Journal of Universal Computer Science, vol. 21, no. 11 (2015), 1439-1453
submitted: 10/5/14, accepted: 12/1/15, appeared: 1/11/15

Synchronizing Data through Update Queries in Interoperable E-Health and Technology Enhanced Learning Data Sharing Systems

Mehedi Masud
(Computer Science Department, Taif University, Taif, Saudi Arabia
mmasud@tu.edu.sa)

M. Shamim Hossain
(Software Engineering Department, College of Computer and Information Sciences, King Saud University, Riyadh, Saudi Arabia
mshossain@ksu.edu.sa)

Atif Alamri, Ahmad Almogren, Mohammed Zakariah
(Research Chair of Pervasive and Mobile Computing, College of Computer and Information Sciences, King Saud University, Riyadh, Saudi Arabia
atif@ksu.edu.sa, ahalmogren@ksu.edu.sa, mzakariah@ksu.edu.sa)

Abstract: Data interoperability and synchronization among heterogeneous data providers in collaborative e-health systems are challenging research issues. Efficient data management techniques promote an efficient way of sharing data. This paper describes existing approaches to data interoperability (platform independency) for exchanging and synchronizing data between heterogeneous data sources or various platforms. We also illustrate an update query execution protocol, which can reduce query execution cost and query response time. We further perform different experiments to validate the effectiveness of the proposed approaches.
Key Words: e-health, collaborative environment, data synchronization, query update
Category: H.2.0, H.2.4, H.2.5

1 Introduction

Collaborative systems such as e-health [Hossain 2011] and e-learning [Deed and Edwards 2011, Forment et al. 2010] provide an effective distributed data-sharing and multimedia data management environment among data sources. In such an environment, data sources are autonomous in managing data (i.e., curate, revise, update). Although sources update their data independently, at some points they reconcile the updates to maintain data consistency and support full data-sharing collaboration. Therefore, update actions (e.g., insertion, deletion, change) on data executed in a source may update the related data in the collaborative sources.

In general, e-health service providers store medical data (e.g., images and videos of diagnosis, plain text, etc.) on patients and provide remote access to

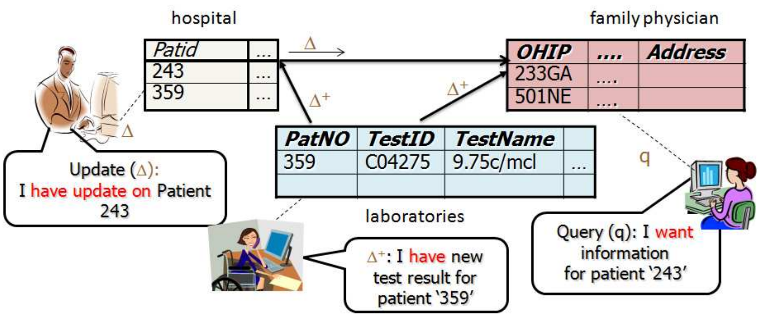

Figure 1: A collaborative e-Health network

these data for healthcare providers (e.g., physicians, clinics, medical researchers, etc.). In a classic system, patients medical data are distributed in autonomous data sources or by e-health service providers. Medical and multimedia data are stored and maintained in these sources without considering any centralized or global architecture. Hence, heterogeneity may result among the sources in terms of data vocabularies and storage schemes. A mechanism is thus needed for the interoperability of data in order to share and exchange data among sources. In such sources, an appropriate data structure is also required to represent the metadata of the multimedia content to provide fast query processing.

To illustrate this, consider the collaborative e-health system in Figure 1. The three healthcare service providers (hospital, family physician, laboratory) share patients medical care information. In such a system, an update action in a source may trigger updates in other service providers or sources in order to synchronize or reconcile the related data, i.e., any medical test result found at the laboratory triggers updates at the family physician and hospital data sources. In the same way, any update at the hospital data source may trigger an update at the family physician data source.

There are two fundamental problems to deal with in order to process updates at these sources:

1. Update translation problem.

To exchange an update action from a source to its related source, the update action needs to be translated with respect to the source storage schema and vocabularies of the related source. This update action should be translated for the related source in such a way that data updated at the original source and the related source satisfy the constraints between these sources. Formally, if update action $\triangle$ is executed at source S_i and its translated version $\triangle'$ is executed at acquainted source S_j, then the execution of $\triangle$ and $\triangle'$ at S_i and S_j makes the data sources consistent with respect to the mappings between S_i and S_j. The authors in [Green et al. 2007, Masud and Kiringa 2011] present approaches for

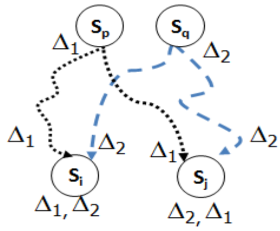

Figure 2: A conflict situation

such a translation.

2. Data synchronization problem.

In distributed systems, each source is designed and administered autonomously and there is no centralized control for update execution in the system. Different circumstances may occur during the execution of an update action, which may lead to data inconsistency between two related sources. One of the reasons for having inconsistent data is the different execution orders of updates at the sources. The different execution order results from a conflict among the updates. In the following, we give an example of such a data inconsistency scenario.

Example 1. Consider two updates $\triangle_1$ and $\triangle_2$ originated at sources S_p and S_q, respectively. Assume that $\triangle_1$ and $\triangle_2$ have been propagated to two sources S_i and S_j from two different paths. This scenario is shown in Figure 2. Assume that $\triangle_1$ and $\triangle_2$ are conflicting updates. Updates $\triangle_1$ and $\triangle_2$ need to be executed in the same order for maintaining data consistency at the data providers. Without sharing the conflict information, S_i and S_j are unable to make a decision on the execution order. These two updates may have been involved in a conflict with other sources and they have been executed in a certain order.

This paper considers the problem of data synchronization during the execution of update queries and proposes a decentralized mechanism for resolving conflicts. It presents a decentralized collaborative approach to resolve conflicts. These conflicts are resolved among the sources by sharing information on update execution status. Essentially, each source that executes a query update exchanges information with the updates originator. The originator of the update action has the prime role in ensuring execution correctness, resolving conflicts, and terminating update execution. A candidate update is selected by the conflicting update originators and executed finally in the system.

The next section presents the update query execution model and the protocol of update execution. Section 3 presents the data consistency notions and framework for synchronizing data. These consistency notions describe the constraints of data consistency. Section 4 describes the assessment results. Section 5 describes the related study and Section 6 concludes the paper.

2 Update Query Execution Model

A query is a read or write, i.e., update operation on data in a source. There are two types of queries: (i) a *read query* and (ii) an *update query.* A read query only reads data. This is not considered for data synchronization in the proposed conflict resolution protocol. The execution of a read query is completed without participating in the conflict resolution protocol. An update query contains write operations. Update queries participate in the proposed data synchronization model if they are involved in a conflict. Therefore, our concern is only with the execution of an update query in the system.

In a collaborative system, if a user possesses query $\triangle_i$ into source S_i, the source executes the query. Then, for $\triangle_i$ source S_i becomes the initiator. In order to maintain data consistency and synchronization between the two sources, if $\triangle_i$ updates data at S_i and if the updated data by $\triangle_i$ has a relation with the data at S_j, which is an acquainted source of S_i, then the related data at S_j have to be updated. Generally, mappings establish relationships between the data in the two sources. When $\triangle_i$ is received by S_j, source S_j executes $\triangle_i$ and forwards $\triangle_i$ to its acquainted sources related to the updated data. The source that has no acquaintees to forward the query is called the terminate source. Hence, a query is propagated from the query originator to all the acquainted sources before the query propagation terminates.

The execution scenario of an update query $\triangle_i$ originated at S_i can be represented as a tree. $\triangle_i$ is viewed as a one-level depth tree with $\triangle_i$ as a root and the participants as children. S_i produces a group of participant updates from $\triangle_i$ for the execution in its acquaintees. Update $\triangle_i$ is transformed into a multi-level update if the acquaintances of S_i produces participant updates. As participants are produced in the system, an *Update Execution Tree (UET)* is evolved. The nodes in the tree depict the sources where the update executes and the label shows the participant updates. An edge from a source S_i to a source S_j represents that S_i has forwarded $\triangle_i$ to S_j for execution. When a source accepts an update query, the source executes the query (if the query is not involved in a conflict with any other update query that is initiated by a different source) or the source halts the execution of the update (if a conflict is detected). When the execution is halted the update is involved in the conflict resolution protocol to be considered for a candidate update in order to be executed further or stopped. If the update is considered for candidate then the update execution continues. Otherwise, the update is compensated and execution is terminated.

The execution scenario of update query $\triangle_i$ originated at S_i can be represented as a tree. $\triangle_i$ is viewed as a one-level depth tree with $\triangle_i$ as a root and the participants as children. $\triangle_i$ produces a group of participant updates from $\triangle_i$ for the execution in its acquaintees. Update $\triangle_i$ is transformed into a multilevel update if the acquaintances of S_i produce participant updates. As participants are produced in the system, an *Update Execution Tree (UET)* is evolved. The nodes in the tree depict the sources where the update is executed and the label shows the participant updates. An edge from source S_i to source S_j represents that S_i has forwarded $\triangle_i$ to S_j for execution. When a source accepts an update query, the source executes the query (if the query is not involved in a conflict with any other update query that is initiated by a different source) or halts the execution of the update (if a conflict is detected). When the execution is halted, the update is involved in the conflict resolution protocol to be considered for a candidate update in order to be executed further or stopped. If the update is considered for a candidate update, update execution continues. Otherwise, the update is compensated and execution is terminated.

The execution of an update query in a collaborative system also differs from the execution model of a transaction in a distributed transaction execution. In a distributed system, users submit a transaction to the global transaction manager (GTM). Then, the GTM decomposes the transaction into a set of sub-transactions and forwards each sub-transaction to the respective local database. Each local database then independently executes the sub-transaction. In our proposed framework, an update is propagated as a single update without being decomposed into sub-transactions. In a distributed system, the GTM controls the execution of transactions. In our model, there is no GTM or controller.

2.1 Update query execution

A source starts building a dynamic tree, called a UET, when a user possesses a query to a source. A UET is used to monitor the execution of the update in the system. The construction of a UET for an update is discussed below.

1. If update query $\triangle_i$ is submitted to source S_i, the source executes $\triangle_i$ and creates a root node to construct a UET for $\triangle_i$. The root node is labeled with $\triangle_i$.

2. If the data updated by $\triangle_i$ in S_i have a relation with the data in source S_j, S_i generates participant update $\triangle_j$ to be executed at S_j for data synchronization and forwards $\triangle_j$ to S_j. If a participant is forwarded, a new $\triangle_j$ is generated. An edge is also inserted between $\triangle_i$ and $\triangle_j$. Note that when the initiator forwards an update, it also forwards its id. In this way, the receiver of the update knows the initiator of the update.

3. When an acquainted source receives a participant update, it performs the following tasks:

i. Executes the update.

ii. The acquainted source informs the initiator by sending a *vote* message. The source then waits for a $forward$ message from the initiator. After receiving the $forward$ message the acquainted source further forwards the update in the system. The vote message includes all the newly produced participant updates that also include the ids of the acquainted sources where new updates will be forwarded by the source.

4. If the vote message is received by the initiator, the initiator produces a group of nodes from the newly generated updates listed in the vote message. The initiator then creates edges in the UET between the new updates and the update from which the new updates are produced. The update initiator next informs the vote message sender by sending a *forward* message.

5. A source forwards new updates to other related sources if the source receives the forward message.

Figure 3 shows the update query execution technique.

3 Data Synchronization Framework

Data sources in an e-health data-sharing system execute updates without any centralized control. In such a system, there is no requirement for a protocol for multisite commitment [Hwang et al. 1994], which blocks the execution of queries. Therefore, the protocol is not feasible for a distributed e-health system. In an e-health system, sources execute updates locally and then forward updates asynchronously to other related sources. Therefore, conflict may occur for the same pair of updates when different sources execute and propagate updates asynchronously. To obtain a consistent execution of conflicting updates in the system, sources must execute the updates involved in a conflict in the same order. In this section, we present the notion of ensuring a consistent execution of updates and propose a framework for achieving such a consistent execution.

3.1 Notion of ensuring consistency

Data in a data source is partitioned into two partitions: (i) shared data (SD) and (ii) local data (LD). Shared data are shared with other sources and local data are not shared. An update can access LD and SD data items where the update is initiated. A local source also maintain the local and shared data consistency. Local data consistency is maintained if an update modify only the local data. However, if an update accesses shared data, then the data synchronization and consistency must be managed in the local as well as in the acquainted sources. Related sources are the sources that store data of common interest. Hence, if a shared item x is updated then the update is forwarded to the acquainted sources for coordination and synchronization of the shared data [Masud and Kiringa 2007].

Definition 1 conflict. Let $\triangle_i$ and $\triangle_j$ be two updates to be executed in a source. Let $WS(\triangle_i)$ and $WS(\triangle_j)$ denote the set of data items on which $\triangle_i$ and $\triangle_j$ perform write operations, respectively. A conflict occurs between $\triangle_i$ and $\triangle_j$ if $WS(\triangle_i) \cap WS(\triangle_j) \neq \emptyset$.

Actor:{Initiator (S_i), Sources ($S_j, S_k, \cdots$)}
Users submit an update query $\triangle_i$ at S_i
Initiator (S_i):
    S_i execute $\triangle_i$
    Starts building $EUT(\triangle_i)$
    Inserts a node with the symbol $\triangle_i$, i.e., root node $\triangle_i$, in $EUT(\triangle_i)$
    Π = Finds acquaintances for execution of $\triangle_i$
    if $|\Pi|$=0 **then**
     Terminates $\triangle_i$
    else
      Generates query updates $\triangle_j, \triangle_k, \cdots$ for each source in Π
      while true **do**
       Initiator waits for a vote message v_j from the sources in Π
       for each vote v_j **do**
        Creates a node $\triangle_j$ in $EUT(\triangle_i)$ for each newly generated update $\triangle_j$ and adds an edge from $\triangle_i$ to $\triangle_j$.
       endfor
       if checkTerminate($\triangle_i$)==true **then**
        Sends a terminate message to all the sources
        terminate while loop
       endif
      endwhile
     endif

Source S_j:
  while true **do**
   halt the execution for receiving an update query message m
   switch type of message m **do**
    case $\triangle_j$
     executes $\triangle_j$
     send vote v_j to the initiator
    case *forward* message
     forward $\triangle_j$ to the acquaintances of S_j
    case *terminate* message
     terminate execution of $\triangle_i$
   endswitch
  endwhile

Figure 3: Update Query Execution Protocol

A typical approach for ensuring data consistency during the execution of conflicting updates is synchronizing updates in such a way that updates' execution is serializable. Generally, in a distributed system, the Global Transaction Manager (GTM) ensures that update's execution order is consistent in the sources. With respect to the execution of updates in two sources (S_i, S_j), the following condition must be satisfied:

– S_i and S_j must execute conflicting updates into the same serial. Formally, for all pair of conflicting updates $(\triangle_p, \triangle_q)$ in a set of updates $\triangle = \{\triangle_1, \triangle_2 \cdots, \triangle_n\}$ executed in all the pair of sources (S_i, S_j) then $\triangle_1 \prec_{O_{S_j}} \triangle_2$ iff $\triangle_1 \prec_{O_{S_i}} \triangle_2$. Here, $\triangle_1 \prec_{O_{S_i}} \triangle_2$ represents execution order of updates $\triangle_1$ and $\triangle_2$ at S_i and S_j is an acquainted source of S_i.

In the following we introduce the notion of update execution consistency in a source as well as into its related sources.

Definition 2 Child-Level Order. A *child-level* order $\mathbb{S}_i^c = O(S_i) \cup (\bigcup_{j=1}^{m} O(S_j)$ w.r.t an order $O(S_i)$ in source S_i for a group of updates Γ is the union of order $O(S_i)$ and all the orders $O(S_j)$ at source S_j $(1 \leq j \leq m)$, such that each S_j is acquaintee of S_i.

Definition 3 Child-Level Consistent Order. A *child-level* order $\mathbb{S}_i^c$ is named *child-level* consistent w.r.t an order $O(S_i)$ in S_i for a set of updates $\Gamma = \{\triangle_1, \triangle_2, \cdots, \triangle_n\}$ and all orders $O(S_j)$ in S_j over each child S_j if

1. for any two updates $\triangle_1$ and $\triangle_2$ in $O(S_i)$, if there exists an order $\mathbf{O} < \triangle_1, \triangle_2 >$, such that $O(S_j) \in \mathbb{S}_i^c (i \neq j)$, the $\mathbf{O}$ is consistent between $\triangle_1$ and $\triangle_2$

Consistent updates' execution is achieved by maintaining the *Child-Level Consistent Order* in each child of the propagation paths of the updates.

Theorem 4. *An execution order $O(S_i)$ consists of a set of updates $\Gamma = \{\triangle_1, \triangle_2, \cdots, \triangle_n\}$ is consistent in a update propagation path $(S_i \rightarrow \cdots \rightarrow S_z)$ if for each child in $(S_i \rightarrow \cdots \rightarrow S_z)$, $\{\mathbb{S}_i^c, \cdots, \mathbb{S}_z^c\}$ is child-level consistent.*

Theorem 5. *An execution order $O(S_i)$ consisting of a set of updates $\Gamma = \{\triangle_1, \triangle_2, \cdots, \triangle_n\}$ is consistent over the system with respect to $O(S_i)$ in S_i if over every propagation path $(S_i \rightarrow \cdots \rightarrow S_z)$, and for each child in individual path $(S_i \rightarrow \cdots \rightarrow S_z)$, $\mathbb{S}_i^c$ is child-level consistent, and all the propagation paths between S_i and S_z is acyclic.*

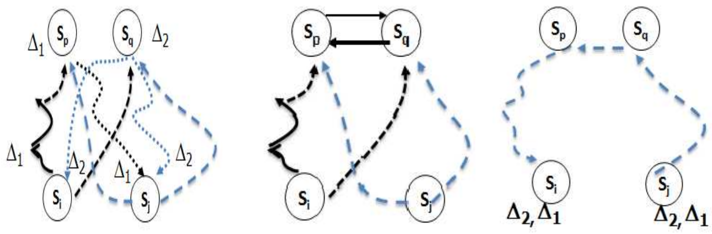

Figure 4: Update synchronization process (a) Process of sending conflict information to the initiators (b) Initiators decide order (c) Decision of update order reaches to the requester

3.2 Data synchronization approach

This section presents a data synchronization framework when two updates initiated from two different initiators are involved in a conflict during their execution in the system. A candidate update is selected from the updates that are conflicting and for which execution is completed in the system. Consider a scenario that source S_k accepts two conflicting updates from two different initiators. Source S_k cannot make a decision on which one to accept and which one to reject without the execution knowledge of the conflicting updates in other sources. A conflict may occur between two updates at different sources during their propagation in a collaborative system for an arbitrary network topology. To maintain data consistency in the sources, updates involved in a conflict must be executed in the same serial order.

Our proposed update-processing approach is optimistic. Optimistic approaches support accessing data while executing update operations. Users can change and read the database in the sources when the sources are disconnected and can synchronize the data with the acquainted sources when the sources reconnect [Petersen et al. 1997, Kermarrec et al. 2001]. Optimistic approaches support relaxed consistency [Saito and Shapiro 2005]. In relaxed consistency, update propagation is asynchronous without halting read queries. Major commercial vendors provide this asynchronous replication mechanism. In the following, we discuss the process of maintaining data consistency.

1: If a source receives updates from two different sources and a conflict is detected, the source halts the execution of the updates. The source also asks the parent about the order of update execution.

Note: A source becomes a parent when it forwards an update and becomes a child when it receives one.

2: The parent asks its parent if it has no data about the execution order. The inquiring message is propagated until it reaches a source that is aware of the updates execution order or the updates' initiators.

Note: An intermediate source may have identified the conflict of the same updates and forwarded an inquiry message to the initiators; hence, the execution order may have been decided already. Therefore, a source that detects a conflict for the same group of updates may have received the conflict resolution result through an intermediate data source from the path to the update initiator [Hossain et al. 2014].

Now we demonstrate the protocols to determine the execution order of the two conflicting updates. By applying two resolution protocols (*friendly resolution* and *absorption resolution*), an execution order is determined. The process is described below.

See the Figure 4(a). Sources S_p and S_q have initiated two updates $\triangle_1$ and $\triangle_2$, respectively. Consider that there is a conflict between the updates $\triangle_1$ and $\triangle_2$ and the sources S_i and S_j have detected the conflict. In Figure 4(b), the propagation of the conflict information message is shown with dotted lines from S_i to S_q and S_p. Since initiators are informed by a vote message by the sources after executing updates, the initiators of the updates know the count of sources where the updates have been executed successfully. We consider this number as the update execution *level* execution. The level is denoted by $level(\triangle_i)$. How the initiators reach an agreement of the update order execution when they receive the conflict information from S_i and S_j (see Figure 4(c) is described below.

See Figure 4(a). Sources S_p and S_q have initiated two updates $\triangle_1$ and $\triangle_2$, respectively. Consider that there is a conflict between updates $\triangle_1$ and $\triangle_2$ and sources S_i and S_j have detected the conflict. In Figure 4(b), the propagation of the conflict information message is shown with dotted lines from S_i to S_q and S_p. Since the initiators are informed by a vote message by the sources after executing the updates, the initiators of the updates know the count of sources where the updates have been executed successfully. We consider this number as the update execution *level*. The level is denoted by $level(\triangle_i)$. How the initiators reach an agreement on the update order execution when they receive the conflict information from S_i and S_j (as shown in Figure 4(c)) is described below.

Friendly Resolution: If ($level(\triangle_1) = level(\triangle_2)$)

When the initiators receive the conflict information of updates they select a candidate update, which one to execute and which to reject. Consider the update $\triangle_1$ is considered for a candidate update. After selecting the candidate the compensation process starts for $\triangle_2$. While the compensation process is running, no update is executed in the sources where $\triangle_1$ and $\triangle_2$ have been executed. The propagation of $\triangle_1$ and $\triangle_2$ is also stopped. To start the compensation process, S_2 forwards a compensation message to the sources those executed $\triangle_2$. Each source now generates a compensate update $\triangle_2^-$ and performs the compensation process. After finishing the compensation task the sources inform to S_2. After that execution of $\triangle_1$ starts.

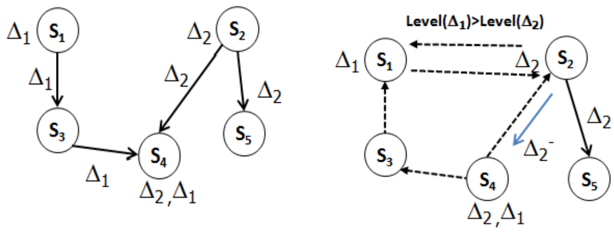

Figure 5: Example of conflict resolution(a) Update propagation and conflict (b) Initiators decide order

Absorption Resolution: If (*level*($\triangle_1$)$\neq$ *level*($\triangle_2$))

When the initiators receive the conflict information they select a candidate based on following protocol:

if level($\triangle_2$)>level($\triangle_1$) then $\triangle_2$ is selected as a candidate update else $\triangle_1$ is selected as a candidate update and the compensation process starts as presented in the friendly resolution.

Example 2. See in Figure 5. Two sources S_1 and S_2 initiated two conflicting updates $\triangle_1$ and $\triangle_2$ in the system.

Step 1: S_1 forwards the update $\triangle_1$ to S_3. Source S_3 executes $\triangle_1$ and forwards $\triangle_1$ to S_4. Meanwhile, S_2 also forward update $\triangle_2$ to S_4 and S_5. Notice that $\triangle_1$ and $\triangle_2$ involve in a conflict at S_4.

Step 2: Sources S_4 sends the information about conflict to S_3 and S_2 after identifying the conflict. However, S_3 has no information about the conflict. Therefore, S_3 forwards the conflict information to S_1

Step 3: When both the initiators S_1 and S_2 receive conflict information they decide the candidate update from the updates. Here, $level(\triangle_1) > level(\triangle_2)$. Hence, the absorption protocol is considered and $\triangle_1$ is selected as a candidate by both S_1 and S_2. The candidate information is sent to S_4.

Step 4: If $\triangle_2$ is executed before $\triangle_1$ then S_4 first executes a compensate update $\triangle_2^-$ and then executes the updates $\triangle_1$.

4 Evaluation

We show the different evaluation results of the proposed data synchronization approach and update processing. We consider a large collaborative setting to evaluate the proposed data synchronization approach. The simulator [Masud and Kiringa 2008] is used to make the environment. The same JVM is used to set up all the sources and a distinct thread is used for each source. For communication between the sources, we implement a FIFO queue. Each source

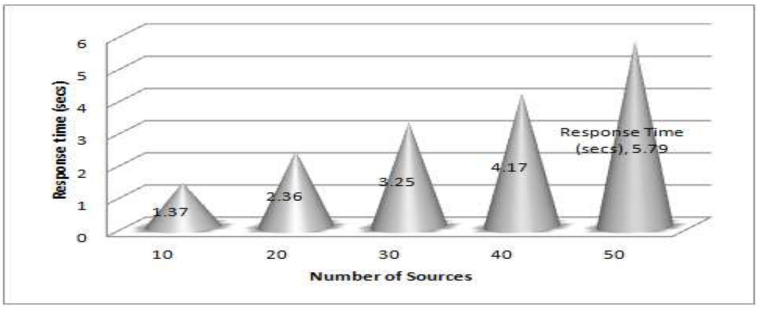

Figure 6: Response time of an update execution in different size of networks

uses a MySQL database for data storage. Update actions are MySQL select commands, i.e. read query, and update commands, i.e., write action. In the system, there is no network delay since all the sources run on the same computer. For the evaluation, we only consider write-write conflicts between update actions.

We first determine the response time of update execution by applying the proposed approach considering various network sizes, i.e., how many data sources are in the system. We consider an arbitrary topology for each network. Figure-Figure 6 shows the result of the observed evaluation. It is observed that response time increases linearly for an increasing network size. This proves that the protocol has scalability.

We evaluate the proposed conflict resolution protocol with different conflict situations among the updates. We monitor how the proposed conflict resolution protocol influences update execution time in the system. The updates are originated concurrently from 10 sources in a 100-source network. For the evaluation, we consider different patterns of conflicts among the updates. In the first pattern, there is no conflict among the updates. In the second pattern, a conflict occurs between two updates, in the third pattern, a conflict occurs among three updates, and so on. A conflict may occur among five updates. Figure 7 shows the evaluation results. It is noted that update execution time increases with increasing number of conflicting updates; however, the increase in time is not serious. We show that execution time gradually increases with an increasing number of conflicts. This proves that the resolution protocol is efficient.

We also conducted performance evaluation of the presented data synchronization technique considering various conflict factors to determine the result of the conflicting factors. A *conflict factor* is calculated as (number of conflicting updates/number of active or running updates) in the system. For this analysis we consider twenty data sources, ten update, and different sources initiated the updates. Figure 8 shows the evaluation results. The updates are selected in such a fashion that they participate in a conflict as defined conflict factors. The conflict factors are 0.1, 0.2, 0.3, 0.4, and 0.5. It is observed that update execution time increases with the increasing value of update conflict factors. However, the

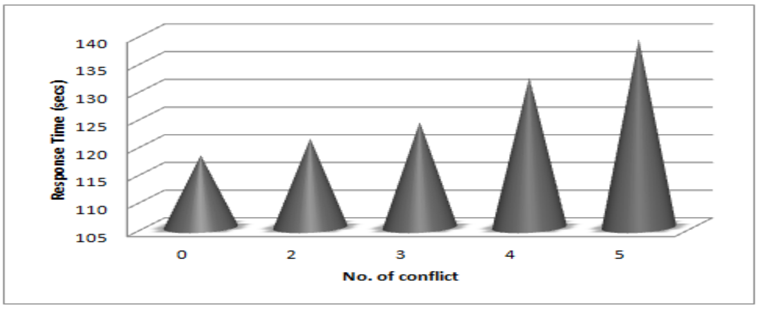

Figure 7: Execution time in update conflicting situations

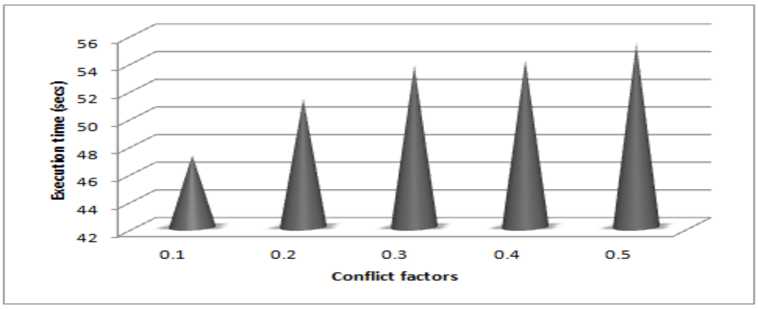

Figure 8: Number of conflicting updates and their execution time

impact is not serious.

We also conduct a performance evaluation of the presented data synchronization technique considering various conflict factors to determine the result of the conflicting factors. A *conflict factor* is calculated as (number of conflicting updates/ number of active or running updates) in the system. For this analysis, we consider 20 data sources, 10 updates, and the different sources initiated by the updates. Figure 8 shows the evaluation results. The updates are selected in such a fashion that they participate in a conflict as defined conflict factors. The conflict factors are 0.1, 0.2, 0.3, 0.4, and 0.5. It is observed that execution time of updates increases with the conflict value factors. However, the impact is not serious.

5 Related works

Multimedia interoperability and content management among healthcare information systems is an important research domain. The authors in [Chang 1994, Yuksel and Dogac 2011] focus on using traditional distributed database query-processing techniques, data interoperability for multimedia interoperability, and content management. The authors in [Yuksel and Dogac 2011, Porumb et al. 1997] focus on multimedia database query processing and data sharing. Yuksel and Dogac [Yuksel and Dogac 2011] proposed a framework for data interoperability in clinical data-sharing applications and medical devices. There are fundamental differences in our approach compared with the above-mentioned approaches. We present a data interoperability solution between data sources in an e-health system that is patient-centric to facilitate collaboration

Porumb et al. [Porumb et al. 1997] presented a prototype of virtual collaboration for providing the real-time collaboration of telemedicine services among users. The approach provides services at the application level. However, we propose a framework for establishing collaboration among e-health service providers.

Mork, Gribble, and Halevy in [Mork et al. 2004] presented update management techniques for a big data-sharing environment, using view maintenance techniques. A source in the system is a receiver of data and the schema is made of a view of other data sources schemas. Other sources act as data providers.

Bertossi and Bravo in [Bertossi and Bravo 2007] presented the semantics of database repair for ensuring data consistency in peer-to-peer data-sharing systems. The repair semantics for data inconsistency between peers is handled at query time. However, we consider an update execution technique in a collaborative data-sharing system where each data source shares its data with others and imports data from related data sources.

6 Conclusion

We present protocols for data synchronization through an exchange of updates in an e-health data-sharing environment. Each data source in the environment independently maintains data consistency. However, data are synchronized among the data sources in a collaborative fashion. Mainly, data sources resolve conflicts through an exchange of conflict information. In addition, we present a collaborative framework for metadata management that handles various medical multimedia content types, such as videos, images, text graphics, and audio. The framework first determines the media content features and then generates metadata to represent the media. Finally, we generate an identifier for the media to support efficient query processing. For efficient query processing, we present an agent-based update query execution protocol.

Acknowledgments

This work was supported by the Research Center of College of Computer and Information Sciences (CCIS), King Saud University through the research project no RC120902. The authors are grateful for this support.

References

[Bertossi and Bravo 2007] Bertossi, L., Bravo, L.: "The semantics of consistency and trust in peer data exchange systems"; Proc. Intl Conf. on Logic for Programming

Artificial Intelligence and Reasoning, 2007, 107-122.

[Chang 1994] Chang, Y.: “Interoperable query processing among heterogeneous databases”; PhD thesis, University of Maryland, (1994), 3-40, Maryland, USA.

[Chen 2010] Chen, S.: “Multimedia databases and data management: a survey”; Int. J. of Multimedia Data Engineering and Management, 1, 1(2010) 1-11.

[Deed and Edwards 2011] Deed, C. and Edwards, A.: “The Role of Outside Affordances in Developing Expertise in Online Collaborative Learning”; Intl. J. Knowledge Society Research, 2, 2(2011) 25-36.

[Forment et al. 2010] Forment, M. A., De Pedro, X., Casa, M. J., Piguillem, J., and Galanis, N.: “Requirements for Successful Wikis in Collaborative Educational Scenarios”; Intl. J. Knowledge Society Research, 1, 3(2011) 44-58.

[Green et al. 2007] Green, T., Karvounarakis, G., Ives, Z., Tannen, V.: “Update Exchange with Mappings and Provenance”; Proc. 33^{rd} Int’l Conf. on Very Large Data Bases (VLDB). 2007, 675-686.

[Hossain 2011] Hossain, M.S.: “Adaptive media service framework for health monitoring”; in Proc. ACM ICIMCS’11, Chengdu, China, August 5-7, 2011.

[Hossain et al. 2014] M. Shamim Hossain, M. Masud, G. Muhammad, M. Rawashdeh, M. M. Hassan, ”Automated and user involved data synchronization in collaborative e-health environments,” Elsevier Computers in Human Behavior, vol. 30, pp. 485-490, Jan 2014.

[Hwang et al. 1994] Hwang, S., Srivastava, J., Li., J.: “Transaction Recovery in Federated Autonomous Database Systems”; Distributed and Parallel Databases, 2, 2 (1994) 151-182.

[Kermarrec et al. 2001] Kermarrec, AM., Rowstron, A., Shapiro, M., Druschel, P.: “IceCube Approach to The Reconciliation of Divergent Replicas”; Proc. ACM symposium on Principles of Distributed Computing (PODC), 2001, 210-218.

[Masud and Kiringa 2011] Masud, M., Kiringa, I.: “Update Translation in Instance-Mapped Heterogeneous Peer Databases”; Int’l Journal of Semantic Computing, 5, 2 (2011) 211-234.

[Masud and Kiringa 2008] Masud, M., Kiringa, I.: “PDST: A Peer Database Simulation Tool for Data Sharing Systems”; Proc. Int’l Conf. on Simulation Tools (SIMUTools), 2008, 1-6.

[Masud and Kiringa 2007] Masud, M., Kiringa, I.: “Acquaintance based consistency in an instance-mapped P2P data sharing system during transaction processing”; Proc. OTM Confederated International Conference on On the Move to Meaningful Internet Systems: CoopIS, 2007, 169-187.

[Mork et al. 2004] Mork, P., Gribble, S., Halevy, A.: “Managing Change in large-scale data sharing systems”; UW CSE Technical Reports UW-CSE-04-04-01.pdf., 2004.

[Petersen et al. 1997] Petersen, K., Spreitzer, M., Terry, D., Theimer, M., Demers, A.: “Flexible Update Prpagation for Weakly Consistent Replication”; Proc. ACM Symposium on Operating Systems Principles, 1997, 288-301.

[Porumb et al. 1997] Porumb, C., Porumb, S., Orza, B., and Budura, D.: “Computer-Supported collaborative work and its application to e-Health”; Proc. Conf. on Advances in Mesh Networks, 2000, 75-80.

[Saito and Shapiro 2005] Saito, Y., Shapiro, M.: “Optimistic Replication Algorithms”; ACM Computing Surveys (CSUR), 37, 1(2005) 42-81.

[Yuksel and Dogac 2011] Yuksel, M. and Dogac A.: “Interoperability of medical device information and the clinical applications: an HL7 RMIM based on the ISO/IEEE 11073 DIM”; IEEE Trans. On Information Technology in Biomedicine, 15, 4(2011) 557-566.