Transactions

Distributed Software Systems

Transactions

- Motivation
  - Provide atomic operations at servers that maintain shared data for clients
  - Provide recoverability from server crashes
- Properties
  - Atomicity, Consistency, Isolation, Durability (ACID)
- Concepts: commit, abort

Concurrency control

- Motivation: without concurrency control, we have lost updates, inconsistent retrievals, dirty reads, etc. (see following slides)
- Concurrency control schemes are designed to allow two or more transactions to be executed correctly while maintaining serial equivalence
  - Serial Equivalence is correctness criterion
    - Schedule produced by concurrency control scheme should be equivalent to a serial schedule in which transactions are executed one after the other
- Schemes: locking, optimistic concurrency control, time-stamp based concurrency control

Figure 12.1 A client's banking transaction

Transaction:T:
Bank$Withdraw(A, 100);
Bank$Deposit(B, 100);
Bank$Withdraw(C, 200);
Bank$Deposit(B, 200);

Figure 12.2 Transactional service operations.

OpenTransaction $\rightarrow$ *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) $\rightarrow$ *(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.

Figure 12.3 Transaction life histories.

Successful	*Aborted by client*		*Aborted by server*
OpenTransaction	*OpenTransaction*		*OpenTransaction*
operation	*operation*		*operation*
operation	*operation*		*operation*
•	•	SERVER	•
•	•	ABORTS →	•
operation	*operation*		*operation ERROR reported to client*
CloseTransaction	*AbortTransaction*		

Figure 12.4 The lost update problem.

Transaction T: *Bank$Withdraw(A, 4);* *Bank$Deposit(B, 4)*		**Transaction U:** *Bank$Withdraw(C, 3);* *Bank$Deposit(B, 3)*	
balance := A.Read( )	$100		
A.Write (balance – 4)	$96		
		balance := C.Read( )	$300
		C.Write (balance – 3)	$297
balance := B.Read( )	$200		
		balance := B.Read( )	$200
		B.Write (balance + 3)	$203
B.Write (balance + 4)	$204		

Figure 12.5 The inconsistent retrievals problem.

Transaction T: *Bank$Withdraw(A, 100); Bank$Deposit(B, 100)*		**Transaction U:** *Bank$BranchTotal( )*	
balance := A.Read()	$200		
A.Write (balance - 100)	$100		
		balance := A.Read()	$100
		balance := balance + B.Read()	$300
		balance := balance + C.Read()	$300+.
balance := B.Read()	$200	•	
B.Write (balance + 100)	$300	•	

Figure 12.6 A serially equivalent interleaving of T and U.

Transaction T: *Bank$Withdraw(A, 4);* *Bank$Deposit(B, 4)*		**Transaction U:** *Bank$Withdraw(C, 3);* *Bank$Deposit(B, 3)*	
balance := A.Read( )	$100		
A.Write(balance – 4)	$96		
		balance := C.Read( )	$300
		C.Write(balance – 3)	$297
balance := B.Read( )	$200		
B.Write (balance + 4)	$204		
		balance := B.Read( )	$204
		B.Write(balance + 3)	$207

Figure 12.7 An inconsistent retrievals solution

Transaction T: *Bank$Withdraw(A, 100);* *Bank$Deposit(B, 100)*		**Transaction U:** *Bank$BranchTotal( )*	
balance :=A.Read( )	$200		
A.Write(balance – 100)	$100		
balance := B.Read( )	$200		
B.Write(balance + 100)	$300		
		balance := A.Read( )	$100
		balance := balance +B.Read( )	$400
		balance := balance +C.Read( )	$400+
		...	

Figure 12.8 A dirty read when transaction T aborts.

Transaction T: *Bank.$Deposit(A, 3)*		**Transaction U:** *Bank$Deposit(A, 5)*	
balance :=A.Read( )	$100		
A.Write(balance + 3)	$103		
		balance := A.Read( )	$103
		A.Write (balance + 5)	$108
		Commit transaction	
Abort transaction			

Figure 12.9 Over-writing uncommitted values.

Transaction T: *Bank$Deposit(A, 3)*		**Transaction U:** *Bank$Deposit(A, 5)*	
balance :=A.Read( )	$100		
A.Write(balance + 3)	$103		
		balance := A.Read( )	$103
		A.Write (balance + 5)	$108
		Abort transaction	

Figure 12.11 Nested *Transfer* transaction.