CHAPTER 4

TRANSACTION DATA FLOW GRAPHS FOR REPLICATED DATABASE SYSTEMS

4.1 Introduction

In this chapter, we have considered a distributed system approach for coordination transactions in a replicated database system. In a distributed system, scheduling can be carried out by constructing partial graphs for individual transactions. Such a graph is referred to as transaction data flow graph (TDFG). This approach is referred to as a transaction data flow graph based approach. In this, a transaction data flow graph is constructed each submitted transaction. This process of scheduling is shown to be equivalent to scheduling of transactions through a data flow graph constructed for centralized database systems.

The technique is based on the use of precedence graphs [BER87]. By constructing a type of precedence graphs and enforcing control for accessing data items, a procedure is evolved for processing transactions. In the proposed algorithm, a transaction ($T_i$) arrives at a site. A transaction number (see section 2.5.1 of chapter 2) is assigned to it. It is executed

--- An earlier version of a portion of this chapter is to appear in ACM Operating Systems Review [RED93b].

--- A paper based on a part of this chapter has been submitted to Second International Conference on Information Systems and management of Data [RED93a].

--- A paper based on a part this chapter has been submitted to IEEE Transactions on Knowledge and Data Engineering [RED93c].
and its read-set and write-set are identified. These are sent to visit a (majority) number of sites. During the visit, if no transaction \( T_j \) is found to be in conflict with transaction \( T_i \), then \( T_i \) returns to home site, and commits the update values. In case of a conflict, the conflicting transactions are ordered with the help of their own TDFGs. This process results in a non-blocking protocol that provides a level of fault tolerance and availability that is same as the level provided by voting based protocols [THO79, GIF79] proposed earlier. The read only transactions (queries) need to visit a majority of sites in order to read a current value of the data items.

The proposed distributed transaction execution model, can be used with an optimistic or pessimistic technique for synchronization. We have adopted the use of pessimistic technique in order to compare it with other voting based approaches. The next section presents a system model. Sections 4.3, 4.4, and 4.5 describe the TDFG approach, the algorithm for generation of TDFGs, and a case example respectively. In the latter sections, proof of correctness (section 4.6) and comparison with other approaches (section 4.8) are discussed. Section 4.7 considers site and media failures. The implementation issues are considered in section 4.9. The last section concludes the chapter.

4.2 System model

In addition to the assumptions presented in chapter two (see section 2.1 and 2.2 of chapter 2), here we give more assumptions to explain the algorithm.

In order to consider a simplified case, we assume that the database is fully replicated. However, the algorithm does not depend on the support of full replication. The
network is assumed to detect failures, whenever these occur. We assume that the sites are fail-stop [SCH82]. Communication links may fail by crashing, or by failing to deliver messages. Combination of site failures and communication link failures may lead to partition failures [DEV85]. In a partitioned state, sites in a partition can communicate with each other, but no communication can occur between sites in different partitions.

A transaction is modeled as a sequence of read and write operations. A read only transaction does not issue write operations. The transactions represent complete and correct computations, i.e., if executed alone on a consistent database, these leave the database in a consistent state. The data items read by the transaction constitute its read-set (RS), and the items written by the transaction constitute its write-set (WS), respectively. Both RS and WS of a transaction constitute transaction variables (TVs). We assume that TVs of a transaction are fully determined at the end of an initial computation. The transactions $T_i$, $T_j$ are said to have R(read)-W(write), W-R or W-W conflict, if $RS(T_i) \cap WS(T_j) \neq \emptyset$; $WS(T_i) \cap RS(T_j) \neq \emptyset$; or $WS(T_i) \cap WS(T_j) \neq \emptyset$, respectively. In this chapter, we say $T_i \cap T_j \neq \emptyset$, to imply that a conflict exists between $T_i$, $T_j$.

4.3. Data flow graphs in a distributed system

A transaction data flow graph is a directed graph, whose nodes represent transactions to be executed, and edges indicate scheduling constraints between transactions. The order of transactions to be executed is directed by the edges in the graph.

In the following section, we examine the notion of data flow graphs. In section 4.3.2, we describe the transaction data flow graphs.
4.3.1. Data flow graph

Construction of DFG in the case of centralized database systems has been studied in [EIC88]. Similar to this study, a data flow graph for the present situation is defined as follows. Let, \( T = \{T_1, T_2, \ldots, T_n\} \) be a set of active transactions in a replicated distributed system. A data flow graph (DFG) is a directed acyclic graph, that represents an execution of \( T \). That is, a DFG is a graph \( G(V,E) \) where \( V \subseteq T \) and \( E = \{<T_i,T_j> | T_i \cap T_j = \emptyset \text{ and } T_i < T_j\} \).

Example 4.1: Consider the transactions \( T_a, T_b, T_c, T_d \) and \( T_e \). Let \( X, Y \) and \( Z \) be the replicated data items. Also, \( 'w_i' \) indicates a write operation by \( T_i \), and \( 'r_i' \) indicates a read operation by \( T_i \). Also given that,

\[
\begin{align*}
T_a &= w_a(X) w_a(Y) w_a(Z) & T_b &= r_b(X) r_b(Z) w_b(X) \\
T_c &= r_c(Y) r_c(Z) w_c(Y) & T_d &= r_d(Z) w_d(X) w_d(Z) \\
T_e &= r_e(X) r_e(Y) r_e(Z).
\end{align*}
\]

The notion of correctness of transaction execution in this context, is that of serializability. That is, an execution of a set of transactions is serializable, if its overall effect is equivalent to some serial execution of these transactions [BER85, BER87]. Therefore, in order to execute the above transactions many DFGs are possible, if we consider different serial executions.
Considering the serial execution: \( T_a < T_b < T_c < T_d < T_e \), an equivalent DFG corresponding to this serial execution is shown in figure 4.1.

![DFG for serial execution](image)

Figure 4.1. DFG for serial execution \( T_a < T_b < T_c < T_d < T_e \).

4.3.2. Transaction data flow graph (TDFG)

The TDFG is a data flow graph with certain properties. For each transaction \( T_i \), there exists a TDFG\(_i\). The TDFG\(_i\) is a graph \( G(V, E) \), where \( V \subseteq T_i \), and \( E = \{ <T_j, T_i> | T_j \cap T_i \neq \emptyset \) and \( T_j < T_i \} \).

In a distributed system, the transactions arrive for execution at different sites. The above TDFGs can be constructed by making each transaction request visit multiple sites. Also, union of TDFGs of active transactions is equivalent to a DFG, which ensures serial execution. The proof of equivalence between the two is described in section 4.3.3.
Consider the DFG of example 4.1. The equivalent collection of TDFGs of given transactions is shown in figure 4.2.

In figure 4.2, edge \( <T_i, T_j> \) denotes that \( T_j \) needs some data item which is also required by \( T_i \). Thus, in the beginning, \( T_a \) accesses the database. In figure 2(b), \( T_a \) and \( T_b \) are conflicting type. So, \( T_b \) accesses the database after \( T_a \). Similarly, in figure 2(d), \( T_d \) accesses the database after \( T_a, T_b \) and \( T_c \). Hence, the TDFG \( i \) includes all transactions \( T_j \) in the distributed system, which are in conflict and \( T_j \not< T_i \).
4.3.3. Theorem 4.1

Let \( T = \{ T_1, T_2, \ldots, T_n \} \) in the distributed system.

Then, \[ \bigcup_{i=1}^{n} \text{TDFG}_i = \text{DFG}. \]

Proof: We prove the following.

(a) an edge of DFG belongs to one of the TDFG; and

(b) an edge of TDFG belongs to DFG.

(a) Consider that, edge \(<T_k, T_m> \in \text{DFG}\) for some \(k\) and \(m\).
By definition of DFG, \( T_k \cap T_m \neq \emptyset \) and \( T_k \leq T_m \).
Also by definition,
\[ \forall T_i, \text{ if } T_i \cap T_m \neq \emptyset \text{ then } <T_i, T_m> \in \text{TDFG}_m. \] (1)

From (2), edge \(<T_k, T_m> \in \text{TDFG}_m\).
Hence, an edge \(<T_i, T_j>\) in the DFG also belongs to TDFG.

(b) Consider the edge \(<T_q, T_s> \in \text{TDFG}_s\) for some \(q\) and \(s\).
By definition of TDFG, \( T_q \cap T_s \neq \emptyset \) and \( T_q \leq T_s \).
Also, by definition,
\[ \forall T_i, T_j, \text{ if } T_i \cap T_j \neq \emptyset \text{ and } T_i \leq T_j \text{ then } <T_i, T_j> \in \text{DFG}. \] (4)
From (4), edge \(<T_q, T_s> \in \text{DFG}\).
Hence, an edge \(<T_i, T_j>\) in the TDFG also belongs to DFG.

From (a) and (b) above, we get \[ \bigcup_{i=1}^{n} \text{TDFG}_i = \text{DFG}. \]
4.4. An algorithm to construct TDFG

For a transaction, the transaction variables are identified at the end of its computation. A transaction request is prepared and sent to visit a majority of sites. During this visit, transaction request forms the TDFG and returns. In the next step, the commit of a transaction results in changes being applied to the replicated database. Alternatively in this step, an abort results in the changes being discarded. A two-phase commit protocol [GRA78] is employed to guarantee the atomicity of transactions at multiple sites. The formation of TDFG and processing of transactions is presented in the following section.

4.4.1. Definitions

Home Site (SH_i)

The site of origin of transaction T_i, is designated as the home site (SH_i) of T_i.

Transaction Request (TR_i)

Transaction request TR_i for transaction T_i consists of transaction number (TN_i), Transaction variables (TVs_i), Initial TDFG_i and number of sites visited (N_i). It is initialized by the SH_i on arrival of each transaction.

Transaction Number (TN_i)

The TN_i is a unique number (S,I,C), assigned to the transaction on its arrival by the SH_i. In this chapter, both the notations TN_i and T_i are used interchangeably and represent individual transactions identities.
Transaction Variables (TVs)

For a transaction (Tj), the read-set (Tj) and write-set (Tj) are identified as transaction variables. Thus, TVs = RS(Tj)∪WS(Tj).

Number of Visited Sites (Nj)

A variable (Nj), indicates the number of sites visited by the TRj. Nj is incremented by one, whenever a request (TRj) proceeds to a new site.

Active Transaction List (ATL)

The ATL is maintained by every site. ATL of site Sj = { (TNj, TVsj) | Tj has visited Sj }. Whenever a transaction commits, its entries are deleted from the ATL.

Odd Edge, Even Edge

An edge <Tj, Tk>, such that TNj > TNk, is called an odd edge. Otherwise, it is called an even edge.

Majority (M)

Majority indicates the number of sites that each TRj must visit. If 'P' is the total number of sites in the system then,

\[ \text{Majority}(M) = \begin{cases} \frac{P+1}{2} & \text{if } P \text{ is odd;} \\ \frac{P}{2}+1 & \text{if } P \text{ is even}. \end{cases} \]
Message Edge Table of $T_i$ (MET$_i$)

The MET$_i$ is maintained at the SH$_i$. It contains the list of edges and committed transactions identities. These messages are sent during confirmation of odd edges, or during transaction commit and update. MET$_i$ is divided into two parts.

**MET$_i$.edges:**

After a majority visit, when $T_j$ returns to its home site, if $\text{TDFG}_j$ contains the odd edge $<T_i,T_j>$, the information is communicated to the SH$_i$. At the SH$_i$, if $T_i$ has not returned from majority visit, then even edge $<T_j,T_i>$ is stored in the "MET$_i$.edges". Later, the edge is inserted into the $\text{TDFG}_i$ on return of $T_i$ at SH$_i$.

**MET$_i$.commit:**

MET$_i$.commit contains the list of committed transaction identities which have been communicated to SH$_i$. While $T_i$ is under majority-visit, if SH$_i$ receives the commit of transaction $T_k$, such that $T_i \cap T_k \neq \emptyset$, then $T_k$ is stored in the "MET$_i$.commit".

4.4.2. Informal description of the algorithm

In this algorithm, whenever a transaction arrives at a site for execution, it is assigned a TN value. Let $T_i$ be the TN. A TR$_i$ is prepared. It is sent to visit a majority of sites starting with the home site. After a majority visit, the TR$_i$ returns to SH$_i$ with the $\text{TDFG}_i$. If $\text{TDFG}_i$,
contains odd edge <T_j, T_i>, then it is confirmed by consulting the SH_j. At the SH_j, if T_j is in execution state or in the process of committing the update values, then the odd edge <T_j, T_i> is not deleted from TDFG_i. If T_j has not started execution, the even edge <T_i, T_j> is inserted into the "MET_j.edges", the odd edge <T_j, T_i> is deleted from TDFG_i. In this way, odd edges are confirmed, resulting in a deadlock free environment. (see section 4.9 on Implementation Issues). The algorithm is described below.

4.4.3. Formal description of the algorithm

I. Notations

SH_i ---> Home site of transaction T_i.

T_i.state ---> After submission of a transaction, until commit, the state of transaction (T_i.state), indicates one of the three possibilities. This value is maintained at the SH_i.

T_i.state = majority-visit; T_i has not returned from majority visit.

T_i.state = waiting; T_i is waiting for the commit messages from preceding transactions in its TDFG_i.

T_i.state = executing; T_i has started execution, and in the process of committing the update values.

TDFG_i.conflict-set ---> Set of transactions in TDFG_i which are in conflict with T_i. That is,

TDFG_i.conflict-set = { vertex set of TDFG_i - vertex T_i }.

MET_i ---> Message edge table of T_i.
On arrival of a transaction, the home site assigns a transaction number \( T_j \), and executes the transaction. Next, the site prepares the transaction request as \( TR_j = (TN_j, TVs_j, \text{TDFG}_j, N_j) \). The \( \text{TDFG}(V,E) \) is initialized to \( V = \{ T_j \} \), \( E = \emptyset \). Also, \( N_j := 0 \). The tables, \( \text{MET}_j\text{.edges} \) and \( \text{MET}_j\text{.commit} \) are initialized to element \( \emptyset \). \( T_j\text{.state} \) is assigned to majority-visit. The message \( TR_j \) is issued to the local site.

II Types of messages

The algorithm exchanges the following different messages. At any site \( S_j \), depending on the message received, a specific action is invoked (see appendix B.2).

\[ TR_i \text{ (Transaction request of } T_j) \):

Whenever a site receives a \( TR_i \) message, the following steps are performed:

\[ \begin{align*}
\text{If } N_i < M & \text{ (} TR_i \text{ has not completed majority visit), the } (TN_i, TVs_i) \text{ are stored into the ATL. An edge } <T_j, T_i> \text{ is inserted into the } \text{TDFG}_j \text{, for all transactions } T_j \text{ which are in conflict with } T_i \text{. The value } N_i \text{ is incremented by one. If } N_i = M \text{ then, } TR_i \text{ is sent to } SH_i \text{. Otherwise, } TR_i \text{ is forwarded to another site.} \\
\text{Else, if } N_i = M & \text{ (} TR_i \text{ is returned to } SH_i \text{), the edges from } "\text{MET}_i\text{.edges}" \text{ are inserted into the } \text{TDFG}_i \text{. The committed transactions of } "\text{MET}_i\text{.commit}" \text{ are deleted from } \text{TDFG}_i \text{. Also, } T_i\text{.state is assigned to waiting. If } T_i\text{.conflict-set}=\emptyset \text{ and } \text{MET}_i\text{.commit}=\emptyset \text{, then } T_i\text{.state is assigned to executing and the update values are committed. Else, if } T_i\text{.conflict-set}=\emptyset \text{ and } \text{MET}_i\text{.commit} \neq \emptyset \text{, then } T_i \text{ is executed and the}
\end{align*} \]

79
update values are committed. Otherwise, if odd edge \(<T_j,T_i>\) exists in the TDFG, for each odd edge \(<T_j,T_i>\), the message DELETE\(<T_j,T_i>\) is sent to SH\(_j\). MET\(_i\) is deleted.

**DELETE\(<T_j,T_i>\):**

When a site receives the message 'DELETE\(<T_j,T_i>\)', if \(T_j\).state = majority-visit, the edge \(<T_i,T_j>\) is stored in "MET\(_j\).edges" and message 'YES\(<T_j,T_i>\)' is sent to SH\(_j\). However, if \(T_j\).state=waiting, then the edge \(<T_i,T_j>\) is inserted in the TDFG\(_j\), and the message 'YES\(<T_j,T_i>\)' is sent to SH\(_j\).

**YES\(<T_j,T_i>\):**

The message YES\(<T_j,T_i>\) is the response to the message DELETE\(<T_j,T_i>\). The vertex \(T_j\) is deleted from TDFG\(_j\). If \(T_i\).conflict-set=\(\emptyset\), then \(T_i\).state is assigned to executing. Next \(T_i\) is executed and the update values are committed.

**COMMIT \(T_j\):**

The entries concerning \(T_j\) are deleted from the site ATL. At SH\(_j\), for all active transactions \(T_i\) such that \(T_i \cap T_j \neq \emptyset\), if \(T_i\).state=majority-visit, then \(T_j\) is stored in "MET\(_i\).commit". Else if, \(T_i\).state=waiting, then the vertex \(T_j\) is deleted from TDFG\(_i\). Also, if \(T_i\).conflict-set=\(\emptyset\), then \(T_i\).state is assigned to executing. Next, \(T_i\) is executed, and the update values are committed.
4.5. Case example

Consider the following example using the algorithm. There are five sites in the network (\(S_1, S_2, S_3, S_4,\) and \(S_5\)). For the sake of simplicity, given a TN(S,I,C), we consider S and I components to form the TNs.

<table>
<thead>
<tr>
<th>TN</th>
<th>Site traversal sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T_{1,20})</td>
<td>1,2,4,1</td>
</tr>
<tr>
<td>(T_{5,20})</td>
<td>5,4,2,5</td>
</tr>
<tr>
<td>(T_{3,21})</td>
<td>3,1,5,3</td>
</tr>
</tbody>
</table>

(a)

Figure 4.3. An example of site visits by transactions (a) Path of traversal for transactions. (b) ATLs at various sites.

Transactions requests are processed by various sites. The majority is defined by three sites. The path of traversal for the requests is shown in figure 4.3(a). Assume that the
transmission time between two sites is one unit, and processing time of requests at different sites is comparatively small (negligible).

Initially, transactions arrive for execution at $S_1$, and at $S_5$ at local clock time 20. These transactions are assigned the TNs $(1, 20)$ and $(5, 20)$. Figure 4.4, depicts the transaction traversal pattern over sites. From the table, it can be noticed that at $t$ (time) = 20, $T_{1,20}$ is at $S_1$ and $T_{5,20}$ is at $S_5$; at $t = 21$, $T_{1,20}$ is at $S_2$, $T_{5,20}$ is at $S_4$, and $T_{3,21}$ is at $S_3$. Let us assume that, the three transactions are in conflict with each other for access to data item ‘X’ (say).

<table>
<thead>
<tr>
<th>Requests/time</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{1,20}$</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$T_{5,20}$</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>$T_{3,21}$</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.4 Transaction traversal pattern over units of time.

Using the proposed algorithm, the process can be explained as follows. In figure 4.3(b), the contents of ATLs of sites are shown. Initially, ATLs of $S_1$, $S_5$ are empty. These transactions (TRs) are processed at the initial sites and corresponding TVs are stored at the site ATL. Next, the TRs are forwarded to other sites. When $T_{1,20}$ visits $S_2$, its TVs are added to the ATL of $S_2$, and it is further sent to $S_4$. Similarly, $T_{5,20}$ visits $S_4$, and its TVs are...
included in the ATL of $S_4$. On arrival of $T_{1,20}$ at $S_4$, on comparison with transaction $T_{5,20}$, the two requests are conflicting type, so $T_{1,20}$ inserts the edge $<T_{5,20}, T_{1,20}>$ in the TDFG of $T_{1,20}$. On completion of visit to a majority of sites, the TDFGs of $T_{1,20}$, $T_{5,20}$ and $T_{3,21}$ are shown in the figure 4.5.

![TDFGs on completion of majority visit.](image)

After visiting majority of sites, when $T_{1,20}$ finds the odd edge $<T_{5,20}, T_{1,20}>$ in the TDFG of $T_{1,20}$, the message 'DELETE $<T_{5,20}, T_{1,20}>$' is sent to $S_5$. From $S_5$, message 'YES $<T_{5,20}, T_{1,20}>$' is sent to $S_1$. Node $T_{5,20}$ is deleted from TDFG of $T_{1,20}$. The TDFGs are shown in figure 4.6.
The execution sequence of transactions is as per the partial order $T_{1,20} < T_{5,20} < T_{3,21}$.

### 4.6. Proof of correctness

**Theorem 4.2**: Let, $RD(H)$ be the replicated data history over $\{T_1, T_2, \ldots, T_n\}$. If $G_T = \bigcup TDFG_i$, where $i=1$ to $n$, then $G_T = RDSG(H)$.

**Proof**: For any two transactions $T_i$ and $T_j$, if they are conflicting on a replicated copy of the data item then (because every transaction request is visiting a majority sites) either an edge $<T_i, T_j>$ is inserted into the $TDFG_j$, or an edge $<T_j, T_i>$ is inserted into the $TDFG_i$. So, all conflicting operations in replicated database system, are ordered in the $G_T$. So, from the definition of $RDSG$ (see section 2.4 of chapter 2), $G_T = RDSG$. 

---

Figure 4.6 TDFGs after elimination of odd edge.
**Theorem 4.3:** $G_T$ is acyclic.

**Proof:** For any blocked $T_i$ in $G_T$:

(i) $T_i$ forms an even edge with some $T_j$ ($<T_j, T_i> \in \text{TDFG}_i$). In this case, $T_i$ will be executed after $T_j$, resulting in a serializable sequence.

(ii) $T_i$ forms an odd edge with $T_k$ ($<T_k, T_i> \in \text{TDFG}_i$). In this case, the odd edge is removed as a result of odd edge confirmation, which results in a serializable sequence.

From theorem 3.2 (see chapter 2), the deadlock does not exist without an odd edge, and an even edge. In this way, all odd edges are removed and corresponding even edges are inserted in the respective TDFGs. So, $G_T$ is acyclic.

**Theorem 4.4:** The algorithm results in a one-copy serializable sequence.

**Proof:** From the theorem 4.2, $G_T = \text{RDSG}$, and from the theorem 4.3, $\text{RDSG}$ is acyclic. So, from the correctness criteria of replicated databases (see section 2.4 of chapter 2), the execution is one-copy serializable.

**4.7. Site and media failures**

In a real life database system, sites and the communication medium are prone to failures. These failures may result in loss of a message, site (computation facility) failures, or partitioning of the network. In this section, measures to cope with these failures are discussed.
A. Loss of Messages

Loss of messages can be handled by using the time-out mechanism. When a site $S_j$ sends a DELETE<$T_j,T_i>$ message to $S_j$, it initiates a timer. If the timer expires before receiving the acknowledgment, the $S_i$ sends the DELETE<$T_j,T_i>$ message to $S_j$ again. If the subsequent messages do not reach the receiver site, it is assumed to be cut off from the system, or the system is assumed to be in a partition state. In this way, if the site $SH_i$ fails, the corresponding edges <$T_j,T_i>$ are deleted from all TDFG_i.

B. Site failures

When a site fails, the subsequent TRs are not sent to the site. The entries of the transactions belonging to the failed site are deleted from ATLs, TDFGs and METs at other sites. If the site fails during TR processing, the situation can be dealt with in the following manner. If broadcast based communication is used for majority visit, the home site knows about the failed site (and it continues the processing by ignoring such a site). In case the daisy chaining mode of communication is used, a site may fail while handling a request. In this case, the home site employs a time-out mechanism. Within a time-out period, if a TR_i does not return, then the home site sends an abort message to all the sites.

C. Partition Failures

In the face of network partitioning, the proposed approach can continue update processing by using pessimistic strategy (which guarantees correctness at the cost of availability), or the optimistic strategy (that provides high availability by sacrificing
correctness) [DEV85].

In case of pessimistic strategy, a partition must ensure that no conflicting updates are executed in other partitions. The proposed approach can allow the execution of transactions within the majority group. The execution is suspended in the other group or groups.

In optimistic strategy, each partition processes transactions, as if it were an independent database system. The consistency within a partition is maintained. However, the global consistency is likely to be violated, because execution of transactions in different partitions is an un-coordinated activity. Consequently, when the system recovers, all inconsistencies among the partitions must be detected and resolved. Thus, each partition keeps a journal, which is the sequence of actions performed on its database copy during partitioning. On recovery, it derives a globally consistent state from the individual journals by using the precedence graph method [DEV84], or the log transformation technique [DEV85].

4.8. Comparison with voting based approaches

Voting based approaches are robust with respect to site failures and network partitioning [JAJ90, GIF79]. However, if there are conflicts, the conflicting transactions are rejected. In the worst case scenario, it is possible that a transaction is rejected a large number of times, and has to be resubmitted again and again (repeated roll-back). This phenomenon can occur in a high volume transaction activity environment. Also, increasing the number of sites adds to processing cost, in terms of number of messages. We first explain the voting based approach and compare it with the proposed approach.
4.8.1. Voting based approach

This approach has been suggested by Thomas [THO79]. It is a solution to the redundant update problem for a replicated databases. In this study, we only consider the daisy-chain version of this algorithm. The detail version of the approach is described in [THO79]. Here, we only give an extremely brief outline of the algorithm [THO79, GAR79].

In this algorithm, the sites in the system form a daisy chain or ring. Before a transaction can be performed, it must move along this chain obtaining votes. After a transaction gets a majority of "OK" votes, it can be performed. A transaction may also receive a "REJECT" vote, in which case it may not be performed.

When an update transaction $T_i$ arrives at a site, it immediately reads the data items desired (and their time-stamps) from the local database. Then the new update values are computed. Next, comes the voting phase where $T_i$ visits the sites along the chain. Each site votes on $T_i$ using the voting rule given below. As the transaction moves along the chain, it carries with it the time-stamps of the base-set items that were read at the originating site. After each vote, a site uses the request resolution rule, also given below, describes how the "accept" messages for $T_i$ (i.e., "perform update $T_i$" messages) are processed at each site. The steps followed by an update transaction are illustrated through a simple example in figure 4.7. We now give the rules used by the voting based approach.

1) Voting rule: Two transactions conflict if the intersection of the base set of one and the write set of the other is not empty. Each update transaction is assigned a priority equal to the site identification number of its originating site. Between the time a site
votes for a transaction and a transaction is resolved, the transaction is said to be pending at that site. The voting rule consists of five steps.

a) Compare the time-stamps for the transaction base set items with the corresponding time stamps in the local database.

b) Vote "Reject" if any base set item is obsolete. That is if we find a time-stamp which is more recent than the one that was read at the transactions originating site.

c) Vote "OK" is each base-set item is current and the transaction does not conflict with any pending transactions at the site.

d) Vote "Deadlock reject" if each base set item is current but the transaction conflicts with a pending request for higher priority.

e) Otherwise, defer voting and remember the transaction for later consideration.

2) Update resolution rule: After voting a transaction, each site uses this rule to decide what must be done next. The update resolution rule consists of four parts.

a) If the vote was "OK" and a majority of "OK" votes for the transaction exist, then the transaction is accepted. Accept messages are sent to all sites.
I. Transaction T_1 arrives at S_1 and gets its first OK vote.

II. Transaction T_1 visits S_2 where it gets its second OK vote.

III. Transaction T_1 visits S_3 where it gets third OK vote, and is accepted (notice that three votes constitutes majority in this five site system).

IV. "Perform update T_1" messages are sent to all sites.

V. Transaction T_2 arrives at S_4 and gets its first vote.

VI. Transaction T_2 visits S_5 and gets its reject vote.

VII. 'Reject T_2' messages sent to all sites.

Figure 4.7. Voting based approach: An example
b) If the vote was "Reject", then reject the transaction by sending out "Reject" messages to all sites.

c) If the vote was "deadlock reject" and a majority consensus is no longer possible, that is the transaction received a majority of "Deadlock Reject" votes), then reject the transaction by sending out "reject" messages to all sites.

d) Otherwise forward the transaction and the votes accumulated so far to the next site in the chain.

3) The Update application Rule: When a site learns that a transaction has been resolved, it uses the update application rule to either perform the update or reject it.

a) If a site receives an "accept $T_i$" message, the new values which are not obsolete are stored in the local database. That is, for each item in $T_i$'s write set, the site compares the item value time-stamp (TS), with the $TS(T_j)$. If $TS$ is less than $TS(T_j)$, and the item is modified as indicated and time-stamp for the item is set to $TS(T_j)$. If $TS$ is greater than $TS(T_j)$ then no modification is performed since the value is obsolete. All conflicting transactions that were deferred at the site because of $T_i$ are rejected.

b) If the site receives a "reject $T_i$" message, then the site uses the voting rule to reconsider conflicting requests that were deferred because of $T_i$. 

Gifford [GIF79] presents a generalization of majority consensus approach in which data items can be assigned different weights. However, we assume that all copies are equally weighted since the generalization has no impact on the ideas presented here. In this protocol, every transaction request visits a majority of sites. Operations on data items can be executed only when a transaction request obtains necessary locks on a set of data items from majority of sites. A transaction that writes to a data item, reads the version numbers from majority copies and then generates a higher version number. Version numbers are assigned to data items, to determine the current data copies. After a majority visit, if it is found that a transaction has been executed by using an obsolete copy of data, then it must be rolled back. Such a transaction is resubmitted again.

4.8.2. Comparison

In distributed systems, each extra message constitutes additional cost and processing time, leading to delays in transaction execution. We consider the number of messages required to accomplish a consensus decision to update the database for a given transaction. We assume that the number of messages required to commit the update values of a transaction is same in all approaches. Consider ‘N’ sites in the system. Communication between any two sites is termed as one message. A pure daisy chaining communication is assumed for the sake of comparison. A broadcast communication environment can be substituted without a loss of generality (also see section 4.9). To simplify the process, we assume that weight of each site is one. The comparison is divided over three headings (A-C) as given below. Also, we assume that for both approaches, the home site commits the final
update values.

A. No Transaction Conflicts

In no conflict situation, in both approaches, a transaction visits strictly majority sites and returns to home site. Next, the commit processing starts. So, for both approaches strictly ‘M’ number of messages needed for consensus.

B. Average Cost Comparison

A simulation study has been carried out to compare the average costs (see appendix C.2). In the simulation model, the number of data items at each site is fixed at 10,000 (figure 4.8). The number of sites (N) within the system is fixed at 10. The number of items requested by a transaction is generated by selecting a value between 4 and 12. All remote sites have an equal chance of being chosen for a request. Multiprogramming level (MPL) which is the total number of concurrent transactions active in the system is varied. It is changed over a set of tests.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database size at each site</td>
<td>10,000 data items</td>
</tr>
<tr>
<td>Number of sites (N)</td>
<td>10</td>
</tr>
<tr>
<td>Transaction size</td>
<td>4-12 data items</td>
</tr>
<tr>
<td>Multiprogramming level (MPL)</td>
<td>Variable (10, 20,...,80)</td>
</tr>
</tbody>
</table>

Figure 4.8. Simulation parameters.
In the simulation experiment, clocks are maintained at various sites. After arrival of a transaction at a site, the lock requests are sent to other sites. To consider a more generalized form, the data access sites are traversed one by one. Every message updates the clock of the visited site (see chapter 2). Messages are stored in First-Come-First-Served site queues. We have run the simulation for 5000 transactions, by fixing a MPL level. An average of all runs has been taken. We have evaluated the number of transactions rejected and average number of messages per transaction.

Figure 4.9 shows the trend of percentage of transactions rejected. As said before, in voting based approach, whenever the two transactions requests a common data item, only one transaction gets consensus vote (after visiting all sites in worst case), and the other one is rejected. As MPL increases, the number of transactions requesting common data items also increases, which results in more rejections. In the TDFG approach none of the above transactions is rejected. But, after returning from the majority, if there is a change in the transaction variables, then the transaction has to visit majority sites again to get a consensus for new set of transaction variables. Such a case will occur rarely in commonly used environments. We assumed that, this type of rejections are very less and are shown in the graph. In figure 4.10, average number of messages per transaction are shown. In voting based approach, the rejected transaction has to inform about the rejection information to all sites it has visited. After, it tries to get consensus, which might be rejected again. So, as the MPL level increases, the average number of messages increases in the voting based approach. But in the TDFG approach, only the odd edges formed in TDFGs increase. The graph in figure 4.10 shows the trends. Figure 4.11, shows the trend of number of rejected transactions as database size increases.
Figure 4.9. Multiprogramming level versus % of transactions rejected
Figure 4.10. Multiprogramming level versus average number of messages per transaction.
Figure 4.11. Database size versus % of transactions rejected

- Voting based
- TDFG approach

Multiprogramming level = 50
Number of sites = 10
C. Load comparison

Consider that, ‘k’ number of conflicting transactions enter into the system. We consider a resultant difference in the volume of transactions generated within the system, within the two approaches.

(i) Voting based approach

In the first round, among k conflicting transactions one transaction is accepted, and the remaining ‘k-1’ transactions are rejected. These ‘k-1’ transactions are submitted again. Similarly, in the second round ‘k-2’ transactions are rejected. Finally, it takes k rounds of submissions, to resolve k conflicts. So, in this approach, the effect of submitting k transactions is equivalent to the effect of submitting k(k+1)/2 non-conflicting transactions. Therefore, the overall transaction load (number of active transactions) on the system increases. The load effect (volume of active transactions) for k (k=10) conflicting transactions is compared for the above approach in figure 4.12.

(ii) TDFG approach.

In the first round, all k transactions are synchronized by forming the respective TDFGs. No transaction is rejected. So, the task of resubmission is eliminated.
In the normal mode (no conflict situation), voting based approach and TDFG approach exhibit similar behavior. However, as the number of conflicting transactions increases the performance characteristics vary. The TDFG approach takes less number of messages by avoiding the rejections in case of conflicts. The performance characteristics also vary with changes in number of processing sites. For networks with large amount of replication (in the event of conflicts) the TDFG approach performs better than the voting based approaches.

**4.9. Implementation issues**

Considering the data structures and algorithms required, the implementation of TDFG approach requires three types of data structures.

i. Active transaction list (ATL);

ii. Message edge table (MET); and

iii. Transaction data flow graph (TDFG).

The ATL contains transaction number and transaction variables for a transaction, that has visited the site. When a site receives the commit message, the entries corresponding to
the transaction are deleted from ATL. The size of ATL depends on the arrival rate of transactions and the processing rate of transactions by the system. This is not considered to be a significant overhead and optimum searching techniques can be used to reduce the local processing delay.

The MET is a table which stores, either the edges, or the information concerning recently committed transactions. It does not occupy much space. It is deleted if corresponding transaction returns from the majority visit.

The TDFG is a one level graph structure. In this graph, each node represents a transaction. The information in the TDFG is not much as compared to the information carried in the case of voting based approach in the process of visiting majority of sites.

The following diagram (figure 4.13) shows the different states of processing for the approach.

```
Figure 4.13. State diagram of transaction processing.
```
As shown in the diagram, the transaction is submitted to a site (S1), it generates initial update values (IUVs) and prepares a TR. After visiting S2, S3,...,SM, TR returns to S1 with its TDFG. If no conflict is found, then it commits the update values (IUVs). Otherwise, if TDFG contains odd edges, the odd edges are confirmed. Further, subsequent to the receipt of update values for transactions that precede Tj in execution, the Tj is reexecuted. We assume that new update values are generated within the scope of the TVs, for which consensus has been obtained. The transaction commits its updates. However, in case of transactions that do not permit evaluation of final TVs at the time of initial update value computation, and if the final update values lead to a change in TVs, the transaction will require a resubmission for obtaining a consensus. Such a transaction must be declared as an aborted transaction (null values for updates) to allow other waiting transactions to proceed.

4.9.1. Odd edge confirmation

The TRj request, on completion of majority, may be made to visit additional sites, to obtain a majority with enough even edges, or add an enroute visit to odd edge elimination site (site SHj due to odd edge <Tj, Tj>).

Alternatively, as a part of implementation convenience, the odd edge elimination message 'DELETE<Tj,Tj>' can be part of the pre-commit message of the commit protocol [GRA78]. This is so, because the response to DELETE<Tj,Tj>, is based on a quick calculation. If the response is YES<Tj, Tj>, the transaction proceeds with its commit processing. Otherwise, the transaction can abort the pre-commit stage.
4.9.2. Choice of communication medium

The communication medium can be one based on the broadcast based communication. In this environment, a TR\textsubscript{j} will be sent to all sites. The response is to be evaluated by the home site to form the TDFG\textsubscript{j}. If the sites follow a first-come first-served discipline to process TR\textsubscript{j} requests, it can lead to reduced possibilities of odd edge introduction. There will not be any necessity for elimination of odd edges, if the above discipline is used to serve message buffers at local sites. This will also result in a higher level of failure resilience.

4.10. Conclusions

In the existing voting based approaches (also quorum based approaches), the update request is executed, and the update variables are sent to other sites for majority approval. In these approaches, if the submitted transactions are in serializability conflict, then some of the transactions are rejected. These transactions are resubmitted for execution, and by this, these incur additional processing delays and overheads. We have proposed a technique that generates a partial data flow graph (TDFG) on visits to sites by a transaction. These TDFGs are then used to achieve one-copy serializability.

In the proposed algorithm, the possibility of a transaction rejection, or abort, is removed, with one exception of an extreme case (see implementation issues above). On the whole, this algorithm works well in case of conflicts and a large number of copies with replication. It needs less number of messages than voting based approaches under conflicting situations. The algorithm is similar to the voting based approach. It is also robust to site crashes and network partitioning, in a similar manner. The transactions ignore the failed site
without stopping normal processing. In case of network partitioning, the partition with majority of sites active, can process transactions. In order to get benefits of weighted voting, the sites may be assigned different weights. In this way, the number of sites to be visited can vary [GIF79]. This can be used to reflect the needed accessibility level of a site.

In case of a conflict among 'n' transactions as \( T_1, T_2, \ldots, T_n \) (assumed serial order as: \(<T_1, T_2, \ldots, T_n>\) ), transaction \( T_1 \) commits on its return. Transaction \( T_2 \) must compute again. On completion, its TVs are assumed to be same as earlier (either these do not change or the final TVs are assumed to be computed at the first instant for voting) and its update values are committed. In this way, the drawback of the voting based approach, such as repeated roll-back, is eliminated. The algorithm provides a frame work for more deterministic estimates for processing time-critical transactions.

After presenting the approaches for concurrency control in distributed and replicated databases based on data flow graphs, in the next chapter, we examine the commit processing in the data flow graph environment.