ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
School of Computer and Communication Sciences | Distributed Programming Lab

CS-451 Bonus Project

A Storage System for Social Networks

$ Last update: September 25, 2015
$ Rev: 2

This document describes the requirements for the bonus project of the Distributed Algorithms class, Fall 2015. The project is optional, with a deadline on 23rd of December 2015, and it may result in a bonus of up to 1 point added to the student's final grade.

The goal of this project is to put in practice some of the topics discussed in this course, namely, the *Reliable Broadcast* and *Causal-ordering* algorithms.

Overview

We consider the problem of data storage for an online social network service. The data storage system has two design goals: to be reliable and to expose useful semantics.

First, to ensure reliability, the system will comprise three replicas, *i.e.* nodes, such that any node can read or write to the storage. All these three nodes replicate the storage system. To maintain their individual storage copies consistent, the nodes coordinate their actions through message passing.

Second, since the storage will serve as the backend for a social networking service, we require that all operations on the objects respect causality. Any semantic guarantee weaker than causality can be problematic (*e.g.*, with eventual consistency, there is no guarantee that users will be able to see their most recent posts or photos), but semantics stronger than causality are impractical because they are too expensive in practice. Thus, we focus on causal consistency, this model being a useful middle-ground between eventual consistency and strong consistency.

As a shorthand, we will call this data storage system *S4*: a student's simple storage service.

Deployment conditions

Messages exchanged between nodes may be dropped, delayed or reordered. Also, the execution on a node may be paused for an arbitrary amount of time and resumed later. A minority of nodes (in this case, one out the three) may also fail by crashing at an arbitrary point of its execution.

It is important that nodes don't crash simply due to unexpected or anomalous network conditions. A particular concern in this project is buffer sizes and queue lengths, which may lead to out-of-memory (OOM) errors. If such errors arise, then the implementation is incorrect. Thus, students should not rely on unbounded available memory, as it is assumed, for instance, in non-garbage-collected algorithms.

Technical specification

Nodes

The storage system implements a multi-writer multi-reader register – i.e., a key-value store – replicated at three nodes. Each node is represented by one Linux process. Process `n` is started by executing:

```
s4 addr-0 port-0 addr-1 port-1 addr-2 port-2 n f
```

where `addr-`*k* is the IP address of process *k* and port-*k* is the port on which process *k* listens for incoming network packets. The next parameter `n` $\in \{0, 1, 2\}$ is the ID of the process, and `f` is the name of the input file; the input file contains the execution trace which process *k* must run.

Typically, the input file for process *k* has the name *k*`.input`; similarly, the output file has the name *k*`.output`. We describe the structure of these files in sections *Input trace format* and *Output trace format*, respectively.

For example, to run all processes locally, each executing a trace from a specific file "*k*`.input`", we'll start them using the following commands:

```
s4 127.0.0.1 8900 127.0.0.1 8901 127.0.0.1 8902 0 0.input
s4 127.0.0.1 8900 127.0.0.1 8901 127.0.0.1 8902 1 1.input
s4 127.0.0.1 8900 127.0.0.1 8901 127.0.0.1 8902 2 2.input
```

Each process performs all necessary initialization tasks on startup, but it does not automatically start executing the trace from its input file. This allows time for all processes to start and initialize their state, otherwise some process might start executing before the others have the chance to finish their initialization routines. A process only starts executing its trace after receiving the `INT` signal.

To produce the crashing of a process we use a `KILL` signal, which no process can mask. It is safe to assume that at most one process crashes in an execution – i.e., only a minority of processes may fail. In addition, we also employ `STOP` and `CONT` signals to safely pause and subsequently resume the execution of processes; these two signals must not be masked by the processes.

We may send the following signals to a process:

- `kill -INT` *`PID`* – to make a process start the execution of its trace from the input file;
- `kill -KILL` *`PID`* – to make one of the processes crash;
- `kill -STOP` *`PID`* – to pause the execution of a process;
- `kill -CONT` *`PID`* – to resume the execution of a process.

Messages

Processes may be deployed on different machines in the network, so we must not assume any inter-process communication (IPC) primitives except the network channels. As a network communication primitive, the nodes can only rely on UDP; TCP or other reliable protocols are prohibited.

In terms of properties, UDP sockets are equivalent with fair-loss links, as presented in the lectures and the reference book.

Storage

All the stored objects in the system have a simple `key-value` representation, where the `key` is an identifier for the object, and `value` is the content of the object. For simplicity sake, we assume that both `keys` and `values` are short ASCII strings of at most 10 characters each.

S4 Template

We provide a template that demonstrates a design of S4 in Java. It contains the following classes:

The template uses Apache Maven as a management tool. On Ubuntu, install Maven with the command:

```
$ sudo apt-get install maven
```

- *App.java*: the main class, providing initialization and skeleton code to run the execution trace;
- *StoreMap.java*: provides the in-memory backend of the register, replicated at every node;
- *PointToPointLink.java*: a simple UDP sockets wrapper that simulates fair-loss links;
- *NodeInfo.java*: a representation of the IP:Port tuple identifying some node in the system.

These classes provide a simple code skeleton that may serve as a starting point for developing the S4 application. The template is available at:

`https://github.com/LPD-EPFL/da15-s4`

Compilation

All submitted applications will be tested using Ubuntu 14.04 running on a *64- bit* architecture. The submission has to contain all sources of the application. All submitted files are to be placed in a single folder hierarchy, such that executing `make` in that folder produces the necessary executable `s4`, which will be called by the testing scripts.

If a language other than Java is used, please indicate any prerequisite packages that need to be installed on the system in a file called *README.TXT*.

Testing and grading

Input trace format

The input file contains the execution trace, i.e., the workload, and has the following format:

```
operation,parameter1[,parameter2]
```

Sample input files can be found at: ➦ `https://github.com/LPD-EPFL/da15-s4/blob/master/0.input`, ➦ `https://github.com/LPD-EPFL/da15-s4/blob/master/1.input`, and ➦ `https://github.com/LPD-EPFL/da15-s4/blob/master/2.input`.

where `operation` $\in$ {`get`,`put`}, to read or write to the storage, respectively. `parameter1` is the key of an object, and `parameter2` is the value of an object. Note that `parameter2` is skipped if the `operation` is a `get`. For example, the following input specifies two write (`put`) operations and two subsequent read (`get`) operations:

```
put,A,1       /* write to object A value 1  */
put,B,2       /* write to object B value 2  */
get,A         /* read the value of object A */
get,B         /* read the value of object B */
```

Output trace format

An output trace contains the results of all the `get` operations specified in the given input file, in the form of `key,value`, one per line. For instance, given the earlier input with four operations, the matching output would contain only two lines, and it could look as follows:

We consider that `put` operations return a NULL ($\perp$) value, e.g., `void` in Java; the output file must not reflect these values.

`.input`	`.output`
`put,A,1`	$\perp$ (ignore in the output file)
`put,B,2`	$\perp$ (ignore in the output file)
`get,A`	`A,1`
`get,B`	`B,2`

Table 1: Sample input and output.

Testing criteria

All testing is performed based on the content of the output file. There are two testing criteria: *correctness* and *performance*.

The submitted applications will be first tested for correctness under various conditions, such as packet loss, reordering, network delays, simulated asynchrony of processes, crashes, etc. An application passes the correctness criteria if it runs the execution trace from the input file and prints a correct output to the output file. We consider an output correct if the values for all the `get` operations on every object in the output respect causal consistency.

Informally, causal consistency requires that all replicas agree on the ordering of causally-related operations; operations that are not causally-related may appear in different order at different processes.

To establish which operations are causally related, we can rely on the causal-order property given in the class or defined in the book:

> We say that an operation op_1 causally precedes another operation op_2, denoted as $op_1 \longrightarrow op_2$, if any of the following three cases applies:
>
> (a) **FIFO**: A process invokes op_1 and then invokes op_2 (see Figure 1);
>
> (b) **Local**: A process invokes the `put` operation op_1 and another process invokes the `get` operation op_2, where op_2 observes the written value of op_1 (see Figure 2); or
>
> (c) **Transitivity**: There exists an intermediate operation op' such that $op_1 \longrightarrow op'$ and $op' \longrightarrow op_2$ (see Figure 3).

The following figures depict the three possible cases for establishing causal precedence among two operations $op_1 \longrightarrow op_2$; dotted arrows highlight causal dependencies:

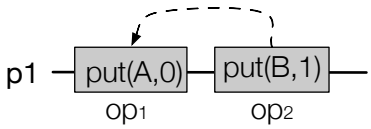

Figure 1: Causal dependencies, case (a).

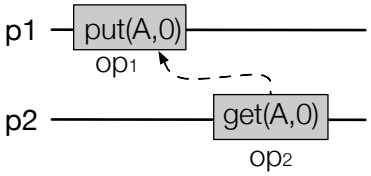

Figure 2: Causal dependencies, case (b).

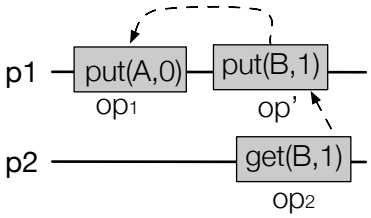

Figure 3: Causal dependencies, case (c).

In Figure 4 (below) and Figure 5 (next page) we depict an execution which does not respect causal consistency and an execution which respects causal consistency, respectively.

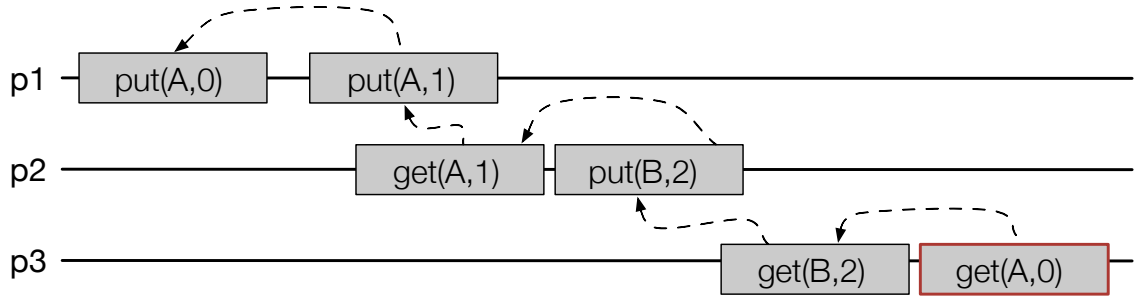

Figure 4: An execution that does not satisfy causal consistency. The last operation on process `p3` should return `(A,1)` in order to satisfy all causal dependencies.

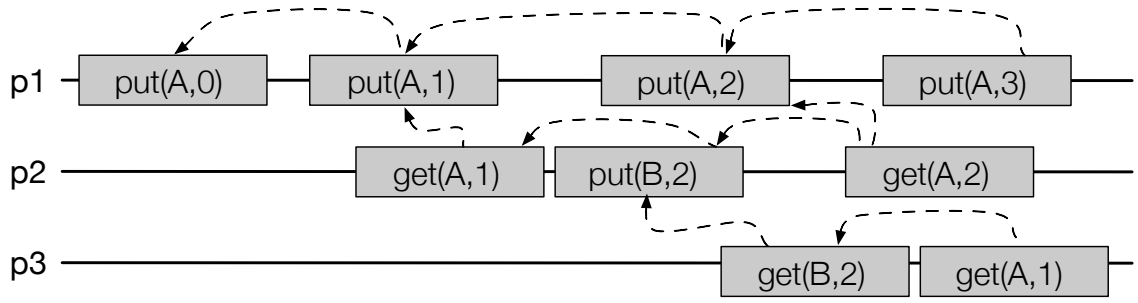

Figure 5: An execution which satisfies causal consistency.

If an implementation passes all correctness tests, it will be tested for performance, which we measure in terms of throughput. When testing performance, we will not introduce any artificial network anomalies (such as packet loss or reordering) or process delays/crashes.

Bonus points

In accordance with the testing criteria, a project may pay off in one out of two ways. First, all correct implementations receive a 0.5 bonus points on top of their final grade. Second, the five correct implementations which are also the fastest ones – in terms of throughput – will be rewarded with an additional bump of 0.5, resulting in a bonus of 1 point to the student's final grade.

Obviously, no bonuses apply beyond a grade of 6.

Cooperation

This project is meant to be completed individually. Copying of other students' solutions is prohibited. You are free to discuss the projects with others, but the submitted source codes must be your own work. Multiple copies of the same code will be disregarded without investigating which is the "original" and which is the "copy".

Further reading

- Reference book: Cachin, C., Guerraoui, R., & Rodrigues, L. (2011). *Introduction to reliable and secure distributed programming.* Springer Science & Business Media
- Ahamad, M., Neiger, G., Burns, J. E., Kohli, P., & Hutto, P. W. (1995). *Causal memory: definitions, implementation, and programming.* Distributed Computing, 9(1), 37–49. doi:10.1007/BF01784241