MULTIMEDIA UNIVERSITY OF KENYA

FACULTY OF COMPUTING INFORMATION TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

COURSE: *SOFTWARE ENGINEERING*

Name: Danny Ngatia Gitonga

Admission no. : CIT-227-O67/2024

Operating Systems – II Review Questions Week 3 - Answer all the Questions

Remote Procedure Calls

1. Evaluate the effectiveness of RPCs compared to message-passing systems for implementing distributed applications. Justify your answer based on performance, transparency, and error handling.
   \>> ***RPC is more developer-friendly and transparent but less performant and harder to make robust under failure. Message-passing is better for performance-critical or highly reliable custom protocols.***

	RPCS	***Message passing systems***
Performance:	*Reduce overhead by adding abstraction layers allowing direct procedure invocation*	*provide better performance in asynchronous, loosely-coupled scenarios where buffering and queuing are beneficial*
Transparency:	*Hide network complexity hence invoking procedural calls as if they were local*	*Exposes messages hence requiring explicit handling of message formats, endpoints and sequencing hence reducing transparency*
Error handling:	*struggle with network failures, requiring additional mechanisms for handling partial failures, timeouts, and idempotency*	*handle errors more gracefully through timeouts, acknowledgments, and retry mechanisms built into the messaging infrastructure*

2. Given two implementations of an RPC system—one using static binding and the other dynamic binding—which would you recommend for a highly scalable cloud-based service, and why?

>>Static binding: Port/endpoint resolved at compile or load time.
>>Dynamic binding: matchmaker/registry (e.g., service discovery via DNS, etcd, or gRPC's name resolver).
>> I would recommend dynamic binding since it offers essential cloud-native scalable services(eg. Kubernetes, microservices) where endpoints change frequently and services auto scale

3. Critically assess the trade-offs between exactly-once and at-least-once invocation semantics in RPC. Which would you recommend for financial transaction systems, and why?

>> At-least-once: Guarantees delivery but may duplicate requests (e.g., due to timeouts). Simpler to implement but risks double-charging.
Exactly-once: Ensures a request is processed once and only once. Requires idempotency keys, deduplication logs, or transactional coordination (e.g., two-phase commit).

>> I would recommend Exactly-once as mandatory for financial transactions because it preserves consistency and correctness, despite higher complexity.

4. An organization is choosing between implementing gRPC and a custom lightweight RPC framework. As a system analyst, evaluate the two options based on extensibility, maintainability, and support for multiple languages.

	gRPC	***Custom lightweight RPC framework***
Extensibility:	***High (interceptors, middleware, streaming)***	***Limited to in-house design***
Maintainability:	***Excellent (mature, well-documented, community support)***	***High long-term cost (testing, bug fixes, docs)***
Support for multiple languages:	***First-class support (Protobuf + 10+ languages)***	***Requires manual bindings per language***

5. Judge the suitability of RPC in a real-time system (e.g., embedded control system in aviation). What are the critical limitations, and would you recommend using RPC in such a scenario?

>> Critical limitations:

- ✓ ***Unbounded latency: Network delays, retries, or GC pauses violate hard deadlines.***
- ✓ ***Non-determinism: RPC relies on OS/network stacks not designed for real-time guarantees.***
- ✓ ***Failure modes: Timeouts or lost messages can't be tolerated in safety-critical contexts.***

>> I would not recommend using RPC as it is **not suitable for hard real-time. Only acceptable in soft real-time, with caution.**

6. You are tasked with designing a distributed file storage system. Evaluate whether RPC or RESTful APIs would be more appropriate for client-server communication. Justify your choice.

>> RPC (e.g., NFS, gRPC):

- ***Supports binary protocols, efficient for file ops (read/write blocks).***
- ***Enables stateful sessions, strong consistency, and low-latency batch operations.***

>> REST:

- ***Stateless, text-based (JSON/XML), higher overhead.***
- ***Better for web clients, caching (HTTP), and loose coupling—but poor for streaming or partial updates.***

>> Recommendation: RPC would be more appropriate for client-server communications as it is more appropriate for performance-sensitive file storage (e.g., cloud storage backends like Ceph or HDFS use RPC-like protocols).

7. Evaluate the fault tolerance mechanisms typically used in RPC systems. Are they sufficient for mission-critical applications? What would you recommend improving?

- ***Mechanisms used are like : retries, timeouts, duplicate suppression, client-side caching***
  They help, but aren't enough for mission-critical apps. Recommend adding checkpointing, replication, and consensus protocols.

8. Assess the impact of asynchronous RPC on system responsiveness and resource utilization in a microservices architecture. Would you recommend asynchronous over synchronous RPC in such contexts?

>> Impact:

- ***Responsiveness: Clients don't block; better user experience.***
- ***Resource utilization: Threads/connections freed quickly; higher throughput.***
- ***Complexity: Requires callbacks, futures, or event loops; harder debugging.***

>> Yes I would recommend use of asynchronous RPC in microservices to avoid cascading failures and improve scalability (e.g., gRPC async, message queues like Kafka for decoupling) especially where responsiveness matters.

9. A university student project team wants to use RPC over HTTP to develop a distributed voting system. Critique their approach and suggest whether this is advisable. Support your judgment with reasoning.

>> RPC over HTTP can be a suitable communication protocol between trusted or internal components of the voting system (e.g., a front-end server communicating with an internal database service)
However, it is not advisable for the core voting submission mechanism to the central component without incorporating higher-level, specialized protocols. The core challenges of a distributed voting system are not merely communication efficiency, but securing the fundamental properties of the election itself:

10. Compare and evaluate the use of middleware frameworks (like CORBA, Java RMI, or gRPC) for implementing RPC in a distributed e-commerce platform. Which framework would you recommend, and on what basis?

Frameworks	**pros**	**cons**
COBRA	*Language-neutral, mature*	*Complex, legacy, poor tooling*
Java RMI	*Simple for Java-only apps*	*Java-only, no web support*
gRPC	*High performance, HTTP/2, Protobuf, multi-language, streaming*	*Steeper learning curve*

>> I would recommend gRPC—best balance of performance, scalability, and polyglot support for modern e-commerce (e.g., order, payment, inventory microservices).

Distributed Processing

1. Evaluate the design principles of a distributed operating system (DOS). Which principle (e.g., transparency, fault tolerance, scalability) do you consider most critical for system performance, and why?

>>Fault tolerance is most critical — without it, failures bring down the whole system. Scalability and transparency matter, but reliability is fundamental.

2. Given a choice between a centralized system and a distributed operating system for a smart city infrastructure project, which would you recommend? Justify your recommendation based on system requirements such as fault tolerance, scalability, and responsiveness.

>> I would recommend Distributed OS as it is essential for resilience and scale in smart infrastructure, offering:

- ***Fault tolerance: Local failures don't cripple the whole city.***
- ***Scalability: Add traffic, energy, or surveillance nodes seamlessly.***
- ***Responsiveness: Edge processing reduces latency (e.g., real-time traffic control).***

3. Critique the rationale for adopting distributed systems in large-scale enterprises. Are the benefits (e.g., resource sharing, modular growth) always worth the increased complexity and overhead?

>> Benefits: resource sharing, modular growth, reliability.
>> Downside: higher complexity/overhead.
>> Yeah, still worth it for large-scale enterprises where resilience outweighs complexity.

4. Assess the effectiveness of location transparency in distributed operating systems. When could this feature become a liability rather than an advantage?

Advantage: Users/apps don't need to know where data/services reside.
Liability:

- ***Performance: Accessing a distant replica increases latency.***
- ***Compliance: Data residency laws (e.g., GDPR) require knowing physical location.***
- ***Debugging: Obscures root cause of failures.***

Transparency should be optional—allow apps to opt out for performance or legal reasons.

5. You are tasked with building a distributed application across a heterogeneous network of devices. Evaluate how the principles of distributed OS design (such as transparency and concurrency) help or hinder your objective.

Helps:

- ***Transparency abstracts hardware differences (CPU, OS).***
- ***Concurrency enables parallel task execution across devices.***

Hinders:

- ***Heterogeneity complicates data representation (endianness, word size)—requires standard formats (e.g., XDR, Protobuf).***
- ***Resource disparity (e.g., IoT vs. server) challenges load balancing.***

6. Judge the appropriateness of using a distributed operating system to manage resources in a university campus network. What limitations or risks should be considered?

>> Distributed OS helps manage shared resources.
>> Risks: admin complexity, higher maintenance costs, and possible security issues.

7. Evaluate the role of fault tolerance and recovery mechanisms in a distributed OS versus a traditional centralized OS. Are the mechanisms in distributed OSs sufficient for mission-critical systems?

>> Centralized: Backups and RAID help, but single point of failure remains.
>> Distributed: Replication, consensus, and automatic failover provide high availability.
>> Modern distributed OSs (e.g., Google Borg, Kubernetes) are sufficient when designed with Byzantine fault tolerance, quorum writes, and chaos engineering.

8. Compare and assess two architectures for distributed systems: peer-to-peer (P2P) and client-server. Which architecture better supports the principles of a distributed OS and under what conditions?

Aspect	*P2P*	*Client-server*
Transparency	*Harder (dynamic nodes)*	*Easier (central naming)*
Scalability	*Highly scalable*	*Limited by server*
Resource sharing	*Organic, efficient*	*Controlled*
Fault tolerance	*Resilient (no central node)*	*Server = SPOF*

>> For distributed OS principles, P2P is usually better unless strong central control is needed.

9. A startup is considering implementing a distributed system for its logistics operations. Evaluate whether this decision is suitable at their current scale and justify what conditions must be met for distributed processing to be beneficial.

>> Not suitable initially. Startups need speed, simplicity, and low cost. A monolith or simple cloud app (e.g., AWS Lambda + RDS) suffices.

Conditions to adopt distributed processing:

- ***10K daily orders***
- ***Need for real-time tracking across regions***
- ***SLA requiring 99.9% uptime***
- ***Multiple warehouses/suppliers***

10. Assess the trade-offs between performance and transparency in distributed operating systems. Should system designers prioritize one over the other? Defend your position with examples.

    >> Trade-off is unavoidable. For system design, balance is needed, but performance should take priority in critical systems (e.g., banking), while transparency can be favored in user-facing systems.