Foundations of Blockchains - Consensus Lab

Open Source Version

The lab is an open-source version of the Consensus Lab used in the Foundations of Blockchains course at Carnegie Mellon University. This version includes lab instructions, starter code, and setup instructions for the auto-grader using Gradescope.

Students For students, the content in this repository is sufficient for learning and practicing distributed consensus. You can test your code with the provided test cases in the "Test" folder. Your working file would be:

- Checkpoint 0: `Majority_Vote_Adversary.java`
- Checkpoint 1: `Student_Player.java`
- Checkpoint 2: `Dolev_Strong_FRound_Adversary.java`

Instructors: For instructors interested in using this lab in their course, please contact Junxi Song at 20001218sjx@gmail.com for the solution and additional test cases. In our current autograder setup, each checkpoint has test cases assigned point values:

- Checkpoint 0: 16 test cases, each worth 1 point
- Checkpoint 1: 41 test cases, each worth 1 point
- Checkpoint 2: 12 test cases, each worth 1 point

1 Introduction

In this lab, you will learn how to construct and analyze consensus protocols. In particular, we will start by implementing a toy "majority vote" protocol and understand why this protocol is insecure by designing attacks that violate the security properties (validity and consistency) of a consensus protocol. We will then implement the Dolev-Strong protocol, which is indeed a secure consensus mechanism. We will try understand why the Dolev-Strong protocol is designed exactly the way it is, and why certain variations and simplifications of it would break the protocol's security. To do this, you will be asked to design attacks that violate the consistency or the validity properties of insecure variants of Dolev-Strong.

2 Project Structure

Starter code. The starter code is in the folder `Code`. All the code for Checkpoint 0, 1, and 2, including the project dependency and submission files, are included in this zip file.

Directory structure. The main directory for the source code is

`code/src`,

henceforth we also refer to this directory as `./`. There are four folders under `./`.

1. The `./Main` folder has two factory for the list of adversaries and players in this project.
2. The `./Protocol` folder has all protocols and their corresponding adversaries we provided to you, everything you need to change is in `./Protocol` folder,
3. The `./Simulator` folder is the framework of the simulator, and we will explain what each class does shortly.
4. Finally, the `./Test` folder contains some tests we have created for you. In our Autograder, we have additional hidden test cases, so we encourage you to write more adversaries and tests yourself.

For each protocol or attack, we have created a file for you where your code should be, you should complete your work without changing anything outside the designated file. To test your programs locally, you could run some tests we provided to you in the `./Test` folder.

2.1 Consensus Simulator Framework

The following modules in the consensus framework will be relevant to you. Below, we explain them one by one.

2.1.1 Idealized Signature Module

In your implementation, you can call the `sign` function in the `Player` class (or any class that extends `Player`) class to sign messages on behalf of the player, and call the `verify` function to verify if the purported signature is valid.

Specifically, `Player.sign` returns a string that includes the message itself, the signature, and the signer. This string is the `json` serialization of a `SignedM` object which we will explain shortly[1].

[1] If you are interested in JSON, you can learn more on how it works at https://sites.google.com/site/gson/gson-user-guide.

2.1.2 Authenticated Channel and Message Sending

The communication between players is through an authenticated channel class `Fauth`. The channel is authenticated in the sense that the message recipient can know who is the true sender. To work with the authenticated channel, in your implementation, you need to know that the messages that a player receives is an object of the `Package` class, which includes the `sender`, the `receiver`, and the actual message `strSignedM` whose type is a raw `String`. It is guaranteed that the receiver identity agrees with the recipient's identity.

To parse the received string `strSignedM` (of type `String`), you can call the `Player.parseSingleMsg` function (or call it from any object that extends `Player`) to convert it to an object of the `SignedM` class. The `SignedM` class includes the actual message `msg` as a `String`, the `signer`, and the signature `sig` as a `String`.

To send a message, you can call `Player.send` which takes the message of the type `String` to send, and the recipient identity. If you want the message to be signed, you could directly pass the output of `Player.sign` to `Player.send`, since `Player.sign` is a wrapper that did all the serialization work for you (as we explained earlier).

2.1.3 Simulation Engine

The simulation engine, i.e., the `Simulation_Engine` class, will initialize the idealized signature, the authenticated channel, all players, as well as the provided adversary if any. After the initialization, the simulator begins a round-by-round simulation. In each round, the engine will call the `action` method of each honest player and then call the adversary's `attack` method. There will be one global instance of the `Simulation_Engine` class called `engine`. Your code may need to call the following functions inside `engine`:

- `engine.roundNumber`: returns the current round number;
- `engine.numOfPlayers`: returns the total number of players;
- `engine.numOfFaultyPlayers`: returns the maximum number of faulty players.
- `engine.checkOutput`: returns `false` if either consistency or validity is broken during an execution of the protocol, else returns `true`.

2.1.4 Adversary

You will be asked to implement adversaries for some broken consensus protocols. Your attack code will extend the `Adversary` class. `Adversary.faulty_players` returns the list of faulty players. The messages sent by faulty players are fully controlled by the adversary. The adversary can just call the faulty players' `send` function to send messages on their behalf. The adversary has the capability of looking at what messages honest players intend to send in the present round, before deciding what messages the faulty players want to send in this round. To look at the honest messages of the present round, you can call the `Adversary.honestMsgs` function which returns a list of `Package`s.

3 Checkpoint 0

In Checkpoint 0, you will familiarize yourself with the toy "Majority vote" consensus protocol, implemented in the `./Protocol/Majority_Vote_Player.java` file. This protocol runs for two rounds. In the first round, each player checks whether it received a single valid message from the

designated sender, and if so, signs the designated sender's bit and sends this message to all other players. Otherwise, it signs and sends a default bit instead. In the second round, players output the bit that received the majority of votes or output 0 if no bit or both bits received the majority of votes.

Your goal is to construct an adversary that breaks either consistency or validity of this majority vote protocol. You should implement your attack in the `./Protocol/Majority_Vote_Adversary` file. Specifically, your task is to implement the `attack` method of the `Majority_Vote_Adversary` class. You can query the faulty players by accessing `Majority_Vote_Adversary.faulty_players`. In some test cases, the designated sender is faulty, and in others, it is honest. If the setting is unbreakable, you can simply return from the `attack` method without doing anything. You can test whether your attack works using `engine.checkOutput` method (check the comments in the code for more information).

Tests: In `./Test/MVTestCases.java` we provided a test case for you. You may add more tests cases with different parameters for the `Simulation_Engine` to check on your code and see what will happen.

4 Checkpoint 1

For Checkpoint 1, your task will be to implement the Dolev-Strong protocol as it was given in the lecture. Both the Dolev-Strong protocol and the majority voting protocol extend the `Player` class. Specifically, your task is to implement the method `action` in the `Student_Player` class which extends `Player`. The base class `Player` takes care of some boiler-plate code for you, including networking and signature creation. You can read the "Consensus Simulator Framework" section for more details.

Remember to test your code both in the honest setting (i.e., all players follow the protocol), and the adversarial setting (i.e., some players might deviate from the protocol). We highly encourage you to thoroughly test your implementation — for example, you could output the state of each player in each round and check whether its current extracted set is exactly as expected. You could also try to see what happens if some players deviate from the protocol (for example, are not responsive).

Tests: In `./Test/DSTestCases.java` we provided four test case for you. You may add more tests cases with different parameters for the `Simulation_Engine` to check on your code and see what will happen.

Important Your Dolev-Strong player must use the following wrapper to send and receive messages.

- `DS_Serialize`: takes in a list of `SignedM` objects, and creates a serialized message of the `String` type. In the r-th round, you should call `DS_Serialize` with a list of $r + 1$ `SignedM` objects, all signing the same message, and the senders should be distinct.
- `DS_Parse`: you should receive an object of type `Package`. You can pass the `strSignedM` string contained in the received `Package` object to `DS_Parse`, which will return to you a list of `SignedM` objects. To parse correctly, all of these `SignedM` objects should include the same message. The `DS_Parse` function will throw an exception if the received message cannot be parsed as a message signed by a list of players.

Note that neither `DS_Serialize` nor `DS_Parse` will check signature validity for you, they also do not check that the signers are distinct, or who the signers are. You should check such protocol-specific semantic behaviors yourself.

5 Checkpoint 2

In Checkpoint 2, the goal is to understand why the Dolev-Strong protocol is designed exactly the way it is, and why we must be extremely careful when making any changes to the version that has been proven secure. Specifically, we will try to modify the original protocol by letting it run only a single round less than the version we saw in class, that is, run it in f rounds.

You may modify your `Student_Player` class to be an implementation of the Dolev-Strong protocol in `Dolev_Strong_FRound_Player` class by updating only 1 line of code. After that, you need to implement a successful attacker for this variation.

Your task is to update the `Dolev_Strong_FRound_Player` class and implement the method `attack` in the `Dolev_Strong_FRound_Adversary` class under the `./Protocol` directory. The attack is successful if at the end of the f-th round either the validity or the consistency of the protocol is violated.

Tests: In `./Test/DSAttackTestCases.java` we provided a test case for you. In the test case, you can read the number of faulty players from `engine.numOfFaultyPlayers`, and your `Dolev_Strong_FRound_Player` class is parametrized to run for `engine.numOfFaultyPlayers` number of rounds. The list of faulty players can be accessed through the `Adversary.faulty_players` variable.

6 Recommendations for Programming Environment

This lab runs in Java 11, you can use any editor and development environment that allows you to edit, compile and run the code.

We recommend using either Intellij to edit/compile/run or use your favorite editor to edit code and shell scripts provided by us to compile/run code as described below.

6.1 Intellij

You can download Intellij unlimited from here , and get a free Intellij License from from.

After you have downloaded the Intellij and the source code for this lab, open Intellij, and select `Open`, then choose the `code/src` folder. Once the project folder is loaded in Intellij, we will setup the project SDK and add the library dependencies.

- Project SDK: click `File` → `Project Structure`, go to `Project` → `Project Settings` and for the SDK field, choose 11.
- Dependencies: click `File` → `Project Structure`, go to `Project` → `Modules` → `Dependencies`, click the + sign (see Figure **??**) and then click `JARs or Directories`. At this point, add `gson-2.8.5.jar`, `hamcrest-core-1.3.jar`, and `junit-4.13.2.jar` JAR files from the `Code/lib` directory.

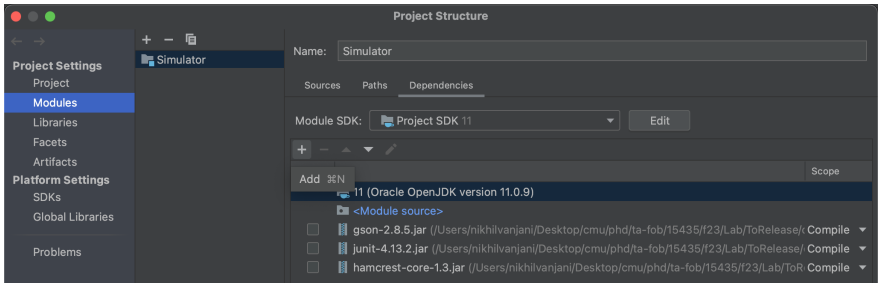

Figure 1: Adding dependencies in Intellij

After these steps are done, you should be able to successfully compile the starter code.

Once the code is compiled, you can run any test in the `./Test/` directory. For example, to run `testDrop1` in `DSTestCases`, go the the `./Test/DSTestCases.java` file and click the green triangle next to line 12. If you want to run all the tests in `DSTestCases`, click the green triangle next to line 10. If you want to run all the tests for all the checkpoints, click the green triangle next to line 15 in the `./Test/RunTests.java` file.

6.2 Shell scripts

In the `Code` folder, you can find two shell scripts `compile.sh` and `run.sh`. The command `./compile.sh` compiles all of the code along with the dependencies. The command `./run.sh` runs all the test cases that we have provided you. Alternately, test cases for specific checkpoints can be run as follows.

- Checkpoint 0: `./run.sh Majority_Vote_Adversary`
- Checkpoint 1: `./run.sh Student_Player`
- Checkpoint 2: `./run.sh Dolev_Strong_FRound_Adversary`

7 Autograder Integration

We currently support Gradescope for autograding. There are two options available for integrating the autograder with Gradescope.

Set Up Your Own Autograder Our project is compatible with Gradescope, as we utilize JUnit tests in the `Test` class. To integrate your autograder, follow the instructions provided in the Gradescope Autograder Documentation to build your base image and upload it to Gradescope.

Specifically, you will need to create two files

- `setup.sh` This file instructs the autograder to install the necessary dependencies for the project.
- `run_autograder` This file guides the autograder to copy the students' code into the project, compile it, and execute the tests.

Details could be fined in the link above.

Modify Our Autograder If you prefer not to build your autograder from scratch, you can contact Junxi Song for access to our autograder configuration, including our test cases. Although the file hierarchy differs slightly from the open-source version, it remains fully functional.