Distributed Systems

Dr. Kamandi

Parsa Mohammadpour

Introduction:

In this assignment we were asked to implement **randomized coordinated attack problem** by code and by simulation, see how often does this algorithm works right.

First, we specify the algorithm description, then we go through the code and after that we show some of our simulation results, then in the end we run simulations for some worst case and show the results.

The codes are provided as notebook file and some classes with this document. And in the end, we provide references.

Algorithm and problem description:

Coordinated attack problem is a famous problem in distributed systems. Here we provide the question with an example and then we go for formal definition.

Imagine that there are some generals and each of them wants to attack to specified object. They are going to succeed if **all of them attack simultaneously** otherwise they are all going to die. In first view, this problem is so easy to solve, they just need to send their states by messengers to each other, then after that they all decide to weather attack or not. But what if we have some failures, for example a messenger is captured in the way? Or what if some messenger is spy and say wrong things? First scenario is known as link failures and the second one is known as byzantine failures. In any of these two scenarios, we have to solve the problem in harder circumstances. It can be proved that "**No deterministic solution that can solve the consensus problem for the system if we have link failures**". The proof is by **indistinguishability** of two scenario, but it is not in our content, so won't go any further.

Formal definition of problem:

Here we have some process named 1, …, n respectively and have an initial value (zero or one for example in database commit or attack or not to attack in generals' problem). Each of them is in a complete graph[1], or we can say all nodes are aware of the entire graph, and they can send messages to each other and these messages can be delivered or can be lost due to link failure at each round. We want all of these nodes decide at some points with following rules(requirements). These requirements are as follows:

- **Agreement:**
  No two process can decide on different value at the end
- **Validity:**
  If all processes started with value zero, then zero is the only possible choice.
  If all processes started with value one and we had **no link failures**, then one is the only possible choice.
- **Termination:**
  All processes eventually decide.

First two requirements (agreement and validity) are known as safety properties and the third one is known as liveness property of a distributed system.

Now we have formally defined the problem, so we go for the solution. Remember as we mentioned earlier, there is no deterministic solution that can solve this problem in case of having even one link failure.

[1] Graph that all its nodes are connected to each other

Randomized coordinated attack algorithm:

Initialization and values:

In this algorithm, we assume that we run the algorithm r step (in r rounds). In the beginning one process (process number on for example or simply the leader that we know how to elect that from before lessons) generates a random key in range of [0, r] (both zero and r, are inclusive). In each process we have some variables. These variables and their bounds are as follows:

- **Uid:**
  Process identifier. (simply process number among other processes)

- **V (values):**
  This is a vector with n index, each of these indexes corresponds to a process initial value.
  Each index of this vector can have a valid value for decision. (for example, in our case of having commit or not commit, we can have zero and one respectively)
  Initially it is *undefined* for all other processes (all indexes value is *undefined*) except for its own. We set *v(uid)* to its own initial value. (the initial value that a process has in the beginning)

- **Information levels:**
  This is also a vector with n index, each of these indexes corresponds to a process information level that we are going to define later on. Initially we set all indexes of this vector to zero except its own index. We set information_level(uid) to one.

- **Key:**
  This is the key that we talked about above. This is the random generated key of the process one.[1] (which can be from zero to r) Initially, it is set to *undefined.* But the leader process (process number one for example) generates a **random uniformly distributed** number of the range [0, r] inclusively.

- **Decision:**
  The final decision that a process is going to make. It must be in all possible decisions union *unknown*. (in case of database commit, one and zero and one and *unknown*) initially, it is set to *unknown.*

At each step all processes send their all states (information levels, values, key and its own uid) to all[2] other processes, with the consideration that some of these messages might not be received due to link failures. So now we have to define our trans[3] function.

[1] here for simplification, we consider process one as key selector but, in the implementation, we can have different process as leader or the key selector.
[2] Here for simplification, we assumed that we have a complete graph, but in the implementation, we can have different communication graphs and see how the algorithm works.
[3] The function that is going to be executed at each round (step) of the algorithm.

Trans function:

At each step all process does the same thing on all their received messages. For each of them, they update their variables as follows[1]:

- **Values:**
  We iterate over value vector, if we have an *unknown* value in an index, we replace it with the value in the message.

- **Information levels:**
  We iterate over the process (own information level indexes) and we assign to that index, max of that value (processes own information level) with the received message information level. Something like: "for i in 1, …, n info_level(i) = max (info_level(i), received_info_level(i))". Then for its own information level (its uid index in information level vector), it iterates over the information level vector, and assign the min value that it had seen plus one to its own information level.
  Something like:"info_level(uid) = min(info_levels) + 1"

- **Key:**
  If the key is *undefined*, replace it with the new key from the message. If it is *undefined* we will have *undefined* again, but if it has value, we will set the value to it. And if we have a value in key variable, we don't change it.

There is an interesting point about these information level values. Each process information level[2] can be at most one difference with other processes information levels. (**one number higher or lower that the other processes information level**) in other word, even in any information level vector, each index value, can only be at most one higher or lower than the others. (they can all be same too, but in general we know that they can't have difference more than one)

At the end of this algorithm, when processes find out that this is the last round, for example they keep a variable related to round, and increase it at each step until it is equal to one at the end of an execution. Then at that time, each process decides by the following decision rule:

If the key is not *undefined*, and values in v variables are not unique (we have at least two different values in v vector) and if key is smaller or equal to all values in information level variable, we return one (the unanimous decision) otherwise we return zero[3].

[1] Uid always remain the same, so we ignore its relevant variable in update (trans) function.
[2] Process information level is somehow the value of uid index in information level vector. Something like information_level(uid).
[3] Zero is the default decision we considered here. In database commit, it is as same as not committing. In the code we can have anything as default decision.

Implementation:

For this part we have implemented some classes. We also provide a little bit of explanation for each of them. These classes are located in 'randomized-attack/classes' folder and contains following files:

- **Message.py:**
  In this file we have only one class, Message class, which if fact it is a holder for the message content that processes exchange with each other in each round. This class contains some properties that are required for generating an object of this type, which we are going to say few about them. These properties are as follows:
  - **Sender_uid**: It contains the sender process uid.
  - **Initial_values**: It contains initial value variable of the sender process in that moment.
  - **Info_levels**: It contains info_level variable vector of that process in that moment.
  - **Key**: It has that process key variable of that process in that moment.

  Every process that in simulation sends a message, it produces an object of this type and sends it to other processes. This class functions are mostly for setting and getting result so we omit them.

- **Process.py:**
  This file contains a process class which is the same as process we have in algorithm. It has some variables (we talked about them previously) and have some functions to perform some actions on every situation, like sending a message, receiving a message, updating its status, getting final decision and etc. this class have some properties that we have to pass to it in the generation time. These properties are as follows:
  - ***uid***: Process uid (number between 1 to n) that we give to each process
  - ***initial_val***: Initial value of process (must be in any of all possible decisions)
  - ***process_count***: Number of all processes that we have (we need this number for generating processes variable like initial values vector and information level vector)
  - ***default_decision_val***: Process default decision value in case of not having a united decision.
  - ***prob_func***: probability function for not having link failure. The function that takes no input and returns a Boolean for the result. If the returned Boolean is *True*, the process can send the message to that process (we have no link failure), and if it returned *False*, the process will send *None* to the process we wanted to send a message to. (it means we had a link failure)
  - ***is_key_generator***: It is a Boolean specifying that this process is the key generator (or simply the leader) or not. It has the default value of *False*.

  This class also have some functions in order to perform some actions like receiving message, sending message and etc. these functions with a little explanation of them, are as follows:

  - ***__initial_process_values***: Set the process initial values (v vector in algorithm). Setting all indexes to *None* except its own index. Set this index to its own initial value.

- ***__update_initial_values***: This function gets a new an initial value vector, check if it has the same size as this process initial values vector[1], if not raise an error, if it has same size, then update the initial values vector by replacing all *None* indexes of this input vector.
- ***__initial_info_levels***: Initial process info levels variable by setting all indexes o minus one except its own index, it sets its own index to zero.
- ***__update_my_info_level***: iterate all over the information level vector, and find the minimum number in it. Then it sets its own information level (the index corresponding to its own uid) and set its value to the minimum computed earlier plus one.
- ***__update_info_levels***: this function gets an information level vector as an input. Then it checks whether or not it has the same size as the process own information level vector, if it doesn't have same size, it rase an error. Otherwise, it goes index by index and apply the maximum function between the process information level vector and the new information level vector. Then it updates its own information level by calling *__update_my_info_level* function.
- ***set_neighbors***: Setting some processes as this process neighbor. It means that this process has links to some other processes. (in case of complete graph, this contains all processes except its own)[2]
- ***generate_key***: This function is used for generating the key, or setting the key in the key selector (leader process) to a value. It gets number of rounds we have in this algorithm and if we don't want our simulation to be for some specified key, we can randomly generate it or if we want the simulation for a certain key, we can just pass that key to this function.
- ***__set_key_from_msg***: Setting the key variable value from the message object if the received message variable's key is not *None*.
- ***get_key***: Returns processes key variable value.
- ***get_uid***: Returns processes uid value.[3]
- ***get_initial_val***: Returns process initial value.
- ***get_information_level***: Returns a copy of the process initial value.
- ***__generate_message***: This function is used to generate a message object according to the process current state, in order to be able to sends it to others.
- ***send_messages***: This function is used for sending message to neighbors of that process. It generates a message object first, then it iterates over its neighbors and by applying the *prob_function*, function we used for link failure probability, and sends the message to that process or sends *None* to that process in case of link failure.
- ***receive_message***: We gets a message from a process and we append that message to our received message list that we have.
- ***update_from_messages***: This function iterate all over the received message list, and apply the required updates. (which is implemented in *__update_single_message* function)
- ***__update_single_message***: Here we call some other functions that we have implemented. We call *__set_key_from_msg*, *__update_info_levels* and *__update_initial_values*.

[1] I used list in python implementation for each vector in the algorithm.

[2] Here we support the case of having any type of communication graph, even directed ones.

[3] For simplification, we keep process uid minus one. Which is relevant to starting index of lists in python. Because otherwise every time we wanted to access the index corresponding to that process, we had to find uid minus one.

- ***get_final_decision***: Computing and returning the final decision which was taken by the specified decision rule.

- **Randomized_coordinated_attack_sim.py:**

This file contains all processes and somehow it specifies algorithm steps. It makes process to sends message to each other, execute received message functions, apply update for received messages and etc. somehow it can be said that it manages processes in algorithm steps. As from previous ones, here we begin with properties that are passes to this class for generating an object of this type. But before that, we have two function at the beginning of this file, *save_plot* and *generate_file*. These two functions are used for saving a matplotlib plot and generating a directory for a path respectively. These two functions are used in this file couple of times. Now let's get into the required properties for initializing this class objects. These properties are as follows:

- ***process_count***: Number of processes that we want in our simulation.
- ***round_num***: Round numbers that our algorithm will have. (r in the formal algorithm)
- ***key_selector***: Number of the process (between one and *process_count*) whose is in charge of selecting the key at first step in the algorithm.
- ***decisions_list***: The list of all possible decisions that we can have.
- ***process_initial_vals***: It's a list containing of all process's initial values. If the value of this variable was *None*, we randomly generate initial values of the processes from all possible decision list.
- ***communication_graph***: It is the graph which specifies our links between processes. If the variable value was *None*, we generate a complete graph as the communication graph.
- ***prob_func***: the probability function that is used in processes.
- ***simulation_num***: number of simulations that we are going to have in each simulation. (number of times that we are going to run the algorithm)
- ***differ_initial_val_on_round***: It is a Boolean which determines whether or not generate a new initial value for each process at each simulation.
- ***base_path***: the base path that we are going to save results in it.
- ***fixed_key***: If it is *None*, it means that we have to generate a random key at each simulation, but if an integer was given, we will apply our simulations with that key only. Ad it has to be between zero and number of rounds that we have.

Then, as before, we are going to give a little bit of explanation about functions that we have in this class. These functions are:

- ***__initial_process_initial_values***: Initializing process's initial values. Either generates them randomly or just setting the by the input.
- ***__initial_process***: Initializing processes and also setting their neighbors. These two actions are performed by calling two other functions. "*__generate_process*" and "*__set_neighbors*" used for generating a process number *i* and setting neighbors of all processes respectively.

- ***__set_neighbors***: Iterate over all processes and gets neighbours corresponding to that process from communication graph and call *set_neighbors* function of that process with the neighbors process that it had found from communication graph.
- ***__generate_process***: generates and returns a process with the uid and specified properties that we had at the beginning of the class definition.
- ***__set_communication_graph***: This function is used for setting communication graph. If the communication graph doesn't have nodes number equal to the process number variable we had, we raise an error. Otherwise, we set that graph as communication graph. If the input graph was *None*, we set default communication graph, which is complete graph.
- ***get_communication_graph***: Returns a copy of communication graph.
- ***plot_communication_graph***: Showing communication graph. Which in this representation the key selector node (process who's in charge of selecting the key at the beginning) will be represented with different color. (This is the default but colors can be given as input and change them to whatever valid matplot library color that you want)
- ***__reset***: This function is used for resetting everything to the default value after each simulation or group simulation[1]. (change processes initial values if it has to and regenerates the processes)
- ***simulate***: the function that is used for simulation. Each time it runs the algorithm over the processes and gets the results and returns them as *data frame* object. In each time of running algorithm, it calls *__single_simulate* method, which is in charge of running algorithm for the provided input and processes.
- ***__single_simulate***: This function is in charge of running algorithm for the provided processes and their inputs for the specified rounds. First it calls reset function. Then it calls the process whose is in charge of selecting the key and selects the key (or set the key if simulations are for same key) and run the algoritm. After running algorithm until r rounds, it returns the result as a dictionary containing previous function *data frame rows*. In each round of algorithm this functions calls *__simulate_round* function which is in charge of running each step of algorithm.
- ***__simulate_round***: This function is in charge of running each round of algorithm. First it changes the seed in order not to have same results in each running of program (this could be done many other places to, but I put it here because it doesn't matter that much where it be and by being here, we change it couple of times) and then run each step of the program. Iterate all over the processes and make them to generate and send messages to each other (to their neighbors) and update their states (variables) with the received messages.
- ***__check_correct_answer***: This function is in charge of checking which the results answer is correct or not. This function checks our three requirements (agreement, validity and termination) and returns a Boolean which determines if our result satisfies requirements and a string to show which of the requirements is not satisfied in case of not having correct answer. This function calls *__has_termination_violation*, *__has_agreement_violation* and *__has_validation_violation* functions respectively for checking termination violation, agreement violation and validation violation, which we talked about in earlier sections.
- ***__has_termination_violation***: Checks whether all processes have decided or not.

[1] Group simulation is the concept of applying the simulation, which was running the algorithm number of times, number of times too. Just for getting the probabilities and other stuff. We have a function corresponding to this named *simulate_n_times*.

- ***__has_agreement_violation***: Checks whether or not we have two or more processes that have different decisions.
- ***__has_validation_violation***: Checks if we have started with same decisions and we had no link failure, all processes have decided on their initial values or not. (notice that all their initial values were checked to be same before)
- ***get_results_compare***: receives a *data frame* and returns group by result of that *data frame* on different columns. For example, if we select column to be 'is-correct-answer', we get number of correct and incorrect answer we had in that *data frame*.
- ***__change_stdout_to_file***: This is a static function which is in charge of setting the *system output* of the *print* method in python. Because the console in notebook, each cell output has a limit and if it didn't have a limit, we had too many lines in there and we might lose trach of them, we write the print result in a file. For this purpose, we have to change the *system output* of *print* to file. This action is done by this function and the next function. This functions changes *print* function *system out* to file.
- ***__change_stdout_to_console***: This function, as was explained in previous function, is used for same purpose, but in opposite way. After we have finished our work for writing logs, we will reset the *system out* from file to *console* for *print* function.
- ***simulate_n_times***: This is the group simulation that we have pointed to earlier. This group simulation is just calling simulation function number of times. First it plots communication graph if we determined to log. Then after some preprocessing like changing the print output to file instead of console and …, we call the *__apply_simulation_n_times* function which is in charge of applying the simulation n times. And returns some results. Then in this function, we show the results and draw some plots and return *data frame* object which was made from *__apply_simulation_n_times* results and by calling *__generate_simulation_n_times_df_plot* function, we draw its plot it was determined to draw a plot.
- ***__apply_simulation_n_times***: This function is in charge of applying the simulation couple of times and return all simulation results in two list, *failures_count* and *success_count* respectively for successful simulation number and failed repetition in each simulation number we have had.
- ***__generate_simulation_n_times_df_plot***: Draw a plot the group simulation result we had.

• Notebook files:

These files are provided for having a conclusion and by running the simulation class for different inputs like different graphs and …. So, this part will be explained in the conclusion section but then we will focus on the results and getting some conclusion from them not the code. Code part of these notebooks is not explained because they are simple and easily understandable.

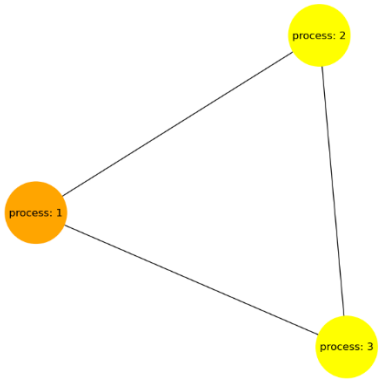
Conclusion:

Here we will do our simulation part for four main cases in which we will apply simulation for complete graphs. And these three main cases are simulated twice, with three processes. Simulation results are provided here and you can even see them in the notebook folder. These four main parts are as follows:

1- Random cases:

The processes inputs are generated randomly and we simulate everything for it and see how often we have failures. We do this simulation some times as see the results.

We do this simulation for 3 processes and see the result. First, we will show the communication graph which is a complete graph containing three processes. The communication graph is shown in the below figure:



As you can see, the communication graph is a complete graph with three nodes. And also, we declared process 1 as the leader (key selector) and we represented it with different color in comparison with other processes. Lest go for the simulation results with five rounds at each simulation step. The result is as follows:

	initial-vals	decisions	info-levels	received-count	failed-count	key	is-correct-answer	reason
0	[0, 1, 0]	[0, 0, 0]	[2, 3, 2]	18	12	[2, 2, 2]	True	-
1	[1, 0, 0]	[0, 0, 0]	[2, 2, 1]	15	15	[3, 3, 3]	True	-
2	[1, 1, 0]	[0, 0, 0]	[1, 1, 1]	16	14	[2, 2, 2]	True	-
3	[0, 0, 0]	[0, 0, 0]	[2, 2, 1]	13	17	[4, 4, 4]	True	-
4	[0, 0, 0]	[0, 0, 0]	[2, 2, 2]	15	15	[4, 4, 4]	True	-
5	[1, 0, 1]	[0, 0, 0]	[1, 0, 1]	16	14	[3, None, 3]	True	-
6	[0, 1, 1]	[0, 0, 0]	[2, 2, 3]	12	18	[1, 1, 1]	True	-
7	[0, 0, 1]	[0, 0, 0]	[2, 1, 1]	9	21	[3, 3, 3]	True	-
8	[0, 1, 0]	[0, 0, 0]	[2, 2, 3]	16	14	[3, 3, 3]	True	-
9	[1, 1, 0]	[0, 0, 0]	[3, 3, 3]	19	11	[3, 3, 3]	True	-

This is the *data frame* file that contains each process initial values, final decisions, info levels, number of link failures we have had, the key variable of each process, the result of simulation that if satisfies requirements and the requirements that we don't satisfy if we don't satisfy any. We have one thousand simulations but we only give first ten simulation to just show few of them. The complete results are provided with this file.

For a more specific look, in following figures we show different values of is-correct-answer column and their repetition and do the same for reason column too. The figures are for these columns:

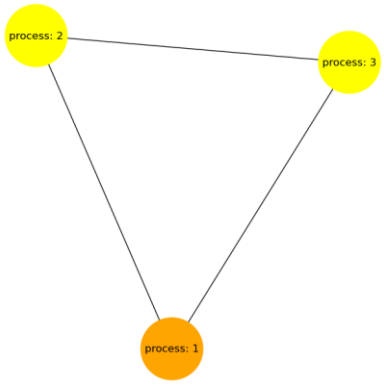
is-correct-answer	<unnamed>
False	19
True	981

reason	<unnamed>
-	981
Agreement	19

As you can see in following figures, we only have agreement violation in some (few) cases. It is the exact same thing that we talked about in the class. And also, in 1000 simulation that we had, we only had 19 wrong answers. So, the probability of not having the correct answer in general case not so much. (As we discussed in the class the 1/r probability of failure, was for the case that all processes started with the value one, but here we choose initial values randomly and in the next section we go for the case that we have only initial values of one.) if we concentrate about the results, every time that we had failures, we had only initial values of one for all processes. So, the worst case is to have all initial values of ones. The exact same thing that we said in the class.

2- Worst case:

We do this simulation for 3 processes but here we only start with initial values of one for all processes and see the result. (from previous section we know that it is the worst case and only in this case we might have a requirement violation) So, first we will show the communication graph which is a complete graph containing three processes. The communication graph is shown in the below figure:



As you can see, the communication graph remains the same because we don't change it in this step. So, the same thing can be told. We have a complete graph and process one is the key selector.

Lest go for the simulation results with five rounds at each simulation step. The result is as follows:

	initial-vals	decisions	info-levels	received-count	failed-count	key	is-correct-answer	reason
0	[1, 1, 1]	[0, 0, 0]	[2, 2, 2]	15	15	[4, 4, 4]	True	-
1	[1, 1, 1]	[1, 1, 1]	[2, 3, 2]	16	14	[1, 1, 1]	True	-
2	[1, 1, 1]	[0, 0, 0]	[1, 1, 2]	14	16	[3, 3, 3]	True	-
3	[1, 1, 1]	[0, 0, 0]	[3, 3, 2]	17	13	[5, 5, 5]	True	-
4	[1, 1, 1]	[0, 0, 0]	[3, 3, 3]	21	9	[4, 4, 4]	True	-
5	[1, 1, 1]	[1, 1, 1]	[2, 2, 2]	16	14	[1, 1, 1]	True	-
6	[1, 1, 1]	[0, 0, 0]	[2, 2, 1]	14	16	[4, 4, 4]	True	-
7	[1, 1, 1]	[0, 0, 0]	[1, 1, 1]	14	16	[2, 2, 2]	True	-
8	[1, 1, 1]	[0, 0, 0]	[2, 1, 1]	16	14	[5, 5, 5]	True	-
9	[1, 1, 1]	[0, 0, 0]	[2, 2, 1]	14	16	[5, 5, 5]	True	-

As you can see, in this data frame now all the initial values are a list of ones with the size of three. By chance in first ten rows, we don't have any wrong answer produced by the simulation, but in the later figures we can see some failures too.

Now just like previous part, for a more specific look, in following figures we show different values of is-correct-answer column and their repetition and do the same for reason column too. But now we expect to see more wrong answers. Let's see if we are right or not. The figures are for these columns:

is-correct-answer	<unnamed>
False	143
True	857

reason	<unnamed>
-	857
Agreement	143

As you can see in following figures, we only have agreement violation in some (few) cases, but there are more failures as we expected. It is the exact same thing that we talked about in the class. And also, in 1000 simulation that we had, we only had 19 wrong answers. So, the probability of not having the correct answer in general case not so much. So, by computations, **the results are not exactly 1/r** of the simulation numbers. But why? We will answer this in the next section.

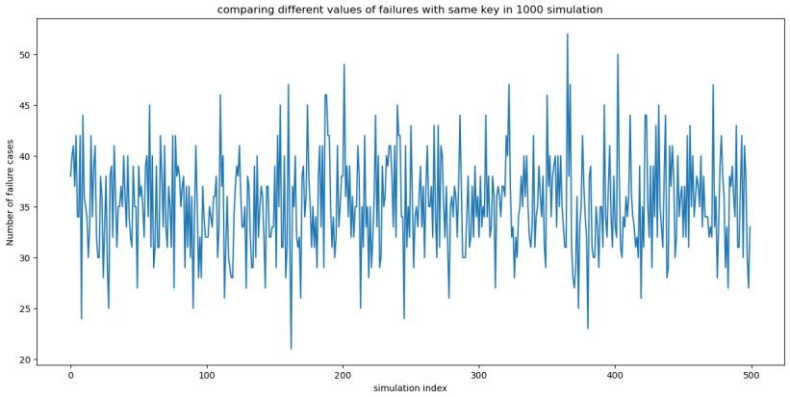
3- Worst case, same key:

In previous section we saw that the probability of having wrong answer is less than 1/r. But why? The reason is that because we **didn't do the simulations for a same key** to get results. For example, once we simulate for key equal to five, another time we simulate for the key equal to three and …. But if we simulate the program for a same key for this example, with the link failure probability of 50 percent, if we simulate for key equal to two, we get pretty good results. For one simulation, the results are as follows:

is-correct-answer	<unnamed>
False	329
True	671

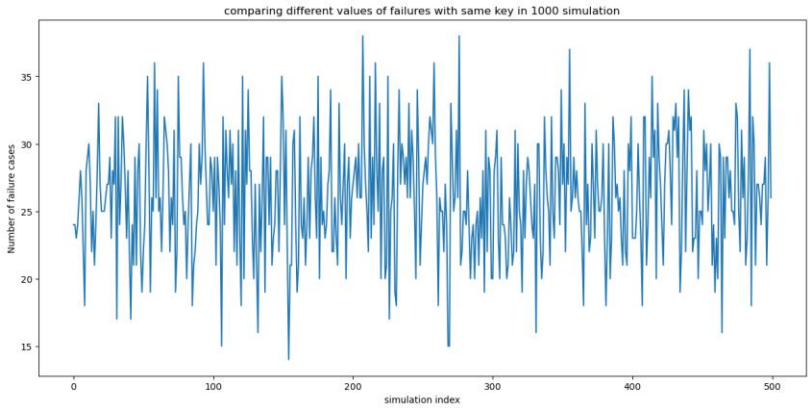
reason	<unnamed>
-	671
Agreement	329

It is approximately the 1/3 of the 1000 simulation, we got wrong answer for key equal to two and rounds number of 3 for these three processes. (before this, number of rounds was five and we didn't have fixed key) but one simulation result is not good enough. Lest simulate this multiple times and see if the results stand in range of 1/3 of simulations. Now we show the figure for running the simulation 500 times and in each time that we run simulation we store number of failures to show in a plot. (this time in simulation we run algorithm with same key and initial values 100 times) the result is as follows:

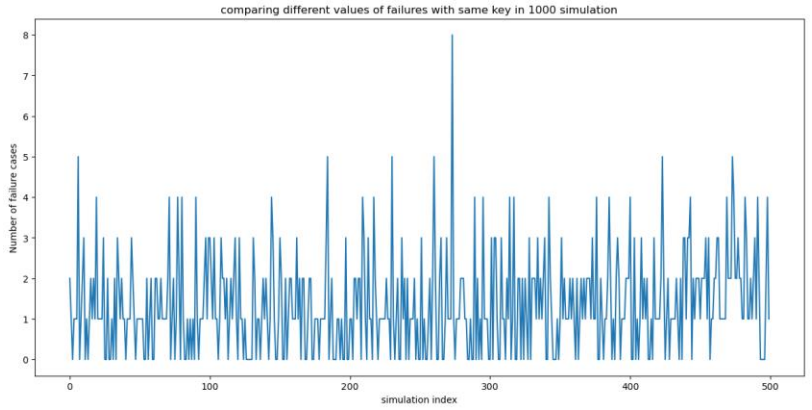


Also, the mean was 35.226 which is somehow good for our purpose. But why it is not exactly 1/r or in this simulation, why it is not exactly 1/3 of times? Why we don't have 33.3 failures as the mean? And

also, let's see if the results stand if we change the key from 2 to one. The following is the figure of the same simulation but for key equal to one. The result figure is as follows:



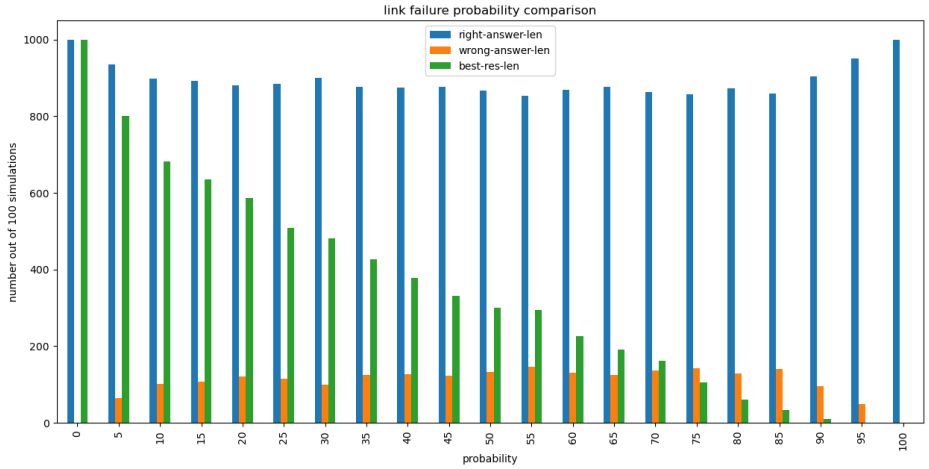
The probability was decreased and it is less than 1/3 of times (33.333) but yet acceptable somehow. But let's see the result would stay the same if we change the link failure probability. The results until now, were for link failure probability of 50 percent. Let's see the results for link failure probability of 25 percent, three process, complete graph, 3 rounds and key equal to one. The results are as follows:



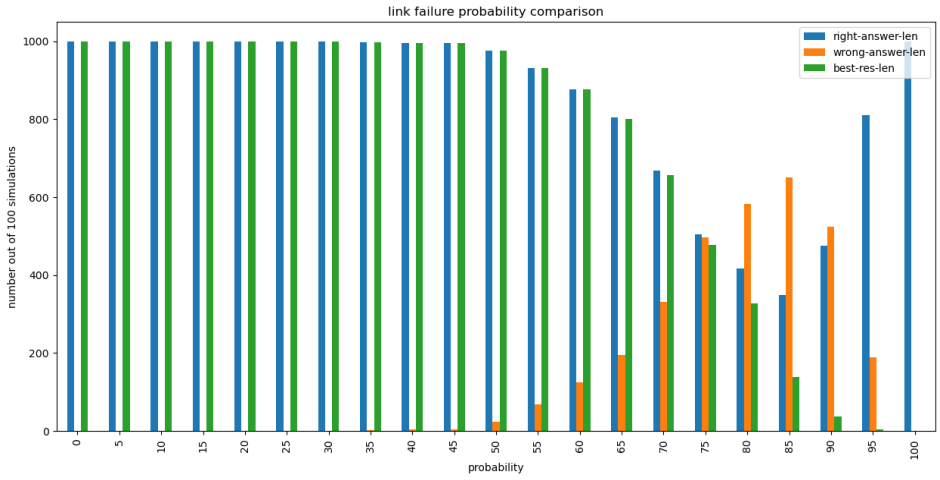
As you can see in the results, the probability for each round of simulation (number of times out of 100 simulations at each round comparing results of 500 simulations) we can see these numbers are not even close to 1/3 or 33.3. there mean is 1.338. but why? Shouldn't be 1/3 or 33.3? the answer is **definitely not**. Because that 1/r probability was for the case that at the end we know we had information levels between two different numbers and the probability that at the beginning we chose that number, is 1/r but what about the probability of having those two numbers at the end in information level? What about the probability of link failures which produces different results? In that time, we didn't consider that and we just say we have a number such that our agreement might be broken, but we didn't specify the probability of having those numbers as information levels at the end. So, as you saw in the first figure, the probability of having wrong answer is very few.

4- Link failure probability:

In this section, we want to see the results of different link failures probability on the simulation results. For this case we consider the simulation for three processes, five rounds of at each execution of algorithm (at each simulation), initial values of ones (worst case that in which we can have wrong answer), simulation number equal to 1000, and for link probabilities of a list which is from zero percent to 100 percent moving each 5 percent.[1] We run the simulation and store the values from the simulation result. We save: correct answers, wrong answers and also best answers. What are best answers? The thing that I myself defined it. It means that when we start with one as initial values of other processes, we get to the result decision of one as final decision or not. in other words, it means that how many times we had the result of one as final decision for all processes. With this declaration, let's see the results. The result of this simulation is as follows:

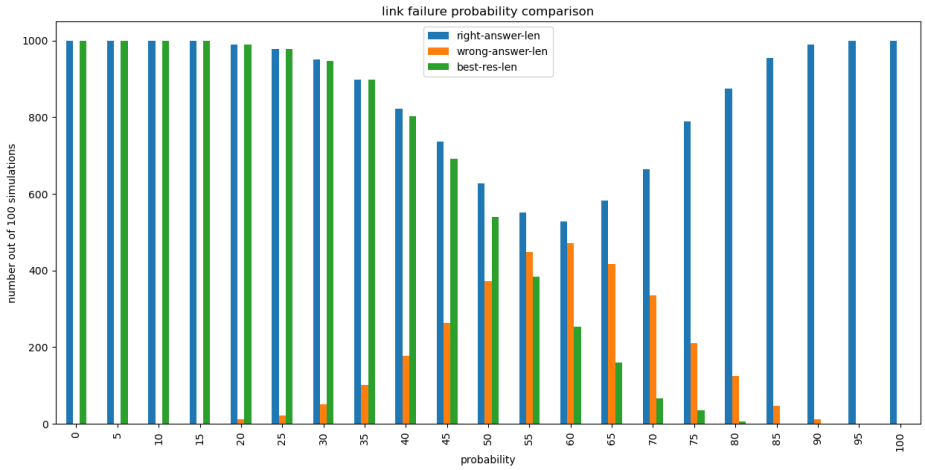


As you can see, we can't say that with increasing the link failure probability, we will always have more wrong answers, because as you can see in the above figure, from zero percent of having link failure, to having link failures f 85 percent, number of wrong answers increases, but from 85 percent to 100 percent, number of wrong answers decreases. We can run the simulation with fixed key and compare the results to see if it is different or not. by setting the key to one, we have the following figure:

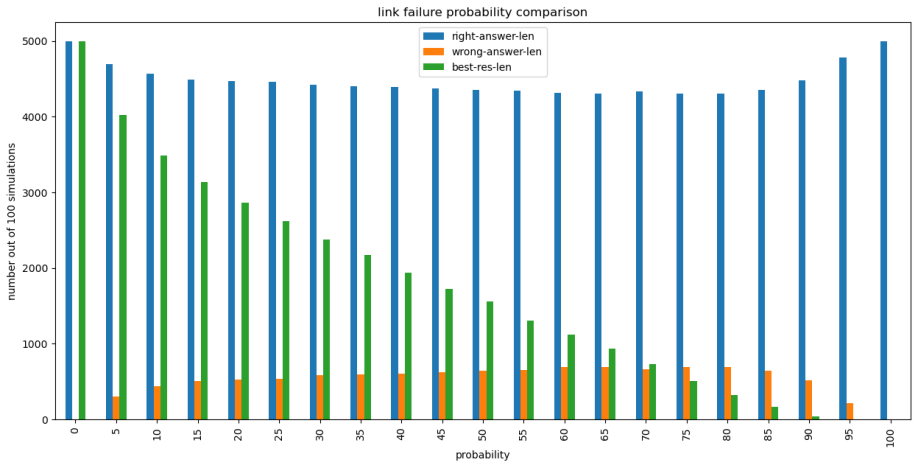


[1] For example, the list is like: [0, 5, 10, 15, 20, ..., 100]

The results of simulating for fixed key equal to two is as follows:



Now let's apply these simulations for all possible keys and combine them together to check whether our results are true or not. the following figure is the result of applying all possible keys along with all probabilities. The result is as follows:



As you can see the result is same as the case that we didn't do the simulation for a specific key. So, our simulations are correct.

There also is another notebook file, *main.ipynb*, in next to the previous, which shows the algorithm and simulation for different type of graphs, for example for directed graphs, for graphs that are not complete and …. And simulate different scenarios and save the results if we want to. But due to big amount of data, these files are not uploaded and simulations are only for *probability-comparison.ipynb* files. Some of the above figures was the result of running the notebook cells with different values and because of it, there are some plots and data frames that are not in the *prob-comp-res* folder but they have been generated using this code. And because of this, at the end of writing this document, I rerun the notebook and values in that folder are these values and might be different from figures. But they were generated by this code.

References:

- Slides
- Class
- Search