A SECURE EPIDEMIC BASED UPDATE PROTOCOL FOR P2P SYSTEMS

[1]Manghui Tu and [2]Dianxiang Xu

[1]Department of CITG Purdue University Northwest 2200 169th Street Hammond, IN, 46323, USA
[2]Department of Computer Science Boise State University 1910 University Drive Boise, ID, 83725, USA

ABSTRACT

Epidemic Based Update Protocols Are Designed To Address The Consistency Issues For Data Replication In P2p Systems. However, Update Protocols Also Raise Security Issues. An Epidemic Based Update Protocol May Be Exposed To Security Threats When It Is Operated In An Untrustworthy P2p Environment. To Address This Issue, Security Prevention And Detection Protocols Are Designed In The Epidemic Based Update Protocol To Protect Update And Their Timestamps From Being Tampered By Compromised Malicious Peers. Theoretic Analysis Shows That The Secured Update Protocol Can Detect All Manipulations On The Timestamps Of Updates And Can Eventually Identify The Compromised Peers In The System.

KEYWORDS:

Peer-to-Peer;Epidemic based update protocols; consistency; timestamps; security

1. INTRODUCTION

Peer-to-Peer (P2P) systems aim to support collaboration and data sharing among participants [3, 18, 20, 24, 28]. A major barrier to support fast data accesses on large scale distributed systems is the high latencies of wide area networks. To overcome the barrier of high network latency, data stored in the P2P systems can be replicated at peers to improve both data access performance and availability [3, 4, 18, 20, 25]. However, replication raises consistency issues to be addressed. Asynchronous update protocols such as epidemic update protocols [2, 7, 9, 24] have been proposed to improve the response time and avoid single point of failures. In an epidemic update protocol, updates can be executed locally at any single site, and the updates are then propagated to all other sites. The key to the success of these protocols is to maintain the causal order, by using the transaction logs and a happened-before logic [2], which is implemented by a vector clock based two-dimensional timetable [2, 4, 8, 9, 17]. Security is another important issue in P2P environments where peers may be compromised and become malicious even with trust management, access control, and security protection in place [22, 23, 24, 27]. Adversaries may tamper the updates or update records and introduce conflicting updates into the system. Research works addressing security issues in P2P systems mainly focus on trust management [1, 21], data accesses, data storage, and routing mechanisms [26], and few of them addresses the security issues on the update dissemination. Some research works propose mechanisms to protect the integrity of the update itself [14, 15, 16, 19], but none considers the security protection of timestamps or vector clocks of the updates, which are essential to the correctness of the update protocol.

DOI : 10.5121/ijp2p.2017.8301

The security protection of timestamps of update transactions is not trivial. Timestamps are generated by peers to maintain the causal order of update events in the system. A compromised peer may generate fake timestamps for its own site, signed with its own private key and then propagate to other site. Such an attack cannot be prevented or detected by using digital signature based technologies. Also, a site may falsely claim that it has received an update that has not been propagated to the site or deny the fact that the site has received an update from another site. This cannot be prevented through the use of digital signature of the site that creates the update. Without appropriate protection, the tampered timestamps can poison update propagation by introducing non-existent conflicting updates, too many of which may lead to the crash of the system [7, 24], or introduce inconsistency to the system without being detected [24]. Thus, sophisticated mechanisms are needed to detect such malicious activities timely and precisely.

In this paper, to address security issues of the update protocol, detection procedures are designed in a two-level update protocol to secure the update propagation protocol in such a way that a non-compromised peer can detect tampered updates and prevent the updates from being further propagated to other sites. The remainder of this paper is organized as follows. Section 2 discusses related works on securing timestamps in distributed systems. Section 3 describes the system model of the P2P system and Section 4 gives background information about timestamps and the basics of the epidemic based lazy update protocol. Section 5 conducts security analysis and proposes a secure update dissemination protocol and Section 6 presents simulation results. Section 7 gives the conclusion of this paper.

2. RELATED WORK

The secure update dissemination issue has been recently studied in some research works [14, 15, 16, 19, 11]. In distributed systems, nodes may be compromised and can compromise the data availability and integrity. Secure dissemination schemes without public key signature has been developed in [14, 15, 16, 19]. The epidemic-style update diffusion in distributed systems that may suffer Byzantine component failures was first discussed in [14]. In this work, two protocols are developed based on the principle that a replica site accepts an update only if b+1 correct nodes have accepted the updates. A more efficient update dissemination algorithm is proposed in [15] and an optimal protocol is proposed in [16]. The algorithm may require less rounds of computation optimality of the algorithm. All these research works assume that there are b compromised nodes which become malicious, and thus require data to be written to at least b+1 non-faulty nodes initially before the update dissemination, which makes the dissemination slow. In [19], a new update dissemination algorithm is proposed by using the path verification protocol, which allows an update to be forwarded even when the forwarding host has not accepted that update. A host accepts an update only receiving b+1 propagations with the same update and disjoint gossip path. Even the sender cannot remove its own identity to fake a path, but it cannot prevent multiple compromised servers to fake multiple disjoint paths by modifying the existing path, without server integrity protection. Also, it cannot prevent compromised servers to impersonate other servers without proper authentication. Most of research works focus on public key algorithms to protect the integrity of the update from groups of clients [5, 11]. The digital signature method makes the integrity protection much efficient, but all of those research works do not consider the security protection of vector clocks, which is essential to the efficiency of update dissemination.

3. SYSTEM MODELING

Some peers are dedicated for the P2P system with strong security management (e.g. timely patch installation, appropriate firewall setup, well enforced security policies, strict access control, and secured communication and information storage). Some other peers participate as non-dedicated

servers without strong security management. Generally speaking, those peers without strong security management may have a higher probability to be compromised than those well managed peers. In this research, some of the peers are assumed to be untrustworthy.

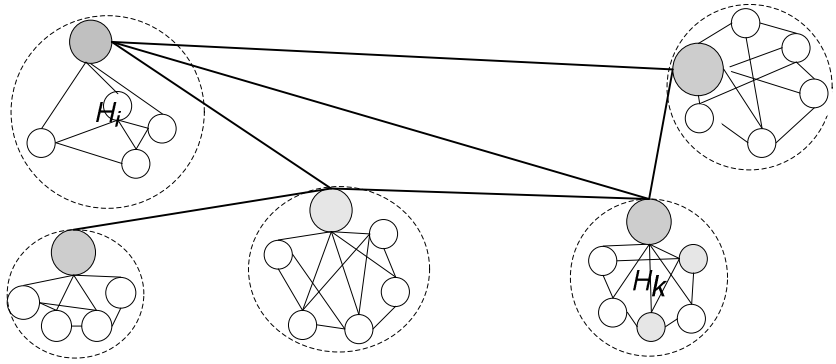


Fig.1 The topology of the P2P system modeled as a two level hierarchy.

To provide reliable and provable communication, P2P systems may be organized into structured overlay networks [6, 10, 13, 25, 28]. In such systems, peers are organized into groups and each group is managed by a super peer. In this research, we consider structured P2P overlay systems with two-level hierarchy, in which peers are grouped into non-overlap local groups. Each local group will be managed by a super peer, which represents the local group to communicate with other groups for update information exchange. The super peers are organized into a super group. A peer can only communicate with another peer in the same group (a super peer can communicate with other super peers and all peers in its local group). Let HC = {H0, …,HM-1} denote the set of M local groups and Pk = {Pk,0, Pk,1, … } denote the set of Nkpeers in group Hk, where Pk,0is the super peer. Also, let HS denote the super group, hence, HS = {P0,0, P1,0, …, PM-1,0}. Thetopology of the P2P system considered in this research is shown in Fig.1. Note that systems with millions of nodes can be extended to form a multi-layered hierarchical structure with more than 2 layers.

To improve access performance, data are replicated at peers in such way that the super group holds the primary copy of the data andeach super peer may hold a subset of the primary copy of the data. Note that data set hosted by the two super peers may be overlapping with each other. We assume that once a local group holds a replica of a data object, the super peer in that group also holds a copy of that data object (it could be the primary copy of the data object or a non-primary copy of such data object). Data can be read and updated at any peer, but only a limited number of users are allowed to make updates. Let t denote a transaction, site(t) denote the peer at which the update t is committed, TS(t) and p(t) denote the logical (timestamps) and physical time of transaction t, RS(t) and WS(t) denote the read and update data set of transaction t, and let r(t) denote the record of updatet. When a user needs to access a data object, it can access at any peer that holds a copy of that data object. For an update t, site(t) needs to propagate t to every peer that holds a copy of any non-empty subset of WS(t). To fit the large scale of the system, an epidemic based lazy update mechanism is chosen for update dissemination [2, 8, 9].

4. BACKGROUND INFORMATION

In an epidemic based update protocol, timestamps is critical to ensure the *happened-before* property, i.e., $\forall e, f \in E$, $e \rightarrow f iff TS(e) < TS(f)$, where $TS(e)$ and $TS(f)$ denote the timestamps of event e and event f, and E is the set of all events [4, 8, 9, 17]. The timestamps is maintained by the vector clock mechanism. The vector clock mechanism was proposed by [12] to ordering events in a distributed systems, in which each process needs to maintain a vector with the size of the number of all processes in the system. In such a system, each process is indexed and corresponds to an element in the vector clock. In the vector, each element corresponds to the number of events has taken place at its corresponding process. Replica sites maintain logs and then exchange log information to keep each other informed about the transactions that have occurred on their sites. This ensures that eventually all data replicas will incorporate all the transactions that have occurred in the system [4, 8, 9, 17]. Logs and timestamps are combined to a two-dimensional timetable to make the update dissemination more efficient [8, 9]. Essentially, each replica site P_i keeps a timetable T_i(shown in Fig. 2),the k^{th} row of which ($T_i[k, *]$) represents P_i's knowledge of the updates that peer P_k has received (through the update exchange information sent by P_k). If $T_i[k, j] = v$, then P_i knows that P_k has received the v^{th} update (namely, t') that is originally issued at peer P_j and all updates that are causally preceding t'(the v^{th} update issued at peer P_j).The row $T_i[i, *]$ represents P_i's record of the received updates that are originally issued at each replica site, e.g., $T_i[i, j] = u$ means that P_i has received the u^{th} update (namely, t) that is issued at P_j and all other updates that are causally preceding t. Also, each peer P_i maintains a local log, denoted as L_i, to log all updates issued locally at P_i or propagated to P_i.

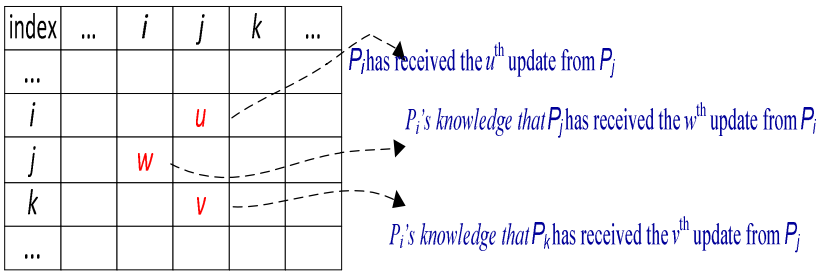


Fig.2 A sample timetable Ti at peer Pi.

Thus, the timetable can be used to define the well-known timetable property, e.g., $HasRecvd(T_i, t, P_k) \equiv (T_i[k, \text{site}(t)] \geq TS(t)[\text{site}(t)])$. That is, site P_i is sure that P_k must have received update transaction t if $T_i[k, \text{site}(t)] \geq TS(t)[\text{site}(t)]$. When P_i performs an update operation, it places a record in the log L_i. When P_i sends a message to P_k it includes all of such update t that $HasRecvd(T_i, t, P_k)$ is false, together with the time table T_i. When P_i receives a message from P_k it applies all non-conflict updates and updates its time-table in an atomic step to reflect the new information received from P_k. When a site receives a log record, it knows that the log records of all causally preceding events either were received in previous messages, or are included in the current message. This is referred as the log property and such property is stated as what follows: $\forall e, f, (e \rightarrow f) \wedge (f \in L_i)$, then $e \in L_i$ ($L_i$is the local log maintained by site P_i). This protocol ensures that whenever a site is aware of an update, it is aware of all causally preceding updates.

Now let's discuss how to apply timestamps to detect conflicting transactions. When $P_j$receives an update t issued at P_i, it first searches its local (update record) log $L_j$to see if there exists such a transaction t' that $TS(t) <> TS(t')$ (t and t' are executed concurrently, denoted as $t \parallel t'$), and the data accessed are overlapping (i.e., $(WS(t) \cap WS(t') \neq \phi) \vee (RS(t) \cap WS(t') \neq \phi) \vee (WS(t) \cap \text{RS}(t') \neq \phi)$ [2, 8, 9]. If we only consider transactions accessing a single data object and allow users to read

old data, then the condition of ($WS(t) \cap WS(t') \neq \phi$) is sufficient to determine the conflict. If such a transaction t' exists, then a conflicting flag is set with the record of update t (i.e.,$r(t)$), and transaction reconciliation procedure is to be launched to abort both update t and update t'.

The correctness of the protocol relies on the exchange of timestamps among all peers, and conflicts are identified and reconciled based on the causal order defined by timestamps. However, in an untrustworthy P2P system, a compromised peer may manipulate the timestamps of some updates, which can poison the updates in the system through propagating malicious update information to other sites. This may either introduce non-existing conflicts into the system which implements a denial of service attack, or introduce the inconsistency into the system without detection by hiding existing conflicts, which may lead the system into an unrecoverable state [7, 24].

5. SECURING THE UPDATE PROTOCOL

To secure the update protocol, security protection, malicious detection, and fault tolerance procedures will be designed. For simplicity, the description of the secure update protocol will focus on a single group first. Without loss of generality, an arbitrary group Hk will be considered.

A. SECURITY THREATS TO THE UPDATE PROTOCOL

The following attacks are considered in the research, (a) manipulating updates; (b) manipulating update value; (c) impersonating other peers to propagate updates; (d) manipulating timetable; and (e) manipulating timestamps for updates executed locally.

secureSendLocal ($P_{k,i}$, *msg*, $P_{k,j}$) {
if $RV_{k,i}(t)[i] \neq 0$ return ;
updateSet = {$r(t) | r(t) \in L_{k,i}$,
$\wedge \neg HasRecvd(T_{k,i}[j, *], t, P_{k,j})$};
sort the *updateSet* such that each $r(t)$ *is followed by* $r(t')$ *such that t' immediate concurrent or succeeds t*.
msg.updateSet= *updateSet*; and *msg.timestable* $= T_{k,i}$;
$P_{k,i}$ forms a signature, $sig_{k,i} = \{msg\}_{K_{k,i}}$
$r(t_1)$ = send (*msg*, $sig_{k,i}$) to $P_{k,j}$;
if $r(t_1)$ is valid $\wedge$ site(t_1) = $P_{k,j} \forall\ r(t) \in$ *updateSet*
if ($r(t).RV_{k,i}(t)[i] \leq TS(t_1)[i]$) $r(t).RV_{k,i}(t)[i] = TS(t_1)[i]$;
else *malicious_resolve*($P_{k,j}$); }

Fig 3 (a). The secured update execution protocol

secureUpdateExecution ($P_{k,i}$, *t*, *cid*) {
begin mutex
acquire write lock on $WS(t)$ and execute t;
$T_{k,i}[i, i] = T_{k,i}[i, i]$ ++;
$TS(t) = T_{k,i}[i, *]$;
commit(t);
client *cid* sign a signature, $sig_{cid} = \{cid, t.id\ (t), TS(t), p(t), WS(t), WS(t).value, site(t), t\}_{Kcid}$;
builds a transaction record $r(t) = \{\{cid, Tid(t), TS(t), WS(t), WS(t).value, site(t), t\}, sig_{cid}\}$;
$r(t)$. $RV_{k,i}(t)$ is initialized to be all 0;
$L_{k,i} = L_{k,i} \cup \{r(t)\}$;
end mutex;}

Fig 3 (b) The secured update forwarding protocol

To fight against attack methods (a) and (b), authentication mechanisms such as digital signature can be used to digitally sign t together with the value of the update t (WS(t).value), TS(t), client ID (cid), WS(t), local transaction ID (t.id), and site(t). Whenever a peer receives an update forwarding message, it verifies the digital signature of the message by using the client's public key stored locally. Therefore, no update message can be modified and no fake update can be generated by any compromised peer. The update execution procedures incorporated with authentication procedures are shown in Fig. 3(a). Similarly, to fight against attack method (c), the update forwarding message can be digitally signed with the ID of the peer who forwards the message and then verified by the receiving peer. The update forwarding procedures incorporated with authentication procedures are shown in Fig. 3(b).

B. VECTOR CLOCK MANIPULATIONS AND COUNTERMEASURES

It is much more complex to fight against attack methods (*d*) and (*e*). To prevent *TS*(*t*) from being manipulated by a peer other than *site*(*t*), *TS*(*t*) can be digitally signed together with the update (shown in Fig.3 (a)), and then verified by the receiving peer (shown in Part A of the secured update protocol described in Fig. 8). However, it cannot easily prevent and detect a compromised peer $P_{k,i}$ ($P_{k,i} \equiv site(t)$) to generate an arbitrary *TS*(*t*) for update *t* as the following 5 cases. Case 1, increasing$P_{k,i}$'s own entry in *TS*(*t*) from *TS*(*t*)[*i*] to *TS'*(*t*)[*i*], e.g., *TS'*(*t*)[*i*] > *TS*(*t*)[*i*]. Case 2, decreasing$P_{k,i}$'s own entry in *TS*(*t*) from *TS*(*t*)[*i*] to *TS'*(*t*)[*i*], e.g., *TS'*(*t*)[*i*]< *TS*(*t*)[*i*]. Case 3, increasinganother peer$P_{k,j}$'s entry in *TS*(*t*) from *TS*(*t*)[*j*] to *TS'*(*t*)[*j*], e.g., *TS'*(*t*)[*j*] > *TS*(*t*)[*j*]. Case 4, decreasing *another peer* $P_{k,j}$'s entry in *TS*(*t*) from *TS*(*t*)[*j*] to *TS'*(*t*)[*j*], e.g., *TS'*(*t*)[*j*] < *TS*(*t*)[*j*]. Case 5, any combination of the above 4 cases. Note that$P_{k,i}$ can also be a super peer, i.e., *i*= 0.

Consider case 1. $P_{k,i}$ ($P_{k,i} \equiv site(t)$) generates a manipulated *TS*(*t*) for an update *t* by increasing its i^{th} entry. For example, $P_{k,3}$generates *TS'*(*t*) = (1, 3, 3, **6**, 4) for the update *t* whose legitimate *TS*(*t*) should be (1, 3, 3, **5**, 4), as shown in Fig. 4.

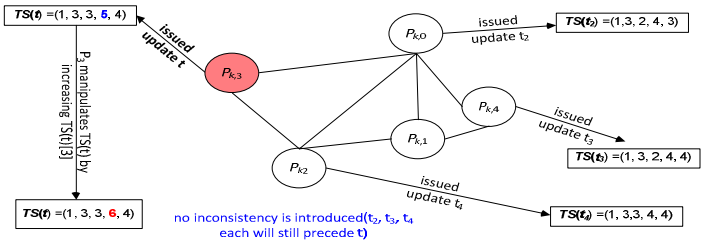


Fig. 4. *TS*(*t*) is manipulated by the compromised peer $P_{k,3}$ (which is site(*t*)) by increasing *TS*(*t*)[3].

Based on the conflict detection procedure defined in Section IV and Section V, this manipulation does not introduce any "*new*" inconsistency to the system (shown in Theorem 1). To detect this type of manipulation, we can enforce an ***immediate concurrent*or*immediate succeeding*** update execution order at the receiving site $P_{k,i}$ such that only an *immediate concurrent* update or an *immediate succeeding* update (denoted as t^f) of *t* can be selected as the next update to be tested locally for execution. An update t^f immediate succeeds another update *t*, denoted as $t \rightarrow_{im} t^f$, if $\forall j \neq site(t^f)$, $TS(t^f)[j] \equiv TS(t)[j] \wedge TS(t^f)[site(t^f)] \equiv TS(t)\ [site(t^f)] +1$. An update t^f isimmediateconcurrentwith another update *t*, denoted as $t <>_{im} t^f$, if $\exists i \in \{1, 2, \ldots, N_k\}$, $TS(t^f)[i] \equiv TS(t)[i]+1 \wedge \forall l \neq i$, $TS(t^f)[l] \leq TS(t)[l]$. Similarly, a timestamp V_1*immediate succeeds* another timestamp V_2 ($V_2 \rightarrow_{im} V_1$) if $\exists i \in \{1, 2, \ldots, N_k\}$ $V_1[i] \equiv V_2[i] +1 \wedge \forall j \neq i$, $V_1[j] \equiv V_2[j]$. A timestamp V_1 is *immediateconcurrent* with another timestamp V_2 ($V_2 <>_{im} V_1$) if $\exists i \in \{1, 2, \ldots, N_k\}$, $V_1[i] \equiv V_2[i] + 1 \wedge \forall l \neq i$, $V_1[l] \leq V_2[l]$. With this immediate succeeding or immediate concurrent execution order, whenever there is such a t^f missing in the propagation message from $P_{k,j}$, $P_{k,i}$ can conclude that a manipulation has been committed either by $P_{k,j}$ or the peer$P_{k,j}$ has been misled by other peers through the timetables mechanism (shown in Theorem 2), which requires a tracking process to identify the malicious peer (shown in the proof of Theorem 2). If itis misled,$P_{k,j}$ can simply resend the missing updates.

Theorem 1. If $P_{k,i}$ (*site*(*t*)) generates a manipulated timestamps for *t* by increasing the i^{th} entry of *TS*(*t*), then, the manipulation of *t* will not introduce any non-existent conflict into the system.
Proof: Suppose that an update *t* with *TS*(*t*) = (1, 3, 3, **5**, 4) is originally issued at peer $P_{k,i}$ (*i* = 3) and is manipulated to *TS'*(*t*) = (1, 3, 3, **6**, 4). Based on the current conflict detection procedure defined in Section IV, two updates (*t* and *t'*) conflict with each other if and only if $TS(t) <> TS(t') \wedge ((WS(t) \cap WS(t') \neq \phi)$. Suppose that there is another update t_1 conflicts with the manipulated update *t* but t_1 does not conflict with the original update *t*, then $TS'(t) <> TS(t_1) \wedge (TS(t) \rightarrow TS(t_1)$

$\vee TS(t_1) \rightarrow TS(t)$). First consider $TS(t_1) \rightarrow TS(t)$. We know that $TS(t) \rightarrow TS'(t)$ since $\forall j \neq i$, $TS(t)[j] \equiv TS'(t))[j] \wedge TS(t)[i] < TS'(t))[i]$, therefore, we have $TS(t_1) \rightarrow TS'(t)$ and this contradicts with the condition $TS'(t) <> TS(t_1)$. Then consider $TS(t) \rightarrow TS(t_1)$. Since t_1 reads from t, no matter what changes applied to $TS(t)$, $TS'(t) \rightarrow TS(t_1)$ will always hold based on the current epidemic update protocol. Therefore, it contradicts the condition $TS'(t) <> TS(t_1)$. Therefore, it is impossible that the manipulation of t as Case 1 will lead to $TS'(t) <> TS(t_1)$. Hence, it is true that such an update t with manipulated timestamps $TS'(t)$ will not introduce non-existent conflict into the system.

Theorem 2. Assume the ***immediateconcurrent* or *immediate succeeding*** update execution order is enforced in the entire system. If the update message $P_{k,i}$ received from $P_{k,j}$ missesan *immediate concurrent* or *immediate succeeding* update t^f, then either $P_{k,j}$ could have been misled by other peers through manipulated timetable or the site $P_{k,j}$ is malicious.

Proof: Case 1.1: if $\forall t_1 \in Z(t) = \{t | r(t) \in updateSet \wedge \neg HasRecvd(T_{k,i}, t, P_{k,i})\}$, $t^f \rightarrow t_1 \vee t^f <> t_1$, then t^f either has a smaller timestamps than or concurrent with the updates in the update propagation message. Let $t_2 \in Z(t)$ and has the smallest timestamps, then one or more updates preceding t_2 are missing in the update propagation message. Based on the sending protocol shown in Fig. 3(b), this can happen only if $P_{k,j}$ itself is maliciousor $P_{k,j}$ "*thought*" that $P_{k,i}$ has received such missing updates based on information provided by the two-dimensional timetable $T_{k,j}$, which is updated when $P_{k,j}$ receives update propagation message from another peer. This happens only if another peer has provided some false information to $T_{k,j}$ that the site $P_{k,i}$ has received such update t^f. To track which peer has provided such false timetable information, another two dimension table is needed for each site. Let $X_{k,j}$ denote such timetable at site $P_{k,j}$. If the knowledge of $P_{k,j}$ on the update reception status of a site $P_{k,i}$ is updated, e.g., an entry $T_{k,j}[i, m]$ is updated, and this update is done with the knowledge provided by $P_{k,l}$, then $X_{k,j}[i, m] = l$. Through this mechanism, it will always be able to identify which peer lied about the update reception status of $P_{k,i}$, together with the update propagation message log. Note that the tracking mechanism is very expensive since it may need to investigate multiple sites. Case 1.2: If $\exists t_1 \in Z(t)$ and $t_1 \rightarrow t^f \vee t^f <> t_1$, then t^f has a smaller timestamps than some but not all of the propagated updates in the update propagation message. Since each site $P_{k,j}$ strictly follows the ***immediate*** *concurrent* or ***immediate*** *succeeding* update execution order, it is impossible that those missing updates are propagated by other peers. Therefore, $P_{k,j}$ can be determined to be malicious.∈

Consider case 2. $P_{k,i}$ ($P_{k,i} \equiv site(t)$) generates a manipulated $TS(t)$ for t by decreasing its i^{th} entry. For example, $P_{k,3}$generates $TS'(t)$ = (1, 3, 3, **4**, 4)for the update t whose legitimate $TS(t)$ should be (1, 3, 3, **5**, 4), as shown in Fig. 5.

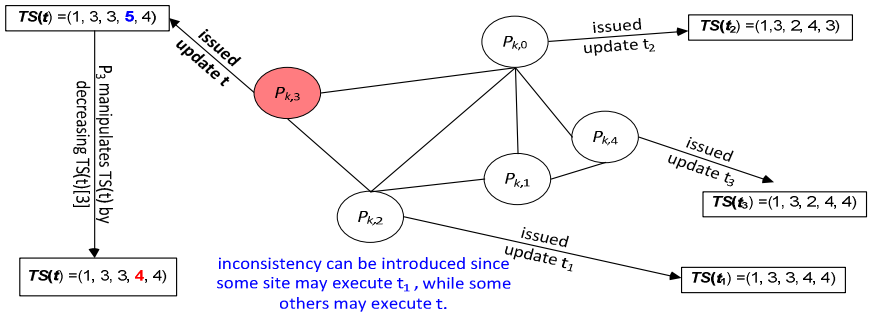


Fig. 5. $TS(t)$ is manipulated by the compromised peer $P_{k,3}$ (which is site(t)) by decreasing $TS(t)$[3].

Note that $TS'(t)$ is the timestamps of another legitimate update t_1(i.e., $t_1 \rightarrow t$). t_1 and t may arrive at different sites which introduces inconsistency undetected. Also, suppose that there is another update t_2 at $P_{k,0}$, $WS(t) \cap WS(t_2) \neq \phi$ reads from update t_1, then, $TS(t_2)$ = (**2**, 3, 3, 4, 4). The update protocol will treat the relationship between t and t_2 as $t \rightarrow t_2$ ($TS'(t) \rightarrow TS(t_2)$), while t and t_2 should actually conflict with each other since $WS(t) \cap WS(t_2) \neq \phi \wedge TS(t) <> TS(t_2)$ is true. Too many of such undetected inconsistencies and conflicts can bring the system into an unrecoverable state. To fight against this, a detection procedure should be designed in the update receiving protocol. Whenever a peer $P_{k,m}$receives an update propagation containing an update with timestamps (1, 3, 3, **4**, 4), the update is either the manipulated update t or the update t_1 *that* causallyprecedes t asshown in Fig. 5. Based on the current update protocol, either t or t_1 can be integrated in the system and the other will be ignored. To determine whether the update is legitimate, $P_{k,m}$ can check the value of the update t (or t_1) ($WS(t).value$ or $WS(t_1).value$) against the value of $WS(t)$ in the local database ($WS'(t).value$) or $WS'(t_1).value$ after t (or t_1) is executed locally. If the two values do not match with each other, then $P_{k,m}$ can determine that t is manipulated and site(t), $P_{k,i}$, is malicious.

Theorem 3. If $P_{k,i}$ ($site(t)$) generates a manipulated timestamps for t by decreasing the i^{th} entry of $TS(t)$, then any inconsistency introduced by the manipulation of t can be detected by a legitimate receiving peer $P_{k,m}$ and the malicious peer $P_{k,i}$ can be identified. Also, the manipulated update t will not create conflict with any other update.

Proof: There will only be two cases. Case 2.1, $TS'(t)[i] < T_m[m, i] + 1$, then t will be ignored by $P_{k,m}$. Case 2.2, $TS'(t)[i] \equiv T_{k,m}[m, i] + 1$, which means that $P_{k,m}$ has not received the update t_1 with $TS(t_1)[i] \equiv TS'(t)[i]$. If $TS(t_1) \neq TS'(t)$, then t will be detected by applying the *immediate concurrent* or *succeeding* order. If $TS(t_1) \equiv TS'(t)$. Since $t_1 \rightarrow t$ at $P_{k,i}$, by applying t to the local database at $P_{k,m}$ before t_1, $WS(t).value$ will definitely be different from $WS'(t).value$ (the value of the local database at $P_{k,m}$), which can be detected by $P_{k,m}$ and the malicious peer can be identified as $P_{k,i}$. Also, t_1 is an existing update and thus t with $TS'(t) \equiv TS(t_1)$ will not introduce non-existent conflict into the system. ∈

Consider case 3, $P_{k,i}$ ($P_{k,i} \equiv site(t)$) generates a manipulated $TS(t)$ for t by increasing its j^{th} entry. For example, $P_{k,3}$generates $TS'(t)$ = (1, 3, **4**, 5, 4)for the update t whose legitimate$TS(t)$ should be (1, 3, **3**, 5, 4), as shown in Fig. 6

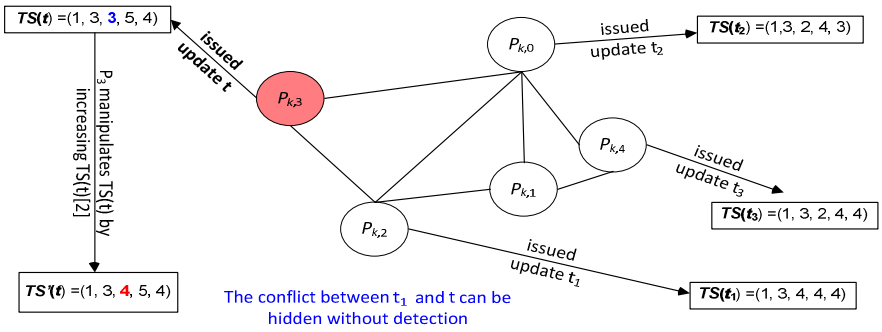


Fig. 6. $TS(t)$ is manipulated by the compromised peer $P_{k,3}$ (which is site(t)) by increasing $TS(t)$[2].

This manipulation may introduce inconsistency to the system undetected without appropriate handling. For example, assume that there is another update t_1($TS(t_1)$ = (1, **3**, 4, 4, 4)) is issued at site $P_{k,2}$ concurrently with t and $WS(t_1) \wedge WS(t) \neq \phi$, then t_1 should conflicts with $t$since $TS(t)$ = (1, 3, **3**, 5, 4). Without a detection procedure, the system will consider the relationship between t_1 and t as $t_1 \rightarrow t$ sincethe timestamps (1, **3**, 4, 4, 4) precedes ($\rightarrow$) the timestamps (1, 3, **4**, 5, 4)). In order to avoid being detected by enforcing the *immediate concurrent* or *succeeding* execution order, $P_{k,i}$can wait until t_1 arrives at $P_{k,i}$ before it propagates t to other sites. To detect the inconsistency

introduced by this attack, a peer $P_{k,m}$ can check the value of the update t ($WS(t).value$) against the value of $WS(t)$ in the local database ($WS'(t).value$) after t is executed locally. If the two values do not match with each other, then $P_{k,m}$ can determine that update $t$conflicts with update t_1 and site(t), $P_{k,i}$, is malicious.

Theorem 4. If a peer $P_{k,i}$ ($site(t)$) generates a manipulated timestamps for t by increasing the j^{th} entry of $TS(t)$, then, any inconsistency introduced (or any conflict hidden) by the manipulation of t can be detected by a legitimate receiving peer $P_{k,m}$ and the malicious peer $P_{k,i}$ can be identified.
Proof: if the manipulation is successfully implemented, both the j^h entry and i^h entry have been increased (j^h entry is increased for manipulation while i^h entry is increased for recording the legitimate update event).To pass the *immediate concurrent* or *succeeding* update execution order at $P_{k,m}$, $P_{k,i}$ has to propagate together with or after the update t_1 which is immediately succeeded by t. If site(t_1) $\neq P_{k,j}$, then whenever $P_{k,m}$ receives both t and t_1, $P_{k,i}$ (site(t)) will be determined as a malicious site. Consider site(t_1) $\equiv P_{k,j}$, which is essentially the same case as shown in case 3. Since $t_1$conflicts with t at $P_{k,i}$ and $t_1 \rightarrow t$, then there will be a difference between $WS(t).value$ and $WS'(t).value$ and it will be detected by a non-compromised peer $P_{k,m}$ and this can happen only because $P_{k,i}$ (site(t)) is malicious. €

Consider case 4, $P_{k,i}$ ($P_{k,i} \equiv site(t)$) generates a manipulated $TS(t)$ for t by decreasing its j^{th} entry. For example, $P_{k,3}$generates $TS'(t)$ = (1, 3, **1**, 5, 4)for the update t whose legitimate$TS(t)$ should be (1, 3, **3**, 5, 4), as shown in Fig. 7.

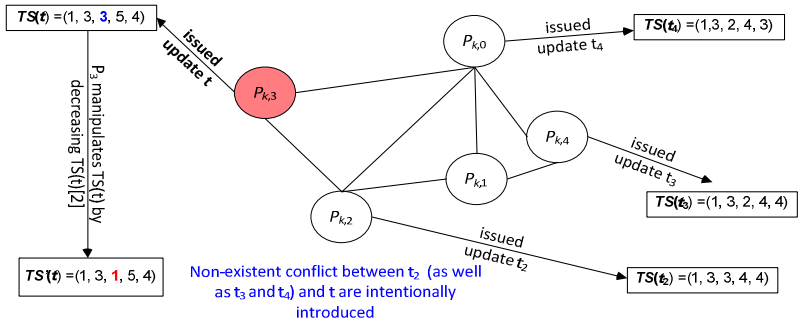


Fig. 7. $TS(t)$ is manipulated by the compromised peer $P_{k,3}$ (which is site(t)) by decreasing $TS(t)[2]$.

Without detection, such manipulation may introduce non-existent conflicts into the system. Examples of such updates could be t_2 and t_3, with $TS(t_2)$ = (1, 3, **3**, 4, 4) and $TS(t_3)$ = (1, 3, **2**, 4, 4). With $TS(t)$ manipulated, t_2 (or t_3) and $t$will conflict with each other, but actually their relationship should be t_2 (or t_3) $\rightarrow t$. To detect such attacks, $P_{k,m}$ can examine the timestamps of t (site(t) = $P_{k,i}$) against $T_{k,m}[i, j]$ for $\forall j$, where $j \in \{1, 2, \ldots, N_k\} \wedge j \neq i$(site($t$) = $P_{k,i}$). If $\exists j$ such that $TS'(t)[j] < T_{k,m}[i, j]$, ($P_{k,m}$ has known that P_i has received the update t_1 with $TS(t_1) = T_{k,m}[i, *]$), $P_{k,i}$ can be determined to be malicious. If $\forall j \in \{1, 2, \ldots, N_k\} \wedge j \neq i$(site($t$) = $P_{k,i}$), $TS'(t)[j] \geq T_{k,m}[i, j]$, the timestamps of t could either be manipulated or authentic. For example, $P_{k,3}$ has received both t_2 and t_3 but has not propagated them to any other site yet, therefore, no other peer will have the knowledge that $P_{k,3}$ received t_2 and t_3. The update t's timestamps, (1, 3, **1**, 5, 4), can be authentic (t isissued before the receipt of t_2 and t_3) or can be manipulated as described above (i.e., t isissued after the receipt of t_2 and t_3 and the original $TS(t)$ = (1, 3, **3**, 5, 4) is manipulated to $TS'(t)$ = (1, 3, **1**, 5, 4)). If $P_{k,i}$ keeps generating such timestamps, the system can potentially conflict all the updates in the system and lead to a denial of service attack.

The successful implementation of the attack described above is due to the missing of the records proving that t_2 and t_3 have been sent to $P_{k,3}$ before t is issued at $P_{k,3}$. Thus, an extra data structure is needed for the sending site to record the number of updatesthat have issued at the receiving site

$P_{k,i}$, i.e. $T_{k,i}[i, i]$. Such piece of information can be obtained through the acknowledge with the latest update t_2 issued at $P_{k,i}$ ($TS(t_2)[i] \equiv T_{k,i}[i, i]$). Although $P_{k,i}$ may acknowledge with an update that is not the latest update issued locally, this can be detected or can be ignored by other non-compromised peers as shown in Theorem 6. Since an update t_1 may be propagated to N_k-1sites by $P_{k,m}$, a timestamp, $RV_{k,m}(t_1)$, with N_k entries is sufficient to record such information. This procedure is described in Fig.3 (b). The detection procedure is designed in the receiving part described in Part A of the secured update receiving protocol shown in Fig. 8. Whenever a conflict update $t$is identified (which is a rare event), $P_{k,m}$ needs to examine the value of $RV_{k,m}(t_1)[\text{site}(t)]$ and $TS(t)[\text{site}(t)]$. If $RV_{k,m}(t_1)[\text{site}(t)] < TS(t)[\text{site}(t)]$ (which means that t_1 has been received by site(t) before t is executed at site(t) and the relationship between t_1 and t should be $t_1 \rightarrow t$), then site(t) can be identified as a malicious peer. It also indicates that a smaller $RV_{k,m}(t_1)[\text{site}(t)]$ acknowledged by site(t) (e.g., $P_{k,i}$) will be detected (note that site(t) e.g., $P_{k,i}$, cannot decrease its own entry in a vector without detection as discussed in case 2). However, this works only if $P_{k,m}$ has propagated t_1 to site(t) (e.g., $P_{k,i}$). Without the knowledge of the reception of t_1 by site(t), after a site $P_{k,m}$ determines that t_1 and t conflict with each other, $P_{k,m}$ will launch the procedure to resolve the conflicts. Therefore, the above detection procedure should be designed into the conflicting resolve procedure at each site$P_{k,m}$, and the input should include the record of t and all such conflicting updates t_1 logged at $P_{k,m}$ (who initializes the conflicting resolve procedure). A part of the detection procedure is described in Part B of the secured update receiving protocol shown in Fig. 8. In Theorem 5, we show that with all these detection procedure in place, the type of attack described in case 4 can be detected and such malicious site $P_{k,i}$ can be identified.

Theorem 5. A peer $P_{k,i}$ decreases the j^{th} entry of $TS(t)$ with $site(t) \equiv P_{k,i}$, and any inconsistency or non-existent conflict introduced by t will be detected by a legitimate receiving peer $P_{k,m}$ and the malicious peer $P_{k,i}$ can eventually be identified.

Proof: Suppose that there exists an update t_1 such that $t_1 <> t$, then $TS(t_1)[i] < TS(t)[i] \wedge \exists l \in \{1, 2, \ldots, N_k\}$, $TS(t)[l] < TS(t_1)[l]$. However, the manipulation as case 4 will not change this fact and it will not introduce any undetected inconsistency into the system. Suppose that the manipulated update t with $TS'(t)$ as case 4 conflicts with t_1 and $TS(t_1) \rightarrow TS(t)$ (if $TS(t_1) <> TS(t) \vee TS(t) \rightarrow TS(t_1)$, the manipulation as case 4 does not introduce any inconsistency or conflict. Since it is impossible to have $t_1 <> t$ if site(t) ≡ site(t_1), then consider site(t) ≠ site(t_1). Since $TS(t_1) \rightarrow TS(t)$, then t_1 must be propagated to $P_{k,i}$ by at least one peer $P_{k,l}$, $l \in \{1, 2, \ldots, N_k\} \wedge l \neq i$, and $TS(t_1)[i] < TS(t)[i] \wedge TS(t_1)[n] \leq TS(t)[n]$, $n \in \{1, 2, \ldots, N_k\} \wedge n \neq i$. Based on the secured update sending protocol described in Fig. 3 (b), at site $P_{k,l}$, it will be true that $RV_{k,l}(t_1)[i] < TS(t)[i]$ since $t_1 \rightarrow t$. In Part A of the secure update receiving protocol, $P_{k,m}$ calls the detection procedure defined in the algorithm *malicious_detected*($r(t)$, $P_{k,i}$) described in Part B of the secure update receiving protocol for each peer including peer$P_{k,l}$, the inconsistency or conflict introduced by the manipulated$t$will be detected and $P_{k,i}$ will be identified as the malicious peer.∈

Theorem 6. Upon receiving an update propagation message including update t_1, if $P_{k,i}$ acknowledges with an update t_2 that is not the latest update issued locally and manipulates the update t as case 4 so that $TS(t_1) <> TS'(t)$, it will be detected by another non-compromised peer$P_{k,m}$.

Proof: Known from the proof in Theorem 5, the introduced conflict between can be detected by a peer $P_{k,m}$ by asking site $P_{k,l}$ to evaluate whether $RV_{k,l}(t_1)[i] < TS(t)[i]$. if $TS(t_1) <> TS'(t)$ and $TS(t_1) \rightarrow TS(t)$, then $RV_{k,l}(t_1)[i] < TS(t)[i]$ will be true. If P_i had acknowledged with an update t_2 that is not the latest update issued locally, then $RV_{k,l}(t_1)[i]$ will be assigned with $TS(t_2)[i]$ and became $RV'_{k,l}(t_1)[i]$, since we know that $TS(t_2)[i] < RV_{k,l}(t_1)[i]$, it will be followed that $RV'_{k,l}(t_1)[i] < RV_{k,l}(t_1)[i] < TS(t)[i]$, then $P_{k,i}$ will be identified as a malicious peer. ∈

Now consider case 5, which is any combination of the above 4 cases. If there is any manipulation of update t at$P_{k,i}$ as case 1 and case 3, it will create miss of update t_1 that t immediate succeeds (i.e., $t_1^f \equiv t$) or concurrent. If site(t) ≡ site(t_1) ≡$P_{k,i}$, to fill in the missing of t_1, $P_{k,i}$ needs to wait for another update t_2 executed locally, and manipulates $TS(t_2)$ to $TS'(t_2) \equiv TS(t_1)$. Since $t_1 \rightarrow t \rightarrow t_2$ and the execution at another peer $P_{k,m}$ is $t_2 \rightarrow t$, then the difference between $WS(t).value$ and $WS'(t).value$ will be detected by$P_{k,m}$ and P_i will be identified a malicious. If site(t_1) ≠ site(t) ≡$P_{k,i}$,then the miss of t_1 cannot be made up and it will be detected by$P_{k,m}$ and $P_{k,i}$ will be identified a malicious. If an update t is manipulated by $P_{k,i}$ as case 2 combined with or without case 4, any inconsistency or non-existent conflict introduced by this type of attack will be detected by $P_{k,m}$ as shown in Theorem 3. For any entry of $TS(t)$ is manipulated by $P_{k,i}$ using the method shown in case 4, then it will be detected by using the detection procedure described above, i.e., evaluating the relationship between $RV_{k,l}(t_1)[i]$ and $TS(t)[i]$ for every update t_1 that t conflicts with. If any t_1 that $P_{k,i}$ has received before t, then the sending peer $P_{k,j}$ can detect the manipulation. Thus, manipulating more entries of $TS(t)$ by $P_{k,i}$ will only increase the opportunity to be detected by other peers.

Now we consider attack method (*d*), i.e., *manipulating timetable*. We first discuss some attacks that are related to the attack method (*d*). If $P_{k,i}$keeps locally issued update for a long time before propagate to other sites, then it will increase the opportunity to conflict with the updates issued at other sites. To deal with this issue, one choice for the protocol is to require each site send out updates periodically. Therefore, other peers will send updates to $P_{k,i}$ which cannot hide its most recent update as shown in Theorem 5 and a simple detection procedure can identify this attack. A compromised peer $P_{k,i}$may repeatedly send out the same update record $r(t)$ to some peers, which may introduce traffic to the system. To precisely detect this type of attack, we can require each sending site to first check $RV_{k,i}(t)[m]$ for each update record $r(t)$ to see if it has been sent to$P_{k,m}$. Also, each update record $r(t)$ at receiving site $P_{k,m}$ can use a data structure $SV_{k,m}(t)$ to record the send peer ID P_i. Note that $SV_{k,m}(t)$ can be a simple integer with each bit representing the corresponding peer ID. If bit i is 1, then $P_{k,i}$has sent $r(t)$ to $P_{k,m}$ and any receiving of $r(t)$ from $P_{k,i}$, $P_{k,m}$ can identify $P_{k,i}$ as a malicious site.

A compromised peer $P_{k,i}$ may provide false information about the reception of updates at other site $P_{k,j}$ by modifying the j^{th} row of the timetable $T_{k,i}$. If $T_{k,i}[j, *]$ is decreased, then other sites may be informed to send out some updates to $P_{k,j}$ that $P_{k,j}$ has already received from others. This will only introduce limited extra traffic into the system since $P_{k,l}$ may know that $P_{k,j}$ has already received those updates from other sources. If $T_{k,i}[j, *]$ is increased, then another site $P_{k,l}$ may be falsely informed that $P_{k,j}$ has already received some updates that $P_{k,j}$ never received. This *may* prevent some updates to be propagated to $P_{k,j}$ which may hurt the converge speed and increase the opportunity of those updates to conflict with updates issued at $P_{k,j}$. Note that it will be extremely difficult for $P_{k,i}$ to prevent all other peers to propagate those updates to $P_{k,j}$. Even if $P_{k,i}$ successfully does this, whenever $P_{k,l}$ sends out some other updates to $P_{k,j}$, $P_{k,j}$ can detect that there is a malicious attack against the system since some updates are missing by applying the *immediate concurrent* or*succeeding* order (Theorem 2). The system can choose to ignore this if no conflicts incurred or launch a forensics investigation to identify the malicious peer. Note that at the malicious peer $P_{k,i}$, there is no such timetable in the received update propagation message shows that $P_{k,j}$ have received those missing updates as $P_{k,i}$ claimed. Also, if $i \equiv 0$, then any update t originally executed in H_k will be considered as an update executed locally at $P_{k,0}$ (t^S), which will generate a new timestamps $TS(t^S)$ and then propagate t^S to other super peers. Since the update itself is signed by client, then no fake update can be generated by super peer. Therefore, for any $i \in \{1, 2, \ldots, N_k\}$, the above theorems still follow.

```
SecureReceive (P_{k,i}, msg, P_{k,j}) {
archive a copy of msg;
select r(t_1) ∈ L_{k,i} where
   site(t_1) ≡ P_{k,i} ∧ TS(t_1)[i] ≡ T_{k,i}[i, i];
acknowledged with digitally signed r(t_1) back to
P_{k,j};
T_{k,j} = msg. timetable;
for each n from 1 to N_k {
 if (T_{k,j}[j, n] < T_{k,i}[j, n]){
   malicious_resolve(P_{k,j});
     break;}}
updateSet = msg. updateSet;
sort the updateSet such that each r(t) is followed
by r(t') such that t' immediate or succeeds t.
If there is a missing of such r(t') to enforce the
immediate concurrent or succeeding order
execution  // case 1
 then malicious_resolve(P_{k,j});
for ∀t∈ {t | r(t)∈ updateSet} {
If TS(t)→T_{k,j}[j, *], then malicious_resolve(P_{k,j});
if (HasRecvd(T_i, t, P_{k,i})) {
if r'(t).SV_{k,i}(t)≡0 where r'(t) ∈ L_{k,i} , r'(t).SV_{k,i}(t)=1 ;
else   malicious_resolve(P_{k,j}); // re-send attack
updateSet.remove(r(t) and S = S ∪ r(t); }
for ∀t where r(t)∈ updateSet
 {verify the signature of t using client's public
key;
      if not valid then malicious_resolve(P_{k,j});
if TS(t) !immediate concurrent or succeeds T_{k,i}[i, *]
       then malicious_resolve(P_{k,j}); //case 1
for ∀t {t| r(t)∈ updateSet }{
begin mutex
 if {∃ t'∈ L_{k,i}|(TS(t)<>TS(t'))∧(WS(t)∩WS(t') ≠
∅) }{
    if ∃l ∈ {1, 2, …, N_k}∧ l ≠i) ∧
    TS'(t)[l] < T_{k,i}[j, l] ∧ site(t) ≡ P_{k,j}
      malicious_resolve(P_{k,j});
         boolean is_malicious = false; //case 4
         for each P_{k,l}where l ∈ {1, 2, …, N_k} {
is_malicious = malicious_detected(r(t), P_{k,l});
   if (is_malicious) break;}
     if (is_malicious != true) {
     resolveConflict(t);
          t.inconflict = true;
         resolveConflict(t');
          t'.inconflict = true;
     }
     else {
     malicious_resolve(P_{k,j});          }
if {∃ t'∈ L_{k,i}| (t'.inconflict) ∧ (t read from t') ∧
     (WS(t) ∧ WS(t') ≠ ∅) } {
         resolveConflict(t);
         t.inconflict = true;        }
else if (¬t.inconflict) {
         acquire write lock on WS(t)
         execute update t locally;
     if WS(t).value ≡ WS'(t).value.{
      commit(t) and release write lock; }
     else {
        rollback the execution of t
          malicious_resolve(P_{k,j)} // case 2, 3
         if (T_{k,i}[i, site(t)] < TS(t)[site(t)]) {
                T_{k,i}[i, site(t)] = TS(t)[site(t)] ;
                X_{k,i}[i, site(t)] = j;
                 r(t).SV_{k,i}(t)=1 ;
                L_{k,i}= L_{k,i} ∪ {r(t)};
                end mutex };
         begin mutex
         ∀ m,n, {
           value = max (T_{k,i}[m, n], T_{k,j}[m, n]);
           if (T_{k,i}[m, n] < value) {
                T_{k,i}[m, n] =  value;
                X_{k,i}[i, site(t)] = j;
                } // record who propagate the
         knowledge
         L_{k,i} = {t| r(t)∈L_{k,i} ∧ ∃ n| HasRecvd(T_{k,i},
t,P_{k,n})
         };
         end mutex }
}
```

Part A. The secured receiving protocol

```
malicious_detected(r(t), P_{k,i}){
 boolean flag =false;
 for {∀t_1∈ L_{k,i}|(TS(t)<>TS(t_1))∧(WS(t)∧WS(t_1)≠∅)
 { if (RV_{k,m}(t_1)[site(t)] < TS(t)[site(t)]){
         flag = true;
          break;}}
       return flag; }
```

Part B. A part of malicious detection protocol

Fig. 8. The secured update receiving protocol with malicious detection.

Let sig_{cid} denote the signature by client with ID *cid*, K_{cid} and K_{cid}^{-1}denote the private and public key pair of that client, *value*(*t*) denote the resulted value of *WS*(*t*), *Tid*(*t*) denotes the transaction ID of *t*. Let sig_{cid} ={*cid*, *Tid*(*t*), *site*(*t*), *TS*(*t*), *p*(*t*), *WS*(*t*), *value*(*t*), *t*}$_{Kcid}$. Note that client and the peer should execute the mutual authentication protocol and the client can obtain the site ID. The executing part and sending part of the secured update protocol is shown in Fig. 3(a) and Fig. 3(b), respectively, while the receiving part of the secure update protocol is shown in Fig. 8.

6. SIMULATION RESULTS

To evaluate the effectiveness of the two-level vector based update protocol on improving the consistency control in the system, we compare it with classic one level update protocol proposed in [2, 9]. Due to scale issues, it is not feasible to design a real system to simulate the update protocol, and thus a simulation is designed to simulate a system with large amount of peers, organized into a one-level system or a two-level group based hierarchy. Also, there is indeed no standard procedure to evaluate the system delusion problem discussed in [7]. In this simulation, we use the number of *conflicting units* as the parameter to evaluate the effectiveness of system consistency. Here, *conflicting units* is defined as the sum of the conflicting transactions that all of the peers have received, whenever a conflicting is detected. The reason to use *conflicting units* is that when there is a higher number of conflicting transactions introduced in the system, the higher the possibility of system will reach the state that cannot be recovered from the inconsistency.

A. THE EFFECTIVENESS OF THE SCALING UP MECHANISM

In the simulation, the two-level system consists of a number of groups and each group has one super peer, which further forms a super group together with the super peers from other groups. The simulated one-level system consists of the same number of peers as the two-level system. The system consists of a certain number of data objects with coarse granularity. Each update transaction is defined to access a single data object only, while the data object to be accessed is totally random. Update rate is simulated as the number of transactions generated by a peer between two consecutive propagations. Even though the number of updates each peer generated is totally random, the total number of transactions generated is constant.To effectively compare the two update protocols, the same set of update transactions, the same update rate and propagation rate are applied to both simulated system.

In this simulation, three factors, e.g., system size (the number of peers), the number of data objects, and the update propagation rate (simulate the propagation frequency, i.e., the number of peers will be chosen to propagate update to within each time period) are considered. We evaluate how these three factors impact the effectiveness of the two-level vector clock based update protocol on update inconsistency control. The results are shown in Fig. 9 (a), Fig. 9(b), and Fig. 9(c), respectively.

Fig. 9 (a) shows the impact of system size on the effectiveness of the two protocols. The parameter setups are: the update propagation rate is 2 and the number of data objects is 100. As you can see from the result, the larger the system size, the more effective the two-level vector based protocol. This is mainly due to the factor that the two-level vector clock is more scalable to large systems, compared with the one level vector clock based update protocol. Each local group consists of less number of peers, thus conflicting is much easier and earlier to be detected. Also, the conflicting updates originally executed at different local groups will be forwarded to the super group and the conflicting will be detected by the super group before they are forwarded to other local groups. This further helps the update inconsistency early detection.

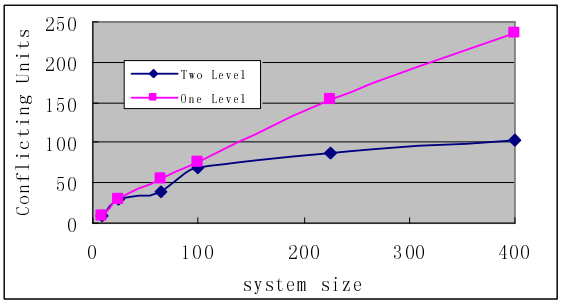
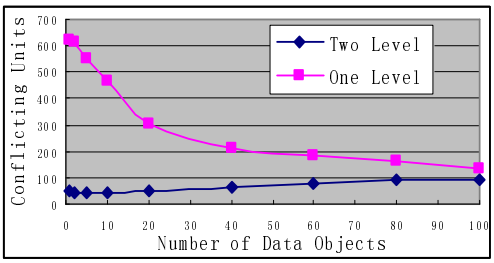


Fig.9 (a). The impact of system sizeFig.



9 (b). The impact of the number of data objects

Fig. 9 (b) shows the impact of the number of data objects on the effectiveness of the two protocols on update inconsistency control. The parameters setups are: the update propagation rate is 2 and the system size is 225. As you can see, the smaller the number of data objects, the more effective the two-level vector based protocol on update inconsistency control, compared with the one level vector clock based update protocol. This is due to the factor that with less number of data objects, the higher the possibility for two updates to be conflicting with each other, and the two-level update protocol can detect such conflicting earlier (less number of peers will receive such update). The larger the number of the data objects are accessed, the lower the possibility of two updates will be conflicting with other since the data objects accessed by the two updates have lower opportunity to overlap. Therefore, the effectiveness of the two level vector clock based update protocol on update inconsistency control is close to that of the one level vector clock based update protocol, when the number of data objects is large. Note that the system size is only 225 and update rate is 2 per peer. When the system size increases and/or the update rate increases, the two level vector clock based update protocol will have big advantage, as seen in Fig. 9(a) and Fig. 9(c.)

Fig. 9 (c) shows the impact of the update propagation rate on the effectiveness of the two protocols on update inconsistency control. The parameters setups are: the number of data object is 60 and the system size is 225. As you can see, the larger the update propagation rate, the more effective the two-level vector based protocol on update inconsistency control, compared with the one level vector clock based update dissemination protocol. This is due to the factor that the larger the propagation rate, the larger the number of peers will receive those conflicting updates, and the two-level update protocol can detect such conflicting earlier than the one-level update protocol. The smaller the update propagation rate, the smaller the number of peers will receive those conflicting updates. Therefore, the effectiveness of the two level vector clock based update protocol on inconsistency control is close to that of the one level vector clock based update protocol.

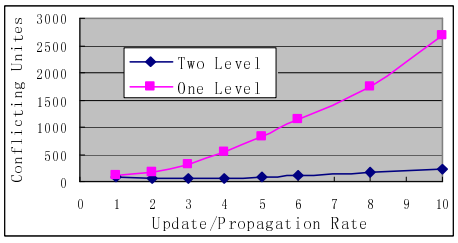


Fig.9 (c). The impact of update propagation rate

7. CONCLUSION

The P2P system considered in this research is modeled as a super peer based two-level hierarchy. A two-level vector clock mechanism is proposed to address the vector clock size issue andupdate dissemination protocolbased on the proposed vector clock mechanism is proposed to address the efficiency and scalability issues (e.g., the capability for the system to recover from update inconsistency) in widely distributed systems. To evaluate the effectiveness of the proposed two level vector clock based update dissemination protocol, simulations are conducted and the results show that inconsistency control is greatly improved compared to that of the traditional one level vector clock based update dissemination protocol. Also, we expect that our two-level vector clock based update dissemination protocol can also be scaled to a multi-level dissemination protocol with trivial adjustment. Due to the untrustworthy environment of the P2P data sharing, peers can be compromised by adversaries and malicious attacks can be launched to compromise the entire system. Without appropriate control, a single compromised peer can bring down the entire P2P system by manipulating the timestamps of the updates. To protect the P2P system against possible malicious attacks, security protocols are designed and integrated into the update dissemination protocol. Our analysis shows that the security protocol can achieve reasonable security protection and can detect all such malicious activities and identify malicious peers.

REFERENCE

[1] Aberer, K. and Despotovic, Z.. Managing Trust in a Peer-2-Peer Information System. Conference on Information and Knowledge Management, Atlanta, Georgia. 2000.

[2] D. Agrawal, A. El, Abbadi, and R. C. Steinke. Epidemic algorithms in replicated databases (extended abstract). In Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '97), 1997.

[3] S. Androutsellis-Theotokis and D. Spinellis. A survey of Peer-to-Peer content Distribution Technology. ACM Computing Surveys, Vol.36, No. 4, 2004.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati,A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall,and W. Vogels. Dynamo: amazon's highly available keyvaluestore. In Proceedings of twenty-first ACM SIGOPSsymposium on Operating systems principles, pp. 205–220, 2007.

[5] D. Ellard and J. Megquier. DISP: practical, efficient, secure and fault-tolerant distributed data storage. ACM Transactions on Storage. Vol.1, No. 1. 2004.

[6] Fiorano Software. Super peer architectures for distributed computing. White paper of Fiorano Software, 2007.

[7] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution.In Proceedings of the 96 ACM SIGMOD International Conference on Management of Data. 1996.

[8] J. Holliday, D. Agrawal, and D. Abbadi. Partial replication using epidemic communication. In Proceedings of the 22nd International Conference on Distributed Computing Systems, 2002.

[9] J. Holliday, R. Steinke, D. Agrawal, and D. Abbadi. Epidemic algorithms for replicated databases. IEEE Transactions on Knowledge and Data Engineering, Vol. 15, No. 3. 2002.

[10] M. Kleis, E. K. Lua, and X. Zhou, "Hierarchical Peer-to-Peer Networks using Lightweight Superpeer Topologies," In Proceedings of the 10th IEEE Symp. Comp. and Commun. (ISCC 2005), La Manga del Mar Menor, Cartagena, Spain, June 27–30 2005.

[11] S. Lakshmanan, M. Ahamad, and H. Venkateswaran. Responsive security for stored Data. IEEE Transactions on Parallel and Distributed Systems. Vol 14, No. 9, 2003.

[12] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM, Vol. 21, No. 7. 1978.

[13] E.K. Lua, J. Crowcroft, M. Pias, R. Sharma, S. Lim. A survey and comparison of peer-to-peer overlay network schemes. IEEE Communications Surveys & Tutorials, 7 (2) (2005), pp. 72–93.

[14] D. Malkhi, M. Reiter, O. Rodeh, and Y. Sella. On diffusing updates in a Byzantine environment. Proceedings of 18th IEEE Symposium on Reliable Distributed Systems, 1999.

[15] D. Malkhi, M. Reiter, O. Rodeh, and Y. Sella. Efficient update diffusion in Byzantine Environments. Proceedings of 20th IEEE Symposium on Reliable Distributed Systems, 2001.

[16] D. Malkhi, E. Pavlov, and Y. Sella. Optimal unconditional information diffusion. Proceedings of 15th International Symposium on Distributed Computing, 2001.
[17] F. Mattern. Virtual time and global states ofdistributed systems. InProceedings of the Workshop on Parallel and Distributed Algorithms. Pp. 215–226. 1988.
[18] Microsoft. Peer-to-peer transactional replication. Retrieved from http://technet.microsoft.com/en-us/library/ms151196.aspx. 2011
[19] Y. Minsky and F. Schneider. Tolerating Malicious Gossip Technical Report. Cornell University Computer Science TR2001-1853, 2001.
[20] SSWUG. Distributed database: one real world solution. http://www.sswug.org/editorials/default.aspx?id=2039. 2010
[21] G. Suryanarayana and R. N. Taylor. A survey of trust management and resource discovery technologies in Peer-to-Peer applications. ISR Technical Report # UCI-ISR-04-6. 2004.
[22] B.M. Thuraisingham and J.A. Maurer, Information survivability for evolvable and adaptable real-time command and control systems. IEEE Transactions on Knowledge and Data Engineering, Vol. 11, No. 1, 1999.
[23] M. Tu, P. Li, I. Yen, B. Thuraisingham, L. Khan. Secure data objects storage in data grids. IEEE Transactions on Dependable and Secure Computing. Vol. 7, No.1. 2010.
[24] M. Tu, D. Xu, Z. Xia. Securing epidemic based update protocol for P2P systems. In Proceedings of theIASTED PDCS 2011(Best Paper). December. 2011.
[25] M. Tu, H. Ma, I. Yen, F. Bastani, and D. Xu. Availability, security, access performance an load balance in P2P data grid.Journal of Grid Computing. Vol. 11, No. 1, 2013.
[26] B. Traversat and C. Haywood. A Survey of Peer-to-Peer Security Issues. Lecture Notes in Computer Science, Vol. 2609. 2003.
[27] D. Xu, M. Tu, M. Sanford, L. Thomas, D. Woodraska. Automated security testing with formal threat models. IEEE Transactions on Dependable and Secure Computing. Vol 9, No. 4. 2012
[28] B. Yang and H. Garcia-Molina. Designing a super-peer Network. In Proceedings ofIEEE International Conference on Data Engineering, 2003