Two-Tier vs Three-Tier Architecture Comparison of Client Side, Middleware Side and Server Side

1 Two-Tier Architecture

1.1 Structure

Client ↔ Server (Database)

- Client directly communicates with database.
- Suitable for small-scale applications.

1.2 Two-Tier Architecture – Layers

1.2.1 Client Side:

- User Interface
- Business Logic

1.2.2 Server Side:

- Database
- Data processing

1.2.3 Middleware Side:

- Not present

2 Three-Tier Architecture

2.1 Structure

Client ↔ Middleware ↔ Server (Database)

- Middleware handles business logic and security.
- Used for large-scale systems.

2.2 Three-Tier Architecture – Layers

2.2.1 Client Side:

- User Interface only

2.2.2 Middleware Side:

- Business Logic
- Validation
- Security

2.2.3 Server Side:

- Database storage

3 Comparison: Two-Tier vs Three-Tier

3.1 Client Side:

- **Two-Tier:** UI + Logic
- **Three-Tier:** UI only

3.2 Middleware:

- **Two-Tier:** No
- **Three-Tier:** Yes

3.3 Server Side:

- **Two-Tier:** DB + Logic
- **Three-Tier:** DB only

4 Advantages

4.1 Two-Tier:

- Simple design
- Fast for few users

4.2 Three-Tier:

- Scalable
- Secure
- Easy maintenance

5 Summary

- Two-Tier fits small applications.
- Three-Tier suits enterprise and web applications.

6 Transaction Servers

6.1 Transactions

Transactions represent an application design philosophy that guarantees robustness in distributed systems.

- Under the control of a TP Monitor, a transaction can be managed from its point of origin—typically on the client—across one or more servers, and then back to the originating client.
- When a transaction ends, all the parties involved are in agreement as to whether it succeeded or failed.
- The transaction becomes the contract that binds the client to one or more servers.

6.2 ACID Properties

6.2.1 Atomicity

A transaction is an indivisible unit of work: all of its actions succeed or they all fail.

- The actions under the transaction's umbrella may include the message queues, updates to a database, and the display of results on the client's screen.
- Atomicity is defined from the perspective of the consumer of the transaction.

6.2.2 Consistency

Consistency means that after a transaction executes, it must leave the system in a correct state or it must abort.

- If the transaction cannot achieve a stable end state, it must return the system to its initial state.

6.2.3 Isolation

Isolation means that a transaction's behavior is not affected by other transactions that execute concurrently.

- The transaction must serialize all accesses to shared resources and guarantee that concurrent programs will not corrupt each other's operations.
- A multiuser program running under transaction protection must behave exactly as it would in a single-user environment.
- The changes that a transaction makes to shared resources must not become visible outside the transaction until it commits.

6.2.4 Durability

Durability means that a transaction's effects are permanent after it commits.

- Its changes should survive system failures.
- The term "persistent" is a synonym for "durable."

6.3 Transactional Integrity

- A transaction becomes the fundamental unit of recovery, consistency, and concurrency in a client/server system.
- The application, in turn, relies on the underlying system—usually the TP Monitor—to help achieve this level of transactional integrity.
- The programmer should not have to develop tons of code that reinvents the transaction wheel.
- A more subtle point is that all the participating programs must adhere to the transactional discipline because a single faulty program can corrupt an entire system.
- A transaction that unknowingly uses corrupted initial data—produced by a non-transactional program—builds on top of a corrupt foundation.
- In an ideal world, all client/server programs are written as transactions.

7 Transaction Models

7.1 Flat Transaction

7.1.1 The Flat Transaction: An All-or-Nothing Proposition

All the work done within a transaction's boundaries is at the same level.

- The transaction starts with `begin_transaction` and ends with either a `commit_transaction` or `abort_transaction`.
- It provides an excellent fit for modeling short activities.

7.1.2 Characteristics

- The major virtue of the flat transaction is its simplicity and the ease with which it provides the ACID features.
- Flat transaction model does not provide the best fit in all environments.
- Flat transactions using two-phase commits are usually not allowed to cross inter-corporate boundaries.
- A typical flat transaction does not last more than two or three seconds to avoid monopolizing critical system resources such as database locks.
- OLTP client/server programs are divided into short transactions that execute back-to-back to produce results.

7.2 The Distributed Flat Transaction

A distributed flat transaction is a flat transaction run on multiple sites and updates resources located within multiple resource managers.

- The programmer is not aware of the considerable amount of "under-the-cover" activity that's required to make the multisite transaction appear flat.
- The transaction must travel across multiple sites to get to the resources it needs.
- Each site's TP Monitor must manage its local piece of the transaction.
- Within a site, the TP Monitor coordinates the transactions with the local ACID subsystems and resource managers—including database managers, queue managers, persistent objects, and message transports.

7.3 Two-Phase Commit Protocol

The two-phase commit protocol is used to synchronize updates on different programs and machines so that they either all fail or all succeed.

- This is done by centralizing the decision to commit but giving each participant the right of veto.
- If none of the parties present object, the transaction takes place.
- In 1993, ISO published its OSI TP standard that defines very rigidly how a two-phase commit is to be implemented.

7.3.1 First Phase of a Commit

- The commit manager node—also known as the root node or the transaction coordinator—sends prepare-to-commit commands to all the subordinate nodes that were directly asked to participate in the transaction.
- The first phase of the commit terminates when the root node receives ready-to-commit signals from all its direct subordinate nodes that participate in the transaction.
- This means that the transaction has executed successfully so far on all the nodes, and they're now ready to do a final commit.

7.3.2 The Second Phase of the Commit Starts

- The root node makes the decision to commit the transaction—based on the unanimous yes vote. It tells its subordinates to commit.
- The second phase of the commit terminates when all the nodes involved have safely committed their part of the transaction and made the transaction durable.
- The root receives all the confirmations and can tell its client that the transaction completed.

7.3.3 The Two-Phase Commit Aborts

The two-phase commit aborts if any of the participants return a refuse indication, meaning that their part of the transaction failed.

7.3.4 Limitations of Two-Phase Commit

- Performance overhead
- Hazard windows, where certain failures can be a problem. For example, if the root node crashes after the first phase of the commit, the subordinates may be left in disarray.

7.4 Limitations of Flat Transaction

Flat transactions have limitations when dealing with:

- Compound business transactions that need to be partially rolled back
- Business transactions with humans in the loop
- Business transactions with a lot of bulk
- Business transactions that span across companies or the Internet

8 Chained and Nested Transactions

8.1 Syncpoints

The simplest form of chaining is to use syncpoints—also known as savepoints—within a flat transaction that allow periodic saves of accumulated work.

- The syncpoint lets you roll back work and still maintain a live transaction.
- In contrast, a commit ends a transaction.
- Syncpoints also give you better granularity of control over what you save and undo.
- But the big difference is that the commit is durable while the syncpoint is volatile.
- If the system crashes during a transaction, all data accumulated in syncpoints is lost.

8.2 Chained Transactions

Chained transactions are a variation of syncpoints that make the accumulated work durable.

- They allow you to commit work while staying within the transaction.
- But the ability to roll back an entire chain's worth of work is lost.

8.3 Sagas

Sagas extend the chained transactions to let you roll back the entire chain.

- They do that by maintaining a chain of compensating transactions.
- You still get the crash resistance of the intermediate commits.
- You have the choice of rolling back the entire chain under program control.
- Treat the entire chain as an atomic unit of work.

8.4 Nested Transactions

Nested Transactions provide the ability to define transactions within other transactions.

- This is done by breaking a transaction into hierarchies of "subtransactions".
- The main transaction starts the subtransactions, which behave as dependent transactions.
- A subtransaction can also start its own subtransactions, thus making the entire structure very recursive.