Lost Messages

There are a number of reasons that messages may be lost:

- Hardware failure
- Software failure (e.g Buffer overflow)
- Failure is more common in inter-machine communication

How can we tell if a message is lost?

- Each message can be responded to by the receiver with an acknowledge (or “ack”) message
- If the sender does not receive an acknowledge within a certain period of time, it retransmits the message
  - We can keep doing this until we receive the “ack”

Problems with Retransmission

The same message may be sent more than once due to retransmission in two cases:

- The "ack" was lost
- The message (or "ack") was delayed beyond the timer but was still received

This leads to the same message being sent more than once.

- If the message was a request to perform an *idempotent* operation (an operation that if executed twice will produce no ill side-effects) then there is no problem
- A *non-idempotent* operation request is one where executing the operation more than once does cause a problem

Problems with Retransmission

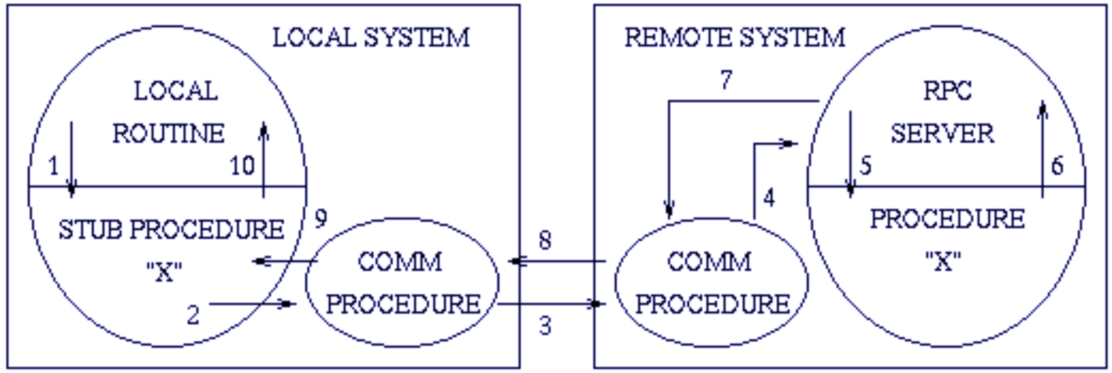
- We can deal with retransmission problems by assigning each message packet a number in a predetermined sequence
  - Receiver discards any packet whose number matches one that has already been received from same sender
- Provide an "at-most-once" IPC operation for non-idempotent operations
  - Reply received? → operation occurred exactly once
  - No reply received? → operation occurred at most once
- We also may have to deal with orphaned requests (ones with no waiting client). We can do this by:
  - *extermination*, *expiration*, or *reincarnation*

Remote Procedure Call

RPC is a way to simplify IPC by modeling communication as procedure calls.

Consider calling a local procedure:

- When process A makes a call to the local subroutine X, it passes the parameters to X and the result from X is returned
- To make this into a remote procedure call:
  - Replace X with a “stub” procedure that packages up the parameters and sends them to the RPC server on remote machine
  - RPC server then calls X on the remote machine and packages up the return value and transmits it back to A



1. local routine calls procedure (stub) X.
2. stub X packs up the arguments, and sends them to the communication process.
3. communication process transfers the arguments to the communication process on the remote machine.
4. remote communications process "replies" the request to RPC server.
5. RPC server calls procedure X (the real one, not the stub).
6. Procedure X returns to RPC server.
7. RPC server sends the results back to the remote communication process.
8. remote communication process transfers results back to the local communication process.
9. local communication process "replies" the result to the stub X.
10. stub X unpacks the results and returns them to the local routine.

Problems with RPC

- Parameters must be passed using strictly machine-independent data structures
- Passing pointers is a big problem because machines do not typically share memory address space
  - Pointer value is meaningless on remote machine!

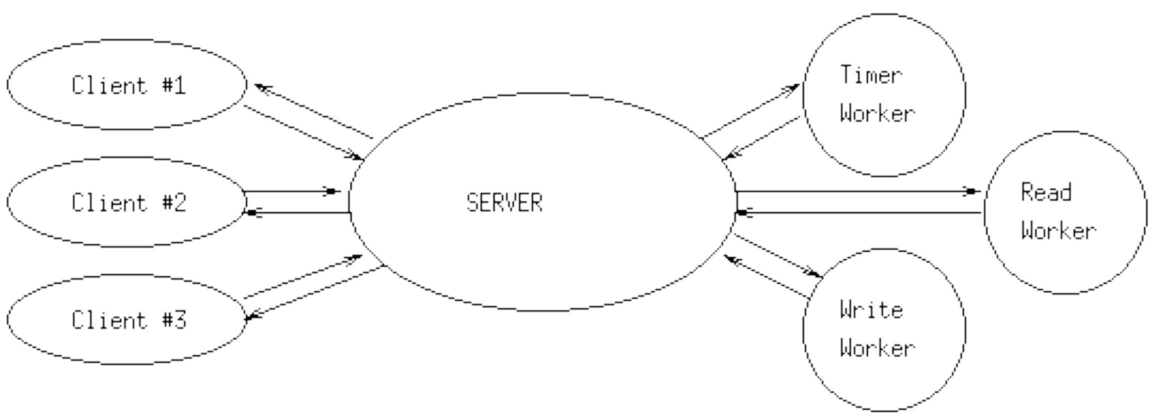
Some possible solutions to this:

- Copy the data pointed to by the pointer, and restore the new data on return
- Disallow passing of pointers for RPC calls
- Have standardized external memory address space representation

Administrator Model of IPC

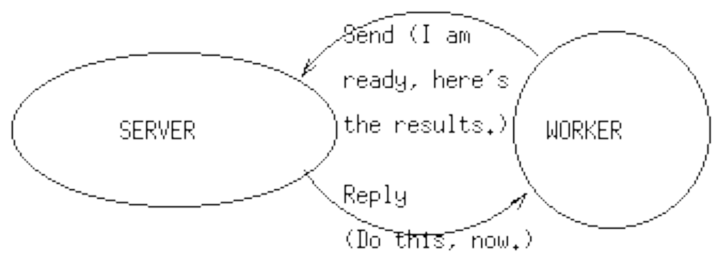
If we have many clients sending requests to a server, then we can run into a problem if the server is doing blocking operations, or other long operations. In this case the clients will possibly have to wait a long while for a response.

Solution: Use worker processes to do the blocking and/or long operations



Administrator Model of IPC

- The server communicates with the workers using Send/Receive/Reply style IPC
- Server cannot "send" to a worker as that would make the server block until a reply was received. Instead:



  - Server "replies" to worker, and worker "sends" results of the operation, and/or an indication of readiness
  - Worker blocks until server "replies" with more work to do

Administrator Model of IPC

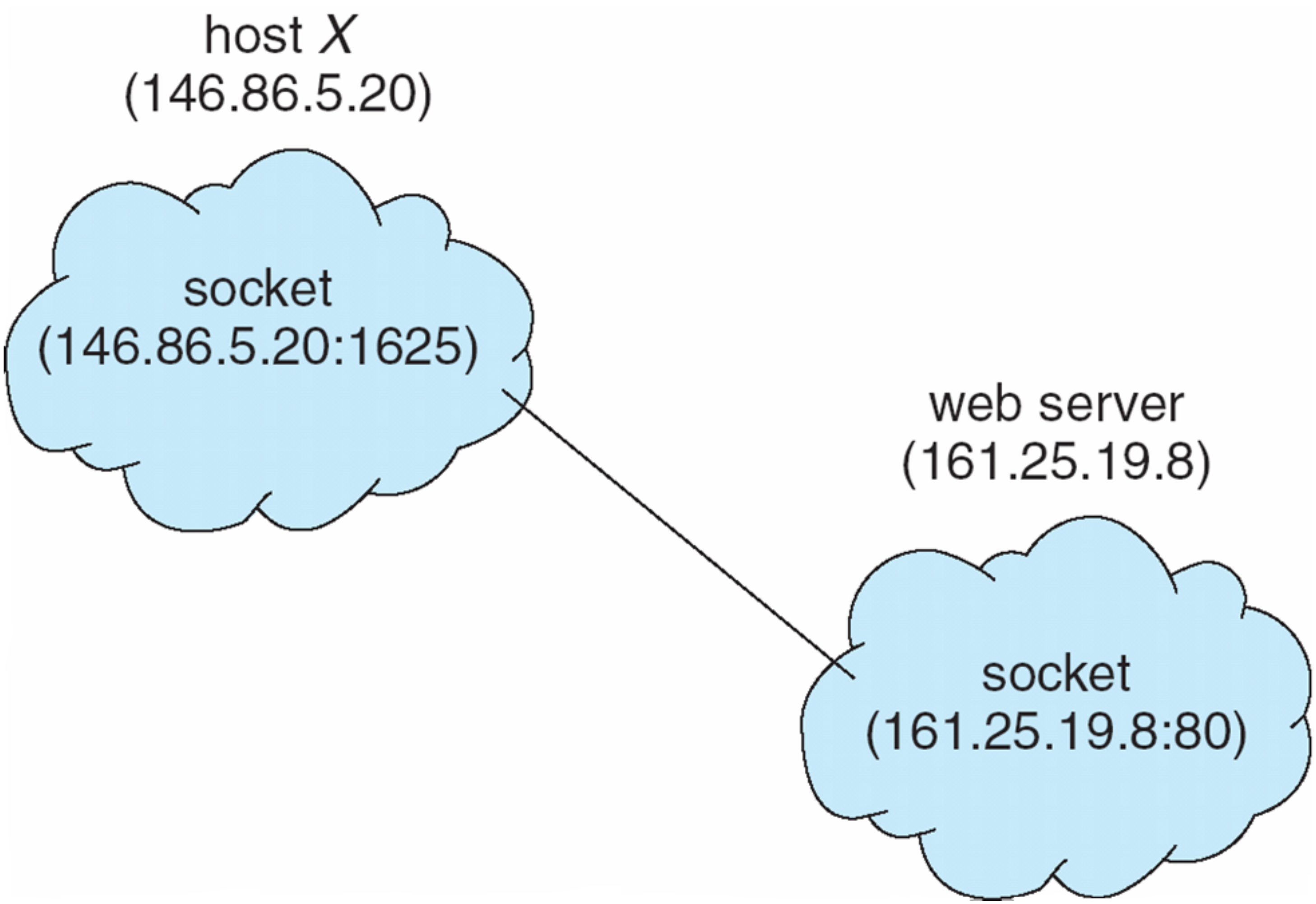
- The administrator (server) can create new workers as needed to handle the load, or destroy workers if they are no longer needed (to save resources)
- Typically, the administrator runs the following:

```
While(1) {
        Receive();
        If (the message was from a worker) {
            - return the result to the client
            - mark the worker as "available"
        }
        If (the message was from a client) {
            - find an available worker
            - pass the client request to an
              available worker.
        }
}
```

Client-Server Communication: 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

Socket Communication



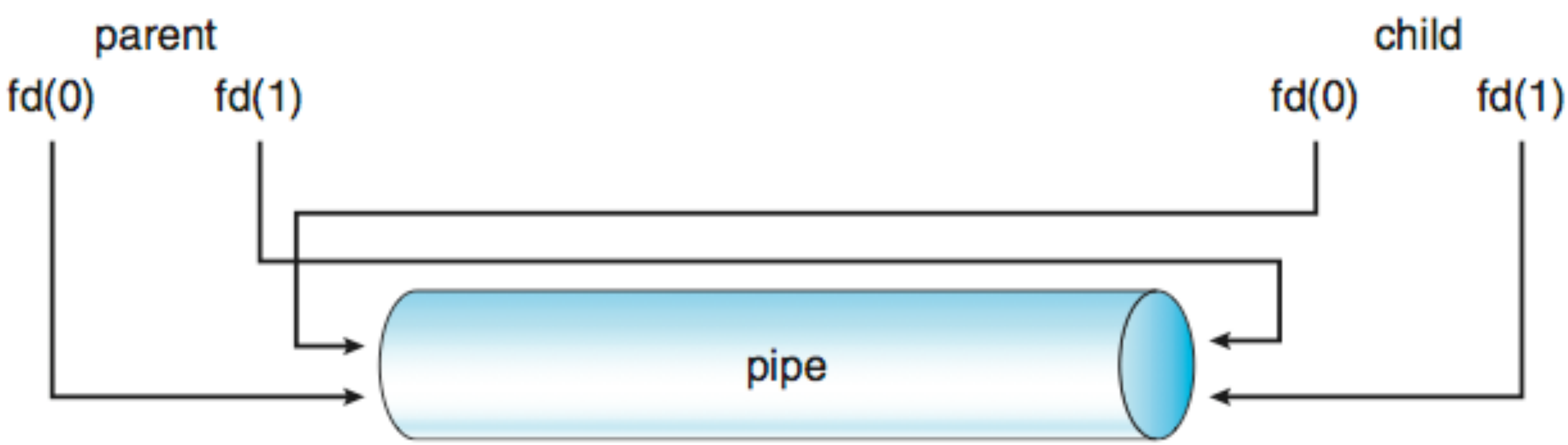
Client-Server Communication: Pipes

- Acts as a conduit allowing two processes to communicate
  - –basically a form of IPC
  - –underlying system creates pipeline
- **Issues**
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e. parent-child) between the communicating processes?
  - Can the pipes be used over a network?

Ordinary Pipes

- **Ordinary Pipes** allow communication in standard producer-consumer style
- Producer writes to one end (the *write-end* of the pipe)
- Consumer reads from the other end (the *read-end* of the pipe
- Ordinary pipes are unidirectional
- Requires parent-child relationship between communicating processes

Ordinary Pipes



Named Pipes

- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems