Client-server

- Common way to structure a distributed application:
  - Server provides some centralized service
  - Client makes request to server, then waits for response

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

- Example: Producer-consumer
  - Server manages state associated with coke machine
  - Clients call `client_produce()` or `client_consume()`, which send request to the server and return when done
  - Client requests block at the server until they are satisfied
Producer-consumer in client-server paradigm

```java
client_produce() {
    send produce request to server
    wait for response
}

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

Problems?

How to fix?
Producer-consumer in client-server paradigm

```java
client_produce() {
    send produce request to server
    wait for response
}

server() {
    receive request
    if (produce request) {
        while(machine is full) { wait }
        add coke to machine
    } else {
        take coke out of machine
    }
    send response
}
```
Producer-consumer in client-server paradigm

```java
server() {
    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
    }
    put coke in machine
    unlock
    unlock
    send response
}
```
Producer-consumer in client-server paradigm

- How to lower overhead of creating threads?
  - Maintain pool of worker threads

- There are other ways to structure the server
  - Basic goal: Account for “slow” operations

- Examples:
  - Polling (via `select`)
  - Threads + Signals
Producer-consumer in client-server paradigm

```c
client_produce() {
    send produce request to server
    wait for response
}

server() {
    receive request
    if (produce request) {
        thread(server_produce())
    } else {
        thread(server_consume())
    }
    send response
}
```
Remote Procedure Call

- Hide complexity of message-based communication from developers
- Procedure calls more natural for inter-process communication

Goals of RPC:
- Client sending request → function call
- Client receiving response → returning from function
- Server receiving request → function invocation
- Server sending response → returning to caller
RPC abstraction via stub functions on client and server

Client machine

client

client stub

call

return

receive

send

Server machine

server

server stub

call

return

send

receive

Client machine

Server machine
RPC stubs

- **Client stub:**
  - Constructs message with function name and parameters
  - Sends request message to server
  - Receives response from server
  - Returns response to client

- **Server stub:**
  - Receives request message
  - Invokes correct function with specified parameters
  - Constructs response message with return value
  - Sends response to client stub
RPC abstraction via stub functions on client and server

![Diagram of RPC abstraction]

- **Client machine**
  - **client**
  - **client stub**
  - **call**
  - **return**
  - **receive**
  - **send**

- **Server machine**
  - **server**
  - **server stub**
  - **call**
  - **return**
  - **send**
  - **receive**
Producer-consumer using RPC

- **Client stub**
  ```c
  int produce (int n) {
    int status;
    send (sock, &n, sizeof(n));
    recv (sock, &status, sizeof(status));
    return(status);
  }
  ```

- **Server stub**
  ```c
  void produce_stub () {
    int n;
    int status;
    recv (sock, &n, sizeof(n));
    status = produce(n);
    send (sock, &status, sizeof(status));
  }
  ```
Generation of stubs

- Stubs can be generated automatically
- What do we need to know to do this?

- Interface description:
  - Types of arguments and return value

- e.g. rpcgen on Linux
RPC Transparency

- RPC makes remote communication look like local procedure calls
  - Basis of CORBA, Thrift, SOAP, Java RMI, …
  - Examples in this class?

- What factors break illusion?
  - Failures – remote nodes/networks can fail
  - Performance – remote communication is inherently slower
  - Service discovery – client stub needs to bind to server stub on appropriate machine
RPC Arguments

- Can I have pointers as arguments?
- How to pass a pointer as argument?
  - Client stub transfers data at the pointer
  - Server stub stores received data and passes pointer
- Challenge:
  - Data representation should be same on either end
  - Example: I want to send a 4-byte integer:
    » 0xDE AD BE EF
    » Send byte 0, then byte 1, byte 2, byte 3
    » What is byte 0?
Endianness

- int x = 0xDE AD BE EF
- Little endian:
  - Byte 0 is 0xEF
- Big endian:
  - Byte 0 is 0xDE

- If a little endian machine sends to a big endian:
  - 0xDE AD BE EF will become 0xEF BE AD DE