|  |  |
| --- | --- |
| **GDPARCM Lecture – Inter-process communication** | Instructor: Neil Patrick Del Gallego |

**Inter-process Communication**

Recall that: Pass messages between applications. Message passing is the core operation for all distributed systems.

You can do a lot with message passing:

* Chat system
* Send images, videos through file streams.
* Trigger remote operations

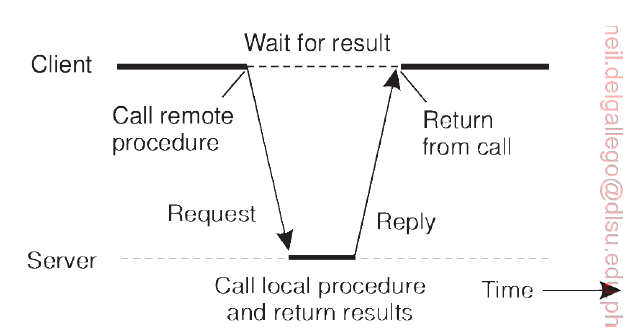
Patterns and protocols exist and were proposed in the field of distributed systems, to support complex operations being executed across different servers/PCs. Two common used models for inter-process communication

* Remote procedure call (RPC)
* Message-oriented middleware (MOM).

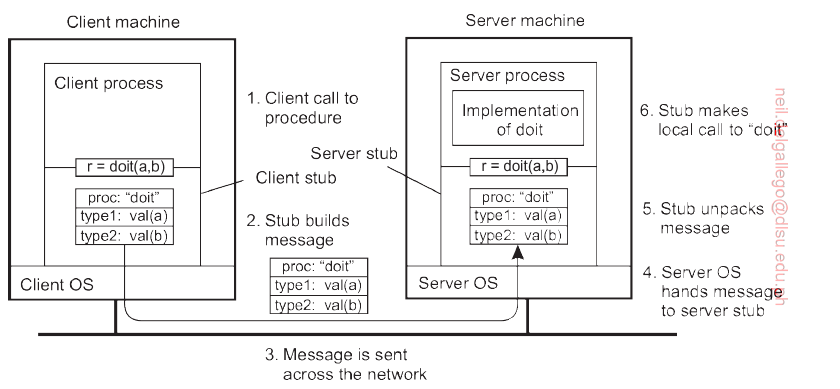
**Remote procedure call (RPC)**

* Many distributed systems are based on explicit message exchanges between processes.
* Background: Chat system has a very basic message passing system 🡪 a fixed character buffer length to accommodate the message string to be sent/received.
* Problem: What if we need to do more complex operations beyond a chat system? Distributed systems only communicate through message passing!
  + Upload and synchronize multiple files across different servers.
  + Stream audio/image/video files.
  + Perform real-time character actions in an MMO game.
* Objectives:
  + Efficiently encode these complex commands as meaningful messages.

Schematic diagram:



A more concrete example 🡪 Consider a function **doit(a,b)** that returns an integer number. The function **doit** is implemented in the server. Client requests for the result.



General steps:

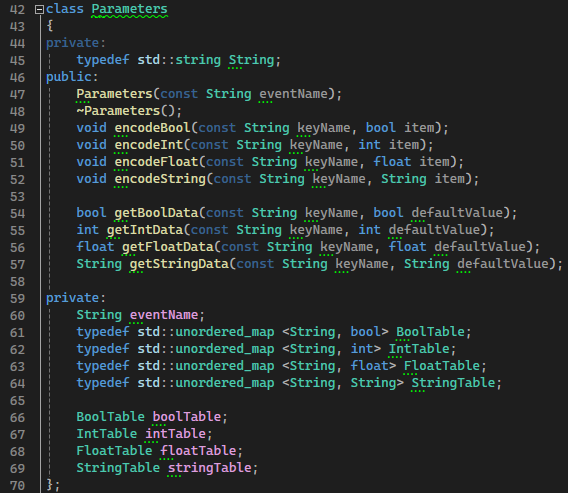
1. The client creates a human-readable, but also code-compatible “message” called a **stub**.
2. The client stub gets sent to the server via socket.
3. The server received the **stub** and parses it into a compatible instruction.
4. The result of the server is packed as a new stub, and sent back to client, for interpretation.

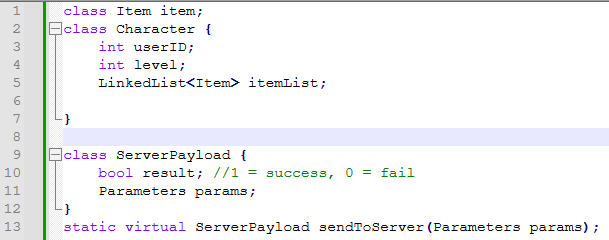
More specific steps:

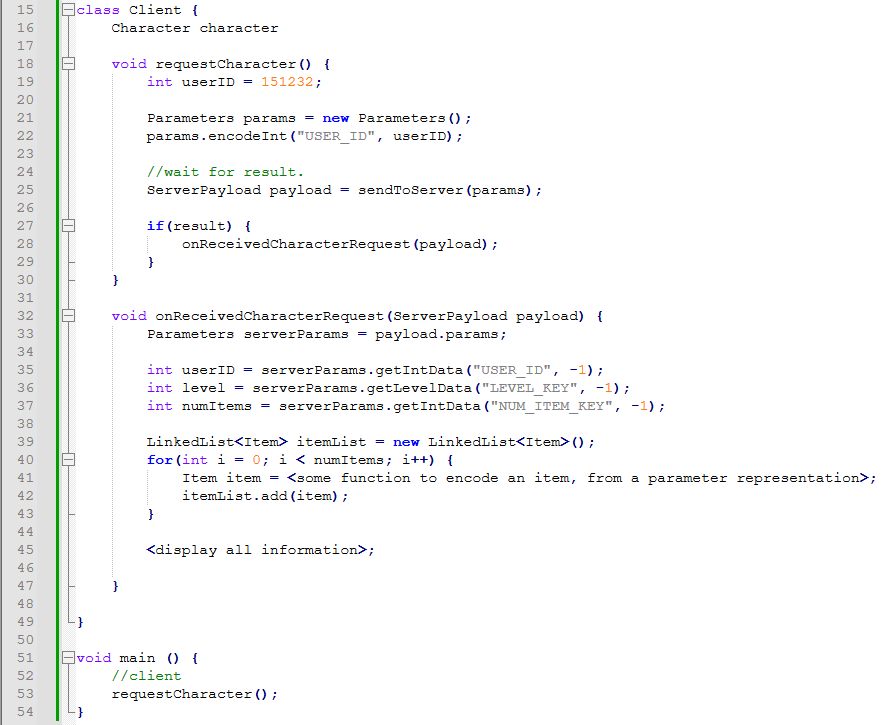
1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message and calls the local operating system.
3. The client’s OS sends the message to the remote OS.
4. The remote OS gives the messages to the server stub.
5. The server stub unpacks the parameter(s) and calls the serer.
6. The server does the work and returns the result of the stub.
7. The server stub packs the result in a message and calls its local OS.
8. The server’s OS sends the message to the client’s OS.
9. The client’s OS gives the message to the client stub.
10. The stub unpacks the results and returns it to the client.

**A C++ pseudocode example of RPC with stub creation.**

SCENARIO: A client requesting for all items equipped by a character. We use the **Parameter** class sample discussed in GDPARCM\_H07.



****

****

**Elements of RPC, as shown in the code snippet above:**

1. **Client-Server Interaction:**
   * The **Client** class initiates a request to the server using the **sendToServer** method. This is akin to a client making a remote call to a procedure or method on the server.
2. **sendToServer Method:**
   * The **sendToServer** method is a representation of the remote procedure call. It is assumed to be implemented elsewhere, and it is responsible for sending a request to the server and returning a response. The server's procedure is invoked remotely as a result of this method call.
3. **ServerPayload:**
   * The **ServerPayload** class represents the response or result of the remote procedure call. It contains information such as whether the operation was successful (**result**) and additional parameters (**params**) that are sent back to the client.
4. **Asynchronous Processing:**
   * The client appears to wait for the result of the remote procedure call, as indicated by the comment "//wait for result." This suggests that the interaction may be asynchronous, with the client continuing its execution after initiating the remote call and later processing the result.

While the specific details of the RPC mechanism (e.g., transport protocol, serialization/deserialization, etc.) are not provided in the code snippet, the overall structure aligns with the client-server interaction typical of remote procedure calls in distributed systems. In practice, RPC is a common approach for communication between components in distributed systems, enabling the invocation of procedures or methods on remote servers as if they were local.

**Challenges:**

While the basic idea sounds simple and elegant, subtle problems exist.

* To start with, because the calling and called procedures run on different machines, they execute in different address spaces, which causes complications.
* Parameters and results also have to be passed, which can be complicated, especially if the machines are not identical. E.g. Web browser (client), a C++ application (server).
* Either or both machines can crash and each of the possible failures causes different problems.

Still, most of these can be dealt with, and RPC is a widely-used technique that underlies many distributed systems.

A screenshot of a phone

Description automatically generatedA screenshot of a white screen

Description automatically generated

**Introduction to gRPC**: <https://grpc.io/>

* Developed by Google. It is a proposed universal remote procedure call (RPC) framework that can run in any environment.

**Motivation**

* Google has been using a single general-purpose framework called Stubby to connect a large number of microservices.
* Stubby is too tightly coupled to the internal infrastructure 🡪 difficult to scale and add applications. Example: An old C++ system, to a latest Python client. Hence, the creation and release of gRPC.
* Standardize message passing across different application/services.
* A screenshot of a phone

  Description automatically generatedIn our event broadcasting example, it is barebones. gRPC aims to handle basic message passing, server-client handling, security, layered protocols, timeout and cancellation policies, and many more.

Sample case studies:

* + List: <https://www.cncf.io/case-studies/?_sft_lf-project=grpc>
  + How cloud native technology helps Mux simplify online video streaming: <https://www.cncf.io/case-studies/mux/>
  + How Netflix increased developer productivity and defeated the thundering herd with gRPC: <https://www.cncf.io/case-studies/netflix/>
  + OpenAI: Launching and scaling up experiments, made simple: <https://www.cncf.io/case-studies/openai/>

**Design Principles**

* Services not Objects, Messages not References.
  + Passing objects tend to be very large. E.g. Person class instance to another client. What if you requested N person instances?
  + References to certain remote functions can fail miserably (fallacies of distributed computing).
    - Example 1: Consider distributed image processing 🡪 delay in receiving a response would cause a missing patch.
    - Example 2: What if we upgrade/change the infrastructure or even rename a function? E.g. Server::requestPerson() 🡪 Server::RequestPerson()
  + Coarse-grained messaging allows us to send staggered bits of data. E.g. Instead of person instance, we first send <user ID + name>, <birthday + address>, … <account balance>.
* Coverage and Simplicity
  + Should be available on every popular development platform.
  + Support different applications coded in different programming languages and API. E.g. A web client communicating with a C++ application.
* Free and open source
* Interoperability and reach
  + The protocol must be capable of surviving traversal over common internet infrastructure.
* General purpose and performance
  + Should cover a wide range of use-cases. E.g. simple web viewer, streaming media, MMORPG, Instagram, chat systems, etc.
* Layered
  + Key facets of the stack must be able to evolve independently. E.g. Revising security protocols should not affect the messaging protocol.
* Payload agnostic
  + Different services may use different messages and encodings (protocol buffer, JSON, XML, YAML, etc). It must be easy to convert the message to compatible encodings. E.g. easily parse the message to a JSON file
* Streaming
  + Must be able to stream and express large data. E.g. Video streaming, cloud AI training, downloading of game update.
* Blocking and non blocking
  + Must support synchronous and asynchronous RPC.
* Cancellation and timeout
  + Operations can be expensive and long-lived. At any time during server-client communication, there must be a way to cancel/timeout the request 🡪 prevents blocking issue and cascade. E.g. Client N + 1 starves due to Client N waiting for a very long time to get results from the server.
* Lameducking
  + Servers must gracefully shut down by rejecting new requests and continuing to process in-flight ones. Example: Game server maintenance in 15 minutes. Login is disabled, but players in-game have time to finish one more dungeon or put their character in a safe spot before logging off.
* Flow control
  + Computing power and network capacity are often unbalanced between client and server. Flow control allows for better buffer management as well as providing protection from DOS by an overly active peer.
* Pluggable and API extensible
  + gRPC is only one of possibly many running infrastructures. Large distributed systems need security, health-checking, load-balancing and failover, monitoring, tracing, logging, and so on. Implementations should provide extensions points to allow for plugging in these features and, where useful, default implementations.
* Standardized status codes
  + Client typically respond to errors returned by API calls in a limited number of ways. he status code namespace should be constrained to make these error handling decisions clearer. E.g. error 404 not found.

**GETTING STARTED WITH GRPC**

* The “message” format uses a proto3 syntax: <https://protobuf.dev/programming-guides/proto3/>

A screenshot of a computer code

Description automatically generated

* The gRPC is an API that handles the message passing, with several needed RPC essentials 🡪 see design principles.

**Typical workflow**

1. Formulate your customized message in a **.proto** file.

A screenshot of a computer program

Description automatically generated

1. Build your .proto file using the provided compilers. This would generate a .h and a .cc file you could import to your application. Both client and server classes would be generated.

A screen shot of a computer code

Description automatically generated

A screenshot of a computer code

Description automatically generated

1. Add your customized implementations to the server and clients.

Refer to GDPARCM\_HO8-gRPCDemo

* Note on directory structure: gRPC and protoc executables are needed. Protoc is a command-line-based compiler and is sensitive to whitespaces. Put your project in a **non-whitespace** directory.
* Install gRPC through the vcpkg. Protoc is automatically installed.

A screenshot of a computer

Description automatically generated

* A “proto\_compile.sh” is provided to facilitate the compilation of .proto files. To execute the bash script, you can download Git bash, which could have been downloaded with Github Desktop.

A screenshot of a computer code

Description automatically generated

**EXAMPLE:** Controlling a remote machine with the following operations: **UPLOAD A FILE, CREATE NEW FOLDER, OPEN A FOLDER.**

Proto file:

|  |
| --- |
| syntax = "proto3";  service RemoteOperations {  rpc UploadFile(FileRequest) returns (Result);  rpc CreateFolder(FolderRequest) returns (Result);  rpc OpenFolder(FolderRequest) returns (Result);  }  message FileRequest {  string file\_content = 1;  }  message FolderRequest {  string folder\_name = 1;  }  message Result {  bool success = 1;  string message = 2;  } |

Server CPP file:

A screenshot of a computer program

Description automatically generated

**Notes:**

* Wait() is a blocking call.
* Refer to the following link for a sample on async/non-blocking server implementation: <https://github.com/grpc/grpc/blob/master/examples/cpp/helloworld/greeter_async_server.cc>

Client CPP file:

+A screenshot of a computer program

Description automatically generated

**Activities:**

1. Propose a format for your client and server stubs and how this will be interpreted for your P3 project. Provide a .proto-related format of the procedures and parameters needed.
2. In your respective P3 project, present an initial working demo of your server and client design. Discuss which features are going to be implemented by the server and which would be your client.