Summary

Quantum Key Distribution Simulation: BB84 Protocol 2
Idea 2
Implementation 2
BB84 Protocol 3
Intro 3
What is QKD? 3
Classical Cryptography 3
Basic principle of BB84 protocol 3
The model 3
The method 4
How is the key distributed in ideal conditions? 4
Table <a>/<b>/qubit 4
What happens when a malicious person tries to eavesdrop? 5
Documentation 7
Server 7
Client 7
Alice 8
Bob 9
Eve 9
BB84lib 9
CUlib 10
Usage 12
Prerequisites 12
General 12
Main Rules 12
How to interact with CLI 12
Example 1: QKD between Alice and Bob in ideal conditions 12
Example 2: QKD between Alice and Bob with the presence of eavesdropper Eve 14
Appendix 17
Appendix A: Acronyms and Notations 17
Appendix B: Tests 17

Quantum Key Distribution Simulation: BB84 Protocol

Project by Manuel Maiuolo

Idea

To simulate the example used in the explanation of the BB84 Quantum Key Distribution protocol during the lecture.
Two people, Alice and Bob, create a secret key that will be used for encryption and decryption of future messages.
The simulation will also consider the possible presence of a malicious person, Eve, that will try to eavesdrop; it will be difficult for her to intercept the key without leaving a trace.

Implementation

The simulation will be implemented in Python code.
To build this simulation, I will leverage socket programming, a technique I explored in the "Fondamenti di Comunicazioni e Internet" course at Politecnico di Milano.
The code will be organized into the following distinct files, each with a specific role:

File name	From this point onwards referred to as:
BB84_server.py	Server
BB84_client.py	Client
BB84_Alice.py	Alice
BB84_Bob.py	Bob
BB84_Eve.py	Eve
BB84lib.py	BB84lib
CUlib.py	CUlib

BB84 Protocol

Intro

What is QKD?

Quantum Key Distribution (QKD) is a secure communication method linked to quantum cryptography; it relies on principles from quantum mechanics. To understand QKD better, let's first explore the basics of classical cryptography.

Classical Cryptography

Cryptography is a method of storing and transmitting data in a particular form so that only those for whom it is intended can read and process it. Classical cryptography has two major branches: secret (or symmetric) key cryptography and public (or asymmetric) key distribution.
In secret-key cryptography, both parties share a single secret key for encryption and decryption, but securely sharing this key beforehand is a major challenge.
Public-key cryptography uses key pairs (public and private) – data encrypted with the public key can only be decrypted with the corresponding private key, and vice versa. While convenient, public-key cryptography relies on the difficulty of solving mathematical problems like integer factorization, a vulnerability that could be exploited by advancements in computing power, especially with the rise of quantum computers and algorithms like Shor's algorithm.

Basic principle of BB84 protocol

In 19**84** Charles **B**ennett and Gilles **B**rassard published the **BB84 protocol**. It relies on principles of quantum mechanics, in particular the superposition of qubits and the No Cloning Theorem.
This theorem states that it is impossible to create identical copies of an unknown quantum state (e.g. the qubits).
This unique property allows sender and receiver to detect eavesdropping, because (almost) any attempt to intercept the message will alter its quantum state and be noticeable.

The model

The basic model to understand the protocol consists of two parties, referred to as Alice and Bob, having access to both a public quantum channel (QC) and a public classical channel (CC).
The QC involves sharing a secret key by exchanging quantum particles (e.g. photons) represented by qubits.
The CC involves basis reconciliation and detection of possible eavesdroppers.

We assume that an eavesdropper, referred to as Eve, can access both channels.

The method

In the BB84 protocol, Alice can transmit a random secret key to Bob by sending a string of qubits with the private key encoded in their orientation. The No Cloning Theorem guarantees that Eve cannot measure these qubits and transmit them to Bob without disturbing the qubits' state in a detectable way.
At the end, Alice and Bob will have the same key if there is no eavesdropper. Otherwise, even one discordance in the key is a signal of eavesdropping.
In case no eavesdropper is detected, the key will be used for encryption and decryption of future messages.

How is the key distributed in ideal conditions?

In the first step, Alice and Bob communicate over a QC.
Alice randomly selects a string of bits <a> and a string of basis <b> (computational basis, referred to as Z, or Hadamard / rectilinear basis, referred to as X) of equal length.
Then she prepares a qubit for each bit of <a> with the corresponding orientation encoded in <b>, following this table:

Table <a>/<b>/qubit

Bit in <a>	Bit in <b> [basis]	Correlated qubit
0	0 [Z]	|0>
1	0 [Z]	|1>
0	1 [X]	|+>
1	1 [X]	|->

Then Alice transmits the quantum states (the qubits) through the QC to Bob.
Bob randomly chooses a basis for each qubit to measure its orientation. If Bob selects the same basis as Alice for a particular qubit, he will correctly find the bit Alice wanted to share as he measured the same orientation. If he doesn't guess correctly, he will get a random bit, because at the measurement the qubit will collapse to one of the possible values with the basis of chosen by Bob.
In the second step, Alice and Bob communicate over the CC.
Bob tells Alice the bases he used to measure each qubit, and vice versa. So: strings <b> and <b'> are announced on the CC.
After that, Alice and Bob remove the encoded and measured bits on different basis. Now, Alice and Bob have an identical bit-string: <a>

and <a'>. This can be used as the key for encryption and decryption of future messages.
To check the presence of Eve, Alice and Bob can share a few bits from the key, which are supposed to be the same. Any disagreement in the compared bits will expose the presence of Eve.
Following is an example where Eve is eavesdropping on the qubits used in the QC.

What happens when a malicious person tries to eavesdrop?

Eve wants to discover the secret key. Suppose that she has access to both the QC and the CC.
A necessary condition for discovering the key is that Eve eavesdrops on the qubits, then resends them to Bob to go unnoticed.
However, Eve cannot copy and resend the qubits to Bob because of the No Cloning Theorem: she must measure the quantum states.
In the absence of knowledge regarding Alice's selection of the qubits' basis, Eve must choose a random basis for each measurement of the quantum state.
So, qubits sent by Alice may be disturbed. In particular: if Eve uses a different basis for a qubit, and Bob measures in the same basis as Alice (so different from Eve), also Bob will have a random result, that may be different from Alice's, but measured in the same basis as Alice prepared! Following are some examples, to clarify ideas:

Alice's sent qubit [basis used]	Eve's measured qubit [basis used]	Bob's measured qubit [basis used]	Can Eve be detected?
|0> [Z]	|0> [Z]	|0> [Z]	No
|0> [Z]	|+> [X]	|0> [Z]	No
|0> [Z]	|+> [X]	|1> [Z]	Yes

To detect if Eve is eavesdropping, Alice and Bob can share a few bits from the key, which are supposed to be the same. Any disagreement in the compared bits will expose the presence of Eve.
Called $|check|$ the number of bits from <a> and <a'> used as control, the following is possible.
In one qubit system ($|check| = 1$), the probability that the eavesdropper is detected is $\frac{1}{4}$ in ideal case.
This is because:

1) Alice and Bob used the same basis. Otherwise, the corresponding bits would have been discarded after the announcement of strings <b> and <b'> in the CC.
2) Eve has $\frac{1}{2}$ probability of choosing the wrong basis.

3) Bob has $\frac{1}{2}$ probability of choosing the same qubit as Alice, knowing (1).

So: the probability of not detecting Eve is $p = 1 - \frac{1}{4} = \frac{3}{4}$.

To generalize: the probability of detecting Eve is $p = 1 - \left(\frac{3}{4}\right)^{|check|}$.

Some significant cases:

|check|	**probability of detecting Eve is greater than**
25	99,9%
17	99%
11	95%
9	90%
6	80%
3	50%

Note that $|check|$ is upper bounded by the number of bits left in the strings <a> and <a'> after the process of discarding the bits correlated to qubits where Bob measured on a different basis than Alice prepared.

Documentation

For each file there is a list of functions and classes.
For each class are described important fields (▫) and methods (+ and −).

Server

- **class ServerActions**:

The possible actions for the server are stored here.

- **class BB84Server**:
  + **alice_request(request_type, request_info)**:
  Passed as input one of Alice's possible actions (from AliceActions) as request_type *and the parameters about it as* request_info*, server performs the requested simulation (e.g. the QC or CC).*
  bob_request and **eve_request** are analogous to alice_request.

  + **start()**:

Makes the instance of server start.

  − **__send_b()**:
  Makes Alice and Bob announce the strings <b> and <b'>.

  − **__detect_eavesdropping()**:
  After user enters |check|, asks Alice and Bob |check| bits from strings <a> and <a'> in the same positions (randomly chosen). Then confronts the requested bits and prints conclusions about the detection of eavesdropping (and the probability of its correctness).

  + **handle_input()**:

Manages Server's action input from CLI.

  + **handle_client(client_name, client_info)**:
  After a new client is connected, calling this method allows to communicate with it.

Client

- **class BB84Client**:
  − **__init__(client_name, function_handle_response, menu_functions)**:
  Initializes the client with name client_name *(can be 'Alice', 'Bob' or 'Eve'). Furthermore, it sets:*

→ function_handle_response. *This is the function to call when server sends a message to the client.*
→ menu_functions, *that must be the following tuple:(*menu_max_choices, menu_choice, show_menu*).*
- menu_max_choices *is the number of choices in the menu's actions' enumeration.*
- menu_choice(choice) *is the function to call when an action is done for the client;* choice *will be the number correlated to the action chosen.*
- show_menu() *is the function that shows the menu of the client.*

+ **connect()**:
Connects the client to the server.

+ **disconnect(log)**:
Prints the message log *in the client CLI and disconnects the client from the server.*

+ **handle_menu()**:
Handles menu input. When the program does not allow input, it ignores it.

+ **handle_responses()**:
Handles the messages from the server.

Alice

- **class AliceActions**:
The possible actions for Alice are stored here.

- **class Alice(BB84Client)**:
  - **__generate_bits()**:
  After user enters 'n' (the length of <a> and <b>), the strings <a> and <b> are randomly generated and stored in class fields.

  - **__prepare_qubits()**:
  Basis are calculated according to string <b>. Then, according to desired (random) basis and to the string <a>, the qubits are prepared.

  - **__receive_b1(b1)**:
  Receives string <b'>, then discards qubits (and correlated bits in <a> and <b>) where Bob measured in different basis than Alice prepared.

  - **__send_some_a(req_a)**:

Server requested for some bits of <a> to try to detect eavesdropping. req_a *will consist in a string where each character is 'x' if the correlated bit in <a> is not requested from the server, '?' otherwise. This method sends to the server a string like* req_a*, where '?' are replaced with the bits in <a>.*

Bob

- **class BobActions**:

The possible actions for Bob are stored here.

- **class Bob(BB84Client)**:
  - **__set_receiving_qubits_rate()**:
  Makes user interact with CLI to set the rate at which to show received qubits from Alice.

  - **__receive_qubits(qubits_str)**:
  *Receives qubits (sender: Alice) as compact string from server (*qubits_str*), then creates string <b'> and correlated list of basis, performs qubits measurements and based on the results creates the string <a'>.*

  → **__receive_b(b)** is analogous to Alice's **__receive_b1(b1).**
  → **__send_some_a1(req_a1)** is analogous to Alice's **__send_some_a(req_a).**

Eve

- **class EveActions**:

The possible actions for Eve are stored here.

- **class Eve(BB84Client)**:
  - **__set_receiving_qubits_rate()**:
  Makes user interact with CLI to set the rate at which to show eavesdropped qubits from Alice.

  → **__receive_qubits(qubits_str)** is analogous to Bob's **__receive_qubits(qubits_str).**

Note: the difference between Eve and Bob is that in Eve's method handle_response the measured qubits are afterwards sent to Bob.

BB84lib

- **class Basis**:
  - **value**:

Basis character is stored in this field.

- **__init__(value = None)**:
Initializes the basis to 'Z' (computational) or 'X' (rectilinear). If value *is not set or not valid, computational basis is set as default.*

- **__str__()**:
Defines how the basis should be shown: by the character 'Z' or 'X'.

+ **set_from_b(b)**:
Defines the value of the basis for a particular bit in <b> string: 'Z' if b *is 0, 'X' if* b *is 1.*

- **class Qubit**:
  - **value**:
The character representing the qubit ('0' and '1' for computational basis, '+' and '-' for rectilinear basis) is stored in this field.

- **__init__(value = None)**:
Initializes the qubit to '0', '1', '+' or '-'. If value *is not set or not valid, '0' is set as default.*

- **__str__()**:
Defines how the qubit should be shown: by ket notation. E.g. if value is 0, it is shown |0>.

+ **set_from_a_and_basis(a, basis)**:
Given the bit a *from string <a> and the polarization* basis *(from b), the value is set accordingly to Table <a>/<b>/qubit.*

+ **measure(basis)**:
Simulates qubit measurement: the measurement is simulated with the same orientation as the basis *passed as argument. If orientation is the same as the qubit's: the qubit collapses on the same value, so no change is applied. Otherwise, the qubit randomly collapses to a random value on the other basis's orientation.*
Then returns the collapsed value of the qubit.

CUlib

- **input_int(minVal, maxVal, errorSentence='')**:

*Loops input until a valid integer value in [*minVal..maxVal*] is obtained, then returns it.*
Every time the input is not valid, errorSentence *is shown.*

- **receive(connection_socket)**:

Waits for message from connection_socket*, then returns it.*

- **send(connection_socket, message)**:

Sends message *to* connection_socket.

- **clear()**:

Clears CLI's screen, valid as 'cls' command for Windows and as 'clear' command for Mac and Linux.

- **set_title(title)**:

Sets CLI's title as title*, valid both for Windows and Mac / Linux.*

- **print_in_box(lines)**:

Given the list lines*, prints list elements as lines in a box.*

- **print_in_table(rows, min_cols=0)**:

Prints each list in the parent-list rows *as row in a table: each element in the row-list will be put in a different column.*
If in a particular row there are less elements than min_cols*, then the row is filled with empty cells to get the minimum number of columns required.*

- **print_menu_options(structure)**:

Given the list structure, prints list elements as numbered options in a menu.

Usage

Prerequisites

- A device that can run Python codes.

To download Python it's possible to visit the official webpage: https://www.python.org/downloads/.
After having a device that can run Python codes, to run a script it should be enough to just double-click on the desired file with '.py' extension.

General

Main Rules

- Use Server as the main manager of the project; execute it first.
- Use Alice and Bob to make the codes worth running.
- Use Eve too if an eavesdropper is desired in the simulation.
- Do not execute: Client, BB84lib and CUlib.

How to interact with CLI

1) Read the CLI.
2) A list of actions is enumerated almost always on top: choose what action should be performed from that CLI.
3) Input the number correlated to the desired action and press Enter.
4) {Possibly} a direct interaction between the user and the CLI is proposed: please, follow the instructions shown.

Example 1: QKD between Alice and Bob in ideal conditions

1) Execute Server, then Alice and Bob.
2) From Alice: action [1] > generate two random strings of n-bits: a, b.
3) Choose the length 'n' for the strings of bits <a> and <b>: it will be the number of qubits simulated for the QKD BB84 protocol. It must be an integer between 0 and 50; if 0: <a> and <b> will be set empty, else: Alice will randomly generate the two strings with the desired length. From Alice: input the number, then press Enter.
4) {Possibly} if the table of Alice's information is not shown properly, from Alice: action [5] > change how Alice's current information is shown. This will show Alice's information in compact method, so the ket notation for qubits will be omitted and there will be no spaces between the bits.
5) From Alice: action [2] > prepare n qubits accordingly to a and b.

6) {Optional} From Bob: action [1] > set the rate at which to show received qubits from Alice. Then, enter the number correlated to one of the options shown. {Note: default is ‘fast’}.
7) {Possibly} if the table of Bob's information will not be shown properly (this will happen if Alice’s information has not been shown properly in non-compact method), from Bob: action [3] > change how Bob's current information is shown. This will show Bob’s information in compact method, so the ket notation for qubits will be omitted and there will be no spaces between the bits.
8) From Alice: action [3] > send quantum state to Bob via public quantum channel.

If in step 6 the rate has not been set to ‘immediate’:
- in Alice’s window should appear “[Simulation in progress]”
- In Bob’s window should appear a simulation of the following process. For each qubit received:
    - A random bit for string <b’> is generated.
    - The bit is used as the encoding for the base to use for the measurement.
    - The received qubit is measured in the randomly chosen base.

  Due to quantum effects:
  - The qubit will not change if the base chosen for the measurement is the same as the base used by Alice to encode the qubit itself.
  - The qubit will randomly collapse to a new state otherwise.

When in Alice’s window and in Bob’s window the main menu is shown (with the latest information): the simulation is finished.
At this point, both Alice and Bob have all the information (strings <a>, <b>, <a'> and <b'>) synchronized. The last step is to "prune" the strings <a> and <a'>, discarding the bits correlated to qubits where Bob measured on a different basis than Alice prepared. To do this, the simulation proceeds as follows:

9) From Server: action [1] > make Alice and Bob announce the strings b and b' via public classical channel.

At this point, Alice and Bob have the same key: <a> = <a’>

10) {Optional} Clear CLI screen for Alice and Bob:
    a. From Alice: action [4] > clear the CLI screen.
    b. From Bob: action [2] > clear the CLI screen.
11) From Server: action [2] > try to detect the presence of Eve thanks to a possible inconsistency in the strings <a> and <a'>.
12) Choose the number of bits from <a> and <a’> to share, to detect the presence of Eve. Since in this example Eve is not

present, the strings <a> and <a’> after step 9 will always be equals. It must be an integer between 1 and the number of bits left in the strings <a> and <a’>. Input the number chosen. From this point onwards this number will be referred to as |check|.

Server will randomly select |check| positions and will communicate with Alice and Bob to get bits in the strings <a> and <a’> at those selected positions. The conclusion about detection of eavesdropping is shown in Server’s window.

Example 2: QKD between Alice and Bob with the presence of eavesdropper Eve

1) Execute Server, then Alice, Bob and Eve.
2) From Alice: action [1] > generate two random strings of n-bits: a, b.
3) Choose the length ‘n’ for the strings of bits <a> and <b>: it will be the number of qubits simulated for the QKD BB84 protocol. It must be an integer between 0 and 50; if 0: <a> and <b> will be set empty, else: Alice will randomly generate the two strings with the desired length. From Alice: Input the number, then press Enter.
4) {Possibly} if the table of Alice’s information is not shown properly, from Alice: action [5] > change how Alice's current information is shown. This will show Alice’s information in compact method, so the ket notation for qubits will be omitted and there will be no spaces between the bits.
5) From Alice: action [2] > prepare n qubits accordingly to a and b.
6) {Optional}
   a. From Eve: action [1] > set the rate at which to show eavesdropped qubits from Alice. Then, enter the number correlated to one of the options shown. {Note: default is ‘fast’}.
7) From Bob: action [1] > set the rate at which to show received qubits from Alice. Then, enter the number correlated to one of the options shown. {Note: default is ‘fast’}.
8) {Possibly} if the table of Eve's information and the one of Bob’s will not be shown properly (this will happen if Alice’s information has not been shown properly in non-compact method):
   a. from Eve: action [3] > change how Eve's current information is shown. This will show Eve’s information in compact method, so the ket notation for qubits will be omitted and there will be no spaces between the bits.

   b. from Bob: action [3] > change how Bob's current information is shown. This will show Bob’s information in compact method, so the ket notation for qubits will be omitted and there will be no spaces between the bits.

9) From Alice: action [3] > send quantum state to Bob via public quantum channel.

If in step 6a rate has not been set to ‘immediate’:

- in Alice’s and Bob’s windows should appear “[Simulation in progress]”
- In Eve’s window a simulation of the following process should appear. For each qubit received:
  - A random bit for string <b> for Eve is generated.
  - The bit is used as the encoding for the base to use for the measurement.
  - The received qubit is measured in the randomly chosen base.

  Due to quantum effects:
  - The qubit will not change if the base chosen for the measurement is the same as the base used by Alice to encode the qubit itself.
  - The qubit will randomly collapse to a new state otherwise.

When in Eve’s window the main menu is shown (with the latest information): the simulation of eavesdropping is almost complete. Eve must send the measured qubits to Bob; the action is done automatically.

Indeed, the same simulation should appear in Bob’s window, if in step 6b rate has not been set to ‘immediate’.

When in Alice’s window and in Bob’s window the main menu is shown (with the latest information): the simulation is finished.
At this point, both Alice and Bob have all the information (strings <a>, <b>, <a'> and <b'>) synchronized, with the effects of eavesdropping. The last step is to "prune" the strings <a> and <a'>, discarding the bits correlated to qubits where Bob measured on a different basis than Alice prepared. To do this, the simulation proceeds as follows:

10) From Server: action [1] > make Alice and Bob announce the strings b and b' via public classical channel.
11) At this point, Alice and Bob should have the same key: <a> = <a’>. However, the effects of eavesdropping can lead to <a> ≠ <a’> for some bit. Any difference in the key is a signal of eavesdropping, however having no differences does not imply that Eve is not eavesdropping.
12) {Optional} Clear CLI screen for Alice and Bob:
   a. From Alice: action [4] > clear the CLI screen.

b. From Bob: action [2] > clear the CLI screen.

13) From Server: action [2] > try to detect the presence of Eve thanks to a possible inconsistency in the strings <a> and <a'>.

14) Choose the number of bits from <a> and <a’> to share, to detect the presence of Eve. Since in this example Eve is present, the string <a> could be different from <a’> even after step 10. The number must be an integer between 1 and the number of bits left in the strings <a> and <a’>. Input the number chosen. From this point onwards this number will be referred to as |check|.

Server will randomly select |check| positions and will communicate with Alice and Bob to get bits in the strings <a> and <a’> at those selected positions. The conclusion about detection of eavesdropping is shown in Server’s window.

Appendix

Appendix A: Acronyms and Notations

- ***QKD***

Quantum Key Distribution.

- ***QC***

(public) Quantum Channel.

- ***CC***

(public) Classical Channel.

- ***Z***

Character used to represent computational basis.

- ***X***

Character used to represent Hadamard / rectilinear basis.

- ***From XYZ: action [w]***

Focus on *XYZ*'s window, then write *w*, then press Enter.

- ***CLI / window***

Command Line Interface window.
Used to identify the "program" opened when executing one of the Python files (e.g. BB84_server.py).

Appendix B: Tests

The tests done have been taken under these conditions:

- **Operating System**: *Microsoft Windows [Version 10.0.19045.4291]*
- **Python version**: *3.10.2*