Introduction to

Parallel and Distributed Programming

CS2
Communication

Dr. Saikishor Jangiti

Agenda

- Lab 1
- IPC fundamentals
- UNIX sockets
- Remote procedural call

Lab 1

- **Discover nodes and status to create a group/cluster**
  a. Set of nodes must discover other nodes and form a group that can communicate and track membership status and health when added to a cluster.
  b. Assume a master-slave architecture and designate a master node to drive this status collection and state maintenance. Slaves send heartbeats to their master. Why shouldn't the master initiate this? Helps with scale/fault tolerance.
  c. Master collects resource status on slaves, current utilisation etc. Master detects a slave crash and lets others know.
  d. Demonstrate using multi-node communication to implement a simple cluster management protocol for membership and health/resource status.
  e. Use secures transport mechanisms (SSL), so nodes cannot impersonate. Use basic secure socket programming, RPC etc.
  f. Slaves can crash and rejoin.

IPC Fundamentals

- What is IPC?
  - Mechanisms to transfer data between processes
- Why is it needed?
  - Not all procedures can be easily built in a single process

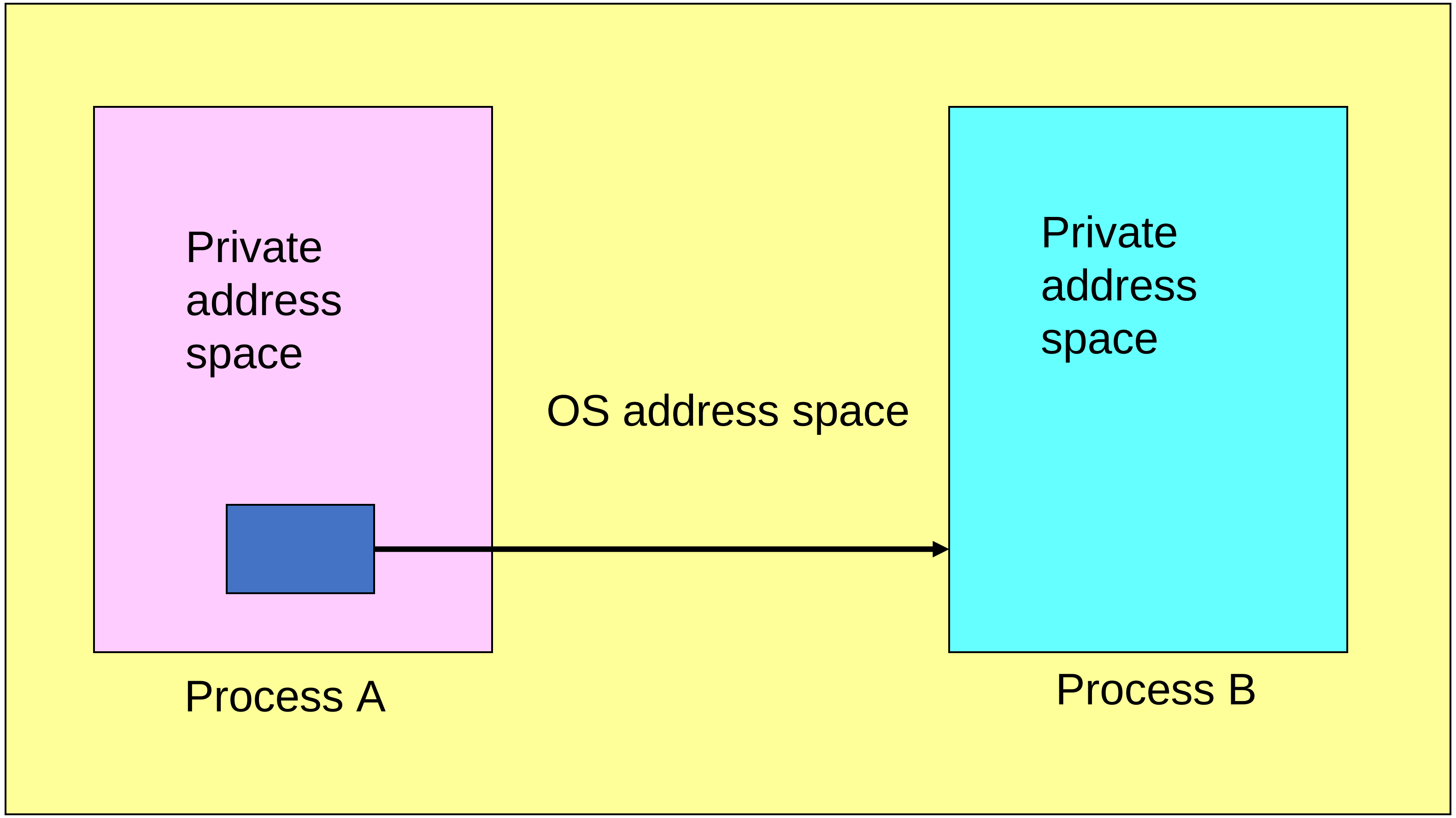
Why do processes communicate?

- To share resources
- Client/server paradigms
- Inherently distributed applications
- Reusable software components
- Other good software engineering reasons

The Basic Concept of IPC

- A process needs to send data to a receiving process
  - Sender wants to avoid details of receiver’s condition
  - Receiver wants to get the data in an organized way

IPC from the OS Point of View



Fundamental IPC Problem for the OS

- Each process has a private address space
- Normally, no process can write to another process's space
- How to get important data from process A to process B?

OS Solutions to IPC Problem

- Fundamentally, two options

1. Support some form of shared address space
   - Shared memory
2. Use OS mechanisms to transport data from one address space to another
   - Files, messages, pipes, RPC

Fundamental Differences in OS Treatment of IPC Solutions

- Shared memory
  - OS has job of setting it up
  - And perhaps synchronizing
  - But not transporting data
- Messages, etc
  - OS involved in every IPC
  - OS transports data

Desirable IPC Characteristics

- Fast
- Easy to use
- Well defined synchronization model
- Versatile
- Easy to implement
- Works remotely

IPC and Synchronization

- Synchronization is a major concern for IPC
  - Allowing sender to indicate when data is transmitted
  - Allowing receiver to know when data is ready
  - Allowing both to know when more IPC is possible

IPC and Connections

- IPC mechanisms can be connectionless or require connection
  - Connectionless IPC mechanisms require no preliminary setup
  - Connection IPC mechanisms require negotiation and setup before data flows

Connectionless IPC

- Data simply flows
- Typically, no permanent data structures are shared in OS by the sender and receiver

+ Good for quick, short communication

- Less efficient for large, frequent communications

- Each communication takes more OS resources per byte

Connection-oriented IPC

- Sender and receiver pre-arrange IPC delivery details
- OS typically saves IPC-related info for them
- Pros/cons pretty much the opposites of connectionless IPC

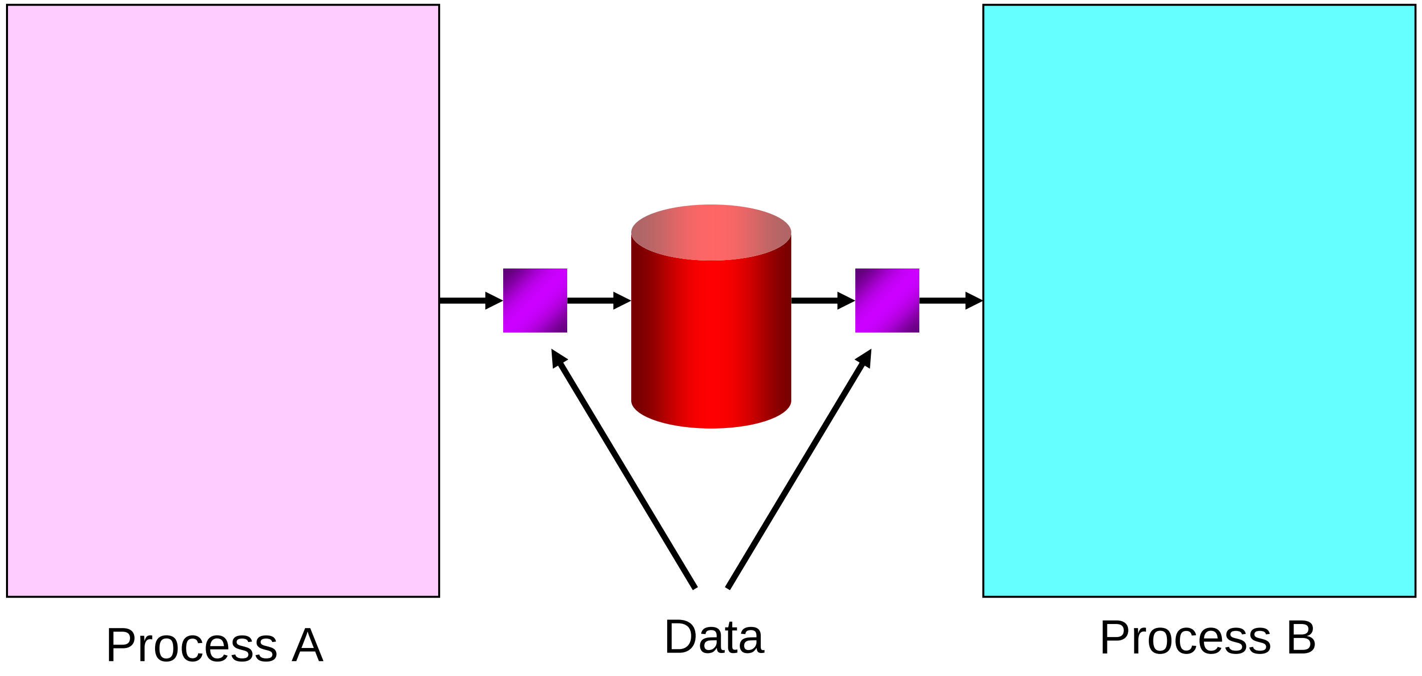
Basic IPC Mechanisms

- File system
- Message-based
- Procedure call
- Shared memory

IPC Through the File System

- Sender writes to a file
- Receiver reads from it
- But when does the receiver do the read?
    - Often synchronized with file locking or lock files
- Special types of files can make file-based IPC easier

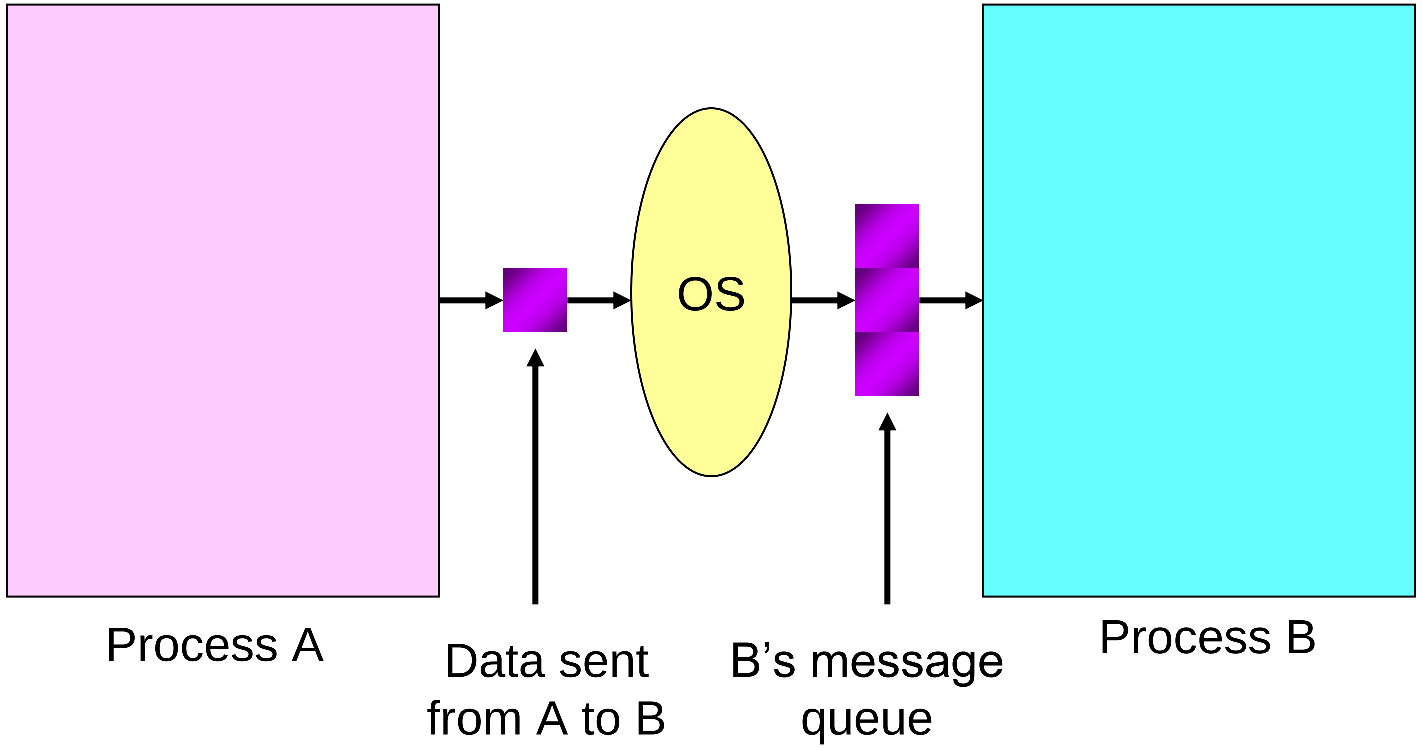
File IPC Diagram



Message-based IPC

- Sender formats data into a formal message
    - With some form of address for receiver
- OS delivers message to receiver's message input queue (might signal too)
- Receiver (when ready) reads a message from the queue
- Sender might or might not block

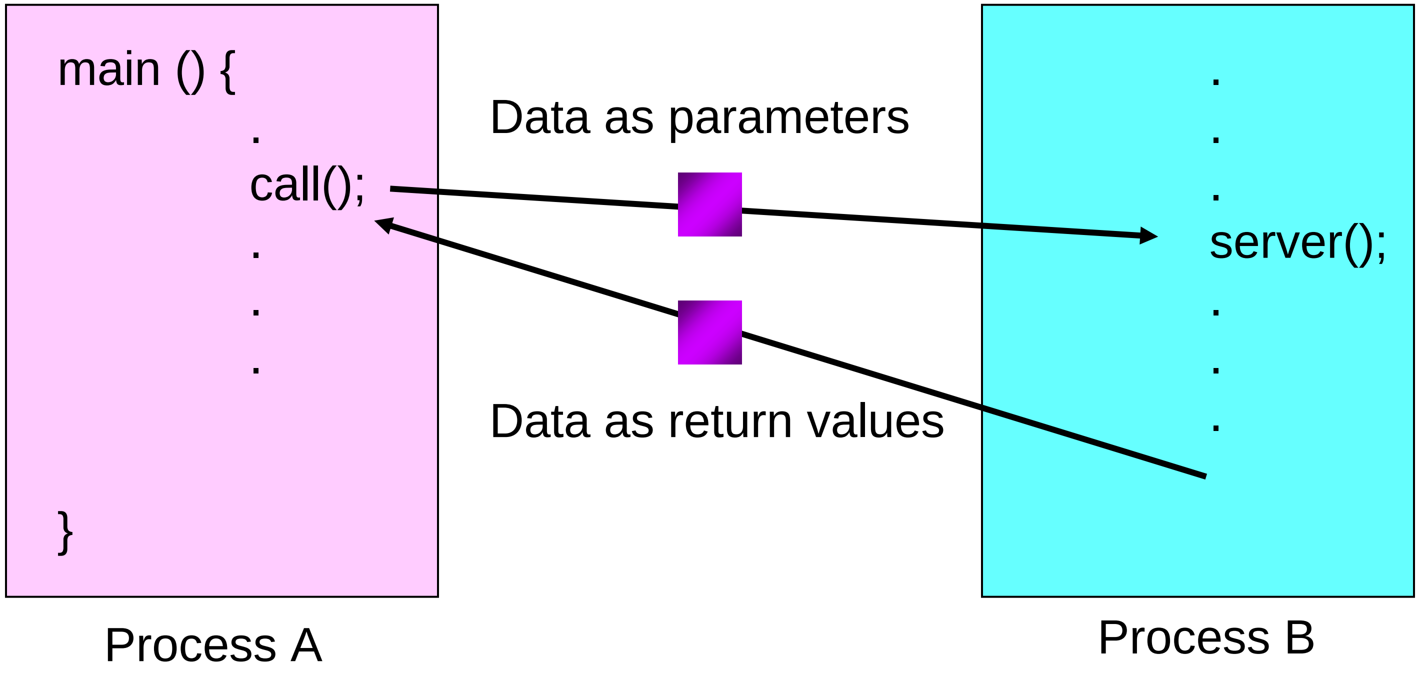
Message-based IPC Diagram



Procedure Call IPC

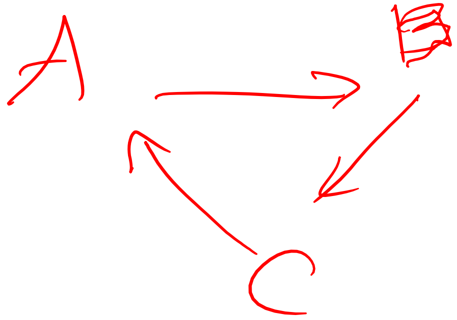
- Uses same procedure call interface as intraprocess
  - Data passed as parameters
  - Info returned via return values
- Complicated since destination procedure is in a different address space
- Generally, calling procedure blocks till call returns

Procedure Call IPC Diagram

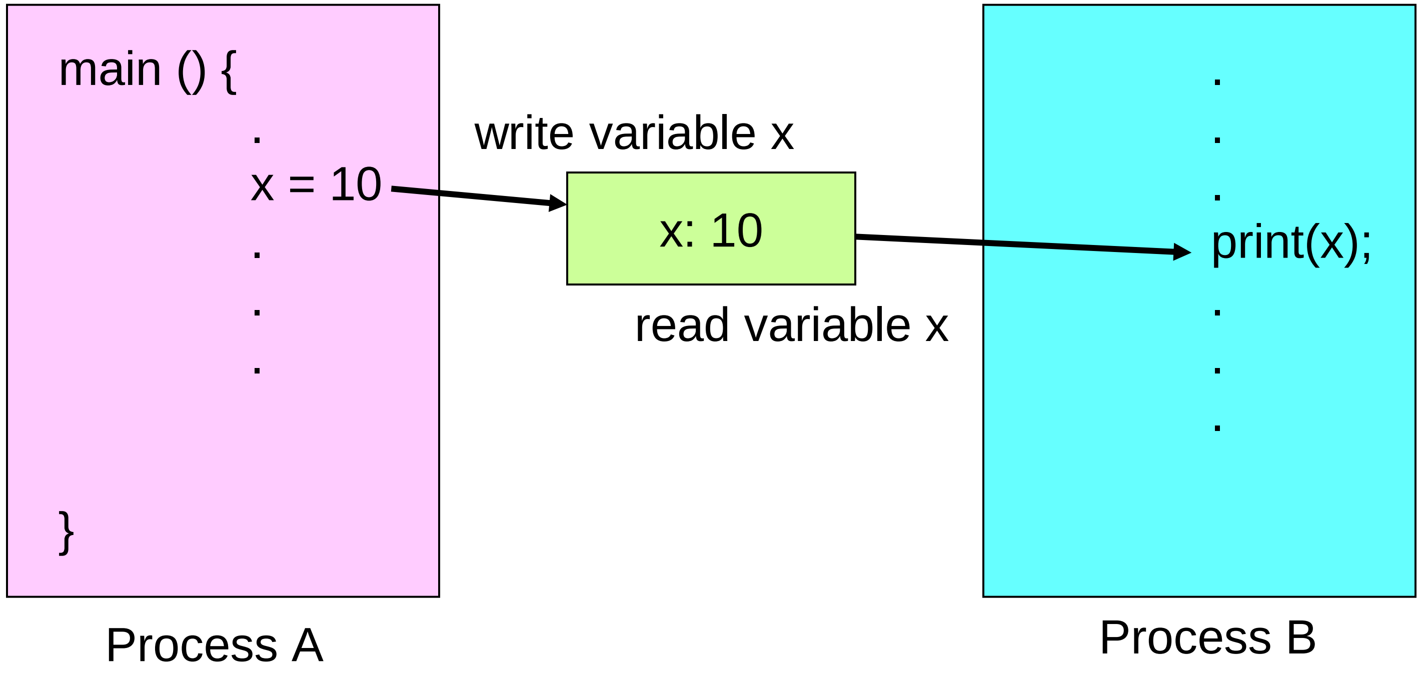


Shared Memory IPC

- Different processes share a piece of memory
  - Either physically or virtually
- Communications via normal reads/writes
- May need semaphores or locks
  - In or associated with the shared memory



Shared Memory IPC Diagram

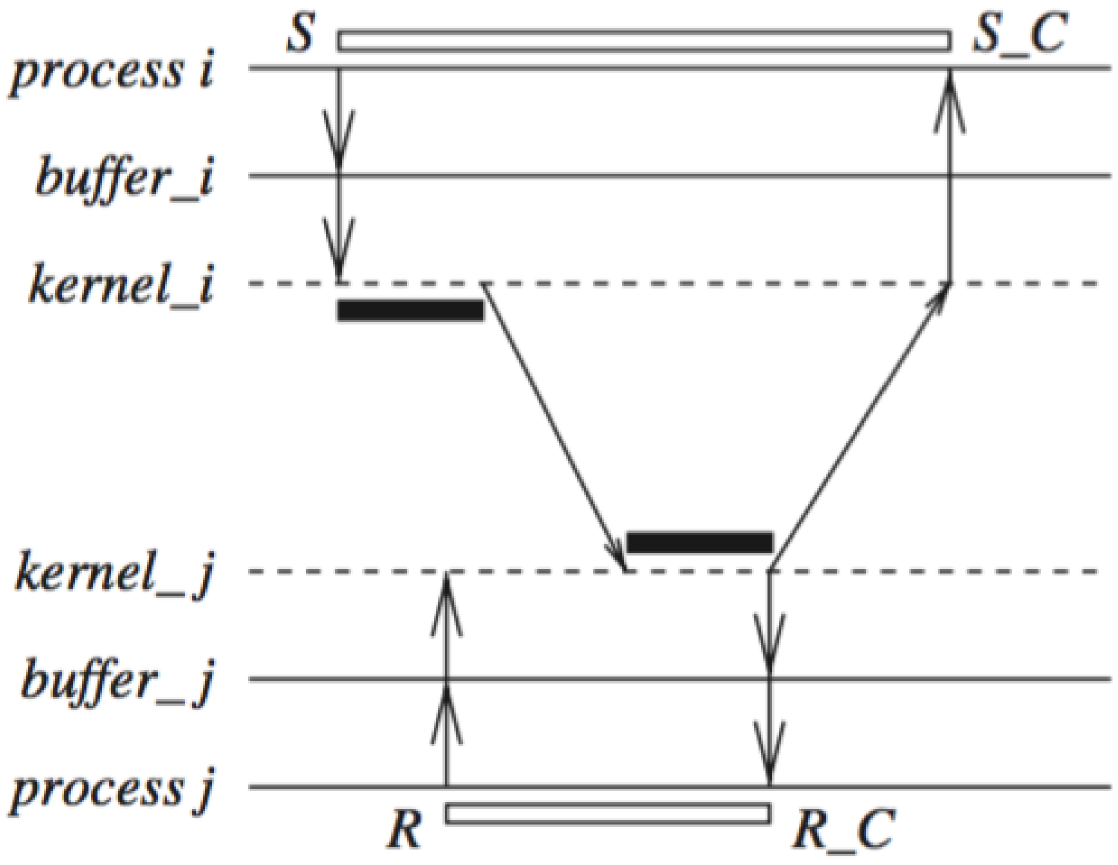


Synchronizing in IPC

- How do sending and receiving process synchronize their communications?
- Many possibilities
  - Based on which process block when
- Examples that follow in message context, but more generally applicable

Blocking Send, Blocking Receive

- Both sender and receiver block
  - Sender blocks till receiver receives
  - Receiver blocks until sender sends
  - Often called message *rendezvous*

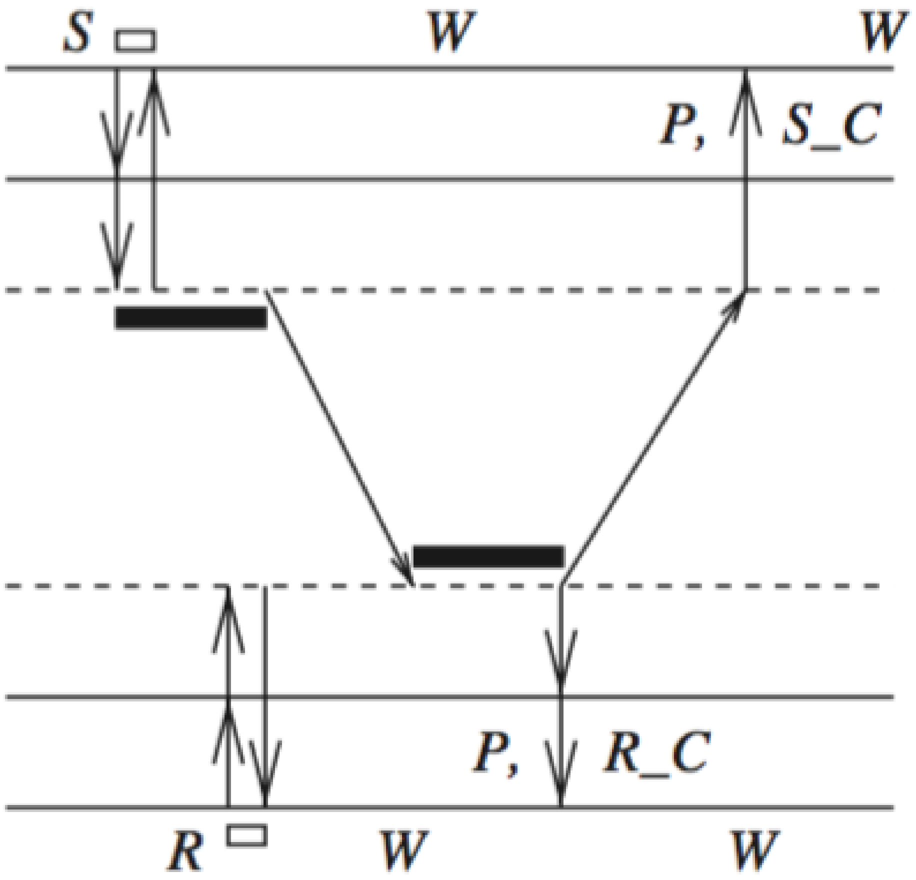


Non-Blocking Send, Blocking Receive

- Sender issues send, can proceed without waiting to discover fate of message
- Receiver waits for message arrival before proceeding
  - Essentially, receiver is message-driven

Non-Blocking Send, Non-Blocking Receive

- Neither party blocks
- Sender proceeds after sending message
- Receiver works until message arrives
  - Either receiver periodically checks in non-blocking fashion
  - Or some form of interrupt delivered



Addressing in IPC

- How does the sender specify where the data goes?
- In some cases, the mechanism makes it explicit (e.g., shared memory and RPC)
- In others, there are options

Direct Addressing

- Sender specifies name of the receiving process
- Using some form of unique process name
- Receiver can either specify name of expected sender
  - Or take stuff from anyone

Indirect Addressing

- Data is sent to queues, mailboxes, or some other form of shared data structure
- Receiver performs some form of read operations on that structure
- Much more flexible than direct addressing

Duality in IPC Mechanisms

- Many aspects of IPC mechanisms are duals of each other
  - These mechanisms have the same power
    - First recognized in context of messages vs. procedure calls
  - IPC mechanisms can be simulated by each other

So which IPC mechanism to build/choose/use?

- Depends on the model of computation
  - And on the philosophy of the user
- In particular cases, hardware or existing software may make one perform better

Typical UNIX IPC Mechanisms

- Different versions of UNIX introduced different IPC mechanisms
  - Pipes
  - Message queues
  - Semaphores
  - Shared memory
  - Sockets
  - RPC

Pipes

- Only IPC mechanism in early UNIX systems (other than files)
  - Uni-directional
  - Unformatted
  - Uninterpreted
  - Interprocess byte streams
- Accessed in file-like way

Pipe Details

- One process feeds bytes into pipe
  - A second process reads the bytes from it
- Potentially blocking communication mechanism
- Requires close cooperation between processes to set up
  - Named pipes allow more flexibility

Pipes and Blocking

- Writing more bytes than pipe capacity blocks the sender
  - Until the receiver reads some of them
- Reading bytes when none are available blocks the receiver
  - Until the sender writes some
- Single pipe with bounded capacity can't cause deadlock

UNIX Message Queues

- Introduced in System V Release 3 UNIX
- Like pipes, but data organised into messages
- Message components include
  - Type identifier
  - Length
  - Data

Semaphores

- Also introduced in System V Release 3 UNIX
- Mostly for synchronization only
  - Since they only communicate one bit of information
- Often used in conjunction with shared memory

UNIX Shared Memory

- Also introduced in System V Release 3
- Allows two or more processes to share some memory segments
- With some control over read/write permissions
- Often used to implement threads packages for UNIX

Sockets

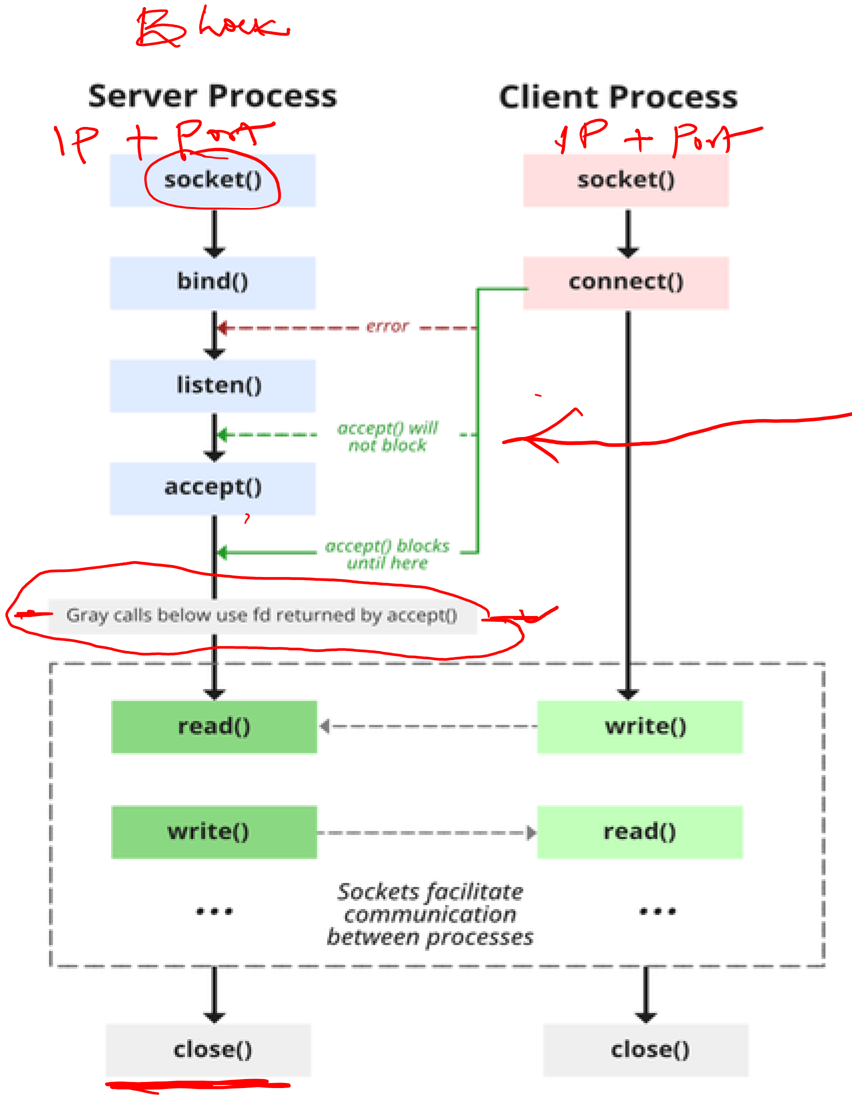
- Introduced in 4.3 BSD
- A socket is an IPC channel with generated endpoints
- Great flexibility in its characteristics
  - Intended as building block for communication
- Endpoints established by the source and destination processes

UNIX Remote Procedure Calls

- Procedure calls from one address space to another
  - On the same or different machines
- Requires cooperation from both processes
- In UNIX, often built on sockets
- Often used in client/server computing

More on Sockets

- Created using the socket() system call
- Specifying domain, type, and protocol
- Sockets can be connected or connectionless
  - Each side responsible for proper setup/access

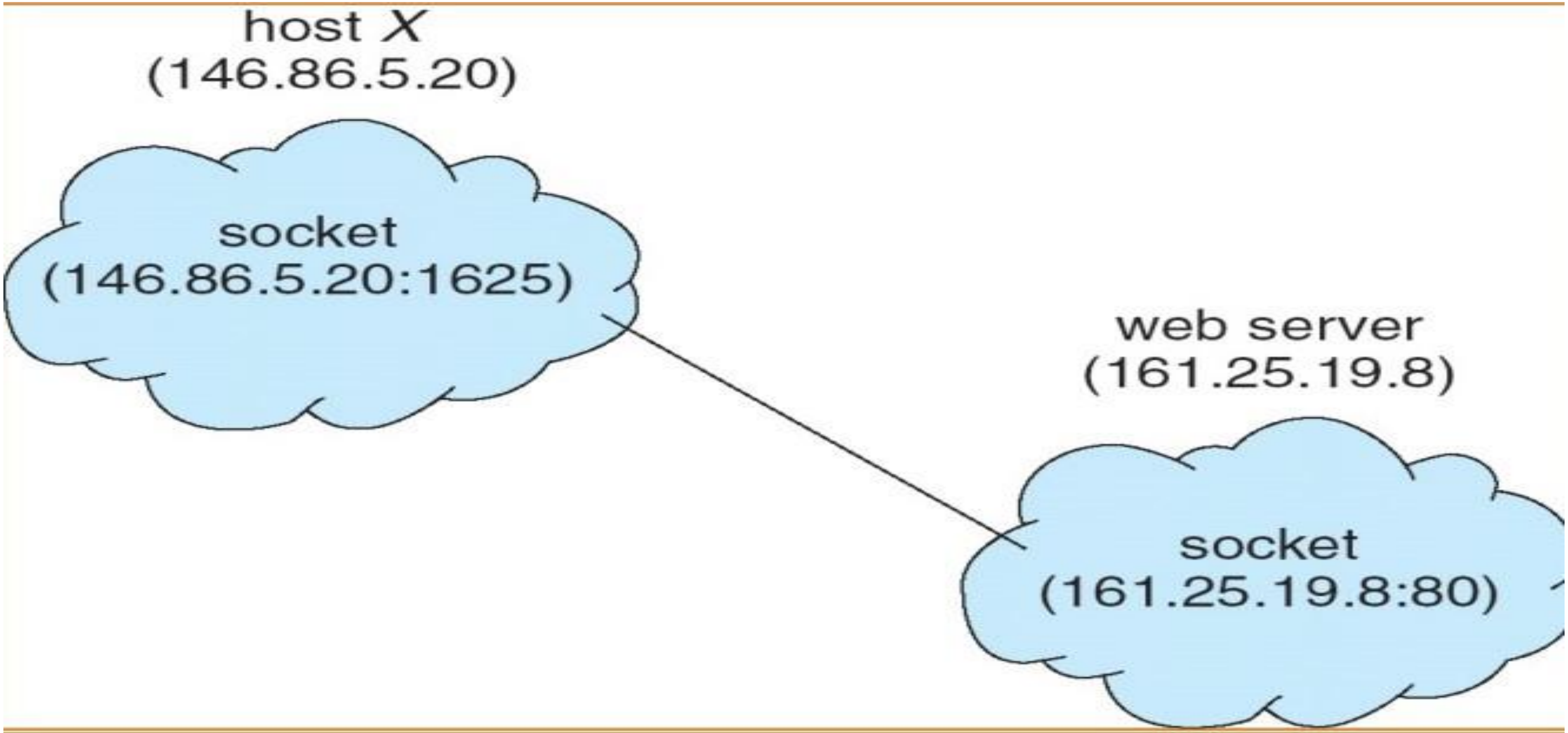


State diagram for server and client model of Socket

Sockets

- A socket is defined as an endpoint for communication.
- Concatenation of IP address and port.
- The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
- Communication consists between a pair of sockets
- Low level: Sends/receives a stream of bytes
- Sockets are either connection-oriented (TCP) or connectionless (UDP)

Socket Communication



Socket Domains

- the socket domain describes a protocol family used by the socket
  - Generally related to the address family
- Domains can be:
  - Internal protocols
  - Internet protocols
  - IMP (interface message processors) link layer protocols

Socket Types

- The socket type describes what the socket does
- Several types are defined
  - SOCK_STREAM
  - SOCK_DGRAM
  - SOCK_SEQPACKET
  - SOCK_RAW
  - SOCK_RDM (reliable data gram w/o ordering guarantees)

Socket Protocols

- This parameter specifies a particular protocol to be used by the socket
- Must match other parameters
  - Not all protocols usable with all domains and types
- Generally, only one protocol per socket type available

Some Examples of Sockets

- Socket streams
- Socket sequential packets
- Socket datagrams

Socket Streams

- Of type SOCK_STREAM
- Full-duplex reliable byte streams
- Like 2-way pipes
- Requires other side to connect

Socket Sequential Packets

- Similar to streams
    - But for fixed-sized packets
- So reads always return a fixed number of bytes
    - Allow easy use of buffers

Socket Datagrams

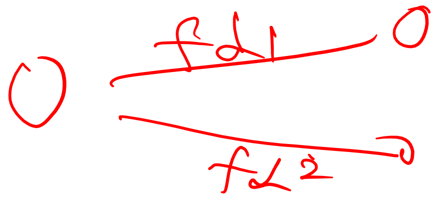
- Like sequential packets
    - But non-reliable delivery
    - Which implies certain simplifications
    - And lower overhead
- send(), rather than write(), used to send data

Socket Options

- Connection or connectionless
- Blocking or non-blocking
- Out-of-band information
- Broadcast
- Buffer sizes (input and output)
- Routing options
- And others

Binding Sockets

- *Binding* prepares a socket for use by a process
- Sockets are typically bound to local names
  - For IPC on a single machine
- Often accessed using descriptors
- Requires clean-up when done
- Binding can be to IP addresses, as well



Connecting to Sockets

- Method for setting up the receiving end of a socket
- In local domain, similar to opening file
- Multiple clients can connect to a socket
    - Program establishing socket can limit connections

Remote Procedural Call

- Method of calling procedures in other address spaces
- Either on the same machine
  - Or other machines
- Attempts to provide interface just like local procedure call
- Request/reply communications model

RPC Case Studies

- RPC in Cedar
- UNIX RPC

Semantics of RPC

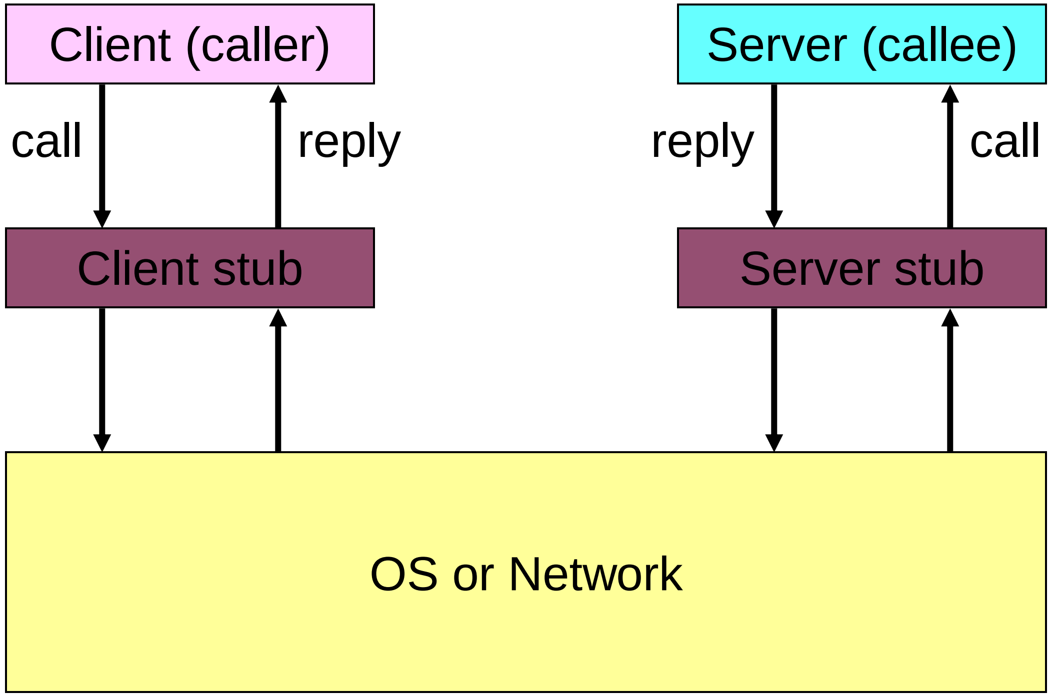
- Similar to regular procedure call

1. Calling procedure blocks
2. Parameters transferred to called procedure
3. Called procedure computes till it returns
4. Return value delivered to calling procedure
5. Calling procedure continues

High-Level RPC Mechanics

- Hide details from applications
- Clients pass requests to stub programs
- Client-end stub sends request to server stub
- Server-end stub calls user-level server
- Results travel in reverse direction
- Network transport or OS actually moves data

Diagram of RPC in Action



What do the stubs do?

- Stubs handle complex details like:
  - Marshaling arguments
  - Message construction
  - Data format translation
  - Finding the server process

Setting Up RPC

- Caller must have a way to find the called procedure
- But it can’t be found at link time
    - Unlike local procedure
- Potential servers must make their presence known

Registering Servers

- Either register the server at a “well-known” port
- Or register it with a name server
  - Which in turn is at a “well-known” port
- Calling procedure “addresses” RPC to that port

Binding to a Service

- A client process *binds* to the service it wants
- Two major sub-problem:
  - Naming
  - Location

Binding: The Naming Problem

- How does a caller name the server program?
- Depends on the RPC mechanism
  - And perhaps the protocol or type within it
- Do you name the server explicitly?
- Or do you name the service and let some other authority choose which server?

Binding: The Location Problem

- Where is the remote server?
- Some naming schemes make it explicit
    - Some don’t
- If it’s not explicit, system must convert symbolic names to physical locations

Binding in Cedar RPC

- Client applications bind to a symbolic name
  - Composed of:
    - Type
    - Instance
- Type is which kind of service
- Instance is which particular implementer

Locating Cedar RPC Service

- Names do not contain physical location
- So Cedar consults a database of services
- Services register with database
- Binding call automatically looks up location in database

Binding in UNIX RPC

- bind() system call used by servers
  - Allows servers to bind naming/location information to particular socket
- connect() system call used by clients
  - Using information similar to that bound by the server
- Automatic code generation hides details

UNIX Binding Information

- Like most socket operations, it's flexible
- Fill in all or part of a socket address data structure
- Create an appropriate socket
- Then call ***bind*** to link socket to address information
- ***connect*** works similarly

UNIX Binding Example

- On server side,

```
struct sockaddr_un sin;
int sd;

strcpy(sin.sun_path,”./socket”);
sd = socket(AF_UNIX, SOCK_STREAM, 0);
bind(sd, &sin, sizeof(sin));
```

UNIX Binding Example, Con't

- For client side,

```
struct sockaddr_un sin;
int sd;

strcpy(sin.sun_path, “./socket”);
sd = socket(AF_UNIX, SOCK_STREAM, 0);
connect(sd, &sin, sizeof(sin));
```

Locating Remote Services in UNIX RPC

- Similar to Cedar methods
- Register services with the *portmapper*
- The portmapper is typically called through automatically generated code
- the portmapper runs on each machine
- Another service (e.g., NIS) deals with intermachine requests

Using RPC

- Once it’s bound, how does the client use RPC?
- Just call the routines
  - As if they were local
- And the results come back

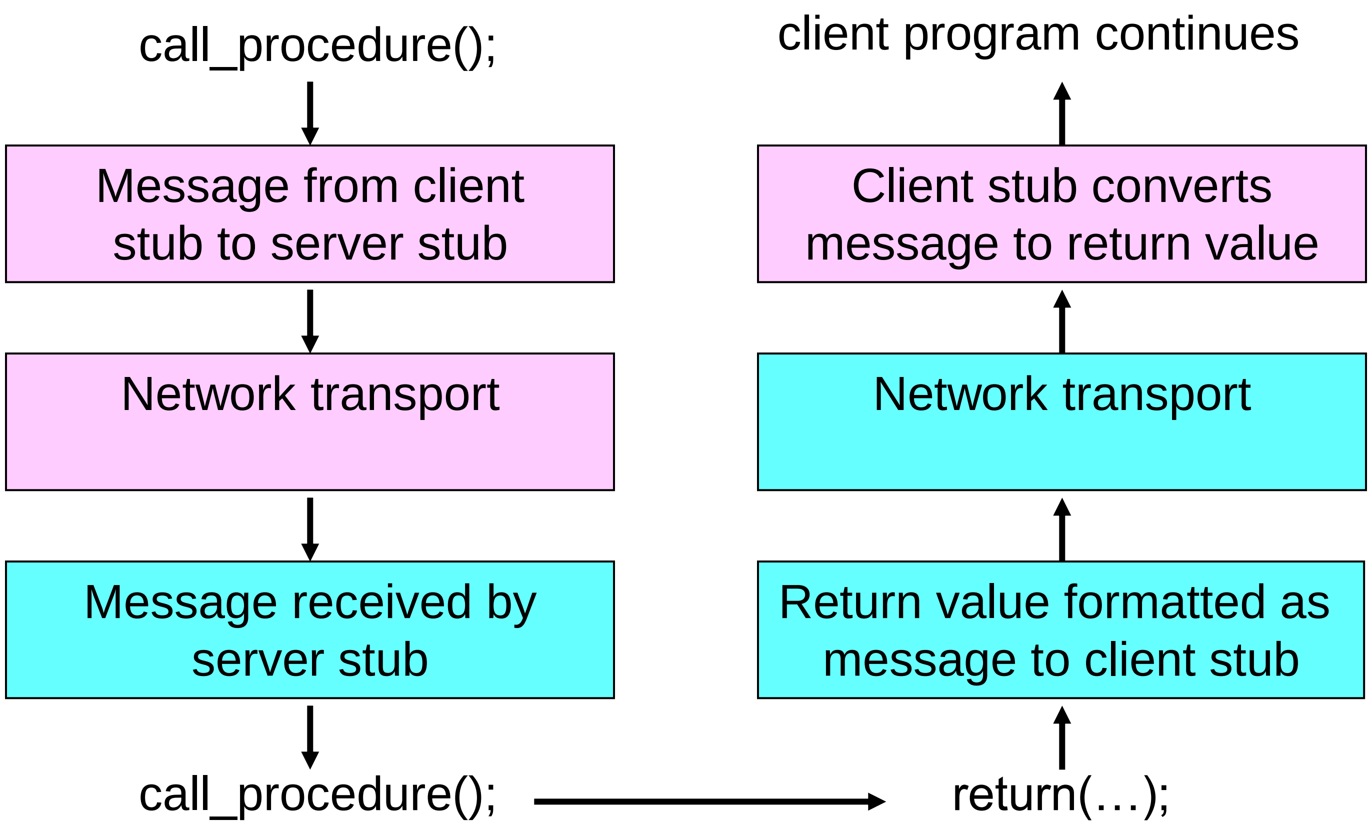
What's happening under the covers?

- When a client calls a remote routine, he really calls a local stub program
- Stub program packages up request to remote server
- And sends it to local transport code
- When reply arrives, local transport code returns results to stub
- Which returns to client program

What happens at the server side?

- A request comes in to the RPC transport code
- It routes it to the appropriate server stub
- Which converts it into a local procedure call
- Which is made within the context of the server

Conceptual Diagram of RPC



Transport for RPC

- In Cedar, special-purpose RPC transport protocol
- In UNIX RPC, can use either UDP or TCP for transport
    - Typically, a protocol is chosen automatically by the stub generator program

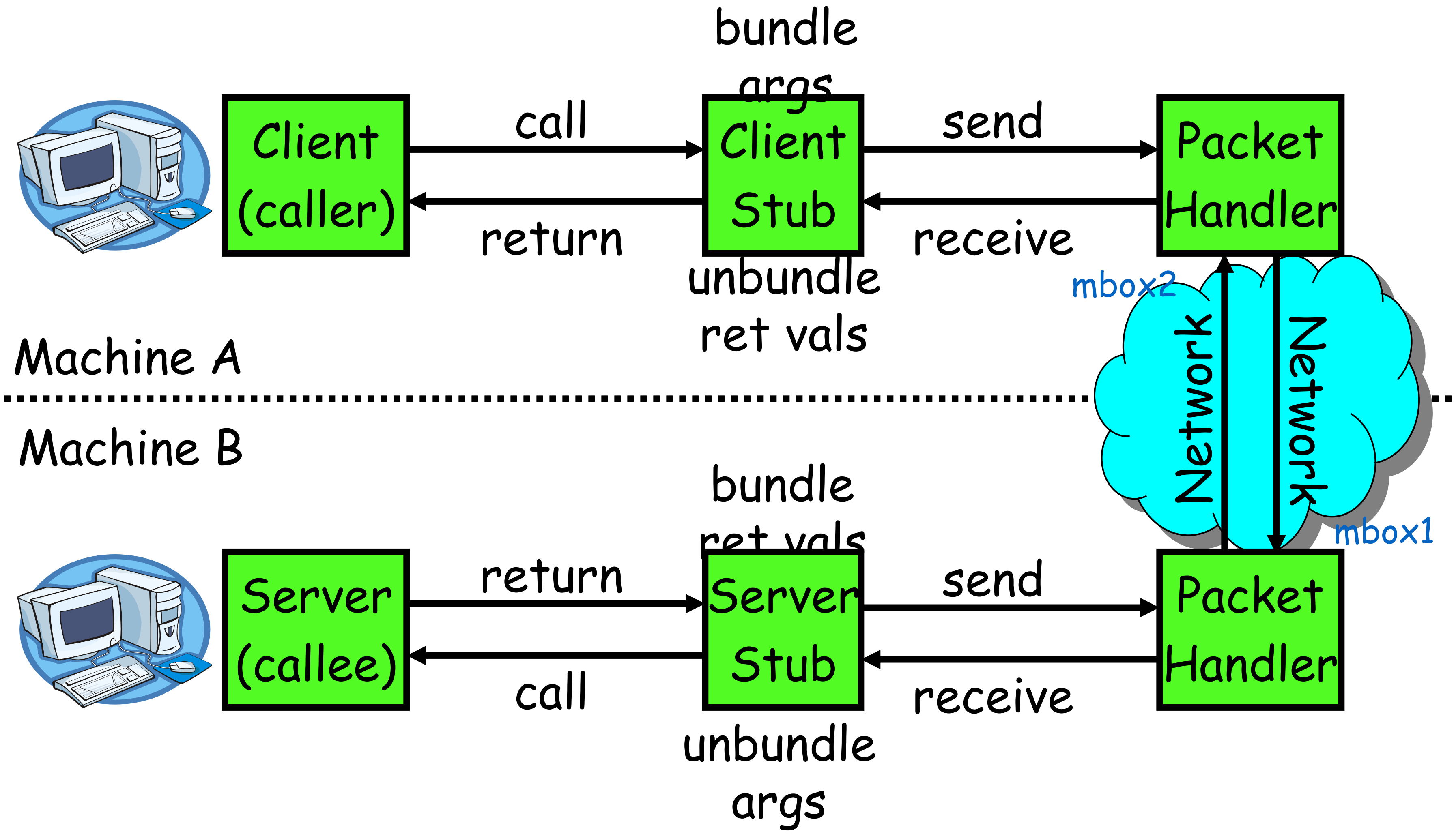
Other RPC Issues

- Mostly related to intermachine RPC
  - Data format conversions
  - Security and authentication
  - Locating remote services
  - Multipacket RPC

Remote Procedure Call

- Implementation:
  - Request-response message passing (under covers!)
  - “Stub” provides glue on client/server
    - Client stub is responsible for “marshalling” arguments and “unmarshalling” the return values
    - Server-side stub is responsible for “unmarshalling” arguments and “marshalling” the return values.
- Marshalling involves (depending on system)
  - Converting values to a canonical form, serializing objects, copying arguments passed by reference, etc.

RPC Information Flow



RPC Implementation

Problems with RPC

- Non-Atomic failures
  - Different failure modes in distributed system than on a single machine
  - Consider many different types of failures
    - User-level bug causes address space to crash
    - Machine failure, kernel bug causes all processes on same machine to fail
    - Some machine is compromised by malicious party
  - Before RPC: whole system would crash/die
  - After RPC: One machine crashes/compromised while others keep working
  - Can easily result in inconsistent view of the world
    - Did my cached data get written back or not?
    - Did server do what I requested or not?
  - Answer? Distributed transactions/Byzantine Commit
- Performance
  - Cost of Procedure call « same-machine RPC « network RPC
  - Means programmers must be aware that RPC is not free
    - Caching can help, but may make failure handling complex

Cross-Domain Communication/Location Transparency

- How do address spaces communicate with one another?
  - Shared Memory with Semaphores, monitors, etc…
  - File System
  - Pipes (1-way communication)
  - “Remote” procedure call (2-way communication)
- RPC’s can be used to communicate between address spaces on different machines or the same machine
  - Services can be run wherever it’s most appropriate
  - Access to local and remote services looks the same
- Examples of modern RPC systems:
  - CORBA (Common Object Request Broker Architecture)
  - DCOM (Distributed COM)
  - RMI (Java Remote Method Invocation)

System Resource Utilization

- System(cat /proc/cpuinfo)
- cat /proc/meminfo

Read CPU utilization

```
#define _GNU_SOURCE
#include <stdio.h>
 #include <stdlib.h>
int main(int argc, char **argv) {
FILE *cpuinfo = fopen("/proc/cpuinfo", "rb");
char *arg = 0;
size_t size = 0;
while(getdelim(&arg, &size, 0, cpuinfo) != -1) { puts(arg); }
free(arg);
fclose(cpuinfo);
return 0; }
```

Read Memory utilization

```
#define _GNU_SOURCE
#include <stdio.h>
 #include <stdlib.h>
int main(int argc, char **argv) {
FILE *cpuinfo = fopen("/proc/meminfo", "rb");
char *arg = 0;
size_t size = 0;
while(getdelim(&arg, &size, 0, cpuinfo) != -1) { puts(arg); }
free(arg);
fclose(cpuinfo);
return 0; }
```

Key, Value, Hashmap

- ```
  #include <stdio.h>
  #include <string.h>
  int main() {
  char* map[] = {"", "172.31.14.27", "172.31.11.125", "172.31.12.40"};
  int listLen = 4;
  // try to get “172.31.14.27”
  int i, find = -1;
  for (i = 0; i < listLen; i++) {
  if (!strcmp(map[i], "172.31.14.27")) {
  find = i;
  break;
  }
  }
  if (find != -1) {
  printf("Ayumu->%d", find);
  } else {
  printf("Not found");
  }
  return 0;
  }
  ```
  ```

Thank You!