Cache Breakdown Simulation

Overview

This project simulates a real-world **cache breakdown (cache stampede)** scenario in a backend system. The objective is to demonstrate how cache expiry can lead to database overload when multiple concurrent requests attempt to access the same data, and how this issue can be mitigated using synchronization techniques.

The system is designed to replicate high-traffic production environments such as e-commerce, banking, and large-scale web applications where caching is essential for performance and scalability.

System Components

The solution consists of the following key modules:

1. Fake Database
2. In-memory Cache with TTL
3. Service Layer
4. Concurrency Simulation
5. Lock-based Protection

The architecture mimics real-world distributed systems where a cache layer sits between the application and the database.

Approach

The system is implemented using Python and multithreading to simulate concurrent user requests.

Cache Layer

An in-memory cache is used to store frequently accessed product data. Each cache entry is associated with a TTL (Time-To-Live), after which the data expires.

When a request arrives: - The system checks whether the data exists in the cache. - If the data is valid, it is returned immediately. - If the cache expires or is missing, the system fetches the data from the database and updates the cache.

This behavior is like industry tools such as Redis and Memcached.

Cache Breakdown Scenario

A cache breakdown occurs when a popular or frequently accessed cache entry expires.

Simulation Steps

1. The cache is preloaded with product data.
2. The TTL is set to a short duration.
3. After expiry, multiple concurrent user requests are generated.
4. Since the cache is empty, all requests hit the database simultaneously.

This results in: - High database load - Increased latency - Potential system failure

Solution Strategy

To mitigate the breakdown, a mutex lock mechanism is implemented.

Lock-based Protection

The system ensures that: - Only one thread fetches data from the database when the cache expires. - Other concurrent threads wait until the cache is refreshed. - Once the cache is updated, the remaining threads read from the cache instead of the database.

This reduces database load and stabilizes system performance.

Design Decisions

The following design choices were made:

- A simple in-memory cache was used to focus on system behavior rather than infrastructure.
- TTL-based expiration was selected to mimic real caching strategies.
- Multithreading was used to simulate concurrent traffic.
- A lock mechanism was used as a lightweight concurrency control method.

This design provides clarity and ease of understanding while demonstrating key backend engineering concepts.

Observations

Without Lock

- Multiple threads are fetched from the database.
- High resource utilization.
- Poor scalability.

With Lock

- Only one database call occurs.
- Reduced system load.
- Improved performance and stability.

Limitations

- The cache is local and not distributed.
- The database is simulated.
- No monitoring or logging framework is included.
- The system does not support dynamic scaling.

Future Enhancements

The project can be extended by implementing:

- Redis-based distributed cache
- Distributed locking
- Random TTL to avoid cache avalanche
- Background cache refresh
- Circuit breaker pattern
- Performance benchmarking and monitoring

Conclusion

This project successfully demonstrates a critical failure scenario in distributed systems and provides a practical solution. The approach aligns with modern backend engineering practices and highlights the importance of concurrency control and system reliability.

This simulation reflects real-world production challenges and showcases the ability to design scalable and fault-tolerant systems.