Distributed Systems - COMP SCI 3012
Collaborative Session 2

Gia Bao Hoang - a1814824

Marcus Hoang - a1814303

Ran Qi - a1675122

Question 1

"Reduce blocking and you reduce latency." Discuss whether this sentence is or is not correct from the perspective of Distributed Systems and provide examples to explain your answer. Carefully explain all of your terms.

In distributed systems, the concepts of blocking and latency are intertwined. Here is a case where reduce blocking lead to reduce latency:

- Network Communication: In distributed architectures, components often communicate over networks. If a component sends a synchronous request to another and waits for a response, it's blocked during that period. This waiting time contributes to the overall latency. By adopting non-blocking or asynchronous communication patterns, we can often reduce this waiting time, thereby reducing latency.
- Example: Consider a distributed database system where a client sends a read request. If the client waits for the data (blocking), the total operation time (latency) includes the round-trip network time and the data retrieval time. But if the client uses an asynchronous pattern, it can potentially initiate other operations while waiting, optimizing the perceived latency.

Here is a case where reduce blocking does not lead to reduce latency:

- Overhead and Complexity: While non-blocking operations can reduce latency, they introduce their own challenges. Asynchronous operations often require more complex programming patterns, error handling, and state management. This added complexity can sometimes introduce overhead, which might offset the latency benefits of non-blocking operations.
- Example: In a distributed microservices architecture, Service A might need data from Service B and Service C. If Service A sends asynchronous requests to both but doesn't handle the responses efficiently, it might end up with increased processing overhead, potentially affecting the overall latency.

The principle "Reduce blocking and you reduce latency" holds merit in many distributed system scenarios. However, it's not a silver bullet. While reducing blocking can often lead to latency improvements, the associated complexities and potential overheads mean that system designers must carefully weigh the pros and cons in each specific context.

Question 2

Which latency reduction and latency hiding method would you choose and when?

Choosing the right reduction and latency hiding method is crucial to the performance of a distributed system. The choices should be made based on the specific requirement of the application, the hardware and software architecture of the distributed system, and the trade-offs you are willing to make. Some of the common latency reduction/hiding methods are:

1) Caching: Store data that is frequently requested in a place that is easy and fast to retrieve so that expensive remote calls can be avoided. Used when there are data that doesn't get changed very often but requested frequently. In the case of Assignment 2, this might not be a good choice as the weather data gets updated on a rather high frequency and accuracy is a high priority.
2) Load balancing: balance the load evenly between multiple servers to avoid some servers get overwhelmed. Used when there are multiple service servers. In the case of assignment 2, there will be multiple aggregation servers that handle requests from clients. Making sure that the work load is evenly distributed among the servers is important for latency reduction.
3) Replication: having multiple copies of the same data at different locations so that the latency can be reduced by retrieving data from the closest location. It also helps when high availability and fault tolerance is required. In the case of assignment 2, the aggregations servers will be run from the same machine so I guess this method won't apply.
4) Using asynchronous calls: use futures and promises to handle the results and errors of operations so that multiple calls can be issued at the same time.Used when there are lots of independent tasks that can be executed separately and the subsequent calls don't depend on the results of previous calls.

Question 3

Write the pseudo-code for the client implementation of a failure mechanism. Think about all the possible failure scenarios.

Scenarios that could result in failure on the client side are:

1) Network issues: client cannot connect to the server
    - solution: set timeout
2) Server being unavailable: client cannot connect to server

    - solution: set timeout
3) Server-side errors: client can connect to server but not getting the expected response. e.g. requests not correctly parsed.
    - solution: handle invalid response.

Use try-catch blocks to capture failures and handle them gracefully. Pseudo code:

```
try {
        create socket
        set timeout = 30 seconds
        initiate connection.
        response = sendRequest()
        if (IsValidResponse(response)) {
                showWeather(response)
        } else {
                handleInvalidResponse()
        }
} catch (ConnectException e) {
        handleConnectionFailue()
} catch (IOException e) {
        handleIOFailure()
}
```

Question 4

You were just asked to design and implement an architecture similar to BOINC Links to an external site.. List and discuss three trade-offs that you think will have to be made.

Architecture components:

1. Client Application:

- Downloads scientific computing jobs.

- Runs jobs in the background.

- Reports results back to the server.

2. Server Backend:

- Manages job distribution.

- Validates and stores results.

- Provides APIs for project management.

3. Database:

- Stores information about jobs, results, and users

4. Web Interface:

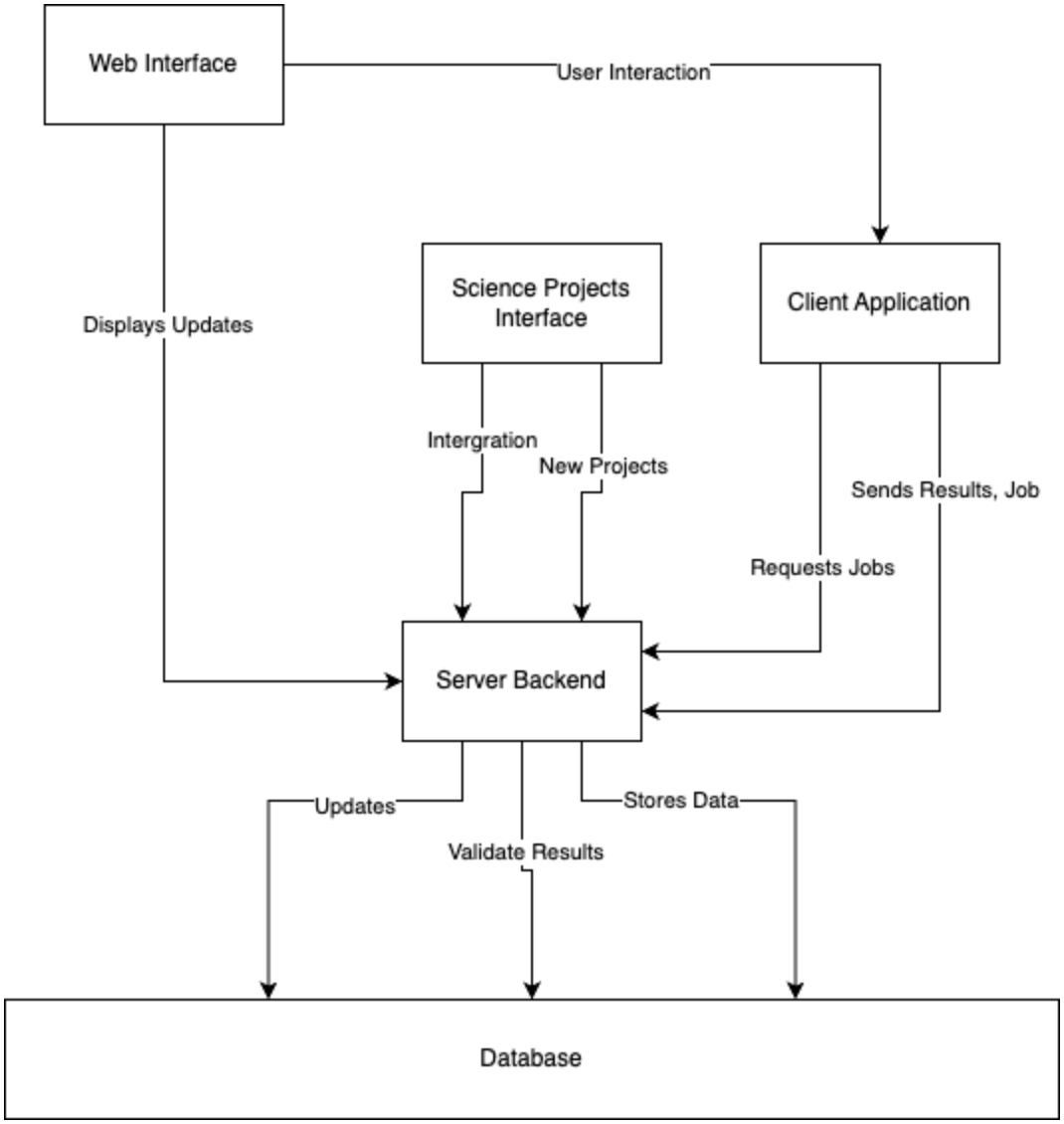
- Allows users to choose specific projects.

- Provides updates and news related to BOINC projects.

5. Science Projects Interface:

- Allows integration of various scientific projects.

- Provides APIs for adding new projects.



Three Trade-offs:

1. Scalability vs Complexity

As the number of clients and projects increases, the system will need to scale horizontally or vertically, which could add complexity in terms of management and costs.

2. Security vs Usability

Ensuring the security of scientific data and user information may require implementing strict security measures, which could make the system less user-friendly.

3. GETAccuracy vs Speed

Ensuring that the computations are accurate may require additional validation steps, which could slow down the overall processing speed.

Question 5

Present and discuss your pseudo-code for testing Assignment 2. What test cases are you considering? How would you implement each of them?

Pseudo-code for testing Assignment 2:

```
Initialize Test Suite for the Entire Project:

1. Test JSONHandler Functionality
    1.1 Test the 'convertTextToJSON' method for successful conversion and edge cases.
    1.2 Test the 'convertJSONToText' method for successful conversion and edge cases.
    1.3 Test the 'extractJSONContent' method for both valid and invalid inputs.
    1.4 Test the 'toJSONObject' method by passing valid and invalid JSON strings and assessing the outcome.

2. Test ContentServer Functionality
    2.1 Ensure ContentServer starts with appropriate settings.
    2.2 Validate ContentServer's ability to serve content upon request.
    2.3 Assess if ContentServer's Lamport clock increments correctly upon operations.
    2.4 Simulate various failure scenarios to check resilience and error handling.

3. Test GETClient Functionality
    3.1 Validate GETClient's ability to make a successful GET request to the AggregationServer.
    3.2 Compare the data retrieved by GETClient against expected data to ensure consistency.
    3.3 Emulate conditions where the AggregationServer is unavailable and assess GETClient's response.

4. Test AggregationServer Basic Functionality
```

4.1 Check initialization process with both default and user-defined port numbers.
4.2 Validate the proper shutdown mechanism and resource release.
4.3 Ensure it processes both PUT and GET requests effectively.
4.4 Confirm it returns expected HTTP status codes for various scenarios.

5. Test AggregationServer Advanced Functionality
5.1 Ensure Lamport Clock is maintained and updated appropriately with operations.
5.2 Validate its ability to store, retrieve, and manage entries in its intermediate data storage.
5.3 Check mechanisms to clear old or stale entries.
5.4 Simulate crash scenarios and validate recovery to the last known stable state.

6. Integration Tests
6.1 Sequentially start ContentServer, AggregationServer, and GETClient, ensuring smooth initialization.
6.2 From ContentServer, make PUT requests and then check if the AggregationServer stores the data accurately.
6.3 Using GETClient, retrieve data and compare against the initial PUT data from ContentServer to ensure consistency.
6.4 Throughout the integrated operations, monitor Lamport clocks across all components to ensure synchronization.

End of Test Suite.

Test Cases for the Project and Their Implementation:

1. JSONHandler Tests:

a. Test Case: extractJSONContent Functionality

Goal: To ensure that the function extracts the content of the first pair of {} correctly from a given string.

Implementation:

- Input: Craft strings with multiple {}, nested {}, no {}, and just one pair of {}.
- Use JUnit's assertEquals to compare the extracted content against the expected output.

b. Test Case: toJSONObject Conversion

Goal: To ensure the correct conversion of a valid JSON string into a JSONObject.

Implementation:

- Input: Provide valid and invalid JSON strings.
- Use JUnit's assertions to check if the conversion is successful and if it fails gracefully for invalid inputs.

2. ContentServer Tests:

a. Test Case: Server Initialization

Goal: Ensure that the server starts and initializes all required components, including the Lamport clock.

Implementation:

- Use JUnit's lifecycle methods (@Before, @After) to start and shut down the server.
- Assert that all initialization tasks have been completed successfully.

b. Test Case: Content Serving

Goal: Ensure the server responds correctly to a GET request.

Implementation:

- Mock a GET request to the server.
- Assert the response matches the expected content.

c. Test Case: Clock Update on PUT

Goal: Ensure the Lamport clock's time increments correctly when a PUT request is made.

Implementation:

- Mock a PUT request.
- Check the Lamport clock's time before and after the request to verify it increments as expected.

3. GETClient Tests:

a. Test Case: Data Retrieval

Goal: Ensure the client retrieves data correctly from the server.

Implementation:

- Simulate a GET request from the client.
- Assert the returned data matches the expected server response.

b. Test Case: Error Handling on Server Unavailability

Goal: Ensure the client handles server unavailability gracefully.

Implementation:

- Use the network operation abstraction to simulate server unavailability.
- Make a request from the client and assert that it handles the error gracefully.

4. AggregationServer Tests:

a. Test Case: Aggregation Server Initialization

Goal: Ensure the AggregationServer starts correctly and initializes all its components, including the network handler and Lamport clock.

Implementation:

- Use JUnit's lifecycle methods (@Before, @After) to start and shut down the server.
- Assert that the network handler, Lamport clock, and other server components are initialized as expected.

b. Test Case: Data Aggregation and Storage

Goal: Ensure the AggregationServer receives, aggregates, and stores incoming data correctly.

Implementation:

- Send mock PUT requests with data to the AggregationServer.
- Assert that the server correctly stores and aggregates the data.
- Check the data store to ensure the data is stored correctly.

c. Test Case: Retrieving Aggregated Data

Goal: Ensure the server responds with correct aggregated data to a GET request.

Implementation:

- Mock a GET request to the AggregationServer after sending some PUT requests.
- Assert that the response contains the aggregated data from the PUT requests.

d. Test Case: Lamport Clock Update on Data Receipt

Goal: Ensure that the Lamport clock of the AggregationServer increments correctly when data is received.

Implementation:

- Send a mock PUT request with data and a Lamport timestamp.
- Check the AggregationServer's Lamport clock before and after the data receipt to verify it updates correctly.

e. Test Case: Error Handling for Malformed Data

Goal: Ensure the server handles malformed or invalid data gracefully.

Implementation:

- Send mock PUT requests with malformed data (e.g., invalid JSON, missing data fields).
- Assert that the server handles errors gracefully, possibly responding with error codes or messages.

f. Test Case: Shutdown Mechanism

Goal: Ensure the AggregationServer can be shut down gracefully using the "SHUTDOWN" input command.

Implementation:

- Start the AggregationServer and then send a "SHUTDOWN" command.
- Assert that the server stops its operations and releases any resources or connections.

g. Test Case: Integration with JSONHandler

Goal: Verify the correct utilization of JSONHandler methods within the AggregationServer.

Implementation:

- Send data to the AggregationServer that requires JSON parsing and handling.
- Use spies or mocks for JSONHandler methods to assert they are called and used correctly.

5. Integration Tests:

a. Test Case: Component Interaction

Goal: Ensure the ContentServer, AggregationServer, and GETClient interact correctly.

Implementation:

- Start the ContentServer, AggregationServer, and GETClient.
- Simulate a flow of requests and interactions between them.
- Assert the data flows as expected between components.

b. Test Case: Lamport Clock Consistency Across Components

Goal: Ensure Lamport clocks update correctly across components during interactions.

Implementation:

- As the systems interact, especially during PUT requests and data retrieval, check the state of the Lamport clocks in each component.
- Assert that the times are consistent and updated correctly.