CS302

gRPC

Spring 2025

Arkaprava Basu & Babak Falsafi

parsa.epfl.ch/course-info/cs302

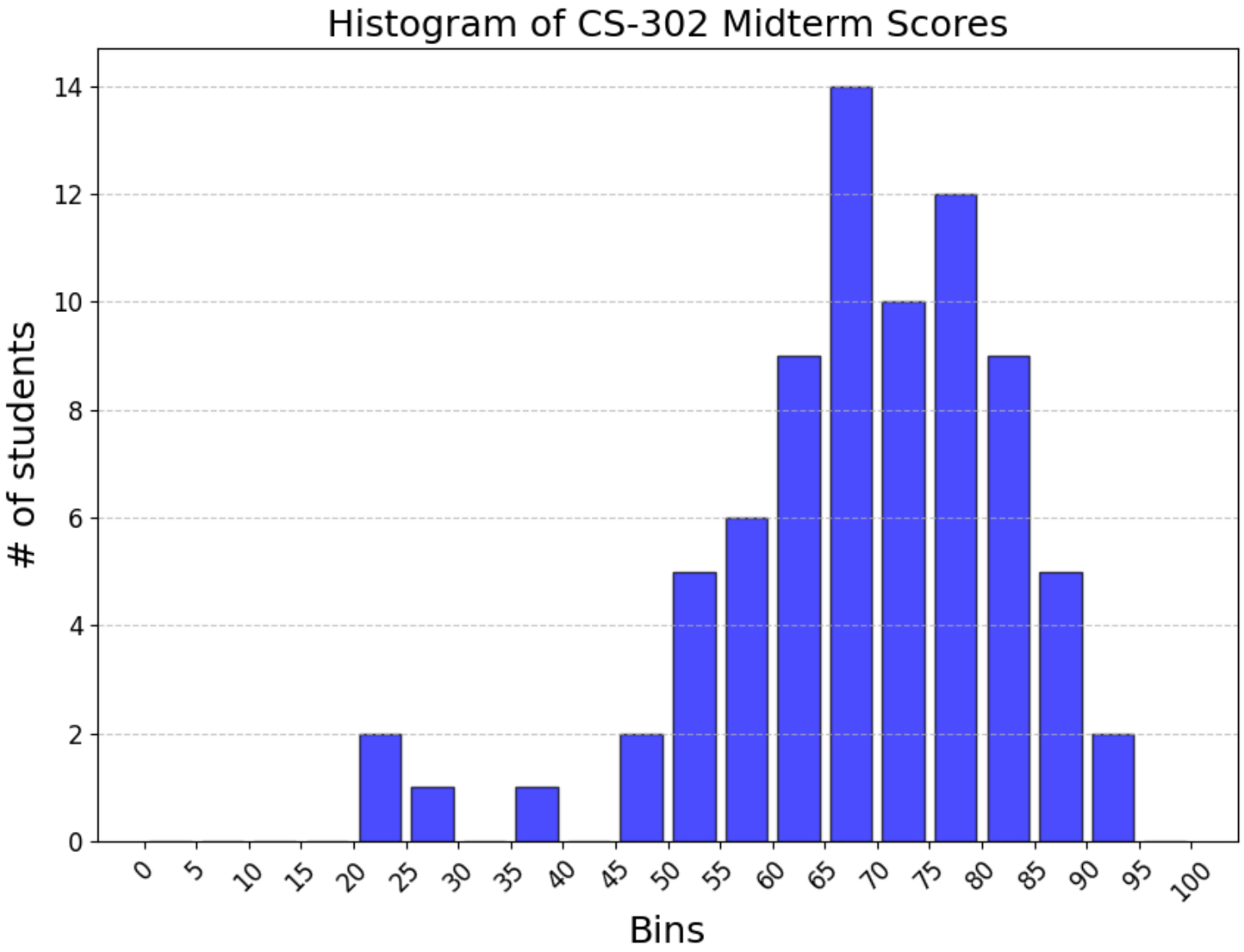
Adapted from slides originally developed by Profs. Falsafi, Fatahalian, Mowry, Wenisch of CMU, Michigan

Where are We?

M	T	W	T	F
17-Feb	18-Feb	19-Feb	20-Feb	21-Feb
24-Feb	25-Feb	26-Feb	27-Feb	28-Feb
3-Mar	4-Mar	5-Mar	6-Mar	7-Mar
10-Mar	11-Mar	12-Mar	13-Mar	14-Mar
17-Mar	18-Mar	19-Mar	20-Mar	21-Mar
24-Mar	25-Mar	26-Mar	27-Mar	28-Mar
31-Mar	1-Apr	2-Apr	3-Apr	4-Apr
7-Apr	8-Apr	9-Apr	10-Apr	11-Apr
14-Apr	15-Apr	16-Apr	17-Apr	18-Apr
21-Apr	22-Apr	23-Apr	24-Apr	25-Apr
	29-Apr	30-Apr	1-May	2-May
5-May	6-May	7-May	8-May	9-May
12-May	13-May	14-May	15-May	16-May
19-May	20-May	21-May	22-May	23-May
26-May	27-May	28-May	29-May	30-May

- RPC + gRPC
  - Basics
  - Example with “Hello World”
- Thursday
  - Lecture: HW Multithreading
  - Exercise session: Coroutines and RPC examples
- Next Tuesday:
  - Intro to GPUs

Midterm Scores



Average = 68

Maximum = 92

Std Dev = 14.2

Midterm Exam Sheet Viewing Sessions

- There are two exam sheet viewing sessions
  - **Tuesday April 29th 5 pm - 6 pm** in BC 229 (Today)
  - **Wednesday April 30th 5 pm - 6 pm** in BC 229
- Exam regrading request deadline: **April 30th 11:59 pm**
  - Let us know in-person at one of the exam viewing sessions or send an email
  - Must have a valid reason
- Exam solutions are on Moodle
- April 10th exercise session also went over the exam solutions
  - Recording link available on Moodle
- A1 grading will be done by end of the day today

Heads Up

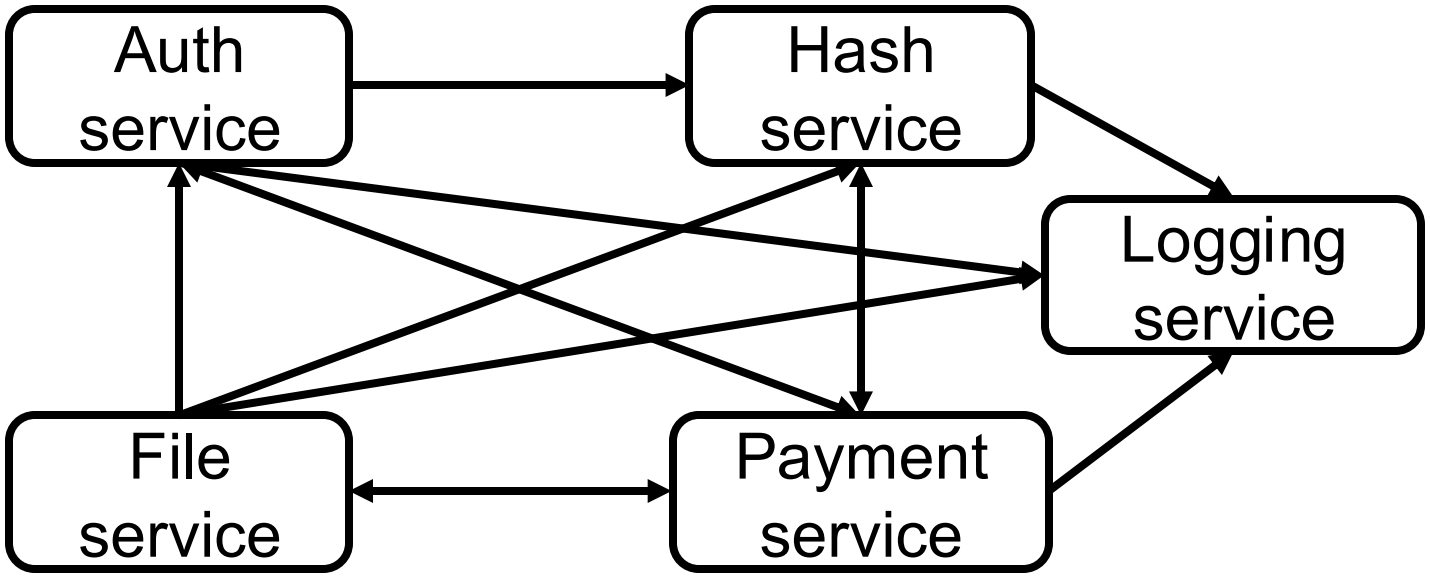
- Assignment 2 deadline is on May 4th 11:59 pm

- FAQs:
  - B1 and B2 are not used for progD (collectives do not transmit data in chunks)
  - If you have time limitations for experiments, you can measure the time taken for 5 iterations and multiply by 4 to estimate the time for 20 iterations and report it
  - Keep regular backups of your code!

- HW 6 is now available on Moodle
  - Deadline to submit: Monday May 5th 11:59pm

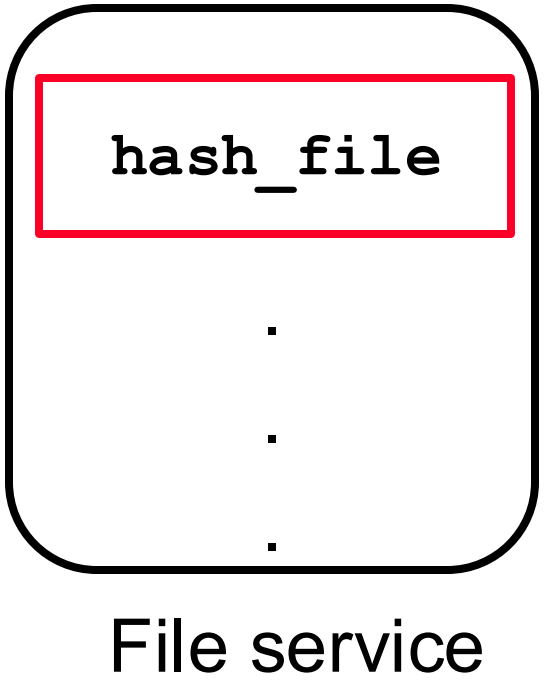
Recap: Microservices

- Microservice architecture: a set of small (micro), independent services
  - Isolated services
  - Scale services based on their needs
  - Heterogenous tech stack



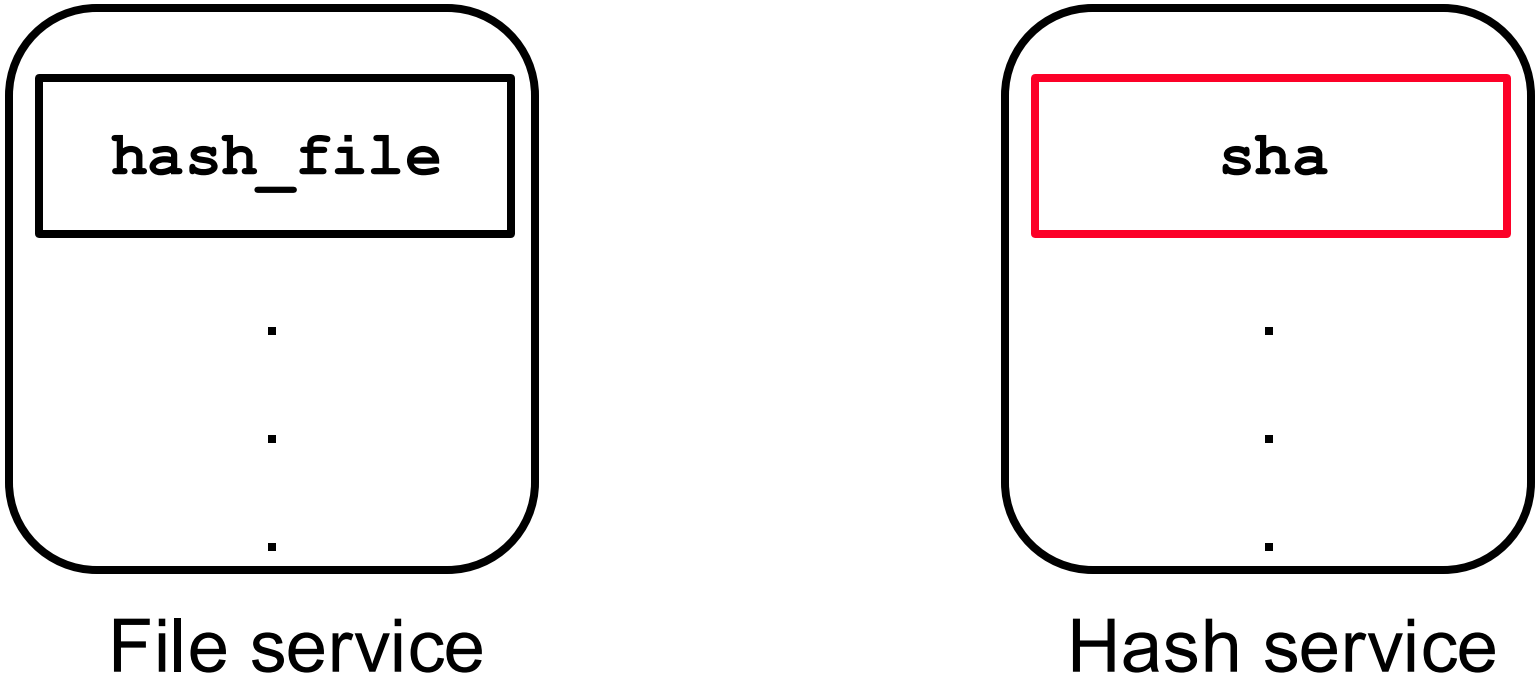
Recap: File & Hash Services

- File service
  - Serving file-handling related requests
  - **`hash_file`**: Hashes the content of a file



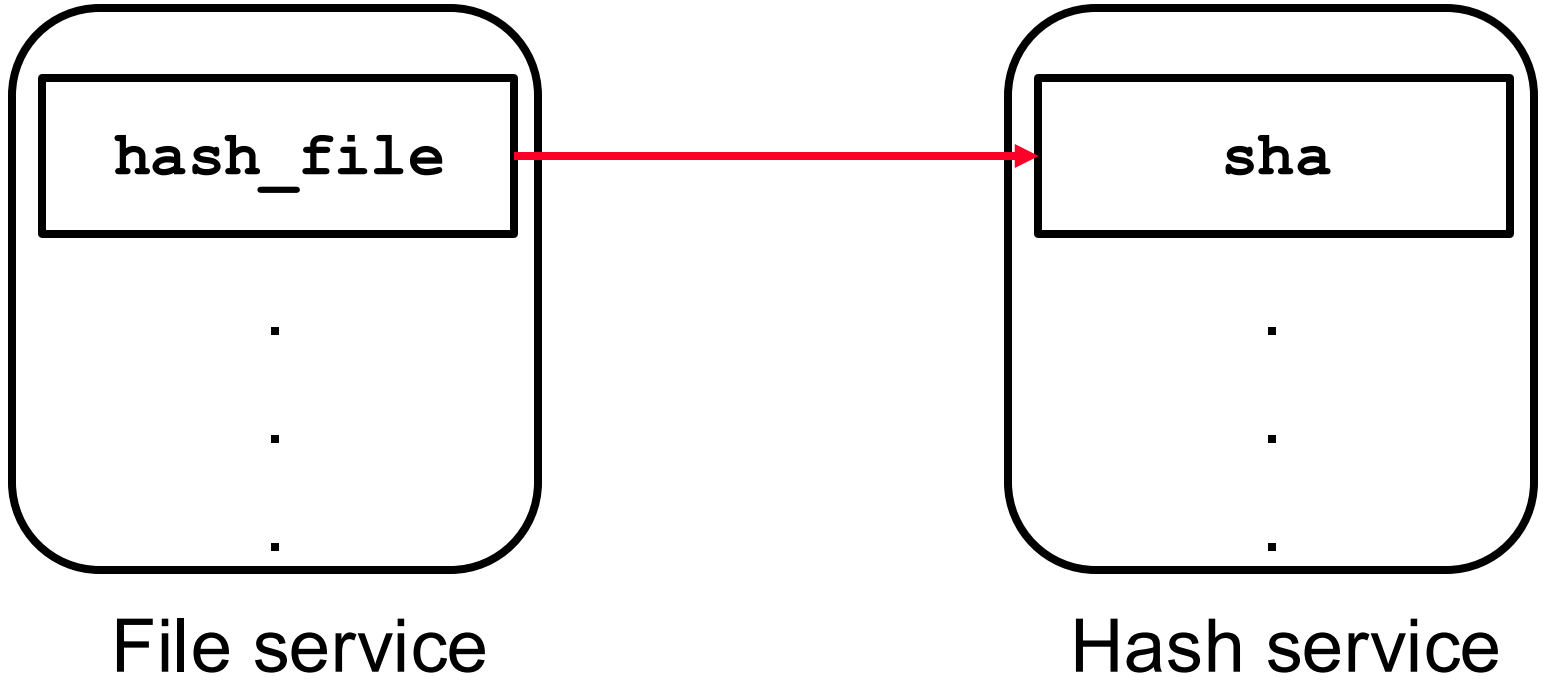
Recap: File & Hash Services

- Hash service
  - Serving hashing requests
  - **`sha`**: Computes hash



Recap: File & Hash Services

- Want to reuse **`sha`** in **`hash_file`**
  - Both file and hash service need **`sha`**'s definition
  - Need a common interface definition language (IDL)



Why IDL?

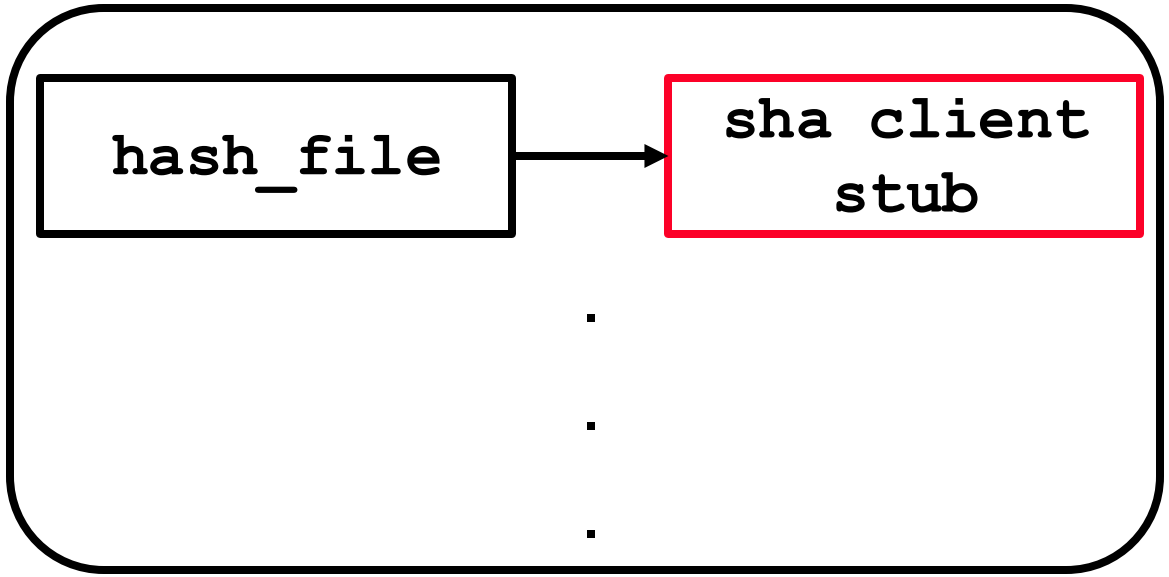
- Services can be in various languages
- Languages define functions differently
- Define using IDL once, generate code in different languages

```
# Python client

def sha(input: str)->str:
   ...
```

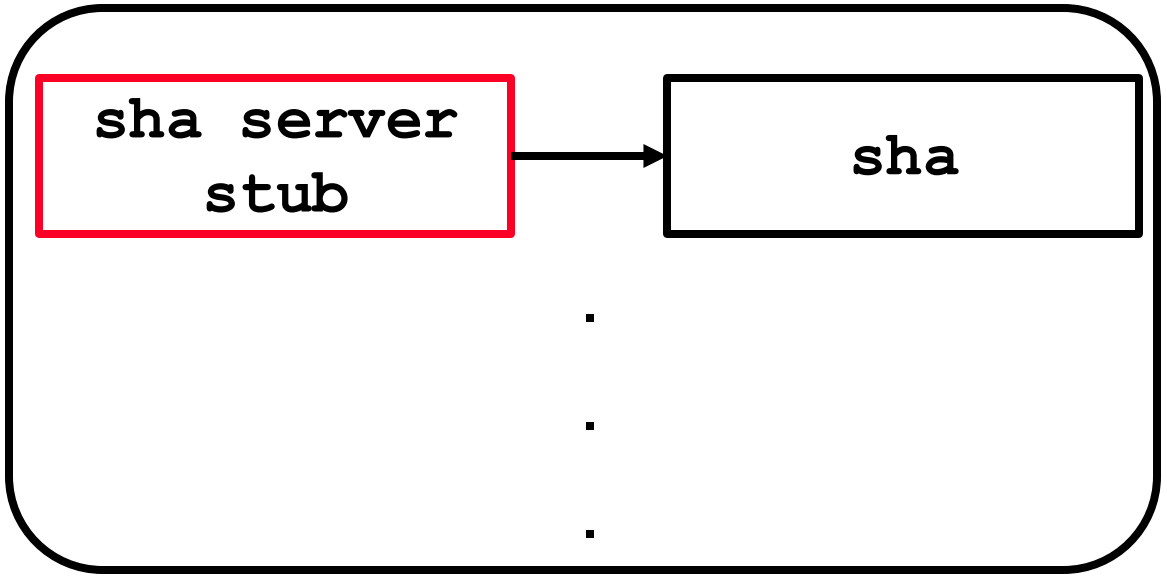
```
// C++ server

char* sha(char* input){
   ...
```

RPC Overview

- RPC generates code from interface definition language (IDL)
  - Client stub: Code generated for the service calling a function (client)
  - Server stub: Code generated for the service serving a function (server)
  - Client and server stub can be in different languages



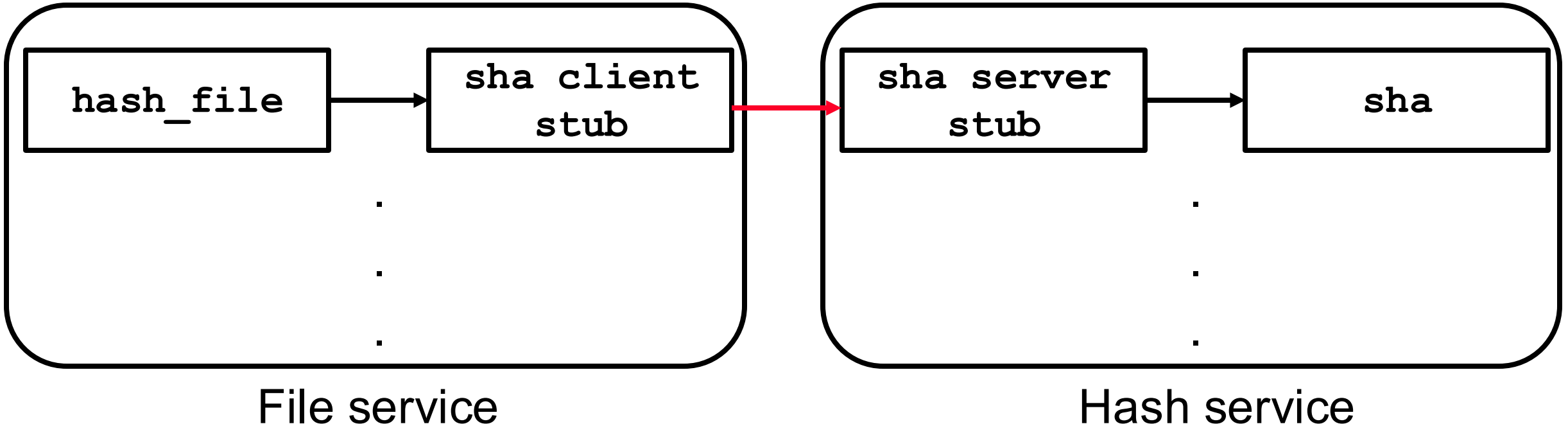
File service



Hash service

RPC Overview

- Can we transfer data directly between stubs?
  - memcpy the input and the output
  - Needs data to be in the same format in memory, for all languages



Data Memory Representation

- Languages store types in various formats in memory
  - Let's consider the variable "**`input`**" which is a string
  - Python has a hierarchical structure to define a string

```
# Python str
class str:
      data: PyASCIIObject
      ...
class PyASCIIObject:
      char *utf8;
      ...
```

Data Memory Representation

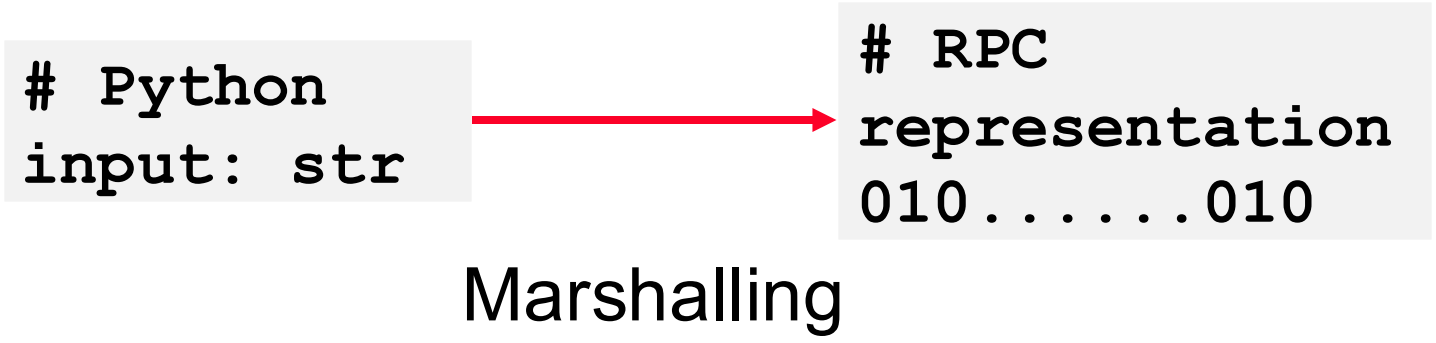
- Languages store types in various formats in memory
  - Let's consider the variable "**`input`**" which is a string
  - Python has a hierarchical structure to define a string
  - C++ has a flat structure
- Therefore, we can not do a memcpy

```
# Python str
class str:
      data: PyASCIIObject
      ...
class PyASCIIObject:
      char *utf8;
      ...
```

```
// C++ std::string
class basic_string {
  size_t size;
  size_t capacity;
  char* data;
}
```

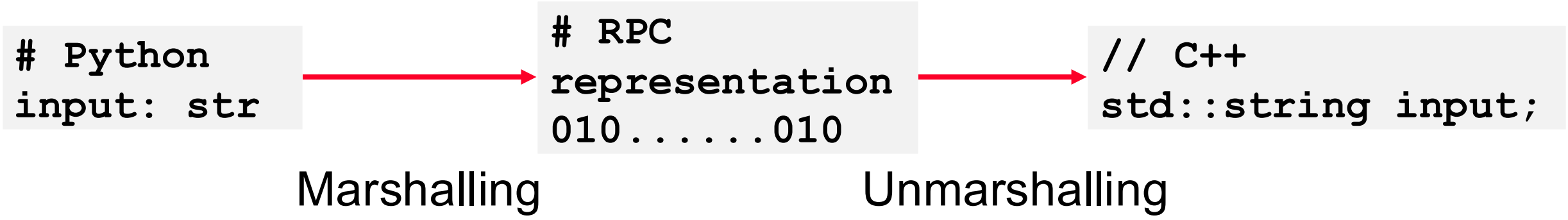
Marshalling/Unmarshalling

- **Marshalling**: Turn data from a memory representation into a representation that can be sent over the network
- RPC unifies marshalling data for all services
  - Once received, RPC needs to be able to save it to memory again



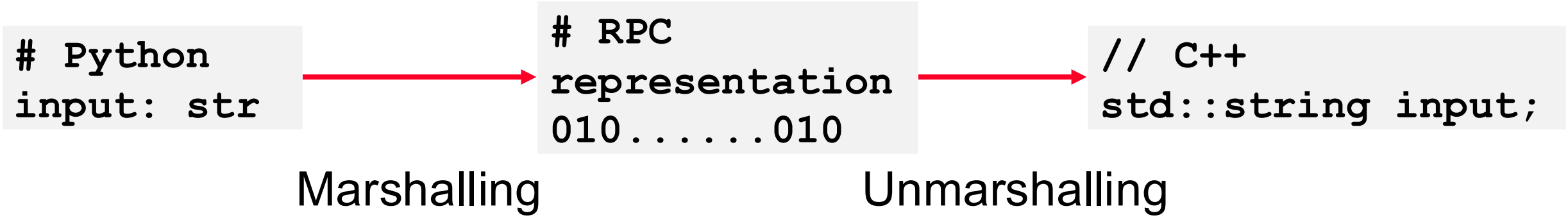
Marshalling/Unmarshalling

- **Marshalling**: Turn data from a memory representation of one language into an IDL representation sent over the network
- RPC unifies marshalling data for all services
- **Unmarshalling**: Turn data received from the network in an IDL representation into a memory representation of the receiving language



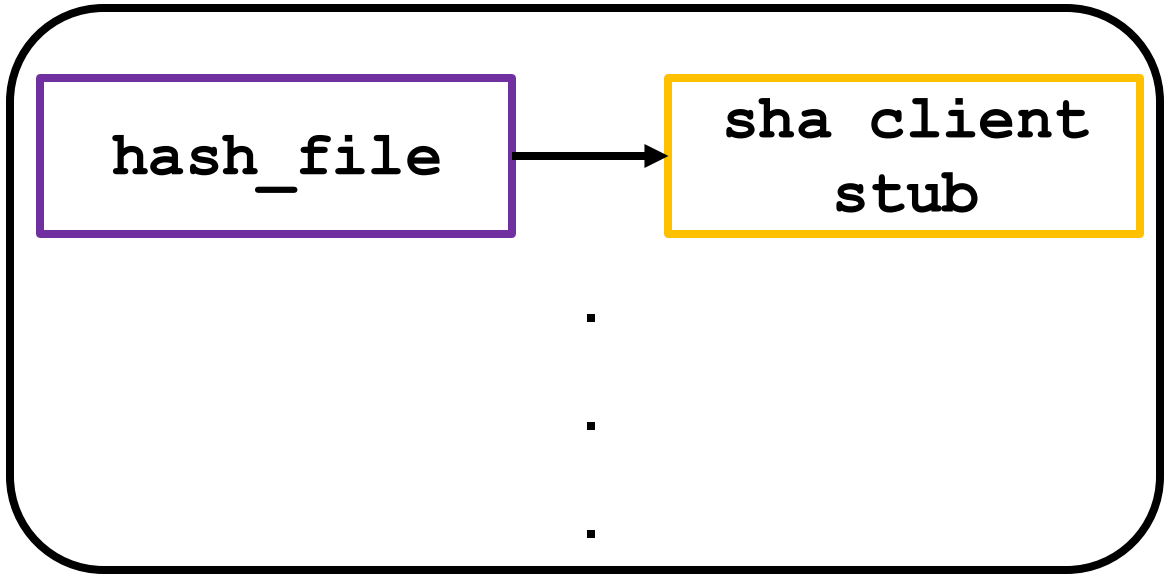
Marshalling/Unmarshalling

- **Marshalling**: Turn data from a memory representation of one language into an IDL representation sent over the network
- RPC unifies marshalling data for all services
- **Unmarshalling**: Turn data received from the network in an IDL representation into a memory representation of the receiving language
- We will see how RPC uses this

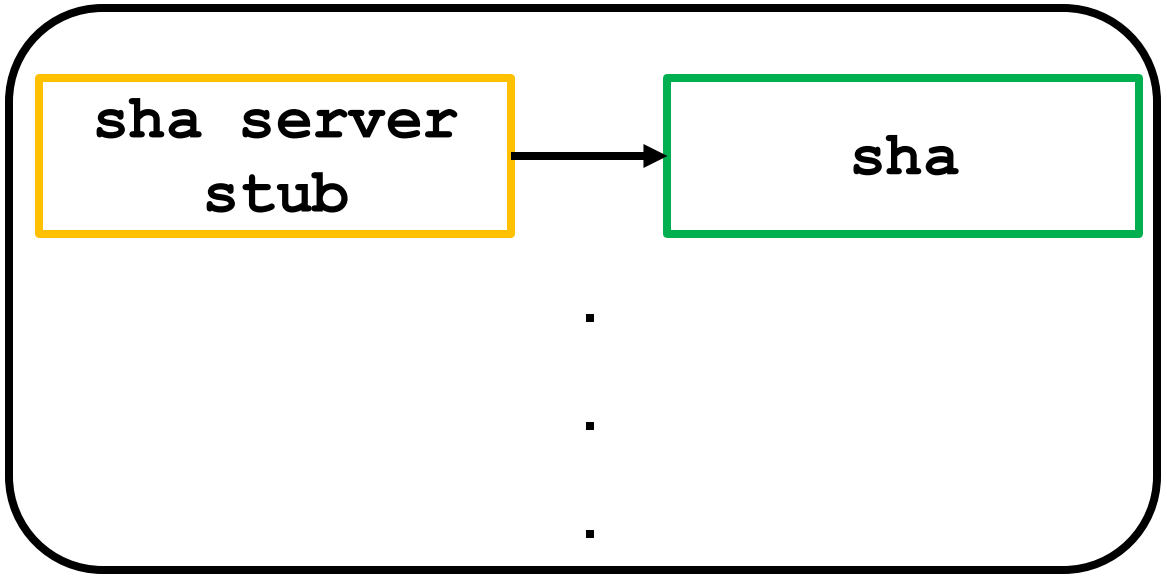


RPC

- Color coding for the following slides (red remains as before)
  - **Client code**
  - **IDL and RPC generated code**
  - **Server code**



File service



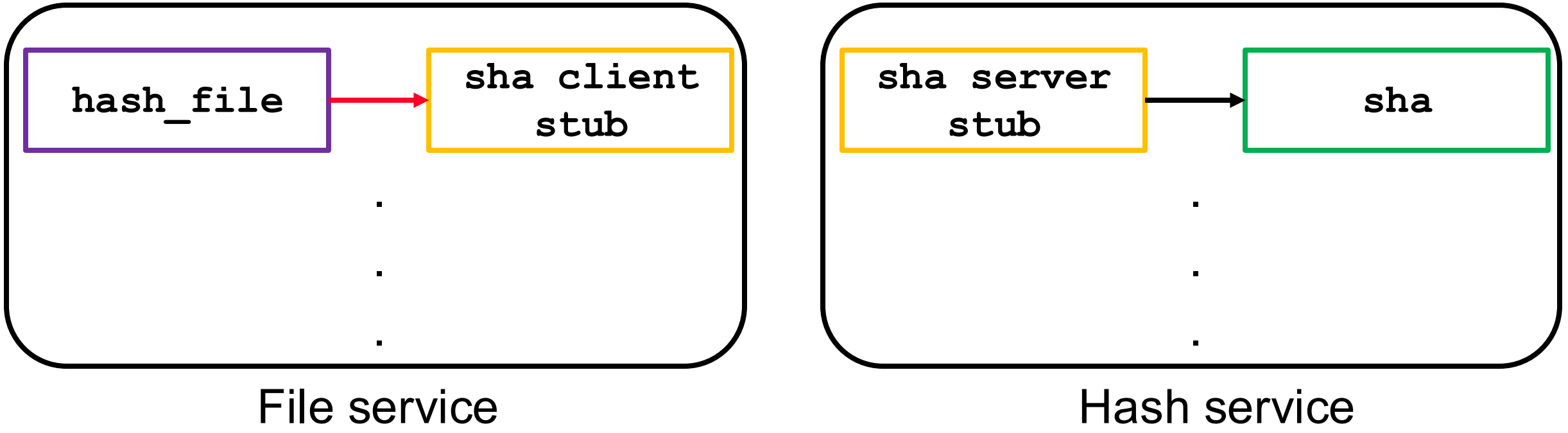
Hash service

RPC Function Call

- File service calls the client stub

RPC Function Call

- **`hash_file`** calls the client stub's **`sha`** function with its inputs



RPC Function Call

- Client stub's **`sha`** function takes the input

```
# generated client stub by rpc
# library for file service
class HashService:
      sha(self, input):
            req = marshall(input)
            resp = send(req)
            ...
```

RPC Function Call

- Client stub's **`sha`** function takes the input
- It marshalls data from memory into a representation that can be sent over the network

```
# generated client stub by rpc
# library for file service
class HashService:
      sha(self, input):
            req = marshall(input)
            resp = send(req)
            ...
```

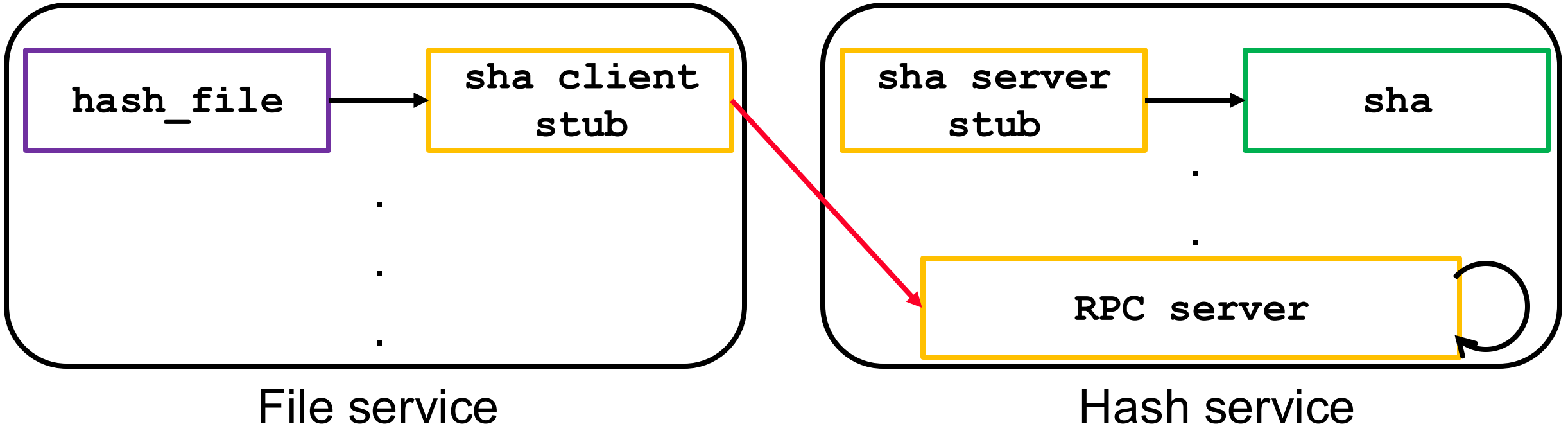
RPC Function Call

- Client stub's **`sha`** function takes the input
- It marshalls data from memory into a representation that can be sent over the network
- RPC provides the send function to send marshalled data

```
# generated client stub by rpc
# library for file service
class HashService:
      sha(self, input):
            req = marshall(input)
            resp = send(req)
            ...
```

RPC Function Call

- Marshalled data is sent to hash service
- RPC libraries include a built-in server that listens for RPC requests coming in from the network

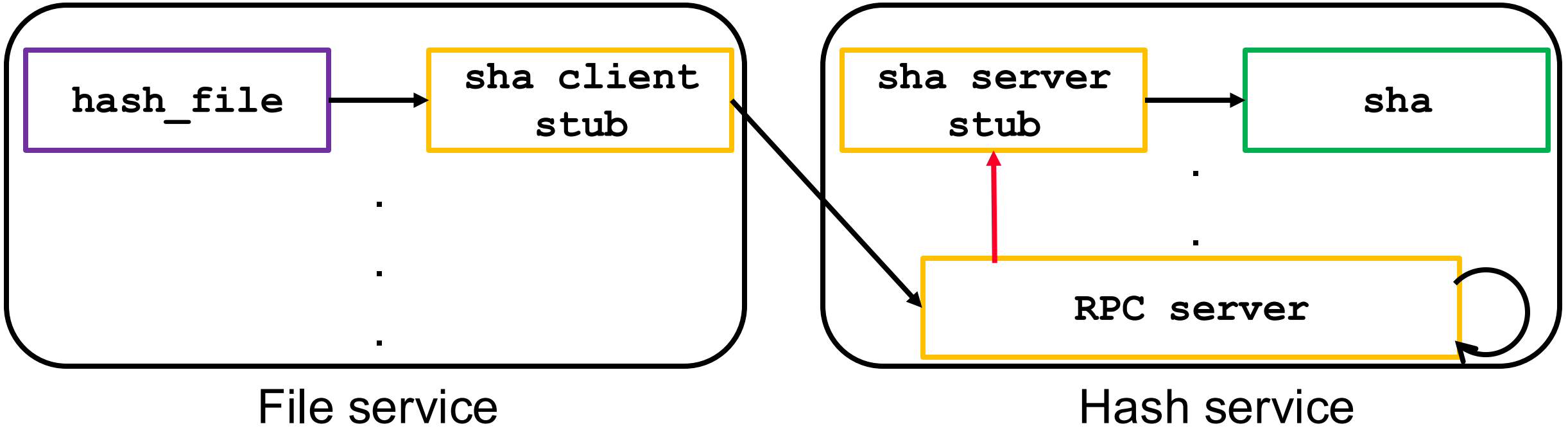


RPC Running Function

- RPC server runs the server stub

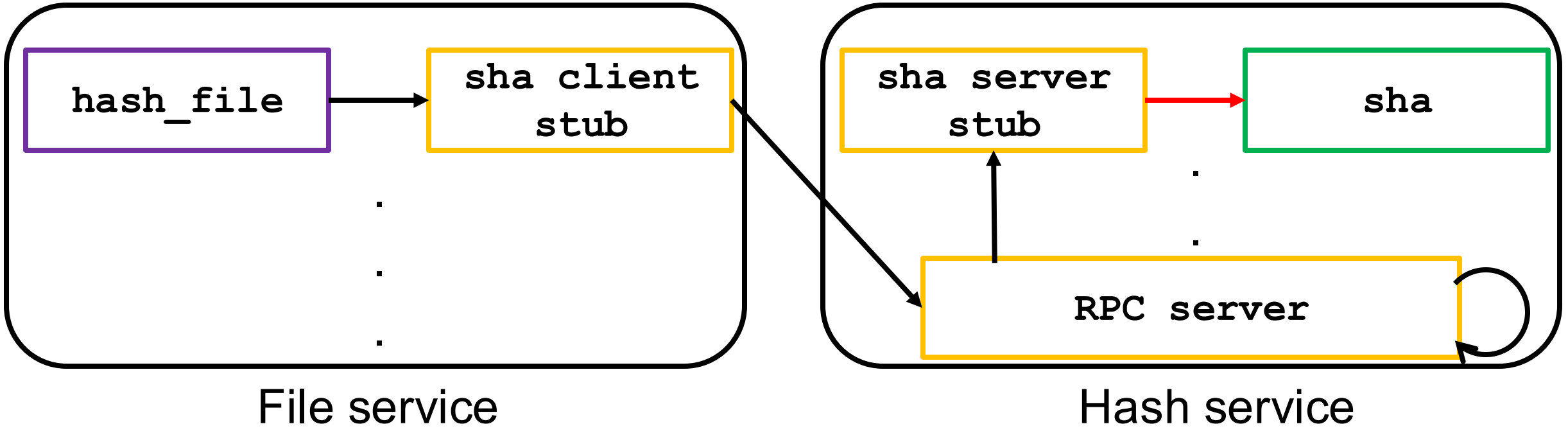
RPC Running Function

- Server stub's **`sha`** function takes the input



RPC Running Function

- Server stub's **`sha`** function takes the input
- It unmarshalls the data into variables that can be used by functions
- It calls the function **`sha`**

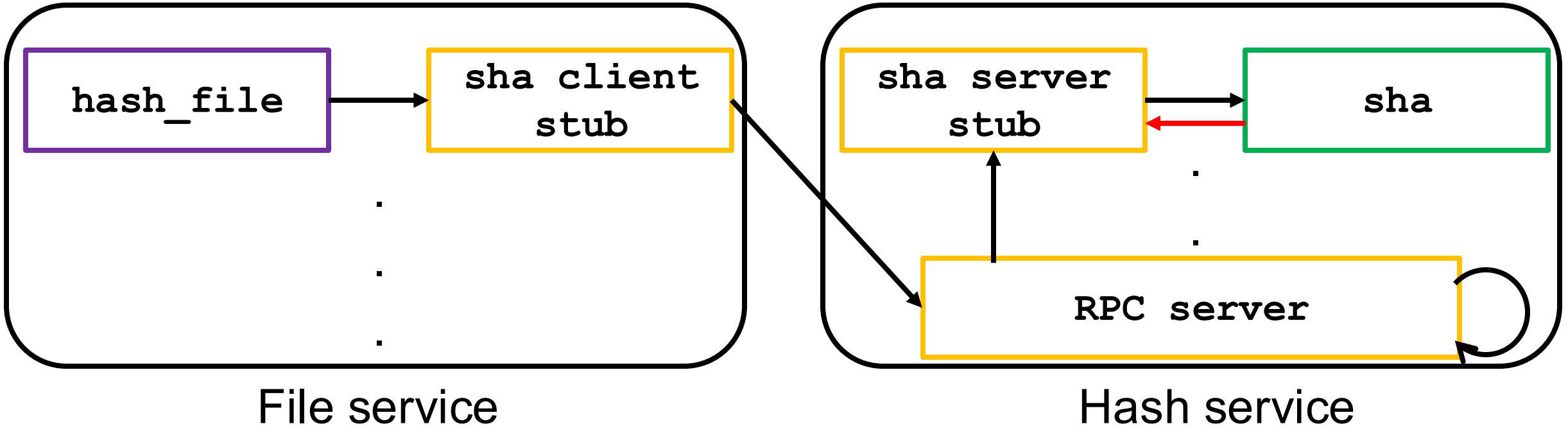


RPC Preparing Result

- Server stub prepares the result to return

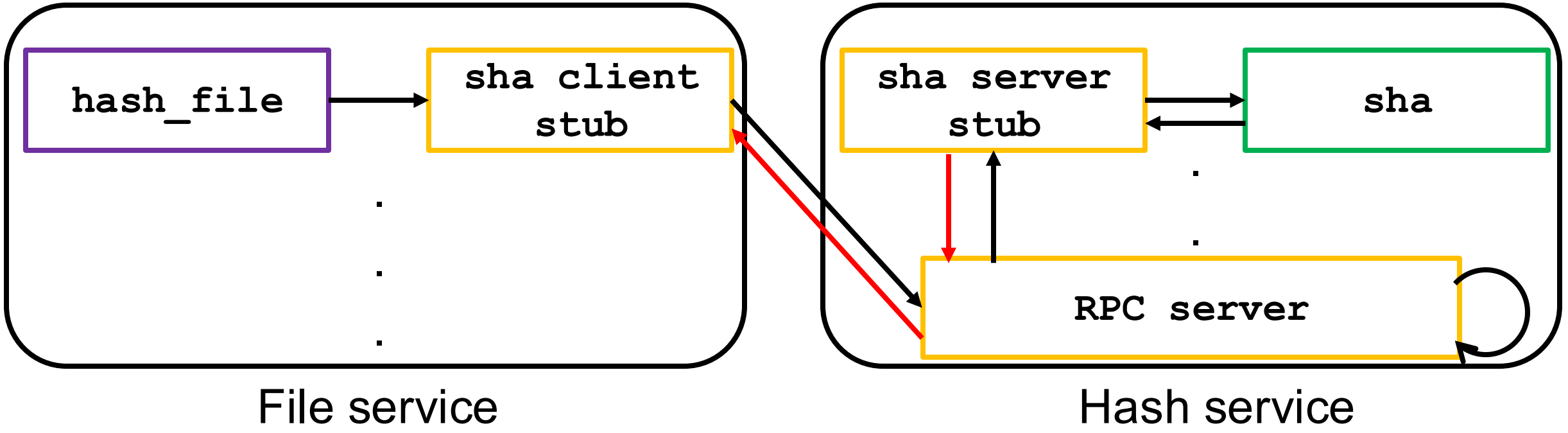
RPC Preparing Result

- Server stub marshalls the result



RPC Preparing Result

- Server stub marshalls the result
- Sends it back to the caller

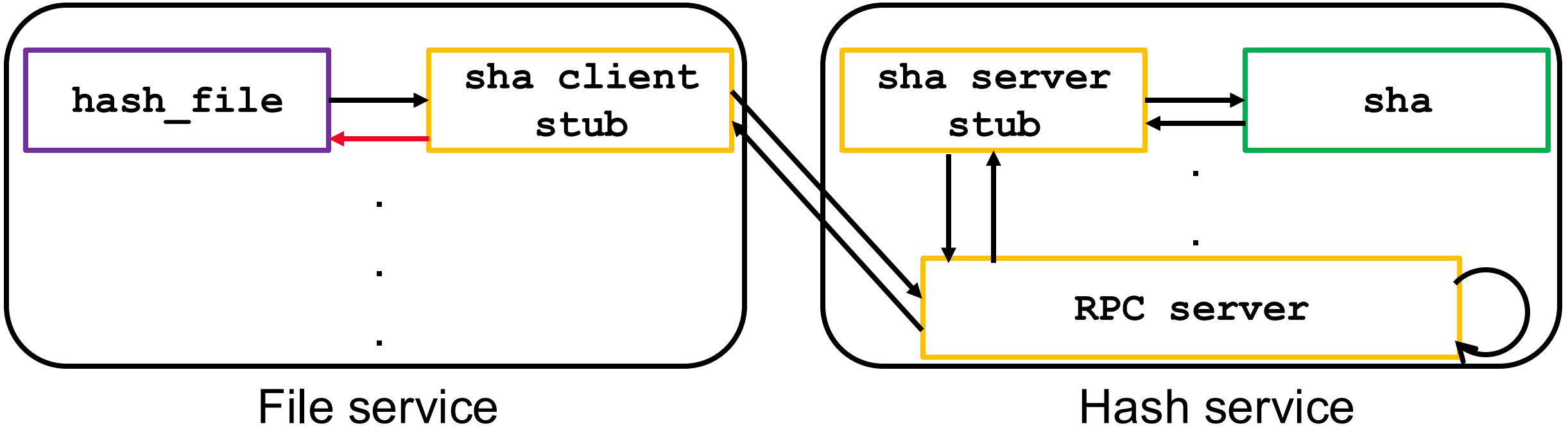


RPC Processing Returned Result

- Client receives the marshalled result

RPC Processing Returned Result

- Client stub unmarshalls the result and returns it to **`hash_file`**



RPC Processing Returned Result

- Client stub unmarshalls the result and returns it to **`hash_file`**
- All these steps are generated by RPC

```
# generated client stub by rpc library for file service
class HashService:
      sha(self, input):
            req = marshall(input)
            resp = send(req)
            return unmarshall(resp)
```

RPC Basics Summary

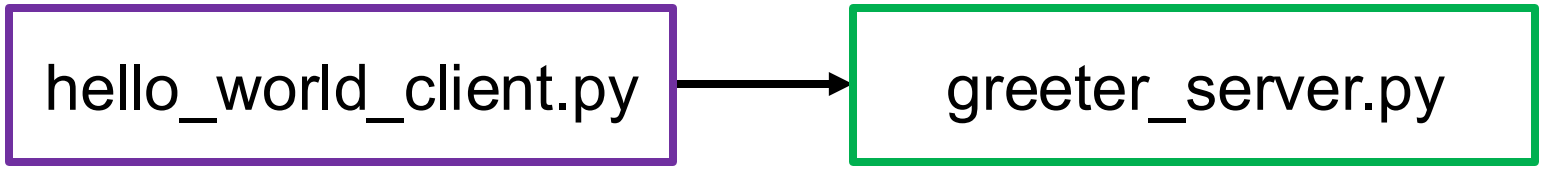
- RPC abstracts away the complexities of remote procedure calls
- On the client side
  - client only needs to call the client stub
  - stub marshalls and sends the inputs, waits for and unmarshalls the result
- On the server side
  - server only needs to start the RPC server
  - stub listens to requests, unmarshalls the input, calls the function, marshalls and sends the result back

RPC Implementation

- Various libraries implement RPC
  - Apache Thrift: uses Thrift as IDL (Facebook)
  - gRPC: uses protobuf as IDL (Google)
- When choosing an RPC library consider:
  - Interface definition language (IDL)
  - Supported programming languages
  - Performance
  - Community and project backers
- We focus on gRPC

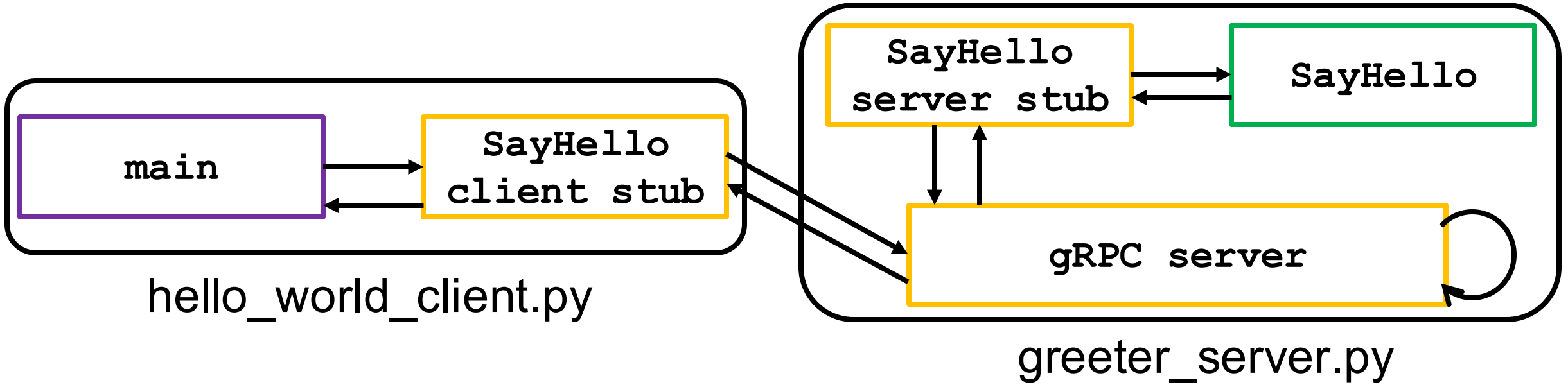
gRPC Hello World

- Imagine a simple “greeter” adds “Hello,” to the input it receives
  - E.g., service called with “World” returns “Hello, World”
- How can a client Python script use this service to get “Hello World”?



gRPC Hello World

- It's time to create the stubs
- gRPC uses protobuf 3.0 or proto3 as the IDL



gRPC Hello World Protobuf

- Define the service
  - Similar to creating an entry in a header file

```
// hello.proto
syntax = "proto3";

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply);
}
```

gRPC Hello World Protobuf

- Define the service
  - Then define the functions of the service

```
// hello.proto
syntax = "proto3";

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply);
}
```

gRPC Hello World Protobuf

- Define the inputs and outputs needed for the functions

```
// hello.proto
message HelloRequest {
   string name = 1;
}

message HelloReply {
   string message = 1;
}
```

gRPC Hello World Protobuf

- Define the inputs and outputs needed for the functions
  - Define the attributes of the input and the output
  - To keep marshalling consistent each attribute is given a field number

```
// hello.proto
message HelloRequest {
   string name = 1;
}

message HelloReply {
   string message = 1;
}
```

gRPC Hello World Protobuf

- Define the inputs and outputs needed for the functions
  - Define the attributes of the input and the output
  - To keep marshalling consistent each attribute is given a field number
  - Numbers are assigned to parameters in numerical order

```
// hello.proto
message HelloRequest {
   string name = 1;
}

message HelloReply {
   string message = 1;
}
```

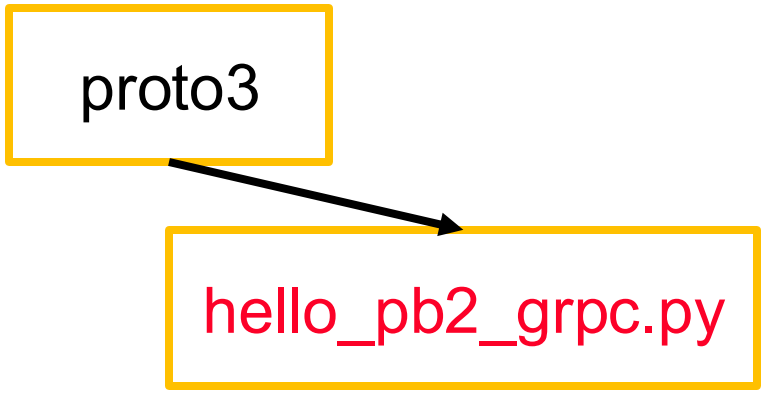
gRPC Hello World Protobuf Compilation

- Use Protocol Buffers compiler
  - Called protoc
  - grpc_tools comes with protoc for compiling to Python

```
python -m grpc_tools.protoc \
      -I .  --pyi_out=. \
      --python_out=.  \
      --grpc_python_out=. \
      hello.proto
```

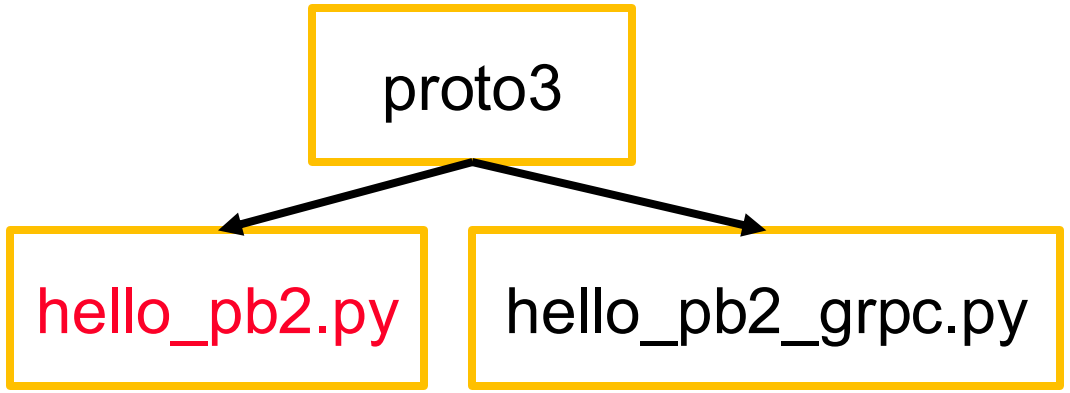
gRPC Hello World Protobuf Compilation

- Protoc generates file containing:
  - the classes for both client and server stub



gRPC Hello World Protobuf Compilation

- Protoc generates file containing:
  - the classes for both client and server stub
  - all the messages and how to marshall and unmarshall them
- Note: _pb.py is a popular ending for Python files so they use _pb2.py



gRPC Hello World Protobuf Compilation

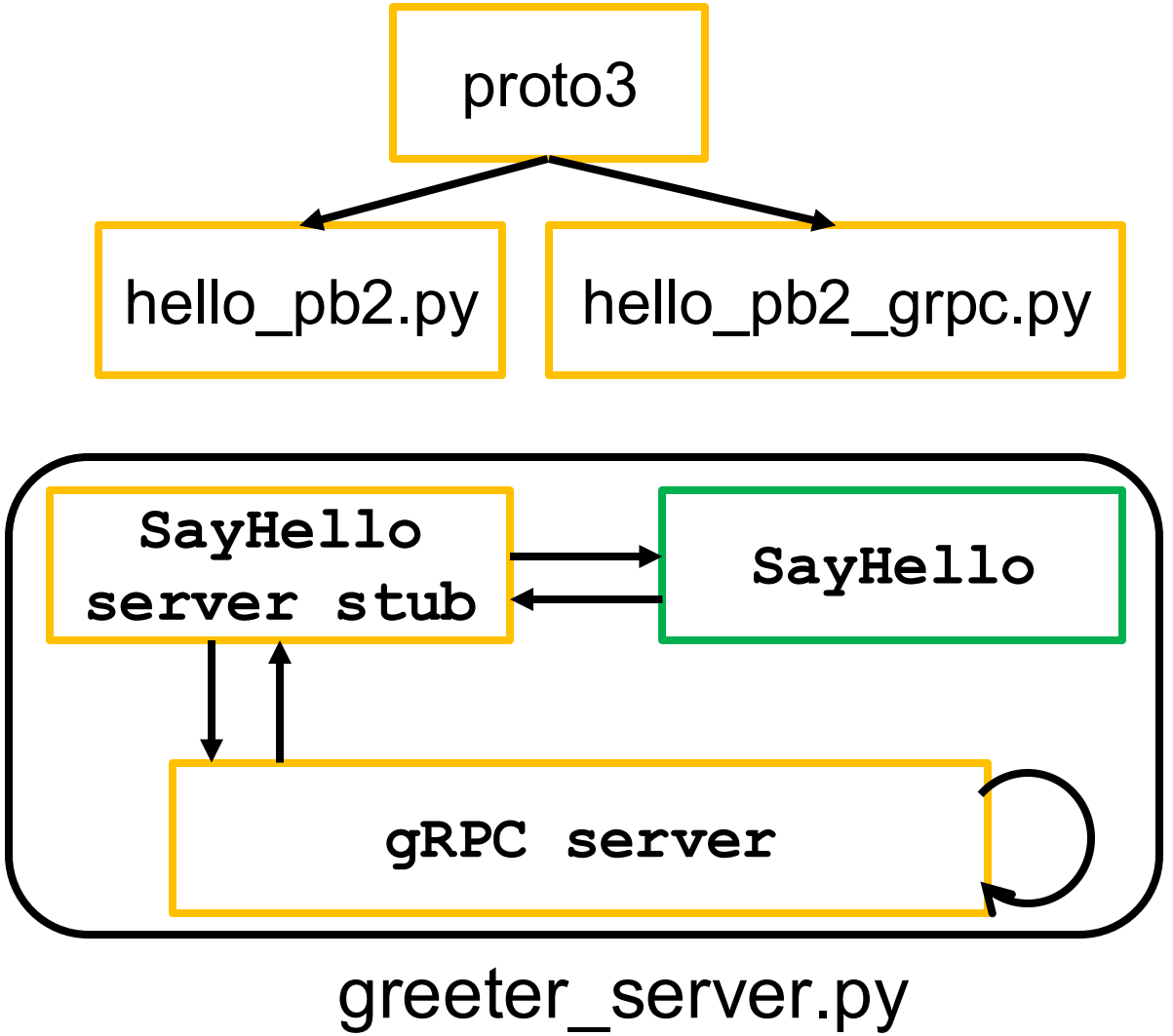
- Protoc generated stubs include:
  - Python class defining the service
  - GreeterServicer with a to-be-implemented service

```
# rest of hello_pb2_grpc.py
class GreeterServicer(object):
      def SayHello(self, request, context):
            context.set_code(grpc.StatusCode.UNIMPLEMENTED)
            context.set_details('Method not implemented!')
            raise NotImplementedError('Method not implemented!')
```

Take a break!

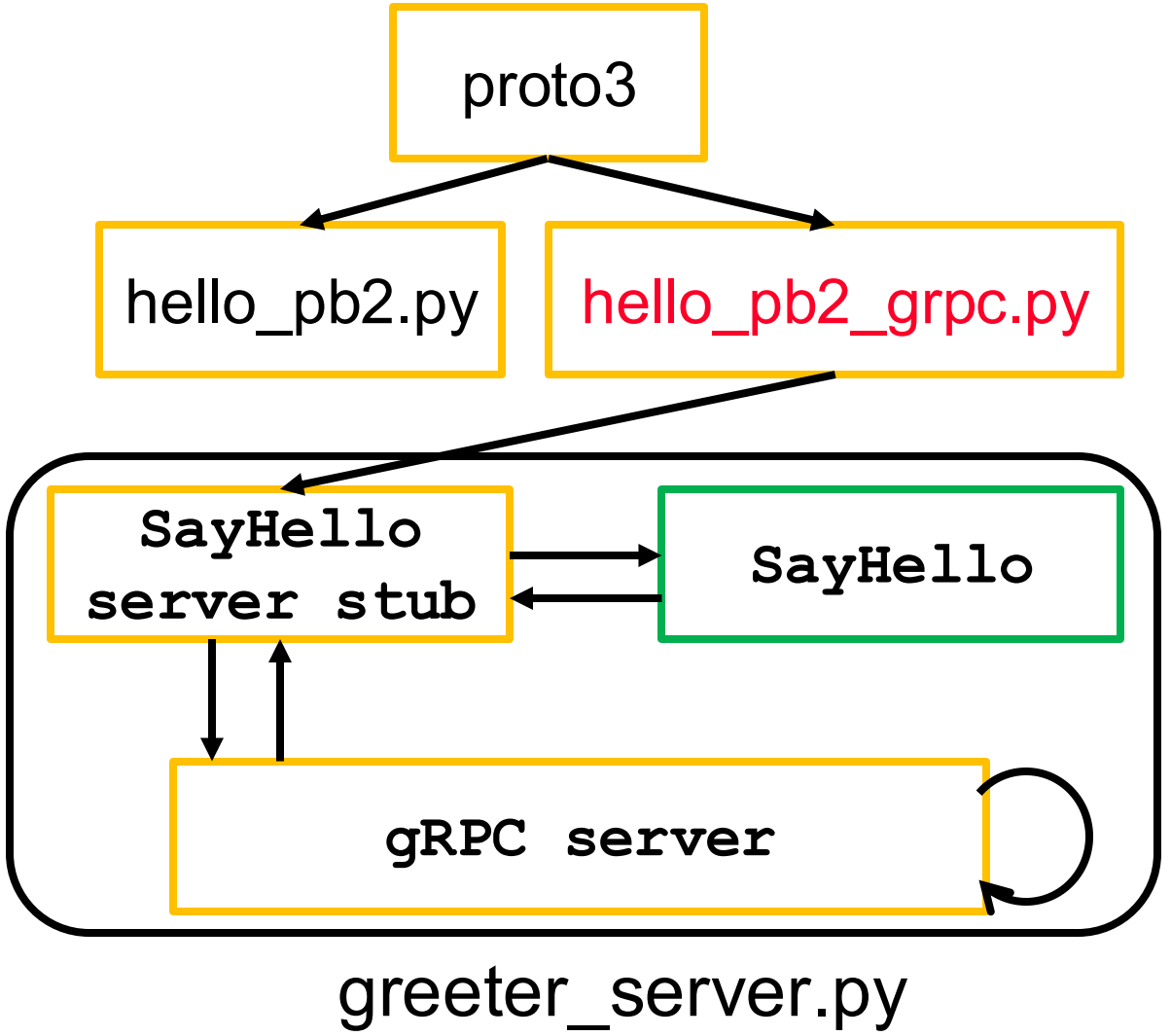
gRPC Hello World Server

- Inherit and implement the abstract class (GreeterServicer) generated by protoc



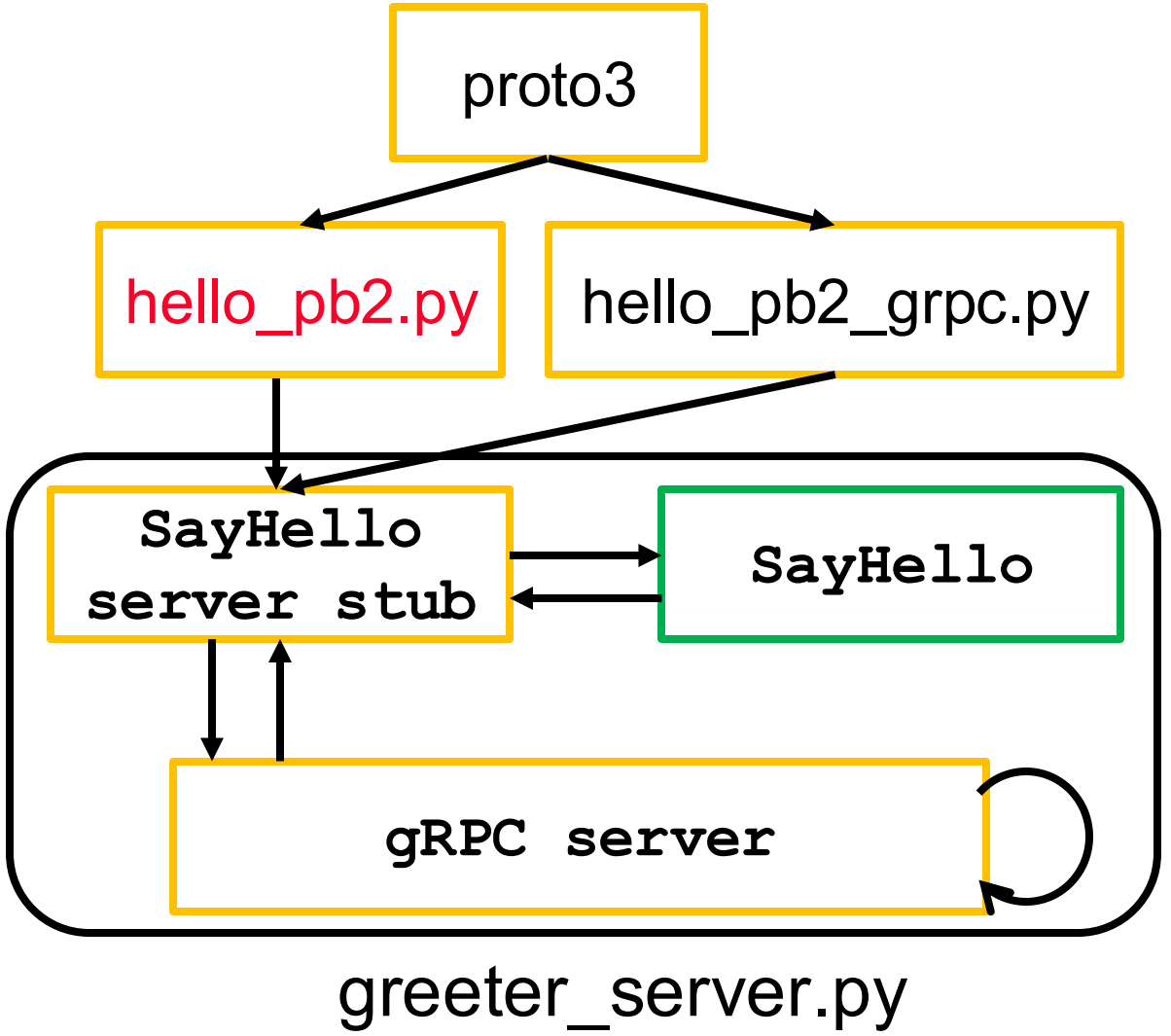
gRPC Hello World Server

- Inherit and implement the abstract class (GreeterServicer) generated by protoc
- Import
  - the class defining the service



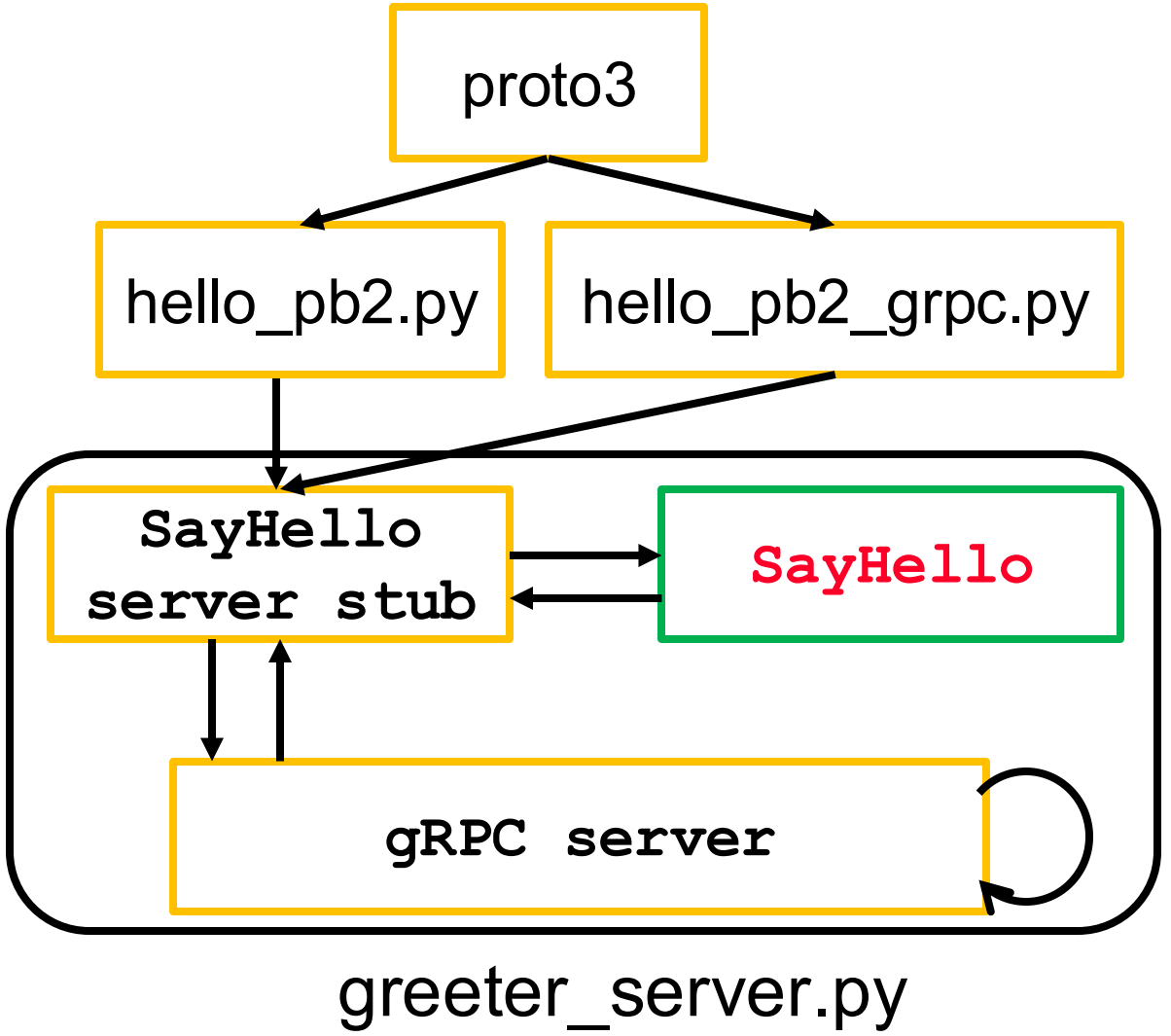
gRPC Hello World Server

- Inherit and implement the abstract class (GreeterServicer) generated by protoc
- Import
  - the class defining the service
  - input and output definition



gRPC Hello World Server

- Inherit and implement the abstract class (GreeterServicer) generated by protoc
- Import
  - the class defining the service
  - input and output definition
- Implement the abstract class and connect it to the server



gRPC Hello World Server

- Import the code generated by protoc
  - The class that defines the service
  - The input and output definition

```
# greeter_server.py
from hello_pb2_grpc import (
    GreeterServicer
)
from hello_pb2 import (
    HelloRequest,
    HelloReply
)
```

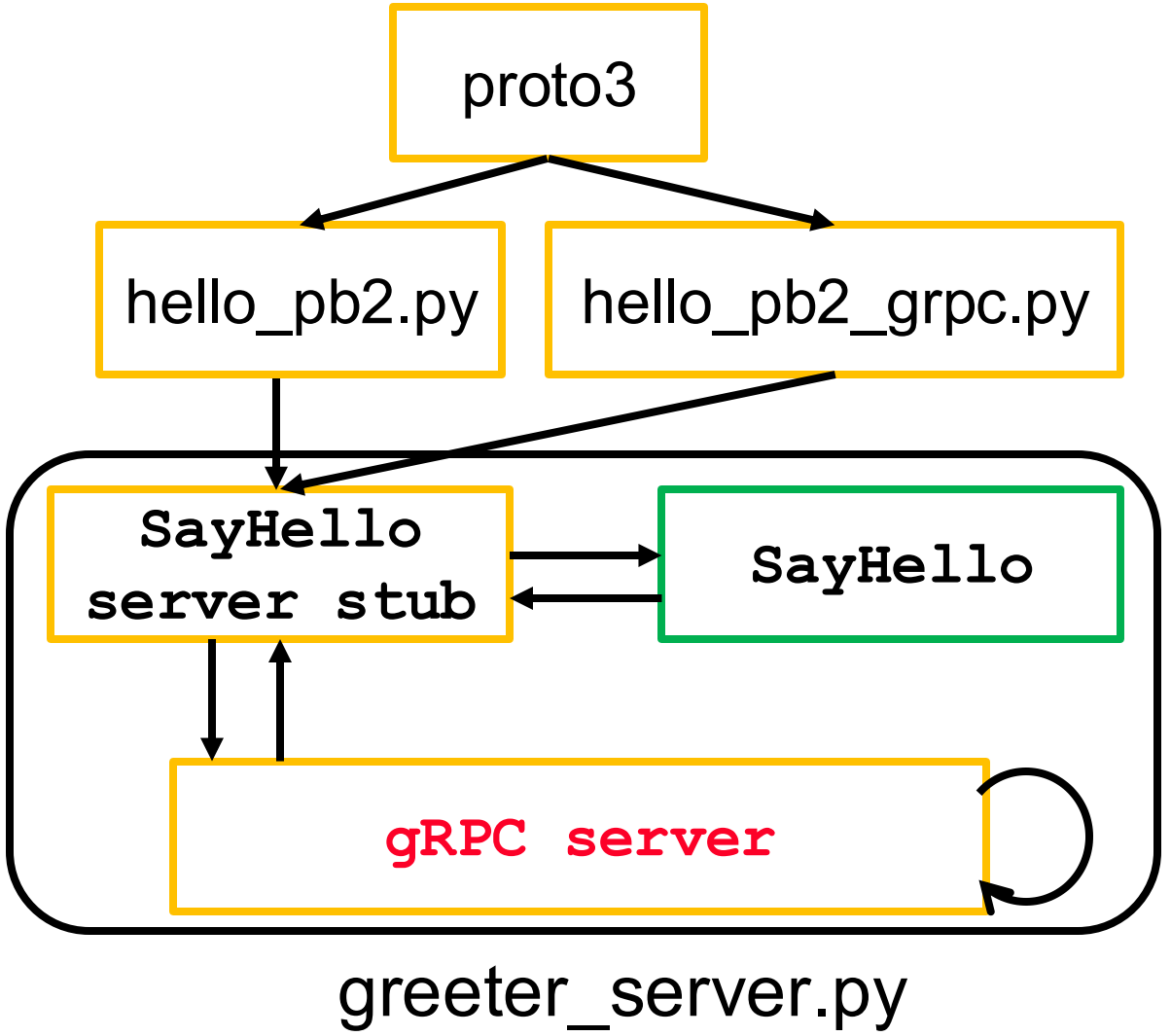
gRPC Hello World Server

- Implement the service

```
# greeter_server.py
class GreeterServiceImpl(GreeterServicer):
    def SayHello(self, request: HelloRequest, context):
        response = HelloReply()
        response.message = f"Hello, {request.name}!"
        return response
```

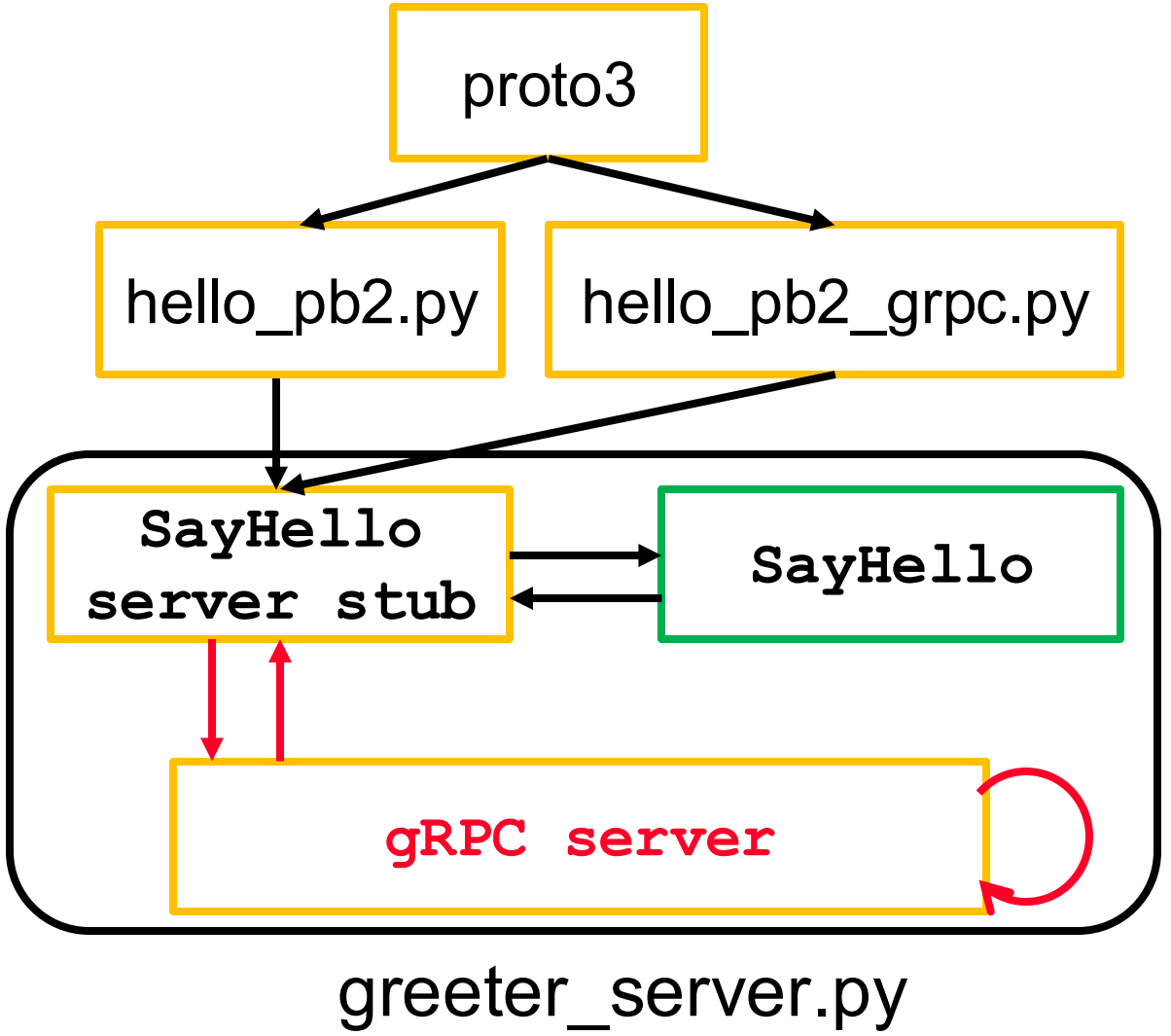
gRPC Hello World Server

- Create a gRPC server



gRPC Hello World Server

- Create a gRPC server
  - The server listens for requests while the stub is running
  - We saw in L8.2 and L9.2 this can be done with spawning new threads
  - This is called a thread pool in Python



gRPC Hello World Server

- Create a thread pool

```
# greeter_server.py
from concurrent import futures
requests_thread_pool =
      futures.ThreadPoolExecutor(max_workers=10)
```

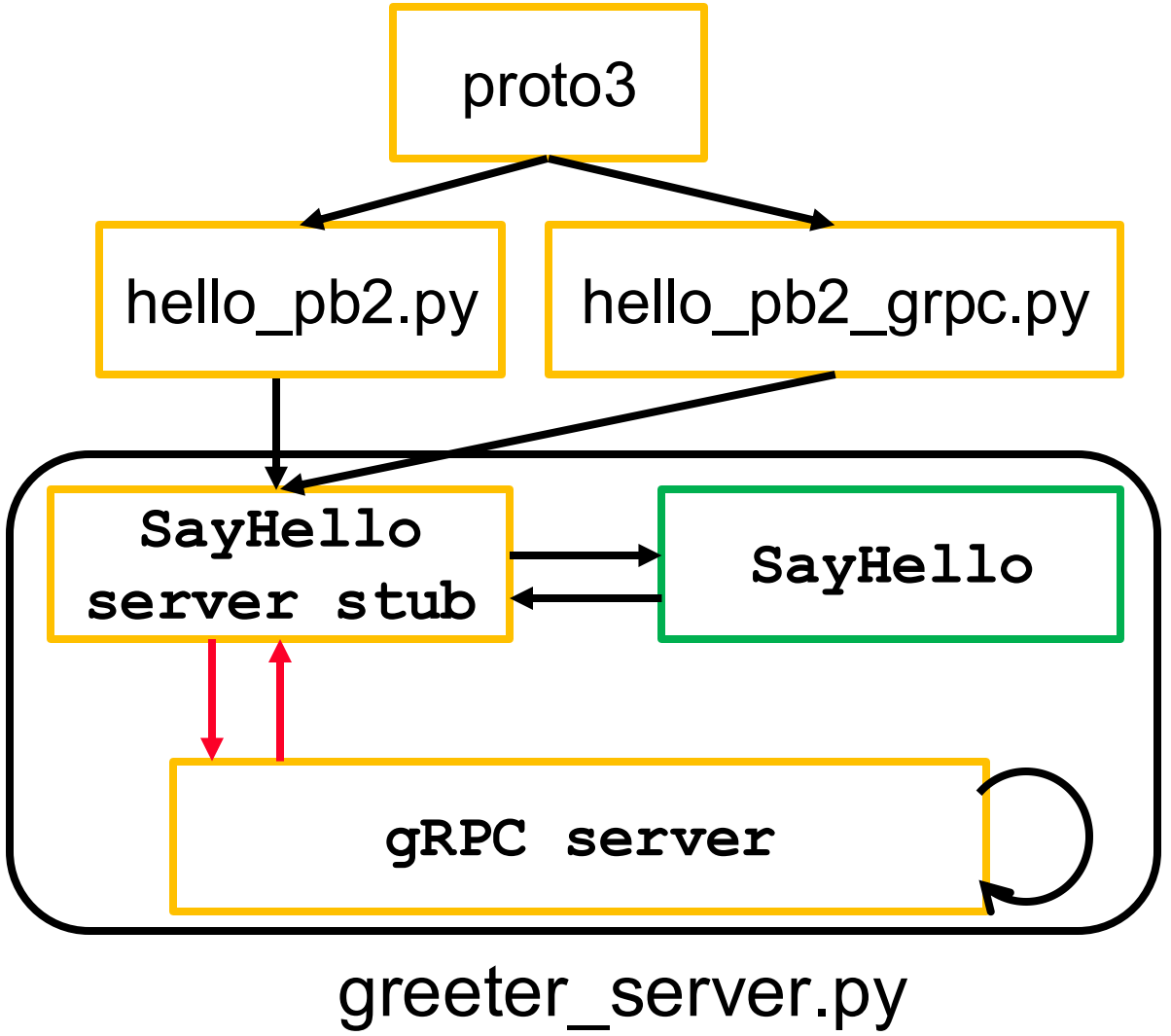
gRPC Hello World Server

- Create a thread pool
- Create a gRPC server with access to the thread pool

```
# greeter_server.py
from concurrent import futures
requests_thread_pool =
      futures.ThreadPoolExecutor(max_workers=10)

import grpc
# Server implemented by gRPC
server = grpc.server(requests_thread_pool)
```

gRPC Hello World Server

- gRPC server calls the service we implemented (**`GreeterServiceImpl`**)
- gRPC generates a function to connect implemented services to the server



gRPC Hello World Server

- **`add_GreeterServicer_to_server`**
  - is generated by gRPC in the same file as the abstract class (**`GreeterServicer`**)
  - accepts any instance of a class that inherits and implements the abstract class

```
# rest of hello_pb2_grpc.py
# takes in servicer and registers it to server
# related requests will now go to this servicer
def add_GreeterServicer_to_server(servicer, server):
    ...
```

gRPC Hello World Server

- Make an instance of the implemented service

```
# greeter_server.py
service_implementation = GreeterServiceImpl()
```

gRPC Hello World Server

- Make an instance of the implemented service
- Add an instance of the service to the server

```
# greeter_server.py
service_implementation = GreeterServiceImpl()

from hello_pb2_grpc import add_GreeterServicer_to_server
add_GreeterServicer_to_server(service_implementation, server)
```

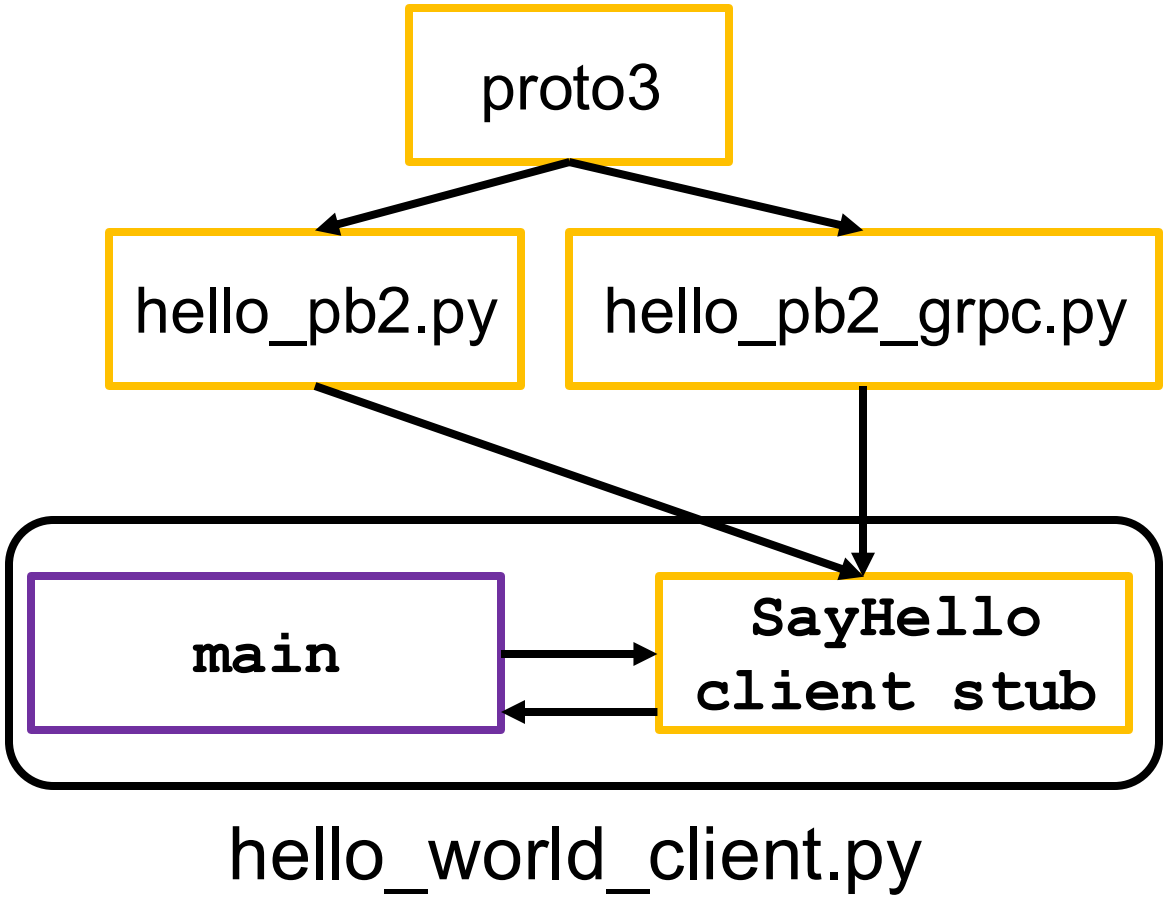
gRPC Hello World Server

- Start the gRPC server and listen for requests
  - We listen on the local network for this example

```
# greeter_server.py
# Only use insecure port for testing
server.add_insecure_port('[::]:50051')
server.start()
print("Server started on port 50051...")
server.wait_for_termination()
```

gRPC Hello World Client

- The client imports the same classes and uses gRPC to communicate



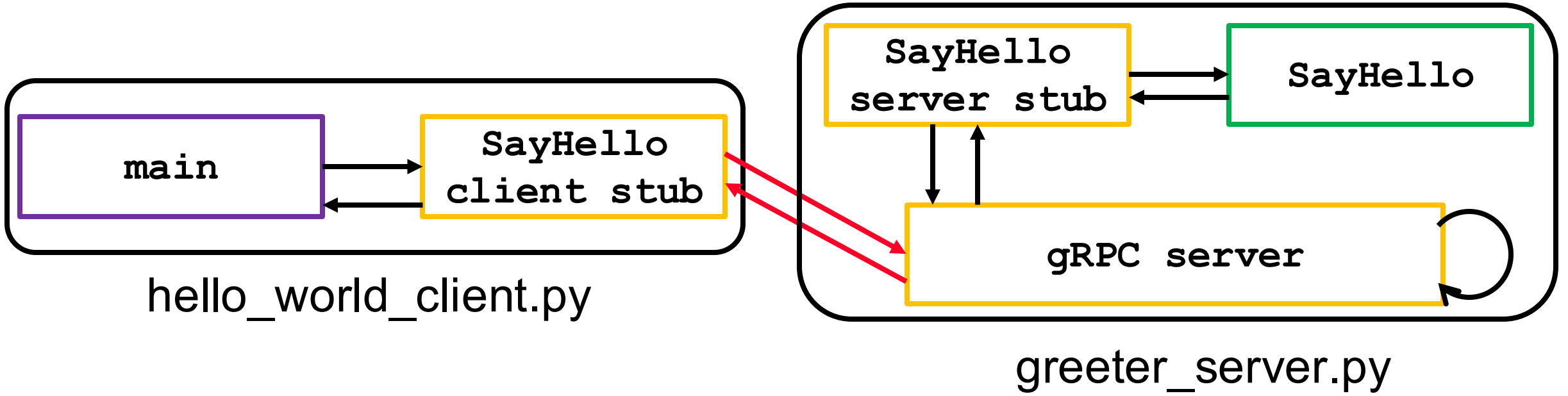
gRPC Hello World Client

- Import classes in client
  - The client stub
  - Input and output classes

```
# hello_world_client.py
from hello_pb2_grpc import (
    GreeterStub,
)

from hello_pb2 import (
    HelloRequest,
    HelloReply
)
```

gRPC Hello World Client

- The client needs a way to reach other services
- gRPC uses channels
  - **Channel**: is an endpoint you can send and receive messages from
  - Can be an IP address



gRPC Hello World Client

- Create a channel (can be encrypted or insecure)

```
# hello_world_client.py
# Where to send requests to
channel = grpc.insecure_channel('localhost:50051’)
```

gRPC Hello World Client

- Send a request by creating an instance of client stub
  - The instance handles communication

```
# hello_world_client.py
# Where to send requests to
channel = grpc.insecure_channel('localhost:50051’)
stub = GreeterStub(channel)
```

gRPC Hello World Client

- Send a request by creating an instance of client stub
  - The instance handles communication
  - The GreeterStub acts the same as the service we implemented (**`GreeterServiceImpl`**)

```
# hello_world_client.py
# Where to send requests to
channel = grpc.insecure_channel('localhost:50051’)
stub = GreeterStub(channel)
request = HelloRequest(name="World")
response: HelloReply = stub.SayHello(request)
print("Client received: " + response.message)
```

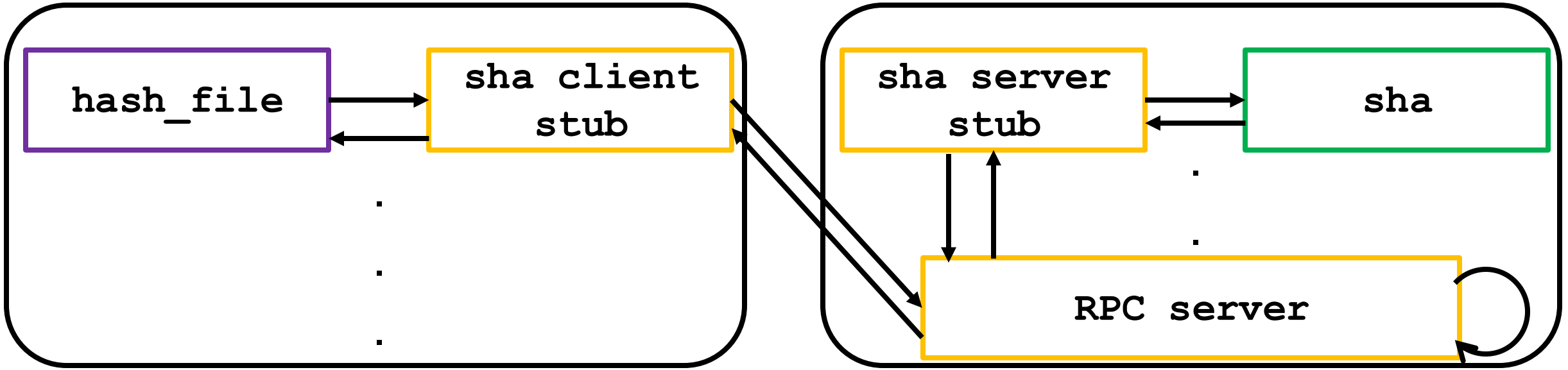
gRPC Hello World Client

- Send a request by creating an instance of client stub
  - The instance handles communication
  - The GreeterStub acts the same as the service we implemented (**`GreeterServiceImpl`**)

```
# hello_world_client.py
# Where to send requests to
channel = grpc.insecure_channel('localhost:50051’)
stub = GreeterStub(channel)
request = HelloRequest(name="World")
response: HelloReply = stub.SayHello(request)
print("Client received: " + response.message)
```

gRPC File & Hash Service

- Similar to Hello World
- Use gRPC to:
  - Create hash service's server stubs in C++
  - Create file service's client stubs in Python



file_service.py
(File service)

hash_service.cpp
(Hash service)

Summary

- RPC abstracts away the complexities of a remote call
  - Client only needs to call the client stubs with its input
  - Server only needs to implement the service and start the RPC server

- gRPC is an implementation of RPC
  - Originally created by Google but is now open source
  - Uses protobuf 3.0 or proto3
  - Supports many languages including Python, C++, Golang, etc.