Distributed Algorithms 2019/20
Practical Project

Implementation of Broadcast Algorithms

Distributed Computing Lab – EPFL

September 23, 2019

1 Overview

The goal of this practical project is to implement certain building blocks necessary for a decentralized system. To this end, some underlying abstractions will be used:

1. Perfect Links,
2. Uniform Reliable Broadcast,
3. FIFO Broadcast (submission #1),
4. Localized Causal Broadcast (submission #2)

Various applications (e.g., a payment system) can be built upon these lower-level abstractions. We will check your submissions (see Section 7) for correctness and performance as part of the final evaluation (see Section 5).

The implementation must take into account that messages exchanged between processes may be dropped, delayed or reordered by the network. The execution of processes may be paused for an arbitrary amount of time and resumed later. Processes may also fail by crashing at arbitrary points of their execution.

2 Technical specification

2.1 Processes

One process is represented by one Linux process. Process `n` is started by executing:

```
da_proc n membership [extra_params...]
```

where `n` $\in \{1, 2, 3, 4, 5, \ldots\}$ is the ID of the process and `membership` is the name of the membership file. `extra_params` are extra parameters specific for each algorithm; we discuss these parameters in more detail later (Section 7).

The membership file describes the system membership, i.e., the identities of processes that participate in the protocol. The first line of this file contains the number `n` of processes in the system. Subsequently, each line of this file contains one process identity per line. The remaining lines of this file are algorithm-dependent (may be empty), and we discuss this later in detail in Section 7.

A process identity consists of a numerical process identifier, the IP address of the process and the port number on which the process listens to incoming messages. The entries of each process identity are separated by at least one white space character. The following is an example of the contents of a membership file:

```
5
1 127.0.0.1 11001
2 127.0.0.1 11002
3 127.0.0.1 11003
4 127.0.0.1 11004
5 127.0.0.1 11005
(... algorithm-dependent data ...)
```

A process performs all necessary initialization tasks on startup, but it does not automatically start sending / broadcasting messages. This enables all processes to start and initialize. A process only starts broadcasting messages after receiving the USR2 signal.

A process that receives a TERM or INT signal must immediately stop its execution with the exception of writing to an output log file (see Section 2.3). In particular, it must not send or handle any received network packets. This is used to simulate process crashes. You can assume that at most a minority (e.g., 1 out of 3; 2 out of 5; 4 out of 10, ...) processes may crash in one execution.

2.2 Messages

Inter-process point-to-point messages (at the low level) must be carried exclusively by UDP packets in their most basic form, not utilizing any additional features (e.g., any form of feedback about packet delivery) provided by the network stack, the operating system or external libraries. Everything must be implemented on top of these low-level point to point messages.

The application messages (i.e., those broadcast by processes) are numbered sequentially at each process, starting from 1. Thus, each process broadcasts messages 1 to `m`. By default, the payload carried by an application message is only the sequence number of that message.

2.3 Output format

The output of each process is a text file. For process `n`, the text file is named `da_proc_n.out` and contains a log of events. Each event is represented by one line of the output file, terminated by a Unix-style line break (`'\n'`). There are two types of events to be logged:

- broadcast of application message, using the format
  `b` *`seq_nr`*
  where *`seq_nr`* is the sequence number of the message

- delivery of application message, using the format
  `d` *`sender seq_nr`*
  where *`sender`* is the number of the process that broadcast the message and *`seq_nr`* is the sequence number of the message (as numbered by the broadcasting process).

An example of the content of an output file:

```
b 1
b 2
b 3
d 2 1
d 4 2
b 4
```

Note: The most straight-forward way of logging the output is to append a line to the output file on every broadcast or delivery event. However, this may harm the performance of the implementation. You might consider more sophisticated logging approaches. Also note that even a crashed process needs to output the sequence of events that occurred before the crash. You can assume that a process crash will be simulated only by the TERM or INT signals. Remember that writing to files is the only action we allow a process to do after receiving a TERM or INT signal.

3 Compilation

All submitted implementations will be tested using Ubuntu 14.04 running on a 64-bit architecture. The submission has to contain all sources of the implementation. All submitted files are to be placed in one folder, such that executing `make` in that folder produces all necessary executables which will be called by the testing scripts. We expect all implementations to be in either C/C++ or Java.

You are **strongly encouraged** to test the compilation of your code in the virtualbox VM (see https://dcl.epfl.ch/site/education/da). For validating your submission, we will upload your submitted code to our test machine (similar to the VM) and **only** run the **original** `validate.sh` (any modifications to this script will be discarded). This script executes `make` to obtain the executable `da_proc` (or `Da_proc.class`). Any submissions that fail to pass this test will be completely ignored!

4 Template

A simple C template is provided that shows a possible high-level structure of the implementation. The file `da_proc.c` contains a simple code skeleton that may serve as a starting point for developing. Along with `validate.sh` (see Section 3) we provide a shell script (`test_performance.sh`) to give you an estimate of how performance will be tested.

5 Testing and grading

The submitted implementations will be first tested for correctness under various conditions (packet loss, reordering, delays, simulated asynchrony of processes, crashes, etc...).

If they pass all correctness tests (i.e. if the resulting output logs are consistent with the definition of the corresonding broadcast abstraction), they will be tested for performance in terms of throughput, i.e. the total number of messages delivered by all processes per second. When testing throughput, no artificial network artifacts (such as packet loss or reordering) or process delays/crashes will be simulated.

6 Cooperation

This project is meant to be completed in teams of 2 to 3 people. Please submit your team preference (see https://dcl.epfl.ch/site/education/da).

Copying of other teams' solutions is prohibited. You are free (and encouraged) to discuss the projects with other teams, but the submitted source code must be the exclusive work of your own team. Multiple copies of the same code will be disregarded without investigating which is the "original" and which is the "copy". Note: code similarity tools will be used to check copying.

7 Submissions

This project comprises two submissions:

1. A runnable application implementing FIFO Broadcast, and
2. A runnable application implementing Localized Causal Broadcast.

Note that these submissions are *incremental*. This means that your work towards the first will help you in your work towards the second. We are only interested in the FIFO and Localized Causal broadcast algorithms.

We define several details for each algorithms below.

7.1 FIFO Broadcast application

- You must implement this on top of uniform reliable broadcast (URB).
- The `extra_params` command-line argument for this algorithm consists of an integer `m`, which defines how many messages each process should broadcast.
- We do not specify any algorithm-dependent data at the end of the `membership` file for this algorithm.

7.2 Localized Causal Broadcast

- The `extra_params` command-line argument for this algorithm consists of an integer `m`, which defines how many messages each process should broadcast.

- The algorithm-dependent data at the end of the `membership` file for this algorithm consists of `n` lines. Each line *i* corresponds to process *i*, and such a line indicates the identities of other processes which can affect process *i*. See the example below.

- The FIFO property still needs to be maintained by localized causal broadcast. That is, messages broadcast by the same process must not be delivered in a different order then they were broadcast.

- The output format for localized causal broadcast remains the same as before, i.e., adhering to the description in Section 2.3.

Example of `membership` file:

```
5
1 127.0.0.1 11001
2 127.0.0.1 11002
3 127.0.0.1 11003
4 127.0.0.1 11004
5 127.0.0.1 11005
1 4 5
2 1
3 1 2
4
5 3 4
```

In this example we specify that process `1` is affected by messages broadcast by processes `4` and `5`. Similarly, we specify that process `2` is only affected by process `1`. Process `4` is not affected by any other processes. Process `5` is affected by processes `3` and `4`.

We say that a process `x` is affected by a process `z` if all the messages which process `z` broadcasts and which process `x` delivers become dependencies for all future messages broadcast by process `x`. We call these dependencies *localized*. If a process is not affected by any other process, messages it broadcasts only depend on its previously broadcast messages (due to the FIFO property).

Note: In the default causal broadcast (this algorithm will be discussed in one of the classes) each process affects *all* processes. In this algorithm we can selectively define which process affects some other process.