Distributed and Reliable Publish-Subscribe System

Luis Neves, Pedro do Vale, Ricardo Abreu
group 14
Instituto Superior Tcnico
Design and Implementation of Distributed Applications
1st Semester 2015/16

Abstract

Our system is a simplified implementation of a reliable,distributed message broker supporting the Publish-Subscribe system

1. Introduction

The publish-subscribe system we are aiming at involves 3 types of processes: publishers subscribers, and message Brokers. Publishers are processes that produce events on one or more topics. Multiple publishers may publish events on the same topic. Subscribers register their interest in receiving events of a given set of topics. Brokers organize themselves in a overlay network, to route events from the publishers to the interested subscribers. Communication among publishers and subscribers is indirect, via the network of Brokers. Both publishers and subscribers connect to one Broker (typically, the nearest Broker in terms of network latency) and send/receive events to/from that Broker. Brokers coordinate among each other to propagate events in the overlay network.

2. Assumptions

We assume that processes can run on different locations (or sites). For simplicity, we assume that all the sites are displayed in a tree shaped way and there is also a "config.txt" file that describes that display. A Publisher always connects to the Broker at its site, and similarly for a Subscriber. 3 Brokers are assign to each site. When a Publisher publishes an event, he associates an unique and incremental sequence number to that event. A Subscriber does not unsubscribe a topic he hasn't subscribed.

3 Problem

Publishing events can be routed in two different ways: flooding and filtering. In the flooding approach, events are broadcasted across the tree. In the filtering approach events only follow the path that lead to someone interested in that event. Subscriptions and Unsubscriptions are always flooded through the tree. In terms of ordering of these events, three orders of increasing magnitude are provided. No order, FIFO order, Total order. In the FIFO order, events are guaranteed to be delivered in the same order that the (one) publisher published them. There is no consideration for other publisher's events. In the final and strongest order method, Total order, events are guaranteed to be delivered in the same order at all matching subscribers, even for different publishers. Finally, the fact that a Broker may crash means that a Publisher should be able to connect to any of its three Brokers, and the same for a Subscriber. Brokers should be able to connects to anyone in its trios of neighbors. This should ensure that the system remains operational even if a Broker fails at each site.

4 Events and Routing

We opted to split the processing of the events in two parts. A notifying part and a propagation part. The first one is where the Broker checks if any of the Subscribers connected to his site (therefore to him) are interested in that Event's topic, and if so, notifies them. In this part we deal with the ordering requisites such as FIFO and Total Order. The propagation part is the acting of forwarding the Event across the tree, either filtering or flooding mode (routing requisites). For all the cases, when a subscription/unsubscription arrives, if it came from the Subscriber himself, it is added/removed to/from the list of subscriptions of that Broker. Otherwise (it is from a neighbor Broker) the routing table is updated. This updated is not the same for both types of event though. When an unsbscription arrives we can only remove it from the corresponding routing table

if the Subscriber that just unsubscribed was the only one interested in that topic. In contrast, when a subscription arrives it is always added to the list of the corresponding Broker.

4.1 Flooding

The flooding approach is pretty straight forward, a Broker maintains a list of its neighbors' sites and, upon propagation, sends the event he just got to everyone of them. As noted before, each neighbor site (Parent and Child in the tree overlay), has 3 Brokers. Each Broker also keeps track of the leader of each one of those (neighbor) sites, so that it only floods the event to it, not its replicas.

4.2 Filtering

For the filter approach, the neighbor's list is not enough. We need to know what topics are of interest for every neighbor. This way, we created a routing table which consisted of a neighbor site and a list of all the topics that should be routed to it. This way, upon propagation, we match the received event's topic against every one of these lists in the routing table. If there is a topic match the event is propagated through that path.

5 Event Ordering Solution

5.1 FIFO

In order to perform a FIFO delivery for a given publisher's events, a Broker needs to keep track of the current sequence number it is expecting for every Publisher. This way, when a Broker receives a subscription from a site S, checks if it has already received any subscriptions from that site. If the answer is negative, he then proceeds to send its expected sequence numbers to "S", so that the newly joined site knows what event numbers to be expecting for the publishers. There are two parts: When a Broker receives a publishing event (either from a Publisher or one of its neighbors) it checks if the event's sequence number is the one it is expecting. If it is, he notifies any of its subscribers that may be interested in that event's topic and increases its expected sequence number for that Publisher. If not, the event is added to a waiting list. After this, the Broker iterates through the waiting list to check if he can process any of its messages. If the system is running in FIFO order and filter routing, there is a slight modification to the usual forwarding of events. In filter mode the events are only forwarded to a given site if there is a path through that site that leads to an interested Subscriber. This means that a Broker may not be expecting the current sequence number for a given Publisher. This way, a Broker has to keep the highest sequence number it has sent to each of its neighbors. This number will be used before propagation to renumber the event with the sequence number the receiving Broker is expecting.

5.2 Total Order

To perform a Total Order of all the events in the system, we assigned a site to be the sequencer of the Events. Events now, are ordered by a special sequence number - given only by the sequencer - instead of the former sequence number which was assigned by the creator of the Event (Publisher). Given that our overlay of Brokers is distributed in a tree, we chose the root node to be the Sequencer. When a (non root) Broker receives an event forwarding it checks if that event has already passed through the root by checking if the special sequence number is assigned. If it has not, the Broker sends the just received Event to its parent. This way, the Event will eventually reach the root node. When the received Event has already gone to the root, the process is that same as the one used for the FIFO order. But this time, the coordination is based on the special sequence number. The root Broker has now more responsibilities when compared to a regular Broker. When an event is received and its special sequence number isn't set, the root checks if it has already assigned a total order sequence number for that event. It does this by keeping every sequence number of a given Publisher it has already assign a special sequence number to.

6. Fault Tolerance

The basic architecture described above has the disadvantage that, if a Broker fails, the entire system stops operating, because events can no longer be routed from publishers to providers. The fault-tolerant variant of the architecture aims at overcoming this limitations. In this version of the system, 3 Brokers are assigned to each site. This should ensure that a site remains operational in face of the failure of a single Broker at each site. The way we solved this problem was by stating that subscribers and publishers send their events to three Brokers, these Brokers are replicas. Only one of those Brokers will then propagate the event to other Brokers, the leader. Broker leaders only send events to other neighbor leaders. After receiving an event the leader informs its replicas of the changes made.

6.1. Types of Failure

There are two kinds of failure. A crash is detected when some process tries to communicate with a different process and the connection fails. The other kind of failure is harder to detected because it's when the process doesn't receive a reply after a given time frame. The question is: should we

assume that the process which is not responding crashed? We will explain how the different processes reacts when a Broker fails.

6.2. Crash

This type of failure is pretty straight forward. A Publisher/Subscriber can't establish a connection to send an event to a Broker and it sends a message to the other replicas of that site and the receiving replicas elect a new leader. The new leader forwards the event upon which the subscriber/publisher detected the failure.

6.3 Freeze and Unfreeze

We say that a process is frozen when we are simulating the lack of a response due to a high latency scenario. This way, whenever the method freeze() is called on a Broker, we set a flag to true. Upon receiving any event, if the Broker is "frozen" the event is added to a queue. This queue stores all the events received while frozen. When an unfreeze() call is received the Broker will first iterate through the frozen events list trying to find any event that tells him if there was a new leader elected (IamLeader events). Afterwards, he does the same but for events responsible for updating control structures. After these 2 checks and if he is still the leader for that site (which is unlikely) the unfrozen Broker can process the events on the frozen queue.

6.4. Publisher/Subscriber - Broker

When a publisher/subscriber sends a message to the Brokers he is connected to, it waits for a reply message from each Broker. If he doesn't receive a message from one of the Brokers after a given time, i.e. it reaches timeout, it sends a "are you alive" message. This message is basically a ping to the unresponsive Broker. If the Broker responds the publisher/subscriber continues its execution. If not and the new timeout is reached (this second timeout is the double of the first one), the publisher/subscriber informs the other replicas that the Broker is assumed dead.

6.5. Broker - Publisher

If a Broker receives a message stating that one of its replicas is assumed dead it checks if the unresponsive replica is the leader. If so, it elects a new leader.

6.6. New Leader - Replicas

The new leader notifies the other replicas, including the assumed dead. This is necessary because if the assumed dead replica (leader) is in fact just frozen it must know that all the events that it received in the meantime are not supposed to be forwarded. Also it forwards the event in which the old leader failed in case it wasn't forwarded and sends a message to all leader neighbors stating that he is the new leader for that site.

6.7. Replicas - New Leader

After receiving a message from the new leader update that information.

6.8 New leader - neighbor leader

When a new leader is elected, it must inform the other neighbor leaders of that change in the site, this is necessary because from then on, all messages addressed to that site must me sent to this new leader

6.9. Neighbor leader - New Leader

After receiving a message from a new leader from a neighbor site it stores that update and informs its replicas of the neighbor's site new leader.

7 Limitations

The mechanism to deal with unsubscriptions described in section 4 was not implemented due to the fact that we didn't realise it existed until close to the deadline. This way, we opted by not removing the topic from the routing table. This approach can make a Broker forward an event to a path that no longer leads to an interested Subscriber. As described in section 5.1 the FIFO ordering requires the notion of current sequence number to each neighbor. In our implementation we didn't do this. Instead, whenever a neighbor Broker did not lead to any interested Subscriber, an update message with no topic and no content was sent to him just so that the he could update its expected sequence number for that Publisher. This approach has the disadvantage of having the same message overhead as the flooding approach. In terms of Total Order, the fact that we assign the root to always be the sequencer is clearly a limitation both in terms of availability and performance. A solution to this problem could be, for instance, to run a Consensus algorithm (like Paxos) to agree on which value to assign to each Event without the need of a centralized Sequencer. Finally, we discussed some Load Balancing improvements such as: given the 3 Brokers per site and the tree topology of our overlay, we could assign one Broker to deal with the events from the parent node and the other two to deal with the events coming through the child nodes. There was also the possibility of implementing a simple probabilistic load

balancer that would choose a Broker among the three available. No type of load balancing was implemented due to the fact that our fault tolerance algorithms were based on a leader.

8 Evaluation

Unfortunately, our final version didn't implement all the features we proposed as mentioned above. The system was able to switch between filtering and flooding of events both in FIFO, Total Order and No order, although using the update message scheme described in 7. instead of the originally intended renumbering of events. As mentioned before we can only tolerate a faulty Broker per site at a given time. We support the failure of a Broker that is receiving events from a Publisher/Subscriber, which means that even if the Broker which started processing the events fails, the Publisher/Subscriber will notify its (Broker's) replicas and another one will be elected leader to continue the process started by the failing one. This method is similar for a Broker to Broker interaction.

Plus, the fact that we changed the core logic to deal with the faulty Brokers added some bugs that weren't fully fixed. Sometimes the system can enter in an infinite loop of message exchanging, and we believe this to be happening due to poor concurrency management in the fault tolerance algorithms that were later added. One of the bugs that were found after submitting the final version was: when a new leader is elected because a Subscriber detected a fault, the newly elected leader informs its replicas and its neighbors that he just got promoted (this is the intended action). When the fault is detected by a Publisher the newly elected leader only informs its replicas. Clearly, this is wrong. The process when dealing with a faulty node in a Publisher/Subscriber-Broker interaction should be the same, the new leader should also inform its leader neighbors for future communication.