Running the project

How to run the project

You will have to open 4 terminals in the root of the project.

You will have to enter the following command in each terminal, where you replace x with the port number you wish for the shell to use.

This number have to be in the range of `[0..3]`

```
go run . x
```

Example:

```
Terminal_1: go run . 0
Terminal_2: go run . 1
Terminal_3: go run . 2
Terminal_4: go run . 3
```

The program will not start before all shells have been opened.

Terminal_1 on port 0 will always represent the hospital and the rest will represent the patients (alice,bob,charlie).

When everything is started correctly. You will be able to send messages from each patient by typing in the terminal.

These messages will represent the secret value that needs to be aggregated.

When every patient has put in a value, the algorithm will start and the hospital will receive the aggregated values.

The hospital will sum the send values into a total sum and the program will terminate.

To ease the pain of opening the terminals i have provided

- A powershell script that is **guaranteed** to work on windows. The terminal output will be logged in logs/textx (with x being the port number)
- A bash script that **might** to work on macos/linux. The terminal output will be logged in logs/textx (with x being the port number). <- This might not work

Note: An example of the output i have generated myself can already be found in the logs directory.

Compiling the project

In case that the project doesn't run, then you might need to compile it before running it.

You need to have go installed to compile the project. There might also be a dependency on protobuffers. (all these can be installed using a package manager)

Open the root of the project folder in your favorite terminal shell and enter the following commands

```
go mod tidy
```

```
protoc --go_out=. --go_opt=paths=source_relative --go-grpc_out=. --go-
grpc_opt=paths=source_relative MPC/MPC.proto
```

Now the project should be compiled and you can run it using the guide above.

If the certificates aren't available you have to follow the guide in ./certificate/how_to_generate.md on how to generate them.

Adversarial model

For this algorithm to work, we have to assume that

- The parties in involved are:
  - Semi-honest and will follow the protocol established by the hospital.
  - With an honest majority meaning that the adversary may only corrupt a minority of the parties.
  - Static adversary meaning that an adversary must choose which party to corrupt before the protocol execution starts.
- The presence of a Dolev-yao adversary As the information is transfered over an insecure network I.E the internet.
- Further that the responsibility to sum the total value across all patients into a single value can be delegated to the hospital.

> **Note:**
> If we don't trust the hospital to do this, we would have to do another run of the value exchange between the patients.
>
> - This would result in the patients having all the aggregated values, which they can sum and send to the hospital.
> - The hospital would then receive the total sum from n parties, where n is the amount of patients.

Building blocks

This program is made using peer to peer . Meaning the parties will act as both the server and the client. We will in the implementation also treat both the patients and the hospital as a client.

When i use "peer" in the following explanation it will refer to both the patients and the hospital and will be used to explain a general concept of the implementation.

I will use patient and hospital respectively when addressing a specific peer type and their behavior in the program.

TLS

We use TLS to avoid message tampering and to keep all communication private between the peers. This will be our best protection against a dolev-yao adversary.

In order to do this I generate a certificate using OpenSSL. I sign it using localhost. This however is not ideal and it would be best to use a proper certificate authority. For this assignment it is however okay to use localhost.

In the case that we did want to use a proper certificate authority we would need to do minimum code changes in the go file, such that we can handle the case where the certificate is not valid.

We would also need to generate a certificate for each peer, instead of the one certificate to rule them all implementation that we have right now.

We create our TLS configuration with the following code snippet

```
openssl req -nodes -x509 -sha256 -newkey rsa:4096 -keyout certificate/priv.key -
out certificate/server.crt -days 356 -subj
"/C=DK/ST=Copenhagen/L=Copenhagen/O=Me/OU=mpc/CN=localhost" -addext
"subjectAltName = DNS:localhost,IP:0.0.0.0"
```

Now that the configuration has been generated we can use it with the credentials library from the standard go library.

We use and declare our TLS configuration in go with the following code snippet

```
 serverCert, err := credentials.NewServerTLSFromFile("certificate/server.crt",
"certificate/priv.key")
 if err != nil {
  log.Fatalln("failed to create cert", err)
 }
```

This will be a peers own TLS record.

We will also need to create a new for each client peer we connect to.

```
 //Set up client connections with TLS
  clientCert, err := credentials.NewClientTLSFromFile("certificate/server.crt",
"")
  if err != nil {
   log.Fatalln("failed to create cert", err)
  }
```

```
  // Dial the server, and store the connection in the clients map
  conn, err := grpc.Dial(fmt.Sprintf("localhost:%v", port),
grpc.WithTransportCredentials(clientCert), grpc.WithBlock())
  if err != nil {
   log.Fatalf("Patient (ID: %v):did not connect: %s", p.id, err)
  }
```

This will establish a tls connection with all the other peers.

Now that the connection is established in a secure way. (apart from the fact that we use the same certificate for all the peers) we can let go handle the tls and communication implementation.

It has been abstracted away which is fine as TLS is not the focus of the assignment.

Secure multi-party computation

Chunking

The user can now input their "secret" value and the algorithm will run.

We first split this secret into chunks using the following function. We need the secret to be split into chunks equal to the total number of patients. Which in our case is 3 patients. One chunk for the patient themselves and one for each of the other patients. The hospital doesn't get one.

This number also has to be random generated, we do that using a non-negative pseudo-random number generator in the std go library, with the only rule that the total sum of the chunks should be equal to the secret.

```
func splitChunks(secret int, fromId int, n int) map[int32]int {
 // Split the secret into chunks, while avoiding floating point numbers but still
keeping the sum of the chunks equal to the secret
 n = n - 1 // Minus one because our implementation is 0 indexed
 remainder := secret
 chunks := make(map[int32]int)
 for i := 0; i < n; i++ {
  rando := 0
  if secret < n {
   // 0 is ok to send if the user has chosen a secret less than n
   // 0 can lead to an edge case where the secret value can be deduced by the
hospital/patients when every patient gets unlucky and sends 0 chunks to n-1
patients. It actually happens pretty often. Try inputting 1 from each patient.
   rando = rand.Intn(remainder)
  } else {
   // This is preferred but not usable if the value is less than n.
   // I probably should have used floating point numbers.
   rando = improved_rand(remainder)
  }
  chunks[int32((fromId + i))] = rando
  fmt.Printf("Iteration %v (for id:%v): Generated following chunk: %v from the
remainder: %v and our secret was: %v\n", i, fromId+i, rando, remainder, secret)
```

```
  remainder -= rando
 }
 fmt.Printf("Iteration %v (for id:%v): Parsing the remainder: %v and our secret
was: %v\n", n, fromId+n, remainder, secret)
 chunks[int32((fromId + n))] = remainder // The remainder gets parsed to the last
peer
 return chunks

}
```

Sending the chunks

After the chunking, a peer send a piece of the chunks to each patient. With the following logic

- A piece should only be send once
- A piece needs to be saved for the peer them self
- The hospital shouldn't get one

when the chunks have been send, we have to wait for the other patients to send their chunks otherwise we won't be able to continue.

Calculating and sending the aggregated value

When every patient has sent their chunks to each other, we can calculate the aggregated value by summing all saved chunks. (I.E our own chunk + the chunks received from the other patients) and as a final step send this value to the hospital.

Receiving the aggregated values

The hospital peer will remain idle throughout the program, until the patients have calculated and sent their aggregated values. The hospital will save each aggregated value and as a final step calculate the final sum of all the aggregated values.

```
func HospitalLogic(p *peer) {
 // The following logic is aimed at the hospital side
 scanner := bufio.NewScanner(os.Stdin)
 for (len(p.chunks)) < len(p.clients) {
  // Wait for all the patients to send their chunks. Using a channel to avoid race
conditions
  fmt.Printf("Hospital (ID: %v): Waiting for data from other patients. Got %v out
of %v\n", p.id, len(p.chunks), len(p.clients))
  result := <-p.channel
  fmt.Printf("Hospital (ID: %v): I just got result %v from id %v\n", p.id,
result.Result, result.Id)
  p.chunks[result.Id] = int(result.Result)
 }
 superChunk := int(0)
 for id, chunk := range p.chunks {
  fmt.Printf("Hospital (ID: %v): Patient (Id: %v) sent the following result:
%v\n", p.id, id, chunk)
```

```
    superChunk += chunk
   }
   fmt.Printf("Hospital (ID: %v): The total sum for all the patients is %v\n", p.id,
  superChunk)

   fmt.Print("Press any button to quit\n")
   scanner.Scan()
  }
```

Given our earlier assumptions we can guarantee that no adversary who corrupts only one party can recover the input of any other party since each party does not have all additive shares of other parties' inputs, which makes the protocol secure.

This also mean that neither curious patients can learn the secret value of the other patients this is also not possible for the hospital.

It is however possible to learn the sum of the patients secrets which is deemed as tolerable concession.

An example output has been generated for the program, and it is included in the logs folder. This is for convenience sake, but can be generated using the instructions above.