SHARED DATA AND TRANSACTIONS

12.1 Introduction
12.2 Conversations between a client and a server
12.3 Fault tolerance and recovery
12.4 Transactions
12.5 Nested transactions
12.6 Summary

This chapter discusses the design of server programs that manage data shared between clients. A server encapsulates resources that are accessed by means of operations invoked by clients. The only way for clients to access a server's resources is by invoking one of the server's operations.

The use of threads in a server offers concurrent access to clients. When a server has multiple threads, it must ensure that its operations are atomic in their effect on its data items. Some servers can enhance the cooperation between clients by arranging for one client to wait until another client has provided a resource needed by the first client.

In some cases, such as the transfer of a stream of data items to a client, the interaction between a client and a server is more like a conversation than a simple operation invocation.

When clients need to use a server to store data over a long period of time, the server must be designed to guarantee that the data will survive even when the server process fails.

Many applications require client transactions—a transaction defines a sequence of server operations that is guaranteed by the server to be atomic. The last section of this chapter introduces transactions—which guarantee atomicity in the presence of multiple clients and server failures.

Nesting allows transactions to be structured from sets of other transactions. Nested transactions are particularly useful in distributed systems because they allow additional concurrency.
12.1 Introduction

We have seen in Chapter 2 that a client is a process that initiates an activity, whereas the server is a process that waits for requests from clients and then performs whatever it is asked to do. This chapter and the three following chapters are concerned with issues relating to consistency and reliability in server programs and their clients. We have two contrasting requirements – on the one hand, clients' operations must be prevented from interfering with one another; on the other hand, clients should be able to use servers to share and exchange information. Some servers are required to maintain data on behalf of their clients for long periods of time – such servers should be based on recoverable data – that is data that can be recovered after the server crashes. In some applications, clients need to perform sequences of related server operations as indivisible units known as transactions. A service may be based on several servers with data partitioned between them – for example, a banking service may be based on a set of servers, each of which maintains the accounts for a single branch. A service based on several servers will need to deal with distributed transactions.

This chapter discusses issues of cooperation between clients and maintenance of long lived data in a single single process server and then introduces transactions. The provision for transactions in a server requires more advanced techniques to prevent interference between clients and to ensure the data is recoverable. Chapter 13 is concerned with concurrency control, Chapter 14 introduces distributed transactions and relates them to issues of concurrency control and replication. The first half of Chapter 15 is concerned with recoverable data in both single server and distributed transactions. The second half of Chapter 15 is concerned with other approaches to the provision of fault tolerant services.

We regard a server as a component of a distributed system that manages a single type of resource, as introduced in Chapter 2. The resources may be application-related such as email messages or bank-account records or they may be generic, such as files, printers or windows.

A server encapsulates the resources that it manages, allowing them to be created, accessed and manipulated by means of operations that can be invoked by clients. Chapter 5 we saw that the operations available to clients are defined in the server interface. The only way for clients to access a server's data items is by invoking one of the server's operations. A server process contains within it data items that store the state of its resources. The effect of a request to perform an operation on a server depends on the actual request, its arguments and the current values of the server's data items.

In general, the data items that represent the resources managed by a server may be stored in volatile memory (for example, RAM) or persistent memory (for example, disk). Even if they are stored in volatile memory, the server may use persistent memory to store sufficient information for the state of the resources to be recovered in case a server process fails.

The resources that a server manages are determined by the needs of its clients:

- a directory server might encapsulate the names, addresses and other details of a group of people and provide operations to look up, add or modify names and addresses;

- the resources of the binding service described in Chapter 5 consist of mappings from service names to service ports. When a server starts up it becomes a client of the binder and supplies a new mapping. Clients of a binder access its resources in order to find out the location of services;

- the resources of the Sun NIS service consist of mappings from keys to values. The maps can hold for example, password file information or host names and their internet addresses. Clients of NIS access its resources for example, when users login;

- a server could be defined to provide a diary database that can be viewed and updated interactively by a number of users. A user who wants to make an appointment with someone views that person's diary and writes in an appointment.

We will use a simple directory service called Address to illustrate some of the points made in this chapter. The Address service provides the following operations on a set of resources containing names and addresses:

- LookUp(name) \rightarrow address
  
  returns the address corresponding to the given name.

- AddAddress(entry)
  
  adds a new entry containing a name and address.

- ModifyAddress(name, address)
  
  modifies the address associated with a name.

- DeleteAddress(name)
  
  deletes the entry with the given name.

For example, a client may invoke the LookUp operation in a server of the name and address service named Address as follows:

- Address$LookUp("Napoleon");

We now return to the main issue of this chapter which is that unless a server is carefully designed, its operations performed on behalf of different clients may sometimes interfere with one another. Such interference may result in incorrect values in the data items.

**Atomic operations at the server** - We have seen in earlier chapters that the use of multiple threads is beneficial to performance in many servers. If a server has more than one thread, it needs to make its operations atomic in order to keep its data items consistent. This means that the effect of performing any single operation is free from interference from concurrent operations being performed in other threads.

For example, in the Address service it would be possible to perform two concurrent ModifyAddress requests on the same entry in a consistent manner. To perform each operation a thread may read and/or write the values of some of the data items. To ensure that the effect of each operation is atomic, the server must ensure that
no other thread can access the same data items until the first thread has finished. This may be achieved by the use of a mutual exclusion mechanism such as the mutex.

Enhancing client cooperation by synchronization of server operations. The client may use a server as a means of sharing some resources for example, names and addresses. This is achieved by some clients using operations to update the server resources and other clients using operations to access the resources. In some services, this situation may arise in which the operation requested by one client cannot be completed until an operation requested by another client has been performed. This can happen when some clients are producers and others are consumers - the consumers may have to wait until a producer has supplied some more of the commodity in question. It can also occur when clients are sharing a resource - clients needing the resource may have to wait for other clients to release it. We shall see in Chapter 13 that similar situations arise when locks or timestamps are used for concurrency control.

Consider the situation when a consumer client requests a resource that is not currently available. It is unsatisfactory to tell such a client to try again later if the client requires the requested resource to continue. It would involve the client in busy waiting and the server in extra requests (because the client will have to keep trying the request). It is also potentially unfair because other clients may make their requests before the waiting client tries again. A preferable solution is for the server to hold on to the request and the client to wait for a reply until another client has produced a resource. After that the consumer operation can proceed.

To deal with such situations a server must be able to suspend requests that cannot be executed immediately; to continue to receive other client requests and to resume suspended requests when they can be executed. To allow for the need to execute more than one client request at a time, a server uses a new thread to execute each request, possibly with some limit on the number of available threads. In some cases a thread may be suspended when it cannot usefully proceed - in our example when the required resource is not available - and later resumed. Chapter 6 discussed the provision of threads, condition variables and synchronization primitives in operating systems. A thread that cannot continue execution requests its own suspension by using the Suspend operation. Another thread can cause a suspended thread to resume by using the Suspend operation. For a comprehensive study of these and other issues in concurrent programming see Andrews [1991] and Bacon [1993].

12.2 Conversations between a client and a server

In some applications, a client request may cause a server to perform a lengthy calculation to produce multiple items of output gradually, for example, to request a database server to return all entries matching a particular key. It might be preferable for the server to be able to transmit them one by one to the client, allowing server and client to work concurrently.

As an example, consider the design of an operation in the name and address service for returning the details of all the names and addresses currently stored. If the server has a very large number of names and addresses, it is better for clients to get them one by one or in small batches, rather than receiving them all in a single message. This enables clients to process each name and address when it arrives.

The desired interaction between client and server can be regarded as a conversation in which the server keeps track of where a particular client has got to. To support conversations, a service will require two new operations for use by clients:

---

OpenConversation ( ) → conversationId
ask to start a conversation with the server - an identifier for the conversation is returned.

CloseConversation (conversationId)
indicate the end of a conversation.

A conversation is generally about the server resources. In our Addresses example, it is about the entries. The role of the server is to give the client each of the name and address entries in turn. It provides an operation for the client to request the next entry or batch:

---

NextAddress (conversationId) → entry — REPORTS EndOfSequence
returns the details of the next name and address; if there are no more entries, returns an error report EndOfSequence.

---

Thus a client might make the following requests to get details of all the names and addresses:

conversationId := Address$OpenConversation ();
REPEAT
entry := Address$NextAddress (conversationId);
UNTIL ErrorReport = EndOfSequence
Address$CloseConversation(conversationId)

---

Each conversation has a conversationId that is returned by the server when a conversation starts. The name and address service will be extended to hold for each client that is currently conversing with it, a variable containing the conversationId and a reference to the next name and address entry whose details are to be returned to that client. Each time OpenConversation is executed by the server, such a variable is allocated and when CloseConversation is called, the variable used by that conversation is freed.

A threaded server could use one thread for each conversation. This would simplify the programming because each thread would need to remember only its own position in the sequence of data items to be accessed.

Stateful servers. In the last example the server maintains information on behalf of each of its clients with whom it is currently conversing. The information consists of the conversation identifier and the next entry for each client. It may exhibit the problems associated with stateful servers - in that it is vulnerable to poorly designed or crashing
12.3 Fault tolerance and recovery

A fault tolerant server should be able to continue to provide a service in spite of processes crashing and the loss of messages.

In Chapter 4 we showed that a request-reply protocol may be selected to provide the desired level of tolerance to the loss of messages. This level will vary according to the RPC call semantics demanded by the particular server application. Consider an Address service: the data items can be arranged as set of entries, each member of which is indexed uniquely by name. The update operations can be designed to be repeatable and at-least-once call semantics are sufficient. In general, the use of idempotent operations in a server reduces the required RPC semantics to at-least-once.

A simple way of designing a server that can tolerate crashing clients is to design it to avoid holding information on behalf of particular clients. Servers designed on this principle are usually called stateless. However, simple stateless servers do not meet application requirements. We saw in the previous section that some servers can benefit from a design that enables them to hold conversations with clients. In Chapter 8 we saw that the Andrew file system uses a call-back mechanism to ensure that clients have to-date copies of files in their caches. This involves the servers in retaining state and which files each client currently has cached. The designers chose this solution preference to a stateless one because it reduced significantly the number of calls to servers. The last section of this chapter introduces servers that provide clients with the ability to use transactions – such servers need to hold quite complex information on behalf of their clients.

A fault tolerant server should be able to provide a service even if a server process crashes. To provide an apparently continuous service in the presence of a process crash, the service will be based on a group of replicas of the service running in different computers. The replicas will monitor one another and will be able to provide continuous service in the presence of the failure of a limited number of replicas. Techniques for fault tolerant services are discussed in Chapter 15 and methods of replication in Chapter 11.

**Recoverable data items**

A simple way to provide a service based on a service process that can recover from a crash is to keep the values of the data items represented in a network of storage that will survive a server crash for, for example, a fixed disk – called the recovery file. When such a server is restarted it will recover its items to the state before the crash by initializing them from the values in the recovery file. In order to make this signal agent to monitor the server and to report any failures. The service should guarantee that after a reply message has been sent to the client the effects of an update operation will remain permanent, even if the server crashes. This requires that after each operation is complete, all the changes to the values of the data items should be written to the recovery file before sending the reply message.

The techniques for organizing a recovery file for a simple server are simplified versions of the techniques used for transactions, which are discussed in Chapter 15.

12.4 Transactions

When we say a server provides atomic operations this means that the effect of performing any operation on behalf of one client is free from interference from operations being performed on behalf of other concurrent clients; and either an operation must be completed successfully or it must have no effect at all in the presence ofserver crashes. In some situations clients require that a sequence of separate requests to a server is atomic in the sense that the combined execution of the corresponding server operations is atomic. A conversation between a client and a server is one example of such a sequence of operations. Returning to our example, the client may wish to retrieve all of the name and address details from the address server without any other clients being allowed to update them (for example, by adding new ones or modifying existing ones) during the retrieval.

We shall use as another example a server that holds data for all of the accounts of a branch of a bank and provides operations to deposit or withdraw money in these accounts. This service called Bank provides the server operations Deposit, Withdraw, GetBalance and BranchTotal on a set of bank accounts.

- **Deposit**(Name, Amount)
  
  deposit amount Amount in account Name.

- **Withdraw**(Name, Amount)
  
  withdraw amount Amount from account Name.

- **GetBalance**(Name) → Amount
  
  return the balance of account Name.

- **BranchTotal**( ) → total
  
  return the sum of all the balances.

Each account has a name (A, B, C,...) used by the client – the server maps account names onto the corresponding data items holding account balances in dollars.

Consider a client that wishes to perform a series of related actions involving bank accounts called A, B, and C. The first action transfers $100 from A to B and the second transfers $200 from C to B. A client achieves a transfer operation by doing a withdrawal followed by a deposit as shown in Figure 12.1.

Transactions originate from the field of databases. In that context a transaction is an execution of a program that accesses a database. Transactions were introduced to...
Figure 12.1 A client’s banking transaction

Transaction T:
BankSWithdraw(A, 100);
BankSDeposit(B, 100);
BankSWithdraw(C, 200);
BankSDeposit(B, 200);

distributed systems in the form of transactional file servers such as XDFS [Mispagel and Dion 1982]. In the context of a transactional file server, a transaction is an execution of a sequence of client requests for file operations. Subsequently transactions are used in the context of servers of recoverable data as for example in the Argus [Williams and Argyris 1988] and Arjuna [Shrivastava et al. 1991] systems. In this last context a transaction consists of the execution of a sequence of client requests as for example in Figure 12.1. From the client’s point of view, a transaction is a sequence of operations that single step, transforming the server data from one consistent state to another.

In all of these contexts, a transaction applies to recoverable data and is intended to be atomic. It is often called an atomic transaction.

There are two aspects to atomicity:

- all-or-nothing: a transaction either completes successfully and the effects of its operations are recorded in the data items or (if it fails) it has no effect.

This all-or-nothing effect has two further aspects of its own:

- failure atomicity: the effects are atomic even when the server fails.

ACID properties: Hinder and Rauter [1982] suggest the main reasons to remember the properties of transactions as follows:

Atomicity: a transaction must be all-or-nothing.
Consistency: a transaction takes the system from one consistent state.
Isolation: the effects of the operations are independent.
Durability: the effects are atomic even when the server fails.

To support the requirement for failure atomicity and durability, the data items must be recoverable; when a server halts unexpectedly due to a hardware fault or a software error, the changes due to all completed transactions must be available in permanent storage so that the server can recover its data items to reflect the all-or-nothing effect. By the time the server acknowledges the completion of a client’s transaction, all of the transaction’s changes to the data items must have been recorded in permanent storage.

A server that supports transactions must synchronize the operations sufficiently to ensure that the isolation requirement is met. One way of doing this is to perform the transactions serially – one at a time in some arbitrary order. Unfortunately this solution would generally be unacceptable for servers whose resources are shared by multiple interactive users. In our banking example, it is desirable to allow several bank clerks to perform on-line banking transactions at the same time as one another.

The aim for any server that supports transactions is to maximise concurrency. Therefore transactions are allowed to execute concurrently if they would have the same effect as a serial execution – that is if they are they are serially equivalent.

A transactional service is an extension of a service such as the banking service, the name and address service or a file service to provide access to its resources via transactions. An atomic transaction is achieved by cooperation between a client program and a transactional service; the client specifies the sequence of operations that are to comprise a transaction and the transactional service guarantees to preserve the atomic property of the whole sequence. The client specifies a transaction as a sequence of operations, prefacing the sequence with an OpenTransaction operation to introduce each new transaction and concluding it with a CloseTransaction operation to indicate its end. See Figure 12.2 in which the operations are defined according to our usual notational convention (introduced in Chapter 7) and the following new argument name is used:

Trans: transaction identifiers or TIDs.

The client uses the transaction identifier returned by OpenTransaction to indicate which of the subsequent operations (up to the close of the transaction) are to be included in the particular transaction it identifies. See for example, the operations of a transactional file service in Figure 12.2.

Normally, a transactional service notes the start of each new transaction and performs the client’s requests until it receives a CloseTransaction request. If the transaction has progressed normally the server then reports to the client that the transaction is committed – this constitutes an undertaking by the service to the client that all of the changes requested in the transaction are permanently recorded and that any future transactions that access the same data will see the results of all of the changes made during the transaction.
Transactional service operations.

OpenTransaction → Trans
starts a new transaction and delivers a unique TID Trans. This identifier will be used in the other operations in the transaction.

CloseTransaction(Trans) → (Commit, Abort)
ends a transaction: a Commit returned value indicates that the transaction has committed; an Abort returned value indicates that it has aborted.

AbortTransaction(Trans)
aborts the transaction.

Alternatively, the transaction may have to abort for one of several reasons related to the nature of the transaction itself, to conflicts with another transaction or to the failure of processes or computers. When a transaction is aborted the transactional service must ensure that none of its effects are visible to future transactions, either in the data items or in their copies in permanent storage.

A transaction is either successful or it is aborted in one of two ways - the client aborts it (using an AbortTransaction call to the server) or the server aborts it. Figure 12.3 shows these three alternative life histories for transactions.

Service actions related to failures If a server halts unexpectedly it aborts any uncommitted transactions when it starts up again and uses a recovery procedure to restore the values of the data items to the values produced by the most recently committed transaction. To deal with a client that halts unexpectedly during a transaction, servers can give each transaction an expiry time and abort any transaction that has not completed before its expiry time.

Client actions related to failures of a server If a server halts while a transaction is in progress the client will become aware of this when one of the operations returns an error report after a time-out. If a server halts and then restarts during the progress of a transaction, the transaction will no longer be valid and the client must be informed as a result of the next operation. In either case, the client must then formulate a plan, possibly in consultation with the human user, for the completion or abandonment of the task of which the transaction was a part.

Banking service operations in terms of Read and Write operations Many of the techniques for handling transactions have been developed in the context of databases and distributed file services. The Read and Write operations are used to access and update both database records and portions of files. Servers such as the banking service provide operations such as Deposit and Withdraw instead of Read and Write on the data items. However the implementation of such operations can be decomposed in terms of an operation Read that accesses the value of a data item and Write that replaces it with a new value.

In the discussion of concurrency control and recovery, we use the notation D.Read and D.Write(newValue) to denote operations within a server that access and update a data item representing the resource D.

For example, we may define informally Deposit(Name, amount) as a server operation that reads the balance of account Name, increases it by amount, in dollars and then writes the balance:

\[
\text{Deposit(Name, amount):} \\
\text{balance} := \text{Name.Read()}; \\
\text{Name.Write(balance + amount)}
\]

Concurrency control

This section illustrates two well-known problems of concurrent transactions - the 'lost update' problem and the 'inconsistent retrievals' problem in the context of the banking service. This section then shows how both of these problems can be avoided by using serially equivalent executions of transactions.

The lost update problem The lost update problem is illustrated by the following pair of transactions on bank accounts A, B and C whose initial balances are $100, $200 and $300 respectively. Transaction T transfers $4 from account A to account B. Transaction U transfers $3 from account C to account B. The net effects of executing the transactions T and U should be to:

- decrease the balance of accounts A by $4 and C by $3;
- increase the balance of account B by $7.

Now consider the effects of allowing the transactions T and U to run concurrently as in Figure 12.4. Both transactions read the balance of B and then write it. The result is incorrect, increasing the balance of account B by $4 instead of $7. This is an illustration of the 'lost update' problem. U's update is lost because T overwrites it without seeing it. Both transactions have read the old value before either writes the new value.

In Figure 12.4 onwards we show the operations that read or write the balance of an account on successive lines down the page, and the reader should assume that an operation on a particular line is executed at a later time than the one on the lines above it.
### Inconsistent retrievals

Figure 12.5 shows another example related to a bank account in which transaction T transfers a sum from account A to B and transaction U obtains the sum of the balances of all the accounts in the bank. The balances of the two bank accounts, A and B, are both initially $200. The result includes the sum of A and B as $300, which is wrong. This is an illustration of the "inconsistent retrievals" problem. U’s retrievals are inconsistent because T has performed only the withdrawal part of a transfer at the time the sum is calculated.

### Serial equivalence

If each of several transactions is known to have the correct effect when it is done on its own, then we can infer that if these transactions are done one at a time in some order the combined effect will also be correct. An interleaving of the operations of transactions in which the combined effect is the same as if the transactions had been performed one at a time in some order is a serially equivalent interleaving.

---

### The inconsistent retrievals problem

<table>
<thead>
<tr>
<th>Transaction T:</th>
<th>Transaction U:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank$Withdraw(A, 100); Bank$Deposit(B, 100)</td>
<td>Bank$BranchTotal( )</td>
</tr>
<tr>
<td>balance := A.Read()</td>
<td>balance := A.Read()</td>
</tr>
<tr>
<td>A.Write( balance - 100)</td>
<td>$100</td>
</tr>
<tr>
<td>balance := balance + B.Read()</td>
<td>balance := balance + B.Read()</td>
</tr>
<tr>
<td>balance := balance + C.Read()</td>
<td>$100</td>
</tr>
<tr>
<td>$200</td>
<td>$300</td>
</tr>
<tr>
<td>$100</td>
<td>$300</td>
</tr>
<tr>
<td>$200</td>
<td>$300</td>
</tr>
<tr>
<td>B.Write( balance + 100)</td>
<td>B.Write( balance + 100)</td>
</tr>
<tr>
<td>balance := balance + B.Read()</td>
<td>balance := balance + C.Read()</td>
</tr>
<tr>
<td>$200</td>
<td>$300</td>
</tr>
<tr>
<td>$200</td>
<td>$300</td>
</tr>
<tr>
<td>$300</td>
<td>$300</td>
</tr>
</tbody>
</table>

When we say that two different transactions have the same effect as one another, we mean that the read operations return the same values and that the data items have the same values at the end.

The use of serial equivalence as a criterion for correct concurrent execution prevents the occurrence of lost updates and inconsistent retrievals.

The lost update problem occurs when two transactions read the old value of a variable and then use it to calculate the new value. This cannot happen if one transaction is performed before the other, because the later transaction will read the value written by the earlier one. As a serially equivalent interleaving of two transactions produces the same effect as a serial one, we can solve the lost update problem by means of serial equivalence. Figure 12.6 shows one such interleaving in which the operations that affect the shared account, B, are actually serial, for transaction T does all its operations on B before transaction U does. Another interleaving of T and U that has this property is one in which transaction U completes its operations on account B before transaction T starts.

We now consider the effect of serial equivalence in relation to the inconsistent retrievals problem in which Transaction T is transferring a sum from account A to B and Transaction U is obtaining the sum of all the balances (see Figure 12.5). The inconsistent retrievals problem can occur when a retrieval transaction runs concurrently with an update transaction. It cannot occur if the retrieval transaction is performed before or after the update transaction. A serially equivalent interleaving of a retrieval transaction and an update transaction (for example, as in Figure 12.7) will also prevent it occurring.

Serial equivalence requires that all of a transaction’s accesses to a particular data item should be serialized with respect to accesses by other transactions. All pairs of conflicting operations of two transactions should be executed in the same order.

Serial equivalence is used as a criterion for the derivation of concurrency control protocols. These protocols attempt to serialize transactions in their access to data items. Chapter 13 introduces three approaches to concurrency control:
locking: In which each data item is locked by the first transaction that accesses it so that no other transaction may access the item until the first transaction has committed or aborted.

optimistic concurrency control: In which it is hoped that no conflicts of access will occur. Transactions proceed until they are ready to commit, when there is a check. If conflicts with other concurrent transactions have occurred, a transaction is aborted and must be restarted.

timestamps: In which each transaction has a timestamp and data items are timestamped each time they are accessed. Transactions are aborted and restarted when they are too late to perform an operation on a particular item.

Such methods for concurrency control are designed to allow two or more transactions to be executed concurrently while maintaining the serial equivalence property.

Recoverability

A transactional service must record the effects of all committed transactions and none of the effects of aborted transactions. A transactional service must therefore allow for the fact that a transaction may abort by preventing it from affecting other concurrent transactions.

This section illustrates two problems associated with aborting transactions in the context of the banking service. These problems are called ‘dirty reads’ and ‘premature writes’ and both of them can occur in the presence of serially equivalent executions of transactions.

Dirty reads  The isolation property of transactions requires that transactions should not see the uncommitted state of other transactions. The ‘dirty read’ problem is caused by the interaction between a read operation in one transaction and an earlier write operation in another transaction on the same data item. Consider the executions illustrated in Figure 12.8, in which T reads and writes A and then U reads and writes A and the two executions are serially equivalent. Now suppose that the transaction T aborts after U has committed. Then the transaction U will have seen a value that never existed, since A will be restored to its original value. We say that the transaction U has performed a dirty read. As it has committed, it cannot be undone.

Recoverability of transactions  If a transaction (like U) has committed after it has seen the effects of a transaction that subsequently aborted, the situation is not recoverable. To ensure that such situations will not arise, any transaction (like U) that is in danger of having a dirty read delays its commit operation. The strategy for recoverability is to delay commits until after the commitment of any other transaction whose uncommitted state has been observed. In our example, U delays its commit until after T commits. In the case that T aborts, then U must abort as well.

Cascading aborts  In Figure 12.8 suppose that the transaction U delays committing until after T aborts. As we have said, U must abort as well. Unfortunately if any other transactions have seen the effects due to U, they too must be aborted. The aborting of these latter transactions may cause still further transactions to be aborted. Such situations are called cascading aborts. To avoid cascading aborts, transactions are only allowed to read data items that were written by committed transactions. To ensure that this is the case, any Read operation must be delayed until other transactions that applied a Write operation to the same data item have committed or aborted. The avoidance of cascading aborts is a stronger condition than recoverability.

Premature writes  Consider another implication of the possibility that a transaction may abort. This one is related to the interaction between Write operations on the same data item belonging to different transactions. For an illustration, we introduce to the Bank service, a new operation SetBalance(Name, Amount) which sets the balance of account Name to Amount. We consider two SetBalance transactions T and U on account A, as shown in Figure 12.9. Before the transactions, the balance of account A was $100.
The two executions are serially equivalent, with T setting the balance to $3 and U setting it to $5. If the transaction U aborts and T commits, the balance should be $3.

Some database systems implement the action of Abort by restoring 'before images' of all the Writes of a transaction. In our example, A is $100 initially, which is the 'before image' of T's Write, similarly $3 is the 'before image' of U's Write. Thus if U aborts, we get the correct balance of $3.

Now consider the case when U commits and then T aborts. The balance should be $5 but as the 'before image' of T's Write is $100, we get the wrong balance of $100. Similarly if T aborts and then U aborts, the 'before image' of U's Write is $3 and we get the wrong balance of $3 – the balance should of course revert to $100.

To ensure correct results, Write operations must be delayed until earlier transactions that updated the same data items have either committed or aborted.

**Strict executions of transactions** □ Generally it is required that transactions should delay both their Read and Write operations so as to avoid both ‘dirty reads’ and ‘premature writes’. The executions of transactions are called strict if the service delays both Read and Write operations on a data item until all transactions that previously wrote that data item have either committed or aborted. The strict execution of transactions enforces the desired property of isolation.

**Tentative versions** □ A transactional service must be designed so that any updates of the data items can be removed if and when a transaction aborts. To make this possible, all of the update operations performed during a transaction are done in tentative versions of data items in volatile memory. Each transaction is provided with its own private set of tentative versions of any data items that it has altered. All the update operations of a transaction store values in the transaction’s own private set. Access operations in a transaction take values from the transaction’s own private set if possible, or failing that, from the data items.

The tentative versions are transferred to the data items only when a transaction commits, by which time they will also have been recorded in permanent storage. This is performed in a single step during which other transactions are excluded from access to the data items that are being altered. When a transaction aborts, its tentative versions are deleted.

**A transactional file service**

A transactional file service is a form of file service that supports atomic transactions on its files. It supports a construct to allow a client program to group together the file service operations that comprise an atomic transaction. Figure 12.10 shows the definitions of the operations in a transactional file service interface. In addition to the usual file service operations, the operations OpenTransaction, CloseTransaction and AbortTransaction are provided to open, close and abort transactions, see Figure 12.2. The file operations are similar to those defined in Figure 7.4 but each one has an additional argument for the transaction identifier.

When a transaction is opened, the service delivers a unique transaction identifier (TID) to the client. All of the procedures with the exception of OpenTransaction would report an error for an invalid TID, but these errors are not shown in Figure 12.10. The client uses the transaction identifier returned by OpenTransaction to indicate which of the subsequent file operations (up to the close of the transaction) are to be included in the particular transaction it identifies. All of the file service operations defined in Chapter 7 are included in the transaction service with a modified interface. In this modified interface, each operation invocation requires an additional argument to specify the transaction identifier and the procedure names are modified to indicate the difference: TRead, TWrite, TCreatet, TDelete, TTruncate and TLength.

Normally, when a client has performed the file operations that comprise the transaction, it terminates the transaction using CloseTransaction, which delivers a Commit result, indicating that the transaction has committed and that subsequent transactions by other clients will see the results of all of the changes to files made within
the transaction. If the transaction has been aborted an \textit{Abort} result is delivered from \textit{CloseTransaction} or is reported as an error after an earlier request.

Concurrency control and recovery

Although concurrency control and recovery are essential parts of any transaction system, they also have independent uses. This chapter has shown that concurrency control is required when multiple clients share data and that recovery is required for fault tolerance. When transactions apply to recoverable data that is not shared, concurrency control is not required. For these reasons the design and implementation of mechanisms for concurrency control and recovery should be independent of one another. We therefore present concurrency control and recovery separately in Chapters 13 and 15.

12.5 Nested transactions

A transaction may be structured as a set of \textit{nested transactions}, each of which may itself consist of a set of further nested transactions. For example, a \textit{Transfer} transaction in the banking service could be structured as a pair of nested transactions, one of which is a \textit{Withdraw} operation and the other a \textit{Deposit} operation, as illustrated in Figure 12.11. The atomicity properties stated in Section 12.4 apply to nested transactions — in which the client cannot observe the hierarchic structure.

Nested transactions are useful for two reasons:

1. Nested transactions at one level may run concurrently with other nested transactions at the same level in the hierarchy. This can allow additional concurrency in a transaction. Concurrency control is used to isolate the effects of concurrent nested transactions from one another.

2. Nested transactions can commit or abort independently. In comparison with a single transaction, a nested transaction is more robust. The aborting of a nested transaction does not necessarily imply that its parent must abort. In fact the parent can perform different actions according to whether the child aborts or commits. On the other hand, the committing of a nested transaction is conditional on the committing of its parent.

Consider the \textit{Transfer} transaction. When the two nested transactions (T₁ and T₂) both commit the \textit{Transfer} transaction can also commit. Suppose that a \textit{Withdraw} transaction aborts whenever an account is overdrawn. Now consider the case when the \textit{Withdraw} transaction aborts and the \textit{Deposit} transaction commits — and recall that the commitment of a child transaction is conditional on the parent transaction committing. We presume that the parent (\textit{Transfer}) transaction will decide to abort. The aborting of the parent transaction causes the nested transactions to abort — so the \textit{Deposit} transaction is aborted and all its effects are undone.

In some nested transactions the parent transaction may decide to commit in spite of the fact that one or more of its child transactions have aborted. For example, a transaction to deliver a mail message to a list of recipients could be structured as a set of nested transactions, each of which delivers the message to one of the recipients. If one or more of the nested transactions fails, the parent transaction could record the fact and then commit, with the result that all the successful child transactions commit.

Nested transactions are particularly useful in distributed systems because child transactions may run concurrently in different servers. We return to this issue in Chapter 14.

12.6 Summary

Services provide their clients with operations that enable them to use shared resources, which are represented in the server by data items. The interleaving of threads in a single server due to concurrent clients could cause inconsistencies in these data items. Server operations on shared data must therefore be designed to be atomic.

Conversations between client and server allow the two to work in parallel. Servers that provide conversations cannot be stateless and may require concurrency control at the level of a conversation. The techniques used in transactions can be used for conversations.

Servers need to provide recoverable data if they hold resources that may be used by client processes over a long period of time. This is particularly applicable when a server provides information that users will expect to last for long periods of time.

Transactions provide a means by which clients can specify sequences of operations that are atomic in the presence of other concurrent transactions and server failures. The first aspect of atomicity is achieved by running transactions so that their effects are serially equivalent. Concurrency control protocols are derived from the criterion of serial equivalence. The effects of committed transactions are recorded in permanent storage so that the transaction service can recover from failures. To allow transactions the ability to abort, without having harmful side effects on other transactions, executions must be strict — that is, reads and writes of one transaction must be delayed until other transactions that wrote the same data items have either committed or aborted. To allow transactions the choice of either committing or aborting, their operations are performed in tentative versions that cannot be accessed by other transactions. The tentative versions of data items are copied to the real data items and to permanent storage when a transaction commits.

Nested transactions are formed by structuring transactions from other sub-transactions. Nesting is particularly useful in distributed systems because it allows concurrent execution of sub-transactions in separate servers. Nesting also has the advantage of allowing independent recovery of parts of a transaction.