SESDAD Report

Miguel Pires
IST – Technical University of Lisbon
Avenida Prof. Dr. Cavaco Silva, 2744-016 Porto Salvo, Portugal
miguel.pires@tecnico.ulisboa.pt

Abstract

SESDAD aims to be a simplified distributed publish-subscribe system supported by reliable message brokers. A filtering scheme was built to improve on plain flooding of the network. To meet the requirement of reliability in the brokering of messages while still providing good performance and scalability a non-generic approach was used to ensure uniform reliability at site level. This multicast scheme leverages knowledge about the supported operations to reduce the number of messages sent through the network to a minimum while still providing the necessary level of consistency between processes. Several mechanisms are used to deal with message loss due to broker failure or due to message omission in special cases of the optimized multicast approach implemented.

1. Introduction

Publish-subscribe is a paradigm known for it's flexibility and loose-coupling in which subscribers may listen to events posted by publishers without forcing each entity to be aware of the other. This also makes publish-subscribe systems more scalable, since to improve throughput one can just add nodes without that change having to be known and dealt with by other nodes. To achieve high scalability we must also take into account the network topology. In particular, we must choose a topology that is dynamic - in the sense that allows adding and removing nodes to improve scalability with relative ease - without restricting communication.

Publish-subscribe systems make use of message brokers to relay events posted by publishers to any interested subscribers. To achieve this, the brokers' function is one of store and forward [1]. The forwarding of events can be based on topic, content and other different properties. In this implementation, SESDAD uses a topic-based subscription model in which subscribers subscribe to specific subjects. The brokers filter published events by comparing them to the subscriptions known to them. However, this limits the system's expressiveness since subscribers will only be notified of events that fall into categories they explicitly subscribed to. This is counteracted by organizing the topic space into hierarchies. By using hierarchical subjects, a subscriber that subscribes to "/IST/*" will receive events published in the more specific topic of "/IST/MEIC" and any other topic that shares the same "/IST/" prefix.

To allow for efficient routing and high scalability, the broker overlay is arranged in a tree structure composed of multiple nodes. A node may have an arbitrary number of child nodes. Each node of the tree is regarded as a *site* that houses three brokers and zero or more client processes. The term client is used as a simplification where, locally, brokers act as servers and publishers and subscribers act as clients. Thus, the terms *client process* and *server process* may be used interchangeably with subscriber or publisher and broker, respectively.

It's interesting to provide different ordering and reliability guarantees on the delivery of events. Namely FIFO ordering between subscribers and publishers, where any subscriber must receive events in the same ordering in which they were published. SESDAD also guarantees reliable delivery of events by using a mechanism similar to uniform reliable multicast, albeit adapted to the problem at hand. The implementation of this mechanism took into consideration various aspects such as performance and load-balancing, while still guaranteeing reliability.

2. Architecture

The network's structure is dictated by the broker overlay and consists of a tree of brokers in which each broker communicates with the neighbouring brokers. Each node abstracts a site where a broker acts as server and zero or more clients interact with it. This exhibits an obvious limitation in terms of fault tolerance since a single broker failure would cause a partition in the tree. If a partition were to occur then messages would no longer circulate through the tree and the system would not function properly. To overcome this limitation, each site is instead composed of three brokers that are aware of each other. This improvement allows us to tolerate the fault of one broker.

However, this introduces a new problem of maintaining a consistent state between replicas. To reconcile these two requirements one could use *state machine replication* (SMR) where a broker is regarded as a deterministic state machine whose outputs are completely determined by the sequence of processed inputs. Using this technique we could ensure that the replicas would be consistent with each other by ensuring that the sequence of inputs is the same. However, since this essentially implies coordinating between replicas to agree on what inputs may be processed and processing them in each replica, it also implies the cost of an agreement protocol and the cost of spending three times as much computing power and bandwidth. So it would be desirable to find a different way to achieve a consistent state shared between replicas that also allows us to amortize the cost of having multiple machines by using them to increase the system's performance and improving scalability. Such an approach requires an in-depth analysis of the specific problem at hand and, naturally, an increase in complexity. This is discussed in section 5.

3. Routing policies

SESDAD supports two routing policies: *flooding* and *filtering*. These modes are defined at configuration time through the system's configuration file and change only the broker's functioning. We start by describing the simplest, flooding mode.

3.1. Flooding

When a broker receives a publication in flooding mode it first sends the publication to any local processes that should receive it and then forwards it to every neighbouring site (except the site that sent the publication - otherwise distributed infinite loops would occur). The verification of which local processes should receive the delivered event is done with the aid of a routing table. This table is gradually constructed as follows: The message brokers receive topic subscriptions and register them in the routing table that, for each subject, aggregates a list of sites and processes to which subsequent events should be forwarded to. When a message broker obtains the *matchlist* - the list of sites and processes to which the events should be forwarded to - of a given topic, it then iterates through it and checks, for each entry, if the site matches the broker's own site. If it does, then the corresponding process is a local subscriber and the broker sends it the event.

Note that since a published event will be sent to every node in the broker overlay irregardless of which nodes may actually have interested processes, then the subscriptions do not need to be forwarded remotely and may only be registered at each site. So, in practice, the routing table only contains local subscriptions and the broker can send the delivered event to every entry. Keeping subscriptions locally prevents sending unnecessary traffic through the network and wasting bandwidth and computational resources. Notice that, since in flooding mode every event is sent to every site, the tree is quickly overloaded with messages, which causes delays and, consequently, other messages to block. This quick overloading of the message brokers causes a general decrease in throughput. This is obviously undesirable and motivates the next routing policy, filtering mode.

3.2. Filtering

When a broker receives a publication in filtering mode it sends it to local processes through the same mechanism that was described in the previous subsection. The main difference is that subscriptions are now forwarded to other sites after being registered locally. This implies that the brokers must check their routing tables and either send events to local processes or forward them to neighbouring sites according to the existence of subscribed processes in that direction. In order to do this, a subscription is delivered with, not only the topic and process name of the subscriber, but also the last site from which the subscription came (i.e., the other site in last hop). This information is necessary because each broker knows only about sites in it's immediate vicinity. Thus, when forwarding an event a broker must send it to the closest site in the path to the target site.

4. Ordering guarantees

SESDAD supports two ordering guarantees, the most interesting of them being FIFO ordering which will be described in the next subsection. The system also provides the option of having no ordering guarantee. This can be chosen as a setting in the configuration file.

4.1. FIFO ordering

As was stated previously, SESDAD provides an end-to-end guarantee of FIFO ordering. This ordering guarantee ensures that subscribers receive events in the same order they were published by a publisher. More formally, if a publisher p publishes event e_1 followed by e_2 then any subscriber s will deliver events e_1 and e_2 in the same order. To implement this, SESDAD makes use of sequence numbers to order events upon arrival. When a publisher p publishes events e_0, e_1, ... e_n, it assigns them with sequence numbers s_1^p, s_2^p, ... s_n^p so that receiving processes are able to determine if an event can be delivered or if it should be blocked in a hold-back queue. For instance, if due to the asynchronous nature of the network, the event e_1 was received by a subscriber s before event e_0 then s could examine the sequence number of e_1 and determine that it differs from the last known sequence number, s_0^p, by more than a single unit. With that information s would queue e_1 until event e_0 had been processed. After processing e_0, e_1 would be unblocked.

There is a situation where the usage of sequence numbers and filtering collide: If a subscriber s subscribes to a topic on which a publisher p has been continuously publishing events then p's sequence number s_x^p has advanced from it's initial value of s_0^p and can't be delivered as is to the subscriber. This is dealt with by maintaining a mapping of sites and sequence numbers at each broker. The inbound sequence number is replaced by a sequence number corresponding to the outbound site. So, if in site $site_0$ ten events have been published on topic t and forwarded to $site_1$ and a subscription arrives coming from $site_2$, then the next event published on topic t will be forwarded to $site_1$ with sequence number s_{11}^p and to $site_2$ with sequence number s_1^p.

5. Fault tolerance

As was discussed in section 2, to deal with the problem of any broker failure partitioning the network and halting the system's execution, it was determined that a site would be composed of three brokers. This introduces a new problem of maintaining a consistent shared state. To avoid using a generic approach like *state machine replication* (SMR), in which a broker is regarded as a deterministic state machine and every replica processes the same inputs [2], we analyze the problem to a greater level of detail. This allows us to take advantage of specific particularities in the operations of the fault tolerance problem we're trying to solve.

Instead of the aforementioned approach of using SMR, the implemented scheme uses an adapted *uniform reliable broadcast* (URB) in which a publish event is sent to a randomly chosen broker. This broker multicasts the event and waits for one reply - there are three replicas and we wish to tolerate one fault. Upon reception of the acknowledgment, the replica that received the initial message begin the delivery process. This implies checking the event's topic against the broker's routing table to check to which local subscribers should receive it and also to which sites should it forward the event[1]. Note that, when forwarding the event, the broker also chooses a random broker in order to distribute the load between the replicas. The replicas that receive the multicasted event only update their state (i.e., routing tables, sequence numbers, etc) without fully processing the event.

This optimization is possible due to the usage of sequence numbers between each broker and allows us to use multiple techniques to cope with message loss and process failure without processing and sending events redundantly. These techniques will be described in section 6.

There is a problematic situation that is caused by this optimization. Since now there is no agreed upon order of events that is attributed to inputs (i.e., subscription messages are not required to be ordered in any particular way to publications) the following situation can occur: Let there be two client processes, a subscriber s and a publisher p, and three brokers b_1, b_2 and b_3 in a site $site_0$. Suppose that s and p concurrently send subscription and publication messages, e_s and e_p, respectively, on the same topic to a randomly chosen broker. Suppose also that e_s is received by b_1 and e_p is received by b_3 and, since there is no agreed upon ordering, b_1 delivers e_s first and then the e_p update and b_3 delivers e_p first and then the e_s update. In this situation, b_1 will advance it's outbound sequence number, from s_0^p to s_1^p, since e_s was already delivered and e_p is sent to s. However, since e_s wasn't delivered at b_3, it will discard e_p without advancing s_0^p and then register s's subscription. In the end of this simple hypothetical case, both brokers have registered the same subscription but have different outbound sequence numbers s^p and, therefore, inconsistent states. It is obvious how subsequent events would be affected by this inconsistency. The next published event might be delivered at b_3 and it would be sent to s with s_1^p. The subscriber would discard it as it already received an event with that sequence number.

To cope with this situation, a broker, the *designated*

[1]Assuming a filtered system. In flooding mode the last check would be ignored

broker, is chosen[2] during the system's start-up phase to receive subscriptions. The broker that receives the initial subscriptions processes them and sends updates to the other replicas. In order to achieve a consistent shared state, the update carries with it the designated broker's state which the other replicas use to replace their own states with. By imposing the designated broker's state on the other replicas we avoid the situation where some other broker would receive a different sequence of inputs that resulted in a different state and, consequently, different outputs. If a replica b_r receives an event e_p before it replaces it's state with the designated broker's b_d state and b_d receives e_p after it has sent it's state, then the most that will happen is that b_r won't advance it's sequence number like b_d will when it receives e_p. This situation is similar to a message loss and so, is not problematic because we already have mechanisms to deal with message loss due to a broker failure. These mechanisms will be described in section 6.

6. Coping with message loss

Several simple mechanisms were implemented to allow the system to recover from broker crashes and other nefarious situations. These mechanisms only come into effect after a certain time has passed since an event and, as such, may be configured to be more or less aggressive. If the waiting period is too short, then we may act unnecessarily, but if it's too long, we increase latency due to failures even more. The two main mechanisms for dealing with message loss are *retransmission requests* and *notifications*.

6.1. Retransmission requests

In section 5 we described how failure of a broker could be tolerated by ensuring that a broker only delivers an event if other brokers are guaranteed to deliver it as well. However, even if the state between multiple replicas is consistent, we need to have a mechanism to retransmit lost messages as needed. For that purpose, when a delayed message is detected and an event e_x is queued, a thread, that initially sleeps for a second and a half, is launched. After waking, the thread checks if there are any events blocked in the hold-back queue and checks if it's sequence number is equal to the minimum sequence number of the blocked messages. If there are blocked messages and the minimum sequence number is the same as it's own, then we know that the event e_{x-1} is still missing and is delaying the delivery of subsequent events. In this case, the process requests a retransmission either from the site from which e_x came from or from the publisher itself. Both publishers and brokers keep a history of processed messages in order to be able to quickly respond to retransmission requests.

This simple mechanism, used by both subscribers and brokers, proves effective in dealing with messages lost due to crashes and also helps to deal with situations where a message is omitted in a special situation of the fault tolerance scheme described in section 5.

6.2. Notifications

The mechanism described in the previous subsection relies on the continuous stream of events to detect message loss. In particular, it detects that a given event e_x is missing if the e_{x+1} is blocked in a queue for a certain period of time. This works well and is an efficient way of detecting possible messages losses since no additional traffic, other than the request, is generated. There is, however, one situation that is not covered by the previous message: the last message sent. Consider the situation where the broker chosen to deliver the last message multicasts the event and then crashes. The event is accounted for in the replicas' state but wasn't neither forwarded to other sites nor sent to local subscribers. Since this event is the last in the stream then it's loss wouldn't be noted using only the previously described mechanism.

To deal with this case, a thread is launched much like in the requesting mechanism and sleeps for three seconds. After it awakes, the thread checks if it's sequence number is the last to be received by checking both the inbound sequence numbers and the hold-back queue. It also checks if the subscriber unsubscribed from the topic since, if it did, then it doesn't make sense to receive any other publication about it. If the sequence number corresponds to the last event received then the subscriber notifies a broker of it's value. Upon arrival of the notification at the broker, two things happen: First, the broker checks if it has newer messages in it's history. If it does, then it resends every message, on the same topic, with a sequence number larger than the value of the sequence number sent with the notification. Second, the broker sends the notification "up the chain". It sends the message to where the corresponding event came from but this time the sequence number is updated to reflect the messages it has already resent so as to avoid other processes resending events that it has already resent. In this way, the notification travels the inverse path to the published event and, if at any point, a missing message is detected then the process that detects it resends the missing events.

[2]For example, the one with the lowest process identifier

The path stops at the publisher, who also checks if the sequence number corresponds to the one it last sent and resends missing events. Notice that since the brokers map outbound sequence numbers onto inbound sequence numbers due to the filtering process, in this reverse path they must do the opposite.

7. Other mechanisms

Other mechanisms worth mentioning are the ones for log and command delivery, that work in a way similar to each other. Upon start-up the PuppetMaster"Master" and PuppetMaster"Slaves" launch two threads. Both read from buffers that are filled with commands, in one case, and log entries, in the other. This is similar to the producers-consumers problem studied in the Operating Systems course but with the simplification that the buffer is not a rotating one and doesn't allow for as much concurrency. When a log is delivered there is an access to a critical section followed by the insertion of the log line in a queue and a "Pulse" instruction that warns the reading thread that there are new items in the buffer. The thread awakens and writes whatever it finds in the GUI. In this way, log and command delivery can be done efficiently without becoming a bottleneck for both site processes and PuppetMasters.

8. Evaluation

The system's evaluation will be based in the comparison of the designed solution with other possible solutions and results observed for locally run tests. Real-life, distributed tests could not be run since the laboratory computers available have restrictive firewall rules due to security reasons and therefore didn't allow for the testing of SESDAD.

We shall first discuss how the designed solution improves on a generic solution such as *state machine replication* (SMR). SMR would require exchanging messages between the three brokers to agree on the imposed ordering of events received in a site. In particular, a broker would broadcast a request to which each replica would reply with a broadcast with the maximum sequence number it has seen or accepted. Each replica would then accept the maximum candidate sequence number observed after receiving a response from every non-faulty replica[3].

In the designed solution, the broker that first receives the event broadcasts it and waits for exactly one response. Both replicas that receive the update also broadcast and deliver immediately. This is possible because the chosen broker must only receive one acknowledgment before delivering and any replica that receives the update already knows that another replica is ready to deliver. This improves greatly the waiting period between the initial reception of the event and the delivery by every replica. This improvement in parallelism is possible because there is no imposed total order without sacrificing consistency.

The desired load-balancing is also provided since every process will randomly choose a broker to which it will send a published event. In fact, although there is no load-balancing for subscription and unsubscription events, the load-balancing and scalability improvement is still significant since, under a normal workload, publish events greatly outnumber any other type of events.

9. Conclusions

Given the initial proposition of building a hierarchical, topic-filtered publish-subscribe system while providing reliability and ordering guarantees, we've seen how simple hierarchical filtering could be implemented. We've also studied different approaches on how to provide reliability and found that a generic approach could be outperformed by analyzing the possible operations and using a protocol similar to URB to guarantee reliability while allowing for a greater level of parallelism.

References

[1] `https://www.wikiwand.com/en/Store_and_forward`.

[2] F. B. S. Author. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4), December 1990.

[3]Note that this is as described in Schneider's "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial", irregardless of any optimization later proposed