SESDAD

Desenvolvimento de Aplicações Distribuídas
Project - 2015-16 (IST/DAD): MEIC-A / MEIC-T / METI

October 1, 2015

Abstract

The DAD project aims at implementing a simplified (and therefore far from complete) implementation of a reliable, distributed message broker supporting the publish-subscribe paradigm (in Portuguese, "Sistema de Edição-Subscrição", SES).

1 Introduction

The goal of this project is to design and implement **SESDAD**, a simplified (and therefore far from complete) implementation of a reliable, distributed, message broker supporting the publish-subscribe paradigm (in Portuguese, "Sistema de Edição-Subscrição", SES).

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.

In **SESDAD**, we consider that events only have two fields: a *topic* and a *content*, both strings. The string that identifies the topic can be seen as representing a hierarchical name space. For instance, an event with topic "/edu/ulisboa/tecnico/meic/dad" would contain information regarding this course and an event with topic "/edu/ulisboa/tecnico/meic" would contain information relevant for all students and professors of the MEIC. In order to start receiving events, a subscriber need to perform a *subscription*. A subscription has, as input parameter, a prefix of the name space. For instance, a process may subscribe to "/edu/ulisboa/*" to receive events for all topics under the "/edu/ulisboa/" prefix. A subscriber may make more than one subscription. Whenever a message is delivered to the subscriber,it should be displayed on that subscriber's user interface (console or GUI). When a subscriber is no longer interested in receiving events from a set of topics it has previously subscribed, it can issue an *unsubscribe* request.

2 Basic Architecture

We consider a distributed system where processes may execute in different geographic locations, or *sites*. For simplicity's sake, sites may be emulated within the same physical machine.

To simplify the project, we assume that there is a configuration file that describes the entire network: how many sites exist and which processes belong to each site. Also, for the purpose or building the overlay of brokers, we assume that sites are organised in a tree. The first part of the configuration file enumerates the existing sites and specifies how the tree of sites is organised. This part of the configuration file is composed of a sequence of lines in the form:

`Site` *sitename* `Parent` *sitename*|*none*

For instance, the following file describes a system composed of 3 sites, *site0*, *site1*, and *site2* organised in a binary tree where *site0* is the root:

`Site` *site0* `Parent` *none*
`Site` *site1* `Parent` *site0*
`Site` *site2* `Parent` *site0*

The rest of the configuration file specifies the logical names of the processes, their role, and the site they belong to. This part of the configuration file is composed of a sequence of lines in the form:

`Process` *processname* `Is` *publisher* |*subscriber*|*broker* `On` *sitename* `URL` *process-url*

The final `URL` parameter designates the URL where a process is providing it's services. If the URL is a remote URL, the process will be started by a remote PuppetMaster (see the PuppetMaster section ahead). For instance, the following file indicates that there is one broker (*broker0*), one publisher (*publisher0*), and two subscribers (*subscriber0* and *subscriber1*) on *site0*

`Process` *broker0* `Is` *broker* `On` *site0* `URL` *tcp://1.2.3.4:3333/broker*
`Process` *publisher0* `Is` *publisher* `On` *site0* `URL` *tcp://1.2.3.4:3334/pub*
`Process` *subscriber0* `Is` *subscriber* `On` *site0* `URL` *tcp://1.2.3.4:3335/sub*
`Process` *subscriber1* `Is` *subscriber* `On` *site0* `URL` *tcp://1.2.3.4:3336/sub*

For the basic architecture, assume that there is always a single broker at each site. Also, publishers and subscribers always connect to the broker on their own site.

The interactions among processes are as follows:

- *Broker-Broker:* As noted before, brokers are organised in a tree. Therefore, each broker only interacts directly with its parent and its children in the tree (if any). The communication among brokers aims at: propagating information regarding subscriptions and unsubscriptions, such that event routing can be optimised (in particular, such that events are not propagated in tree branches where there are no subscribers for those events); and to propagate the events along the tree.
- *Publisher-Broker:* The publishers interacts with a broker to publish events. A publisher sends the events it produces to the broker at its site without being aware of other brokers nor of the number or location of subscribers for that event.
- *Subscriber-Broker:* Subscribers interact with the broker at their site. Subscribers forward to their broker the subscription and unsubscription requests. Subscribers are not aware of the number and location of publishers for the topics they subscribe. Also, subscribers export a callback that can be invoked by the broker to deliver events to the subscriber. There may be an arbitrary delay between the action of subscribing a topic and starting to receive messages on tha topic. Once messages on a topic start being delivered to a subscriber, no message on that topic may be lost or delivered out of order.

3 Event Routing

The **SESDAD** system shall implement two different policies for event routing:

- *Flooding*: In this approach events are broadcast across the tree.
- *Filtering based*: In this approach, events are only forwarded only along paths leading to interested subscribers. To this end, brokers maintain information regarding which events should be forwarded to their neighbours.

The above policies are intentionally defined in an abstract manner. It is up to the students to instantiate concrete algorithms that implement these three event routing policies, while striving to optimize system's efficiency, scalability and load balancing. In particular, attention shall be placed to exploit in an efficient way the tree-based topology of the broker's overlay.

The event routing policy to be used by the **SESDAD** system is defined in the configuration file, in a line with the format:

`RoutingPolicy` *flooding|filter*

If the line is missing, the system should default to flooding.

3.1 Ordering guarantees

The **SESDAD** system provides three types of ordering guarantees for the notification of events, namely *Total order*, *FIFO order* and *No ordering*.

The semantics of the three ordering guarantees supported by **SESDAD** are specified in the following:

- *Total order:* all events published with total order guarantees are delivered in the same order at all matching subscribers. More formally, if two subscribers $s1, s2$ deliver events $e1, e2$, $s1$ and $s2$ deliver $e1$ and $e2$ in the same order. This ordering property is established on *all* events published with total order guarantee, independently of the identity of the producer and of the topic of the event.
- *FIFO order:* all events published with FIFO order guarantee by a publiser p are delivered in the same order according to which p published them.
- *No ordering:* as the name suggests, no guarantee is provided on the order of notification of events.

The event ordering guarantee is defined in the configuration file provided to all nodes, in a line with the format:

`Ordering` $NO|FIFO|TOTAL$

If the line is missing, the system should default to FIFO. The specified ordering guarantee is applied to *all* events produced in the system.

4 Fault-Tolerant Architecture

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. Note

that, in the fault-tolerant version, sites are still organised in a tree. However, a broker at a site can now connect to one or more of the 3 servers that may be operational in its parent site. Similarly, a broker may connect to more than one broker at each of its children sites. Finally, publishers and subscribers may connect to any of the brokers in their site. In case of failures, the system should reconfigure automatically, without disrupting the reliable stream of events. For instance, if the broker to which a publisher is connected crashes, the publisher should automatically reconnect to another broker at its site and subscribers should receive, nevertheless, all the events in the order by which they have been sent.

The students are free to use the replication technique they feel more appropriate to solve the problem at hand. Also, the students are encouraged to use techniques that may leverage from the availability of different brokers to perform some load balancing whenever possible.

5 PuppetMaster

To simplify project testing, brokers, publishers, and subscribers will also connect to a centralised *PuppetMaster*. The role of the PuppetMaster process is to provide a singe console from where it is possible to control experiments. The activation of the PuppetMasters will be performed manually, for simplicity, although they could be configured to be executed as daemons.Thus, after connected to a broker, publishers and subscribers just wait for instructions from the PuppetMaster, before issuing notifications, subscriptions and unsubscriptions. It is the PuppetMaster that reads the system configuration file and starts all the relevant processes. All PuppetMasters should expose a service at an URL, called PuppetMasterURL, that creates worker pro- cesses on the local machine. This service can be used by PuppetMasters on other machines to create new processes of any type. For simplicity, we assume that all the PuppetMasters know the URLs of the entire set of PuppetMasters. This information can be provided, for instance, via configuration file or command line. The PuppetMaster can send the following commands to the other processes:

- `Subscriber` *processname* `Subscribe` *topicname*. This command is used to force a subscriber to subscribe to the given topic.
- `Subscriber` *processname* `Unsubscribe` *topicname*. This command is used to force a subscriber to unsubscribe to the given topic.
- `Publisher` *processname* `Publish` *numberofevents* `Ontopic` *topicname* `Interval` *x_ms*. This command is used to force a publisher to produce a sequence of *numberofevents* on a given topic. The publisher should sleep x milliseconds between two consecutive events. The content of these events should be a string that includes the name of the publisher and a sequence number.
- `Status`: This command makes all nodes in the system print their current status. The status command should present brief information about the state of the system (who is present, which nodes are presumed failed, which subscriptions are active, etc...). Status information can be printed on each nodes' console and does not need to be centralized at the PuppetMaster.

In addition, the PuppetMaster may also send to the other processes commands that aim at simplifying debugging, for instance, by forcing certain processes to crash. Namely, the PuppetMaster can send the additional commands to the other processes:

- `Crash` *processname*. This command is used to force a process to crash (can be sent to publishers, subscribers or brokers).

- `Freeze` *processname*. This command is used to simulate a delay in the process (can be sent to publishers, subscribers or brokers). After receiving a freeze, the process continues receiving messages but stops processing them until the PuppetMaster "unfreezes" it.
- `Unfreeze` *processname*. This command is used to put a process back to normal operation. Pending messages that were received while the process was frozen, should be processed when this command is received.

The PuppetMaster should have a simple console where an human operator may type the commands above, when running experiments with the system. Also, to further automate testing, the PuppetMaster can also read a sequence of such commands from a *script* file. A script file can have an additional command that controls the behaviour of the PuppetMaster itself:

- `Wait` *x_ms*. This command instructs the pupper master to sleep for x milliseconds before reading and executing the following command in the script file.

For instance, the following sequence in a script file will force broker *broker0* to freeze to $100ms$:

`Freeze` *broker0*
`Wait` *100*
`Unfreeze` *broker0*

The PuppetMaster should produce a time ordered log of all events it triggers or observes. All events triggered by the PuppetMaster (Subscriber-Subscribe, Subscriber-Unsubscribe, Publisher, Freeze, Wait, Unfreeze, Crash) appear in the log with the same syntax as in the script files. Publishers, brokers and subscribers should notify the PuppetMaster whenever they publish, forward or receive a message.

The publication of an event by a publisher should appear in the log as:
`PubEvent` *publisher-processname, publisher-processname, topicname, event-number*
The forwarding of an event by a broker should appear in the log as:
`BroEvent` *broker-processname, publisher-processname, topicname, event-number*
The delivery of an event to a subscriber should appear in the log as:
`SubEvent` *subscriber-processname, publisher-processname, topicname, event-number*

There are two levels of logging, light and full. The logging level to be used by the **SESDAD** system is defined in the configuration file, in a line with the format:

`LoggingLevel` *full|light*

If the line is missing, the system should default to light.

In the full logging mode, all events described above should be included in the log. In the light logging mode, the forwarding of events by the brokers need not be included in the log and therefore the brokers need not notify the PuppetMaster of those events.

6 Final report

Students should prepare a final report describing the developed solution (max. 6 pages). In this report, students should follow the typical approach of a technical paper, first describing the problem they are going to solve, the proposed solutions, and the relative advantages of each solution. The report should include an explanation of the algorithms used and

justifications for the design decisions. The project's final report should also include some qualitative and quantitative evaluation of the implementation. The quantitative evaluation should focus on the following metrics:

- Routing of events.
- Fault-tolerance features.
- Other optimisations.

This should motivate a brief discussion on the overall quality of the protocols developed. The final reports should be written using LaTeX. A template of the paper format will be provided to the students.

7 Checkpoint and Final Submission

In the evaluation process, an intermediate step named *project checkpoint* has been scheduled. In the checkpoint the students may submit a preliminary implementation of the project; if they do so, they may gain a bonus in the final grade. The goal of the checkpoint is to control the evolution of the implementation effort. Given that students are expected to perform an experimental evaluation of the prototype, it is desirable that they have a working version by the checkpoint time. In contrast to the final evaluation, in the checkpoint only the functionality of the project will be evaluated and not the quality of the solution.

For the checkpoint, students should implement the entire base system, excluding: i) total ordering of events (i.e., only no order and fifo order guarantees shall be supported); ii) the fault-tolerant architecture. After the checkpoint, the students will have time to perform the experimental evaluation and to fix any bugs detected during the checkpoint. The final submission should include the source code (in electronic format) and the associated report (max. 6 pages). The project *must* run in the Lab's PCs for the final demonstration.

8 Relevant Dates

- November 6^{th} - Electronic submission of the checkpoint code;
- November 9^{th} to November 13^{th} - Checkpoint evaluation;
- December 4^{th} - Electronic submission of the final code.
- December 7^{th} - Electronic submission of the final report.

9 Grading

A perfect project without any of the fault-tolerant features will receive 15 points out of 20. The fault-tolerant features are worth 5 additional points for a total of 20 points.

The project grading will depend on a discussion at the end of the semester where all members of the groups must be present and where individual grades will be determined. That grade will depend on, besides the quality of the project, the individual performance in the discussion and the lecturer's evaluation.

The project grade (45% of the course's grade) is the *best* of the following two:

- Final_Project_Grade
- 85% of the Final_Project_Grade + 15% of Checkpoint_Grade

10 Cooperation among Groups

Students must not, *in any case*, see the code of other groups or provide their code to other groups. If copies of code are detected, both groups will fail the course.

11 "Época especial"

Students being evaluated on "Época especial" will be required to do a different project and an exam. The project will be announced on January 25th, 2016, must be delivered January 29th, and will be discussed on February 1st, 2016.

The weight for the project is of 45% (as for the "Época normal"), if the student has given the paper presentation (which will have weight 15%) during the semester (and the exam will account for the remaining 40% of the grade).

If the student has **not** given the paper presentation during the semester, the weight of the project will be 60% (and the exam will account for 40% of the grade).