

Maintaining Update Consistency in Replicated Peer-to-Peer Systems*

Minyoung Choi **Haengrae Cho**
Dept. of Computer Engineering
Yeungnam University
Gyungsan, Gyungbuk 712-749, South Korea

Dongha Lee
Dept. of IT SW Research Team
DGIST
Daegu 700-742, South Korea

Youngmin Park
Dept. of Multimedia Engineering
Kyungwoon University
Gumi, Gyungbuk 730-852, South Korea

Abstract

Peer-to-peer (P2P) systems have become a popular medium through which to share huge amounts of data. On the basis of network topology, P2P systems are divided into three types: centralized, structured distribution, and unstructured distribution. Unstructured P2P systems such as Gnutella are novel in the sense that they are extensible and reliable. However, as the number of participating nodes increases, unstructured P2P systems suffer from the high complexity of search operations that have to scan the network to find the required data items. Efficient replication of data items can reduce the complexity, but it introduces another problem of maintaining consistency among replicated data items when each data item could be updated. In this paper, we propose a new update propagation algorithm that propagates an updated data item to all of its replica. The proposed algorithm can reduce the message transfer overhead by adopting the notion of timestamp and hybrid push/pull messaging.

Keywords: peer-to-peer system, data replication, update propagation, consistency management, performance evaluation

*This research was supported by Korean Ministry of Information and Communication under the University IT Research Center program supervised by the IITA (Institute of Information Technology Assessment).

1 Introduction

Peer-to-peer (P2P) systems have become a popular medium through which to share huge amounts of data [1, 11]. They usually adopt a decentralized architecture to data sharing. By distributing data storage and processing across autonomous nodes in the network, P2P systems can scale without the need for powerful servers. Representative P2P systems such as Gnutella [5] and Kazaa [8] have millions of nodes sharing petabytes of data over Internet.

The complexity of a search operation in P2P systems would be very high since it may require scanning the entire network to find a required data item. Efficient replication of data items can reduce the complexity [3, 4, 6, 7, 9]. However, it introduces another problem of maintaining consistency among replicated data items when some data items are updated. Propagating updates consistently through P2P systems are challenging because nodes are autonomous and may be off-line frequently and that no global knowledge on the system exists. This is especially relevant for upcoming mobile environments.

In this paper, we propose a new update propagation algorithm that propagates an updated data item to its replica. The novel features of our algorithm are as follows.

- The algorithm is *purely decentralized*. Specifically, it is based on rumor spreading.

All updates must be eventually propagated to their replica, but each update propagation among nodes are performed in asynchronous manner. This is a great advantage in P2P systems that may experience transient failures and network congestion.

- The algorithm achieves *lower communication overhead*. It tries to reduce duplicate propagation of an update message to a node.
- The algorithm is *adaptive* to the dynamic behavior of P2P network. When most neighbor nodes leave the network, a node reconstructs the neighborhood with new nodes by itself. This contributes to fault-tolerance and fast update propagation.

The rest of this paper is organized as follows. Section 2 introduces related work and Section 3 describes the proposed algorithm in detail. Section 4 presents the experiment model and Section 5 summarizes the experiment results. Finally, Section 6 concludes the paper.

2 Related Work

There are two types of update propagation algorithms in P2P systems: *push* and *pull*. In the push-based algorithm [7], a new update is pushed by an initiator to neighbor nodes, which in turn propagate it to their neighbor nodes. In the pull-based algorithm [2], a node polls periodically to one of neighbors if there are any new updates. The push-based algorithm can provide good consistency guarantees for nodes that are online and reachable from the initiator. However, it has high communication overheads and is suitable only for static networks. The pull-based algorithm is better suited for dynamic networks but suffers from longer propagation delay and weaker consistency guarantees - the consistency guarantees in pull are critically dependent on the effectiveness of polling [9].

To combine the best features of push and pull, *hybrid push-pull* [3, 9] algorithms are proposed. In [9], each data item has a unique owner node. Updates on a data item can only be made by its owner. The owner pushes an invalidation message upon each update of a data item to its neighbors, who then propagate the message to their

neighbors. The hybrid technique requires nodes to occasionally poll the owner to check whether the data item was updated. However, the pull phase does not work when the owner leaves the network. A duplicate message propagation may also be happened at the push phase. The hybrid algorithm of [3] allows any node to be able to update a data item. Its notable features are two-fold. First, at the push phase, an update message includes a receiver list. The receiver list is a list of node identifiers to which the same message has been sent. By not forwarding the update message to any node in the receiver list, the algorithm can avoid duplicate message propagation. However, if the message is lost or a node in the receiver list is off-line at the push phase, the node cannot receive the message again. Next, the update message includes an updated value of the data item. This means that the pull phase may work even the initiator of the push phase leaves the network. A node may contact several nodes and select the most up to date versions among them. However, the algorithm does not describe how to get the most recent version in the system.

3 The Proposed Algorithm

In this section, we propose a new hybrid push-pull algorithm, named PRC (*Peer-to-Peer Replication Control*). PRC guarantees complete propagation of updates to every replica with reasonable overhead.

3.1 Data Structures

We assume that each node creates a new neighborhood with other nodes when it joins the P2P system. Gnutella's ping-pong protocol [1] is an example to make such neighborhood. Then a node N_i maintains the information of its neighbor nodes at $neighbor(i)$.

Every node in PRC maintains three data structures to synchronize propagations and to detect missed updates. First, a node N_i maintains an *update counter* $UC(i)$. N_i increments $UC(i)$ by one whenever N_i updates a data item. $UC(i)$ is broadcast to other nodes at the push phase. Next, N_i maintains an *update counter vector* UV_i for other nodes. $UV_i[k]$ stores a value of $UC(k)$ that N_i knows. Finally, N_i maintains an *update history table* H_i . If $H_i[k, UC(k)] = d$, then N_i

knows that another node N_k updates a data item d at $UC(k)$. The initial settings of the data structures are $UC(i) = 0$, $UV_i[*] = 0$, and $H_i[*] = \emptyset$. The contents of UV and H are updated at push and pull phases.

3.2 Push Phase

When a node updates a data item, the node makes a new update message and propagates it to the neighbor nodes. The update message consists of six attributes: $\langle \text{sender, initiator, identifier of data item, updated value of data item, update counter, sender list} \rangle$. The sender list is a set of nodes that have sent the same message. Let us suppose that a node N_i updates a data item d . Then the following steps are performed.

- (1) $UC(i) = UC(i) + 1$
- (2) $H_i[i, UC(i)] = d$, $UV_i[i] = UC(i)$
- (3) To every node in $neighbor(i)$, send an update message $\langle i, i, d, \text{value of } d, UC(i), \{i\} \rangle$.

Since N_i is an initiator of the update propagation, the update message includes i twice. The sender list includes only i due to the same reason. Now let us suppose that a node N_k receives the update message $\langle s, i, d, \text{value of } d, UC(i), \{i, s\} \rangle$ from a node N_s . Then the following steps are performed to process the message.

- (1) If $UC(i) == UV_k[i] + 1$, N_k has received every update message of N_i that was sent before $UC(i)$. So N_k needs to process this message only as follows.
 - $H_k[i, UC(i)] = d$, $UV_k[i] = UC(i)$
 - Replace the local copy of d by the value of d at the message.
 - To every node in $neighbor(k) - \{i, s\}$, send an update message $\langle k, i, d, \text{value of } d, UC(i), \{i, s, k\} \rangle$.
- (2) Else if $UC(i) == UV_k[i]$, this is a duplicate message. So N_k just neglects the message.
- (3) Else if $UC(i) > UV_k[i] + 1$, there are some updates that N_k missed. Then N_k requests the missed updates to N_s at the pull phase.

There are two comments on the push phase. First, when a node is in the sender list, it actually received the message. So other nodes do not need to send the message to any nodes in the sender list. As a result, compared with sending the receiver list as [3], PRC can increase the

probability of receiving the update message at the push phase. However, the probability of duplicate message propagation would be higher. Next, the update counter can guarantee complete propagation of updates to every replica. If some data items have been updated during a node is off-line, the node can eventually detect the missed updates after reconnection by comparing the update counter and its update counter vector. The node can get the missed updates at the pull phase.

3.3 Pull Phase

A node performs the pull phase at two cases: (a) when the node detects any missed updates at the push phase, or (b) just after the node joins the P2P system. A *selective pull* is performed at the former case. The node performs an *entire pull* at the latter case. We describe a selective pull first and then describe an entire pull.

3.3.1 Selective Pull

Let us suppose that a node N_k receives an update message $\langle s, i, d, \text{value of } d, UC(i), \{i, s\} \rangle$ from a node N_s , and N_k finds that $UC(i) > UV_k[i] + 1$. Then N_k performs the selective pull as follows.

- (1) N_k sends a pull message $\langle k, i, UV_k[i] \rangle$ to N_s .
- (2) When N_s receives the message, it makes a response message PR as follows.
 - $PR = \{ \}$
 - For $(t = UV_k[i]; t \leq UV_s[i]; t = t + 1)$, $PR = PR \cup \langle i, H_s[i, t], \text{value of } H_s[i, t], t \rangle$.
- (3) N_s returns PR to N_k . N_k updates its data structures and local copies as follows.
 - For each tuple $\langle i, d, v, t \rangle \in PR$, set $H_k[i, t] = d$ and replace the local copy of d by v .
 - $UV_k[i]$ is set to the maximum value of t for all tuples in PR .

3.3.2 Entire Pull

When a node joins the P2P system, it needs to validate the currency of its local copies by performing the entire pull. Let us suppose that a node N_k joins the P2P system. The entire pull consists of the following steps.

Table 1: Simulation Parameters

Parameter	Description	Setting
<i>CPU Speed</i>	Speed of nodes' CPU	1 Gips
<i>NetBandwidth</i>	Network bandwidth	100 Mbps
<i>NumNode</i>	Number of nodes	500
<i>NumNeighbor</i>	Number of neighbor nodes	8
<i>DiskTime</i>	Disk access time	10 ms ~ 30 ms
<i>MsgInst</i>	CPU instructions to process a message	22000
<i>PerIOInst</i>	CPU instructions for a disk I/O	5000
<i>NumDataItem</i>	Number of data items	1000
<i>UpdateRate</i>	Probability of a data item being updated	1.0
<i>OnlineRate</i>	Probability of a node being online	0.2
<i>LeaveRate</i>	Probability of an online node leaving network	0.0 ~ 0.5
<i>MsgLossRate</i>	Probability of a message being lost	0.0 ~ 0.3
<i>MsgDelay</i>	Delay to transfer a message	1 ms ~ 5 ms

- (1) N_k selects a node $N_s \in neighbor(k)$ in a random manner. Then N_k sends a request message $\langle k, UV_k \rangle$ to N_s .
- (2) When N_s receives the message, it makes a response message PR as follows.
 - $PR = \{ \}$
 - For each node n where $UV_k[n] < UV_s[n]$, perform the following steps.
For $(t = UV_k[n]; t \leq UV_s[n]; t = t + 1)$, $PR = PR \cup \langle n, H_s[n, t], \text{value of } H_s[n, t], t \rangle$.
- (3) N_s returns PR to N_k . N_k updates its data structures and local copies as follows.
 - For each tuple $\langle i, d, v, t \rangle \in PR$, set $H_k[i, t] = d$ and replace the local copy of d by v .
 - $UV_k[i]$ is set to the maximum value of t for all tuples in PR .

Note that the random selection of target node would not be optimal. This is especially true when the target node has missed some updates also. However, both the push phase and the selective pull can guarantee to get the missed updates eventually. If a node contacts multiple nodes at the entire pull, it may reduce the amount of missed updates. This in turn increases the communication overhead and consumes CPU cycles of more neighbor nodes.

3.4 Reconstruction of the Neighborhood

The correctness of PRC depends on the connectivity of a node with its neighbor nodes. When most neighbors leave the P2P system, the node would not receive any push messages. To detect such condition, each node sends periodically an `are_you_alive` message to every neighbor node. The living nodes reply to the message. If the number of living nodes drops down under the threshold value, the node has to reconstruct the neighborhood with new nodes by Gnutella's ping-pong protocol [1]. Furthermore, if the node has been isolated completely (i.e, the number of living nodes is zero), it has to perform the entire pull to one of the new neighbors.

4 Experiment Methodology

We develop an experiment model of a P2P system using the CSIM [10] simulation package to compare the performance of PRC with other algorithms. We consider two target algorithms, PO (push only) and LRA (List of Receivers Algorithm). PO implements a pure push algorithm. LRA is a hybrid push-pull algorithm, which implements the push phase of [3] and the pull phase of PRC. We do not consider another hybrid push-pull algorithm of [9], because it assumes that only an owner node can update a data item.

Table 1 summarizes the simulation parameters. We model a P2P system with a limited number

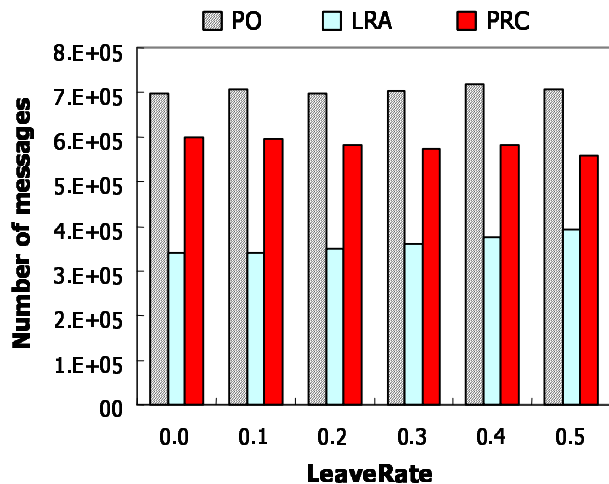


Figure 1: Number of messages
($MsgLossRate = 0$)

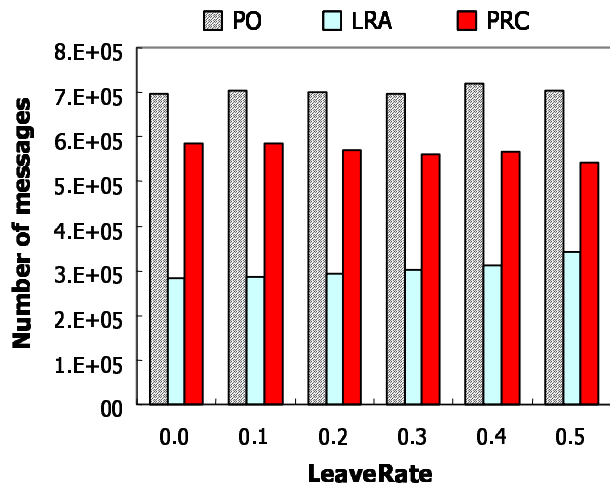


Figure 2: Number of messages
($MsgLossRate = 0.3$)

of data items and high update ratio. This setting helps us investigate the differences between algorithms. There are three types of nodes: online, join, and leaving. Online nodes are alive through the experiment. Join nodes are off-line initially and then join the network with a probability of $OnlineRate$. Leaving nodes are online initially, but leaves the network with a probability of $LeaveRate$. $MsgLossRate$ is a probability of a message being lost. These three parameters are used to model the dynamics of the P2P system.

5 Experiment Results

We have compared three algorithm (PRC, PO, and LRA) under a wide variety of network conditions. Specifically, we first compare the number of total messages propagated during the experiment. Then we compare the number of lost updates for each algorithm. Both experiments are performed by changing the dynamics of a network. When the network is stable, the value of $MsgLossRate$ is zero. An unstable network has nonzero value of $MsgLossRate$.

5.1 Experiment 1: Number of Messages

We first compare the message overhead of three algorithms. The unstable network has $MsgLossRate$ of 0.3. Figure 1 and Figure 2 show the experiment results at the stable network and the unstable network, respectively.

PO suffers from the heavy communication traf-

fic due to large number of messages. Note that PO does not filter any duplicate update messages. Every node just propagates an update message to all of its neighbors. PRC reduces the number of messages about 35% compared with PO. This is because PRC attaches a sender list at the update message so that it can reduce the duplicate message propagation significantly. As expected, LRA reduces the number of messages the most. It was about half of PO. The receiver list can prevent the duplicate update propagation completely, even though lost updates would happen.

An interesting observation is that the number of messages of LRA is reduced when $MsgLossRate = 0.3$. This means that some nodes do not receive messages at LRA when the messages could be lost. On the other hand, both PO and PRC perform similarly at both experiments. They allow a node to receive a message from multiple network links; hence, they are more resilient to the message lost. We will evaluate the effect of $MsgLossRate$ and $LeaveRate$ in detail at the next section.

5.2 Experiment 2: Lost Updates

We evaluate the number of lost updates for each algorithm by varying $MsgLossRate$ and $LeaveRate$. Figure 3 shows performance graphs when $LeaveRate$ varies from zero to 0.5 but $MsgLossRate$ is zero. This parameter setting models a stable network. As $LeaveRate$ increases, PO suffers from large number of lost updates. When

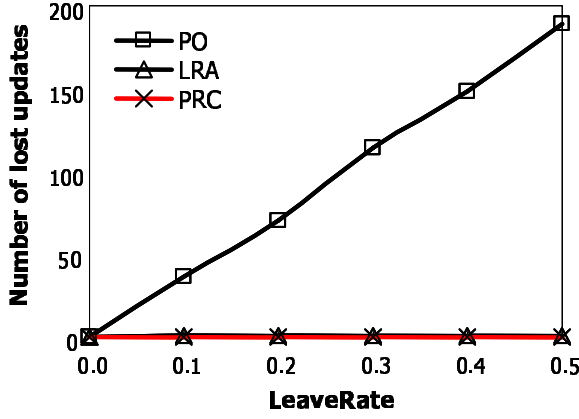


Figure 3: Number of lost updates
($MsgLossRate = 0$)

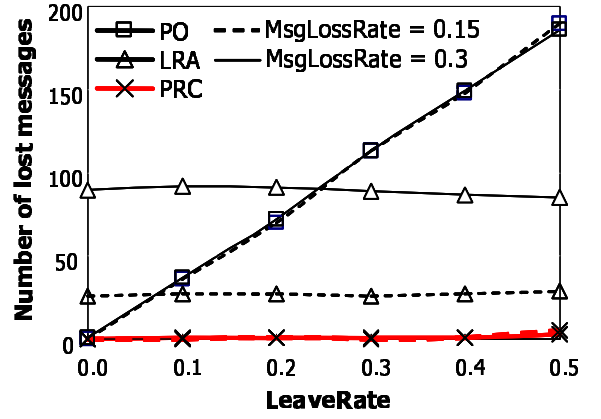


Figure 5: Number of lost updates
($MsgLossRate = 0.15$ and 0.3)

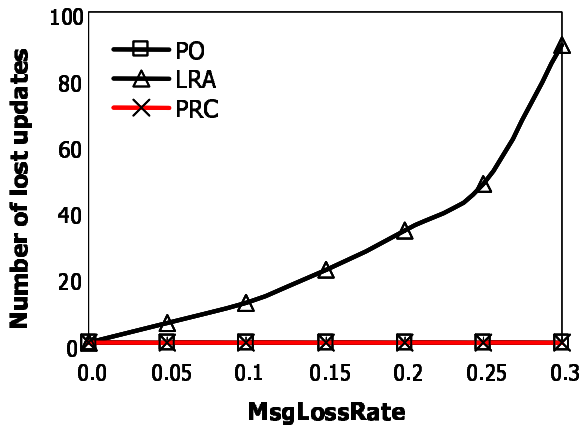


Figure 4: Number of lost updates
($LeaveRate = 0$)

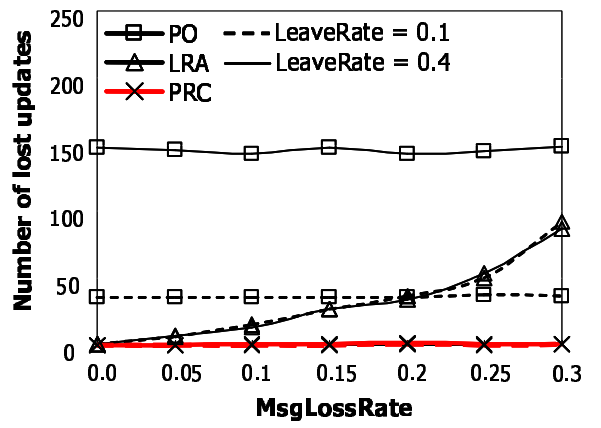


Figure 6: Number of lost updates
($LeaveRate = 0.1$ and 0.4)

$LeaveRate$ is 0.5, PO loses about 20% of updates. This is because PO does not include a pull phase. If a node is off-line during a push phase of an update, it should lose the update. On the other hand, both PRC and LRA do not cause any lost updates. They can allow a new joining node receive any missing updates at the pull phase.

Figure 4 shows experiment results when $MsgLossRate$ varies from zero to 0.3 but $LeaveRate$ is set to zero throughout the experiment. When every node is alive ($LeaveRate = 0$) and there is no message lost ($MsgLossRate = 0$), every algorithm propagates all of the updates completely. As $MsgLossRate$ increases, LRA loses more updates. When the $MsgLossRate$ is 0.3, LRA loses about 10% of updates. The reason is same to that we described at Figure 2. The receiver list approach of LRA restricts a node to receive an

update message from only one of its neighbor nodes. If the message would be lost, the node cannot receive the message again from any other nodes. Figure 4 also shows that both PRC and PO do not cause any lost updates. The duplicate message propagation is beneficial at this setting. Note that PRC does not cause any lost updates at both experiments.

The next two experiments compare three algorithms when both $MsgLossRate$ and $LeaveRate$ are not zero. Figure 5 shows the experiment results by varying $LeaveRate$ while $MsgLossRate$ is set to either 0.15 or 0.3. The experiment results are similar from the performance graphs of Figure 3 and Figure 4. PO performs worse as $LeaveRate$ increases due to the lack of pull phase. Its performance behavior is nearly equal at both settings of $MsgLossRate$. On the other hand, LRA

does not respond significantly to the change of *LeaveRate*. Its performance degrades just when *MsgLossRate* is high. PRC still performs best at this experiment.

Figure 6 shows the experiment results by varying *MsgLossRate* while *LeaveRate* is set to either 0.1 or 0.4. Most results correspond with the previous experiments. PRC can guarantee safe update propagation at unstable network when messages could be lost and many nodes leave or join the network.

6 Concluding Remarks

This paper has described PRC (*Peer-to-Peer Replication Control*), a new update propagation algorithm for P2P systems. PRC is novel in the sense that it is purely decentralized and has lower communication overhead. More importantly, PRC is adaptive: it can reconstruct the neighborhood and is resilient to the message lost.

We have demonstrated the efficacy of PRC using a number of different experiments. In the experiments, PRC propagates most update messages completely even when messages could be lost or nodes are often disconnected. This means that PRC can support P2P systems at mobile ad-hoc network. Furthermore, PRC also reduces the amount of duplicate message delivery significantly compared with the push only algorithm.

References

- [1] S. Androutsellis-Theotokis, and D. Spinellis, "A Survey of Peer-to-Peer Content Distribution Technologies," *ACM Computing Surveys*, 36(4), 2004.
- [2] U. Cetintemel, P. Keleher, B. Bhattacharjee, and M. Franklin, "Deno: A Decentralized, Peer-to-Peer Object-Replication System for Weakly Connected Environments," *IEEE Trans. Computers*, 52(7), 2003.
- [3] A. Datta, M. Hauswirth, and K. Aberer, "Updates in Highly Unreliable, Replicated Peer-to-Peer Systems," *Proc. of 23rd ICDCS*, 2003.
- [4] E. Franconi, G. Kuper, A. Lopatenko, and I. Zaihrayeu, "A Distributed Algorithm for

Robust Data Sharing and Updates in P2P Database Networks," *LNCS 3268*, 2004.

- [5] Gnutella, <http://www.gnutelliums.com/>.
- [6] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, P. Keleher, "Adaptive Replication in Peer-to-Peer Systems," *Proc. of 24th ICDCS*, 2004.
- [7] J. Holliday, R. Steinke, D. Agrawal, and A. E. Abbadi, "Epidemic Algorithms for Replicated Databases," *IEEE Trans. Knowledge and Data Eng.*, 15(3), 2003.
- [8] Kazaa, <http://www.kazaa.com/>.
- [9] J. Lan, X. Liu, P. Shenoy and K. Ramamritham, "Consistency Maintenance In Peer-to-Peer File Sharing Networks," *Proc. of 3rd IEEE Workshop on Internet Applications*, 2003.
- [10] H. Schwetmann, *User's Guide of CSIM18 Simulation Engine*, Mesquite Software, Inc. 1996.
- [11] P. Valduriez and E. Pacitti, "Data Management in Large-Scale P2P Systems," *LNCS 3402*, 2004.