TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Communication in Distributed Systems: RPC

Corso di Sistemi Distribuiti e Cloud Computing

A.A. 2026/27

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

Communication in distributed systems

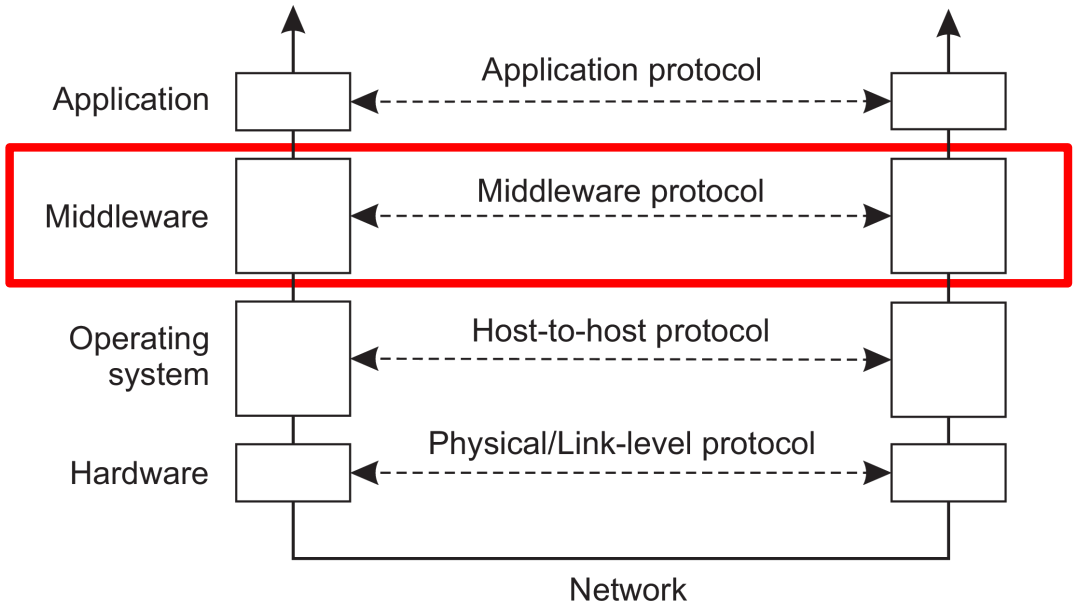
- Message passing: the basics
- Communication is based on sending and receiving messages
- To make this work, parties must agree on many low-level communication details, such as:
  - How many volts represent a 0 or 1 bit?
  - How many bits represent an integer?
  - How does the receiver know when the message ends?
  - How can the receiver detect message corruption and what to do then?

- The good news: we don't care about these low-level details ☺

No AI/ML Processing

Basic networking model and its adaptation

- We already know the solution: divide network communication in layers
  - The well-known ISO/OSI reference model
  - For DS we do not deal with low-level details: we focus on higher-level abstractions built over the network layer
- Adapted layering scheme in distributed systems



Middleware layer

- **Middleware** provides common services and protocols that support different distributed applications and systems
  - General-purpose
  - Application-independent
- Common middleware services
  - Communication: remote procedures calls, message queues, multicasting
  - Naming: identifies and enables sharing of resources
  - Security: ensures secure communication among distributed nodes
  - Distributed consensus: coordinates agreement across multiple distributed nodes
  - Distributed locking: prevents concurrent access to shared resources (e.g., multiple clients editing a file)
  - Data consistency: maintains data consistency across distributed nodes

Communication types in DS

- Let's distinguish communication types by these key aspects:

1. Persistency
   - Transient vs. persistent communication
2. Synchronization
   - Synchronous vs. asynchronous communication
3. Time dependence
   - Discrete vs. streaming communication

No AI/ML Processing

Persistent vs. transient communication

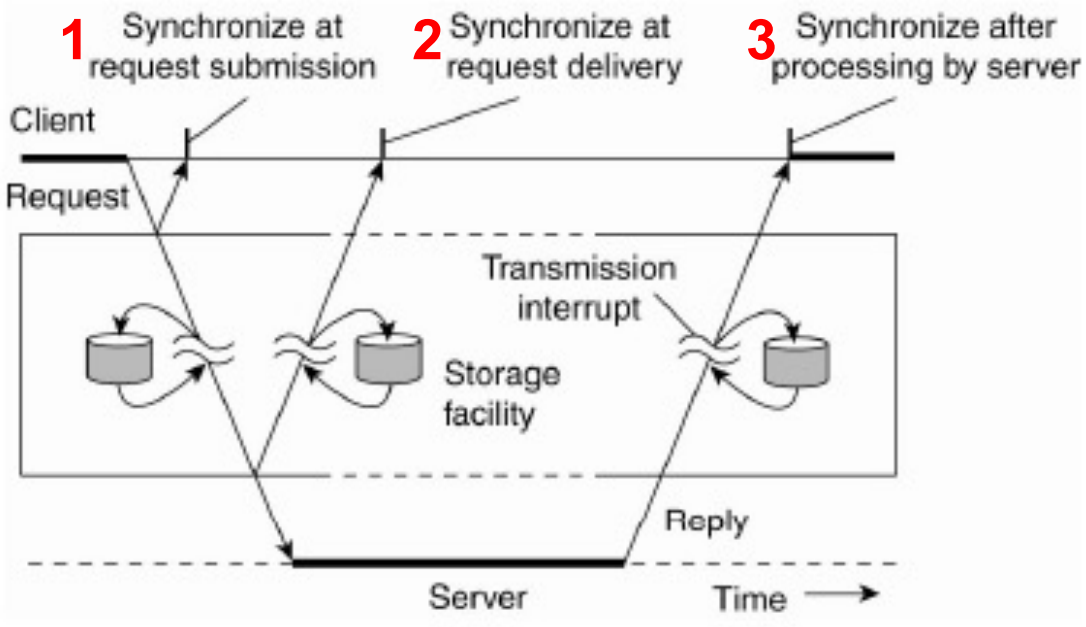
- Persistent communication
  - Message is stored by middleware until it can be delivered
  - Sender can continue execution after submitting the message
  - Receiver does not need to be active at the time of submission

- Transient communication
  - Message is kept only while sender and receiver are both active
  - If delivery is not possible, message is discarded
  - Requires both parties to be executing at the time of communication
  - Transport-layer example: routers store and forward messages, but discard them if forwarding fails

No AI/ML Processing

Synchronous communication…

- Once a message is submitted, the sender is blocked until the operation completes
- Send and receive are *blocking* operations
- Synchronization can occur:
  1. At request submission
  2. At request delivery
  3. After request processing



… vs asynchronous communication

- *Non-blocking send*
  - Once the message is submitted, the sender continues processing
  - The message is temporarily stored by middleware until it is transmitted
- *Receive* can be *blocking or non-blocking*
  - Non-blocking receive: immediately returns, either with data (if available) or with an indication that there is nothing to receive yet
- Benefits: decouples sender and receiver, enabling higher concurrency and responsiveness

No AI/ML Processing

Discrete vs. streaming communication

- Discrete communication
  - Each message forms a complete unit of information

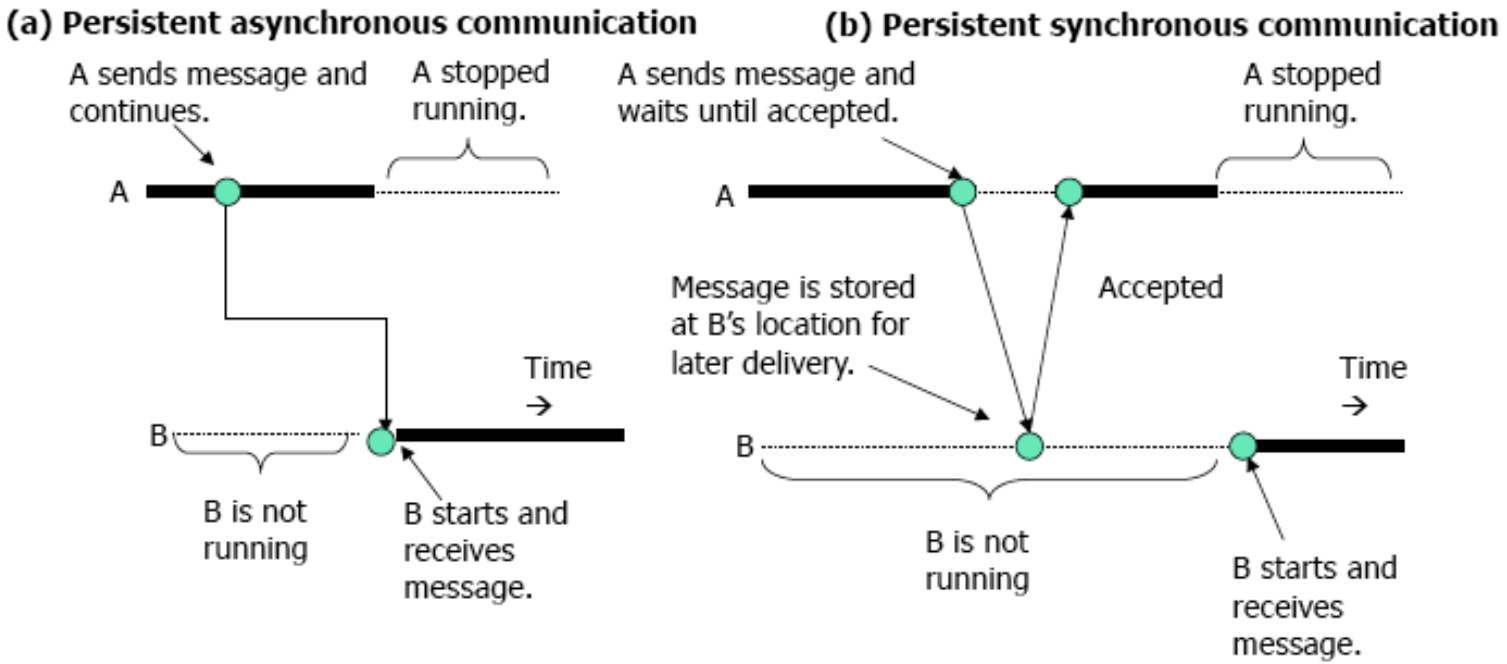
- Streaming communication
  - Involves sending multiple messages, in temporal relationship or related to each other by sending order, which is needed to reconstruct complete information
  - E.g., audio, video

No AI/ML Processing

Combining persistence and synchronization

a) Persistent asynchronous communication
   - Message is stored until receiver is ready, sender does not wait for receiver
   - E.g., email, Teams chat, message-oriented middleware

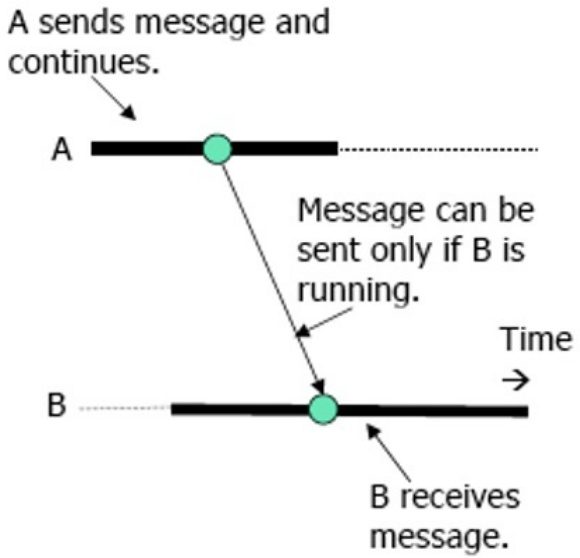
b) Persistent synchronous communication
   - Sender is blocked until message is delivered to receiver
   - Message is stored until delivery is confirmed



Combining persistence and synchronization

c) Transient asynchronous communication

- Sender does not wait
- Message is not stored
- If the receiver is offline or unreachable, message is lost (e.g., UDP)

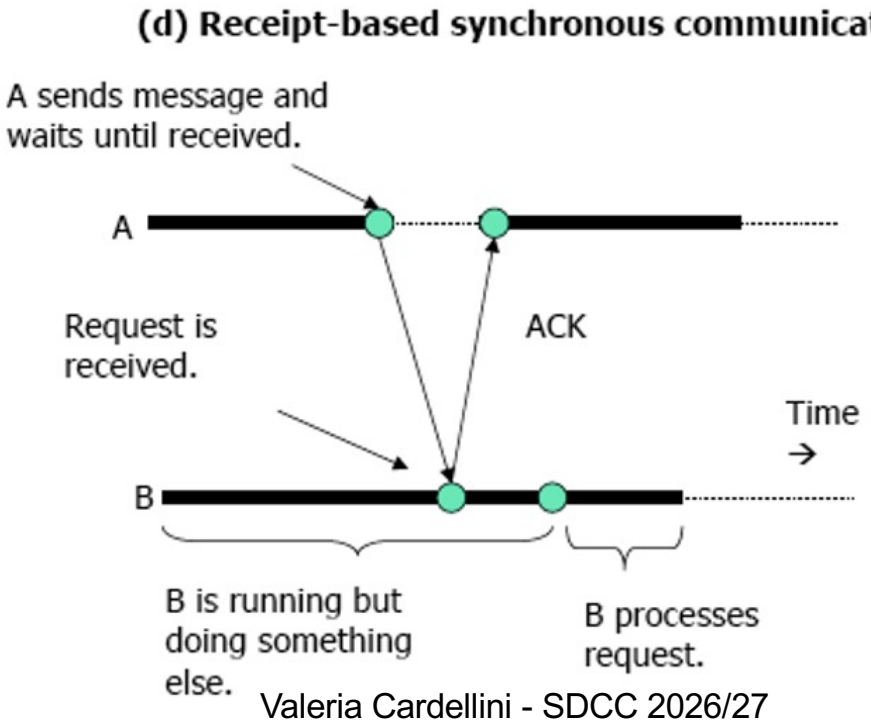


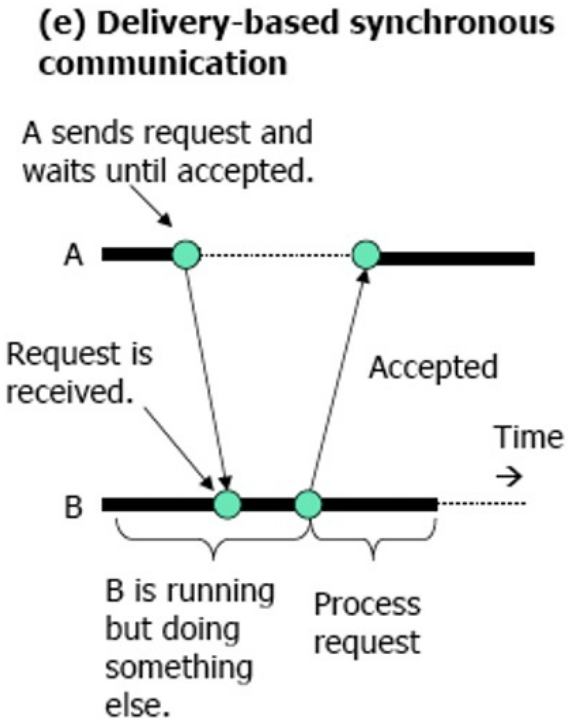
(c)Asynchronous communication

No AI/ML Processing

Combining persistence and synchronization

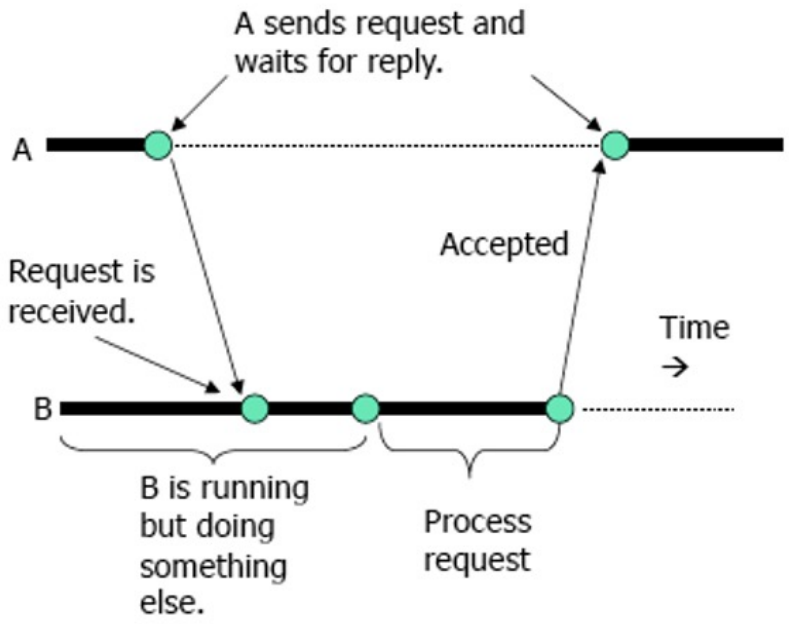
- Transient: message is not stored; sender is blocked until a specific condition is met
- Alternatives
  - d) Receipt-based synchronous: sender is blocked until message reaches the receiver buffer (e.g., asynchronous RPC)
  - e) Delivery-based synchronous: sender is blocked until message is actually delivered to receiver





Combining persistence and synchronization

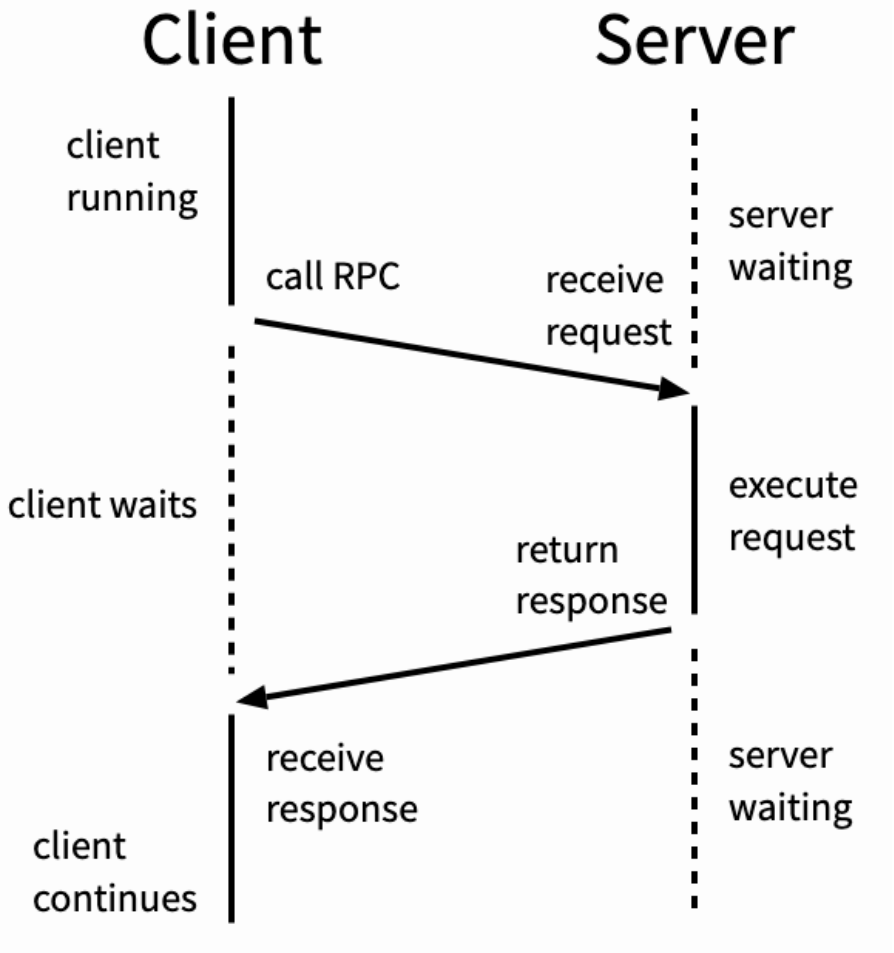
f) Response-based synchronous: sender is blocked until it receives a reply from receiver (e.g., synchronous RPC)



(f) Response-based synchronous communication

No AI/ML Processing

What happens when a client calls a RPC



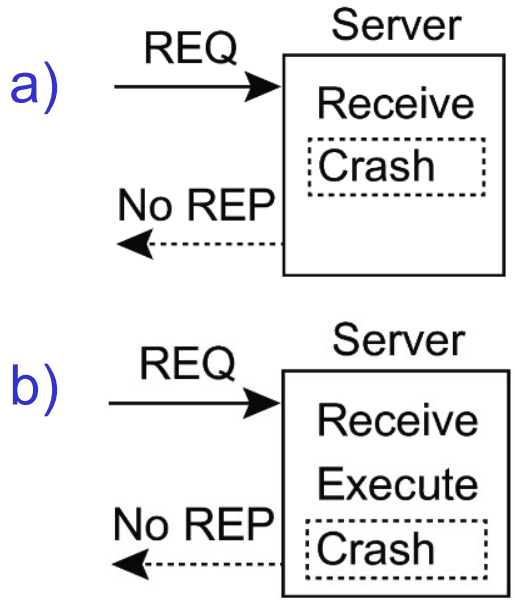
- Which failures can happen?

No AI/ML Processing

Communication failures

- Different communication failures between sender (client) and receiver (server): what can go wrong?

1. Message loss or delay
   - Request or reply is lost or delayed
   - Connection is reset
   - Pitfall: *Network is reliable*
2. Server crashes
   a) Before performing the service
   b) After performing the service
   - Client can't distinguish between a) and b)
3. Client crashes after sending request



No AI/ML Processing

Failure semantics during communication

- In a DS, communication semantics define what guarantees the DS gives about message delivery in the presence of failures
  - **May-be** semantics
  - **At-least-once** semantics
  - **At-most-once** semantics
  - **Exactly-once** semantics



More guarantees, more complexity

- Failure semantics applies both to service processing (e.g., RPC) and message delivery (e.g., MOM)
- We focus on service processing
  - sender = client, receiver = server

No AI/ML Processing

Failure semantics in a nutshell

- **Maybe**: no guarantee that service has been executed or not by server
- **At-least-once**: service is executed at least once (i.e., >= 1)
  - If client does not receive the reply: service *might* have been executed or it might not, or it might have been executed multiple times
- **At-most-once**: service is executed at most once (i.e., 0 or 1)
  - If client receives the reply, it has been processed by server *no more than once*
  - The service may not be executed at all
- **Exactly-once**: service is executed precisely once
  - Even in the face of retries, crashes, or delays
  - Ideal for critical operations, but complex and costly

No AI/ML Processing

Basic mechanisms for implementation

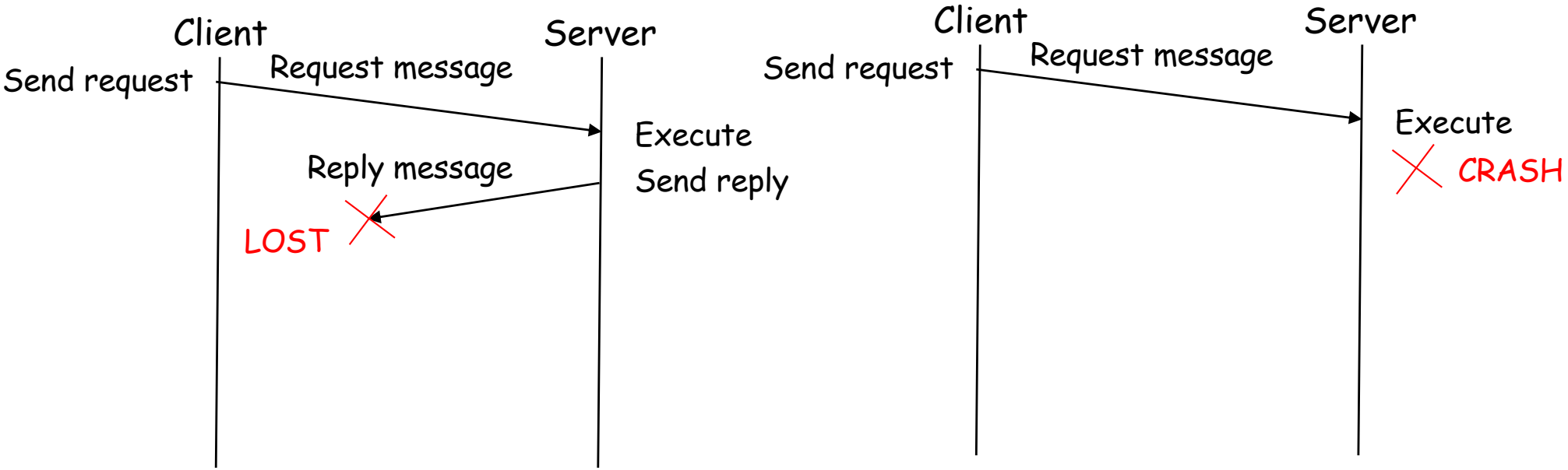
- Basic mechanisms to implement failure semantics

1. Client side: Request Retry (RR1)
   - Client keeps trying until it gets a reply or is confident about server failure after a certain number of failed retries
2. Server side: Duplicate Filtering (DF)
   - Server discards any duplicate request from same client
3. Server side: Result Retransmit (RR2)
   - Server keeps result (reply) so that it can be retransmitted without being computed again when server receives duplicate requests

No AI/ML Processing

May-be semantics

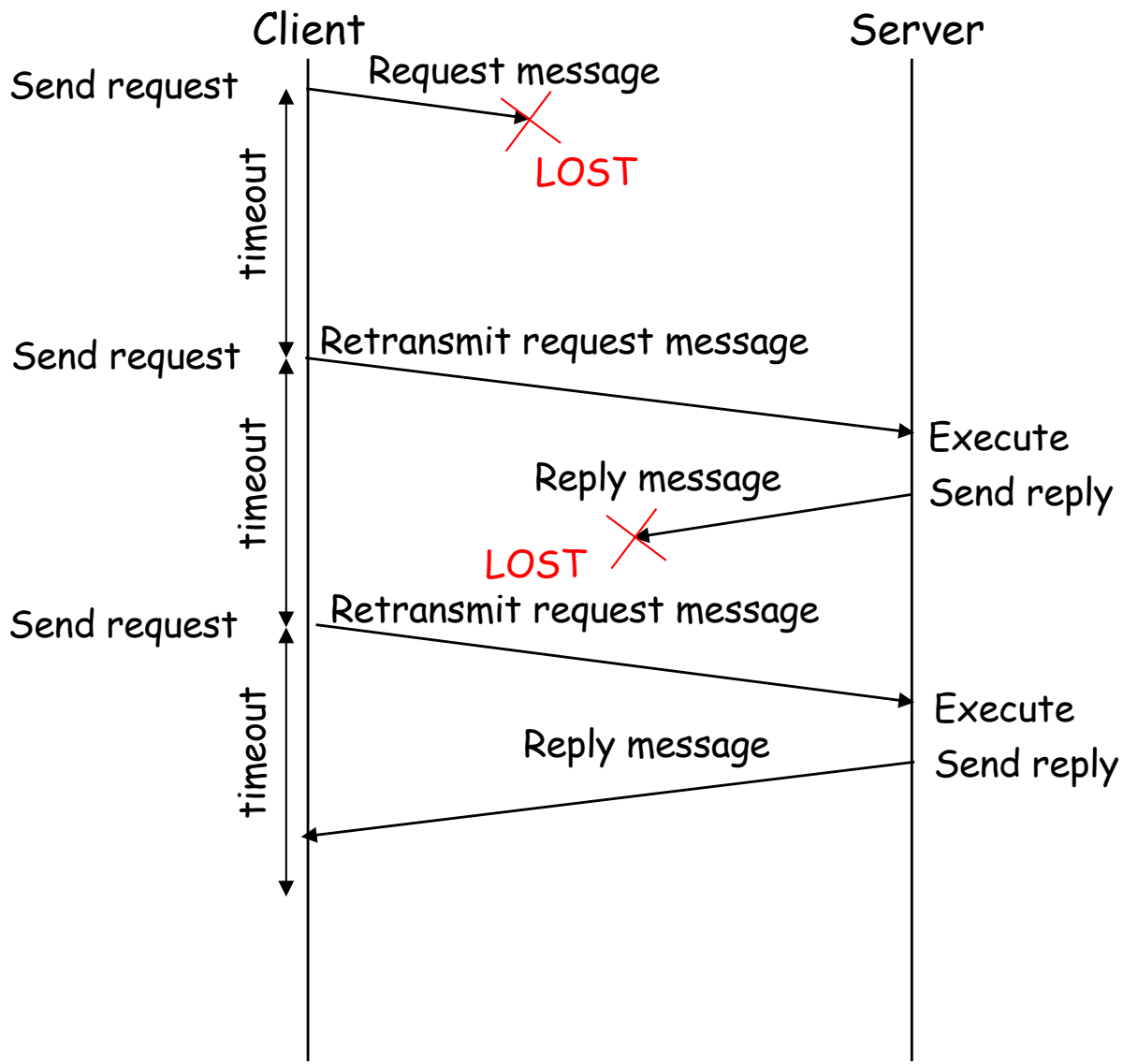
- No guarantee that service has been executed or not by server
- No action to ensure reliable communication: no mechanism (among RR1, DF, RR2) is used
- E.g., best-effort in UDP
- Use cases: logging, sensor data (if occasional loss is tolerable)



At-least-once semantics

- Service, if executed, has been executed at least once
  - Perhaps more than once due to request duplicates caused by retransmissions
- Client uses RR1, server uses neither DF nor RR2
- Upon receiving the reply, client cannot know how many times its request has been processed by server
  - If server executes the service once or multiple times because the reply was lost or delayed
  - If server executed the service but crashed before replying and then recovered, and client retries the request after timeout; server processes the request again
- It works for idempotent services
  - **Idempotent** service: a service where applying it multiple times has the same effect as applying it once to the same input
    - E.g., `get(key)` or `put(key,value)` in KV data store

At-least-once semantics



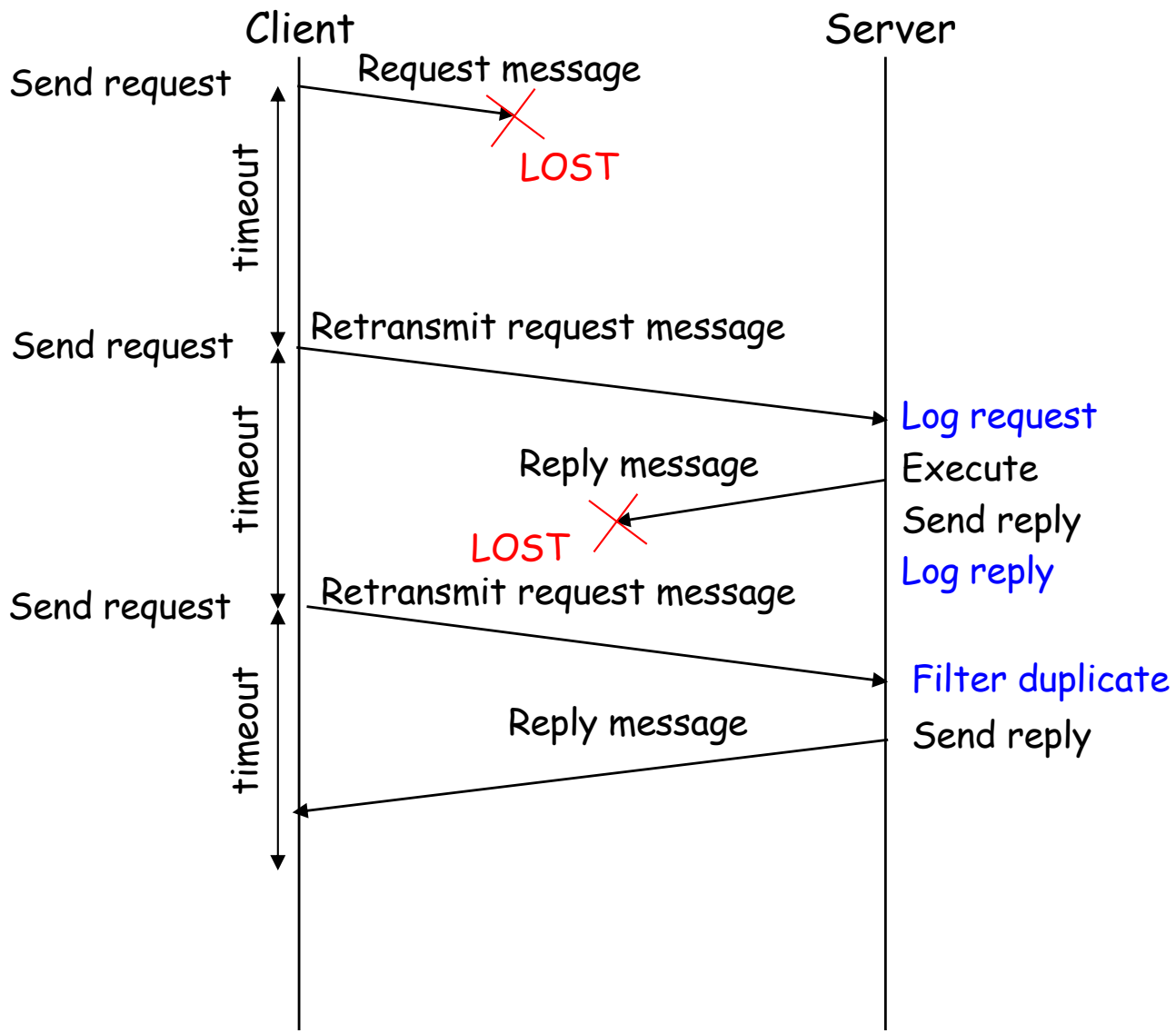
No AI/ML Processing

At-most-once semantics

- Service, if executed, has been executed at most once
  - *If* client receives the reply, it knows that the service was executed *once*
  - In case of failure, client cannot determine whether the request never reached the server, or server executed the service but the reply was lost (at-most-once: at most once, but possibly not all)
- Mechanisms used: RR1, DF, and RR2
  - Client retransmits request when timeout expires
  - Server maintains some state to identify duplicate requests and avoid processing the same request more than once
- It works also for non-idempotent services
  - E.g., `increment(counter)` or `withdraw(amount)`
- No strict coordination between client and server
  - Client does not know if the service was executed (in case of no reply), server does not know if client received the reply

No AI/ML Processing

At-most-once semantics



No AI/ML Processing

At-most-once: implementation

- Server must:
  - Detect duplicate requests
  - Manage duplicate requests returning saved reply instead of re-running service `handler()`
  - Discard saved replies when it is safe

- How to detect duplicates
  - Client includes a unique ID (`xid`) with each request and uses same `xid` when retransmitting the request due to timeout
  - Server uses `xid` to detect duplicates
  - If server knows the request with that `xid`
    - If reply is saved, resend it
    - If original request is still executing, don't re-run service

Server:

```
if seen[xid]
 reply = old[xid]
else
  reply = handler()
  old[xid] = reply
  seen[xid] = true
```


No AI/ML Processing

At-most-once: implementation

- How to ensure unique `xid`?
  - Globally unique per client (and per server if needed)
  - Common pattern: `xid = client_id + sequence_number`
  - Sequence number: increments with every new request and wraps around only after large gaps
- How to handle duplicate requests during execution?
  - Server maintains also a status flag: `xid → {status, reply}`
  - Status is `pending` if service still executing
  - Status is `completed` if reply cached
  - If duplicate arrives while status is pending, server ignores it or queues it to wait for the original to finish
- Server must eventually discard cached replies: how?
  - Use sliding windows to track recent `xids`
  - Retain reply for a maximum lifetime

Idempotent Receiver pattern
https://martinfowler.com/articles/patterns-of-distributed-systems/idempotent-receiver.html

No AI/ML Processing

Exactly-once semantics

- Strongest and hardest guarantee to implement, especially in large-scale DS
- Requires full agreement on interaction between client and server
  - Service is run only once or not run at all: all-or-nothing semantics
    - If everything goes well: service runs only once, duplicates are found
    - If something goes wrong: client or server knows if service has run (once - all) or not (never - nothing)
- Semantics with concordant knowledge of each other's state and without constraint on maximum duration of interaction protocol between client and server
  - No constraint on maximum duration: barely practical in real systems!

No AI/ML Processing

Exactly-once semantics: more mechanisms

- Basic mechanisms (RR1, DF, RR2) are not enough
  - Do not protect against server failures
- Additional mechanisms for fault tolerance
  - Transparent server replication
    - Multiple servers replicate the same service
    - E.g., leader-follower or primary-backup
  - Write-ahead logging (WAL)
    - Before executing a service, the server logs the request to persistent storage
  - Recovery mechanisms
    - After failure, the server resumes processing from a safe state
    - *State checkpoint*: saves a local snapshot of server's state to persistent storage
      - Combined with WAL to reduce recovery time (only replay log entries after last checkpoint)
    - *Snapshot*: captures a consistent global state of replicated servers: more complex, requires coordination

No AI/ML Processing

Write-ahead logging pattern

- aka Commit log
- Goal
  - Ensure durability by persisting every state change so server can survive crashes and restarts.
  - Log is stored
- How
  - Each state change is stored as log entry in an append-only commit log file written on persistent storage (e.g., disk or SSD)
  - Entries are appended sequentially, ensuring fast write performance and crash-consistent ordering
  - On server restart, log is read and all entries are replayed in order to rebuild the in-memory state
  - No state is lost, as long as the log was flushed to disk before crash

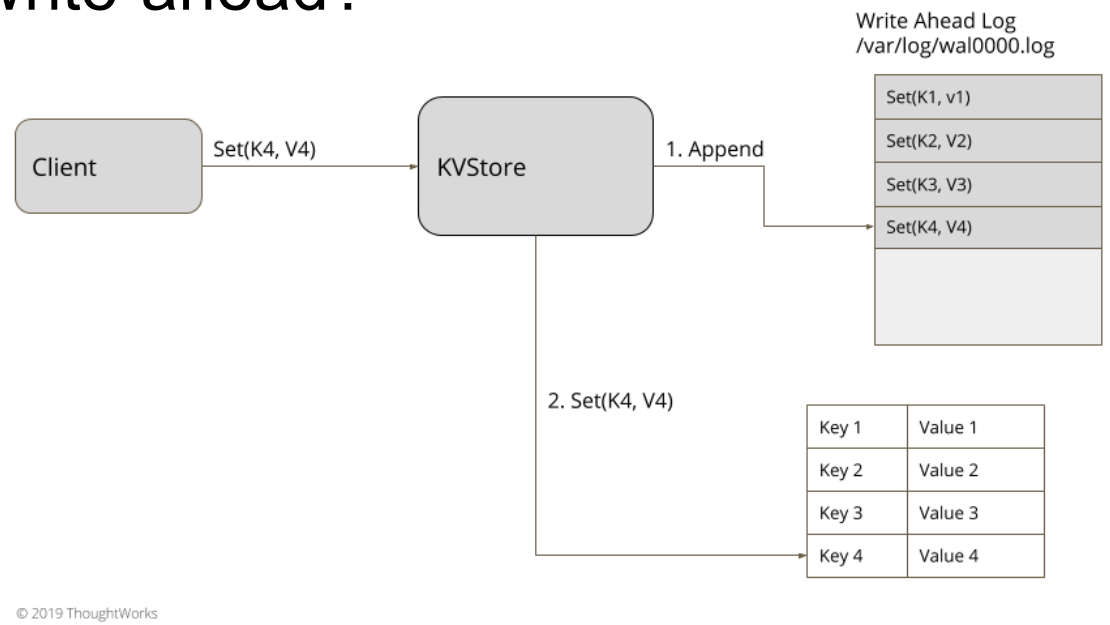
WAL pattern

https://martinfowler.com/articles/patterns-of-distributed-systems/wal.html

No AI/ML Processing

Write-ahead logging pattern

- Why write-ahead?




```
public void put(String key, String value) {
    appendLog(key, value);
    kv.put(key, value);
}

private Long appendLog(String key, String value) {
    return wal.writeEntry(new SetValueCommand(key, value).serialize());
}
```


No AI/ML Processing

Summing up failure semantics

- In practice, at-least once and at-most once semantics are both feasible and widely used in DS
- Exactly-once is ideal, but often too complex, slow, or expensive to achieve in practice

- We often choose the lesser of two evils, i.e., at-least-once semantics
  - Easier to scale
    - Stateless retries
    - No need for coordination between sender and receiver
  - Works well with idempotent services
  - Can be combined with application-level deduplication if needed

Distributed systems are all about trade-offs!

No AI/ML Processing

Distributed application programming

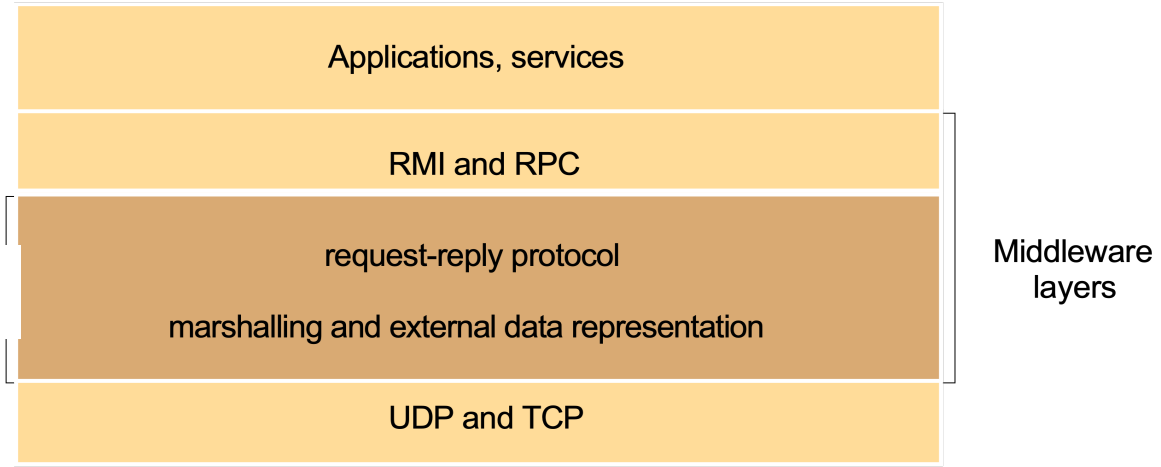
- Explicit network programming
  - Operating system constructs based on socket API and explicit management of message exchange
  - Foundational for network applications (e.g. web browser, web server, chat app)
  - ✗ Distribution is not transparent and requires developer effort

- How to increase the abstraction level?
- Communication middleware between OS and applications
  - Provides a higher-level API for communication
  - Hides complexity of underlying hw and sw layers
  - Frees developers from automatable tasks
  - Improves software quality by providing reusable, correct, and efficient solutions to common DS challenges

No AI/ML Processing

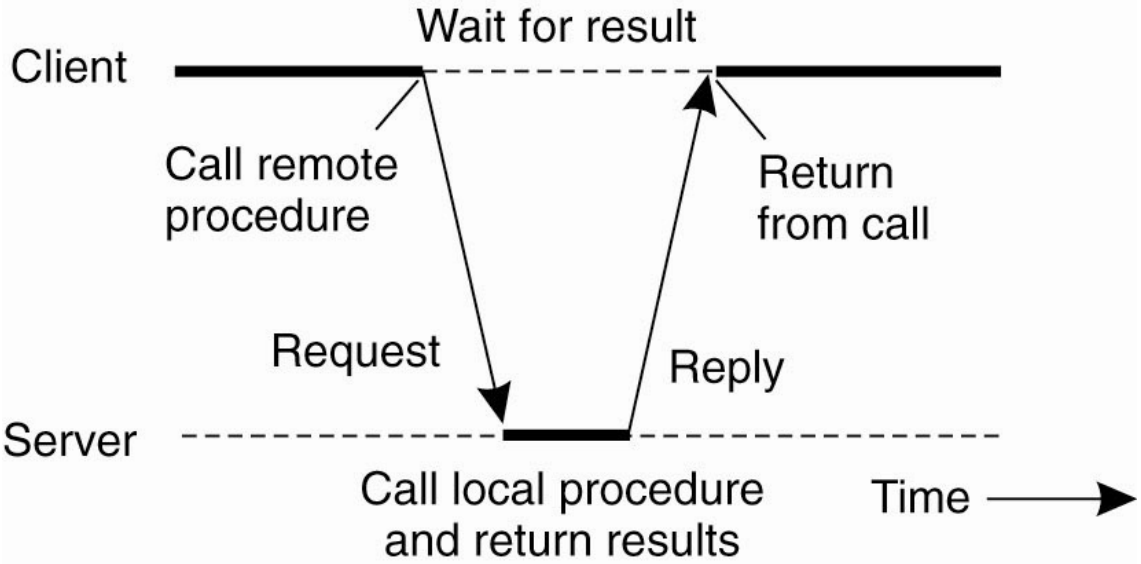
Distributed application programming

- Implicit network programming
  - Language-level constructs simplify distributed programming by hiding network details from developers
  - Developers work with **remote procedure calls (RPC)** or **remote method invocations (RMI)** as if they were local calls
  - In reality, the caller (client) and callee (server) are located on separate machines



Remote Procedure Call (RPC)

- Core idea (Birrel and Nelson, 1984): use client/server model to call procedures executed on remote machines
  - Client on machine A calls a procedure on machine B
  - Calling process on A is suspended
  - Called procedure is execute on B
  - Input/output parameters are transported as messages
  - No message passing is visible to developer



Why RPC

- Widely used in many distributed systems, including cloud computing, microservice architectures
- Many languages and frameworks have they own implementation of RPC:
  - C (Sun RCP)
  - Java (Java RMI)
  - Python (RPyC)
  - Go
  - gRPC
  - Distributed Ruby https://github.com/ruby/drb
  - Ice https://zeroc.com/ice
  - Microsoft .NET https://dotnet.microsoft.com/
  - JSON-RPC https://www.jsonrpc.org

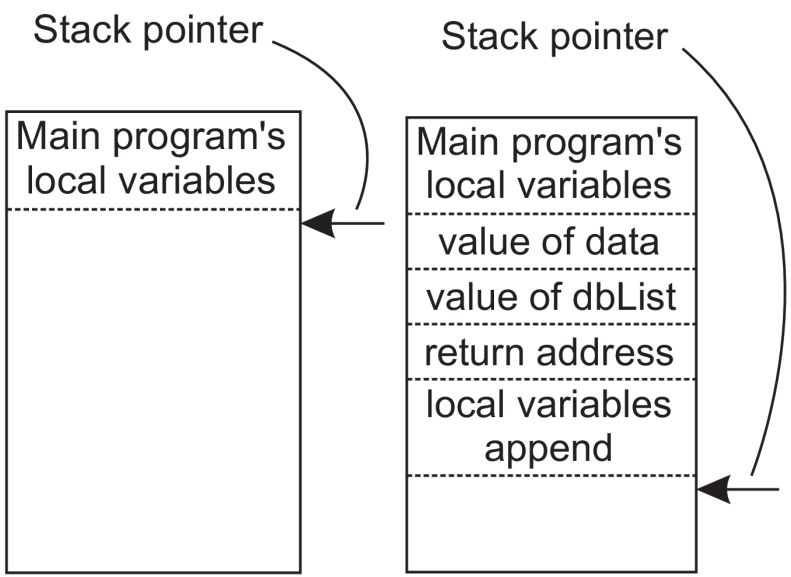
Case studies

Local procedure call

- Example of local procedure call:

```
newlist = append(data, dbList)
```

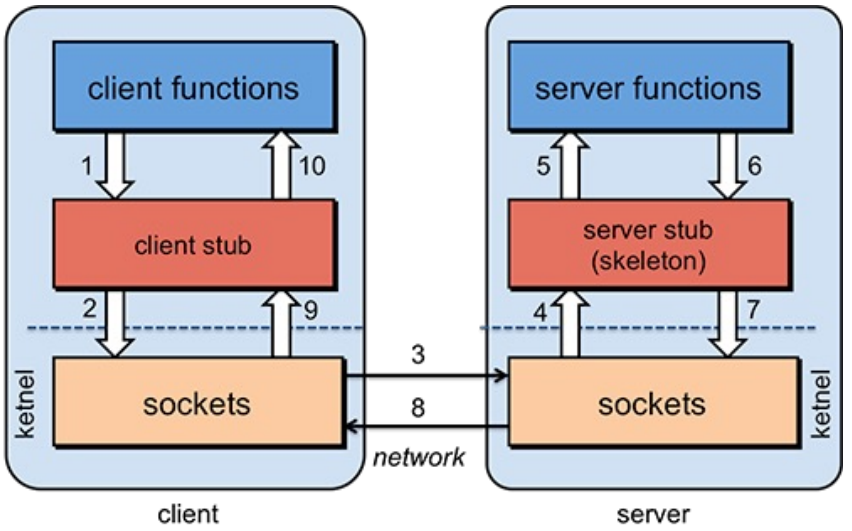
- Caller pushes to stack input parameters (`data`, `dbList`) and return address
- When callee returns, control is back to caller



How can RPC appear as close as possible to a local procedure call?

RPC: architecture

- Solution: proxies (aka **stubs**) to make RPC calls feel like local calls
- Client side: **client stub** exposes service's interface
  - Client calls client stub, which manages all communication details
- Server side: **server stub** receives request and dispatches it to the local procedure on the server
- Goal: distribution transparency
  - Automatic stub generation
  - Focus on application logic



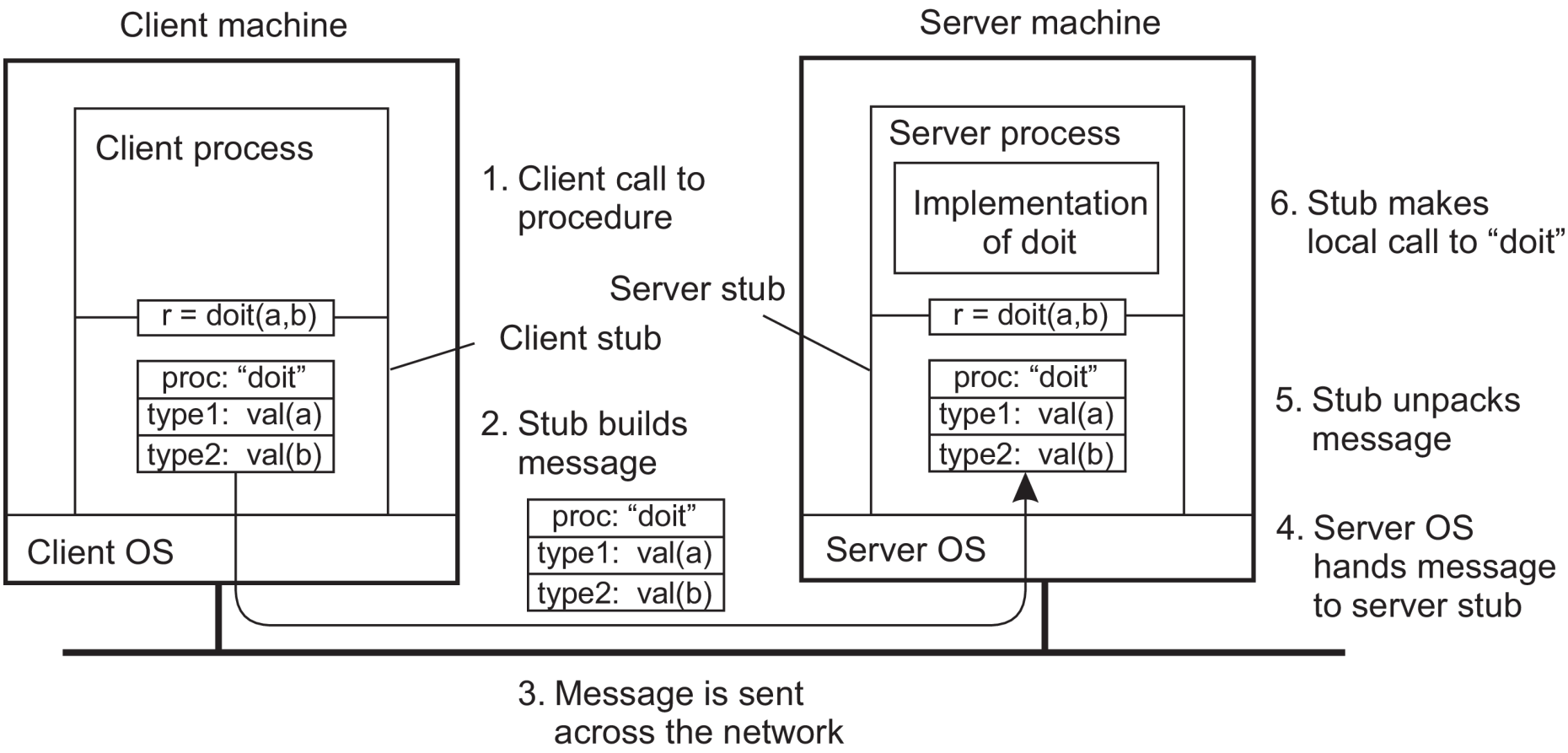
RPC: basic steps

1. On client side, client calls a local procedure, the client stub
2. Client stub marshals parameters (converts them to a common format), packages them into a request message, and call local OS
3. Client OS sends request message to remote OS
4. On server side, remote OS delivers request message to server stub
5. Server stub unmarshals parameters and calls server as it were a local procedure
6. Server executes local call and returns result to server stub
7. Server stubs marshals return value(s) and calls OS
8. Server OS sends reply message to client OS
9. On client side, Client OS delivers reply message to client stub
10. Client stub unmarshals return value(s) and returns result to client

No AI/ML Processing

RPC: example

- RPC `doit(a,b)`



RPC: key requirements

- To make an RPC appear as local procedure call, the middleware must:
- Exchange messages between client and server
  - Identify request/reply messages and remote procedure
  - Pass parameters and return values
- Manage data heterogeneity
  - Parameters, return value(s)
  - Marshaling vs. serialization:
    - ***Marshaling***: bundle parameters into a transferable form that can be unmarshaled (reconstructed) by another process
    - ***Serialization***: convert objects into a sequence of bytes for transmission; serialization is a subset of marshaling
- Handle failures
  - During communication (e.g., message loss, delay)
  - User errors (e.g., invalid input)

No AI/ML Processing

RPC: implementation challenges

1. Manage heterogeneity in data representation
   - In addition, client and server must agree on transport protocol for message passing: TPC, UDP, both?
2. Perform parameter passing by reference
   - Client and server run on different machines with separate address space
3. Define failure semantics
   - Local procedure call: exactly-once
   - Remote procedure call: typically at-least-once or at-most-once
4. Bind client to server
   - Locate server endpoint before performing the call

No AI/ML Processing

Data representation heterogeneity

- Client and server may use different data representations, e.g.,
  - Byte ordering: little-endian vs big-endian
  - Size of types, e.g., bits for `int`,
  - Character encoding
  - Memory layout: struct alignment and padding
- Copying raw memory onto the network is not an option
- RPC must define must define an explicit wire encoding

No AI/ML Processing

Data representation heterogeneity

- Alternatives for data conversion (*general*, not only RPC):

1. Encoding specified in the message
   - The receiver converts
2. Sender converts into receiver's encoding
   - The sender must know it
   - $O(N^2)$ converters (N = number of systems with ≠ formats)
3. Common encoding agreed by both parties
   - Sender converts local → common
   - Receiver converts common → local
   - O(N) converters
4. Intermediary converts between different encodings

- Most RPC systems use common encoding

No AI/ML Processing

Data representation heterogeneity: patterns

- Proxy (aka stub)
  - Local representative of a remote service, with the same interface
  - Gives access transparency (marshaling)
  - Generated automatically
- Broker
  - Separates communication details from the application logic
  - Handles message routing and communication hiding
  - E.g. CORBA ORB
- RPC systems are built on the proxy pattern

No AI/ML Processing

Parameter passing methods

- Call by value
  - Parameter value is copied into local storage (usually stack)
  - Callee acts on copy; changes do not affect caller data
- Call by reference
  - Reference (pointer) to parameter is passed (copied into stack)
  - Callee acts directly on caller data
- Call by copy-restore
  - Special case of call by reference
  - Data is copied into callee stack; upon procedure return, updated data is copied back (restored) to caller
  - Supported in a few programming languages (e.g., Ada, Fortran)

No AI/ML Processing

RPC parameter passing

- A reference is a memory address
  - Valid only in its context (local machine)
  - We need a pointer-less representation

- Solution: simulate call by reference in RPC by using call by copy-restore
  - Client stub copies the pointed data into the request message
  - Server stub works on copy in its own address space
  - If modified, the updated copy is restored by client stub overwriting original data
  - Size of data to be copied must be known
  - What happens if data contains pointers?

No AI/ML Processing

RPC failure semantics

- Exactly-once semantics
  - Guarantees remote procedure is executed exactly once
  - Costly and complex to implement
- At-least-once semantics
  - If client receives a reply, the remote call has been executed at least once
  - Possible duplicate executions due to retransmissions
  - Suitable for idempotent operations
- At-most-once semantics
  - If client receives a reply, the remote call has been executed at most once
  - No duplicate executions, but call may not have been executed at all in failure scenarios
  - Requires duplicate request detection and state management on server

No AI/ML Processing

Server binding

- Binding: locate the server endpoint, including address and specific process (e.g., port)
- Types of binding: static or dynamic
- Static binding
  - Server endpoint known at design time (hard-coded)
  - ✓ Simple, no runtime overhead
  - ✗ Lacks transparency and flexibility
    - Changes require redeployment
- Dynamic binding
  - Server endpoint determined at run-time
  - ✓ Gains transparency and flexibility
    - Request redirection in case of server replication
  - ✗ Increased overhead: how to limit it?

No AI/ML Processing

Dynamic server binding

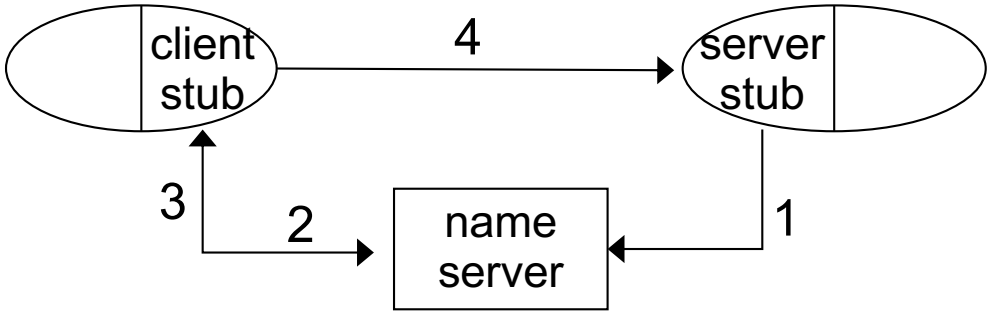
- Two phases in client/server relationship

1. Naming: static phase *before* execution
   - Client specifies whom it wants to connect to, using a unique service name
   - Unique names correspond to operations or abstract interfaces
   - Binding is made to the specific service interface, not a concrete address
2. Addressing: dynamic phase *during* execution
   - Actual server endpoint is resolved when client invokes the service
   - RPC middleware may dynamically select among multiple replica servers

No AI/ML Processing

Dynamic server binding: addressing

- Addressing can be explicit or implicit
  - Explicit: client sends request using broadcast or multicast, waiting for the first reply
  - Implicit addressing: uses a name server (aka binder, directory service, registry service) that registers services and manages a binding table
    - Supports service registration, lookup, update, and deletion

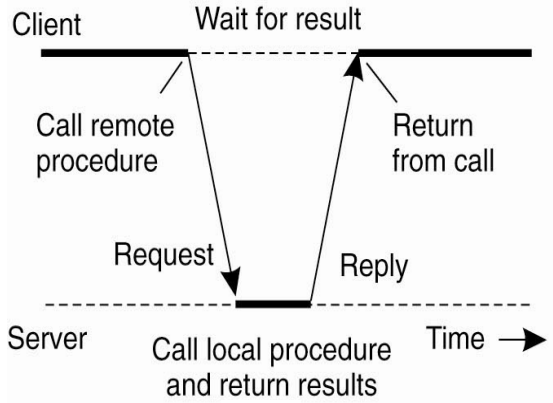


- Addressing frequency
  - Each RPC call requires addressing
  - To reduce overhead, binding results are cached and re-used

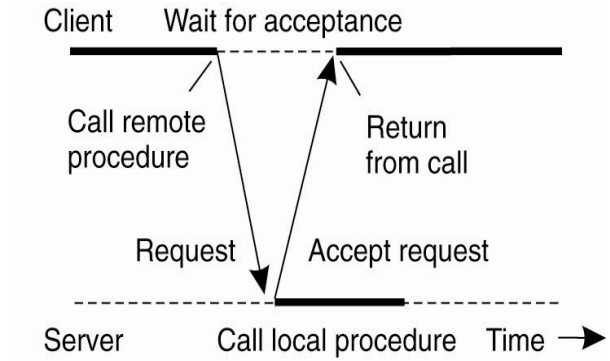
No AI/ML Processing

Other issues: Synchronous vs asynchronous RPC

- Synchronous RPC
  - RPC call blocks client that waits for server reply
- Some RPC middleware supports asynchronous RPC
  - Client does not wait for server reply: continues execution
  - Server can send an ack upon receiving the request is received and execute procedure later



Synchronous RPC



Asynchronous RPC

Other issues: transparency

- Is RPC truly transparent? Can RPCs be treated like local calls?
  - Performance, failures, concurrent requests, replication, migration, …

- Performance
  - RPC is much slower than local calls: why?
  - Local procedure call: ~10 cycles (~3 ns)
  - RPC on LAN takes 0.1-1 ms: ~100000 times slower
    - Major overheads: context switching, data copying, communication
  - RPC over WAN: can be millions of times slower

Other issues: transparency

- Common failures in RPC
  - Client cannot locate server
  - Lost request messages during transmission
  - Server crashes during execution
  - Lost reply messages
  - Client crashes before or after request

Other issues: security

- Authentication
  - Does client send messages to genuine server or to impostor?
  - Is the client who it claims to be?
- Authorization and access control
  - Does server accept messages only from legitimate clients?
  - Can server identify users at client side?
- Messages confidentiality and integrity
  - Messages may be visible and sniffed on network
  - Can messages be encrypted?
  - Are in-transit messages protected against accidental corruption or truncation?
- Replay attacks
  - Can an attacker capture an RPC message and retransmit it later?

Programming with RPC

- Language-level support
  - Some languages (e.g., C, C++) lack built-in RPC support
  - Other languages (Java, Python, Go, Erlang, Haskell) provide direct support
  - Challenge: heterogeneous languages (e.g., Java client and Python server)

- Common solution
  - Use an **Interface Definition Language (IDL)** to describe remote procedures
  - An IDL compiler automatically generates client and server stubs

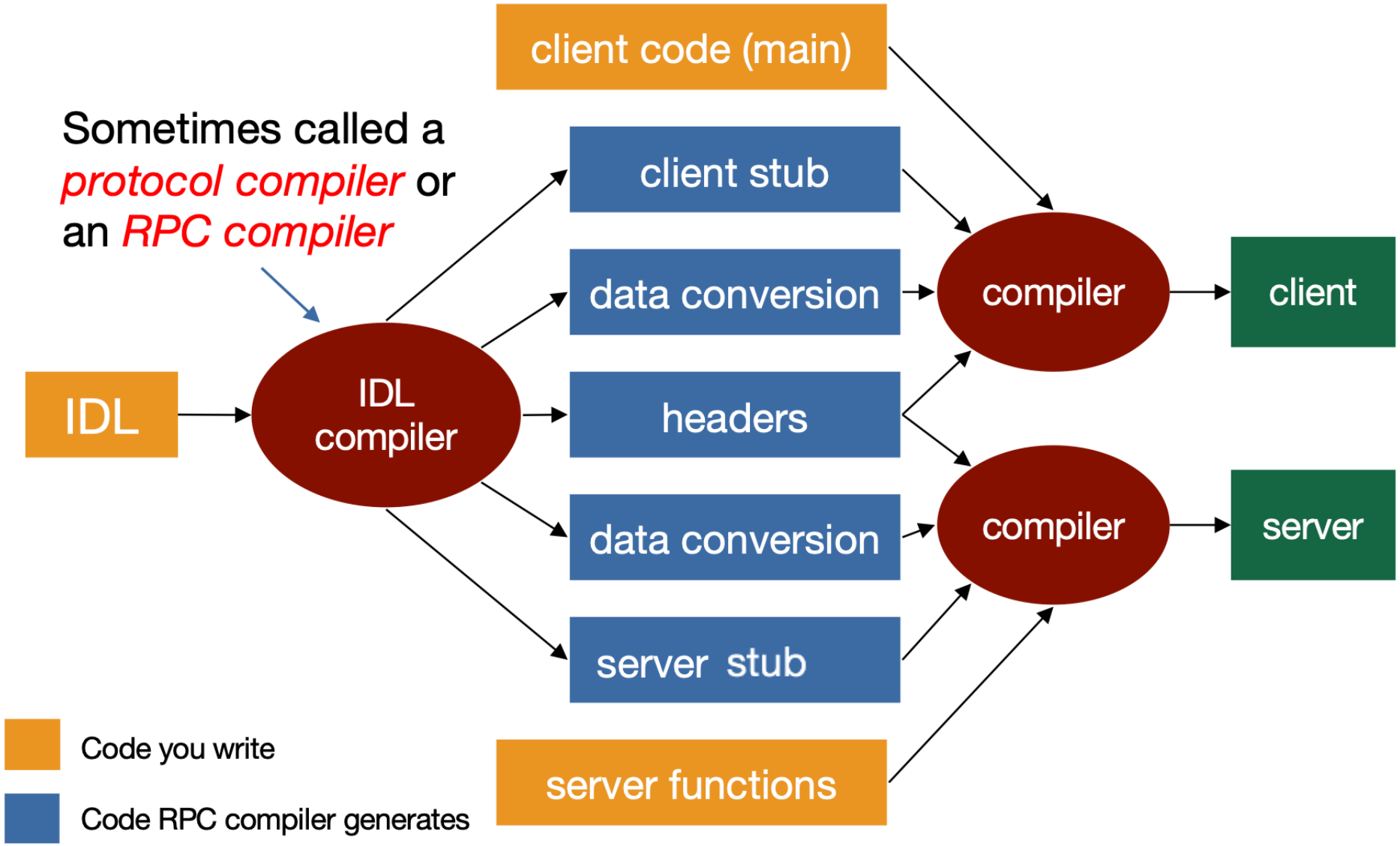
Interface Definition Language (IDL)

- IDL allows developers to specify remote procedure interfaces (including *names, parameters, return values*) in a machine-independent way
- IDL compiler uses these interfaces to generate client and server stubs, containing:
  - Code to marshal
  - Code to unmarshal
  - Network transport code
- IDL definitions look similar to function prototypes
- Example: Thrift IDL

```
service KeyValueStore {
  i32 put(1: string key, 2: string value),
  string get(1: string key),
}
```


No AI/ML Processing

IDL and its compiler



No AI/ML Processing

RPC middleware: case studies

- Sun RPC
- Java RMI
- Python RPyC
- Go
- gRPC

No AI/ML Processing

RPC implementation: Sun RPC

- First-generation RPC
- Created by Sun Microsystems (now Oracle)
  - RFC 1831 (1995), RFC 5531 (2009)
  - Still used mainly for NFS (Network File System)

- Example: remote procedure to calculate integer square

1 Define service (`square.x`)

RPC program, version, and procedures

2 `rpcgen`

Automatic generation of header, client stub, server stub, and XDR routines for marshaling

3 Developer

Write `client.c` and `server.c`

No AI/ML Processing

Sun RPC: define RPC program using XDR

- Service interfaces are defined using XDR (eXternal Data Representation) as IDL
  - IDL compiler: `rpcgen`
  - Provides built-in conversion functions for:
    - Primitive types, e.g., `xdr_int()`
    - Structured types, e.g., `xdr_string()`
  - Binary format with *implicit typing*
    - Only values sent, no type or parameter metadata
- Two parts in `.x` file
  1. Definition: procedures with their name and parameters' data types
  2. XDR definitions: custom data definitions (if not built-in)

No AI/ML Processing

Sun RPC: define remote procedure

square.x

```
struct square_in {        /* input (argument) */
  long arg1;
};
struct square_out {       /* output (result) */
  long res1;
};
program SQUARE_PROG {
  version SQUARE_VERS {
  square_out  SQUAREPROC(square_in) = 1; /* procedure number = 1 */
  } = 1;                  /* version number */
} = 0x31230000;           /* program number */
```

- Define remote procedure SQUAREPROC
  - Only one input parameter and one output parameter
  - Identifiers are written in uppercase
  - Unique procedure number within RPC program (e.g., 1)

No AI/ML Processing

Sun RPC: implement RPC program

- Developer task
- Client program: implements `main()` (`square_client.c`)
  - Who locate the remote procedure and bind to its server?
- Server program: implements remote procedures provided by RPC server (`square_server.c`)
  - No server-side `main()`: who calls the remote procedure on server side?

No AI/ML Processing

Sun RPC: remote procedure

- Remote procedure: *similar* to local one

server.c

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "square.h"    /* generated by rpcgen */
square_out *squareproc_1_svc(square_in *inp, struct svc_req
   *rqstp) {
  static square_out out;

  out.res1 = inp->arg1 * inp->arg1;
  return(&out);
}
```

- Input and output parameters use pointers
- Output must point to a static variable so it persists after procedure returns
- Procedure naming convention: original RPC procedure name becomes `<procedure_name>_<version>_svc`, e.g., `squareproc_1_svc`, all lowercase)

No AI/ML Processing

Sun RPC: client

- Run RPC client:

  `./square_client <server-hostname> <integer>`

client.c

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "square.h"   /* generated by rpcgen */
int main(int argc, char **argv) {
  CLIENT *clnt;
  char *host;
  square_in in;
  square_out *result;
   if (argc != 3) {
    printf("usage: client <hostname> <integer-value>\n");
    exit(1);
  }
  host = argv[1];
  clnt = clnt_create(host, SQUARE_PROG, SQUARE_VERS, "tcp");
```

```
CLIENT *clnt_create(char *host, unsigned long prog,
unsigned number vers, char *proto)
```


No AI/ML Processing

Sun RPC: client

```
  if (clnt == NULL) {
    clnt_pcreateerror(host);
    exit(1);
  }
  in.arg1 = atol(argv[2]);
  if ((result = squareproc_1(&in, clnt)) == NULL) {
    printf("%s", clnt_sperror(clnt, argv[1]));
    exit(1);
  }
  printf("result: %ld\n", result->res1);
  exit(0);
}
```


No AI/ML Processing

Sun RPC: client

- Create client transport manager: `clnt_create()`
  - Handles communication (TPC or UDP)
  - Default request timeout and retransmission
  - Requires:
    - Server hostname
    - Remote procedure info: program (`SQUARE_PROG`), version (`1`), procedure (`squareproc`)
- Call the remote procedure:
  - Procedure name: lowercase + _ + version (`squareproc_1`)
  - Input: actual parameter + client transport manager
  - Output: pointer to result
  - RPC failure: function returns `NULL`
- Handle errors:
  - `clnt_pcreateerror()` and `clnt_perror()`

No AI/ML Processing

SUN RPC: development process

1 Define service (square.x)

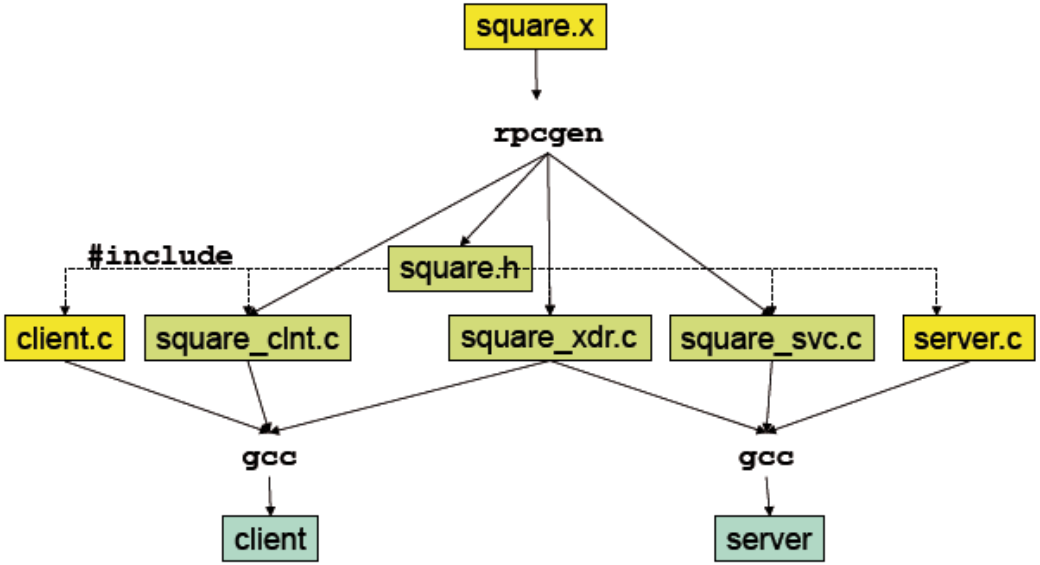
RPC, Program, version, and procedures; only one input/output parameter

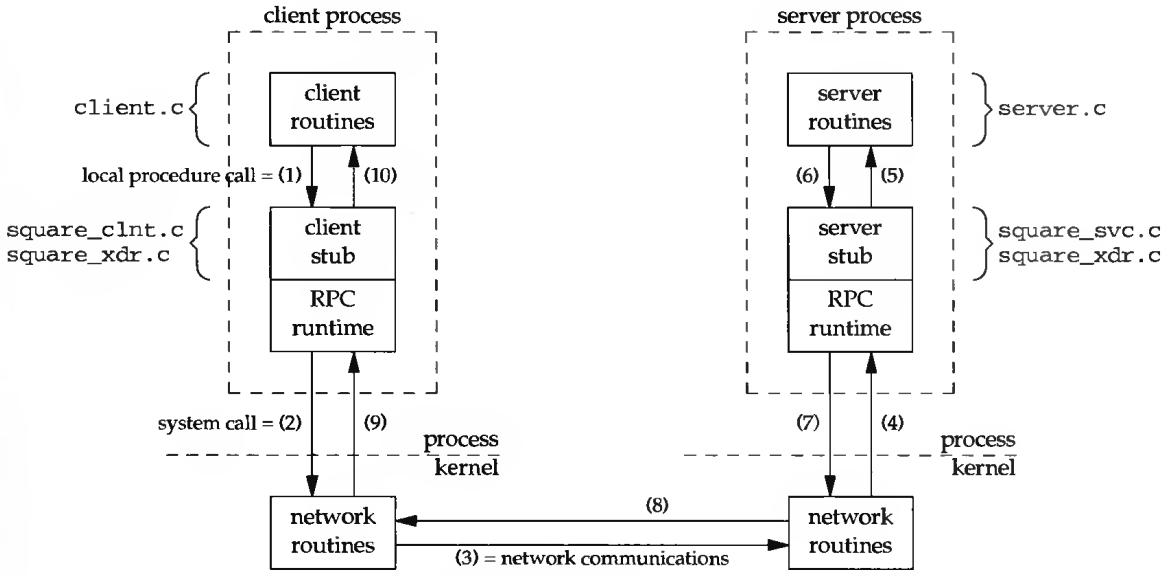
2 rpcgen

Automatic generation of header, client stub, server stub, and XDR routines for marshaling

3 Developer

Write client.c and server.c; server main() is in stub





No AI/ML Processing

Sun RPC: server binding

- Problem: the client stub needs to know the port of a server stub that does not listen on a fixed one
- Solution: one well-known service, called port mapper (`rpcbind`), listens on port 111 (TCP and UDP) on every host
- The port mapper keeps a table (*port map*) that maps <program, version, protocol> to the port where that RPC service is actually listening

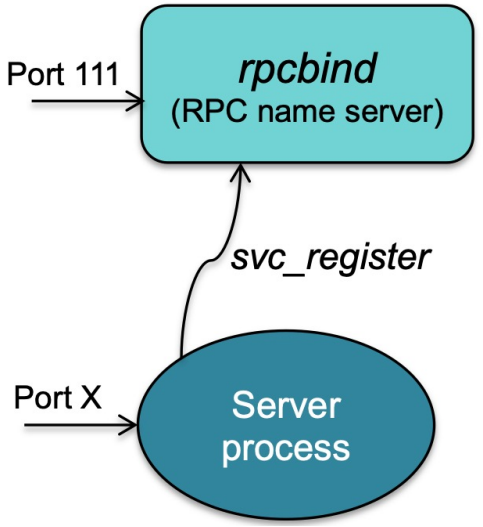
```
>$ rpcinfo -p
 program          vers     proto    port
   100000         4        tcp      111       rpcbind
   100000         4        udp      111       rpcbind
   824377344      1        udp      59528
   824377344      1        tcp      49311
```

824377344 = 0x31230000

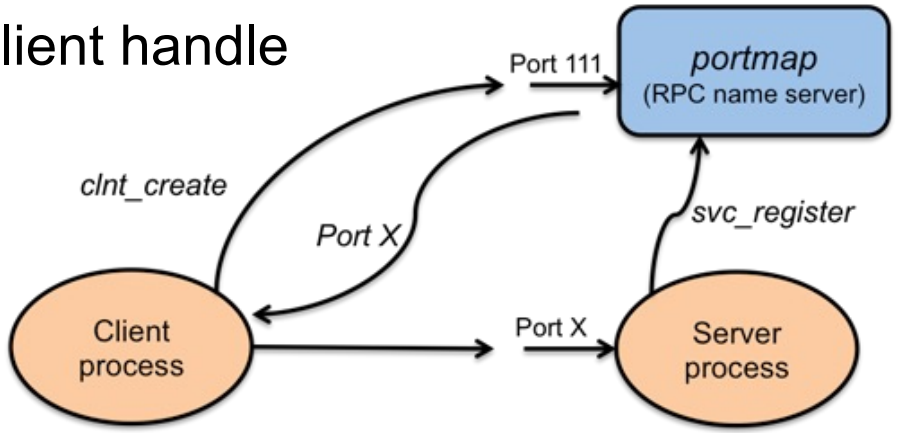
No AI/ML Processing

Sun RPC: server binding

- The server stub registers the RPC service with the local port mapper
  - Example: program 8243777344, version 1, protocol tpc, port 49311



- The client stub contacts the port mapper on port 111 of the server host to lookup the port number, then sends the request to server stub
  - Early binding: done once per client handle



No AI/ML Processing

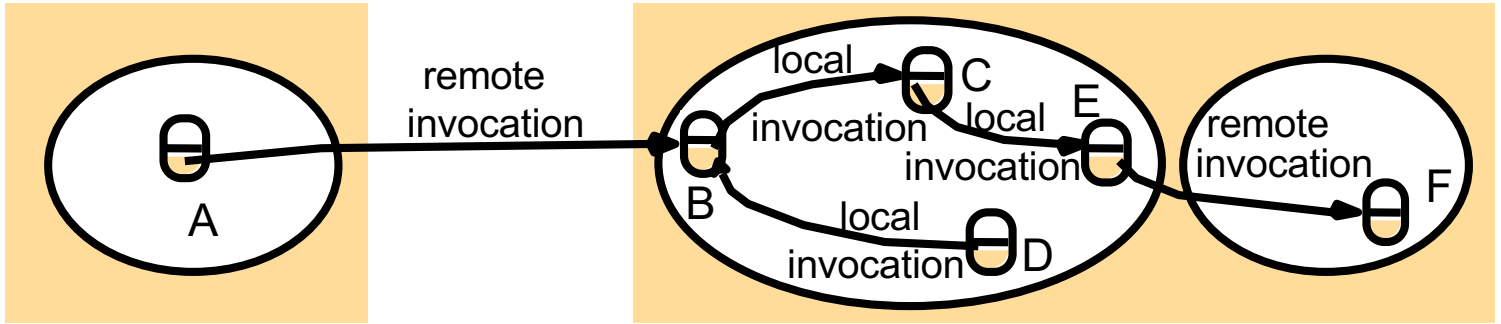
Sun RPC: summing up

- Remote procedures
  - Multiple procedures with the same RPC program, multiple versions of the RPC program
  - One input parameter, one return value
  - Parameter passing: by copy-restore
- Transport independent: TCP or UDP at runtime
- Synchronous, blocking calls
- Concurrency: no concurrency by default
- Stub code: visible, auto-generated by `rpcgen`
- Failure semantics: at-least-once
  - Retransmission on timeout, no server-side deduplication
- Server binding: port mapper
- Security: authentication added by Secure RPC

No AI/ML Processing

Java RMI

- RMI: Remote Method Invocation
- Second-generation RPC
  - Objects in one JVM can invoke methods on objects in another JVM
- Logical separation enables physical separation
  - Definition (interface) and implementation (class)
- Remote interface: methods for remote invocation
- Remote object: instance of class implementing remote interface
  - The object state is not distributed, but lives in the server



No AI/ML Processing

RMI: basics

- Remote method invocation: invokes methods of a remote interface on a remote object
- Access transparency: syntax of a remote call = syntax of a local call, via proxy pattern
  - Client side: a **stub** represents the remote object locally
  - Server side: a dispatcher invokes the actual object (a skeleton in JDK 1.1; the RMI runtime since JDK 1.2)

- Example: remote echo

No AI/ML Processing

RMI example: echo remote interface

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface EchoInterface extends Remote {
 String getEcho(String echo) throws RemoteException; }
```

- Remote interface: extends **`Remote`**
  - Interface invocable from other JVMs
- Remote method
  - Throws **`RemoteException`**
    - Exposes network, protocol and server failures
    - Remote invocation is not fully transparent: failures are visible

No AI/ML Processing

RMI: Common format and serialization

- Common format: built-in Java serialization
  - Serialization: converts object state to byte stream (`writeObject`)
  - Deserialization: rebuilds object state from byte stream (`readObject`)
- Hides hardware heterogeneity (byte order, type sizes), but no language heterogeneity
  - Both sides must run Java
- In RMI, marshaling = serialization + codebase annotation
  - Codebase: the URL from which a JVM can download the bytecode of a class it does not have locally

No AI/ML Processing

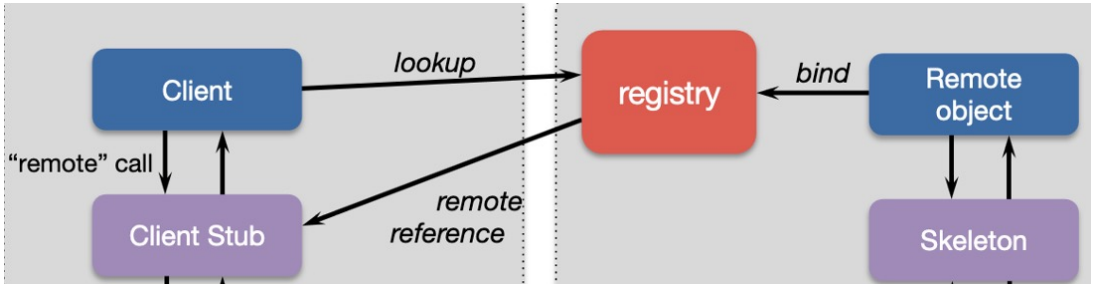
RMI: Parameter passing

- Parameters and return values are passed depending on their type
- By value: primitives and `Serializable` objects
  - The receiver gets a copy; changes are not propagated back
- By reference: remote objects
  - Only the stub is passed, the object state stays in the JVM that hosts it
  - A remote reference is a data structure {IP address, port, object ID, remote interface}, not a pointer
  - Why pass a remote object? When the receiver must act on the original, not on a copy
    - Typical case: callback
- Exceptions are serialized and rethrown on client

No AI/ML Processing

RMI: call flow

- The server exports the remote object and registers it in the **RMI registry** (name → stub)
- The client looks it up by name and receives a stub
- The client calls the stub, which serializes method ID and parameters and sends them
- The server runtime (or skeleton) deserializes, invokes the method and sends back the serialized result (or exception)
- The stub deserializes the result and returns it to the caller



No AI/ML Processing

RMI example: echo server

- Class implements remote interface
  - Extend **`UnicastRemoteObject`**
  - Call class constructor **`UnicastRemoteObject`** which enables server to receive requests
  - Implement remote method, which throws remote exception

```
public class EchoRMIServer
extends UnicastRemoteObject
implements EchoInterface {

// Constructor
public EchoRMIServer()
 throws RemoteException {
   super();
}

// Implement remote method
public String getEcho(String s)
  throws RemoteException {
    return s;
  }
```


No AI/ML Processing

RMI example: echo server

- Create instance of remote object
- Start RMI registry
- Register remote object with registry by name
- RMI registry and server run on same host

```
public static void main(String[] args) {
  try {
    // Create an instance of echo server
    EchoRMIServer serverRMI = new EchoRMIServer();
    // Bind remote object with RMI registry
    Registry reg = LocateRegistry.createRegistry(1099);
    reg.bind("EchoService", serverRMI);
  }
  catch (Exception e) {
    System.err.println("EchoRMIServer exception: ");
    e.printStackTrace();
  }
 }
}
```


No AI/ML Processing

RMI example: echo client

- Lookup remote object in server's RMI registry by name
- RMI registry returns a remote reference (stub)
- Client invokes remote method
  - Synchronous, blocking

```
public class EchoRMIClient {
  // Start RMI client
  public static void main(String[] args) {
    try {
      // Connect to remote RMI service
      EchoInterface service = (EchoInterface)
         Naming.lookup("EchoService");
      // Interact with user
      bufferedReader stdIn =
       new BufferedReader(
         new InputStreamReader(System.in));

      String message, echo;
      System.out.print("Message? ");
      message = stdIn.readLine();
      // Call remote method
      echo = service.getEcho(message);
      System.out.println("Echo: "+echo+"\n");
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  } // end main
}
```


No AI/ML Processing

RMI: summing up

- Access transparency, but failures are visible
- Java serialization
- No IDL compiler: stub is auto-generated
- Synchronous, blocking calls
- Concurrency: server code must be thread-safe (`synchronized`)
- Failure semantics: at-most-once
- Server binding: RMI registry
- Exception handling: `RemoteException`
- Security: requires TLS-based socket factories
- Distributed garbage collection
  - Remote references are leases that clients must renew

No AI/ML Processing

Python RPC

- Several RPC implementations
  - Pyro https://pyro5.readthedocs.io
  - **RPyC** (Remote Python Call) https://rpyc.readthedocs.io
  - ZeroRPC https://www.zerorpc.com

- Why transparency is easy in Python
  - Dynamic typing
  - Runtime introspection: ability to examine objects during execution via `inspect` module https://docs.python.org/3/library/inspect.html
    - Inspect live objects, examine class contents, retrieve method source code, and extract function argument list

No AI/ML Processing

RPyC: features

- RPyC: transparent RPC in Python
  - Transparent access to a remote object through a dynamic proxy (*network reference* or netref)
  - Proxies are created at runtime, so no IDL service definition, no IDL compiler, no generated stubs
  - Same syntax as a local object: operations are forwarded to the remote side, but they can fail
- Symmetric
  - Client and server can call each other (e.g., server can invoke callbacks on client)
- Synchronous and asynchronous calls
- Security
  - Only exposed members are reachable
  - Runs over TLS or through SSH tunnel

No AI/ML Processing

RPyC: server and client

- Server
  - Multiple implementations
    - `ThreadedServer`: one thread per connection
    - `OneShotServer`, `ThreadPoolServer`, `ForkingServer`: other concurrency models
  - Binds to default port 18812 (18821 with SSL)
    - Developer can specify host IP and port
- Client
  - On connection, the client gets a netref to `conn.root`, a netref to the service's root object
  - Further netrefs are created whenever calls return mutable objects
  - Transparent proxy forwards operations to the remote object

No AI/ML Processing

RPyC: parameter passing

- RPyC uses marshaling (serialization) internally
- By value
  - Immutable built-in types (e.g., `int`, `string`, `tuple`)
  - Values are sent directly to remote side
  - Remote copy is independent of the original
- By reference
  - Mutable objects (e.g., open files)
  - Passed as `netref` (network reference)
  - Remote server contacts client to access attributes or call methods on the original object
    - Changes affect the original local object
  - Enables location-sensitive objects (files, OS resources, `sys.stdout`)
    - E.g., remote process can print to local console via `sys.stdout`

No AI/ML Processing

RPyC: services

- Each connection exposes a service, which defines the set of operations that can be remotely invoked (its policy)
- Service definition
  - A service is a class derived from `rpyc.Service`
  - It defines objects and methods that can be accessed remotely (exposed)
  - The client reaches it through `conn.root`
  - The interface is implicit: no separate contract file, unlike SunRPC or Java RMI

No AI/ML Processing

RPyC: services

- Exposing methods
  - A method can be called remotely if its name starts with `exposed_`
- Exposing classes
  - A class named `exposed_` can be instantiated by the client
  - The object lives on the server and the client holds a netref
  - Only its exposed methods are accessible
- Everything else is private to the server
- Security: what you expose is your remote attack surface

No AI/ML Processing

RPyC example: calculator server

```
import rpyc

class CalculatorService(rpyc.Service):
    def exposed_add(self, a, b):
        return a + b
    def exposed_div(self, a, b):
        return a / b
    def exposed_fib(self,n):
        seq = []
        a, b = 0,1
        while a < n:
            seq.append(a)
            a, b = b, a+b
        return seq
    def foo(self):  # foo is private
        print("foo")

if __name__ == "__main__":
    from rpyc.utils.server import ThreadedServer
    t = ThreadedServer(CalculatorService, port=18861)
    print('Service started on port 18861')
    t.start()
```


No AI/ML Processing

RPyC example: calculator client

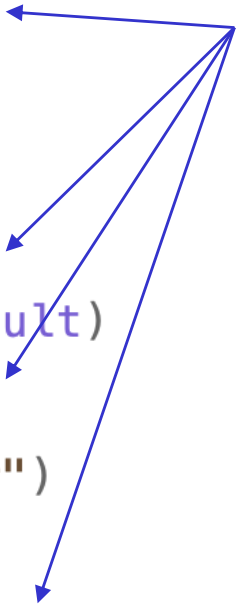
```python
import rpyc

conn = rpyc.connect("localhost", 18861)

# RPC for add
x = conn.root.add(4,7)
assert x == 11

# RPC for div
try:
    result = conn.root.div(4, 2)
    print("Division result:", result)
    conn.root.div(4, 0)
except ZeroDivisionError:
    print("Division by zero error")

# RPC for fib
print(conn.root.fib(1000))
```



The remote service is accessed via the root object of the connection (`conn.root`).
This object is a network reference (`netref`) to the service instance running on the server.
Any operation on `conn.root` is automatically forwarded to the server, making the remote service appear local.
This mechanism allows the client to transparently call exposed methods.

No AI/ML Processing

RPyC: async calls and future pattern

- Wrap call with `async_()` to make it asynchronous
- Client starts remote call and continues without blocking
- Call returns immediately with `AsyncResult` object, a future that will *eventually* hold the result
- Using the *future pattern*, client can:
  - test `AsyncResult` for completion using `ready`
  - block until completion using `wait()`
  - retrieve result using `value`; blocks if it has not arrived, and raises the exception if the call failed
  - set timeout for result using `set_expiry()`
  - register a callback function to execute when result arrives using `add_callback()`

See https://rpyc.readthedocs.io/en/latest/docs/async.html

No AI/ML Processing

RPyC: async calls and future pattern

- Calculator example
- Server

```
    def exposed_slow_add(self, a, b):
        time.sleep(3) # simulate a long computation return a + b
        return a + b
```

- Client

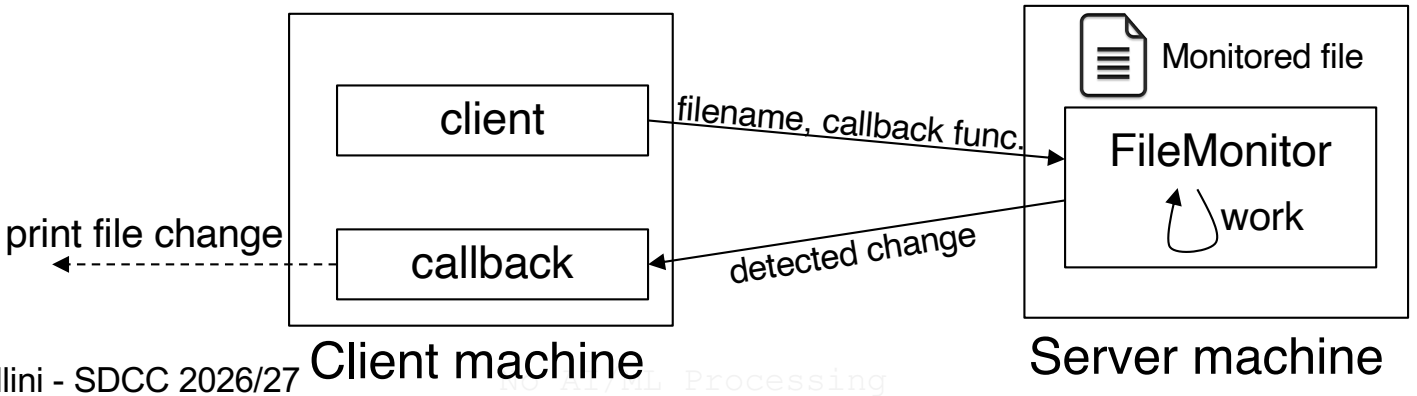
```
# RPC for slow add
print("Slow add")
slow_add = rpyc.async_(conn.root.slow_add)

res = slow_add(4, 7)     # returns immediately: AsyncResult
print(res.ready)         # False: the result has not arrived yet
# ... the client can do other work here ...
res.wait()               # block until the result arrives
print(res.ready, res.value) # True 11
```


No AI/ML Processing

RPyC: asynchronous callbacks

- Events can be implemented as asynchronous callbacks
  - Server produces events, client consumes them asynchronously
  - Symmetric: server calls client
- Example: FileMonitor
  - Server monitors a file and detects change
  - On file change, server sends an event to client (through async callback) providing old and new `stat` results
  - Client needs a background thread to receive notifications
  - Client is the observer → *observer pattern*



No AI/ML Processing

RPyC example: FileMonitor server

```
import rpyc
import os
import time
from threading import Thread

class FileMonitorService(rpyc.Service):
    class exposed_FileMonitor(object): # expose class
        def __init__(self, filename, callback, interval = 1):
            self.filename = filename    # file to monitor
            self.interval = interval    # monitoring interval
            self.last_stat = None
            self.callback = rpyc.async_(callback) # async callback
            self.active = True
            self.thread = Thread(target = self.work)
            self.thread.start()
        def exposed_stop(self): # must be exposed since client calls it remotely
            self.active = False
            self.thread.join()
        def work(self):          # check for file changes
            while self.active:
                stat = os.stat(self.filename) # get status of monitored file
                if self.last_stat is not None and self.last_stat != stat:
                    self.callback(self.last_stat, stat) # notify client of change
                self.last_stat = stat
                time.sleep(self.interval) # monitor file status periodically

if __name__ == "__main__":
    from rpyc.utils.server import ThreadedServer
    # spawn a thread for each connection
    ThreadedServer(FileMonitorService, port = 18871).start()
```

Exposing the class, client can call remotely its constructor with `conn.root.FileMonitor(...)`, but only the `exposed_` methods are accessible remotely

No AI/ML Processing

RPyC example: FileMonitor client

```python
import rpyc
import time
import os

# File to monitor
filename = "/tmp/floop.bloop"
if os.path.exists(filename):
    os.remove(filename)

# Open file to monitor
f = open(filename, "wb", buffering=0)

# Connect to RPyC server that monitors file changes
conn = rpyc.connect("localhost", 18871)

# Start background thread that handles incoming asynchronous events from server
# Necessary for receiving callbacks without blocking the client
bgsrv = rpyc.BgServingThread(conn)

# Define the callback function on client side
# Called by server whenever the monitored file changes
def on_file_changed(oldstat, newstat):
    print("\nfile changed")
    print(f"    old stat: {oldstat}")
    print(f"    new stat: {newstat}")

# Create the file monitor object and pass filename and callback function as arguments
filemon = conn.root.FileMonitor(filename, on_file_changed)
```

Client passes the callback function (by reference)

No AI/ML Processing

RPyC example: FileMonitor client

```
# Wait for file monitor to check the initial file state before changing it
time.sleep(2)

# Change file size; server detects the change and calls callback notifying the event
print("\n---> Change file")
f.write(b"loop")
time.sleep(2)

# Change file size again
print("\n---> Change file again")
f.write(b"groop")
time.sleep(2)

# Cleanup
f.close()
filemon.stop()
bgsrv.stop()    # stop background thread handling server callbacks
conn.close()
```


No AI/ML Processing

RPyC: summing up key features

- Transparent access through netrefs
- No IDL, no generated stubs, service registry optional
- Only exposed classes and methods are visible remotely
- Synchronous and asynchronous calls
- Symmetric (callbacks)
- Concurrency: shared state needs explicit synchronization
- Failure semantics: best effort, requires explicit failure handling
  - No built-in retries, no server deduplication

No AI/ML Processing

RPC in Go

- We introduce the Go programming language
  https://www.ce.uniroma2.it/courses/sdcc2627/slides/Go.pdf

- What about RPC in Go?

No AI/ML Processing

RPC in Go

- Go provides built-in support for RPC
  - Package `net/rpc` https://pkg.go.dev/net/rpc
  - Type-safe RPC mechanism

- Main features
  - Remote access to exported methods of object
  - TCP or HTTP as “transport” protocols
  - Concurrent client requests

No AI/ML Processing

RPC in Go: defining an RPC service

- Define a type (struct) whose methods will be exposed remotely
- The exposed method must follow the rules:

```
func (t *T) MethodName(args ArgType, reply *ReplyType) error
```

  - Method type and name are exported
    - In Go: start with capital letter
  - Method takes two arguments, both exported or built-in types
    - First argument: input (usually a struct)
    - Second argument: pointer to reply struct for response
  - Method returns a single error
    - If non-nil, the client gets the error string and the reply is not sent

No AI/ML Processing

Defining an RPC service: example

```
package arith
import "errors"

type Arith struct{}  // Service struct
type Args struct {A, B int}  // Input arguments
type Result int  // Result type for Multiply method
type Quotient struct {Quo, Rem int}  //Result type for Divide method

func (t *Arith) Multiply(args *Args, res *Result) error {
        *res = args.A * args.B
        return nil
}
func (t *Arith) Divide(args *Args, quo *Quotient) error {
        if args.B == 0 {
                return errors.New("divide by zero")
        }
        quo.Quo = args.A / args.B
        quo.Rem = args.A % args.B
        return nil
}
```

RPC in Go: server

- On server side
  - Create a TCP (or HTTP) server to receive incoming requests
  - Use **`Register`** (or **`RegisterName`**) to make an object's exported methods visible as RPC procedures and publish them on server

```
func (server *Server) Register(rcvr any) error
func (server *Server) RegisterName(name string, rcvr
any) error
```

  - Takes interface as input: `any` is an alias for `interface{}`, allows any value (typically a pointer to a struct)
  - `Register` uses the name of the concrete type and registers methods
  - `RegisterName` allows to specify a custom service name

No AI/ML Processing

RPC in Go: server

- Use **`Listen`** to create a network listener and announce the server on a specific port

```
func Listen(network, address string) (Listener, error)
```

- Use **`Accept`** to accept connections on the listener and serve requests

```
func (server *Server) Accept(lis net.Listener)
```

  - Concurrency out of the box: `Accept` spawns a separate goroutine for each request
  - `Accept` blocks; to handle other tasks concurrently in main, call `Accept` in a goroutine

- Can also use an HTTP handler for RPC messages
  - Use `HandleHTTP` and `http.Serve`
  - See example on course site

No AI/ML Processing

RPC in Go: client

- On client side
  - Use **`Dial`** to connect to RPC server at the specified address and port using the given transport protocol (e.g., `"tcp"`)

```
func Dial(network, address string) (*Client, error)
```

    - Use `DialHTTP` for HTTP connection
  - Use **`Call`** to call synchronous RPC
    - `Call` waits for the remote procedure to complete

```
func (client *Client) Call(serviceMethod string, args
any, reply any) error
```

  - Use **`Go`** to call asynchronous RPC
    - `Go` invokes the call asynchronously and signals completion via `Done` channel in `Call` struct

```
func (client *Client) Go(serviceMethod string, args
any, reply any, done chan *Call) *Call
```


No AI/ML Processing

RPC in Go: client

- On client side
  - Struct **`Call`** represents an active RPC

```
type Call struct {
    ServiceMethod string  // The name of the service and method
                          // to call.
    Args any // The argument to the function (*struct)
    Reply any // The reply from the function (*struct)
    Error error // After completion, the error status.
    Done chan *Call // Receives *Call when Go is complete.
}
```


No AI/ML Processing

RPC in Go example: calculator

- We consider a simple RPC calculator with two functions: multiply and divide two integers
- Code available on course site

No AI/ML Processing

RPC in Go: synchronous call

- Need some setup in advance of this, see example
- `Call` makes a blocking RPC call: invokes the remote function, waits for it to complete, and returns its error status

```
// Synchronous call
args := arith.Args{n1, n2}
var mulReply arith.Result
err = client.Call("Arithmetic.Multiply", args, &mulReply)
if err != nil {
   log.Fatal("Error in Arithmetic.Multiply:", err)
}
fmt.Printf("Arithmetic.Multiply: %d*%d=%d", args.A, args.B,
   mulReply)
```

```
func (client *Client) Call(serviceMethod string, args any, reply any)
error
```


No AI/ML Processing

RPC in Go: asynchronous call

- Go uses a channel as parameter to retrieve the RPC reply when the call is complete
- The Done channel will signal when the call is complete by returning the same `Call` object
  - If no channel in input, a new one will be allocated by Go

```
// Asynchronous call
divReply := new(Quotient)
divCall := client.Go("Arithmetic.Divide", args, divReply, nil)
replyCall = <- divCall.Done
// check errors, print, etc.
```

channel

```
func (client *Client) Go(serviceMethod string, args any, reply any,
done chan *Call) *Call
```

- Go internal implementation: see https://go.dev/src/net/rpc/client.go?s=8029:8135 - L284

No AI/ML Processing

Go's RPC semantics

- Maybe semantics
- Client-side: no automatic retries, no built-in timeouts
- Client returns an error if:
  - Connection breaks
  - Server method returns an error
- Server-side: no deduplication of requests
  - Client request has a unique sequence number `Seq` but it is only for matching replies on the client, and each retry gets a new `Seq`
- Retries/timeouts/dedup (request IDs) are developer responsibility

No AI/ML Processing

RPC in Go: marshaling and unmarshaling

- `net/rpc` uses `encoding/gob` by default, applied automatically to args and reply
  https://pkg.go.dev/encoding/gob
- `gob` is a binary, self-describing, stream-based format
  - Type is described once per stream, then values follow
  - Pointers are flattened (values only): no aliasing, no cycles
  - Only exported fields are sent, and zero values are omitted
  - Fields are matched by name, so sender and receiver types may differ
- Use `gob` directly for custom Go-to-Go serialization or persistence
- `gob` is Go-specific
  - RPC client and server must be written in Go
- For interoperability use `net/rpc/jsonrpc` or a custom codec

No AI/ML Processing

Go RPC: summing up key features

- Transparent access to remote structs and methods
- Only exported and registered methods
- Both synchronous and asynchronous calls
- Concurrency: one goroutine per request
- No IDL
- No service registry (client needs host:port)
- Marshalling via gob, Go-to-Go
- Maybe semantics
  - No built-in timeouts, retries, server deduplication
- No security or versioning

No AI/ML Processing

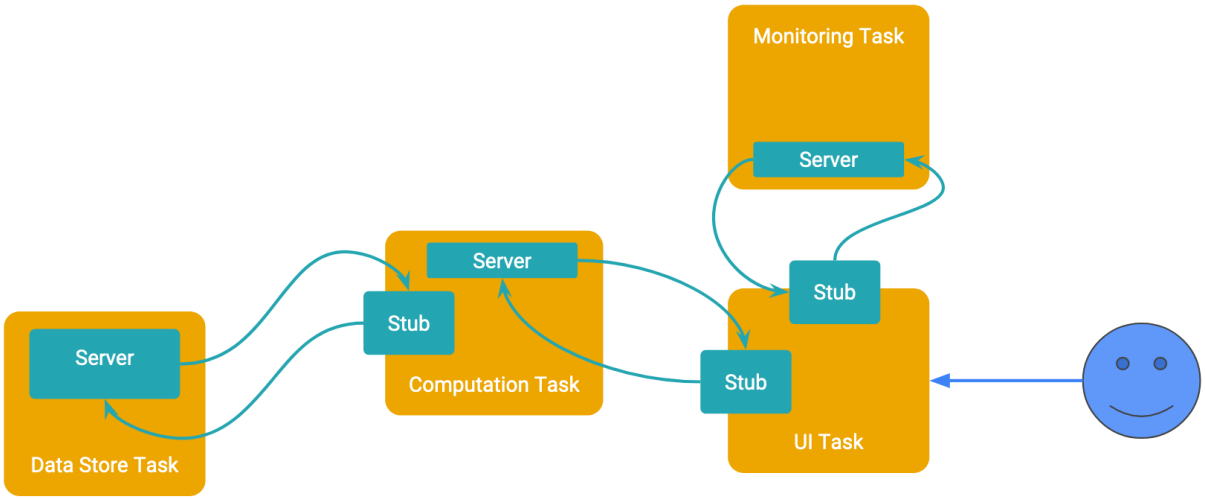
Comparing RPC so far

- We compare RPC implementations and how they differ in terms of distribution transparency
  - Access
  - Location
  - Concurrency
  - Failure
  - Replication
    - How? A server-side load balancer acting as proxy between RPC clients and replicated RPC servers, so clients see a single endpoint
      - Does not provide failure transparency: in-flight calls are lost when a replica fails
      - Single point of failure

No AI/ML Processing

Motivation for new RPC middleware

- Large-scale distributed applications composed of **microservices**
  - Microservices architecture: build application as a collection of *independent*, *autonomous* (developed, deployed, and scaled independently), *business capability-oriented*, and *loosely coupled* services
  - Multi-language (i.e., polyglot) development
  - Communication mainly structured as RPCs



No AI/ML Processing

gRPC

- Open-source, high-performance RPC framework
  https://grpc.io/
- Can run in any environment
  - Multi-language, multi-platform support
- Background
  - Developed by Google, open-sourced in 2015, now CNCF project
- Adoption
  - Used by large companies (Cisco, Square, Netflix, and others)
  - Used in cloud-native infrastructure: etcd, containerd, and Kubernetes interfaces



No AI/ML Processing

gRPC

- Main use cases
  - Polyglot microservices within and across data centers
    - gRPC supports load balancing, tracing, health checks, authentication
  - Mobile and devices apps to backends
  - Generating client libraries

- Widely used for model serving and distributed training
  - Model serving: TensorFlow Serving, NVIDIA Triton expose gRPC alongside REST https://github.com/tensorflow/serving/blob/master/tensorflow_serving/apis/prediction_service.proto
  - TensorFlow's distributed runtime use gRPC among workers
  - gRPC access to Gemini through Google APIs

No AI/ML Processing

gRPC: key features

- Transport and communication
  - Uses **HTTP/2** for efficient transport
  - Supports flow control, multiplexing, bidirectional streaming, and header compression
- Interface definition and code generation
  - **Protocol Buffers** as IDL
  - Simple service definition, automatic client/server code generation
  - Schema evolution
- Advanced features
  - Blocking or non-blocking client bindings (language-dependent)
  - Reliability: deadlines (set by client and propagated to server), cancellation, status codes, retry policies (idempotency still required)
  - Extensibility: interceptors (logging, metrics, tracing
  - Security: TLS/mTLS and token authentication

No AI/ML Processing

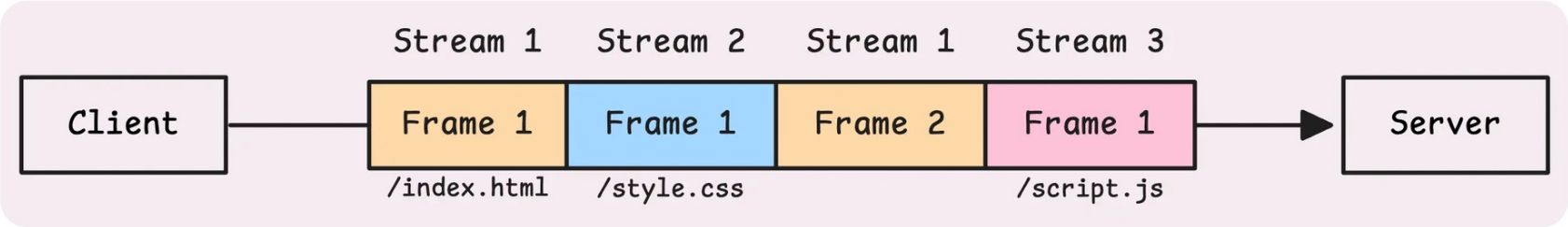
gRPC transport: HTTP/2

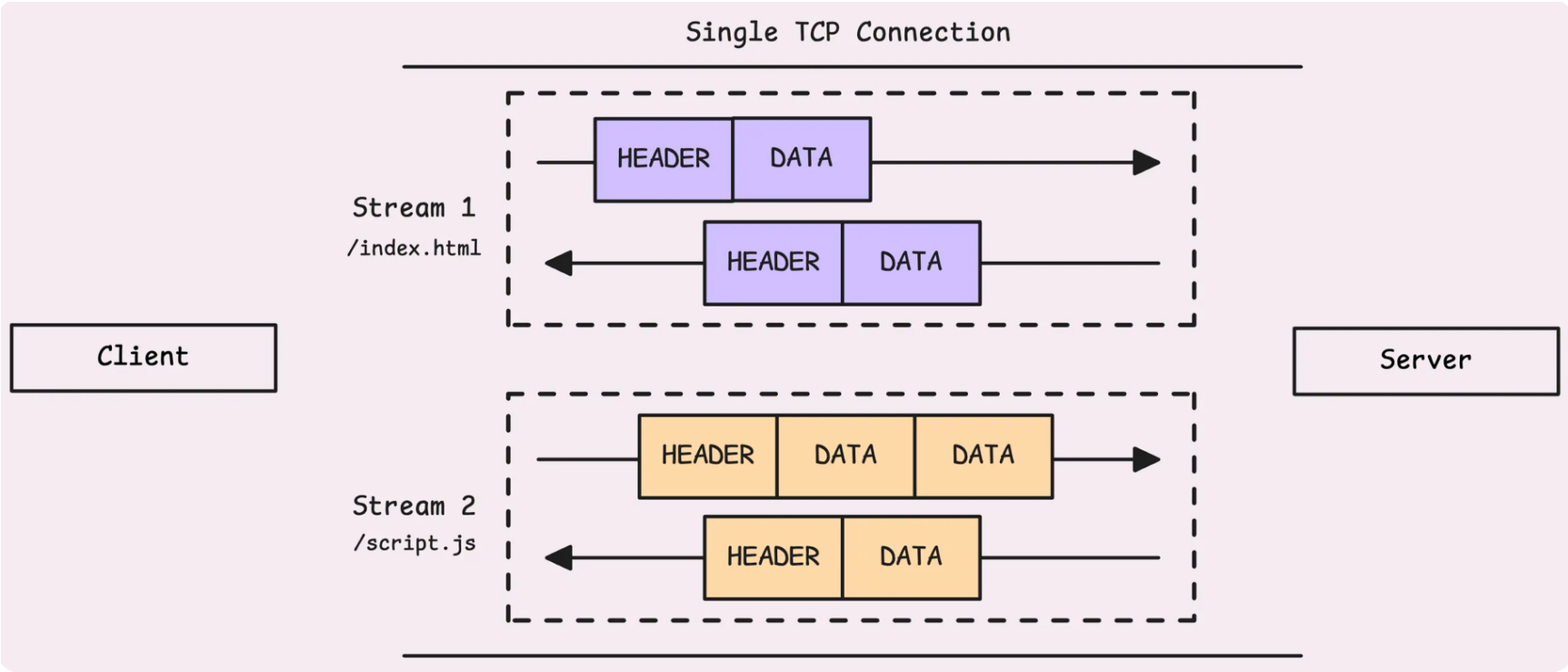
- Why HTTP/2?
  - Same HTTP semantics (methods, status codes, headers), but significant performance improvements over HTTP 1.x
  - Enables low-latency, high-throughput communication
- HTTP/2 in a nutshell
  - Binary framing layer: requests and responses are divided into small messages framed in binary
  - From TPC connections to frames
    - Connection: TPC connection between client and server
    - Stream: bidirectional flow of bytes within the connection, identified by an ID; carries one or more messages
    - Message: made of one or more frames, represents the logical request or response
    - Frame: smallest unit, its header carries the stream ID
  - Frames from different streams are interleaved on the connection and reassembled at the receiver

No AI/ML Processing

gRPC transport: HTTP/2

- HTTP/2 in a nutshell





No AI/ML Processing

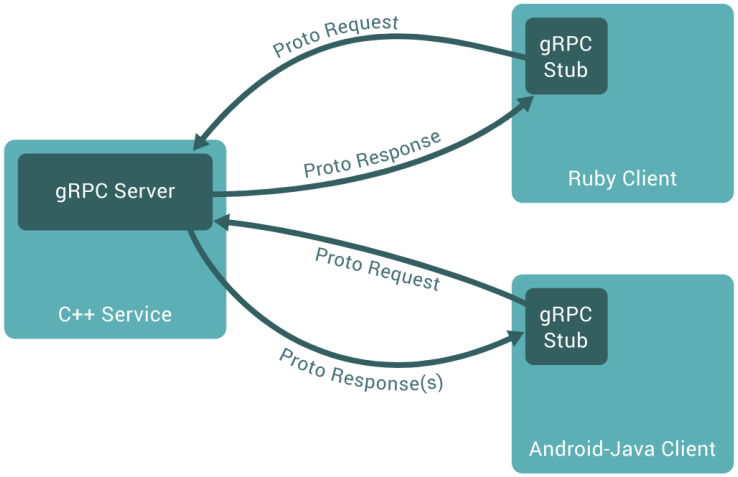
gRPC transport: HTTP/2

- HTTP/2 in a nutshell
  - Multiplexing: many concurrent requests/responses per client share the same TCP connection
    - Efficient use of TCP connections
    - No HTTP/1.1 head-of-line blocking
  - Header compression: repeated headers are not resent in full, reducing overhead
  - Flow control: per stream and per connection, giving backpressure
  - Full-duplex streams: both sides can send at the same time
  - Limit: a lost TCP packet still stalls all streams (TCP-level head-of-line blocking)
    - Motivation for HTTP/3 over QUIC

No AI/ML Processing

gRPC: Protocol buffers

- gRPC uses Protocol buffers (aka *protobuf*) as:
  - IDL for service interface definition, independent of language
  - Serialization format: compact binary encoding by field number; not self-describing (schema is needed to decode)
  - `protoc` plus gRPC plugin generate message classes, client stubs and server interfaces or base classes in each language
- Design pattern: proxy pattern
  - Client stub acts as local proxy for remote service
  - Server skeleton handles incoming RPC calls



No AI/ML processing

Protocol buffers

- Open-source, language- and platform-neutral mechanism for serializing structured data, developed at Google
- Binary encoding: compact and efficient
- Data types structured as messages
  - Message: record made of fields, each with a name, a type and a unique number within the message (e.g., `string name = 1`)
    - Field types: scalars, enums, nested messages, repeated, map, oneof
    - Field numbers are used in the binary encoding to identify data

```
syntax = "proto3";

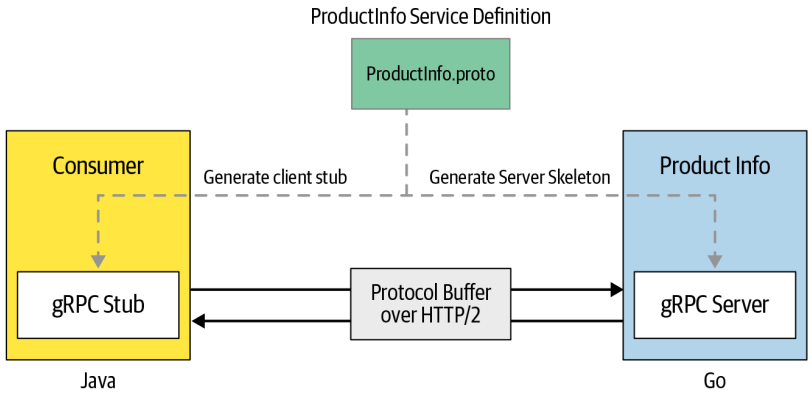
message Person {
  string name = 1;
  int32 id = 2;
  string email = 3;
}
```

- Typed fields
  - Generated code gives type-checked access to them
    - E.g., `id = "abc"` gives a compile error as the compiler knows `id` is `int32` because of the schema

No AI/ML Processing

Protocol buffers: example

- ProductInfo service interface



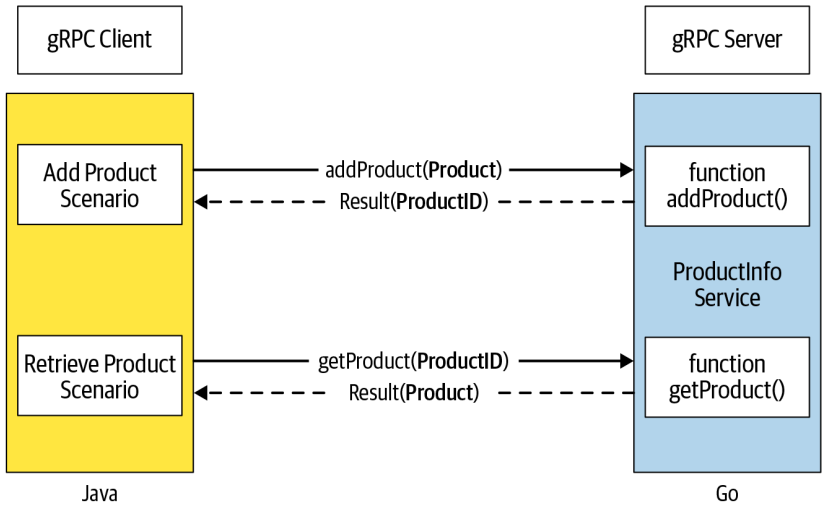

```
// ProductInfo.proto
syntax = "proto3";
package ecommerce;

service ProductInfo {
    rpc addProduct(Product) returns (ProductID); // returns product's id
    rpc getProduct(ProductID) returns (Product);
}

message Product {
    string id = 1; // assigned by server
    string name = 2;
    string description = 3;
}

message ProductID {
    string value = 1;
}
```

Client-server interaction



No AI/ML Processing

gRPC: basic steps

1. Define service and messages: write `.proto` file using protobufs as IDL and describing
   - Service: collection of remote methods
   - Message types exchanged between client and server
2. Generate server and client code
   - Run `protoc` (protocol buffer compiler) for the chosen language(s) from proto definition
     - Go: run `protoc` with the language plugins
     - Java: Maven/Gradle plugins
3. Implement service and client
   - Use language-specific gRPC API (e.g., Go, Java, Python) to write client and server applications
   - Focus: Go using gRPC-Go https://grpc.io/docs/languages/go

No AI/ML Processing

gRPC: helloworld example in Go

- See https://grpc.io/docs/languages/go/quickstart/ and https://github.com/grpc/grpc-go/tree/master/examples/helloworld/

1. Define service and messages (`helloworld.proto`)

```
package helloworld;

// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```


No AI/ML Processing

gRPC: helloworld example in Go

2. Compile the service definition

```
$ protoc --go_out=. --go_opt=paths=source_relative \
    --go-grpc_out=. --go-grpc_opt=paths=source_relative \
    helloworld/helloworld.proto
```

- Automatically generated files:
  - `helloworld.pb.go` (option `--go_out`)
    - A Go struct per message, with getters
    - Serialization by protofub runtime, called by gRPC
  - `helloworld_grpc.pb.go` (option `--go-grpc_out`)
    - Client: `GreeterClient` interface, its stub implementation, constructor `NewGreeterClient`
    - Server: `GreeterServer` interface to implement (embedding `UnimplementedGreeterServer`)
    - Dispatching by gRPC runtime, registered through `RegisterGreeterServer`
    - Uses the message structs from `helloworld.pb.go`

No AI/ML Processing

gRPC: helloworld example in Go

3. Create and run the server; composed of two parts
   a. Implement the generated service: the actual “work”

```
func (s *server) SayHello(ctx context.Context,
    in *pb.HelloRequest) (*pb.HelloReply, error) {
     // Implement service logic here
}
```

   - `ctx`: deadlines, cancellation, and metadata
   - `in`: the request message

No AI/ML Processing

gRPC: helloworld example in Go

3. Create and run the server; composed of two parts

   b. `Listen` on a TCP port; create a new gRPC server instance; `Register` the service implementation; `Serve` blocks, accepting connections

```
lis, err := net.Listen("tcp", port)
if err != nil {
    log.Fatalf("failed to listen: %v", err)
}
s := grpc.NewServer()
pb.RegisterGreeterServer(s, &server{})
s.Serve(lis)
```

   – Each RPC runs in its own goroutine (shared state needs a mutex)

No AI/ML Processing

gRPC: helloworld example in Go

3. Create and run the client
   - Create a *gRPC channel* to server; channel is long-lived and safe for concurrent use (shared among calls)

```
conn, err := grpc.NewClient(address, opts...)
```

     - Close with `defer`
     - `opts` can include TLS credentials, etc.
     - Insecure: use `grpc.WithTransportCredentials(insecure.NewCredentials())`
   - Create a client stub with `NewGreeterClient` (generated from `.proto`)

```
c := pb.NewGreeterClient(conn)
```

   - Call the method on the stub: send request, receive response

```
r, err := c.SayHello(ctx, &pb.HelloRequest{Name: name})
```

     - Create and populate the request message
     - Pass a context for deadline, cancellation or metadata
     - Block until response message or error

No AI/ML Processing

gRPC: updating a service

- gRPC simplifies application updates
  - Incremental updates without breaking existing functionalities
- Adding a method
  1. Edit `.proto` file: add the method and any new message
  2. Regenerate stubs
  3. Update server implementation to implement the new method
  4. Update client code to call the new method
- Changes
  - Compatible: add methods; add fields with new numbers; remove fields and reserve their numbers
  - Breaking: change or reuse field numbers or types; rename package, service or method; change existing method messages → develop new package version

No AI/ML Processing

gRPC: updating helloworld example

- We update helloworld adding `SayHelloAgain()`
  1. Update `.proto` file
  2. Regenerate gRPC code using `protoc`
  3. Add the new method to server implementation

```
func (s *server) SayHelloAgain(ctx context.Context, in
   *pb.HelloRequest) (*pb.HelloReply, error) {
    log.Printf("Received: %v", in.GetName())
    return &pb.HelloReply{Message: "Hello again " +
   in.GetName()}, nil
}
```

  4. Update client code calling the new method

```
r, err = c.SayHelloAgain(ctx, &pb.HelloRequest{Name: name})
 if err != nil {
    log.Fatalf("could not greet: %v", err)
 }
 log.Printf("Greeting: %s", r.GetMessage())
```


No AI/ML Processing

gRPC: ProductInfo example

- ProductInfo microservice which manages products and their information
- Operations:
  - Add new products, retrieve product details

1. Define service in `.proto` file (see slide 118)
2. Implement server and client in Go
3. Implement server and client in Java

Go code: Teams/course website

Java code: https://github.com/grpc-up-and-running/samples/tree/master/ch02

See chapter 2 of gRPC: Up and Running
https://www.oreilly.com/library/view/grpc-up-and/9781492058328/

No AI/ML Processing

gRPC: service method types

- gRPC supports 4 kinds of service methods
  - Defined in `.proto` file
  - Example: route guide

  https://github.com/grpc/grpc-go/tree/master/examples/route_guide

1. Simple (unary) RPC
   - Client sends a single request to server and waits for a single response
   - A normal function call

```
rpc GetFeature(Point) returns (Feature) {}
```

2. Server-streaming RPC
   - Client sends one request to server
   - Server responds with a stream of messages
   - Client reads until the stream ends
   - Guaranteed message ordering from server to client

```
rpc ListFeatures(Rectangle) returns (stream Feature) {}
```


No AI/ML Processing

gRPC: service method types

3. Client-streaming RPC
   - Client sends a stream of messages to server
   - Server reads all messages and returns a single response
   - Guaranteed message ordering from client to server

```
rpc RecordRoute(stream Point) returns (RouteSummary) {}
```

4. Bidirectional streaming RPC
   - Client and server each send a stream of messages, simultaneously (i.e., full duplex)
   - Each side can read and write independently
   - Server can wait for all client messages before replying, or server and client can play “ping-pong”
   - Message ordering is preserved in each direction

```
rpc RouteChat(stream RouteNote) returns (stream
RouteNote) {}
```


No AI/ML Processing

gRPC: route guide example in Go

- Goal: demonstrates all four types of gRPC communication in Go using a location-based route guide service
- protobufs: service and message definitions in `routeguide/route_guide.proto`
- Server: implements gRPC service methods (`GetFeature`, `ListFeatures`, `RecordRoute`, `RouteChat`)
- Client: calls RPCs using Go gRPC stubs
- Transport: supports both plaintext and TLS-secured connections

No AI/ML Processing

gRPC: distribution transparency

- Access transparency
  - Achieved through the proxy pattern
  - Cross-language: one `.proto`, stubs in many languages
  - Partial: remote failures are visible (e.g., status codes, deadlines)

- Location transparency
  - Clients use a logical name
  - A name resolver (DNS by default) maps it to one or more server addresses
  - Custom resolvers can be implemented through resolver API https://grpc.io/docs/guides/custom-name-resolution
  - External service discovery (e.g., Eureka) can be integrated to discover services dynamically
  - Service meshes (e.g., Istio, Envoy) can provide service discovery and additional infrastructure-level capabilities for service-to-service communication

No AI/ML Processing

gRPC: distribution transparency

- Concurrency transparency
  - Provides concurrency-safe APIs, but concurrency management is application-dependent (e.g., shared state) → partial
  - gRPC-Go uses goroutines and synchronization mechanisms https://github.com/grpc/grpc-go/blob/master/Documentation/concurrency.md
- Replication transparency
  - Integrated on client-side (`round_robin` instead of `pick_first`) or through external load balancers (e.g., Envoy) to distribute requests across replicated servers https://grpc.io/docs/guides/custom-load-balancing
  - Replication logic (e.g., keeping replica consistent) must be handled by the application

No AI/ML Processing

gRPC: distribution transparency

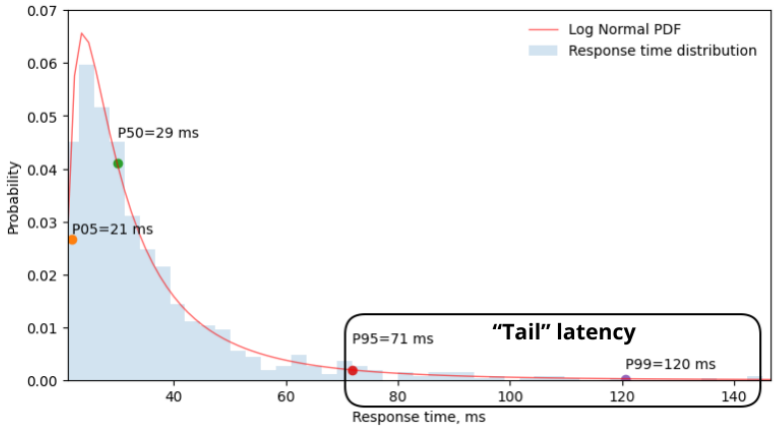
- Failure transparency
  - Depends on the configuration
  - By default, only *transparent retry* → maybe
    - Retry is performed only when the transport guarantees gRPC server never saw the call (request never left client or server refused it)
- Failure tools include policy-based retry and hedging
  - Must be configured, not automatic
  - Retry with exponential backoff for transient failures (retry with backoff pattern) https://grpc.io/docs/guides/retry
  - Retry configured → at-least-once, duplicates are possible
  - Retry + server-side deduplication → at-most-once
    - But developer has to implement application-level deduplication
  - Server crash mid-call may result in *partial execution*, with no automatic rollback (no transactions) → no exactly-once

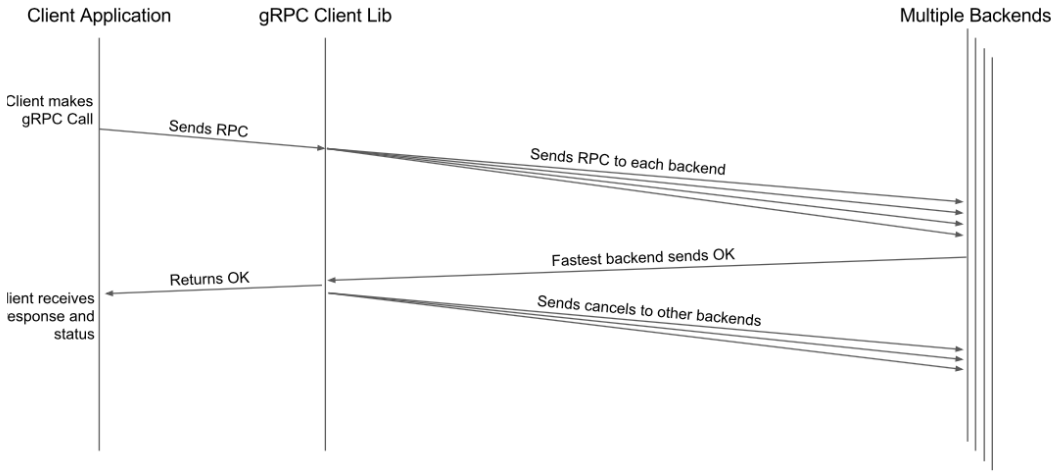
No AI/ML Processing

gRPC: distribution transparency

- Request hedging
  - Send copies of the same request to multiple replica
  - Client uses first response and cancels outstanding requests

  https://grpc.io/docs/guides/request-hedging
  - Goal: reduce tail latency
    - Dean and Barroso, The Tail at Scale, *ACM Comm.* 2013 https://research.google/pubs/the-tail-at-scale/
  - X Can increase load: gRPC uses throttling token bucket (shared with retries)
  - X Safe only for idempotent requests





No AI/ML Processing

gRPC: additional features

- Security: TLS support
  - Gives confidentiality, integrity, and server authentication
- How to use
  - Generate certificates
    - Development: private or self-signed certificate
    - Production: trusted certificate from Certificate Authority (CA)
  - Server side: configure TLS certificates + private key
  - Client side: trusts signing CA and verifies server name
  - mTLS (mutual TLS): for client-server authentication, ensuring both sides are verified
  - Example: https://github.com/grpc/grpc-go/tree/master/examples/features/encryption

No AI/ML Processing

gRPC: additional features

- Benchmarking and load testing
  - Use load testing tool for gRPC, e.g., ghz https://ghz.sh
    - Load testing tool: generates concurrent requests against a service to measure how it behaves under load, reporting throughput, latency (including percentiles) and error rate
  - Test gRPC call types (e.g., unary vs. streaming), payload sizes and concurrency to simulate real traffic
  - Measure throughput, latency percentiles and errors; compare with and without TLS

No AI/ML Processing

gRPC: weaknesses

- Browser support
  - Browers cannot call gRPC directly
  - Requires gRPC-Web with a proxy, but limit on supported call types https://grpc.io/docs/platforms/web/basics/
- Binary protocol
  - protobufs is efficient but not human-readable
  - Requires additional tools, e.g., gRPC command-line tool like `grpcurl` or Wireshark https://wiki.wireshark.org/gRPC
- Load balancing and traffic management
  - No built-in server-side load balancing, only client-side
  - Requires a service mesh for traffic management
- Operational complexity at scale
  - Many service definitions (`.proto` files and their changes) and certificates

No AI/ML Processing

References

- Chapter 4 of van Steen & Tanenbaum book
- Remote Procedure Calls https://www.linuxjournal.com/article/2204
- Java Remote Method Invocation Specification https://docs.oracle.com/en/java/javase/27/docs/specs/rmi
- Trail: RMI https://docs.oracle.com/javase/tutorial/rmi/
- RPyC tutorial https://rpyc.readthedocs.io/en/latest/tutorial.html
- RPyC documentation https://rpyc.readthedocs.io
- Go's package rpc https://pkg.go.dev/net/rpc
- RPC in the Go standard library, in Building Microservices with Go https://subscription.packtpub.com/book/web-development/9781786468666/1/ch01lvl1sec12/rpc-in-the-go-standard-library
- gRPC - Up and Running https://www.oreilly.com/library/view/grpc-up-and/9781492058328/

No AI/ML Processing