EECS 482
Introduction to Operating Systems

Winter 2019

Baris Kasikci

Slides by: Harsha V. Madhyastha
## OS abstraction of network

<table>
<thead>
<tr>
<th>Hardware reality</th>
<th>Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple computers connected via a network</td>
<td>Single computer</td>
</tr>
<tr>
<td>Machine-to-machine communication</td>
<td>Process-to-process communication</td>
</tr>
<tr>
<td>Unreliable and unordered delivery of finite messages</td>
<td>Reliable and ordered delivery of byte stream</td>
</tr>
</tbody>
</table>
OS abstraction of network

- Hardware reality
  - Machine 1
  - Machine 2
  - Machine 3

- OS abstraction
  - Process A
  - Process B
  - Process C
Changing communication from inter-machine to inter-process

- Every process thinks it has its own:
  - Multiprocessor (threads)
  - Memory (address space)
  - Network interface cards (sockets)

- **Socket**
  - Virtual network interface card
  - Endpoint for communication
  - NIC named by MAC address; socket named by “port number” (via bind)
  - Programming interface: BSD sockets
OS multiplexes multiple sockets onto a single NIC

- UDP (user datagram protocol): IP + sockets
- TCP (transmission control protocol): IP + sockets + reliable, ordered streams

April 1, 2019
Ordered messages

- Hardware interface: Messages can be re-ordered by IP
  - Sender: A, B
  - Receiver: B, A
- Application interface: Messages received in order sent

How to provide ordered messages?

Ordering of messages per-"connection"
- TCP: process opens connection (via `connect`), sends sequence of messages, then closes connection
  - Sequence number specific to a socket-to-socket connection
Ordered messages

- Example:
  - Sender sends 0, 1, 2, 3, 4, ...
  - Receiver receives 0, 1, 3, 2, 4, ...

- How should receiver deal with reordering?
Reliable messages

- Hardware interface: Messages can be dropped, duplicated, or corrupted
- Application interface: Each message is delivered exactly once without corruption

- How to fix a dropped message?
- How does sender know message was dropped?
- Does timing out mean the message was dropped?
Reliable messages

- How to deal with duplicate messages?
- How to deal with corrupted messages?

Transform:
  - Corrupted messages → dropped messages
  - Potential dropped messages → potential duplicates
- Solve duplicates by dropping duplicate messages
Byte streams

- Hardware interface: Send/receive messages
- Application interface: Abstraction of data stream
- TCP: Sender sends messages of arbitrary size, which are combined into a single stream
- Implementation
  - Break up stream into fragments
  - Sends fragments as distinct messages
  - Reassembles fragments at destination
Message boundaries

- TCP has no message boundaries (unlike UDP)
  - Example: Sender sends 100 bytes, then 50 bytes; Receiver could receive 1-150 bytes
- Receiver must loop until all bytes received
- How to know # of bytes to receive?
Project 4

- **Use assertions** to catch errors early
  - No. 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

- There are no boundaries in TCP byte streams

- **A char** is not a string!
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?

April 1, 2019
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
}
```

April 1, 2019
Producer-consumer in client-server paradigm

```c
server() {

}

server_produce() {
    lock
    while (machine is full) {
        wait
    }
    put coke in machine
    unlock
    send response
}
```

April 1, 2019
Producer-consumer in client-server paradigm

- How to lower overhead of creating 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

```java
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

Server machine

server

server stub

April 1, 2019
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

April 1, 2019
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

April 1, 2019
Producer-consumer using RPC

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

- **Server stub**
  ```
  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?
RPC Arguments

- Can I have pointers as arguments?
- How to pass a pointer as argument?

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 (Intel):
  - Byte 0 is 0xEF
- Big endian (Power PC):
  - Byte 0 is 0xDE

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