CHAPTER 2

THE SYSTEM MODEL, CORRECTNESS CRITERIA, AND DATA FLOW GRAPHS

In this chapter, an outline of the serializability considerations for distributed, and replicated databases has been studied. After this, we describe a model of time, within which we explain the process of assigning unique numbers to the transactions in DDBS. These unique numbers are used in describing the algorithms in the coming chapters. Next, we present a concurrency control approach based on data flow graphs for centralized database system [EIC85].

2.1. The transaction model

A database is a collection of named data items. A database is in a consistent state, if it obeys all of the consistency constraints (integrity) defined over it. State changes occur due to updates, insertions and deletions. Due to these state changes, database should not enter into an inconsistent state. The database can be temporarily inconsistent during the execution of a transaction. The database should be consistent when the transaction terminates. Also, the database should be in a consistent state even if there are a number of user requests that are concurrently accessing (reading or updating) the database. Transaction management deals with the problems of keeping the database in a consistent state even when concurrent accesses
and failures occur.

The concept of a transaction is used within the database domain as a basic unit of (consistent and reliable) computation [GRA78, SPE83]. If the database was consistent before the execution of the transaction, it will be consistent at the end of an execution, regardless of the fact that, (1) the transaction may have been executed concurrently with others, and (2) failures may have occurred during the execution.

The consistency and reliability aspects of transactions are due to four properties: (1) atomicity, (2) consistency, (3) isolation, and (4) durability [OZS91]. Together, these are commonly referred to as the "ACIDity" of transactions.

The *atomicity* property of a transaction implies that it will run to a completion as an indivisible unit. That is at the end of the transaction, either no changes have occurred to the database or the database has been changed in a consistent manner. At the end of the transaction, the updates made by the transaction will be accessible to other transactions and the processes outside the transaction.

The *consistency* property of a transaction implies that if the database was in a consistent state before the start of a transaction, then on termination of a transaction the database will also be in a consistent state.

The *isolation* property of a transaction indicates that actions performed by a transaction will be isolated or hidden from outside the transaction until the transaction terminates. The property gives the transaction a measure of relative independence.

The *durability* property of a transaction ensures that the commit action of a transaction, on its termination, will be reflected in the database. The permanence of the
communication of a transaction requires that any failures after the commit operation will not cause loss of the updates made by the transaction.

We define a transaction as a set of atomic operations on data items. An operation is either a read (returns the value of the data item), or a write (updates the item with a specified new value). For simplicity, we assume that every transaction can read and write on any data item at most once. Then, for any $T_i$ and data item $X$, $r_i[X]$ denotes a read executed by $T_i$ on $X$. Similarly, $w_i[X]$ denotes a write executed by $T_i$ on $X$. The notation $O_i$ denotes an operation of transaction $T_i$, i.e., either $r_i$ or $w_i$. In general, a transaction does not have to be a totally ordered sequence. When two operations are not ordered relative to each other, these can be executed in any order. However a read and a write on the same element, must be ordered.

**Definition 2.1:** Two operations $O_i[X]$ and $O_j[X]$ conflict with each other, if they operate on data item $(X)$ and at least one of them is a write.

**Definition 2.2:** A transaction $T_i$ is a partially ordered set (see appendix A) with ordering relation $<_i$ such that,

1. $T_i \subseteq \{r_i[X]: X \in D\} \cup \{w_i[X]: X \in D\}$
2. $\forall X \in D$, if $r_i[X] \in T_i$ and $w_i[X] \in T_i$, then, either $r_i[X] <_i w_i[X]$ or $w_i[X] <_i r_i[X]$.

The items to be locked by the transaction for the purpose of read and write steps are termed as read-set (RS) and write-set (WS), respectively. The union of the read set and the
write set of a transaction $T_i$ constitute the locking variables (LVs$_i$). 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. Also $T_i, T_j$ are said to be in conflict, if at least one of the above conflicts exists.

2.2. A distributed database model

A distributed database is a collection of multiple, logically interrelated databases distributed over a computer network. A distributed database management system is defined as the software system that permits the management of the and makes the database distribution transparent to users [BER82, BER87].

The distributed database system (DDBS) consists of a set of data items (set $D$ (say)). A data item is the smallest accessible unit of data. It may be a file, a record, an array, a page, or an object. In this paper, $X, Y, Z$ represent data items. Each data item is stored at one site only. Transactions are represented by $T_i, T_j, \ldots$; and sites are represented by $S_i, S_j, \ldots$; where, $i, j, \ldots$ are integer values. The data items are stored at database sites connected by a computer network. Each site has a system wide unique identifier. Each site of a DDBS runs one or more of the following software modules: a transaction manager (TM), a data manager (DM) and a scheduler (Figure 2.1). Transactions communicate with TMs, TMs communicate with schedulers, schedulers communicate with DMs, and DMs manage data. (TMs do not communicate with other TMs, nor do DMs communicate with other DMs.) The TMs supervise the execution of the transactions, while the DMs manage individual databases. Each transaction issues all of its reads and writes to a single TM. Each transaction executed in the
DDBS is supervised by a single TM. The TM forwards each read and write of a transaction to the schedular. The schedular controls the order in which the DMs process reads and writes. The schedular depends on the concurrency control algorithm that is to be followed to order conflicting operations. The DM executes each read and write it receives.

Figure 2.1. DDBS architecture
The network is assumed to detect failures, as and when these occur. When a site fails, it simply stops running and other sites detect this fact. The communication medium is assumed to provide the facility of message transfer between sites. When a site has to send a message to some other site, it hands over the message to the communication medium, which delivers it to the destination site in finite time. We assume that, for any pair of sites \( S_i \) and \( S_j \), the communication medium always delivers the messages to \( S_j \) in the same order in which they were handed to the medium by \( S_i \).

2.3. Serializability in a distributed database system

The correctness of the concurrency control algorithms is tested by collection of mathematical rules which is termed as a serializability theory. In this section we give an overview of serializability theory in distributed database system [BER87, BER82, CER85].

Let \( T = \{ T_1, \ldots, T_n \} \) be a set of active transactions in a DDBS. When a set of transactions execute concurrently, there operations may be interleaved. We model such an execution by a structure called a history. A history indicates the order in which the operations of the transactions (committed) were executed relative to each other. Since some of these operations may be executed in parallel, a history is defined as a partial order. If a transaction \( T_i \) specifies the order of two of its operations, these two operations must appear in that order in any history that includes \( T_i \). In addition, a history specifies the order of all conflicting operations that appear in it.

**Definition 2.3:** A complete history \( H \) over \( T \) is a partial order with ordering relation \( \prec_H \).
where,

\[ H = \bigcup_{i=1}^{n} T_i; \]
\[ \preceq_H = \bigcup_{i=1}^{n} \preceq_i; \] and
\[ \text{for any conflicting operations } O_i, O_j \in H, \text{ either } O_i \preceq_H O_j \text{ or } O_j \preceq_H O_i. \]

Condition (1) says that the execution represented by \( H \) involves precisely the operations submitted by \( T_1, T_2, \ldots, T_n \). Condition (2) says that the execution honors all operation orderings specified within each transaction. Finally, condition (3) says that the ordering of every pair of conflicting operations is determined by \( \preceq_H \).

**Definition 2.4:** A serial history is a totally ordered history if for every pair of transactions \( T_i \) and \( T_j \), either all of \( T_i \)'s operations precede \( T_j \)'s operations or vice versa.

In a distributed database, each transaction performs operations at several sites. A sequence of operations performed by transactions at a site is a local history.

**Definition 2.5:** A local history of operations for a set of transactions \( T' \) (where \( T' \subseteq T \)), at \( S_i \), is serializable, if it is computationally equivalent to some serial history of such operations of \( T' \) at \( S_i \).
Definition 2.6: The execution of set $T$ ($\prec_T$), modeled by histories at sites $S_1, \ldots, S_m$ is correct (serializable), if there exist a partial ordering of conflicting operations $O_i$ and $O_j$. The $O_i \prec_H O_j$ in any history $(H)$ of $S_1, \ldots, S_m$, if and only if, $T_i \prec_T T_j$ in the total ordering.

We can determine whether a history $(H)$ is serializable by analyzing a graph derived from the history called a serialization graph (SG) for $H$ denoted as $SG(H)$.

Definition 2.7: A serialization graph is a directed graph whose nodes are the transactions in $T$ that are committed in $H$ and whose edges are all $<T_i, T_j>$ ($i \neq j$) such that one of the $T_i$'s operations precedes and conflicts with one of the $T_j$'s operations in $H$.

Theorem 2.1: A history $H$ is serializable if and only if $SG(H)$ is acyclic [BER87].

2.4. Serializability in a replicated database system

In this section, we outline the serializability theory for replicated data [BER87, BER85].

A replicated database is a distributed database in which multiple copies of some data items are stored at multiple sites. The main goal is to improve system reliability. By storing critical data at multiple sites, the system can operate even though some sites have failed.

For a replicated database system, it must behave like a database system managing a one-copy database insofar as users can tell. In a one-copy database, users expect the interleaved execution of their transactions to be equivalent to a serial execution of those
transactions. Since replicated data should be transparent to them, they would like the interleaved execution of their transactions on a replicated database to be equivalent to a serial execution of those transactions on a one-copy database. Such executions are called one-copy serializable. This is a goal of concurrency control for replicated databases.

To execute a transaction on replicated data, we must translate every operation on a data item into an operation on one copy (in the case of read), or several copies (in the case of write). Let $h$ be a translation function, i.e., $h(r_i[X]) = \{r_i[X_j]\}$ and $h(w_i[X]) = \{w_i[X_{j1}], \ldots, w_i[X_{jr}]\}$, where $X_{j1}, X_{j2}, \ldots, X_{jr}$ are copies of $X$. Thus the translation of given operation on a data item is a set of operations on copies (a singleton in the case of read). To simplify notations, let $h(T_i)$ denote the union of translations of all operations in $T_i$. That is,

$$h(T_i) = \bigcup_{O_i[X] \in T_i} h(O_i[X])$$

$$h(T) = \bigcup_{T_i \in T} h(T_i)$$

**Definition 2.8:** Two operations $O_i[X_k]$ and $O_j[X_k]$ conflict with each other if they are on the same copy of a data item ($X_k$) and at least one of them is a write.

**Definition 2.9:** A replicated data history $H$ over a set of transactions $T$ is a partial order with ordering relation $\prec_H$ such that:

1. $H = h(T)$.

2. For any $T_i \in T$ and all $X, Y$ such that $r_i[X], w_i[Y] \in T_i$, if $r_i[X] \prec_i w_i[Y]$ and $r_i[X_j], w_i[Y_k] \in H$, then $r_i[X_j] \prec_H w_i[Y_k]$; and

3. All pairs of conflicting operations in $H$ are ordered by $\prec_H$.  

27
Informally, a (replicated data) history is a partially ordered set of operations on copies, that is consistent with the order of operations on copies within transactions (condition(2)). In addition, since a history is intended to model a particular execution of a set of transactions on replicated data, it should order all conflicting operations (condition(3)).

**Definition 2.10:** Given a replicated data history $H$ over a set of transactions $T$ and transactions $T_i, T_j$ belongs to $T$, we say that $T_i$ reads-X-from $T_j$ if $w_j[X_m] \in H$, $r_i[X_m] \in H$, $w_j[X_m] < H r_i[X_m]$, and there exists no $k$, such that, $w_j[X_m] < H w_k[X_m] < H w_j[X_m]$.

**Definition 2.11:** Two replicated data histories over transactions $T_1, ... , T_m$ are said to be equivalent if they have same read-from's; i.e., if $T_j$ reads $X$ from $T_i$ in one history, then this relationship holds in the other history as well.

**Definition 2.12:** A serial history $H$ is a totally ordered replicated data history, such that, for every pair of transactions $T_i$ and $T_j$ in $H$, either all of $T_i$’s operations precede all of $T_j$’s operations or vice versa.

**Definition 2.13:** A serial history is a one-copy serial history, if for all $i, j$ and $X$ in $H$, if $T_j$ reads-X-from $T_i$, then $T_i$ is the last transaction before $T_i$ to write into any copy of $X$.

**Definition 2.14:** We say that a replicated data history is one-copy serializable, if it is equivalent to a one-copy serial history.
Thus, in this model, an execution of transactions is correct, if it is equivalent to a serial execution of the same transactions on a single copy database. Therefore, one-copy serializable histories hide all aspects of data replication from user transactions and give a transactions one-copy view of the database.

To test for one-copy serializability of a history, one usually makes use of a replicated data serialization graph, defined as follows.

Let $v$ and $w$ be nodes of a directed graph $G$. If there is a path from $v$ to $w$ we denote this fact by $v \rightarrow \rightarrow w$.

**Definition 2.15:** A replicated data serialization graph (RDSG) for history $H$, is a directed graph whose nodes are transactions in $T$ and whose (directed) edges constitute a superset of edges of $SG(H)$, such that, for all $X \in D$ the following conditions hold:

1. if $w_i[X] \in T_i$ and $w_j[X] \in T_j$, then either $T_i \rightarrow \rightarrow T_j$, or $T_j \rightarrow \rightarrow T_i$;

2. if $T_j$ reads-$X$-from $T_i$, $w_k[X] \in T_k$ for some $k$ ($k \neq i, k \neq j$), and $T_i \rightarrow \rightarrow T_k$, then $T_j \rightarrow \rightarrow T_k$.

**Theorem 2.2** [BER87]: Let $H$ be a replicated data history. If $H$ has an acyclic RDSG, then $H$ is one-copy serializable.

**2.5. Time-stamps and Transaction numbers.**

This section presents the notion of time, time-stamps, transaction numbers and ordering of events in a distributed system [LAM78].
A distributed system consists of a collection of distinct processes, which are spatially separated and which communicate with one another by exchanging messages. In such systems, the message transmission delay is not negligible. In a distributed system, it is sometimes necessary to know, if an event $e_i$ at some site happened before or after, an event $e_j$ at a different site. Determining the order of events is simple in a centralized system, since it is possible to use the same clock to determine the time at which an event occurs. Instead in a distributed system, it is not realistic to assume that perfectly synchronized clocks are available at all sites.

Several distributed concurrency control and deadlock prevention algorithms need the determination of an ordering of events. The determination of an ordering of events consists in assigning to each event $e_j$, which occurs in the distributed system a time-stamp $TS(e_j)$ having the following properties:

1. $TS(e_i)$ uniquely identifies $e_i$ (i.e., different events have different time-stamps):

2. For any two events $e_i$ and $e_j$, if $e_i$ occurred before $e_j$, then $TS(e_i) < TS(e_j)$.

The main inconvenience of above definition is that the meaning of the relationship "occurred before" is not precisely defined. Because, if the two events $e_i$ and $e_j$ occurred at two different sites, we do not possess a "global clock" for measuring the exact time of occurrence of events in the distributed system.

A precise definition of occurred before relationship in a distributed system is the following. Assume that we know the meaning of the statement "event $e_i$ occurred before event
e" at site S_i", i.e., we know the meaning of time ordering at a single site. Let E={e_1, e_2,...} be the set of all events in distributed system. The relation occurred before, denoted E, can be generalized to a distributed environment by the following rules:

1. If e_i and e_j are two events at the same site and e_i occurred before e_j then e_i E e_j.
2. If the even e_i consists in sending a message and event e_j consists in receiving the same message, then e_i E e_j.
3. if e_i E e_j and e_j E e_k, then e_i E e_k.

The relation E is a partial ordering. The second property of the definition of time-stamps can therefore be restated as follows.

2. For any two events e_i and e_j, if e_i E e_j, then TS(e_i) < TS(e_j)

There are number of ways that time-stamps can be assigned. One method is to use a global (system wide) monotonically increasing counter. However, the maintenance of global counters is a problem in distributed systems. Therefore, it is preferable that each site (autonomously) assign time stamps based on its local counter. To maintain uniqueness, each site appends its own identifier to the counter value. Thus the time-stamp is a two-tuple of the form "<site identifier, local counter value>". If each system can access its own system clock, it is possible to use system clock values instead of counter values.

It is not difficult to ensure that the time-stamps are unique. A site need only take care to never assign the same clock (counter) value to time-stamps for different transactions.
However, in this study, the concept of transaction numbers is used. Because, while in the process of synchronizing the clocks of different sites, the time-stamp of a request which visits more than one site, may not be the same value (but it is a unique value). So, we add another field to the time-stamp (I-field), which denotes the unique identification number, which is used for constructing the graphs. The process of assigning and comparing transaction numbers is explained below.

2.5.1 Transaction Number (TN)

Every site $S_i$ has a logical clock $C_i$, which takes a monotonically nondecreasing integer value [LAM78]. A TN is assigned to the transaction $T_i$, on its arrival, by a site $S_i$. It is triple element value, as $(S_i, I_i, C_i)$. $S$ is a site identifier ($S_i$). The I field is the transaction identifier, which is the value of the local clock ($C_i$) at the instant of $T_i$'s arrival at site $S_i$. The C field is used to synchronize different clocks, in a manner described below.

For any transaction message, $T(S_i, I_i, C_i)$ being sent from site $S_i$ to site $S_j$:

- $C_i = C_i$ of the message sender site.
- $C_j$ of the site $S_j$ on receipt of a transaction $T(message)$
  
  $C_j = \max(C+1, C_j)$.
- $C_j$ of the site $S_j$ before dispatch of a transaction $T(message)$
  
  $C_j = C_j + 1$.

Let, $TN_1 = (S_1, I_1, C_1)$ and $TN_2 = (S_2, I_2, C_2)$, then any pair of TNs can be compared using the following criteria.

Equal to ($\equiv$) : $TN_1 = TN_2$, if and only if, $S_1 = S_2$ and $I_1 = I_2$;
In the algorithms proposed in the subsequent chapters, the transaction number is assigned to the transaction by its home site on its arrival. The algorithms use transaction numbers for constructing data flow graphs of respective transactions. The data flow graphs are used to order the transactions.

2.6. Graph directed locking

The application of graphs for concurrency control has been first studied by Eich [EIC88, EIC88a] for centralized database system. Although two-phase locking is extremely simple in concept and implementation, it suffers from the problem of deadlock occurrence and reduced concurrency because of its two-phase requirement [EIC90]. Graph directed locking is developed to overcome these problems. This approach uses data flow graphs to direct the progression of locks between transactions and yields a concurrency control technique which is consistent, serializable, deadlock free, and not two-phase. In this section, the approach has been examined for its implementation in a distributed system environment.

This approach assumes availability of a predeclared read-set and write-set. After getting an access to all data items the transaction starts execution. A graph is constructed to access the database. It is called database flow graph (DBFG). A DBFG is an acyclic graph used to show the data dependencies between database operations (both intratransaction & inter-transaction) and is an extension of normal data flow graphs. The dependencies shown
in the DBFG define a multiple transaction schedule of database operations which is serializable. When a transaction enters a system, utilizing DBFG scheduling the query parse tree associated with the transaction must be merged with the existing DBFG. Likewise when a transaction terminates, it must be removed from the DBFG.

In this technique, nodes in the DBFG either represent a transaction or a data item. There is one source node in the relation and one transaction node per transaction in the system. Transaction nodes are never source nodes. When a transaction is requested (terminated) the associated graph node is added (removed) from the DBFG. each arc \(<T_1,T_2>\) is associated with some data item \(X\), and implies that the equivalent serial order in which these transactions will see data item \(X\) is \(T_1\) then \(T_2\). DBFG locking enforces this order unless both \(T_1\) and \(T_2\) require read only access to \(X\). In this case, either one of the order is acceptable.

Figure 2.2. Database flow graph
For each transaction, there is one input arc and one output arc per data item seen by the transaction. Following the arcs for a data item X from associated source node, a directed X path is defined. Figure 2.2, illustrates a DBFG for three data items (X,Y,Z) and four transactions (T₁,T₂,T₃,T₄). Arcs are labeled with the data item name. The directed X path is <T₁,T₂,T₄>, directed Y path is <T₁,T₂>, and directed Z path is <T₂,T₃,T₄>. The equivalent serial order obtained is, thus <T₁,T₂,T₃,T₄>. Figure 2.3 shows the resulting DBFG when a transaction T₅ using data items Y,Z is merged into the DBFG. During system execution, granting of locks must be such that this desired order of lock transition is guaranteed.
There are three types of activities that are performed as a part of the graph directed locking protocol:

1) When a transaction requests data items, MERGE a transaction node into the DBFG.

2) During the transaction execution, perform locks, un locks, and lock upgrades as requested. Granting of lock requests is made, based upon the status of locks already granted and the structure of the DBFG.

3) When a transaction commits, remove it from the DBFG.

2.7. Conclusions

In this chapter, at first, we have presented the transaction model and the distributed database model. After that, the criteria of serializability for distributed and replicated databases has been presented. The concept of time-stamps, transaction numbers further described is presented in detail. At the end, concurrency control algorithm based on data flow graphs is presented. In the next chapter, based on the foundations of serializability theory and data flow graphs presented in this chapter, we proceed to present the concurrency control algorithm for distributed databases.