Distributed algorithms - Project description

Nicolas Wagner, Diana Petrescu, Rishi Sharma

October 7, 2024

Contents

1 Abstraction blocks for decentralized systems

The goal of this practical project is to implement certain building blocks necessary for a decentralized system, to ensure reliable message delivery to all participants in the system. To this end, some underlying abstractions will be used, whose interfaces are listed in the next subsections. Interface's sections are guidelines on what you should implement and what properties it should respect to fulfill correctness. It is not supposed to be teaching material, if you need more explanation please refer to the lectures or the excellent book Introduction to reliable and secure distributed programming (freely available online for EPFL students).

1.1 System model

The description of the properties and assumptions we take for the system in which you have to implement the abstractions, is as follow:

Processes. We assume a static system of $\Psi = \{P_1, ..., P_n\}$ of n processes.

Failures. Among $n = 2f + 1$ processes, at most f processes can fail by crashing. A process that fails is said to be faulty (e.g., a non-faulty process is correct), they stop executing instructions/actions.

Asynchrony. The processes are asynchronous: a process proceeds at its own arbitrary (and non-deterministic) speed. Moreover, the communication network is also asynchronous: message delays are finite, but arbitrarily big.

Communication. Processes communicate by exchanging messages over an authenticated point-to-point network. In addition to the communication being asynchronous, messages can also be lost, delayed or reordered.

1.2 Perfect Links

The first abstraction is perfect links, its specification is written in Interface 1. This interface consists of two events: a request event (for sending messages) and an indication event (for delivering messages), note that the request event consist of sending a message to a **single** process.

Interface 1 Perfect links

Module:

Name: Perfect Links, **instance** *pl*.

Events:

Request: ⟨*pl*, Send | q, m⟩: Sends message m to process q.

Indication: ⟨*pl*, Deliver | p, m⟩: Delivers message m sent by process p.

Properties:

PL1: *Reliable delivery:* If a correct process p sends a message m to a correct process q, then q eventually delivers m.

PL2: *No duplication:* No message is delivered by a process more than once.

PL3: *No creation:* If some process q delivers a message m with sender p, then m was previously sent to q by process p.

1.3 FIFO Broadcast

The second abstraction is FIFO-order uniform reliable broadcast, its specification is written in Interface 2. It is obtained from the uniform reliable broadcast specification extended with the FIFO delivery property. This interface consists of two events: a request event (for broadcasting messages) and an indication event (for delivering messages), note here that the request event consist of sending a message to **all** process **including** itself.

Interface 2 FIFO-order uniform reliable broadcast

Module:

Name: FIFO broadcast, **instance** *frb*.

Events:

Request: $\langle frb,$ Send | m$\rangle$: Broadcasts a message m to all processes.

Indication: $\langle frb,$ Deliver | p, m$\rangle$: Delivers a message m broadcast by process p.

Properties:

FRB1: *Validity:* If a correct process p broadcasts a message m, then p eventually delivers m.

FRB2: *No duplication:* No message is delivered more than once.

FRB3: *No creation:* If a process delivers a message m with sender s, then m was previously broadcast by process s.

FRB4: *Uniform agreement:* If a message m is delivered by some process (whether correct or faulty), then m is eventually delivered by every correct process.

FRB5: *FIFO delivery:* If some process broadcasts message m1 before it broadcasts message m2, then no correct process delivers m2 unless it has already delivered m1.

1.4 Lattice Agreement

The last abstraction is multi-shot lattice agreement, its specification, for the single-shot version, is written in Interface 3. Let $\mathcal{V}$ denote the set of values, the lattice agreement problem allows processes to agree on "similar" decisions despite failures. The proposal of a process P_i is denoted by I_i, whereas the decision of a process P_i is denoted by O_i.

Here is a description of lattice agreement's properties. The validity property states that (1) the decided set must include the proposal set, and (2) the decided set includes the proposals of other processes (i.e., the decided set cannot include values which were not proposed). Consistency claims that decided values must be comparable. Finally, termination states that all correct processes eventually decide.

1.4.1 Algorithm's pseudocode

The algorithm considers two roles: proposers and acceptors. Every process plays both roles, the separation is included solely for simplicity and clarity. This pseudocode is made to solve a single instance of the lattice agreement problem. but your goal is to implement a version that can handle multiple instances of the lattice agreement problem in parallel.

However, your goal is to extend it to handle multiple instances of the lattice agreement problem in parallel. Refer to

Interface 3 Lattice Agreement

Module:

Name: Lattice Agreement, **instance** *la*.

Events:

Request: $\langle la,\ \text{Propose} \mid I_i \rangle$: A process proposes a set $I_i \subseteq \mathcal{V}$.

Indication: $\langle la,\ \text{Decide} \mid O_i \rangle$: A process decides a set $O_i \subseteq \mathcal{V}$.

Properties:

LA1: *Validity:* Let a process P_i decide a set O_i, then $I_i \subseteq O_i$ and $O_i \subseteq \bigcup_{j \in [1,n]} I_j$.

LA2: *Consistency:* Let a process P_i decide a set O_i and let a process P_j decide a set O_j, then $O_i \subseteq O_j$ or $O_i \supset O_j$.

LA3: *Termination:* Every correct process eventually decides.

Algorithm 1 Lattice Agreement Algorithm: Pseudocode of proposer for P_i

1: **upon** init:
2: Boolean $active_i = false$
3: Integer $ack\_count_i = 0$
4: Integer $nack\_count_i = 0$
5: Integer $active\_proposal\_number_i = 0$
6: Set $proposed\_value_i = \perp$
7: **upon** propose(Set *proposal*):
8: $proposed\_value_i \leftarrow proposal$
9: $active_i \leftarrow true$
10: $active\_proposal\_number_i \leftarrow active\_proposal\_number_i + 1$
11: $ack\_count_i \leftarrow 0$
12: $nack\_count_i \leftarrow 0$
13: **trigger** $beb.broadcast(\langle \texttt{proposal}, proposed\_value_i, active\_proposal\_number_i \rangle)$
14: **upon** reception of $\langle \texttt{ack}, \text{Integer } proposal\_number \rangle$ **such that** $proposal\_number = active\_proposal\_number_i$:
15: $ack\_count_i \leftarrow ack\_count_i + 1$
16: **upon** reception of $\langle \texttt{nack}, \text{Integer } proposal\_number, \text{Set } value \rangle$ **such that** $proposal\_number = active\_proposal\_number_i$:
17: $proposed\_value \leftarrow proposed\_value \cup value$
18: $nack\_count_i \leftarrow nack\_count_i + 1$
19: **upon** $nack\_count_i > 0$ **and** $ack\_count_i + nack\_count_i \geq f + 1$ **and** $active_i = true$:
20: $active\_proposal\_number_i \leftarrow active\_proposal\_number_i + 1$
21: $ack\_count_i \leftarrow 0$
22: $nack\_count_i \leftarrow 0$
23: **trigger** $beb.broadcast(\langle \texttt{proposal}, proposed\_value_i, active\_proposal\_number_i \rangle)$
24: **upon** $ack\_count_i \geq f + 1$ **and** $active_i = true$:
25: **trigger** decide($proposed\_value_i$)
26: $active_i \leftarrow false$

Algorithm 2 Lattice Agreement Algorithm: Pseudocode of acceptor for P_i

1: **upon** init:
2: Set $accepted\_value_i = \bot$
3: **upon** reception of $\langle$`proposal`, Set $proposed\_value$, Integer $proposal\_number\rangle$ from proposer P_j **such that** $accepted\_value_i \subseteq proposed\_value$:
4: $accepted\_value_i \leftarrow proposed\_value$
5: send $\langle$`ack`, $proposal\_number\rangle$ to P_j
6: **upon** reception of $\langle$`proposal`, Set $proposed\_value$, Integer $proposal\_number\rangle$ from proposer P_j **such that** $accepted\_value_i \not\subseteq proposed\_value$:
7: $accepted\_value_i \leftarrow accepted\_value_i \cup proposed\_value$
8: send $\langle$`nack`, $proposal\_number, accepted\_value_i\rangle$ to P_j

2 Project

The description is quite long and there is a lot of information to digest, we suggest you take your time reading it. Before starting, you should take note of the following important points:

- **You are expected to be fluent in Java, C or C++ for this project.** Don't forget, there are plenty of great resources easily available online. We can try to help you, but since implementing the abstractions is the purpose of this project, we might not be able to answer you without giving you the solution. If you are not sure which language to choose, you should pick the one you are the more comfortable with. This choice will **not** impact your grade.
- **Concurrent programming can be quite complicated and error-prone**. We advise you to not start the project last minute, as it can take you a substantial amount of time.
- **Testing your implementation is as important as your implementation itself.** We want to emphasize that testing is supposed to be a big part of the project, you should not underestimate it.

2.1 Practical constraints

Languages In order to have a fair comparison among implementations, as well as provide support to students, the project must be developed using the following tools. You have the choice between:

- C11 and/or C++17, with build system CMake
- Java, with build system Maven

Additionally, you are **not** allowed to use any 3rd party libraries in your project. C++17 and Java 11 come with an extensive standard library that can easily satisfy all your needs, including UDP sockets which you will need. However, you are allowed to use code from the internet (e.g., a publicly available skip list), as long as you give appropriate credit.

Projects that fail to compile with the following configuration will not be considered. Submitted implementations will be tested using Ubuntu 18.04 running on a 64-bit architecture. These are the specific versions of tool chains where your project will be tested upon:

- gcc (Ubuntu 7.5.0-3ubuntu1 18.04) 7.5.0
- g++ (Ubuntu 7.5.0-3ubuntu1 18.04) 7.5.0
- cmake version 3.10.2
- OpenJDK Runtime Environment (build 11.0.8+10-post-Ubuntu-0ubuntu118.04.1)
- Apache Maven 3.6.3 (cecedd343002696d0abb50b32b541b8a6ba2883f)

VM You are free to set up your own development environment if you know what you are doing. But in the case that you are not comfortable with it, we provide you with a virtual machine, which include the exact versions of the build tools mentioned previously.

To use the VM Image, you need to install VirtualBox in your computer. The installation process is straight-forward. The provided images were created with VirtualBox 6.1, please make sure you use a recent version of VirtualBox. There are several variants to the image:

- For Windows/Linux/Mac machines with Intel/AMD CPUs, use this image. It also includes GuestAdditions, it allows for easy transfer (drag and drop) of files between the VM and the HOST, as well as copying from clipboard between the VM and HOST. You can enable these features from Menu→Devices→Shared Clipboard and Menu→Devices→Drag and Drop.
- For Mac machines with ARM CPUs, use this image. Additionally, install UTM.

The VM's credentials are username *dcl* and password *dcl*. We provide support for the VM, feel free to ask if you have any issue or question.

Hardware limits Naturally, you don't have an unlimited amount of resources available, which is one of the challenge of this project. Therefore, we enforce certain limitations during testing. First, you are given 2 CPU cores and 4 GiB RAM. In addition to that, we impose a global limit of 8 threads per process (e.g., if the test is of 10 process, you are allowed to use up to 80 threads in total). It is essential that you take these limitations into account when developing your project. Projects that are not well engineered may fail some tests (or even not get any points) due to them. Finally, there are two more limitations in places: your output files cannot exceed 64 MiB, and your console output files will be trimmed if they exceed 1 MiB.
The upper bound for the number of processes is 128 and the number of messages is `MAX_INT`.

Asynchrony limits As we defined previously, the entire project implements abstractions that operate in the asynchronous model. However, this is not interesting in practice, as this would mean, you could just provide an empty submission, and it would fulfill correctness. To prevent this, during the evaluation of your project, we set a maximum execution time limit. This time is defined by assuming that every message sent via UDP takes at most 1 second to be delivered. We adjust this number accordingly when the network and processes are not well behaving (messages delayed, lost, ...). In the context of this project, **this is considered as a huge upper bound**, if you have to worry about this limitation, you should probably review your implementation.

2.2 Code structure

We provide you with a template for both C/C++ and Java, it is mandatory to use the template in your project. The template already includes some source code under src, that will help you with parsing the arguments provided to the executable. You are **not** allowed to change any of the file names or the function signatures in this template, more specifically, don't make changes when you see the message "*Do not edit this section!*". You shouldn't edit any files outside the src directory, as your changes will be overwritten during the evaluation.

You can run:

- `build.sh` to compile your project.
- `run.sh` <`arguments`> to run your project.
- `cleanup.sh` to delete the build artifacts. Run this command when submitting your project for evaluation.

You should place your source code under the src directory, and you can arrange and structure those project files as you see fit. Yet, we encourage modular design and usual best practices. As this will make your code cleaner, the project easier and help you spot bugs.

Finally, your executable should **not** create/use directories named "deploy" and/or "logs" in the current working directory. These directories are reserved for evaluation!

Java The code structure is:

```
.
├── bin
│   ├── deploy # Do not use in executable
│   │   └── README
│   ├── logs # Do not use in executable
│   │   └── README
│   └── README
├── build.sh
├── cleanup.sh
├── pom.xml # Do not edit
├── run.sh
└── src
    └── main
        └── java
            └── cs451 # Your code goes here
                └── ...
```

C/C++ The code structure is:

```
.
├── bin
│   ├── deploy # Do not use in executable
│   │   └── README
│   ├── logs # Do not use in executable
│   │   └── README
│   └── README
├── build.sh
├── cleanup.sh
├── CMakeLists.txt # Do not edit
├── run.sh
└── src # Your code goes here
    ├── CMakeLists.txt # Do not edit
    └── ...
```

2.3 General instructions

The following sections are concepts that are valid for all milestones of the project. Take your time reading them, as they are important for the understanding of the project.

2.3.1 Communication

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. Everything you need is included in the standard libraries.

Each process should only use 1 port to receive messages, therefore should only use a single UDP socket too. With regard to sending messages, it is usually the case that the UDP socket will assign a port automatically from available ones, we tolerate this behavior.

The messages exchanged by processes contain a payload, which differs based on the submission. For the first two submission, the payload is an integer number. For the third submission, it is a list of integer numbers. Apart from the payload, the exchanged messages should also contain other metadata information which is necessary to implement the required abstractions. Your implementation must take into account that messages exchanged between processes may be dropped, delayed or reordered by the network.

2.3.2 Signals

We simulate process crashes by relying on Linux's signals, processes may fail at arbitrary points of their execution. A process that receives a `SIGTERM` or `SIGINT` signal must immediately stop its execution, except for writing to an output log file. In particular, it must not send or handle any received network packets. You can assume that a process crash will be simulated only by the `SIGTERM` or `SIGINT` signals. The execution of processes may also be paused for an arbitrary amount of time and resumed later (using signals `SIGSTOP` and `SIGCONT`).

But don't worry, you don't have to handle signals directly, the templates provided include code that does that for you. In Java, we use a *shutdownHook* and in C/C++, we use the *signal* function. Those will allow you to still run code before the process stops completely. In particular, you should stop and close any network activity used and log all values to output files, as explained in the next section.

2.3.3 Logging

In all milestones, you have to produce an output file per process, and we will use them to evaluate your implementation.

An output file contains a log of send/receive events for milestone 1 and 2, or decisions events for milestone 3. Each event is represented by one line of the output file, terminated by a Unix-style line break $\backslash n$. These are the events to be logged:

Milestone 1 & 2

- Sending (milestone 1) / Broadcasting (milestone 2) of an application message, using the format `b seq_nr`, where `seq_nr` is the sequence number of the message. These messages are numbered sequentially at each process, **starting from 1**.
- Delivery of an application message, using the format `d sender seq_nr`, where `sender` is the id of the process that sent the message and `seq_nr` is the sequence number of the message (as numbered by the sending process).

Milestone 3

- Decision of the process for a single instance of lattice agreement. The order of the lines in the output file must be the same as the order of the lines (proposals) in the config file. Given that proposals are a set of integers, so are decisions. Thus, a decision should contain integers separated by single spaces (and terminated by a Unix-style line break $\backslash n$).

Note 1: For milestone 1 and 2, the payload carried by an application message is only the sequence number of that message. Though the payload is known in advance, your implementation should not utilize this information. In other words, your implementation should be agnostic to the contents of the payload. For example, your implementation should work correctly if the payload is arbitrary text instead of sequential numbers. In addition, your implementation should not rely on the fact that the total number of messages (to be broadcast) is known in advance, i.e., your implementation should work correctly if messages arrive as a stream.

Note 2: 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 may consider more sophisticated logging approaches, such as storing the logs in memory and periodically write them to the output file. Also note that even a crashed process needs to output the sequence of events that occurred before the crash. Remember that writing to files is the only action we allow a process to do after receiving a SIGINT or SIGTERM signal.

Note 3: Be especially careful to respect this format for output files. For example, you should not have unfinished lines (i.e., `b` or `d 1` are unfinished and could happen if you are writing lines character by character). You should not have duplicate events, as events are unique and should be logged only once (e.g., you should

not log what happens at the UDP level). Output files that don't respect the format will be discarded and considered as empty.

2.3.4 Application interface

The templates provided come with a command line interface (CLI) that you should use for your deliverables. The implementation for the CLI is given to you for convenience.

The supported arguments are:

```
./run.sh --id ID --hosts HOSTS --output OUTPUT CONFIG
```

Where:

- `ID` specifies the unique identifier of the process. In a system of n processes, the identifiers are $1...n$.
- `HOSTS` specifies the path to a file that contains the information about every process in the system, i.e., it describes the system membership. The file contains as many lines as processes in the system. A process identity consists of a numerical process identifier, the IP address or name of the process and the port number on which the process is listening for incoming messages. The entries of each process identity are separated by a single space character. The following is an example of the contents of a `HOSTS` file for a system of 5 processes:

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

- `OUTPUT` specifies the path to a text file where a process stores its output. The formatting of the output text file depends on the submission.
- `CONFIG` specifies the path to a file that contains additional information for the experimented abstraction (e.g., how many messages to broadcast).

2.3.5 Tools

For your convenience, we provide a tool to stress test your implementation. You can edit it under *tools/stress.py* in order to test different scenarios. But this test does **not** cover everything nor is suitable to completely validate your implementation. You **should** still create your own tests, inspiring yourself from this one, if you want.

Depending on the milestone, here are the appropriate commands to run it:

```
./stress.py perfect -r RUNSCRIPT -l LOGSDIR -p PROCESSES -m MESSAGES
./stress.py fifo -r RUNSCRIPT -l LOGSDIR -p PROCESSES -m MESSAGES
./stress.py agreement -r RUNSCRIPT -l LOGSDIR -p PROCESSES -n PROPOSALS
    -v PROPOSAL_MAX_VALUES -d PROPOSALS_DISTINCT_VALUES
```

Where:

- `RUNSCRIPT` is the path to run.sh. Remember to build your project first!
- `LOGSDIR` is the path to a directory where stdout/stderr and output of each process will be stored. It also stores generated `HOSTS` and `CONFIG` files. The directory must exist as it will not be created for you.
- `PROCESSES` (for perfect, fifo and agreement) specifies the number of processes spawn during validation.
- `MESSAGES` (for perfect and fifo) specifies the number of messages each process is broadcasting.
- `PROPOSALS` (for agreement) specifies the number of proposals each process is proposing.

- `PROPOSAL_MAX_VALUES` (for agreement) specifies the maximum size of the proposal set, i.e., the maximum number of integers that each proposal contains.
- `PROPOSALS_DISTINCT_VALUES` (for agreement) specifies the number of distinct elements across all proposals of all processes, i.e., it specifies the maximum size of the union of all proposals.

You can edit *testConfig* at the bottom of *stress.py* in order to test different scenarios.

```
testConfig = {
    'concurrency' : 8, # How many threads are interfering with the
       running processes.
    'attempts' : 8, # How many successful operations (SIGCONT,
       SIGSTOP, SIGTERM) each thread will attempt before
       stopping. Threads stop if a minority of processes has been
       terminated.
    'attemptsDistribution' : { # Each thread selects a process
       randomly and issues one of these operations with the given
       probability.
        'STOP': 0.48,
        'CONT': 0.48,
        'TERM':0.04
    }
}
```

To change the rate at which interfering threads interfere with the processes, modify this line, under the *StressTest* class:

```
time.sleep(float(random.randint(50, 500)) / 1000.0)
```

To introduce delay, loss or reordering in communications, you can use the *tc* Linux utility to alter network characteristics (man page). As an example:

```
sudo tc qdisc add dev lo root netem 2>/dev/null
sudo tc qdisc change dev lo root netem delay 200ms 50ms loss 10% 25% reorder 25% 50%
```

What it does:

- Adds packet delay of 200ms ± 50ms [150ms-250ms] that is normally distributed.
- 10% of the packets are lost. Each successive probability depends on 25% of the last one: P(n) = 0.25 * Prob(n-1) + 0.75 * Random (e.g., Emulated burst losses)
- 25% of packets (with a correlation of 50%) will get sent immediately, others will be delayed by the aforementioned delay factor

For convenience, *tools/tc.py* applies the TC commands previously mentioned, so you only have to run:

```
python3 tc.py
```

The tc rules are applied for as long as this program is running. When this program exits, the tc rules are reset. So you should run *tc.py* before *stress.py*. To change the TC configuration, you can edit *tools/tc.py*:

```
config = {
  'delay': ('200ms', '50ms'),
  'loss': ('10%', '25%'),
  'reordering': ('25%', '50%')
}
```

2.4 Milestone 1: Perfect Links

In this application, a set of processes exchange messages using perfect links. In particular, a single process only receives messages while the rest of processes only send messages. The communication of every sender with the receiver is realized using the perfect links abstraction. Refer to Interface 1.

Config The config file contains two integers m i in its first line. The integers are separated by a single space character. m defines how many messages each sender process should send. i is the index of the receiver process. The receiver process only receives messages while sender processes only send messages. All $n-1$ sender processes, send m messages each. Sender processes send messages 1 to m in order. Example of 100 messages with 3 processes:

```
100 3
```

Example An example of the content of output files with the previous config file (ellipsis are not included and denotes skipped content)

Process 1:
```
b 1
b 2
b 3
b 4
b 5
b 6
...
```

Process 2:
```
b 1
b 2
b 3
b 4
b 5
b 6
...
```

Process 3:
```
d 1 1
d 1 2
d 2 2
d 2 1
d 1 3
d 2 3
...
```

Manual Execution The following example builds and starts 3 processes (run from within the *template_cpp* or the *template_java* directory):

```
# Build the application:
./build.sh

# In first terminal window:
./run.sh --id 1 --hosts ../example/hosts --output
   ../example/output/1.output ../example/configs/perfect-links.config

# In second terminal window:
./run.sh --id 2 --hosts ../example/hosts --output
   ../example/output/2.output ../example/configs/perfect-links.config

# In third terminal window:
./run.sh --id 3 --hosts ../example/hosts --output
   ../example/output/3.output ../example/configs/perfect-links.config

# Wait enough time for all processes to finish processing messages.
# Type Ctrl-C in every terminal window.
```

2.5 Milestone 2: FIFO broadcast

Refer to Interface 2 for the FIFO broadcast abstraction. Informally, every process is both broadcasting and delivering messages from every other process (including itself). You must implement FIFO broadcast on top of Uniform Reliable Broadcast (URB).

Config The config file contains an integer m in its first line. m defines how many messages each process should broadcast. Processes broadcast messages 1 to m in order. Example of broadcasting 100 messages with 3 processes:

```
100
```

Example An example of the content of output files with the previous config file (ellipsis are not included and denotes skipped content)

Process 1:
```
b 1
d 3 1
d 3 2
b 2
d 2 1
d 1 1
d 1 2
...
```

Process 2:
```
b 1
b 2
d 2 1
d 2 2
d 3 1
d 3 2
d 1 1
...
```

Process 3:
```
b 1
d 1 1
d 1 2
b 2
d 2 1
d 3 1
d 3 2
...
```

Manual Execution The following example builds and starts 3 processes (run from within the *template_cpp* or the *template_java* directory):

```
# Build the application:
./build.sh

# In first terminal window:
./run.sh --id 1 --hosts ../example/hosts --output
   ../example/output/1.output ../example/configs/fifo-broadcast.config

# In second terminal window:
./run.sh --id 2 --hosts ../example/hosts --output
   ../example/output/2.output ../example/configs/fifo-broadcast.config

# In third terminal window:
./run.sh --id 3 --hosts ../example/hosts --output
   ../example/output/3.output ../example/configs/fifo-broadcast.config

# Wait enough time for all processes to finish processing messages.
# Type Ctrl-C in every terminal window.
```

2.6 Milestone 3: Lattice Agreement

Lattice Agreement is strictly weaker than consensus, as it can be solved in the asynchronous model. The formal specification, as well as an algorithm that implements it, is provided to you in Algorithms 1 and 2. Note that the provided algorithm is single-shot, i.e., processes propose and decide only once.

The goal of this milestone is to implement multi-shot lattice agreement, in which processes decide on a sequence of proposals. In other words, in multi-shot lattice agreement, processes run single-shot lattice agreement on a series of slots. The specification (Interface 3) of lattice agreement must be satisfied individually in every slot.

Config The config file consists of multiple lines. The first line contains three integers, p *vs* *ds* (separated by single spaces). p denotes the number of proposals for each process, *vs* denotes the maximum number of elements in a proposal, and *ds* denotes the maximum number of distinct elements across all proposals of all

processes. The subsequent p lines contain proposals. Each proposal is a set of positive integers, written as a list of integers separated by single spaces. Every line can have up to vs integers. An example of the content of a config files with 3 processes (ellipsis are not included and denotes skipped content):

Config 1:
```
100 3 5
1
2 3 4
2 5
...
```

Config 2:
```
100 3 5
2 4
1 5
2 3 5
...
```

Config 3:
```
100 3 5
1 3
4
1 4 5
...
```

Note: In the previous milestones, every process received the same config file. In this milestone, every process receives a different config file. These config files have identical first lines and differ in the rest of the lines.

Example An example of the content of output files with the previous config file (ellipsis are not included and denotes skipped content)

Process 1:
```
1 2 4
1 2 3 4
1 2 4 5
...
```

Process 2:
```
1 2 3 4
1 2 3 4 5
1 2 3 4 5
...
```

Process 3:
```
1 2 3 4
1 4
1 2 3 4 5
...
```

Manual Execution The following example builds and starts 3 processes (run from within the *template_cpp* or the *template_java* directory):

```
# Build the application:
./build.sh

# In first terminal window:
./run.sh --id 1 --hosts ../example/hosts --output
   ../example/output/1.output
   ../example/configs/lattice-agreement-1.config

# In second terminal window:
./run.sh --id 2 --hosts ../example/hosts --output
   ../example/output/2.output
   ../example/configs/lattice-agreement-2.config

# In third terminal window:
./run.sh --id 3 --hosts ../example/hosts --output
   ../example/output/3.output
   ../example/configs/lattice-agreement-3.config

# Wait enough time for all processes to finish processing messages.
# Type Ctrl-C in every terminal window.
```

3 Grading

3.1 Guidelines

This project is meant to be completed **individually**. Copying from others is prohibited. You are free (and encouraged) to discuss the projects with others, but the submitted source code must be your exclusive work (code similarity tools will be used to check copying). Multiple copies of the same code will be disregarded without investigating which is the "original" and which is the "copy". Furthermore, please give appropriate credit to pieces of code you found online (e.g., in Stack Overflow) by adding a simple comment with the link. As this is not a programming course, we do not grade code quality, documentation, or modularization, nevertheless it is still a good practice to follow those points.

3.2 Grading Scheme

This project accounts for 30% of the final grade and comprises three submissions:

- A runnable application implementing Perfect Links, interface 1 (20%)
- A runnable application implementing FIFO Broadcast, interface 2 (40%)
- A runnable application implementing Lattice Agreement, interface 3 (40%).

Note that these submissions are **incremental**. This means that your work towards the first will help you in your work towards the second.

The project evaluation consist of two criteria: correctness and performance. We prioritize correctness: a correct implementation (i.e., that passes all the test cases) will receive (at least) a score of 4.5-out-of-6. The rest 1.5-out-of-6 is given based on the performance of your implementation compared to the performance of the implementations submitted by your colleagues. The fastest correct implementation will receive a perfect score 6.

We purposely do not give the exact tests that we use for grading. We understand this can be frustrating, but you should focus on your implementation being able to handle all cases and optimize performance for the general use case. **Spend some time on testing your implementation** under various configuration, concurrent programs are complicated, you might oversee some errors if you don't take the time for it.

For the correctness part of the grading, network issues or process issues can be introduced. This includes packet delay, packet loss, packet reordering, slow processes and crashed processes.

For the performance part of the grading, you can assume that there will be no network issues or process issues. Performance will be measured by the numbered of messages you successfully delivered across all processes in a fixed amount of time (i.e., the aggregate throughput of all processes). The exact procedure is as follows, with the number of messages being large enough such that no process finishes before it is stopped:

1. Start all processes
2. Wait x minutes (e.g. 2 minutes)
3. Stop all processes
4. Wait y minutes (for the logs to be written, e.g. 1 minute)
5. Compute the aggregated throughput based on the generated logs.

3.3 Submissions

The project might not look difficult for you, but it can be really time-consuming. **We advise you to not start the project at the last moment.** The three deliverables deadlines are:

- Milestone 1, Perfect Links: November 3rd 2024, 23:59
- Milestone 2, FIFO Broadcast: December 1st 2024, 23:59
- Milestone 3, Lattice Agreement: December 22nd 2024, 23:59.

These deadlines mentioned above are definitive. We do not accept submissions over email. Submissions sent over email will not be considered.

4 FAQ

Q1. Can I use multi-threading ?

Q2. Can I put multiple messages in the same packet ?

Q3. Can I compress the messages ?

Q4. I implemented the whole project but it does not work correctly (or does not compile). Will I get some points for the implementation ?

Q5. Will there be separate performance rankings for C/C++ and Java ?

Q6. How can I validate my output ?

Q7. Is it ok that the processes terminate before they are able to deliver all the messages ?

Q8. How can I implement a Perfect Failure Detector ?

Q9. Should I log a message that I have broadcast to myself ?

Q10. Can I use JSON encoding or XML parser ?

Q11. I am having trouble running many proccesses on the VM

Q12. I am worried about the hardware limtis

Q13. Should message fragmentation be taken into account ?

Q14. My process exit with a non zero error code

Q15. I don't know why my programm crashes here ?

Q16. In the project, can packet be corrupted ?

Q17. Will the IP addresses of the HOSTS file always be localhost ?

Q18. Can I use docker ?

Q19. Why competitive grading?

Q20. How can I implement this abstraction ?

Q21. Isn't best-effort broadcast the same as using UDP without any additional mechanism ?

Q22. Can I change language between Milestones ?

Q23. Are *.stderr* checked ?

Q24. Can Java programs be given more execution time than C++?

Q25. Will the target folder for output be emptied for each test?

Q26. What happens if my process does not exit within the log-writting time window after receiving `SIGTERM`?

Q27. What should I do if the scheduler delays process starts, affecting message delivery?

Q28. What should I assume about proposal sets?

Q29. What should I do if my receiver continues logging messages after receiving `SIGSTOP`?

Q30. How should I handle large-scale tests with many processes?

Q31. How should I handle the number of threads under the JVM?

Q32. How will tests be structured when failures are present?

Q33. How do the worst test case parameters compare to those in stress.py?

Q1. Can I use multi-threading?

Yes ! However, only spawn threads (don't spawn child processes) from your process. C11/C++17 and Java support threads natively.

Q2. Can I put multiple messages in the same packet?

Yes, but we limit the number of messages per same packet to 8. Moreover, you should not use the fact that the payloads are sequential integers nor that the total number of messages is known in advance. For example, your code should work correctly if the payload is some arbitrary text.

Q3. Can I compress the messages?

Yes. This is an implementation detail that is up to you.

Q4. I implemented the whole project but it does not work correctly (or does not compile). Will I get some points for the implementation?

No. Submissions that fail to compile will NOT be considered for grading. Similarly, submissions that fail to produce any output files or produce faulty output files (e.g., empty files) will NOT be graded.

Q5. Will there be separate performance rankings for C/C++ and Java ?

Yes. C/C++ and Java will be graded separately since the performance may be drastically different. It is up to you to choose the language you are the most comfortable with. We will calibrate the performance of both languages.

Q6. How can I validate my output ?

It is your responsibility to ensure that your implementation is correct. For example, you should write some scripts that use the output files generated by the processes after they terminate their execution and validate them.

Q7. Is it ok that the processes terminate before they are able to deliver all the messages ?

Yes, as soon as you receive a SIGTERM signal, you need to terminate the process and write to the logs. You may not have delivered all the messages by that time which is ok. You should only deliver the message that you can deliver (i.e., that does not violate FIFO for example). Otherwise, you may be violating correctness.

Q8. How can I implement a Perfect Failure Detector ?

There are other alternatives to implementing URB that do not use a Perfect Failure Detector (PFD). We advise you to take a look at the book for a URB algorithm that does not involve a PFD. Remember that we cannot implement a perfect failure detector in our asynchronous environment.

Q9. Should I log a message that I have broadcast to myself ?

You definitely need to deliver to yourself. Whether you contact yourself through the network or without it is up to you. Both options are correct, it is mainly a matter of performance.

Q10. Can I use JSON encoding or XML parser ?

Yes, you can, as long as you respect all limits of the project. If it works directly on the VM, you can use it. Otherwise, it's prohibited.

Q11. I am having trouble running many proccesses on the VM

If your implementation is crashing on VM, check again how you use the resources. Your program should not crash, inspire yourself from *stress.py* to launch multiple processes.

Q12. I am worried about the hardware limtis

The bound of `MAX_INT` can be problematic in terms of memory limits, but we won't expect you to send all `MAX_INT` messages. Nevertheless, your implementation should still be able to take `MAX_INT` as input in the config file. We are aware of hardware constraints and expect them to be a bottleneck at some point.

Q13. Should message fragmentation be taken into account ?

We ensure that a message up to 64KiB can fit in a single UDP packet, thus a sane implementation should not worry about message fragmentation.

Q14. My process exit with a non zero error code

Firstly, during the communication phase, your processes should always be running. Processes should exit only after receiving the appropriate signal. If your processes exits with a non zero error code after the appropriate signal, this will not be an issue, as we will only check the output files.

Q15. I don't know why my program crashes here ?

Using multithreading is expected to be difficult and error prone. If it is on the provided template, we are very happy to help you. But for other issues, we might not be able to help you too much.

Q16. In the project, can packet be corrupted ?

No !

Q17. Will the IP addresses of the HOSTS file always be localhost ?

You cannot assume that. The IPs are given in the "HOSTS" file. But all processes have access to the same host file and know the port of every other port by reading it.

Q18. Can I use docker ?

You test your implementation using what you want. But your submission should respect our limitations.

Q19. Why competitive grading?

The project algorithms can fairly easily be implemented without optimization. The purpose of the competitive grading is for you to find better and innovative solutions in terms of design and data structures in order to maximize the number of messages broadcasted and delivered while guaranteeing the different properties of each algorithm.

Q20. How can I implement this abstraction ?

It is your job to design and think about the correct implementation. We do provide a pseudocode for the last abstraction, but you are not obligate to follow it.

Q21. Isn't best-effort broadcast the same as using UDP without any additional mechanism ?

No ! The best-effort broadcast still assumes perfect links. For example, under best-effort broadcast (with perfect links) if the sender and recipient are correct processes, then both processes will deliver the message sent by the sender. However, if these two processes communicate with UDP (without additional mechanisms), there is no guarantee that a message sent by the sender will ever arrive at the receiver

(even if they are both correct)! That's because the message may be lost inside the network (UDP allows for such scenarios).

Q22. Can I change language between Milestones ?

Yes, you can change language. But you will have to implement everything from scratch again.

Q23. Are *.stderr* checked ?

No !

Q24. Can Java programs be given more execution time than C++?

Execution time limits are guidelines. What matters is progress and reasonable speed, adhering to the 1-second rule.

Q25. Will the target folder for output be emptied for each test?

Yes, the target folder will be emptied to ensure correct output without appending to existing files.

Q26. What happens if my process does not exit within the log-writting time window after receiving `SIGTERM`?

We do not remove grades for this, but it may result in corrupted output, affecting correctness.

Q27. What should I do if the scheduler delays process starts, affecting message delivery?

This is acceptable. We account for such delays in our testing, and your implementation should handle them gracefully.

Q28. What should I assume about proposal sets?

You can assume that proposal sets are non-empty.

Q29. What should I do if my receiver continues logging messages after receiving `SIGSTOP`?

Check your signal handling code. `SIGSTOP` should stop the process, preventing further message logging.

Q30. How should I handle large-scale tests with many processes?

Optimize your implementation for scalability, ensuring memory management and efficient process coordination.

Q31. How should I handle the number of threads under the JVM?

There is no max number of threads for the VM, but there is a max per process. Do not worry about JVM threads.

Q32. How will tests be structured when failures are present?

When failures are present, we ensure tests where memory is not a problem. Without failures, we run larger tests which will fail unless you have a memory reclamation scheme.

Q33. How do the worst test case parameters compare to those in stress.py?

The worst test case parameters are no worse than those given in stress.py. Correct implementations work in all environments, as concurrent programs are inherently hard to debug.