Remote Method Invocation (RMI) Documentation

Aim

The aim of this document is to provide a comprehensive overview of Remote Method Invocation (RMI), detailing its theoretical background, operational mechanics, performance analysis, output evaluation, bonus extensions, and final conclusions.

Theory

RMI is a Java-specific API that allows Java objects to invoke methods on remote objects. It abstracts the complexities of network communication, enabling seamless interaction between distributed Java applications while leveraging the benefits of object-oriented programming.

Working

1. **Remote Interface**: RMI starts with defining a remote interface that declares the methods that can be called remotely.
2. **Object Implementation**: A class implements this remote interface, providing the actual functionality for the remote methods.
3. **Registry**: The server registers the remote object with the RMI registry, which allows clients to look up the remote object by name.
4. **Client Interaction**: The client obtains a reference to the remote object via the registry and invokes methods on it as if it were a local object.
5. **Serialization**: Java serialization is used to convert the remote objects into a byte stream for transmission over the network.
6. **Communication**: RMI uses its protocol (JRMP) for communication, handling the transmission of method calls and responses.

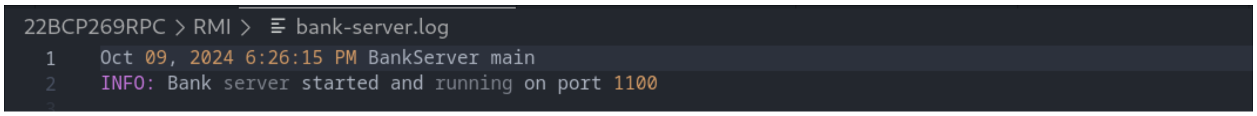
Performance Analysis

- **Throughput**: RMI supports multiple concurrent method calls, enhancing throughput but can become bottlenecked under heavy loads.
- **Latency**: The overhead of Java serialization may introduce additional latency compared to more optimized RPC serialization methods.
- **Scalability**: RMI can scale well within Java environments but may face challenges in mixed-language systems.

Output Analysis

Output analysis focuses on the correctness and performance of method invocations. Each remote method call should return the expected results efficiently. Testing should include response time measurements and logging of method invocation details.

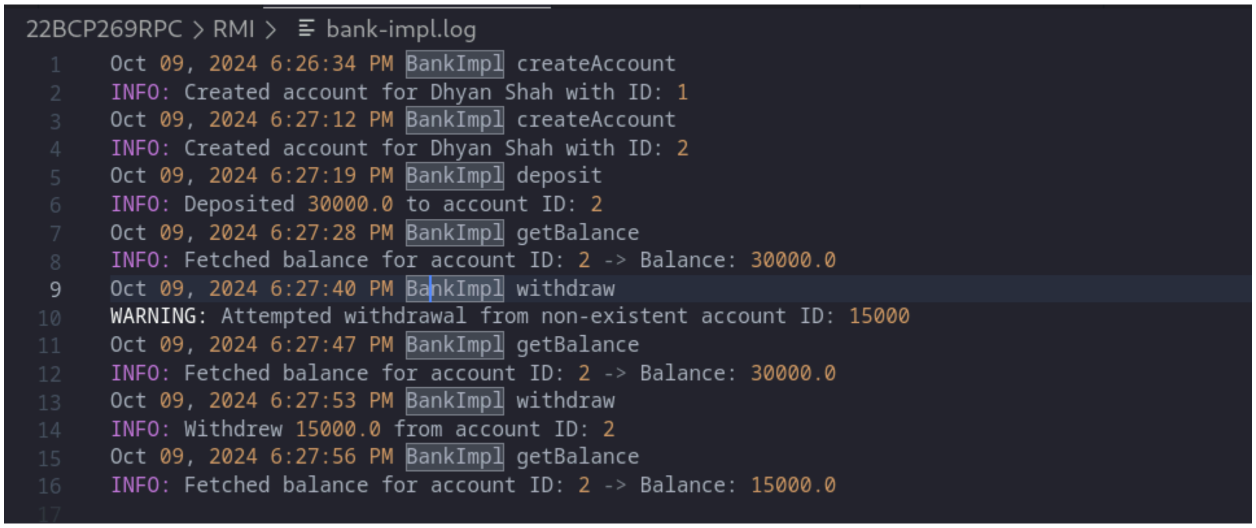
BankServer.log



22BCP269RPC > RMI > bank-server.log

```
Oct 09, 2024 6:26:15 PM BankServer main
INFO: Bank server started and running on port 1100
```

BankImpl.log



22BCP269RPC > RMI > bank-impl.log

```
Oct 09, 2024 6:26:34 PM BankImpl createAccount
INFO: Created account for Dhyan Shah with ID: 1
Oct 09, 2024 6:27:12 PM BankImpl createAccount
INFO: Created account for Dhyan Shah with ID: 2
Oct 09, 2024 6:27:19 PM BankImpl deposit
INFO: Deposited 30000.0 to account ID: 2
Oct 09, 2024 6:27:28 PM BankImpl getBalance
INFO: Fetched balance for account ID: 2 -> Balance: 30000.0
Oct 09, 2024 6:27:40 PM BankImpl withdraw
WARNING: Attempted withdrawal from non-existent account ID: 15000
Oct 09, 2024 6:27:47 PM BankImpl getBalance
INFO: Fetched balance for account ID: 2 -> Balance: 30000.0
Oct 09, 2024 6:27:53 PM BankImpl withdraw
INFO: Withdrew 15000.0 from account ID: 2
Oct 09, 2024 6:27:56 PM BankImpl getBalance
INFO: Fetched balance for account ID: 2 -> Balance: 15000.0
```

Bonus Extensions

- **Fault Tolerance**: Implementing retries on failed method calls to ensure reliability.
- **Secure Communication**: Utilizing SSL/TLS to secure data exchanged between clients and servers.
- **Logging Mechanism**: Implementing logging to track method calls and responses, aiding in debugging and performance monitoring.

Conclusion

RMI provides a powerful mechanism for Java applications to communicate across networks, allowing for remote method invocations as if they were local. While RMI simplifies the distribution of Java objects, its limitations in language support and potential performance overheads necessitate careful consideration in system design. RMI remains a viable choice for Java-centric applications, particularly in enterprise environments.