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. 2023/24

Valeria Cardellini

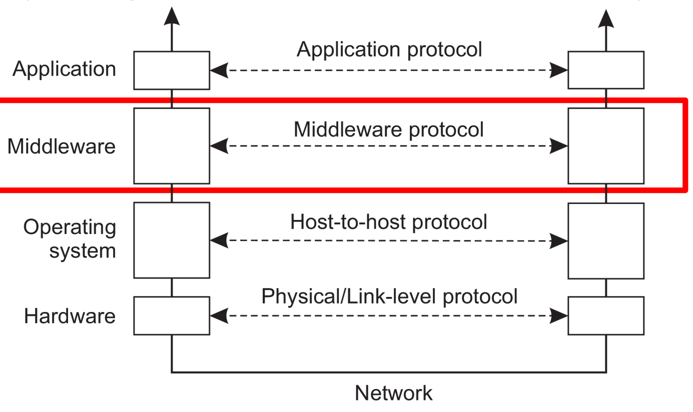
Laurea Magistrale in Ingegneria Informatica

Communication in distributed systems

- Based on message passing
  - Send and receive messages
- To allow for message passing, parties must agree on many low-level details
  - How many volts to signal a 0 bit and how many for a 1 bit?
  - How many bits for an integer?
  - How does the receiver know which is the last bit of the message?
  - How can the receiver find if a message has been corrupted and what to do then?

Basic networking model and its adaptation

- We know the solution: divide network communication in layers
  - The well known ISO/OSI reference model
  - We don't care about low-level details: for many distributed systems, the lowest-level interface is that of the network layer
- Adapted layering scheme for distributed systems



Middleware layer

- **Middleware** provides common services and protocols that can be used by many different distributed applications and systems
  - General-purpose
  - Application-independent
- Some examples of middleware services and protocols:
  - Communication: remote procedures/methods, queue messages, multicasting
  - Naming: to allow easy sharing of resources
  - Security: to allow applications to communicate securely
  - Distributed consensus, including distributed commit
  - Distributed locking: to protect a shared resource against simultaneous access by a collection of distributed processes (e.g., multiple clients update a file in shared storage)
  - Data consistency

Types of communication

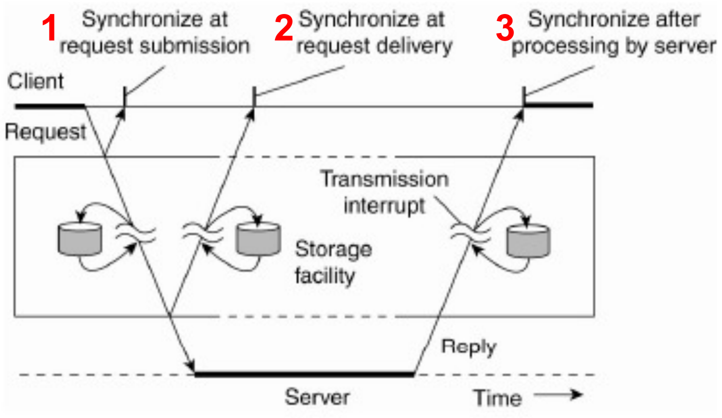
- Let's distinguish
  - Persistency
    - Transient versus persistent communication
  - Synchronization
    - Synchronous vs. asynchronous communication
  - Time dependence
    - Discrete vs. streaming communication

Persistent vs. transient communication

- Persistent communication
  - Message is stored by middleware as long as it takes to deliver it to receiver
  - Sender does not need to continue execution after submitting message
  - Receiver does not need to be executing when message is submitted
- Transient communication
  - Message is stored by middleware only as long as sender and receiver are executing: sender and receiver have to be active at time of communication
  - If delivery is not possible, message is discarded
  - Transport-layer example: routers store and forward, but discard if forward is not possible

Synchronous communication…

- Once message has been submitted, sender is blocked until operation is completed
- Send and receive are *blocking* operations
- Places for synchronization:
  1. At request submission
  2. At request delivery
  3. After request processing



1 Synchronize at request submission
2 Synchronize at request delivery
3 Synchronize after processing by server
Client
Request
Transmission interrupt
Storage facility
Reply
Server
Time

… vs synchronous communication

- Once message has been submitted, sender continues its processing: message is temporarily stored by middleware until it is transmitted
- Send is non-blocking, receive can be blocking or non-blocking

Discrete vs. streaming communication

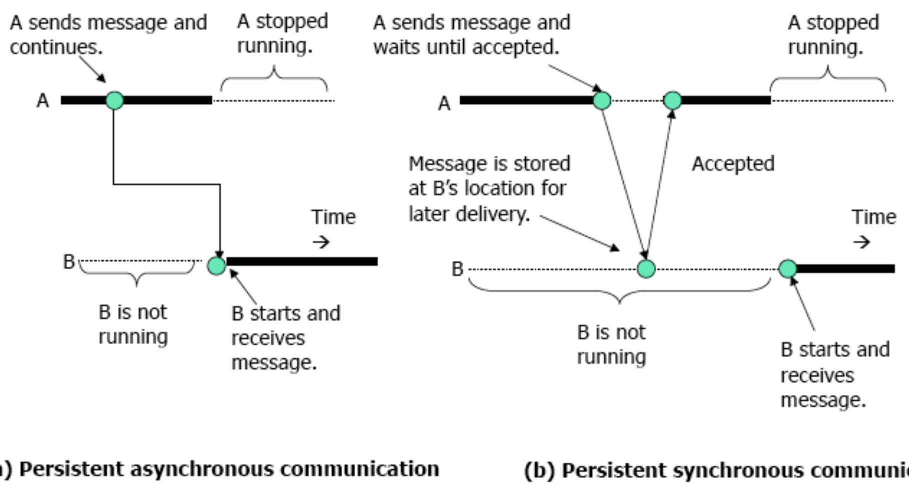
- Discrete communication
  - Each message forms a complete unit of information
- Streaming communication
  - Involves sending multiple messages, in temporal relationship or related to each other by sending order, which is needed to reconstruct complete information
  - E.g., audio, video

Combining communication types

- Combining persistence and synchronization

a) Persistent asynchronous communication
  - E.g., email, Teams chat, message-oriented middleware

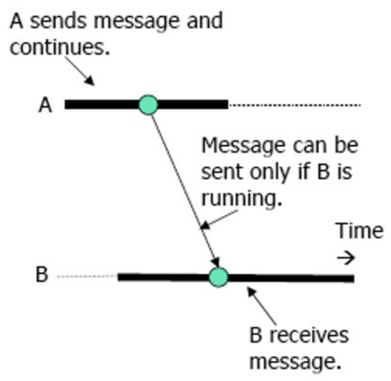
b) Persistent synchronous communication
  - Sender is blocked until message is delivered to receiver


A sends message and continues.
A stopped running.
A
Time →
B
B is not running
B starts and receives message.
(a) Persistent asynchronous communication
A sends message and waits until accepted.
A stopped running.
A
Message is stored at B's location for later delivery.
Accepted
Time →
B
B is not running
B starts and receives message.
(b) Persistent synchronous communication

Combining communication types

- Combining persistence and synchronization

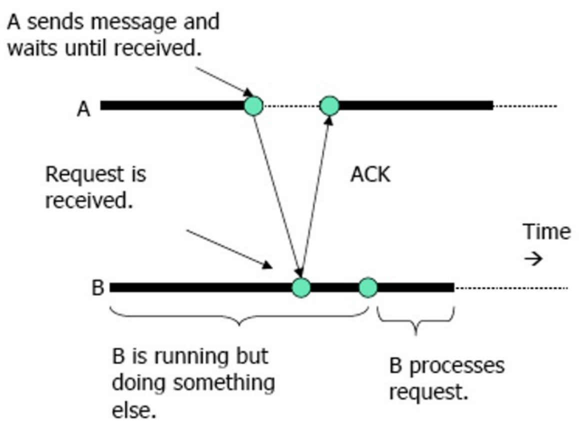
c) Transient asynchronous communication
   - Sender does not wait but message can be lost if receiver is unreachable (e.g., UDP)


A sends message and continues.
A
Message can be sent only if B is running.
Time →
B
B receives message.

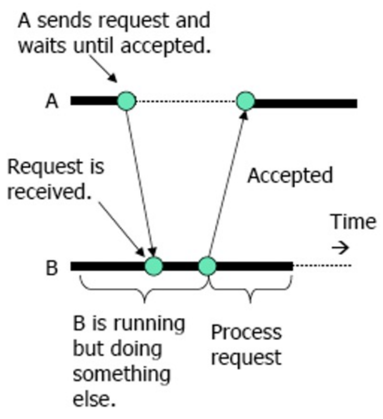
(c)Asynchronous communication

Combining communication types

- Alternatives for transient synchronous communication
  d) Receipt-based synchronous: sender is blocked until message is in receiver space (e.g., asynchronous RPC)
  e) Delivery-based synchronous: sender is blocked until message is delivered to receiver


A sends message and waits until received.
A
Request is received.
ACK
Time →
B
B is running but doing something else.
B processes request.

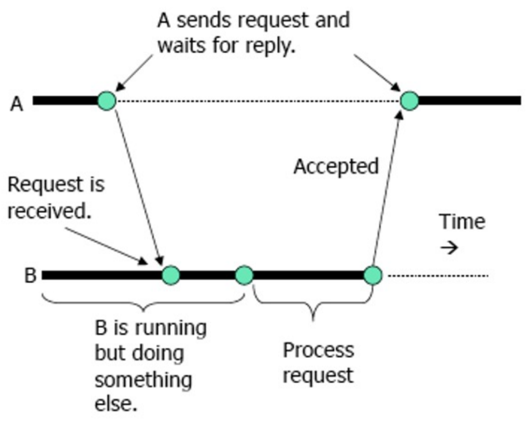
(d) Receipt-based synchronous communication


A sends request and waits until accepted.
A
Request is received.
Accepted
Time →
B
B is running but doing something else.
Process request

(e) Delivery-based synchronous communication

Combining communication types

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

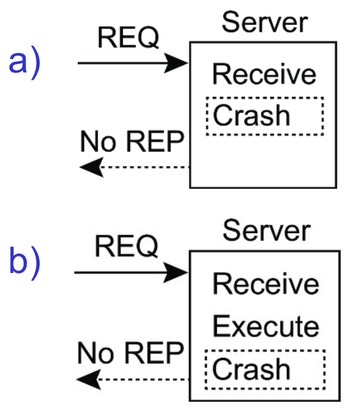

A sends request and waits for reply.
A
Request is received.
Accepted
Time →
B
B is running but doing something else.
Process request

(f) Response-based synchronous communication

Failure semantics during communication

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

1. Request or reply message is lost or delayed, connection is reset
   - *Network is reliable* ("The Eight Fallacies of Distributed Computing")
2. Server crashes
   a) before performing service
   b) after performing service
   - Client cannot distinguish between a) and b)
3. Client crashes after sending request



Failure semantics during communication

- What is the semantics of communication in the presence of failures in a DS?
  - **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)
  - Let's focus on service processing
    - Sender -> client, receiver -> server

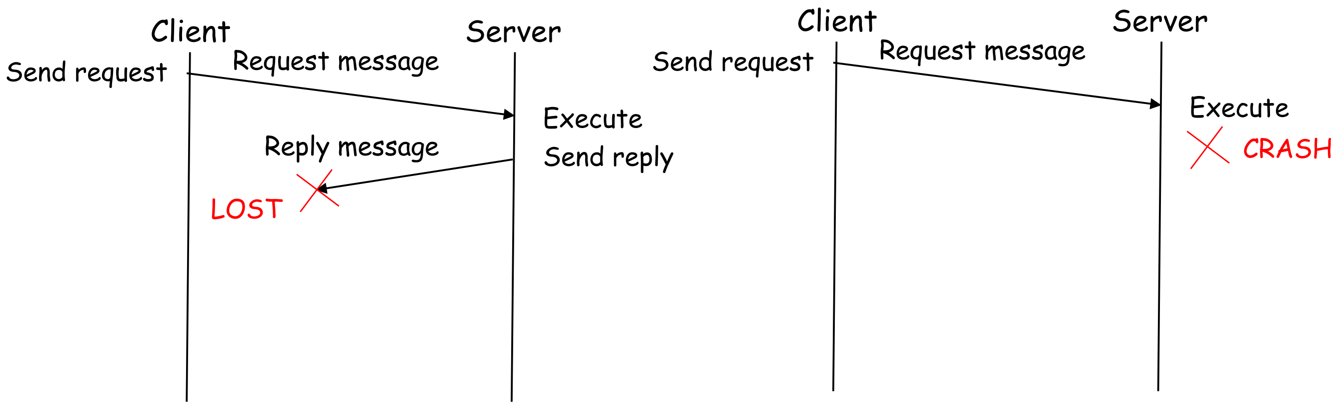
Basic mechanisms for failure semantics

- 3 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 the same client
3. Server side: Result Retransmit (RR2)
   - Server keeps result (response) so that it can retransmit result without processing it again when server receives duplicate request
     - Needed if service performed by server is non-idempotent
   - **Idempotent** service (i.e., without side effects): multiple service executions produce the same effect/result as a single service execution, e.g., read value, set key and value in key-value store

May-be semantics

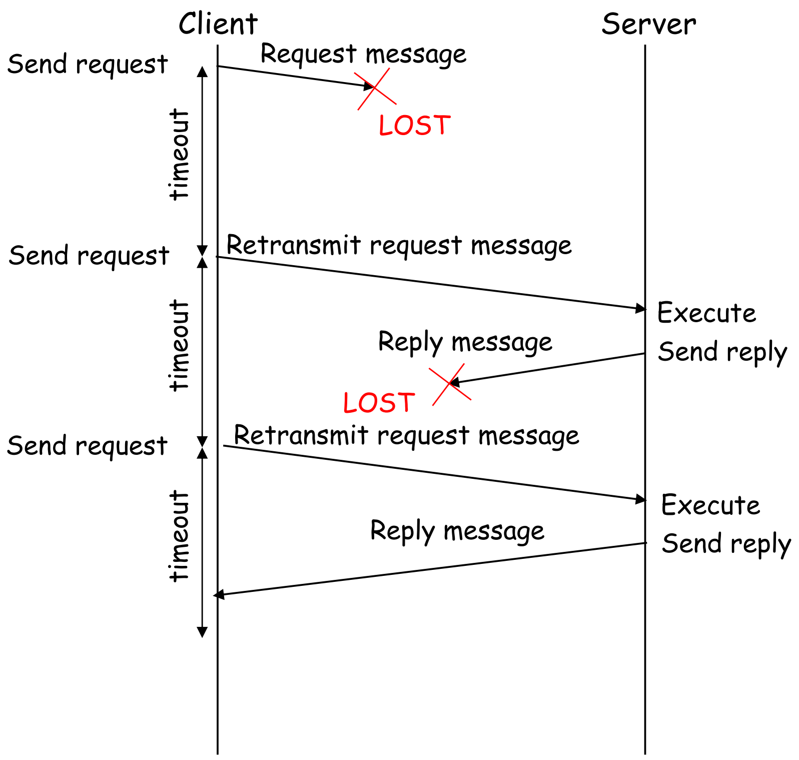
- No guarantee that request has been processed or not on server
- No action is taken to ensure reliable communication: no mechanism (RR1, DF, RR2) is used
- E.g., best-effort in UDP



At-least-once semantics

- Service, if executed, has been executed at least once
  - Could be several times, because of request duplication due to retransmissions
- Client uses RR1, server uses neither DF nor RR2
  - Server is not aware of duplicates
- Suitable for idempotent services
- Upon response receipt, client does not know how many times its request has been processed by server (at-least-once): client does not know about server status
  - Server may have executed the requested service but crashed before sending the response: when timeout expires, client resends the request, server processes it again and sends the response to client

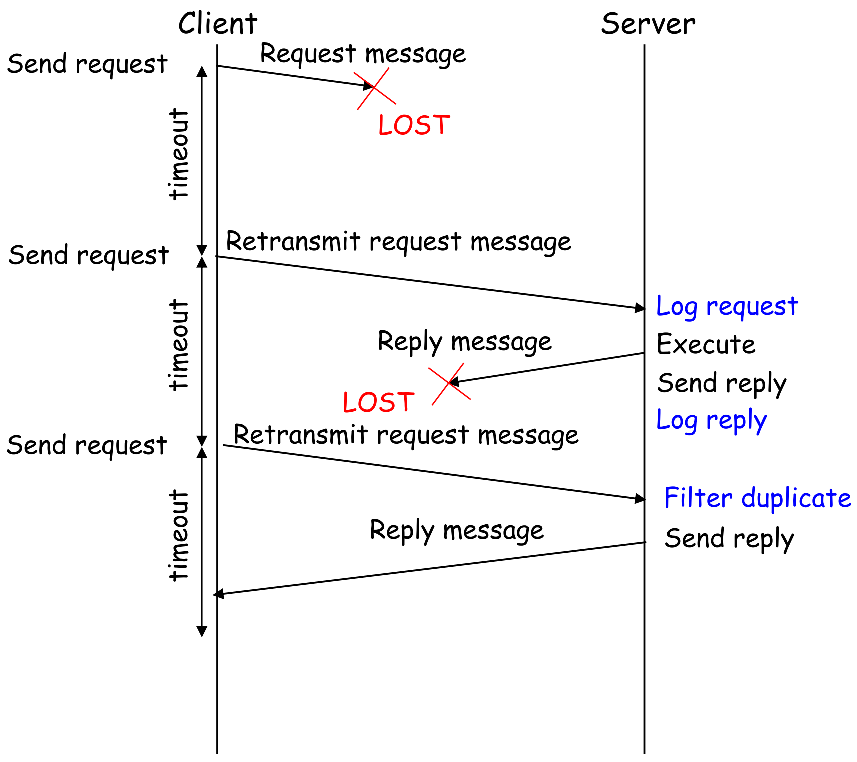
At-least-once semantics



At-most-once semantics

- Service, if performed, is carried out at most once
  - Client knows that, *if it receives the reply*, it has been processed by server *only once*
  - In case of failure, no information (at-most-once: reply has been calculated at most once, but possibly also none)
- All basic mechanisms (RR1, DF, RR2) are used
  - Client retransmits request when timeout expires
  - Server maintains some state to identify duplicate requests and avoid processing the same request more than once
- Suitable also for non-idempotent services
- No constraint on consequent actions
  - No strict coordination between client and server: in case of failure, client does not know if server run the service, while server ignores if client knows that the service run

At-most-once semantics



At-most-once semantics: implementation

- Server detects duplicate requests and returns saved response instead of re-running service `handler()`
- How to detect duplicate request?
  - Client includes a unique ID (`xid`) with each request and uses same `xid` when retransmitting request
- Issues to address
  - How to ensure unique `xid`?
  - Server must eventually discard info about saved responses: when is discard safe?
    - Use sliding windows and sequence numbers
    - Discard information older than maximum message lifetime
  - How to handle duplicate requests while original one is still executing?

Server:
```
if seen[xid]
 r = old[xid]
else
  r = handler()
  old[xid] = r
  seen[xid] = true
```

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

Exactly-once semantics

- Strongest but most complex 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!

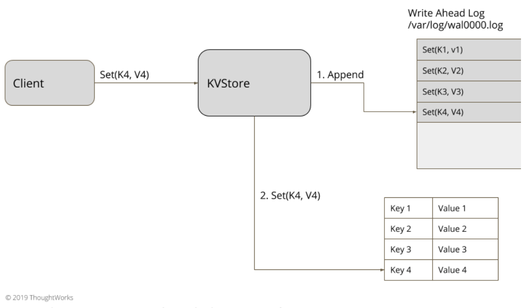
Exactly-once semantics: mechanisms

- Server-side basic mechanisms (RR1, DF, RR2) are not enough
- Need more mechanisms to tolerate server-side faults
  - Transparent server replication
  - Write-ahead logging
  - Recovery
    - Mechanisms to recover from whatever state failed server left behind and begin processing from a safe point
    - *Distributed snapshot*: captures a consistent global state of DS
    - *State checkpointing*: saves a snapshot of DS state

Write-ahead logging (WAL) pattern

- aka Commit log
- Goal
  - Provide durability guarantee by persisting every state change as command to append-only log
- How
  - Each state change is stored as log entry in a file on disk and log is appended sequentially
  - File can be read on every restart and state can be recovered by replaying all log entries

WAL pattern martinfowler.com/articles/patterns-of-distributed-systems/wal.html



```
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());
}
```

Summing up failure semantics

- At-least once and at-most once semantics are feasible and widely used in DS
- We often choose the lesser of two evils, i.e., at-least-once semantics
  - At-least once is also easier to scale

Distributed systems are all about trade-offs!

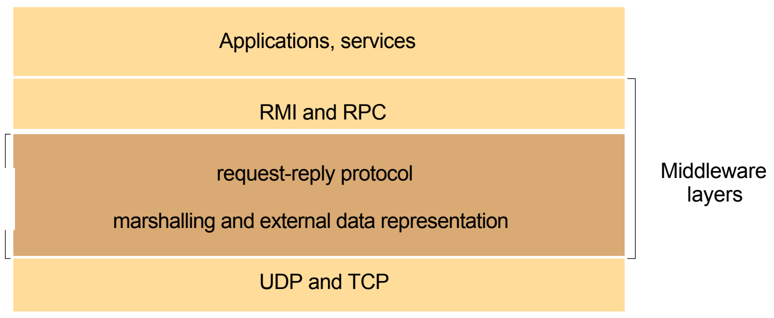
Distributed application programming

- You know explicit network programming
  - Operating system construct based on socket API and explicit management of message exchange
  - Used in most network applications (e.g. web browser, web server)
  - But distribution is not transparent and requires developer effort

- How to increase the abstraction level of distributed programming? By means of communication middleware between OS and applications
  - Hide complexity of underlying hw and sw layers
  - Free programmer from automatable tasks
  - Improve software quality by reusing known, correct and efficient solutions

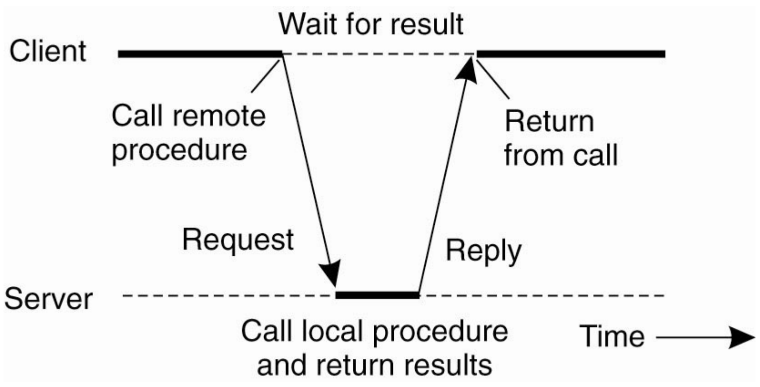
Distributed application programming

- Implicit network programming
  - Language-level construct
  - **Remote Procedure Call (RPC)**
    - Distributed app is realized through procedure calls, but caller (client) and callee (server) are located on remote machines and communication details are hidden to programmer
  - **Remote method invocation (Java RMI)**
    - Distributed app in Java is realized by invoking object methods running on remote machines



Remote Procedure Call (RPC)

- Idea (by Birrel and Nelson, 1984): use client/server model to call procedures executed on other machines
  - Process on machine A calls procedure on machine B
  - Calling process on A is suspended
  - Called procedure is execute on B
  - Input and output parameters are transported into messages
  - No message passing is visible to programmer



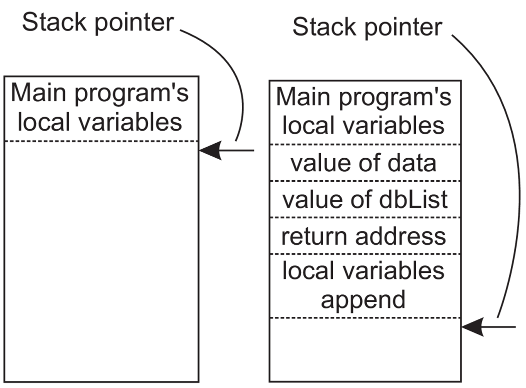
Why RPC

- Used in many distributed systems, including cloud computing ones
- Developed and employed in many languages and frameworks, among which:
  - **C (Sun RCP)**
  - **Java (Java RMI)**
  - **Python (RPyC)**
  - **Go**
  - **gRPC**
  - Distributed Ruby (DRb)
  - Ice
  - Microsoft .NET
  - JSON-RPC

Our case studies

Local procedure call

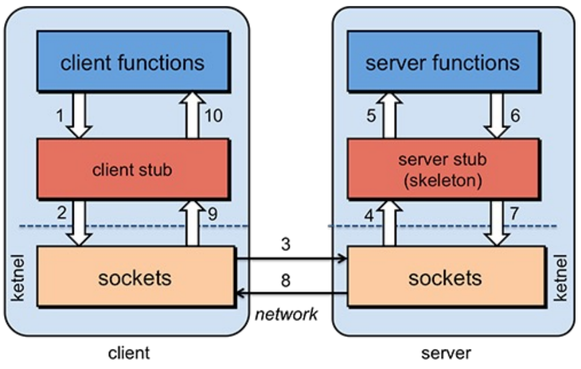
- Example of local procedure call:
  `newlist = append(data, dbList)`
- Caller pushes to stack input parameters (`data`, `dbList`) and returns address
- When callee returns, control is back to caller



How to make RPC look as much as possible as a local procedure call?

RPC: architecture

- Solution: create proxies (aka **stubs**)
- Client side: **client stub** exposes service's interface
  - Client calls client stub that manages all the details
- Server side: **server stub** receives request and calls local procedure
- Goal: distribution transparency
  - Stubs are automatically generated
  - Developer focuses on application logic

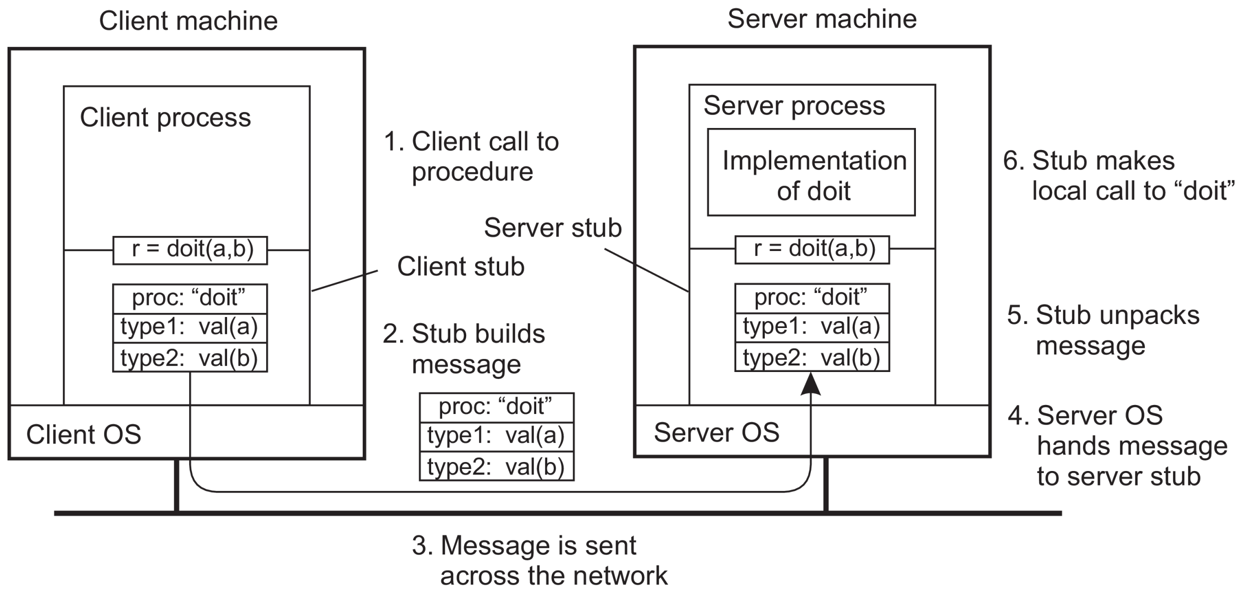


RPC: basic steps

1. On client side, client calls a local procedure, called client stub
2. Client stub packs request message and call local OS
   - Parameter marshaling: arguments are converted from local to common format and packaged into a message
3. Client OS sends request message to remote OS
4. Remote OS delivers request message to server stub
5. On server side, server stub unpacks request message and calls server as it was a local procedure
   - Parameter unmarshaling: arguments are extracted from message and converted from common to local format
6. Server executes local call and returns result to server stub
7. Server stubs packs reply message (marshals return value(s)) and calls OS
8. Server OS sends reply message to client OS
9. Client OS delivers message to client stub
10. Client stub unpacks reply message (unmarshals return value(s)) and returns result to client

RPC: example

- RPC `doit(a,b)`



RPC: what is needed

- Exchange messages, so to make it appear to programmer that procedure call is local; we need to:
  - Identify request and reply messages, remote procedure
  - Pass parameters
- Manage data heterogeneity
  - Which data? Parameters, return value(s)
  - Marshaling vs. serialization:
    - ***Marshaling***: bundle parameters into a form that can be reconstructed (unmarshaled) by another process
    - ***Serialization***: convert object into a sequence of bytes that can be sent over a network; serialization is used in marshaling
- Handle failures due to distribution
  - During communication
  - User errors

RPC: issues

- Main issues to address to make remote procedure call as simple and straightforward as local procedure call

1. Manage heterogeneity in data representation
   - Client and server also need to agree on transport protocol for message passing: TPC, UDP, both?
2. Perform parameter passing by reference
   - Client and server run on different machines with their own address space
3. Define failure semantics
   - Local procedure call: exactly-once
   - Remote procedure call: at-least-once or at-most-once (in most cases)
4. Bind client to server, i.e., locate server endpoint

Data heterogeneity

- Client and server may use different data representations
  - E.g., byte ordering (little endian vs big endian), data size, padding, …
  - RPC needs to define the details of how RPC messages are sent on the wire

- Alternatives (*general*, not only RPC) to handle heterogeneity in data representation:
  1. Specify encoding within message itself
  2. Let sender convert data into receiver encoding
  3. Convert data into common encoding agreed between parties
     - Sender: converts from local to common
     - Receiver: converts from common to local
  4. Let an intermediary convert between different encodings

Data heterogeneity

- Let's compare alternatives #2 and #3, assuming *N* distributed components
- #2: each component knows all conversion functions
  - ✓ Conversion is faster
  - ✗ Higher number of conversion functions: *N**(*N*-1)
- #3: all components agree on common encoding for data representation and each component knows how to convert from local to common format and vice versa
  - ✗ Conversion is slower
  - ✓ Lower number of conversion functions: 2*(*N*-1)
- Alternative 3: standard choice in RPC systems

Data heterogeneity: patterns

- Patterns to implement alternatives #3 and #4
- Proxy
  - Goal: support access (and location) transparency
  - Manage access to an object using another proxy object
    - Proxy is created in local address space to represent remote object and exposes same interface of remote object
- Broker
  - Goal: separate and encapsulate communication details from its functionality
  - Enable components to interact without handling remote concerns by themselves
  - Locate server for client, hide communication details, etc.
- Proxy (aka stub): standard choice in RPC systems
  - Who automatically generates stubs?

Parameter passing techniques

- Call by value
  - Parameter value is copied in a local isolated storage (usually stack)
  - Callee acts on copied data and changes will not affect caller
- Call by reference
  - Reference (pointer) to parameter is copied into stack
  - Callee acts directly on caller data
- Call by copy-restore
  - A somehow special case of call by reference: data is copied into caller stack; when procedure returns, updated content is copied back (restored)
  - Available in few programming languages (e.g., Ada, Fortran)

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 by using call by copy-restore
  - Client stub copies the pointed data in the request message and sends the message to server stub
  - Server stub acts on copy, using the address space of the receiver host
  - If the copy is modified, it will be then restored by client stub overwriting the original data
  - Size of data to be copied should be known
  - What happens if data contains a pointer?

Semantics of remote call/method

- Exactly once semantics is costly: most RPC systems implement weaker semantics
- At-least-once semantics: if client receives reply from server, then remote call has been executed at least once by server
- At-most-once semantics: if client receives reply from server, then remote call/method has been executed at most once by server

Server binding

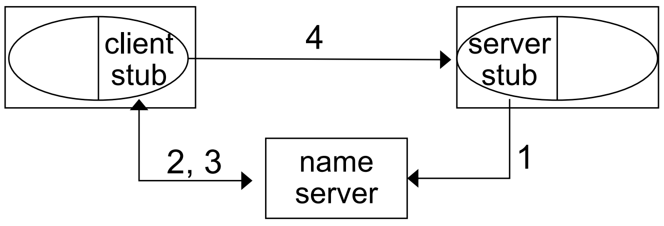
- Binding: how to locate the server endpoint, including the proper process (port or transport address) on it
  - In principle: can be static or dynamic
- Static binding
  - Binding is known at design time: server address and other info (e.g., port) are hard-coded
  - Easy and no overhead, but lacks transparency and flexibility
- Dynamic binding
  - At run-time
  - Increased overhead, but gains transparency and flexibility
    - E.g., we can redirect requests in case of server replication
  - Try to limit overhead

Server binding: dynamic

- Two phases in client/server relationship
- **Naming**: static phase *before* execution
  - Client specifies to whom it wants to be connected, using a unique name that identifies the service
  - Unique names are associated with operations or abstract interfaces and binding is made to the specific service interface
- **Addressing**: dynamic phase *during* execution
  - Server effectively binds to client when client invokes service
  - Depending on middleware implementation, multiple replica servers can be looked for
  - Addressing can be *explicit* or *implicit*
  - **Explicit addressing**: client sends request using broadcast or multicast, waiting only for first reply

Server binding: dynamic

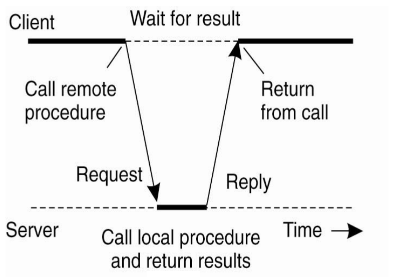
- **Implicit addressing**: there is a **name server** (aka binder, directory service, registry service) that registers services and manages a binding table
  - Service lookup, registration, update, and deletion



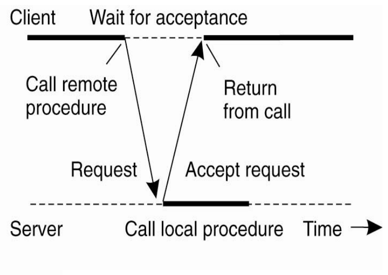
- Dynamic binding frequency
  - Each procedure call requires addressing
  - To reduce cost, binding result can be cached and re-used

More issues: Synchronous vs. asynchronous RPC

- Synchronous RPC: strict request-reply behavior
  - RPC call blocks client that waits for server reply
- Some RPC middleware supports asynchronous RPC
  - Client continues without waiting for server reply
  - Server can reply as soon as request is received and execute procedure later



Synchronous RPC



Asynchronous RPC

More issues: transparency

- Is RPC truly transparent? Can we really just treat remote procedure calls as local procedure calls?
  - Performance, failures, concurrent requests, replication, migration, …

- Performance
  - RPC is slower ... a lot slower: why?
  - Local call: maybe 10 cycles = ~3 ns
  - RPC: 0.1-1 ms on a LAN => ~100K slower
    - Major source of overhead: context switching, copies, inter-process communication
    - In WAN: can easily be millions of times slower

More issues: transparency

- Failures
  - Different failures can occur
    - Client cannot locate server
    - Lost request messages
    - Server crashes
    - Lost reply messages
    - Client crashes

More issues: security

- Authenticate client? Authenticate server?
  - Is client sending messages to correct server or to impostor?
  - Is server accepting messages only from legitimate clients? Can server identify user at client side?
- Messages may be visible over network
  - Messages may be sniffed (and modified) while they traverse the network: can we encrypt them?
  - Have messages been accidentally corrupted or truncated while on network?
- RPC protocol may be subject to replay attacks
  - Can a malicious host capture a message and retransmit it at a later time?

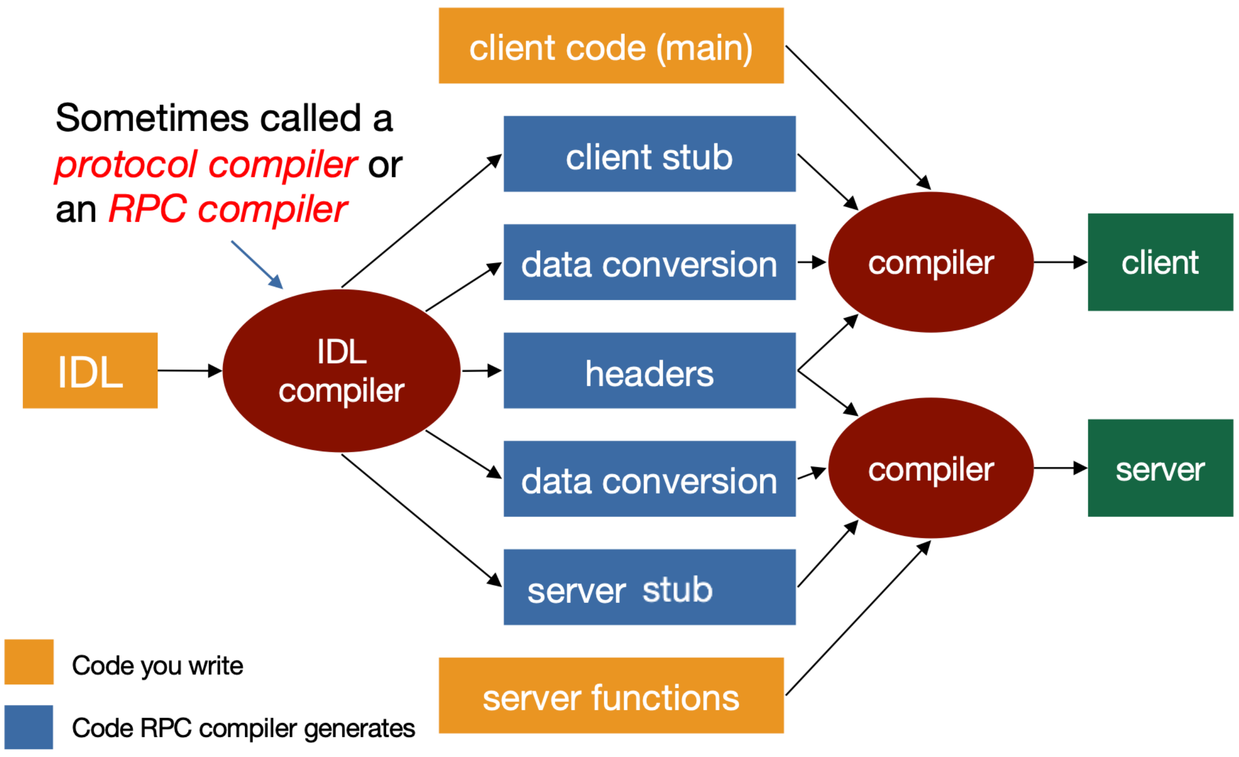
Programming with RPC

- Language support
  - Some programming languages have no language-level concept of remote procedure calls (e.g., C, C++)
    - Their compilers will not automatically generate stubs
  - Some languages directly support RPC (Java, Python, Haskell, Go, Erlang)
    - But we may need to deal with heterogeneous environments (e.g., Java service communicating with Python service)
- Common solution
  - **Interface Definition Language (IDL)**: describes remote procedures
  - Separate compiler that generates stubs (pre-compiler)

Interface Definition Language (IDL)

- Allow programmer to specify remote procedure interfaces (*names, parameters, return values*)
- IDL compiler can use this to generate client and server stubs
  - Marshaling code
  - Unmarshaling code
  - Network transport code
- Conform to defined interface
  - An IDL looks similar to function prototypes

IDL and RPC compiler



RPC middleware: case studies

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

RPC implementation: Sun RPC

- First-generation RPC
- Created by Sun (now Oracle): Sun RPC
  - RFC 1831 (1995), RFC 5531 (2009)
  - Remains in use mostly because of NFS (Network File System)
- Interfaces defined in an IDL called XDR

Sun RPC: XDR

- Sun RPC uses **XDR** (eXternal Data Representation) as IDL to address data heterogeneity
  - Standard to describe and encode machine-independent data (RFC 4506)
  - IDL compiler is `rpcgen`
- XDR provides built-in conversion functions for:
  - Predefined primitive types, e.g., `xdr_int()`
  - Predefined structured types, e.g., `xdr_string()`
- XDR is a binary format using *implicit typing*
  - *Implicit typing*: only values are transmitted, not data types or parameter info

Sun RPC: define RPC program using XDR

- Two descriptive parts written in XDR and grouped in a file with extension .x
  1. Definition: specifics of procedures (services) to identify procedures and their parameters' data types
  2. XDR definitions: definitions of parameters' data types (if not built-in)
- Our Sun RPC example: calculate square of integer number

Example: 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`
  - Each procedure has only one input parameter and one output parameter
  - Identifiers are written in uppercase
  - Each procedure is associated with a procedure number which is unique within RPC program (e.g., 1)

Sun RPC: how to implement RPC program

- Programmer develops:
  - **Client program**: implements `main()` and logic needed to find remote procedure and bind to it (example: square_client.c)
  - **Server program**: implements remote procedures provided by RPC server (example: square_server.c)

- Note: programmer does not write server-side `main()`
  - Who calls remote procedure on server side?

Example: local procedure

- Let’s first consider standard local procedure

square_local.c

```
#include <stdio.h>
#include <stdlib.h>

struct square_in {  /* input (argument) */
  long arg;
};
struct square_out {  /* output (result) */
  long res;
};
typedef struct square_in square_in;
typedef struct square_out square_out;

square_out *squareproc(square_in *inp) {
  static square_out out;

  out.res = inp->arg * inp->arg;
  return(&out);
}
```

Example: local procedure

- Local procedure *(continue)*

```
int main(int argc, char **argv) {
  square_in in;
  square_out *outp;

  if (argc != 2) {
    printf("usage: %s <integer-value>\n", argv[0]);
    exit(1);
  }
  in.arg = atol(argv[1]);

  outp = squareproc(&in);
  printf("result: %ld\n", outp->res);
  exit(0);
}
```

Which changes in case of remote procedure?

Example: remote procedure

- Remote procedure is *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);
}
```

- Notes:
  – Input and output parameters use pointers
  – Output parameter must be pointer to static variable (i.e., global memory allocation) so that pointed area exists when procedure returns
  – Name of RPC procedure changes slightly (add _ suffixed by version number and _svc, e.g., _1_svc, all in lowercase)

Example: client

- Run client with remote hostname and integer value; it calls remote procedure

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)
```

Example: 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);
}
```

Example: client

- `clnt_create()`: creates client transport manager to handle communication with remote server
  - TPC or UDP, default timeout for request retransmission
- Client must know:
  - Remote server hostname
  - Info to call remote procedure: program name (`SQUARE_PROG`), version number (1) and procedure name (`square_proc`)
- To call remote procedure:
  - Procedure name changes slightly: add _ followed by version number and write name in lowercase
  - Two input parameters:
    - Effective input parameter plus client transport manager
  - Client gets *pointer* to result
    - To identify failed RPC: `NULL` return
- Handling of failures that may occur during remote call
  - `clnt_pcreateerror()` and `clnt_perror()`

Sun RPC: features

- RPC program contains multiple remote procedures
  - Versioning support
  - Each procedure has one input and one output parameter
  - Call by copy-restore
- Transport independent
  - Transport protocol can be selected at run-time
- Mutual exclusion guaranteed by server
  - Default: no concurrency on server side
- Synchronous client: blocked until server replies
- At-least-once semantics
  - Request retransmission when timeout expires
- Security? Authentication mechanism added later with Secure RPC
  - Uses DES encryption

Sun RPC: server binding

- Client stub needs to know port number: how?
- Server stub registers RPC program
  - Each procedure is identified by: program number, procedure number, version number
- Where? In *port map*
  - Dynamic table of RPC services on that host machine
  - port map is managed by *port mapper* (`rpcbind`): one per host, listens on port 111
  - Client stub contacts port mapper to find out port number and then sends request message to server stub

Port mapper (rpcbind)

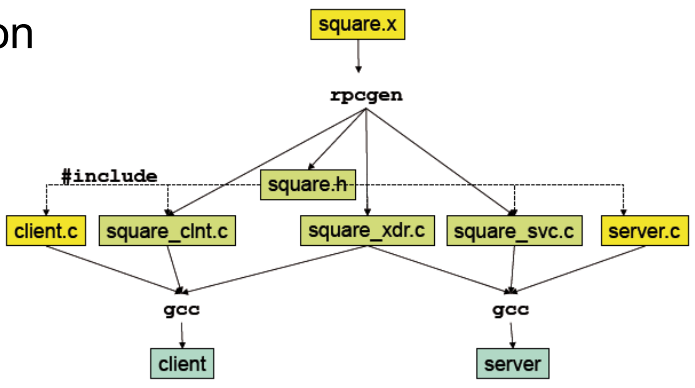
- List RPC programs on a given host

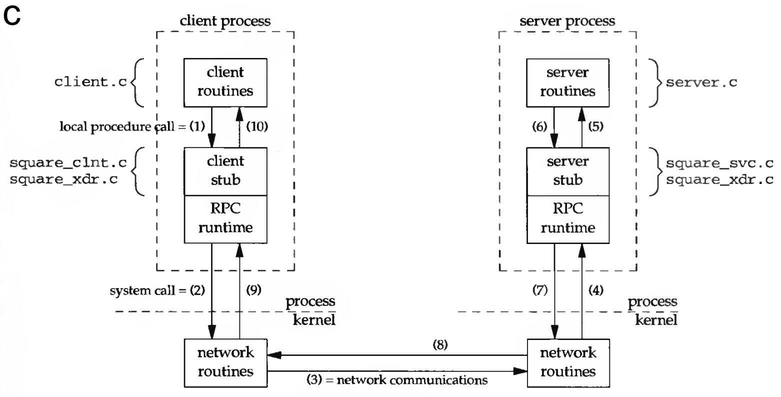
```
>$ rpcinfo -p
 program        vers    proto   port
   100000       4       tcp     111      rpcbind
   100000       4       udp     111      rpcbind
   824377344    1       udp     59528
   824377344    1       tcp     49311
```

Our RPC program `SQUAREPROC`
- 824377344 (= 0x31230000) is the program number in square.x
- Server supports both TCP and UDP: transport-protocol independent

SUN RPC: Development process

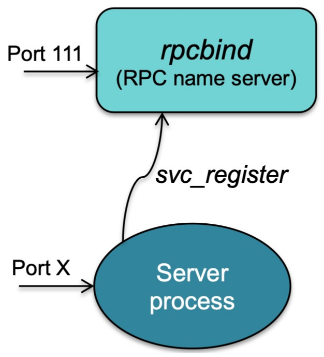
1. Define service specification
   - Written in XDR: square.x
2. Use `rpcgen` to generate
   - header: square.h
   - client stub: square_clnt.c
   - server stub: square_svc.c
   - XDR routines: square_xdr.c
3. Write
   - client program: client.c
   - server program: server.c





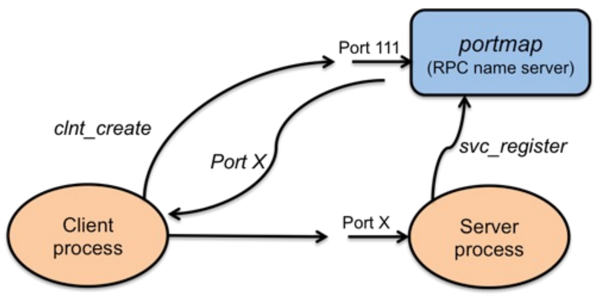
What goes on: server side

- Let's analyze server stub code (square_svc.c)
- In `main()` server stub creates a socket and binds any available local port to it
- Calls `svc_register` (RPC library function)
  - To register procedures with port mapper
    - Associates the specified program and version number pair with the specified dispatch routine
- Then waits for requests by calling `svc_run` (RPC library)
  - `svc_run` invokes specific service procedures in response to RPC call messages



What goes on: client side

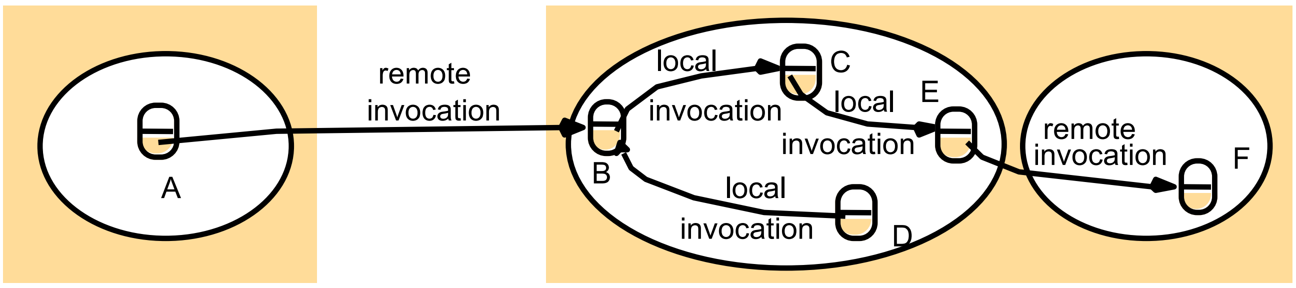
- When we start client program, `clnt_create` contacts port mapper on server side to find port for that interface
  - Early binding: done once, not per each procedure call



- Client stub (square_clnt.c) manages communication
  - Request timeout
  - Marshaling from local representation to XDR format and unmarshaling from XDR format to local representation

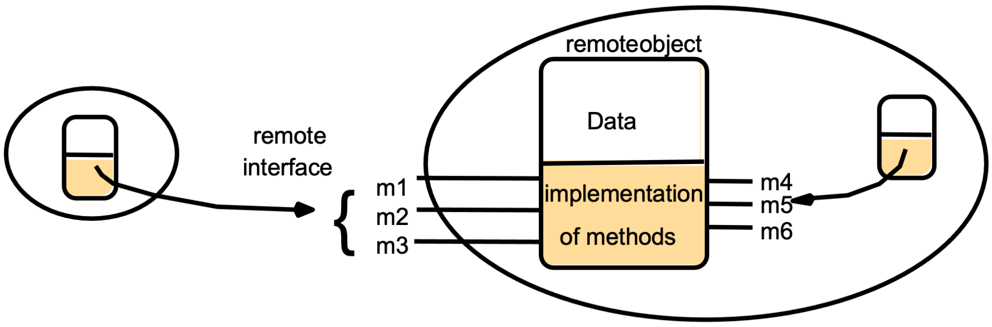
Java RMI

- Second-generation RPC
- **Java RMI** (Remote Method Invocation): RPC in Java
- Extends RPC to distributed objects
  - Allows to develop distributed applications in Java where an object on one JVM invokes methods on an object in another JVM
  - Goal: access transparency, but distribution transparency is still not full



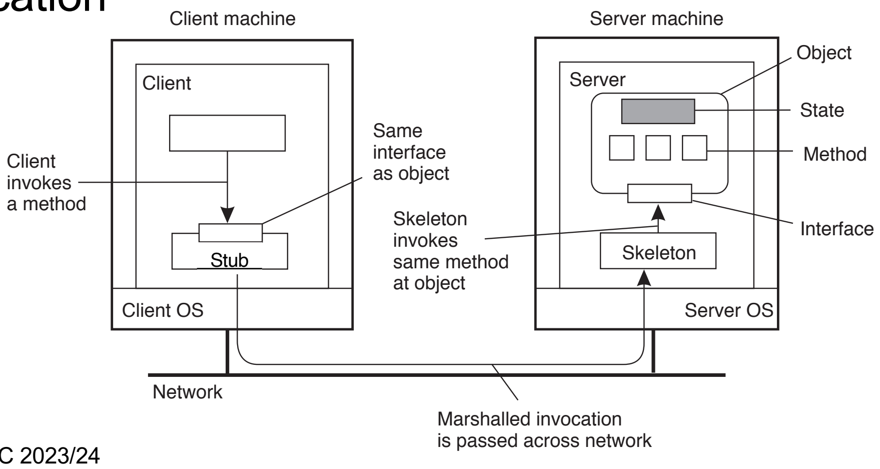
Java RMI: basics

- Recall Java separation between definition (interface) and implementation (class)
- Idea: logical separation between interface and object allows for their physical separation
- *Remote interface*: specifies set of methods to be invoked remotely
- *Remote object*: instance of a class that implements a remote interface
  - But internal state of remote object is not distributed!



Java RMI: basics

- *Remote method invocation*: invoke methods of a remote interface on a remote object
  - Goal: keep same syntax as local invocation
  - How to achieve it?
- Once again, proxy pattern: client-side ***stub*** and server-side ***skeleton*** to hide distributed nature of application



Java RMI: Serialization/deserialization

- Serialization/deserialization directly supported by Java
  - Thanks to Java bytecode, no need to (un)marshal, but data is (de)serialized using language-level features
- Serialization: converts object that is passed as parameter into byte stream
  - `writeObject` on output stream
- Deserialization: decodes byte stream and builds copy of original object
  - `readObject` from input stream
- Stub and skeleton use serialization/deserialization to exchange input and output parameters between different JVMs

Marshaling vs serialization

- Loosely synonymous but semantically different
- Marshaling: stub converts local data into network data (using encoding/deconding routines) and packages network data for transmission
- Serialization: object state is converted into byte stream, which can be converted back into object copy
- Difference becomes noticeable for objects
  - Objects have a codebase that also needs to be marshaled
  - Java serialization relies on codebase being present at receiver
- In Python (`Pickle` module) marshaling and serialization are considered the same, but not in Java

Java RMI: stub and skeleton interaction

- Client obtains stub instance through RMI registry
  - Naming service for Java RMI that maps names to remote objects
- Client invokes remote method on stub
  - Remote invocation syntax is identical to local one
- Stub serializes data needed to invoke method (method’s ID and input parameters) and sends them to skeleton in a message
- Skeleton receives message, deserializes received data, invokes method, serializes return value and sends it to stub in a message
- Stub receives message, deserializes return value and returns it to client

Example: echo remote interface

- Remote interface extends **`Remote`**
  - **`Remote`** identifies interfaces whose methods may be invoked from a non-local JVM
- Remote method
  - throws **`RemoteException`**
    - To handle communication failure or protocol error
    - Remote method invocation is not fully transparent
  - passes parameters
    - by value in case of primitive data types (int, char, …) or objects that implement **`java.io.Serializable`** interface: serialization/deserialization managed by stub/skeleton
    - by reference in case of **`Remote`** objects

```
import java.rmi.Remote;
import java.rmi.RemoteException;

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

Example: echo server

- Class implements remote interface
  - Extends **`UnicastRemoteObject`**
  - **`super()`** calls class constructor **`UnicastRemoteObject`** which executes all is needed to allow server to wait for requests and serve them
  - Implements method, which throws remote exception
- In **`main`** server object instance is created
- Server registers by name its remote objects with RMI registry
  - `java.rmi.Naming` class provides methods to look up, bind, rebind, unbind, and list the name-object pairings maintained on a host
  - RMI registry and server must run on same host

```
public class EchoRMIServer
  extends UnicastRemoteObject
  implements EchoInterface{
  // Constructor
  public EchoRMIServer()
   throws RemoteException
  { super(); }
  // Implement the remote method declared
  // in the interface
  public String getEcho(String echo)
    throws RemoteException
   { return echo; }
  public static void main(String[] args) {
    // Service registration
  try
  { EchoRMIServer serverRMI =
         new EchoRMIServer();
Naming.rebind("EchoService", serverRMI); }
   catch (Exception e)
  {e.printStackTrace(); System.exit(1); }
 }
}
```

Example: echo client

- Service is used through interface variable, obtained through **lookup** on RMI registry
- Client contacts RMI registry to look up the remote object using its name
  - The registry returns a reference to the remote object
- Client invokes remote method
  - Synchronous blocking call

```
public class EchoRMIClient
{
  // Start RMI client
  public static void main(String[] args)
 {
   bufferedReader stdIn =
     new BufferedReader(
       new InputStreamReader(System.in));
  try
  {
    // Connect to remote RMI service
    EchoInterface serverRMI = (EchoInterface)
         Naming.lookup("EchoService");
    // Interact with user
    String message, echo;
    System.out.print("Message? ");
    message = stdIn.readLine();
    // Invoke remote service
    echo = serverRMI.getEcho(message);
    System.out.println("Echo: "+echo+"\n");
  }
  catch (Exception e)
  {e.printStackTrace(); System.exit(1); }
 }
}
```

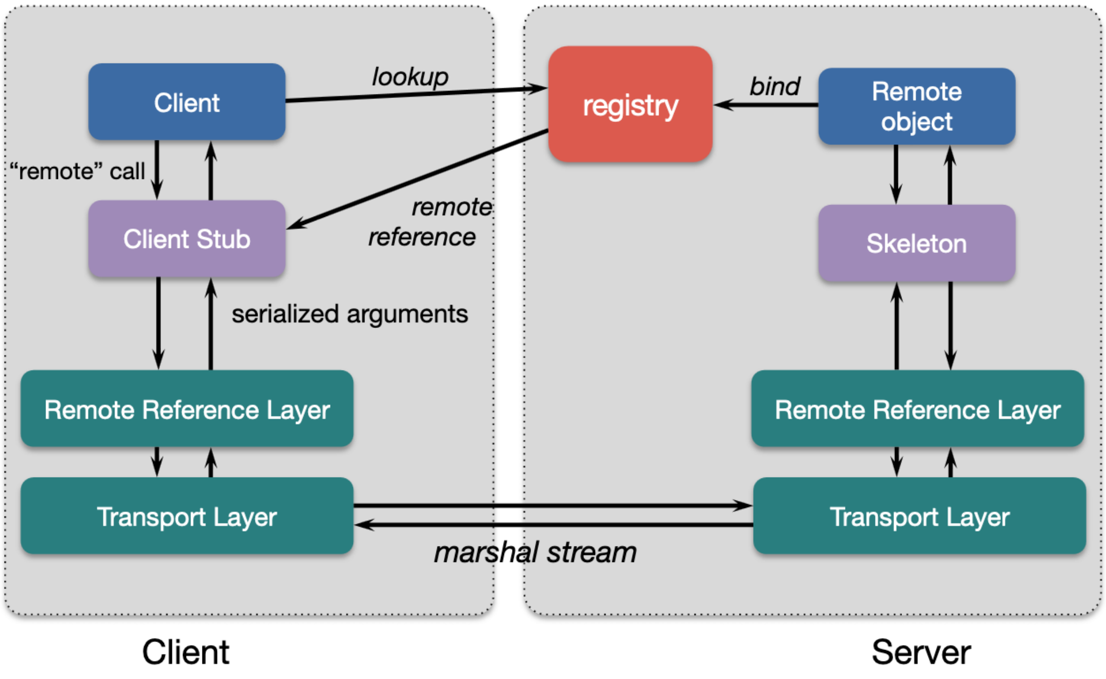
Java RMI: other features

- Synchronous blocking client
- At-least-once semantics
- Concurrency: a remote method can be invoked concurrently by multiple clients

  From Java RMI specification: "*Since remote method invocation on the same remote object may execute concurrently, a remote object implementation needs to make sure its implementation is thread-safe*"
  docs.oracle.com/en/java/javase/21/docs/specs/rmi/arch.html
  - To protect a remote method from concurrent accesses while guaranteeing thread safety, define it as `synchronized`
- Dynamic class loading: since class definitions are required for serializing/deserializing objects passed as parameters, RMI also provides a facility for dynamically loading class definitions

Java RMI: infrastructure

- RMI system consists of 3 layers: *stub/skeleton* layer, *remote reference* layer, and *transport* layer



RPC in Python

- Various implementations: PyRO, PyInvoke, RPyC, ZeroRPC
- What helps Python achieve transparency
  – Inspection of live objects through the `inspect` module
    - Examine the contents of a class, retrieve source code for a method, and extract the argument list for a function
- General idea of implementing RPC in Python
  – Create a connection using an RPC object
  – Then invoke remote methods using that object
- Let's analyze RPyC (Remote Python Call)

RPyC features

- Transparent RPC interface
  - No definition file, IDL compiler, name server, transport service
- Symmetric operation
  - Both sides can invoke RPCs on each other – enables callback functions
- Server
  - RPyC ThreadedServer started on server
    - Besides Threaded, RPyC supports also thread pool, forking and one shot server
  - Binds to a default port (18812, 18821 for SSL) or you specify host's IP address and port
- Client
  - Connects to server
  - Performs remote operations through the `modules` property, which exposes the server module's namespace

Serialization in RPyC: passing data

- By value
  - Simple types (immutable objects: string, int, tuple)
    - Sent directly to remote side
- By reference
  - Objects: reference (object name) to an object is passed
    - Remote contacts the client to access attributes and invoke methods on these objects
    - Changes will be reflected onto actual object
  - Enables passing of location-sensitive objects, like files or other OS resources
    - Remote process can write to `stdout` of a local process by getting its `sys.stdout`
  - Implementation: netrefs = transparent object proxies
    - Local objects that forward all operations to the corresponding remote object
    - They make remote objects look and feel like local objects

RPyC: stubs

- Client creates local proxy objects for remote modules
  - Allows for transparent access
  - Reference wrapped in a special object called a proxy that looks like the actual object
  - Any operation on the proxy is delivered to the target
  - Client is unaware of this
- Synchronous and asynchronous calls
  - Synchronous: client waits for a return
  - Asynchronous: immediate return, notification when complete
  - Calls can be made asynchronous by wrapping the proxy with an asynchronous wrapper

RPyC: services and security

- RPyC is built around services
  - Each end of the connection exposes a service that is responsible for the policy
  - Policy = set of supported remote operations

- Services are classes that derive from `rpyc.core.service.Service` and define exposed methods
  - Methods whose names begin with `exposed_` or use `@rpyc.exposed` decorator
  - All exposed members of a service class will be available to the other side

RPyC example: server

```
import rpyc

class CalculatorService(rpyc.Service):
    def exposed_add(self, a, b):
        return a + b
    def exposed_sub(self, a, b):
        return a - b
    def exposed_mul(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):
        print("foo")

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

RPyC example: client

```
import rpyc

conn = rpyc.connect("localhost", 18861)
x = conn.root.add(4,7)
assert x == 11
print(conn.root.fib(1000))
# print(conn.root.div(1,0))
```

To the remote party, the service is exposed as the root object of the connection (`conn.root`).
This root object is a network reference (`netref`) to the service instance living in the server process

RPyC: async operation

- Asynchronous operation is a key feature of RPyC
  - Client starts the request and continues rather than blocking
  - It gets an `AsyncResult` object that will *eventually* hold the result
- Asynchronous behavior must be explicitly enabled: to turn the invocation of a remote method (or any callable object) asynchronous, wrap it with `async_()`
- Then, the client can
  - test an `AsyncResult` object for completion using `ready`
  - wait for completion using `wait()`
  - get the result using `value`
  - set a timeout for the result using `set_expiry()`
  - register a callback function to be invoked when the result arrives using `add_callback()`
- No guarantee on execution order for async requests

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

RPyC: async operation and events

- Events can be implemented as asynchronous callbacks
  - Server produces an event which is consumed by client
- Let's analyze FileMonitor example
  - Server periodically monitors a file using os.stat() for detecting file change
  - Server will send event to client (invoke an async callback) whenever a file is changed (providing old and new `stat` results)

RPC in Go

- Let’s introduce Go programming language
  www.ce.uniroma2.it/courses/sdcc2324/slides/Go.pdf

- What about RPC in Go?

RPC in Go

- Go standard library supports RPC right out-of-the-box
  - Package `net/rpc`
  - Provides access to the exported methods of an object across a network
- TCP or HTTP as “transport” protocols
- Requirements for server RPC methods
  - Method type and method are exported (capital letter)
  - Only two arguments, both exported
  - Second argument is a pointer to a reply struct that stores the corresponding data
  - An error is always returned

```
func (t *T) MethodName(argType T1, replyType *T2) error
```

RPC in Go: server

- On server side
  - Create a TPC server (or an HTTP server) to receive data
  - Use **Register** (or **RegisterName**): register an object, making it visible as a service

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

    - One input parameter, which is the interface: `any` is an alias for `interface{}`
    - It publishes the methods that are part of the given interface on the RPC server and allows them to be called by clients connecting to the service

  - Use **Listen** to announce on the local network address

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

RPC in Go: server

  - Use **Accept** to accept connections on the listener and serve requests for each incoming connection

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

    - `Accept` blocks; if the server wishes to do other work as well, it should call this in a goroutine (`go` statement)

  - Can also use HTTP handler for RPC messages
    - See example on course site

RPC in Go: client

- On client side
  - Use **Dial** to connect to RPC server at the specified network address (and port)

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

  - Use DialHTTP for HTTP connection
- Use **Call** to call synchronous RPC: Call waits for the remote call 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 using the Call structure's Done channel

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

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.
}
```

RPC in Go: example

- Let’s consider a simple RPC calculators with two functions: multiply and divide two integers
- Code available on course site

RPC in Go: synchronous call

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

```
// Synchronous call
args := &server.Args{7,8}
var reply int
err = client.Call("Arith.Multiply", args, &reply)
if err != nil {
   log.Fatal("arith error:", err)
}
fmt.Printf("Arith: %d*%d=%d", args.A, args.B, reply)
```

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

RPC in Go: asynchronous call

- How to make asynchronous RPC? Go uses a channel as parameter to retrieve RPC reply when the call is complete
- Done channel will signal when the call is complete by returning the same object of Call
  - If Done is nil, Go will allocate a new channel

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

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

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

Go's RPC semantics

- Go's RPC is at-most-once
- Client opens TCP connections and writes the request
  - TCP may retransmit but server's TCP receiver will filter out duplicates internally, with sequence numbers
  - No request retry in Go's RPC code (i.e. will not create a second TCP connection)
- However, client returns an error if it does not get a reply
  - Perhaps after a TCP timeout
  - Perhaps server did not see the request
  - Perhaps server processed the request but server or network failed before reply came back

RPC in Go: marshaling and unmarshaling

- By default Go uses package encoding/gob for parameters marshaling (encode) and unmarshaling (decode)
  - Package `gob` manages streams of gobs (Go binary values) exchanged between Encoder (transmitter) and Decoder (receiver)
  - A stream of gobs is *self-describing*: each data item in the stream is preceded by a specification of its type, expressed in terms of a small set of predefined types; pointers are not transmitted, but the values they point to are transmitted
  - Basic usage: create an encoder, transmit some values, receive them with a decoder
  - Requires that RPC client and server are both written in Go

RPC in Go: marshaling and unmarshaling

- Two alternatives to gob

- `net/rpc/jsonrpc` package in Go's standard library
  - Implements a JSON-RPC 1.0 ClientCodec and ServerCodec for `rpc` package

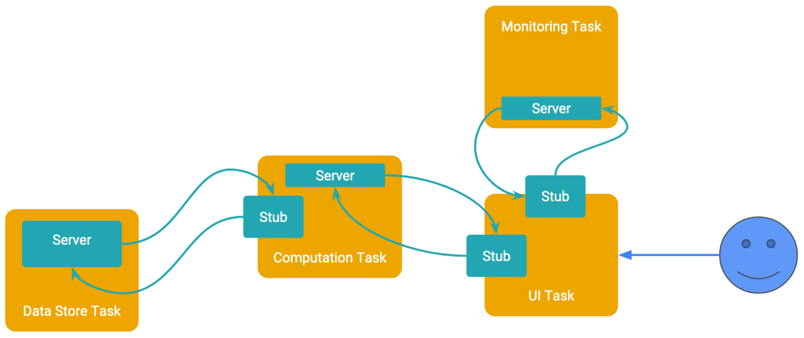
- gRPC
  - See next slides

Comparing RPC so far

- Let's compare RPC implementations
  - How do RPC implementations differ in terms of distribution transparency?
  - Access transparency?
  - Location transparency?
  - Replication transparency?
  - Concurrency transparency?
  - Failure transparency?

Motivation for new RPC middleware

- Large-scale distributed applications composed of **microservices**
  - Microservices architecture: building a software 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
  - Use communication predominantly structured as RPCs



gRPC

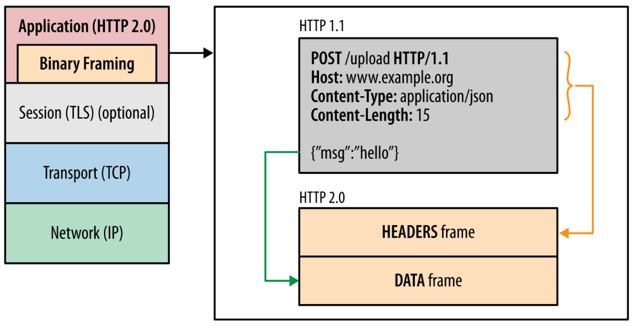
- High-performance, open source universal RPC framework grpc.io
- Can run in any environment
  - Multi-language, multi-platform framework
- Main usage scenarios
  - Connect polyglot microservices that use request-response style communication
  - Connect mobile devices and browsers to backend services
  - Generate efficient client libraries
- Developed by Google and now a CNCF project
- Used by many companies and in many distributed systems
  - E.g., Google, Dropbox, Netflix, Square, etcd, CockroachDB

gRPC: main features

- **HTTP/2** for transport
  - Bidirectional streaming and multiplexing
- **Protocol buffers** as IDL
  - Automatic code generation
  - Protobufs strict contracts prevent errors
- Plus authentication, bidirectional streaming and flow control, blocking or non-blocking bindings, timeouts and cancellation

gRPC: HTTP/2

- Transport over HTTP/2
  - Basic idea of gRPC: treat RPCs as references to HTTP objects
- HTTP/2: major revision of HTTP that provides significant performance benefits over HTTP 1.x
- HTTP/2 in a nutshell
  - Binary framing layer: HTTP/2 request/response is divided into small messages and framed in binary format, making message transmission efficient

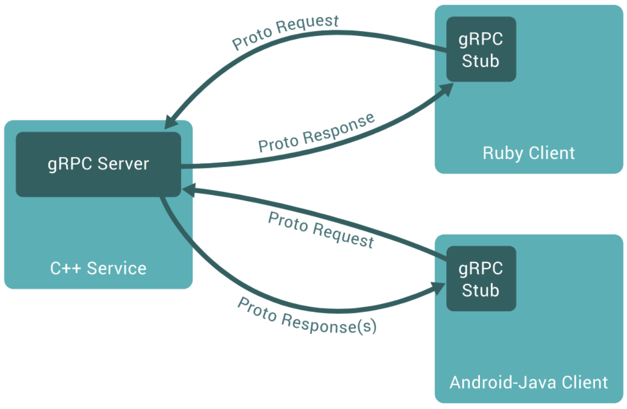


gRPC: HTTP/2

- HTTP/2 in a nutshell
  - From request/response messages to streams
    - Stream: bidirectional flow of bytes within an established connection, which may carry one or more messages
    - Message: complete sequence of frames that map to a logical request or response message
    - Frame: smallest unit of communication in HTTP/2, each containing a frame header, which at least identifies the stream to which the frame belongs
  - Request/response multiplexing (usage of a single connection per client): allows for efficient use of TCP connections and avoids head-of-line blocking at HTTP level
  - Native support for bidirectional streaming
  - HTTP header compression: to reduce protocol overhead
- See web.dev/performance-http2

gRPC: Protocol buffers

- gRPC uses protocol buffers (aka *protobufs*) as:
  - IDL to define service interface: automatic generation of client stubs and abstract server classes
  - Message interchange format: gRPC messages are serialized using protocol buffers, thus resulting in small message payloads
- Based on usual proxy pattern (*stub* and *server*)



Protocol buffers

- Google's mature open-source mechanism to serialize structured data
- Binary data representation
- Strongly typed

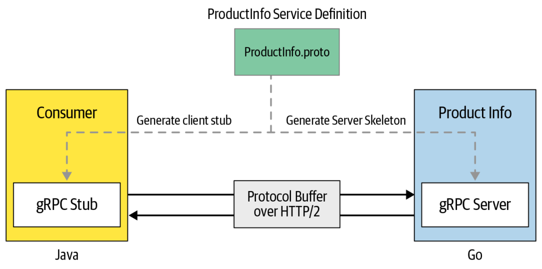
```
message Person {
  string name = 1;
  int32 id = 2;
  bool has_ponycopter = 3;
}
```

- Data types are structured as messages
  - message: small logical record of information containing a series of name-value pairs called *fields*
  - Fields have unique field numbers (e.g., `string name = 1`), used to identify fields in message binary format

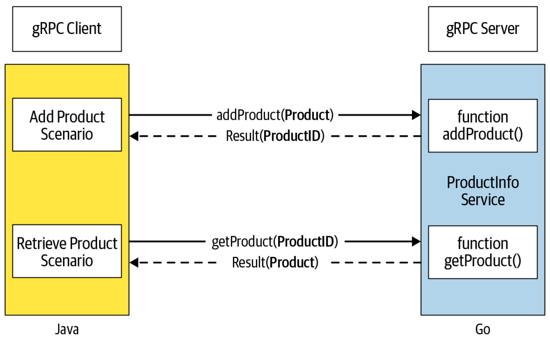
Protocol buffers: example

- ProductInfo service interface

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

service ProductInfo {
    rpc addProduct(Product) returns (ProductID);
    rpc getProduct(ProductID) returns (Product);
}

message Product {
    string id = 1;
    string name = 2;
    string description = 3;
}

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



Client-server interaction



gRPC: basic steps

1. Define service (collection of remote methods) and message types that are exchanged between client and service in `.proto` file using protobufs as IDL
2. Generate server and client code using `protoc` (protocol buffer compiler) in your preferred language(s) from your proto definition
   – Go: compile manually using `protoc` command
   – Java: use build automation tools like Bazel, Maven, or Gradle
3. Use gRPC API in your preferred language (e.g., Go, Java, Python) to write service client and server
   – Let's consider Go: gRPC-Go

gRPC: greeting example in Go

- See Quick start and helloworld example on grpc.io

1. Define service (`helloworld.proto` file)

```
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;
}
```

gRPC: greeting example

2. Compile 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`: contains protobuf code to populate, serialize, and retrieve request and response message types
  - `helloworld_grpc.pb.go`: contains
    - interface type (or *stub*) for clients to call with methods defined in Helloworld service
    - interface type for servers to implement, also with methods defined in Helloworld service

gRPC: greeting example

3. Create server: composed of two parts
   a. Implement service interface generated from service definition: the actual “work”

```
func (s *server) SayHello(ctx context.Context,
   in *pb.HelloRequest) (*pb.HelloReply, error) {
    …
}
```

   b. Create and run a gRPC server to listen for requests from clients and dispatch them to service implementation

```
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)
```

gRPC: greeting example

4. Create client
   – To call service methods, we first need to create a *gRPC channel* to communicate with the target server using `Dial`

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

   – Then we need a client stub to perform RPCs: we get it using `pb.NewGreeterClient` provided by pb package generated from `.proto` file

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

   – Then we call service methods on client stub: we create and populate a request protobuf object (`HelloRequest`) and pass a `context` object which lets us change RPC’s behavior if necessary (e.g., time-out/cancel RPC in flight)

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

gRPC: greeting example

- Let's update gRPC service adding new method `SayHelloAgain()`
  1. Update `.proto` file
  2. Regenerate gRPC code using `protoc`
  3. Update server code to implement new method

```
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 to call 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())
```

gRPC: types of RPC methods

- gRPC supports 4 kinds of service methods that can be defined in `.proto` file
- See routeguide example on grpc.io
- *Simple RPC*: client sends a request to server and waits for a single response to come back (i.e., unary)
  – A normal function call

```
rpc SayHello (HelloRequest) returns (HelloReply) {}
```

- *Server-side streaming RPC:* client sends a request to server and gets a stream to read a sequence of messages back
  – gRPC guarantees message ordering within an individual RPC call

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

gRPC: types of RPC methods

- *Client-side streaming RPC*: client writes a sequence of messages and sends them to server, waits for the server to read them and return its response
  - gRPC guarantees message ordering within an individual RPC call

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

- *Bidirectional streaming RPC*: both sides send a sequence of messages using a read-write stream (i.e., full duplex)
  - gRPC preserves the order of messages in each stream

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

gRPC: ProductInfo example

- Online retail scenario has a ProductInfo microservice which is responsible for managing products and their information; clients can add and retrieve products

1. Define service in .proto file (see slide 110)
2. Implement server and client in Go
3. Implement server and client in Java

Go code on course web site

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

gRPC: weaknesses

- Limited support in browsers
  - Lack of full HTTP/2 support
  - gRPC-Web can provide gRPC support to browser but limited features (only simple RPC and limited server streaming)
- Non-human readable format
  - Protobuf is efficient to send and receive, but its binary format is not human readable
  - Developers need additional tools (e.g., gRPC command-line tool) to analyze protobuf payloads on the wire, write manual requests, and perform debugging

References

- Chapter 4 of van Steen & Tanenbaum book
- Petron, Remote Procedure Calls
- Java Remote Method Invocation Specification, docs.oracle.com/en/java/javase/21/docs/specs/rmi/
- Trail: RMI, docs.oracle.com/javase/tutorial/rmi/
- RPyC tutorial, rpyc.readthedocs.io/en/latest/tutorial.html
- RPyC documentation, rpyc.readthedocs.io
- Go's package rpc, pkg.go.dev/net/rpc
- Jackson, RPC in the Go standard library, in Building Microservices with Go, 2017 subscription.packtpub.com/book/web-development/9781786468666/1/ch01lvl1sec12/rpc-in-the-go-standard-library
- Indrasiri and Kuruppu, gRPC - Up and Running, O'Reilly, 2020