Project Presentation
CS744 Autumn 2024

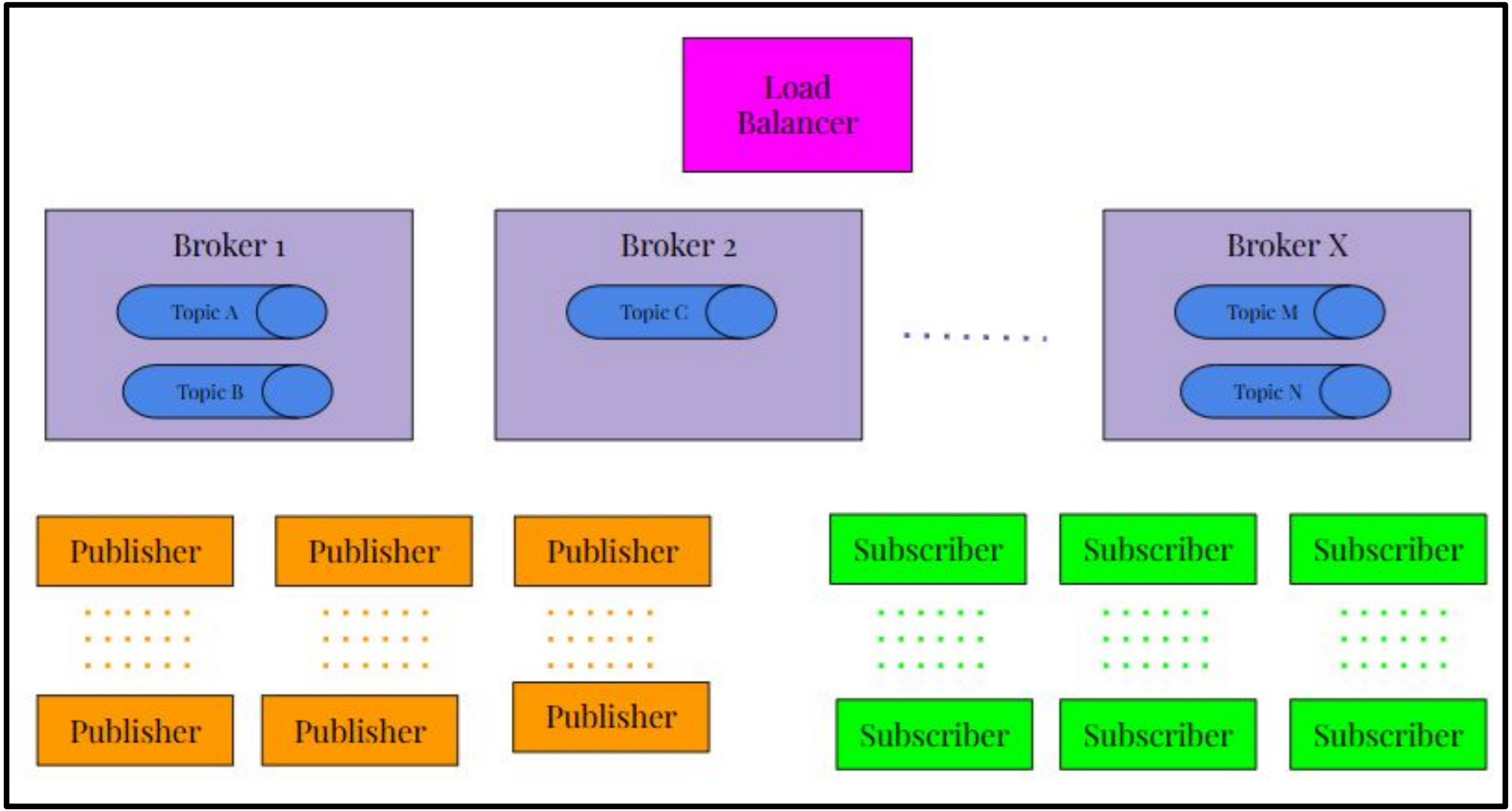
Distributed Publish-Subscribe Platform

Byte Architects

24m0767 - Ayush Pratap Singh
24m0775 - Praveen Kapildev Maurya

https://github.com/ayush-0727/Pub-Sub-Platform.git

Distributed Publisher - Subscriber Model



Context

- **Some major Systems /software Architecture Patterns are:**
  - Client-server, Event-driven, Peer-to-peer, Broker
- **If our goal is to create :**
  - Applications which need to communicate information to multiple consumers, which may have different availability requirements or uptime schedules than the sender
  - If we want a distributed system where different parts interact with each other, but are **'not tightly coupled'**
  - Publisher - Subscriber model can come to the rescue
- **What is loose coupling of parts / entities / senders / receivers?**
  - Senders / receivers (of message) do not interact directly with each other; senders don't know who receivers are, receivers don't know who senders are

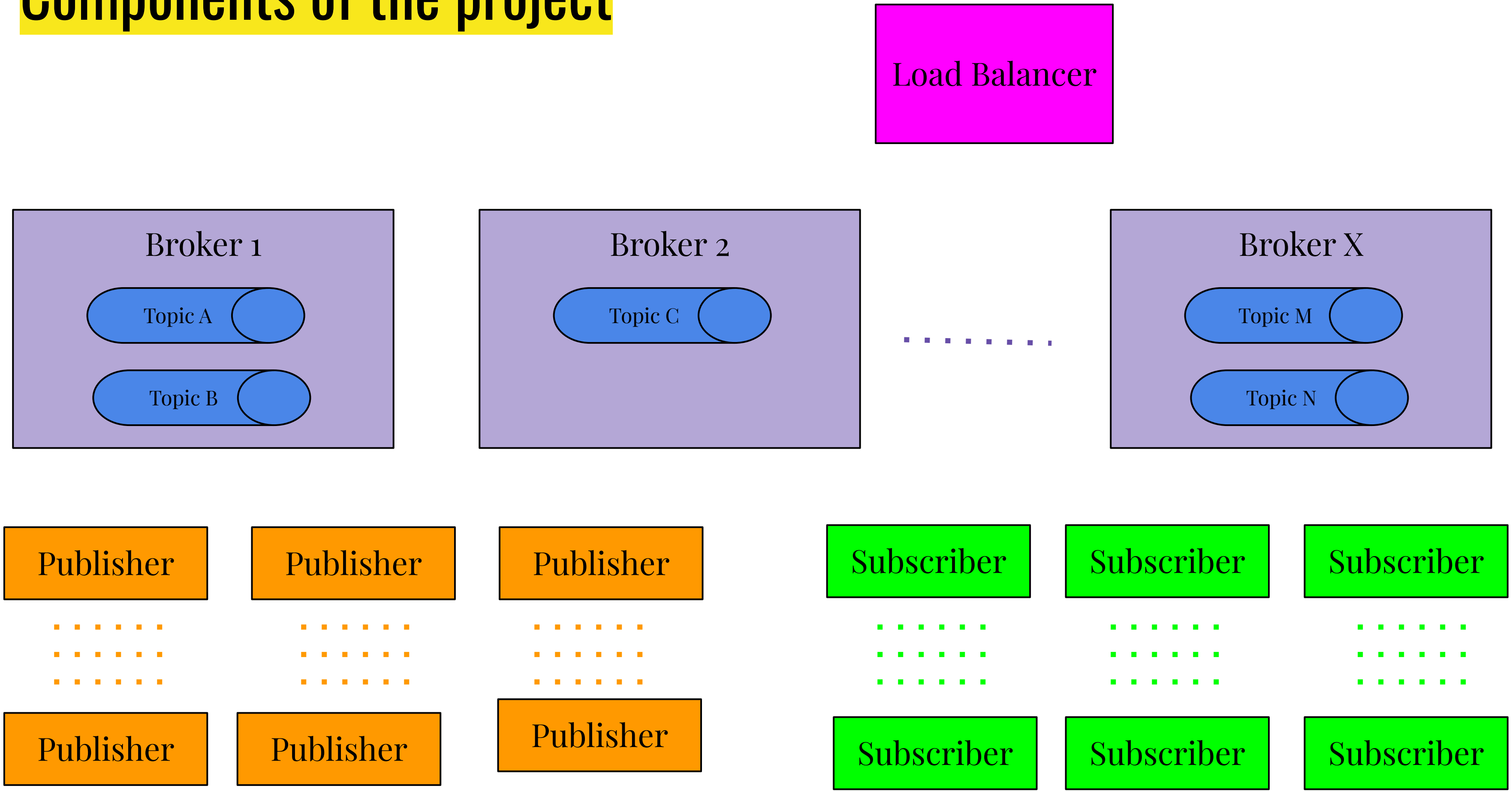
Problem description

- **Problem Statements:**
  - Design a <u>scalable</u> and <u>reliable</u> message delivery system where senders (publishers) and receivers (subscribers) interact with each other loosely
  - Publishers can <u>publish messages to multiple topics</u> / channels, Subscribers can <u>subscribe and receive messages from a select set of topics</u>
- **Scope / Goals / Deliverables:**
  1. **Load balancer (server) :** distributes client (publishers & subscribers) requests across multiple brokers
  2. **Multiple brokers (servers) :** handle incoming requests from publishers and subscribers

Problem description

- ➔ **Scope / Goals / Deliverables:**
  3. **Multiple Publishers :** who publish to multiple topics
  4. **Multiple Subscribers :** who subscribe and receive messages from multiple topics

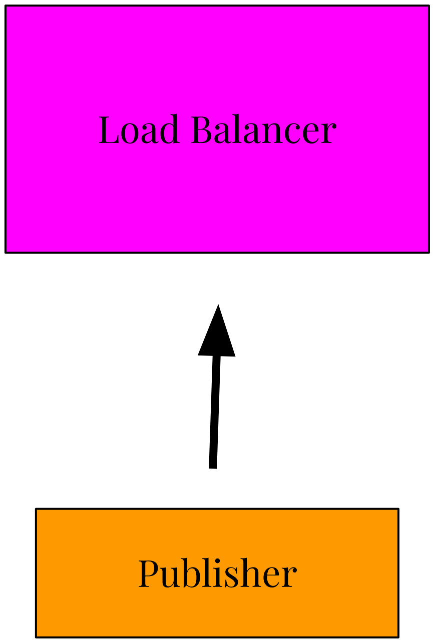
Components of the project



Design

New Topic creation

- Publisher expresses request to create a new topic
- Load balancer selects a broker in round-robin manner and maps the topic to chosen broker



Subscriber subscribes to a topic

Step 1:

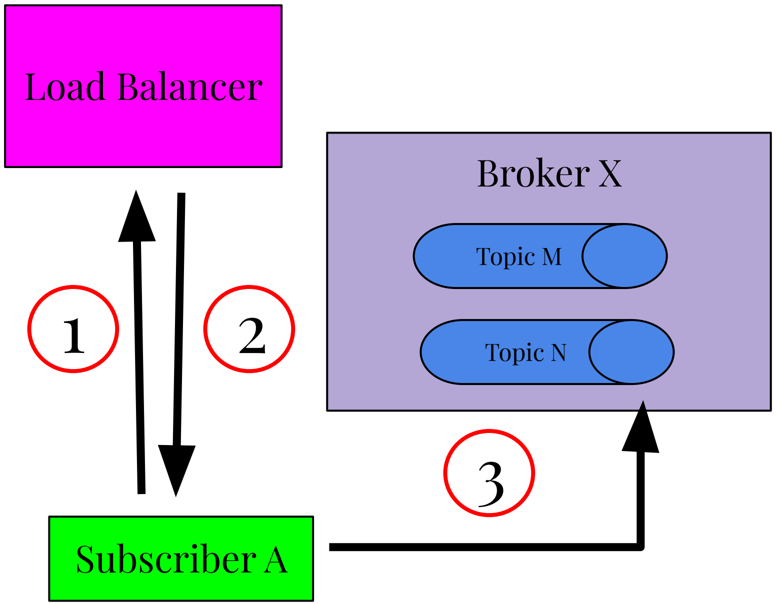
- 'Subscriber A' sends message to load balancer for subscribing to topic (e.g. topic M)

Step 2:

- Load balancer checks mapping of topic vs broker
- Sends back IP address and port of respective broker (e.g. Broker X)
- Connection terminates with load balancer

Step 3:

- Subscriber creates connection with Broker X
- Broker X saves connection details of 'Subscriber A' along with topic name in a 'subscribers list'



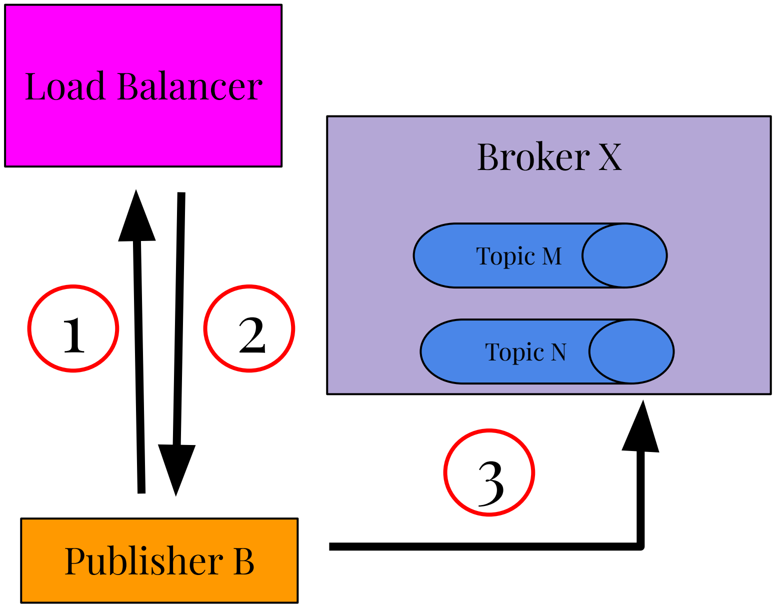
Publisher publishes to a topic

Step 1:

- 'Publisher B' sends a publish message (e.g. Topic M) request to load balancer

Step 2:

- Load balancer checks the mapping of topic vs. broker
- Sends back IP address and port of the respective broker (e.g. Broker X)
- Connection terminates between load balancer & Publisher B



Step 3:

- Publisher B creates connection with Broker X and sends <topic, msg> details
- Broker X extracts the topic name sent by publisher, searches each subscriber in the 'subscriber's list' and sends <msg> to all respective subscribers

Implementation details

Data Structures used:

1. Common data structure:

```c
typedef struct {
    MessageType type;           // enum
    char topic[50];             // topic name
    char data[MAX_DATA_LEN];    // message
} Message;
```

```c
typedef enum {
        PUBLISH,
        SUBSCRIBE,
        CREATE_TOPIC
} MessageType;
```

2. Broker specific data structures:

a)

```c
typedef struct {
    int fd;                     // TCP connection file descriptor
    char topic[BUFFER_SIZE];    // topic name
} Subscriber;
```

b)

```c
Subscriber subscribers[MAX_CLIENTS];  // Array storing all connected subscribers
```

3. Load balancer specific data structures:

a)

```c
typedef struct {            // broker entity
    char ip[BUFFER_SIZE];   // broker's IP address
    int port;               // broker's port no
} Broker;
```

b)

```c
Broker brokers[MAX_BROKERS]; // array of live brokers
```

c)

```c
typedef struct {                  // topic vs. broker mapping
    char topic[BUFFER_SIZE];      // topic name
    char broker_ip[BUFFER_SIZE];  // broker ip
    int broker_port;              // broker port
} TopicBrokerMapping;
```

d)

```c
TopicBrokerMapping topic_mapping[MAX_TOPICS]; // array of mappings
```

4. Subscriber specific data structures:

a)
```
typedef struct {                    // get broker ip/port from load balancer
    char topic[50];                 // topic name
    char broker_ip[BUFFER_SIZE];    // IP of broker mapped to topic
    int broker_port;                // Broker's port
} BrokerInfo;
```

b)
```
typedef struct {        // stores socket fd after connecting with broker
    char topic[50];     // topic name
    int broker_sock;    // connection socket fd with broker
} BrokerConnection;
```

c)
```
// array of connections with brokers
BrokerConnection broker_connections[MAX_TOPICS];
```

❖ Interface seen by the publisher:

Publisher sees an input menu repeatedly for actions:

1) Create a new topic; 2) Publish a new message; 3) Exit

Activity 1 : Publisher creates a new topic

1. Publisher establishes connection with load balancer
2. Creates a *struct Message Msg*
3. Takes user input for *topic* name
4. Sets *MessageType* as CREATE_TOPIC
5. Sends creation request via *Msg* to load balancer using *send(sock, &msg, ...)*
6. Load balancer receives *Msg*, sees that message type is CREATE_TOPIC, assigns topic to brokers in round-robin fashion, finds ip/port details of mapped broker and stores it in topic_mapping[MAX_TOPICS] array

❖ Interface seen by the subscriber:

Subscriber sees an input menu repeatedly for actions:

1) Subscriber to a new topic
2) Listen for new message
3) View subscribed topics
4) Exit

Activity 2 : Subscriber subscribes to a topic

1. Subscriber takes user input for *topic* name
2. Establishes connection with load balancer at *sock*
3. Creates a *struct Message Msg* and stores user input in *msg.topic*
4. Sets *MessageType* as SUBSCRIBE
5. Sends *Msg* to load balancer via *send(sock, &msg, ...)*
6. Receives response from load balancer containing broker ip/port

Activity 2 : Subscriber subscribes to a topic (continued........)

7. Establishes connection with broker at *broker_ sock*
8. Creates a *struct Message Msg* and stores user input in *msg.topic*
9. Sets *MessageType* as SUBSCRIBE
10. Sends *Msg* (subscription request) to broker via *broker_ sock* using *send(broker_ sock, &msg, ...)*
11. Broker uses *Select* system call for I/O multiplexing. Broker listens the subscription request on its listening port *server_ fd* (which is part of fd_set *read_ fds*)
12. Broker identifies msg's *MessageType* as SUBSCRIBE and adds requesting subscriber's connection_fd and topic to subscribers[MAX_CLIENTS] array
13. Subscriber stores *broker_ sock* & topic mappings into broker_connections [MAX_TOPICS]

Activity 3 : Publisher publishes to a topic

1. Publisher establishes connection with load balancer at *sock*
2. Creates a *struct Message Msg*
3. Takes user input for *topic* name and *message* and store in *Msg*
4. Set *MessageType* as PUBLISH
5. Sends publish request to load balancer using *send(sock, &msg, ...)*
6. Load balancer receives *Msg*, sees that message type is PUBLISH, finds the broker assigned to the topic
7. Load balancer establishes connection with the broker and forwards message to the broker
8. Broker accepts the connection request of load balancer on listening port, and adds the connected socket to fd_set *read_fds*

Activity 3 : Publisher publishes to a topic (continued....)

9. Broker reads message received on newly added fd in *read_ fds* and identifies its *MessageType* as PUBLISH
10. Broker broadcast the received message to all valid subscribers of the topic by scanning the subscriber's list *subscribers*

Activity 4 : Subscriber listens for new topical messages

11. Subscribers maintain list of connected brokers in *broker_ connections* array; and keep checking for activity on those fds via select system call
12. When broker broadcasts the published message, subscriber detects activity on fd and reads message

Evaluation

Questions :

1. Can a single load balancer handle and scale adequately for increasing number of publishers and subscribers ? Can it become a bottleneck?
2. How threading of load balancer impacts the overall efficiency of the system?
3. How threading of brokers impact the performance of the system?

Evaluation

- **Experimental Setup :**
  a. Using a load tester to simulate concurrent running of multiple publishers and subscribers
  b. Tracking the number of messages exchanged and time taken for varying thread pool sizes of load balancer
  c. Observed the data and plotted the graph between thread pool sizes and throughput

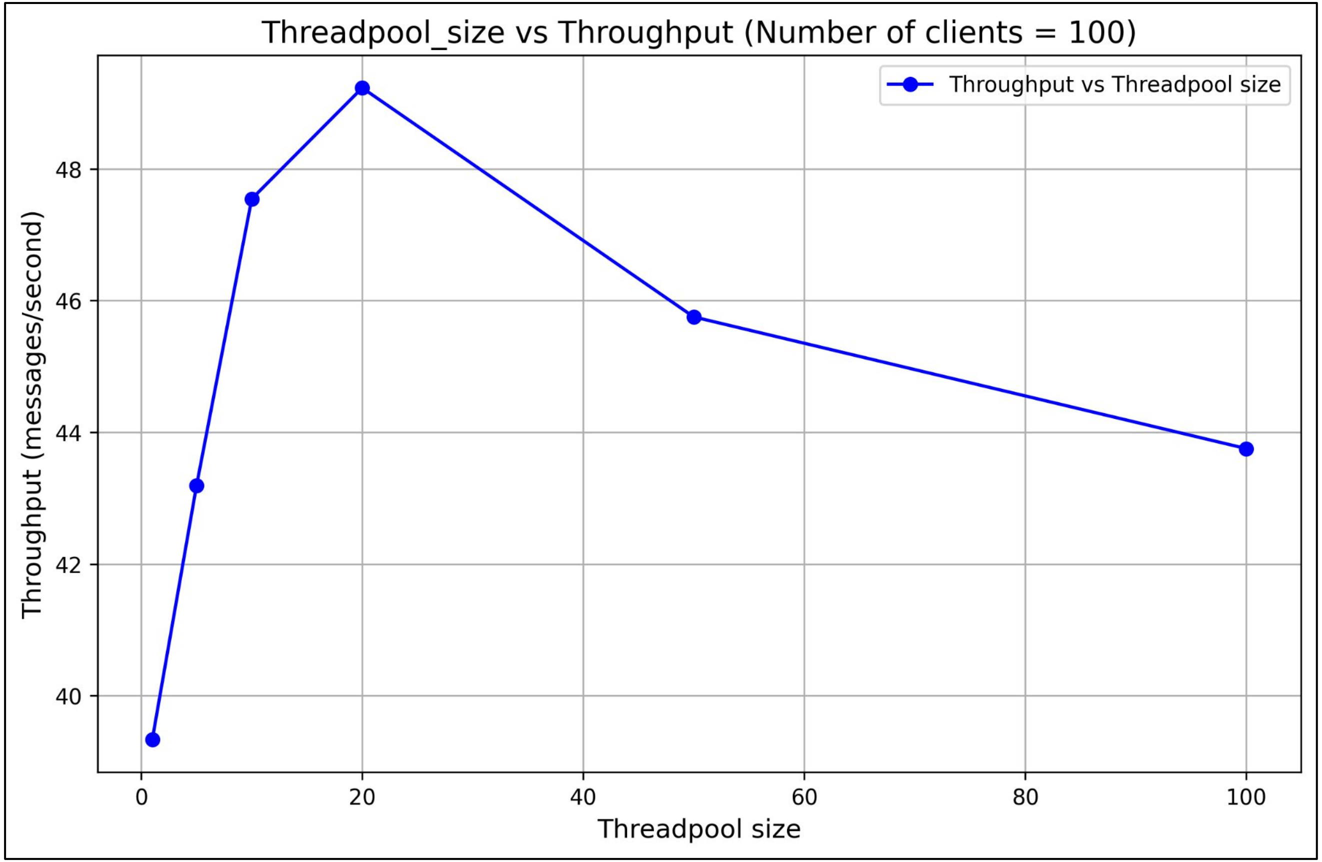
Evaluation

- **Parameters :**
  a. Number of Publishers
  b. Number of Subscribers
  c. Message Frequency
  d. Number of Topics

- **Metrics :** Throughput (messages delivered / second)

Title of experiment

Tracking the system throughput (messages delivered / second) for varying thread pool sizes of load balancer



Summary of results

- Increase in threadpool size enhances throughput upto a certain limit
  - Maximum throughput of 50 messages/sec occurred at threadpool size = 20
- **Inferences:**
  - Initial Increase in Throughput due to:
    - Improved Parallelism
    - Better resource utilization
  - Throughput decreases after a certain point due to:
    - Resource Contention
    - Context Switching Overhead
    - I/O Bottlenecks
    - Memory Overhead

Unfinished scope

- **Map topics to multiple brokers :**
  - Currently each topic is mapped to a single broker leading to <u>low fault tolerance capabilities</u>
  - Leads to scenarios where the broker goes down and respective topics become inaccessible to publishers and subscriber
- **Use of persistent storage mechanism :**
  - Use of in-memory storage for data compromises reliability of the whole distributed system
  - Leads to scenarios where broker goes down and all topic-subscriber mappings are lost
- **Implementation of back-up load balancers**

Challenges

- Implementing multiple publishers and subscribers on multiple systems was a challenging aspect of the project
- Defining metrics for measuring performance
- Designing an efficient Load tester to measure system performance and correctness

Reflection

- **Interesting aspect:** opportunity of implementing multiple entities running on different systems

- **What would we do differently?**
  - Implement the system with either files or databases (SQLite/Redis)
  - Authentication based publisher and subscriber

Conclusions

- Publisher - Subscriber models scale effectively in scenarios where senders and receivers interact with each other asynchronously

References

1. https://www.geeksforgeeks.org/types-of-software-architecture-patterns/
2. https://learn.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber
3.