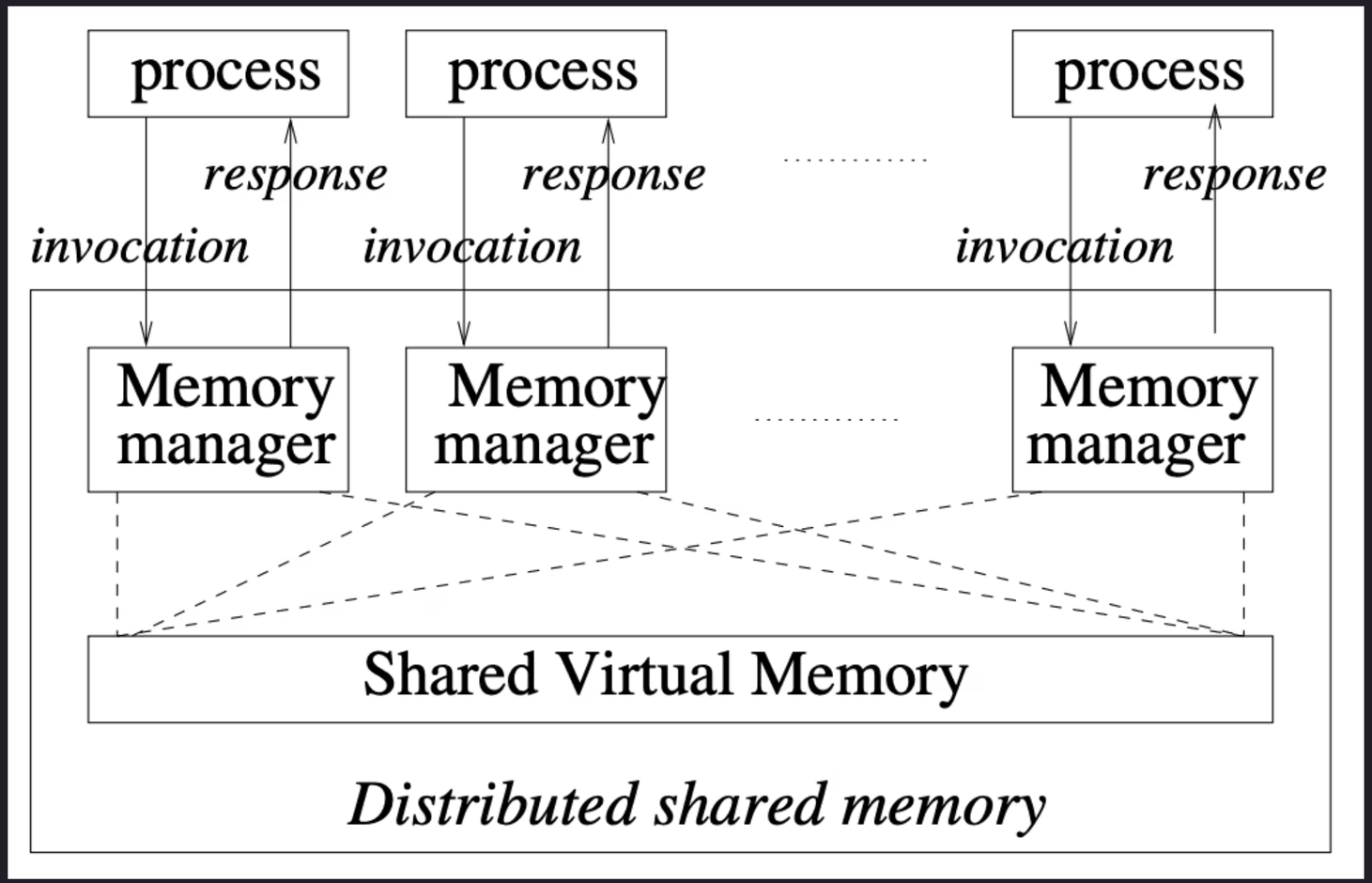
DISTRIBUTED SHARED MEMORY

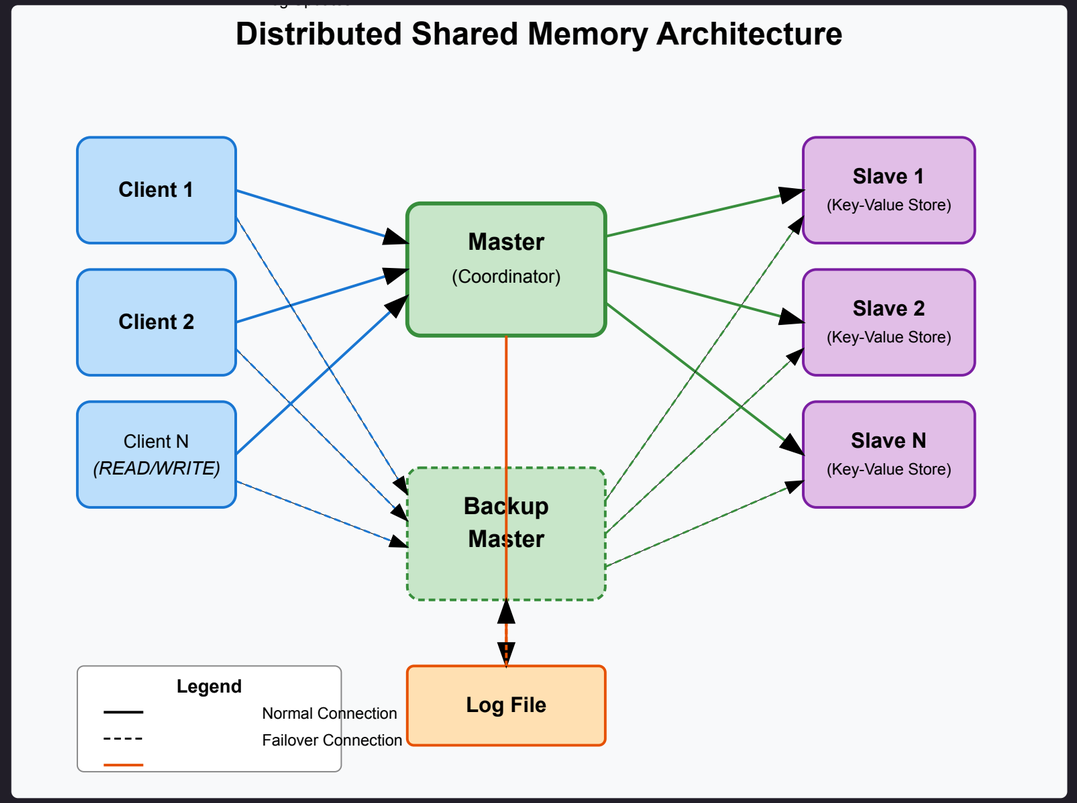
ROLL NUMBER	NAME
2021101047	Manuj Garg
2021101095	Pranav Gupta
2021101113	Gowlapalli Rohit



Try Pitch

Introduction

The Distributed Shared Memory is a system that is used in order to store key-value pairs locally across multiple nodes, so that the master node no more remains a single point of failure. The master node acts as a metadata server keeping the info of the slaves who have the value corresponding to a particular key and does not itself stores the value of any key in the system. The system demonstrates READ and WRITE operations between multiple clients and slaves, coordinated by a master server, using Golang and sockets on a single machine. The system emphasizes fault tolerance through ACK bits and data consistency by distributing WRITE operations to a subset of slaves.



Try Pitch

Files Structure and Functionality

- **client.go:** Manages the interaction of clients initiating READ and WRITE requests to the master and waits for the Master for a particular time, after which it times out and connects to the Backup Master if required.
- **master.go**: Coordinates all operations between clients and slaves, maintaining lists of active slaves and keys, and the set of slaves storing the value for all keys.
- **backup_master.go**: Performs the same functionality as the master, and is required only in the case when the master fails.
- **slave.go**: Handles storage and processing of data, sending ACK bits to the master/backup master after operations.

Try Pitch

Client Overall Functionality

Manages client interactions, allowing users to issue READ and WRITE commands. The client performs the following tasks:

- Connects to the master via a socket request at the master port.
- Sends commands based on user input and displays responses from the master.
- Has a particular timeout interval, after which it assumes that the master is down, and instead connects to the backup master.

Try Pitch

Master's Read Operation

- For READ requests, the master checks its records to determine which slaves last stored the data associated with the requested key.
- In case it is a new key (i.e. a key not in the master's records), then it first asks all the slaves whether they have the value for that particular key. If yes, then the master keeps a record and maintains a list of slaves who replied as the slaves storing that key.
- It retrieves the value from one of these slaves. If multiple writes have occurred, the value retrieved is the one from the most recent successful write, ensuring that the client always receives the latest data.
- It attempts to retrieve the value from one of these slaves. If the initial attempt fails to receive a response (e.g., due to a slave failure or network issues), the master employs a continuous pinging mechanism. This process involves repeatedly sending the READ request until a response is received.
- If the key is not found in any slave (e.g., if all slaves that stored the key have failed), the master responds with a "NOT_FOUND" message.

Try Pitch

Master's Write Operation

- When a WRITE request is received from a client, the master selects a subset of slave nodes to handle the operation. The slave nodes are selected randomly in the code.
- The master sends the WRITE command along with the data to the chosen slaves and waits for ACK bits to confirm that the operation has been successfully completed.
- If ACK bits are not received from any slave within a specified timeout, the master assumes the slave has failed and removes it from the list of active slaves. This ensures that only responsive and reliable slaves are used for future operations.

Try Pitch

Slave Operation

It performs the following tasks:

- **Registers** with the master and **waits for commands** from the master. The slaves performs wait based upon **exponential backoff** to avoid overloading the master servers.
- On receiving a **WRITE** command, stores the provided data and sends an ACK bit back to the master. The use of ACK bits allows the master to monitor the health and responsiveness of slave nodes. Non-responsive slaves are quickly identified and excluded from future operations, which helps in maintaining the overall robustness of the system and helps the system being **fault-tolerant**.
- On receiving a **READ** command, retrieves and sends the requested data back to the master.

Try Pitch

Backup Master operation

Backup Master Server

- **Function**: Provides failover redundancy for the primary master server
- **Implementation**:
  - Runs on separate port from the master.
  - Maintains identical functionality to primary master
  - Automatically takes over when primary master fails
  - Clients connect to backup when primary is unreachable
- **Advantage**: Ensures high availability of the key-value store system

Try Pitch

Consistency by Log File

- **Implementation**:
  - Every successful operation is recorded in a shared log file
  - Format: OPERATION KEY VALUE (e.g., WRITE user1 Alice)
  - The log serves as a persistent record of all state changes
  - The log is the source of truth for the distributed system
- **Benefits**:
  - Provides an ordered history of all operations
  - Enables recovery and synchronization between masters
  - Creates a durable record that survives server crashes

Try Pitch

Backup Master Synchronization

- Synchronization Process
  - **Initial Loading**:

    When the backup master starts up, it loads the existing data from the log file:

    1. The backup master calls loadDataFromLog() function
    2. It opens and reads the "kv_store.log" file that contains all operations performed by the primary master
    3. For each entry in the log file, it extracts the key and value
    4. It populates its local keyValueData map with these key-value pairs
    5. This ensures the backup has the same state as the primary master when it starts
  - **Continuous Monitoring:**

    After initialization, the backup master continuously monitors the log file for changes:

    1. The watchLogFile() function runs as a goroutine (separate thread)
    2. It periodically checks the size of the log file (typically every second)
    3. When it detects that the file has grown (new entries added):
       - It seeks to the position where it last read
       - Reads only the new entries
       - Updates its local keyValueData map with the new keys and values
    4. This ensures the backup master stays synchronized with the primary master's state in near real-time

Try Pitch

Client Failover Process

Automatic Failover

- **Connection Process:**

  When a client attempts to connect, it first tries the primary master server:
  - Makes a TCP connection attempt to the primary port
  - Uses a timeout (2 seconds) to detect if the primary is unreachable

  If the primary connection fails:
  - The client automatically attempts to connect to the backup master (port 12346)
  - Uses the same timeout mechanism to detect availability

  Connection status is returned:
  - If connected to primary: returns connection and true
  - If connected to backup: returns connection and false
  - If both fail: returns nil and false

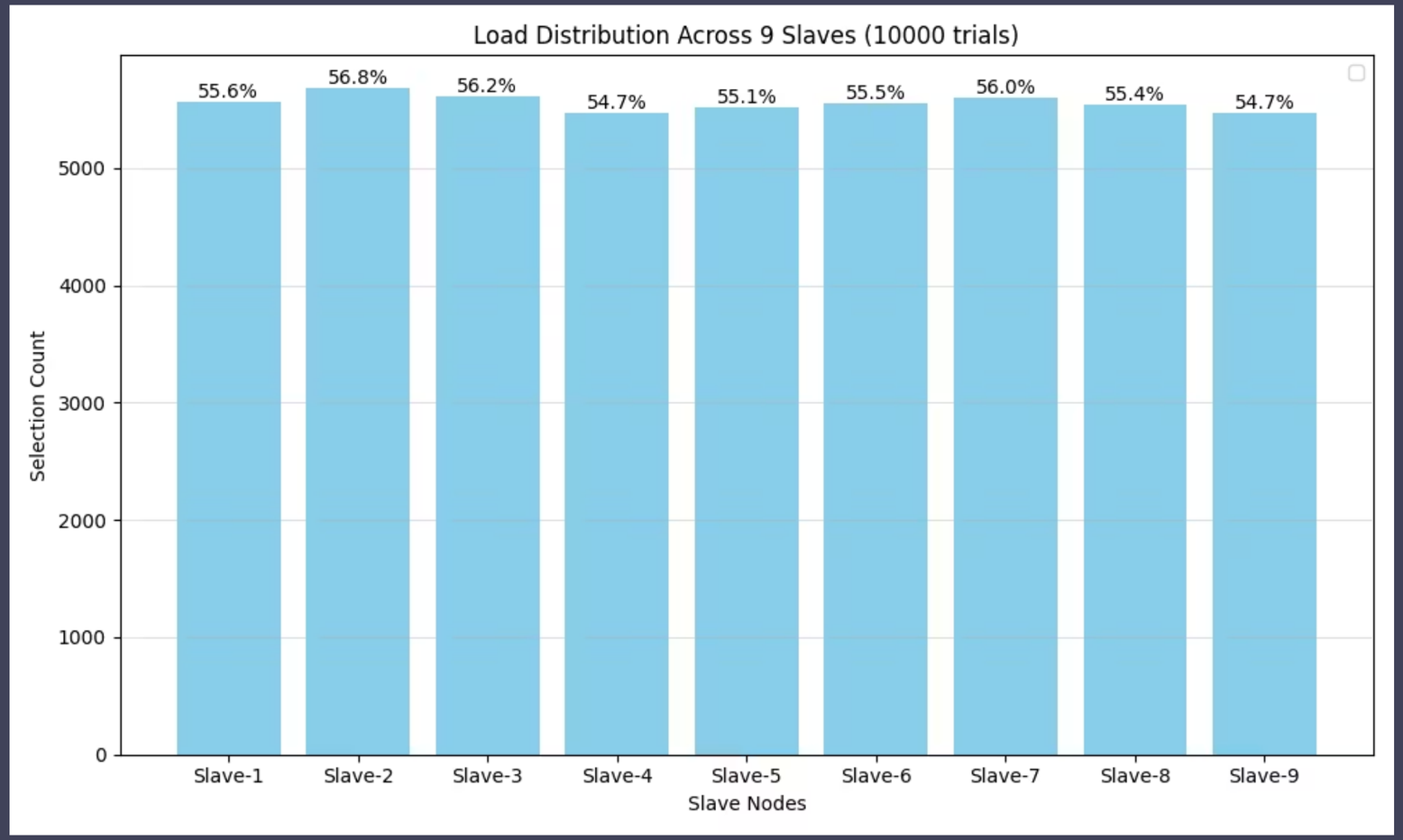
  The client can then retry with an exponential backoff strategy if both servers are down

Try Pitch

Scaled-Testing

Try Pitch

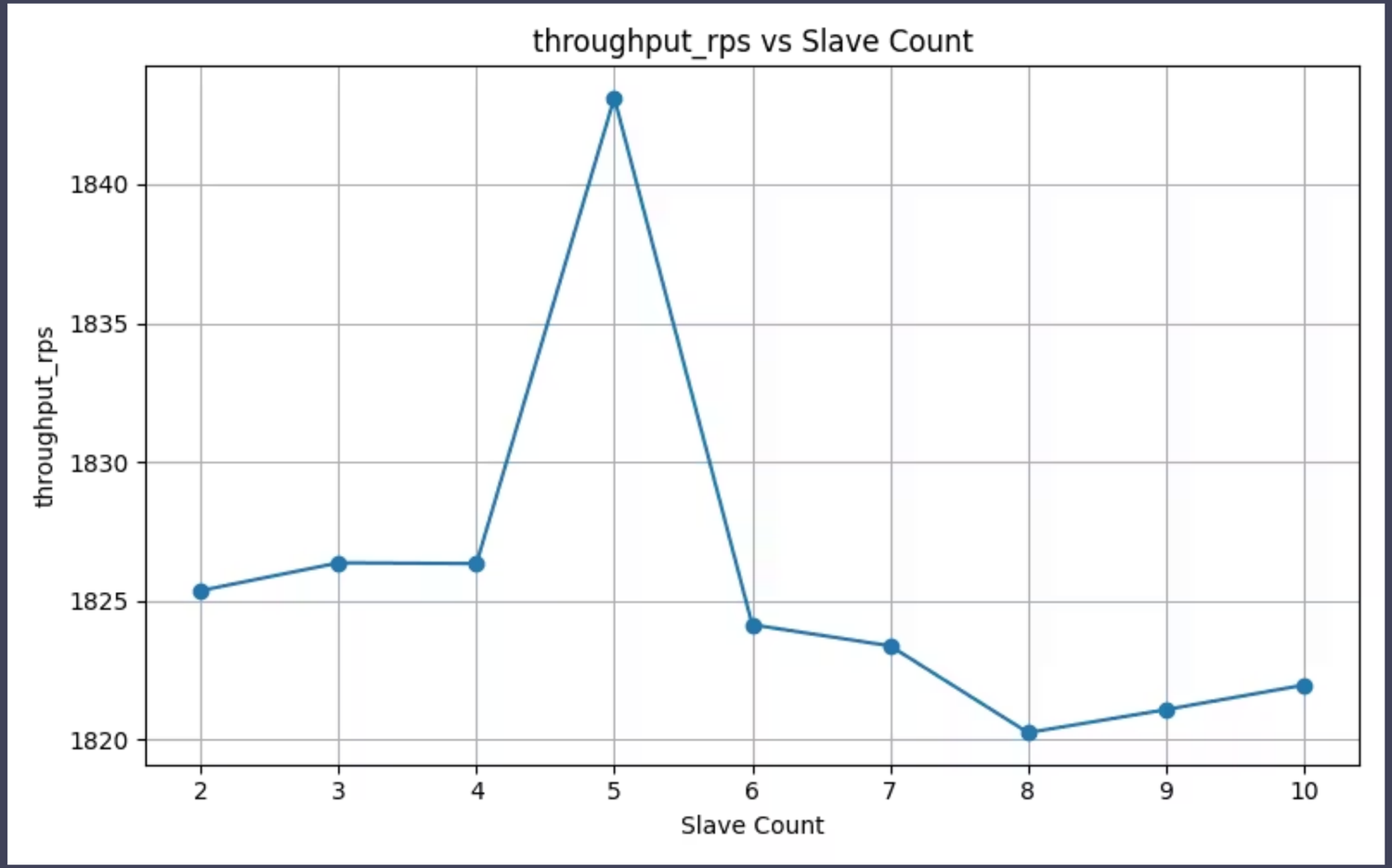
Load Distribution



The graph shows how requests are distributed across 9 slaves. For each WRITE, the master randomly selects a quorum of slaves (ceil(n/2)) to avoid hotspots. Key-to-slave mappings ensure READs are routed only to relevant slaves, reducing load. In 10,000 trials, the distribution should be roughly equal, with minor deviations due to random selection, quorum size, and slave removal.

Try Pitch

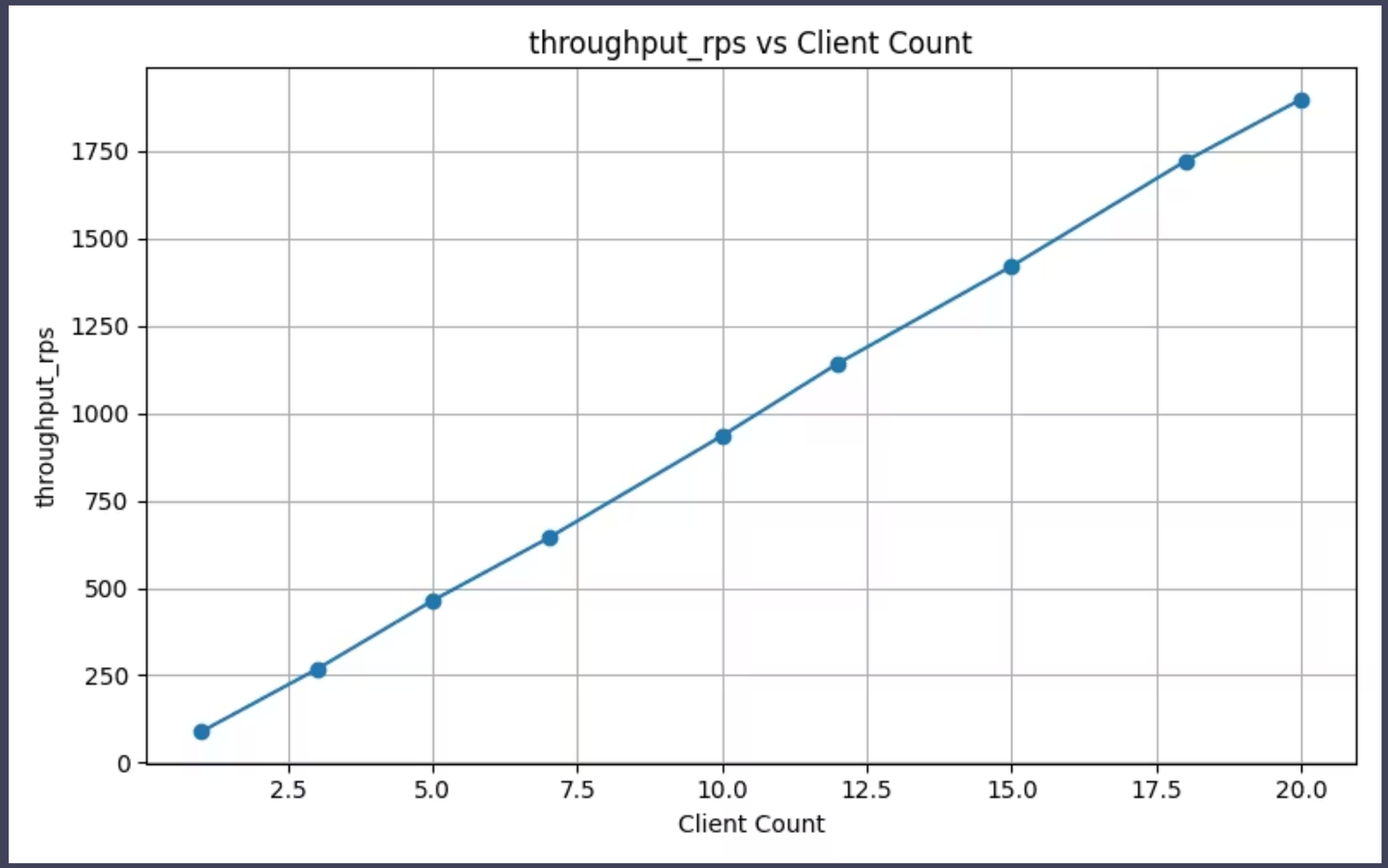
Throughput



Throughput remains steady (~1820–1835 RPS) as slave count increases, indicating a bottleneck (likely at the master or network) and suggesting the workload is read-heavy.

Try Pitch

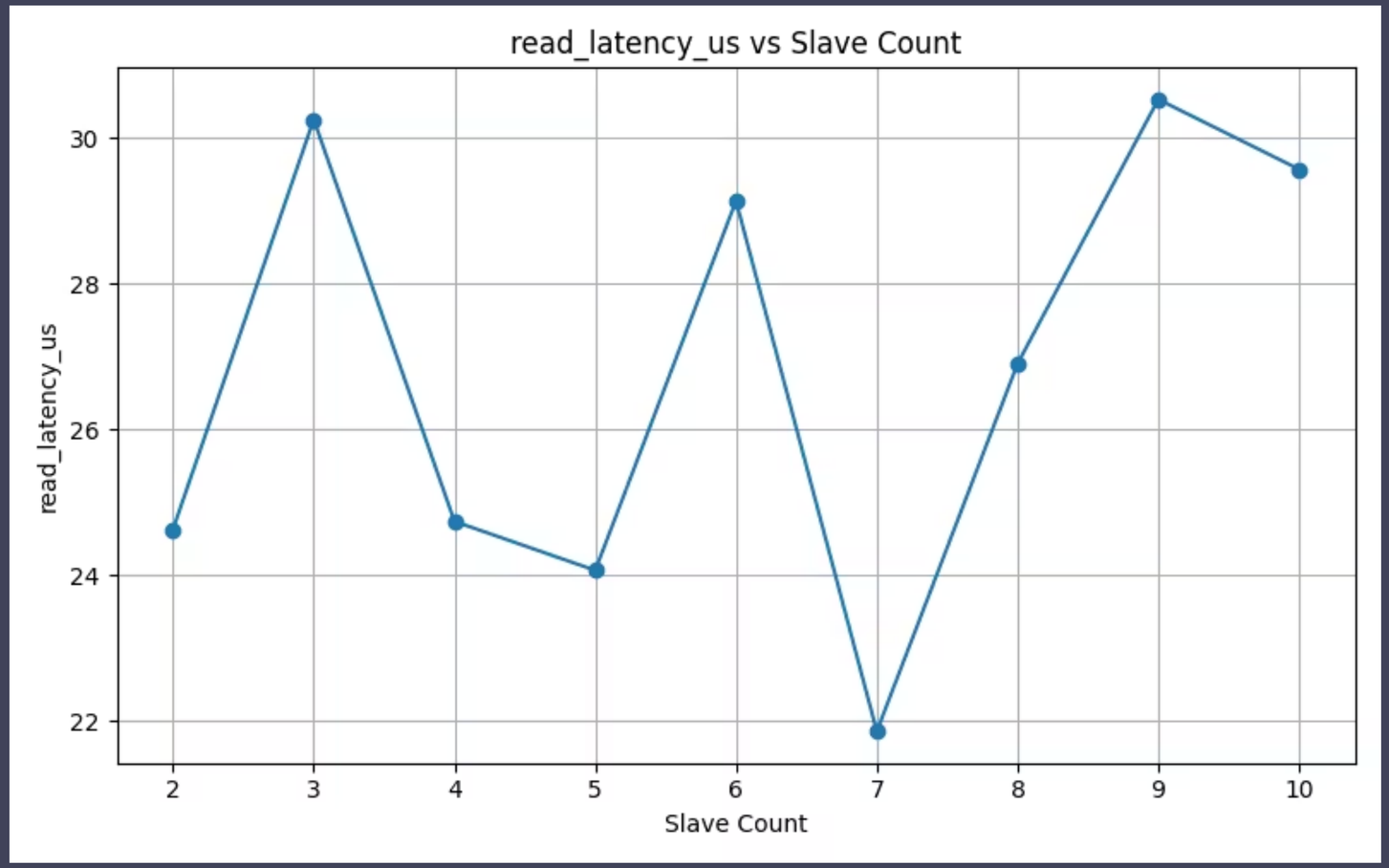
Throughput



The graph shows a **linear increase in throughput** (100 RPS per 1.25 clients) as client count rises from 1 to 20, indicating **efficient scaling** in the distributed shared memory system. This suggests **low contention** and effective synchronization, with no sign of saturation or bottleneck up to 20 clients.

Try Pitch

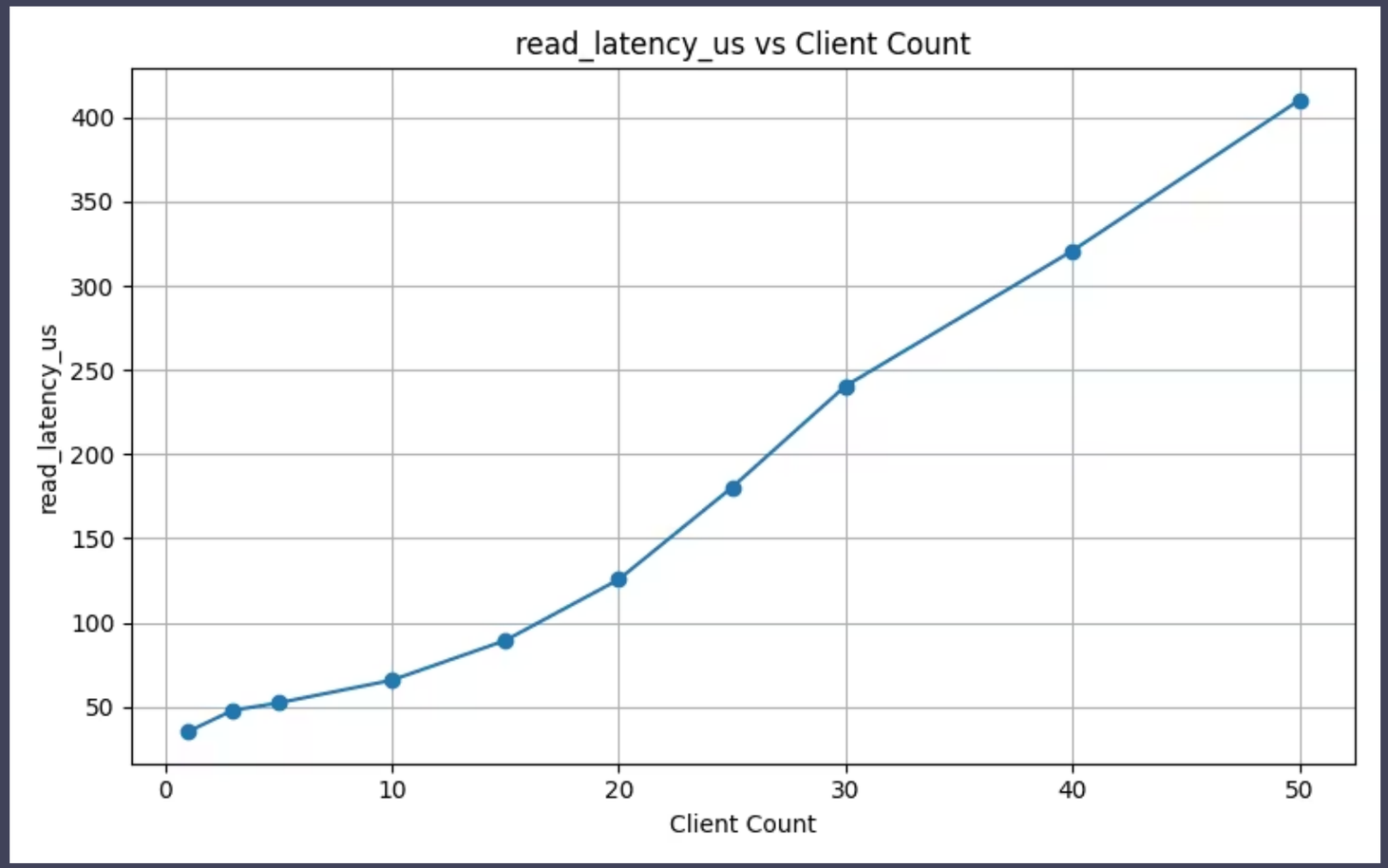
Read Latency



The graph shows that read latency remains **fairly steady**, fluctuating mildly between 22–31 µs as slave count increases. This indicates that the distributed shared memory system maintains **consistent read performance**, with only minor variations likely due to coordination or network overhead.

Try Pitch

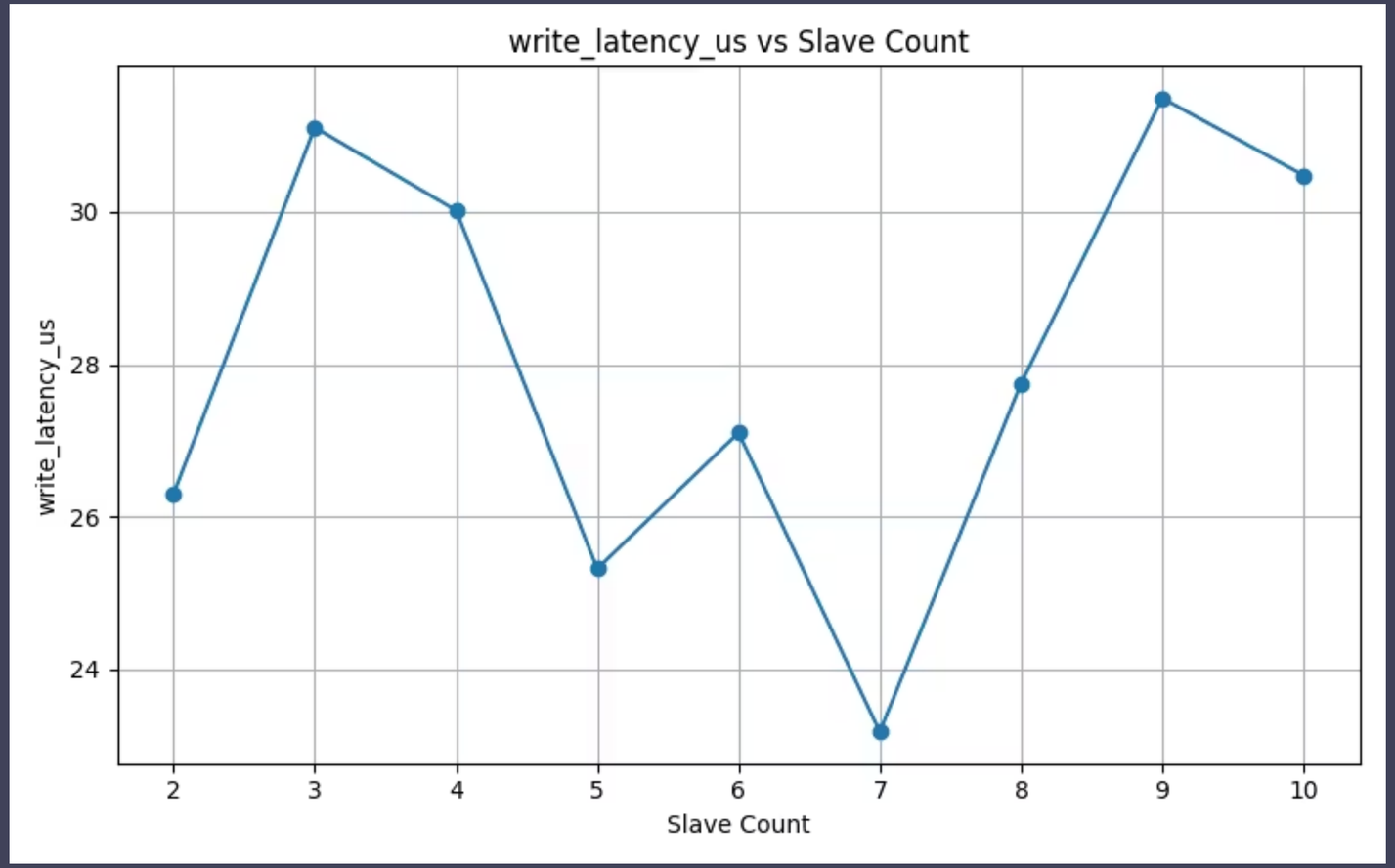
Read Latency



The graph shows that **read latency increases steadily** from ~35 µs to over 400 µs as client count rises in a distributed shared memory system. This indicates **growing contention or coordination overhead** with higher client load, reflecting **scalability limits** in concurrent access.

Try Pitch

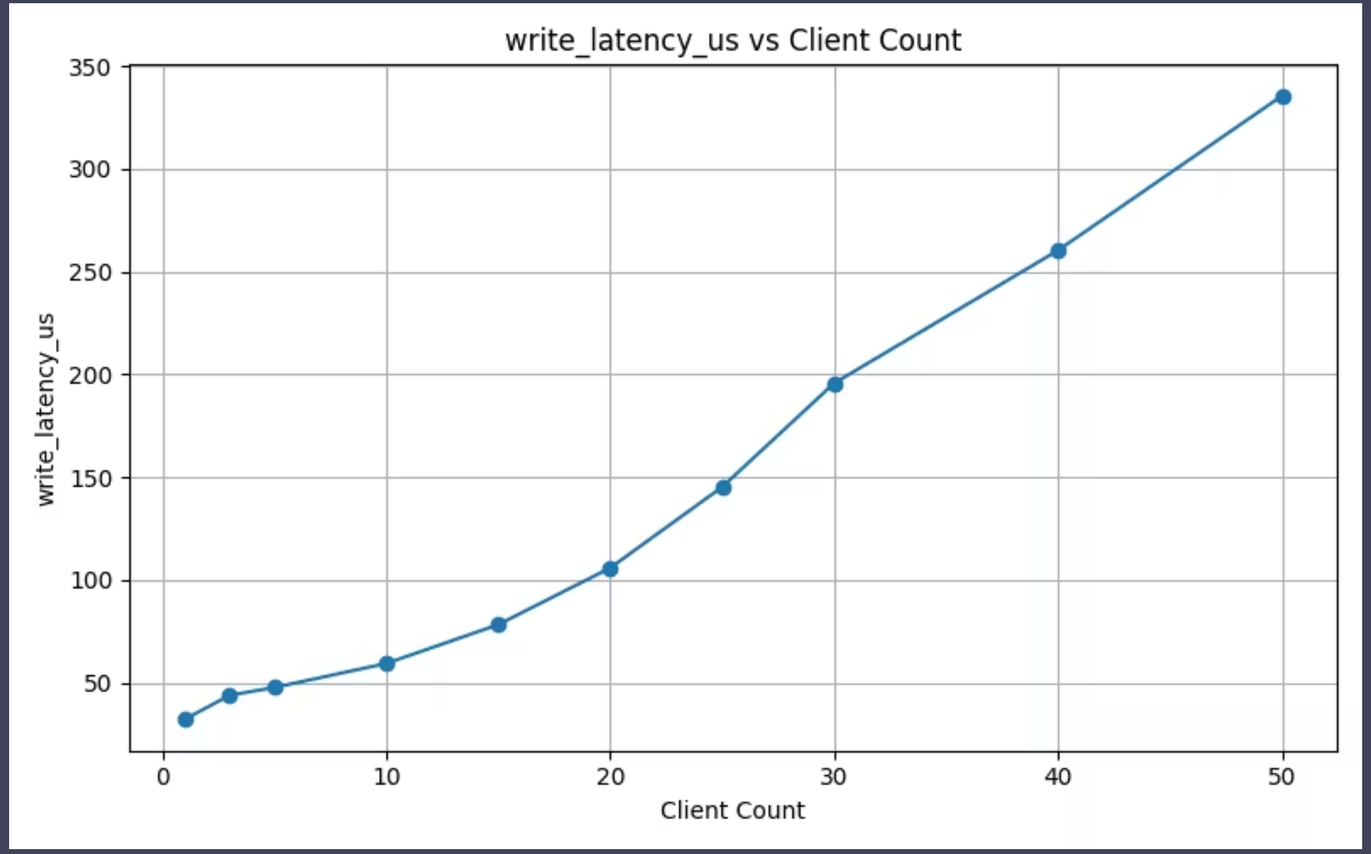
Write Latency



The graph shows that **write latency remains fairly steady** between ~23–31 µs across varying slave counts in a distributed shared memory system. The fluctuations suggest **inconsistent synchronization or propagation delays**, but no strong upward or downward trend is observed with increasing slaves.

Try Pitch

Write Latency



The graph shows that **write latency increases significantly** with client count—from ~30 µs at 1 client to over 330 µs at 50 clients—indicating growing contention or coordination overhead as more clients attempt simultaneous writes.

Try Pitch

THANKYOU

Distributed Shared Memory - TEAM 6

Try Pitch

Pitch

Want to make a presentation like this one?

Start with a fully customizable template, create a beautiful deck in minutes, then easily share it with anyone.

Create a presentation (It’s free)