Pawn

Jingyi Cui	Haocheng Sun	Jason Lok
19204584	19203637	20366363

Synopsis

The intention of our proposed system is the creation of an auction, where clients are able to view, create and bid on items existing in the market. The main features of our system consists of:

- **Create and Bid:** Accepts create and bid executions from our clients, presenting the fundamental functionality of an auction system.
- **Live feed of bids:** Displays bids that come in from all clients regardless if bid exceeds the current bid.
- **Auto-refresh on updated items:** Provides clients with the latest snapshot of the auctioned items.
- **Local Time:** Localises the time for clients, adapting for multi-timezone access.

Technology Stack

The main distribution technologies used within this project are:

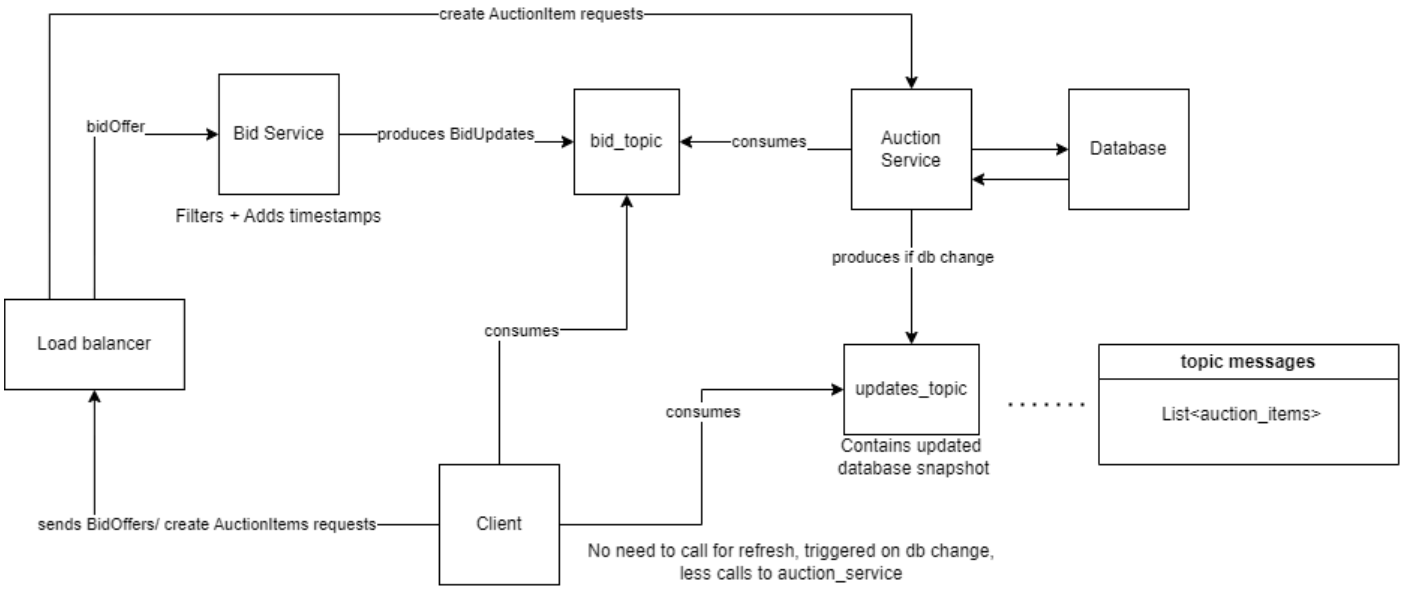
- **Nginx**: Considering our system is dealing with an unknown number of customers with an expectation of heavy traffic, our system must be required to deal with a large number of clients and incoming bids and new items. Nginx acts as a load balancer, capable of distributing incoming traffic across multiple replicated services removing bottlenecks on single servers being overwhelmed with requests. Additionally, Nginx is designed to be able to handle large volumes of concurrent connection with low memory usage.
- **Kafka**: Kafka is a distributed streaming platform used for real-time data pipelines and streaming applications. With the messages in our auction system being real time and high volume, the use of kafka is advantageous with the added benefits of built in partitioning and replication. Partitioning aligns with the architecture of our system where the services scale horizontally as partitions are divided amongst the services that read them depending on the configured group. Replication on multiple Kafka brokers is also useful for dealing with fault tolerance. Additionally the one-to-many messaging in our system requires a message broker rather than direct messaging.
- **SQL**: SQL provides a robust information store for our auction system where data integrity is vital with mechanisms such as transactions. In a scenario where placing bids simultaneously is expected to happen, the availability of ACID (Atomocity, Consistency, Isolation, Durability) transactions ensures consistency and reliability.

- **Kubernetes**: Acting as a container orchestration tool, kubernetes automates deployment, scaling and management of containerised applications. This allows our auction system to horizontally scale through the addition and removing of bid services and auction services. Additionally, it provides features for recovery and fault tolerance, enhancing reliability and minimising downtime.
- **Zookeeper**: The primary use for zookeeper in the auction system is leader election. When encountering an issue with duplicated production to our kafka topic from a scaled service, the need for a single differentiated service was required. Zookeeper being one of the main distributed coordination services allowing for the electing of a leader to handle the single task with the fallback of re-election if the leader service fails was the most suitable option to resolve this issue.

System Overview

The main components of the system are:

- **Bid Service:** This service receives bid offers from clients and applies a filter to ensure the correct format is presented. It then assigns a timestamp to the offer and labelled a bid update to be processed by our auction service.
- **Auction Service:** Our auction service is responsible for comparing incoming bids with the current bids stored in our system's database and applies updates if necessary i.e higher bid, new item and broadcasts the changes to clients.
- **Client:** The client listens to bids placed by other clients in addition to the most up to date auction details and is capable of placing bids and creating auction items of their own.
- **Load balancer:** Nginx load balancer to distribute workload across services.
- **Bid_topic**: Holds all bids from clients which were filtered for bad data.
- **Updates_topic:** Contains updated view of the items in the auction.
- **Database:** SQL database used to hold information on items in our auction system.



The typical workflow of our program is:

- With a database containing all the information on our auction system, our client is able to send bid offers and auction items to be added to the load balancer.
- From here, the load balancer forwards the request to either the bid service or auction service depending on the type of request.
- The bid offers are processed by one of the bid services which checks for negative bid values and adds the timestamp that the bid was received in our system.
- This is then forwarded to the bid_topic where all the clients are consuming from and are informed of the bid.
- The auction service also consumes these bids and determines if the newly offered price is higher than the current offered price for a given item in the database.
- Simultaneously, the auction service also deals with new items being added to the database.
- Additionally, the one instance of the auction service checks on auction item expiries when the bid session for an item is over.
- These changes act as triggers for refresh on client side so that if any changes occur, the auction services produce the current snapshot of the items to the updates_topic.#

Scalability is supported in our system with the use of Kubernetes, Nginx, Kafka, UTC and Row level locking.

- **Kubernetes**: Scalability is accounted for with the use of kubernetes for horizontally scaling the services (bid and auction) allowing for more load to be handled through the addition of more services.
- **Nginx**: The routing to these scaled services is handled by Nginx, balancing the load between the services, reducing bottlenecks and removing idle services given high traffic.
- **Kafka**: Using Kafka as our message broker has the benefits of partitions where messages are stored in partitions in the topic. These partitions are divided between auction services which similar to nginx allows for distributed workload.
- **UTC**: Another feature that supports scalability is the use of UTC (Coordinated Universal Time) for our timestamps. Utilising a universal time within our system but displaying local time on clients accommodates for geographical scalability.
- **Row Level Locking**: Lastly, the use of row-level locking in SQL supports scalability on our auction services that are to be horizontally scaled. Leveraging concurrent updates on different items in the database while avoiding race conditions preserves data integrity and allows for the scaled auction services to execute in parallel rather than wait.

Fault tolerance is handled with the use of Kubernetes' automatic recovery mechanism, Architecture design choices, SQL volumes and Zookeeper leader election.

- **Kubernetes automatic recovery mechanism**: Fault tolerance is maintained with kubernetes where the services have been replicated across multiple pods so even if one service goes down, the other replicas are able to handle the workload while it restarts. Kubernetes also provides an automatic recovery mechanism for failing pods.
- **Architecture design choice**: For failing components such as kafka and nginx, availability is reduced during downtime but is promptly restarted and continues functioning on client thereafter without crashing the client. There is loss of transaction logs within the kafka topics in our system which can be managed with replication to multiple kafka brokers but since the client-facing updates_topic is dependent on our SQL database data which should be updating often and the bid_topic's logs isn't a priority, this wasn't implemented in this version.
- **SQL Volume**: The SQL database utilises volumes to maintain its record of item data even after it fails over and restarts but also incorporates transaction rollbacks if update transactions have not been committed.
- **Zookeeper leader election**: Lastly, the use of zookeeper for leader election also helps with ensuring the checking of item expiry times is functional as when a leader is terminated, a new leader will be elected.

Contributions

Jingyi Cui:

- Proposed project architecture, designed the components of the project, and represented the operational processes between each part through diagrams.
- Introduced Nginx as load balancer
- Implemented the bid service which will handle the bid requests and forward to message middleware.
- Designed DB table schema and Kafka messages schema.
- Implemented client console interface that allows users to interact with the system in multi threads fashion.

Haocheng Sun:

- Worked on Restructuring of project architecture
- Worked on auto-refresh feature
- Worked on UTC timestamp for geographically scalability
- Helped with Zookeeper for leader election on auction services

Jason Lok:

- Implemented the foundation for auction service
- Set up SQL and SQL initialization
- Transitioned system to Kubernetes
- Introduced Zookeeper for leader election
- Documentation

Reflections

The key challenges involved:

- **Architecture Design:** There were conflicting opinions within the group on which features would be required as well as the functionality of each component. Each of us had our own version of an auction system in mind and this resulted in clashes in design. This was quickly resolved with further discussion and a unified design was accepted.
- **Exposing kubernetes clusters to local machines:** Attempting to expose our pods to the local machine for our Client caused a lot of issues. Various methods were attempted such as Ingress and NodePorts but led to further complications. Inevitably, the use of port-forward was used, mapping our local port to a pod port which is simple and commonly used for development.
- **UTC Timestamp:** The use of a universal timestamp was important for the scalability of our auction system. The problem was initially discovered after large-scale refactoring of the structure of the codebase, requiring type changes on our different time variables from LocalDateTime -> Timestamp to be eligible for Kafka Topics. After which inconsistencies in timestamps appeared for the returned items in the table leading to issues with verifying auction item expiry times too. This led to further refactoring to utilise UTC as our core timestamp for geographic scalability.

Things I would do differently are:

- **Experimenting with a different database system:** Although SQL worked sufficiently well in this small project, the use of a distributed database could be more suitable with better fault tolerance mechanism and high availability with the use of sharding and partitions.
- **Clean up the architecture:** There were some components that could have been extracted into its standalone service i.e adding items to database, updating bids in database and checking for expired item times could have been separated into several microservices.
- **Use additional Kafka brokers:** Using additional kafka brokers with an increased replication factor allows for our system to become more robust in terms of fault tolerance and availability. When the kafka pod fails, we lose messages that were in the topic. Our client, on startup, checks the latest updates_topic message for an initial auction item set so a scenario exists where they may receive nothing given a polling time of 10 second. This information can also be useful if expansion into logging and sales history was implemented.

Learnings from technologies used:

- **Zookeeper-less Kafka**: During investigation, we have learned that the newer version of Kafka is no longer dependent on external services (Apache Zookeeper) for leader election which introduced both configuration overhead and stability issues since the Zookeeper cluster may go down. Instead, Kafka implemented the Raft Algorithm to achieve consensus during leader election.

- **Nginx limitations:** Nginx is an extremely performant reverse proxy and load balancer, however it depends on static configuration files, and hence lacking dynamically scalability. Comparatively, Kubernetes' native Service object can serve a similar job as load balancer, and since it is running on Kubernetes, scaling out can be just issuing one command to increase the number of replicas in a replica set.
- **Kubernetes:** Kubernetes is an extremely useful tool for orchestration of containers allowing for automated deployment, scaling and management of containers. It carters for scalability with the built in deployment scaling mechanisms, high availability with automatic failover and offers built in service discovery and load balancing. However, there is a lot to learn from Kubernetes and only a small segment of its available tools have been used in this project. A downside of using minikube is its single-node cluster, preventing the exploration of node level fault tolerance with affinities.
- **Zookeeper:** The use of zookeeper for leader election was interesting as a means for distinguishing between a single instance of a service and the other replicas. Alongside this, there is an automatic failover mechanism for re-electing leaders. However, zookeeper does have practical limits on the number of services it can support as well as having a single point of failure. If the leader fails, clients may not receive updates on expired auction items until a new leader is elected resulting in downtime.