lc(n, CS) + 2dc(n, n_{CS}) + lc(n, AS) + dc(n, n_{AS})$
4	$2lc(n, AS) + 3dc(n, n_{AS}) + lc(n, CS) + 3dc(n, n_{CS})$
5	$lc(n, AS) + 3dc(n, n_{AS}) + lc(n, CS) + dc(n, n_{CS})$
6	$lc(n, CS) + 3dc(n, n_{CS}) + lc(n, AS) + 2dc(n, n_{AS}) + lc(n, S) + dc(n, n_S)$

As an example, let us explain *cost(2, n)*. In the second step of DSR-P2P (*i=2*), node n takes actions from the action log R (L_R) and produces the action groups of R (G_R). Thus, the first term in the associated formula ($lc(n,L_R)$) represents the lookup cost for finding L_R provider. The second term ($2dc(n,n_{L_R})$) corresponds to the direct costs for taking actions from L_R provider (request and reply). The third term ($lc(n,G_R)$) represents the lookup cost for finding G_R provider, and the last term ($dc(n,n_{G_R})$) corresponds to the direct cost for storing groups in G_R provider (only request). Similarly, all formulas can be explained.

4.2 Allocating Nodes

Node allocation is the first step of DSR-P2P algorithm. It aims to select for every succeeding step a set of reconciler nodes that can perform reconciliation

with good performance. In this subsection, we describe how reconciler nodes are chosen and we illustrate that with an example.

The cost provider, i.e. the node that currently holds the communication costs reconciliation object, is the node responsible for allocating reconcilers. The allocation works as follows. Replica nodes locally estimate the costs for executing every DSR-P2P step, according to the DSR-P2P cost model, and provide this information to cost provider. The node that starts reconciliation computes the maximal number of reconcilers per step (noted *maxRec*), as described in [10], and asks cost provider for allocating at most *maxRec* reconciler nodes per DSR-P2P step. As a result, the cost provider selects the best nodes for each step, and notifies these nodes about DSR-P2P steps they should execute.

In our solution, the cost management is parallel and independent of reconciliation. Moreover, it is network optimized since replica nodes do not send messages to cost provider, informing their estimated costs, if the node step costs overtake the *cost limit*. For these reasons, the cost provider does not become a bottleneck.

We now illustrate the allocation algorithm using an example. Table 2 shows the lookup and direct costs of our example, which were computed using a Chord DHT [14] with 4 connected nodes (i.e. n_0, n_1, n_4, and n_6). In a DHT, a node that is close to a reconciliation object (e.g. n_0 is close to AS (*id=1*)) may be far distant of others (e.g. n_0 is far distant of L_R (*id=5*)). As a result, a node that is suitable for a DSR-P2P step may not be worth in other steps. For this reason, every DSR-P2P step has its own set of reconcilers.

Table 2. Lookup and direct costs based on the DHT cost model. Each column has the identifier of a reconciliation object (*id*) and the node that holds this object (*home(id)*). Reconciliation object identifiers are: *CS*-0, *AS*-1, L_R-5, G_R-6, *S*-7. Each cell provides a specific lookup or direct cost, e.g. the cell in the 1^{st} line and 3^{rd} column indicates that n_0 spends 148.8ms to lookup L_R (*id=5*) stored in n_6 whereas the cell in the 2^{nd} line and 3^{rd} column indicates that a direct access between n_0 and n_6 costs 81.8ms.

Node	**Cost Metric**	**Reconciliation Objects (*id* → *home(id)***				
		0 → n_0	**1** → n_1	**5** → n_6	**6** → n_6	**7** → n_0
n_0	**Lookup *id***	0	0	148.8	148.8	0
	Access *home(id)*	0	37.8	81.8	81.8	0
n_1	**Lookup *id***	132.0	0	116.8	116.8	132.0
	Access *home(id)*	37.8	0	66.0	66.0	37.8
n_4	**Lookup *id***	35.4	148.8	0	0	35.4
	Access *home(id)*	74.4	58.4	17.7	17.7	74.4
n_6	**Lookup *id***	0	163.6	0	0	0
	Access *home(id)*	81.8	66.0	0	0	81.8

Table 3 shows the estimated costs that the cost provider receives from the replica nodes. These costs are computed by applying on the DSR-P2P cost model (Table 1) the lookup and direct costs of the DHT cost model (Table 2). We show in bold the two less expensive costs associated with each DSR-P2P step. Thus, in our example, if the maximal number of reconcilers is 2, the cost provider selects as reconcilers for each DSR-P2P step the nodes of Table 3 whose costs

are in bold (i.e. $Step_2 = \{n_4, n_6\}, Step_3 = \{n_0, n_6\}, Step_4 = \{n_0, n_1\}, Step_5 = \{n_0, n_1\}, Step_6 = \{n_0, n_1\}$), and notifies its decision to these nodes.

Table 3. Node step costs associated with the DHT considered in Table 2.

Node	**DSR-P2P steps (i)**				
	2	**3**	**4**	**5**	**6**
n_0	543.0	**432.0**	**113.4**	**113.4**	**75.6**
n_1	431.6	522.4	**245.4**	**169.8**	**415.2**
n_4	**53.1**	444.5	731.4	433.8	634.0
n_6	**0**	**393.2**	770.6	443.4	622.8

4.3 Managing the Dynamic Behavior of DSR-P2P Costs

The costs estimated by replica nodes for executing DSR-P2P steps change as a result of disconnections and reconnections. To cope with this dynamic behavior and assure reliable cost estimations, a replica node n_i works as follows:

- **Initialization:** whenever n_i joins the system, n_i estimates its costs for executing every DSR-P2P step. If these costs do not overtake the cost limit, n_i supplies the cost provider with this information.
- **Refreshment:** while n_i is connected, the join or leave of another node n_j may invalidate n_i's estimated costs due to routing changes. Thus, if the join or leave of n_j is relevant to n_i, n_i recomputes its DSR-P2P estimated costs and refreshes them at the cost provider.
- **Termination:** when n_i leaves the system, if its DSR-P2P estimated costs are smaller than cost limit (i.e. the cost provider holds n_i's estimated costs), n_i notifies its departure to the cost provider.

5 Validation and Performance Evaluation

To validate and study the performance behavior of DSR-P2P, we implemented it and simulated the overlay P2P network based on Chord (we used SimJava [4] for simulations). In this section, we present our performance model and the experimental results.

The performance model takes into account the strategy for selecting reconciler nodes (noted *Allocation*), the action log size (i.e. the number of actions to be reconciled, noted *Nb-Actions*) based on IceCube setup, and the network topology based on BRITE [2]. We define three strategies for selecting reconcilers: random selection (RDM); cost-based selection using precise costs for direct communication (CB/P); and cost-based selection using estimated costs for direct communication (CB/E). A network topology is defined by its bandwidth (noted *Bandwidth*), the number of connected nodes (noted *Nb-Nodes*), the average latency among these nodes (noted *Avg-Latency*), and the associated standard deviation (noted *Sd-Latency*). Latency values follow a uniform distribution. We produced 3 network instances for every network topology definition. We also produced 3 action logs for each action log size. By combining action logs with

network instances, we generate several distinct reconciliation scenarios that avoid over fitted results. Table 4 describes the parameters of the performance model.

The first experiment (Figure 3a) studies the reconciliation performance with locally estimated direct costs (recall that this approach reduces network load and avoids the overload of provider nodes, but it is not precise). For this experiment, we defined 4 network topologies and produced 12 network instances that are different only wrt. latency parameters (all topologies have *Bandwidth* = 1Mbps and *Nb-Nodes* = 1024). We used 3 action logs with *Nb-Actions* = 1005. Figure 3a shows the reconciliation performance using precise costs (CB/P), estimated costs (CB/E), and random allocation (RDM). In 3 topologies, the cost-based approaches (i.e. CB/P and CB/E) are equivalent and more efficient than the random approach. In the best case, which corresponds to a real P2P network, the CB/P reduces the reconciliation time of RDM in 37% whereas CB/E provides a performance improvement of 30%. Due to the small difference between CB/P and CB/E (i.e. 7%), we consider the estimated approach worth to avoid overload problems. Notice that the experimental conditions (i.e. constant bandwidth and uniform distribution of latencies) are strongly promising for random selection. We can improve the performance of cost-based approaches by changing these conditions (i.e. by providing variable bandwidths and distributing latencies in a way that some nodes are very close to each other making up clusters of nodes).

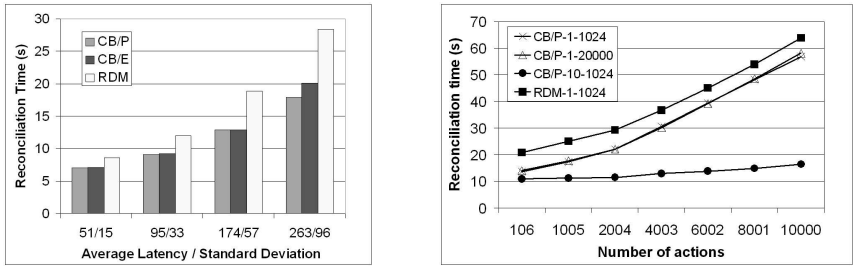
Table 4. Performance Parameters

Parameter	**Definition**	**Values**
Allocation	Strategy for selecting reconciler nodes	CB/P; CB/E; RDM
Nb-Actions	Number of actions to be reconciled	106 - 10000
Nb-Nodes	Number of connected nodes	1024; 20000
Bandwidth	Network bandwidth	1Mbps; 10Mbps
Avg-Latency	Average latency among nodes	51ms - 263ms
Sd-Latency	Standard deviation of network latency	15ms - 96ms

Due to the lack of space, we describe three additional experiments in a single graph, which corresponds to Figure 3b. The goal of these experiments is to show that the reconciliation time is improved because cost-based selection is used, and for faster network we have the best improvements compared with RDM. For instance, for a network of 10 Mbps and 1024 connected nodes using cost-based selection (CB/P-10-1024) we improved the random approach (RDM-1-1024) by a factor of 4. Notice that in this case both network bandwidths are different. For equal network bandwidths, the cost-based approach (CB/P-1-1024) still outperforms the random approach. Finally, increasing the number of connected nodes up to 20000 (CB/P-1-20000) does not degrade the DSR-P2P performance because it relies on a DHT and due to our allocation algorithm.

Liveness is an important issue in dynamic systems. DSR-P2P provides a greater degree of availability, scalability and fault-tolerance than the centralized solution. In contrast, since DSR-P2P depends on network communication, its reconciliation time (e.g. 57s for 10000 actions in a 1Mbps network with average

latency of 229ms) is worse than the centralized counterpart (e.g. about 3s for 10000 actions). However, 57s remains an acceptable time for reconciling 10000 actions in a P2P network. The centralized solution, although more efficient than DSR-P2P, is unsuitable for P2P networks due to its low availability in dynamic environments.



(a) Varying network latencies (b) Varying actions, nodes, band.

Fig. 3. DSR-P2P Reconciliation Time

6 Related Work

In the context of P2P networks, there has been little work on managing data replication in the presence of updates. Most of data sharing P2P networks consider the data they provide to be very static or even read-only. Freenet [3] partially addresses updates which are propagated from the updating peer downward to close peers that are connected. However, peers that are disconnected do not get updated. P-Grid [1] is a structured P2P network that exploits epidemic algorithms to address updates. It assumes that conflicts are rare and their resolution is not necessary in general. In addition, P-Grid assumes that probabilistic guarantees instead of strict consistency are sufficient. Moreover, it only considers updates at the file level in a single master-mode. In OceanStore [8] every update creates a new version of the data object. Consistency is achieved by a two-tiered architecture: a client sends an update to the object's *inner ring* (primary copies) and some secondary replicas in parallel. Once the update is committed, the *inner ring* multicasts the result of the update down the dissemination tree. OceanStore assumes an infrastructure comprised of servers that are connected by high-speed links. Different from the previous works, we propose to distribute the reconciliation engine in order to provide high availability. Our approach assures eventual consistency among replicas, which enables asynchronous collaboration among users. In addition, we provide multi-master replication and we do not assume servers linked by high-speed links.

7 Conclusion

In this paper, we proposed the DSR-P2P, a distributed algorithm for semantic reconciliation in P2P networks. Our main contributions are a cost model for computing communication costs in DHTs and an algorithm that takes into account these costs and the DSR-P2P steps to select the best reconciler nodes. For

computing communication costs, we use local information and we deal with the dynamic behavior of nodes. In addition, we limit the scope of event propagation (e.g. joins or leaves) in order to avoid network overload.

We validated DSR-P2P through implementation and simulation. The experimental results showed that our cost-based reconciliation outperforms the random approach by a factor of 30% over scenarios that are favorable for the random approach (constant bandwidth and uniform distribution of latencies). In addition, the number of connected nodes is not important to determine the reconciliation performance due to the DHT scalability and the fact that reconcilers are as close as possible to the reconciliation objects. Compared with the centralized solution, which is more efficient but lowly available, our algorithm yields high data availability with acceptable performance and limited overhead. As future work, we plan to include variable bandwidths in our cost model.

References

1. K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-grid: a self-organizing structured p2p system. *SIGMOD Rec.*, 32(3), 2003.
2. BRITE. http://www.cs.bu.edu/brite/.
3. I. Clarke, T.W. Hong, S.G. Miller, O. Sandberg, and B. Wiley. Protecting free expression online with freenet. *IEEE Internet Computing*, 6(1), 2002.
4. F. Howell and R. McNab. Simjava: a discrete event simulation library for java. In *Web-based Modeling and Simulation*, 1998.
5. R. Huebsch, J.M. Hellerstein, N. Lanham, I. Stoica, B.T. Loo, and S. Shenker. Querying the internet with pier. In *Proc. of VLDB Conference*, 2003.
6. A. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The icecube approach to the reconciliation of divergent replicas. In *Proc. of ACM PODC*, 2001.
7. P. Knezevic, A. Wombacher, and T. Risse. Enabling high data availability in a dht. In *Proc. of DEXA Workshops*, 2005.
8. J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of ACM ASPLOS*, 2000.
9. V. Martins, E. Pacitti, and P. Valduriez. Distributed semantic reconciliation of replicated data. In *Proc. of CDUR*. IEEE France and ACM SIGOPS France, 2005.
10. V. Martins, E. Pacitti, and P. Valduriez. A dynamic distributed algorithm for semantic reconciliation. In *Distributed Data and Structures 7 (WDAS)*, 2006.
11. N. Preguiça, M. Shapiro, and C. Matheson. Semantics-based reconciliation for collaborative and mobile environments. In *Proc. of IFCIS CoopIS*, 2003.
12. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proc. of ACM SIGCOMM*, 2001.
13. Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1), 2005.
14. I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of ACM SIGCOMM*, 2001.
15. S. Whittaker, E. Isaacs, and V. O'Day. Widening the net: workshop report on the theory and practice of physical and network communities. *ACM SIGCHI Bulletin*, 29(3), 1997.