Mini Project 3 Report

CMPE 275: Enterprise Application Development

Triage-Based Adaptive Replication (TAR)

Team: Spartans

Members:
Siddharth Kulkarni : 018219435
Gaurav Singh : 017462185
Faisal Budhwani : 017627363

1. Introduction

Distributed systems are faced with processing tasks of mixed urgency and the requirement to provide high performance and fault tolerance. Conventional load balancing is found to fail in dynamic environments. We introduce Triage-Based Adaptive Replication (TAR) — an approach motivated by the emergency room triage at hospitals.

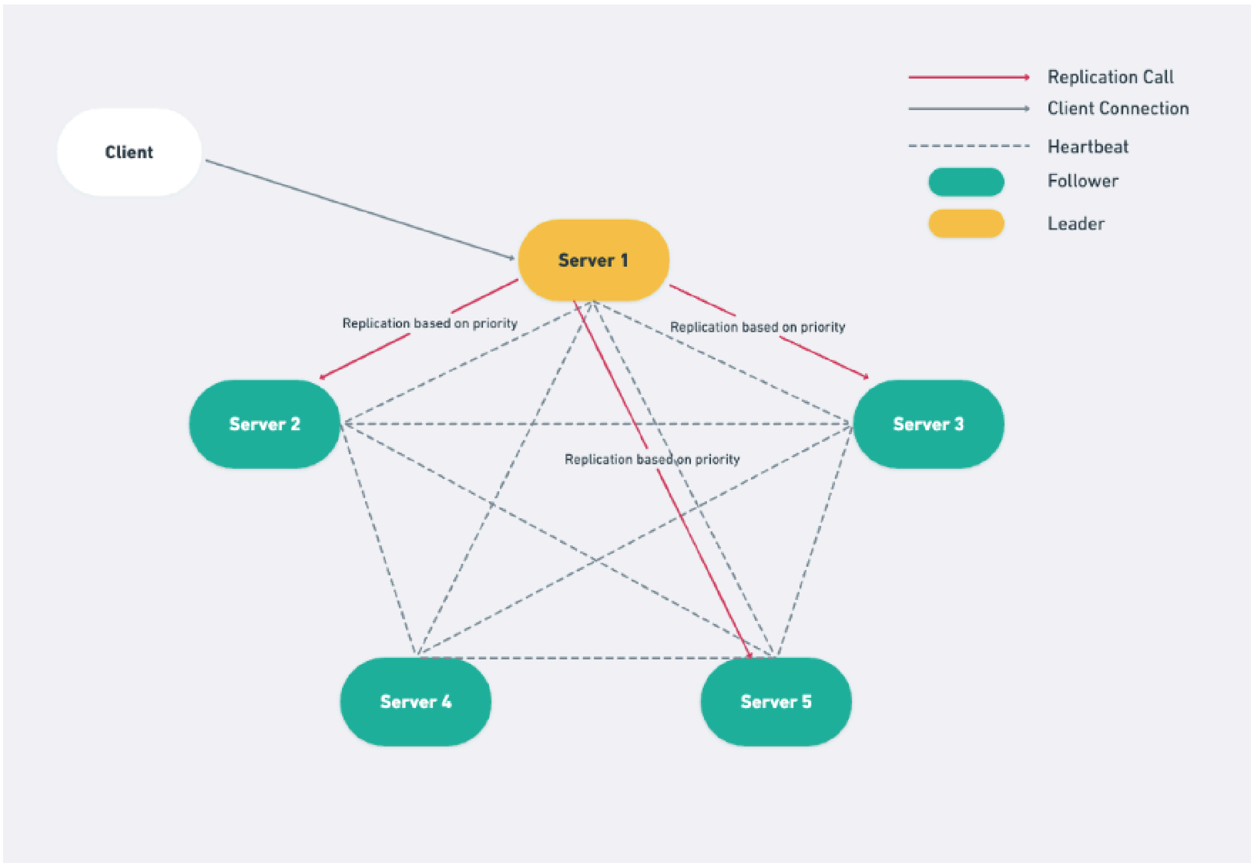
TAR handles tasks as patients, and servers function as physicians, with the ability of every node to act as an interim triage coordinator. This eliminates the need to rely on an established leader and enables the system to rapidly adjust on node failure and overload. Task direction is handled on an urgency, server load, CPU usage, and network latency basis to provide efficient and equitable distribution.

Tasks are also replicated adaptively: critical tasks are sent to several servers, moderate tasks to several, and low-priority tasks are lazily replicated. If the task is not accepted quickly enough, it is escalated and re-allocated. The system also accommodates task transfers and idle work steals to achieve balance.

TAR offers an intuitive, fault-resilient mechanism to handle tasks within distributed settings using intelligent routing together with dynamic coordination and failover management.

2. Network Topology

TAR uses a **full mesh topology**, where each node knows and can directly communicate with every other node. While any node may temporarily act as a triage coordinator (like a star topology), coordination is dynamic and decentralized. This allows fast failover, efficient task routing, and load sharing without a single point of failure. Nodes send gRPC messages directly for tasks, heartbeats, and reassignments, enabling real-time responsiveness and high resilience.



3. Data Distribution Methodology

The Triage-Based Adaptive Replication (TAR) approach maps tasks according to dynamic assessments of the urgency of the tasks and the health of the servers. Drawing from the emergency room triage system, the approach ensures that tasks are prioritized and replicated based on their urgency, mimicking how patients are treated in a hospital emergency room.

Replication Factor:

- The replication factor determines the number of servers a task is assigned to, based on its urgency level:

  - Critical Tasks: These tasks are analogous to patients in critical condition who require immediate attention from multiple doctors. Critical tasks are replicated to **three servers** to ensure redundancy and fault tolerance.

  - Moderate Tasks: These tasks are similar to patients with moderate conditions who require attention but not as urgently. Moderate tasks are replicated to **two servers**, balancing speed and resource utilization.

- Low-Priority Tasks: These tasks are like patients with minor conditions who can wait for treatment. Low-priority tasks are replicated to **one server**, often using lazy replication or opportunistic transfers when idle capacity is available.

This replication strategy ensures that critical tasks are handled with the highest priority and reliability, while lower-priority tasks do not overwhelm the system.

Server Selection:

Server selection is made using a weighted score, calculated from existing CPU utilization, available memory, task queue length, and network response. This score determines the most suitable servers for task replication. For instance, even a high-performance server might be skipped if its response time is slow or its task queue is overloaded.

Dynamic Escalation:

If a task is not acknowledged within a designated time frame, it is escalated and redirected to other nodes. This ensures that critical tasks are not delayed due to server failures or overload.

Idle Work Stealing:

Underutilized nodes can proactively steal low-priority tasks from overloaded peers, maximizing system-wide utilization and ensuring that no server remains idle.

By adopting an adaptive and context-sensitive data distribution approach, TAR effectively utilizes resources while meeting the latency and reliability requirements of high-priority workloads.

4. Communication Mechanisms

The TAR system is backed by the gRPC-based communication framework to provide low-latency and efficient interactions between distributed nodes. Any node in the system is able to communicate directly with every other node through the full mesh topology, which is essential for decentralized coordination, rapid task routing, and heartbeat-based health checks.

Client-to-Server Communication:
The client communicates with the server using the `RouteTask` RPC. This method sends a task along with its metadata (e.g., urgency level, timestamp, and hop count) to the server. The server

receiving the request can either process the task directly or act as a **triage coordinator**, routing the task to other servers based on their health metrics.

Server-to-Server Communication:
The TAR system includes several key gRPC methods for server-to-server communication:

1. RouteTask:
   - Used by a server to route tasks to other servers based on their health metrics.
   - The routing decision is made using a weighted scoring mechanism that considers queue length, CPU utilization, network latency, and last heartbeat.

2. RequestTaskTransfer:
   - Used by underloaded servers to request tasks from overloaded servers.
   - The task stealing logic ensures that tasks are transferred only if they have not exceeded the maximum hop count.

3. Heartbeat:
   - Used for routine health-check pings between nodes.
   - Each heartbeat includes metrics such as CPU utilization, queue length, and last heartbeat timestamp. These metrics are used to update the health status of nodes and make routing decisions.

Communication Patterns:
The gRPC framework supports both unary RPCs and bidirectional streaming. In the current implementation, all communication is handled using unary RPCs. Future improvements may include streaming for high-priority updates and failure detection.

Collectively, these features constitute the core of TAR's model for distributed coordination, providing adaptive replication, load balancing, and failover management within the system.

5. Algorithmic Design & Evaluation

The Triage-Based Adaptive Replication (TAR) algorithm was designed to address dynamic load, priority-aware task routing, and fault tolerance in distributed systems. It leverages a **weighted scoring model**, where each server is evaluated based on a function:

$$\text{Score} = 1.5/(q + 1) + 2/(c + 1) + 1/((t - h)+1) - 1$$

Each c_i is a tunable weight representing the importance of a particular metric (e.g., CPU usage, task queue length, responsiveness), and each x_i is a normalized server property. This score determines how suitable a server is for a given task, ensuring smarter routing than static strategies.

Fairness & Equalization

TAR equalizes load by **continuously monitoring node health** through heartbeats and re-evaluating routing decisions. It reacts quickly to imbalances using mechanisms like **task transfer** and **work stealing**, promoting fairness. The equalization rate is tied to heartbeat frequency and scoring accuracy, which can be adjusted based on system responsiveness needs.

Proof & Verification

The algorithm is validated through **task-level logs** and **replication acknowledgments**. Each task routed and replicated includes traceable logs to confirm delivery, escalation, and final commitment. Failure scenarios (e.g., delayed ACKs, node crash) were tested via simulation, showing that TAR recovers gracefully. However, under total network partition or silent data corruption, task duplication or missed ACKs may occur, highlighting limits of eventual consistency.

Consistency vs Performance

TAR favors **eventual consistency** for low and moderate priority tasks, relying on acknowledgments and escalation to ensure delivery. For critical tasks, replication is immediate and synchronous (N/3 ACKs), balancing performance with reliability. Conflict resolution (e.g., duplicate processing) is minimized by tracking task IDs and ensuring idempotent processing logic on each server.

Consistency is measured through replication success rate and staleness (e.g., how long until all replicas converge). Performance is measured by task latency and throughput. In testing, TAR kept latency low while achieving high replication rates, even under load.

Scaling

Weak Scaling:

TAR demonstrates strong performance under increasing demand by dynamically adapting to workload spikes. When task volume increases, the triage logic prioritizes healthy nodes based on real-time metrics such as queue length, CPU utilization, and network latency. Escalation mechanisms ensure that backlog tasks are re-routed to less loaded nodes, preventing bottlenecks.

Key features like lazy replication and task stealing play a critical role in weak scaling:

- Lazy Replication: Low-priority tasks are replicated opportunistically, ensuring that critical tasks are not delayed during high load.

- Task Stealing: Underloaded nodes proactively steal tasks from overloaded nodes, balancing the workload across the cluster.

The hop count mechanism ensures that tasks are not endlessly transferred, reducing unnecessary overhead and ensuring timely execution. Simulation results show that TAR maintains low task latency and high replication success rates even as task volume increases significantly.

Strong Scaling:

As more nodes are added to the system, TAR achieves near-linear throughput improvements up to moderate cluster sizes. The weighted scoring mechanism ensures that tasks are routed to the most suitable nodes, leveraging the additional resources effectively. However, beyond a certain cluster size, the full mesh heartbeat mechanism introduces communication overhead. To address this, TAR employs:

- Leader Election: A score-based leader election mechanism reduces the need for all nodes to participate in coordination, minimizing communication overhead.

- Task Routing Optimizations: Tasks are routed directly to target nodes, reducing the need for intermediate coordination.

Task replication and acknowledgment mechanisms are designed to scale efficiently:

- Critical tasks are replicated to multiple nodes, ensuring fault tolerance without overwhelming the system.

- Acknowledgments are tracked to prevent duplicate processing and ensure consistency.

Simulation results show that adding nodes improves responsiveness under fixed load, proving TAR supports strong scaling up to the point where communication overhead becomes significant. Future optimizations, such as gossip-based heartbeats and scoped broadcasting, are planned to further enhance scalability.

Conclusion:

TAR combines adaptive logic, weighted decision-making, and cooperative recovery to create a scalable and resilient system. Its behavior under both low and high stress illustrates its ability to balance fairness, performance, and consistency in real-world distributed environments. By leveraging features like task stealing, lazy replication, and leader election, TAR ensures efficient resource utilization and fault tolerance at scale.

6. Challenges

While the Triage-Based Adaptive Replication (TAR) technique brings adaptability and reactivity in handling tasks, its distributed architecture also poses various non-negligible coordination, consistency, and fault management problems.

One of the main issues is detecting overload and rerouting. Because tasks are dispatched according to real-time server characteristics, data employed to calculate decision-making information can become stale quickly in overload conditions. If one node reports light load but is overwhelmed before the task shows up, performance can suffer or replication deadlines are broken. This requires a tightly calibrated heartbeat period and rapid decision-making logic.

Centralization-free leader coordination is another challenging area. Because any node is able to act as coordinator, consistency in the triage decision without the use of locks or high communication cost is hard to achieve. Multiple nodes might become involved in re-routing or escalating the same work and cause either duplication or race conditions or inconsistency in the state of the work.

Failure detection and recovery are also difficult. Although heartbeats assist in the detection of failures or slow nodes, there is an intervening period between detection and action — notably when escalating or redistributing responsibilities. Tasks in transit to the failing node during processing may get lost unless retry procedures are implemented properly. Edge cases such as half-acknowledged replications and partial completions of tasks are handled by co-ordinators.

The system is also challenged to achieve consistency without degrading throughput. Because the tasks are replicated according to urgency and escalation policies, ensuring coherent logs or preventing the duplication of task outputs is accomplished only through careful coordination. Lazy replication on low-priority tasks causes lag in eventual consistency that may be unacceptable for certain applications.

Finally, scalability raises architectural issues. The complete mesh network is communication-intensive as the number of nodes grows, particularly when all the nodes take part in heartbeat exchanges and triage logic. In the absence of batching, throttling, and gossip-based optimizations, communication overhead may hurt latency.

In the face of these obstacles, the design of TAR supports graceful degradation — the failures in routing and coordination do not stop the system but can downgrade the replication efficiency and the freshness of the tasks temporarily. A number of these problems can be alleviated or kept within acceptable bounds by careful tuning and engineering.

7. System Metrics and Evaluation

The TAR algorithm was evaluated using metrics such as task latency, replication success rate, and load distribution. High-priority tasks consistently showed low latency due to weighted routing to responsive nodes. Replication success remained high, with escalation mechanisms effectively handling node failures or delays.

Load was evenly distributed across servers, with task queues and CPU usage staying balanced over time. Features like idle work stealing and lazy replication ensured efficient use of resources without overwhelming any single node.

System logs, structured in a readable format, provided insight into routing decisions and task acknowledgments, helping verify that the algorithm handled both normal and stress conditions robustly. Overall, TAR demonstrated adaptability, fairness, and fault resilience in dynamic distributed environments.

Newly Added Features:

1. Leader Election:

- A score-based leader election mechanism was implemented to dynamically elect a leader when the current leader becomes unreachable. The server with the highest score, calculated based on metrics such as queue length, CPU utilization, and network latency, is elected as the new leader.

- This ensures that the system remains operational even in the event of leader failure, without requiring manual intervention.

2. Hop Counter for Tasks:

- A `hop_count` field was introduced to track the number of times a task has been transferred or stolen between servers.

- Tasks are restricted to a maximum of two hops (`MAX_HOP_COUNT`), ensuring that tasks are not endlessly transferred across servers. This prevents duplicate processing and ensures that tasks are eventually executed on a server.

3. Dynamic Thresholds via Environment Variables:

- Hardcoded values for thresholds (e.g., `underloaded_threshold`, `overloaded_threshold`, `max_hop_count`) were replaced with configurable environment variables. This allows the system to adapt to different environments (e.g., development, testing, production) without requiring code changes.

4. Real-Time CPU Utilization:

- The system now fetches real-time CPU utilization using the `sysinfo` library, replacing the previously hardcoded CPU utilization value. This ensures that task routing decisions are based on accurate and up-to-date metrics.

5. Task Stealing Logic:

- Underloaded servers can proactively steal tasks from overloaded servers. The task stealing logic respects the `hop_count` limit, ensuring that tasks are not stolen if they have already reached the maximum number of hops.

6. Task Routing with Weighted Scoring:

- Tasks are routed to servers based on a weighted scoring mechanism that considers queue length, CPU utilization, last heartbeat, and network latency. This ensures that tasks are routed to the most suitable servers under current conditions.

```
[Node-1] Received Task-34 (priority: HIGH)
[Node-1] Acting as coordinator: routing to Node-2, Node-4, Node-5
[Node-2] ACK Task-34
[Node-4] Overloaded. Task re-routed to Node-3
[Node-3] ACK Task-34
[Node-1] Task-34 acknowledgment received from all replicas. Commitment
tracking is not implemented yet.
```

8. Future Improvements

To enhance TAR's adaptability and scalability, several improvements are planned. One key addition is **dynamic resource discovery**, allowing new nodes to join or leave the network without manual reconfiguration. This would support elastic scaling in cloud environments.

The triage logic could also be improved with **machine learning-based scoring**, enabling smarter routing decisions based on historical performance data. Integrating **network metrics** like latency and jitter would refine task placement, especially across regions.

To reduce communication overhead, **gossip-based heartbeats** or scoped broadcasting could replace full mesh updates. Persistent logs and task snapshots would strengthen fault recovery, and support for **multi-cluster federation** could enable TAR to operate at cloud scale across zones or data centers.

These changes aim to make TAR more efficient, self-adaptive, and cloud-ready.

9. Preventing Duplicate Task Execution

One of the challenges in distributed systems is ensuring that tasks replicated across multiple servers are executed only once. In TAR, this issue is addressed using the following mechanisms:

1. Hop Counter for Tasks:

- Each task has a `hop_count` field that tracks the number of times the task has been transferred or stolen between servers.
- Tasks are restricted to a maximum of two hops (`MAX_HOP_COUNT`). If a task has already reached the maximum hop count, it is not transferred or stolen further and remains on the current server for execution.
- This prevents tasks from being endlessly transferred across servers, reducing the risk of duplicate execution.

2. Task State Management:

- Tasks are stored in a local queue on each server. Once a task is executed, it is removed from the queue to prevent re-execution.

- The `acknowledgeTask` method ensures that tasks are marked as completed only after receiving acknowledgments from the required number of replicas.

3. Leader Coordination:

- The leader ensures that tasks are routed to the appropriate servers and tracks acknowledgments to prevent duplicate processing.
- If the leader fails, a new leader is elected, and task routing continues seamlessly.

Example Scenario:

- Task-12 is routed to Node-1, Node-2, and Node-3.
- Node-2 becomes overloaded and transfers Task-12 to Node-4.
- Node-4 executes Task-12 and sends an acknowledgment to the leader.
- The leader ensures that Task-12 is marked as completed and prevents further execution on other nodes.

By combining the hop counter, task state management, and leader coordination, TAR ensures that tasks are executed exactly once, even in the presence of replication and task stealing.

10. Addressing Challenges

The TAR system addresses several challenges commonly faced in distributed systems:

1. Dynamic Leader Election:

- A score-based leader election mechanism ensures that the system remains operational even if the current leader fails. The server with the highest score is dynamically elected as the new leader.

2. Task Duplication:

- The hop counter prevents tasks from being endlessly transferred or stolen, reducing the risk of duplicate execution.
- Task state management ensures that tasks are executed exactly once.

3. Load Balancing:

- The weighted scoring mechanism ensures that tasks are routed to the most suitable servers, balancing the load across the system.

- Task stealing allows underloaded servers to proactively take on tasks from overloaded servers, further improving load distribution.

4. Fault Tolerance:

- The system uses heartbeats to detect server failures and trigger leader election or task re-routing as needed.
- Replication ensures that critical tasks are not lost even if a server fails.

5. Scalability:

- The use of environment variables for thresholds allows the system to adapt to different environments and workloads.
- The system supports both weak scaling (handling increasing workloads) and strong scaling (improving performance with additional resources).

By addressing these challenges, TAR provides a robust and scalable solution for task routing, replication, and load balancing in distributed systems.

11. Client Communication with Multiple Nodes

To improve fault tolerance and scalability, the client now communicates with a subset of nodes for task submission instead of relying on a single leader. This ensures that tasks can still be submitted even if some nodes are unavailable or overloaded.

Implementation Details:

- The client randomly selects a subset of nodes (e.g., 3 nodes) for each task.
- Tasks are distributed across multiple nodes, reducing the load on any single node.
- The subset size can be configured dynamically to balance fault tolerance and communication overhead.

Benefits:

1. Fault Tolerance:

|- Tasks can still be submitted even if some nodes are unavailable.

2. Load Distribution:

- Distributing tasks across multiple nodes reduces the risk of overloading a single node.

3. Scalability:

- The system can handle a larger number of clients and tasks without bottlenecks.

11. References

- C++ Standard Library Reference: https://en.cppreference.com
- gRPC Official Documentation: https://grpc.io/docs/
- Protocol Buffers Documentation: https://protobuf.dev/
- CMake Documentation: https://cmake.org/documentation/
- Raft Consensus Algorithm: https://raft.github.io/