Remote Procedure Call

IPADS, Shanghai Jiao Tong University

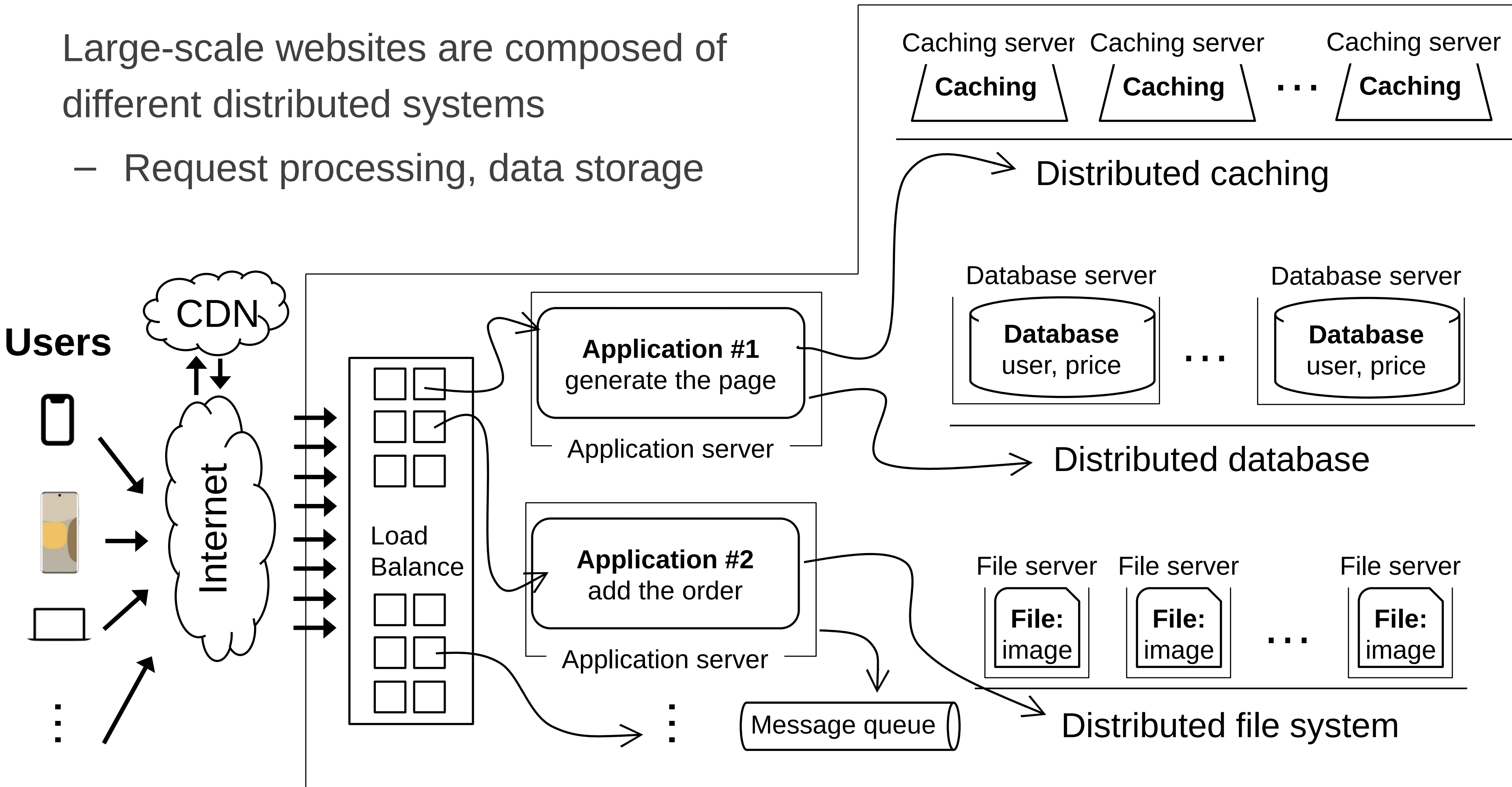
https://www.sjtu.edu.cn

Credits: Rong Chen@IPADS

Distributed system that support large-scale websites

Large-scale websites are composed of different distributed systems

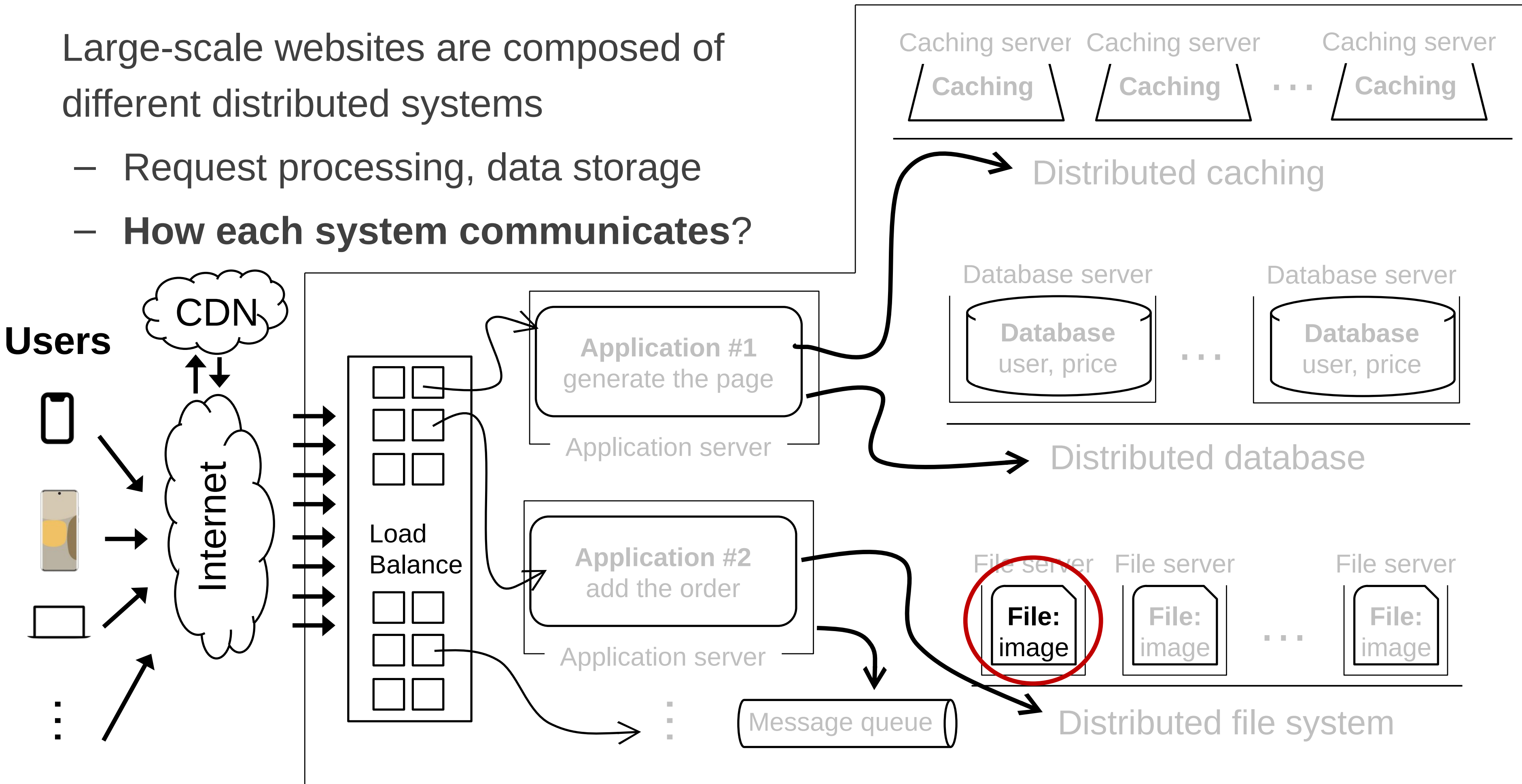
- Request processing, data storage



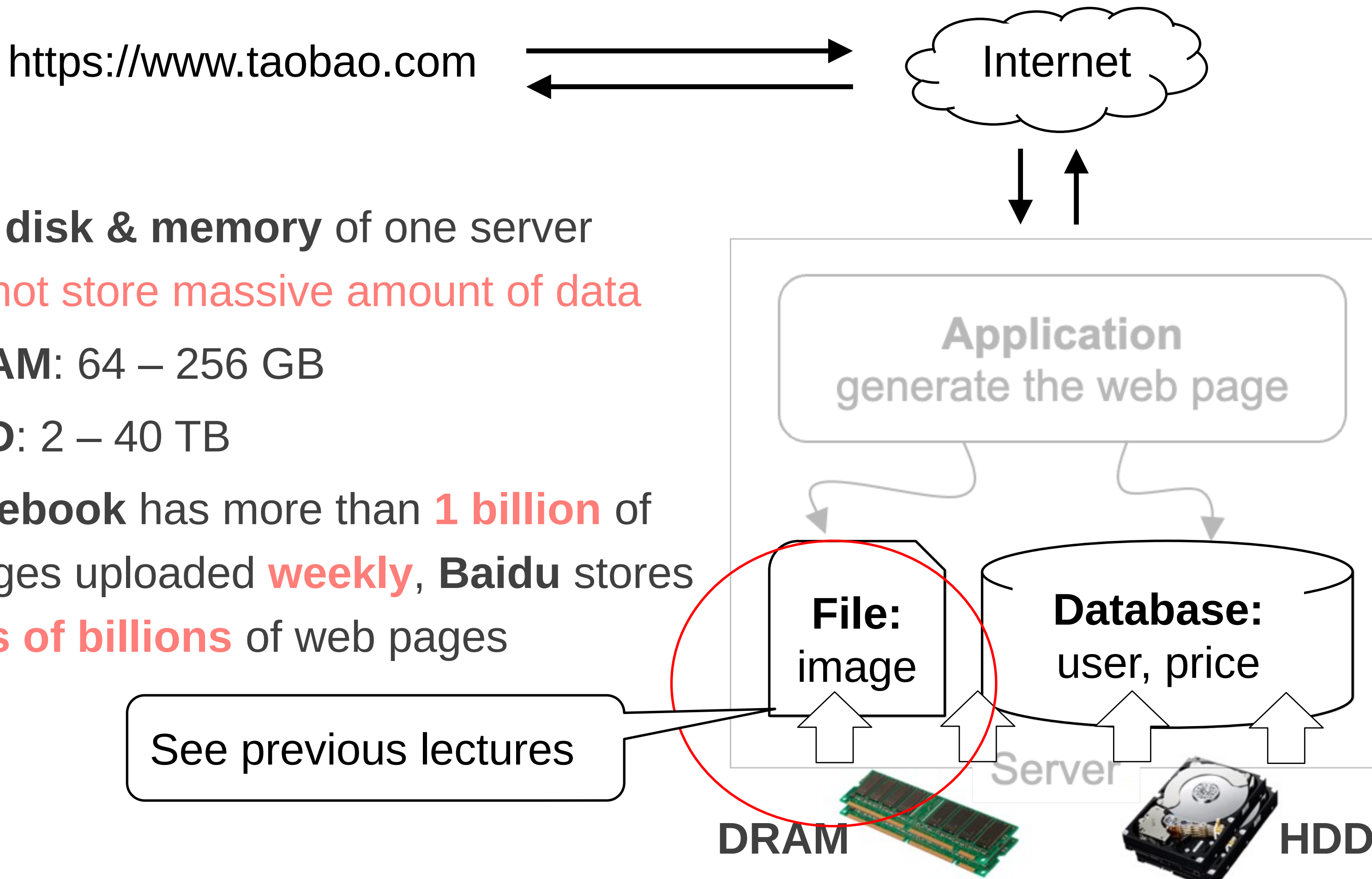
Example: extend a single-node filesystem to a distributed one

Large-scale websites are composed of different distributed systems

- Request processing, data storage
- **How each system communicates**?



Recall: The architecture of LAMP cannot scale!



1. The **disk & memory** of one server cannot store massive amount of data
   - **DRAM**: 64 – 256 GB
   - **HDD**: 2 – 40 TB
   - **Facebook** has more than **1 billion** of images uploaded **weekly**, **Baidu** stores **tens of billions** of web pages

Step #1 for scalability: disaggregating application & data

Application: handles application logic

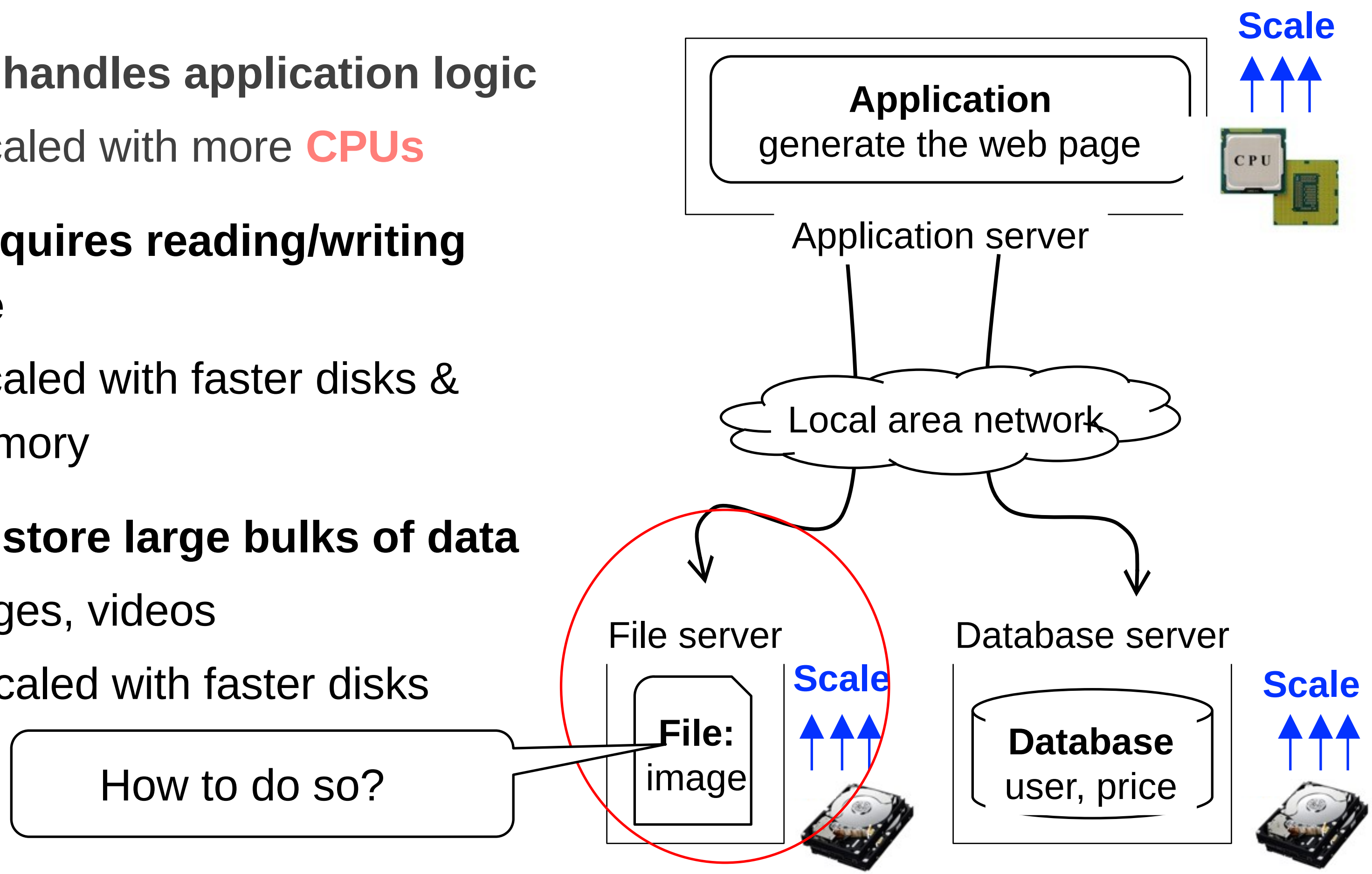
- Can be scaled with more **CPUs**

Database: requires reading/writing disk & cache

- Can be scaled with faster disks & larger memory

File system: store large bulks of data

- E.g., images, videos
- Can be scaled with faster disks



Remote Procedure Call (RPC)

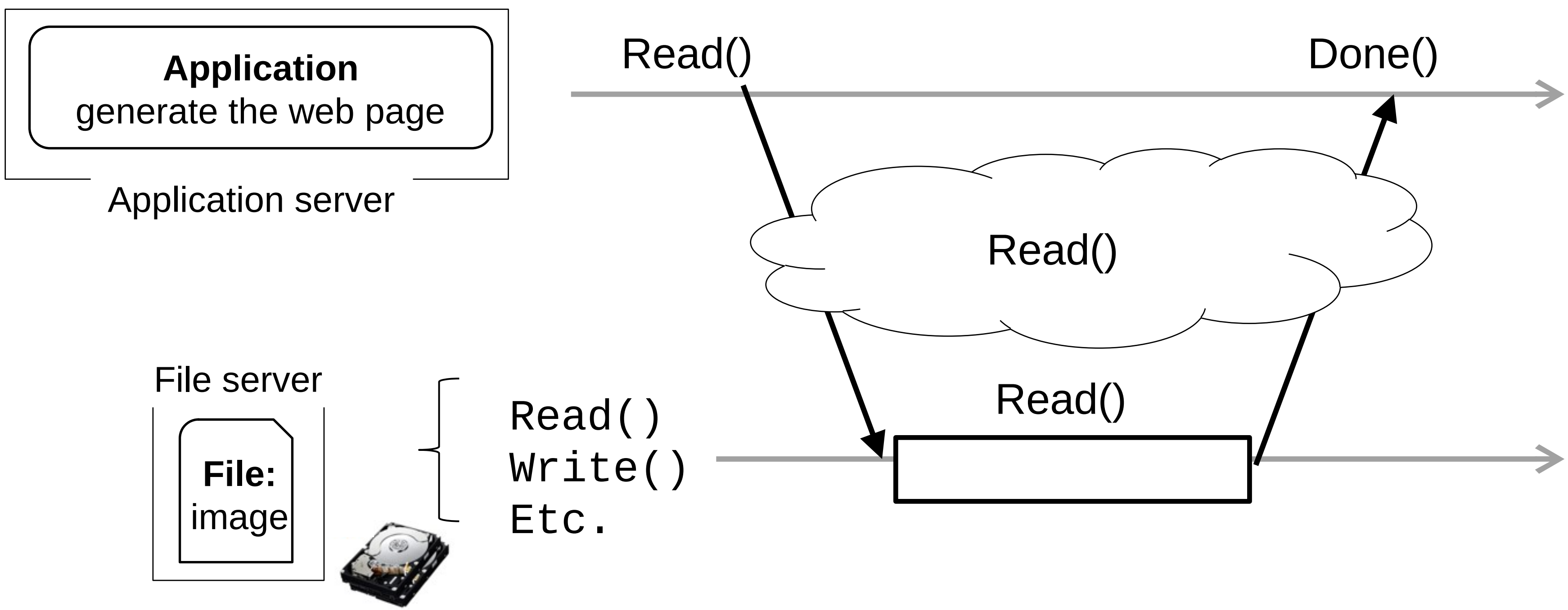
“Remote procedure calls (RPC) appear to be a useful paradigm.”

— Birrel & Nelson, 1984

Andrew Birrell, Bruce Nelson, ***“Implementing Remote Procedure Calls”***. ACM Transactions on Computer Systems (TOCS), 2(1), **1984**

filesystem + RPC, a form of distributed filesystem

Calling a function on a remote server like a local one !



RPC vs. Sockets API

The sockets interface forces a read/write mechanism

- read(), write(), sendmsg(), etc.

Programming is often easier with a functional interface

- To make **distributed** computing look more like **centralized** computing

Goal of RPC:

- It should appear to the programmer that a normal call is taking place

Idea: build the RPC atop of the socket interface

- Hide the construction of messages and remote invocation logic from the developers

RPC: Remote Procedure Call

Allow a procedure to execute in another address space without coding the details for the remote interaction

RPC History

- Idea goes back in 1976
- Sun's RPC: first popular implementation on Unix
  - Used as the basis for NFS
- Many modern RPC frameworks: gRPC, bRPC, etc.
- **RMI** (Remote Method Invocation)
  - Object-oriented version of RPC, e.g. in Java

Example of RPC

```
procedure MEASURE (func)
    start ← GET_TIME (SECONDS)
    func () // invoke the function
    end ← GET_TIME (SECONDS)
    return end – start
```

```
procedure GET_TIME (units)
    time ← CLOCK
    time ← CONVERT_TO_UNITS (time, units)
    return time
```

The implementation of GET_TIME.

Suppose we want to measure the execution time of *func()*

Assumption:

– Only the server has the implementation of GET_TIME

How can the client call server's GET_TIME?

Call GET_TIME in a single machine case

```
procedure MEASURE (func)
    start ← GET_TIME (SECONDS)
    func () // invoke the function
    end ← GET_TIME (SECONDS)
    return end – start
```

```
procedure GET_TIME (units)
    time ← CLOCK
    time ← CONVERT_TO_UNITS (time, units)
    return time
```

GET_TIME

func()

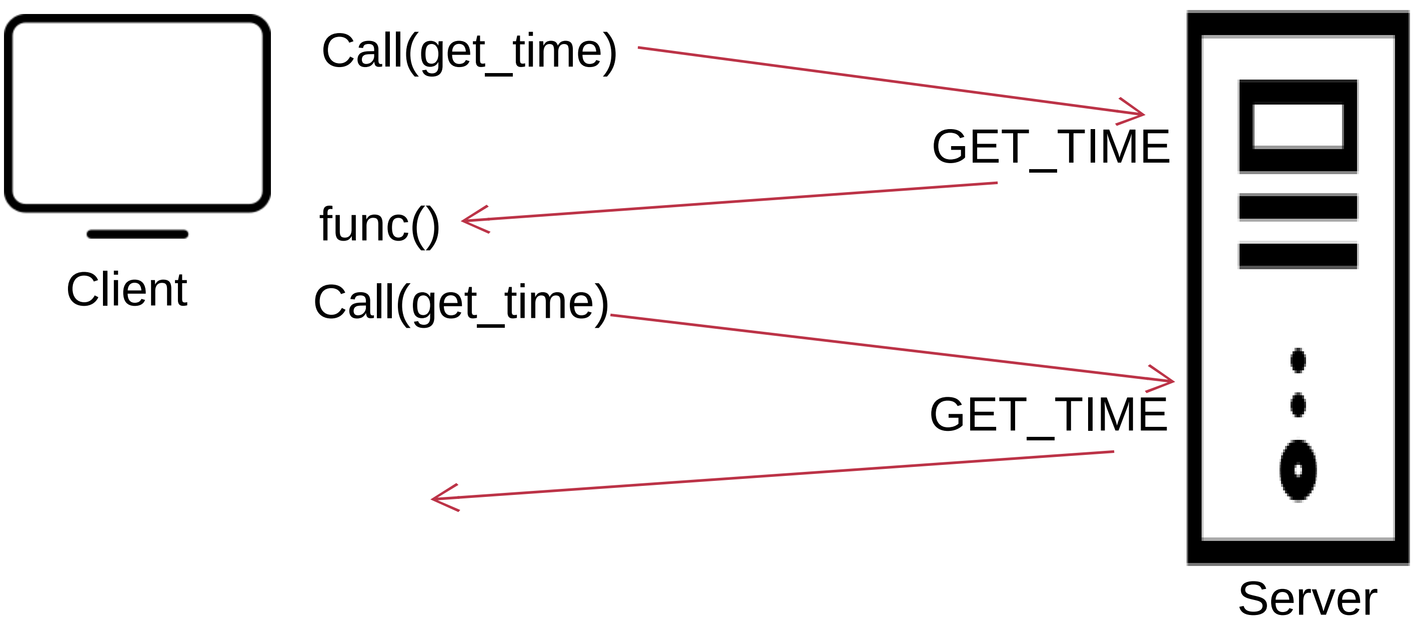
GET_TIME

Client

Single-machine call vs. distributed call （RPC）

```
procedure MEASURE (func)
    start ← GET_TIME (SECONDS)
    func () // invoke the function
    end ← GET_TIME (SECONDS)
    return end − start
```

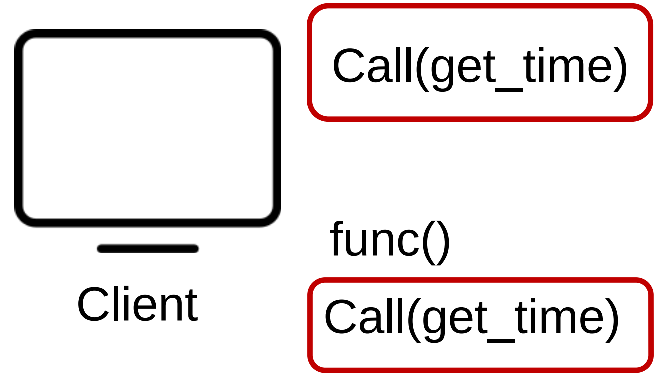
```
procedure GET_TIME (units)
    time ← CLOCK
    time ← CONVERT_TO_UNITS (time, units)
    return time
```



Handwritten code: client

Local

```
procedure MEASURE (func)
    start ← GET_TIME (SECONDS)
    func () // invoke the function
    end ← GET_TIME (SECONDS)
    return end – start
```

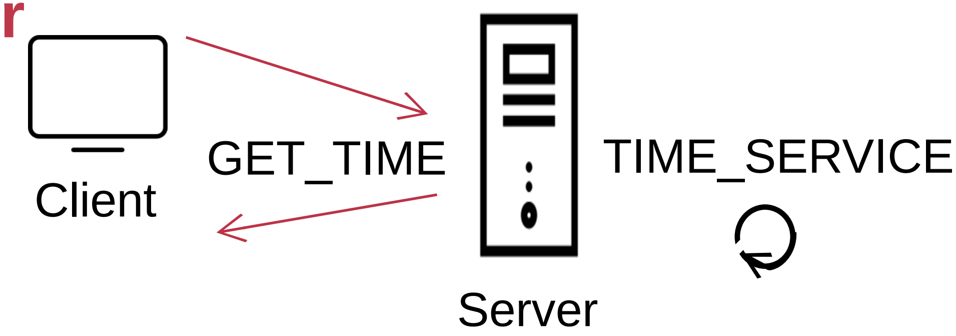


RPC

Client program

```
procedure MEASURE (func)
    SEND_MESSAGE (NameForTimeService, {"Get time", CONVERT2EXTERNAL(SECONDS)})
    response ← RECEIVE_MESSAGE (NameForClient)
    start ← CONVERT2INTERNAL (response)
    func ()        // invoke the function
    SEND_MESSAGE (NameForTimeService, {"Get time", CONVERT2EXTERNAL(SECONDS)})
    response ← RECEIVE_MESSAGE (NameForClient)
    end ← CONVERT2INTERNAL (response)
    return end – start
```

Handwritten code: server




```
procedure TIME_SERVICE ()
    do forever
        request ← RECEIVE_MESSAGE (NameForTimeService)
        opcode ← GET_OPCODE (request)
        unit ← CONVERT2INTERNAL(GET_ARGUMENT (request))
        if opcode = "Get time" and (unit = SECONDS or unit = MINUTES) then
            time ← CONVERT_TO_UNITS (CLOCK, unit)
            response ← {"OK", CONVERT2EXTERNAL (time)}
        else
            response ←{"Bad request"}
        SEND_MESSAGE (NameForClient, response)
```

RPC simplifies the implementation of remote calls

Abstracts away the common parts with stub

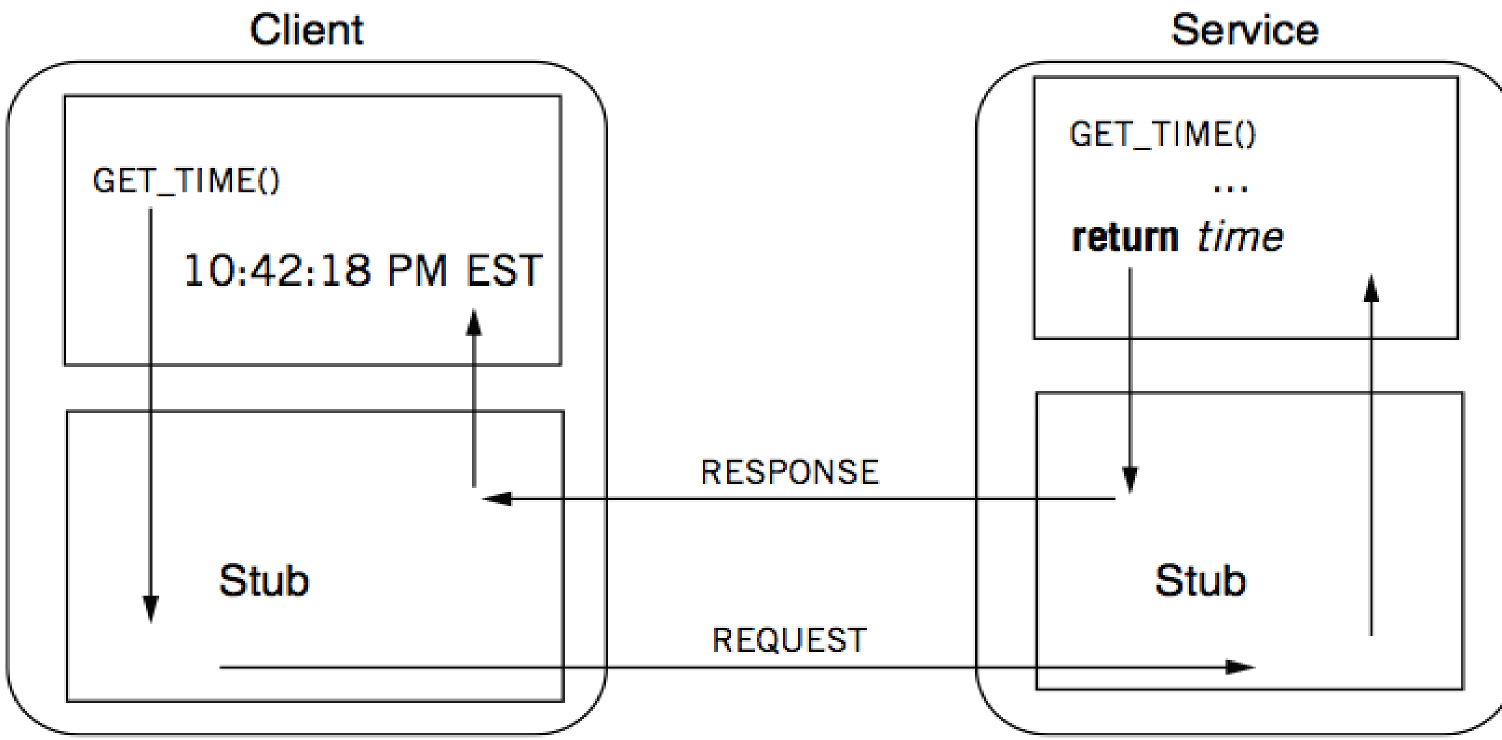
Provided in RPC's stub

Client program

```
procedure MEASURE (func)
    SEND_MESSAGE (NameForTimeService, {"Get time", CONVERT2EXTERNAL(SECONDS)})
    response ← RECEIVE_MESSAGE (NameForClient)
    start ← CONVERT2INTERNAL (response)
    func ()        // invoke the function
    SEND_MESSAGE (NameForTimeService, {"Get time", CONVERT2EXTERNAL(SECONDS)})
    response ← RECEIVE_MESSAGE (NameForClient)
    end ← CONVERT2INTERNAL (response)
    return end − start
```

```
procedure TIME_SERVICE ()
    do forever
        request ← RECEIVE_MESSAGE (NameForTimeService)
        opcode ← GET_OPCODE (request)
        unit ← CONVERT2INTERNAL(GET_ARGUMENT (request))
        if opcode = "Get time" and (unit = SECONDS or unit = MINUTES) then
            time ← CONVERT_TO_UNITS (CLOCK, unit)
            response ← {"OK", CONVERT2EXTERNAL (time)}
        else
            response ←{"Bad request"}
        SEND_MESSAGE (NameForClient, response)
```

RPC: a complete calling process



RPC stub

Client stub

- Put the arguments into a request
- Send the request to the server
- Wait for a response

***Stub**: hide communication details from up-level code, so that up-level code does not change.*

Service stub

- Wait for a message
- Get the parameters from the request
- Call a procedure according to the parameters (e.g. GET_TIME)
- Put the result into a response
- Send the response to the client

Client Program using RPC

```
procedure MEASURE (func)
    start <- GET_TIME(SECONDS)
    func()
    end <- GET_TIME(SECONDS)
    return end - start
```

Note: code is not changed comparing with the single-machine case

This is the **stub** of client

```
procedure GET_TIME (units)
    SEND_MESSAGE(ServerName, {"Get time",
CONVERT2EXTERNAL(units)})
    response <- RECEIVE_MESSAGE(ClientName)
    if GET_RETCODE(response) != "OK"
        HANDLE_ERROR(response)
    else
        return CONVERT2INTERNAL(GET_ARGUMENT(response))
```

Server Program using RPC

```
procedure GET_TIME (units)
    time <- CLOCK
    time <- CONVERT_TO_UNITS(time,
units)
    return time
```

Note: this code is not changed

This is the **stub** of server

```
procedure TIME_SERVICE ()
    do forever
        request <- RECEIVE_MESSAGE(ServerName)
        opcode <- GET_OPCODE(request)
        arg <- CONVERT2INTERNAL(GET_ARGUMENT(request))
        if opcode = "Get time" and (arg = SECONDS or arg = MINUTES)
then
            retval <- GET_TIME(arg)
            response <- {"OK", CONVERT2EXTERNAL(retval)}
        else
            response <- {"Bad request"}
        SEND_MESSAGE(ClientName, response)
```

Question: what is inside a message?

```
procedure GET_TIME (units)
    SEND_MESSAGE(ServerName, {"Get time", CONVERT2EXTERNAL(units)})
    response <- RECEIVE_MESSAGE(ClientName)
    if GET_RETCODE(response) != "OK"
        HANDLE_ERROR(response)
    else
        return CONVERT2INTERNAL(GET_ARGUMENT(response))
```

```
procedure TIME_SERVICE ()
    do forever
        request <- RECEIVE_MESSAGE(ServerName)
        opcode <- GET_OPCODE(request)
        arg <- CONVERT2INTERNAL(GET_ARGUMENT(request))
        if opcode = "Get time" and (arg = SECONDS or arg = MINUTES) then
            retval <- GET_TIME(arg)
            response <- {"OK", CONVERT2EXTERNAL(retval)}
        else
            response <- {"Bad request"}
        SEND_MESSAGE(ClientName, response)
```

Question: what is inside a message?

A message may contain:

- Service ID (e.g., function ID)
- Service parameter (e.g., function parameter)
- Using marshal / unmarshal

RPC request message

RPC request:

- **Xid** → X is short for "transaction"
  Client reply dispatch uses xid
  Client remembers the xid of each call
- call/reply
- rpc version
- **program #**
- program version
- **procedure #**
- auth stuff
- arguments

Server dispatch uses prog#, proc#

RPC reply message

RPC reply:

- **Xid**
- call/reply
- **accepted**? (Yes, or No due to bad RPC version, auth failure, etc.)
- auth stuff
- **success**? (Yes, or No due to bad prog/proc #, etc.)
- **results**

Binding: find the server

Can implement with other network name services

- E.g., 192.168.10.233:8888 + function ID

Example: gRPC

gRPC client

```
grpc::CreateChannel("localhost:50051",

grpc::InsecureChannelCredentials());
```

binding

gRPC server

```
std::stringserver_address("0.0.0.0:50051");
...
... AddListeningPort(server_address, ...);
```

How to pass the data between

client & server?

Parameter passing

Pass by value?

– Easy: just copy data to network message

Pass by reference?

– Makes no sense without shared memory

Needs a conversion between data used in a program vs. data that can be transferred through the network

– Client converts data structure into **pointerless** representation

– Client transmits data to the server

– Server reconstructs structure with local pointers

Parameter passing is challenging across machines

Distributed systems have the **incompatibility** problem,

- which does not exist on a single machine

For example, remote machine may have **different**

- byte ordering,
- sizes of integers and other types,
- floating point representations,
- character sets,
- alignment requirements
- etc.

Represent 0x89abcdef

Big endian, e.g., Power processor

89	AB	CD	EF
0	1	2	3

Little endian, e.g., X86

EF	CD	AB	89
0	1	2	3

Parameter passing is more challenging

Application is changing over time

- What would happen if a server **upgrades** while the client is intact?
- E.g., server upgrades and adds a priority field to its message, what would happen if the client does not change?

Evolvability: we should built systems that are easy to adapt to changes!

- **Backward compatibility**: newer code can read data that was written by older code
- **Forward compatibility**: older code can read that was written by newer code

Representing data: encoding

Need standard **encoding** to enable communication between heterogeneous systems & different versions of software

- Sun’s RPC uses XDR (eXternal Data Representation)
- ASN.1 (ISO Abstract Syntax Notation)
- JSON
- Google Protocol Buffers
- W3C XML Schema Language

Why not using language-specific formats?

Many languages have **built-in support** for encoding in-memory objects

- Java's `java.io.Serializable`
- Python's `pickle`

Drawbacks:

- The encoding is tied to a particular programming language
  - E.g., it's challenging to use a Java client to call a python server
- No versioning -> no forward and backward compatibility

System requirements for encoding/decoding

Transfer objects through the network

- Correctly encode and decode a object to a byte stream

Compatibility

- Support multiple language, multiple versions of program

Efficiency

- Reduce the traffic transferred from the network
- Network bandwidth is a scarce resource

Standardized encoding: JSON, XML & etc.

Independent to a specific programming language

Textual formats:

- JSON
- XML
- CSV

Logic client request format

- Xid
- call/reply
- rpc version
- program #
- program version
- procedure #
- auth stuff
- arguments

JSON representation

```
{ “xid” : 12, ”call”: true,
  “rpc_version”: 73,
  ...
}
```

Standardized encoding: JSON, XML & etc.

Independent to a specific programming language

Textual formats:

- JSON
- XML
- CSV

Benefits:

- Human-readable: easy to debug

Standardized encoding: JSON, XML & etc

Independent to a specific programming language

Textual formats:

- JSON
- XML
- CSV

Drawbacks:

1. Ambiguity around encoding of numbers
2. How to support **binary strings**?
    - Programmers have to **re-encode** the string as Base64, etc.
3. **Verbose**: use more bytes to store the data. E.g., tags like <xx> </xx> in XML

How may bytes should I use to store the number?

JSON representation

```
{ “xid” : 12, ”call”: true,
  “rpc_version”: 73,
  ...
}
```

Binary formats

Encode the data using **binary encoding**

- **Pros**: more compact, more accurate, faster to parse and store
- **Cons**: less human-readablity
  - Not a problem when the data is only used internally
  - E.g., RPC message metadata

Typical example: **Thrift & Protocol Buffers**

Thrift, by Facebook, now in Apache

Protocol Buffers, by Google

```
{ "userName": "Martin", "favoriteNumber": 1337, "interests": ["daydreaming", "hacking"] }
```

JSON representation takes 81B in total (w/o spaces)

Binary formats can reduce it to **34B or even less**

Binary formats: schema

Both Thrift and Protocol Buffers require a **schema** for any data that is encoded

- **Benefits**: no need to encode things such as userName in the encoded data

Thrift interface definition language (IDL)

```
struct Person {
  1: required string userName,
  2: optional i64 favoriteNumber,
  3: optional list<string> interests
}
```

Protocol Buffers IDL

```
message Person {
    required string user_name = 1;
    optional int64 favorite_number = 2;
    repeated string interests = 3;
}
```

```
{ "userName": "Martin", "favoriteNumber":
1337, "interests": ["daydreaming",
"hacking"] }
```

JSON representation takes 81B in total (w/o spaces)

The BinaryProtocol of Thrift

Each field has:

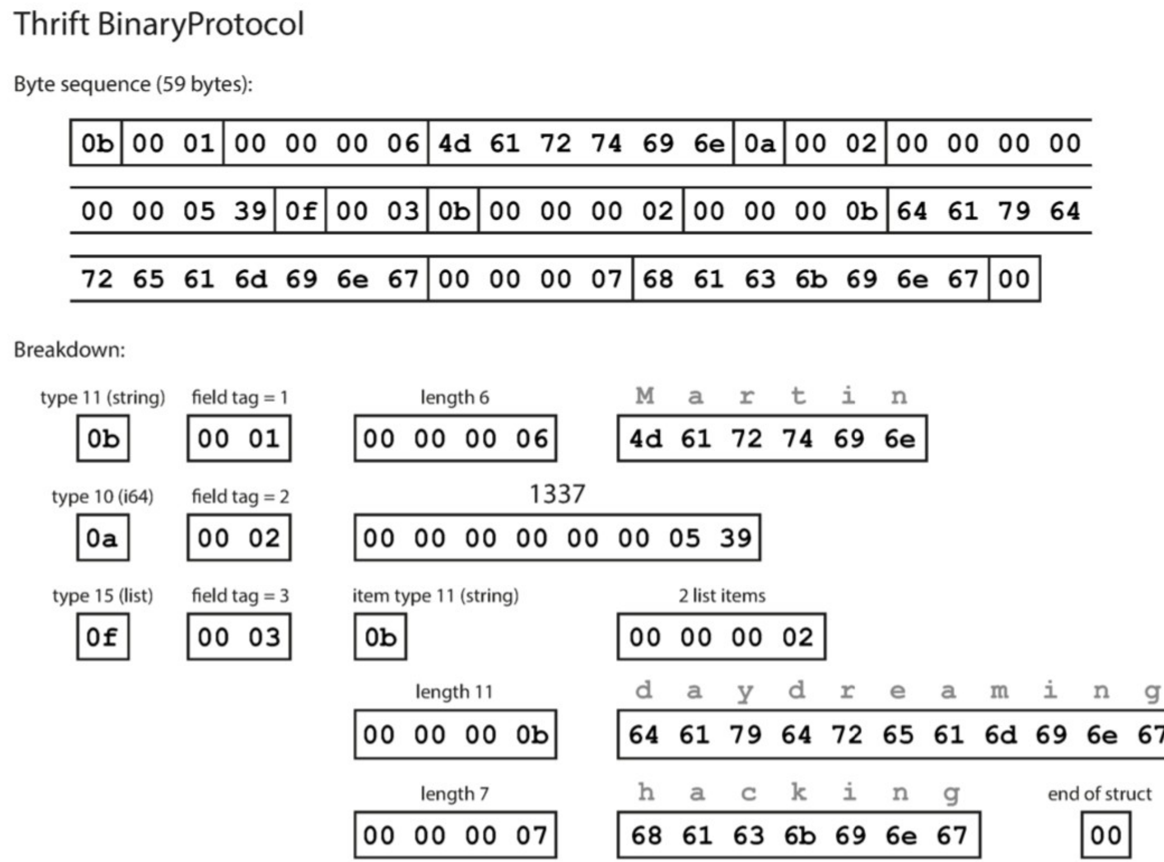
- Type annotation (1B)
- Type field (1B)
- A length indication (optional, required for string, list, etc.)
- Data (like json)

For each type, the protocol assumes an internal encoding

- E.g., little endian

Total 59B

- no field names (userName, favoriteNumber, interests)

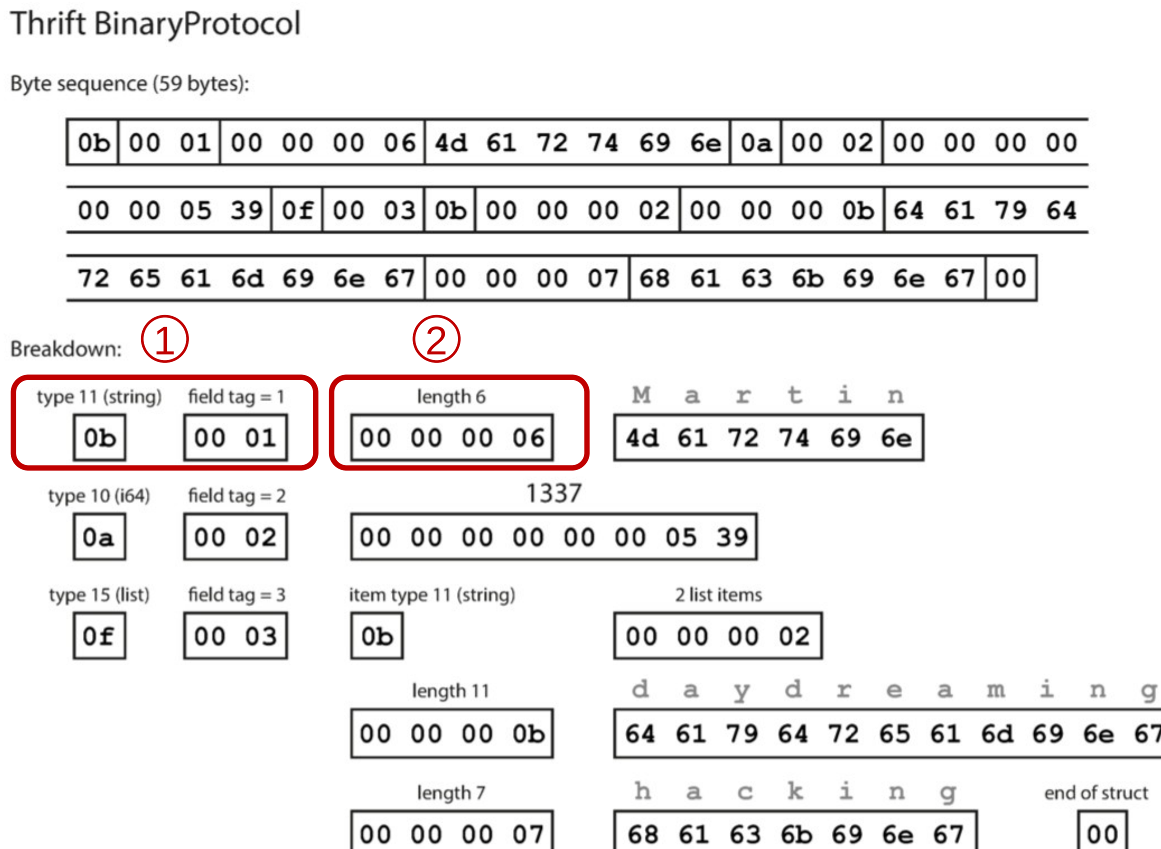


Being more compact: Thrift CompactProtocol

Techniques:

① Packing field type & field tag in 1B

② Variable-length integer: *top-bit of each byte indicates whether there are more bytes*



Being more compact: Thrift CompactProtocol

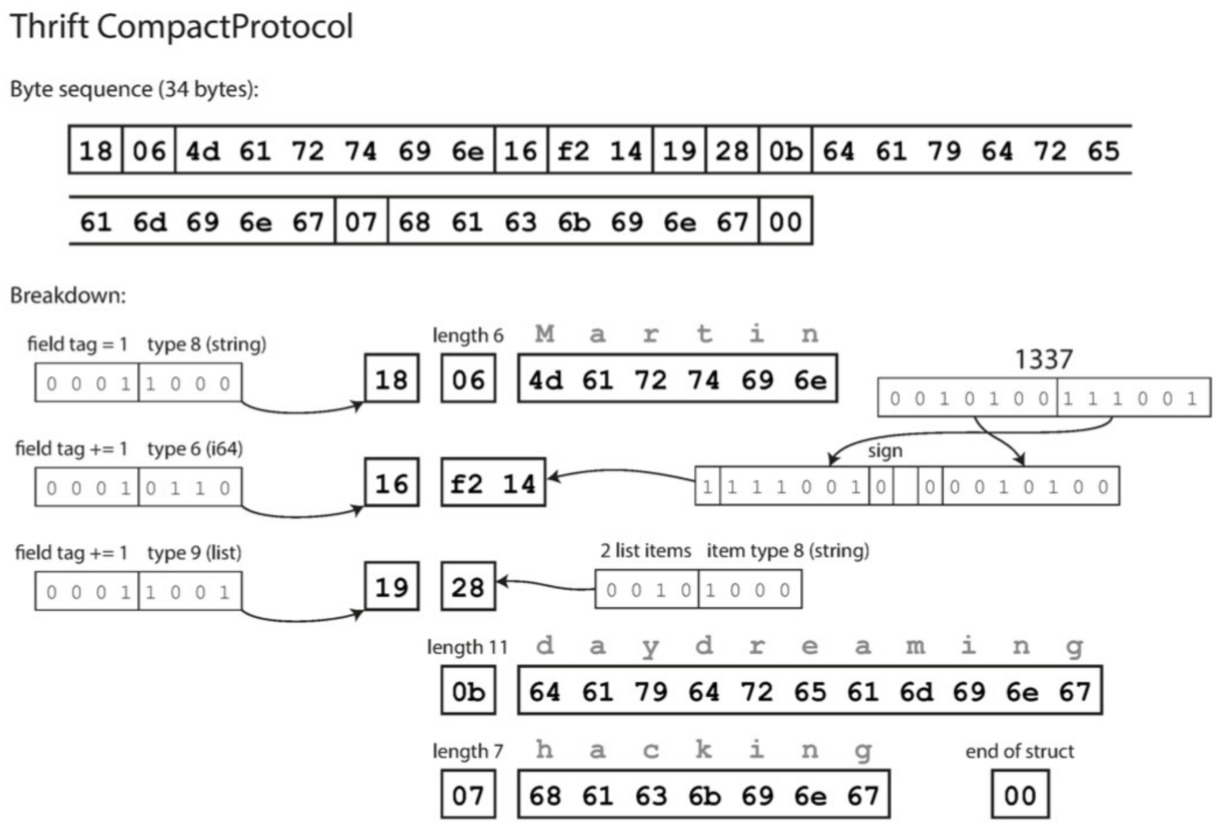
Techniques:

① Packing field type & field tag in 1B

② Variable-length integer: *top-bit of each byte indicates whether there are more bytes*

1337: from 8B to 2B

Now only consumes **34B**

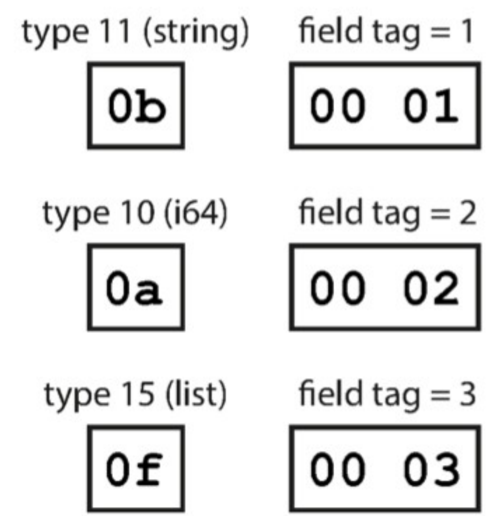


Schema simplifies supporting compatibility

Key observation: we only need to keep the field tags & field type compatible

Forward compatible: if new schema preserves the presentation of old schema, then new code can trivially decode the data encoded with old schema

Backward compatible: old code can simply **skip** fields with **unknown field tag**



Short summary of parameter passing in RPC

Passing data using RPC is non-trivial, should ensure:

- Correctness
- Compact
- Evolvable

Encoding/decoding

- Convert an object into **an array of bytes** with enough annotation so that the decode procedure can convert it back into an object
- Also known as **marshal / unmarshal** in RPC
- With other names, e.g., **serialize / unserialize or deserialize**

Automatic stub generation

Generate stubs from an interface specification

- Tool to look at argument and return types
- Generate the **marshal** and **unmarshal** code
- Generate stub procedures

Benefits:

- Saves programming (thus less error)
- Ensures agreement on argument types
  - E.g., consistent function ID

Typically a standard component in an RPC library

Transport protocol of RPC

Stubs also include implementations of sending/receiving messages

- **TCP** or **UDP**, which one to use?
- What about new networking features, e.g., RDMA?

Hide the transport protocol from the user

- **Benefits**: e.g., transparent migrate to a more advanced network

Most support several

- Allow programmer (or end user) to choose at runtime based on their hardware setup or performance requirements

Short summary of encoding & decoding data for RPC

Transfer objects through the network

- Correctly encode and decode a object to a byte stream

Compatibility

- Support multiple language, multiple versions of program

Uniformed format + Schema

Efficiency

- Reduce the traffic transferred from the network
- Network bandwidth is a scarce resource

Binary format

Short summary of encoding & decoding data for RPC

Efficiency

- Reduce the traffic transferred from the network
- Network bandwidth is a scarce resource

Question

- Does binary format always improve the efficiency of RPC?
- E.g., What if the network is becoming faster?

Short summary: RPC so far

RPC simplifies programming w/ an interface similar to local function call

RPC uses stubs to avoid handling argument **encoding/decoding** and send/receiving messages, etc.

- Ensure correctness & efficiency

Remain challenges of RPC：

- How to handle failure?
  - As we have mentioned, failure is common especially in large-scale distributed systems!

When RPC meets failure

When RPC meets failures

Local procedure calls do not fail (most of the time)

- If that happens, the entire process dies

RPC may meet many different failures

- Parameter error -> just return
- Server crash -> how the client copes with it?

Crash can happen even the remote procedure runs correctly!

When RPC meets failures

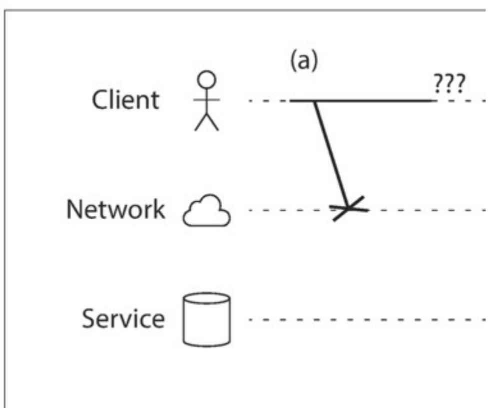
RPC could fail in a **partial failure** way

– Like what we have presented in lecture 02

A client uses RPC to call a function at a service through the network

– The client does not get any reply

– What could possibly happen?



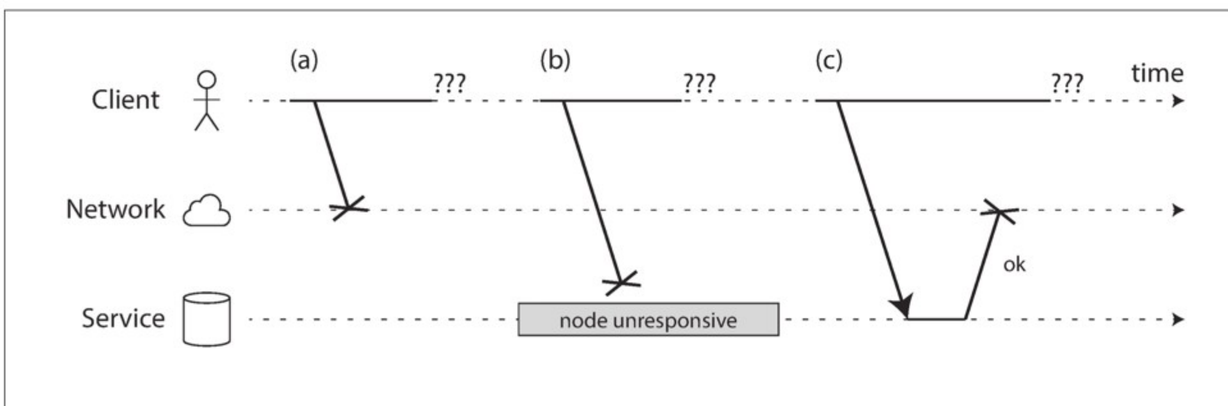
When RPC meets failures

A user sends **an RPC** but **the server does not reply**, possible reasons:

1. Request may have been **lost** (e.g., someone unplugged a network cable).

2. Request may be waiting in a queue and will be delivered **later** (e.g., the network or the recipient is overloaded).

3. The remote node may have **failed** (e.g., it crashed or it was powered down).



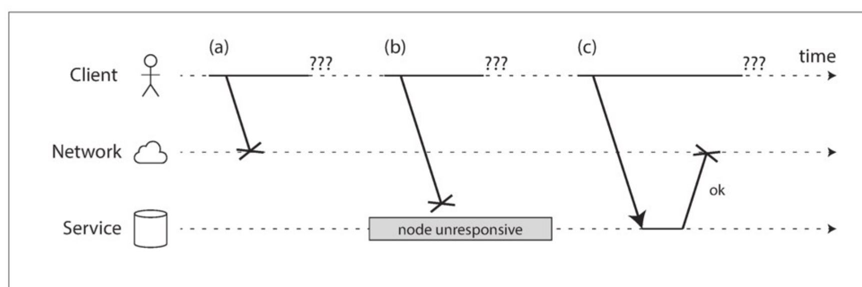
When RPC meets failures

A user sends **an RPC** but **the server does not reply**, possible reasons:

4. The remote node may **have temporarily stopped** responding (e.g., it is experiencing a long **garbage collection pause**)

5. The remote node may **have executed the function**, but the **response** has been **lost** on the network (e.g., a network switch has been misconfigured).

6. The remote node may **have executed the function**, but the response has been **delayed** and will be delivered later (e.g., the network or your own machine is overloaded).



RPC != PC

The **transparency** of RPC breaks here:

– Applications should be prepared to deal with RPC failure

– E.g., gRPC will return a status for application to check

```
// Here we can use the stub's newly available method we just adde
Status status = stub_->SayHelloAgain(&context, request, &reply);
if (status.ok()) {
  return reply.message();
} else {
  std::cout << status.error_code() << ": " << status.error_messag
            << std::endl;
  return "RPC failed";
}
```

When RPC meets failures

Semantics of remote procedure calls

- Local procedure call: **Exactly once**

A remote procedure call may be called:

- **0 time**: server crashed or serve process died before executing server code
- **1 time**: everything worked well, as expected
- **1 or more**: excess latency or lost reply from server and client retransmission
- **0 or 1 time**: the function can execute at most once

What is the most desirable RPC semantic for the developers?

What is the desirable RPC semantic for system developers?

RPC semantic

Most RPC systems will offer either:

- **At-least-once** semantics
- **At-most-once** semantics

Simple **retransmission** leads to "**at-least-once**"

Birrell's RPC semantics (1984) :

- server says **OK**: executed once
- server says **CRASH**: zero or one time

much **easier** than exactly once, more **useful** than at-least-once

RPC semantic

Understand application:

- **Idempotent**: may be run any number of times without harm (e.g., i = 1)
- **Non-idempotent**: those with side-effects (e.g., i++)

When **at-least-once** is **OK**?

- if no side effects (e.g., read-only operation)
- if app has its own plan for detecting duplication

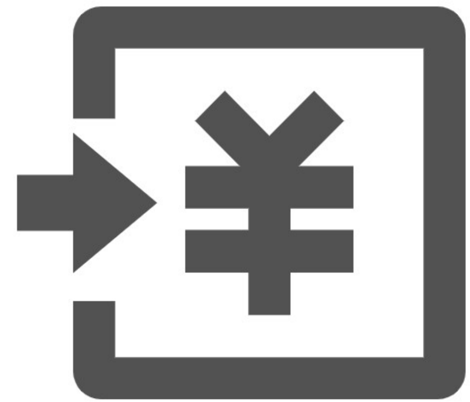
Cases of idempotence

“Idempotence means that multiple invocations of some work are identical to exactly one invocation.”

——Pat Helland

Sweeping the floor is idempotent

Pressing the elevator button is idempotent



Saving money is non-idempotent

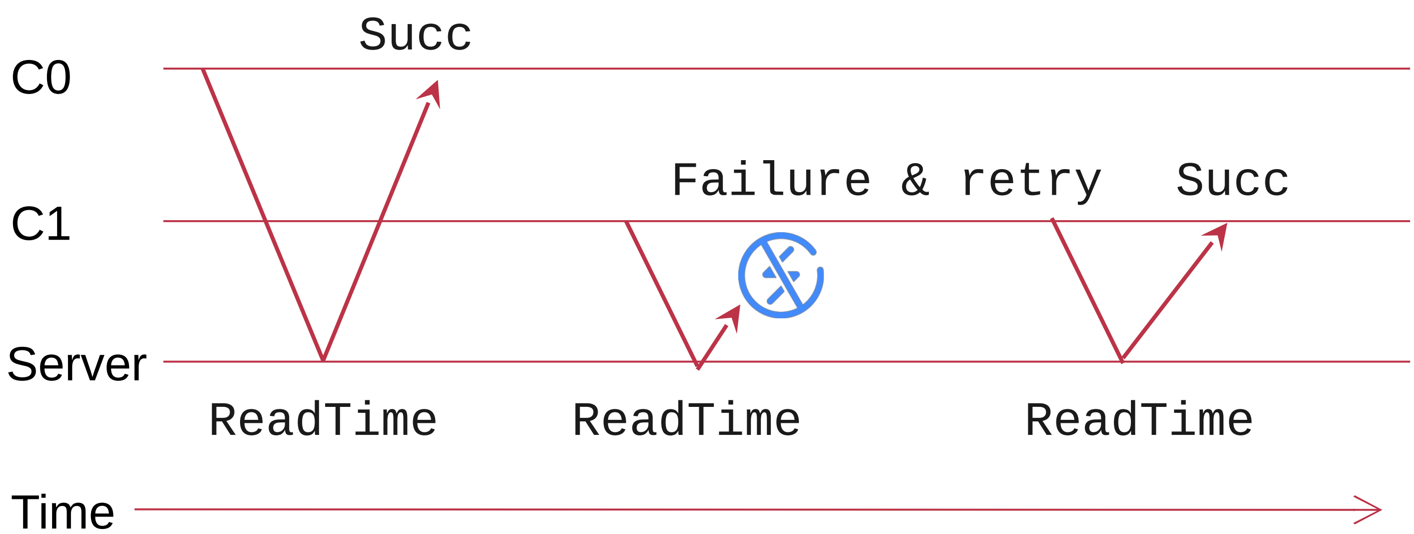
Idempotence in RPC

RPC client

```
Call(server, “ReadTime”);
```

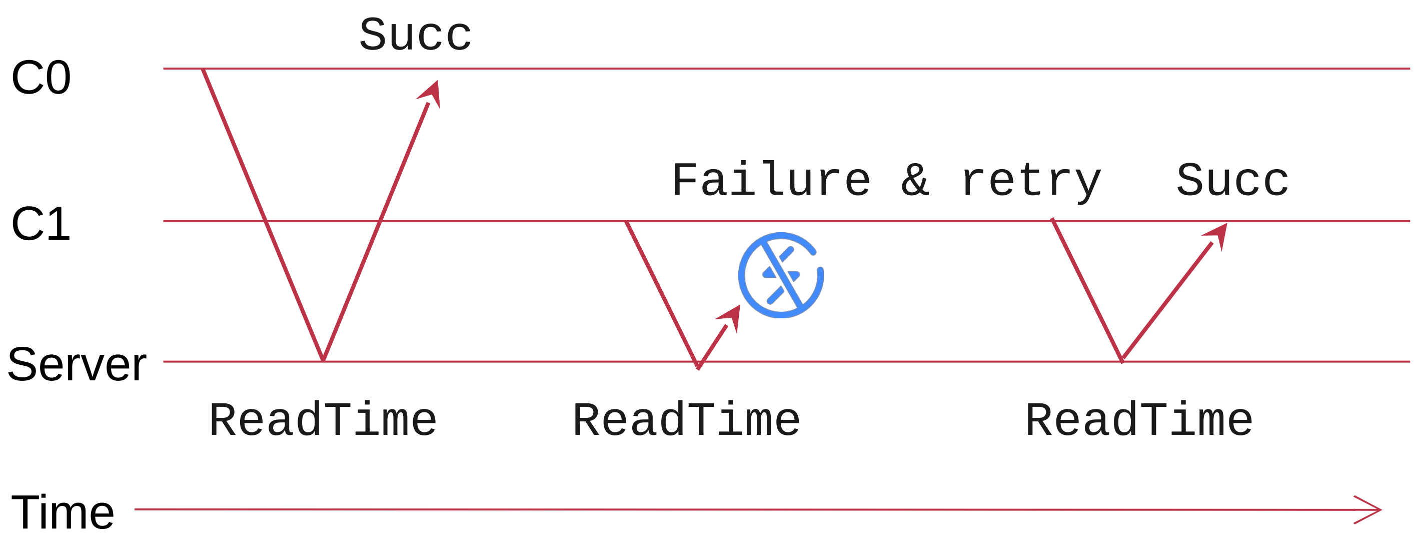
RPC server

```
ReadTime() {
  return current_time;
}
```



Idempotence in RPC

Expect ReadTime is called **twice**
Actually called **3 times**
Is it ok? Why?



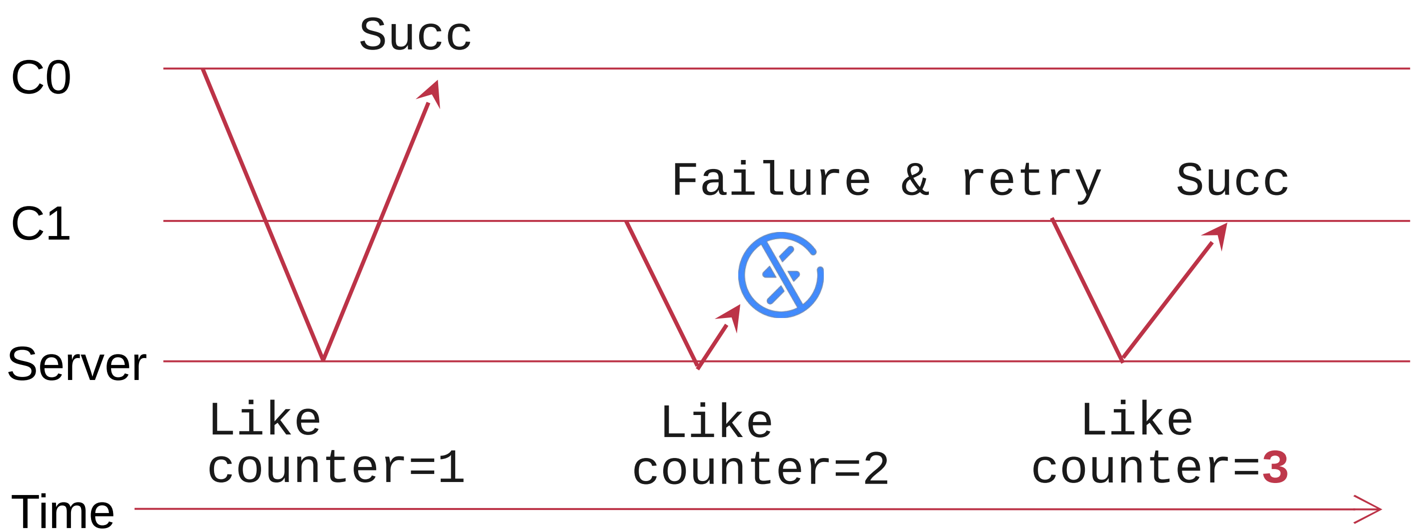
Idempotence in RPC

RPC client

```
Call(server, “Like”);
```

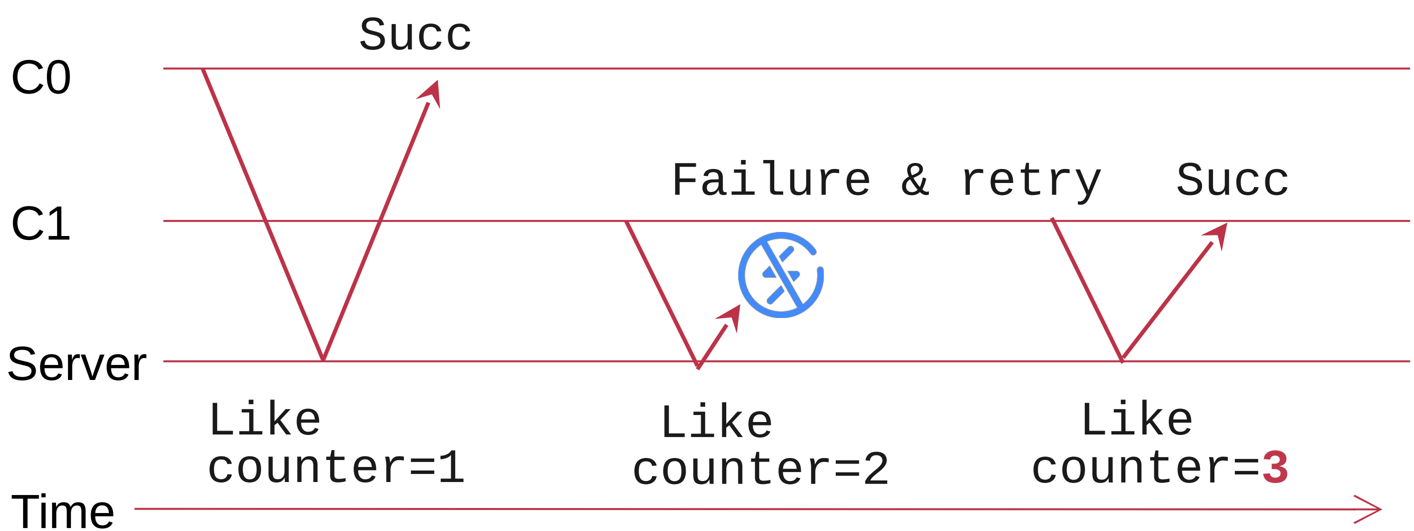
RPC server

```
Like() {
  counter++;
}
```



Idempotence in RPC

Expect Like is called **twice**
Actually called **3 times**
Is it ok? Why?
What about something like **Vote**?



RPC semantic

Understand application:

- **Idempotent**: may be run any number of times without harm
- **Non-idempotent**: those with side-effects (e.g., call i++ at the server)

When **at-least-once** is **OK**?

- If no side effect (e.g., read-only operation, ReadTime)
- If app has its own plan for detecting duplication
  - E.g., record how many times a counter has been added

Ideal RPC Semantics: exactly-once

Like single-machine function call

Implement exactly-once semantics:

- Server remembers the requests it has seen and replies to executed RPCs (need to across reboots)
- **Detect duplicates**, requests need unique IDs (XIDs)

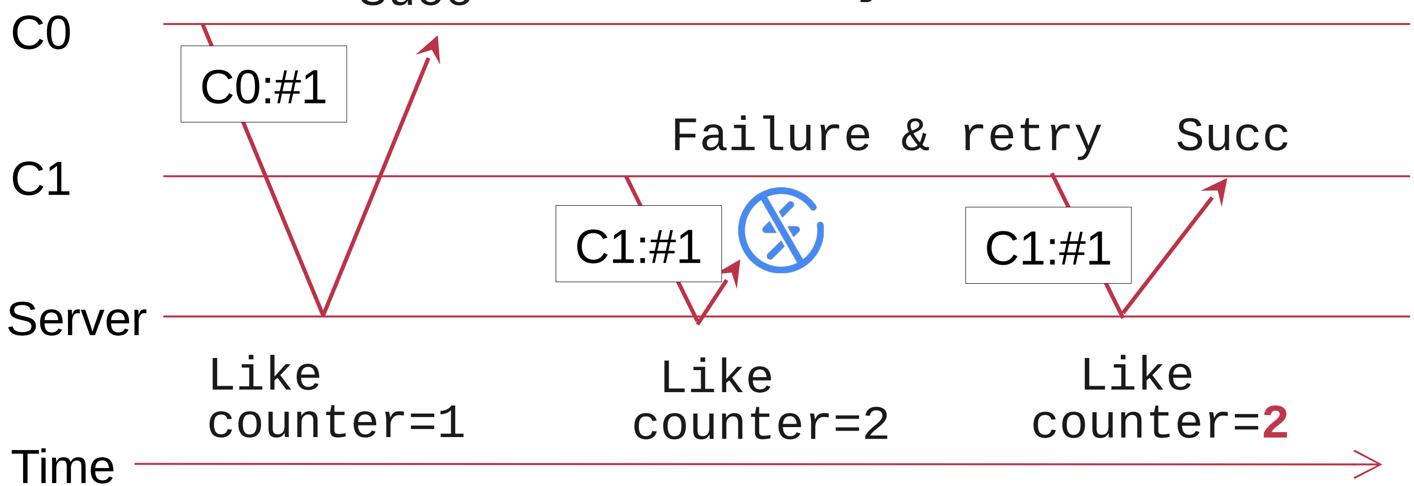
Idempotence in RPC

RPC client

```
Call(server, “Like”);
```

RPC server

```
Like(xid) {
  if not see xid {
     counter++;
  }
}
```



Ideal RPC Semantics: exactly-once

Like single-machine function call

Implement exactly-once semantics:

- Server remember the requests it has seen and replies to executed RPCs (across reboots)
- Detect duplicates, reqs need unique IDs (XIDs)

Assumption: failures are **eventually** repaired, and client retries **forever**

- How to correctly recover from failure? See later lectures

Put it all together: RPC system components

1. Standards for wire format of RPC message and data types

2. Library of routines to **marshal / unmarshal** data

3. Stub generator, or RPC compiler, to produce "stubs"
   - For client: marshal arguments, call, wait, unmarshal reply
   - For server: unmarshal arguments, call real function, marshal reply

Put it all together: RPC system components

4. Server framework:
   - Dispatch each call message to correct server stub
   - Recall each called functions ,if provide **at-most-once** semantic or **exactly-once semantic**

5. Client framework:
   - Give each reply to correct waiting thread / callback
   - Retry if timeout or server cache

6. Binding: how does client find the right server?