MSDAD

Design and Implementation of Distributed Applications 2019-2020
IST (MEIC-A / MEIC-T / METI)
Project Statement

1 Introduction

The goal of this project is to design, implement and evaluate a replicated distributed meeting scheduling system. A replicated set of servers will provide services for reading the current open meeting proposals, creating and closing meetings and joining/rejecting meeting invitations. Meeting proposals once created are broadcast among system clients. The meeting scheduler is accessed via a remote client application on which users perform all schedule operations as requests to the servers.

2 Overview

The meeting scheduling is initiated by a user that creates a *meeting proposal.* A meeting proposal has a coordinator (the user that creates the proposal), a topic (just a string, for instance, "Budget_2020"), a minimum number of participants (say "5"), a set of slots, where each slot represents a potential date and location (for instance, "Lisboa,2019-11-14 or "Porto,2020-02-03") and, optionally, a list restricting the meeting to a set of invitees. The coordinator of the meeting proposal is not required to participate in the meeting (but is likely to participate).

Each meeting proposal should be shared among all clients (or just among the invited users if the proposal includes a list of invitees). The load of disseminating the proposal among all participants should not be taken exclusively by the sender. Students should choose an algorithm that distibutes the load among the participants, such as a gossip or structured peer-to-peer broadcast algorithm.

On the scheduling server (that can be replicated), users can see the current set of meeting proposals and express their willingness to participate. When a user joins a meeting, she adds to the meeting record her own name and the subset of dates/location where she is available. For instance, Maria could express her willingness to join the "Budget for 2020" meeting by adding a record with "Maria, (2019-11-15, Porto)".

The coordinator can close the proposal at any moment. When the coordinator closes the proposal, the system automatically selects a date and a location that matches a set of constraints (for instance, the minimum number of participants is satisfied). The meeting becomes scheduled and no new participants can join.

Each meeting location has a set of meeting rooms as attributes. Each meeting room has a maximum capacity. The set of locations and rooms should be preconfigured in the system. For instance, a possible configuration would be:

Locations:

- Lisboa, (“Room A”, 20) (“Room B”, 10)
- Porto, (“Room C”, 15)

To simplify the program, we assume that a meeting occupies a room for the entire day. To extend the system in order to consider time intervals in each day would be trivial but is not relevant for this project. Thus, on each day, a room is either free or booked. A room becomes booked when a proposed meeting is closed and the meeting is scheduled for that room.

When a meeting proposal is closed, the system automatically selects the “best” data and location, i.e., the one that maximises the number of participants, subject to the following constraints:

- There is a free meeting room, in the desired location.
- The number of participants is equal or larger than the minimum number of participants specified in the proposal.
- If there are more registered participants than the capacity of the selected meeting room, the system must automatically chose which participants are excluded (using some user-defined criterium).

If there is no date/location that matches the constraints above, the meeting is cancelled. If the meeting is scheduled, the selected room becomes booked.

The students should implement two versions of the system:

- A centralised version, based on a client server architecture that uses a single server (the purpose of this version is just to debug the business logic). This version may exclude the algorithm to disseminate proposals among clients.
- A distributed system composed of multiple servers, one server running in each location. Clients connect to a single server to interact with the system. Typically clients interact with the “closest” server but, if the network is down, they may contact one of the other servers. If needed, the server coordinates with other servers before responding to the client request. The system should maximise availability

in face of network partitions. Thus, any request that can be satisfied by a server without immediate coordination with other servers, should be satisfied locally. For instance, a user should be able to initiate a meeting proposal in the “Lisboa” server, even if “Lisboa” cannot contact “Porto” at the time the request is made. Later, the “Lisboa” server should propagate information to the “Porto” server, such that clients connected to the “Porto” server can participate in the meeting. Clients can migrate from one server to another. For instance, “Maria” can open a meeting proposal by contacting the “Lisboa” server, take a train to Porto, and later contact the “Porto” server to participate in the meeting she has created earlier. The clients' interaction with the system must respect causality. For instance, using the same example, “Maria” should be able to observe the proposal she created in “Lisboa” when she gets a response from the “Porto” server. All meeting closing operations must, besides respecting causality, respect a system-wide total order.

The system should provide strong guarantees on the reliability of the distributed execution and liveness in the presence of failures (assuming that the maximum number of faults is bounded).

The project should be implemented using C# and .Net Remoting using Microsoft Visual Studio and the CLR runtime. (or alternatively, but less desireably, MonoDevelop and the mono runtime).

3 System Architecture

3.1 Servers

The **MSDAD** will consist of a set of server processes, where all of them can be contacted to perform scheduling operations. We consider a distributed system in which the scheduler state is stored on a set of machines — although, for simplicity's sake, multiple server processes may be deployed within the same physical machine. It can be assumed that servers know all clients in the system.

3.2 Clients

In **MSDAD**, there will also be an arbitrary number of client processes. A **MSDAD** client uses a client library providing an API to contact the **MSDAD** servers and execute scheduling operations. Any application using the client library can be a client.

3.2.1 Client Scripts

A special client, script-client, should be developed that simply executes an input script.

The client scripts are text files submitted to the script-client as command line parameters. They are assumed to be available on the client machines on the same directory as the client program. The scripts should be executed synchronously.

The commands a script may contain are:

- `list` : lists all available meetings;
- `create` *meeting_topic min_attendees number_of_slots number_of_invitees slot_1 ... slot_n invitee_1 ... invitee_n*: creates a new meeting identified by *meeting_topic* with a *min_attendees* required number of atendees, with a *number_of_slots* large set of possible dates and locations and with a *number_of_invitees* large group of invited users. *meeting_topic* is a string which may contain letters and the underscore character such as "budget_2020". Each *slot_n* is a location followed by a date with all elements separated by a comma and hyphens such as "Lisboa,2020-01-02". Each *invitee_n* is the username of an invited client or user (see 4 below).
- `join` *meeting_topic* : joins an existing meeting
- `close` *meeting_topic* : closes a meeting
- `wait` x : Delays the execution of the next command for x milliseconds.

3.3 Fault Tolerance

The reason for having multiple replicas of the scheduler is to provide fault-tolerance and lower latency. If a replica fails, the system's functionality will be still available at other replicas. We assume that replicas can only fail by crashing and that, when a replica fails this fact can be reliably detected by other nodes (in other words, you can assume the availability of a perfect failure detector). Thus, when a replica fails, all the other nodes are eventually informed of the failure and can update the view of active servers accordingly. A node that crashes never recovers. Also, no new nodes join the execution.

For this project, it is assumed that at most f faults may occur (where $f < N$, being N the number of servers), but that only one fault may happen at each moment and that the system will have time to recover before the next fault. Obviously, how much time is required for that recovery will be taken into account in the project grading.

3.3.1 Message Delivery Delays

The implementations should ensure that it is possible to arbitrarily delay the arrival of a message at a server. When servers are started they will should be configured to delay any incoming message for a random number of milliseconds chosen from a given interval (see Section 4 below).

4 PuppetMaster

To simplify project testing, all nodes will also connect to a centralised *PuppetMaster*. The role of the PuppetMaster process is to provide a single console from where it is possible to control experiments. Each physical machine used in the system (except for the one where the PuppetMaster is running) will also execute a process, called PCS (Process Creation Service), which the PuppetMaster can contact to launch processes (clients or servers) on remote machines. Once a process is created, it should interact with the PuppetMaster directly. For simplicity, the activation of the PuppetMaster and of the PCS will be performed manually. It should be, in principle, possible to operate the system without the need for a PuppetMaster or PCS.

It is the PuppetMaster that reads the system configuration file and starts all the relevant processes. The PCS on each machine should expose a service at an URL on port 10000 for the PuppetMaster to request the creation of a process. For simplicity, we assume that the PuppetMaster knows the URLs of the entire set of available PCSs. This information can be provided, for instance, via configuration file or command line. The PuppetMaster can send the following commands to the nodes in the system:

- `Server` *server_id URL max_faults min_delay max_delay*: This command creates a server process identified by *server_id*, available at *URL* that delays any incoming message for a random amount of time (specified in milliseconds) between *min_delay* and *max_delay*. If the value of both *min_delay* and *max_delay* is set to 0, the server should not add any delay to incoming messages. Note that the delay should affect *all* communications with the server. The parameter *max_faults* determines how many simultaneous faults may happen, i.e., how many servers may fail before an operation is guaranteed to be replicated. *Important note: delays should not cause messages to be re-ordered. For instance, if you are using TCP, and TCP delivers messages in a given order, this order should be preserved.*

- `Client` *username client_URL server_URL script_file*: This command creates a client process identified by the string *username*, available at *client_URL*, that will connect to a preferred server at *server_URL*, and that will execute the commands in the script file *script_file*. It can be assumed that the script file is located in the same disk folder as the executable.

- `AddRoom` *location capacity room_name*: This command adds a room called *room_name* with a certain *capacity* to a location *location*.

- `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, etc...). Status information can be printed on each nodes' console and does not need to be centralised at the PuppetMaster.

Additionally, the PuppetMaster may also send to the server replicas debugging commands:

- `Crash` *server_id*. This command is used to force a process to crash.
- `Freeze` *server_id*. This command is used to simulate a delay in the process. After receiving a freeze, the process continues receiving messages but stops processing them until the PuppetMaster "unfreezes" it.
- `Unfreeze` *server_id*. 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, preferably with a GUI, 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 and execute them either sequentially or step by step. A script file can have an additional command that adjusts the behaviour of the PuppetMaster itself:

- `Wait` *x_ms*. This command instructs the PuppetMaster 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 a server *broker0* to freeze to 100ms:

```
Freeze server_url
Wait 100
Unfreeze server_url
```

All PuppetMaster commands should be executed asynchronously except for the `Wait` command. Port 10001 is reserved for the PuppetMaster and can be used to expose a service that collects information from the system's nodes.

5 Performance Evaluation

Each group must evaluate the system's performance by identifying the workloads for which the implementation performs best and worst and design clients that test those situations as well as baseline scenario. The resulting performance data should be discussed in the report (see next section).

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 very briefly the problem they are going to solve, the proposed implementation solutions, and the relative advantages of each solution. Please avoid including in the report any information included in this project description. The report should include an explanation of the algorithms used and justifications for the design decisions. 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

The grading process includes an intermediate optional checkpoint and a mandatory final submission. For 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 produce working version of the systems including the replicated scheduling servers but excluding fault detection and management as well as the meeting diffusion algorithm implementation. After the checkpoint, the students will have time to add the fault tolerance mechanisms, add the meeting diffusion algorithm, 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 8th - Electronic submission of the checkpoint code;
- November 11th to November 15th - Checkpoint evaluation;
- December 6th - Electronic submission of the final code.
- December 9th - Electronic submission of the final report.

9 Grading

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, all groups involved 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 July 17, 2019, must be delivered by July 23, and will be discussed on July 24, 2019.