SESDAD

Carlos Faria
Instituto Superior Técnico
MEIC-A
carlosfaria@tecnico.ulisboa.pt

Nuno Vasconcelos
Instituto Superior Técnico
MEIC-A
nunovasconcelos@tecnico.ulisboa.pt

Sérgio Mendes
Instituto Superior Técnico
MEIC-A
sergiosmendes@tecnico.ulisboa.pt

Abstract

In this report we present an implementation of a reliable, distributed message broker supporting the Publish-Subscribe paradigm that assures that events that are subscribed by a given subscriber, are successfully delivered to him - SESDAD (Sistema de Edição Subscrição, Desenvolvimento de Aplicações Distribuídas). Our system implements, two types of event routing policies: Flooding where the events are broadcasted through the whole tree, and Filtering where events are only forwarded only along paths leading to interested subscribers; three types of ordering policies: No Ordering where the events can be received at the subscribers at any order , FIFO Ordering where all events published by a publisher p are received at a subscriber by the same order they were published by publisher p and Total Ordering that guarantees that subscribers deliver matching events in the same order. Our system provides fault-tolerance where we use a passive based replication technique to guarantee that the system keeps working even when a broker fails or a broker is slow.

1. Introduction

Indirect communication paradigms (group communication, publish-subscribe, message queues, shared memory approaches) aim to implement a communication through intermediaries nodes therefore not having direct communication between the sender and the receiver. In our system we implement a Publish-Subscribe system which is a system where publishers publish events to an event service and subscribers express interest in particular events through subscriptions [1]. In our implementation the "event service" mentioned before is a broker. These are responsible for routing the events received from the publisher to the subscribers that subscribe to the given topic. Subscribers can subscribe to a topic by sending a subscription to a broker in order to tell the broker that he wants to receive events on the topic he subscribed, they can also send unsubscribes in order to stop receiving events from that topic. In a Publish-Subscribe system there a few issues that have to be taken in consideration: Routing of events; Ordering of events; Failures. The routing policy used for events is important to assure a system efficiency and scalability. A Publish-Subscribe system demands guarantees in terms of the ordering of messages delivered to multiple subscribers. Since ordering is not guaranteed by the communication protocols used to exchange events, we must implement these ordering policies. In all type of systems, failures can occur, a node can crash or become too slow and the system must keep working as if any of those two never happened. For this a fault-tolerant implementation is required. The next section will describe how we implement the routing of events. In section 3 we will describe the ordering policies we used. In section 4 we will show how we managed to solve failure issues. Section 5 presents a quantitative and qualitative evaluation of our system. Section 6 concludes. The figures mentioned in the following sections are located in the Appendix. The Appendix also contains the Brokers states (used for fault tolerance), the script files we used in the evaluation of our project and the results of the evaluation.

2. Routing of events

The routing of events in our system can be done in two ways: through a flooding approach, and by using a filtering. We will know describe how each one was implemented. We will start by looking at flooding.

2.1 Flooding implementation

When a publisher publishes an event, it sends the event to its broker, the broker then will route the event to its parent (with the exception of the root node that doesn't have a parent) and to his sons. When a broker receives the event coming from a broker it resends to its broker Parent and his sons, however the broker doesn't send the event to the broker that sent him the event. This approach allows that if a broker is added to the system, it starts receiving events without doing any type of action so it provides a good scalability. In terms of efficiency, the broker only sends one message to each broker that are connected to it, now we at flooding counterpart, filtering.

2.2 Filtering implementation

To implement the filtering routing policy, we first needed to create a routing table for each broker. This is required so that the broker knows to which broker he needs to send the event in order to reach the subscriber that subscribed for that event. This assures that the events only go through the necessary brokers in order to reach the desired destination. Next we describe how we create the routing table for each broker. The routing table (tabelaEncaminhamento in the project) is a C# Dictionary where the key is the id of the broker, e.g. Broker1, and the value is the URL of the broker. Brokers have a list of their sons and Parent, for example, Broker2 (figure1) knows that Broker1 is his Parent node and that Broker4 and Broker5 are his sons. Let's first consider the case that Broker1 is creating his routing table. He starts by adding his sons directly in the routing table. So at this moment he has the following routing table:

[Broker2][Broker2URL]

[Broker3][Broker3URL]

He still needs to add Broker4, Broker5, Broker6 and Broker7 to his routing table, for that we call a recursive function that will iterate through Broker2 and Broker3 sons then call the function again on their sons and keep doing this until it reaches a broker that doesn't have sons (in this case Broker7, Broker5 and Broker6). When Broker1 calls this recursive function on Broker2 it will add the following entries to Broker1 routing table:

[Broker4][Broker2URL]

[Broker5][Broker2URL]

Broker2's URL is used instead of Broker4's URL for example, because for Broker1 to reach Broker4, he first needs to send to Broker2 because he can't send messages directly to Broker4. Then Broker4 will the run recursive function and add Broker7 entry to the Broker1's routing table

[Broker7][Broker2URL] and then finally Broker7 will stop the recursive function because he doesn't have any sons.

Now let's consider the case that Broker2 is creating his routing table. He first applies the same procedure we applied for Broker1 that is, we iterate through his sons and then call a recursive functions that will be run by the sons of his sons. We still need to the Broker1, Broker3 and Broker6 to this routing table. This is where the routing process becomes efficient. Broker2 will simply add Broker1 in the routing table, and then he will ask for Broker1's routing table and all the entries from Broker's 1 table that are not already added in the Broker2's table, in this case Broker3 and Broker6, are added directly to Broker2's routing table without needing to call any recursive function like we did for the sons. Broker2's routing table will then have the following entries added to it:

[Broker1][Broker1URL]

[Broker3][Broker1URL]

[Broker6][Broker1URL]

Our approach becomes more efficient has we go down in the tree, for example to create Broker7's routing table, we simply we get routing table already created by Broker4.

The Brokers now know to which sons or parent they need to send a message in order to reach a Broker that is outside his boundaries, for example Broker2 knows that to send a message to Broker7 he needs to send to Broker4 and then Broker4 will deliver the message to Broker7.

Now we still need to perform an additional step. When Brokers receive an event published by a Publisher, they need to know which subscribers are interested in these events and where are they at. To accomplish this, when Brokers receive a subscription, they will flood the subscription to entire tree using the same flooding process that is described in 2.1. These subscriptions are stored in a C# List (tabelaSubscricoes) where it stores information about the subscription, like the ID of the subscriber, the name of the topic and the URL of the subscriber. By having the URL of the subscriber we know to which broker this subscriber connects.

For a broker to send an event from a publisher he first checks tabelaSubscricoes to see if there's any subscriber interested in this event, if there is, he then checks the URL of the broker he needs to send the subscription to and then checks the routing table to see to whom he needs to forward the event in order to reach its final destination. Doing an efficiency and scalability analysis, we can conclude that if a broker is added to the system, he would be able to create his routing table very quickly since he simply had to ask his parent for his routing table and add it has his own has seen in the Broker7's example we provided above, we shall confirm this in 5.

Our filtering approach provides an efficient routing of events because has we have seen previously, the brokers only route the events through the necessary brokers in or to reach its final destination. Next we will introduce our ordering policies that "work" together with these routing policies we just described.

3 Ordering Policies

Our Publish-Subscribe system supports three types of ordering policies: No ordering, FIFO ordering and Total ordering. In the following sub-sections we describe we implemented these three policies.

3.1 No ordering

The no ordering policy, as the name says, it doesn't guarantee any type of ordering, that is, the events can be delivered in any order.

3.2 FIFO ordering

FIFO ordering guarantees that a message from a publisher P with a sequence number n+1 can only be delivered if a message from the same publisher with sequence number n has already been delivered. We implemented FIFO ordering differently to both routing policies, flooding and filtering, for efficiency reasons.

3.2.1 Flooding

The events are only ordered in the brokers with subscribers. There isn't any kind of ordering between brokers, all brokers will receive all events without any order whatsoever, then they will organize them by publisher, i.e. the brokers start by expecting a message n=1 from a publisher, if the message has n>1, the event will be placed in a queue, if we receive the message n=1, we check if the subscribers are interested in the topic of the event so we can deliver, and then start expecting the message n=2, and check the queue if event n=2 was already waiting. If the subscriber is not interested, the event is discarded and we start expecting the message n=2 anyway, and also check the queue if event n=2 was already waiting.

3.2.2 Filtering

In flooding we could organize them only in the brokers with subscribers because we know that we are going to receive all events from all publishers. In the filtering routing policy there is not guarantee that we're going to receive all events from a publisher. For example, a publisher P is publishing events with topics A and B, and a subscriber is only interested in topic A. The broker that has the subscriber won't ever receive the events with topic B, so he will get blocked forever. The publisher might have published the events in the following order: A1 > A2 > B3 > A4

After the broker receiving A2, he will be awaiting some message from that publisher with the number 3, but it will never get there since this is the filtering routing policy.

To overcome this problem the events are ordered between brokers, all connections between brokers are FIFO channels. Also we added a private sequence number to the events, so it can be rearranged between brokers. There will be a reorder in every broker if needed. For example we have the same publisher from above connected to a broker that has two sons that each of them have a subscriber interested in one of these topics. Let's call this broker Broker1, the left son Broker2 and the right son Broker3.

Broker2 will be interested only in A, and Broker3 will be interested only in B. Broker1 will receive the events from the publisher in the following order:

A1 (1) > A2 (2) > B3 (3) > A4 (4), being the number in parenthesis the private sequence number. He will rearrange the events and:

Send to Broker2: A1 (1) > A2 (2) > A4 (3),

Send to Broker3: B3 (1)

Now the sons will be awaiting event with the private sequence number 1, and if any of the others arrive first, they'll be placed in a queue. When the private sequence number 1 arrives, we start expecting the number 2, and check the queue if number 2 was already waiting.

3.2.3 Efficiency comparison

The implementations were different as said before for efficiency reasons. The FIFO in flooding only needs to order the events in the broker with the subscriber. In the filtering one, in every broker they reorder the events (even if the order will stay the same, the broker goes through the same process or reordering), so naturally the events will take longer to get to their destination.

3.3 Total ordering

Total ordering needs to guarantee that subscriber that are interested in the same topics receive the events in the same topics. To do this we also implemented in 2 different ways for the same reason as in FIFO, in filtering we don't receive all events, so there will be "gaps" in the sequence numbers.

The main idea in total ordering was that there has to be a global sequencer, i.e. every time an event is published, the broker needs to ask the global sequencer a number so he can attach that number to the event. This way all events will have a unique sequence number, that is used to find out which event was publish first between brokers.

The global sequencer will always be the root node, because of being "fair" the distance of the brokers to the global sequencer. If the root node was chosen randomly, he could be on the left part of the tree, and all brokers on the left part of the tree were always closer to the global sequencer, hence they would get their global sequence numbers much faster.

Our implementation of total order not only guarantees that the subscribers receive in the same order, but also guarantee the order that the publisher published them.

3.3.1 Flooding

As said before, when an event is published, the broker asks the global sequencer for a number. This request is not broadcasted through the tree, it only travels to the root node and comes back down to the requester through the routing table. These requests are numbered so that we also guarantee the order that the publisher published. When the root node receives a request with the number 1, he knows that the broker that requested has a publisher and will be asking for global sequence numbers from now on, so he adds the broker to a list so that he can wait for the request is order. Again the same system is used with the queues, if a request with number 5 arrives and we are still needing the request number 3, it is added to a queue. Every time we receive the request that we were indeed waiting for, we check the queue and so on.

When the root node receives a request, it will attach the global sequence number to the request, and send it back, incrementing the number locally afterwards.

As soon as the broker sends the request, he won't be waiting all the time for the request to go up and down to the root node, so he broker add it to a list with the events associated with the request number. As soon as the request gets back with the global sequence number, we search the list with what even that request is associated with and attach the global sequence number that came with the request to the event, and now the event can be broadcasted along the tree.

Now it is ordered like in FIFO was in section 3.2.1 above. But in total order, they are ordered by the global sequence number, i.e. in the brokers with the subscribers there is a reorder to deliver to the subscriber. They start expecting the event with the global sequence number 1, if events with a number greater than 1 arrive, they are added to the queue, and once the number 1 gets there, the broker checks the queue for the number 2 and so on.

3.3.2 Filtering

The same system of requesting a global sequence number to the root node is used when applying the total order policy to filtering. But here we will have the same problem of the "gaps" between global sequence numbers, since in the filtering routing policy the brokers only get the events that its subscribers are interested in.

To overcome this problem, when the broker that makes the request will attach additionally a list of all subscribers interested in that topic. A request number is also attached to the request exactly like in flooding total order for the same reason.

The root node will receive the request ordered by the request number, if the request number is not the one that we are expecting we add it to the queue, if it is we accept it, treat it and then check the queue to see if the next request is there.

Now the root node will treat the request in a different way. The root node keeps a list of all subscribers that exist in the tree. Every time a request is received, the root node will increment its local list the subscribers that came in the request, and then after incrementing, will attach the result of each corresponding subscriber to the list on the request so that it can be sent down again to the requester. Every subscriber's order is decided in the root node.

When the broker receives the request back, like in flooding, he will get the corresponding event and attach each different sequence number that came on the list. For example if the list came back saying:

Susbcriber1	5
Subscriber2	7

We will now make 2 copies of the event, and attach the sequence number 5 with the destination to subscriber1, and attach the sequence number 7 with the destination to subscriber2.

The brokers that are the final destination won't have to worry about the gaps anymore since every subscriber has its own private global sequence number.

3.3.3 Efficiency comparison

Again the flooding will be faster because of the treatment of the events that has to be done differently. In flooding the events are only ordered on the destination brokers when in filtering there is a lot more process to be done to guarantee total order, for example the broker that makes the request needs to be making copies of the events and attaching on them each respective global sequence number.

Next we will introduce our solutions to treat a failure or a slow broker in our system.

4 Fault Tolerance

In the non-fault tolerant version of our system, the system only has 1 broker per site has seen in Figure1. In our fault tolerant version, each site has 3 brokers. To ensure that our system is fault tolerant, we decided to use a solution based on passive replication. We thought about implementing an active replication where the messages received on a site, would be balanced between the 3 brokers of the site but due to time restrictions we didn't think we could implement this in time so we decided for an approach that doesn't provide load balancing but provides fault tolerance. Besides keeping the system working in the occurrence of a crash we also tried to minimize the false crashes detection.

4.1 Broker states

Brokers can have the states depicted in Figure 10. The broker that receives events and forwards events is the broker that has the state ATIVO. The other two brokers have the state ESPERA. The first broker of a site to be created is the one that receives the ATIVO state, the other two brokers receive the state ESPERA. When a broker is in ESPERA state he is in a waiting state meaning his ready to become the active broker in case the ATIVO broker fails. Let's consider site1 which has Broker1, Broker11 and Broker111. Since Broker1 is the first to be created, he will be in the ATIVO state while the other two are in ESPERA state. When a broker is the state SUSPEITO, it means that he is suspected to have failed and when a broker is in MORTO state it means that it has crashed.

4.2 Replication

The ATIVO Broker, in our example, Broker1, will replicate the routing table information and subscription updates to the replicas, Broker11 and Broker111, so in the case Broker1 fails, Broker11 or Broker111 can keep doing Broker1 job. When a broker receives an update of a state of a broker, it will also send these updates to the replicas.

4.3 Broker Suspicion

When a broker receives a message, he must send an ACK back to the entity (Broker, Subscriber or Publisher) that sent him the message in order to assure that the message was successfully delivered. If the entity that sent him the ACK did not receive the ACK in a certain period of time, this entity will warn an ESPERA state broker that the current ATIVO is not responding. The ESPERA state broker that the entity warns is the first one to be in ESPERA state, in our example, Broker11. Broker11 sets himself as ATIVO and sets Broker1 in state SUSPEITO. Broker11 will now warn the publishers and subscribers on his site that the new ATIVO is him and that all messages should be sent to him. He will also warn his sons and parent that he is the new ATIVO and that Broker1 is in state SUSPEITO and finally, he also warns Broker111 that Broker1 is SUSPEITO and that Broker11 is ATIVO.

The new ATIVO broker, in this case Broker11, must now check if Broker1 has really failed or if he was just slow. He will send an ACK to Broker1 with a longer waiting period. If Broker1 doesn't respond in that time period Broker11 will put Broker1 in state MORTO meaning that he crashed. Broker11 will then do the same procedure he done in when Broker1 was in state SUSPEITO and warn everyone but in this case, to set Broker1 has MORTO. If Broker1 responds and is indeed still alive, he will be set to state ESPERA and Broker11 will warn everyone else and they will swap Broker1 state from SUSPEITO to ESPERA.

4.4 False Crash detected

There can be a case where a Broker is really slow and can't respond within the second ACK time period. In this case when a Broker that has been declared crashed is actually alive, he will send a message to the ATIVO Broker on his site to warn that he didn't crashed and his state will be set to ESPERA and everyone will be warned to change that Broker state from MORTO to ESPERA. Let's consider again the previous example, Broker11 is the current active broker, and Broker1 has been declared MORTO. When Broker11 warns Broker111 that Broker1 is crashed, Broker11 also sends that same message to Broker1, this might seem pointless but it's actually a third measure (besides the two ACKs) to assure that a false crash doesn't happen. If Broker1 has really crashed then 1 message was sent for nothing but if he was really really slow, we will then eventually recover and receive the message that he was declared MORTO and by seeing this message he will then do the previous procedure.

The message sent from Broker11 to Broker1 stating that he was declared MORTO also serves a different purpose. Lets imagine Broker1 was slow and is declared SUSPEITO or MORTO and then he recovers, the messages he was going to deliver must not be delivered because they were already delivered by Broker11. If Broker1 receives a message stating he is SUSPEITO or MORTO he will discard all the messages he had to deliver.

4.5 Limitations

We implemented fault tolerance for total ordering but we did not had time to test it so it probably has a lot of bugs. As for the rest of the orderings and routing policies, some issues came up that we also did not have time to fix. These were mainly timing issues that made sometimes made the system not work but most of the time it led to subscribers receiving fewer messages than they were supposed to receive. Next we will see the implications of a crash on the performance of the system.

5 Evaluation

To evaluate our work we did three types of tests. In the first sub section we present the results of the tests on the creation of the routing tables with different number of brokers. In subsection 5.2 we present the results of tests made on the routing of events using the different types of ordering and routing policies using different numbers of publishers, subscribers, subscriptions and brokers. In subsection 5.3 we compare the results of running with a broker crashing and without crashes. We did 3 tests for each case, and then we did the average of those 3 results to get the final value, the one presented in the graphs.

5.1 Testing the routing tables

To the test the creation of the routing tables, we timed how long it took for all brokers have their routing tables created. We started with 3 brokers as depicted in figure 9 and we kept adding brokers and the results are expressed in figure 18. In the X axis is represented the numbers of brokers while in Y axis is represented the time required to create all the routing tables, measured in seconds. As we can see in the graph, the results obtained are linear, more specifically, for each broker added it took approximately more 3 seconds to create the table than before, for example, with 3 brokers it took 8.06 seconds, with 4 it took 11,24 seconds , with 5 14.33 seconds and so on. These results show that the creation of the routing tables have a good scalability because has the numbers of brokers increase, the time to create the routing table is linear with the number of brokers.

5.2 Testing the routing of events

Figure 15 and figure 16 show the results on the performance of delivering events, to note that the time measured is when subscribers start receiving events they are supposed to receive. In the X axis is represented the network topology used to test the routing and ordering policies and in the Y axis is the time it took to deliver all the events published.

Let's first consider the graph depicted in figure 15, we keep the numbers of brokers the same and just change the numbers publishers, subscribers, publications and subscriptions. For the test that resulted in this graph the following network topologies in figures, 2, 5 and 8 were used. Regarding figure 3, the script file in figure 11 was used. For figure 6 the script file in figure 12 was used and for figure 8, the script in figure 13 was used. By comparing the three network topologies used we can see that the time required to deliver the events increases as the number of the topology increases. We can also see that flooding performs much better when using a small network topology but is nearly the same to filtering when using a bigger topology. Considering flooding, we can see that total ordering perform better than no ordering and FIFO has the topology increases, filtering however no ordering keeps performing better and better than FIFO and total ordering has the network topology increases.

Now let's analyze the results depicted in figure 16, we use the same script files we used in the previous graph but this time we increase the number of brokers to 6. The network topologies are then following: figure 3 with script of figure 11; figure 6 with script of figure 12; figure 7 with script of figure 13. We can see that FIFO has big performance drop has we increase the number of brokers when used with flooding when compared to the results of figure 15. When we increase the topology, no ordering catches up FIFO in this performance drop but total ordering performs remarkably well. Considering filtering we can see that total ordering keeps loosing track to other ordering policies while we increase the network topology.

5.3 Comparing with a crash and without a crash

For our final test we tested the performance of the delivery of events when a crash occurs and when a crash doesn't occur, to note that time measured is when subscribers start receiving events they are supposed to receive. The result is shown in the graph in figure 17. For this test, the network topology of figure 4 was used together with script file of figure 14 and the same file for the non-crash version but without the crash line. By comparing both results, we can see without surprise that the non-crash version is considerably faster to deliver the events than with a crash, although we can see the performance cost caused by a crash is much greater in flooding than in filtering.

Comparing only the results obtained in the crash version, concerning flooding, total ordering performs better than no ordering and FIFO ordering, but using a filtering routing policy, we can see that using no ordering performs better than using total ordering. An interesting note is that if we look the performance achieved without a crash, we can see that flooding performs better than filtering but when a crash occurs, filtering performs better.

6 Conclusion

In this project we have successfully implemented a simplified reliable, distributed message broker supporting the publish-subscribe paradigm. This system supports flooding and filtering routing policies, and FIFO, Total and No ordering policies. Our system is fault tolerant however it has some limitations specially regarding failures when using total order.

7 References

[1] Distributed Systems: Concepts and Design (5th Edition) [George Coulouris, Jean Dollimore,Tim Kindberg, Gordon Blair]

8 Appendix

8.1 Network Topologies Used

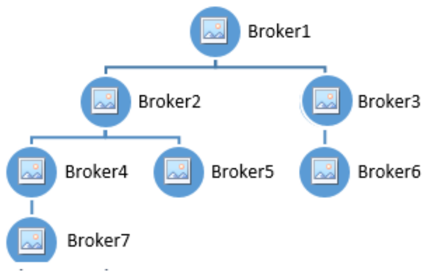


Figure 1. Broker topology

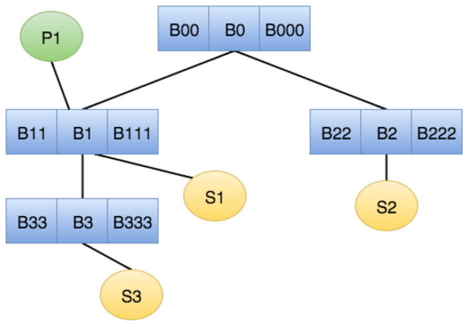


Figure 2. Network topology

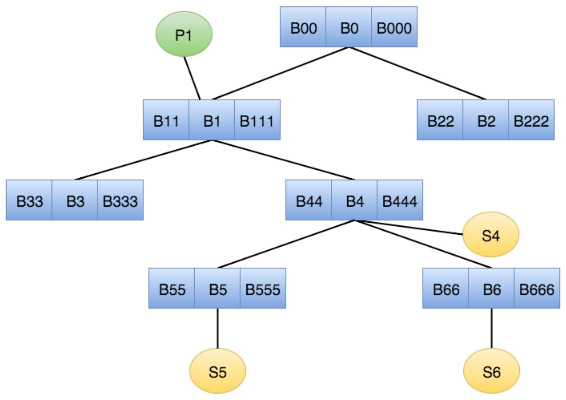


Figure 3. Network topology

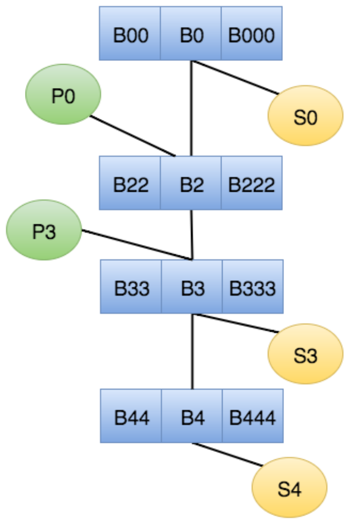


Figure 4. Network topology

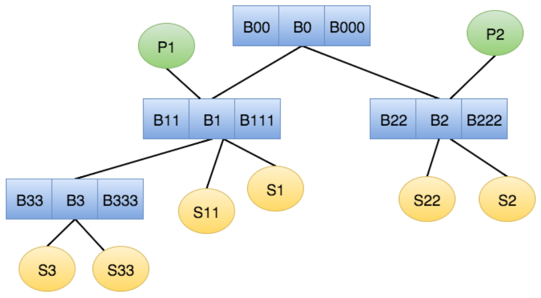


Figure 5. Network topology

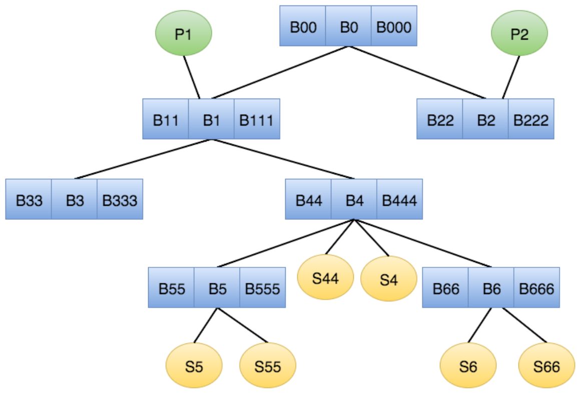


Figure 6. Network topology

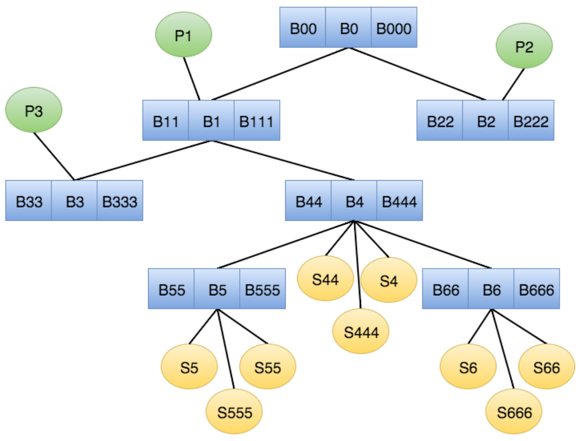


Figure 7. Network topology

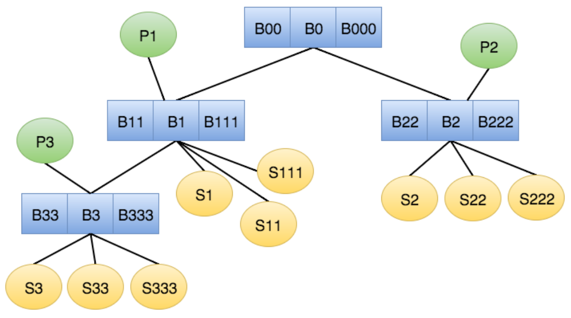


Figure 8. Network topology

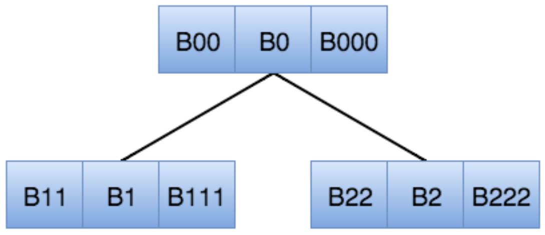


Figure 9. Broker topology

Broker1	Broker11	Broker111
Initial state: ATIVO Other possible states: ESPERA SUSPEITO MORTO	Initial state: ESPERA Other possible states: ATIVO SUSPEITO MORTO	Initial state: ESPERA Other possible states: ATIVO SUSPEITO MORTO

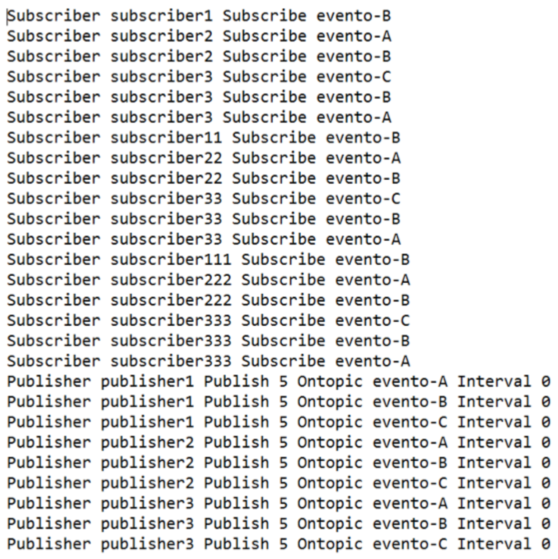
Figure 10. Broker's possible states

```
Subscriber subscriber1 Subscribe evento-B
Subscriber subscriber2 Subscribe evento-A
Subscriber subscriber2 Subscribe evento-B
Subscriber subscriber3 Subscribe evento-C
Subscriber subscriber3 Subscribe evento-B
Subscriber subscriber3 Subscribe evento-A
Publisher publisher1 Publish 5 Ontopic evento-A Interval 0
Publisher publisher1 Publish 5 Ontopic evento-B Interval 0
Publisher publisher1 Publish 5 Ontopic evento-C Interval 0
```

Figure 11. Script file used for the network topology in figure 3

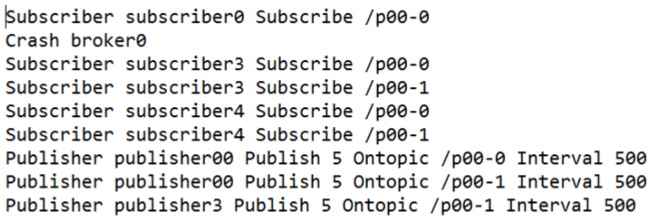
```
Subscriber subscriber1 Subscribe evento-B
Subscriber subscriber2 Subscribe evento-A
Subscriber subscriber2 Subscribe evento-B
Subscriber subscriber3 Subscribe evento-C
Subscriber subscriber3 Subscribe evento-B
Subscriber subscriber3 Subscribe evento-A
Subscriber subscriber11 Subscribe evento-B
Subscriber subscriber22 Subscribe evento-A
Subscriber subscriber22 Subscribe evento-B
Subscriber subscriber33 Subscribe evento-C
Subscriber subscriber33 Subscribe evento-B
Subscriber subscriber33 Subscribe evento-A
Publisher publisher1 Publish 5 Ontopic evento-A Interval 0
Publisher publisher1 Publish 5 Ontopic evento-B Interval 0
Publisher publisher1 Publish 5 Ontopic evento-C Interval 0
Publisher publisher2 Publish 5 Ontopic evento-A Interval 0
Publisher publisher2 Publish 5 Ontopic evento-B Interval 0
Publisher publisher2 Publish 5 Ontopic evento-C Interval 0
```

Figure 12. Script file used for the network topology in figure 5



```
Subscriber subscriber1 Subscribe evento-B
Subscriber subscriber2 Subscribe evento-A
Subscriber subscriber2 Subscribe evento-B
Subscriber subscriber3 Subscribe evento-C
Subscriber subscriber3 Subscribe evento-B
Subscriber subscriber3 Subscribe evento-A
Subscriber subscriber11 Subscribe evento-B
Subscriber subscriber22 Subscribe evento-A
Subscriber subscriber22 Subscribe evento-B
Subscriber subscriber33 Subscribe evento-C
Subscriber subscriber33 Subscribe evento-B
Subscriber subscriber33 Subscribe evento-A
Subscriber subscriber111 Subscribe evento-B
Subscriber subscriber222 Subscribe evento-A
Subscriber subscriber222 Subscribe evento-B
Subscriber subscriber333 Subscribe evento-C
Subscriber subscriber333 Subscribe evento-B
Subscriber subscriber333 Subscribe evento-A
Publisher publisher1 Publish 5 Ontopic evento-A Interval 0
Publisher publisher1 Publish 5 Ontopic evento-B Interval 0
Publisher publisher1 Publish 5 Ontopic evento-C Interval 0
Publisher publisher2 Publish 5 Ontopic evento-A Interval 0
Publisher publisher2 Publish 5 Ontopic evento-B Interval 0
Publisher publisher2 Publish 5 Ontopic evento-C Interval 0
Publisher publisher3 Publish 5 Ontopic evento-A Interval 0
Publisher publisher3 Publish 5 Ontopic evento-B Interval 0
Publisher publisher3 Publish 5 Ontopic evento-C Interval 0
```

Figure 13. Script file used for the network topology in figure 8



```
Subscriber subscriber0 Subscribe /p00-0
Crash broker0
Subscriber subscriber3 Subscribe /p00-0
Subscriber subscriber3 Subscribe /p00-1
Subscriber subscriber4 Subscribe /p00-0
Subscriber subscriber4 Subscribe /p00-1
Publisher publisher00 Publish 5 Ontopic /p00-0 Interval 500
Publisher publisher00 Publish 5 Ontopic /p00-1 Interval 500
Publisher publisher3 Publish 5 Ontopic /p00-1 Interval 500
```

Figure 14. Script file used for the network topology in figure 4

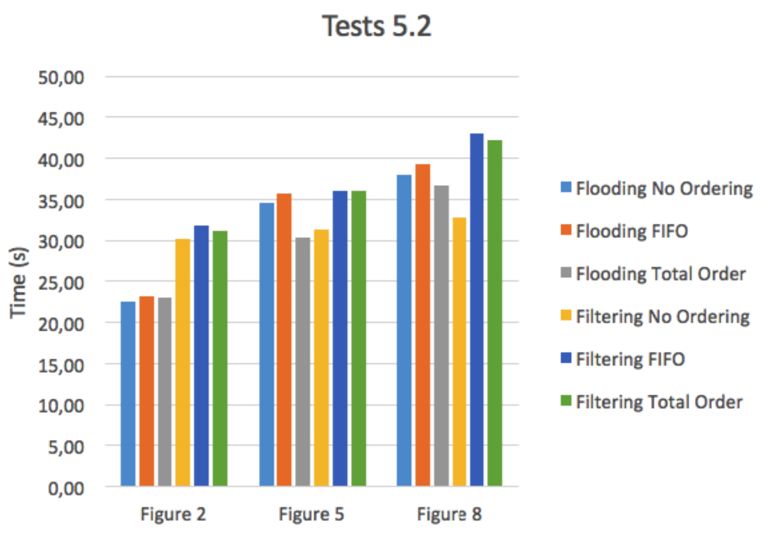


Figure 15. Results using network topologies of figures 2, 5 and 8

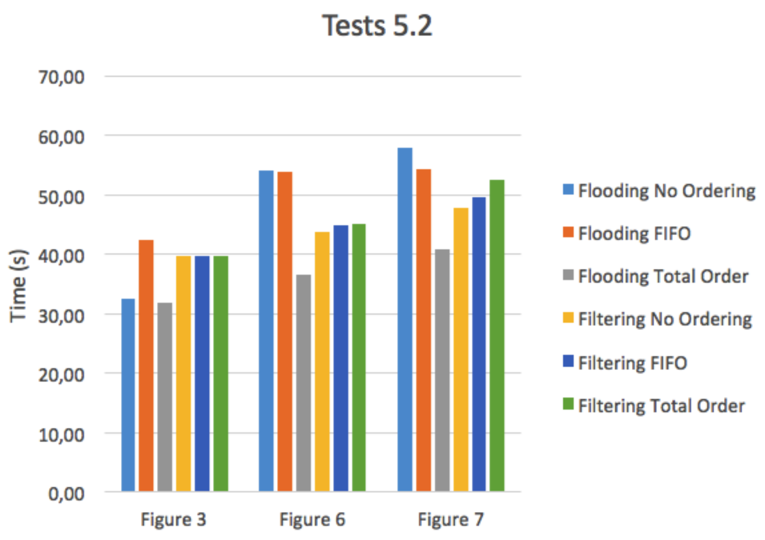


Figure 16. Results using network topologies of figures 3, 6 and 7

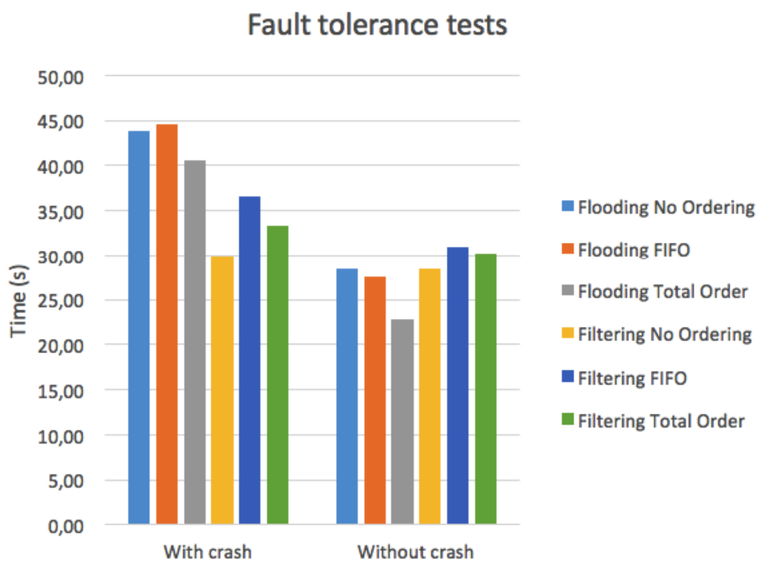


Figure 17. Results of running a script (figure 14) with a crash and without a crash using the network topology of figure 4

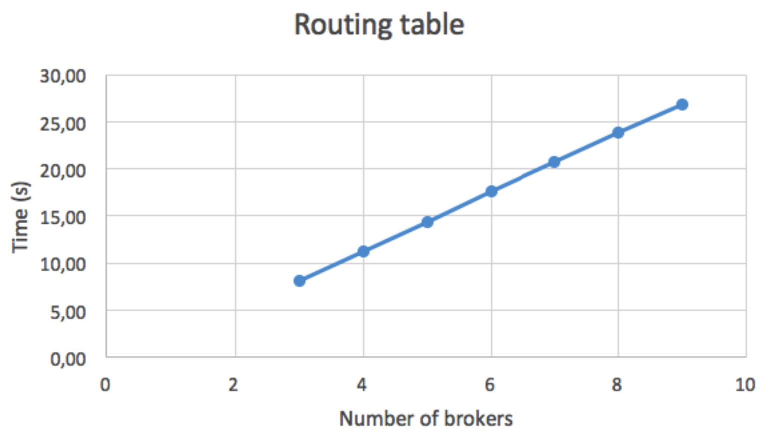


Figure 18. Results of the test on the creation of routing table