School of Computing Sciences
Pak-Austria Fachhochschule: Institute of Applied Sciences and Technology, Haripur, Pakistan

Lab 07
Multi-Tier Client Server Architecture

Course: COMP-352L (Computer Networks Lab)

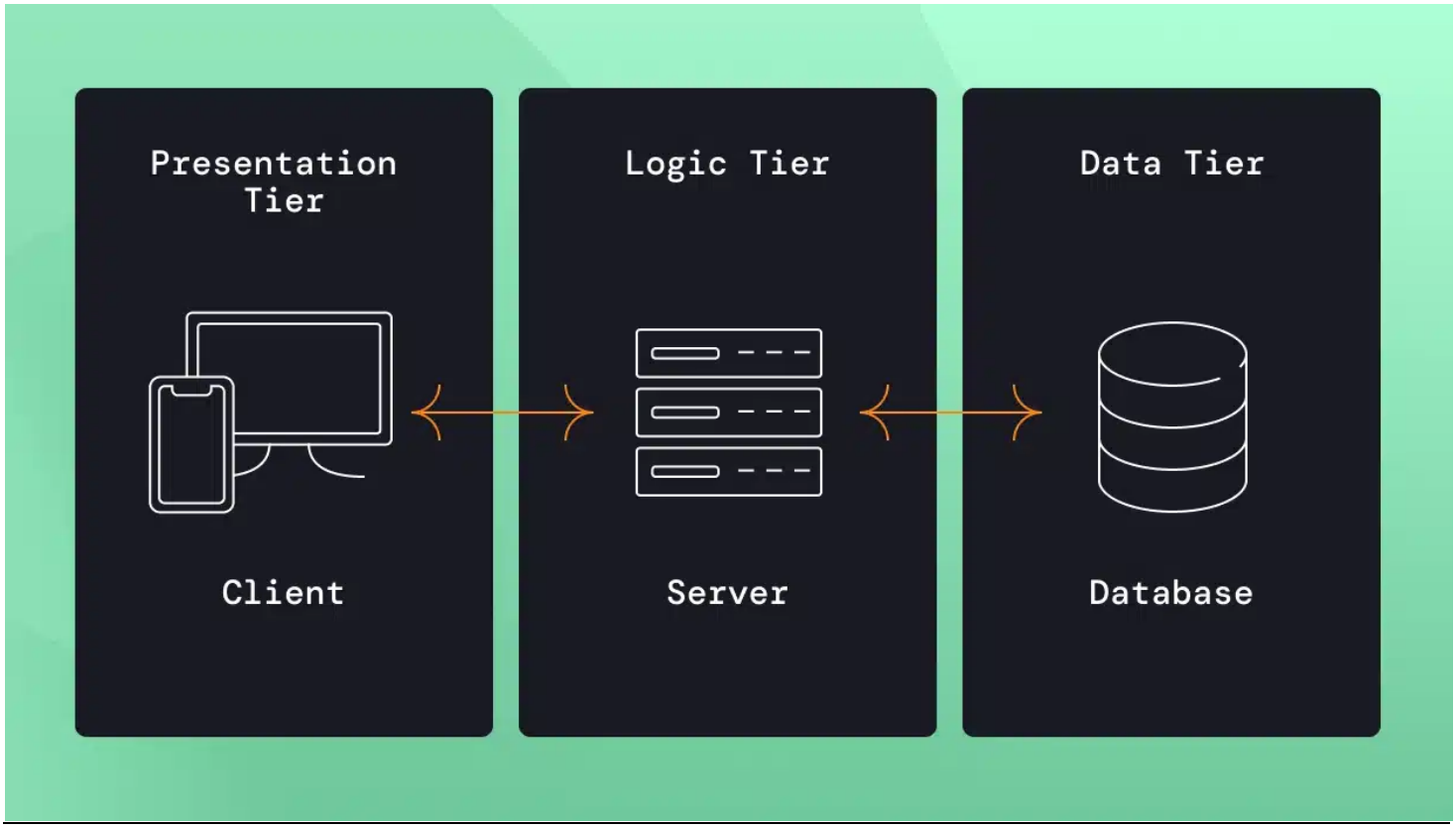
Lab Demonstrator: Jarullah Khan

School of Computing Sciences
Pak-Austria Fachhochschule: Institute of Applied Sciences and Technology, Haripur, Pakistan

Multi-Tier (Three-Tier) Architecture

A **multi-tier architecture** (often three-tier) is a way of structuring a distributed software system so that different responsibilities are separated into distinct layers (tiers). The typical three tiers are:

1. **Presentation / Client Tier**
   The user interface. This could be a desktop application, web browser, mobile app, or thin client. It's the front-end that users interact with.
2. **Application / Logic / Business Tier**
   Business logic lives here: processing client requests, applying rules, orchestrating actions, validation, etc. This tier mediates between presentation and data tiers.
3. **Data / Storage Tier**
   Responsible for data persistence and retrieval (databases, file systems, etc.). It stores the data, manages queries, transactions, backups.



Data / Request Flow

Here's a typical request lifecycle in a three-tier system:

1. **User Action**
   The user interacts with the presentation tier (e.g. filling a form, clicking a button).
2. **Request Sent**
   The presentation tier sends a structured request (could be over sockets, HTTP, RPC, etc.) to the application tier. The request includes all needed parameters.
3. **Business Logic Processing**
   The application tier receives this request, validates it, applies business rules, maybe performs computations, aggregates results.
4. **Database Interaction**
   If data is needed (read or write), the application tier sends queries/commands to the data tier.
5. **Data Response**
   The data tier fetches/manipulates the data and returns result(s) to application tier.
6. **Response Formatting & Return**
   The application tier formats the result (e.g. JSON, XML, objects), handles error cases, and sends response back to the presentation tier.
7. **Display to User**
   Presentation tier receives the response, shows it to user (UI update, print, etc.).

Benefits of Three-Tier Architecture

Benefit	Description
Separation of Concerns	Each tier has its distinct responsibility: UI, logic, data. Changes in one tier (e.g. UI redesign) don't necessarily force changes in other tiers.
Maintainability	Easier to maintain: debug, test, evolve parts independently. For example, updating business logic or data schema is more controlled.
Scalability	Tiers can be scaled independently. If database becomes a bottleneck, scale just that tier; if logic tier is busy, replicate application servers.
Reusability	The application (business logic) tier can serve different kinds of presentation tiers (web, mobile, desktop).
Improved Security	The data tier can be shielded; one can enforce authentication/authorization in logic tier; sensitive data is not exposed directly to clients.
Flexibility in Deployment	Tiers can be on different machines, even geographically distributed. Can use different technologies in different tiers.
Better Load Distribution	Workload (request parsing, business logic, database operations) gets divided so no single component becomes a single point of failure or performance bottleneck.

Trade-Offs / Challenges

- More complex to design and implement.
- Added latency: having more network hops (client → logic → data → logic → client) may increase response time.
- More moving parts: more servers, more services, more configuration – potential for more things to go wrong.
- Requires good protocols/interfaces between tiers (clear APIs, message format, error handling).
- Deployment, monitoring, and versioning of multiple tiers can be more work.

Implementation Techniques

Techniques, patterns, and considerations when implementing a multi-/three-tier system (especially for socket-based or networked systems):

1. **Clear Protocol / Message Format**
   - Use structured data formats (JSON, XML, Protobuf, etc.) for communication between tiers.
   - Define standard message types / fields (e.g. action / command / request id / status / error).
   - Maybe message framing (prefix length, or delimiter) to know where each message starts/ends over sockets.
2. **Concurrency & Threading / Asynchronous Design**
   - Logic and database servers often need to handle multiple concurrent requests (multi-threading, thread pools).
   - Use locks / synchronization where shared resources are accessed.
   - Use timeouts for connections to prevent resource hanging.
3. **Error Handling & Reliability**
   - Handle invalid input / malformed messages gracefully.
   - Handle disconnections, partial failures.
   - Retries, logging.
4. **Modularity & Decoupling**
   - Each tier should expose a well defined API / interface. Presentation tier should not tightly depend on internal structure of the business logic tier.
   - Data tier should expose only necessary operations; hide internal schema if possible.
5. **Security**
   - Authentication / authorization in logic tier.
   - Secure communication channels (e.g. encryption or at least ensuring trust).
   - Avoid exposing the database directly to clients.
6. **Performance Optimizations**

- Caching: results of frequent database queries can be cached in logic/application tier or even in presentation.
- Connection pooling to database.
- Load balancing logic servers if many clients.
- Batch requests if possible.

7. **Scalability & Deployment Considerations**
   - Horizontal scaling: multiple instances of logic servers; perhaps master-slave or replicas for database.
   - Distribution: tiers can live on separate machines or different data centers.
   - Use of message queues if tasks are asynchronous.
8. **Monitoring, Logging & Metrics**
   - Each tier must log interactions, errors.
   - Collect metrics: latency, request counts, error rates.
   - Instruments to help debug across tiers (Tracing).

Use Cases

- Web applications (frontend → API server → database)
- Mobile apps communicating with backend servers for services
- Enterprise systems (e.g. order processing, banking)
- Any system where business rules / logic is central and you want to separate data storage / UI concerns.

School of Computing Sciences
Pak-Austria Fachhochschule: Institute of Applied Sciences and Technology, Haripur, Pakistan

Demonstration

tier_1_database_server.py

```
import socket
import threading
import sqlite3
import json
import os

HOST = "127.0.0.1"
PORT = 6000
DB_FILE = "students.db"

# Initialize database
def init_db():
    create_table = """
    CREATE TABLE IF NOT EXISTS students (
        id TEXT PRIMARY KEY,
        name TEXT NOT NULL,
        gpa REAL
    );
    """
    default_data = [
        ("1001", "Alice", 3.8),
        ("1002", "Bob", 3.5),
        ("1003", "Charlie", 3.9)
    ]
    conn = sqlite3.connect(DB_FILE)
    cur = conn.cursor()
    cur.execute(create_table)
    cur.execute("SELECT COUNT(*) FROM students")
    if cur.fetchone()[0] == 0:
        cur.executemany("INSERT INTO students VALUES (?, ?, ?)", default_data)
    conn.commit()
    conn.close()

def handle_request(conn, addr):
    print(f"[DB] Connection from {addr}")
    conn_db = sqlite3.connect(DB_FILE)
    cur = conn_db.cursor()
    try:
        data = conn.recv(4096).decode("utf-8")
        if not data:
            return

        request = json.loads(data)
        action = request.get("action")

        if action == "get_student":
```

```
            student_id = request.get("id")
            cur.execute("SELECT name, gpa FROM students WHERE id = ?", (student_id,))
            row = cur.fetchone()
            if row:
                response = {"status": "ok", "data": {"id": student_id, "name": row[0], "gpa":
row[1]}}
            else:
                response = {"status": "error", "message": "Student not found"}

        elif action == "list_all":
            cur.execute("SELECT id, name, gpa FROM students")
            rows = cur.fetchall()
            response = {"status": "ok", "data": [{"id": r[0], "name": r[1], "gpa": r[2]} for r in
rows]}

        else:
            response = {"status": "error", "message": "Unknown action"}

        conn.send(json.dumps(response).encode("utf-8"))

    except Exception as e:
        conn.send(json.dumps({"status": "error", "message": str(e)}).encode("utf-8"))
    finally:
        conn_db.close()
        conn.close()
        print(f"[DB] Closed connection with {addr}")

def main():
    init_db()
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.bind((HOST, PORT))
        s.listen(5)
        print(f"[DB] Database server running on {HOST}:{PORT}")
        while True:
            conn, addr = s.accept()
            threading.Thread(target=handle_request, args=(conn, addr), daemon=True).start()

if __name__ == "__main__":
    main()
```

tier_2_application_server.py

```python
import socket
import threading
import json

HOST = "127.0.0.1"
PORT = 5000
DB_HOST = "127.0.0.1"
DB_PORT = 6000

def forward_to_database(request_json):
    """Send JSON to database server and return JSON response."""
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as db_sock:
        db_sock.connect((DB_HOST, DB_PORT))
        db_sock.send(json.dumps(request_json).encode("utf-8"))
        response = db_sock.recv(4096).decode("utf-8")
    return json.loads(response)

def handle_client(conn, addr):
    print(f"[APP] Connected to client {addr}")
    try:
        data = conn.recv(4096).decode("utf-8")
        if not data:
            return
        request = json.loads(data)

        # Pass request to database server
        db_response = forward_to_database(request)

        # Wrap response for client
        response = {"from": "application_server", "db_response": db_response}
        conn.send(json.dumps(response).encode("utf-8"))

    except Exception as e:
        conn.send(json.dumps({"status": "error", "message": str(e)}).encode("utf-8"))
    finally:
        conn.close()
        print(f"[APP] Disconnected from client {addr}")

def main():
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.bind((HOST, PORT))
        s.listen(5)
        print(f"[APP] Application server running on {HOST}:{PORT}")
        while True:
            conn, addr = s.accept()
            threading.Thread(target=handle_client, args=(conn, addr), daemon=True).start()

if __name__ == "__main__":
    main()
```

tier_3_client.py

```python
import socket
import json

HOST = "127.0.0.1"
PORT = 5000

def send_request(request):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((HOST, PORT))
        s.send(json.dumps(request).encode("utf-8"))
        response = s.recv(4096).decode("utf-8")
        return json.loads(response)

def main():
    print("=== Student Info Client (JSON-based) ===")
    while True:

        student_id = input("Enter Student ID: ").strip()
        request = {"action": "get_student", "id": student_id}

        response = send_request(request)
        print("\n[Server Response]:")
        print(json.dumps(response, indent=4))

if __name__ == "__main__":
    main()
```

Output

Database Server

```
jarullah@saturn:lab07 | $ clear
jarullah@saturn:lab07 | $ python3 tier_1_db_server.py
[DB] Database server running on 127.0.0.1:6000
[DB] Connection from ('127.0.0.1', 51159)
[DB] Closed connection with ('127.0.0.1', 51159)
[DB] Connection from ('127.0.0.1', 51161)
[DB] Closed connection with ('127.0.0.1', 51161)
```

Application Server

```
jarullah@saturn:lab07 | $ clear
jarullah@saturn:lab07 | $ python3 tier_2_application_server.py
[APP] Application server running on 127.0.0.1:5000
[APP] Connected to client ('127.0.0.1', 51158)
[APP] Disconnected from client ('127.0.0.1', 51158)
[APP] Connected to client ('127.0.0.1', 51160)
[APP] Disconnected from client ('127.0.0.1', 51160)
[APP] Connected to client ('127.0.0.1', 51162)
[APP] Disconnected from client ('127.0.0.1', 51162)
[APP] Disconnected from client ('127.0.0.1', 51162)
```

Client

```
jarullah@saturn:lab07 | $ clear
jarullah@saturn:lab07 | $ python3 tier_3_client.py
=== Student Info Client ===
Enter Student ID: 1001

[Server Response]:
{
    "from": "application_server",
    "db_response": {
        "status": "ok",
        "data": {
            "id": "1001",
            "name": "Alice",
            "gpa": 3.8
        }
    }
}
Enter Student ID: 1002

[Server Response]:
{
    "from": "application_server",
    "db_response": {
        "status": "ok",
        "data": {
            "id": "1002",
            "name": "Bob",
            "gpa": 3.5
        }
    }
}
```

Client (invalid id and exit command)

```
Enter Student ID: 100000

[Server Response]:
{
    "from": "application_server",
    "db_response": {
        "status": "error",
        "message": "Student not found"
    }
}
Enter Student ID: exit

[Server Response]:
{
    "from": "application_server",
    "status": "Disconnected"
}
jarullah@saturn:lab07 | $
```

School of Computing Sciences
Pak-Austria Fachhochschule: Institute of Applied Sciences and Technology, Haripur, Pakistan

TASKS

Task 01	Task:Add Registration, Login functionality to the multi-tier system
	
	• Client should be able to Add New Users through Registration.
	  ○ Allow retry at failure.
	• Client should be able to Log In to the system with valid credentials.
	  ○ Allow retries.
	
	Test:
	
	• Register 3 New students.
	• Log In all 3 simultaneously.
	• Log In with invalid account.

Task 02	Task: Add the Update and Delete functionality: • A user, once logged in, can update their information like "address", "phone number", and "email" but Not "ID", and "GPA". • A client should be able to delete their account by providing the correct credentials. Test your program: • Update the "address", and attempt to update the "GPA". • Delete a user account and try to login through it.
Task 03	Task: Implement search functionality • Client should be able to search student by name. • Have multiple students registered with the same name. Test: • Return all students matching the search term. • Search for no existant student.