# Distributed Database Project

JI-DUNG, LO, 07700005729, jlo3@scu.edu,

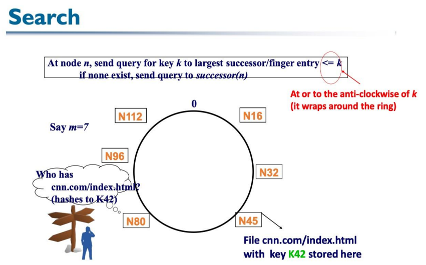
HAO-PENG, GAO, 07700019769, hgao2@scu.edu

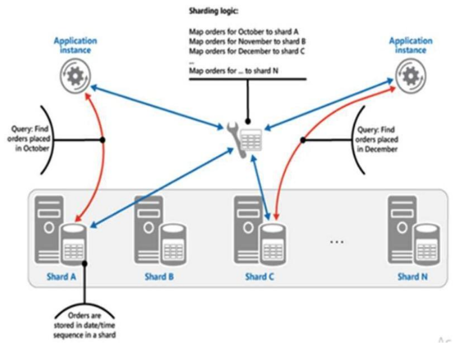
## Introduction

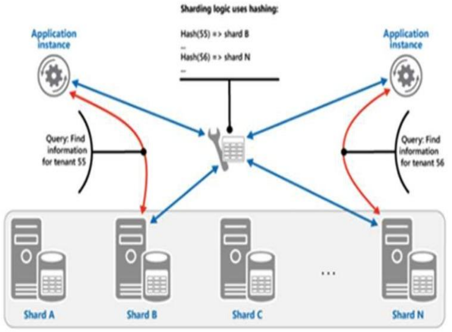
- Project Goals
  - Data Sharding, Fault tolerance, Scalability and data consistency for a distributed database web application
- Key implementation
  - Data sharding for Member/Order entities, consistent data product catalog vis message queue, failure detection and custom gossip protocol, encrypted communication, P2P routing, and static file replication
- Motivation
  - Single-node database bottleneck performance
  - Distributed system scales easily
  - Provide redundancy to increase fault resilience

# Related work/systems

- Sharding strategy
  - Range strategy, Hash strategy, Lookup strategy
- Multi-Version Concurrency Control (MVCC)
  - locking overhead
  - allows multiple versions of data so readers don’t block writers
- P2P chord searching
  - At node n, send query for key k to largest successor/finger entry <= k
  - Wraps around the ring


Search
At node *n*, send query for key *k* to largest successor/finger entry <= *k*
if none exist, send query to *successor(n)*
At or to the anti-clockwise of *k* (it wraps around the ring)
Say *m*=7
0
N112
N16
N96
N32
N80
N45
Who has cnn.com/index.html? (hashes to K42)
File cnn.com/index.html with key K42 stored here


Sharding logic:
Map orders for October to shard A
Map orders for November to shard B
Map orders for December to shard C
...
Map orders for ... to shard N
Application instance
Application instance
Query: Find orders placed in October
Query: Find orders placed in December
Shard A
Shard B
Shard C
...
Shard N
Orders are stored in date/time sequence in a shard



## Key design choices

- **Heterogeneity:**
  - Standard protocols (HTTP/REST and CloudAMQP for messaging). Self-contained Spring Boot application (Java 8+) and SQLite
- **Openness:**
  - RESTful APIs and standard networking protocols. HTTP requests or UDP for gossip, and data consistency uses RabbitMQ.
- **Security:**
  - Each client or forwarding request includes an X-Signature header – an HMAC SHA-256 signature of the request content or parameters, generated using a shared secret key known to all servers.
- **Failure Handling:**
  - Gossip-based failure detection. Each node runs a Heartbeat Service that periodically exchanges heartbeat messages (over HTTP) with all peers.
  - Static files are also replicated to their next neighbor, so a static content request that would have gone to a down node can be forwarded or fetched from another node holding that file.
  - RabbitMQ also acts as a buffer during failures

## Key design choices

- **Concurrency:**
  - Natural lock from SQLite
  - Simulate MVCC from multiple tries
- **Quality of Service (QoS):**
  - Throughput is improved by sharding
  - For requests that do need to hop (e.g., a user request routed to another shard), we minimize overhead by using asynchronous, non-blocking forwarding (e.g., RestTemplate)
- **Scalability:**
  - handle increasing amounts of work by adding more EC2 instances
  - Data sharding - each node handle a fraction of the orders.
  - Consistent hashing (with the finger table) means we can add a new server node and assign it a portion of the hash space
- **Transparency:** The system provides distribution transparency in several forms:
  - **Location transparency:** do not need to know which server or database their data resides on.
  - **Replication transparency:** Product data and static files are replicated and hidden from users.
  - **Failure transparency:** Through gossip, the system attempts to mask node failures from the user.
  - **Concurrency transparency:** The user is unaware of the complex concurrency control under the hood.

## Overview of components, architecture and features

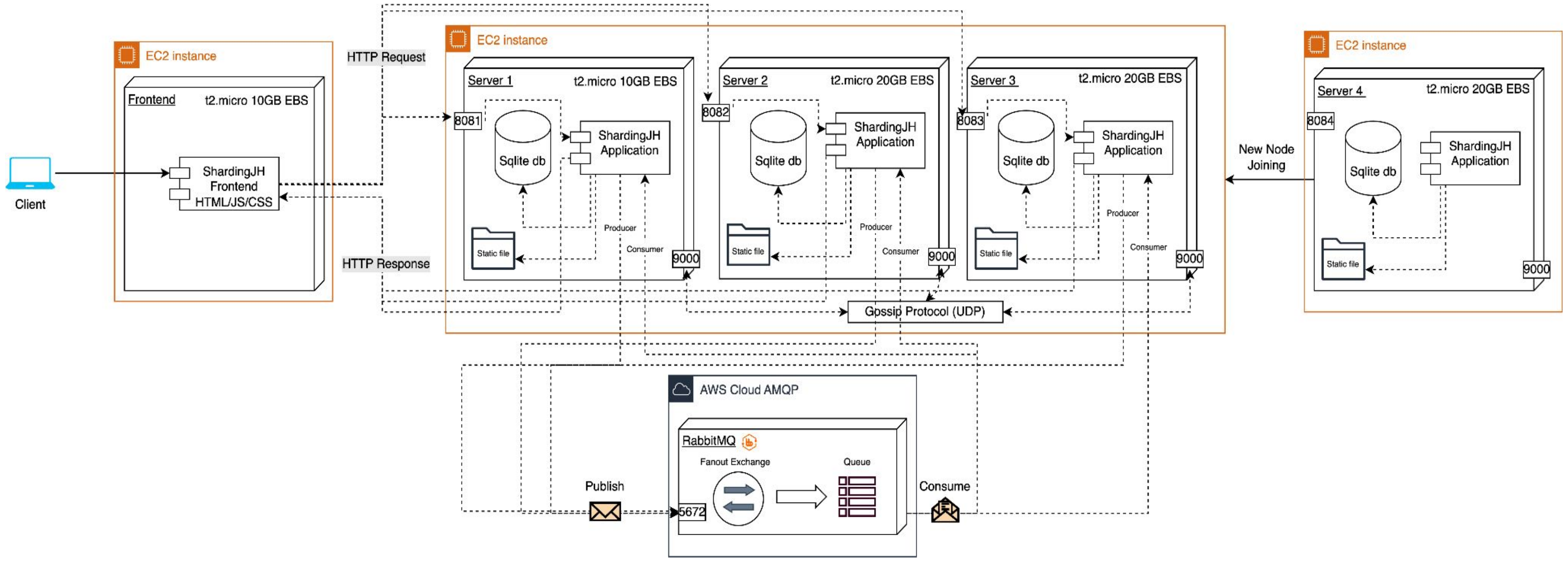
**System Architecture**

We have 4 server nodes (EC2 instances) running identical Spring Boot applications, each with its own local database instance. A RabbitMQ message broker (hosted in the cloud) connects to all nodes to facilitate messaging for data consistency.

Each node's internal architecture follows a layered design: a REST API Controller layer (handles client HTTP requests for members, orders, products, static files), a Service layer (business logic and coordination of operations), and a Repository/Data Access layer which uses a custom routing DataSource to direct queries to the appropriate shard DB.

In addition, each node runs background services – a Gossip module for cluster membership, a Heartbeat service for failure detection, a Server Router for forwarding requests to other nodes when needed, and RabbitMQ Producer/Consumer components for product updates.

# Deployment Diagram



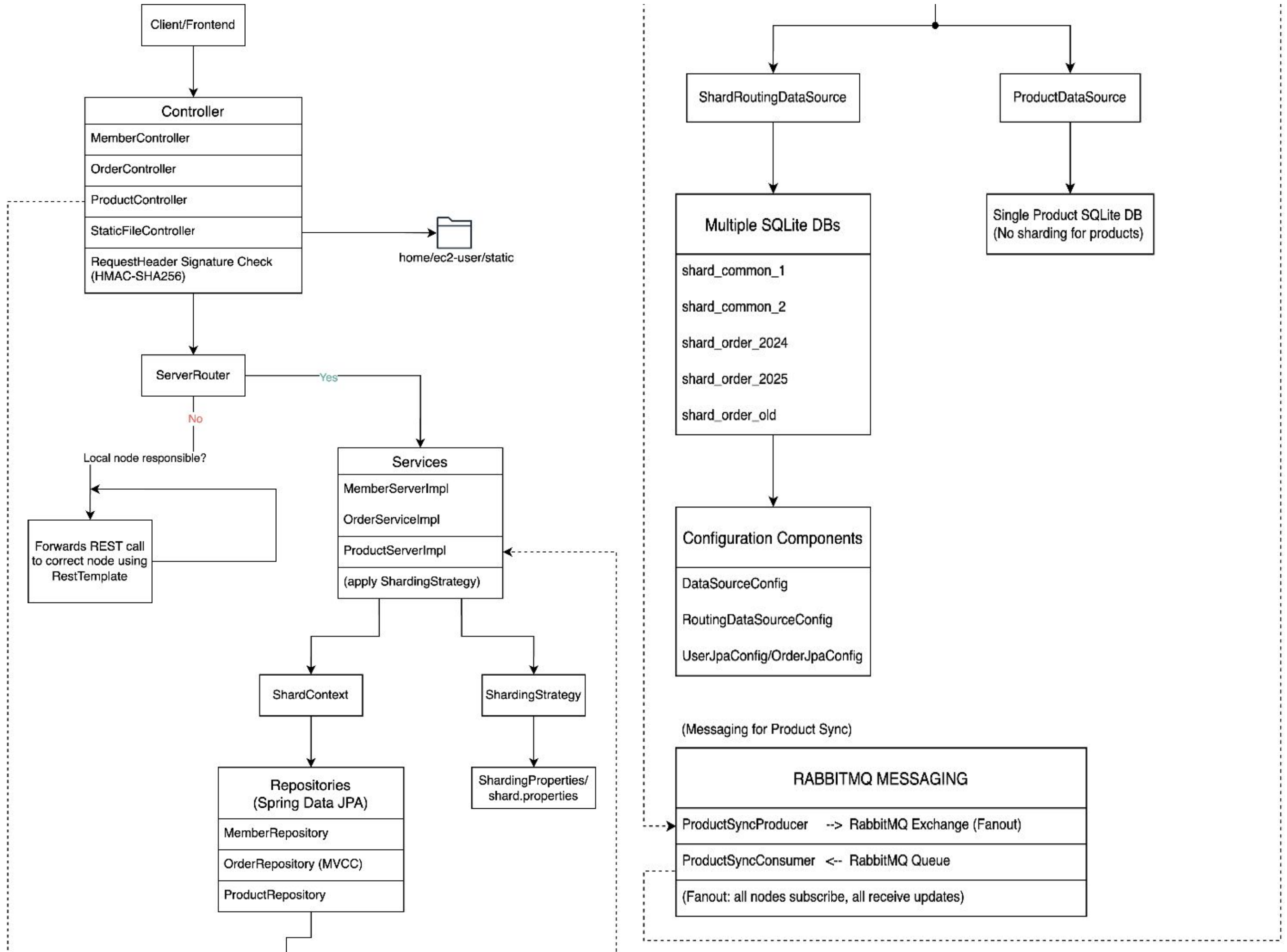
## Overview of components, architecture and features

**Data Sharding Strategy**

We use two partitioning strategies in tandem:

- Hash-based sharding for Member data: We hash the member’s ID and take modulo N to assign each member to one of N member shards. In our setup N=2 for members, so e.g. MemberID “Alice” might hash to shard 1, “Bob” to shard 2. This evenly distributes load across nodes (avoiding hotspots), though it means assembling results for range queries would require querying all shards.
- Range-based sharding for Order data: We partition orders by year of creation (with an additional “old” bucket for archives). For example, all 2024 orders go to one shard, 2025 orders to another, etc. This allows efficient time-range queries (each year’s data is localized) and reflects a realistic partitioning by time. The range boundaries are configured (e.g., a new shard per year).

# Component Diagram



## Overview of components, architecture and features

**Web/API Controller Layer:** This includes the REST controllers such as MemberController, OrderController, ProductController, and StaticFileController which handle HTTP requests from clients (or forwarded from other servers).

**Service Layer:** This contains the business logic and interacts with repositories. For example, MemberServiceImpl, OrderServiceImpl, ProductServiceImpl contain the core logic for manipulating each entity.

**Repository/Data Access Layer:** Using Spring Data JPA for each entity, backed by the routing data sources. The repositories perform the actual database operations, and they operate on the correct physical database depending on ShardContext and ShardRoutingDataSource.

**Queue Component:** The cloud RabbitMQ Producer/Consumer on each node. The ProductSyncProducer on whichever node modifies a product will send a message to the RabbitMQ exchange. The ProductSyncConsumer on every node listens on the queue and triggers product updates to the local DB.

## Overview of components, architecture and features

**Gossip Protocol for Membership**
Each node periodically sends out heartbeats (HTTP pings) to all other nodes; if a node misses heartbeats or cannot be reached, it's suspected failed. The detecting node then initiates a gossip message to inform others of the failure.

Gossip messages (sent via UDP) come in two types: HOST_ADD (when a new node joins) and HOST_DOWN (when a node is detected failed), containing the updated membership list (finger table). Upon receiving a gossip message, nodes merge this information – adding new nodes or removing failed ones from their finger table (the data structure mapping hash ranges to node addresses).

This decentralized approach ensures eventual consistency of the cluster view: within a few gossip rounds, all nodes agree on the active members. We note that this provides a quick propagation; for example, in our tests, a node failure was broadcast to all others in ~10 milliseconds.

## Overview of components, architecture and features

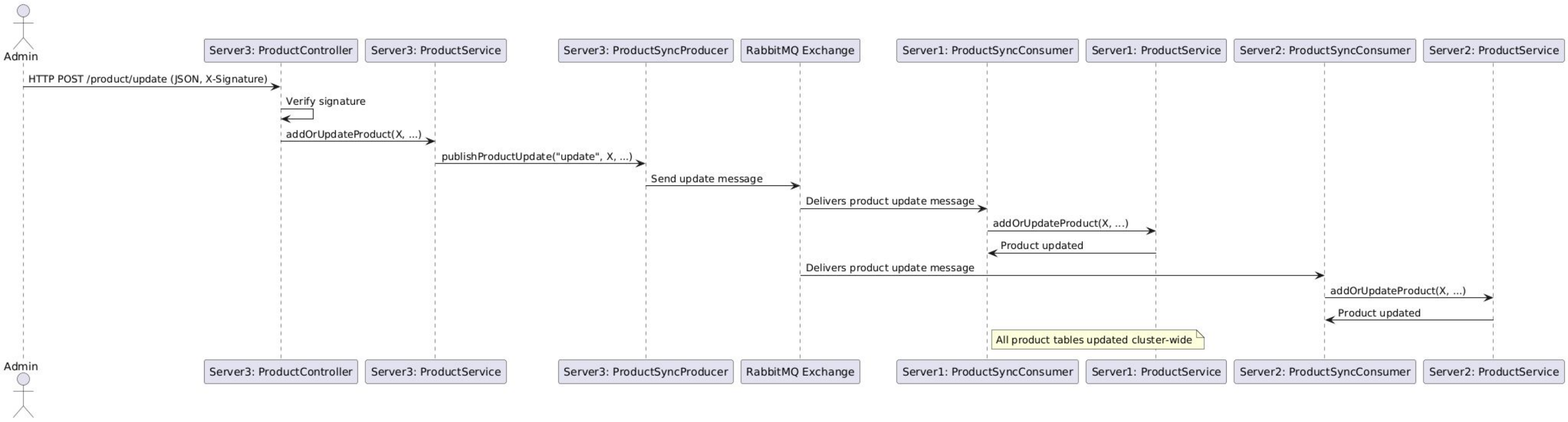
**Product Update Flow (RabbitMQ Integration)**

When a product is added or updated on any node, that node’s ProductService saves it to the local database and then produces a message to a RabbitMQ fanout exchange via the ProductSyncProducer. The message (containing the product details and operation type) is delivered to queues on all nodes.

Each node runs a ProductSyncConsumer (RabbitMQ listener) that receives the update message and applies the change to its own local product database (insert/update or delete) inside a transaction. Because we use a fan-out exchange, every node (including the originator) gets the message and thus converges to the same product state.

This design provides eventual consistency for the product catalog across the cluster, with minimal delay (on the order of a few hundred milliseconds for all nodes to sync in our tests).

# Product Sequence Diagram


Admin
Server3: ProductController
Server3: ProductService
Server3: ProductSyncProducer
RabbitMQ Exchange
Server1: ProductSyncConsumer
Server1: ProductService
Server2: ProductSyncConsumer
Server2: ProductService
HTTP POST /product/update (JSON, X-Signature)
Verify signature
addOrUpdateProduct(X, ...)
publishProductUpdate("update", X, ...)
Send update message
Delivers product update message
addOrUpdateProduct(X, ...)
Product updated
Delivers product update message
addOrUpdateProduct(X, ...)
Product updated
All product tables updated cluster-wide

## Live demo of JiDung

- Encryption
- P2P routing
- Sharding
- In the update order, demo MVCC and roll back
- Static file upload and replication
- RabbitMQ product update sync

See our demo video at: https://youtu.be/kl9JrnHCmfs

## Live demo of Haopeng

- Gossip
- HeartBeat Messaging
- Dynamic hash allocating

- Test cases: Nodes fail and recover

## Overall impact and evaluation

**Failure Detection Speed:** Our heartbeat + gossip mechanism detects node failures and informs the entire cluster almost instantaneously. In testing, when a server was killed, the failure was detected and a gossip message propagated to all other nodes in about **10 milliseconds**. This rapid detection means the system can update routing tables quickly to avoid sending requests to downed nodes.

**Product Update Consistency:** We measured the lag for **product updates** to sync cluster-wide. The worst-case delay between one node publishing an update and all other nodes applying it was about **0.3 seconds** (293 ms). This sub-second propagation ensures that the product catalog remains **eventually consistent** across the cluster in near-real-time, which is acceptable for our use case.

**Concurrent Update Handling:** We validated that our concurrency control works as designed. In a test where two nodes attempted to update the same order simultaneously, one update succeeded and the other was rejected with a **version mismatch**. The final state had only the single committed update (no interleaving corruption). This outcome confirms our **MVCC-based approach** successfully prevents lost updates and maintains consistency under concurrent writes.

## Plan on doing next

**Replicate Sharded Data for High Availability:** Currently, if a node holding a particular member/order shard fails, that shard's data is temporarily unavailable (a trade-off we accepted for simplicity). In the future, we would implement replication of those shards so that even if one node goes down, its data could be served from a replica file. This would eliminate the single-point-of-failure for each shard and improve fault tolerance.

**Enhanced Static File Distribution:** Our static files are now replicated to a node's immediate neighbor. We could improve this by replicating files to multiple nodes or using a distributed file system, ensuring files remain accessible even if two adjacent nodes fail.

**Dynamic Rebalancing & Scaling:** As we add more nodes, we might implement automatic data rebalancing. While our consistent hashing limits the scope of data movement when a node joins, a future improvement is to actually migrate a portion of data (e.g., split an overgrown shard) for better load distribution. We'd also like to test the system with a larger number of nodes and higher loads, to observe its behavior at scale and tune parameters (gossip intervals, etc.) accordingly.

**Strong Consistency and Transactions:** For use cases requiring stronger consistency, we could explore distributed transaction coordinators or consensus protocols. For example, ensuring **ACID** across shards (two-phase commit) or using a more robust consensus-backed metadata service for membership might be future directions, although they add complexity.

## Conclusion with a summary of your project's key points.

We built a functional distributed database that **scales horizontally** and handles failures gracefully. By sharding data across multiple nodes and replicating critical data, we improved throughput and availability over a single-node design. The system achieves **over 1200 req/s throughput** with negligible latency overhead, and quickly adapts to node failures(membership updates in milliseconds). We demonstrated effective solutions to key distributed systems challenges – scalability, fault tolerance, consistency, and concurrency – within our implementation.

**In summary,** our distributed database system provides a practical example of a distributed database that balances performance and consistency for web-scale applications.