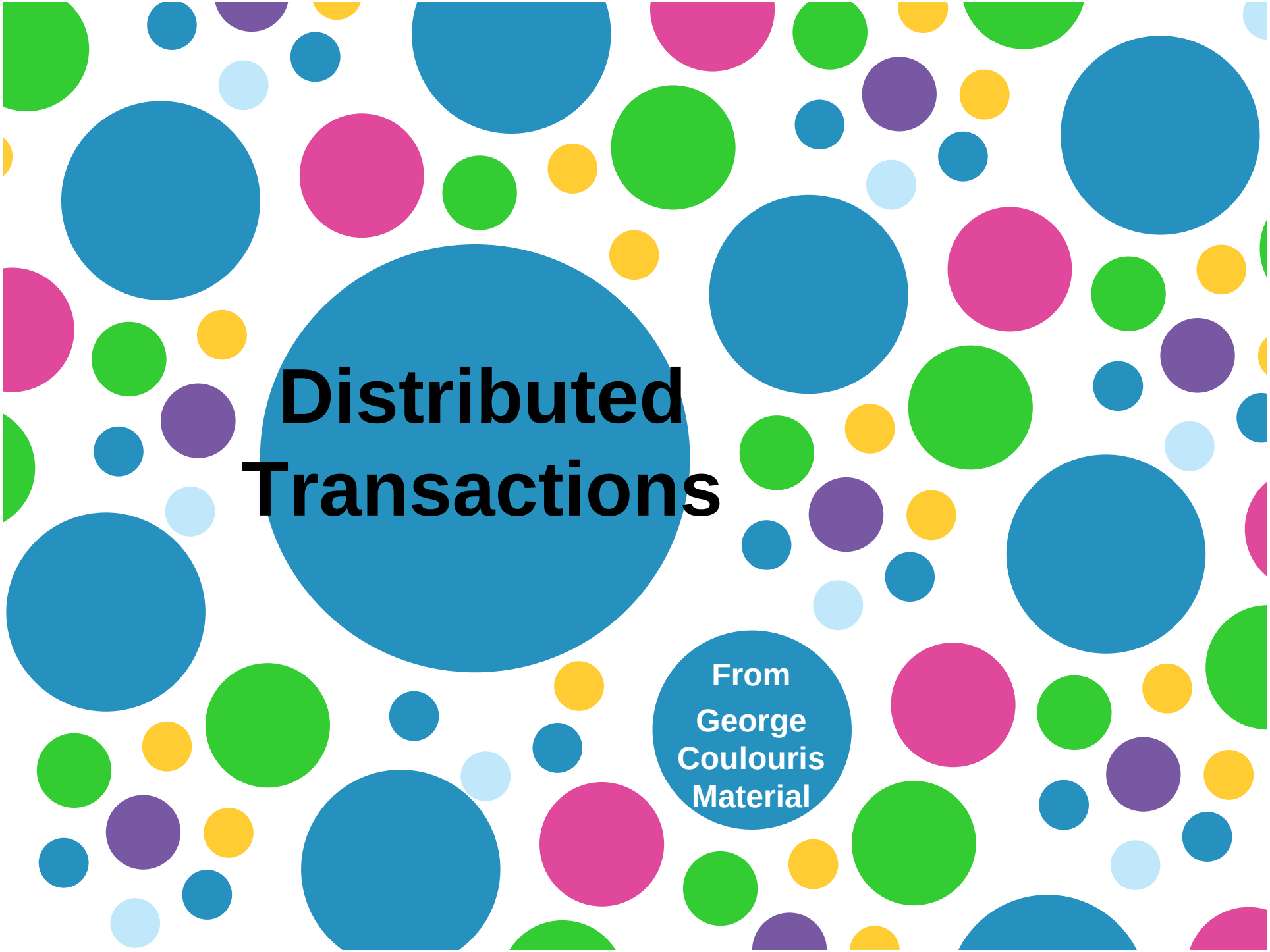
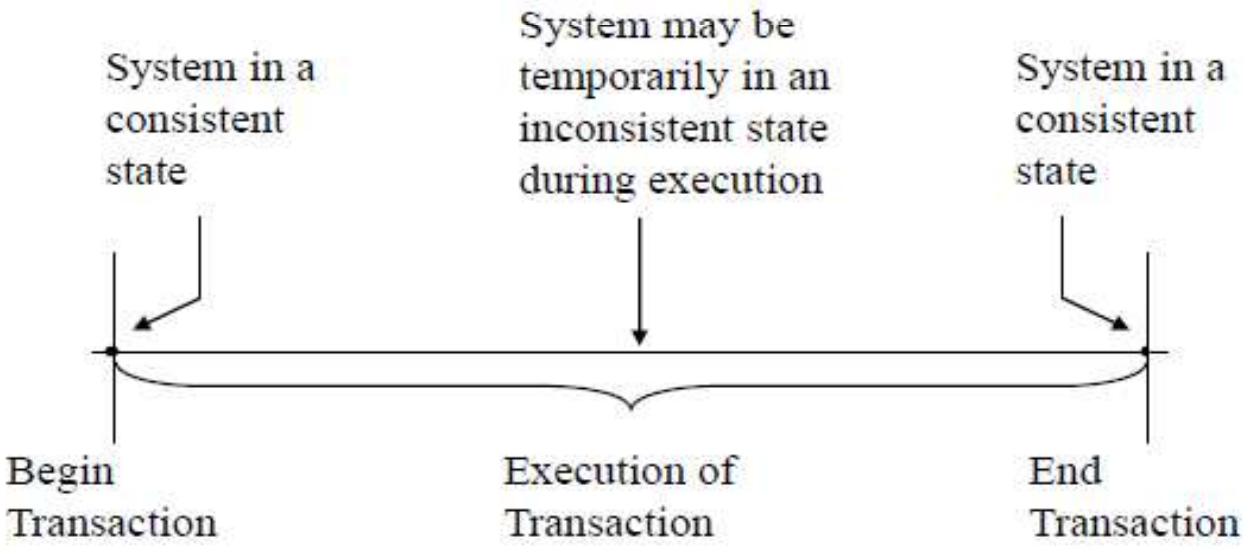

Distributed Transactions
From George Coulouris Material

Transaction

A transaction is a collection of actions that make consistent transformations of system states while preserving system consistency.

- concurrency transparency
- failure transparency



Example of Transaction

```
begin
    input(flight_no, date, customer_name);
    Begin_transaction Reservation
    begin
        Write(flight(date).stsold++);
        Write(flight(date).cname, customer_name);
        Commit
    end. {Reservation}
    output("reservation completed")
...
end
```

Transaction Primitives

- Special primitives required for programming using transactions
  - Supplied by the operating system or by the language runtime system
- Examples of transaction primitives:
  - BEGIN_TRANSACTION: Mark the start of a transaction
  - END_TRANSACTION (EOT): Terminate the transaction and try to commit (there may or may not be a separate COMMIT command)
  - ABORT_TRANSACTION: Kill the transaction and restore the old values
  - READ: Read data from a file (or other object)
  - WRITE: Write data to a file (or other object)

Operations of the *Account* interface

deposit(amount)
deposit amount in the account
withdraw(amount)
withdraw amount from the account
getBalance() -> amount
return the balance of the account
setBalance(amount)
set the balance of the account to amount

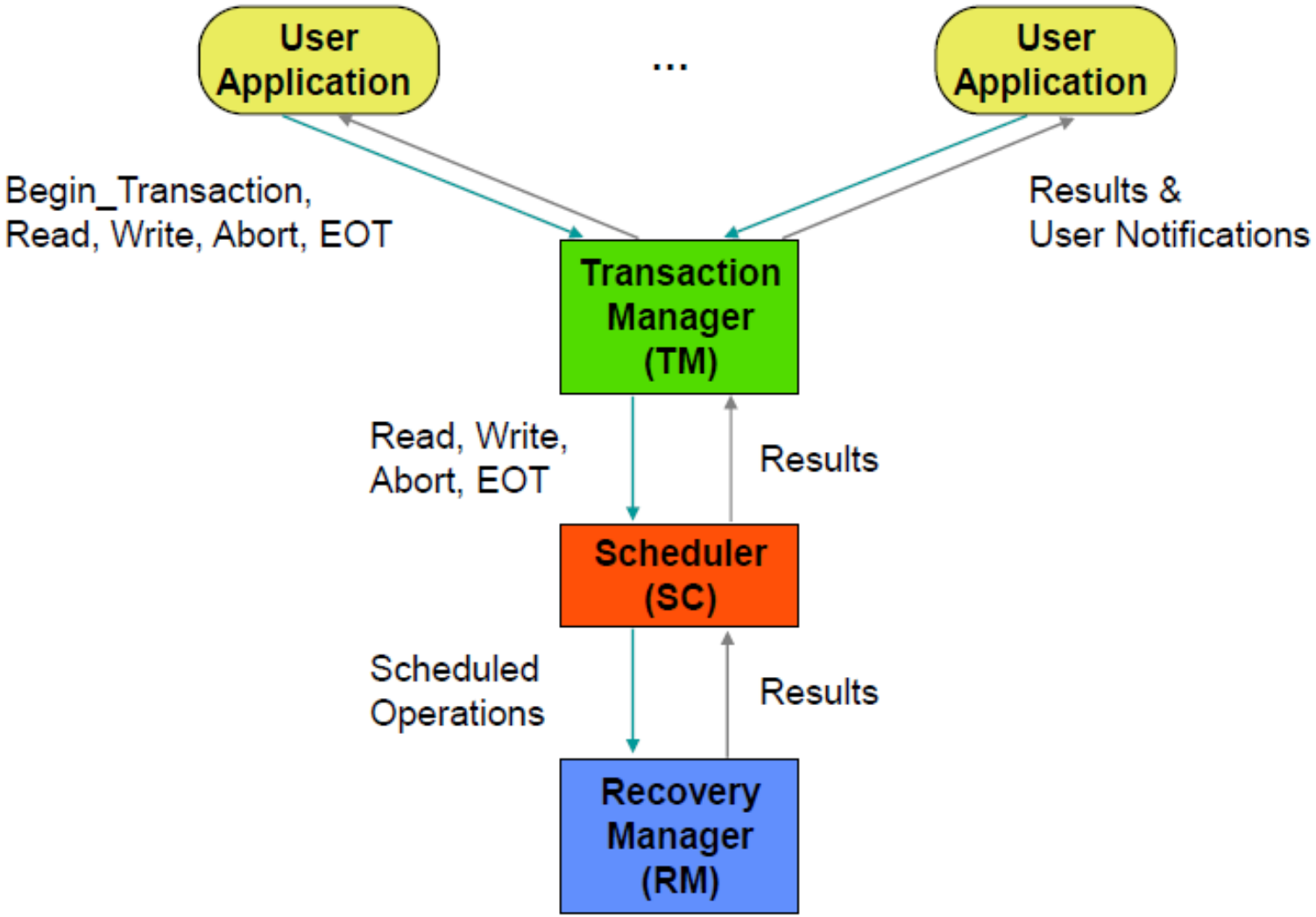
Operations of the Branch interface

create(name) -> account
create a new account with a given name
lookUp(name) -> account
return a reference to the account with the given name
branchTotal() -> amount
return the total of all the balances at the branch

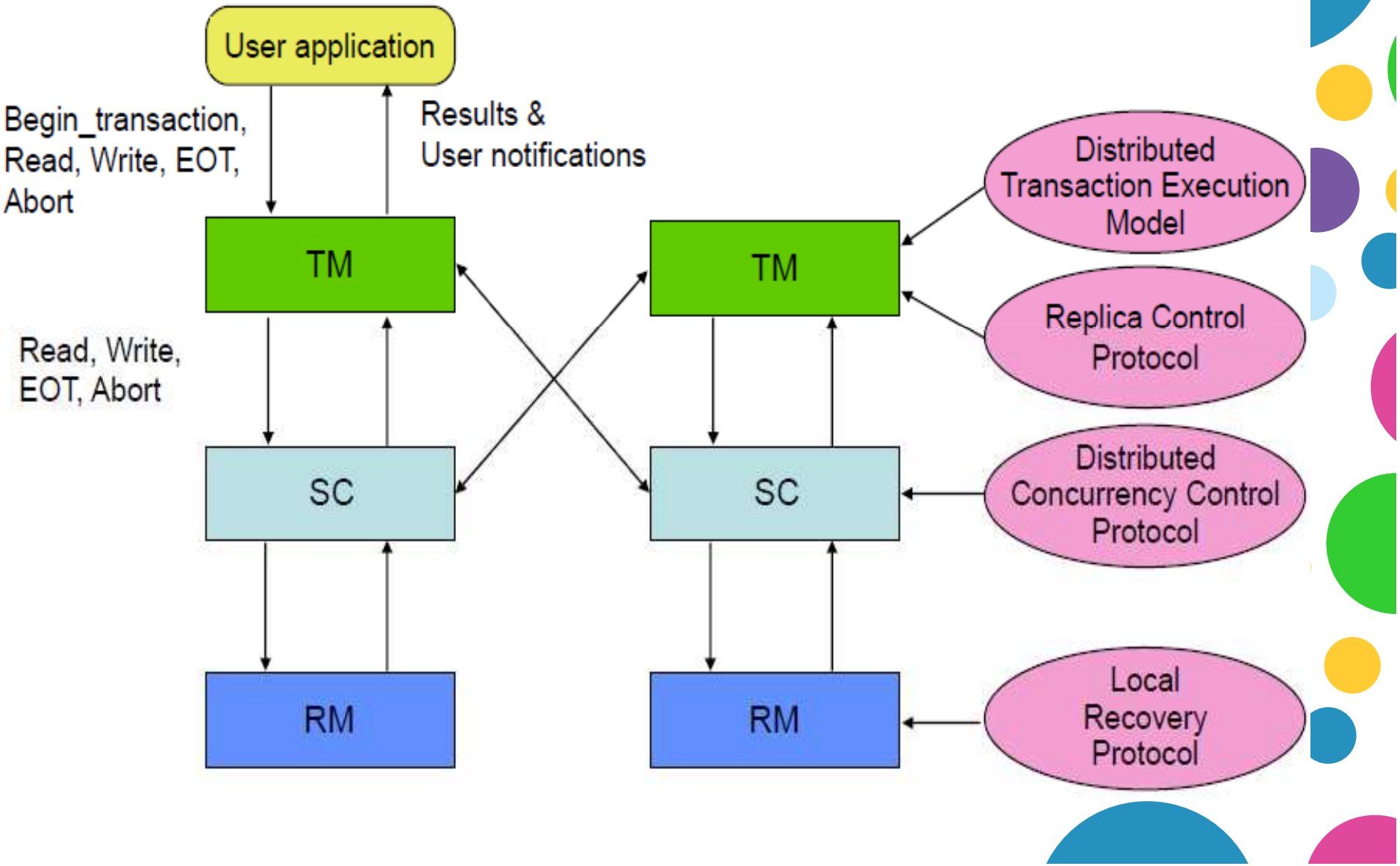
Enhancing Client Cooperation by Signaling

- **Clients** may use a **server** as a means of **sharing** some **resources**. E.g. some clients update the server's objects and other clients access them.
- However, in some applications, **threads** need to **communicate** and **coordinate** their actions.
- **Producer** and **Consumer** problem.
  - **Wait** and **Notify** actions.

Centralized Transaction Execution



Distributed Transaction Execution



Properties of Transactions

ATOMICITY

- All or nothing
- Multiple operations combined as an atomic transaction

CONSISTENCY

- No violation of integrity constraints
  - → System specific rules. In a distributed database, the index should always reflect the data, foreign key constraints are not violated, triggers are issued, replicas have the same value, etc.
- Transactions are correct programs

ISOLATION ⇐ Our focus in this module

- Concurrent changes invisible → serializable

DURABILITY

- Committed updates persist
- Database recovery

Serializability of Transactions

- Serializability
  - If several transactions are executed concurrently, the results must be the same as if they were executed serially in some order.
- Incomplete results
  - An incomplete transaction cannot reveal its results to other transactions before its commitment.
  - Necessary to avoid cascading aborts.
  - Anomalies:
    - → Lost updates
      - The effects of some transactions are not reflected on the database.
    - → Inconsistent retrievals
      - E.g. a transaction, if it reads the same object more than once, should always read the same value.

Concurrency Control – Lost Update Problem

Initial values: A=100, B=200, C=300

Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance(); *b.setBalance(balance*1.1);* *a.withdraw(balance/10)*		*balance = b.getBalance();* *b.setBalance(balance*1.1);* *c.withdraw(balance/10)*	
balance = b.getBalance();	$200		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$280

a, b and c initially have bank account balance are: 100, 200, and 300.
T transfers an amount from a to b. U transfers an amount from c to b.
b is increased by 10% on its balance in each. Totally 20 % hike

Concurrency Control – Inconsistent Retrieval Problem

Initial values: A=200, B=200

Transaction *V*:		**Transaction *W*:**	
a.withdraw(100) *b.deposit(100)*		*aBranch.branchTotal()*	
a.withdraw(100);	$100		
		total = a.getBalance()	$100
		total = total+b.getBalance()	$300
		total = total+c.getBalance()	
b.deposit(100)	$300	⋮	

Serially Equivalent

- **If these transactions are done one at a time in some order, then the final result will be correct.**
- If we do not want to sacrifice the concurrency, an **interleaving** of the **operations** of **transactions** may lead to the **same effect** as if the **transactions** had been **performed one** at a **time** in **some order**.
- We say it is a **serially equivalent interleaving**.
- The use of **serial equivalence** is a criterion for **correct concurrent execution** to prevent lost updates and inconsistent retrievals.

A serially equivalent interleaving of *T* and *U*

Transaction *T*:		Transaction *U*:	
balance = b.getBalance()		*balance = b.getBalance()*	
*b.setBalance(balance*1.1)*		*b.setBalance(balance*1.1)*	
a.withdraw(balance/10)		*c.withdraw(balance/10)*	
balance = b.getBalance()	$200		
*b.setBalance(balance*1.1)*	$220		
		balance = b.getBalance()	$220
		*b.setBalance(balance*1.1)*	$242
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$278

A serially equivalent interleaving of *V* and *W*

Transaction*V*:		Transaction*W*:	
a.withdraw(100);		***aBranch.branchTotal()***	
b.deposit(100)			
a.withdraw(100);	**$100**		
b.deposit(100)	**$300**		
		total = a.getBalance()	**$100**
		total = total+b.getBalance()	**$400**
		total = total+c.getBalance()	
		...	

Read and *write* operation conflict rules

Operations of different transactions		Conflict	Reason
read	read	No	Because the effect of a pair of read operations does not depend on the order in which they are executed
read	write	Yes	Because the effect of a read and a write operation depends on the order of their execution
write	write	Yes	Because the effect of a pair of write operations depends on the order of their execution

A non-serially equivalent interleaving of operations of transactions *T* and *U*

Transaction *T*:	**Transaction *U*:**
x = read(i)	
write(i. 10)	
	y = read(j)
	write(j, 30)
write(j, 20)	
	z = read (i)

Access to objects i & j are serial, but transactions are nor serially equivalent

Solutions to Concurrency control problems

- **Locks** used to order transactions that access the same object according to request order.
- **Optimistic concurrency control** allows transactions to proceed until they are ready to commit, whereupon a check is made to see any conflicting operation on objects.
- **Timestamp ordering** uses timestamps to order transactions that access the same object according to their starting time

A dirty read when transaction *T* aborts

Transaction *T*:		Transaction *U*:	
a.getBalance()		*a.getBalance()*	
a.setBalance(balance + 10)		*a.setBalance(balance + 20)*	
balance = a.getBalance()	$100		
a.setBalance(balance + 10)	$110		
		balance = a.getBalance()	$110
		a.setBalance(balance + 20)	$130
		commit transaction	
abort transaction			

Recoverability of Transactions

- 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

- Abort of one transaction will cause other transactions to abort.
- Transactions are **only allowed** to **read objects** 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 object*** *have* ***committed*** *or* ***aborted****.*

Overwriting uncommitted values or Pre-mature Writes

Transaction *T*:		**Transaction *U*:**	
a.setBalance(105)		*a.setBalance(110)*	
	$100		
a.setBalance(105)	$105		
		a.setBalance(110)	$110

Before Image : 100

Before Image : 105

Overwriting uncommitted values or Pre-mature Writes

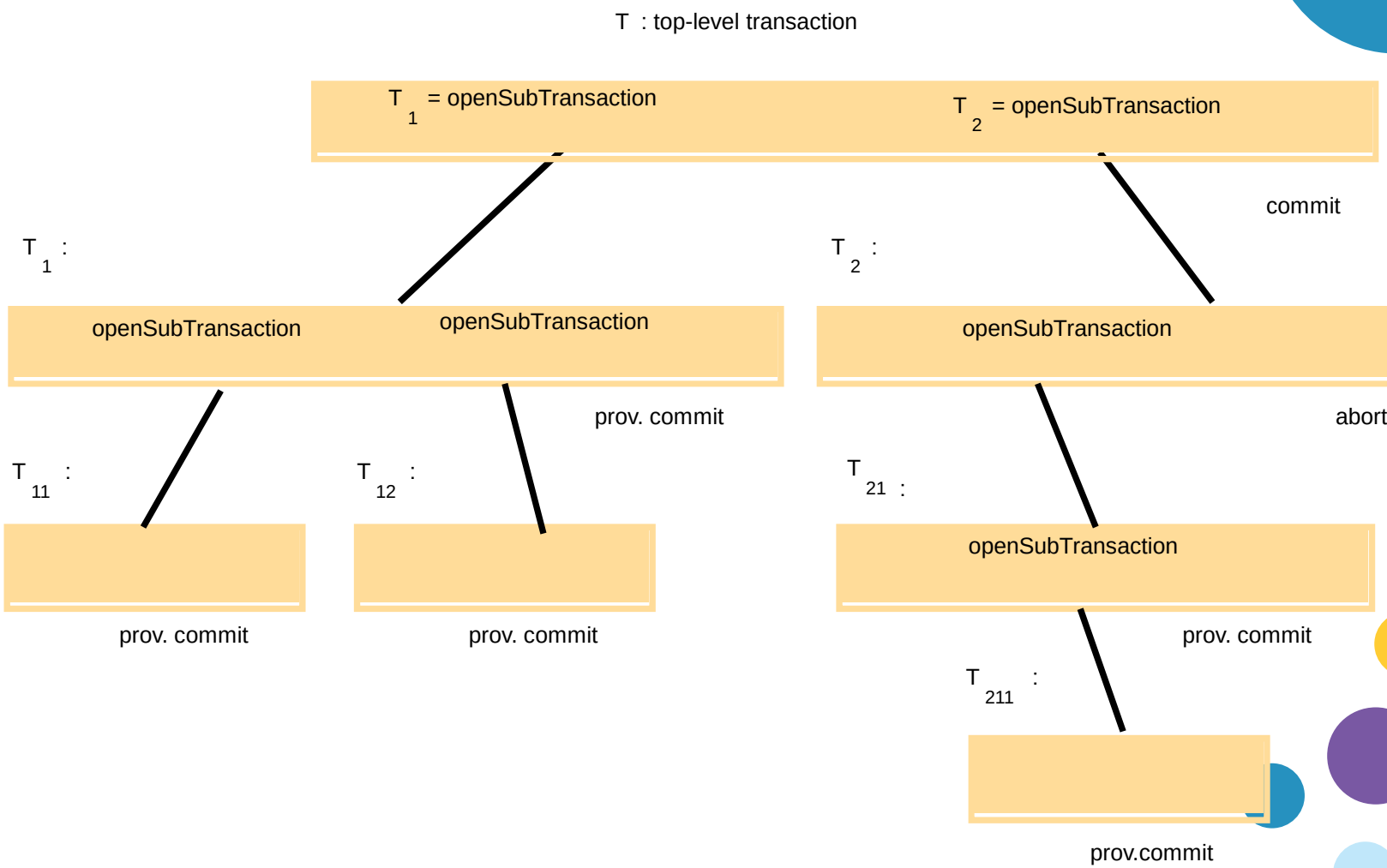
- **Strict Execution of Transactions**:

  If the transaction **delays** both **read** and **write** operations on an **object until** all **transactions** that **previously wrote** that **object** have either **committed** or **aborted**.
- **Tentative Version**: Make changes to tentative versions of objects. If transaction commits, transfer updates to objects, else delete tentative version.

Nested transactions

- **Nested transactions** extend the above transaction model by allowing **transactions** to be **composed** of **other transactions**.
- Thus **several transactions** may be **started** from **within** a **transaction**.

Nested transactions

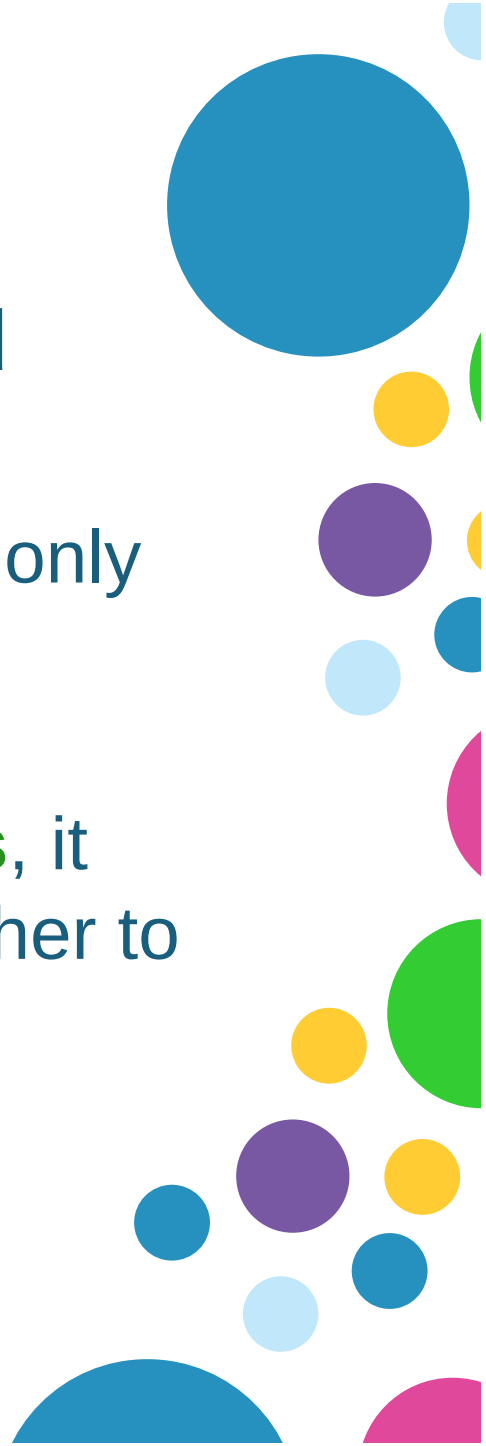


Nested transactions

- **Subtransactions** at one **level** (and their descendants) may **run concurrently** with other **subtransactions** at the same level in the hierarchy.
- **Subtransactions** can **commit** or **abort independently**.

Nested transactions

- The rules for committing of nested transactions
  - A transaction may **commit** or **abort** only after its **child** transactions have completed.
  - When a **subtransaction completes**, it makes an **independent** decision either to commit provisionally or to abort. Its decision to abort is final.
  - When a **parent aborts**, **all** of its **subtransactions** are aborted.

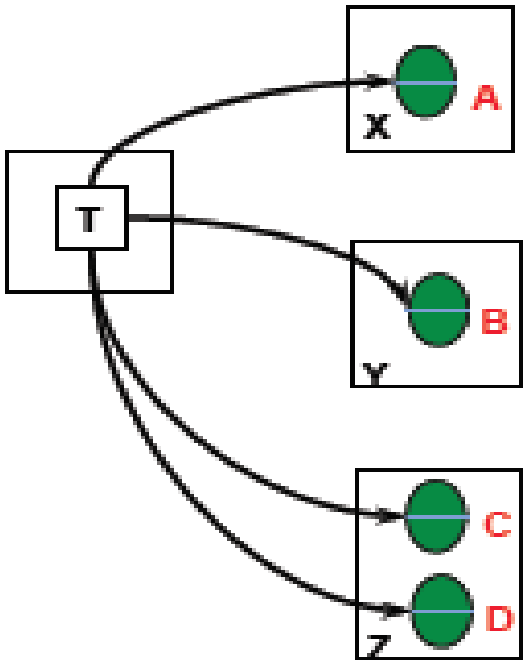


Nested transactions

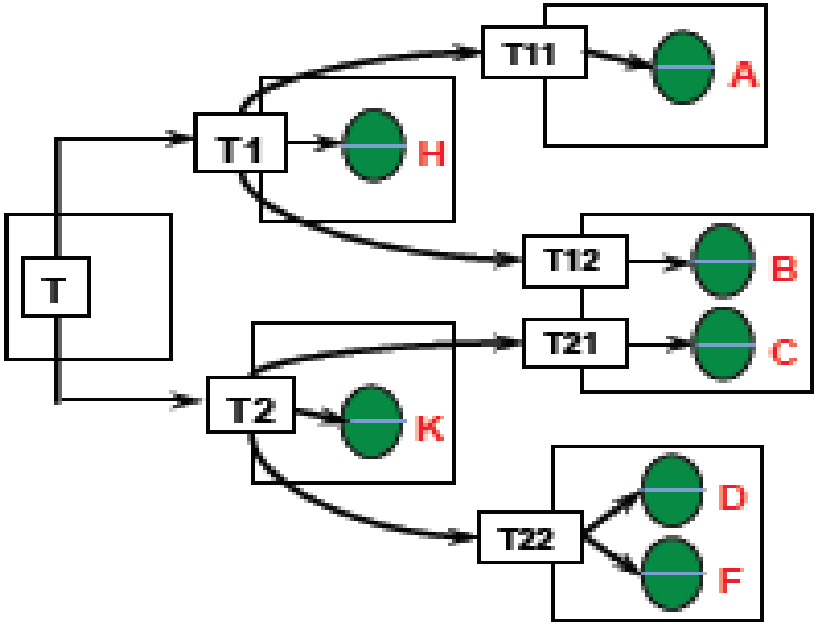
- When a **subtransaction aborts**, the **parent** can **decide** whether to **abort** or **not**.
- If the **top-level transaction commits**, then all of the **subtransactions** that have provisionally **committed** can **commit** too, provided that **none** of their **ancestors** has **aborted**.

Distributed Transactions

- Transactions that invoke operations at multiple servers



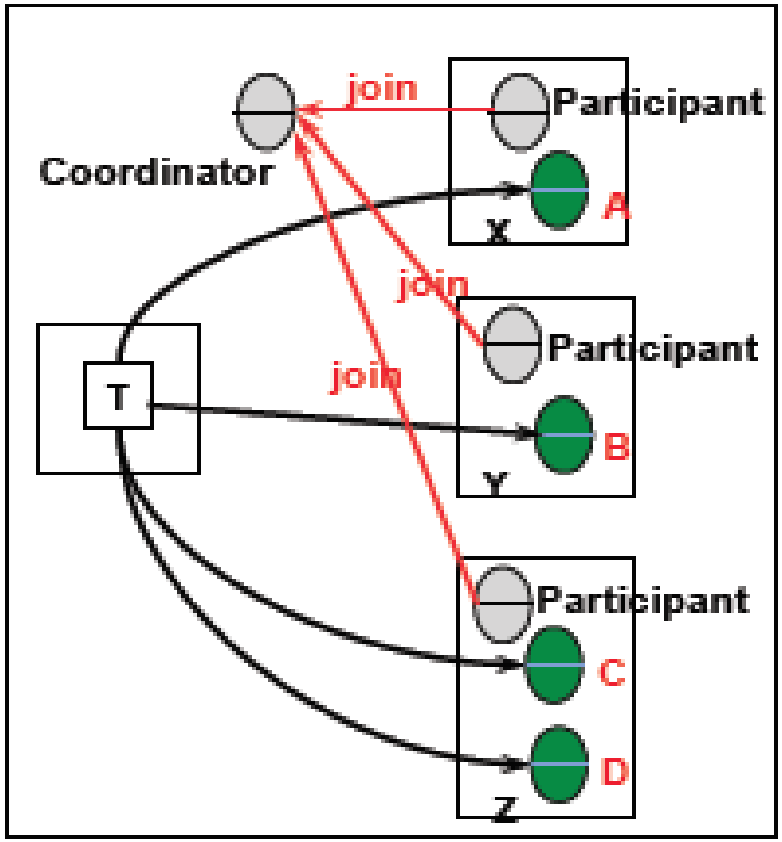
Flat Distributed Transaction



Nested Distributed Transaction

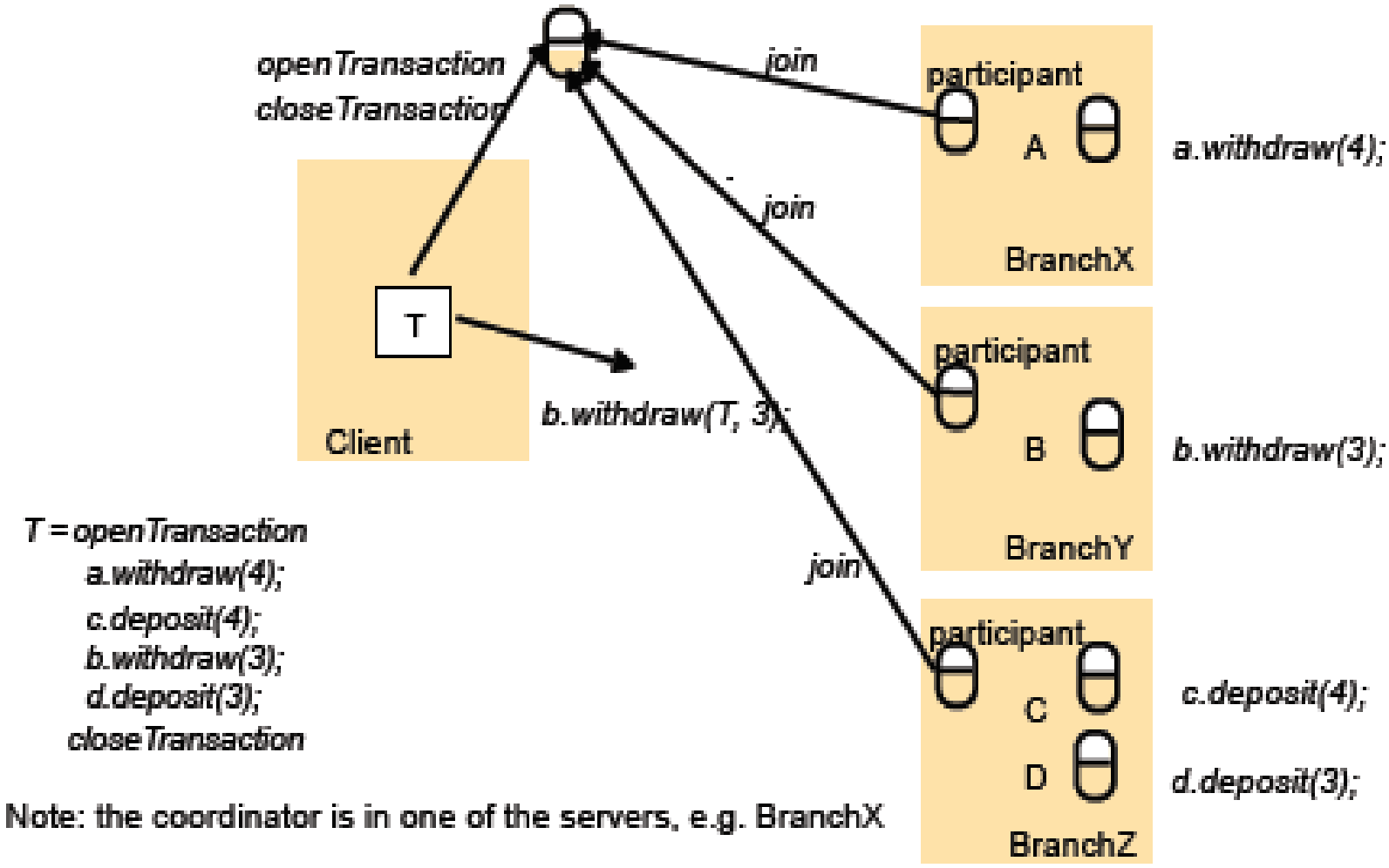
Distributed Transactions

- Coordinator
  - In charge of begin, commit, and abort
- Participants
  - Server processes that handle local operations



Coordinator & Participants

Distributed Transactions - Example



Thank You