uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Mehedi Masud

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

Ph.D. (Computer Science)

GRADE / DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Update and Transaction Processing in Peer Data Sharing Systems

TITRE DE LA THÈSE / TITLE OF THESIS

Iluju Kiringa

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

Hassan Ural

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Leopoldo Bertossi

Azzedine Boukerche

Bettina Kemme (McGill University)

Liam Peyton

Gary W. Slater

Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

UPDATE AND TRANSACTION PROCESSING IN PEER DATA SHARING SYSTEMS

BY

MD. MEHEDI MASUD

uOttawa

A THESIS SUBMITTED IN CONFORMITY WITH THE REQUIREMENTS
FOR THE DEGREE OF **Doctor of Philosophy in Computer Science**,
SCHOOL OF INFORMATION TECHNOLOGY AND ENGINEERING (SITE),
AT THE **University of Ottawa, Canada.**

Your file Votre référence
ISBN: 978-0-494-61388-7
Our file Notre référence
ISBN: 978-0-494-61388-7

Update and Transaction Processing in Peer Data Sharing Systems

Doctor of Philosophy in Computer Science Thesis
Ottawa-Carleton Institute for Computer Science
University of Ottawa, Canada

by Md. Mehedi Masud
September 2009

Abstract

Our thesis investigates an update exchange mechanism in data sharing systems. In these systems, databases in peers are created independently and may have semantic inter-dependencies with regards to data. Therefore, each peer specifies pair-wise mappings with other peers for sharing and exchanging related data. The mappings express how data in one peer relate to data in another peer. In our thesis, we first define settings for sharing and exchanging related data in a data sharing system, and then we discuss an update exchange semantics considering these settings. The update exchange models how changes made to a peer's instance are applied to the peer's local database instance and then propagated to related peers for maintaining data consistency in data sharing systems. The propagation is subject to transformations of the updates wrt the schema and data vocabularies (i.e., constants and relation names) of the related peers, and the results of the updates are consistent wrt the defined data sharing settings. Second, we propose an update translation strategy between a peer and its acquainted peers. The translation strategy first filters those updates from consideration that are ineligible for execution at the acquainted peers. During the translation, the locally expressed updates are transformed in terms of the schema of acquainted peers. Moreover, the translation ensures that insertions, deletions, and modifications of the tuples made by an update

in a peer and by the translated version of the update in an acquainted peer are related through the mappings between both peers. The translation is performed by considering the mappings between peers and through a syntactic analysis of update operations. Third, we investigate an update conflict detection and resolution mechanism during the propagation of updates. Fourth, we investigate the processing of transactions in a data sharing system. We mainly focus on how to maintain a consistent execution view of concurrent transactions in peers without a global transaction coordinator. For this purpose, we investigate potential problems that arise when maintaining a consistent execution of concurrent transactions. In order to guarantee consistent execution, we introduce a correctness criteria and propose two approaches, namely the Merged Transactions method and the Ticket method. We assume that one single peer initiates the concurrent transactions. In this thesis, we also discuss the architecture and functionalities of a simulation tool, called PDST, that we developed for simulating large peer data sharing systems. We evaluate our proposed strategies using this tool.

Acknowledgments

First of all, I glorify the greatness and bounties of almighty Allah who has bestowed on me the strength and ability without which it would not have been possible to carry out this thesis.

I am indeed grateful to my supervisor Dr. Iluju Kiringa who has given me the opportunity to do research in the exiting and evolving field of peer data management systems, and guided me to the way of innovation and novelty. I am also grateful to Dr. Hasan Ural, my co-supervisor, who has inspired and motivated me constantly by providing valuable guidelines and suggestions. Their invaluable suggestions and profound research experience kept me enthusiastic and optimistic all the way to the completion of this thesis. I thank them for their many hours of patience for listening to my problems. Their assistance, comments, constructive criticism, and positive attitude helped me proceed towards completing each step of this thesis.

I would also like to thank Dr. Leo Bertossi for his wisdom and constructive suggestions from the theoretical point of view on my thesis topic. I must acknowledge that his valuable comments in the thesis proposal defense have further enriched the content of this thesis. I would like to acknowledge the support that I received from the staff of the School of Information Technology and Engineering.

I am also grateful to my friends Anwar, Shamim, Anis, and other colleagues in my work place for their constant encouragement, inspiration, and suggestions. I give a special thank to Anwar for his generous cooperation and concern that has motivated me to pursue higher education in Canada.

No thankful worlds are sufficient to express my gratitude towards my mother and late father, and in-laws for love, encouragement and prayers (doas).

I would especially like to thank my beloved wife for her patience, unending support, and constant doas during the most critical periods of my PhD research.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Context

In the past few years, peer-to-peer (P2P) computing has emerged as a new paradigm for distributed data sharing. In P2P systems, all participating computing units (or peers) have equivalent capabilities and responsibilities, and exchange resources and services through a pair-wise communication, thus eliminating the need for centralized servers. Until now, there are many domain specific P2P systems (e.g. Freenet, Gnutella, SETI@home, ICQ, etc.) that are deployed. With a few notable exceptions, currently implemented P2P systems lack data management capabilities that are typically found in a database management system (DBMS), such as query processing and transaction processing.

In the last few years, steady progress has been made in research on various issues related to peer data management systems, such as data integration models [38], mediation methods [39], coordination mechanisms [81, 75], and data-level mappings [48] among the peer databases. These systems combine both P2P and database management system

functionalities. Contrary to the traditional data integration systems where a global mediated schema is required for data exchange, in peer data management systems semantic relationships exist between two peers, or among a small set of peers for exchanging data. The data are accessed globally from any peer by traversing the network of peers.

There is an increasing interest in the creation of peer data management systems, which includes establishing and maintaining mappings between peers and processing queries using appropriate propagation techniques. While there is a rich body of research concerning frameworks and mapping issues among peers, dynamic aspects of data in such systems have received much less attention. For example, in many collaborative data sharing efforts, particularly in biological and health sciences, data between sources are exchanged for sharing and coordinating information with each other. Generally, in collaborative data sharing, independent researchers or groups with different goals, schemas, and data agree to share data with one another. Each group independently curates, revises, and extends this shared data. At some point sources need to exchange the updates and reconcile their changes with each other in order to keep the collaborating sources updated. The reconciliation process in a collaborative data sharing system is not to provide one globally consistent data instance, but rather to give each source internally consistent instance that this particular source accepts from other sources.

In traditional relational database systems, two well-known update propagation scenarios exist, namely the view maintenance [36, 37, 15, 16] and view update problems [47, 7, 82, 23, 65]. A view is a derived relation that is defined in terms of base relations, where one side only contains base tuples, while the other side only contains data derived from those base tuples. In the view maintenance problem, updates made to

base relations are propagated to the materialized view [10] and in the view update problem updates made to a materialized view are propagated to the respective base relations. Authors in [35] introduce an update exchange mechanism in data exchange settings [29]. In a data exchange setting, one peer acts as a source (data provider) and another peer acts as a target (data receiver); and the peers are related through schema mappings in the form of tuples-generating dependencies (tgds) [9]. The mappings are equivalent to so-called global-local-as-view or GLAV mappings [56]. In the update exchange mechanism [35], users located at a peer P update the local database instance in an "offline" fashion and records the changes in a *local edit log*. Periodically, P publishes changes and accepts updates that other peers have published. Updates are also filtered based on $P's$ trust conditions that use the provenance of the data in the updates. The update exchange mechanism in [35] is roughly analogous to the view maintenance problem, i.e., a target peer gets updated in response to the updates made at a source peer. Similar to this work [35], authors in [67] proposed a framework for managing updates in a large-scale data sharing system applying the concept of view maintenance. In the system [67], a peer may act either as a data provider or as a data receiver. The schema of a receiver peer is defined as a view of the schema of data providers.

Authors in [13, 14] also proposed a data consistency semantics for maintaining consistency in peer data exchange systems without physically changing the data in peers. The inconsistency between databases of peers is managed at query time using the repair semantics [6]. The computation of repairs is based on the insertion and deletion of tuples to the local database instances at query time so that the resulting database satisfies all constraints (local integrity constraints and data exchange constraints).

Update Propagation

In this thesis, we investigate update exchange mechanisms in collaborative data sharing systems, where peers have stand-alone databases, independently developed that are linked to each other for sharing and coordinating data. Mappings or data sharing constraints between peers specify semantic relationship and coordination of data. In this system, an update submitted from a local user at a peer P is applied to the local instance and then P generates remote updates for execution into its acquainted peers. Remote updates are translated versions of the original update corresponding to the schemas of the acquainted peers. Peer P performs update translation using the mappings to generate corresponding updates over acquainted peers' schemas. As the updates are being translated, they are also checked for the correctness of translation such that the updates affect only the tuples in two collaborating peers that are related wrt mappings.

Contrary to the traditional data integration settings [53, 86, 56] where a global mediated schema is required for data exchange, in a data sharing setting, semantic relationships that exist between peers are exploited for sharing data. Moreover, in typical data integration and data exchange settings, we are often dealing with *one world*, for example, a set of sources all containing information about videos or music files. Hence, data integration or data exchange between data sources is provided mainly through the use of views or schema mappings, i.e., queries that map and restructure data between schemas. However, in a data sharing setting, sources may represent different worlds with different schemas and the real world entities denoted by a symbol in different sources may be related [11]. Moreover, the symbols representing the entities in two sources can be represented by two different vocabularies since sources are designed independently. In order to represent this situation, one may use a global schema with appropriate mappings

to/from each local source schema. However, building a global schema is not feasible in a P2P setting since (i) P2P networks are open-ended and continuously evolve, (ii) it requires huge effort and time to build a global schema for large number of peers, and (iii) it is not practical to build a global schema for every peer and her acquaintances as acquaintances keep changing [81]. Instead, we adopt a solution to this problem by creating a domain relation [81] between sources through pairwise mappings. The mappings associate the data elements of a source domain to the data elements of another source domain. In order to create a domain relation, we consider value correspondences between sources through the use of mapping tables [48]. In general, the mappings relate two objects (e.g. tuples) between peers to be shared.

Consider an example from the Health Care domain, where family physicians, walk-in clinics, hospitals, medical laboratories, pharmacists, and other stakeholders are willing to share information about patients treatments, medications, and test results. These databases need to remain acquainted and coordinate their contents for every shared patient. Coordination may mean something as simple as propagating updates to each other. For example, the patient database of a family physician and that of a pharmacist may want to coordinate their information about a particular patient, the prescription she has been administered, the dates when these prescriptions were fulfilled and the like. In such a coordination, data updates (insert, delete, or modify) in a database of a peer may, in turn, affect the database of other peers. For example, information for a certain patient may not exist in a walk-in clinic or in a pharmacy database until the patient visits for the first time. If the patient visits a walk-in clinic, the database in the walk-in clinic may import information about previous treatments from the family physician database, medication information from the pharmacy database, or medical test data from

the laboratories database. Any update of the patient information (e.g. address, blood pressure, new medication, cause of visit, medical tests, etc.) in the walk-in clinic database may trigger updates in the family physician database to keep it current.

Example 1 *We illustrate the above scenario more precisely considering two parties (i) A family physician and (ii) A specialist doctor. The sample database instances of the two parties are shown in Figure 1.1. Figure 1.1 shows that there are differences in the way patients' information are represented both at the schema and at the data instance level in the two peer databases since the databases are created independently. This representational discrepancy raises the need for some way of mapping one representation to the other, both at the schema and at the data levels. Such a mapping will permit interpretability between these databases. As mentioned earlier, we adopt a solution to this problem by creating a domain relation [81] between sources. In order to create a domain relation, we consider value correspondences between sources through the use of mapping tables [48]. In Section 2.1, we formally discuss how to create an acquaintance between two coordinating peers using value correspondences.*

Assume that the two peers (family doctor and specialist) want to coordinate data of the patients "Andrew Lucas" and "Ricky Ponting". Therefore, any update made in a peer on these shared patients should be exchanged to another peer in order to coordinate information of the patients. For example, assume the patient "Andrew Lucas" has visited the specialist. The specialist needs a medical examination that "Andrew Lucas" has gone through. To insert the medical examination report, the specialist posed the following local update in standard SQL against the local peer database SDB:

INSERT INTO SDB

SET TestID=1107, TestName=C0427512, Result=12.8 g/dL, Date=18/02/2009, PATID=235–

OHIP	Name	address	weight
233GA	Andrew Lucas	170 Lees Ave	65kg
501NE	Ley Davidson	300 Somerset St	75kg
359RA	Ricky Ponting	245 Main St	82kg

(a) Family physician patients database instance

OHIP	TID	TestName	Result	date
233GA	B6115	whitebloodcount	9755 c/mcL	10/12/2008
501NE	B5666	hemoglobin	14.6 g/dL	05/01/2009

(b) Family physician patients test database instance

PATID	Name	address	Reason	weight
235-02-09	Lucas, Andrew	170 Lees Ave	Sinus	65kg
430-08-08	Ray, Richard	652 Queen St	Brain Tumor	76kg
659-07-08	Ponting, Ricky	245 Main St	Headache	82kg

(c) Specialist patients database instance

TestID	Test	Result	Dt	PATID
1101	C0427512	6339 c/mcL	13/01/2009	235-02-09
1107	C0518015	12.5 g/dL	14/01/2009	235-02-09
1107	C0518015	09.5 g/dL	18/02/2009	659-07-08

(d) Specialist patients test database instance

Figure 1.1: Database Instances

$02-09$

Since the specialist shares information of "Ricky Ponting" with the family doctor, the following insert query has to be propagated to the family doctor database FDB:

INSERT INTO FDB

SET OHIP=233GA, TID=B5666, Test=hemoglobin, Result=12.8 g/dL, Dt=18/02/2009, PATID=$235-02-09$

Assume that TestID=1107 in SDB is represented with TID=5666 in FDB and TestName=C0427512 in SDB is represented with Test=hemoglobin in FDB. However, both databases represent test results and dates using the same vocabularies.

Consider another update where family doctor changes the weight of the patient "Ricky Ponting":

$$UPDATE\ FDB\ SET\ weight = 85\ WHERE\ name = "Ricky\ Ponting"$$

Since, family doctor shares information of "Ricky Ponting" with the specialist, hence, the following replace query has to be exchanged to the specialist database:

$$UPDATE\ SDB\ SET\ weight = 85\ WHERE\ name = "Ponting,\ Ricky"$$

Generally, due to the autonomy of peer databases, the specialist and family doctor should be allowed to pose their updates in terms of the schema and data values of their respective local peer database. However, due to data coordination policy, updates in one peer need to be propagated to a remote peer (acquainted peer) after transformation with respect to the data values and schema of the remote peer. The local peer uses the mapping information for transforming a local update into a remote update. Once an update is propagated to a remote peer the sender does not have any control of the execution of remote updates. There are some situations that can occur where a transformed update may apply to the tuples in an acquainted peer that from the point of view of the mapping are not related with the tuples that are updated in the local database of the initiator of the update. This means that updates happened in two peers that do not satisfy the data coordination policy that are in place between these two peers. These situations are described more elaborately in Chapter 4. Therefore, the main challenge is how a peer translates an update for a remote peer such that the tuples affected in two peers are related wrt the mappings between them. Therefore, we need an update translation algorithm which changes an update over the schema S_1 of a peer P_1 to an update over schema S_2 of a peer P_2, such that insertions, deletions, and modifications of the tuples by two updates in two collaborating peers are related wrt mappings between these peers. In order to achieve this translation, we first need to give a semantics of update exchange

between peers that precisely defines when an update to a local database is relevant for an acquainted peer and what it means to execute an update in a local database and into the acquainted peer. Moreover, the translation algorithm must be shown to be sound wrt the semantics.

Another concern wrt updates is how to support efficient mechanisms for propagating updates that may originate from any given peer on the network. Since peers are autonomous and there is no centralized control of the propagation of updates, a crucial concern is how each site independently executes multiple conflicting updates received from different initiators. There are two types of conflict that can occur between a pair of update operations: (i) semantic conflict and (ii) ordering conflict. Two update actions u_1, u_2 semantically conflict iff

1. u_1, u_2 insert tuples with same key values but different values in at least one attribute, or

2. either u_1 or u_2 is a deleting a tuple and the other is either inserting or modifying a tuple with the same key values, or

3. both u_1 and u_2 are modifying a tuple with different values.

Ordering conflict occurs if the execution order of two updates on a data item in two peers is different.

Example 2 *Consider that a peer P_2 has executed two updates u_1 and u_2 on a data item d in the order u_1u_2 sent by a peer P_1. Also assume that P_2 has received two new updates u_3 from P_1 and u_4 from P_3 that access d. Suppose that the execution order of the updates in P_2 is $u_1u_2u_4u_3$. Although, P_2 executed the updates in the same order $u_1u_2u_3$ as sent by P_1, u_3 may read different values of d at P_2 as read in P_1, if u_3 and u_4 access d.*

Transaction Processing

In this thesis we also investigate the execution of transactions in a data sharing system. We observe that transaction processing in a data sharing system is similar to processing in a multidatabase system (MDBS) [57, 18] in the sense that each system consists of a collection of independently created local database systems (LDBSs), each of which may follow different concurrency control mechanisms. Moreover, each system supports the management of transactions at both local and global levels. Global transactions are those that execute at several sites and local transactions are those that execute at a single site. In an MDBS, global level transactions are issued to the global transaction manager (GTM), and are decomposed into a set of global subtransactions to be individually submitted to the corresponding LDBSs. However, in a data sharing system, a global transaction is not decomposed, but rather propagated as entire transaction (that is, not the individual read and write operations that constitute the transaction). The remote peer that receives the transaction considers the transaction as submitted by local users. In MDBSs, transactions are executed under the control of the GTM. One of the main problem in MDBSs is ensuring serializability of the global schedule under the assumption that local schedules at each LDBS are serializable. A schedule S_k in a site s_k is a sequence of operations resulting from the execution of transactions located on that site. A serialization graph for a schedule S_k is a directed graph with nodes corresponding to the transactions that are committed in S_k and with a set of edges such that $T_i \rightarrow T_j$ if T_i conflicts with T_j in S_k. In what follows, $o_i(x)$ denotes an operation o_i of transaction T_i on database object x; $w_i(x)$ and $r_i(x)$ denote write and read operations, respectively. A transaction T_i and T_j are in direct conflict in schedule S_k, denoted by $T_i \rightarrow T_j$, at site s_k if and only if there exists operations $o_i(x)$ in T_i and $o_j(x)$ in T_j, $T_i \neq T_j$, such that

$o_i(x)$ followed by $o_j(x)$, where one of them is a write operation on a data item x and T_i does not abort before $o_j(x)$ is executed [18]. A schedule S_k is serializable if and only if the serialization graph is acyclic [12]. A global schedule S is a partial ordered set of all operations belonging to local and global transactions such that, for any local site s_k, a projection of S on a set of global and local transactions executing at site s_k is the local schedule S_k at site s_k [18]. Global serializability is ensured if there exists a total order defined over global transactions that is consistent with the serialization order of global transactions at each of the LDBSs. Since global transactions are under the control of the global transaction manager in MDBSs, the global transactions' serializability is enforced by a GTM.

In contrast, a data sharing system is built on a network of peers without a global transaction manager or controller. However, we can assume that each LDBS ensures the local serializabilty using the local concurrency protocol. Since the execution of concurrent transactions in a data sharing system is similar to the execution in an MDBS, we also consider the concurrent transactions' execution consistency criteria as ensuring serializabilty. The absence of a GTM in a data sharing system makes it more challenging to ensure global serializabilty since peers execute transactions independently beyond any centralized control. We observe that, although we assume that the LDBS of each peer guarantees serializability, concurrent transactions that execute in multiple peers may be serialized in different orders in different peers, resulting in an inconsistent execution of the transactions in the system.

Example 3 *Consider a data sharing environment consisting of three peers:* P_1 *and* P_2 *with data items a, b, and c and* P_3 *with data items* a, c. *Consider the following transactions* T_1 *and* T_2 *executed at* P_1, *concurrently.*

$$T_1 : w_1(a)w_1(c),\ T_2 : r_2(c)w_2(c)$$

Assume that the LDBS of P_1 produced the following schedule:

$$S_1 = w_1(a)w_1(c)r_2(c)w_2(c)$$

Suppose that peers P_2 and P_3 are connected with P_1. Hence, after execution of T_1 and T_2, P_1 propagates them to P_2 and P_3. Assume that when P_2 executes the transactions it also executes the following local transaction L_2.

$$L_2 = r_{L2}(a)r_{L2}(c)$$

Assume that the LDBS of P_2 generates the following schedule.

$$S_2 = r_{L2}(a)r_{L2}(c)w_1(a)w_1(c)r_2(c)w_2(c),$$

Similarly, when P_3 receives the transactions it also executes them using its LDBS and produces the following schedule:

$$S_3 = w_1(a)r_2(c)w_2(c)w_1(c)$$

If we observe we find that the resulting serialization orders of T_1 and T_2 at P_1, P_2, and P_3 are as follows:

$$SO_1 : T_1 \rightarrow T_2, \quad SO_2 : L_2 \rightarrow T_1 \rightarrow T_2, \quad SO_3 : T_2 \rightarrow T_1$$

Notice that each local schedule in each peer is serializable, but the serialization order or conflict relationship of T_1 and T_2 in the schedule S_3 at P_3 is different with respect to the schedule S_1 at P_1.

For ensuring database consistency in acquainted peers over the acquaintances with respect to the local execution of transactions in a peer, the serialization order of the

transactions must be same in all the acquainted peers if there are direct conflicts between transactions. It turns out to be more difficult when local transactions cause indirect conflict between global transactions that may not conflict directly when they are initiated. Moreover, we need to consider several failure-prone situations during execution of transactions that can occur. For examples, a peer may go offline when a transaction is active in the network, a peer may fail due to power failure or the system crashes, or a peer has successfully executed a transaction but the transaction has failed in one of its acquaintees. Examples of a transaction failure are a transaction abort to timeout, or a failure to pass the validation test by the local transaction manager. We can consider an offline status of a peer as a failure of a peer. Ensuring consistent serialization order of transactions in all participating peers considering failure-prone situations is another important concern in a data sharing system.

1.2 Thesis Overall Goal and Objectives

1.2.1 Overall Goal

The overall goal of this thesis is to investigate the issues of update and transaction processing in data sharing systems. For update processing, the updates originating in a peer are translated in terms of the schema of acquainted peers and propagated as translated updates to the acquainted peers. In order to achieve this translation, our first goal is to define a semantics of update exchange between peers that precisely describes what it means to execute an update in a local database and into the acquainted peer. After defining the semantics, the goal is to present an algorithm for translating updates destined to acquainted peers. The update translation algorithm is shown to be sound

wrt the defined semantics.

In investigating update propagation, our goal is to focus on update propagation mechanisms in a dynamic context where the network changes frequently, and propose conflict detection and resolution mechanisms for updates.

In investigating transaction processing, our goal is to analyze problems to ensure serializability of transactions in data sharing systems that arise due to the absence of a GTM and propose a condition that is both necessary and sufficient for ensuring serializability. Our goal is also to propose mechanisms for ensuring serializability taking into account the aforementioned condition.

1.2.2 Objectives

In Chapter 3, we formally state the problem statement that this thesis investigates. However, we list our overall objectives here in a general way:

1. Give an update semantics and investigate issues related to the translation of updates between heterogeneous peers in data sharing systems. The issues include detecting the conditions when an update in a peer is relevant for the peers' acquaintances and present an update translation algorithm between acquainted peers that is provably correct wrt the update semantics.

2. Search for a suitable update propagation algorithm in peer data sharing systems by investigating the existing optimistic approaches to update propagation. Moreover, the propagation must take into account the dynamic behavior of participating peers and offer a conflict detection and resolution mechanism. We consider two types of conflicts (i) update order conflict and (ii) semantic conflict.

3. Investigate problems for maintaining the same serialization order of concurrent transactions in peers in the absence of a global transaction manager.

1.3 Contributions

With respect to our objectives, contributions of this thesis are:

- We give a semantics of update and update translation between peers. Considering the update semantics, we propose an update translation algorithm where an update expressed wrt the schema of a peer is translated to an update corresponding to the schema of an acquainted peer.

- We propose two approaches for detecting which updates are relevant for propagation to acquainted peers. The approaches consider both the syntactic and semantic analysis of updates. Basically, the update relevancy techniques filter out those updates from consideration that are ineligible to execute at the acquainted peers.

- Inspired by the existing optimistic approaches to update propagation, we propose an update propagation mechanism that takes the dynamic behavior of peers in a network into account; and we propose a mechanism for conflict detection and resolution during the propagation of updates.

- We introduce a correctness condition that guarantees the same serialization order for concurrent transactions in different peers. We propose two approaches for ensuring this correctness condition, namely the Merged Transactions method and the Ticket method without violating the autonomy of local peer database management systems. While the Merged Transactions method is a completely novel approach,

the Ticket method however, exploits the concept of the existing Ticket method [33] for multidatabase systems.

- Finally, we implement the proposed update and transaction processing mechanisms, and show evaluation results which demonstrate the feasibility of the approaches.

Some of the results of this thesis have been published in [60, 61, 62, 63, 64]

1.4 Comparison

We now briefly compare our approach with the approaches that are closely related to our work. The other related works are presented in Section 3.3.

Authors in [35] proposed an update exchange mechanism for data exchange settings [29]. The mechanism is equivalent to the view maintenance problem. Like view maintenance settings, in a data exchange setting, one peer acts as a source (data provider) and another peer acts as a target (data receiver). The target schema is defined as a view over the source schemas. In our work, we investigate an update exchange mechanism in a data sharing system where each peer contributes its own data in the system and where, relationships exist between the data items among peer databases (e.g., acquaintances are created between different parties in a health care network for sharing information about patients, acquaintances between the partner airlines in a flight reservation network are created for sharing data of flights and passengers). We also consider a data sharing scenario where data vocabularies are different in peer databases. To us, obtaining a peer's schema as a view over another peer's schema is not desirable [48] because of peer independence.

Work in [35] performs update exchange by using the concept of *state-transfer* [77]. In

the state-transfer mechanism, updated data are propagated between peers. In our work, however updates are propagated using the concept of *operation-transfer* [77]. In the operation-transfer mechanism, update operations, and not data, are propagated between peers. We believe that the operation-transfer based update propagation mechanism is more feasible in the context of peer to peer systems. The reason is that when the volume of updated data is large then it is costly to propagate the data in a network using the state-transfer based update propagation. In the operation-transfer mechanism, the update operations are propagated which are usually smaller in size compared to the volume of the updated data.

In [35], changes of data are exchanged in an "offline" fashion. The updated records are kept in a local *edit log* and later the updates are published. However, in our case changes of data are exchanged on-the-fly; i.e., if any change on data occurs in a peer database, the change is propagated immediately to the acquainted peers.

In [35], a conflict between updates is resolved by using data provenance information and trust relationships between peers. Therefore, a centralized data provenance repository is required. However, in our proposed conflict resolution mechanism, each peer independently resolves a conflict applying the proposed resolution technique.

Authors in [31] introduced a semantics of data exchange between peers in peer data exchange settings which is a generalization of data exchange settings. Unlike data exchange, however, the target is no longer a passive recipient of source data. The target peer uses constraints, that is absent in data exchange settings, to impose restrictions on the data that it is willing to receive from a source. Moreover, a target may have its own data. The fundamental problem that is addressed in a peer data exchange is augmenting a target instance with the source instance in such a way that the given source instance

and the augmented target instance satisfy all constraints between them. In our work, however, we consider update exchange between peers. The problem we consider is to update a target instance that is triggered by an update initiated in the source instance. To update target instance with the change in the source instance, we perform translation of updates against the source schema, using the mappings to compute corresponding updates over the target schema. As the updates are translated they are also verified for the correctness.

1.5 Outline

This thesis is organized as follows. In Chapter 2, we discuss the technical preliminaries and data sharing system model used throughout the thesis. Chapter 3 contains the problem statement and the objectives of this thesis. In Chapter 4, we introduce the update semantics and the issues related to the translation of updates between peers. Chapter 5 discusses an update propagation strategy and proposes a mechanism for detecting and resolving update conflicts. Chapter 6 analyzes the processing of concurrent transactions in peer data sharing systems. In Chapter 7, we describe a tool developed for simulating peer data sharing systems. Chapter 8 presents experimental results obtained on the update translation and processing, as well as transaction processing mechanisms. Finally, Chapter 9 concludes the thesis and outlines future work.

Chapter 2

Technical Preliminaries and System Model

This chapter contains the precise definitions of many concepts needed for our investigation of a data sharing system. Since mappings are established through pair-wise fashions in a data sharing system, we first describe the data sharing setting where only two peers are involved. Later, we generalize the setting to multiple peers.

2.1 Preliminaries

Attributes are symbols taken from a given finite set $U = \{A_1, \cdots, A_q\}$ called the universe. We use the letters A, B, C, $\cdots$ to denote single attributes and X, Y, $\cdots$ to denote sets of attributes. Each attribute A_j is associated with a finite set of values called the *domain* of A_j and is denoted by $dom(A_j)$. Suppose $X = \{A_1, A_2, \cdots, A_k\} \subseteq U$, with the elements $A_i (1 \leq i \leq k)$ taken in the order shown, then $dom(X) \subseteq dom(A_1) \times dom(A_2) \times \cdots \times dom(A_k)$. A non-empty subset of U is called a *relation schema* R. A *database schema*

is a finite collection $\mathbf{R} = (R_1, \cdots, R_m)$ of relation schemas. A tuple t with attribute X is a *X-value*. A relation is a set of tuples. We shall use r_i to denote a relation that interpreters R_i. An *instance* I over $\mathbf{R}$ is a function that associates each relation schema R_i to a finite relation $r_i = I(R_i)$. For a tuple t and a set $Y \subseteq U$, we denote the restriction of t to Y by $t[Y]$.

Let S be a schema at a peer P_i and T be a schema at another peer P_j. If data sharing constraints are specified from S to T, then we call S as a source schema and T as a target schema. The instances over S and T are denoted by I and J, respectively. Now, we discuss the data sharing constraints.

Source-to-target schema constraint: Source-to-target schema constraints are constituted by a set of assertions of the forms

$$\Sigma_{st}^{s} = q_S \rightarrow q_T$$

where q_S and q_T are two queries, respectively over the source schema S, and over the target schema T. Intuitively, an assertion $q_S \rightarrow q_T$ specifies that the concept represented by the query q_S over the sources corresponds to the concept in the target schema represented by the query q_T. We consider the form of assertions as tuple-generating dependencies [9]. An example of assertions can be specified as logical expressions of the form:

$$\forall_{\mathbf{x}}[\exists_{\mathbf{w}}\phi(\mathbf{x},\mathbf{w}) \rightarrow \exists_{\mathbf{z}}\psi(\mathbf{x},\mathbf{z})]$$

where the left hand side (LHS) of the implication, ϕ, is a conjunction of relation atoms over the schema of S and the right hand side (RHS) of the implication, ψ, is a conjunction of relation atoms over the schema T. The mapping expresses a constraint about the existence of a tuple in the instance satisfying the constraint of the RHS, given a particular combination of tuples satisfying the constraint of the LHS. A tuple-generating

dependency (tgd) is an implication in which no equality atoms occur (in either the left or right hand sides of the implication).

Example 4 *Consider a family physician database (FDB) with the schema S consisting of two relations R_1(OHIP, Name, Address, Illness, DOB) and R_2(OHIP, TestName, Result, Date). Also consider a database in a research cell database (RDB) with the schema T consisting of a relation R_3(OHIP, Name, Illness, DOB, TestName, Result). Assume the following source-to-target schema constraint is designed between FDB and RDB.*

$$\forall_{ohip,name,illness,dob,testname,result}[\ \exists_{address} R_1(ohip, name, address, illness, dob),$$
$$\exists_{date} R_2(ohip, testname, result, date) \rightarrow R_3(ohip, name, illness, dob, testname, result)]$$

The constraint expresses that patients' data (ohip, name, illness, dob, testname, result) are shared from FDB to RDB.

Source and target constraints: A source constraint is a constraint over a source schema S. The constraints are basically local integrity constraints (ICs) on the local database. Examples of ICs are functional dependencies and foreign key constraints. Generally, source constraints do not play any direct role in data sharing. Similarly, a target constraint is a constraint over a target schema. Target constraints play roles in data sharing since data received from a source are validated through target constraints. We assume that each peer is responsible for maintaining its database to keep it consistent with respect to its local integrity constraints independently from other peers, since the local database is independently designed. Notice that source and target constraints are not used for update translation. However, we introduce these constraints to define the semantics of update execution.

Source-to-target data-level constraints: The data-level constraints impose constraints on data values by associating values between two sources, where databases may

OHIP	PATID
233GA	235-02-09
359RA	659-07-08

(a) Mapping table OHIP2PATID

TestName	Test
whitebloodcount	C0427512
hemoglobin	C0518015

(b) Mapping table TestName2Test

Figure 2.1: Mapping tables

use different vocabularies and different domains. Intuitively, the association of values creates a *domain relation* [81] between two semantically related domains and relates data objects (tuples) of two peers. Authors in [48] show how to create data-level mappings by using mapping tables. Using mapping tables, Hyperion [48, 75] introduces a data sharing system for sharing data between peers. Our work is motivated by the Hyperion data sharing system. A mapping table, denoted by $m(X, Y)$, simply is a relation over the sets of attributes X and Y. A tuple $(\mathbf{x}, \mathbf{y})$ in $m(X, Y)$ indicates a mapping that the value $\mathbf{x} \in dom(X)$ is associated with the value $\mathbf{y} \in dom(Y)$.

Example 5 *Consider the database instances in Example 1. Observe that patients' and medical test information are represented in two databases with different data vocabularies. In order to associate or relates patients in two databases for sharing patients' data and resolve data heterogeneity, sample mapping tables in Table 2.1 are designed. Figure 2.1(a) shows a mapping table OHIP2PATID that associates patients' ids of two patients in two peer databases FDB and SDB. It represents mappings from the attribute OHIP to the attribute PATID. Intuitively, according to the interpretation of the table OHIP2PATID, it records associations of patients' identifiers "233GA" in FDB with "235-02-09" in database SDB and "359RA" with "659-07-08". There is another mapping table 2.1(b) which resolves data heterogeneity the way test names are represented in two peers.*

The values appearing in the tables presented thus far only constants. However, to augment the expressiveness of tables variables are used [48]. Specifically, let V be a set

of variables where $V \cap dom(A) = \emptyset$, for every attribute A. Therefore, a mapping in a mapping table is a tuple which may contain constants or variable. Formally, a mapping, alternatively called a tuple t in a mapping table, has the following meaning.

Definition 1 (Mapping) *[48] Given a set of attributes W of $X \cup Y$, t is a mapping over W if for each $A \in W$, $t[A]$ is either a constant in dom(A), a variable in V or an expression of the form $v - F$, where $v \in V$ and F is a finite subset of dom(A).*

Given the presence of variables in mappings, it is necessary to introduce the notion of a valuation.

Definition 2 (Valuation) *[48] A valuation ρ over a mapping table m is a function that maps each constant value in m to itself and each variable v of m to the value in the intersection of the domains of the attributes where v appears. Furthermore, if v appears in an expression of the form v-F, then $\rho(v) \notin F$.*

Introduction of variables in mapping tables offers a compact and convenient way of representing common associations between values. An example of such a mapping table is illustrated in the following example.

Example 6 *Consider the database instance in Figure 1.1. Notice that the patient names are used in two different formats in two databases. There are two alternative mapping tables that we can use to represent the association between the two sets of patient names. The two tables are shown in Figure 2.2. Mapping table Name2Name in Figure 2.2(a) contains a set of mappings of the form (name1; name2), where name1 is a patient name in the FDB and name2 is a patient name in the SDB database. Alternatively, we can construct a more succinct, data independent, mapping table Name2Name containing the single mapping, (v; v'), where v and v' are variables. The alternate mapping table is*

Name	Name
Andrew Lucas	Lucas, Andrew
Ricky Ponting	Ponting, Ricky

(a) Mapping table Name2Name

Name	Name
v	v'

(b) Alternate representation of Name2Name

Figure 2.2: Mapping tables

OHIP	PATID
233GA	233GA
359RA	359RA

(a) Mapping table OHIP2PATID

OHIP	PATID
v	v

(b) Alternate representation of OHIP2PATID

Figure 2.3: Mapping tables

shown in Figure 2.2(b). We can use a valuation function that converts a name value in the FDB database corresponding to a name value in the SDB database.

When the values of attributes in two peers are same then we can also use variables for associating values. The representation of such a mapping table is called an identity mapping table. An example of identity mapping tables is given as follows.

Example 7 *Consider the database instance of Figure 1.1. Assume that the database FDB uses the same patient identifiers as the SDB database. There are two alternative mapping tables that we can use to represent the association between the two sets of patient identifiers. The two tables are shown in Figure 2.3. Mapping table OHIP2PATID in Figure 2.3(a) contains a set of mappings of the form (id; id), where id is a patient identifier in the FDB and SDB databases. Alternatively, we can construct a more succinct, data independent, mapping table OHIP2PATID containing the single identity mapping, (v; v), where v is a variable. The alternate mapping table is shown in Figure 2.3(b).*

In general, if there are multiple mapping tables $M = \{m_1, m_2, \cdots, m_k\}$ then we can combine the tables into a single mapping table using the $\wedge$-operator [48]. Therefore, we

can apply the valuation ρ on M which we represent as $\rho(m_1, m_2, \cdots, m_k)$.

Definition 3 (Mapping Table) *Let X and Y be non-empty disjoint sets of attributes with domains $dom(X)$ and $dom(Y)$, respectively. A mapping table m from X to Y is a finite set of mappings over $X \cup Y$ that are any subset of $dom(X) \times dom(Y)$.*

Since a mapping table m from X to Y associates values from $dom(X)$ to $dom(Y)$, we can determine the set of $Y-$values with which a particular value $\mathbf{x} \in dom(X)$ is associated by the following definition.

Definition 4 ($Y-$values) *Let m be a mapping table from X to Y. We define $Y-$values, denoted as $Y_m(\boldsymbol{x})$, with which a particular value $\boldsymbol{x} \in dom(X)$ is associated as follows: $Y_m(\boldsymbol{x}) = \{y | \exists\ t \in m$ and there exists valuation ρ over m such that $\rho(t[X]) = \boldsymbol{x}$ and $\rho(t[Y]) = \boldsymbol{y}\}$*

Example 8 *Consider the mapping table $OHIP2PATID$ in Figure 2.1. It is not hard to see that $PATID_{OHIP2PATID}(OHIP : 233GA) = \{235 - 02 - 09\}$.*

We now explain how a mapping table creates valid associations of tuples between two relations.

Definition 5 (Association of Tuples) *Consider relations r_1 and r_2 with relation schemas $R_1[U_1]$ and $R_2[U_2]$, respectively, and also consider a mapping table $m(X, Y)$ from X to Y, where $X \subseteq U_1$ and $Y \subseteq U_2$. Consider a relation r_{12}, where $r_{12} = r_1 \times r_2$. We say that a tuple t_{12} in r_{12} associates a tuple $t_1 \in r_1$ to a tuple $t_2 \in r_2$ if there is a valuation ρ over m such that $t_{12}[X] \in \pi_X(\rho(m))$ and $t_{12}[Y] \in \pi_Y(\sigma_{X=t_{12}[X]}(\rho(m)))$.*

The intuition behind this definition is that a tuple $t_1 \in r_1$ such that $t_1[X] = \mathbf{x}$ is associated only to a tuple $t_2 \in r_2$ with respect to the mapping table $m(X, Y)$, for which

$t_2[Y] \in Y_m(\mathbf{x})$. Therefore, we can consider the mapping table m as a condition to filter relation r_{12} that is subset of r_{12} contains only the tuples that m associates the tuples of relations r_1 and r_2.

Now we define how a mapping table associates tuples in two peers P_i and P_j.

Definition 6 (Mapping Table Between Two Peers) *Consider two peers P_i and P_j in a data sharing system. Let S be a schema at a peer P_i and T be a schema at P_j. Assume two relation schemas $R_s[U_s] \in S$ and $R_t[U_t] \in T$ with corresponding relations r_s and r_t. Consider a mapping table $m(X, Y)$ over the attributes $X \subseteq U_s$ and $Y \subseteq U_t$, and a value $\boldsymbol{x} \in dom(X)$. A tuple $t_s \in r_s$ such that $t_s[X] = \boldsymbol{x}$ is associated, with respect to mapping table m, only to tuples $t_r \in r_t$ for which $t_r[Y] \in Y_m(\boldsymbol{x})$. More formally, we can represent with the following logical formula:*

$$\forall\ t \in m\ \forall\ t_s \in r_s[(t_s[X] = t[X]) \rightarrow\ \exists\ t_r \in r_t(t_r[Y] = t[Y])]$$

We observe that source-to-target schema constraints performs schema mappings between a source and a target, i.e., create structural relationship of data between the source and the target. However, it does not resolve data-level heterogeneity. On the contrary, source-to-target data-level constraints resolve data-level heterogeneity and logically relate tuples between the source and the target. In order to incorporate these two constraints, we require a semantic that combines both constraints. We call the combined constraints as data sharing constraints, denoted by Σ_{st}. A data sharing constraint can be represented with the following assertion:

$$\Sigma_{st} = q_S \rightsquigarrow_M q_T$$

where q_S and q_T are two queries, respectively over the source schema S, and over the target schema T. Intuitively, an assertion $q_S \rightsquigarrow_M q_T$ specifies that the concept represented

by the query q_S over the source schema corresponds to the concept in the target schema represented by the query q_T that are associated by the mapping tables M. Intuitively, the formula also tells us that how the concept in the source is converted in terms of the data vocabularies into the concept for the target. An assertion of a data sharing constraint can be specified using the following logical expression:

$$\Sigma_{st} = \forall_{\mathbf{xy}}[\exists_{\mathbf{w}}\phi(\mathbf{x},\mathbf{w}) \wedge \mathbf{y} \in Y_M(\mathbf{x}) \rightarrow \exists_{\mathbf{z}}\psi(\mathbf{y},\mathbf{z})]$$

The set of mapping tables $M = \{m_1, m_2, \cdots, m_k\}$ is called source to target data level constraints Σ_{st}^v.

Let $\Sigma_{st} : q_S \rightsquigarrow_M q_T$ be a source-to-target data sharing constraint, $q_S(I)$ denotes the set of tuples returned by the query q_S over the instance I with the source schema S. We denote the fact that a tuple $t \in q_S(I)$ satisfies the LHS constraint q_S of Σ_{st} by $t \vDash q_S$, or in general by $t \vDash LHS(\Sigma_{st})$. Similarly, the fact that a tuple t' in a target instance J satisfies the RHS constraint q_T of Σ_{st} is denoted by $t' \vDash q_T$ (or, in general, by $t' \vDash RHS(\Sigma_{st})$).

Considering the source-to-target data sharing constraints, we can define how two tuples are related in two peers P_i and P_j, assuming peer P_i has a source instance I with schema S and P_j has a target instance J with schema T.

Definition 7 (Related tuples) *Consider a source instance I and a target instance J with $\Sigma_{st} = q_S \rightsquigarrow_M q_T$ where $\Sigma_{st}^s = q_S \rightarrow q_T$ and $\Sigma_{st}^v = M = \{m_1(X_1, Y_1), m_2(X_2, Y_2), \cdots, m_k(X_k, Y_k)\}$. A tuple t_1 in I is related to a tuple t_2 in J with respect to $\Sigma_{st} = q_S \rightsquigarrow_M q_T$ if $t_1 \in q_S(I)$ i.e. $t_1 \vDash LHS(\Sigma_{st})$ and $t_2 \in q_T(J)$ i.e. $t_2 \vDash RHS(\Sigma_{st})$ such that $t_1[X_1, X_2, \cdots, X_k] \in \Pi_{X_1,X_2,\cdots,X_k}(\rho(m_1, m_2, \cdots, m_k))$ and $t_2[Y_1, Y_2, \cdots, Y_k] \in \Pi_{Y_1,Y_2,\cdots,Y_k}(\sigma_{X_1=t_1[X_1] \wedge X_2=t_1[X_2] \wedge \cdots \wedge X_k=t_1[X_k]}\rho(m_1, m_2, \cdots, m_k))$. When two tuples t_1 and t_2 are related wrt Σ_{st}, we denote it by $t_1, t_2 \vDash \Sigma_{st}$.*

R_1(A, B, C)

(a) Source S with a relation schema R_1

R_2(A, B, D)

(b) Target T with a relation schema R_2

A	B	C
a_1	b_1	c_1
a_2	b_1	c_2
a_3	b_3	c_3

(c) Source with a relation r_1

A	B	D
a'_1	b'_1	d_1
a'_2	b''_1	d_2

(d) Target with a relation r_2

Figure 2.4: Source and target database schemas and instances

Example 9 *Consider the database instances of a source S and a target T in Figure 2.4. Consider Σ_{st} between S and T as follows:*

$$\Sigma_{st} = \forall_{xyx'y'}(\exists_{z} R_1(x,y,z) \rightsquigarrow_M \exists_{w} R_2(x',y',w))$$

where $M = \Sigma^v_{st} = \{m_1, m_2\}$ which is interpreted by the mapping tables in Figure 2.5.

Let us find the tuples in instance I of S that are related with the tuples in instance J of T wrt Σ_{st}. Observe that the tuple $t = (a_1, b_1, c_1)$ in I is related to a tuple $t' = (a'_1, b'_1, d_1)$ in J since $t, t' \vDash \Sigma_{st}$, i.e., $t[A,B] = (a_1, b_1) \in \Pi_{R_1.A,R_1.B}\rho(m_1, m_2)$, and $t'[A,B] = (a'_1, b'_1) \in \pi_{R_2.A,R_2.B}(\sigma_{R_1.A=t[A] \wedge R_1.B=t[B]}(\rho(m_1, m_2)))$.

$R_1.A$	$R_2.A$
a_1	a'_1

(a) Mapping table m_1

$R_1.B$	$R_2.B$
b_1	b'_1

(b) Mapping table m_2

Figure 2.5: Mapping tables

Observation: Only using schema-level constraints, we are not able to link two databases unless the data vocabularies are same. For instance, consider Example 9, without the existence of mapping tables m_1 and m_2, it is not possible to say that the tuples $t = (a_1, b_1, c_1)$ in I and $t = (a'_1, b'_1, d_1)$ in J are related since the data vocabularies of two tuples are different. However, if the data vocabularies of the two tuples are same then

they satisfy the source-to-target schema constraints. By introducing mapping tables, a source can implicitly specify which data in the source relates to a data in the target. Therefore, mapping tables not only resolve data level heterogeneity but also impose constraints on data to be shared [25]. Suppose that a source doesn't want to share information about a certain tuple t in its instance. To accomplish this, it is sufficient to ensure that the mapping tables in Σ_{st}^{v} do not associate any data element of t, namely, there is no mapping $(\mathbf{x}, \mathbf{y})$ in the table $m(X, Y)$ such that $Y_m(t[X]) = \mathbf{y}$.

2.2 Data Sharing Settings

Definition 8 (Data Sharing Setting) *A data sharing setting between two peers, where one peer acts as a source and another peer acts as a target, is a quintuple* $\mathbf{D} = (S, T, \Sigma_{st}, \Sigma_s, \Sigma_t)$ *such that*

- *S is a source schema and T is a target schema;*
- *Σ_{st} is a finite set of source-to-target data sharing constraints;*
- *Σ_s, is a finite set of source constraints.*
- *Σ_t, is a finite set of target constraints;*

2.3 Data Sharing System

Although the definition of a data sharing setting involves two peers, it can be easily extended to a family of peers that share data in a peer to peer network. Depending on the roles, a peer may act as a source or as a target. The data sharing settings with multiple peers constitute a data sharing system.

A data sharing system is a set $\mathcal{P} = \{P_1, P_2, \cdots, P_n\}$ of n peers with autonomous pre-existing local database systems (LDBSs). Each peer P_i, $1 \leq i \leq n$, has its own database, denoted D_i, with its own schema. We assume that each peer P_i is responsible to maintain its database consistent with respect to its local integrity constraints independently from other peers, since the local database is independently designed.

We now define the different concepts related to a data sharing system.

Definition 9 (Peer) *A peer P_i in a data sharing system $\mathcal{N}$ consists of:*

- *a database D_i with its own schema. Note: if P_i acts as a source then its schema is called source schema S_i or if P_i acts a target then its schema is called target schema T_i.*
- *a set of local integrity constraints IC_i on D_i. Note: if P_i is a source then IC_i is treated as source constraints Σ_s or if P_i is a target then IC_i is treated as target constraints Σ_t.*
- *a finite set of source-to-target data sharing constraints from P_i to P_j, denoted by Σ_{ij}.*

Note: If P_i is a source and P_j is a target then Σ_{ij} is denoted by Σ_{st}. The set of data sharing constraints in a source peer P_i for all of its target peers P_j is denoted by Σ_i.

Definition 10 (Peer Data Sharing System) *A peer data sharing system $\mathcal{N} =< \mathcal{P}, \Sigma >$ consists of:*

- *a finite set $\mathcal{P} = \{P_1, P_2, \cdots, P_n\}$ of peers, and*
- *a set $\Sigma = \{\Sigma_1, \Sigma_2, \cdots, \Sigma_n\}$ of data sharing constraints.*

Definition 11 (Acquaintance Graph) *Let $\mathcal{P} = \{P_1, P_2, \cdots, P_n\}$ be a set of peers and $\Sigma = \cup_{i=1}^{n} \Sigma_i$ be a set of non empty data sharing constraints in a data sharing system $\mathcal{N}$. An acquaintance graph is a graph $\Gamma_{\mathcal{N}} = (V, ACQ)$, where $V = \mathcal{P}$ is a set of vertices and $ACQ \subseteq \mathcal{P} \times \mathcal{P}$ is a set of direct edges such that every edge $(P_i, P_j) \in ACQ$, $i \neq j$, is associated with data sharing constraints Σ_{ij}.*

Definition 12 (Acquaintee) *A peer P_j is an acquaintee of a peer P_i in $\mathcal{N}$ if there exists an edge from P_i to P_j in $\Gamma_{\mathcal{N}}$; $\mathcal{N}(P_i)$ denotes the set of acquaintees of P_i.*

Definition 13 (Accessible Peers) *A peer P_j is accessible from a peer P_i in $\mathcal{N}$ if there is a path in $\Gamma_{\mathcal{N}}$ from P_i to P_j; $\mathcal{A}(P_i)$ denotes the set of accessible peers of P_i.*

2.4 Summary

This chapter first presented some technical preliminaries about relational databases. Next, we formalized the data sharing settings and introduced our data sharing system model. In the next chapter, we state the problem of processing updates and transactions in peer data sharing systems.

Chapter 3

Problem Statement and Related Work

In this chapter, we state the algorithmic problems related to update and transaction processing mechanisms in a data sharing system. We also discuss related work, in addition to a preliminary discussion of the most important work related to this thesis that we discussed in Chapter 1.

3.1 Update Processing

Here we address some of the potential problems that need be solved for processing of updates in a data sharing system. Before addressing the problems, we first illustrate a general execution scenario of an update in a data sharing system.

3.1.1 Execution Scenario of Updates

In a data sharing system, when a user submits an update μ_i to a peer P_i, he/she needs only be aware of the local database schema and data vocabularies. When μ_i is submitted to P_i, it is executed at P_i and appropriate changes occur at the database D_i. Whenever a change occurs at P_i, the data in each acquaintee P_j of P_i need to be changed subject to the data sharing constraint Σ_{ij} in order to maintain data consistency between peers. Therefore, when μ_i is submitted to P_i for execution, P_i verifies whether μ_i is relevant for its acquaintees. The update μ_i is relevant to an acquaintee P_j if it is possible to translate μ_i for P_j by considering Σ_{ij}. If μ_i is relevant then it is translated for P_j using the mappings. After translating μ_i, P_i forwards the translated updates to its acquaintees. When an acquaintee P_j of P_i receives a translated update from P_i, it also executes the received update, and translates and forwards the translated updates to its acquaintees that are relevant to these translated updates. This *execute-and-forward* step is repeated in each of the relevant acquaintees, causing in turn further propagations of the update in the system. Note that during the propagation of an update in the system, a peer might receive an update through several paths, however, the peer executes that update only once. According to our update propagation protocol, when an update is initiated by a peer it gets a unique global identifier (GID). The GID is attached to the update message and remains fixed during the propagation into the network. This information facilitates a peer to discover the reception of an update multiple times from different paths.

Based on the execution scenario above, updates are classified into three categories, namely *local*, *remote*, and *global*.

local update: An update originating and executing in a peer P_i, is a *local* update with respect to P_i and is denoted μ_i^i. A local update μ_i^i is called *global update initiator*,

if μ_i^i needs to be executed over the system. For ease of presentation, we denote a local update μ_i^i as μ_i.

remote update: If a local update μ_i is translated for an acquaintee P_j then a translated version of μ_i is a *remote update* with respect to P_i, and is denoted μ_i^j. More precisely, a remote update refers to the execution of μ_i at peer P_j in a translated form. If μ_i is propagated from P_i to P_j, where $P_j \in \mathcal{A}(P_i)$ via a peer P_k then a remote update at P_j is denoted as $\mu_{i[k]}^j$. For ease of presentation, we omit the intermediate peers from the notation of a remote update. For example, the remote update $\mu_{i[k]}^j$ is represented as μ_i^j. A remote update is composed of $q \geq 1$ component updates which form a sequence of updates $\mu_i^{j,1}, \cdots, \mu_i^{j,q}$. Multiple components arise since an update may have multiple translations. Note that each remote update μ_i^j is defined with respect to the corresponding peer P_j.

global update: A *global* update $\mu_i^g = \{\mu_i, \mu_i^j, \mu_i^k, \cdots\}$ consists of an initiator μ_i executed at P_i and a set of *remote* updates μ_i^j, $1 \leq j \leq n$ that are executed at $P_j \in \mathcal{A}(P_i)$.

In our thesis, we consider the following problems for processing an update in a data sharing system.

3.1.2 Translation of Updates

For exchanging updates from a peer to its acquainted peers, first the update that is posed to a peer is translated in terms of the schema and data vocabularies of the acquainted peers. The translation should be such that insertions, deletions, and modifications of the tuples made by an update in a peer and by the translated version of the update in an acquaintee are related through the mappings between both peers. In our thesis, we consider this case of update translation between peers. Particularly, the problem that we consider is as follows:

Given: A data sharing setting $\mathbf{D} = (S, T, \Sigma_{ij}, \Sigma_s, \Sigma_t)$ between two peers P_i and P_j. Suppose an update μ is executed against the database D_i at P_i with schema S.

Find: Translation of μ into an update μ' against the database D_j at P_j with schema T, such that the execution of μ and μ' at P_i and P_j, respectively, leaves the databases D_i and D_j consistent (i.e., databases satisfy their local integrity constraints, Σ_s and Σ_t, respectively) and the tuples updated by μ and μ' in D_i and D_j, respectively, are related through the mappings Σ_{ij}.

Solution to this problem requires answering the following questions:

(i) When is an update μ at P_i relevant to a peer P_j?

(ii) What is the computational complexity of checking the relevancy of updates?

(iii) If μ is relevant to P_j, under what conditions μ' is a correct translation of μ?

(iv) What is a correct translation of μ into μ'.

(v) What is the computational complexity of computing a correct translation of a given update?

3.1.3 Update Propagation and Synchronization

Due to the dynamic nature of peers, the processing of updates in a data sharing system is a challenging task. First, we need to consider the dynamic behavior of peers. Moreover, a peer involved to executing an update is not aware of the global execution and termination of the update in the system. Therefore, we need a protocol that guarantees a successful termination of the execution of an update. Moreover, peers are autonomous in a data sharing system and there is no global control of the execution of updates. Therefore, during propagation of updates, different conflicting situations with respect to updates may occur which lead to data inconsistency to the system. Proposing solutions

to resolve these issues is another important objective of our thesis. We believe that optimistic approaches best suit the P2P settings which, by nature, involve autonomous data sources that must be allowed to query and update data simultaneously. Therefore, a further objective of our thesis is to investigate optimistic approaches for propagating and processing updates and adopt an approach that is best suited for data sharing systems.

3.2 Transaction Processing

With respect to processing of transactions, a data sharing system is similar to a conventional multidatabase system (MDBS) [57, 18] in the sense that each system consists of a collection of independently created local database systems (LDBSs), and the management of transactions is handled at both local and global levels. In an MDBS, global level transactions are issued to the global transaction manager (GTM), where they are decomposed into a set of global subtransactions to be individually submitted to the corresponding LDBSs. Local transactions are directly submitted to the local transaction management systems (LTMs). Each local transaction manager maintains the correct execution of both local transactions and global subtransactions at its site. It is left to the GTM to maintain the correct execution of global transactions.

In contrast, a data sharing system is built on a dynamic network of peers without a global transaction manager or controller. Moreover, global level transactions are initiated by any peer. If a transaction is submitted to a peer and needs to be executed over the network, the transaction is propagated from peer to peer. Note that when a user submits a transaction to a peer, he/she is only aware of the local database schema. The mappings between the peer where the transaction is active and other peers in the network determine the translation of the transaction and propagation and execution of the transaction to

other peers.

Although we assume that an LDBS of each peer guarantees serializability, concurrent transactions that execute in multiple peers may be serialized in different orders in different peers, resulting in an inconsistent execution of the transactions in the system. One of the objectives of our thesis is to propose a correctness criteria that ensures the consistent execution of concurrent transactions in a data sharing system. Second objective is to analyze the execution of transactions considering the dynamic behavior of peers. Due to the dynamic nature of peers, several situations can occur during the execution of transactions in the system. For examples, a peer may become offline when a transaction is active in the system, a peer may fail due to power fails or the local database system crashes, or a transaction is executed successfully in a peer but the transaction may fail in the acquaintee of the peer. Examples of a transaction failure are a transaction abort to timeout, or a failure to pass the validation test by the local transaction manager. In our thesis, we mainly focus on the transaction failures and how the failures cause problems for the consistent execution of transactions.

3.3 Related Work

In traditional relational database systems, two well-known update propagation scenarios exist, namely the view maintenance [36, 37, 15, 16] and view update problems [47, 7, 82, 23, 65]. In the view maintenance problem, updates made to base relations are propagated to the materialized view and in the view update problem updates made to a materialized view are propagated to the respective base relations. Authors in [35] consider the view maintenance concept for exchanging updates in a data exchange setting [29]. In a data exchange setting, one peer acts as a source (data provider) and another peer acts as a

target (data receiver). The ultimate goal of a data exchange setting is to fill a target with the data from sources that satisfy the mappings between the sources and the target. Hence, the update exchange mechanism is roughly analogous to the view maintenance problem, i.e., a target peer gets updated in response to the updates made at a source peer. Authors in [67] also propose a framework for managing updates in a large-scale data sharing system with the concept of view maintenance. In the system, a peer which acts as a data receiver, its schema is defined as a view of the schema of data providers. Authors in [13, 14] also propose a semantics for maintaining data consistency in peer data exchange systems. The inconsistency between databases of peers is managed at query time using the repair semantics [6]. However, in our thesis, we investigate an update exchange mechanism in data sharing systems where each peer typically contributes its own data and may import data from other related peers. Each peer has its own preexisting database and the schema of a peer is not defined as a view of another peer's schema. Peers are related mainly with data sharing constraints or mappings that store related data. The mappings between peers relate two objects (e.g. tuples) between peers. Therefore, an update to an object in a peer that satisfies the mappings is propagated to acquainted peers. Our work is mainly based on an existing data sharing system, Hyperion [5, 75] that uses mapping tables [48] to relate data located in different peers. In this system, mapping tables provide a framework of simple schema mappings and data mappings between peers. Hyperion provides core data sharing features such as schema and data mappings and query processing [49, 50, 59], which we leverage to carry out our work. There are also other related work for processing of updates in peer to peer systems. However, the work mainly focuses on update propagation, conflict resolutions, and processing mechanisms. In the following we discuss some related work.

With respect to related work, peer-to-peer updates have been analyzed in [51, 30, 1, 46, 58]. For example, Kantere et al. [45, 46] describe techniques related to establishing and abolishing acquaintances of peers in P2P systems through triggers. The approach specifies update exchange policies on-the-fly based on the constraints between peers. The update exchange supports through firing of triggers that satisfies the constraints. However, the triggers are explicitly written for each constraint. In this case, the administrator of a peer explicitly mentions in the system which local update should fire which trigger for a remote peer. In our approach, we present the update exchange strategy where the updates are generated automatically to execute in remote peers that are based on the data sharing constraints or mappings between peers. The translation of updates for remote peers is done on-the-fly. Also, authors in [45, 46] assume that when a peer joins a network, it needs to register to certain interest groups in the network and a standard schema exists for each interest group. The standard schema is known to all members of the group. Hence, the standard schema becomes a bottleneck of the system. It increases the complexity for maintaining the standard schema and restricts the change of local database schemas of peers freely.

Authors in [24] present an approach for automatically detecting erroneous mappings in peer data management systems. For resolving mapping inconsistency they built a probabilistic model by taking the advantage of transitive closures of mapping operations to confront local belief on the correctness of a mapping against evidences gathered around the network. However, the model is for detecting and correcting errors of mappings among peers. In our work, we assume that mappings are correct. Based on this assumption, we propose an approach for maintaining data consistency between peers by propagating updates.

In coDB [30] peer database system, peers are interconnected by means of GLAV [56] coordination rules among their schemas. A peer can fetch data from its neighbors by using schema level translations involving the GLAV mappings that are in place between peers. The GLAV mappings play the role of coordination rules. A global update in the network is started when a peer sends out that global update, along with the definitions of appropriate coordination rules request to all acquaintees.

P-Grid [1] addresses an update mechanism in a structured network that support self-organization. The update mechanism uses a rumor spreading algorithm to scale and provides probabilistic guarantees for maintaining consistency between peers. However, it only considers updates at the file level considering that a single peer can update a file and changes are propagated to other peers. There are other update mechanisms in the file sharing environment. Authors in [84] propose a replicated database system, called Bayou, to support collaboration among users in a weakly connected network. Updates are broadcast between sites using an epidemic propagation protocol. Bayou schedules the updates using timestamp ordering. In this system, each update is assigned a tentative timestamp when it arrives to a site. Bayou first executes updates in their tentative order, then rolls back and replays them in final order. If the update is accepted by a primary site, the final timestamp is assigned to the update. Hence, the final execution of updates relies on a primary site that enforces a global continuous order on a growing prefix of history.

Authors in [51] introduce a log-based reconciliation mechanism of update operations in weakly connected systems. The system is called IceCube. In IceCube, concurrent update operations are merged at one site for reconciliation. Therefore, all concurrent operations must be sent at one site for merging. Then, the merged log must be dispatched to all

sites. Consequently, all sites must be connected to the merging site during reconciliation and wait until reconciliation is finished.

In the APPA framework [58], a distributed semantic reconciliation algorithm is proposed for reconciling conflicting updates in peer data sharing systems. The algorithm supports multi-master replication and assures consistency based on application semantics. The solution proposed in [58] rests on a mechanism that replicates data in different peers.

With respect to related work, peer-to-peer transactions have been analyzed in [80, 40, 73, 4, 85]. We discuss them in the following.

In [80, 40, 85], a concept of transaction processing in P2P environment is presented. The authors proposed a decentralized concurrency control for transactions relying on a decentralized serialization graph. Each peer and each transaction maintain a local serialization graph. The serialization graph of the peer reflects the dependencies of the transactions that invoke service calls on that peer, whereas the serialization graph of the transaction includes the dependencies in which the transaction is involved. The approach resolves conflicts of transactions by exchanging the serialization graph that basically changes the transaction semantics in conventional database management systems. However, in our work, we keep the execution semantics of transactions like in a conventional database management system.

In [73], a preliminary approach is presented for agent-based transaction processing in a decentralized P2P network. The main interest of the work is on a cooperative information system based on a P2P model. The model consists of a multi agent system with four components (wrapper, mediator, facilitator, and planner) which are responsible for the management and control of transactions composed by data management operations (read,

write, delete) and their outcomes. However, the approach is still preliminary and the lack of details does not permit of comparison with our approach.

In [4], authors present a preliminary proposal for peer-to-peer e-business transaction processing systems. More specifically, the approach focuses on requirements analysis on different aspects of the collaboration and transaction procedure. However, it lacks precise semantics of transactions and does not describe the execution semantics of transactions.

There are simulation tools for simulating peer-to-peer systems. However, they are not directly related for simulating peer data sharing systems. In the following, we discuss some of the work. Authors in [68] discuss the current trend of simulation tools with respect to simulation usage in P2P research, and present the state of the art of P2P simulations. We notice that most of the simulators are dedicated for content distribution and file sharing systems. In order to evaluate a peer data sharing system, we need different resources, for example, databases, mappings between peers, and acquaintances. The existing tools do not support or provide these facilities. Moreover, tools should support database related actions (e.g. query, update, and transaction). Our goal is to present a tool that can be used to evaluate a peer data sharing system which is unable with the existing tools. We describe some of the P2P simulators in the following.

NS-2 [71] is a popular network simulator that best suits for simulating packet switched networks and small scale networks. Adding new modules is not straightforward, because of it's complex module structure [54].

The simulator [79] is specialized to file-sharing simulations. It mainly models the content distributions, query activity, download behavior etc. Simulations proceed in query cycles representing the time period between issuing a query and receiving a response. Similar to our approach, queries are passed into a queue and handled on FIFO basis.

NeuroGrid [44] is a single-threaded, non-parallel Java-based simulator designed for supporting comparative resource search simulations between FreeNet, Gnutella, and NeuroGrid systems.

3.4 Summary

In this chapter, we stated the problem that we investigate in this thesis. We first presented issues related to translation of updates between peers. Next, we spelled out the challenges raised when updates are propagated in data sharing systems. We also discussed the main issues for ensuring a consistent execution view of concurrent transactions in peers. In the next chapter, we present our proposed update translation mechanism for peer data sharing systems.

Chapter 4

Translation of Updates

In this chapter, we investigate the issues related to the translation of updates between peers. Before discussing these issues, we first introduce a semantics for update execution in a data sharing system. Particularly, we illustrate what it means to execute an update at a peer and the translated versions of that update at the peer's acquaintees. We also present the proposed update translation algorithm in this chapter.

4.1 Update Language

To present update translation issues, we adopt the language Insert-Delete-Modify (IDM) transaction model [2] to describe update operations. The three different updates we consider are insert, delete, and modify. Accordingly, an update over a relation schema R is an insertion, deletion, or modification of tuples in $R's$ relation r. An *insertion* is an expression $ins(t)$, where t is a tuple over $att(R)$ and $att(R)$ denotes the set of attributes in R. Here, $ins(t)$ inserts the tuple t into r. A deletion operation removes from r all the tuples satisfying some stated set of conditions. Formally, a *deletion* update is an

$R_1.B$	$R_2.B$
b_1	b_1'
b_1	b_1''

Figure 4.1: Modified mapping table m_2

expression $del(C)$, where $C = c_1 \wedge c_2 \wedge \quad \wedge c_n$ is a conjunctive boolean expression, where each $c_i, 1 \leq i \leq n$, is an atomic formula of the form $(A = a)$, where $A \in att(R)$ is an attribute name and a is a constant. The atomic formula c_i is positive [2], i.e., it has no negation. Here, $del(C)$ removes from r all the tuples satisfying each condition in C. Finally, a *modification* update is an expression $mod(C \rightarrow C')$, where C, C' are boolean expressions, with C' containing only equalities $A = c$, where A is an attribute name and c is a constant. This update selects all the tuples from r satisfying C and then, for each such tuple and each equality $A = c$ in C', it sets the value of A to c. Note that the updates that we consider do not support negation in their where clauses. The reason is that mapping tables encode only positive equality [48].

4.2 Update Examples

In this section we illustrate some examples of update operations wrt the chosen update language.

Example 10 (Modify update) *Consider Figure 2.4. In order to distinguish the attribute A in P_1 and P_2, we rename A to A' in P_2. Similarly, attribute B in P_2 is renamed to B'. This change is also considered in all the subsequent examples. Suppose the mapping table m_2 is modified as shown in Figure 4.1 and relations R_1 and R_2 contain extra attributes E and E', respectively. Also, assume that the domain of the attributes E and E' are same; thus, we include an identity mapping table $m_3(E, E')$. Suppose that*

the following update is submitted to P_1.

$$\mu_1 = mod(\{A = a_1, B = b_1\} \rightarrow \{E = e_1\});$$

Based on the mapping tables m_1, m_2, *and* m_3, $A'_{m_1}(a_1) = \{a'_1\}$, $B'_{m_2}(b_1) = \{b'_1, b''_1\}$, *and* $E'_{m_3}(e_1) = \{e_1\}$. *Therefore, the corresponding translated updates for* P_2 *are as follows:*

$$\mu_1^{2,1} = mod(\{A' = a'_1, B' = b'_1\} \rightarrow \{E' = e_1\});$$
$$\mu_1^{2,2} = mod(\{A' = a'_1, B' = b''_1\} \rightarrow \{E' = e_1\});$$

Example 11 (Delete update) *Consider the following delete update on database at* P_1.

$$\mu_1 = del(B=b_1);$$

By analyzing the above update, only the mapping table $m_2(B, B')$ *is required to translate* μ_1, *where* $Y'_{m_2}(b_1) = \{b'_1, b''_1\}$. *Therefore,* μ_1 *is translated for peer* P_2 *as follows:*

$$\mu_1^{2,1} = del(B' = b'_1);$$
$$\mu_1^{2,2} = del(B' = b''_1);$$

4.3 Update Execution Semantics

Definition 14 (Local Execution) *Given a local database* D_i *of* P_i, *the result* $upt(\mu_i, D_i)$ *(upt stands for updated tuples) by a local update* μ_i *is a set of tuples* t^+ *(inserted tuples) and* t^- *(deleted tuples) such that*

- $t^+ \cap t^- = \emptyset$
- $\mu_i(D_i) \models IC_i$, *where* $\mu_i(D_i) = D_i \cup t^+ - t^-$

Note that in this execution semantics we do not consider the set of modified tuples since the modified tuples can be logically represented with a combination of deletion and insertion of tuples.

Definition 15 (Global Execution) *Given a local database D_i of P_i, the result $upt(\mu_i^g, D_i)$ by a global update μ_i^g is made of two sets of inserted tuples $t^+, t^{+\prime}$ and two sets of deleted tuples $t^-, t^{-\prime}$ such that*

- $t^+, t^- \in upt(\mu_i, D_i)$, $t^{+\prime}, t^{-\prime} \in upt(\mu_i^j, D_j)$, *and* $\mu_i^j \in \mu_i^g$ *is a translation of* μ_i *for each accessible peer* $P_j \in \mathcal{A}(P_i)$,
- $t^+ \cap t^- = \emptyset$,
- $\mu_i(D_i) \vDash IC_i$, *where* $\mu_i(D_i) = D_i \cup t^+ - t^-$,
- $t^+, t^{+\prime} \vDash \Sigma_{ij}$ *and* $t^-, t^{-\prime} \vDash \Sigma_{ij}$,
- $\mu_i^j(D_j) \vDash IC_j$, *where* $\mu_i^j(D_j) = D_j \cup t^{+\prime} - t^{-\prime}$.

4.4 Relevant Updates

There are cases where the execution of a local update at a peer has no effect to its acquaintees with respect to the data sharing constraints in place. We call these updates *irrelevant* wrt these acquaintees; otherwise called *relevant.* Our goal is to find a correct translation of a relevant update. We demonstrate two approaches for checking whether an update is relevant or not. The first approach considers the semantics of update operations by taking into account the updated tuples caused by update operations. The second approach is syntactic where update relevancy is determined by a syntactic analysis

of update operations (i.e., update values, attributes, conditions, etc. mentioned in update operations).

We describe the approaches considering the settings in Example 9. Also consider the scenarios of Figure 2.4, 2.5, and the mapping table in Figure 4.1.

4.4.1 Approach 1 (Semantic Approach)

Before describing the approach for checking relevancy of updates, we first define the related notions.

Definition 16 (Association of a Tuple) *A tuple t has associations with respect to a mapping table $m(X, Y)$, if $Y_m(t[X]) \neq \emptyset$.*

Definition 17 (Valid/Invalid Tuple) *A tuple t is valid with respect to a data sharing constraint Σ_{st} if $t \models LHS(\Sigma_{st})$ and t has associations wrt the mapping tables M in Σ_{st}; t is invalid otherwise.*

Assume an update $\mu_1 = del(C)$ is processed in P_1. The update deletes tuples from R_1 that satisfy the boolean expression C. Consider that μ_1 deletes a tuple $t = (a_1, b_1, c_1)$ from R_1 that satisfies the expression C. If t is invalid with respect to Σ_{st} then deleting t from R_1 has no effect on relation R_2 of P_2.

Now, we define under what conditions an update at P_i is relevant to an acquaintee P_j of P_i.

Definition 18 (Relevant Update) *An update μ_i at P_i is relevant to an acquaintee $P_j \in \mathcal{N}(P_i)$ if there is at least a tuple t that belongs to $upt(\mu_i, D_i)$ that is valid wrt Σ_{ij}; μ_i is irrelevant otherwise.*

Example 12 (Relevant update) *Consider two peers P_1 and P_2 with relations R_1 and R_2, respectively and the data sharing setting between the peers as illustrated in Example 9. Assume an update $\mu_1 = del(A = a_1)$ is executed at P_1. Notice that μ_1 deletes a tuple $t = (a_1, b_1, c_1)$. We have $t \models LHS(\Sigma_{st})$ and t has associations with respect to the mapping tables $M = \{m_1, m_2\}$ i.e., $A'_{m_1}(t[A]) = a'_1$ and $B'_{m_2}(t[B]) = b'_1$. Therefore, μ_1 is a relevant update to P_2.*

Example 13 (Irrelevant update) *Suppose an update $\mu_1 = del(A = a_2)$ is executed at P_1. Notice that μ_1 deletes a tuple $t = (a_2, b_1, c_2)$. Tuple t is invalid since t has no association with respect to the mapping tables $M = \{m_1, m_2\}$, i.e., $A'_{m_1}(t[A]) = \emptyset$. Therefore, μ_1 is an irrelevant update to P_2.*

Theorem 1 (Relevant update) *For a given update μ_i at P_i with a local database D_i and data sharing constraints $\Sigma_{ij} = q_S \rightsquigarrow_M q_T$ corresponding to a peer P_j, the problem of determining whether μ_i is relevant to P_j, is in co-NP.*

Proof 1 *From Definition 18, we see that an update μ_i at P_i is relevant to P_j if at least $\exists\ t \in upt(\mu_i, D_i)$ that is valid wrt Σ_{ij}. Therefore, verifying the relevancy of an update means checking the validity of t wrt Σ_{ij}. Note that validity of a tuple is determined using the mapping tables M in Σ_{ij}. This turns out to the problem whether t is a certain answer of a query Q posed to M, i.e., $t \in Q(M)$. Note that Q is formed with the condition mentioned in μ_i. Consider I as an instance stores all the tuples is the result of a valuation ρ on M with respect to D_i and satisfying Σ_{ij}.*

Assume that t is not in a certain answer of query Q in M. Then there is M with $I \subseteq \rho(M)$ and t is not in $Q(M)$. Let n be the total number of tuples in I and let k be the number of mapping tables in M. Each tuple in I can be generated by at most

k tuples in M. Therefore, there is a $M' \subseteq M$ with at most nk tuples such that still $I \subseteq \rho(M')$. Because t is not in $Q(M)$, t is also not in $Q(\rho(M'))$. It follows that there is M' whose size is polynomially bounded in the size of I such that $I \subseteq \rho(M')$, and t is not in $Q(\rho(M')$. Moreover, checking that $I \subseteq \rho(M')$ and that t is not in $Q(\rho(M'))$ can be done in polynomial time. ■

4.4.2 Approach 2 (Syntactic Approach)

In this approach, update relevancy is determined through a syntactic analysis of update operations.

Definition 19 (Relevant Mapping Table) *Let $att(\mu)$ denotes the set of attributes in an update μ. A mapping table $m(X, Y)$ is relevant to μ if $X \subseteq att(\mu)$.*

Example 14 (Relevant mapping table) *Assume an update $\mu_1 = del(A = a_1)$ at P_1. In this case, $m_1(A, A')$ is relevant to μ_1 since $A \subseteq att(\mu_1)$, where $att(\mu_1) = \{A\}$*

Definition 20 (Relevant Update) *Let μ be an update and $\mathcal{R}$ be the set of relevant mapping tables for μ. Let $v(X)$ denotes the value that is associated with attribute X in μ. An update μ_i at P_i is relevant to an acquaintee $P_j \in \mathcal{N}(P_i)$ if $Y_{m_i}(x_i) \neq \emptyset$, where $x_i = v(X_i)$, for each relevant mapping table $(m_i(X_i, Y_i))_{i=i\cdots k} \in \mathcal{R}$.*

Example 15 (Relevant update) *Consider Example 14. Here, $att(\mu_1)$ is $\{A\}$ and $a_1 = v(A)$. Only the relevant mapping table corresponding to μ_1 is $m_1(X, X')$, since $A \subseteq att(\mu_1)$. Update μ_1 is relevant for P_2 since $A'_{m_1}(a_1) = \{a'_1\}$.*

Theorem 2 (Relevant update) *For a given update μ_i at P_i and $\mathcal{R}$ be the set of relevant mapping tables in data sharing constraints Σ_{ij} corresponding to a peer P_j. Checking for relevancy of μ_i for P_j can be done in polynomial time.*

Proof 2 *Suppose that* $\mathcal{X} = \{X_1, \cdots, X_k\}$ *are the attributes in* $att(\mu_i)$. *The set relevant mapping tables with respect to the attributes in* $\mathcal{X}$ *is* $\mathcal{R} = \{m_1(X_1, Y_1), \cdots, m_k(X_k, Y_k)\}$. *From the Definition 20, relevancy of an update* μ_i *at* P_i *to* P_j *depends on the existence of values* $v(X_i)$ *in the corresponding mapping table* $m_i(X_i, Y_i)$. *Therefore, verifying relevancy of an update means checking the existence of* $v(X_i)$ *in the corresponding mapping table* $m_i(X_i, Y_i)$. *Checking that* $v(X_i)$ *is in* $m_i(X_i, Y_i)$ *can be done in polynomial time since size of* $m_i(X_i, Y_i)$ *and* $|\mathcal{X}|$ *are polynomially bounded.* ■

4.5 Correctness of Update Translations

In this section, we introduce the notion of correctness for translating updates. Correctness ensures that the updated tuples in two peers are related by the data sharing constraints between them. Correctness is a property that every translation must satisfy.

Definition 21 (Correctness) *Given updates* μ_i *and* μ_i^j *over databases* D_i *and* D_j *of peers* P_i *and* P_j, *respectively. We say that* μ_i^j *is a correct translation of* μ_i *with respect to* $\Sigma_{ij} = q_S \rightsquigarrow_M q_T$, *where* $M = \{m_1(X_1, Y_1), \cdots, m_k(X_k, Y_k)\}$, *if*

(1) μ_i *is relevant to* P_j,

(2) for every tuple $t'[Y_1, \cdots, Y_k] \in upt(\mu_i^j, D_j)$, $\exists\, t \in upt(\mu_i, D_i)$ *such that* $t \models LHS(\Sigma_{ij})$ *and* t *has associations wrt* M, *i.e.* $t'[Y_1, \cdots, Y_k] \in \pi_{Y_1,\cdots,Y_k}(\sigma_{X_1=t[X_1]\wedge\cdots\wedge X_k=t[X_k]}(\rho(m_1(X_1, Y_1), \cdots, m_k(X_k, Y_k))))$.

The intuition of the second condition is that for each tuple t' updated by μ_i^j at D_j there exists a tuple t in the updated tuples by μ_i at D_i, where t and t' are related by the data sharing constraint Σ_{ij}, i.e. $t, t' \models \Sigma_{ij}$ between P_i and P_j.

Example 16 (Correct translation) *According to the definition of correctness, update* $\mu_1^2 = del(A' = a_1')$ *at* P_2 *is a correct translation of an update* $\mu_1 = del(A = a_1)$ *at* P_1. *The translation is correct since* μ_1 *is relevant to* P_2 *(according to case 1 and 2). Also, for the tuple* $t' = (a_1', b_1', d_1) \in upt(\mu_1^2, D_2)$ *there exists a tuple* $t(a_1, b_1, c_1) \in upt(\mu_1, D_1)$ *where* $t \vDash LHS(\Sigma_{st})$ *and* $t'(a_1', b_1') \in \pi_{A',B'}(\sigma_{A=a_1 \wedge B=b_1}(\rho(m_1(A, A'), m_2(B, B'))))$.

Example 17 (Incorrect translation) *Consider Examples 4 and 7. Here,* μ_1^2 *is an incorrect translation of* μ_1 *(although* μ_1 *is relevant to* P_2*), since there exists a tuple* $t_2'(a_1', b_1'', d_2) \in upt(\mu_1^2, D_2)$, *but* $t_2'(a_1', b_1'') \notin \pi_{A',B'}(\sigma_{A=a_1 \wedge B=b_1}(\rho(m_1(A, A'), m_2(B, B'))))$, *though* $t(a_1, b_1, c_1) \in upt(\mu_1, D_1)$.

We observe that the translation of a relevant update is not always correct. If the translation does not satisfy the correctness criteria then the execution semantics of updates is violated. As a result, the databases of peers is inconsistent considering the data sharing constraints. In the following, we discuss the instances of incorrect translation of updates despite the fact that the updates are relevant.

Example 18 (Incorrect translation (Approach 1)) *Suppose an update* $\mu_1 = del(B = b_1)$ *is executed at* P_1. *Observe that* μ_1 *deletes tuples* $t_1 = (a_1, b_1, c_1)$ *and* $t_2 = (a_2, b_1, c_2)$. *Tuple* t_1 *is valid since* $t_1 \vDash LHS(\Sigma_{st})$ *and* $A'_{m_1}(t_1[A]) = a_1'$, $B'_{m_2}(t_1[B]) = \{b_1', b_1''\}$. *Since* μ_1 *is relevant, therefore, its translated versions for* P_2 *are* $\mu_1^{2,1} = del(B' = b_1')$ *and* $\mu_1^{2,2} = del(B' = b_1'')$. *Observe that the translated updates delete tuples* $t_1' = (a_1', b_1', d_1)$ *and* $t_2' = (a_2', b_1'', d_2)$. *Although,* μ_1 *is relevant but it's translation produces incorrect translation since* $t_1, t_2' \nvDash \Sigma_{st}$ *or* $t_2, t_2' \nvDash \Sigma_{st}$. *Hence, we observe that translation of a relevant update is not always correct. On the contrary, if we consider* μ_1 *as an irrelevant update then* t_1' *will not be deleted even though* $t_1, t_1' \vDash \Sigma_{st}$.

Example 19 (Incorrect translation (Approach 2)) *Suppose an update* $\mu_1 = del(B = b_1)$ *is executed at* P_1. *Here,* $att(\mu_1) = \{B\}$, *the relevant mapping table is* $m_2(B, B')$, *and* $b_1 = v(B)$. *Notice that* μ_1 *is relevant to* P_2 *since* $B'_{m_2} = \{b'_1, b''_1\}$. *The translated update* μ_1^2 *of* μ_1 *is* $del(B' = B'_{m_2}(b_1))$, *i.e.,* $\mu_1^{2,1} = del(B' = b'_1)$ *and* $\mu_1^{2,2} = del(B' = b''_1)$. *Notice that tuples deleted by* μ_1^2 *are* $t'_1 = (a'_1, b'_1, d_1)$ *and* $t'_2 = (a'_2, b''_1, d_2)$. *Observe that there does not exist a tuple* $t \in upt(\mu_1, D_1)$ *for which* $t, t'_2 \models \Sigma_{st}$. *Hence,* μ_1^2 *is an incorrect translation of* μ_1 *even it is relevant to* P_2.

Observe that the incorrect translation results if there are one to many mappings in a mapping table. However, the incorrect translation results even though there is one to one mapping of a data item in mapping tables. An example is given below.

Example 20 (One to one mapping) (Approach 2)) *Consider the database instances of Figure 2.4. Assume that a tuple* (a'_4, b'_1, d_4) *exists in* r_2. *Suppose an update* $\mu_1 = del(B = b_1)$ *is executed at* P_1. *According to Example 19, the translated update* μ_1^2 *of* μ_1 *is* $del(B' = B'_{m_2}(b_1))$, *i.e.,* $\mu_1^{2,1} = del(B' = b'_1)$. *Notice that tuples deleted by* μ_1^2 *at* P_2 *are* $t'_1 = (a'_1, b'_1, d_1)$ *and* $t'_2 = (a'_4, b'_1, d_4)$. *However, there does not exit a tuple* $t \in upt(\mu_1, D_1)$ *such that* $t, t'_2 \models \Sigma_{st}$. *Hence,* μ_1^2 *is an incorrect translation of* μ_1 *even though it is relevant to* P_2.

From the above examples, we observe that it is not the case that the translation of a relevant update is always correct. Therefore, we need to impose extra constraints during translation of updates in order to ensure the correctness criteria. We observe that in order to produce the correct translation, a mapping table is needed that associates the key values between two acquainted peers. If there exists no such a mapping table then we assume that peers share the same key values. The problems of incorrect translation are resolved by applying this assumption.

Example 21 (Correct translation) *Assume that the mapping table $m_1(A, A')$ associates primary key values of two peers, where A and A' are the key attributes of relations R_1 and R_2, respectively. Consider Examples 4 and 7. The only valid tuple (according to Definition 17) that is updated by μ_1 is $t(a_1, b_1, c_1)$. Also m_2 is the relevant mapping table (according to Definition 14) and m_1 is the mapping table that associates primary key values. The primary key value that is associated with t is a_1. Therefore, μ_1 is augmented with an extra condition $A = a_1$. Hence, μ_1 becomes $\mu_1 = del(A = a_1, B = b_1)$. Finally, the corresponding translated update μ_2 is as follows:*

$$\mu_1^{2,1} = del(A' = a_1', B' = b_1')$$
$$\mu_1^{2,2} = del(A' = a_1', B' = b_1'')$$

If these two updates are allowed to execute at P_2, they delete the following tuple:

$$t' = (a_1', b_1', d_1)$$

Observe that μ_1^2 is a correct translation of μ_1 since $t, t' \vDash \Sigma_{st}$.

Example 22 (Correct translation (1 to 1 mapping)) *Again assume that the mapping table $m_1(A, A')$ associates primary key values of two peers, where A and A' are the key attributes of relations R_1 and R_2, respectively. Consider the update in Example 20. The only valid tuple (according to Definition 17) that is updated by μ_1 is $t(a_1, b_1, c_1)$. Here m_2 is the relevant mapping table (according to Definition 14) and m_1 is the mapping table that associates primary key values. The primary key value that is associated with t is a_1. Therefore, μ_1 is augmented with the extra condition $A = a_1$. Hence, μ_1 becomes $\mu_1 = del(A = a_1, B = b_1)$. Finally, the corresponding translated update μ_1^2 is as follows:*

$$\mu_1^2 = del(A' = a_1', B' = b_1')$$

If this update is allowed to execute at P_2, it deletes the following tuple:

$$t' = (a'_1, b'_1, d_1)$$

Observe that μ_1^2 is a correct translation of μ_1 since $t'(a'_1, b'_1) \in \pi_{A',B'}(\sigma_{A=a_1 \wedge B=b_1}(\rho(m_1(A, A'), m_2(B, B'))))$.

Proposition 1 *Given an update μ_i on relation R_i at peer P_i and data sharing constraints Σ_{ij} with respect to a peer P_j. Assume μ_i is relevant to P_j. Then, the translation of μ_i to μ_i^j is correct if a mapping table $m_k(X_k, Y_k)$ is considered for the translation that associates key values of the updated tuples made by μ_i and μ_i^j in R_i and R_j.*

Proof 3 *We shall proof that if m_k is considered for the translation μ_i to μ_i^j then μ_i^j is a correct translation of μ_i. By contradiction, we shall assume that μ_i^j is not a correct translation even m_k is considered for translation. This implies that the following two statements hold:*

(1) $t \in upt(\mu_i, D_i)$ such that $t[X_k] \in \pi_{X_k}\rho(m_k(X_k, Y_k))$ and $t' \in upt(\mu_i^j, D_j)$ but $t'[Y_k] \notin Y_{m_k}(t[X_k])$.

(2) $t \in upt(\mu_i)$ such that $t[X_k] \notin \pi_{X_k}\rho(m_k(X_k, Y_k))$ and $t' \in upt(\mu_i^j)$ but $t'[Y_k] \in Y_{m_k}(t[X_k])$.

First we show that the statement 1 can not be true. Since μ_i is relevant to P_j. Therefore, μ_i^j contains a condition $Y_k = y_k$, where $y_k = Y_{m_k}(t[X_k])$. Hence, either $t'[Y_k] \in \pi_{Y_k}(\sigma_{X_k=x_k}(\rho(m_k(X_k, Y_k))))$ or $t' = \emptyset$. This is the contradiction of the first statement.

It is obvious that the second statement can not be true, since μ_i is relevant to P_j it must satisfy that $t \vDash m_k$. Therefore, $t[X_k] \in \pi_{X_k}\rho(m_k(X_k, Y_k))$. This is the contradiction of the second statement. ■

4.6 Discussion

Note that the syntactic approach determines the relevancy of updates by considering only the mapping tables between peers and the values that are associated with the attributes mentioned in the updates. We notice that, in some cases, this approach fails to check the relevancy of an update properly even if each associated value $v(X)$ of each attribute X in the update has an association in the corresponding mapping table. The fail cases arise in two situations: (i) when an update affects tuples in the local database of a peer and the affected tuples have no association wrt the relevant mapping tables and (ii) when the relevant mapping tables contain mappings of value $v(X)$ for each associated attribute X in an update but the values of the affected tuples are not associated wrt the relevant mapping tables. In the following examples, we illustrate the situations.

Example 23 *Consider that the relation R_1 contains a tuple $t = (a_4, b_4, c_4)$ and the relation R_2 contains a tuple $t' = (a'_4, b'_4, d_4)$. Also assume that the mapping table m_2 contains a mapping (b_4, b'_4) and there exists no mapping between the values a_4 and a'_4 of relations R_1 and R_2. Suppose that the following update is posed at P_1.*

$$\mu_1 = del(B = b_4)$$

Observe that the tuple $t = (a_4, b_4, d_4)$ in R_1 is deleted by μ_1. However, t has no associations wrt the mapping table m_1, i.e., $A'_{m_1}(t[A]) = \emptyset$. Therefore, μ_1 should not be relevant for P_2. However, the syntactic approach considers μ_1 as a relevant update. According to the approach, first the relevant mapping tables are determined for an update. In this case, m_2 is the only relevant mapping table. Second, the approach looks for the associations of values in the relevant mapping tables for the values associated with the attributes mentioned in the update. In this update, B is the only attribute and the associated value

of B is b_4. Since, $B'_{m_2}(b_4) \neq \emptyset$, therefore, μ_1 is treated as a relevant update. However, if we consider the semantic approach, the update μ_1 is not considered as relevant since t does not have any associations wrt m_1 and m_2.

Example 24 *Consider that the relation R_2 contains a tuple $t' = (a'_4, b'_4, d_4)$ and m_2 contains a mapping (b_4, b'_4). Suppose that the following update is posed at P_1.*

$$\mu_1 = del(B = b_4)$$

According to the syntactic approach, μ_1 is a relevant update for P_2. Observer that m_2 is the only relevant mapping table for μ_1 and the values associated with the attribute B in μ_1 is b_4. Since, $B'_{m_2}(b_4) \neq \emptyset$, therefore, μ_1 is treated as a relevant update. However, μ_1 should not be considered as relevant since μ_1 does not delete any tuple in R_1. However, if we consider the semantic approach, the update μ_1 is not considered as relevant.

Authors in [49] consider the syntactic approach for translating queries between peers. However, for translation of updates, if we consider the syntactic approach, in some cases irrelevant updates are considered for translation, as illustrated in examples 16 and 17. Meanwhile, the semantic approach guarantees the verification of relevant updates since it mainly considers the affected tuples in the local peer database. Therefore, we select the semantic approach for checking the relevancy of updates in order to translate updates.

4.7 Translation Algorithm

Considering the correctness of update translation, we make the following assumptions for the proposed update translation algorithm.

- To account for translation of updates, we restrict our attention to the closed-world semantics [48] of mapping tables. In the closed-world semantics we are able to constrain the $Y-$values associated with specific $X-$values. Under this semantics, we can specify the complete set of associations between the X and $Y-$values. Thus, for translating updates, only the mappings that are explicitly mentioned in the mapping tables are considered.

- The language that we consider does not support negation. The reason is that mapping tables encode only positive equality [48].

An update may affect large number of tuples to the local database of a peer and the affected tuples may satisfy the data sharing constraints. Hence, the update needs to be processed at the peer's acquaintees. In order to process the update at acquaintees, one choice is to translate the affected tuples in terms of the data vocabularies of the acquaintees and propagate the translated tuples to the acquaintees. This requires much computation time (e.g. translating each tuple into vocabularies of acquaintees plus transmission of large number of messages). Thus, it is not feasible to send updated tuples to acquaintees. Another choice is simply to translate update operations in terms of the vocabularies of acquaintees. It is rather worth to translate an update operation and forward its translated version since the size of update operations are typically short compare to the size of large number of updated tuples. Note that the translation should be such that the translated update operations only update the tuples at the peer's acquaintees that satisfy the data sharing constraints and tuples affected at the peer and its acquaintees are related by the data sharing constraints between them. In this section, we propose such an update translation algorithm. The translation algorithm is depicted in Algorithm 1.

Each peer applies the algorithm for translating updates for its acquaintees. The algorithm has two parameters: (i) an input update μ_i over the local schema of a peer P_i and (ii) the data sharing constraints Σ_{ij} between P_i and an acquaintee P_j of P_i. The algorithm outputs a translated update μ_i^j generated from μ_i for P_j considering Σ_{ij}. In the beginning of translation, the algorithm checks the type of updates (deletion, modification, or insertion). Due to the syntactic similarity of *deletion* and *modification* updates, the translation strategies are almost similar. Therefore, we first discuss the strategies for translating *deletion* and *modification* updates. Next, we discuss the strategy for *insertion* updates, which is treated in different way.

Consider an update μ_i at P_i posed on a relation schema R of P_i. In the first phase, the algorithm first checks the relevancy of μ_i for the acquaintees of P_i. If μ_i is relevant for an acquaintee P_j, it is then translated in the second phase. The translation strategy is described below.

- At first the algorithm computes a set of relevant mapping tables $\mathcal{R}=\{m_1, \cdots, m_k\}$ for translating μ_i. Assume m_p is the mapping table that associates key values. If $m_p \notin \mathcal{R}$ then m_p is added in $\mathcal{R}$. Next the algorithm determines the valid tuples $T = \{t_1, \cdots, t_m\}$ from the affected tuples $upt(\mu_i, D_i)$ wrt data sharing constraints Σ_{ij}. If $T \neq \emptyset$ then μ_i is relevant to P_j. For finding the valid tuples, first a temporary relation R_C is produced with the tuples from R that satisfy the boolean expression C mentioned in μ_i. After that the algorithm extracts tuples from R_C that are valid (according to the definition 17) considering Σ_{ij} and stores the result in a relation R_v. This process is evaluated using the expression $R_v = R_C \bowtie m_1 \bowtie \cdots \bowtie m_k$. For translating μ_i into μ_i^j, the algorithm takes projection on Y attributes from R_v. Note that this set of Y attributes are in the mapping tables in $\mathcal{R}$. The result is

stored in another temporary relation R_Y.

- If μ_i is a *deletion* operation then the algorithm converts each tuple $t \in R_Y$ into an update $\mu_i^{j,p}$=$del(Y_1 = y_1, \cdots, Y_l = y_l)$, where $(1 \leq p \leq k)$, $k = |R_Y|$, $l = |\mathcal{R}|$, $\{Y_1, \cdots, Y_l\}$ are attributes in R_Y, and y_q is a constant of $t[Y_q]$, where $1 \leq q \leq l$. This set of updates $\mu_i^j = \mu_i^{j,1}, \cdots, \mu_i^{j,p}$ is the translated update for P_j.

- If μ_i is a *modification* operation then algorithm translates the boolean condition C as in the case of C in a *deletion* operation. In addition, the algorithm needs to translate the expression C' in μ_i=$mod(C \rightarrow C')$. For translation, the algorithm first determines the mapping table $m(X, Y)$ that contains the attributes in C'. Next it computes the tuples that satisfy the expression C' and stores the result in a temporary relation $R_{C'}$. After that the algorithm takes projection on attributes Y from $R_{C'}$. Finally, for each projected value y of Y the algorithm converts it into a condition $Y = y$.

- If μ_i is an *insertion* operation then the translation of μ_i is straightforward. If a tuple $t[X_1 = x_1, \cdots, X_l = x_l]$ is inserted in the local database of P_i, the algorithm translates each $X_q = x_q$ to $Y_q = y_q$ where $l = |\mathcal{R}|$, $1 \leq q \leq l$, by finding the appropriate mapping table that contains the attribute X_q. The algorithm treats each $X_q = x_q$ as a condition and translates it as described in the condition translation part of a *deletion* update. After the translation of each $X_q = x_q$ to $Y_q = y_q$, the algorithm forms an update $ins(Y_1 = y_1, \cdots, Y_l = y_l)$.

4.7.1 Discussion

Note that the proposed update semantics and translation algorithm consider the close-word semantics of mapping tables [48]. Under this semantics, a complete set of associations exist between the X and $Y-$values. This is a very strong assumption. However, we need to deal with situations when we translate updates by considering incomplete mappings, particularly, situations where an update creates or requires uncertain information. For example, an inserted tuple may have an attribute for which no mapping table entry exists for the specific value of that attribute

In the update translation algorithm, we do not specify an update translation mechanism when there are variables in the mapping tables. We assume that there exists a valuation function for evaluating each variable. This valuation function is application dependent.

Algorithm 1: The proposed update translation algorithm

Input: An update request μ_i and data sharing constraints Σ_{ij}.
Output: Translated updates $\mu_i^j = \{\mu_i^{j,1}, \cdots, \mu_i^{j,k}\}$.
begin
  $\mathcal{R} = \emptyset$ // Set of relevant mapping tables
  case $u_i = del(C)$ *or* $u_i = mod(C \rightarrow C')$
    $Z \leftarrow att(C)$ //Returns the attributes in C
    for *each mapping table* $m(X,Y) \in \Sigma_{ij}^v$ **do**
      if $X \subseteq Z$ **then**
        $\mathcal{R} = \mathcal{R} \cup m$
    Find the mapping table $m_{key} \in \Sigma_{ij}^v$ which associates primary key values.
    if $m_{key} \notin \mathcal{R}$ **then**
      $\mathcal{R}$=$\mathcal{R} \cup m_{key}$
    $l \leftarrow \mid \mathcal{R} \mid$; $R_C \leftarrow \sigma_C(R)$; Remove tuples from R_C those do not satisfy Σ_{ij}^s
    Compute $R_v \leftarrow R_C \bowtie m_1 \bowtie \cdots \bowtie m_l$ where $m_q \in \mathcal{R}$, $1 \leq q \leq l$
    Compute $R_Y \leftarrow \pi_{Y_1,\cdots,Y_l}(R_v)$, where Y_q is in $m_q(X_q, Y_q) \in \mathcal{R}$, $1 \leq q \leq l$;
    Compute $k \leftarrow |R_Y|$
    if $\mu_i = del(C)$ **then**
      for *each tuple* $t \in R_Y$ **do**
        Generate an update $\mu_i^{j,p}$=$del(Y_1 = y_1, \cdots, Y_l = y_l)$,$(1 \leq p \leq k)$ where $Y_q \in att(R_Y)$, $1 \leq q \leq l$, and y_q is a constant of $t[Y_q]$
      return $\mu_i^j = \{\mu_i^{j,1}, \cdots, \mu_i^{j,k}\}$
    if $\mu_i = mod(C \rightarrow C')$ **then**
      Translate C as the translation of a *del* update; $Z \leftarrow att(C')$
      Find mapping table $m(X,Y) \in \Sigma_{ij}^v$ such that $X = Z$
      Compute $R_{C'} \leftarrow \pi_Y \sigma_{C'}(m(X,Y))$
      for *each tuple* $t \in R_v$ **do**
        for *each tuple* $t' \in R_{C'}$ **do**
          Generate an update $\mu_i^{j,p} = mod((Y_1 = y_1, \cdots, Y_l = y_l) \rightarrow (B = b))$, where $(1 \leq p \leq k)$ $(1 \leq q \leq l)$. B is an attribute in $R_{C'}$, b is a constant of $t'[B]$
      return $\mu_i^j = \{\mu_i^{j,1}, \cdots, \mu_i^{j,k}\}$
  case $\mu_i = insert(t[A_1 = a_1, \cdots, A_l = a_l])$
    $R_C = \emptyset$
    for *each* $A_q = a_q$ **do**
      $Z \leftarrow att(A_q)$
      for *each mapping table* $m(X,Y) \in \mathcal{R}$ **do**
        if $X \subseteq Z$ **then**
          Compute $R_c \leftarrow \pi_Y \sigma_{A_q = a_q}(m)$; $R_C = R_C \cup R_c$
    $w = |R_C|$; Compute $R_v \leftarrow (R_1 \times, \cdots, \times R_w)$
    for *each row* $t[B_1 = b_1, \cdots, B_l = b_l]$ **in** R_v **do**
      Translate to an update $\mu_i^{j,p} = ins(B_1 = b_1, \cdots, B_l = b_l]$
    return $\mu_i^j = \{\mu_i^{j,1}, \cdots, \mu_i^{j,p}\}$ $(1 \leq p \leq r)$,$(r = |R_v|)$
end

Theorem 3 (Correctness) *Given two updates μ_i at peer P_i and μ_i^j at peer P_j which is an acquaintee of P_i. Assume μ_i^j is the translated version of μ_i that is produced by the algorithm 1 with the input $\{\mu_i, \Sigma_{ij}\}$. The update μ_i^j is a correct translation of μ_i.*

Proof 4 *To prove the correctness, we need to show that the tuples updated by μ_i^j in the database D_j of P_j are related with the tuples updated by μ_i in the database D_i of P_i with respect to the data sharing constraint Σ_{ij}.*

Consider the relation schema R_j of P_j and a tuple $t' \in r_j$ such that t' is updated by μ_i^j. Also assume a relation schema R_i of P_i and a tuple $t \in r_i$ such that t is updated by μ_i.

Case ($\mu_i = del(C)$: Assume μ_i^j is $del(C')$) and C' is of the form $(Y_1 = y_1, \cdots, Y_l = y_l)$. Since t' is updated by μ_i^j, therefor, t' satisfies the condition C'. Now we need to show that C' is the correct translation of $C = (A_1 = a_1, \cdots, A_l = a_l)$ $(1 \leq q \leq l)$ which is the condition in μ_i. Assume t satisfies C updated by μ_i. According to the algorithm 1, the condition $C' = (Y_1 = y_1, \cdots, Y_l = y_l)$ is computed using the expression $R_Y \leftarrow \Pi_{Y_1,\cdots,Y_l}(R_v)$. The relation R_v is computed by performing a join of R_C with all the relevant mapping tables in $\mathcal{R}$. Note that R_C contains a tuple t that satisfies C. The projection of $Y_q's$ on R_Y retrieves the data values y_q of each Y_q such that $(Y_q = y_q)$ which constitute the individual conditions in C'. Therefore, the update μ_i^j with the condition C' updates t' which is related to t.

Case ($\mu_i = mod(C \rightarrow C')$) : Assume μ_i^j is $mod(C_t \rightarrow C_t')$). The translation of C_t of μ_i to C_t of μ_i^j is done like the translation of $\mu_i = del(C)$, which is proven correct. Here, we need to show that the condition in C_t' of μ_i^j is a correct translation of C' of μ_i. From the algorithm 1, we notice that the value b mentioned in the atom $B = b$ in C_t' is obtained

from the projected B's value of $R_{C'}$. The relation $R_{C'}$ considers a mapping table $m(A, B)$ that associates the values of attribute A, mentioned in C' of μ_i, to the values of attribute B that is mentioned in C'_t of μ_i^j. Therefore, C'_t of μ_i^j updates the values of B in t' which are related to the values update by C' in μ_i and t' is related to t.

Case ($\mu_i = ins(A_1 = a_1, \cdots, A_l = a_l)$) : Assume μ_i^j is $ins(B_1 = b_1, \cdots, B_l = b_l)$. We need to show that each atom $B_q = b_q$ is the correct translation of the corresponding $A_q = a_q$. The algorithm considers each atom $A_q = a_q$ as a condition and translates it accordingly. The translation of a condition is correct as proved before in the case of a del update. Therefore, the value b_q of each B_q is related to a value a_q of A_q. Hence, $t[A_1 = a_1, \cdots, A_l = a_l]$ is related to $t'[B_1 = b_1, \cdots, B_l = b_l]$ due to correct translation of μ_i to μ_i^j. ■

4.7.2 Complexity Analysis

Basically, the complexity of the algorithm 1 depends on the number of relevant mapping tables involved in translating an update. The main cost of the translation algorithm is the number of joins which is equal to the number of relevant mapping tables.

Assume that R is the relation that is updated by an update μ and $m_1, m_2, m_3, \cdots, m_{l-1}, m_l$ is the sequence of relevant mapping tables that decides the order of joins of the mapping tables. Let R_C be the tuples that are affected by the update satisfying condition C and $v(R_C)$ be the size of R_C. Let $v(m_1)$ be the size of the first mapping table in the relevant mapping tables and $s(R_C, m_1)$ is the selectivity between R_C and m_1. When the nested loop join method is employed, the estimated cost of the join between R_C and m_1 is $v(R_C) * v(m_1)$ and the size of the resultant intermediate

relation from joining R_C and m_1 is $v(R_C) * v(m_1) * s(R_C, m_1)$. Let R_{m_1} denotes this intermediate result. Assume that minimum one mapping table is involved in the translation of update. Therefore, the minimum cost of the translation algorithm is $v(R_C) * v(m_1)$.

Then, the second mapping table m_2 is joined. Let R_{m_1,m_2} denotes the intermediate result after joining of R_{m_1} and m_2. Therefore, the cost is $v(R_C) * v(m_1) + v(R_{m_1}) * v(m_2)$. In general, after the $l-$th mapping table (assume $l > 2$) is joined, the cost is given by:

$$v(R_C) * v(m_1) + \Sigma_{i=2}^{l} v(R_{m_1,\cdots,m_{i-1}}) * v(m_i), \text{ where}$$
$$v(R_{m_1,\cdots,m_k}) = v(R_{m_1\cdots,m_{k-1}}) * v(m_k) * s(R_{m_1,\cdots,m_{k-1}}, m_k)$$

Note that R_{m_1} is the joined result of R_C and m_1, R_{m_1,m_2} is the joined result of $R_{m_1} * m_2$, and so on.

4.8 Summary

In this chapter, we mainly proposed a mechanism for translating updates between peers. We first introduced an update execution semantics and then discussed different important issues related to the translation of updates between peers. The issues include of : (i) finding the relevancy of updates (ii) analyzing the complexity of finding relevancy of updates, and (iii) introducing a correctness criteria for translating updates. Finally, this chapter proposed an update translation algorithm that ensures the correctness of update translation. In the next chapter, we discuss several challenging issues related to the propagation of updates in data sharing systems taking into account the dynamic behavior of peers. Moreover, we present a conflict detection and resolution mechanism during the propagation of updates.

Chapter 5

Update Propagation and Synchronization

In this chapter, we propose an update propagation mechanism that takes into account the dynamic behavior of peers that participates in a data sharing system. We also propose a mechanism for detecting and resolving conflicts that inevitably arise during propagation of updates.

5.1 Update Propagation and Execution

The propagation of updates starts in a data sharing system when a peer receives an update from a user or from its acquaintees. When a peer receives an update, first the update is executed in the local database system. After local execution, the update is forwarded to the acquaintees which are relevant to the update. When a peer initiates an update the peer first creates an initial update message. An update message is composed of:

UID	PID	timestamp	tuple	Action
u_1	P_4	June, 06, 12:45	t4	replace
u_3	P_6	June, 06, 05:21	t2	delete

Table 5.1: Update log table

(i) a unique global identifier (GID) of the update. GID is formed combining the identifier of the initiator and an update sequence number. We assume that each peer generates a unique update sequence number in increasing order for each update it initiates.

(ii) a path tag, consisting of peer identifiers (PIDs). Initially, the path tag contains only the PID of the initiator. As the propagation of the update goes on, each peer also includes its PID in the path tag when it executes and forwards the update. Therefore, when a peer receives an update, it knows from the path tag which peers have executed the update. Thus a peer does not forward the update to its acquaintees that are in the path tag.

(iii) the timestamp of the update. The timestamp represents the time when the update is originated in the network. The timestamp remains unchanged in the update message during the update propagation in the network. Therefore, each peer knows when the update is originated by looking at the timestamp. Section 5.2.1 shows how this timestamp is used to resolve conflicts between two updates.

In our system, each peer maintains the following data structures for propagating updates. Note that from now we denote an update μ_i as u_i.

Send queue (SQ_i^j): Each peer P_i maintains a *send queue* SQ_i^j, a FIFO queue, for each of its acquaintee P_j in order to monitor the status of the forwarded updates. When a peer forwards an update to an acquaintee the peer stores the update in the send queue

of the acquaintee peer. A peer also stores a status value in the send queue for each forwarded update. For each forwarded update, a peer maintains four types of status. In Section 5.1.1, we describe different status values of a forwarded update and their purpose.

Update log (UL_{P_i}): Each peer also maintains a log, called update log, that a peer uses to record the history of the executed updates. When peer P_i successfully executes an update, it records an entry in the update log. Each entry for an update u_i in the log consists of (i) the update id (UID), (ii) the peer id (PID) from where u_i is received, (iii) the timestamp of u_i ($ts(u_i)$), (iv) the tuple to be updated by u_i at P_i before the execution of u_i, and (v) the update action (insert, delete, or replace). Table 5.1 shows an implementation of an update log at a peer P_i. For instance, the first entry denotes that an update u_1 is executed in P_i and u_1 is received from P_4. The timestamp of the update is *June*, 06, 12 : 45. This time timestamp remains the same over all the peers where the update is executed. In the entry, $t4$ denotes the tuple updated by u_1, and the operation is *replace*.

5.1.1 Update Propagation

We now describe a strategy for propagating updates in a data sharing network. First, we describe the strategy without considering the dynamic behavior of peers. The dynamic case where peers leave or join the system is described in Section 5.1.2

It is discussed in Section 3.1.1 that when an update u_i is originated in a peer P_i, a set of remote updates is produced. This logically produces a global update u_i^g which consists of the initial update u_i and the set of remote updates produced by u_i. The relationships between the members of a global update are constructed using an *Update Dependency Tree (UDT)*. This logical tree is generated during the propagation of updates. The nodes

in the tree represent updates and there is an edge from update u_i^j to update u_i^k, if u_i^k depends on u_i^j. An update u_i^k on peer P_k is said to depend on update u_i^j, if peer P_j and P_k are acquainted and update u_i^k has resulted from the translation and propagation of update u_i^j from peer P_j to P_k. The root of the UDT is annotated with the global update initiator u_i.

The propagation of an update u_i and the construction of the corresponding $UDT(u_i)$ at P_i is described below:

1. If P_i is the initiator of u_i and u_i needs to be executed in acquaintees, then the construction of $UDT(u_i)$ starts and u_i becomes the root of the $UDT(u_i)$.

2. P_i translates the update u_i for an acquaintee P_j such that P_j is relevant to u_i. Next, P_i sends the translated update u_i^j to P_j and sets the status flag in SQ_i^j to 'S' for u_i. The status flag 'S' denotes that update is sent.

3. A peer P_j replies 'Y' to P_i for u_i^j, if it is the first time P_j receives the update u_i^j. If P_i receives the response 'Y' from P_j, then P_i considers u_i^j as a child of u_i in the $UDT(u_i)$ and changes the status flag from 'S' to 'Y' for u_i in SQ_i^j. If u_i^j is received by P_j previously from another peer P_k, then P_j replies 'N' to P_i for u_i^j. If P_i receives the response 'N' from P_j, then P_i does not consider u_i^j for the $UDT(u_i)$ and changes the status flag from 'S' to 'N' for u_i in SQ_i^j. When P_j accepts u_i^j from P_i, it also participates in constructing $UDT(u_i)$ by translating u_i^j and forwarding to its acquaintees. When no response is received by P_i from P_j for the update u_i^j after a certain period of time (time out), P_i then assumes that P_j is offline. P_i sets the status flag from 'S' to 'X' for u_i in SQ_i^j. When P_j gets back online and requests for missing updates during the unavailable period, P_i then forwards all the updates u_i^j that has status flag 'X' and proceeds accordingly.

4. Each peer terminates its task in propagating u_i when the status flag of u_i in the *send queue* is changed to any of {'Y','N','X'} from 'S' for all acquaintees P_j where u_i is forwarded or when there is no peer to forward the update.

Notice that the above protocol constructs a logical tree with u_i as a root. In the tree, a logical edge is created due to the propagation of u_i from P_i to P_j with the edge where 'Y' response message is sent. Furthermore, the last peer is connected in the tree by a path where 'Y' message is sent to each link. Since every $P_j \neq P_i$ sends exactly one 'Y', the tree contains all the peers that is relevant to u_i, and contains no cycle. Therefore, it is a tree rooted at u_i.

Note that a peer involved in executing an update is not aware of the global execution and termination of the update in the network. Moreover, each peer stores some information about each update in the *send queue* which may grow indefinitely. This may raise concern about the effectiveness of the protocol. Since the protocol constructs a rooted tree during propagation of an update, the *convergecast* [78] algorithm may be used for the global termination of an update and deleting the entries in the *send queue.* Before describing the *convergecast* process, we introduce the role of a peer in constructing a UDT. For each propagation edge of u_i, there is a parent-child relationship between peers. When a peer forwards the update through an edge to a peer and receives 'Y' for the update, the forwarding peer becomes the *parent* and the receiving peer becomes a *child* for the update. The peer which originates the update is the *root* peer and the peers where the update propagation terminates are $leaf$ peers.

The *convergecast* process for an update u_i starts at the leaf peers of the $UDT(u_i)$ when the updates received from their parents are successfully executed. The process is described below:

1. a leaf peer sends a message to its parent about the successful execution of the update.

2. when a parent peer receives the successful execution message from its child; it then removes the entry from the corresponding *send queue*. When the parent receives messages from all its children it also sends a message of the successful execution of the update to its parent.

This way the initiator and all the peers know about the termination of the update in the network and delete all the entries from their *send queues* of the update. Note that UDT for an update u_i creates the relationships among the remote updates generated from u_i and the relationships are used to terminate the global execution of u_i.

5.1.2 Online and offline Update

This section examines the propagation of updates throughout a data sharing network by taking the dynamics of the peers into account. In a P2P network, a peer may be offline at any time and any amount of time. When a peer goes offline, it may miss some updates during the offline period. Therefore, when an offline peer gets back online, it needs to synchronize its data with its acquaintees. During synchronization, the peer may invoke synchronization in further peers that are acquainted with the offline peer.

Based on the dynamics of networks, updates can be categorized further into two different types, namely *offline* and *online updates*:

• *Online update*: Update which is processed in all peers that are currently online when the update is initiated.

• *offline update*: Update which is processed in peers when they come back online from

the offline state.

To deal with these two categories of updates, we use the combination of *push* and *pull* strategies [25, 55, 27, 43, 87]. The *push* method is used to propagate online updates. The intuition behind this choice is that data need to be synchronized immediately when an update is initiated. Also, a push method is suitable when the coherency requirements are stringent and data changes are not frequent [27]. Using a pull method instead would result in periodically checking acquaintances for the latest updates to synchronize data. This would incur significant communication overhead since updates are less frequent in data sharing networks.

We use a *pull* method to process offline updates. The intuition here is that when a peer gets back online, it should be the one that initiates data synchronization. We know that the sender of an update only knows, from the entries in the *send queue*, about the acceptance/refusal of an update. Once the update is accepted by an acquaintee the sender has no knowledge about the execution status of the update until it receives the message of the successful execution during the *convergecast* process. Therefore, it is not a feasible strategy that the sender waits or continuously monitors the status of a peer for an indefinite period of time to send updates. Hence, when an offline peer gets back online it sends a *pull* request to its acquaintees to deliver the updates that the peer missed during the offline period. Once a peer receives the *pull* request the peer sends all the updates that is marked 'X' in the status flag in the *send queue* for the requester.

In this thesis we consider limited dynamics of peers. We assume that peers do not go offline for ever and there is no network partition occurs. Therefore, when a peer forwards an update to an acquainted peer, it waits for a response for a certain period of time. If the peer does not receive any response it assume that the acquainted peer is offline.

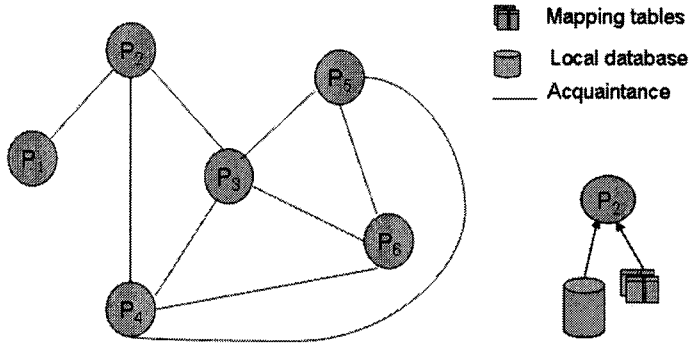
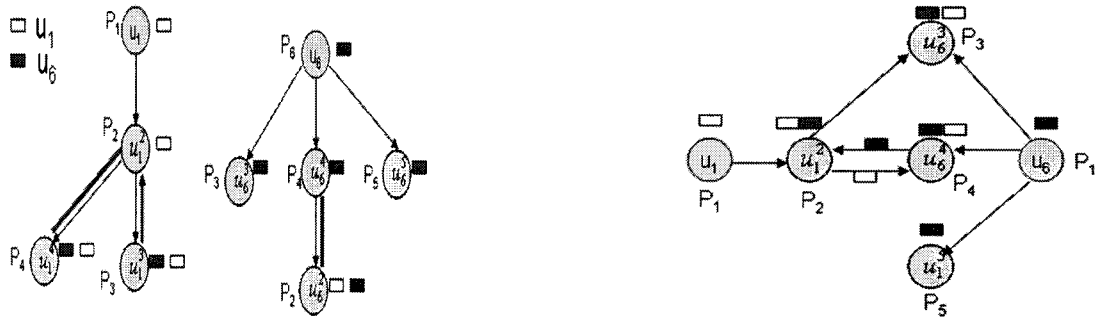


Figure 5.1: A peer data sharing scenario



(a) Construction of UDTs

(b) Conflict detection

Figure 5.2: Conflict detection when two UDTs coincide

Later, when the acquainted peer comes back online, the acquainted peer is responsible to request its neighboring peers for executing the offline updates.

5.2 Update Synchronization

This section introduces an update synchronization model that ensures the consistency of each peer with its respective acquaintees. Recall that each peer is autonomous and has no global control of the execution of updates. During the propagation of updates several situations may occur which lead to data inconsistency between two acquainted peers. In

the following, we give some examples of data inconsistencies.

Case 1: Although, updates are executed in the same order in two acquainted peers, the updates may still read and write two different values. Consider that a peer P_2 has executed the updates u_1^{21} and u_1^{22} in the order $u_1^{21}u_1^{22}$ sent by a peer P_1. Also assume that P_2 has received two new updates u_1^{23} and u_3^{21} from its acquaintees P_1 and P_3, respectively. Suppose that the execution order of the updates in P_2 is $u_1^{21}u_1^{22}u_3^{21}u_1^{23}$. Although P_2 executed the updates in the same order $u_1^{21}u_1^{22}u_1^{23}$ sent by P_1, the database of P_2 is in an inconsistent state if u_1^{23} and u_3^{21} update the same tuple.

Case 2: Consider a situation when two peers participate in the construction of two UDTs which correspond to two different updates. Also assume that peers forward the updates to each other through a common edge in the acquaintance graph and the updates conflict with each other. When two update operations attempt to update the same tuple in a database then we say that the updates are conflicting updates. The formal definition of the update conflict is given in Definition 5.2.1. This conflict situation is depicted in Figure 5.1. We see that peers P_1 and P_6 initiate two updates u_1 and u_6. The constructions of the UDTs for u_1 and u_6 are shown in Figure 5.2(a) and Figure 5.2(b). We now describe the scenario of Figure 5.2.

1. P_1 and P_6 initiate updates u_1 and u_6.
2. P_1 translates u_1 for P_2 and forwards the translated update to P_2. Similarly, P_6 translates u_6 for P_3, P_4, and P_5 and sends the translated updates to the peers.
3. P_2 forwards u_1^2 to P_4 and P_3. P_4 forwards u_6^4 to P_2.

From the above scenario, it is obvious that the execution order of the updates in P_2 is $u_1^2u_6^2$ because P_2 receives u_1^2 before u_6^2. Similarly, the execution orders of the updates in P_3 and P_4 are $u_6^3u_1^3$ and $u_6^4u_1^4$, respectively because both the peers receive translated

u_1 after receiving translated u_6. P_2 can realize that the update order in P_4 is different compared to its own local order, because P_2 forwards u_1^2 to P_4 and receives u_6^2 from P_4. It means that P_2 has received u_1^2 before u_6^2. Similarly, P_4 recognizes the difference of update order in P_2. In that case, we can say that the edge connecting P_2 and P_4 has a "conflict". On the other hand, P_3 has no knowledge of the update order in other peers. In this case, P_3 is unable to take any decision with respect to the order in which the updates should be executed or which one should be accepted or rejected.

Note that due to arbitrary topology of a database network, these conflict situations may occur in many places during propagation of updates.

Case 3: Consider three peers P_1, P_2, and P_3 which, at the beginning, are consistent with respect to data item X. Suppose P_1 initiates an update u_1 locally and executes an action on X. Also assume that P_1 goes offline before forwarding u_1 to P_2 and P_3. Meanwhile, a few updates have already been executed at P_3 and P_2 that are initiated at P_3 while P_1 was offline. When P_1 comes back online, it reconciles its offline updates with P_2 and P_3 by sending the translated update u_1 to P_2 and P_3 and by requesting updates from P_2 and P_3. Hence, P_2 and P_3 send their updates to P_1. As there is no central control in ordering the updates, it is difficult to find the same order for executing updates in all these peers. Moreover, a peer cannot control the execution of updates to other peers. Therefore, if the order is not the same in all the peers, the databases in the peers could be inconsistent with respect to X.

5.2.1 Conflict Resolution

This section discusses the conflict resolution strategies. We categorize the resolution techniques into two types: (i) ordering of conflicting updates and (ii) resolutions of

semantic conflicts. In the first case, we ensure that when two updates are in conflict in a peer then they must be executed in the same order in all peers. For ordering of conflicting updates, each peer independently detects and resolves conflicts and agrees on the same data value with its acquaintees. For semantic conflict resolution, we introduce resolution rules in Section 5.2.2.

Value at Neighbor Resolution

According to the *value at neighbor resolution* strategy, when a peer forwards an update to its acquaintees the peer includes the last value of the data item seen by the update in the database. Note that the peer translates the data item and the update wrt the schema of the acquaintees before forwarding the update to those acquaintances. When a remote peer receives the update it knows the last value of the data item at the sender. The peer then compares the value received from sender with the value in the local source. If the values are the same then both sender and receiver agrees on the last value of the data item before the execution of the update. After that, the update is executed. Otherwise, the peer realizes that another update has already accessed the data item and updated it. In this case, the remote peer uses timestamps of the updates to resolve the conflict. We assume that the system uses Greenwich Mean Time timestamp and updates are executed according to the increasing timestamp order. The Greenwich Mean Time timestamp allows each peer to know a global standard time when an update is originated in the network irrespective of the local times of peers.

We introduce some notations that is used in further discussion. We use $u_i(X)$ to denote an update u_i on data item X and $R_v(u_i, P_i, X)$ the value of X read by u_i at peer P_i before the execution of u_i. When an update u_i is forwarded to an acquaintee

P_j, P_i includes $R_v(u_i, P_i, X)$ with the update message. Note that, when $R_v(u_i, P_i, X)$ is forwarded, it is translated using the corresponding mapping tables. We assume that the translation is complete, i.e. each atomic value of X is translatable. For ease of presentation, we keep the same notations of u_i and X, and their translation in all peers. Now we define the conflict between two updates in a peer.

Definition 22 (Conflict) *Suppose that a peer P_j receives an update $u_i^j(X)$ from P_i. Then $u_i^j(X)$ conflicts with another update $u'_j(X)$ at P_j if $R_v(u_i, P_i, X) \neq R_v(u_i^j, P_j, X)$ and $R_v(u_i, P_i, X) = R_v(u'_j, P_j, X)$.*

The definition states that when an update u_i^j arrives at a peer P_j and the peers finds that the data item the update wants to update is already updated by another update u'_j, then the update is in conflict. The conflict is determined by the last value read by u_i at P_i and by u_i^j at P_j. If $R_v(u_i, P_i, X) \neq R_v(u_i^j, P_j, X)$, that is different value of X is read at P_i and P_j, then there must be an update u'_j that updated X. In this case, $R_v(u_i, P_i, X) = R_v(u'_j, P_j, X)$, that is the last value read by u'_j at P_j is same as the last value read by u_i at P_i for data item X. Therefore, u_i^j and u'_j conflict at P_j. An algorithm to resolve a conflict is described below:

An update $u_i^j(X)$ that reaches a peer P_j from P_i is executed as follows.

1. P_j determines $R_v(u_i^j, P_j, X)$.
2. P_j compares $R_v(u_i^j, P_j, X)$ with $R_v(u_i, P_i, X)$.
3. If $R_v(u_i^j, P_j, X) = R_v(u_i, P_i, X)$ then P_j executes u_i^j and puts an entry in its update log. This guarantees that P_i and P_j agree on the same value of X before executing u_i and u_i^j, respective. The format of an entry in the update log is described in Section 5.1.
4. If $R_v(u_i^j, P_j, X) \neq R_v(u_i, P_i, X)$ then the value of X read by u_i at P_i is not consistent

with the value of X read by u_i^j at P_j. P_j realizes that another update has already updated X. In order to find the update, P_j searches its update log. Suppose that update is u'_j which updated X with the read value of $R_v(u_i, P_i, X)$, i.e. $R_v(u_i, P_i, X)$=$R_v(u'_j, P_j, X)$. Therefore, P_j now proceeds to resolve the conflict using the latest timestamp method as follows:

if $ts(u_i^j)$<$ts(u'_j)$. (Remind that $ts(u_i^j)$ is the timestamp of u_i^j, i.e. the time when u_i^j is originated.)

In this case, P_j finds that u_i^j is an older update than u'_j but u'_j is executed before. Note that, in P_j there may be other updates executed after the execution of u'_j. In order to execute the updates according to the timestamp order, all the updates which have timestamp greater than the timestamp of u_i^j need to be undone. Therefore, P_j finds all the updates $u(X)$ from the update log such that $ts(u)$ >$ts(u_i^j)$. Let U be such a set of updates.

else $ts(u_i^j)$>$ts(u'_j)$

(a) In this case, P_j finds that u'_j is an update which is older than u_i^j but u'_j is executed before. Note that after execution of u'_j there may be other updates $u(X)$ that were executed in P_j and $ts(u)$ >$ts(u_i^j)$. This situation may occur when u_i^j reaches P_j after $u(X)$. Therefore, P_j finds all the updates $u(X)$ such that $ts(u)$ >$ts(u_i^j)$. Let U be such a sequence of updates.

(b) undo all the updates in U; form a new sequence of updates $U' = \{u_i^j\} \cup U$ and re-execute the updates in U' according to increasing timestamp order.

(c) update the log accordingly.

There are several advantages of this strategy. First, each peer performs the conflict resolution independently. Second, it does not require any special session or exchange of extra messages between two peers for synchronizing data. Finally, the strategy also supports the execution of updates when a peer returns to online status. Only the peer that comes back online will request its acquaintees to send the updates which it has missed during the offline period. When the acquaintees send the updates stored in their *send queues*, the peer starts processing the updates like any other online peer.

We now prove the correctness of the *value at neighbor* conflict resolution protocol i.e., if two conflicting updates are executed in a peer in a certain order then the protocol ensures that all the peers where these two updates are executed maintain the same order.

Theorem 4 (Correctness) *If a peer P_k executes two conflicting updates $u_i^k(X)$ and $u_j^k(X)$ in an order $u_i^k u_j^k$, where $ts(u_i^k)<ts(u_j^k)$, then the protocol* value at neighbor *guarantees that all the peers those execute these two updates maintain the same order as in P_j.*

Proof 5 *Consider the two possible cases where the two updates u_i and u_j are initiated at peers P_i and P_j, respectively. Assume that the updates are conflicting. Let us denote the translated versions of these two updates at peers P_v and P_w by u_i and u_j.*

Case 1. Updates u_i and u_j are at P_v and P_w, where $P_v = P_w$. According to the protocol, P_v executes them in the timestamp order, say $u_i u_j$. Now we need to show that u_i and u_j are executed in all peers relevant to u_i and u_j in the same order as executed by P_v. Suppose that u_i has reached P_v along the path $(P_i \rightarrow, \cdots, \rightarrow P_v)$ and u_j has reached P_v along the path $(P_j \rightarrow, .., \rightarrow P_v)$. If $ts(u_i) < ts(u_j)$, then P_v must execute u_i before u_j. When u_j is forwarded to the direction of u_i's initiator (the sender of u_i to P_v), the order is maintained automatically there, since u_i has already been executed there before u_j arrives.

Therefore, the order is maintained during the propagation of u_j in all the peers along the path starting from P_v to P_i. In order to execute u_i before u_j at all peers along the path from P_v to P_j, all peers perform three actions: undo u_j, execute u_i, and execute u_j again. Henceforth, the orders are maintained.

Case 2. Updates u_i and u_j are in conflict at P_v and P_w, where $P_v \neq P_w$. Let us keep the same order of execution of $u_i u_j$ as in case 1. Clearly, u_i is executed before u_j at P_v and u_j is executed before u_i at P_w. But the same order must hold at P_v and P_w and at all the peers along the propagation paths where u_i and u_j are executed. If $ts(u_i) < ts(u_j)$, according to the algorithm, P_v has executed in proper order but P_w must undo u_j, execute u_i, and execute u_j again. This ensures the same order of execution at P_w as executed by P_v and the execution will continue at all the peers in the path from P_w to P_j where u_i and u_j are executed. ■

5.2.2 Resolution Rules

In this section, we introduce rules to resolve semantic conflicts between two updates u_i and u_j in a peer. An example of a semantic conflict is given below.

Example 25 (Semantic Conflict) *An update u_i deletes a tuple t at a peer P_i and P_i forwards u_i to P_j to delete the tuple that is related with t. On the other hand, another peer P_k requests P_j to replace the tuple t without knowing that the tuple is already deleted. The semantic conflict can also occur when two insert updates arrive in a peer to insert a tuple that has the same key value. These and similar situations are called* semantic conflicts.

Peers deal with semantic conflicts by means of resolution rules. The rules specify actions to be taken to resolve conflicts. Each peer contains each of the rules. Recall that

updates are executed in a peer immediately upon their arrival. During update execution, a peer first stores the current value of the data item that is to be updated its update log before the update is executed in the data source. In order to detect a semantic conflict, a peer first checks the update log for records indicating that the data item has already been changed by a previously executed update. If such a record is found in the update log, a semantic conflict is detected and the conflict resolution rules are applied to resolve this conflict. Otherwise, the peer proceeds with the normal execution of the update. In describing the resolution rules, we use the following notations:

- I_{UL} instance of update log
- r_j^+ : tuple to be inserted by u_j
- t_i^+ : tuple inserted by u_i in the update log
- r_j^- : tuple to be deleted by u_j
- t_i^- : tuple deleted by u_i in the update log
- r_j^* : tuple to be replaced by u_j
- t_i^* : tuple replaced by u_i in the update log
- $key(t_i)$: values of key attributes in t_i

Rule 1 : $(u_j = insert)$ //Update u_j arrives at a peer.

$\exists\ t_i \in I_{UL}$ where $(t_i = t_i^+)$ or $(t_i = t_i^-)$

a. if $u_i = insert$

if $key(r_j^+) = key(t_i^+)$

if $ts(u_i) < ts(u_j)$ then

undo u_i; execute u_j

replace the entry $(u_i, P_i, ts(u_i), t_i^+, insert)$ with

$(u_j, P_j, ts(u_j), r_j^+, insert)$ in the update log

else discard u_j.

b. if $u_i = delete$ and $key(r_j^+) = key(t_i^-)$ then

if $ts(u_i) < ts(u_j)$ then

execute u_j

insert $(u_j, P_j, ts(u_j), r_j^+, insert)$ in the update log

else discard u_j

c. if there is no update that conflicts with u_j then

execute u_j

insert $(u_j, P_j, ts(u_j), r_j^+, insert)$ in the update log.

Rule 1 deals with the case where an insert update u_j conflicts with an insert and a delete update u_i. Rule 1(a) addresses the case where u_j attempts to insert a tuple that is already inserted by another update u_j with the same key values; here, the effect of u_i is undone if timestamp $ts(u_i) < ts(u_j)$. The intuition behind this conflict resolution is that u_i is an obsolete update which must be replaced by the recent update u_j. After the execution of u_j, the update log is modified accordingly. Rule 1(b) describes the case where u_i is a *delete* operation. In this case, u_j is executed if $ts(u_i) < ts(u_j)$, since u_i deleted a tuple that is later to be inserted by u_j. Therefore u_j should be accepted. The update log is modified accordingly. Finally, rule 1(c) expresses the falling through case: if there is no conflict, then u_j is executed.

The intuition behind the next rule (Rule 2) is to resolve conflicts between a delete update u_j and any other updates u_i that is already executed in a peer.

Rule 2 : $(u_j = delete)$

$\exists\ t_i \in I_{UL}$ where $(t_i = t_i^+)$ or $(t_i = t_i^-)$ or $(t_i = t_i^*)$

a. if $u_i = delete$ and $key(r_j^-) = key(t_i^-)$

discard u_j

b. if $u_i = insert$ and $key(r_j^-) = key(t_i^+)$ then

if $ts(u_i) < ts(u_j)$ then

execute u_j,

insert $(u_j, P_j, ts(u_j), r_j^-, delete)$ in the update log

else discard u_j

c. if $u_i = replace$ and $key(r_j^-) = key(t_i^*)$ then

if $ts(u_i) < ts(u_j)$ then

execute u_j,

insert $(u_j, P_j, ts(u_j), r_j^-, delete)$ in the update log

else discard u_j

Rule 2(a) deals with the case where two delete updates target the same tuple for deletion. In this case, we simply discard one of them. Execution of any of them is sufficient to delete the tuple. Rule 2(b) describes the actions for a delete-insert conflict. This situation occurs when a delete update u_j arrives at a peer to delete a tuple that was previously inserted by u_i. In this case u_j is executed if $ts(u_i) < ts(u_j)$. Rule 2(c) describes the resolution strategy for a delete-replace conflict. When a delete update u_j arrives at a peer to delete a tuple t and finds that a replace update u_i already changed some values of t, then a delete-replace conflict occurs. In this case, u_j is executed if $ts(u_i) < ts(u_j)$; otherwise, u_j is discarded.

Our last rule (Rule 3) resolves conflicts when a replace update u_j is to be executed on a tuple which has already been updated by another update u_i.

Rule 3 : (u_j = *replace*)

$\exists\ t_i \in I_{UL}$ where (t_i=t_i^-) or (t_i=t_i^*)

a. if u_i = *replace* and $key(r_j^*) = key(t_i^*)$

if $ts(u_i) < ts(u_j)$ then

undo u_i, execute u_j,

delete $(u_i, P_i, ts(u_i), t_i^*, replace)$ from the update log

insert $(u_j, P_j, ts(u_j), r_j^*, replace)$ in the update log

else discard u_j.

b. if u_i = *delete* and $key(r_j^*) = key(t_i^-)$

if $ts(u_i) < ts(u_j)$ then

undo u_i, execute u_j

delete $(u_i, P_i, ts(u_i), t_i^-, delete)$ from the update log

insert $(u_j, P_j, ts(u_j), r_j^*, replace)$ in the update log

else

discard u_j.

Rule 3(a) describes the replace-replace conflict. In this case, the last update is considered. In Rule 3(b), the replace-delete conflict is considered; here, the delete update u_i is undone if $ts(u_i) < ts(u_j)$, otherwise u_j is discarded.

5.2.3 Justification of the Rules

In this section, we justify the resolution rules. We notice that an *insert-replace* or *replace-insert* conflict can not occur in the system. We also justify this case.

Assume that the system is consistent for a data item X which is at the beginning empty. It means that X is not present in any peer.

The following situations may happen for an *insert* update on the data item X.

Case 1: A single peer initiates an insert update to insert the data item X, and the update may propagate to other peers. If the update is propagated to other peers, the update is executed in peers without any conflict. The situation is addressed by the rule 1(c).

Case 2: Multiple peers initiate the insertion of X, results in an insert-insert conflict. The conflict is solved by the rule 1(a).

Now assume that all the peer databases are consistent with a certain value of X. The following situations may happen for a *delete* update.

Case 1: A peer initiates the delete update for the deletion of X. This update may propagate to other peers. In this case, the update is successfully executed in peers. (No conflict)

Case 2: Multiple peers initiate the deletion of X, results in a delete-delete conflict. This conflict is solved by the rule 2(a).

Case 3: A single peer initiates the deletion of X, and another peer initiates an replace update to modify the value of X. This cause a delete-replace conflict. This conflict is solved either by rule 2(c) or 3(b).

Case 4: Multiple peers initiate replace updates to modify data of X. Thus, a replace-replace conflict occurs. This conflict is solved by the rule 3(a).

An insert-delete or a delete-insert conflict can occur in some exceptional cases. For example, a peer initiated an insert update on X, and executed in all peers. On the other hand, another peer initiated a delete update. If there is no replace update executed on X between the insert and delete updates, a delete-insert or an insert-delete conflict occurs.

Also, an insert-replace or a replace-insert conflict can occur in some exceptional cases.

For example, a peer initiated an insert update on X, and executed in all peers. On the other hand, another peer initiated a replace update. If there is no delete update executed on X between the insert and replace updates, a replace-insert or an insert-replace conflict occurs.

5.3 Summary

The current chapter presented an update propagation algorithm. During the propagation of updates, an update dependency tree is built which shows the relationship between the component updates of a global update. We also discussed several challenging issues for the execution of updates in a dynamic environment. Moreover, we presented a conflict detection and resolution mechanism for updates. In our next chapter, we investigate the processing of concurrent transactions in a data sharing system.

Chapter 6

Transaction Processing

In this chapter, we propose a transaction processing mechanism in a peer data sharing system. We assume that each peer uses transaction services for querying and updating local as well as remote data. We mainly focus on the correct execution of concurrent transactions over a data sharing system initiated by a peer. We also present a correctness criteria, called the acquaintance-level consistent execution view of transactions, for ensuring the consistent execution view of concurrent transactions in the system. We propose two approaches for ensuring the correctness criteria, namely the Merged Transactions method and the Ticket method.

With respect to transaction processing, a data sharing system is similar to a conventional multidatabase system (MDBS) [57, 18] in the sense that each system consists of a collection of independently created local database systems (LDBSs), and transaction management is handled at both the global and local levels. In an MDBS, global level transactions are issued to the global transaction manager (GTM), where they are decomposed into a set of global subtransactions to be individually submitted to the corresponding LDBSs. Local transactions are directly submitted to the local transaction

management managers (LTMs). Each local transaction manager maintains the correct execution of both local transactions and global subtransactions at its site. It is left to the GTM to maintain the correct execution of global transactions.

In contrast, a data sharing system is built on a dynamic network of peers without a global transaction manager or controller. In a data sharing system, global level transactions are initiated by any peer. If a transaction is submitted to a peer and needs to be executed over the network, then the transaction is propagated from peer to peer. Note that when a user submits a transaction to a peer, he/she is only aware of the local database schema. The mappings between the peer where the transaction is active and other peers in the network determine the translation of the transaction, and propagation and execution of the transaction to other peers.

6.1 Objectives and Assumptions

One of the objectives of this chapter is to propose an approach for ensuring a consistent execution view of concurrently executing transactions in a data sharing system. A transaction is a sequence of read (e.g. SQL select) and write (e.g. SQL update, delete, and insert) actions in a database. Although we assume that each LDBS of each peer guarantees serializability, but the transactions that execute concurrently in multiple peers, may have different execution views at different peers. We first identify some potential problems for ensuring a consistent execution view of transactions in a data sharing system, and introduce a correctness criteria in order to ensure the consistent execution view of transactions. We propose two approaches that guarantee the consistent execution view of transactions.

In general, the system discussed in this chapter is based on the following assumptions:

1. When a user submits a transaction in a peer, he/she is only aware of the local database schema, and there is no global transaction manager or coordinator in the system.

2. A peer is not able to control or synchronize the execution of transactions in another peer.

3. Each LDBS has a mechanism for ensuring the local serializability.

6.2 Preliminaries

For ease of presentation, we use the well-known read-write model of transactions. We now recall the basics of this model. Let a database be a (finite) set $D = \{a, b, c, \cdots\}$ of data objects. A transaction T is a sequence of database operations applied to a subset of data objects D. Formally, $T=(O_T, \prec_T)$, where O_T, is a finite set of operations and $\prec_T$ is a partial order of operations that have been invoked by a transaction T. The operations of a transaction T consist of *reads* (denoted by $r(a)$) and *writes* (denoted by $w(a)$) operations. Further, each T has begin and termination operations commit (c) or abort (a).

The concurrent execution of transactions results in a schedule. A schedule S is a pair $(\Gamma_S, \prec_S)$, where Γ_S is a finite set of transactions and $\prec_S$ is a partial order over the operations of transactions in Γ_S. The partial order $\prec_S$ satisfies the property that it preserves the order of steps within each transaction, (that is, $\prec_{T_i} \subseteq \prec_S$, for each $T_i \in \Gamma_S$).

The most commonly used correctness criteria for an acceptable schedule is conflict serializability [90]. Consider a schedule S consisting of transactions T_i and T_j, then a transaction T_i is said to conflict (direct conflict) with T_j, denoted by $T_i \rightarrow T_j$, if there

exist operations o_i in T_i and o_j in T_j, $T_i \neq T_j$, such that $o_i \prec_S o_j$, and o_i, o_j access the same data item and one of them is a write operation. By $\xrightarrow{*}$, we denote the transitive closure (indirect conflict) of the $\rightarrow$ relation. A *serialization order* of a set of transactions Γ_S in a schedule S is a total order $\prec$ of Γ_S such that, for any pair of transactions T_i and T_j in Γ_S, $T_i \prec T_j$ holds if T_j conflicts with T_i in the execution.

The execution views of a set of transactions Γ in two schedules S_i and S_j are same if the serialization orders of the transactions are same in their execution. We assume the commit order of two transactions as the serialization order if there is no conflict between the transactions. A schedule S is called conflict serializable (serializable) if there exists a serial schedule S' such that the transactions in S have the same serialization order as in S'.

Similar to the execution semantics and classification of updates as discussed in Section 3.1.1, transactions can be classified into three categories, namely *local*, *remote*, and *global*. We denote a transaction initiated into a peer P_i by T_i. If a transaction is local it is executed only by the local database in P_i. We denote a local transaction by L_i. A remote transaction that is generated from a transaction T_i by a peer P_i for one of its acquaintees P_j is denoted T_i^j. A *global* transaction, denoted by G_i, is a set of remote transactions and the initiator of the global transaction. For ease of presentation, we denote remote transactions $T_i^j, T_i^k, \cdots$ generated from T_i as T_i, since they are actually generated from T_i, and a global transaction G_i is represented with the initiator T_i. Intuitively, execution of any component transaction $T_i, T_i^j, T_i^k, \cdots$ is called the execution of G_i.

6.2.1 Properties of a Global Transaction

In a data sharing system each global transaction consists of a set of transactions that includes a global transaction initiator and a set of remote transactions. Each of the transaction is called a component transaction of the global transaction. Note that each component transaction is an atomic transaction resulted from the translation of another component transaction. Each component transaction accesses data items that are located in the peer where the transaction is active. Unlike a global transaction in an MDBS, a component transaction is not decomposed into subtransactions to access data at acquaintees. In order to access data at acquaintees, the component transaction is propagated as an atomic transaction after translation to each of the acquaintees if there are data mappings between the acquainted peers with respect to the data accessed by the transaction.

6.2.2 Transaction Translation

When a transaction is forwarded to an acquaintee, first the transaction is translated in terms of the data vocabularies of the acquaintee. During translation, each $r(a)$ and $w(a)$ operation is translated into the same operation substituting the data value a with the corresponding data value a' using the appropriate mapping table that contains the mapping $a \rightarrow a'$. Due to one to many mappings of a data item, the translation of a single operation may produce multiple operations. For ease of presentation, we assume that a single operation is produced from each operation after translation. During the translation of a transaction, each write operation $w(a)$ is translated using the algorithm proposed in Section 1. However, each read operation is translated using the algorithm proposed in [49]. The translation of a transaction is restricted as specified in the definitions below.

Definition 23 (Order Restriction) *Consider two transactions T_i and T_j with partial orders $\prec_{T_j}$ and $\prec_{T_j}$, respectively. Transaction T_j follows the order restriction of T_i if $O_{T_j} \subseteq O_{T_i}$, and for all o_1,$o_2 \in O_{T_j}$, $o_1 \prec_{O_{T_j}} o_2$ iff $o_1 \prec_{O_{T_i}} o_2$.*

Note that the translation keeps each operation (read/write) same, but only translates the data items mentioned in the operations.

Definition 24 (Schedule Order Restriction) *Consider a schedule $S_i = (\Gamma_{S_i}, \prec_{S_i})$ at P_i. Then a schedule $S_j = (\Gamma_{S_j}, \prec_{S_j})$ at an acquaintee P_j of P_i follows the order restriction of S_i if for any two operations o_1,o_2 in S_j, $o_1 \prec_{S_j} o_2$ iff $o_1 \prec_{S_i} o_2$.*

Translation of a transaction may be either partial or complete over the acquaintance, depending on whether parts or the totality of its operations have been translated.

Definition 25 (Partial Translation) *A transaction T_j is a partial translation of a transaction T_i if $O_{T_j} \subset O_{T_i}$, and T_j follows the order restriction of T_i.*

Definition 26 (Complete Translation) *A transaction T_j is a complete translation of a transaction T_i if $O_{T_j} = O_{T_i}$, and T_j follows the order restriction of T_i.*

Example 26 *Consider a peer data sharing system with four peers P_1, P_2, P_3, and P_4. Assume P_1 has acquaintances with peers P_2 and P_3. Peer P_3 has acquaintance with peer P_4. Suppose the following data mappings are in place among the acquaintances.*

$$(1,2)\text{: } a^1 \rightarrow a^2,\ c^1 \rightarrow c^2,\ d^1 \rightarrow d^2$$

$$(1,3)\text{: } c^1 \rightarrow c^3,\ d^1 \rightarrow d^3, \quad (3,4)\text{: } c^3 \rightarrow c^4,\ d^3 \rightarrow d^4$$

Suppose a global transaction T_1 is initiated at P_1 in the network.

$$T_1 = w_1(a^1) r_1(b^1) w_1(c^1) w_1(d^1)$$

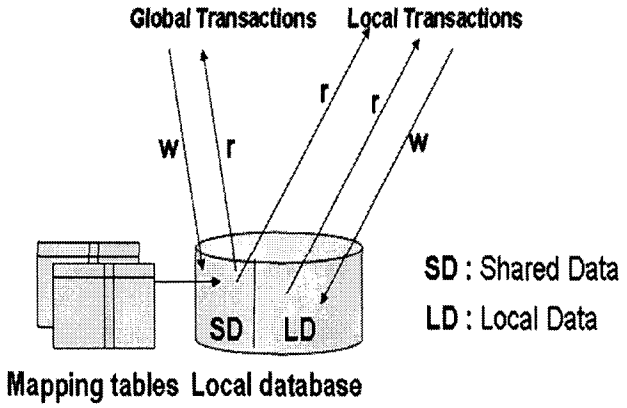


Figure 6.1: Data constraint property

Based on these mappings, P_1 generates the following remote transactions from T_1 for P_2 and P_3.

$$T_1^2 = w_1(a^2)w_1(c^2)w_1(d^2), \quad T_1^3 = w_1(c^3)w_1(d^3)$$

When P_3 receives T_1^3, it also generates the following remote transaction for P_4 using the data mappings that exist between P_3 and P_4.

$$T_1^4 = w_1(c^4)w_1(d^4)$$

From the above translation, we can say that T_1^2 and T_1^3 are partial translation of T_1. Meanwhile, T_1^4 is a complete translation of T_1^3. All of the translations also follow the order restrictions.

6.2.3 Data Constraint Property

Mapping tables store data sharing constraints by associating data vocabularies between peers. Although mapping tables impose constraints on the associations of values, they respect the autonomy of the sources whose values they associate. Based on constraints

in mapping tables, we can categorize the data stored in a peer into two categories. The data in a peer that are shared using mapping tables are called shared data (SD), and the data that have no mappings in mapping tables are called local data (LD).

In a data sharing system, a transaction in a peer can access local as well as shared data items. Therefore, a peer is responsible to maintain the consistency of the local as well as shared data. An LDBS in a peer maintains the local database consistency when a transaction accesses only LD items. Meanwhile, if a transaction accesses SD items, then the consistency of SD must be maintained in the local peer as well as in acquaintees. Therefore, when an update occurs on an SD item x through the interface of LDBS in a peer, then the update must be propagated and executed in an acquaintee of the peer in order to maintain the mutual consistency on SD. We can classify the consistency on data items in a data sharing system into two types:

Local consistency: the consistency of the local data items. The constraints are defined in the local database system.

Peer-to-Peer consistency: the consistency of the data items that are shared between peers. The constraints are defined using mapping tables when two peers share there data.

Therefore, the introduction of the peer-to-peer constraints, partitions the data items D_i in a peer P_i into local data (LD_i) and shared data (SD_i), such that $LD_i \cap SD_i = \phi$ and $D_i = LD_i \cup SD_i$. Intuitively, if there is an association of $a_i \in D_i$ and $a_j \in D_j$, $i \neq j$ in a mapping table, then data item a_i and a_j are shared data items in D_i and D_j, respectively. Figure 6.1 shows the categories of data. From the figure we notice that SD is mapped using the mapping tables, and LD is not mapped with any mapping tables. Note that mapping tables coexist with in a local database.

In order to avoid the inconsistency of shared data items between peers, the local transactions must be restricted to access data items in *SD*. In our work, we restrict a local transaction from updating a shared data item. A local transaction, however, is not restricted to read shared data items. Since mapping tables are placed between peers to share data, and global transactions are generated based on data mappings in mapping tables. Therefore, global transactions only access the shared data items, and they have full access on the shared data items. However, a global transaction is restricted to read and write local data items in a peer. This is logical since global transactions are used only to access shared data.

6.3 Problems for Maintaining a Consistent Execution of Concurrent Transactions

A classic technique for preserving database consistency during concurrent execution of transactions is to organize interleaving transactions such that their executions are atomic, recoverable, and serializable. However, classic serializability has known shortcomings when used as a correctness criteria for a distributed computing environments, such as multidatabase systems, transactional workflow executions [76], or P2P systems. First, it requires close coordination and interaction among sites, that is, the sites must agree on the execution of global transactions in a specific and consistent manner. Second, as distributed transactions tend to be long lived, the use of serializability as a correctness criteria would restrict data availability [32].

One of the important issues for distributed multidatabase systems is to maintain global serializability of global transactions without violating the autonomy of local

databases. The main problem occurs due to indirect conflicts between global transactions which cause different serial order of transactions at different sites. These problems have been widely studied and numerous solutions have been proposed, for example, in [33, 3, 66, 28]. Generally, in an MDBS, the global transaction manager (GTM) plays an important role to ensure the global serializability of global transactions in the system.

However, in a data sharing system there is no GTM, and transactions are executed first locally in each peer before being forwarded to the acquaintees. Since global transactions are propagated in a data sharing system from peer to peer along the acquaintances, the globally consistent execution of concurrently executing global transactions in a data sharing system can be achieved by ensuring the consistent execution in each acquaintance that is included in the propagation paths of the global transactions. With respect to the execution of transactions in an acquaintance (i, j), the acquaintance level consistent execution of transactions between P_i and P_j is maintained if the following two conditions are satisfied:

1. All the operations of a transaction must be executed in the same order in peers P_i and P_j of an acquaintance (i, j). Formally, for all acquaintances (j, k) between P_j and P_k $(1 \leq k \leq m)$, if $T_i^j \mathcal{P} T_i^k$, then for all operations o_1,$o_2 \in O_{T_i^k}$, $o_1 \prec_{T_i^k} o_2$ iff $o_1 \prec_{O_{T_i^j}} o_2$.

2. For any concurrent execution of global transactions, it is required to maintain the consistent execution of the transactions over all the acquaintances in the propagation paths of the global transactions. Formally, for any acquaintance (j, k) between P_j and P_k, if there are schedules $S_j = (\Gamma_{S_j}, \prec_{S_j})$ and $S_k = (\Gamma_{S_k}, \prec_{S_k})$ in P_j and P_k respectively, and each transaction in Γ_{S_k} is a translation of a transaction in Γ_{S_j}, then for all operations o_1,$o_2 \in S_k$, $o_1 \prec_{S_k} o_2$ iff $o_1 \prec_{S_j} o_2$

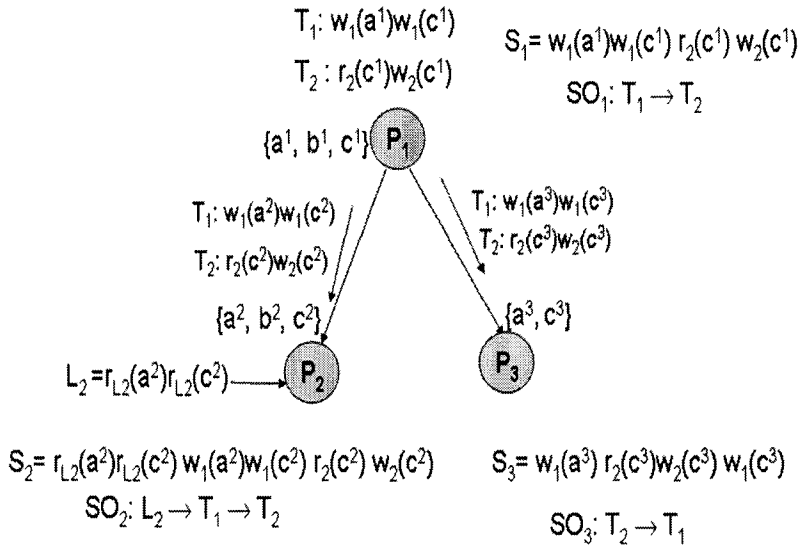


Figure 6.2: Problem to maintain serializability during direct conflicts of transactions

The first condition simply enforces the same execution order of the operations of a transaction at the peers in an acquaintance. The condition can be satisfied easily by forwarding each translated transaction as a single message to the acquaintees. Each acquaintee processes the transaction just like it processes its local transactions. Therefore, the order of the operations of a single transaction is maintained. In order to meet the second condition, we need to ensure the same execution views of transactions in each acquaintee of a peer. Note that the second condition cannot be fulfilled by sending the transactions serially according to the local serialization order of the sender since the sender has no knowledge about the execution order of the transactions at a remote peer. In the following examples, we describe some of the problems that occur during the concurrent execution of global transactions over acquaintees. In Section 6.4, we shall present a correctness criteria to ensure consistent execution of global transactions over an acquaintance.

Example 27 (Direct conflict) *Consider a data sharing system shown in Figure 6.2. Assume that peer P_1 has data items $\{a^1, b^1, c^1\}$, P_2 has data items $\{a^2, b^2, c^2\}$, and P_3 has data items $\{a^3, c^3\}$. Suppose that the transactions T_1 and T_2 executed at P_1 concurrently, and P_1 produced the schedule S_1 as follows:*

$$T_1 : w_1(a^1)w_1(c^1),\ T_2 : r_2(c^1)w_2(c^1) \quad S_1 = w_1(a^1)w_1(c^1)r_2(c^1)w_2(c^1)$$

Suppose the following data mappings exist in the acquaintances.

$$(1,2)\text{: } a^1 \rightarrow a^2,\ b^1 \rightarrow b^2,\ c^1 \rightarrow c^2, \quad (1,3)\text{: } a^1 \rightarrow a^3,\ c^1 \rightarrow c^3.$$

Based on the data accessed by T_1 and T_2, P_1 translates T_1 and T_2, and forwards them to peers P_2 and P_3. For ease of presentation, we keep the same notation of T_1 and T_2, and their translation. The translation of T_1 and T_2 for P_2 and P_3 are as follows:

$$(P_2)\text{: } T_1{=}w_1(a^2)w_1(c^2),\ T_2{=}r_2(c^2)w_2(c^2), \quad (P_3)\text{: } T_1{=}w_1(a^3)w_1(c^3),\ T_2{=}r_2(c^3)w_2(c^3)$$

Assume that peer P_2 executed the following local transaction L_2 concurrently when it executed T_1 and T_2.

$$(P_2) : L_2 = r_{L2}(a^2)r_{L2}(c^2)$$

Consider that after receiving the translated transactions T_1 and T_2, P_2 and P_3 generated the following schedules.

$$S_2 = r_{L2}(a^2)r_{L2}(c^2)w_1(a^2)w_1(c^2)r_2(c^2)w_2(c^2),\ S_3 = w_1(a^3)r_2(c^3)w_2(c^3)w_1(c^3)$$

Therefore, the resulting serialization orders of T_1 and T_2 at P_1, P_2, and P_3 are as follows:

$$SO_1 : T_1 \rightarrow T_2, \quad SO_2 : L_2 \rightarrow T_1 \rightarrow T_2, \quad SO_3 : T_2 \rightarrow T_1$$

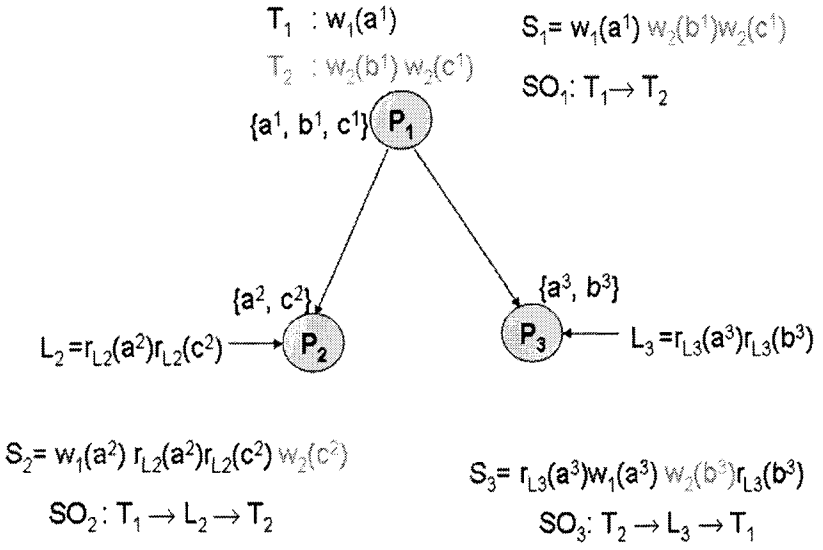


Figure 6.3: Problem to maintain serializability during indirect conflicts of transactions

Notice that each local schedule in each peer is serializable, but the execution view of T_1 and T_2 in the schedule S_3 at P_3 is different with respect to the schedule S_1 at P_1. Since each peer executes transactions independently, and there is no central controller, the resulting schedules at different peers may be different. In order to keep the peer databases consistent with each other, the execution view of the transactions should be the same in each peer.

Example 28 (Indirect conflict) *In this example, we show how the local transactions cause different execution views of transactions in the acquaintees of a peer P_i even though the transactions have no conflict when the transactions executed at P_i. Consider Figure 6.3, where transactions T_1 and T_2 executed concurrently at P_1, and the local transaction manager at P_1 produced the schedule S_1:*

$$T_1 : w_1(a^1),\ T_2 : w_2(b^1)w_2(c^1) \quad S_1 = w_1(a^1)w_2(b^1)w_2(c^1)$$

Based on the data mappings, P_1 generates the following transactions for P_2 and P_3, and forwards the translated transactions to them.

$$(P_2)\text{: } T_1{=}w_1(a^2),\ T_2{=}w_2(c^2),\ \ (P_3)\text{: } T_1{=}w_1(a^3),\ T_2{=}w_2(b^3)$$

Assume that the following local transactions executed at the same time when P_2 and P_3 received T_1 and T_2.

$$(P_2)\ {:}L_2 = r_{L2}(a^2)r_{L2}(c^2),\ (P_3)\ {:}L_3 = r_{L3}(a^3)r_{L3}(b^3)$$

Consider that P_2 and P_3 generated the following schedules.

$$S_2 = w_1(a^2)r_{L2}(a^2)r_{L2}(c^2)w_2(c^2),$$
$$S_3 = r_{L3}(a^3)w_1(a^3)w_2(b^3)r_{L3}(b^3)$$

Notice that when T_1 and T_2 executed at P_1, there was no conflict between the transactions. Meanwhile, when T_1 and T_2 executed at P_3, they involved to indirect conflict due to the presence of the local transaction L_3. Based on the execution views of T_1 and T_2 at P_2 and P_3, we observe the following serialization orders.

$$SO_2 : T_1 \rightarrow L_2 \rightarrow T_2, \quad SO_3 : T_2 \rightarrow L_3 \rightarrow T_1$$

Notice that the serialization orders of T_1 and T_2 at P_2 and P_3 are different. Hence, acquaintance-level consistent execution is not maintained.

In a multidatabase environment, the GTM has the control over the execution of global transactions and the operations they issue. The GTM can ensure the global serializability by a direct or indirect control of the global transactions. For example, altruistic locking [3], 2PC agent [88], site graph [20], and ticketing [33]. All the methods have a global transaction manager which plays an important role in ensuring the global

serializability. Since in a data sharing system there is no such GTM, the only assumption we can make is that each peer ensures the local serializability. Once the transactions are forwarded to the acquaintees, the peer has no control of the execution of transactions at the acquaintees.

6.4 Maintaining Acquaintance-Level Consistent Execution of Transactions

Since in a data sharing system transactions are executed locally and independently of peers, the system does not require multi-site commit protocols (e.g. two-phase commit), which tend to introduce blocking and are thus not easily scalable. Specifically, in a peer data sharing system, transactions are executed locally, and then asynchronously propagated over acquaintances after their commitment. A consistent execution of transactions in a peer data sharing system can be achieved by ensuring the consistent execution of transactions in each peer over each acquaintance that is included in the propagation path of the transactions.

We observe from the examples in Section 6.3 that the inconsistent execution of transactions occur at different peers due to the independent execution. However, for ensuring database consistency in acquainted peers over the acquaintances with respect to the local execution of a peer, the execution views of the transactions must be same in all the acquaintees of a peer if there are direct conflicts between transactions. Fortunately, we don't need to be worried about an indirect conflict between the transactions when it occurs in acquaintees. An indirect conflict occurs due to the local transactions in an acquaintee. If transactions have no conflict at the time they originate in a peer, then these

transactions can be executed in any order in the acquaintees. Since the data constraint property restricts the access of the local and global transactions in a database, different execution views of transactions due to the indirect conflicts do not create any database inconsistency. This is because a global transaction does not read a data item that is written by a local transaction. The conflicts that can occur based on the data constraint property between a local transaction L and a global transaction T_1 are *write-read* and *read-write*. A write-read conflict between T_1 and L occurs for accessing a data item a, when a read operation of L is followed by a write operation of T_1. A read-write conflict occurs when a write operation of T_1 is followed by a read operation of L. Therefore, if L has a write-read conflict with T_1, then L does not create a read-write conflict for the same data item with another transaction T_2. This is because T_1 and T_2 had no conflict when they initiated. Similarly, when L has a read-write conflict with T_1, then L does not create a write-read conflict with another transaction T_2. Therefore, when two global transactions T_1 and T_2 execute at a peer and have no conflict, then their different execution orders in the acquaintees of the peer does not create any inconsistency. In the following, we generalize the two problems, and introduce the notions for ensuring a consistent execution of transactions in a peer and its acquaintees.

Definition 27 (Acquaintance-Level Schedule) *An acquaintance-level schedule* $\mathbb{S}_i = S_i \cup (\bigcup_{j=1}^{m} S_j)$ *with respect to a schedule* S_i *at peer* P_i *for a set of transactions* Γ *is the union of the schedule* S_i *and all the schedules* S_j *at peer* P_j $(1 \leq j \leq m)$*, where each* P_j *is an acquaintee of* P_i*.*

Definition 28 (Acquaintance-Level Consistent Schedule) *An acquaintance-level schedule* $\mathbb{S}_i$ *is called acquaintance-level consistent with respect to a schedule* S_i *in* P_i *for a set of transactions* $\Gamma = \{T_1, T_2, \cdots, T_n\}$ *and all schedules* S_j *in* P_j *over each acquaintance*

(i, j), $(1 \leq j \leq m)$ if

1. *all the local schedules in $\mathbb{S}_i$ are serializable, and*

2. *for any two transactions T_1 and T_2 in S_i, if there exist a serialization order (SO) $T_1 \rightarrow T_2$, then for all schedules $S_j \in \mathbb{S}_i (i \neq j)$, the SO is consistent between T_1 and T_2*

Definition 29 (Acquaintance-Level Serializable) *An acquaintance-level consistent schedule $\mathbb{S}_i$ is called acquaintance-level serializable wrt S_i between peers P_i and all acquaintees P_j of P_i $(1 \leq j \leq m)$ if $\mathbb{S}_i$ is acquaintance-level consistent schedule.*

Definition 30 (Global Schedule) *A global schedule $\mathbb{S} = \mathbb{S}_i \cup (\bigcup_{j=1}^{n} \mathbb{S}_j)$ over a set of global transactions $\Gamma = \{T_1, T_2, \cdots, T_n\}$ initiated at P_i, consists of the acquaintance-level schedule $\mathbb{S}_i$ wrt S_i at P_i and all the acquaintance-level schedules $\mathbb{S}_j$ wrt S_j at P_j, $(1 \leq j \leq n, i \neq j)$ in a data sharing system where Γ executes.*

A global consistent execution of transactions can be achieved by maintaining the *acquaintance-level* serializability over each acquaintance in the propagation paths of the global transactions.

Theorem 5 *A global schedule $\mathbb{S}$ consisting of a set of global transactions $\Gamma = \{T_1, T_2, \cdots, T_n\}$ is consistent over a propagation path $(P_i \rightarrow \cdots \rightarrow P_z)$ with respect to a schedule S_i at P_i if for each acquaintance in $(P_i \rightarrow \cdots \rightarrow P_z)$, $\mathbb{S}$ is acquaintance-level serializable.*

Proof 6 *(By induction over the length of the path from P_i to P_z)*

Let l be the length of the propagation path of Γ from P_i to P_z.

Case $l = 0$: Γ executed only at P_i, and there is no further propagation of Γ. According

to our assumption that each local schedule is serializable. Hence, the global schedule $\mathbb{S}$*, which consists of only the schedule* S_i*, is consistent.*

Case $l = 1$*:* Γ *executed at* P_i *and an acquaintee* P_j *of* P_i *over an acquaintance* (i, j)*. Since* $\mathbb{S}$ *is serializable over a single acquaintance* (i, j) *according to Definition 29, the serialization orders of* Γ *in* S_i *and* S_j *are same. Hence,* $\mathbb{S}$ *is consistent over the path* $(P_i \rightarrow P_j)$*.*

Case $(0 \leq k \leq l)$*: For the induction step, we assume that consistency holds along the path between* P_i *and* P_k *recursively in each acquaintance, where* P_k *is a peer before* P_z*. Now we need to show that serializability holds between* P_k *and* P_z*, where* $l = k+1$*. Since* $\mathbb{S}$ *is consistent over the path* $(P_i \rightarrow \cdots \rightarrow P_k)$ *and* P_k*, and* P_z *are directly acquainted,* $\mathbb{S}$ *is consistent in* $(P_k \rightarrow P_z)$*. Hence, global consistency holds over the path* $(P_i \rightarrow \cdots \rightarrow P_z)$*.* ■

Definition 31 (Global Consistency) *A global schedule* $\mathbb{S}$ *over a set of global transactions* Γ *initiated at* P_i *is globally consistent if*

1. *all local schedules in* $\mathbb{S}$ *are serializable, and*
2. *for each acquaintance* (j, k) *over all the propagation paths* $(P_i \rightarrow \cdots \rightarrow P_z)$*,* $\mathbb{S}$ *is acquaintance-level serializable.*

Theorem 6 *A global schedule* $\mathbb{S}$ *consisting of a set of global transactions* $\Gamma = \{T_1, T_2, \cdots, T_n\}$ *is globally consistent with respect to an initial schedule* S_i *at* P_i *if for all propagation paths* $(P_i \rightarrow \cdots \rightarrow P_z)$*, and for each acquaintance in each path* $(P_i \rightarrow \cdots \rightarrow P_z)$*,* $\mathbb{S}$ *is acquaintance-level serializable, and each path between* P_i *and* P_z *is acyclic.*

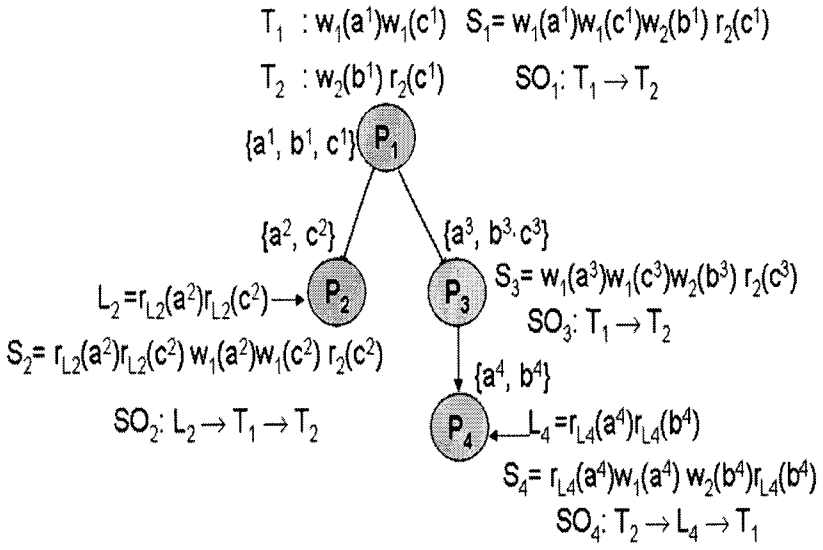


Figure 6.4: An example of global consistency

Proof 7 *According to Theorem 5,* $\mathbb{S}$ *is consistent in a path of* Γ*'s propagation. It means that* Γ *is serializable in each path* $(P_i \rightarrow \cdots \rightarrow P_z)$*. Moreover, each path is acyclic. Therefore,* $\mathbb{S}$ *is globally consistent.* ■

Example 29 *Consider Figure 6.4 where two global transactions* T_1 *and* T_2 *executed concurrently at* P_1*,* P_2*,* P_3 *and* P_4*. Assume that transactions are initiated at* P_1*. In the scenario, the global schedule* $\mathbb{S}$ *is* $\{\mathbb{S}_1, \mathbb{S}_2, \mathbb{S}_3, \mathbb{S}_4\}$*. From the figure we notice that all the local schedules are locally serializable. The initial schedule* S_1 *has the serialization order* $SO_1 = T_1 \rightarrow T_2$*. The acquaintance-level schedule* $\mathbb{S}_1$ *with respect to* S_1 *at* P_1 *is* $\{S_1, S_2, S_3\}$*. We see that* $\mathbb{S}_1$ *is acquaintance-level serializable schedule because both* SO_2 *and* SO_3 *have the serialization order* $T_1 \rightarrow T_2$*, but* $\mathbb{S}_3$ *is not acquaintance-level serializable because* SO_4 *at* P_4 *is not consistent with* SO_3 *of* P_3*. Therefore,* $\mathbb{S}$ *is not globally consistent.*

6.5 Maintaining Acquaintance-Level Consistent Schedule

The acquaintance-level serializability guarantees the same execution view of a set of transactions Γ in a peer and in its acquaintees. Although there is no GTM in a peer data sharing system, a globally consistent execution of transactions can be achieved by ensuring acquaintance-level serializability in each propagation paths of the transactions. In order to maintain the acquaintance-level serializability, we need to guarantee a consistent serialization order of the transactions at all the acquaintees of a peer. We know that when a set of transactions Γ is executed at a peer P_i, the transactions are executed immediately at P_i. Therefore, P_i generates a local schedule S_i without waiting for the execution of Γ in its acquaintees. After execution, P_i forwards Γ to its acquaintees. The execution and forward steps continue until no propagation of Γ is possible. Each of the peer P_j executes Γ with the local concurrency control, and generates a local schedule S_j independently. The main challenge is how to guarantee a consistent serialization order in all S_j with respect to the order in S_i. In the following, we propose two approaches that guarantee acquaintance-level serializability.

6.5.1 First Approach

When a set of global transactions Γ is initiated at a peer or is received by a peer, the peer immediately executes Γ with the local concurrency control protocol. Therefore, the peer generates a local serializable schedule forming a specific serialization order of the transactions in Γ. In this method, we assume that a function called *returnSchedule*() is used by each peer that returns the locally generated schedule of Γ. Note that the local

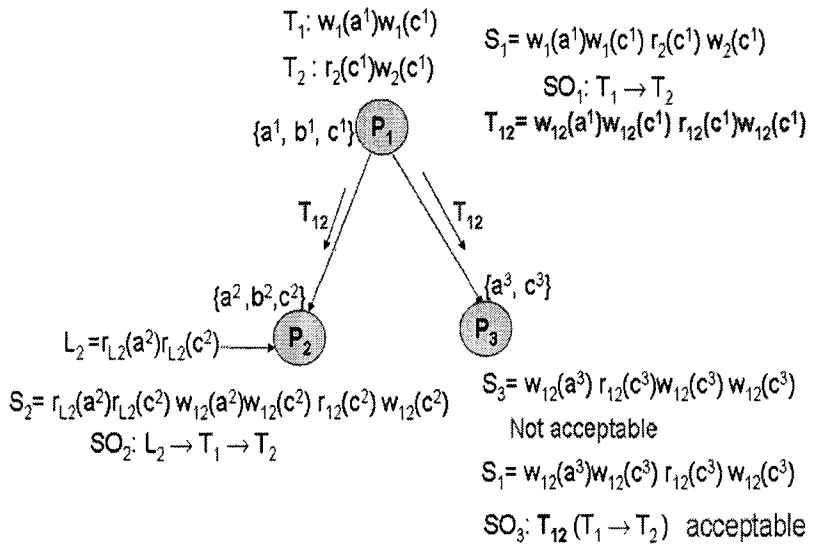


Figure 6.5: Maintaining serializability during direct conflict using the first approach

schedule may contain the operations of the local transactions. The *returnSchedule*() function returns only the operations of Γ in the same order appeared in the schedule. We assume that the function is added externally in the system. A peer treats the schedule returned by *returnSchedule*() as an atomic transaction, and translates the operations in the schedule for each of its acquaintees using the mappings. The peer then forwards the schedule as a new transaction to its acquaintees. When an acquaintee receives the schedule (now a transaction), the acquaintee peer processes the schedule as it processes a transaction. Note that treating a schedule as a single transaction, the receiver keeps the same order of the operations that is generated by the sender in its schedule.

In the following, we describe the method with examples. We show that the acquaintance-level serializability is maintained taking into account of direct and indirect conflicts of transactions.

Example 30 (Direct conflict) *Consider the situation of Example 27. The local schedule generated at P_1 is as follows. Figure 6.5 shows the scenario.*

$$S_1 = w_1(a^1)w_1(c^1)r_2(c^1)w_2(c^1).$$

According to the method, P_1 creates a new transaction T_{12} for its acquaintees P_2 and P_3 from the schedule S_1. The order of the operations of T_{12} follows the order as mentioned in the schedule S_1.

$$(P_2) : T_{12} = w_{12}(a^2)w_{12}(c^2)r_{12}(c^2)w_{12}(c^2), \quad (P_3) : T_{12} = w_{12}(a^3)w_{12}(c^3)r_{12}(c^3)w_{12}(c^3)$$

Consider that P_2 and P_3 received the transaction T_{12} and executed it. Note that it is not possible for peer P_3 to generate the same schedule as described in Example 27 since transactions T_1 and T_2 are considered as a single transaction. Thus, operations $r_2(c^3), w_2(c^3)$ of T_2 can not come before the operation $w_1(c^3)$ of T_1. The only possible schedule P_2 and P_3 can generate are as follows:

$$S_2 = r_{L2}(a^2)r_{L2}(c^2)w_{12}(a^2)w_{12}(c^2)r_{12}(c^2)w_{12}(c^2), \; S_3 = w_{12}(a^3)w_{12}(c^3)r_{12}(c^3)w_{12}(c^3)$$

Note that the transaction T_{12} contains the operations of T_1 and T_2. As T_{12} is an atomic transaction for the transaction manager of P_2 and P_3, therefore the operations in T_{12} are executed in the same order as the operations were executed at P_1. Hence, the serialization order of T_1 and T_2 must be the same at P_1, P_2, and P_3. Therefore, acquaintance-level serializability is maintained with respect to the schedule S_1.

Example 31 (Indirect conflict) *Consider Example 28, there is no conflict between T_1 and T_2 when they executed at P_1. Assume that the following schedule is generated at P_1.*

$$S_1 = w_1(a^1)w_2(b^1)w_2(c^1)$$

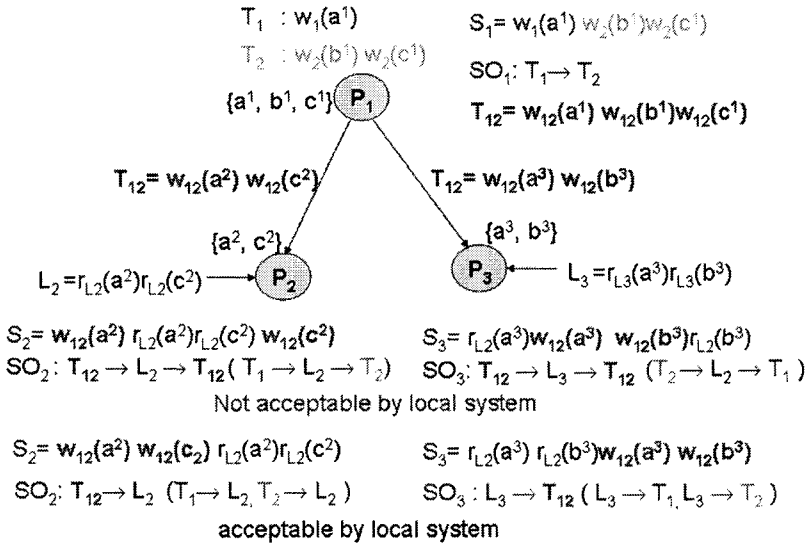


Figure 6.6: Maintaining serializability during indirect conflict using the first approach

According to the method, P_1 creates the following transactions for its acquaintees P_2 and P_3 from the schedule S_1.

$$(P_2) : T_{12}{=}w_1(a^2)w_2(c^2), \quad (P_3) : T_{12}{=}w_1(a^3)w_2(b^3)$$

Consider the same execution view of the operations as mentioned in Example 28. Therefore, P_2 and P_3 try to generate the following schedules. Figure 6.6 shows the schedules generated by each peer.

$$S_2 = w_{12}(a^2)r_{L2}(a^2)r_{L2}(c^2)w_{12}(c^2),\ S_3 = r_{L3}(a^3)w_{12}(a^3)w_{12}(b^3)r_{L3}(b^3)$$

We notice from Figure 6.6 that the schedules S_2 and S_3 are not allowed by the local concurrency control of P_2 and P_3 since both schedules have cycles. In P_2, either L_2 or T_{12} is blocked or aborted. If the local schedule in P_2 were

$$S_2 = r_{L2}(a^2)r_{L2}(c^2)w_{12}(a^2)w_{12}(c^2) \text{ or } S_2 = w_{12}(a^2)w_{12}(c^2)r_{L2}(a^2)r_{L2}(c^2)$$

then the schedule would be permitted by the local concurrency control at P_2.

Similarly, if the local schedule at P_3 were

$$S_3 = w_{12}(a^3)w_{12}(b^3)r_{L3}(a^3)r_{L3}(b^3) \text{ or } S_3 = r_{L3}(a^3)r_{L3}(b^3)w_{12}(a^3)w_{12}(b^3)$$

then the schedule would be permitted by the local concurrency control at P_3 because there is no cycle in the schedule. We notice that from the above executions, both P_2 and P_3 schedule the transactions T_1 and T_2 logically in the same order. Therefore, the acquaintance-level serializability is maintained with respect to S_1.

6.5.2 Second Approach

In this approach, we exploit the concept of the Optimistic Ticket Method (OTM) [33]. According to this approach, a peer includes an extra operation $w(t)$ before the first operation of each transaction when the transactions are forwarded to the acquaintees. The $w(t)$ is a write ticket operation. A ticket is a (logical) timestamp whose value is stored as a regular data item in each LDBS [33]. The intuition behind the use of $w(t)$ operation is to create a relative serialization order of the global transactions. The inclusion of $w(t)$ operation does not violate the autonomy of LDBS nor does pose any restriction in the LDBS. The inclusion of $w(t)$ operation is outside the scope of LTM. We also assume that there is a function called *getSeriliazeOrder*() that returns the serialization order of the executed global transactions in peers. When a peer forwards global transactions, it sends the transactions according to the serialization order as returned by the function *getSeriliazeOrder*(). This can be performed by delaying the propagation of the transactions. When a remote peer receives transactions, it processes the transactions accordingly and is allowed to interleave the operations of the transactions under the control of the LDBS. Note that adding the $w(t)$ operation creates a direct conflict between the

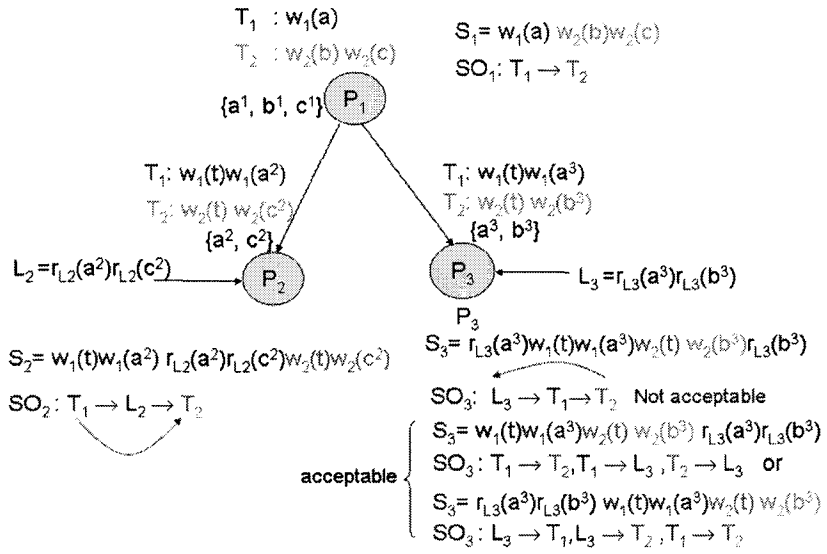


Figure 6.7: Maintaining serializability during indirect conflict by the second approach

transactions at acquaintee peers. Therefore, transactions are serialized in the acquaintee peers as determined by the sender.

In the following, we describe the protocol using an example considering that there is no direct conflict between transactions when the transactions are executed in a peer.

Example 32 (Indirect conflict) *Consider Example 28. The returnSchedule() function returns the following schedule at* P_1.

$$S_1 = w_1(a^1)w_2(b^1)w_2(c^1).$$

According to the method, P_1 *creates the following transactions adding* $w(t)$ *operation to* T_1 *and* T_2 *before forwarding them to the acquaintees* P_2 *and* P_3.

$$(P_2)\text{:}T_1\text{=}w_1(t)w_1(a^2),\ T_2\text{=}w_2(t)w_2(c^2),\ (P_3)\text{:}T_1\text{=}w_1(t)w_1(a^3),\ T_2\text{=}w_2(t)w_2(b^3)$$

Peer P_1 *uses the getSeriliazeOrder() function to find the serialization order of* T_1 *and*

T_2. From the schedule, we see that T_1 is executed before T_2 in S_1. Therefore, P_1 sends T_1 before T_2 to its acquaintees P_2 and P_3, respectively.

Consider that the following schedules are generated by P_2 and P_3 after receiving T_1 and T_2. Figure 6.7 shows the schedules generated by each peer.

$$S_2 = w_1(t)w_1(a^2)r_{L2}(a^2)r_{L2}(c^2)w_2(t)w_2(c^2),\ S_3 = r_{L3}(a^3)w_1(t)w_1(a^3)w_2(t)w_2(b^3)w_{L3}(b^3)$$

We notice from Figure 6.7 that the schedule S_3 is not allowed by the local concurrency control of P_3 because there is a cycle in the schedule. That is, either L_3 or T_2 is blocked or aborted. On the other hand, if the local schedule in P_3 were

$$S_3 = w_1(t)w_1(a^3)w_2(t)w_2(b^3)r_{L3}(a^3)w_{L3}(b^3),\ S_3 = w_1(t)w_1(a^3)r_{L3}(a^3)w_{L3}(b^3)w_2(t)w_2(b^3),$$

then the schedule would be permitted by the local concurrency control at P_3. In this case, we see that the execution views of transactions T_1 and T_2 are same in all the acquainted peers of P_1. Therefore, it ensures acquaintance-level serializability. The situation is depicted in Figure 6.7. Similarly, for Example 27, we can show that acquaintance-level serializability can be maintained using the ticket method.

Example 33 (Direct conflict) *Consider example 27. The returnSchedule() function returns the following schedule at P_1.*

$$S_1 = w_1(a^1)w_1(c^1)r_2(c^1)w_2(c^1).$$

According to the method, P_1 now creates the following transactions by adding $w(t)$ operation to T_1 and T_2 before forwarding them to the acquaintees P_2 and P_3.

$$(P_2)\text{:}T_1{=}w_1(t)w_1(a^2)w_1(c^2),\ T_2{=}w_2(t)r_2(c^2)w_2(c^2),\ (P_3)\text{:}T_1{=}w_1(t)w_1(a^3w_1(c^3)),$$
$$T_2{=}w_2(t)r_2(c^3)w_2(c^3)$$

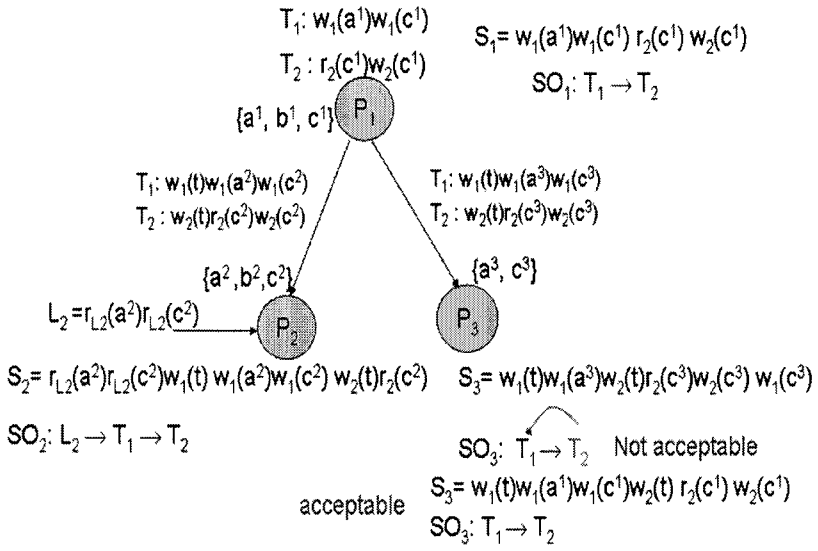


Figure 6.8: Maintaining serializability during direct conflict in the second approach

Now P_1 uses the getSeriliazeOrder() function to find the serialization order of T_1 and T_2. From the schedule, we see that T_1 is serialized before T_2 in S_1. Therefore, P_1 sends T_1 before T_2 to its acquaintees P_2 and P_3.

When P_2 and P_3 receive T_1 and T_2, consider that they try to generate the same schedules as mentioned in Example 27. Therefore, the generated schedules are as follows. Figure 6.8 shows the schedules generated by each peer.

$$S_2 = r_{L2}(a^2)r_{L2}(c^2)w_1(t)w_1(a^2)w_1(c^2)w_2(t)r_2(c^2)w_2(c^2),$$
$$S_3 = w_1(t)w_1(a^3)w_2(t)r_2(c^3)w_2(c^3)w_1(c^3)$$

Notice from Figure 6.8 that the schedule S_3 is not allowed by the local concurrency control of P_3. Since there is a cycle in the schedule. On the other hand, if the local schedule in P_3 is

$$S_3 = w_1(t)w_1(a^3)w_1(c^3)w_1(t)r_2(c^3)w_2(c^3),$$

then the schedule would be permitted by the local concurrency control at P_3. In this case, we see that the execution view of transactions T_1 and T_2 at P_3 is same as in P_1. Therefore, ensures acquaintance-level serializability.

6.6 Discussion

There are some differences in the way our ticket method works with OTM. Although ticket is used in an MDBS, but still GTM is involved to validate the serialization order of the transactions. The validation is performed using Global Serialization Graph (GSG). An edge (T_i, T_j) of GSG means that ticket operation of a global subtransaction T_i is preceded by that of another global subtransaction T_j at least at one site. The set of edges reflects the serialization order. If the GSG has a cycle, the global transaction does not pass the validation test. Then the GTM sends abort message to the sites for restarting again. In our work, we do not need any GSG, since each peer executes the transactions independently. Before forwarding the transactions to a peer, each transaction is augmented with a write ticket operation. The peer which sends the transactions knows the serialization order by its local schedule. Therefore, the order of taking the ticket, i.e. the serialization order of the transactions at a remote peer is predefined and maintained by delaying the propagation of the transactions.

6.7 Ensuring Acquaintance-Level Consistent Execution Considering Failures of Transactions

The last section proposed mechanisms to ensure consistent execution of concurrent transactions over the acquaintances of a peer. However, the approaches assume that the failures of transactions do not occur during their execution.

There are several failure-prone situations can occur in a peer to peer network. For example, a peer may go offline when a transaction is active in the network, a peer may fail due to power fails or the system crashes, or a peer has successfully executed a transaction but the transaction has failed in one of its acquaintees. Examples of a transaction failure are a transaction abort to timeout, or a failure to pass the validation test by the local transaction manager. We can consider an offline status of a peer as a failure of a peer. In this section we mainly focus on the failures of transactions and how these cause problems for guaranteeing consistent execution of transactions in a peer and in its acquaintees. The following examples show that a peer and its acquaintees may have different execution views for the same set of transactions when a transaction fails. We consider the situations of direct and indirect conflicts between transactions at the time transactions are executed in peers.

Example 34 *Consider the situation of a direct conflict between transactions at the time transactions execute in peers. Assume a peer to peer network with three peers shown in Figure 6.9. We see from the figure that peer* P_1 *has data items* $\{a^1, b^1, c^1\}$, P_2 *has data items* $\{a^2, c^2\}$, *and* P_3 *has data items* $\{a^3, b^3\}$. *Suppose that transactions* T_1 *and* T_2 *executed at* P_1 *concurrently, and the local database system at* P_1 *produced the schedule* S_1*:*

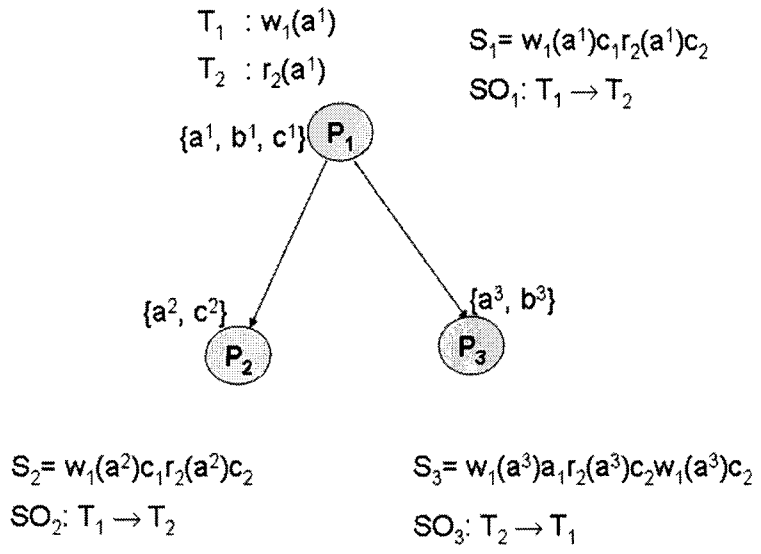


Figure 6.9: Direct conflict

$$T_1 : w_1(a^1),\ T_2 : r_2(a^1)$$
$$S_1 = w_1(a^1)c_1r_2(a^1)c_2$$

Suppose that the following data mappings exist between the acquaintances.

$$(1,2)\text{: } a^1 \rightarrow a^2,\ c^1 \rightarrow c^2,\ \ (1,3)\text{: } a^1 \rightarrow a^3,\ b^1 \rightarrow b^3.$$

Based on the data accessed by T_1 and T_2, P_1 translates T_1 and T_2 and forwards them to P_2 and P_3. The translation of T_1 and T_2 for P_2 and P_3 are as follows:

$$(P_2)\text{: } T_1{=}w_1(a^2),\ T_2{=}r_2(a^2),$$
$$(P_3)\text{:} T_1{=}w_1(a^3),\ T_2{=}r_2(a^3)$$

For ease of presentation, we keep the same notation of T_1 and T_2, and their translations. Assume that after receiving the transactions, P_2 and P_3 generated the following schedules.

$$S_2 = w_1(a^2)c_2r_2(a^2)c_2,\ S_3 = w_1(a^3)a_1r_2(a^3)c_2w_1(a^3)c_1$$

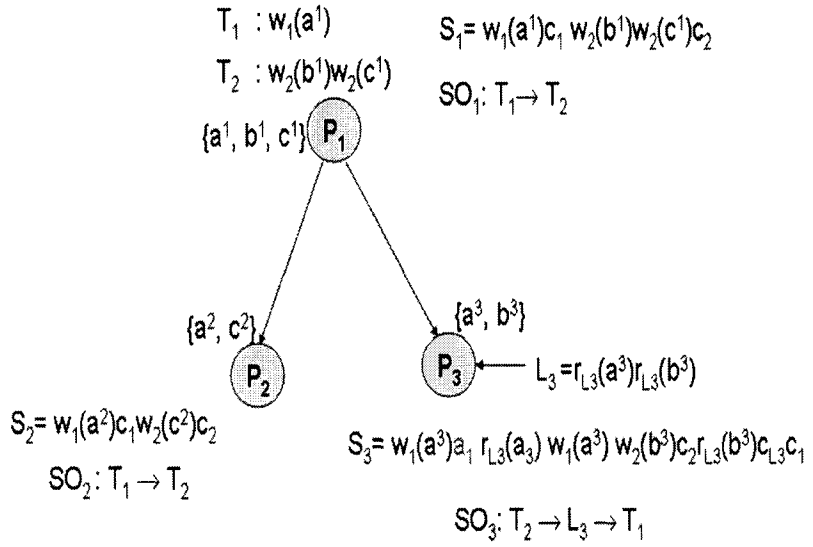


Figure 6.10: Indirect conflict

The resulting serialization orders of the transactions T_1 and T_2 at peers P_1, P_2, and P_3 are as follows:

$$SO_1 : T_1 \rightarrow T_2, \quad SO_2 : T_1 \rightarrow T_2, \quad SO_3 : T_2 \rightarrow T_1$$

Notice that each local schedule in each peer is serializable, but the execution view of T_1 and T_2 in the schedule S_3 of P_3 is different with respect to the schedule S_1 in P_1. The execution view is different since T_1 has aborted in P_1 and later is resubmitted after the execution of T_2.

Example 35 (Indirect conflict) *In this example, we show how the local transactions in a peer cause different execution views of transactions in acquaintees of a peer P_i even though transactions have no conflict when they executed at P_i. Consider Figure 6.10 where the transactions T_1 and T_2 executed at P_1, and the local transaction manager at P_1 produced the schedule S_1:*

$$T_1 : w_1(a^1),\ T_2 : w_2(b^1)w_2(c^1)$$
$$S_1 = w_1(a^1)c_1w_2(b^1)w_2(c^1)c_2$$

According to the data mappings, P_1 generates the following transactions for P_2 and P_3, and forwards the translated transactions to them. Note that T_2 has been translated partially for P_2 and P_3 due to data mappings.

$$(P_2) : T_1{=}w_1(a^2),\ T_2{=}w_2(c^2),$$
$$(P_3) : T_1{=}w_1(a^3),\ T_2{=}w_2(b^3)$$

Assume that peer P_3 has a local transaction L_3 when P_3 executes T_1 and T_2:

$$(P_3) : L_3 = r_{L3}(a^3)r_{L3}(b^3)$$

Consider that P_2 and P_3 generates the following schedules.

$$S_2 = w_1(a^2)c_1w_2(c^2)c_2,$$
$$S_3 = w_1(a^3)a_1r_{L3}(a^3)w_1(a^3)w_2(b^3)c_2r_{L3}(b^3)c_{L3}c_1$$

Notice that when T_1 and T_2 executed at P_1, there was no conflict between the transactions. Meanwhile, when T_1 and T_2 are executed at P_3, they involve in indirect conflict due to the presence of a local transaction L_3 at P_3. Based on the execution views in P_2 and P_3, we observe the following serialization order.

$$SO_2 : T_1 \rightarrow T_2, \quad SO_3 : T_2 \rightarrow L_3 \rightarrow T_1$$

Note that the serialization orders of T_1 and T_2 in P_2 and P_3 are different. In P_3, T_1 fails and later is resubmitted, this causes different execution view of transactions in P_3. Since in peers P_1 and P_2, T_1 and T_2 have no conflict, we consider commit order as serialization order.

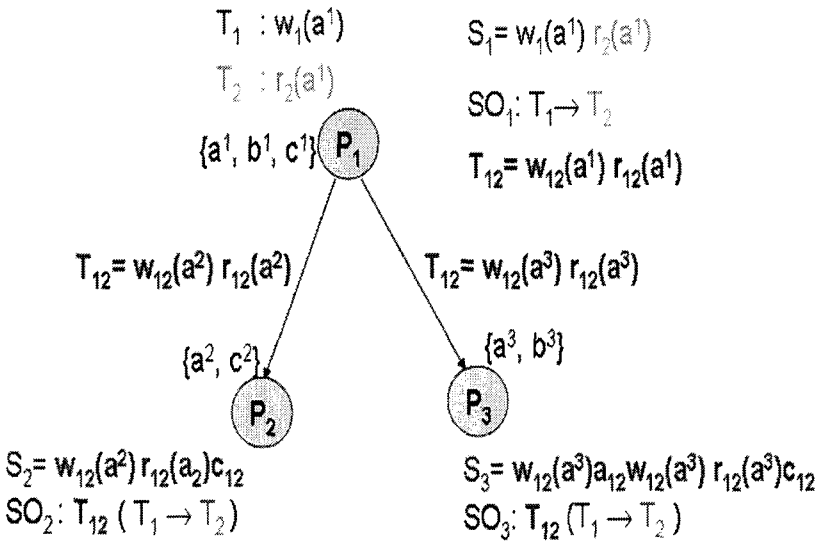


Figure 6.11: Maintaining acquaintance-level consistent execution during direct conflict

In a failure-prone environment, a two phase commit (2PC) protocol is needed to guarantee atomic commitment of subtransactions of a multi-site transaction processing environment [42]. Since an LDBS of a site in such an environment may not support a visible Prepared state of the 2PC protocol, a site uses a 2PC agent or server to simulate the 2PC protocol between a GTM and 2PC agents [88] and servers. Since in a peer data sharing system there is no such GTM, the only assumption we can make is that each peer ensures its local serializability. Once the transactions are forwarded to the acquaintees, the peer has no control of the execution of transactions at the acquaintees.

Now we show how the first approach ensures the acquaintance-level consistent execution during the failures of transactions. Again, we consider both the direct and indirect conflicts between transactions when transactions execute in the system.

Example 36 (Direct conflict) *Consider the situation of Example 34. The local schedule generated at P_1 is as follows:*

$$S_1 = w_1(a^1)c^1r_2(a^1)c^2.$$

According to the approach 1, P_1 creates a transaction T_{12} for its acquaintees P_2 and P_3 from the schedule S_1. The order of operations in T_{12} follows the order as mentioned in the schedule S_1. Figure 6.11 shows the scenario.

$$(P_2) : T_{12} = w_{12}(a^2)r_{12}(a^2),\ (P_3) : T_{12} = w_{12}(a^3)r_{12}(a^3)$$

Consider the same execution of operations at P_2 and P_3 as in Example 6.9.

$$S_2 = w_{12}(a^2)r_{12}(a^2)c_{12},\ S_3 = w_{12}(a^3)a_{12}w_{12}(a^3)r_{12}(a^3)c_{12}$$

Note that at peer P_3, after the operation w_{12} the transaction T_{12} aborts. Therefore, no more operations of T_{12} will be executed until w_{12} successfully executes. After another try T_{12} is executed. Hence, the order of operations of T_1 and T_2 in P_3 remains same as it is in P_1. Therefore, acquaintance-level consistent execution is maintained at P_1, P_2, and P_3 with respect to the schedule S_1.

Example 37 (Indirect conflict) *Consider Example 35, there is no conflict between T_1 and T_2 when they executed at P_1. Assume that the following schedule is generated in P_1.*

$$S_1 = w_1(a^1)w_2(b^1)w_2(c^1)$$

According to the method, P_1 creates the following transactions for its acquaintees P_2 and P_3 from the schedule S_1. Figure 6.12 shows the scenario.

$$(P_2) : T_{12}{=}w_{12}(a^2)w_{12}(c^2),\ \ (P_3) : T_{12}{=}w_{12}(a^3)w_{12}(b^3)$$

Assume that the following schedules result at P_2 and P_3 respectively. We consider the same execution sequence of operations as illustrated in Example 35.

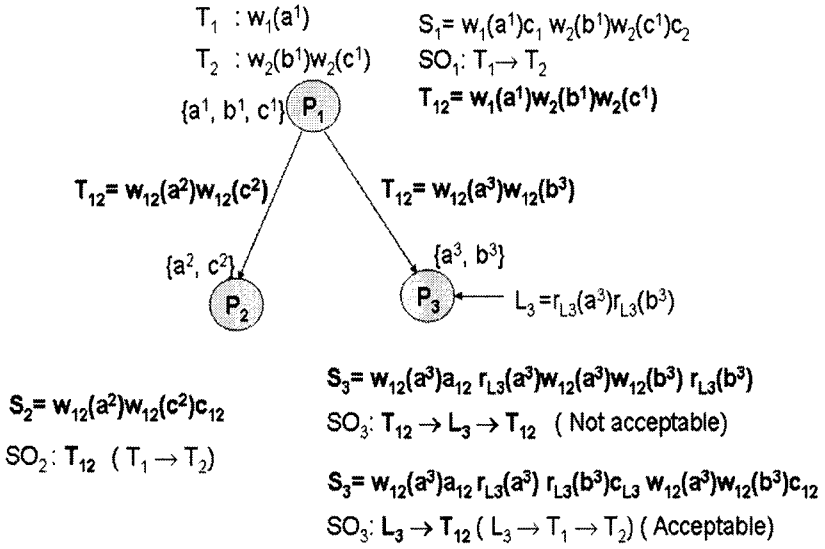


Figure 6.12: Maintaining acquaintance-level consistent execution during indirect conflict

$$S_2 = w_{12}(a^2)w_{12}(c^2)c_{12},\ S_3 = w_{12}(a^3)a_{12}r_{L3}(a^3)w_{12}(a^3)w_{12}(b^3)r_{L3}(b^3)$$

Notice that the schedule S_3 is not allowed by the local concurrency control in P_3 since local schedule contains a cycle ($T_{12} \rightarrow L_3 \rightarrow T_{12}$). If the local schedule in P_3 is

$$S_3 = w_{12}(a^3)a_{12}r_{L3}(a^3)r_{L3}(b^3)c_{L3}w_{12}(a^3)w_{12}(b^3)c_{12}$$

then the schedule would be permitted by the local concurrency controller in P_3. Note that from the above execution, both P_2 and P_3 schedule the transactions T_1 and T_2 logically in the same order. Therefore, the acquaintance-level consistent execution is maintained with respect to S_1.

6.7.1 Discussion

Our assumption of transaction processing in a peer data sharing system is that transactions are not generated continuously. We do not expect that users continuously submit

update requests in a P2P system. In a P2P system, generally, queries are more frequent than updates. Even, if transactions are continuous, according to our system, a peer forwards transactions to acquainted peers after the complete execution in the local peer.

Both the proposed approaches guarantee acquaintance-level serializability, however there are some differences between them. The first approach is more efficient compared to preprocessing of transactions before propagating concurrent transactions to remote peers. In the first approach, transactions are executed and propagated immediately after the local execution. In the second approach each transaction is augmented with an extra write ticket operation before propagation. In the second approach, however, local schedule information is not required and transactions are propagated like the transactions are received from users. In contrast, in the first approach, transactions are merged and propagated to remote peers as a single transaction. Therefore, generating the schedule may take some time. The most important difference between the two approaches is that the Ticket approach can not always guarantee acquaintance-level serializability during transaction failures. Moreover, the Ticket approach only works if transactions are sent in FIFO order according to their commit order.

6.8 Execution Model of Transactions

Generally, in a distributed database system, there is a coordinator and set of participants. The coordinator controls the execution of transactions over the participants, and the participants act as data sources. However, in a peer to peer network, a peer acts both as a coordinator and as a data source. A peer processes both the local transactions, submitted to it directly by the local users, and the remote transactions received from remote peers. In the following, we define the roles of a peer based on the execution of a

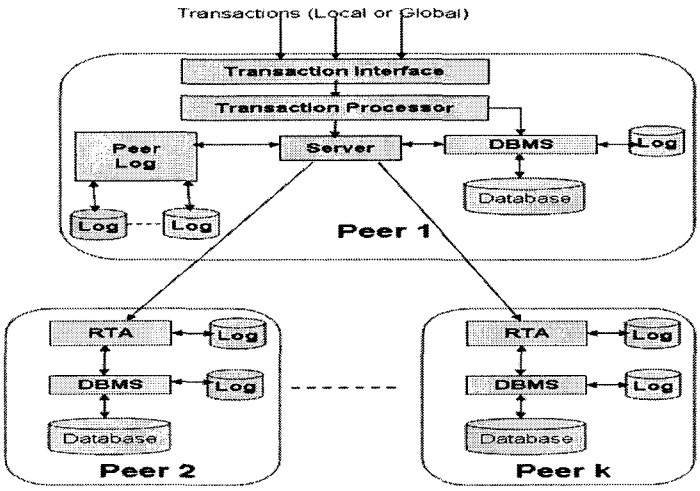


Figure 6.13: Execution model of transactions

transaction.

- *Coordinator (C):* The peer that initiates a global transaction becomes the coordinator of that global transaction if the transaction needs to be executed in other peers.

- *Participant (P):* A peer which receives and processes a global transaction but does not forward the transaction to other peers is called a participant.

- *Coordinator and Participant (CP):* A peer that receives and processes a global transaction from one of its acquaintees and forwards the transaction to other acquaintees, then becomes both a coordinator and a participant. The peer is called a participant with respect to its coordinator, and is called a coordinator with respect to its participants.

In Figure 6.13, we briefly depict an execution model of transactions in a peer data sharing system. Each peer provides a Transaction Interface (TI) which is used by the

users to submit a transaction. When a transaction is submitted through TI, TI forwards the transaction to the Transaction Processor (TP). After receiving a transaction from TI, the Transaction Processor determines the execution scope of the transaction. If the execution scope of a transaction is local, then the transaction is directly submitted to the local database. If the scope is remote, then TP generates remote transactions for the acquaintees of the peer. TP then forwards the remote transactions to Server component. Server component then dispatches remote transactions to the corresponding Remote Transaction Agent (RTA) of each acquaintee. Since we give focus on the transaction execution aspect, we concentrate only on the functionality of RTA and server modules.

Server: A server in a peer P_i maintains a log table for each of its acquaintee which is called *peer log*, and monitors the execution status of a remote transaction that is propagated to an acquaintee. A peer log in P_i for an acquaintee P_j contains the status information of a transaction that is sent to P_j. When a peer P_i propagates a transaction T_i to a peer P_j, then P_i writes a log $< T_i, send >$ in the peer log of P_j. Peer P_i then waits for an acknowledgement for T_i. If P_i receives an acknowledgement for T_i, then P_i writes an entry $< T_i, received >$ in the peer log. If P_i does not receive an acknowledgement from P_j for T_i after a certain period of time, then P_i writes an entry $< T_i, failed >$ in the peer log. In this case, P_i assumes that P_j is offline or crashed. When P_j comes back online, P_i then tries again to send T_i. When T_i is successfully executed by P_j, P_j then sends a response to P_i. When P_i receives a response for T_i, P_i then writes an entry $< T_i, commit >$ in the log.

Remote Transaction Agent: This module is similar to a server module of a multidatabase system. When RTA receives a remote transaction, it sends an acknowledgement to the server to inform about the reception of the transaction. Each RTA is responsible to submit the operations of a remote transaction to its LDBS. A RTA module does not participate to an atomic commitment protocol with a server for executing a remote transaction like in a multidatabase system. Since each remote transaction is an atomic transaction for a specific peer, therefore it does not require to wait for the execution of another remote transaction that is executing in another peer. To maintain the autonomy of LDBSs concurrency mechanism, each RTA is viewed by an LDBS like an application process. Therefore, a remote transaction is processed by an LDBS like a local transaction is processed. However, RTA monitors the execution of operations of a transaction that is sent to the LDBS.

As for recovery purpose of a transaction in a remote peer, we consider the recovery scheme proposed by Brayner et. al. [17] that is used by a server in a multidatabase system. Mainly, we consider the mechanism when failure of a transaction occurs due to aborts of a transaction on behalf of the LDBS [17].

A Remote Transaction Agent of a peer monitors the execution of a transaction in the LDBS by recording logs for each operation of the transaction. A transaction that executes in an LDBS have four different states (active, to-be-committed, committed, and aborted). A transaction is said to be *active*, when a termination operation of the transaction is not submitted to the LDBS by RTA. When RTA submits a commit operation, the transaction enters into the *to-be-committed* state. If the commit operation that is submitted by RTA is successfully executed by the LDBS, the transaction enters into the *committed* state. If the transaction aborts, it enters *aborted* state. Typically, a transaction fails in a peer if

the LDBS decides to abort. A transaction is aborted if the transaction involves in local deadlocks or the transaction faces some internal error conditions during. After such an abort, the effects of the aborted transaction are undone by the LDBSs and locks held by the aborted transaction are released. As soon as RTA recognizes that a particular transaction is aborted by the LDBS, RTA starts execution of its recovery actions. These actions consist of an analysis pass and a redo pass over the log file in RTA. The analysis pass reads the log sequentially in order to identify the status of the aborted transaction, that is active or to-be-committed before the occurrence of the failure. After finding the state, RTA starts redo pass. If the transaction fails in the active state then the transaction is resubmitted. However, if an abort occurs after the transaction is in to-be-committed-state, the write operations are resubmitted to the LDBS. It is important to note that the resubmission of the aborted transaction does not affect other remote transactions in other peers since each remote transaction is an independent atomic transaction. If a remote transaction is successfully executed at a peer, RTA sends a response to the corresponding server of the remote transaction.

From the above execution, it is obvious that a global transaction is executed acquaintance based. There are mainly three advantages of the approach:

1. If a transaction aborts in a peer over an acquaintance path, it does not need to abort the transaction through the path. The peer, where a global transaction is aborted, receives the transaction (reconciliation transaction) again from its immediate coordinator and re-executes the transaction and starts forwarding the transaction.

2. A coordinator does not need to wait for the complete execution of a global transaction over the network. The coordinator only observes the execution over its immediate participants.

3. If a transaction aborts at a coordinator, then the transaction needs to be aborted only over its immediate acquaintances since a transaction commits acquaintance to acquaintance.

6.9 A Typical Transaction Service Request Scenario

We consider a transaction as a transaction service that contains either a single or a group of SQL commands (e.g. Select, Update, etc.). In the following, we describe the different components of the service module which is shown in Figure 6.14:

- **Transaction Service**: A transaction service operates in two modes: local or remote. In the local mode, the service processes request locally without requiring assistance of other peers; in the remote mode, the service needs remote resources (i.e. acquaintance to other peers and services offered by those peers [transaction service]) to finish its job.

- **Transaction Service Manager**: Each peer has a transaction service manager that handles execution of transactions. The transaction service manager takes care of transactions that are received from the local as well as from remote peers. Transaction service manager creates a transaction handler for each of the transactions.

  A local service processes its request using following steps:

  1. Processes the request locally
  2. Returns the result to the calling process if the request is local, or send the result to the original requesting service if the request is remote.

  A remote service processes its request using the following steps:

1. Processes the request locally first, if any.
2. Retrieves or finds remote resources that can handle the request and sends it to the remote resources.
3. Wait until the response from the remote service arrives and processes the response, if any.
4. Returns the result to the calling process if the request is local, or send the result to the original requesting service if the request is remote.

- **Service Handler**: A Transaction Service Manager can coordinate multiple transactions simultaneously by using transaction Service Handlers. When a Transaction Service Manager receives a transaction, it first creates a Service Handler for that transaction and passes over the transaction to the Handler. Afterwards, the Handler uses the LDBS to process the transaction; the service continues to wait for a new transaction. In the following, we describe the processing of a typical transaction service request.

1. When a user on peer A submits a transaction execution request, the Transaction Service Manager creates a service handler to process the request.
2. The service handler processes the request locally and asks the transaction service manager to find the remote resource for execution over peer A's acquaintances.
3. The transaction service manager invokes acquaintance service to get acquaintances of peer A.
4. The manager then consults with Transaction Translation Component to translate the transaction for all the acquaintances and sends the translated transactions to

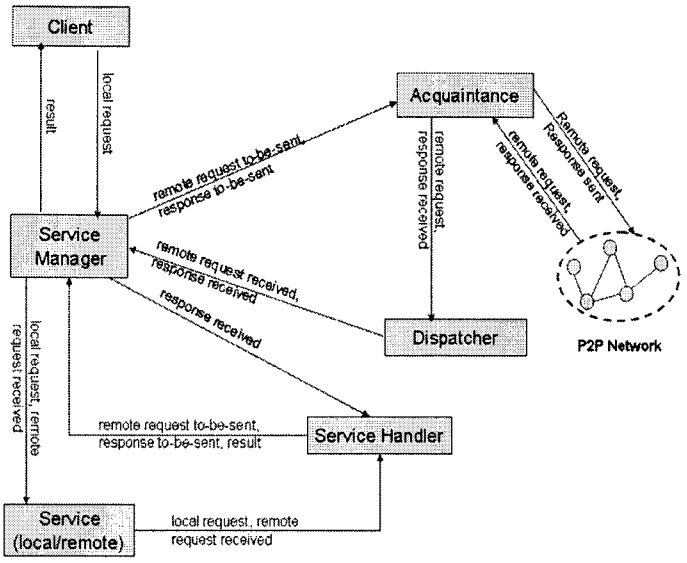


Figure 6.14: Modular structure: Main classes and their functionalities

the corresponding acquaintees.

5. The acquaintee peer B receives the request and forwards to the Transaction Service Manager. The Service Manager of peer B then creates a transaction handler to process the transaction locally.

6. The Service Handler of peer B then sends a response to the Transaction Service Manager. The Transaction Service Manager of peer B then sends the response to peer A.

7. The Transaction Service Manager wakes up the waiting Handler and delivers the response message to it.

8. The Handler finishes processing the transaction and gives the result back to the Transaction Service Manager.

9. Transaction Service Manager notifies the waiting user that the transaction is processed.

6.10 Summary

In this chapter, we discussed execution scenario of transactions in data sharing systems where sources are independent database management systems and share data with each other. We also discussed the problems for maintaining consistent execution view of concurrent transactions and also proposed approaches for maintaining consistent execution. The approaches consider both normal execution of transactions and failures of transactions. Our approaches are scalable because a peer doesn't need any global knowledge of the system, and there is no global coordinator. Transactions are processed by each peer independently and consistency is maintained recursively through acquaintances. At the end of this chapter, we presented a service oriented model for execution of transactions.

Chapter 7

PDST: A Simulation Tool for Peer Database Systems

Currently, there are simulation tools for simulating peer-to-peer systems [79, 71, 44, 68]. However, the tools are used for simulating content distribution systems and file sharing systems. There exists no software tool for simulating and evaluating large networks of peer database systems. In this chapter, we present such a software tool. The tool provides different facilities for generating peers with databases, establishing acquaintances between peers, and creating mappings between peers. The databases in peers are populated with synthetic data. The tool also provides a general framework for processing queries and updates.

We observe that the majority of prototypes of peer database systems do not provide experimental results by considering large networks. Most of the systems are evaluated with few peers and databases; acquaintances and mappings are created manually. Therefore, it is a time consuming approach for evaluating a large system. Hence, we feel that

there is a need to develop a software tool that serves the peer database research community for simulating large peer database systems.

In practice, there has been a clear separation between a simulation model of a P2P system and a real P2P system that operates with real resources (e.g. databases, mappings, queries, etc.). Simulation models are simplified abstractions of real systems that are formalized with the language and modeling concepts that a particular simulation paradigm and environment offers [41]. Particularly, a simulation model in a P2P system is developed in order to validate the scalability of the system. However, sometimes it is necessary to experiment a proposed system with real peers in order to validate the functionalities of the system. The functionalities of a real P2P system is usually provided by P2P applications that are formalized by means of application programming and that include every details of the intended system.

The main objective of this chapter is to present a software tool developed for modeling peer database systems. The tool provides facilities for evaluating peer database systems in a large P2P network. In brief, the tool has the following features.

- combines a database system with P2P functionalities;
- automatically generates databases for each peer and populates databases with synthetic data. The tool also generates mappings (coordination rules, mappings) between peers automatically by analyzing schemas of peers; and
- automatically creates acquaintances among peers based on the data in the peers.

7.1 Features of PDST

PDST, developed using Java, is a simulation toolkit with a library of Java classes. It is developed using Java programming language for the portability and ease of extensibility. Each peer in PDST is a separate Java thread. The tool can be used for query, update, and transaction processing, which are the important services in a peer database system. In PDST, each service is implemented as a thread. The tool provides real-time clock simulation capabilities. The real-time clock simulation, as the name suggests, involves using a real world clock (system clock time obtained using the Java function System.currentTimeMills) for simulating timing [69].

PDST is a message-level event driven simulation framework aimed at modeling peer database systems. It is event based rather than time-driven. Therefore, the simulation time is advanced by the occurrence of events instead of advancing the simulation time in fixed increments that is used in a time-driven simulation system. For example, a query service starts when a peer receives a query from a user or from another peer, and the service finishes when the query is executed in all the peers in the network relevant to the query. Before describing the framework of the tool, we first describe the architecture of a peer that the tool creates for a peer database system. Figure 7.1 shows the architecture. A peer in the system consists of the following components:

P2P User Interface: This interface provides a user with a facility for submitting a service request (e.g. query, update, or transaction) in the system. There are two options for submitting a service request: using either GUI or a text file. The file option allows a user to submit a batch of requests for monitoring the behavior of a system with different loads (e.g. queries arrival rate, different update size, number of concurrent transactions,

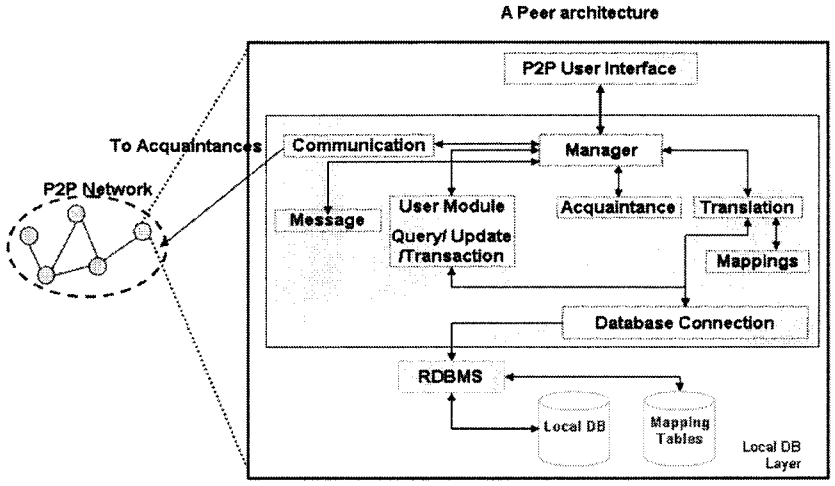


Figure 7.1: Architecture of a peer

etc.). Through the GUI, user can pose one request at a time to verify the functionalities of a system.

User Module: The tool provides a facility to the users for adding modules according to the type of services that the system needs to process. For example, a service may be for processing a query, an update, or a transaction. The tool uses the database connection component for processing a request in the local database. The Manager component handles the services for processing in acquainted peers.

Manager: The Manager handles the execution of services. The Manager takes care of the service requests that are received from the local as well as from remote peers. The Manager interacts with other components that are necessary for processing a service request. For example, if a service needs to be executed locally, it communicates directly with the local database system through the appropriate processing modules (e.g. query, update, or transaction). The service processing modules are externally plugged-in by the users according to the system requirements. If the service needs to be executed remotely,

the Manager interacts with other components, such as, Translation, Acquaintance, Communication, and Message components.

Acquaintance: The Acquaintance component provides the acquaintance information to the Manager of the local peer. The acquaintance information is used by the Manager to translate a service request using mappings in terms of the vocabularies of the acquainted peers. The Manager interacts with the translation component for translating a request and the communication component for forwarding a request to the acquaintees of a peer.

Translation: The Translation component translates a local request into a set of remote requests that need to be executed in the acquainted peers. The translation is performed using the coordination rules or mappings that exist between a local peer and its acquainted peers. The tool considers coordination rules for schema-level mappings, and mapping tables for data-level mappings.

Communication: This module provides the facility to a peer for sending and receiving messages in the network. In PDST, each peer implements a FIFO queue for sending and receiving messages. When a peer wants to send a message to an acquaintee, it enqueues the message to the corresponding queue of the acquaintee. A peer performs dequeue operation for receiving a message from the queue. For receiving a message, a peer listens the queue for the incoming message. If a message is found, the appropriate action is performed. The Communication component uses the Message component to construct a message for a request. Once a message is formed, the Communication component sends the message to the acquaintees. Since the tool implements queue for message exchanges, there is no communication delay in the simulation time. On the other hand, some delays are introduced because of the database access time.

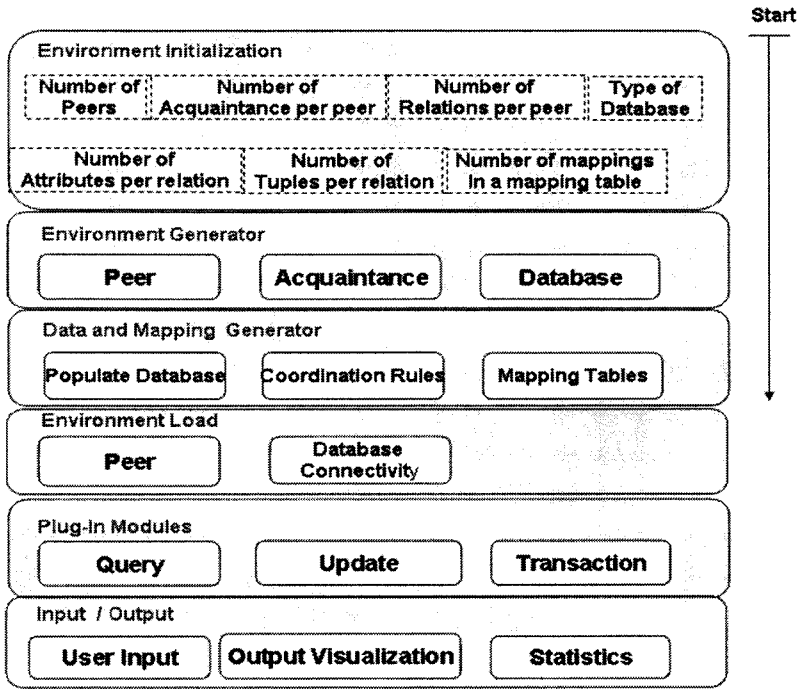


Figure 7.2: PDST simulation framework

Database Connection: This component allows a peer for accessing the local database system for executing a service request. The tool supports different database systems (MySQL, PostgreSQL, Microsoft Access) connectivity. In the environment setup phase, a user needs to specify which database system he/she wants to connect. For this purpose the tool is incorporated with different JDBC driver packages.

Local DB: Each peer has a local database system that is created by the tool with synthetic data. In this tool, each peer uses a relational database system (RDBS).

Parameters	Values
Number of peers	1 to 500
Maximum diameter of a network	10
Number of acquaintances per peer	Min: 2; max: 5
Number of relations per peer	Min: 1; max:3
Number of attributes per relation	Min: 2; max: 4
Number of tuples	Min: 10; max: 100
Domain of the attributes	Integers from 1 to 1000
Number of mapping tables per peer for each acquaintance	Min: 2; max: 8
Number of data mappings in a mapping table	Min: 5; max: 25
Number of coordination rules	Min: 1; max: 3

Table 7.1: Parameters to build a peer database system

7.2 General Framework of PDST

The overall simulation framework of PDST is given in Figure 7.2. This figure shows the phases needed for setting up a complete simulation environment. In the following, we describe different phases a user goes through to build a peer database system using PDST.

7.2.1 Environment Initialization

In this phase, users specify values for different parameters to setup a peer database system. For example, number of peers, maximum number of acquaintances per peer, number of relations per peer, etc. The tool provides an interface to the users for setting a system environment. The parameters that are used to build a system using PDST are given in Table 7.1

7.2.2 Environment Generator

In this phase, users create different resources for a system to be evaluated. The peer module creates peers as defined in the parameter settings. Each peer in the system is implemented as a distinct java thread. The ids of peers are integer numbers starting from 1. After creating peers, the tool creates databases for peers. During the creation of databases, the tool considers parameters that are defined in the environment initialization phase by the users. The database module mainly creates the schemas of databases. In the next phase, databases are populated and the mappings are generated. The acquaintance module creates the acquaintances for each peer. When acquaintances are created, a logical peer database system is created. Information about acquaintances is stored in a text file.

7.2.3 Data and Mapping Generator

The databases are populated in this phase. More importantly, in this phase coordination rules and mapping tables are generated for each acquaintance. The tool considers integer as the domain of the attributes of each relation. Relation names follow the following convention:

$$peerid\_relation(attrbute1, attribute2, \cdots).$$

For example, if a peer "P1" has two relations then the relations are generated as follows:

$$P1\_r1(A1, A2, A3);\ P1\_r2(A4, A5, A6)$$

Consider that peer $P1$ has an acquaintance with peer $P2$. An example of a coordination rule generated from the mapping module is shown below:

$$P1 :: P2 :: P1\_r1(A1, A2, A3) : -P2\_r1(A1, A2, A3)$$

The naming convention of mapping tables is *peerid_m_peerid*. A peer may generate more than one mapping table for a single acquaintee. For example, if a peer $P1$ has two mapping tables for its acquaintee $P2$, then following mapping tables are generated:

$$P1\_m1\_P2;\ P1\_m2\_P2$$

7.2.4 Environment Load

Once the generation of peers, acquaintances, databases, and mappings are completed, this phase loads all the peers in the memory to become active. When a peer is active in the system, the peer makes a connection to its database, loads all the resources in the memory, and waits for events to process.

The simulation process starts when a peer initiates a request and the request needs to be executed in the system. Before forwarding a request to acquaintees, the initiator constructs a message which is composed of (i) peer ID (PID), the sender of the message, (ii) a unique global identifier ($msgID$) of the message. A $msgID$ is formed by combining the identifier of the initiator and a message sequence number. Each peer generates a unique sequence number in increasing order for each message it initiates. If a peer receives a request with the same $msgID$ as seen before, the request is rejected, (iii) the message itself, and (iv) the type of message. There are three types of message: (a) query (Q), (b) response (RS), and (c) result (R). Methods related to creation of a message are defined in the *message* class. Messages are sent to peers using the method *sendMessage* defined in the *message* class. There are also other useful methods in the *message* class.

7.2.5 Plug-in Module

In this phase, users are allowed to include their own modules to extend the tools according to their requirements. For example, processing of queries, updates, and transactions.

7.2.6 Input / Output

PDST provides a GUI to setup a system environment. Figure 7.3 shows the interface for creating a system environment. Through this interface, a user can create peers, databases, acquaintances, and mappings. The interface allows users to see the acquaintance graph of a peer database system. The window in the right side of the main screen in Figure 7.3 shows the generated acquaintances of a peer. In order to run all the peers in the system, the user needs to click on the 'Load Peers' button. On the right side of the main screen, it shows that the peers are active in the system. In order to initiate a request (query, update, or transaction) from a peer, the user selects a peer from the peers' list. When a peer is selected, a window is activated to work with that peer. For example, Figure 7.4 shows an input screen of peer "1" for submitting transactions. The screen also provides the list of peers that received the request after its submission. In this case, the screen shows the received transactions by the peers and the serialization orders of the transactions generated by peers. Note that each peer is a distinct thread of a class "peer.class". In the following, we present some of the important methods of the "peer.class".

getPeerName(): returns the name of the currently selected peer.

getAcquaintances(): returns the names of all the acquainted peers of a peer.

getDBConn(): returns the database connection object of a peer.

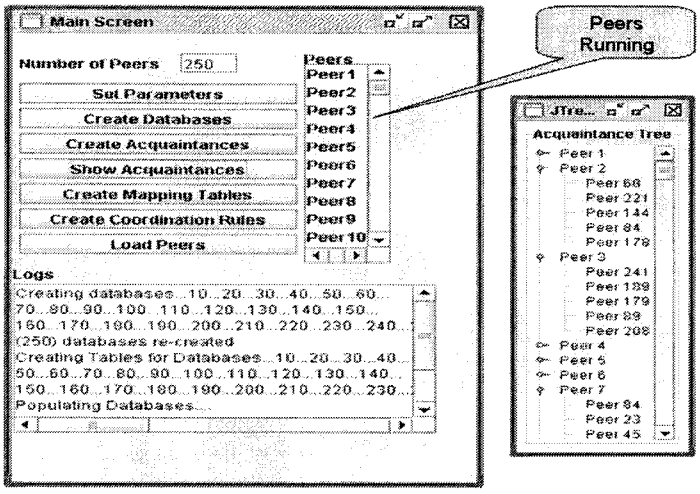


Figure 7.3: Main screen of PDST

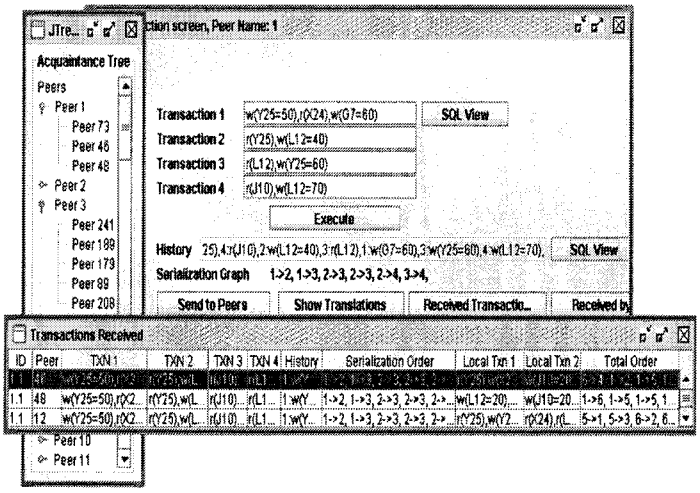


Figure 7.4: A user input screen for peer 1

listenPeerQueue(): continuously listen the queue for any incoming message. This a separate thread running in each peer. If any message is detected, the peer processes the request.

sendMessage(): sends message to the acquaintees.

7.3 Summary

In this chapter, we presented a tool for simulating peer database systems in a large P2P network where mappings between peers are established either by coordination rules or data-level mappings. The tool supports the real database functionalities, for example processing queries, updates, and transactions. We evaluate the transaction processing mechanism [62] using this tool and present some experimental results in Chapter 8.

A future goal is to evaluate the tool considering proper network factors, for example, taking into account network contention and number of exchanged messages or data. Another goal is to investigate the query processing of the approaches presented in [49, 39, 30, 70] and show their comparison results. Moreover, we intend to extend the framework for supporting web interface, so that users can use the tool from any where through the Internet.

Chapter 8

Experiments

In this chapter, we first present the settings of our experiments. Second, we show the evaluation results of the proposed update translation algorithm. Third, we discuss experimental results of the proposed update propagation and conflict resolution strategy. Fourth, we discuss experimental results of the proposed transaction processing mechanisms. Finally, we evaluate our PDST tool.

8.1 Update Translation and Propagation

We implemented our proposed update translation and propagation strategies on top of the freely available JXTA platform [34]. JXTA is a P2P networking platform that provides basic protocols and communication links (called pipes) between peers. Moreover, it provides basic P2P functionalities such as the creation of peers for a P2P network; the creation of messages and message communication onto pipes, the discovery of peers, the creation of peer groups, and the ability for peers to join a peer group. JXTA also provides an IP independent naming space to address peers and other resources.

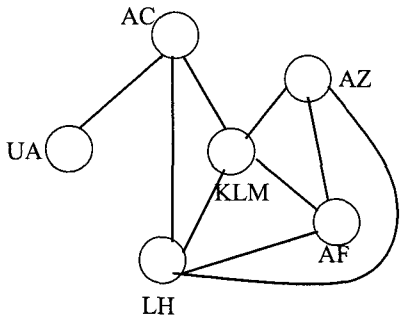


Figure 8.1: Acquaintances between peers

Fno	Date	deptime	Dest
AC856	10/06/2003	18:15	LONDON
AC866	10/06/2003	19:15	LONDON
AC608	10/05/2003	06:45	HALIFAX
AC872	10/06/2003	17:45	FRANKFURT
AC700	10/05/2003	06:50	NEW YORK

(a) Instances of peer AC

fno	date	deptime	dest	ArrTime
UA928	10/05/2003	18:25	LHR	8:15
UA928	10/06/2003	18:25	LHR	8:15
UA940	10/06/2003	20:55	FRA	12:20
UA672	10/06/2003	8:00	LGA	11:05

(b) Instances of peer UA

Figure 8.2: Database instances of peer AC and UA

dest		dest
HALIFAX		YHZ
LONDON		LHR
NEW YORK		EWR
NEW YORK		JFK
NEW YORK		NYC
NEW YORK		LGA

(a) Instances of peer AC

fno		fno
AC856		UA928
AC872		UA940
AC700		UA670
AC700		UA672
AC700		UA674

(b) Mapping table dest2dest

Figure 8.3: Mapping table fno2fno

The experimental environment consists of a single Windows XP machine with Intel Pentium 4 CPU 3.40GHz and 1GB of RAM. The databases are created within the MySQL 5.0 database environment. For the experiments, we built a prototype of a data sharing system that considers a data sharing domain of a flight reservation system. The prototype system has six peers representing six airlines, where the acquaintances between peers are shown in Figure 8.1. The airlines are called Air Canada (AC), Air France (AF), Alitalia (AZ), KLM, United Airlines (UA), and Lufthansa (LH). The data in the system include information for flights of different airlines and passengers. We collected the data from the experiments [49]. After analyzing the data, we noticed that the peers use different data vocabularies to describe flights. Mapping tables were used to map flight numbers between partner airlines for flights to common destinations on the same day. Apart from the distinct flight numbers used by each airline, other differences include the use of city abbreviations. Furthermore, mapping tables were used between the destination attributes of different airlines, e.g. airport codes to city destinations. Finally, non-binary mapping tables were used between flight date and time attributes in different airlines to represent the time difference between the geographical locations of the databases. In total, there were around 50 mapping tables to map flight numbers, destinations, etc, between partner airlines in the system. Each mapping table contains an average of 150 records. Figures 8.2 and 8.3 show part of the database instances and mapping tables between peers AC and UA. We ran all the peers within the same Java Virtual Machine and each peer runs as a distinct thread. Since all the peers are run in a single machine, there is no network delay. Note that the prototype is built using Java programming language.

We performed several experiments for evaluating our proposed update translation and

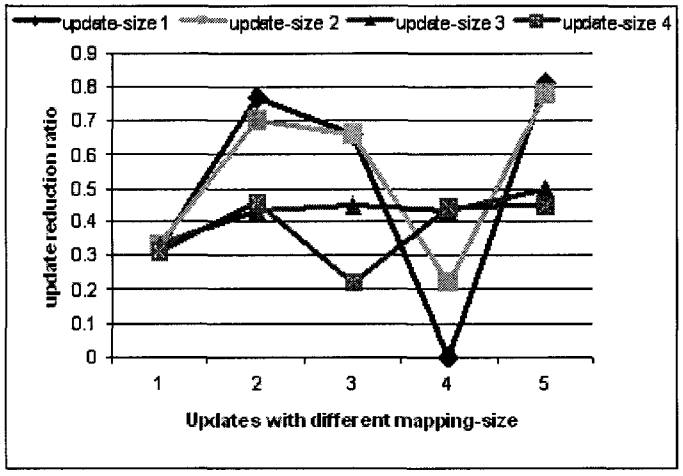


Figure 8.4: Update reduction ratio with different updates

propagation strategies focusing on two main objectives. The first objective is to evaluate the performance of the translation algorithm. We evaluated the algorithm with respect to four problem parameters, namely, the size of mapping tables, the size of the input updates, the number of acquaintees of a peer, and arrival rate of updates. The second objective is to examine the update propagation and execution strategy. We examined the strategy considering two parameters, namely, the number of peers and the number of conflicting updates. We implemented the *value at neighbor* conflict resolution mechanism that each peer applies to resolve update conflicts. We performed all the experiments at least five times and considered the average of the results.

The first experiment shows in what ratio the translation algorithm reduces the number of translated updates that are irrelevant when we consider the mapping table m_k for translation that contains associations of key values. We considered the configuration of Figure 8.1 to perform this experiment. In order to justify the consideration of m_k, we used the metric called *update reduction ratio* which is formulated as *(1-(number of updates*

generated by considering m_k /total number of updates generated without consideration of m_k). The update reduction ratio basically shows the number of irrelevant updates that are eliminated for translation. It also shows the justification of considering the mapping table that stores primary key associations during translation of updates. For this experiment, we used four updates with different update size. We also considered different mapping-size of the attributes mentioned in the updates. The number of attributes in a given update is called the *update-size* and the number of tuples in a mapping table for a given attribute in an update is called the *mapping-size.* We obtained the mapping-size of an attribute in an update by analyzing the corresponding mapping table (e.g. using the SQL *count* and *group by* commands). The updates are selected in such a way that they contain *non-key* and *key* attributes with different mapping-size. The updates are originated from a single peer. The experimental result of this experiment is shown in Figure 8.4. The first observation from the results of the experiment is that, if the attributes are non-key and the mapping-size related to the attributes in an update is large, then *update reduction ratio* is high. This is because m_k is considered for translating the updates and the consideration of m_k filters more irrelevant updates. Hence, it reduces network traffic during the propagation of less number of updates. This also reduces the execution time of updates in the system. For an instance, consider an update with update-size one in Figure 8.4. We see from the results of the experiment that when the attribute in the update is non-key and the mapping-size for the attribute is large, we obtain a value of high *reduction ratio* which is approximately 0.8. Our second observation is that when an update contains key attributes the translation algorithm does not reduce the number of translated updates. This is because, the update already includes m_k. For instance, consider again the update with update-size one in Figure 8.4, but now the

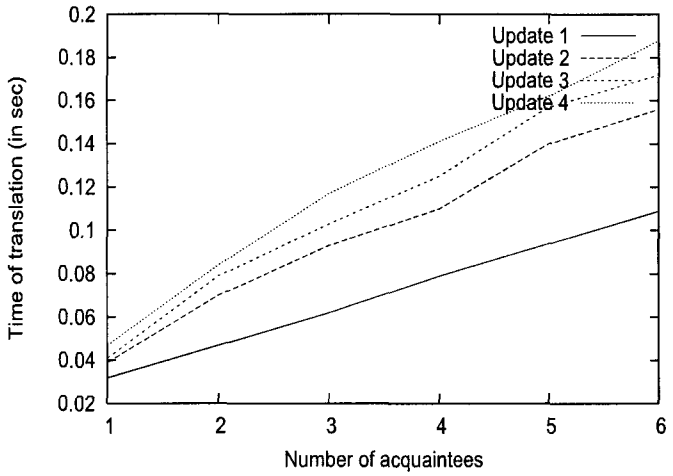


Figure 8.5: Translation time with respect to number of acquaintees

attribute in the update is a key attribute. In this case, we obtain 0.0 value for *update reduction ratio.*

With the second experiment, we examined the efficiency of the proposed update translation algorithm and observed the behavior of the algorithm in translating different types of updates. We considered the translation time as measuring criteria for evaluating the efficiency of the algorithm. For the evaluation, a single peer generates four different updates and the algorithm translates the updates for the peer's acquaintees. The number of acquaintees increased from 1 to 6. We used the following updates which are randomly selected:

Update 1: $del(fno = UA1012)$

Update 2: $mod(\{fno = UA928\} \rightarrow \{tocode = FRA\})$

Update 3: $del(fno = UA5804, date = 10/05/2003, tocode = GRB)$

Update 4: $ins(UA658, 10/05/2003, 18:00, BOS)$

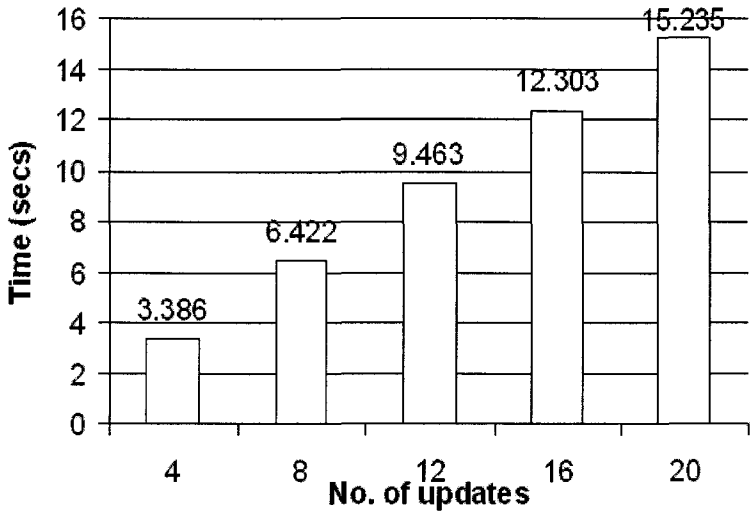


Figure 8.6: Update execution time with different update frequencies

Figure 8.5 shows the results of the experiment. As expected, the translation time increases gradually for each type of update with the increasing number of acquaintees. However, the change of translation time is not rapid, which proofs the efficiency of translation of the algorithm. We also notice from the evaluation that the algorithm takes more time to translate an *insert* update compared to translate other types of updates. The reason is that, it requires more mapping tables for translating an *insert* update.

In the third experiment, we examined the scalability of the proposed update propagation and execution strategy. We mainly count the execution time of updates with different update generation rates as the measuring criteria. For this experiment we considered a network of 20 peers. Since we don't have real data for 20 peers, we considered synthetic data for the database in each peer. We used the PDST tool for creating databases with synthetic data. Each database contains a few hundred different relational tuples and each

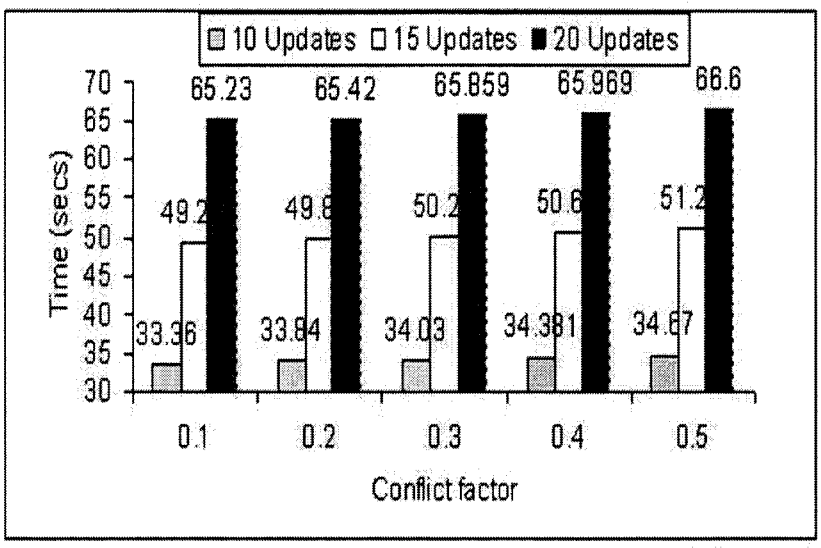


Figure 8.7: Update execution time: different conflict factors and number of concurrent updates

acquaintance is associated with one mapping table that contains 50 tuples in average. The domain of the attributes in each relation is integer. Note that acquaintances are created based on the related data items stored in each peer. For measuring the execution time of updates, a single peer generates updates with different rates. Figure 8.6 shows the execution time with different update generation rates. The annotated values in the figure denote the complete execution time of all the updates in all peers in the network. This time includes the propagation time and the execution time of updates. We observe from the results that the execution time increases linearly with the increasing value of update generation rates in the system. Given that the typical median diameter of a network is about six hops, we anticipate from the results of the experiment that the approach is scalable to hundreds or thousands of peers.

We also evaluated the update propagation and execution strategy considering different

conflict factors of updates. A *conflict factor* denotes the ratio of the number of updates that are involved into conflict and the total number of updates that are active in the system. For example, 0.2 conflict factor means that 20% of the total updates that are active in the system are involved into conflict. Note that updates are generated from different peers. For this evaluation, we considered the same setting as described in the last experiment. The objective of this evaluation is to examine what is the effect of the conflict factors to the proposed update propagation and conflict resolution mechanism. We performed two experiments with two different settings to examine the effect. In the first experiment, we fixed the number of peers to twenty but varied the number of updates generated from peers. Figure 8.7 shows the results of the experiment. Note that we selected the updates in such a way that updates involve into conflict according to the specified conflict factors (0.1, 0.2, 0.3, 0.4, and 0.5). From the result, we observe that the execution time increases with increased value of conflict factors, however, the effect on execution time is not a major inhibiting factor. Consider the execution time of 20 updates with conflict factors 0.4 and 0.5. We see that the execution time are 65.969 and 66.60, respectively. Notice that the execution time increases slowly with the increased conflict factors. We also observe that the execution time increases rapidly with increased number of concurrent updates. For the second experiment, we fixed the number of concurrent updates generated from peers to ten but varied the number of peers and conflict factor in the system. The objective of this experiment is to examine the behavior of the update propagation and execution strategy taking into account the different number of peers with different conflict factors. Figure 8.8 shows the results of the experiment. We observe from the experiment that major time changes occur as a function of number of peers in the system.

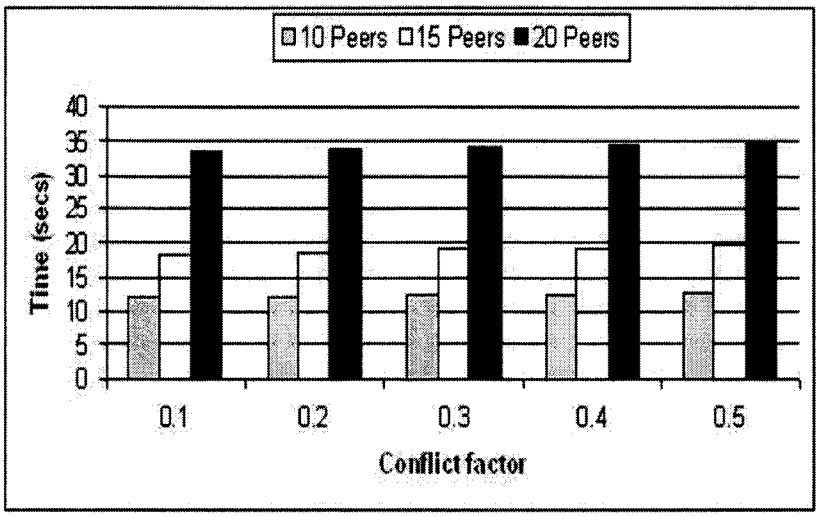


Figure 8.8: Update execution time with different conflict factors

8.2 Transaction Processing

In this section, we show different experimental evaluations of the proposed transaction processing mechanism in data sharing systems. The evaluation is based on the first approach presented in Section 6.5.1. In order to evaluate the performance of the approach over relatively large settings, we used the PDST tool for creating data sharing systems. In the systems, all peers are run within the same Java Virtual Machine. Each peer is implemented as a distinct thread and implements a FIFO queue, which is used by peers for sending and receiving transaction messages.

Our experimental environment consists of a single Windows XP machine with Intel Pentium 4 CPU 3.40GHz and 1GB of RAM. We evaluate the strategy in different size of networks. Namely, we considered networks with 100, 250, and 500 peers. Each peer is connected to a database that is instantiated as a MySQL 5.0 database. For each peer

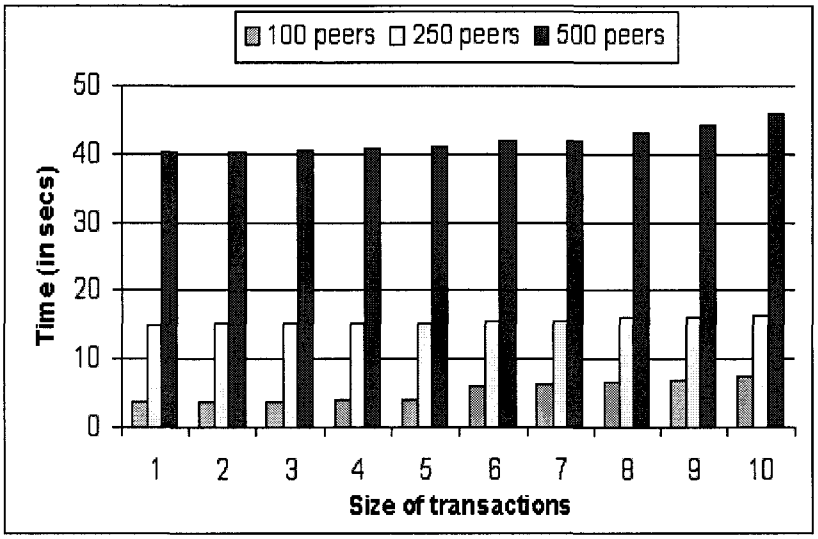


Figure 8.9: Execution time of transactions with different size of transactions

in a network, we generated schemas and contents of the peers' databases, as well as the peers' acquaintances by the tool. All the peers have an equal average number of acquaintances, which we selected randomly. Based on the data in the databases of the peers in an acquaintance, mapping tables are generated using the tool. Table 8.1 provides a summary of the configuration parameters that we use to create a network environment by the tool. The operations of a transaction are MySQL select (read operation) and update (write operation) commands. Since all the peers ran on the same machine, there is no network delay. However, there are some delays because for accessing databases. In the tool, we included a module that simulates the strict two-phase locking protocol in each peer. The strict two-phase locking protocol is chosen since it is used in most of the commercial databases and is easy to implement. Therefore, before executing transactions in the database of a peer, the module determines the serialization order and schedules of transactions. After that, operations are submitted into the database of the peer according

Parameter	Value
Number of peers	100, 250, 500
Maximum diameter of the network	10
Number of acquaintances per peer	Min: 2; max: 5
Number of relations per peer	1
Number of attributes per relation	Min: 2; max: 4
Number of tuples	Min: 10; max: 25
Domain of the key attribute	String, length 4
Domain of non key attributes	Integers from 1 to 26
Number of mapping tables per peer for each acquaintance	Min: 2; max: 4
Number of data mappings in a mapping table	Min: 5; max: 25
Size of a transaction	Min: 1; max: 10
Number of concurrent transaction generated in a peer	Min: 1; max: 4

Table 8.1: Configuration parameters

to the order determined by the module. We performed several experiments for evaluating our proposed transaction processing strategy. In the following, we show the objectives and the experimental results.

The first goal of this experiment is to show the execution time of different sized global transactions in different size of networks. The number of operations in a transaction is called the size of the transaction. In our experiments, we limit the transaction size between 1 and 10. In each transaction, the probability of the read and write operations are same. It makes the number of read and write operations balanced in a transaction.

Figure 8.9 shows the results of this experiment. From the result, we observe that the execution time of the transactions increases gradually and linearly, with the size of transactions. For instance, consider the network of 500 peers, the execution time of

the transactions with size 2 and 10 are 40.375 second and 45.862 second, respectively. The result shows that the execution time does not increase rapidly with the size of transactions. We also observe from the result that the execution time grows rapidly with the increased size of networks. For instance, consider the network of 100 and 500 peers, a transaction with size 5 takes 3.929 second and 41.468 second, respectively to complete its execution. This result shows that the execution time of transactions increases rapidly with the increased size of networks. Note that the execution time of transactions scale badly as compared with the execution time of updates, shown in Figure 8.6. The reason is that a transaction has multiple operations. Therefore, it takes time to translate a transaction as compared to the time of translating a single update. Besides the translation time, additional time is required for the execution of concurrent transactions. This includes involvement of local concurrency control protocol and the proposed mechanism of ensuring acquaintance-level serializability.

The goal of our second experiment is to show the execution time of concurrent global transactions in a data sharing system considering the involvement of the local transactions in peers. We verified the correctness of transactions' execution by examining the serialization order in each peer where the global transactions execute. A snapshot of the global transactions, local transactions, and the corresponding serialization orders of transactions in peers are shown in Figure 7.4. We assumed four global transactions concurrently execute in networks and a peer has maximum two local transactions with transaction size four. The result is shown in Figure 8.10. We notice from the result that the local transactions do not really decrease the performance of the global transactions' execution in the system. For instance, consider a network of 500 peers. The execution time of the global transactions in the system is 38.109 second when no peer in the network

Parameter	Value
Number of peers	100-500
Number of acquaintances per peer	3
Number of relations per peer	1
Number of attributes	4
Number of tuples	25
Number of mapping tables per peer for each acquaintance	4
Number of data mappings in a mapping table	8

Table 8.2: System parameters

executes local transactions during the execution of the global transactions. Meanwhile, when 50% of the peers, i.e. 250 peers execute local transactions at the time they execute global transactions, the execution time is 46.812 second. This shows that there is no significant effect on execution time for global transactions, which is to be expected, when local transactions are considered. However, we notice from that result that the time grows rapidly when the network size changes as it is obvious from the Figure 8.10.

Finally, the goal of our third experiment is to show the execution time of transactions generated from a peer with different arrival rates of transactions. In this case, we only consider 100 peers due to the factor that the number of concurrent connections allowed by the MySQL server is limited. We notice that the execution time increases gradually when the arrival rate increases, which was to be expected; but this increase rate is linear. The transactions arrival rate were 2/sec, 4/sec, 6/sec, 8/sec, and 10/sec. The result is shown in Figure 8.11.

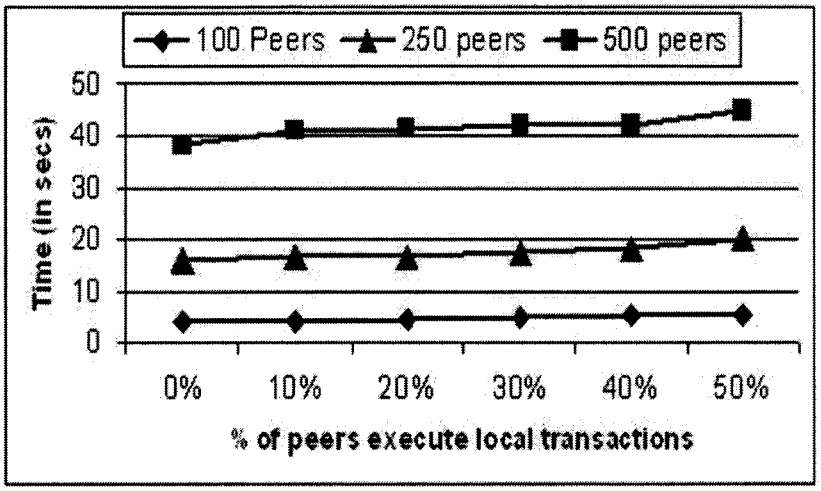


Figure 8.10: Execution time of concurrently executed global transactions with local transactions

8.3 PDST Tool

The first objective of the evaluations of the PDST is to show the time required to build a peer database system with different size of networks. The time includes the creation of peers, databases, acquaintances, and mappings. The second objective is to evaluate the tool by modeling an existing approach of query, update, or transaction processing. Using the tool, we modeled the transaction processing system proposed in [62]. We used the tool in a single Windows XP machine with Intel Pentium 4 CPU 3.40GHz and 1GB of RAM. Since the tool is used in a single machine, there is no communication delay in the simulation time. On the other hand, some delays are introduced because of database access time. In our setting, we considered that each peer is connected to a MySQL 5.0 database, and all the peers have an equal average number of acquaintances. We provide a summary of the configuration parameters in Table 8.2. The results of creating different

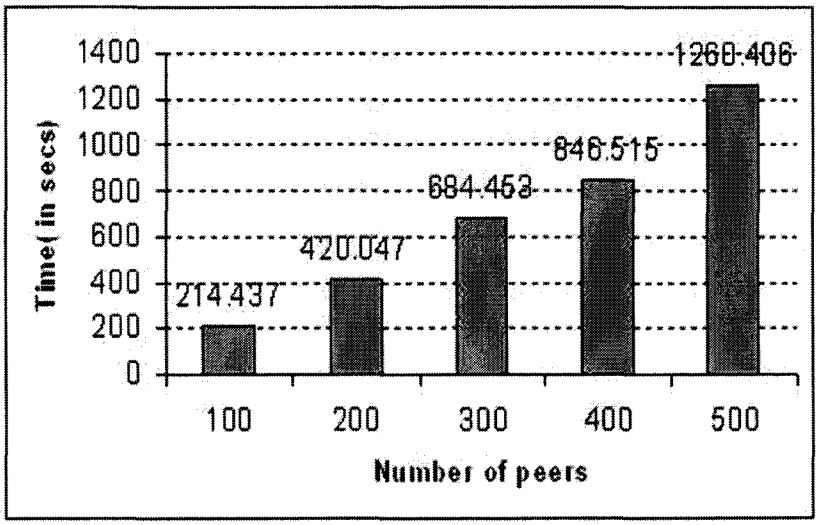


Figure 8.11: Execution time of transactions with different arrival rates of transactions (in a 100 peers network)

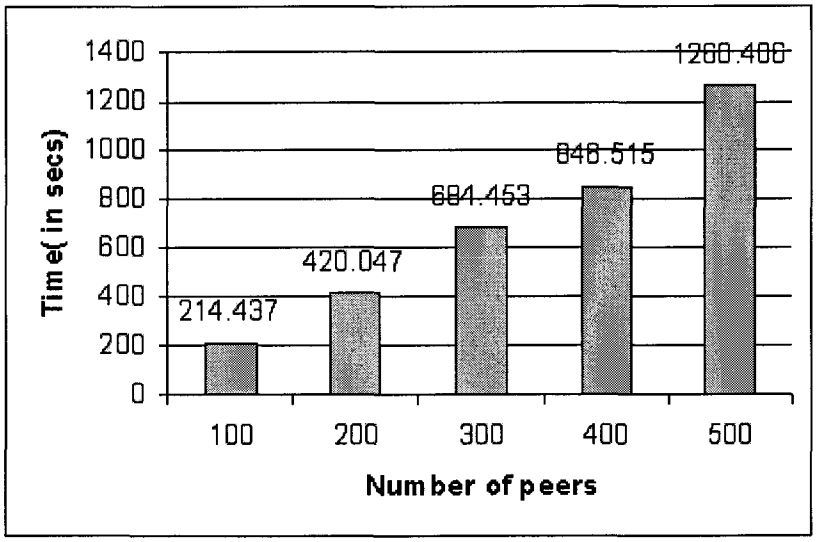


Figure 8.12: Time to generate peers with resources

peer database systems are shown in Figure 8.12. We observe from the result that the time increases sharply when the number of peers increases. Through the analysis of the evaluation, we notice that 90% of the time is required to create databases and generate mappings.

Note that the evaluation results for processing transactions using PDST are described in Section 8.2.

8.4 Summary

This chapter first presented the settings of our experiments. Next, we showed evaluation results of the proposed update and transaction processing algorithms. We also evaluated the PDST tool. In the future, we want to conduct all the experiments in a large peer data sharing system, for example on PlanetLab [74], to show their scalability. Moreover, we want to evaluate the PDST tool considering proper network factors, for example, taking into account network contention and the number of exchanged messages or data. Another goal is to use our simulator to compare the query processing approaches presented in [49, 39, 30, 70].

Chapter 9

Conclusions and Future Work

9.1 What Has Been Achieved

In this thesis, we investigated mechanisms for supporting update exchange and transaction processing in peer data sharing systems. Specifically, we presented a data update problem that is distinct from the well-studied problems of view maintenance and view update problems. The view maintenance and view update problems are generally applied in data integration and data exchange systems. In this thesis, we investigated data update problems in the context of peer-to-peer data management systems for maintaining data consistency between related peers.

In Chapter 1 of this thesis, we highlighted the difference between the update propagation problem in data exchange in one hand, and the same problem in data sharing systems, on the other. We further described the settings of a data sharing system in Chapter 2. The setting is mainly based on the Hyperion peer data sharing system. Hyperion uses mapping tables to relate data located in different peers and provides a framework of simple schema mappings and data mappings between peers. Hyperion also

provides core data sharing features such as schema and data mappings and query processing, which we leveraged to carry out our work. Chapter 3 presented the research issues for processing updates and transactions in peer data sharing systems. Chapter 3 also discussed literature related to this thesis.

Chapter 4 investigated one of the main objectives of this thesis, namely, update processing. We first discussed the semantics of update execution in peer data sharing systems. Second, we discussed the situations when a local update originated at a peer needs to be propagated to other related peers. This propagation of updates brought the issues of finding relevancy and irrelevancy of updates. Hence, we discussed syntactic and semantic mechanisms for checking update relevancy. Since relevant updates are required to be propagated in the system, it requires proper update translation mechanisms between peers. For the correct translation of updates, we introduced a correctness criteria that ensures the correct translation. We also proposed an algorithm for translating updates between peers that guarantees the correctness. Chapter 4 also analyzed the complexities of checking relevancy of updates and the proposed update translation algorithm.

Chapter 5 investigated the mechanism of update propagation and conflict resolution. In this chapter, we first discussed some issues that arise during propagation of updates due to the autonomous and dynamic nature of peer data sharing systems. Considering the dynamic behavior of peers, we presented an update propagation algorithm that is similar to an optimistic update propagation algorithm. We also presented a conflict resolution mechanism of updates where each peer independently resolves a conflict and that requires no global coordination. We also introduced semantic conflict resolution rules for resolving semantic conflicts between updates.

In Chapter 6, we studied mechanisms for transaction processing in peer data sharing

systems. With respect to processing of transactions, we focused on concurrency issues. Maintaining a consistent execution view of concurrent transactions in peers is challenging in a data sharing system since there is no global transaction manager that controls the execution of transactions. In chapter 6, we first discussed issues that cause problems for maintaining a consistent execution view. Later, we introduced a correctness criteria that ensures the correct execution of transactions. We also proposed two approaches for maintaining a consistent execution of concurrent transactions while ensuring the correctness criteria. We further discussed issues related to consistent execution while considering failures of transactions. We also showed how the proposed approaches ensure correctness, even though transactions may fail during their execution.

9.2 Future Work

With respect to update semantics and translation, we consider the mapping table semantics where the mapping tables are complete. This is a very strong assumption. However, we need to deal with situations of update translation where we only have incomplete information. Particularly, we like to study translation issues when an update creates or requires uncertain information. For example, how should the translation look like when there are no values for all the attributes? We also need to consider the processing of updates when there is a cycle in the acquaintance graph. For this, we need to extend the update semantics.

With respect to propagation of updates, our future goal is to extend the proposed algorithm so that it can handle the network partitions and the dynamic behavior of peers. Furthermore, we need to consider the situation where update propagation involves cycles. Notice that we proposed a simple strategy for dealing with cycles that only considers a

syntactic approach by considering the ids of updates. However, we need to extend solution with a semantic approach. An example is given below.

During the propagation of an update, there can be multiple paths from a peer that originates an update to a different peer with relevant data. Further, the propagated updates can in fact be different based on which path is chosen. For example, consider a P2P system with three peers P_1, P_2, and P_3 with all three being inter-connected. Consider an update u_1 originating at P_1, the path $P_1 \rightarrow P_2 \rightarrow P_3$, may produce updates u_1^2 and u_2^3, but $P_1 \rightarrow P_3 \rightarrow P_2$ may produce altogether different updates u_1^3 and u_3^2. According to the current propagation algorithm it will not pick both the above paths since it will not do $P_2 \rightarrow P_3$ and then $P_3 \rightarrow P_2$ too. By arbitrarily choosing one path over the other (to construct the dependency tree), the algorithm may lead to unpredictable consequences.

With respect to processing of transactions, our future goal is to propose an approach for maintaining a consistent execution view of transactions when transactions are initiated from multiple peers and are executed concurrently in the system. Finally, we want to investigate the proposed approaches in a large real model network to show the scalability of the system.

In addition to the above mentioned future work, we feel that there are some interesting research issues that are worth studying in data sharing systems with regards to processing of queries. Here, we list some of the algorithmic problems. These problems are similar to those raised in data exchange [29] and peer data exchange [31] settings.

Given a finite source instance I and a target instance J, find a target instance J' such that $(I, J') \models \Sigma_{st}^s$, $(I, J') \models \Sigma_{st}^v$, and $J' \models \Sigma_t$. Such a J' is called a solution for (I, J). The concept of a solution is introduced in the data exchange literatures [31, 29]. Now the following questions arise.

Q1. Does a solution J' exist for $(I,\ J)$?

Q2. What is the complexity to check whether a solution exists or not, and to find the solution?

Q3. Given I, J, find the certain answers of a query q posed over T wrt to $(I,\ J)$. The intended semantics of certain answers is the one described in [6].

Q4. Under what conditions and for which queries can the certain answers be computed using source and target instances?

Q5. What is the computational complexity of computing the certain answers of target queries? (Data complexity)

Bibliography

[1] K. Aberer, P. Cudre-Mauroux, A. Datta, Z. Despotovic, M. Punceva, and R. Schmidt. P-Grid: A Self-Organizing Structured P2P System. In *ACM SIGMOD Record*, 32(3):29-33, 2003.

[2] S. Abiteboul, R. Hull, and V. Vianu. Foundations of Databases. Addison-Wesley, pages 580-581, 1995.

[3] R. Alonso, H. Garcia-Molina, and K. Salem. Concurrency Control and Recovery for Global Procedures in Federated Database Systems. In *IEEE Data Engineering Bulletin*, 10(3):5-11, 1987.

[4] S. Androutsellis-Theotokis, D. Spinellis, and V. Karakoidas. Performing Peer-to-Peer e-Business Transactions: A Requirements Analysis and Preliminary Design Proposal. In *Proc. of IADIS International Conference in e-Commerce*, pages 399-404, 2004.

[5] M. Arenas, V. Kantere, A. Kementsietsidis, I. Kiringa, R.J. Miller, and J. Mylopoulos. The Hyperion Project: From Data Integration to Data Coordination. In *ACM SIGMOD Record*, 32(3):53-58, 2003.

[6] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent Query Ansers in Inconsistent Databases. In *Proc. of the ACM symposium on Principles of Distributed Computing (PODC)*, pages 68-79, 1999

[7] F. Bancilhon and N. Spyratos. Update Semantics of Relational Views. In *ACM Transactions on Database Systems*, 6(4):557-575, 1981.

[8] M. A. Bassiouni. Single-Site and Distributed Optimistic Protocols for Concurrency. In *IEEE Transaction Software Engineering*, 14(8):1071-1080, 1988.

[9] C. Beeri and M. Y. Vardi. A Proof Procedure for Data Dependencies. In *Journal of the ACM*, 31(4):718-741, 1984.

[10] R. G. Bello, K. Dias, A. Downing, J. James J. Feenan, J. L. Finnerty, W. D. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Materialized Views in Oracle. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 659-664, 1998.

[11] P. Bernstein, F. Giunchiglia, A. Kementsietsidis, and J. Mylopulos. Data Management for Peer-to-Peer Computing: A Vision. In *Proc. of the Int'l Workshop on the WEB and Databases (WebDB)*, 2002.

[12] P. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency Control and Recovery in Database Systems. *Addison Wesley, Reading, MA*, 1987.

[13] L. E. Bertossi and L. Bravo. The Semantics of Consistency and Trust in Peer Data Exchange Systems. In *Proc. of the Int'l Conf. on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, Springer LNCS 4790, ages 107-122, 2007.

[14] L. E. Bertossi and L. Bravo. Information Sharing Agents in a Peer Data Exchange System. In *Proc. of the Int'l Conf. on Data Management in Grid and P2P Systems (Globe)*, Springer LNCS 5187, pages 70-81, 2008.

[15] J. A. Blakeley, N. Coburn, and A. Larson. Updating derived relations: Detecting Irrelevant and Autonomously Computable Updates. In *ACM Transactions on Database Systems (TODS)*, 14(3):369-400, 1989.

[16] J. A. Blakeley, P. A. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In *ACM SIGMOD Record*, 15(2):61-71, 1986.

[17] A. Brayner and T. Harder. Recovery in Multidatabase Systems. In *Proc. of the Brazilian Database Symposium (SBBD)*, 1999.

[18] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of Multidatabase Transaction Management. In *The International Journal on Very Large Data Bases*, 1(2):181-240, 1992.

[19] Y. Breitbart, A. Silberschatz, and G. R. Thompson. Transaction Management Issues in a Failure-Prone Multidatabase System Environment. In *The International Journal on Very Large Data Bases (VLDB Journal*, 1(1):1-40, 1992.

[20] Y. Breitbart and A. Silberschatz. Multidatabase Update Issues. In *Proc. of the Int'l Conf. on the Management of Data (ACMSIGMOD)*, pages 135-142, 1988.

[21] S. Ceri, M.A.W. Houtsma, A.M. Keller, and P. Samarat. Independent Updates and Incremental Agreement in Replicated Databases. In *Journal of Parallel Distributed databases*, 3(3):225-246, 1995.

[22] U. Cetintemel, P.J. Keleher, B. Bhattacharjee, and M.J. Franklin. Deno: A Decentralized, Peer-to-Peer Object-Replication System for Weakly Connected Environments. In *IEEE Transactions on Computers*, 52(7):943-959, 2003.

[23] S. S. Cosmadakis and C. H. Papadimitriou. Updates of Relational Views. In *Journal of the ACM (JACM)*, 31(4):742-760, 1984.

[24] P. Cudre-Mauroux, K. Aberer, and A. Feher. Probabilistic Message Passing in Peer Data Management Systems. In *Proc. of the Int'l Conf. on Data Engineering (ICDE)*, page 41, 2006.

[25] A. Datta, M. Hauswirth, and K. Aberer. Updates in Highly Unreliable, Replicated Peer-to-Peer Systems. In *Proc. of Int'l Conf. on Distributed Computing Systems (ICDCS)*, page 76, 2003.

[26] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in Partitioned Networks: A Survey. In *ACM Computing Surveys*, 17(3):341-370, 1985.

[27] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive Push-Pull: Disseminating Dynamic Web Data. In *IEEE Transaction Software Engineering*, 51(6):652-668, 2002.

[28] W. Du and A. Elmagarmid. Quasi Serializability: A Correctness Criterion for Global Concurrency Control in InterBase. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 347-355, 1989.

[29] R. Fagin, P. Kolaitis, R. J. Miller, and L.Popa. Data Exchange: Semantics and Query Answering. In *Theoretical Computer Science*, 336(1):89-124, 2005.

[30] E. Franconi, G. Kuper, A. Lopatenko, and I. Zaihrayeu. A Distributed Algorithm for Robust Data Sharing and Updates in P2P Database Networks. In *Proc. of the P2P&DB International Workshop*, pages 382-393, 2004.

[31] A. Fuxman, P. G. Kolaitis, R. J. Miller, and W. C. Tan. Peer Data Exchange. In *ACM Trans. Database System*, 31(4):1454-1498, 2005.

[32] H. Garcia-Molina and K. Salem. Sagas. In *Proc. of the Int'l Conf. on the Management of Data (ACMSIGMOD)*, pages 249-259, 1987.

[33] D. Georgakopoulos, M. Rusinkiewicz, and A. Sheth. Using Tickets to Enforce the Serializability of Multidatabase Transactions. In *IEEE Transactions on Knowledge and Data Engineering* , 6(1):166-180, 1994.

[34] J. Gradecki. Mastering JXTA: Building Java Peer-to-Peer Applications. Wiley InterScience Publisher, 2002.

[35] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update Exchange with Mappings and Provenance. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 675-686, 2007.

[36] A. Gupta, D. Katiyar, and I. S. Mumick. Counting Solutions to the View Maintenance Problem. In *Proc. of the Int'l Workshop on Deductive Databases*, pages 185-194, 1992.

[37] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. In *IEEE Data Engineering Bulletin*, 18(2):3-18, 1995.

[38] A. Y. Halevy, Z. G. Ives, D. Suciu, and I. Tatarinov. Schema Mediation in Peer Data Management System. In *Proc. of the Int'l Conf. on Data Engineering*, pages 505-516, 2003.

[39] A. Y. Halevy, Z. G. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov. The Piazza Peer-Data Management System. In *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16(7):787-798, 2004.

[40] K. Haller, H. Schuldt, and C. Türker. Decentralized Coordination of Transactional Processes in Peer-to-Peer Environments. In *Proc. of Int'l Conf. on Information and Knowledge Management (CIKM)*, pages 28-35, 2005.

[41] D. Hildebrandt, L. Bischofs, and W. Hasselbring. RealPeer-A Framework for Simulation-Based Development of Peer-to-Peer Systems. In *Proc. of Euromicro Int'l Conf. on Parallel, Distributed and Network-Based Processing (PDP)*, pages 490-497, 2007.

[42] S. Hwang, J. Srivastava, and J. Li. Transaction Recovery in Federated Autonomous Database Systems. In *Distributed and Parallel Databases*, 2(2):151-182, 1994

[43] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A Decentralized Peer-to-Peer Web Cache. In *Proc. of ACM Symposium on Principles of Distributed Computing (PODC)*, 2002.

[44] S. Joseph and T. Hoshiai. Decentralized Meta-Data Strategies: Effective Peer-to-Peer Search. In *IEICE Transaction on Communication*, E86-B(6), 2003.

[45] V. Kantere, I. Kiringa, and J. Mylopoulos. Supporting Distributed Event-Condition-Action Rules in a Multidatabase Environment. In *International Journal Cooperative Information System*, 16(3/4):467-506, 2007.

[46] V. Kantere, I. Kiringa, and J. Mylopoulos. Coordinating Peer Databases using ECA rules. In *Int'l Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P)*, pages 108-122, 2003.

[47] A. M. Keller. Comment on Bancilhon and Spyratos' Update Semantics and Relational Views. In *ACM Transactions on Database Systems (TODS)*, 12(3):521-523, 1987.

[48] A. Kementsietsidis, M. Arenas, and R.J. Miller. Mapping Data in Peer-to-Peer Systems: Semantics and Algorithmic Issues. In *Proc. of the Int'l Conf. on the Management of Data (ACMSIGMOD)*, pages 325-336, 2003.

[49] A. Kementsietsidis and M. Arenas. Data Sharing Through Query Translation in Autonomous Sources. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 468-479, 2004.

[50] A. Kementsietsidis. Data Sharing and Querying for Peer-to-Peer Data Management Systems. In *Proc. of the EDBT International Workshop on Information Retrieval and Interoperability*, pages 177-186, 2004.

[51] AM. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. IceCube Approach to The Reconciliation of Divergent Replicas. In *Proc. of the ACM symposium on Principles of Distributed Computing (PODC)*, pages 210-218, 2001.

[52] J. J. Kister and M. Satyanarayanan. Disconnected Operation in The Coda File System. In *ACM Transaction on Computer Systems*, 10(1):3-25, 1992.

[53] C. Koch. Data Integration Against Multiple Evolving Autonomous Schemata. *http://www.csd.uoc.gr/ hy562/Papers/thesis_final.pdf*, 2001.

[54] N. Kotilainen, M. Vapa, T. Keltanen, A. Auvinen, and J. Vuori. P2PRealm - Peer-to-Peer Network Simulator In *Proc. of Int'l Workshop on Computer-Aided Modeling, Analysis and Design of Communication Links and Networks (CAMAD)*, pages. 9399, 2006.

[55] J. Lan, X. Liu, P. Shenoy, and K. Ramaritham. Consistency Maintenance in Peer-to-Peer File Sharing Networks. In *Proc. of the IEEE Workshop on Internet Applications*, page 90, 2002.

[56] M. Lenzerini. Data Integration: A Theoretical Prespective. In *Proc. of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 233-246, 2002.

[57] W. Litwin. From Database Systems to Multidatabase Systems: Why and How. British National Conference on Databases, Cambridge Press, London, 1988.

[58] V. Martins, R. Akbarinia, E. Pacitti, and P. Valduriez. Reconciliation in the APPA P2P System. In *Proc. of Int'l Conf. on Parallel and Distributed Systems (ICPADS)*, 2006.

[59] M. Masud, I. Kiringa, and A. Kementsietsidis. Don't Mind Your Vocabulary: Data Sharing Across Heterogeneous Peers. In *Proc. of the Int'l Conf. on Cooperative Information Systems (CoopIS)*, pages 292-309, 2005.

[60] M. Masud, I. Kiringa, and H. Ural. Update Propagation and Data Synchronization in Instance Mapped Peer Data Sharing Systems. In *Proc. of the VLDB Workshop on Database Interoperability (InterDB)*, 2007.

[61] M. Masud and I. Kiringa. Acquaintance Based Consistency in an Instance-Mapped P2P Data Sharing System During Transaction Processing. In *Proc. of the Int'l Conf. on Cooperative Information Systems (CoopIS)*, pages 169-187, 2007.

[62] M. Masud and I. Kiringa. Data Synchronization in an Instance-Mapped P2P Data Sharing System. In *Proc. of the Int'l Conf. on Distributed Objects and Applications (DOA)*, 2007.

[63] M. Masud and I. Kiringa. Maintaining Consistency in a Failure-Prone P2P Database Network During Transaction Processing. In *Proc. of Int'l EDBT Workshop on Data Management in Peer-to-Peer Systems (DAMAP)*, pages 27-34, 2008.

[64] M. Masud and I. Kiringa. PDST: A Peer Database Simulation Tool for Data Sharing Systems. In *Proc. of Int'l Conf. on Simulation Tools (SIMUTools)*, 2008.

[65] Y. Masunaga. A Relational Database View Update Translation Mechanism. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 309-320, 1984.

[66] S. Mehrotra, R. Rastogi, Y. Breitbart, H.F. Korth, and A. Silberschatz. Overcoming Heterogeneity and Autonomy in Multidatabase Systems. In *Information and Computation*, 167(2):132-172, 2001.

[67] P. Mork, S. D. Gribble, and A. Y. Halevy. Managing Change in Large-Scale Data Sharing Systems. UW CSE Technical Reports UW-CSE-04-04-01.pdf., 2004

[68] S. Naicken, B. Livingston, A. Basu, S. Rodhetbhai, I. Wakeman, and D. Chalmers. The State of Peer-to-Peer Simulators and Simulations. In *ACM SIGCOMM Computer Communication Review*, 37(2):95-98, 2007.

[69] R. S. Nair, J. A. Miller, and Z. Zhang. Java-based Query Driven Simulation Environment. In *Proc. of Winter Simulation Conference (WSC)*, pages 786-793, 1996.

[70] W. S. Ng, B. C. Ooi, K. L. Tan, and A. Y. Zhou. PeerDB:A p2p-based System for Distributed Data Sharing. In *Proc. of Int'l Conf. on Data Engineering (ICDE)*, pages 633-644, 2003.

[71] NS-2. http://www.isi.edu/nsnam/ns/.

[72] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible Update Prpagation for Weakly Consistent Replication. In *Proc. of ACM Symposium on Operating Systems Principles*, pages 288-301, 1997.

[73] L. Penserini, M. Panti, and L. Spalazzi. Agent-Based Transactions into Decentralised P2P. In *Proc. of Int'l Conf. on Autonomous Agents and Multiagent Systems (AAMS)*, pages 1288-1289, 2002.

[74] PlanetLab. World Wide Web URL: *http://www.planet-lab.org/*

[75] P. Rodriguez-Gianolli, M. Garzetti, L. Jiang, A. Kementsietsidis, I. Kiringa, M. Masud, R. Miller, and J. Mylopoulos. Data Sharing in the Hyperion Peer Database System. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 1291-1294, 2005.

[76] M. Rusinkiewicz and A. Sheth. Specification and Execution of Transactional Workflows. In *Modern Database Management*, Addison-Wesley, 1995.

[77] Y. Saito and M. Shapiro. Optimistic Replication Algorithms. In *ACM Computing Surveys (CSUR)*, 37(1):42-81, 2005.

[78] N. Santoro. Design and Analysis of Distributed Algorithms. Wiley InterScience Publisher, 2006

[79] M. Schlosser and S. Kamvar. Simulating a P2P File-Sharing Network. In *Proc. of Int'l on Workshop on Semantics in Grid and P2P Networks*, 2002.

[80] C. Schuler, H. Schuldt, C. Türker, R. Weber, and H. Schek. Peer-to-Peer Execution of (Transactional) Processes. In *International Journal of Cooperative Information Systems (IJCIS)*, 14(4):377-406, 2005.

[81] L. Serafini, F. Giunchiglia, J. Molopoulos, and P. Bernstein. Local Relational Model:A Logocal Formalization of Database Coordination. Technical Report, Informatica e Telecomunicazioni, University of Trento, 2003.

[82] H. Shu. Using Constraint Satisfaction for View Update. In *ACM Journal of Intelligent Information Systems*, 15(2):147-173, 2000.

[83] N. E. Taylor and Z. G. Ives. Reconciling while Tolerating Disagreement in Collaborative Data Sharing. In *Proc. of the Int'l Conf. on the Management of Data (ACMSIGMOD)*, pages 13-24, 2006.

[84] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, A Weakly Connected Replicated Storage System. In *Proc. of the ACM Symposium on Operating Systems Principles*, 1995.

[85] Türker, K. Haller, C. Schuler, and H. Schek. How Can We Support Grid Transactions? Towards Peer-to-Peer Transaction Processing. In *Proc. of the Int'l on Conference on Innovative Data Systems Research (CIDR)*, pages, 174-185, 2005.

[86] J. D. Ullman. Information Integration Using Logical Views. In *Proc. of the Int'l Conf. on Database Theory (ICDT)*, pages 19-40, 1997.

[87] Z. Wang, S. K. Das, M. Kumar, and H. Shen. An Efficient Update Propagation Algorithm for P2P Systems. In *Computer Communications*, 30(5):1106-1115, 2007.

[88] A. Wolski and J. Veijalainen. 2PC Agent Method: Achieving Serializability In Presence Of Failures in A Heterogeneous Multidatabase. In *Proc. of Int'l Conf. on Databases, Parallel Architectures and Their Applications (PARBASE)*, pages 321-330, 1990.

[89] W. Yang and N. Abu-Ghazaleh. GPS: A General Peer-to-Peer Simulator and its Use for Modeling BitTorrent. In *Proc. of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* , pages 425-432, 2005.

[90] A. Zhang, M. Nodine, and B. Bhargava. Global Scheduling for Flexible Transactions in Heterogeneous Distributed Database Systems. In *IEEE Transactions on Knowledge and Data Engineering* , 13(3):439-450, 2001.

[91] W. Zhou and W. Jia. A Token-Based Independent Update Protocol for Managing Replicated Objects. In *International Journal of Computer Systems Science and Engineering*, 17(3):189-207, 2002.