Remote invocation

Distributed software systems
CdLM Informatica - Università di Bologna

Agenda

Inter-system communication and coordination

Request-reply protocols

Remote Procedure Call - RPC

Remote Method Invocation - RMI

Interface Description Languages (IDL)

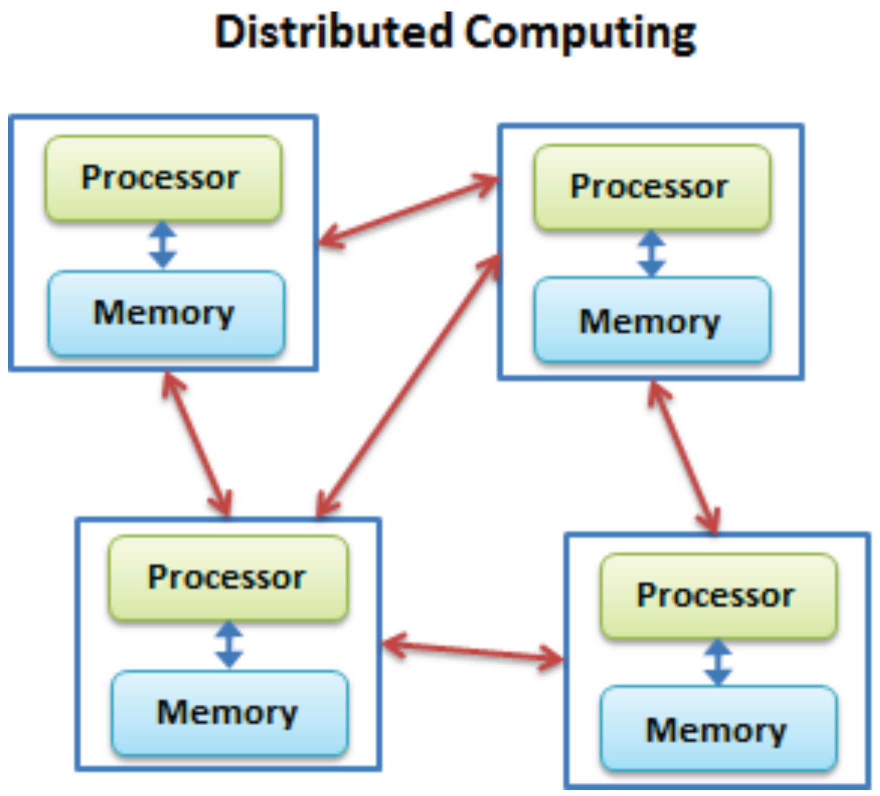
Programming a computer

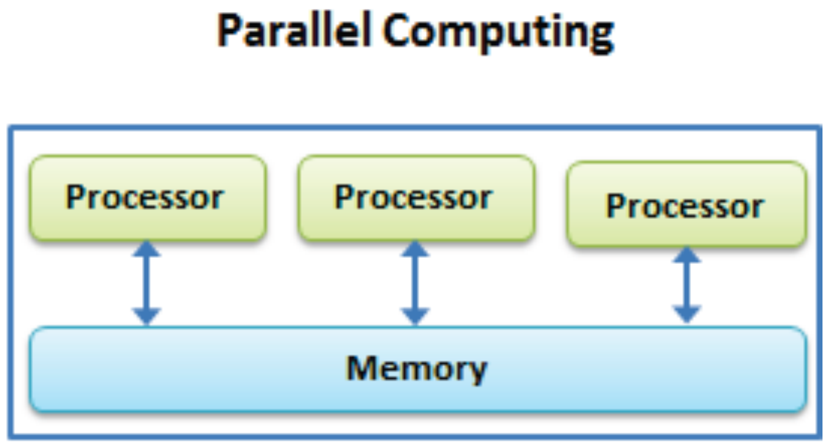
When you program a (single) computer:

1. You need an algorithm that usually is sequential: one step at time
2. You write the program corresponding to the algorithm in some programming language, that is also sequential
3. You ask the operating system to execute the program and give you the result

Remark: The execution, namely the process (or the processes and threads) executing your program, is local to your computer/operating system

Parallel vs distributed computing





Multiprocessor with shared memory (multicore)

Concurrency by Interleaving (one processor only)

1

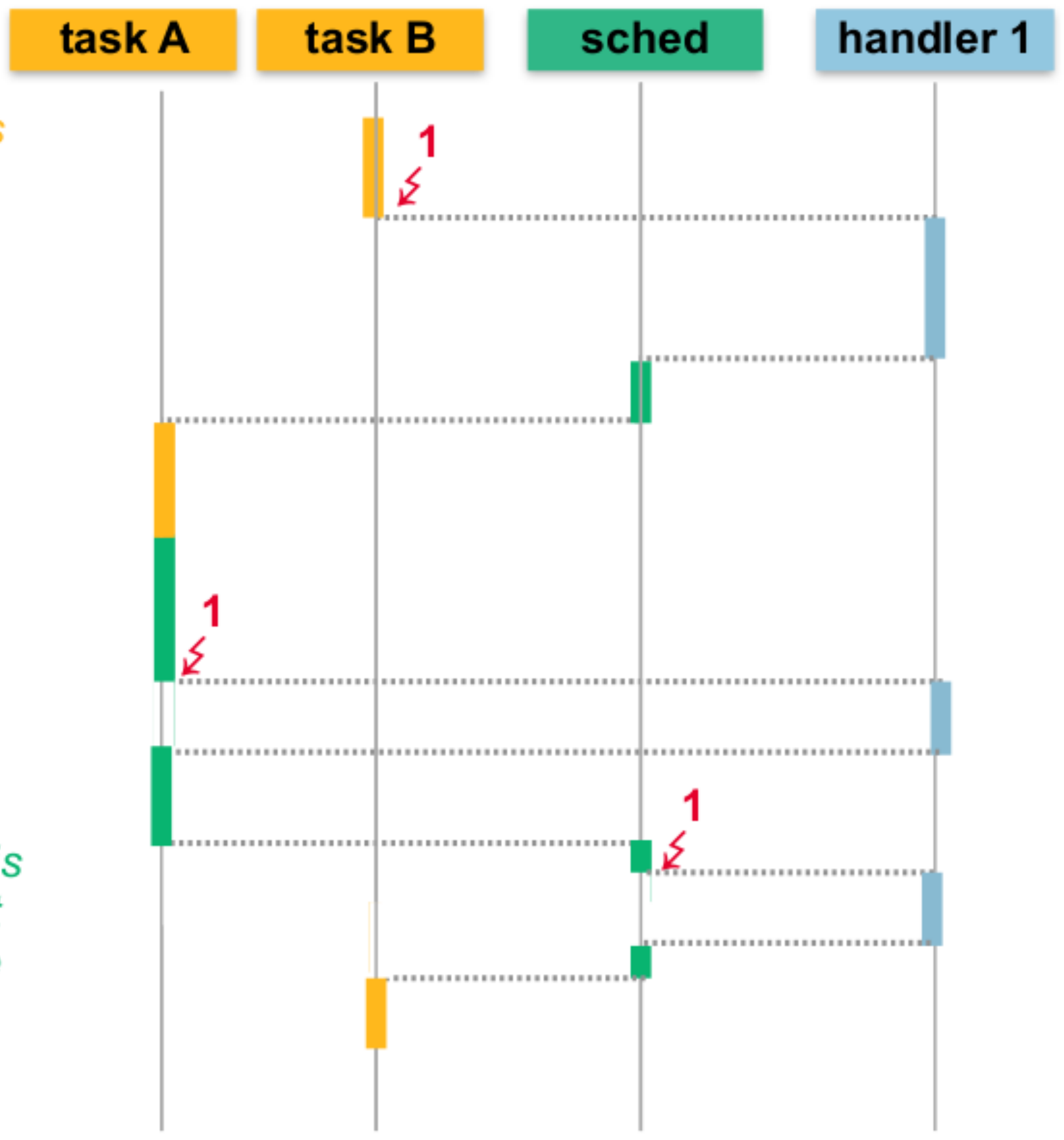
application code is interruptible and preemptible

2

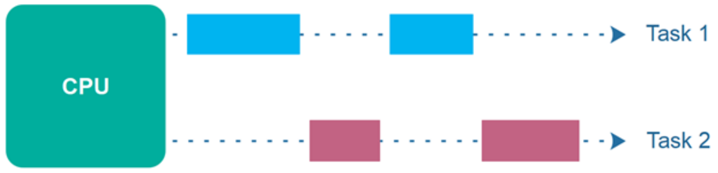
OS code is interruptible but not preemptible

3

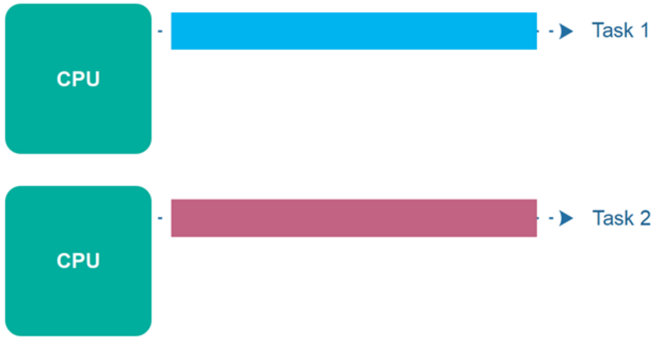
scheduler code is interruptible but not preemptible



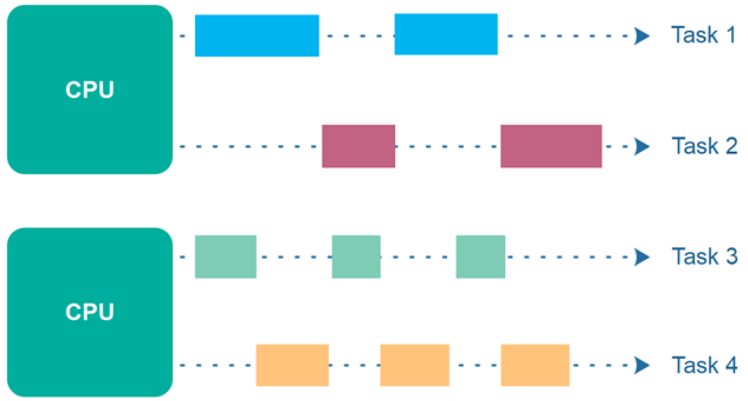
A distributed (multiprocessor) system is true concurrent



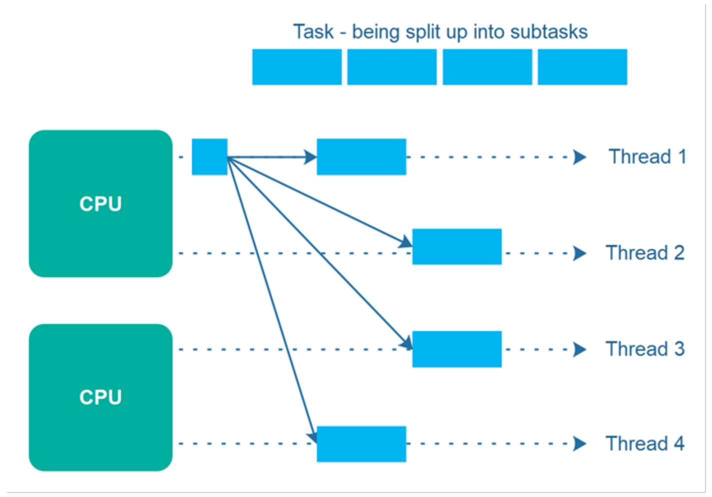
Uniprocessor
(simulated concurrency)



Multiprocessor
(true concurrency)



Simulated + true concurrency



Multithreads: master+workers

Programming locally vs programming remotely

Multiple processors

No shared memory

True concurrent execution

Remote variables?

Remote procedures?

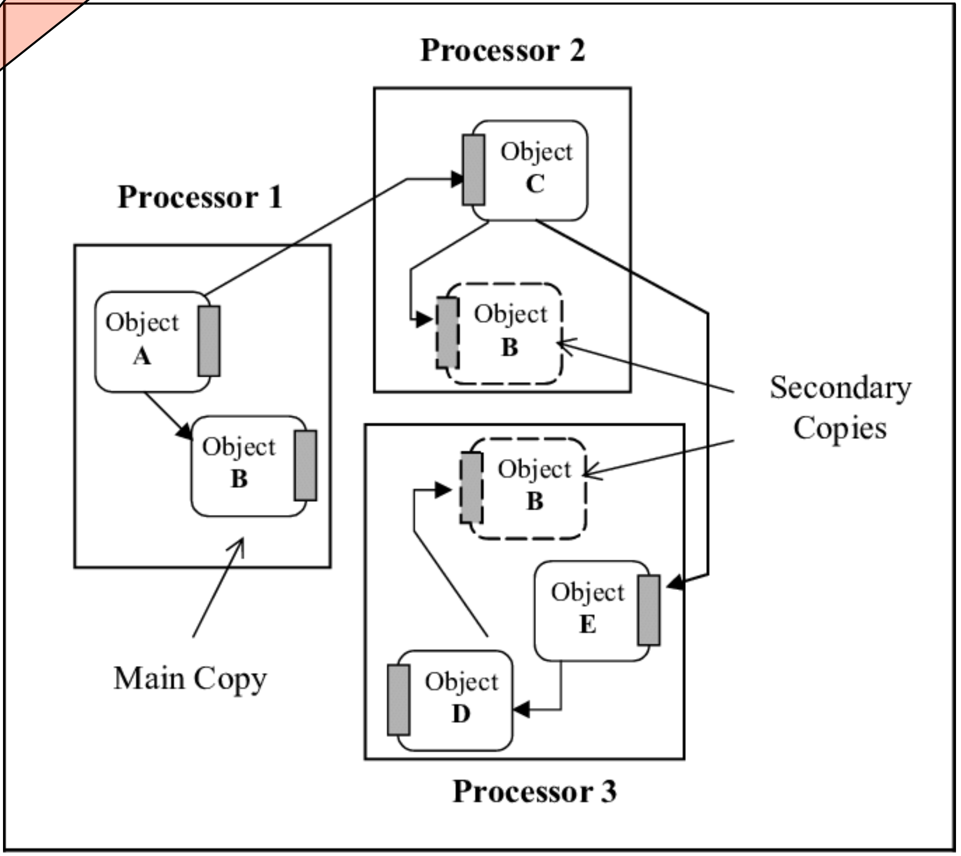
Remote invocation types?

Remote data?

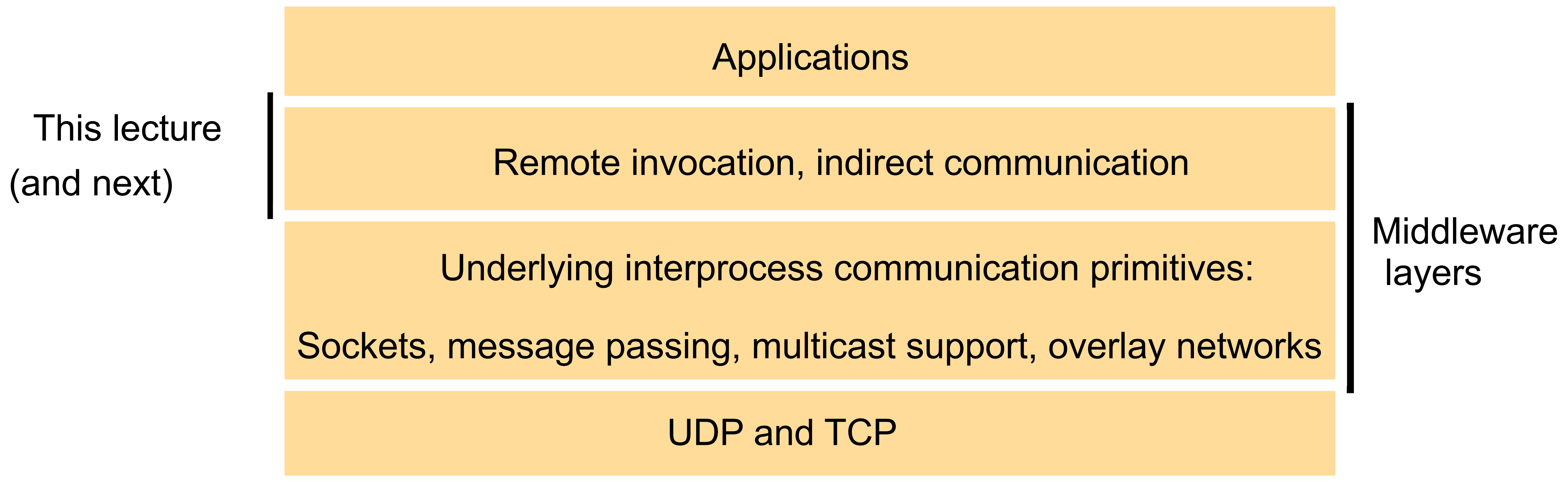
Network partitioning?

Mobile code, data, objects?

Middleware!



Middleware layers

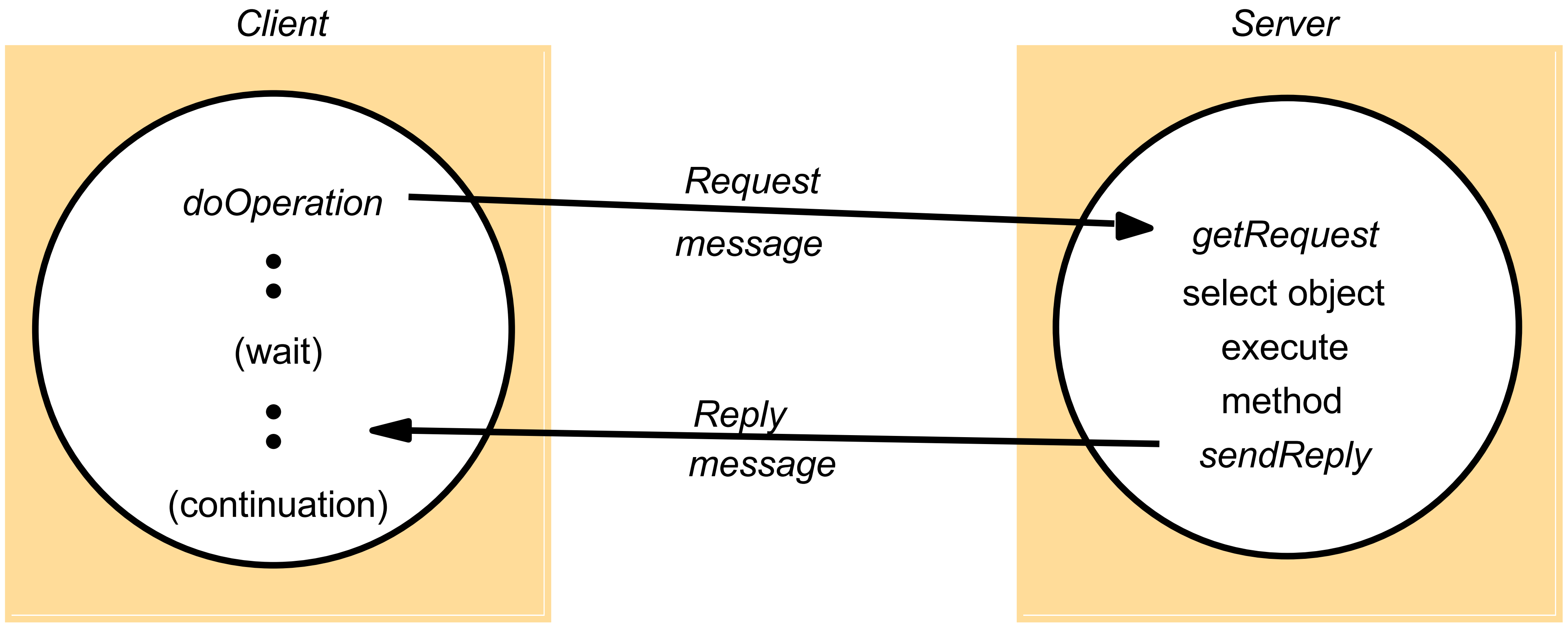


Applications
This lecture
(and next)
Remote invocation, indirect communication
Underlying interprocess communication primitives:
Sockets, message passing, multicast support, overlay networks
Middleware layers
UDP and TCP

Process interaction

Awareness	Relationship	Influence that a process has on the other	Potential control problems
Processes are unaware of each other	Competion (over resources)	• No dependency • Timing maybe affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation
Processes indirectly aware (eg. Shared object)	Cooperation by sharing	• dependency producer-consumer • Timing maybe affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation • Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	• dependency producer-consumer • Timing maybe affected	• Deadlock (consumable resource) • Starvation

Request-reply communication



This request-reply protocol matches requests to replies. It may be designed to provide certain delivery guarantees.
If UDP datagrams are used, the delivery guarantees must be provided by the request-reply protocol, which may use the server reply message as an acknowledgement of the client request message.

Operations of the request-reply protocol

public byte[] doOperation (RemoteRef s, int operationId, byte[] arguments)

sends a request message to the remote server and returns the reply.

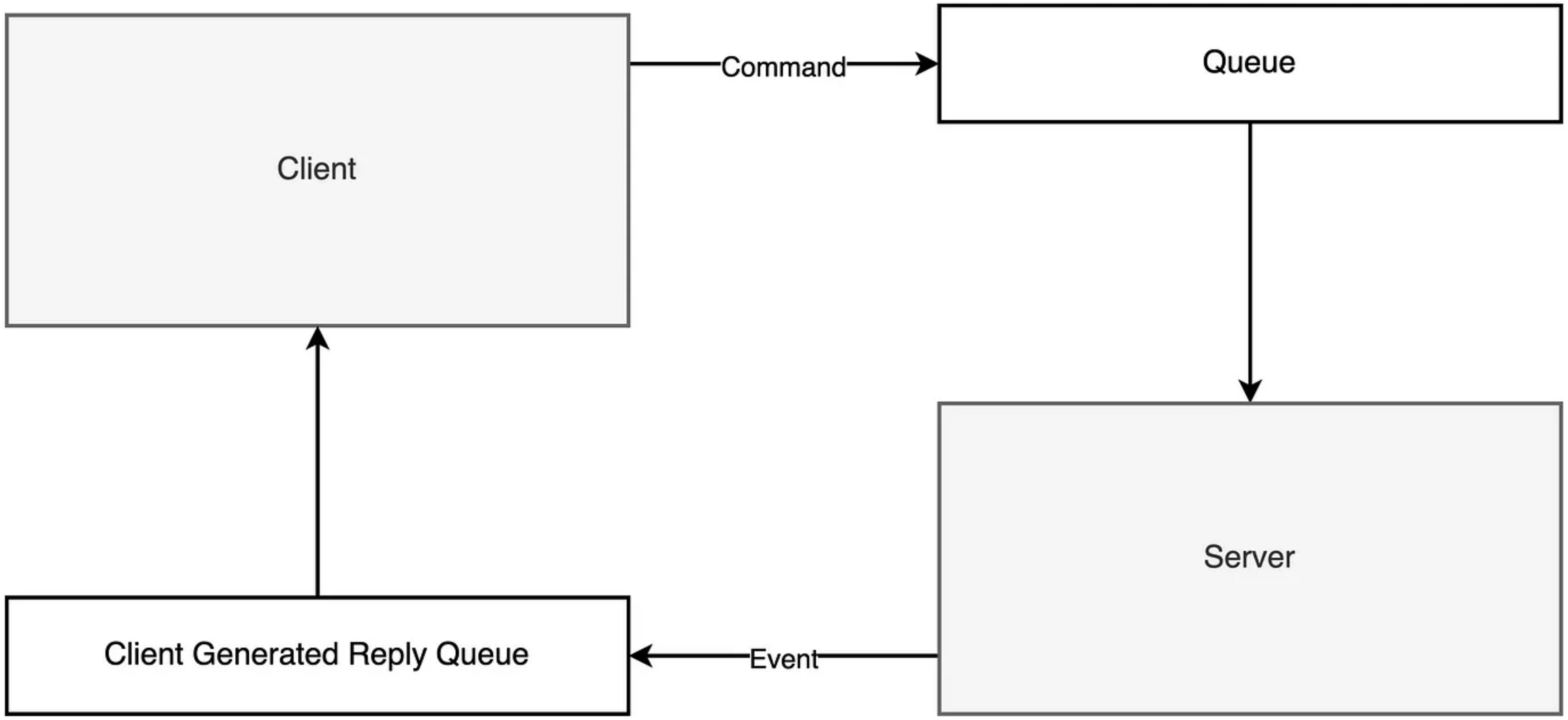
The arguments specify the remote server, the operation to be invoked and the arguments of that operation.

public byte[] getRequest ();

acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);

sends the reply message reply to the client at its Internet address and port.



https://en.wikipedia.org/wiki/Request–response

Request-reply message structure

messageType	*int (0=Request, 1= Reply)*
requestId	*int*
remoteReference	*RemoteRef*
operationId	*int or Operation*
arguments	*array of bytes*

Request reply exchange protocols

Name	*Messages sent by*		
	Client	*Server*	*Client*
R	*Request*		
RR	*Request*	*Reply*	
RRA	*Request*	*Reply*	*Acknowledge reply*

The protocols produce differing behaviours in presence of communication failures and are used for implementing various types of request behaviour:

- the request (R) protocol;
- the request-reply (RR) protocol;
- the request-reply-acknowledge reply (RRA) protocol.

HTTP as a Request-Replay protocol

HTTP is implemented over TCP

In the original version of the protocol, each client-server interaction consisted of the following steps:

1. The client requests and then the server accepts a connection at the default server port or at a port specified in the URL.
2. The client sends a request message to the server.
3. The server sends a reply message to the client.
4. The connection is closed

HTTP methods: GET

- Each client request specifies the name of a **method** to be applied to a resource at the server and the URL of that resource.
- The reply reports on the status of the request. Requests and replies may also contain resource data, the contents of a form or the output of a program resource run on the web server.

GET: Requests the resource whose URL is given as its argument.

If the URL refers to data, then the web server replies by returning the data identified by that URL.

If the URL refers to a program, then the web server runs the program and returns its output to the client.

With GET, all the information for the request is provided in the URL

HTTP methods

HEAD: This is identical to GET, but it does not return any data. However, it returns metadata, such as the time of last modification, its type or its size

POST: Specifies the URL of a resource (for example a program) that can deal with the data supplied in the body of the request. The processing carried out on the data depends on the function of the program specified in the URL. This method is used when the action may modify data on the server.

PUT: Requests that the data supplied in the request is stored with the given URL as its identifier, either as a modification of an existing resource or as a new resource.

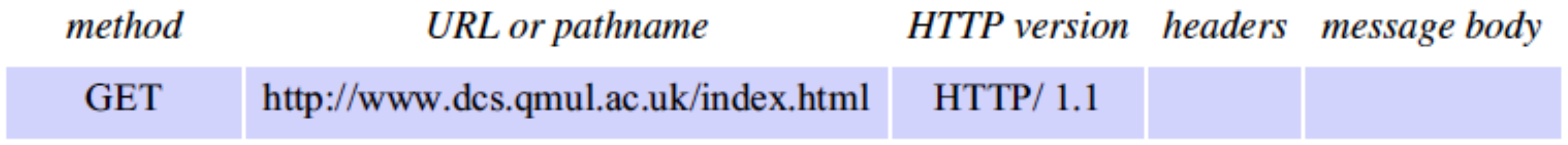
DELETE: The server deletes the resource identified by the given URL. Servers may not always allow this operation, in which case the reply indicates failure. OPTIONS: The server supplies the client with a list of methods it allows to be applied to the given URL (for example GET, HEAD, PUT)

TRACE: The server sends back the request message. Used for diagnostic purposes.

HTTP methods

- The operations PUT and DELETE are idempotent, but POST is not necessarily so because it can change the state of a resource.
- The others are safe operations in that they do not change anything

HTTP request message (method GET)



method	*URL or pathname*	*HTTP version*	*headers*	*message body*
GET	http://www.dcs.qmul.ac.uk/index.html	HTTP/ 1.1		

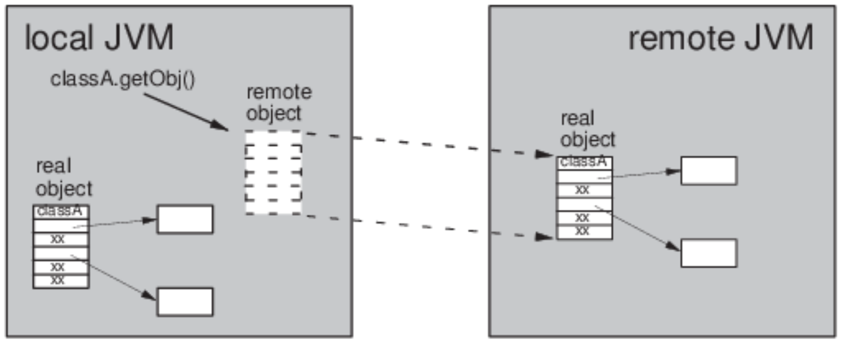
The Request message specifies the name of a method, the URL of a resource, the protocol version, some headers and an optional message body.
We show the contents of an HTTP Request message whose method is GET

HTTP *Reply* message

HTTP version	*status code*	*reason*	*headers*	*message body*
HTTP/1.1	200	OK		resource data

A Reply message specifies the protocol version, a status code and ‘reason’, some headers and an optional message body.
The status code and reason provide a report on the server’s success or otherwise in carrying out the request

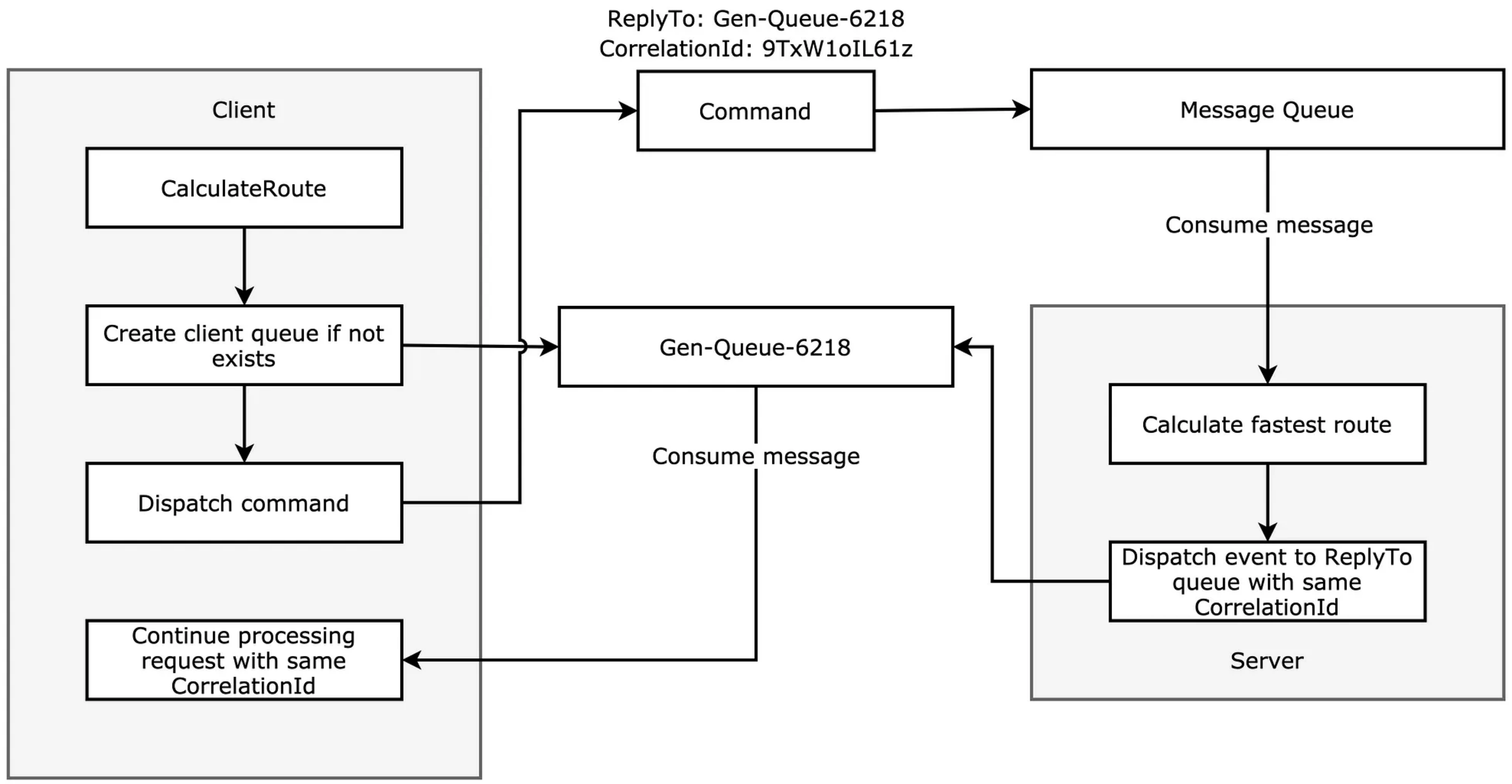
RPC and RMI



There are two main remote invocation techniques for communication in distributed systems:

- The remote procedure call (RPC) approach extends the abstraction of the procedure call to distributed environments, allowing a calling process to call a procedure in a remote node as if it were local
- Remote method invocation (RMI) is similar to RPC but for **distributed objects**, with added benefits in terms of using oo programming concepts: extending the concept of an object reference to distributed environments, and allowing the use of object references as parameters in remote invocations.

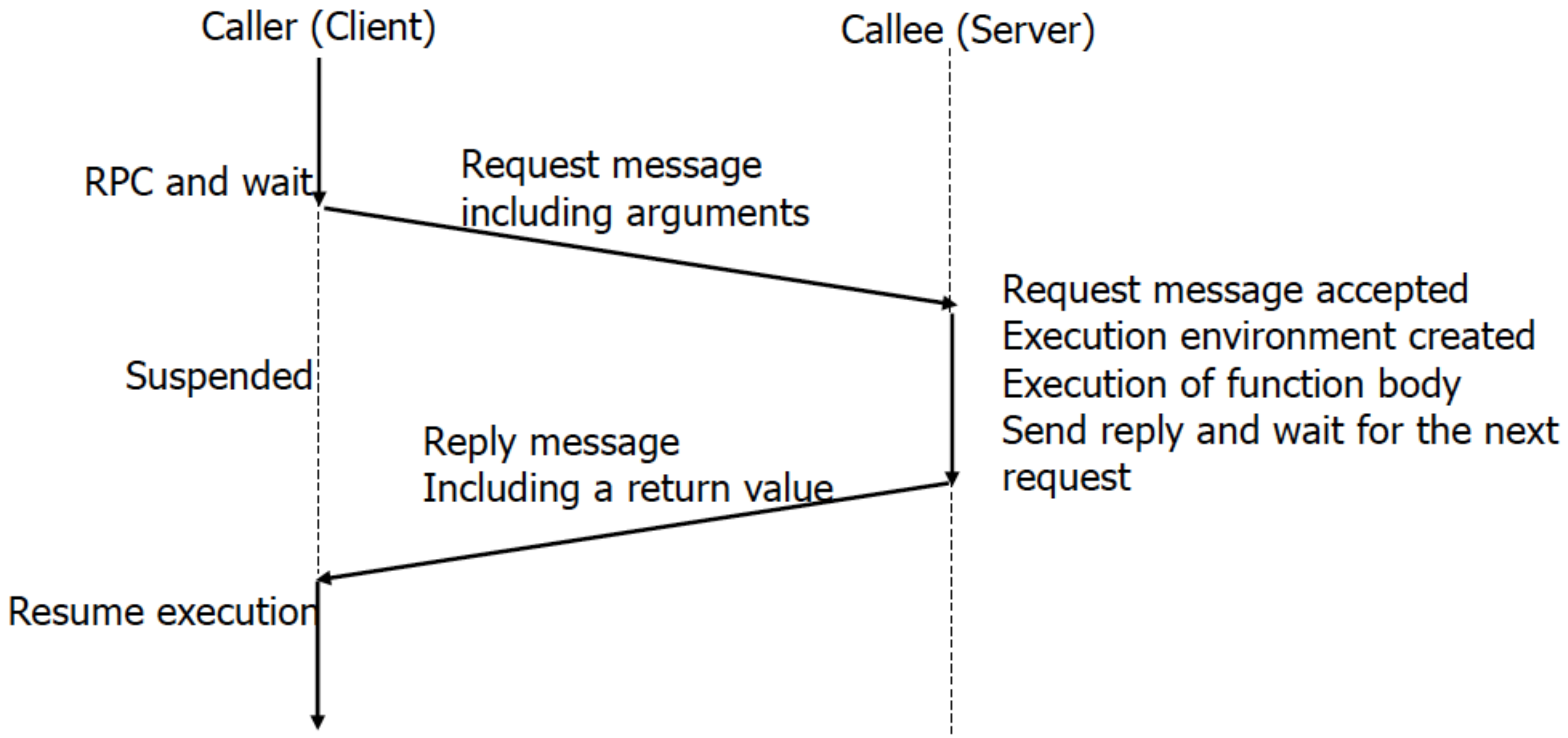
RPC: Application Example


ReplyTo: Gen-Queue-6218
CorrelationId: 9TxW1oIL61z
Client
CalculateRoute
Create client queue if not exists
Dispatch command
Continue processing request with same CorrelationId
Command
Message Queue
Consume message
Gen-Queue-6218
Consume message
Calculate fastest route
Dispatch event to ReplyTo queue with same CorrelationId
Server

application for booking a taxi

1. When a user requests a taxi we will need to calculate the fastest route.
2. Calculating the fastest route is a heavy operation.
3. Because of that, we will move this logic into a separate service that we can scale when necessary.
4. We can call this service by using RPC

RPC and RMI: a common pattern



Implementation issues of remote calls

Transparency property

- Syntactic transparency
- Semantic transparency

Similarity between local and remote procedure calls

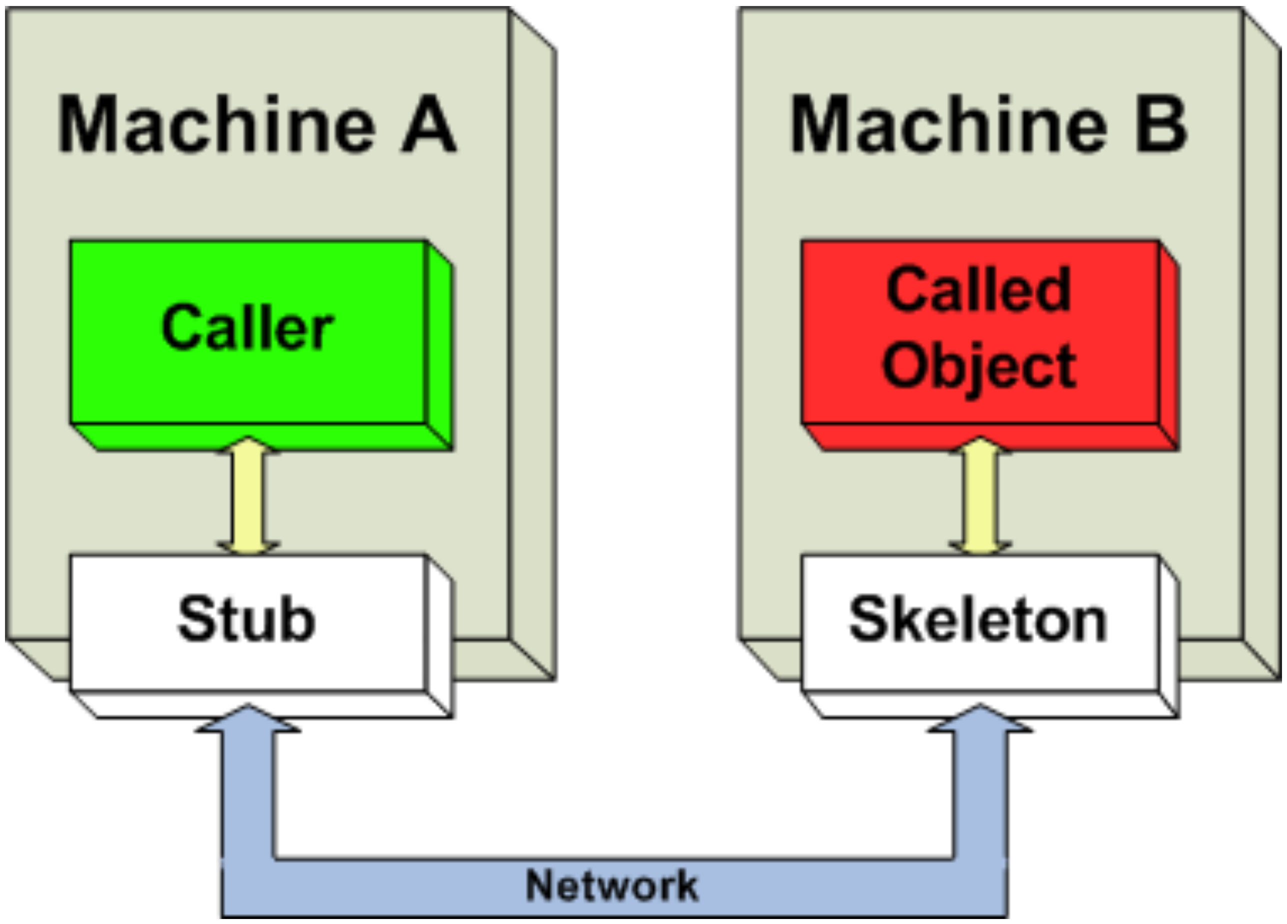
- Caller capable of passing arguments (automatic marshalling)
- Caller suspended till a return from a function
- Callee capable of returning a value to caller

Difference between local and remote procedure calls

- No call by reference and no pointer-involved arguments
- Error handling required for communication (Ex. RemoteException in Java)
- Performance much slower than local calls

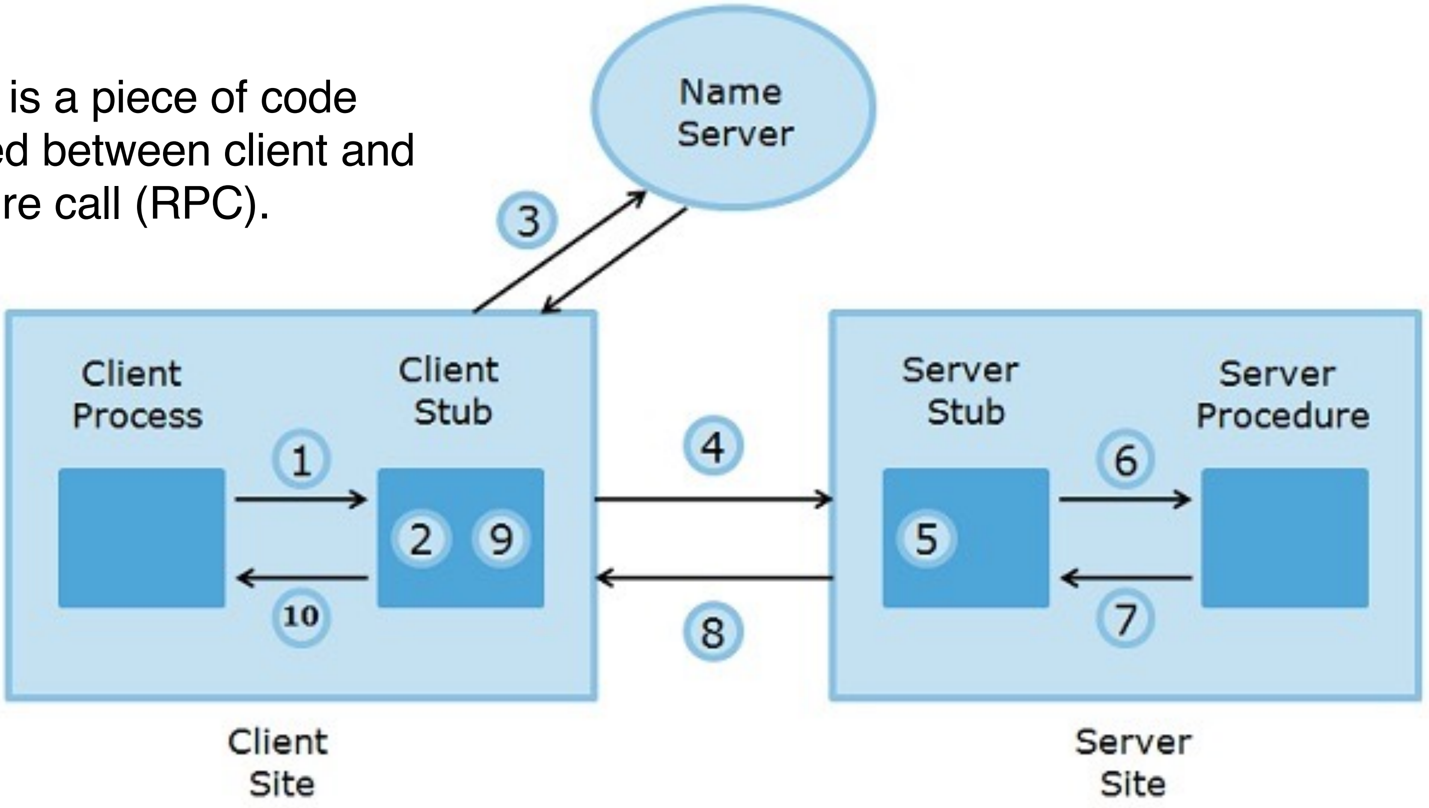
Implementing a RPC

Only the stubs, which are automatically generated by the compiler, know that the call is remote
For the programmer (imagine launching her main program on machine A calling a remote object on B) the remote machine is «transparent»



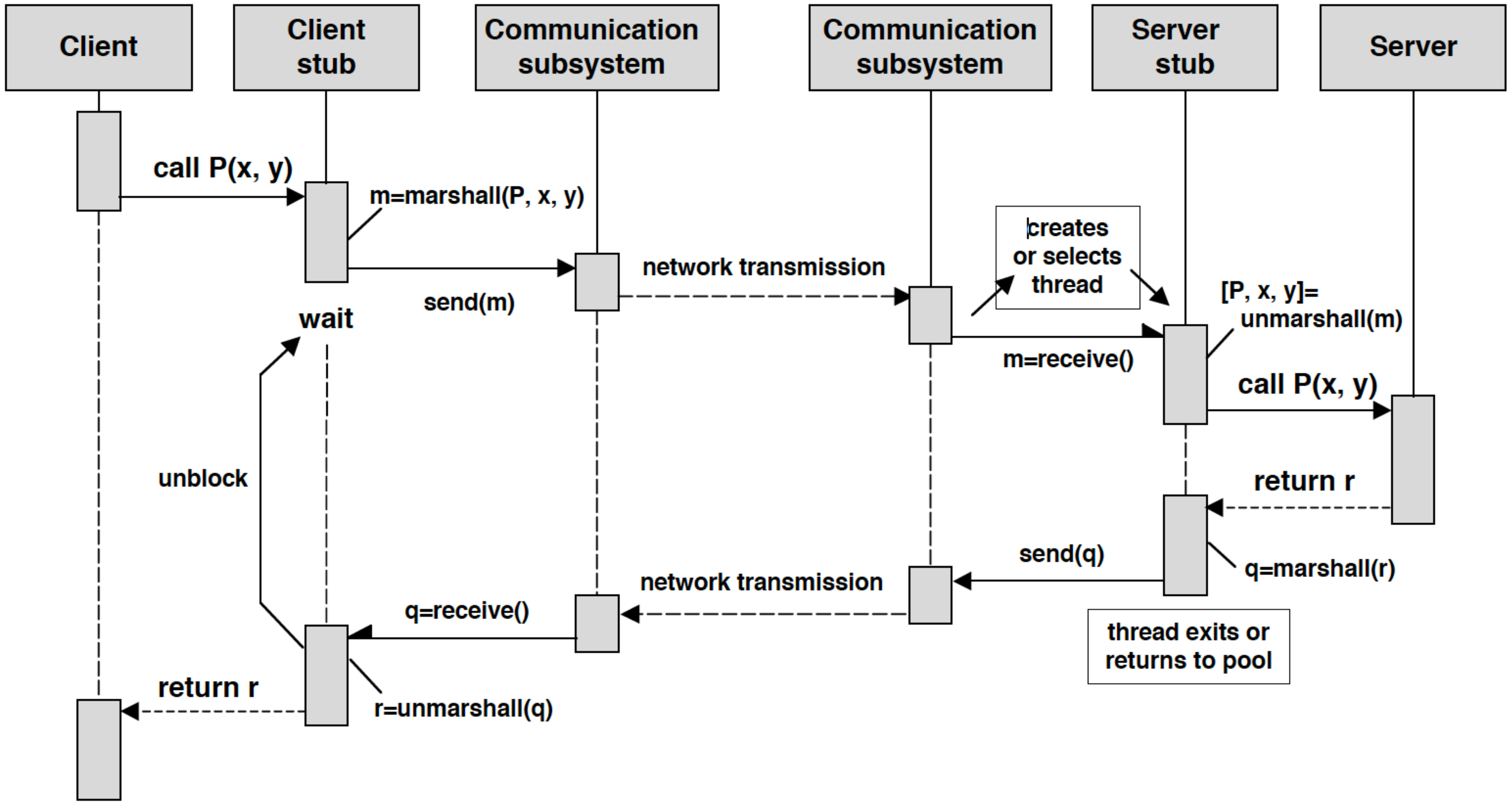
RPC

A **stub** in distributed computing is a piece of code that converts parameters passed between client and server during a remote procedure call (RPC).



The client and server use different address spaces, so parameters used in a function (procedure) call have to be converted, otherwise the values of those parameters could not be used, because pointers to parameters in one computer's memory would point to different data on the other computer.
The client and server may also use different data representations, even for simple parameters (e.g., big-endian versus little-endian for integers).
Stubs perform the conversion of the parameters, so a remote procedure call looks like a local function call for the remote computer.

RPC: overall flow of control



Stub and skeleton in Java RMI

Stub: This is an object that acts as an entry point for the client object to route any outgoing requests. It exists on the client JVM and represents the handle to the remote object. If any object invokes a method on the stub object, the stub establishes RMI by following these steps:

1. It initiates a connection to the remote machine JVM.
2. It marshals (write and transmit) the parameters passed to it via the remote JVM.
3. It waits for a response from the remote object and unmarshals (read) the returned value or exception, then it responds to the caller with that value or exception.

Skeleton: This is an object that behaves like a gateway on the server side. It acts as a remote object with which the client objects interact through the stub. This means that any requests coming from the remote client are routed through it. If the skeleton receives a request, it establishes RMI through these steps:

1. It reads the parameter sent to the remote method.
2. It invokes the actual remote object method.
3. It marshals (writes and transmits) the result back to the caller (stub).

Design issues for RPC

Three issues are important:

1. the style of programming promoted by RPC –that is, programming with interfaces;
2. the call semantics associated with RPC;
3. the issue of transparency and how it relates to remote procedure calls.

Interface description language

- An interface description language (IDL), is a language used to describe a software component's application programming interface (API).
- IDLs describe an interface in a language-independent way, enabling communication between software components written in different languages (for example, between those written in C++ and those written in Java)

https://docs.oracle.com/javase/7/docs/technotes/guides/idl/jidlExample.html

API- based programming

- The use of API to allow teams to collaborate raises the question of how an API can change and be reprogrammed.
- If an API is changed, e.g. by adding a new method, old code written to implement the interface will no longer compile – and in the case of dynamically-loaded or linked plugins, will either fail to load or link, or crash at runtime.
- There are two basic approaches for dealing with this problem:
  1. a new interface may be developed with additional functionality, which might inherit from the old interface
  2. a software versioning policy may be imposed to interface implementors, to allow forward-incompatible, or even backward-incompatible, changes in future "major" versions of the platform
- Both of these approaches have been used in the Java platform.

Interface definition language (IDL)

- An RPC mechanism integrated with a programming language has to include a notation for defining interfaces, allowing input and output parameters to be mapped onto the language’s normal use of parameters.
- This approach is useful when all the parts of a distributed application can be written in the same language. It is also convenient because it allows the programmer to use a single language for local and remote invocation.
- However, many existing useful services are written in different languages. It would be beneficial to allow programs written in a variety of languages, to access them remotely.
- Interface definition languages (IDLs) are designed to allow procedures implemented in different languages to invoke one another: an IDL provides a notation for defining interfaces in which each of the parameters of an operation may be described as for input or output in addition to having its type specified

IDL

- An *interface description* (or *definition*) *language* (IDL), is a specification language used to describe a software component's application programming interface (API).
- IDLs describe an interface in a language-independent way, enabling communication between software components written in different languages.

IDL: examples

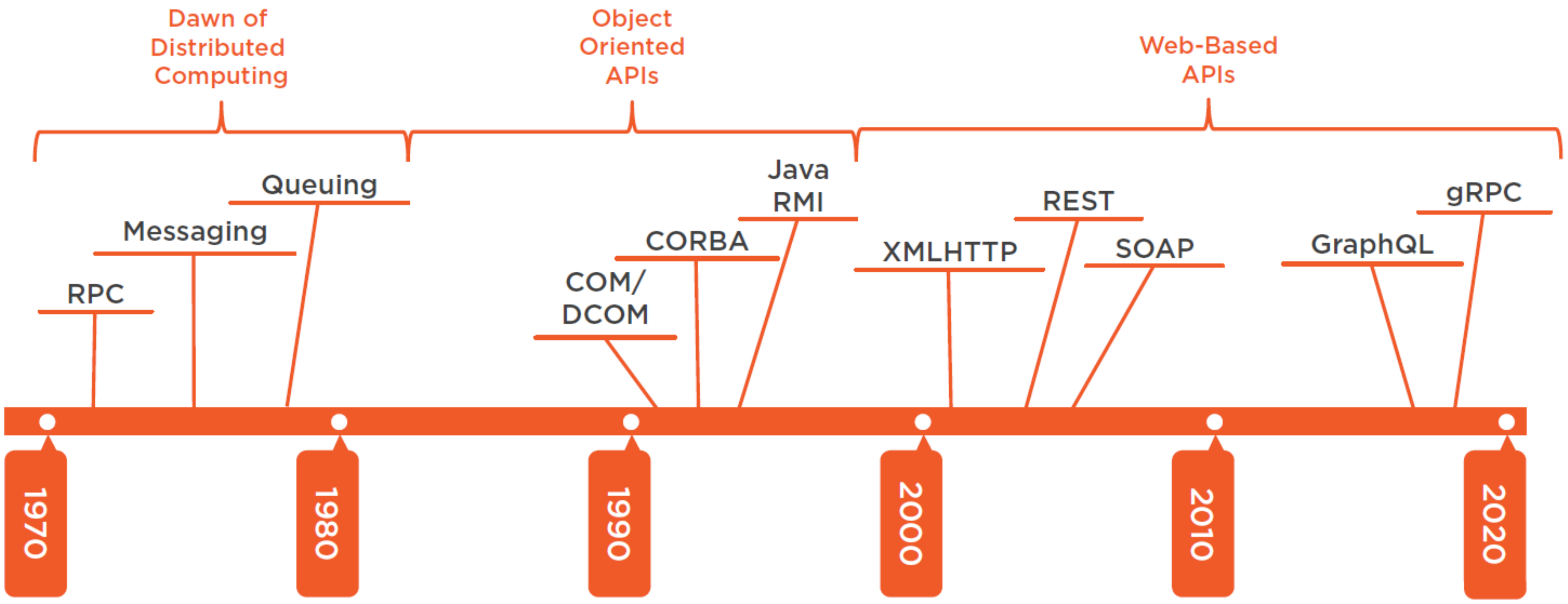
- AIDL: Java-based, for Android; supports local and remote procedure calls, can be accessed from native applications by calling through Java Native Interface (JNI)
- Apache Thrift: from Apache, originally developed by Facebook
- Data Distribution Service: In DDS IDL was identical to OMG IDL until version 3.5 when it branched in order to evolve independently of the OMG specification
- JSON Web-Service Protocol (JSON-WSP)
- Microsoft Interface Definition Language (MIDL) extension of OMG IDL to add support for Component Object Model (COM) and Distributed Component Object Model (DCOM)
- OMG IDL: implemented in CORBA for DCE/RPC services, also selected by the W3C for exposing the DOM of XML, HTML, and CSS documents
- OpenAPI Specification: a standard for REST interfaces.
- Protocol Buffers: Google's IDL for gRPC
- RESTful Service Description Language (RSDL)
- Universal Network Objects: OpenOffice.org's component model
- Web Application Description Language (WADL)
- Web IDL: can be used to describe interfaces intended for implementing in web browsers
- Web Services Description Language (WSDL)

A simple CORBA IDL example

```
// In file Person.idl
struct Person {
        string name;
        string place;
        long year;
} ;
interface PersonList {
        readonly attribute string listname;
        void addPerson(in Person p) ;
        void getPerson(in string name, out Person p);
        long number();
};
```

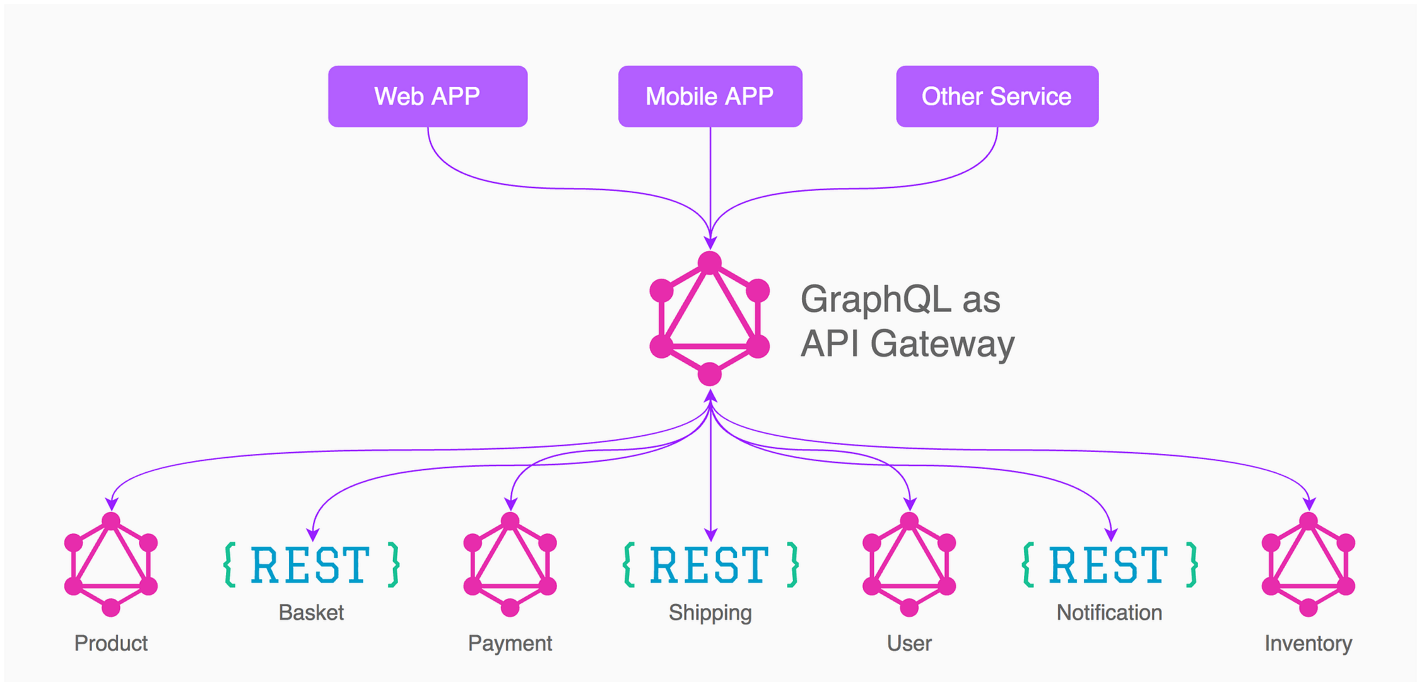
- The interface PersonList specifies the methods available in a remote object that implements that interface.
- Example:
  - the method *addPerson* specifies its argument as input (sent by the client)
  - the method getPerson, that retrieves an instance of Person by name, specifies its second argument as output (computed by the server)

The History of Distributed APIs



GraphQL

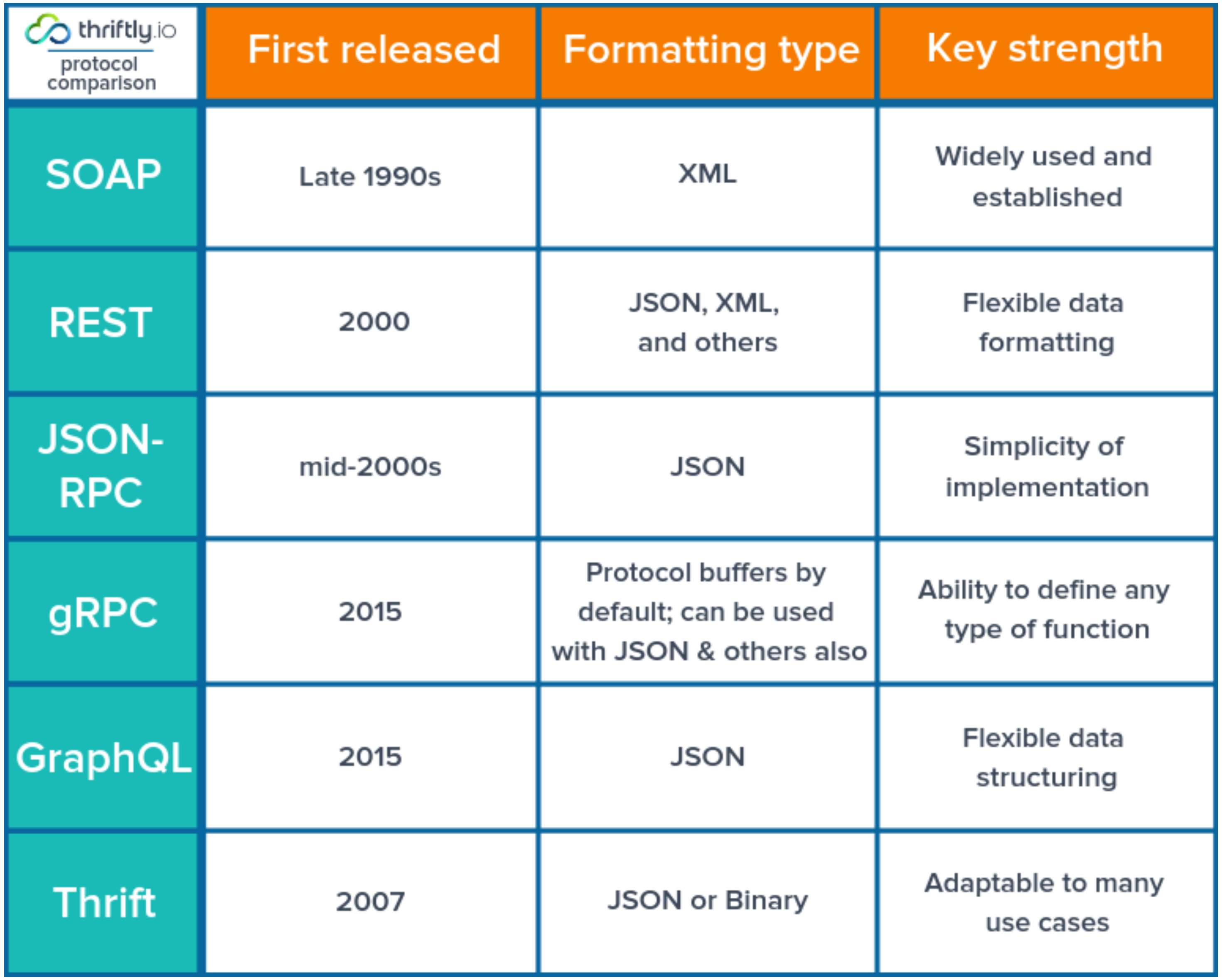
- GraphQL is an open-source data query and manipulation language for APIs and a query runtime engine.
- Introduced by Facebook
- GraphQL enables declarative data fetching where a client can specify what data it needs from an API.
- Instead of multiple endpoints that return separate data, a GraphQL server exposes a single endpoint and responds with the data a client asked for.



Data serialization

- There is a wide variety of data serialization formats, including XML, JSON, BSON, YAML, MessagePack, Protocol Buffers, Thrift and Avro.
- The choice of a specific format for an application is subject to a variety of factors, including data complexity, necessity for humans to read it, latency, and storage space concerns.
- XML is the reference benchmark for the other formats as it was the original implementation. JSON is often described as faster and more light-weight.
- We will look at three newer frameworks: Thrift, Protocol Buffers and Avro, all of which offer efficient, cross-language serialization of data using a scheme, and code generation for Java. Each has a different set of strengths.

Protocol comparison



thriftly.io protocol comparison	First released	Formatting type	Key strength
SOAP	Late 1990s	XML	Widely used and established
REST	2000	JSON, XML, and others	Flexible data formatting
JSON-RPC	mid-2000s	JSON	Simplicity of implementation
gRPC	2015	Protocol buffers by default; can be used with JSON & others also	Ability to define any type of function
GraphQL	2015	JSON	Flexible data structuring
Thrift	2007	JSON or Binary	Adaptable to many use cases

Protocol Buffers (PB)

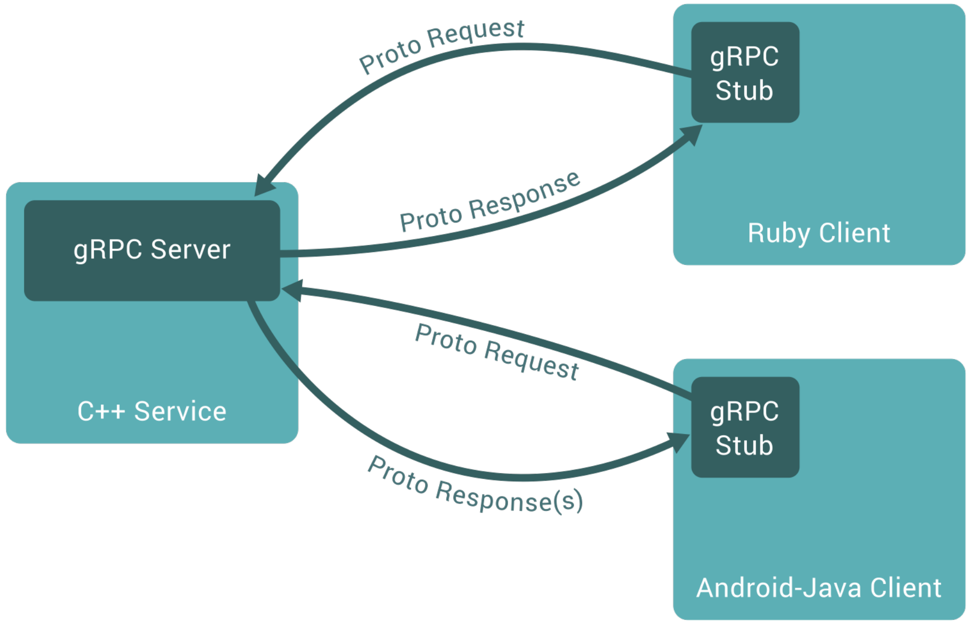
Google’s Protocol Buffers (also known as PB or Protobufs) are quite popular

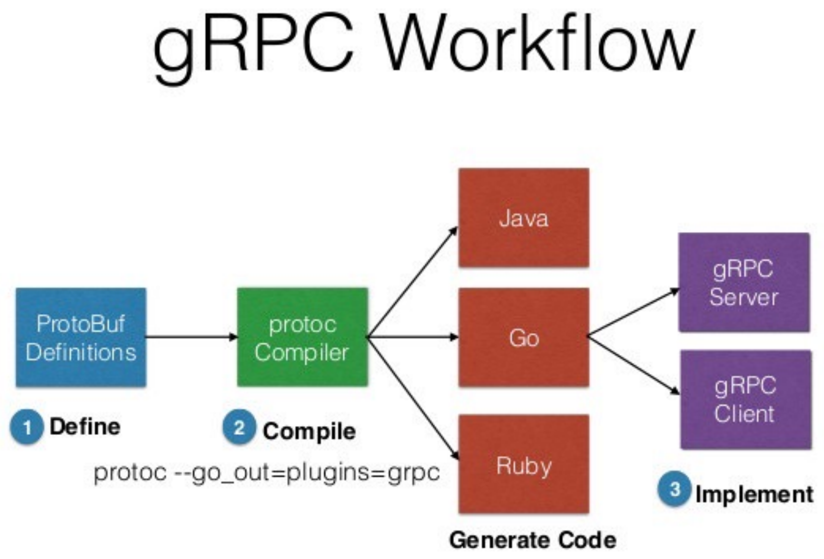
They were designed in 2001 as an alternative to XML for server request/response protocols. They were a proprietary solution at Google until 2008, when they were open-sourced.

They are “the glue to all Google services”, and “battle-tested, very stable and well trusted”.

Google uses PB as the foundation for a custom RPC system that underpins virtually all its inter-machine communication.

gRPC IDL: protocol buffers





https://medium.com/@akshitjain_74512/inter-service-communication-with-grpc-d815a561e3a1

Exercise Creating Simple gRPC Server and Clients

Server Requirements:

1. Create a gRPC server in Python that offers a basic service with the following functionality:
   1. The server should provide a method called Greet that accepts a single string parameter and responds with a greeting message.
   2. Define the service and the Greet method in a .proto file using gRPC IDL.
2. Implement the server logic to handle the Greet method by generating a greeting message using the provided string parameter.

Client Requirements (Client 1):

1. Create a Python client (Client 1) that interacts with the gRPC server.
2. Connect to the server and call the Greet method with a custom name.
3. Print the response received from the server, which should be a greeting message.

Client Requirements (Client 2):

1. Create a second Python client (Client 2) that interacts with the same gRPC server.
2. Connect to the server and call the Greet method with a different custom name.
3. Print the response received from the server, which should be a different greeting message.

Challenge: add a function to clients so that they can chat peer to peer

Conclusions

Programming a distributed computer system is more difficult than programming one computer alone

Controlling concurrency and parallelism is a first problem

Another problem is how different processes on different computers and operating systems, written in different languages, can communicate

The ideas of IDLs (interface description languages) and of interchange formats (eg. XML, JSON, RTF) are important for APIs and microservices

Questions?