Distributed Queuing using RabbitMQ
IN4391 Distributed Computing Systems

Authors: Dominik Harz (4513649) & Jorik Oostenbrink (4169263), TU Delft
Course Instructor: Prof. Alexandru Iosup, PDS Group TU Delft
Teaching Assistant: Alexey Ilyushkin, PDS Group TU Delft
April 22, 2016

Abstract

Tabster is an app focusing on easing the payment process as well as splitting the tab in the hospitality business. Tabster uses the Command and Query Responsibility Segregation (CQRS) pattern to divide write and read access. In this paper an extension to the system is presented to transform the read store updater into a distributed system. Tabster is implemented with PHP framework Symfony2, the distributed message broker RabbitMQ, the database MySQL and a search server cluster based on Elasticsearch. The objective of this paper is to dynamically assign queues and consumers to update the read side as fast as possible. In experiments it is shown that the distributed approach is able to handle bursty workloads better and minimize the events stored in the queues compared to the old non-distributed approach.

1 Introduction

Tabster [1] is an app allowing people to pay and share the tab at a bar or café. It is enabled by recording the consumptions of a group of people sharing a tab. A user is able to join an existing tab or create a new one. Users are able to view a personal tab overview in which they see the events that have occurred since they joined the tab. This includes for example consumptions or other users joining and leaving their tab. Furthermore, users can leave a tab and pay their share accordingly. When the last user leaves the tab, the tab is closed, the full cost is calculated and the money is transferred to the bar or café. Any interaction with the tab is event-sourced and is stored in one large queue. Events are processed and stored in the queue based on their creation. However, when a large number of customers is active, the queue becomes rather full as some events take a comparably long time to process. This results in a large backlog of messages, which decreases the usability for users and hinders them from seeing the latest version of their personal tab overview. In order to eradicate the shortcomings of the current system a distributed approach is proposed.

The current architecture is implemented as follows: The data from the cash register and all requests from the customers are send to the Tabster backend. The backend system is designed with the Command and Query Responsibility Segregation (CQRS) pattern [2] with separate read and write stores. In this pattern operations that read data (queries) are separated from operations that write data (commands) by using separate interfaces (and data models). To increase performance, scalability, and security the "read data" and the "write data" can then be stored in separate stores/databases. Of course, the read store does have to be notified of any changes made to the write store. Tabster has an event driven architecture. When the write store is updated an event is created and put in a queue [3]. The read store takes events from this queue, processes them and updates itself accordingly. An overview of this type of architecture can be seen in figure 1. This design allows the write store to use a strong consistency model, while the read store utilizes eventual consistency.

As a starting point for this project a single system is used, which is extended to be distributed. The webserver of Tabster is written mostly in PHP with the support of the Symfony2 framework[4]. The webserver contains the logic to interact with the Tabster app, as well as update the write and the read datastore. MySQL [5] is used to implement the write side datastore. Tabster further uses RabbitMQ [6] for its queuing operations and Elasticsearch [7] to store its read data. Currently, it processes all events to the read store one by one. Under their current load, this causes the events to slowly build up in the queue. This buildup of events in the queue causes the customers to get an outdated view of their tab and makes the app feel unresponsive.

In this report the authors describe a distributed system for processing incoming events to Tabster's read store. In section 2 the design of the distributed system and some of its (theoretical) properties are described, section 3 gives implementation details of the system, section 4 presents the experiments executed to test the system, section 5 discusses the results and section 6 concludes the article.

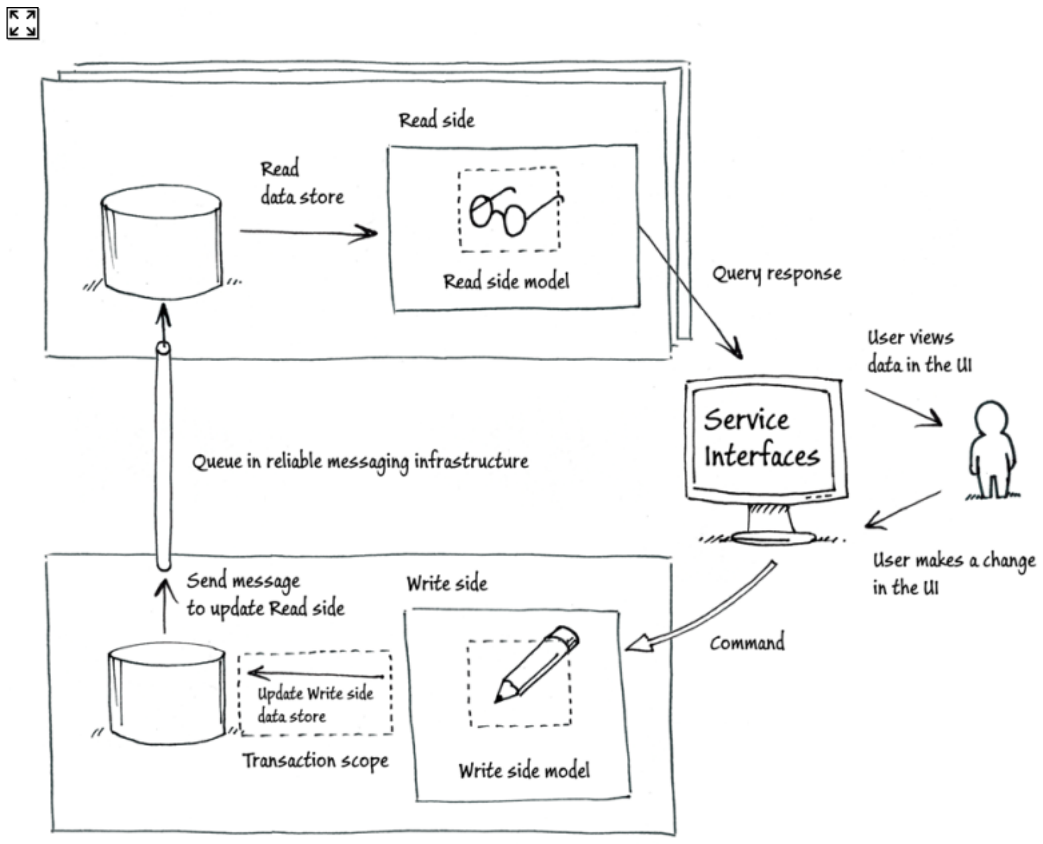


Figure 1: Overview of the CQRS pattern as used in Tabster. Taken from [8]

2 Design

In this section the authors describe the design and properties of the system. The first subsection gives the requirements of the system, the second subsection describes the design of the system and the final subsection discusses some of the properties of the designed system.

2.1 Requirements

The system is designed to fulfill requirements in the fields of system operation, fault tolerance and scalability. Furthermore, real life test data is used to conduct the experiments and the whole system is stress-tested. The detailed requirements are further explained below.

First of all, the system has to successfully update its read model in Elasticsearch based on the updates it receives. Furthermore, updates to the same tab have to be processed in the order they arrive. Tabster (the company) has provided real world data to be used to test the functionality of the system.

The system has to be able to scale horizontally by adding more nodes. At a minimum, the system should be able to handle 10 tabs, with at least 100 events per tab. It should also be able to deal with a scenario in which one tab has 5 times the number of events than another.

Finally, if a component related to the handling of a specific tab fails, this should have no effect on the performance and operation related to any other tab. In addition, the failure of one node should not crash the whole system. For debugging purposes, all system events (like the arrival or the completion of an event) should be logged.

2.2 Design

The system is build entirely around the concept of creating one queue per tab. Consumer Managers are developed and run on each node of the cluster to manage the creation of these queues. They look through all events and create a new queue and corresponding consumer when they encounter a “TabWasCreated” event.

To understand the system, one first needs to understand four RabbitMQ concepts, those of the producer, consumer, exchange and queue[6]. A producer is a process that creates messages (in Tabster’s case events) and

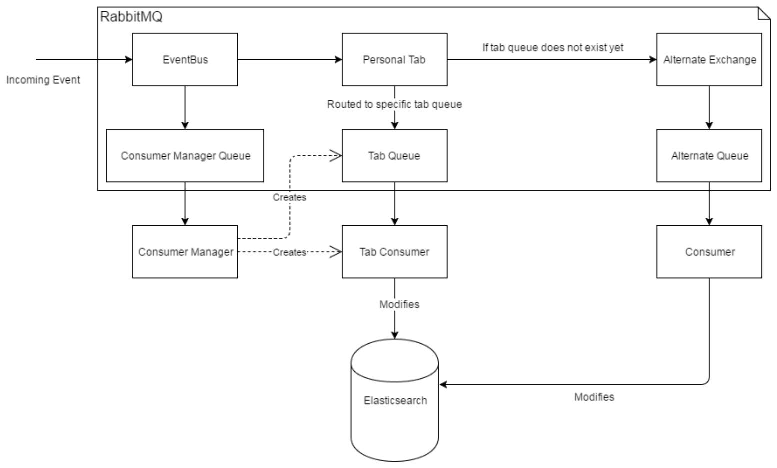


Figure 2: Overview of how the system processes incoming events

sends them to the RabbitMQ cluster to be added to a queue. The consumer is the process that takes those messages from the queue and processes them. Both of these processes run outside of RabbitMQ itself.

To route a message to the correct queue(s), RabbitMQ uses exchanges. An exchange is just an object which routes a message to a set of queues based the exchange's settings and the message's routing key. The queue itself works with the first in, first out principle (FIFO). Messages come in and get placed in the queue. If a consumer requests a new message to process it gets send the oldest message in the queue and the message gets removed from the queue. If the consumer dies without having send an acknowledgment (ACK) back to the queue, the message is put back in the queue (if the queue is setup to do this, which is the case for this project).

A RabbitMQ cluster [9] can contain multiple nodes, each containing multiple queues and exchanges. All communication with a specific exchange or queue has to be done with its "master", potentially via other nodes in the cluster. A queue can be replicated to other nodes (in the so called "High Availability" (HA) mode), but in this case all communication still has to be done with the master node. The master node informs its "slaves" about all operations made to the queue, so they can update themselves. If the master fails a slave can take over and become the new master.

A RabbitMQ node writes almost all its data (including its queues and exchanges) to its disk, so in the case of failure it can easily be restarted losing (almost) no data. Tabster employs a service which automatically tries to restart RabbitMQ nodes after they crash.

In the implemented system, an incoming event is processed as follows: First it arrives at an exchange called the "eventbus", the "eventbus" sends the event to the Consumer Manager Queue and to the "Personal Tab" exchange.

The Consumer managers mostly ignore (and ack) all events except the "TabWasCreated" event. If they receive this event they'll create a new Tab Consumer and corresponding queue.

The "Personal Tab" exchange routes the events to their tab queue. If this queue does not exist yet, it sends the message to the "Alternate Exchange" which immediately routes them to the "Alternate Queue". Multiple Consumers are connected to this queue to quickly process all incoming events for which no tab exists yet (or for which the tab queue has failed for some reason). The authors would have preferred to have the "Personal Tab" send the messages directly to the "Alternate Queue", but in RabbitMQ it's only possible to send unrouted messages to exchanges.

It should be noted that this system might cause messages to be processed in the wrong order. As a message which arrived earlier, and was routed to the alternate queue, might be processed after a later message which was routed to a tab queue and processed immediately. In the next section this will be discussed further and it will also be explained why this won't be a problem in practice. It should also be noted that in this system all "TabWasCreated" are processed by the Consumer Managers and the Alternate Queue Consumers and never by the Tab Consumers, as the tab specific queue has not yet been created when this event arrives.

The Tab Consumers process all incoming events, destroying themselves (and their queue) if they receive a "FinalCostCalculated" event (for every tab, this is the last event received).

An overview of the above is given in figure 2 and figure 3 shows a (nontraditional) sequence diagram where a tab is opened and subsequently closed. To reduce the amount of cluttering, the Alternate Queue has been omitted in this sequence diagram. Finally, figure 4 gives a small overview of the consumers, exchanges and queues in the system.

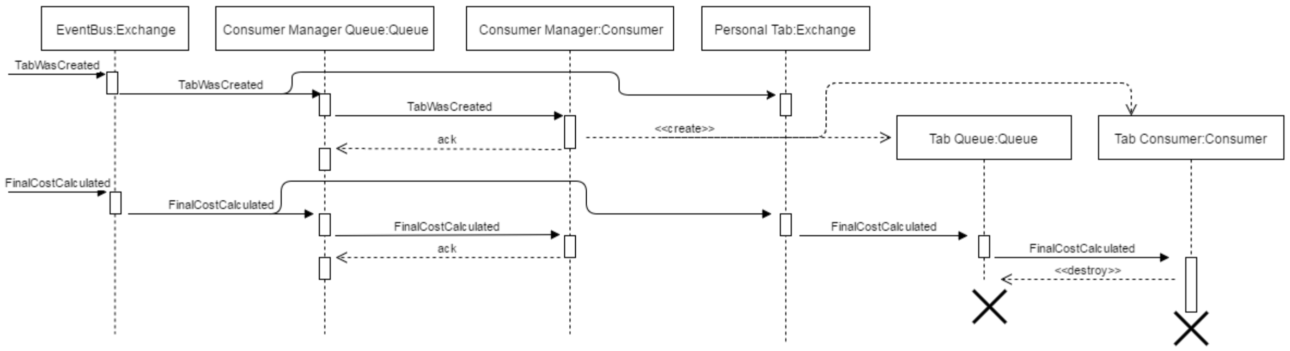


Figure 3: Sequence Diagram: creating and destroying a tab

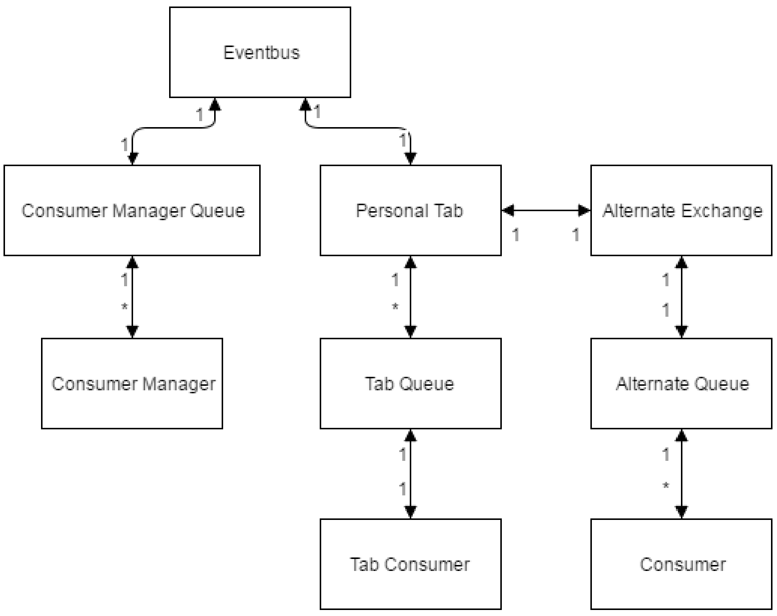


Figure 4: System Overview

2.3 Properties

In this section some of the (theoretical) properties of the system (corresponding to the requirements given earlier) are discussed.

First of all, if the system processes the events from the same tab in the order they arrived it should successfully update the read model in Elasticsearch. Simply because the workers (consumers) making the changes to the read model are still the same workers as Tabster used before. The only change here is that before they used only one worker before, while now there are multiple workers working in tandem.

However, theoretically it is possible that a “TabWasCreated” event might arrive at a very full Alternate Queue while also being placed in a more empty Consumer Manager Queue. In this case the Tab Queue gets created much earlier than the “TabWasCreated” event gets processed. If another event for the tab arrives before “TabWasCreated” gets processed by the Alternate Queue consumer, it might be possible that this new event gets processed (by the Tab Consumer) before the “TabWasCreated” has been processed. Thus causing errors in the system.

The authors believe this worst case scenario wont happen in practice, as it requires that a new event for a tab arrives before “TabWasCreated” has been completely processed. It can be assumed that after a tab has been created at the bar people won’t immediately add consumptions to them. At the very least it takes time to type them in. It can be concluded that there is always a few seconds delay between “TabWasCreated” and any following events. Because only a few events get routed to the alternate queue, its consumers can easily keep the queue practically empty at all times and consume events much quicker than this delay. This property will also be shown in the experiments.

The system can be scaled in 2 ways. By adding more RabbitMQ nodes and by adding more nodes for the consumers to run on. The former mostly increases the number of queues it can keep in memory, while the latter increases the amount of events that can be processed per second.

The system does have a few bottlenecks. A single exchange (or queue) only runs on one node (even in HA mode all work is done by the master node), so if enough events arrive the RabbitMQ node which runs for example the EventBus will slow down the whole system. For this to happen Tabster has to become a lot more popular than it currently is, and if enough requests get send to the server to generate this workload the other parts of Tabster (not the read model) would fail much earlier than the EventBus would slow things down.

The other bottleneck are the Tab Consumers. In our current design only one Tab Consumer is assigned to every Tab Queue. This might slow the system down if a lot of events get send to a single tab. However, realistically the amount of events that would be necessary to slow down the consumer too much would be much more than the amount of events generated by any tab (even those with a lot of users). If this would turn out a problem it would be very easy to add multiple Tab Consumers to each Tab Queue.

Finally, fault tolerance is mostly achieved by using RabbitMQ and by assigning each Tab Consumer to only one queue related to only one tab. RabbitMQ (and Symfony) handles the messaging between queues and consumers, dealing with link failure. If a Consumer Manager crashes this has no effect on the workings of the system. The other Consumer Managers will continue to do the work and any events it was currently processing will be added to the Consumer Manager Queue again. If a Tab Consumer crashes no events for this tab will be processed anymore until it’s restarted, but the events for all other tabs will still be processed. If a Alternate Queue Consumer crashes the other Alternate Queue Consumers can take over and any currently processed events will be send back to the queue. If a RabbitMQ node crashes this has a very large effect on the system (potentially no events could be processed anymore), but each node has saved all its data to its disk and will be restarted automatically as soon as possible. Almost no events will be lost. One should note that the designed system is more fault tolerant than specified in the requirements.

3 Implementation

The system is implemented within Amazon Web Services[1]. It runs on two t2.medium instances with two CPUs (Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz), 4 GB of RAM and 40 GB SSD storage are used. On each instance a Symfony [4] instance, a RabbitMQ node and a Elasticsearch node are running. On one of the instances the MySQL database is running, to which the other instance also connects. RabbitMQ and Elasticsearch are configured as a cluster with two nodes respectively. The configuration of the instances is automated with Ansible[2] and the deployment is executed with boto[3]. The paragraphs below elaborate the additions that are made to the existing Symfony code and change of configuration for RabbitMQ as well as Elasticsearch.

[1]https://aws.amazon.com/
[2]https://www.ansible.com/
[3]https://github.com/boto/boto

3.1 RabbitMQ

RabbitMQ elects a master when nodes are joined together as a cluster. The nodes are configured to automatically join as a cluster based on their Linux host names [6]. To enable these nodes to communicate the Erlang cookie needs to be the same for every node in the cluster[6]. This is achieved by setting the Erlang cookie to the same value in the Ansible script, such that the instance starts up directly with that Erlang cookie. Furthermore, the cluster is configured to use auto-healing when one of it nodes fail to communicate. When a node fails completely (i.e. shutdown of the instance) the other node tries to restore any lost queues and tries to take over the connections from producers and consumers. In case the network connection fails, there will be a partition of the network as the two nodes act independently and try to restore each others state. Once the nodes are connected again, they synchronize their status and decide on the state to use for each exchange and queue.

3.2 Symfony

The Symfony instance on each node is configured the same, except for the database connection. In the implementation only one node runs the MySQL database. To implement the RabbitMQ capabilities the libraries php-amqplib[4] and RabbitMqBundle[5] are used. php-amqplib implements the Advanced Message Queue Protocol for PHP and RabbitMqBundle defines wrappers to utilize php-amqplib within the Symfony framework. Within Symfony the producers, exchanges, queues and consumers are configured. Producers are sending events to defined exchanges. In this implementation the existing producers are extended to include the tab ID as a routing key, when sending the events to the “eventbus” exchange. Furthermore, two new exchanges are defined. Usually only queues are bound to an exchange, however RabbitMQ also offers exchange to exchange bindings in case further routing requirements are needed. This feature is not yet present in the RabbitMqBundle library and is hence implemented within this project. The ”personal tab” exchange is bound to the “eventbus” with the routing key “#”. This means that every event from the “eventbus” is forwarded to that exchange. Lastly, an alternate exchange called “alternate” is defined for the “personal tab” exchange. In case events can not be routed to a queue by “personal tab” (i.e. no queues exist yet or none have a matching routing key), the events are forwarded to “alternate”.

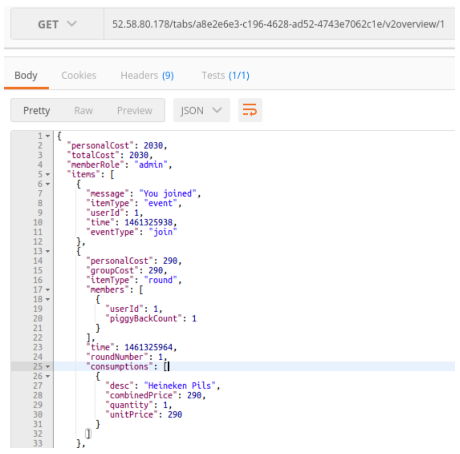
Consumer managers are consumers running on each node bound to one consumer manager queue at the “eventbus”. They are implemented in Symfony and are listening for “TabWasOpenedEvent” on its queue. On “TabWasOpenedEvent” they create a new anonymous consumer based on the tab ID of that event. Consumer managers share a common storage to keep track of running anonymous consumers. On any incoming event they are checking if the anonymous consumers are still running and restart them if required with a binding to the specific queue. When they receive “FinalCostWasCalculatedEvent” the worker and its queue are removed from the list of active workers as they is supposed to be stopped.

An anonymous consumer is bound to the ”personal tab” exchange for each tab and creates a new queue bound to that exchange, which takes only events based on the tab ID as routing key. By default anonymous consumers are used to create queues on the fly in RabbitMQ. However, once the consumer fails or is removed on purpose the queue is also deleted. In order to increase the fault tolerance of the system, this default behavior is adjusted in the rabbitmq library to keep the queue alive in case the anonymous consumer fails. Each anonymous consumer handles a specific tab and the corresponding events to update the personal tab overview in Elasticsearch. On “FinalCostWasCalculatedEvent” the tab is closed and the tab is fully paid. Thus, no more events are to be applied to this tab. Therefore, the anonymous consumer deletes its queue and stops itself.

On the ”alternate” exchange two consumers (one on each instance) similar to the anonymous consumers are working. They handle events like the anonymous consumers, however the have a wildcard (i.e. “#”) routing key to take any event. In case events for a specific tab arrive before the anonymous consumer is started or the queue is created, they can not be routed in the “personal tab” exchange and are forwarded to ”alternate”. These consumers do not stop themselves on the “FinalCostWasCaclculated” event.

All queues in the system are configured to be durable to persist events in the queues to disk. In the case of the anonymous consumers, this requires a change in the RabbitMqBundle library. In case a RabbitMQ node fails, it can pickup its previous state from disk, once it is restarted. The consumer managers and alternate consumers are started and managed within supervisor. The supervisor configuration is setup to restart the service in case they fail.

[4]https://github.com/php-amqplib/php-amqplib
[5]https://github.com/php-amqplib/RabbitMqBundle



52.58.80.178/tabs/a8e2e6e3-c196-4628-ad52-4743e7062c1e/v2overview/1

```
{
  "personalCost": 2030,
  "totalCost": 2030,
  "memberRole": "admin",
  "items": [
    {
      "message": "You joined",
      "itemType": "event",
      "userId": 1,
      "time": 1461325938,
      "eventType": "join"
    },
    {
      "personalCost": 290,
      "groupCost": 290,
      "itemType": "round",
      "members": [
        {
          "userId": 1,
          "piggyBackCount": 1
        }
      ],
      "time": 1461325964,
      "roundNumber": 1,
      "consumptions": [
        {
          "desc": "Heineken Pils",
          "combinedPrice": 290,
          "quantity": 1,
          "unitPrice": 290
        }
      ]
    },
```

Figure 5: Excerpt of a personal tab overview for a test users with some consumptions

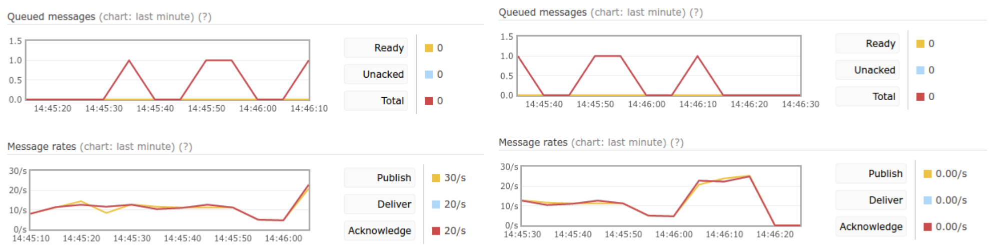


Figure 6: Anonymous consumers from the RabbitMQ interface with their respective message rates and queue length.

3.3 Elasticsearch

Elasticsearch is configured as a cluster with two nodes. Elasticsearch elects a master from the two nodes [10]. The master node manages the connections, while the views can be stored on either on the two nodes. Furthermore, the master handles shards that occur when nodes are joining or leaving as well as during failures. For this project Elasticsearch is configured to automatically detect other nodes on the network and automatically join a cluster when available [10].

4 Experiments

In this section the experiments to test the requirements outlined in section 2 and their results are discussed. The tests are executed with data provided by Tabster. To test the system Postman [11] and Newman[12] are used. These are testing suits to test APIs (Postman) and to automate the testing of APIs based on collections with Javascript (Newman). These collections include adding users to the system, creating tabs, adding consumptions, adding and removing users from a tab and closing the tab.

4.1 System operation requirements

Tab specific queues and workers are created based on specific events ("TabWasCreated" and "FinalCostWas-Calculated") in the general queue. Consumer Managers watches for these events and creates the tab specific queues and workers accordingly. When an event arrives, it is put in the local tab queue or sent to the alternate exchange queue (for errors/unrouted events). Queues are monitored by workers, which execute actions based on the events in the queue. Each queue and worker have a unique identifier. Each event has a unique identifier and a variable duration (its runtime on a single node). To test that the system is working properly all the above mentioned collections are run from Postman afterwards the personal tab overview is requested. Figure 5 shows an excerpt of the result for a user with various consumptions that add up to 20.30 EUR in his tab. As the consumptions are added to the tab correctly and all tests return positive we can assume the system functions properly. The messages arrive at the dynamically created consumer and are processed there. Figure 6 shows two figures of different anonymous consumer queue's from the RabbitMQ management interface. On the top the current length of the queue is displayed, which varies between 0 and 1 and below the messages that are processed are indicated.

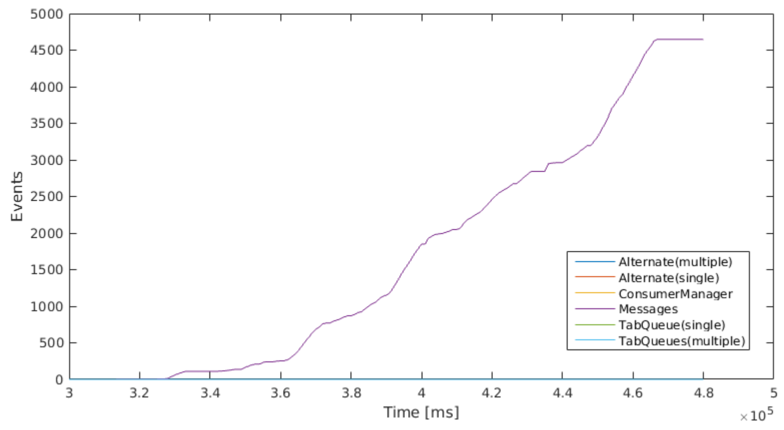


Figure 7: Number of events over time with number of events in specific queues for a single consumer and the multiple (distributed) system

4.2 Fault tolerance

The system is fault tolerant against failure of queues and workers. This means that a failure of one queue or worker does not affect other system components. To test this queues and their workers are killed from the command line of the instance. This does not affect other tabs or their read model in the system. However, if a worker is currently processing an event this leads in some cases to a broken read model for that specific tab. Hence, the read model needs to be fully or partly restored from the write model. This is left as future work to improve the system further.

In addition, all system events (e.g., event arrivals, event starts and completions, queue and worker node restarts) are logged in the order they occur. This is achieved by writing the events with the ID of the worker, the event ID and a timestamp into a syslog file.

4.3 Scalability requirements

The properties of the Tabster system are demonstrated when it contains 10 tabs with at least 100 events per tab, 2 consumer manager, and when the workload consists of at least 1000 events in total. Figure 7 shows the case for 20 tabs, with 3 to 6 users per tab, 110 consumptions per tab and users joining and leaving the tabs. This accumulates to above 4500 events within around 20 seconds. As shown in the graph, the queue length is always around 0 or 1, which means that the consumers are able to process and update the tabs fast enough that there occurs nearly no backlog. However, from experimenting it was found that this also occurs if just a single consumer handles all tab updates. To enable the study of realistic imbalance situations, the ratio of the numbers of events arriving at the most and least loaded tabs is set to be five. In this case the backlog of the queue is also zero for single and multiple consumers.

4.4 Benchmarking

As the backlog of queues is comparably small for 20 tabs, further experiments to stress test the system are executed. The system is stress-tested by creating a customized script which creates a desired number of tabs and runs tests on them concurrently. This is run with 100 tabs and workloads designed to stress particular elements of the system. The results are presented in figure 8. The total number of messages with 100 tabs, 3 to 6 users per tab, on average 110 consumptions per tab and a total of around 20,000 events are produced within around 100 seconds. In this test case the distributed (multiple) consumers empty the queue relatively fast (having 0 to 1 events left in total in the queues). However, in single consumer case, there are up to 900 events left in the queue when within a short time a large number of events are produced. Thus showing the advantage of the distributed system.

5 Discussion

The system requirements as defined in section 2 are fulfilled. However, they are equally fulfilled with just a single consumer and a less complex single system. This means for small workloads, the additional complexity of the system is not beneficial. With an increase of the workload, the distributed system is able to continuously

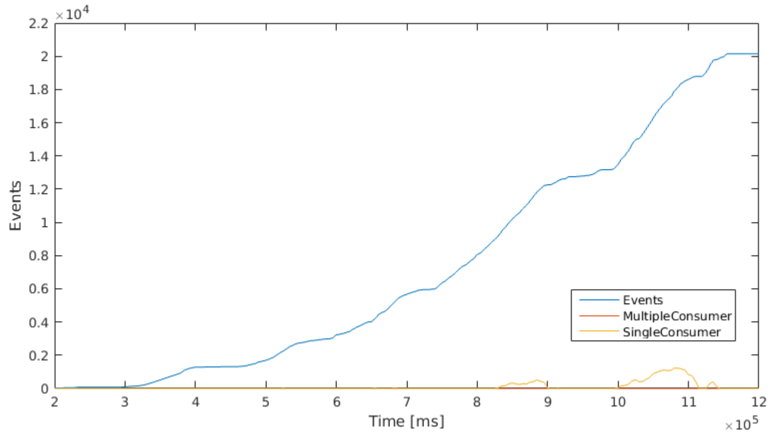


Figure 8: Number of events over time with number of events in specific queues for a single consumer and the multiple (distributed) system

handle the workload without leaving an increased amount of messages in the queue. Hence, the distributed system delivers predictable high performance.

In terms of fault tolerance the distributed system allows for tab individual consumers to fail and to restore them. Also, the Consumer Manager and Alternate consumer are allowed to fail, as long as one survives. In case that they completely fail the events would however still be stored in the message queues, which enables the system to continue working after they Consumer Manager and Alternate consumer are up and running again. Hence, this represents a persistent asynchronous communication model between the producers and consumers. Currently, faulty processed events log an error message, but are still acknowledged as long as the consumer does not crash during execution. The theory behind it, is that if a faulty event gets back in the queue, it will continue to fail and thus congest the queue and the consumer. However, in case the processing of the event relies on a depended service it might be processed correctly at a retry. Hence, the system can be extended to include a maximum number of retries per event, returning the event to the queue as long as the threshold is not achieved.

Furthermore, a severe crash of the system can leave the read model unusable, which requires a manual reset of the read model. This issue is currently only addressed by exception handling in the consumers and producers. By creating checkpoints in the system this can be tackled. By doing this states of the read model can be stored and restored in case the read model fails. This needs to be implemented as future work.

The system uses two types of consistency. The MySQL database and the write side utilize strong consistency, while the read side with Elasticsearch uses eventual consistency [13]. This causes issues in case a user wants to close and pay his share of a tab, since he needs a current version of his spending (i.e. his personal tab overview must be recent). Although the CQRS model ensures that reads are executed comparably fast, the latency between the read and write model updates need to be taken into consideration. The distributed system enables a fast update by splitting up the work among different queues and consumers. However, especially when searching for errors in the system eventual consistency is hard to follow.

The system is tested with extended workloads and is able to handle them. The main issue in extending an existent system and conducting load tests are the dependencies on other components and services. In case of opening a tab, payment methods including the payment services need to be in place. Furthermore, other consumers are also involved to close the tab, which provide overview of a receipt and the receipts in a venue. These are as of now not distributed and thus limit the load tests. When increasing the number of tabs for the benchmark to 200, 500 or more the system becomes less stable. There are numerous time-outs when executing the creation of users at the payment methods and also at the closing of tabs, when receiving the final receipt. These are caused by the above mentioned dependencies. To further stress test the system, these services need to be improved in terms of performance. With the current system configuration, the distributed queues and consumers are potentially still quite far form their limit.

6 Conclusion

To facilitate a larger amount of customers the current read model update system of Tabster is redesigned to be distributed. In theory, the distributed system fulfills all of its requirements in the fields of system operation, scalability and fault tolerance. This is supported by the results of multiple experiments on the system.

The system can easily handle 20 tabs receiving around 4500 events in total within 20 seconds, which is more than initially required. Even a more extreme case of 100 tabs and a total of around 20,000 events generated within 100 seconds is handled without problems for the distributed system. The non-distributed system performs much worse in this last case, with its queue slowly filling up with events.

In this report it is shown that a distributed system can significantly improve the performance of Tabster, especially when dealing with a large number of open tabs.

References

[1] Tabster. `https://www.tabster.nl`.

[2] Command and query responsibility segregation (cqrs) pattern. `https://msdn.microsoft.com/en-us/library/dn568103.aspx`.

[3] J. Thönes. Microservices. *IEEE Software*, 32(1):116–116, Jan 2015.

[4] Wojciech Bancer. *Symfony2 Essentials.* Packt Publishing Ltd, 2015.

[5] Mysql. `https://www.mysql.com/`.

[6] Rabbitmq. `https://www.rabbitmq.com`.

[7] Elasticsearch. `https://www.elastic.co/products/elasticsearch`.

[8] D Betts, J Dominguez, G Melnik, F Simonazzi, and M Subramanian. Exploring cqrs and event sourcing. a journey into high scalability, availability, and maintainabilitywith windows azure, 2012.

[9] Alvaro Videla and Jason J W Williams. *RabbitMQ in action: distributed messaging for everyone . Rabbit MQ in action.* Manning, Shelter Island NY, 2012.

[10] Elasticsearch clustering. `https://www.elastic.co/guide/en/elasticsearch/guide/current/distributed-cluster.html`.

[11] Postman. `https://www.getpostman.com/`.

[12] Newman. `https://www.npmjs.com/package/newman`.

[13] O. Parisot, A. Schlechter, P. Bauler, and F. Feltz. Flexible integration of eventually consistent distributed storage with strongly consistent databases. In *Network Cloud Computing and Applications (NCCA), 2012 Second Symposium on*, pages 65–72, Dec 2012.

A Timesheets

Table 1: Time spent on project

Parts	Jorik	Dominik
total-time	76 hours	90 hours
think-time	40 hours	10 hours
dev-time	15 hours	40 hours
xp-time	4 hours	2 hours
analysis-time	1 hours	10 hours
write-time	16 hours	8 hours
wasted-time	0 hours	10 hours