DIDA-TUPLE

Project
Design and Implementation of Distributed Applications
2018-19 IST (MEIC-A / MEIC-T / METI)

Abstract

The DIDA project aims at comparing two variants of a distributed fault-tolerant tuple space system.

1 Introduction

The goal of this project is to implement and compare two variants of a distributed fault-tolerant tuple space system. A tuple is a sequence of values with an arbitrary length. A tuple space [1] stores a multiset (a set where repetitions are allowed) of tuples.

These tuples are accessible using three tuple operators: *add*, *read* and *take*.

The *add* operator inserts a tuple into the tuple space. The *read* operator submits a tuple schema (a tuple where concrete field values may be replaced by wildcards) and returns a tuple from the tuple space that matches the schema without that tuple being removed from tuple space. The *take* operator submits a schema and returns a tuple from the tuple space that matches the schema and removes that tuple from the tuple space.

The tuple space should provide strong guarantees on the reliability of the distributed computation and on bounded performance in the presence of failures.

Each groups should implement two variants of the tuple space, **DIDA-TUPLE-SMR** and **DIDA-TUPLE-XL**. The students should then identify the workloads for which each of the implementations performs best and design clients that test those situations as well as baseline scenario. The resulting performance data should be discussed.

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

2 System Architecture

2.1 Servers

The **DIDA-TUPLE** tuple space will consist of a set of server processes, where all of them can be contacted to perform tuple operators. We consider a distributed system in which the tuple space is stored on a set of machines — although, for simplicity's sake, multiple server processes may be deployed within the same physical machine. In **DIDA-TUPLE**, a distributed computation can therefore be expressed as a set of programs concurrently executing tuple operators on the tuple space.

2.2 Clients

In **DIDA-TUPLE**, there will also be an arbitrary number of client processes. A **DIDA-TUPLE** client uses a client library providing an API to contact the **DIDA-TUPLE** servers and execute tuple operators. Any application using the client library can be a client.

2.2.1 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:

- `add` $< field_1, ..., field_n >$: Adds a tuple to the tuple space
- `read` $< field_1, ..., field_n >$: Reads a tuple from the tuple space which matches the schema.
- `take` $< field_1, ..., field_n >$: Removes a tuple from the tuple space which matches the schema.
- `wait` x : Delays the execution of the next command for x milliseconds.
- `begin-repeat` x : Repeats x number of times the commands following this command and before the next `end-repeat`. It is not possible to have another `begin-repeat` command before this loop is closed by a `end-repeat` command.
- `end-repeat` : Closes a repeat loop.

A field can be a character string or an object. The string can be written as any lower-case value written between quotes. e.g. "test". An object can be written similarly as a call to a object Constructor, e.g. MyClass(1,"a"). For simplicity, the constructor parameters may only be integer values and lower-case character strings. In the case of tuple fields that are objects, the `read` and `take` operators may either request a particular object written as a constructor call, any instance of

a particular type expressed as the name of the type, which will then in C# translate to an instance of `System.Type`), or any object (indicated by the constant `null`).

Both the `read` and `take` operators have a blocking semantics, i.e. the operation will not complete until there is a tuple in the tuple space that matches the operation wildcard.

In the case of a `read` or a `take`, a field might be a concrete value or a wildcard. Wildcards can be used to read or take from the tuple space any tuple that matches all of the wildcards on tuples can be expressed for both strings and objects. In the case of strings, the allowed wildcards are:

- Any string: any string matches this wildcard. It is written as "*".
- Initial substring: all strings starting with a given substring match this wildcard. It is written as "<substring>*" where <substring>is any string consisting of lower case letters.
- Final substring: all strings ending with a given substring match this wildcard. It is written as "*<substring>" where <substring>is any string consisting of lower case letters.

2.3 System Variants

As mentioned before, all groups must implement two variants of the tuple space: **DIDA-TUPLE-SMR** and **DIDA-TUPLE-XL**. Both variants maintain a set of servers where the tuples are stored and that can be contacted by any clients.

In the **DIDA-TUPLE-SMR** variant, the replication of operations should be based on the state machine replication model where all replicas start in an empty space, receive all commands in the same order and react deterministically to them. Students must implement the message delivery mechanism that ensures messages are received in the same order by all server processes.

In the **DIDA-TUPLE-XL** variant, the students should follow the implementation proposed by Xu and Liskov [2] (also described in the coursebook).

Groups should carefully explore the commonalities between both solutions so as to minimize the work needed to create the two requested variants and to allow to potentially add future variants.

2.4 Fault Tolerance

The reason for having multiple copies of the tuple space is to provide fault-tolerance. If a replica fails, tuples will be still available at other replicas. In the basic version of the project (see also the advanced features in Section 4) the students should assume that is is possible to implement a perfect failure detector. When a replica fails, all the other nodes are eventually informed of the failure and can update the view of active servers accordingly. Note that the algorithm of Xu and Liskov [2] waits for replies from *all* nodes that are active.

For this project, it is assumed that an arbitrary number of faults may occur but that only one fault may happen at each moment and that the system will have time to recover before the next fault. Obviously, the time required for that recovery will be taken into account in the project grading.

2.4.1 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 3 below).

3 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 will also execute a process, called PCS (Process Creation Service), which the PuppetMaster can use to launch operator processes on remote machines. Once a process is created, it may interact with the PuppetMaster either directly or through the local PCS. 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 process creation services on each machine should expose a service at an URL on port 10000 for the PuppetMaster, if it is running on a remote machine, to request the creation of a process on the local machine. For simplicity, we assume that the PuppetMaster knows the URLs of the entire set of process creation services. 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 min_delay max_delay*: This command creates a server process identified by *server_id*, available at *URL* and that delays any incoming message for a random amount of time 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. For instance, you may want to implement a total order layer to implement state-machine replication; in this case, messages sent to to the total order protocol can also be delayed. *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` *client_id URL script_file*: This command creates a client process identified by *client_id*, available at *URL* and that will execute the commands in the script file *script_file*.
- `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 centralized at the PuppetMaster.

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

- `Crash` *processname*. This command is used to force a process to crash.
- `Freeze` *processname*. 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` *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 and execute them either sequentially or step by step. A script file can have an additional command that controls 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.

4 Advanced Features

The students may way to implement the advanced features described below. These features are optional but have an impact on the final grade (see Section 9).

- A version of **DIDA-TUPLE-XL** that does not assume a perfect failure detection and that therefore is able to make progress even if only a majority of the servers respond to requests.
- A version of **DIDA-TUPLE-SMR** that does not assume a perfect failure detection and that therefore is able to make progress even if only a majority of the servers respond to requests.

5 Performance Evaluation

Each group should evaluate the performance of the different implementations. For groups that do not implement the advanced features, they should compare the two variants of the tuple, **DIDA-TUPLE-SMR** and **DIDA-TUPLE-XL**. Students that implement advanced version, should also include those version in the evaluation.

Note that **DIDA-TUPLE-SMR** and **DIDA-TUPLE-XL** may have different performance for different workloads. The students should identify the workloads for which each of the implementations performs best and design clients that test those situations.

For the students that implement the advanced version, it may happen that those version perform worse than the basic version. It is interesting to assess the performance penalty that you get (if any) by not assuming a perfect failure detector.

The students should discuss the results obtained 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 evaluation 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 present working implementations of the two base system variants. After the checkpoint, the students will have time to add additional fault tolerance mechanisms, 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

TODO: CONFIRM THE DATES

- November 9^{th} - Electronic submission of the checkpoint code;
- November 12^{th} to November 16^{th} - Checkpoint evaluation;
- December 7^{th} - Electronic submission of the final code.
- December 10^{th} - Electronic submission of the final report.

9 Grading

A perfect project without any of the advanced features will receive 16 points out of 20. Each advanced feature is worth 2 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 July 19, 2019, must be delivered by July 24, and will be discussed on July 25, 2019.

References

[1] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[2] A. Xu and B. Liskov. A design for a fault-tolerant, distributed implementation of linda. In *1989 The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers(FTCS)*, volume 00, pages 199–206.