Winter 2025

# EECS 482: Introduction to Operating Systems

## Lecture 22: Client-Server

Prof. Ryan Huang

# Client-server

## Common way to structure a distributed application:

- Server provides some centralized service
- Client makes request to server, waits for response

## Example: web server

- Server stores and returns web pages
- Clients run Web browsers, make GET/POST requests

## Example: coke machine

- Server manages state associated with coke machine
- Clients call `client_produce()` or `client_consume()`, which send request to the server and return when done
- Server responds to client after request has been satisfied

# Client-server example

```
client_produce() {
  send produce request to server
  receive response
}
server() {
  while (1) {
    receive request
    if (produce request) {
      add coke to machine
    } else {
      take coke out of machine
    }
    send response
  }
}
```

Problems?

# Example with waiting

```
client_produce() {
  send produce request to server
  receive response
}
server() {
  while (1) {
    receive request
    if (produce request) {
      while (machine is full) wait
      add coke to machine
      signal
    } else {
      ...
    }
    send response
  }
}
```

Problems?

# Example with threads

```
server() {
  while (1) {
    receive request
    if (produce request) {
      create thread that calls server_produce()
    } else {
      create thread that calls server_consume()
    }
  }
}

server_produce() {
  lock
  while (machine is full) {
    wait
  }
  add coke to machine
  signal
  send response
  unlock
}
```

Problems?

# Example with threads

```
server() {
  while (1) {
    wait for new connection from client
    create thread that calls handle_request()
  }
}

handle_request() {
  receive request
  call server_produce() or server_consume()
}

server_produce() {
  lock
  while (machine is full) {
    wait
  }
  add coke to machine
  signal
  unlock
  send response
}
```

# Alternatives

How to lower overhead of creating threads?

How to structure the server that avoid blocking for slow operations

- Synchronous/blocking
  - Threads
- Asynchronous/non-blocking
  - Polling (via `select`)
  - Events

What are the slow operations in producer/consumer?

What are the slow operations in Project 4?

# Client-server interface

Message send/receive

Problem: too low-level and tiresome

- Need to worry about message formats
- Must wrap up information into message at source
- Must decide what to do with message at destination
- Have to pack and unpack data from messages
- May need to sit and wait for multiple messages to arrive

Messages are not a very natural programming model

Alternative?

# Procedure Calls

Procedure calls are a more natural way to communicate

- Every language supports them
- Semantics are well-defined and understood
- Natural for programmers to use

Idea: let servers export procedures that can be called by client programs

- Similar to module interfaces, class definitions, etc.
- Clients just do a procedure call as it they were directly linked with the server
- Under the cover, the procedure call is converted into a message exchange with the server

# Remote Procedure Calls (RPC)

Use procedure call as a model for distributed (remote) communication

Lots of issues

- How do we make this invisible to the programmer?
- What are the semantics of parameter passing?
- How do we bind (locate, connect to) servers?
- How do we support heterogeneity (OS, arch, language)?
- How do we make it perform well?

# RPC model

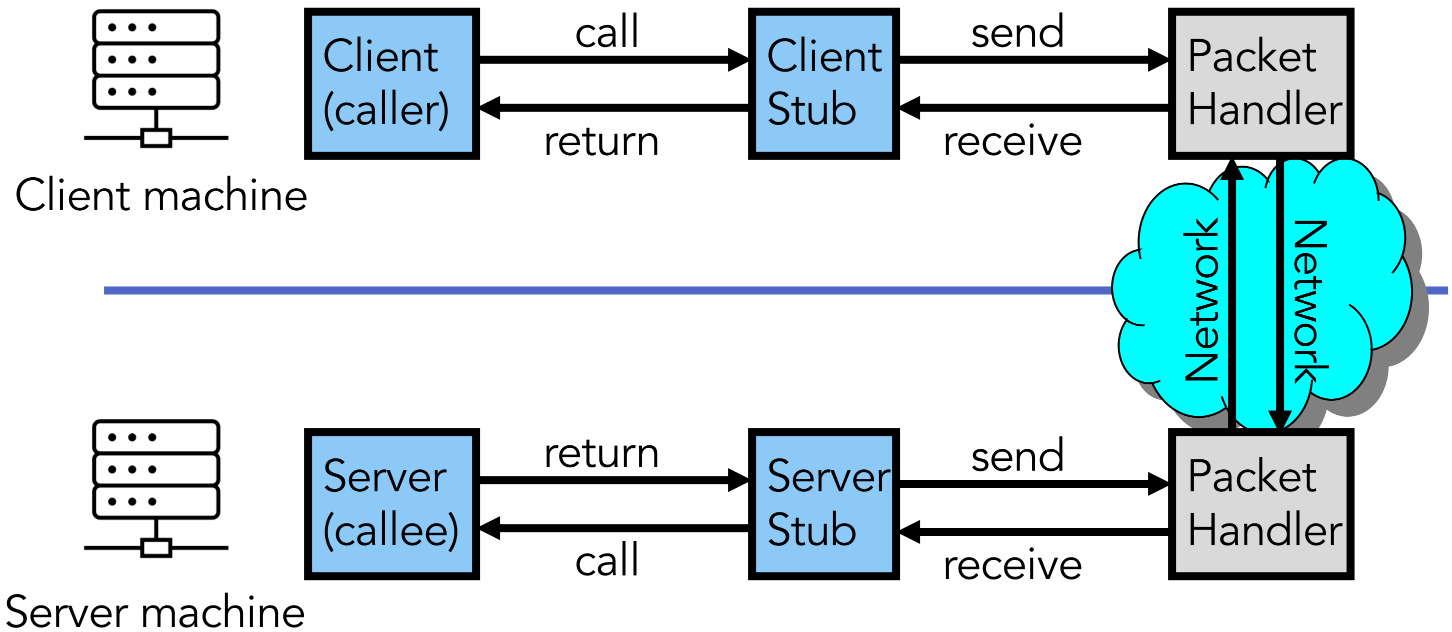
Two stubs for each server procedure: client and server

Server programmer implements the server procedures and links them with server-side stubs

Client programmer implements the client program and links it with client-side stubs

Stubs are the "glues" for managing all details of the remote communication between client and server

# RPC information flow



# RPC stubs

## Client stub:

1. Constructs message with function name and parameters
2. Sends request message to server
7. Receives response from server
8. Returns response to client

## Server stub:

3. Receives request message
4. Invokes correct function with specified parameters
5. Constructs response message with return value
6. Sends response to client stub

# Producer-consumer using RPC

## Client stub

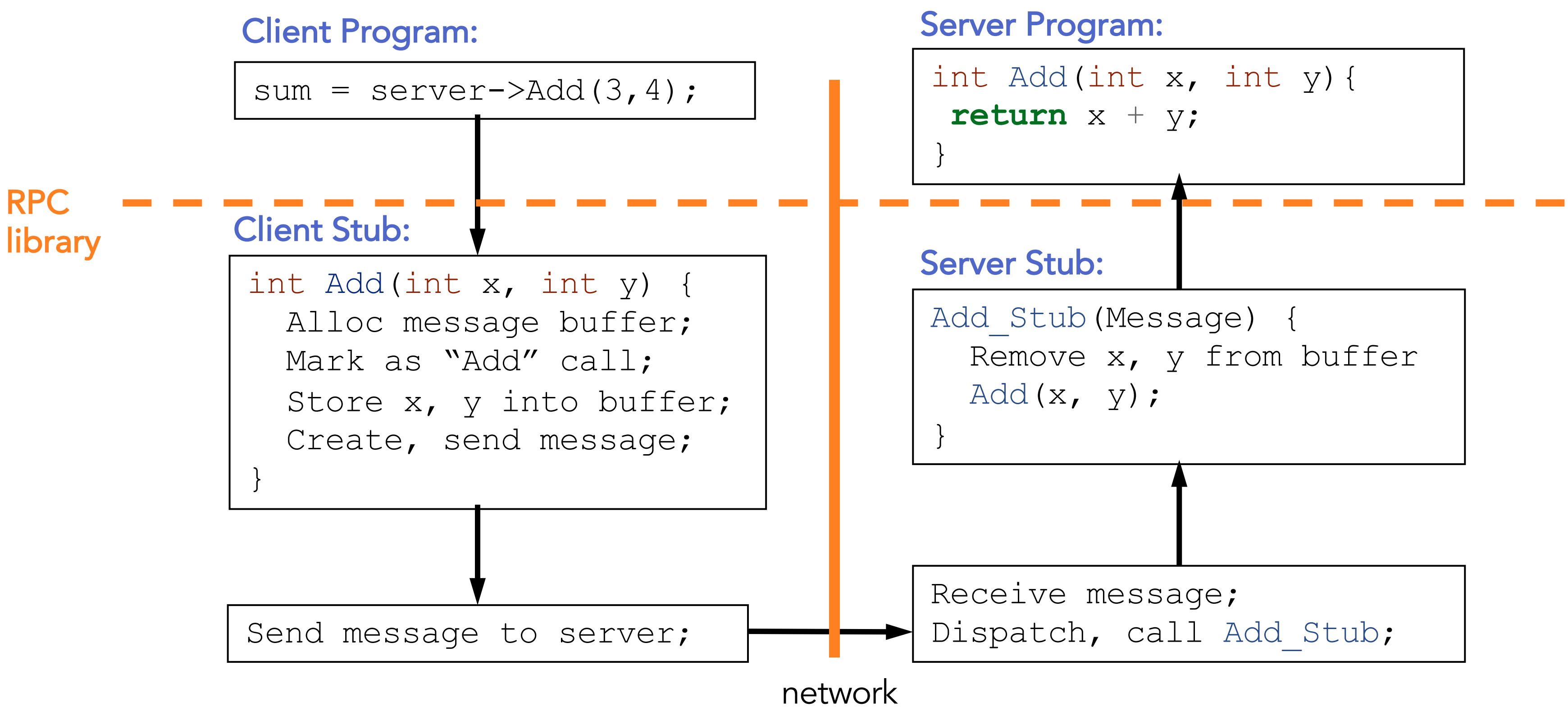
```
int produce (int n) {
  int status;
  send (sock, &n, sizeof(n));
  recv (sock, &status, sizeof(status)); // add MSG_WAITALL or loop
  return(status);
}
```

## Server stub

```
void produce_stub () {
  int n;
  int status;
  recv (sock, &n, sizeof(n));  // add MSG_WAITALL or loop
  status = produce(n);
  send (sock, &status, sizeof(status));
}
```

invoke actual `produce` function in server

# Another RPC example: call



# Another RPC example: return

Client Program:

```
sum = server->Add(3,4);
```

Server Program:

```
int Add(int x, int y){
 return x + y;
}
```

RPC library

Client Stub:

```
int Add(int x, int y) {
  Alloc message buffer;
  Mark as “Add” call;
  Store x, y into buffer;
  Create, send message;
  Remove r from reply;
  return r;
}
```

Server Stub:

```
Add_Stub(Message) {
  Remove x, y from buffer
  r = Add(x, y);
  Store r in buffer;
}
```

network

```
Return reply to stub;
```

```
Send reply to client;
```

# Generation of stubs

Stubs can be generated automatically

What do we need to know to do this?

Interface description

- Server defines an interface using an *interface definition language* (IDL)
- The IDL specifies the names, parameters, and return types for client-callable server procedures

Stub compiler (e.g., rpcgen on Linux) reads the IDL and produces the server and client stubs

# RPC transparency

RPC aims to be as transparent as possible

- Make remote procedure calls look like local procedure calls

What factors might break the transparency?

- or make it difficult to achieve transparency

# Pass a pointer to remote system?

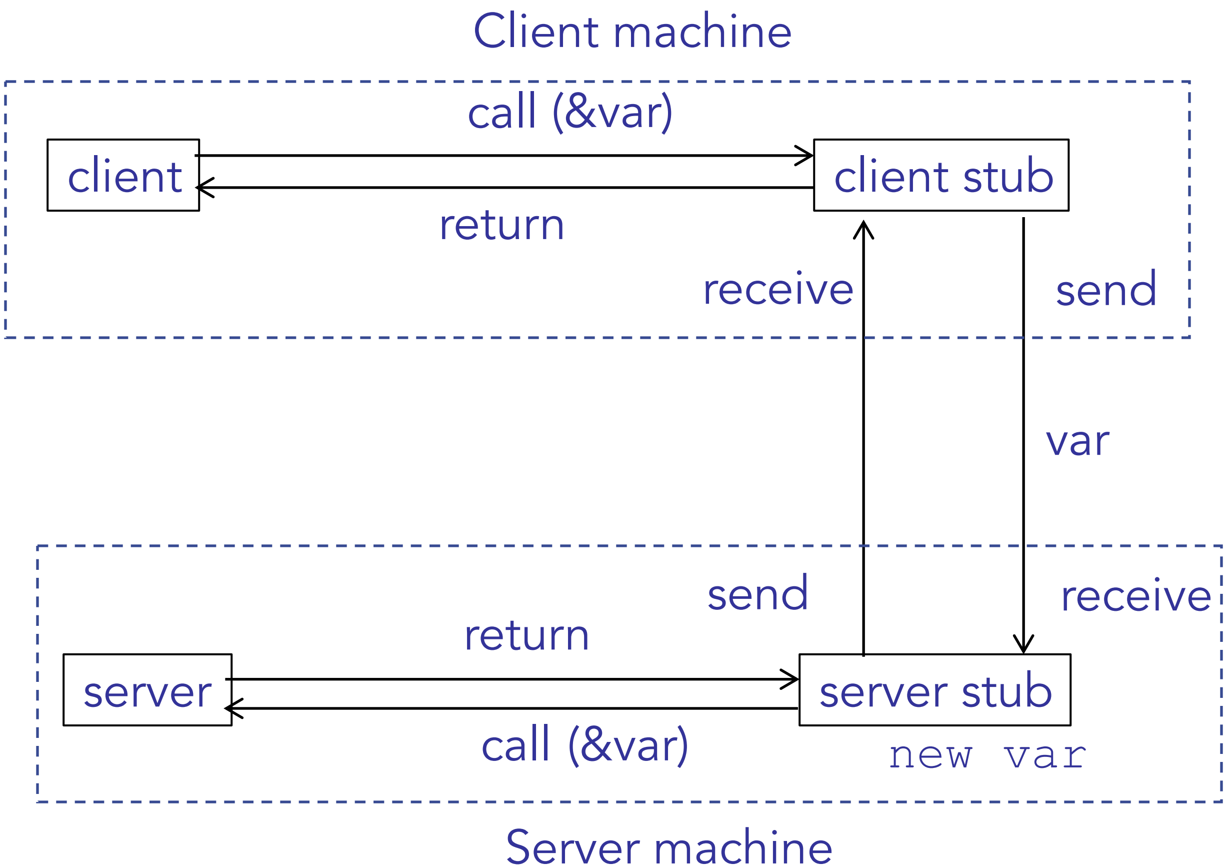
## Client stub

- Client calls client stub with pointer
- Client stub sends …

## Server stub

- Server stub receives …
- Server stub calls server with pointer

# Pass a pointer to remote system?



# Data representation

Client and server must agree on how to represent data

E.g., endianness of multi-byte integers

- In a 4-byte integer, which is the least significant byte?

| | |
|---|---|
| Byte 3 | DE |
| Byte 2 | AD |
| Byte 1 | BE |
| Byte 0 | EF |

Is this 0xDEADBEEF or
0xEFBEADDE?

E.g., character representation

E.g., C-string

# RPC binding

Binding is the process of connecting client to server
- Static: fixed at compile time
- Dynamic: performed at runtime

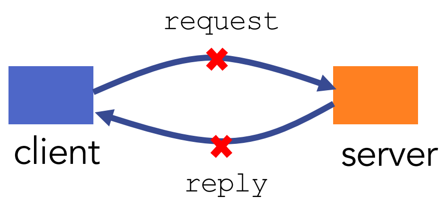
The server exports its interface when it starts up
- Identifies itself to a network name server
- Tells RPC runtime that it's alive and ready to accept calls

The client imports the server before issuing any calls
- RPC runtime uses the name server to find the location of a server and establish a connection

The import and export operations are explicit in the server and client programs

# RPC failures

RPC can have more complex failure modes than than local procedure calls




What does a failure look like to the client RPC library?

- Client never sees a response from the server
- Client does *not* know if the server saw the request
    - Maybe server/net failed just before sending reply

# RPC failure semantics

## Simplest scheme: at-least-once behavior

- RPC library waits for response for time T, if none arrives, re-send the request
- Repeat this a few times
- Still no response → return an error to the application

## Problems?

- E.g., request is "deduct $100 from bank account"

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

- If it's ok to repeat an operation, e.g., `get(key);`
- If the application has its own way of dealing with duplicates

# RPC failure semantics (2)

## Another (better) RPC behavior: at-most-once

- Server RPC code detects duplicate requests returns previous reply instead of re-running handler
- How to detect a duplicate request?
  - client includes unique ID (XID) with each request and uses the same XID for re-send
  - server checks an incoming XID in a table, if an entry is found, directly returns the reply

# RPC failure semantics (3)

## What if an at-most-once server crashes and re-starts?

- If duplicate info is in memory, server will forget and accept duplicate requests after re-start
- It could write the duplicate info to disk
- What if the server fails to restart?
- Replica server could also replicate duplicate info

## What about "exactly-once"?

- at-most-once plus unbounded retries plus fault-tolerant service

# Project 4

**Use assertions to catch errors early**

- # of free disk blocks matches file system contents?
- Are you unlocking a lock that you hold?
- Verify initial file system is not malformed

**Use showfs to verify that contents of file system match your expectations**

Test cases: cover all states of data structures (e.g., direntries array)