A LAZY REPLICATION SCHEME FOR LOOSELY SYNCHRONIZED UDDI REGISTRIES

Mahantesh Surgihalli, K Vidyasankar
Memorial University of Newfoundland
St John’s, Newfoundland
Canada – A1B3X5
{mahan, vidya}@cs.mun.ca

ABSTRACT

Web services have created a platform-neutral environment for business processes to communicate. The number of companies using Web services is expected to grow exponentially. As such, the number of queries to name directory services, Universal Description Discovery and Integration (UDDI) registries, is expected to be very high. To facilitate high availability, UDDI version 3 suggests replication. For quick response and scalability, a lazy replication scheme is appropriate for UDDI registries. Here, update transactions are executed at primary site(s), committed, and then updates are transmitted to other sites asynchronously. Then, for a user accessing different sites in the operations of a session, a session guarantee mechanism is needed to ensure consistent access.

In this paper, we propose a lazy replication scheme that ensures one copy serializability and provides session guarantee. Advantages of our algorithm include — flexibility in executing update transactions pessimistically or optimistically, and a fine-grained session guarantee.

KEY WORDS
Concurrency control, Transactions, Lazy replication, Session guarantee, Web services, UDDI Registries.

1. Introduction

1.1 Background

The growth of Internet technologies has unleashed a wave of innovations that has forced companies towards automation, efficient business processes and global visibility. Web services [1] enable publishing of business functions in the Web and accessing them universally. UDDI registry is a name directory used for Web services. UDDI defines data structures and API’s for publishing service descriptions (publish API) and querying (inquiry API). The main requirements of UDDI registries are high throughput, low response time, high availability and accurate accessibility to its contents. As UDDI is still evolving, consistency issues remain to be addressed to the best of our knowledge.

1.2 Consistency Issues in Replicated Databases

Maintaining consistency among the replica copies (replica control) has been studied extensively in conventional databases. In a replicated database scenario, one copy serializability (1SR) [2], in which a data item appears to clients as in a single non-replicated database, is the normal correctness criterion. The database replication issues can be divided into two parts — execution of transactions and propagation of messages between nodes. A transaction manager executes transactions and broadcast primitives propagate updates. The types of broadcast primitives are reliable, causal and total order.

The two main classes of replica control strategies are lazy [3,4,5] and eager [6,7] replications. In lazy replication, the transaction is first executed and committed at one site called the primary site, and updates are propagated later to other sites called secondary sites asynchronously. Transactions executing at the primary and secondary sites are called primary transactions and refresh transactions, respectively. As soon as a primary transaction is executed, the user gets the response from the system. In eager replication, there is coordination among the replica copies considered in the context and update is executed at all those sites synchronously. The user gets the response from the system only after the transaction is successfully committed at all those sites. [8] claims that the eager replication would never scale. Scalability is not a problem with lazy replication as it decouples the primary transaction execution and propagation of updates to other sites. This decreases the response time, making the system highly available. Therefore, lazy replication is popular among commercially available databases. UDDI has used the lazy replication strategy [9].

Inconsistency arises in lazy replication as replica copies may be out of synchronization for a certain period of time. A user interacting with the system with a sequence of operations may get an inconsistent view if different replica copies are accessed. A session guarantee mechanism aims to give a consistent view to a user individually. Different types of session guarantees are described in [10].
1.3 Related Work

We briefly review the related work of lazy replication schemes in conventional databases and name service directories.

[3] uses lazy replication where primary transactions can be executed at any site. Once they have been successfully executed, they are said to be in the precommit state. They are propagated to other sites by epidemic propagation (exchanging up-to-date information by choosing another site at random; in a way passing them through the system like an infectious disease) of messages to detect conflicts. The transaction commits successfully if no conflicts are detected at any other site.

Providing session guarantees within a transactional framework is presented in [4]. They ensure a new correctness criterion called strong session serializability which is weaker than strong serializability (where every transaction is executed in the same order at all the sites), but stronger than 1SR. Recently, providing a freshness guarantee in partially replicated databases has been dealt with in the PDBREP project in [5]. As it is a lazy replicated database, where all the data items may not be up-to-date, the user is allowed to specify the freshness requirements for transactions.

In traditional name service directories like Grapevine [11] and Clearinghouse [12], weak consistency called eventual consistency is employed. It is not serializable but all data items of all the copies eventually reach the same state. It is a sufficient correctness criterion in the applications for which they are used. Due to the lack of trust between organizations and issues like confidentiality in UDDI, a serialized view of transactions is required. Any weaker consistency criterion should be avoided. In [13], a comparison of UDDI registry replication strategies is given.

1.4 Contribution summary

In this paper, we present a new protocol for loosely synchronized replica copies. Our protocol ensures 1SR. It is essentially a two phase locking (2PL) protocol but allows the user to obtain locks pessimistically or optimistically. The protocol implements a causal broadcast primitive. It eliminates false causality [14]. A session guarantee mechanism is integrated into the protocol. Ours is a fine-grained session guarantee and provides better performance than those in the literature [5].

The remainder of this paper is organized as follows: Section 2 investigates the consistency problems in UDDI. In section 3 we present our replication protocol. A discussion appears in section 5. Section 6 concludes the paper.

2. Preliminaries

2.1 Data structure and configuration in UDDI

UDDI data structure consists of businessEntity (BE), businessService (BS), bindingTemplate (BT), tModel (tM), PublisherAssertion (PA) and Subscription entity types. BE, BS and BT have parent-child relationships among them. BE may have one or more BS. BS may have one or more BT. All these entity types can refer to the tM using reference relationships. The relationship between business organizations is defined by PA. UDDI is a fully replicated registry. Each node has custody of a portion of the data and any changes to the datum must originate at this custodian node [9].

2.2 Local and global transactions

As UDDI is completely replicated, all read-only transactions (ROTs) can read data items at a single node. As such, ROTS are local transactions. Update transactions accessing data items in custody of a single node are also local transactions. Update transactions accessing data items in custody of two or more nodes are called global transactions. According to [9], “Changes to a datum in the registry must originate at the node which is the custodian of the datum.” Update values of the primary transaction are packed into a change-record. The change-record is broadcast to other nodes in the registry. These change-records are executed by individual nodes as refresh transactions.

2.3 Session guarantee

In this work, we are primarily interested in Monotonic reads guarantee [10]: “The Monotonic reads guarantee permits users to observe a database that is increasingly up-to-date over time. It ensures that read operations are made only to database copies containing all writes whose effects were seen by previous reads within the session.”

We know that all the nodes in the registry start with the same initial state. As the system ensures 1SR, all copies will reach the same final state. As different transactions may execute primary transactions at different nodes, there may not be the same sequence of states among the nodes. If the user always reads data items from only one node, session guarantee is ensured trivially. This may be a major restriction, as efficient load balancing cannot be achieved. We provide the flexibility for consecutive transactions in a session to access different nodes.

3. Replication protocol in UDDI

In our protocol, we select a coordinator for the execution of a primary transaction. A primary transaction may access data items which are in the custody of nodes
other than the coordinator. These nodes are called participant nodes. Other nodes in the registry are called non-participant nodes. The custodian node of a data item is responsible for resolving conflicts on that data item. Each node has its own transaction manager which uses a variation of strict 2PL [2] for transaction execution. A node in the registry can request locks from any other node.

Let us consider a registry with \( n \) nodes. Each primary transaction is associated with an Update Sequence Number (USN) which is unique at the coordinator node. The USN at a node increases monotonically. A transaction in a registry is uniquely identified by the pair \(<\text{Node-ID}, \text{USN}>\). The state of the node can be defined in terms of the primary transactions and refresh transactions that have been executed at that node. We list the data structures each node will have with respect to a node, say ‘Node-k’:

1. A two dimensional array \( N_k[i][1,2,3,…n] \). \( N_k[k][k] \) is the USN of the latest transaction executed at Node-k. \( N_k[i][1,2,3,…n] \) denotes the state of Node-i known to Node-k.
2. Present state vector \( P_k[] \) is \( N_k[k][] \).

After the transaction has been executed at the coordinator node, updates of the transaction are broadcast as change-records to other nodes. The USN of the primary transaction is assigned, at commit time, to the change-record. The change-record contains the node identifier of the coordinator, USN, node identifiers of the participants of the write set, write values of the transaction, present state vector \( (P[i]) \) and dependency vector \( D[i] \) (which indicates preceding conflicting transactions). Change-records are stored at all the nodes until they are explicitly deleted by the delete_change-record procedure.

<table>
<thead>
<tr>
<th>REQUEST</th>
<th>PT-S</th>
<th>PT-X</th>
<th>RT-X</th>
<th>ROT-S</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>R</td>
<td></td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A</td>
<td>PT-X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>RT-X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>ROT-S</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table -1 Lock compatibility

We assume that the system ensures reliable broadcast of messages. Our protocol requires the causal delivery for the change-record, lock grant and acknowledgement messages. Later, in the protocol, we show that causal delivery of the change-record occurs only among the conflicting transactions and hence would avoid false causality [14]. We have separate protocols for each type of transaction — primary, refresh and read-only. We also have separate types of locks for each of the transaction types. Primary transactions have two types of locks, namely PT-S and PT-X, representing read and write locks, respectively. Refresh transactions request only write locks, namely RT-X. ROTs request only read locks, namely ROT-S. Lock compatibility is as shown in Table-1. Causal transmission and lock compatibility issues are explained later with the protocol.

For primary transaction Ta

Execution of a primary transaction has four main stages:
1. Lock acquisition, 2. Execution, 3. Validation, 4. Release of locks. When a transaction starts, all locks for its data items are acquired at the coordinator node in stage 1. For global transactions, in addition to those locks, the coordinator acquires locks at its respective custodian nodes. It can be obtained either in stage 1 or 3. If all the required locks are obtained in stage 1, then it is a pessimistic execution. Otherwise, it is optimistic. We do not consider deadlock in this work. We assume that when one node requests a lock from some other node, it will get the lock within a finite amount of time.

The execution of transactions can be explained from the coordinator’s view, participant’s view and non-participant’s view.

For Coordinator

Consider a primary transaction executing at Node-i.

1. Acquiring read and write locks: Transaction requests all PT-S and PT-X locks at Node-i. The coordinator decides to request locks from participants either now or in stage 3. If it decides to acquire a lock on a data item now, it sends a lock-request message to the corresponding custodian node. This request message does not have the present state vector (P). After it gets the response for all its requests, it goes to the execution stage.
2. Execution: Transaction is executed.
3. Validation: Request the locks from those participants which were not requested in Stage 1. If it wants a read lock on data item ‘y,’ S-optimistic-request is sent. Along with its request, it sends the present state vector \( P[i] \). It requests a write lock using the X-optimistic-request message. It will get the locks after a bounded time period. The primary transaction can abort due to S-optimistic-request but it will never abort due to X-optimistic-request. Once it gets responses for all its requests, it goes to the commit/abort phase.

Commit/abort: If the transaction receives negative acknowledgement in response to any of its validation messages then it aborts, releases all its PT locks and sends lock-release-request message to all the participant nodes. On the other hand, if it gets all positive acknowledgements then a change-record is created as explained below. The USN is generated and the pair \(<\text{Node-ID}, \text{USN}>\) is assigned to it.

The present state vector \( P \) is obtained as follows:

\( P[i][1,2,3,…n] ← N_i[i][1,2,3,…n] \)

The dependency vector \( D_k \) for a transaction \( T_k \) is created as follows: Let us assume that the primary transaction \( T_k \) has WR, RW and WW conflicts, with a set of transactions, namely \( T_1, T_2,…,T_m \). Let the change-records corresponding to these transactions be CR1,
participating nodes are released after the execution of the session_update procedure. All the PT locks are released. Then, the change-record at that node. On the other hand, if the transaction aborts, all its locks are released and by the refresh protocol. Then, the primary transaction commits.

4. Release of locks: Once the primary transaction commits, all the PT locks are released. Then, the session_update procedure is executed. Locks obtained at participating nodes are released after the execution of the change-record at that node. On the other hand, if the transaction aborts, all its locks are released and lock-release-request is sent to all the participants.

**For Participants:**

Let us consider a participant node, Node-j.

**Upon receiving PT-S/PT-X lock request:** Node acquires the corresponding PT lock (PT-S or PT-X). Then, it sends the lock grant message along with the dependency vector D. D contains the preceding conflicting transactions on the requested data item.

**Upon receiving the lock-release-request:** Node releases all the PT locks held by transaction Ta.

**Upon receiving S-optimistic or X-optimistic request:** If the node receives an S-optimistic request on data item ‘y,’ PT-S lock on ‘y’ is requested. After the lock is obtained, a check is made to determine if there are any writes on ‘y’ between the state of request message and present state of the node. If false, positive acknowledgement is sent. Otherwise, negative acknowledgement is sent and locks are released right away. On receiving an X-optimistic request, it requests the PT-X lock. Along with the acknowledgement message, the present state vector (P) is also sent. Vector D is associated with both the above message types. These are used for causal delivery of the message (this will be explained later).

**Release of locks:** On execution of the corresponding refresh transaction, all the PT locks are released.

**For refresh transaction Ta**

The protocol is the same for the coordinator and participant nodes.

**For Coordinator and Participant**

**Acquisition of write lock:** Check for conflicting RT-X locks. If they exist, wait until they are released. RT-X locks are acquired on all the data items to be written. An RT-X lock on a data item can be given even when the transaction has a PT-X lock (refer to Table-1).

**Execution:** Transaction is executed and committed.

**Release of locks:** PT and RT-X locks on all the data items are released atomically.

For Non-participant

**Acquisition of write lock:** RT-X locks are acquired on all the data items to be written.

**Execution:** Transaction is executed and committed.

**Release of locks:** RT-X locks on all the data items are released atomically.

**For ROT Ta**

A ROT can be executed at any node.

**Acquiring read lock:** The ROT-S locks are requested on all the data items at the present node.

**Execution:** Once all the requested locks have been acquired, session_read check is performed. Then, the transaction is executed.

**Release of Locks:** It releases all the ROT-S locks.

**Lock compatibility issues**

We have two unique features in our lock compatibility:

1. **PT-X vs ROT-S:** In our protocol, primary transactions do not update data items. Refresh transactions modify data items in the registry. As such, when the primary transaction is holding the lock, ROTs can read the current (stale) data if permitted by session guarantee. This compatibility increases concurrency among transactions.

2. **PT-X vs RT-X:** When a transaction is granted a PT-X lock and a conflicting transaction requests an RT-X lock, they are compatible (please note the asymmetry in Table-1). Our protocol is designed in such a way that when a primary transaction is executing, only the preceding conflicting transactions will request locks and not any of the succeeding transactions. As write order of conflicting transactions is maintained at all the nodes, we show that the execution is indeed correct. An obvious benefit of this is that an optimistic execution for a global transaction does not require validation for write data items.

**Session guarantee for transaction Ta**

A user or organization interacts with the registry in a session. We represent the user session with the two dimensional array S[i][j]. Vector S[i][j] contains the USN of the transactions from which the user session has read. In other words, S contains the state of the registry last seen by the user. S[i][j] denotes the state of Node-i which is known by the user. We differentiate the session guarantee for update transactions and ROTs. ROTs can potentially read stale values, permitted by Monotonic read guarantee. The update transaction reads the latest value in the registry; otherwise our protocol will invalidate the transaction. The session_read procedure is called only before execution of ROTs; session_update is called after all transactions are executed in the system.

**Session_read:** This procedure takes a session’s knowledge about the custodian nodes of data items it is accessing. It calculates the maximum state by comparing and recording the maximum USN for each entry. A transaction can execute only if this state is reached at that node. Otherwise, it waits till that state is reached. Let the user read data items from Node-j. By knowing the data
For $i = 1$ to $n$, where $n$ is the number of nodes in the registry. Then, for $var = 1$ to $n$, where $n$ is the number of nodes

$$R[var] = \text{Max}(S[a][var], S[b][var], S[c][var])$$

If $R[1,2,3,...,n] \leq Nj[j][1,2,3,...,n]$ // $Nj$ belongs to Node-$j$

then execute ROT at Node-$j$

else

wait

**Session_update:** After a transaction $T$ commits, the result of the execution is returned to the user. The next transaction in the session cannot proceed until the **session_update** procedure is executed and the user receives the response. Ensuring Monotonic read guarantee means that the next transaction in the same session should read a higher state of the system than that read by $T$. A simple solution will be to keep track of the present state of the node when $T$ was executed and then execute the next transaction after that state has been reached at the executing node. To achieve higher concurrency, we provide a fine-grained session guarantee. The **session_update** procedure for a committed transaction $T$ can be explained as follows. Let $T$ be uniquely identified by $<ID, USN>$. Let $IDP1, IDP2, ..., IDPm$ be participants of $T$. From the serialization graph, we can find out the transactions from which $T$ has read. There will be change-records corresponding to each of the transactions, at that node. (If there is no change-record, no action is taken. This means that this transaction has been executed at all the nodes in the registry. So, there is no need to keep track of these entries.)

For $var = \{\text{List of change-records from which } T \text{ has read from}\} \cup \{T\}$

For $varp = \{\text{List of participant ids of a given var}\}$

$$S[varp][var] \leftarrow \text{Max}(\text{USN of var}, S[varp][var])$$

**Delete_change-record:** As all change-records generated in the registry are stored at all the nodes, they grow without bound. If some transaction $Tx$ is executed at some Node-$i$, we know that at some point in time, all the nodes in the registry will execute that transaction. Vector $Mk[]$ indicates the transactions that have been executed at all the nodes in the registry as known by Node-$k$. This procedure is invoked whenever there is a change in the $Mk[]$ vector. The procedure deletes all change-records lower than $<i, Mk[i]>$ for all $i$ from 1 to $n$. $Mk[]$ is calculated as follows:

For $i = 1$ to $n$, where $n$ is the number of nodes in registry

$$Mk[i] = \text{Min}(Nk[1][i], Nk[2][i], ..., Nk[i][i], ..., Nk[n][i])$$

where the Min function returns the minimum element of a set of elements.

**Causal transmission of messages**

Our protocol provides causal transmission of messages. If the message has to be received in causal order, it should contain $D[]$. The only messages having $D[]$ vectors are change-record, lock grant and acknowledgement messages; these follow causal ordered delivery. When Node-$i$ receives a message from Node-$j$ for a transaction $Tk$, the following code is executed. ($Ni[]$ is associated with Node-$i$ and $Dk[]$ is associated with the change-record for transaction $Tk$)

if $Ni[i][1,2,3,...,j-1,j+1,...,n] \leq Dk[1,2,3,...,j-1,j+1,...,n]$

then deliver the change-record.

//($Ni[]$ and $Nj[]$ are vectors of Node-$i$ and Node-$j$ //respectively)

//After delivery of above the change-record, Ni is updated

$$Ni[i][j] \leftarrow P[j]$$

$$Ni[j][1,2,3,...,n] \leftarrow P[j][1,2,3,...,n]$$

It should be noted that in our algorithm, before the delivery of each message, the state vector of Node-$i$ ($Ni[]$) is compared with the dependency vector ($Dk[]$). This means that message delivery will be delayed only for the conflicting transactions. This imposes total order delivery between two update messages only if they are conflicting (not total order among all the messages). This eliminates false causality (the perception that because one event occurred before another, the earlier event has caused the later event). In this transactional framework, we require only the conflicting transactions to obey causality. Eliminating false causality relaxes the strict ordering between two non-conflicting transactions.

### 4. Correctness Proof

Due to space limitation, we only provide a brief outline of the proof: (1) At each node, the transaction manager ensures conflict serializability since our algorithm uses 2PL. (2) As conflicts between transactions due to a data item are resolved at its custodian node, the protocol ensures 1SR. (3) As the user session state captures the distributed state of the system, we show that our session guarantee mechanism ensures Monotonic read guarantee. Complete proof can be found in [15].

### 5. Discussion

The main features of the algorithm are:

1. **Flexibility:** The flexibility of requesting locks at hot spots pessimistically (reduces the abort rates due to high conflicts) and requesting locks on other data items during validation optimistically (higher transaction throughput) gives better performance.
2. Scalability: By organizing the data items according to a transaction access pattern, good performance can be obtained even with an increase in the total number of nodes. Minimum book keeping is required for session guarantee as it is proportional to the number of nodes instead of the number of data items. As a result, the protocol is highly scalable with an increase in the data items in the registry.

3. Distributed and fault tolerant: Our algorithm is distributed and there is no single point of failure. If a node fails, the custody of the data item is transferred to another node [9].

A brief comparison of our work with related works in literature follows: Our method is more efficient than [3] as once the primary transaction commits, the user gets the response from the system and synchronization is required only with participant nodes. While in [3], no transaction is allowed to read the write data items of a precommit transaction. Also, in order to check the conflicts between primary transactions, there is a need for synchronization between all the nodes.

Our method is more efficient than in [4] as we provide the flexibility of executing ROTs at any node in the registry whereas they allow ROTs only at specified nodes. As the session keeps track of the transactions from which it has read instead of the present state of the registry, our session guarantee is finer grained and provides more concurrency than theirs. We allow distributed transaction execution whereas theirs is centralized.

In [5], they have a centralized log which keeps track of the present state of the system whereas ours is completely distributed. In their case, if the user gives an invalid freshness requirement, the user may read an inconsistent view of the system. Our protocol provides a valid and consistent state of the system as permitted by Monotonic read guarantee.

6. Conclusion

Researchers have suggested replication in UDDI as the number of queries is expected to be very high. It is widely accepted that tight synchronization of replica copies is practically infeasible as it assumes that all sites are available at all times. Also, in distributed systems, tightly synchronizing copies means reduced availability. We have presented a distributed replication protocol with an embedded session guarantee mechanism. Even though replication has been dealt with extensively in the literature, the configuration we consider in this paper has not been addressed before. Our protocol provides the user with the flexibility to use the algorithm based on the conflict rate in the system. The session guarantee mechanism enhances fair load balancing as the user has liberty to execute consecutive transactions on any node. We ensure Monotonic read guarantee within the transactional framework. In the future, we would like to extend the work to partially replicated databases and consider the semantics of UDDI.

Acknowledgements

This research is supported in part by the Natural Science and Engineering Research Council of Canada Discovery Grant 3182.

References