Implementing a Pub-Sub Architecture for Real-Time Organizational Notifications

Anwitha Arbi | Rebecca Dsouza | Simran Dhawan

Proposed System

- Live news notification system leveraging the publisher-subscriber concept of message delivery

- Enable subscribers to get real time notifications about topics (eg. Finance, Layoffs etc) they have subscribed to within an Organization.

- Example: Organization ‘Maple Newspaper’ readers can subscribe to topics such as Tech / Sports / Finance etc to receive live updates about news in these fields

- Main components :
  1. Publisher
  2. Subscriber
  3. Broker

Related Work

1. A Fair Comparison of Message Queuing Systems

2. Scheduling messages within MQTT shared subscription group in the clustered cloud architecture

3. MQTT-ST: a Spanning Tree Protocol for Distributed MQTT Brokers

4. Kafka: a Distributed Messaging System for Log Processing

5.Efficient Message Diffusion in Distributed Publish-Subscribe Systems

6. PopSub: Improving Resource Utilization in Distributed Content-based Publish/Subscribe Systems

Design Choices

Dual-Broker Architecture with Primary-Secondary Roles:

- *Challenge Addressed: Fault Tolerance and High Availability.*
- By having two brokers, each serving as a primary and a secondary, the system ensures continuous operation even if one broker fails. This design mitigates the risk of a single point of failure, a prevalent challenge in distributed systems.

Deployment on Amazon EC2 Instances:

- *Challenge Addressed: Scalability and Resource Management.*
- Utilizing EC2 instances allows the system to scale resources up or down based on demand. This cloud-based approach addresses the challenge of resource allocation in distributed systems, ensuring the system can handle varying loads without physical hardware limitations.

Message Replication Across Both Brokers:

- *Challenge Addressed: Data Integrity and Redundancy.*
- Discussion: By replicating messages in both the primary and secondary broker queues, the system safeguards against data loss. This approach addresses the challenge of maintaining data integrity in distributed environments where system components can fail.

Heartbeat Mechanism for Real-Time Synchronization:

- ***Challenge Addressed: Consistency and State Synchronization.***
- The heartbeat mechanism ensures that the state of the primary broker is continuously synced with the secondary. This feature addresses the challenge of maintaining consistency across distributed components.

Security Measures for Subscriber Access:

- ***Challenge Addressed: Security and Access Control.***
- Restricting access to registered subscribers ensures secure communication, addressing critical security concerns in distributed systems.

Implementation of Multi-Threading:

- ***Challenge Addressed: Concurrency and Performance Optimization.***
- The use of multi-threading allows simultaneous processing of multiple tasks, such as message handling and heartbeat checks. This design choice effectively addresses the challenge of optimizing performance and managing concurrency in a distributed environment, ensuring that the system can handle multiple operations efficiently without blocking or resource contention.

Design goals

- Fault tolerance
- Scalability
- Data Integrity
- Consistency
- Security
- Concurrency & Performance

Algorithms

Heartbeat Protocol:

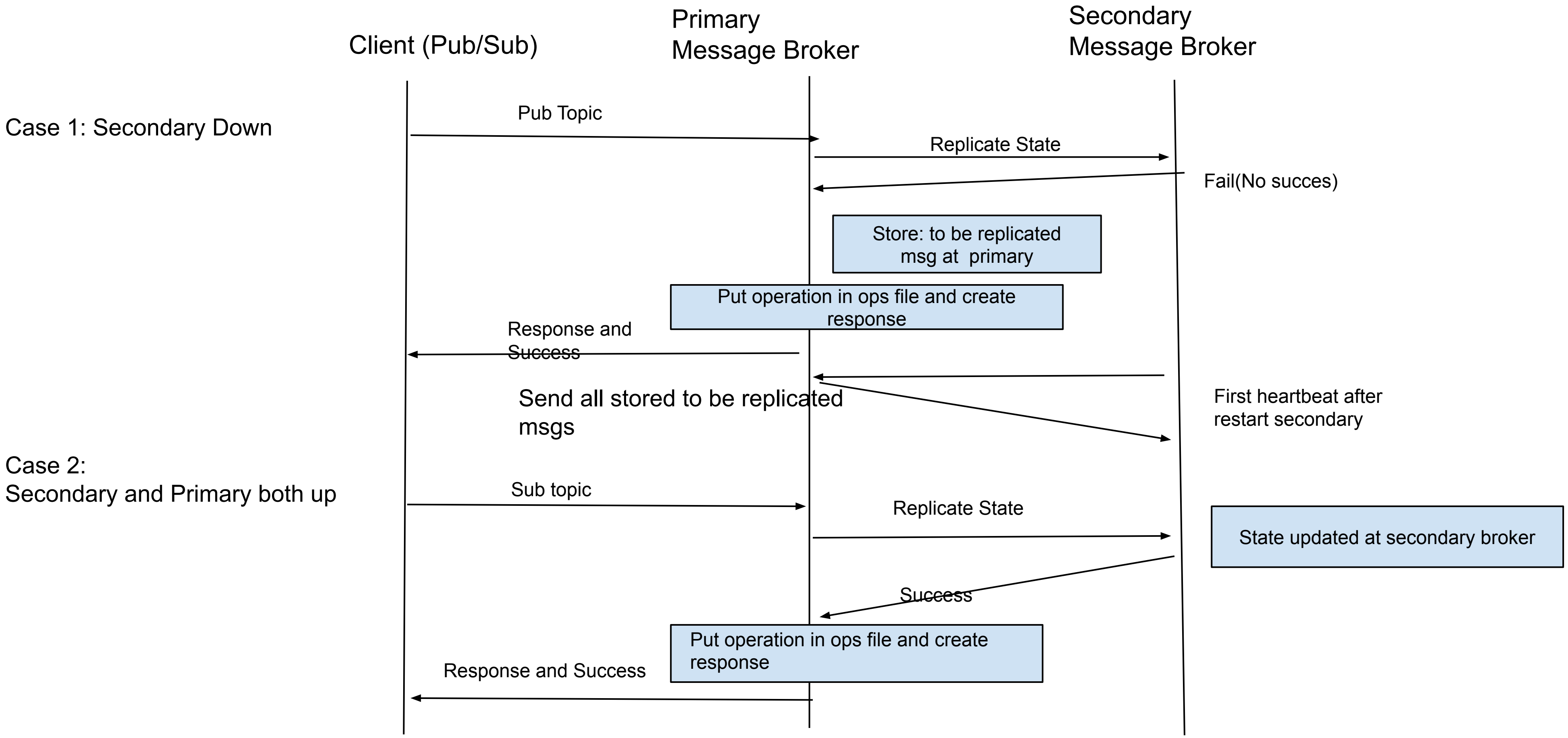
The heartbeat protocol is implemented between two brokers. Each of the broker send the heartbeat to all the replicas with information as its own ip. This way each broker tracks which replica gets information when any broker goes down. The heartbeat is sent every 5 seconds, so failure in the system is detected after 5 seconds.

Replication and Consistency Protocol:

- Primary broker receives a request from the client and the primary broker sends the same request to the replica if it is alive.
- If it is not alive the request towards replica will not succeed and it will store the request as a pending request to be sent to the replica as soon as it comes up(It will receive information about the replica coming up from the heartbeat protocol implemented). Only after the state in secondary broker(if alive) is updated along with the primary broker, it will send the response towards the client.
- Also, whenever any broker receives the message it stores that message in ops file which is used by the broker to recover its state once it goes down.

Thus protocol implemented is **fault tolerant** and states at two brokers can recover as soon as they come up. Since it is two broker system so failure of any one system will not impact the performance.

Architecture (Replication and Consistency Protocol)


Client (Pub/Sub)
Primary Message Broker
Secondary Message Broker
Case 1: Secondary Down
Pub Topic
Replicate State
Fail(No succes)
Store: to be replicated msg at primary
Put operation in ops file and create response
Response and Success
Send all stored to be replicated msgs
First heartbeat after restart secondary
Case 2: Secondary and Primary both up
Sub topic
Replicate State
State updated at secondary broker
Success
Put operation in ops file and create response
Response and Success

Concurrency Protocol:

- **Optimistic concurrency control**
- **Valid assumption** that each object can be touched by one subscriber/ publisher for subscribing or publishing topic. Message broker is creating separate queues for publisher topic and data and separate queue for each subscriber and its topic and data index.
- We have stored operation files we can always **rollback** and restart the system.

Broadcast Protocol

- Two message brokers identifies the presence of each other via broadcast protocol. The two message brokers add each other as list of replica from the menu.
- In future, when more than two brokers are there then there presence can also be known.

Features

Broker

- List Brokers
- Add Replica
- Initialize Broker

Publisher

- Publish Topic

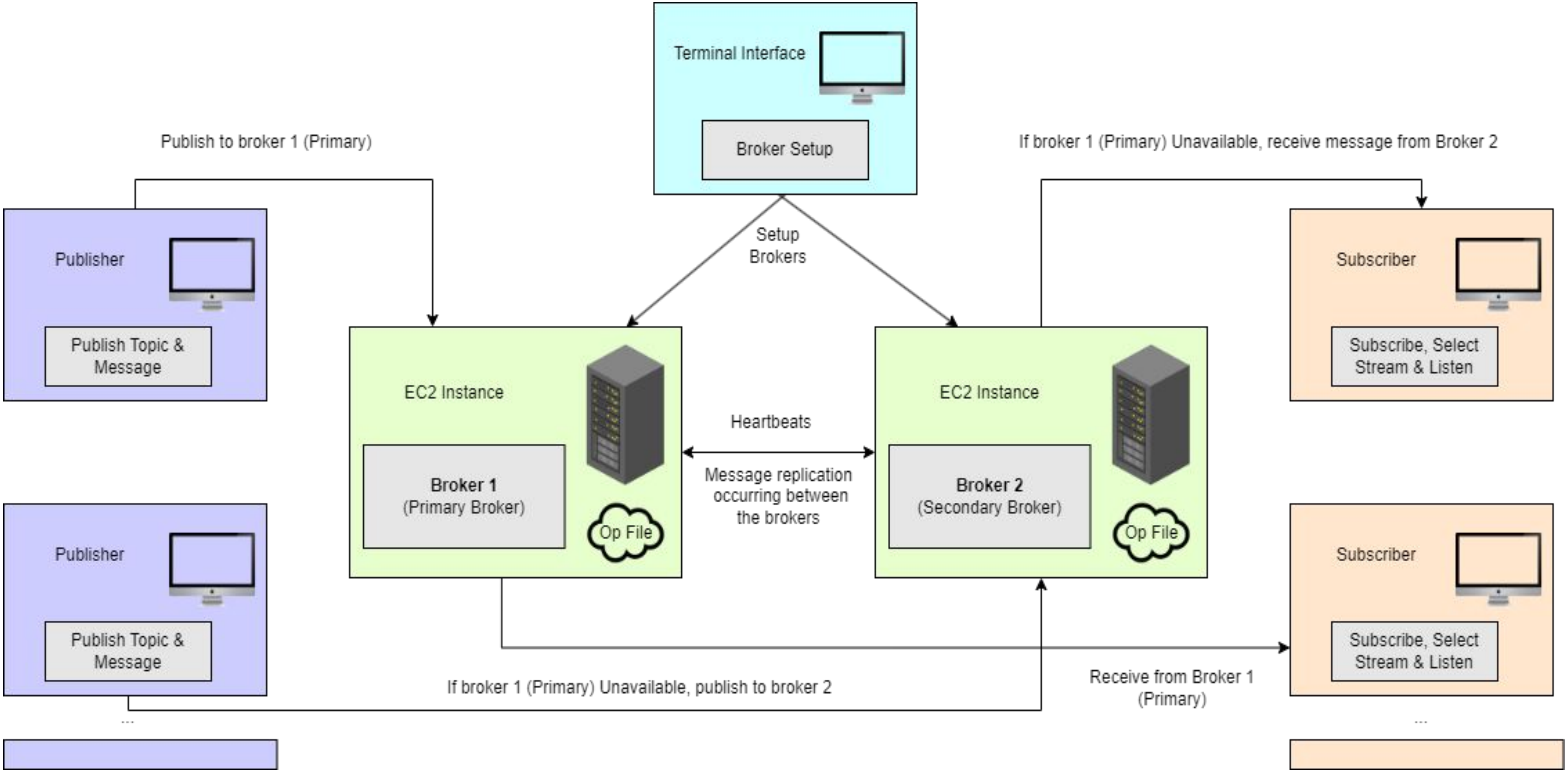
Subscriber

- Register Self and Subscribe
- Stream Messages

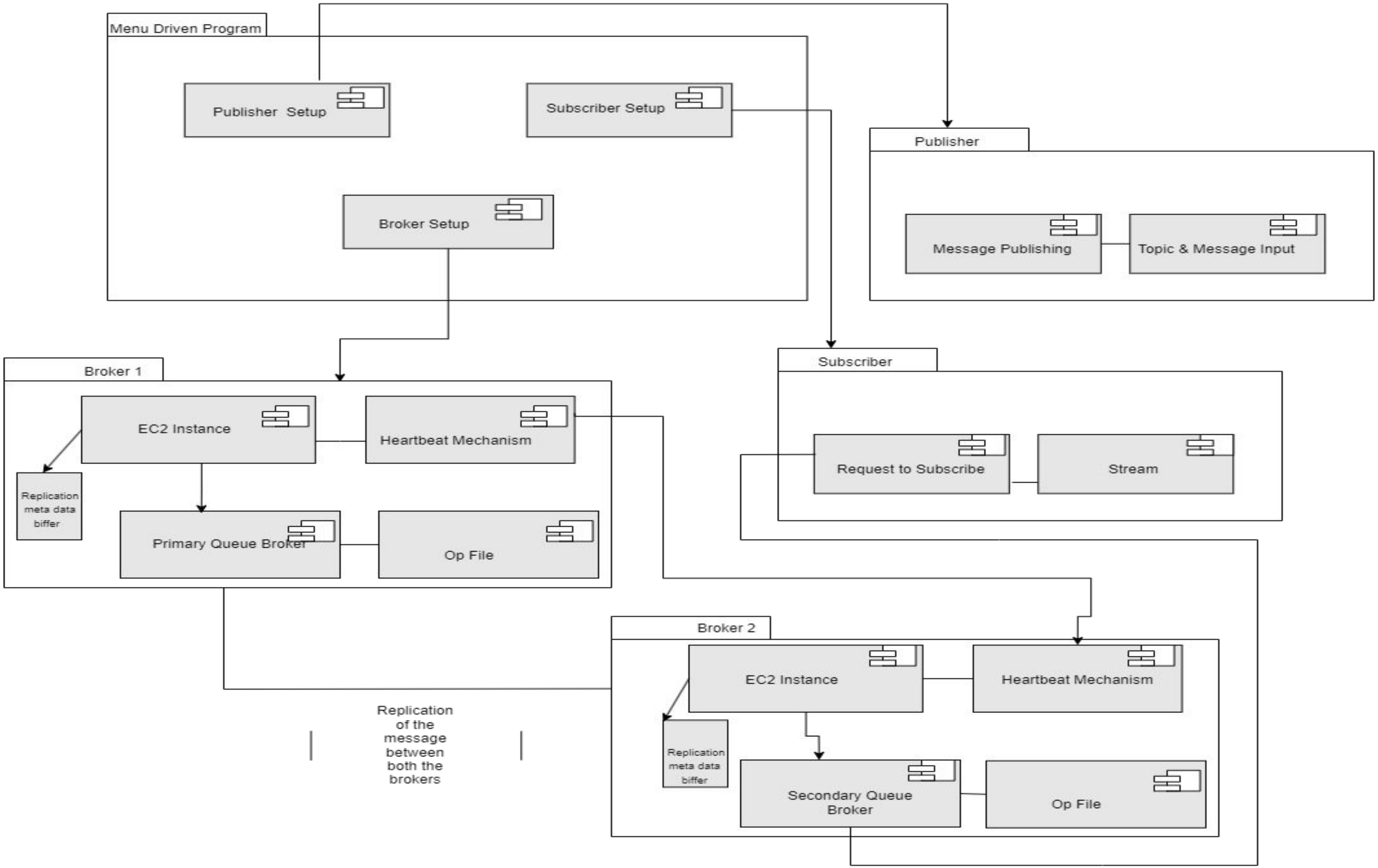
Primary <-> Secondary

- Heartbeat
- Replicate/Publisher
- Replicate/Subscriber
- Replicate/Stream
- Restore state (after failure)
- Storing operations/snapshots of individual system for restart

System Architecture


Terminal Interface
Broker Setup
Setup Brokers
Publish to broker 1 (Primary)
If broker 1 (Primary) Unavailable, receive message from Broker 2
Publisher
Publish Topic & Message
EC2 Instance
Broker 1
(Primary Broker)
Op File
Heartbeats
Message replication occurring between the brokers
EC2 Instance
Broker 2
(Secondary Broker)
Op File
Subscriber
Subscribe, Select Stream & Listen
Publisher
Publish Topic & Message
Subscriber
Subscribe, Select Stream & Listen
If broker 1 (Primary) Unavailable, publish to broker 2
Receive from Broker 1 (Primary)
...
...

Component Diagram



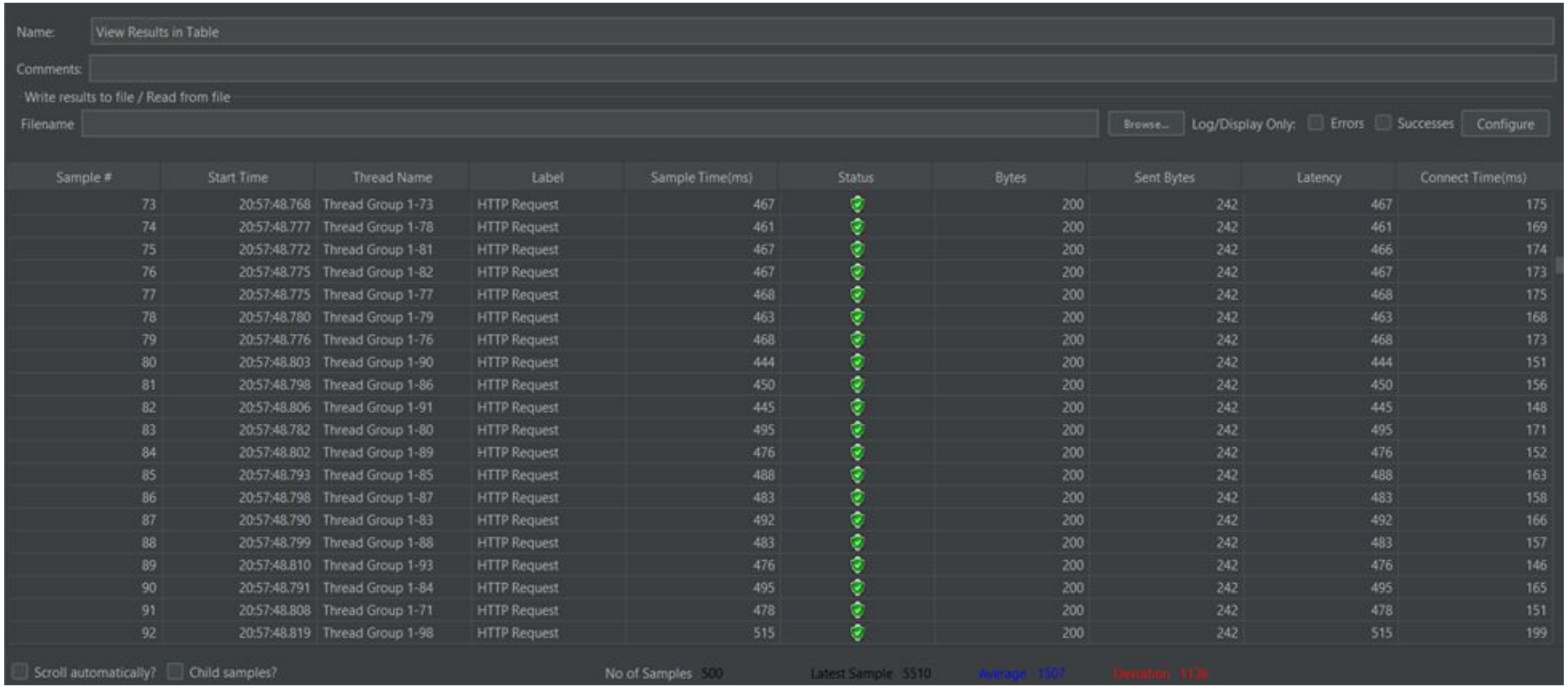
DEMO

Test Case 1 : Multiple users subscribing to the same topic

Test Case 2: Single user subscribed to multiple topics
Test Case 3: Subscribe able to consume the missed messages whenever he is back online in the network.
Test Case 4: Unauthorized subscribers not able to subscribe to the topics.
Test Case 5: Heartbeat protocols
Test Case 6: Subscribers redirected to the secondary message broker when primary message broker fails.
Test Case 7: Secondary message broker restoring it states after the failures.

Jmeter Performance Test for Publishing topics

No of concurrent publishers= 500



Name: View Results in Table

Comments:

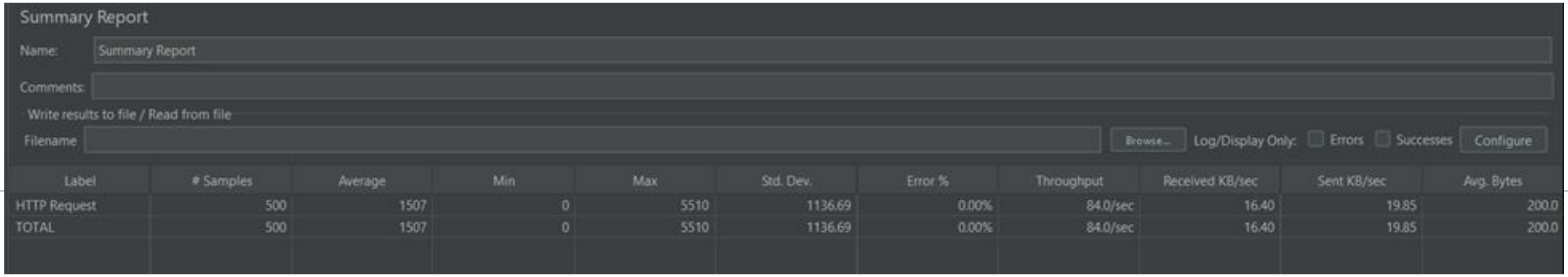
Write results to file / Read from file

Filename

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(ms)
73	20:57:48.768	Thread Group 1-73	HTTP Request	467	✓	200	242	467	175
74	20:57:48.777	Thread Group 1-78	HTTP Request	461	✓	200	242	461	169
75	20:57:48.772	Thread Group 1-81	HTTP Request	467	✓	200	242	466	174
76	20:57:48.775	Thread Group 1-82	HTTP Request	467	✓	200	242	467	173
77	20:57:48.775	Thread Group 1-77	HTTP Request	468	✓	200	242	468	175
78	20:57:48.780	Thread Group 1-79	HTTP Request	463	✓	200	242	463	168
79	20:57:48.776	Thread Group 1-76	HTTP Request	468	✓	200	242	468	173
80	20:57:48.803	Thread Group 1-90	HTTP Request	444	✓	200	242	444	151
81	20:57:48.798	Thread Group 1-86	HTTP Request	450	✓	200	242	450	156
82	20:57:48.806	Thread Group 1-91	HTTP Request	445	✓	200	242	445	148
83	20:57:48.782	Thread Group 1-80	HTTP Request	495	✓	200	242	495	171
84	20:57:48.802	Thread Group 1-89	HTTP Request	476	✓	200	242	476	152
85	20:57:48.793	Thread Group 1-85	HTTP Request	488	✓	200	242	488	163
86	20:57:48.798	Thread Group 1-87	HTTP Request	483	✓	200	242	483	158
87	20:57:48.790	Thread Group 1-83	HTTP Request	492	✓	200	242	492	166
88	20:57:48.799	Thread Group 1-88	HTTP Request	483	✓	200	242	483	157
89	20:57:48.810	Thread Group 1-93	HTTP Request	476	✓	200	242	476	146
90	20:57:48.791	Thread Group 1-84	HTTP Request	495	✓	200	242	495	165
91	20:57:48.808	Thread Group 1-71	HTTP Request	478	✓	200	242	478	151
92	20:57:48.819	Thread Group 1-98	HTTP Request	515	✓	200	242	515	199

□ Scroll automatically? □ Child samples?

No of Samples 500 Latest Sample 5510 Average 1507 Deviation 1136



Summary Report

Name: Summary Report

Comments:

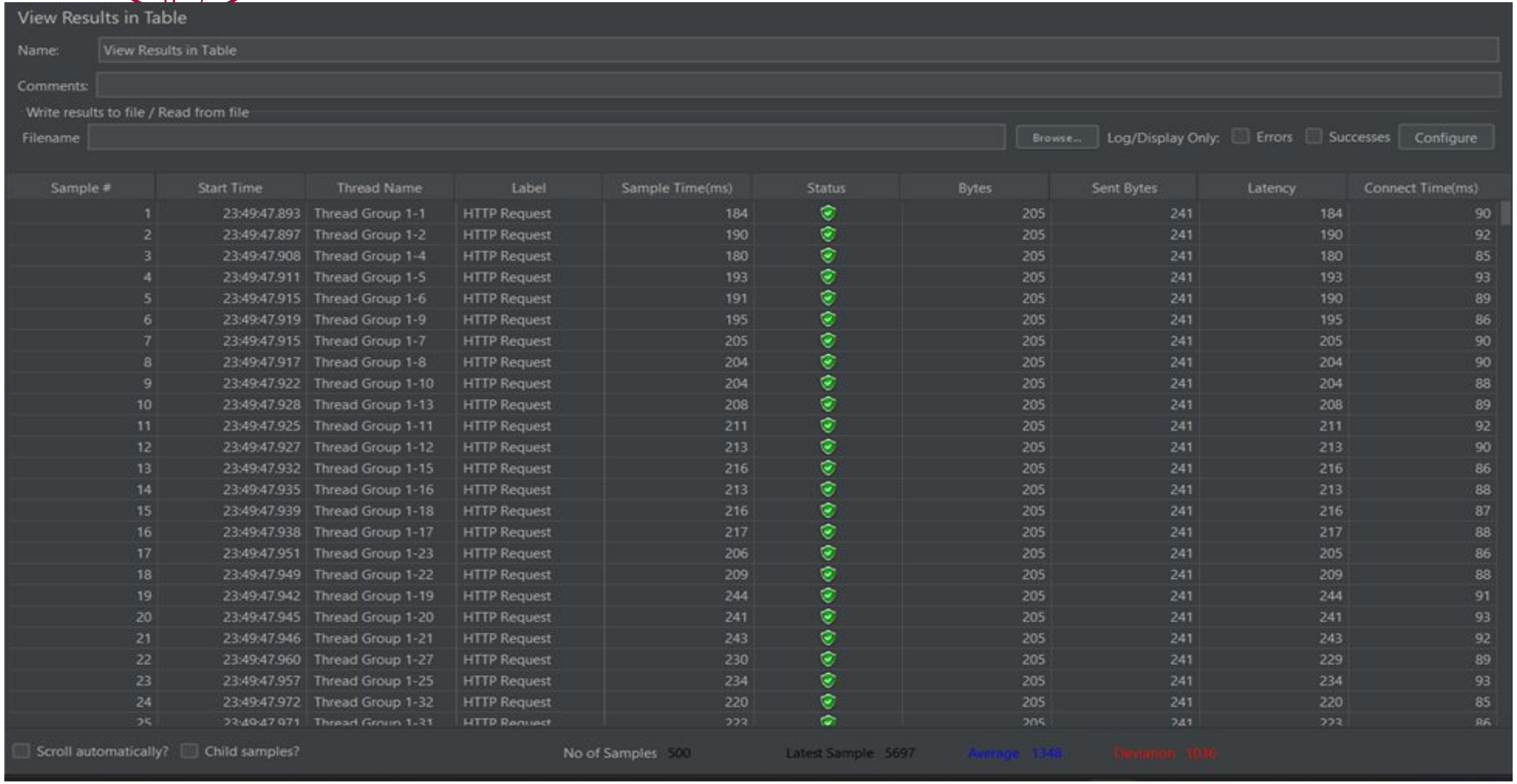
Write results to file / Read from file

Filename

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	500	1507	0	5510	1136.69	0.00%	84.0/sec	16.40	19.85	200.0
TOTAL	500	1507	0	5510	1136.69	0.00%	84.0/sec	16.40	19.85	200.0

Jmeter Performance Test for Subscribing users

No. of concurrent users = 500



View Results in Table

Name: View Results in Table

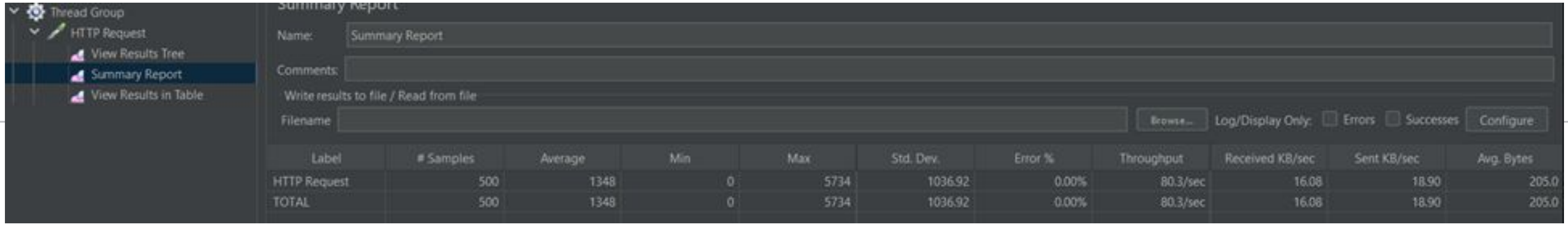
Comments:

Write results to file / Read from file

Filename Browse... Log/Display Only: □ Errors □ Successes Configure

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(ms)
1	23:49:47.893	Thread Group 1-1	HTTP Request	184	✓	205	241	184	90
2	23:49:47.897	Thread Group 1-2	HTTP Request	190	✓	205	241	190	92
3	23:49:47.908	Thread Group 1-4	HTTP Request	180	✓	205	241	180	85
4	23:49:47.911	Thread Group 1-5	HTTP Request	193	✓	205	241	193	93
5	23:49:47.915	Thread Group 1-6	HTTP Request	191	✓	205	241	190	89
6	23:49:47.919	Thread Group 1-9	HTTP Request	195	✓	205	241	195	86
7	23:49:47.915	Thread Group 1-7	HTTP Request	205	✓	205	241	205	90
8	23:49:47.917	Thread Group 1-8	HTTP Request	204	✓	205	241	204	90
9	23:49:47.922	Thread Group 1-10	HTTP Request	204	✓	205	241	204	88
10	23:49:47.928	Thread Group 1-13	HTTP Request	208	✓	205	241	208	89
11	23:49:47.925	Thread Group 1-11	HTTP Request	211	✓	205	241	211	92
12	23:49:47.927	Thread Group 1-12	HTTP Request	213	✓	205	241	213	90
13	23:49:47.932	Thread Group 1-15	HTTP Request	216	✓	205	241	216	86
14	23:49:47.935	Thread Group 1-16	HTTP Request	213	✓	205	241	213	88
15	23:49:47.939	Thread Group 1-18	HTTP Request	216	✓	205	241	216	87
16	23:49:47.938	Thread Group 1-17	HTTP Request	217	✓	205	241	217	88
17	23:49:47.951	Thread Group 1-23	HTTP Request	206	✓	205	241	205	86
18	23:49:47.949	Thread Group 1-22	HTTP Request	209	✓	205	241	209	88
19	23:49:47.942	Thread Group 1-19	HTTP Request	244	✓	205	241	244	91
20	23:49:47.945	Thread Group 1-20	HTTP Request	241	✓	205	241	241	93
21	23:49:47.946	Thread Group 1-21	HTTP Request	243	✓	205	241	243	92
22	23:49:47.960	Thread Group 1-27	HTTP Request	230	✓	205	241	229	89
23	23:49:47.957	Thread Group 1-25	HTTP Request	234	✓	205	241	234	93
24	23:49:47.972	Thread Group 1-32	HTTP Request	220	✓	205	241	220	85
25	23:49:47.971	Thread Group 1-31	HTTP Request	223	✓	205	241	223	86

□ Scroll automatically? □ Child samples? No of Samples 500 Latest Sample 5697 Average 1348 Deviation 1036



Summary Report

Name: Summary Report

Comments:

Write results to file / Read from file

Filename Browse... Log/Display Only: □ Errors □ Successes Configure

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	500	1348	0	5734	1036.92	0.00%	80.3/sec	16.08	18.90	205.0
TOTAL	500	1348	0	5734	1036.92	0.00%	80.3/sec	16.08	18.90	205.0

Future scope

- OAuth implementation for the publisher and subscriber frontend applications

- Since our system is scalable so using api available in broker.py, cluster of brokers can be prepared. More brokers can be added to handle failure of more than one broker. Primary broker can be selected via leader election algorithm.

- Implement a more sophisticated load balancing mechanism that can dynamically distribute traffic among brokers based on real-time load analysis. This could involve using algorithms like Round Robin, Least Connections, or even machine learning-based predictive load balancing.

Summary

This project represents a sophisticated, cloud-based messaging system that excels in fault tolerance, scalability, data integrity, consistency, security, and performance, making it a robust solution for modern communication needs.

- Fault Tolerance: Achieved through a dual broker system with primary-secondary roles, ensuring continuous operation.
- Scalability: Hosted on Amazon EC2 instances, allowing for easy scaling to handle increased load.
- Data Integrity and Consistency: Ensured through message replication and heartbeat mechanisms between brokers.
- Security: Maintained by allowing only registered subscribers to access messages.
- Concurrency & Performance: Optimized to handle multiple publishers and subscribers efficiently, ensuring fast and reliable message delivery.
- The Algorithms used in this project are: Replication and Consistency, Concurrency, Heartbeat and Broadcast Protocols.

Thank you