Atomic Transactions in Distributed Systems

Gregor Gabriel

Institute of Informatics and Applied Mathematics
University of Berne

May 26, 2002

*Zaehringerstr. 55, CH-3012 Bern, ggabriel@student.unibe.ch, Matrikelnr. 99-103-194
## Contents

1 Introduction 3

2 The Transaction Model 3
   2.1 Synchronization Problem 4
   2.2 Recovery Problem 5

3 The Implementation 5
   3.1 Private Workspace 6
   3.2 Writehead Log 6
   3.3 Commit Protocol 7

4 The Concurrency Control 10
   4.1 Locking Protocol 11
   4.2 Optimistic Concurrency Control 14
   4.3 Timestamping 15

5 Concluding Remarks 17
Abstract

The present paper focusses on the applicability of atomic transactions to distributed computer systems. The theory of the transaction model derived from the business world in reference to operations on databases serves for the implementation of atomic transactions in distributed systems. Distributed systems are potentially very reliable. This is because they provide redundant resources on different machines. But performing atomic transactions, obeying the ACID-rules, the demands of a distributed system on synchronization causes problems. Thus the atomic transaction model for a monolithic system has to be modified for a distributed system. A simple way to guarantee data consistency is achieved by providing a private workspace to each transaction. But the most widespread implementation technique to enable atomic transactions is the atomic commit protocol, especially the two-phase commit (2PC). The 2PC provides well-known recovery from failure strategies and it fits the concurrency demands of a distributed system. To ensure concurrency control a distributed system is based on transaction managers. These managers can make use of different concurrency control mechanisms as there are the locking protocol, optimistic concurrency control and timestamping to enable concurrent transactions. Therefore, the concept of atomic transaction with its several approaches to implementation offers a promising technique for building reliable distributed systems.
1 Introduction

Distributed computer systems are potentially very reliable. This is because of the possibility of providing redundant resources on different nodes. However, the distribution of redundant resources also generates some serious problems, such as the lack of global state information, the possibility of partial failure, and the performing of many operations in parallel. For instance there may be several sites participating in the execution of several operations called transactions. The failure of one of these sites, or the failure of a communication link connecting these sites, may result in erroneous computations. These problems imply that it is difficult to maintain data consistency. To overcome these problems the atomic transaction model was invented.

First it is to define what an atomic transaction is. According to H.-J. Schneider [Sch97] a transaction is a sequence of DML-instructions (Data Manipulation Language) called by the user or the operating system by special instructions (e.g. begin-trans, end-trans). This sequence is defined as an atomic sequence. It is running independently and complete by turning the database from a consistent state $S$ in a consistent state $S'$.

The original model of the atomic transaction comes from the world of business: Suppose two companies like to trade with each other. Both sides have their business interests. They will start to bargain until they achieved an agreement in form of a contract. Up until this point, both parties are free to terminate the discussion, in which case the world returns to the state it was in before they started talking. Once the contract is achieved, both companies are legally bound to it and the transaction must be carried out. The computer model is similar.

The concept is actually an outgrowth of the way computer worked in the 1960’s. Before there were disks and online databases, all files were kept on magnetic tape. Imagine a supermarket with an automated inventory system. Every day after closing, a computer run was made with two input tapes. The first one contained the complete inventory as of opening time that morning. The second one contained a list of the day’s updates: products sold to customers and products delivered by suppliers. The computer read both input tapes and produced a new master inventory tape. The great beauty of this scheme is that if a run failed for any reason, all the tape could be rewound and the job be restarted with no harm done. Primitive as it was the magnetic tape system had the all-or-nothing property of an atomic transaction. [Tan92]

In the first section the atomic transaction model will be discussed in detail (section 2). Then the focus will lie on the implementation of transactions in distributed systems (section 3). The following section deals with special problems that arise if there are multiple transactions executed simultaneously and how they will be overcome (section 4). Finally, a short resume will round the paper (section 5).

2 The Transaction Model

As mentioned above the transaction model originates in the database research. A fundamental aspect is the atomicity of a transaction. An atomic action has four equivalent properties.
• Its process is not aware of any other active process or processes and no other process is aware of its activities.

• Its process doesn’t communicate with other processes during its activities.

• Its process cannot detect any state changes except those performed by itself and will not show any state changes until the action is completed.

• It is considered to be indivisible and instantaneous, so far other processes are concerned.

These properties must be viewed as relative rather than absolute [Gos91].

A transaction has to fulfil another four properties. The so called ACID-rules [Hal96].

atomicity changes on the database are atomic.

consistency the database is transferred from a consistent state in another consistent state.

isolation intermediate results associated with the operations are not accessible from outside. And if there are concurrent transactions, they are controlled as if they were performed sequentially.

durability correct changes to the database are permanent.

Applying these demands on atomic transactions to operating systems two major features can be outlined. The transaction has to lock an object before accessing it and it has to unlock the object before it completes. These two aspects are important for synchronization and recovery of the data.

In distributed systems the demands on synchronization is very high and the recovery of the data has to be managed by the system. Thus the atomic transaction model has to be modified to cope the requirements of a distributed environment. A synchronization problem arose with nested transactions, long term transactions and operations on user defined abstract data types. And the recovery problem deals with the concepts of reconstruction and the physical side where data is stored.

2.1 Synchronization Problem

In distributed systems it is probable that a transaction itself starts a new transaction. And this transaction also might start another transaction. So that the success of the root transaction depends on their successors success. These wrapping of sub-transaction is called nested transactions [Cro96]. Using nested transactions can improve the reliability of the conventional transaction system by allowing sub-transactions to restart after failures without aborting the parent transaction. However these nested transactions causes difficulties to synchronization mechanisms because of the higher degree of concurrency and it must be reflected in the fault tolerance of the system.

Some transactions need to be a long time active (long term transactions). This can cause problems within distributed systems because objects that are used by these transactions are locked for a long time. This may limit the performance of the system in terms of concurrency.
The system can also make use of semantic information about the type of objects and the operations to be performed (e.g., read and write locks). This might increase the concurrency again. Semantic information can also be used to permit concurrent access to data types whose behaviour does not lead to inconsistency (e.g., insertion of a new key into a directory service). [Gos91]

2.2 Recovery Problem

There are two major concepts how to keep track on the state during transactions.

- The **backward error recovery** means that when a transaction aborts, all the effects of the transaction must be undone and the system has to come back to the state it was before.

- The **forward error recovery** means that the system is brought up to the next consistent state by ensuring that all operations of the failed transaction are completed.

That brings up another demand on atomic transactions. The initial state of the modified objects must be reconstructible and the modified objects were not available to other objects during the transaction.

To determine how the system should ensure this the following subsection will take a closer look to the devices used for storing data.

Storage

There are three types of stores in computers. The volatile storage (e.g., RAM-memory) does not survive system crashes. The access to such data is extremely fast. The nonvolatile storage (e.g., disks) is somehow more resistant to CPU failures but it is susceptible to failures which may result in loss of data.

Finally the data on stable storage is ‘never’ lost. Here the data is replicated in two or more nonvolatile storage caches (disks) with independent failure modes [Sil97]. When a block is updated, it is first verified and updated on disk one and then on the other disk. When an update failed the two disks can be compared block by block and the old one can be replaced or if there is suddenly an inconsistency in a checksum of one block, it is easy to recover the data from the other disk. 

As a consequence of its implementation, stable storage is well suited to applications that require a high degree of fault tolerance, such as atomic transactions.

3 The Implementation

The transaction model seems to give a solution to inconsistency and serializability problems. But how can they be implemented? And are there further steps to be taken into account for distributed systems? It should be clear that each process executing a transaction just updating the objects it uses (e.g., files,

---

1For further details on replicated storage cf. T. Huber ‘Replication in Distributed Systems’. 5
data base records) in place, then transaction will not be atomic, and changes will not vanish if the transaction aborts. Clearly some other implementation methods are required. In the first part of this section two approaches to handle the consistency demand will be presented, followed by the protocols referring to them and their specific problems. As an example the 2PC protocol will be described more detailed.

3.1 Private Workspace

The most simplest and obvious way to guarantee the consistency of data is to give a process, when it starts the transactions, his private workspace. That means a transaction has his own copies of data (files) from the start until it commits or aborts, respectively. The problem with this technique is the cost of copying everything to a private workspace is prohibitive, but various optimisation are feasible.

A first optimization can be achieved by using information on use of the data. In other words if a transaction just have to read from a file and does not modify it, there is no need to copy it. So a pointer to the real data in the private workspace will serve the needs of this transaction. This optimization also works with nested transaction where a sub-transaction simple uses a pointer for its read-only data to its parents workspace. [Tan92]

A second optimization concerns the write accessed data of a transaction. Even here most of the copying can be avoided. Instead of copying the entire data, only the file’s index will be copied into the private workspace. When a block of the data is modified, a copy of the block is made and the address of the block will be inserted in the index. The block can then be updated without affecting the original and only the modified part of the data was copied. These modified blocks are called shadow blocks. [Tan92]

Only the process running the transaction can see the modified blocks, other processes can only see (and access) the original data. If the transaction aborts, following a backward error recovery approach, the private workspace simply will be deleted. When the transaction commits the private indices are moved into the original file index.

This design allows some kind of version management where one version per outstanding transaction is kept. This is useful for nested transaction where a version manager ensures that changes are only reflected in the stable storage after all sub-transactions commit.

Another aspect especially to distributed systems is a higher amount of failure modes then in centralized ones because not only a single machine might crash even the file server might not be reachable. So a copy in the private workspace of a process could stabilize the system again as long as the file server for instance is only temporary absent. [Cro96]

3.2 Writehead Log

Another method of implementing transactions is the writehead log. Every transaction has to announce its changes to data to an intention list stored at stable storage. There it must declare which files it want to access, what the data is by now and will be after. Only if the log has been successfully written, the changes will be made to the data.
By using a writehead log we have two possible error recovery strategies when a system crash occurs during a transaction sequence. If there is a difference between the writeheads log and the corresponding files after recovery of the system it is possible to detect whether the transaction already has committed or not. Then it is either possible to rollback (rollback error recovery) the whole transaction step by step or to fulfil the transaction (forward error recovery) according to the log (cf. sec. 2.2). These two possibilities are no real alternative choices because they do depend on the actual state in which the transaction was interrupted when the system crash occurred. At the subsection of failures for the 2PC protocol the two error recovery strategies will be discussed in particular at the example.

### 3.3 Commit Protocol

As mentioned above the action of committing a transaction must be an atomic operation. In distributed systems, all sites accessed by a transaction must coordinate their actions so that they unanimously commit or abort that transaction. This is achieved through an atomic commitment protocol launched at the end of the transaction. The current standard of these protocols is the two-phase commit protocol (2PC).

The underlying concept is a master and slave pattern. There must be a distributed transaction facility (also called distributed transaction processing unit (DTP) [Hal96]) that supports standard abstractions such as processes and interprocess communication as well as the execution of transactions and the implementation of objects on which operations can be performed. The distributed transaction facility meets the so called CCR\(^2\) concept [Gos91]. The transaction facility is the transaction coordinator - the master - which manages the transaction processing and termination. This unit is the one who triggers the atomic commitment protocol. All other participating sites of the transaction has to serve the role of slaves.

As it name indicates, the 2PC consists of two phases. In the first phase, called the voting phase, the participating sites are given a right to abort the transaction, called the veto right. In the second phase, called the decision phase, the participating sites have to agree on the same decision (commit or abort) given by the transaction coordinator. The decision phase ensures the atomicity of the transaction [Abd02].

In the voting phase the coordinator sends a request-for-vote message (or prepare message) to all the participating sites in the transaction. Each participant replies by sending its vote (ready or refuse). If a participating site votes ‘yes’, it enters a prepared state during which it can neither commit nor abort the transaction unless it receives the final decision from the coordinator. If a participating site votes ‘no’, it can abort its sub-transaction. If all participating sites have voted ‘yes’, the coordinators decision is commit. Otherwise the decision will be an abort and the whole transaction tree will be aborted. Figure 1 illustrates the 2PC algorithm in pseudocode.

There are several variations and optimisation to the 2PC protocol. For example one can reduce

---

\(^2\)CCR stands for Commitment, Concurrency and Recovery and describes the three fundamental operation of a higher level transaction. For more detail on the CCR concept it is referred to the book of A. Goscinski [Gos91]. In the section 2 the recovery and some aspects due to synchronization of the commitment were already discussed. The concurrency aspect is the topic of section 4.
begin; /*atomic action*/
send(request-for-vote_1);
send(request-for-vote_2);
...
send(request-for-vote_i);
recieve(request-for-vote_i);
...
send(request-for-vote_N);
prepare; /*prepare to commit*/
if (action can be performed)
  then begin
    lock object;
    store an initial state;
    store the request;
    ready(T); /*the i-th slave is able to do its work*/
  end;
else
  refuse;
if (all slaves sent ready(T))
  then begin
    commit(T);
  end;
else
  abort(T);
wait for response;
if (commit(T))
  then begin
    do work;
    unlock object;
  end;
send response;

Figure 1: Two-phase commit algorithm (2PC)

the costs associated with an abortion of the transaction. In the so called Presumed Abort protocol the coordinator does not log information nor wait for acknowledge regarding abort decisions. And no participating sites will acknowledge abort decisions. Another way is to reduce the costs of the committing transactions. The Presumed Commit protocol interprets missing information about transactions as commit decisions. But it has to force-write a membership log record that contains the identities of all participating sites in the transaction on stable store.

As a more fundamental variation to the 2PC the One-Phase Commit protocol (1PC) was proposed. The basic assumption underlying here is that the participating sites do not need to vote. This actually means that the coordinator of 1PC acts as a ‘nice dictator’ and makes sure that no participating site can have any reason to vote ‘no’. This is achieved by guaranteeing that the ACID properties of the local sub-transactions are already ensured before triggering the commit protocol. [Abd02]
Failure Handling in 2PC

Thus the 2PC protocol is the today’s standard of atomic commitment a more precise look on the failure types and the failure handling will be discussed. As mentioned before there is a greater range of failure sources in distributed systems as in monolithic ones. Thus the procedure will be given how failures are handled when a participating site causes a fault, the coordinator fails or the network is disturbed in the midst of the transaction.

This subsection is based on the textbook ‘Operating System Concepts’ of A. Silberschatz and P. B. Galvin [Sil97, chp. 18.3.2, pp. 571-573].

Failure of a Participating Site

When a participating site recovers from a failure, it must examine its log to determine the fate of those transactions that were in the midst of the execution when the failure occurred.

- The log contains a commit(T) record for a transaction T. The site will execute the transaction by a redo(T).
- The log contains an abort(T) record. The site will perform a undo(T).
- The log contains a ready(T) record. In this case the site has to consult its coordinator to determine the fate of the transaction T. The coordinator will inform the site if T has aborted or committed whereupon the site will execute an undo(T) or a redo(T). If the coordinator is down and cannot answer, the site will send a query-status(T) message to all the sites in the system to find out the status of the transaction T. Depending on the answers the site will perform an undo(T), redo(T) or even has to resend its request periodically if it didn’t got information about T. Note that at least the coordinator will have the required information.
- The log contains no control records concerning the transaction T. Because there is no entry concerning T, it is clear that the site failed before it responds to the prepare(T) message from the coordinator. So the coordinator never got a ready(T) message. It has to abort the transaction and will send an undo(T) to the site that is supposed to perform the undo(T).

Failure of the Coordinator

If the coordinator fails while executing the commit protocol for the transaction T, then the participating sites must decide on the fate of the transaction. In some cases a decision will not be possible for the sites. Thus they have to wait for the recovery of the coordinator.

- If an active site contains a commit(T) record in its log, then T must be committed.
- If an active site contains an abort(T) record in its log, then T must be aborted.
• If some active sites does not contain a \texttt{ready}(T) record in its log, Then the failed coordinator cannot have decided \texttt{commit}(T). But it is still possible that the coordinator has decided to \texttt{abort}(T), so the active sites have to wait for the recovery of the coordinator.

• If all active sites contain a \texttt{ready}(T) record in their logs, nevertheless it is not possible to determine wether the coordinator has failed a decision and if so, which one \texttt{commit}(T) or \texttt{abort}(T). So all active sites have to wait until the coordinator has recovered.

When the active sites have to wait for the coordinator to recover, the undecided transaction \( T \) may hold resources (e.g. by locking) because it still is in a critical section. As a result, some data are not available for all the active sites. The number of unavailable data will increase by downtime of the coordinator. This undesirable situation is called the blocking problem.

Failure of the Network

If a link of the distributed system fails, all the messages that are routed through that link cannot arrive their destination. The sites connected throughout that link will treat that absence of messages as a failure of the connected sites. Thus, the previous scenario will apply here as well.

If a number of links fail, the network might be divided in partitions. In this case the whole transaction sequence might be in one partition and will run unconcerned of that failure. Or the coordinator and the participating sites are spread in different partitions, but this will also lead to ‘normal’ link failure scenario for a single site.

4 The Concurrency Control

When multiple transactions in a distributed system perform their actions simultaneously, some mechanism is needed to prevent unwelcome interference of those transactions. These mechanism is called concurrency control algorithm that will be discussed in this section.

It is the function of the transaction manager also named distributed transaction processing unit (DTP) of a distributed system to manage the execution of transactions accessing data stored in a local site (e.g. locking manager, logical recovery unit, commit guards). This can either be a transaction executing their own site or a sub-transaction as a part of a global transaction executing several sites. Each transaction manager has to have records in a log for recovery purposes (see sec. 2.2) and for participating in an appropriate concurrency control mechanism to coordinate concurrent execution of the transactions taking place on its site.

By implementing those demands some properties for the concurrent control mechanism are to be emphasized [Gos91]. The concurrency control mechanism

• must ensure the preservation to consistency of data and

• must ensure the completion of atomic actions in finite time.
• It should show robustness to communication or link failures,
• should permit parallelism to satisfy system performance requirements,
• should have less computational and storage overhead,
• should work in a network environment with possible communication delays and
• should make few constraints on the structure of the atomic actions.

There are several approaches in solving the concurrency control problem in distributed systems. In this part the mechanism known as locking, the optimistic concurrency control algorithm and the method with timestamps will be discussed.

4.1 Locking Protocol

The oldest and most widely used concurrency control approach is locking. The idea of the locking protocol is most simple illustrated by an example: If one wants to stop two people using a single room at the same time, an obvious technique is to provide the room with a lock, and the people with a single key. Locks in concurrent systems are a logical extension of this. An transaction or an atomic action may lock data to ensure their inaccessibility during a temporarily inconsistent state. Locking can be done by using a single centralized locking manager or with a local locking manager on each machine of the distributed system for managing the local sites. The locking manager maintains a list of all locked files, and rejects all attempts to lock a data that is already locked by another process. Setting a lock will guarantee that no other process will change the data during the transaction. This implies that other transactions that want to access the locked data have to wait, abort or preempt the existing transaction. But when two transaction or more are waiting, they may be deadlocked\textsuperscript{3}.

To achieve transaction synchronization with a locking protocol, all transactions must be well-formed. According to A. Goscinski [Gos91] a transaction is well-formed if it

• locks data before accessing,
• does not lock data which is already locked, and
• before it completes, unlocks each data it locked.

A pseudocode for a well-formed, two-phase transaction is illustrated in Figure 2 which shows explicit lock, unlock, read, write operations on two objects $A$ and $B$. This figure is drawn from A. Goscinski [Gos91, chp. 6.8.2, fig. 6.18, p. 274]

To maximize the concurrency in the system, these essential properties should be intensified. First locks have to apply to the smallest granularity of data sensible. For instance, if there are two processes on the same file but accessing different records, they should not be excluded. And second,

\textsuperscript{3}Deadlocks are caused by at least two concurrent processes acquiring resources that the other process requires; in this case the resources is the lock. Deadlock can be dealt with either deadlock avoidance or deadlock detection.
Transaction T

begin-trans
  lock object A (account A)
  read object A (account A obtaining A_Balance)
  lock object B (account B)
  read object B (account B obtaining B_Balance)
  write object A (A_Balance - 100 CHF to account A)
  unlock object A (account A)
  write object B (B_Balance + 100 CHF to account B)
  unlock object B (account B)
end-trans;

Figure 2: Well-formed, two-phase transaction

Locks should be released as early as possible. A lock for a reading access could be released as soon as the appropriate data has been read, but a lock for a writing operation must be kept until the entire transaction invoking the writing has committed [Cro96]. Otherwise, other transactions could be affected by an intermediate state of data. Therefore, the locking mechanism is improved by dividing the locks into two classes:

- **Shared read locks** are used to exclude updates but allow shared read-only access to data from a site where a transaction performs its actions.

- **Exclusive write locks** exclude all other accesses and are used for write access to data from a site of a transaction.

Acquiring and releasing locks precisely at the moment they are needed or no longer needed, respectively, can lead to inconsistency or deadlocks. To avoid these deadlocks locks for the same object (e.g. data, record, file) should always be acquired in the same canonical order. An ordering of locks is part of deadlock avoidance\(^4\). [Cro96]

To improve performance usually a technique called **two-phase locking** is used. Here, as shown in Figure 3 drawn from A. Tanenbaum [Tan92, chp. 11.4.4, fig. 11-22, p. 495], the process first acquires all the locks it needs during the transaction (growing phase) and releases them during the shrinking phase. Sticking close to the two-phase locking algorithm allows an easy way to handle failures (simply release all locks, wait a while and start all over again) and a high serializability of transactions and sub-transactions [Tan92]. In many systems the growing phase last until the transaction commits or aborts, only then the shrinking phase is triggered (**strikted two-phase locking**). The advantage of this technique is that a transaction always reads data written by a committed transaction. Thus, it must never be aborted because of data it was not allowed to see. And furthermore, all lock acquiring and releasing can be handled by the system without the transaction being aware of them. This prevents the system of **cascaded aborts** where committed transactions has to be rewind because it saw data it should not have seen [Tan92].

---

\(^4\)For further detail on deadlocks cf. P. Buchenel ‘Deadlocks in Distributed Systems’.
Until now, not much is said about the implementation of locks. As mentioned above, the distributed system could be implemented with local locking managers and no centralized locking manager. But this architecture only fits for non-replicated data systems. Here, a transaction that wants to access data just send its request to the local locking manager of that site where the data is stored. This locking manager now has to decide whether it can provide the lock and give way to the request or it has to delay the request until it could be granted. This mechanism is easy to implement and only a few messages are required to provide locking of data, but the costs for deadlock handling will increase compared to the centralized locking manager. [Sil97]

With the centralized locking manager approach all requests for locks will be maintained by a single locking manager. The scheme of granting locks is the same as the one with several local locking managers. The only difference lies in the updating of a committed write access. In this case the data is written to every site where a replica of the data is stored. This architecture has the same advantages as the one with distributed managers but the centralized locking manager is a bottleneck because all the message handling depends on it. Furthermore, if the site where the single locking manager is located fails, the concurrency control is lost. Either the processing must stop or a recovery protocol has to be run. [Sil97]

As an intermediate the multiple-coordinator approach was invented, in which the lock-manager function is split up on several sites. Each of them coordinating the locking for a subset of items [Sil97]. This approach increases the expense of deadlock handling, but reduces the bottleneck property.

The majority locking protocol is an outgrow of the non-replicated data scheme and the idea of the voting phase of the 2PC. Here, a lock can be provided if the majority of the locking managers that holds a replica of the requested data are able to grant a lock. These scheme reduces disadvantages of the centralized manager, but suffers from a complex implementation, a high expense on message

---

For further details on replicated data storage cf. T. Huber ‘Replication in Distributed Systems’
sending and going along with it a difficult deadlock handling.

The *biased protocol* tries to reduce the overhead in communication by handling the shared read locks and the exclusive write locks in a different manner. It prefers the shared locks by granting a lock only on one replica on one site of the requested data. The exclusive locks still has to get the lock on all sites where replicas of the data are stored. This is a suitable approach because usually read accesses are far more widespread than write accesses.

Finally there is the *primary copy* approach where a replica is chosen as the primary copy. Each primary copy must reside on the primary site. Now, the requests will be handled by the locking manager of the primary site. This approach again is close to the centralized locking manager approach.

Transactions and their locks may be timed out. This will lead to failures followed by abort and redo operations without guarantee to succeed this time. Suitably randomized timeouts may help to avoid this kind of deadlock. Deadlock detection in a distributed system is much harder because no single site can reliably hold the *wait-for-graph* for the whole system.\(^6\)

### 4.2 Optimistic Concurrency Control

The *optimistic concurrency control* differs from the use of locks fundamentally, in that it is based on the assumption that in many systems, the likelihood of two transactions accessing the same data is low. It was proposed by Kung and Robinson in 1981 as an alternative to the locking protocol with its disadvantages of high costs concerning implementation and maintenance and their susceptibility to deadlocks [Gos91]. The basic idea of the optimistic concurrency control algorithm is to go ahead with the things the process wants to do without caring for what others might do. If there emerges problems, they will be handled later. Therefore optimistic concurrency control avoids the expensive synchronisation and checking necessary for transaction systems.

Usually a distributed system is divided into a number of logical recovery units. Each of these units periodically checks its state, and continuously keeping track on data they have changed since the transaction started in a log to enable a potentially error recovery rollback. These logs consists of two sets of data items accessed within the transaction: a *read set* which contains items read by the transaction and a *write set* containing the items written, created or deleted by the transaction [Gos91]. When the transaction completes a second phase is started where the transaction is reviewed by checking the recovery units. The goal of the second phase is to determine whether or not the write operations could be made permanent. If the validation is successful, the write operations will take place and the transaction will end with a *commit* [Gos91]. A rollback is only necessary if it turns out one of the recovery units failed i.e. some data is found in an inconsistent state; the transaction ends with an *abort*. The technique to perform the validation is called *commit guards*. Commit guards are effectively the list of other recovery unit states that this transaction is dependent on [Cro96]. The recovery unit of a transaction is compared with the recovery units of all other concurrent transactions that reached the end of their first phase before it. A transaction then fails if the write set contains any element that is also in a checked recovery unit of another transaction [Gos91].

---

\(^6\)For further detail on deadlocks cf. P. Buchenel *Deadlocks in Distributed Systems*.
The advantage of the optimistic concurrency control approach is by splitting up the recovering problem in smaller pieces the costs of keeping enough information to carry out an error recovery rollback are not to high in any partition. Furthermore, it is deadlock free and allows a maximum parallelism because no process ever has to wait for a lock. But in some cases it fails and the transaction has to be rerun. And some overhead arises because information about old versions of files corresponding to concurrently committed transactions have to be stored by the transaction manager. Optimistic concurrency control seems to be best suitable for systems using a private workspace scheme (see section 3.1).

4.3 Timestamping

The final approach to concurrency control that will be discussed in this paper is the technique known as **timestamping**. Every file in the system will have a unique read and a unique write timestamp associated with it [Tan92]. The timestamp is a unique number with an increasing value every time a new stamp is given. This timestamps reveal which transaction last read or wrote the file, respectively.

In distributed systems the timestamps are usually generated by using the local time of the day concatenated with the local unique identifier of the participating machines [Gos91]. An other approach is to determine one machine that will provide unique timestamps [Sil97]. Instead of the time of the day a logical counter can also be used. Anyway, the order of the concatenation is important. The identifier must always be in the least significant position to guarantee that the global timestamps of one site are not always greater then those generated at other sites. If the timestamps are generated at different sites, it is possible that one local clock runs faster than the others. To enable a fair timestamping on each site a logical clock is defined to generate the unique local timestamps. The logical clock can be implemented as a counter that increases when a new timestamp is generated. To adjust a local logical clock it is updated at every visit of a transaction with a later timestamp on its local site [Sil97].

The **transaction manager** supervises the timestamp ordering and oversees the complete distributed computation. Placed under its control **object managers** perform actions on their respective objects (e.g. data, file). When a transaction is launched the transaction manager will attach a unique timestamp to it. Furthermore, it will assign the timestamp to all belonging actions to the object managers on the behalf of the transaction. Each object manager has a **timestamp ordering scheduler** that schedules all local actions according to their timestamps. [Gos91]

The steps of the timestamp ordering algorithm\(^7\) are as follows:

- The object manager evaluates the largest timestamp of all completed **read** \((t_{sr})\) and **write** action \((t_{sw})\) of each object.

- For a new **read** request on an object, the scheduler compares the incoming timestamp \((ts)\) with largest write timestamp \((t_{sw})\). If \(ts < t_{sw}\), the scheduler rejects the read access and the transaction manager has to **abort** the transaction (cf. Figure 4(a) and (b) modified after A.

\(^7\)By some others this scheme is also known as **conservative timestamp-ordering scheme** [Sil97].
Tanenbaum [Tan92, chp. 11.4.4, fig. 11-23, p. 497]). Otherwise, the scheduler allows the read access in a strict FIFO ordering and sets $ts_r := \max(ts, ts_r)$ (cf. Figure 4(c) and (d)).

- For an incoming write request the scheduler compares $ts$ with the maximum of $ts_w$ and $ts_r$. If $ts < \max(ts_w, ts_r)$, the scheduler rejects and the transaction manager performs an abort (cf. Figure 4(e) and (f)). Otherwise, the action obeying the FIFO principle takes place and $ts_w := ts$ (cf. Figure 4(g) and (h)).

- When the transaction was aborted, the transaction manager will restart it and allot a new (larger) timestamp to it.

![Figure 4: Timestamping](image)

But coming along with the timestamping some disadvantages must be mentioned. By rejecting and restarting at least two transactions by the transaction manager again and again a cyclic restart might emerge. And when the actual execution sequence is in a reverse order to timestamps a transaction might never get in and restarts again (unnecessary restart). Moreover, it could lead to a starvation-related problem, the infinite restart, when an older transaction, each time it restarts, finds it has to wait for a younger conflicting transaction [Gos91]. Because all transactions have unique timestamps, an ordering by timestamps is free of deadlocks. That is why it is willingly combined within systems using a locking protocol to avoid deadlocks. As a difference to the optimistic concurrency control it is unimportant if a concurrent transaction uses the same file, as long as the earlier timestamped transaction always goes first. Compared to locking, timestamping will have to abort a transaction when it encounters a later timestamped file, whereas the locking protocol has to wait or proceed immediately.

**Timestamping with 2PC**

To give one example how the presented techniques to serializability and concurrency control using atomic transactions can work hand in hand an example that combine the timestamping approach and
the two-phase commit protocol (see section 3.3) are presented. [Gos91]

As an extension of the 2PC a prewrite action is introduced which must be requested by the transaction manager before the write is called. Both actions will be assigned with the same timestamp. The effect is similar to an exclusive write lock (see section 4.1) where the object is locked during the transaction. Now, when the prewrite with the timestamp \( t_{sp} \) was accepted by the scheduler, neither a read nor a write action with a greater timestamp than \( t_{sp} \) must be rejected until the write (belonging to the prewrite) is output. This rule is implemented by buffering read, write, prewrite. These action are assumed to be indivisible operation as far as the local scheduler is concerned. The timestamps are initialized to zero, also the buffered ones.

Taken from A. Goscinski [Gos91] the following algorithm gives the working method of the timestamp ordering with two-phase commit:

- A read with timestamp \( ts \) is
  - rejected if \( ts < ts_w \).
  - handed over to the object manager if \( ts_w =\min(ts_p) \),
  - or if \( ts_w < \min(ts_p) \) (write phase) \& \( ts < \min(ts_p) \).
  - buffered if \( ts_w < \min(ts_p) \) \& \( ts > \min(ts_p) \).

- A prewrite with timestamp \( ts \) is rejected if \( ts < \max(ts_r, ts_w) \). Otherwise, it is buffered.

- A write never has to be rejected when the corresponding prewrite has been accepted. The request is buffered if \( ts > \min(ts_p) \). Otherwise, it is handed over to the object manager. In the second case the prewrite will be deleted from the buffer. If this causes an increase of \( \min(ts_p) \) value buffered, the buffered read and write requests are retested and eventually be handed over to the object manager.

This algorithm fits for only one prewrite operation. Normally a transaction has several write actions, so a lot of prewrites has to take place before the transaction can commit.

5 Concluding Remarks

It was started in describing the transaction model which is an outgrowth of the business world where databases are manipulated through atomic transactions following the so called ACID-rules. In monolithic computer systems the transaction model is easy to implement. By transferring it to distributed systems it is to be modified. It is to be ensured that the system stays in a consistent state or otherwise is set to a consistent state. Therefore, some synchronization techniques and error recovery methods were invented. The kind of storage that is used to keep the data is also an important factor for choosing the appropriate implementation to provide atomic transaction in distributed systems as there are private workspace and the commitment protocols. In detail the 2PC protocol was discussed and its failure handling strategies were explained.
Also the demands on concurrency control have been taken into account. How locks on data in a distributed system are provided by the so-called locking protocol was presented. The optimistic concurrency control approach was compared to the locking followed by the timestamping method. Because the 2PC is the most common technique in terms of data consistency and timestamping is for the concurrency, a closer look on the combination of these two was taken.

All in all, transactions offer many advantages and thus are a promising technique for building reliable distributed systems. Their chief problem is their great implementation complexity, which yields low performance. These problems are being worked on, and A. Tanenbaum [Tan92] hopes that in due course of time they will be solved. Furthermore it shows that the atomic transaction model is not only appropriate for monolithic computer systems but also it turned out that it is suitable to distributed systems.
References


