Network and Distributed File Systems

With content from

Distributed Communication Systems

Christophe Bisciglia, Aaron Kimball, & Sierra Michels-Slettvet

From Local to Network File System

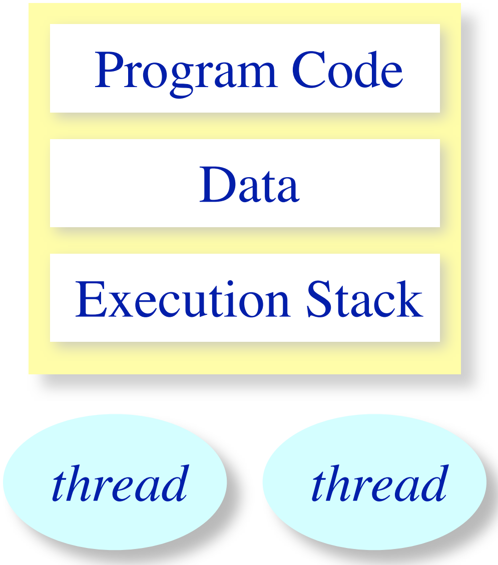
- So far, we have assumed that files are stored on local disk …
- How can we generalize the design to access files stored on a remote server?
- Need to invoke file creation and management methods on the remote server
- Basic mechanisms:
  - Message passing primitives
  - Remote Procedure Calls (RPC)

- A network file system is likely to be better than a local file system in what respects?
  - A. Read/write performance
  - B. Availability
  - C. Fault tolerance
  - D. Ease of management

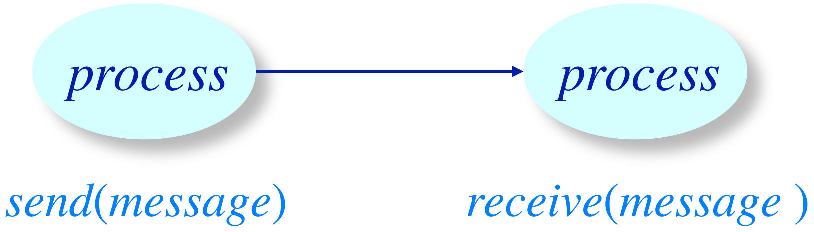
Process Coordination

Two fundamental approaches

- Communication and synchronization based on...
  - Shared memory
    - Assume processes/threads can read & write a set of shared memory locations
    - Inter-process communication is implicit, synchronization is explicit



  - Message passing
    - Inter-process communication is explicit, synchronization is implicit



Process Coordination

Shared Memory *v.* Message Passing

- Shared memory
  - Efficient, familiar
  - Difficult to provide across machine boundaries.

```
            global  int x = 0;
process foo              process bar
begin                    begin
  :                          :
 x := 1                    while(x==0) ;
  :                          :
end foo                  end bar
```

Message passing

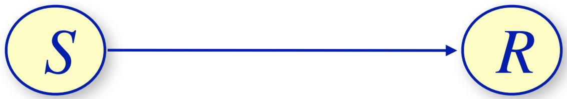
Extensible to communication in distributed systems

Canonical syntax:

```
   send(int id, String message);
receive(int id, String message);
```

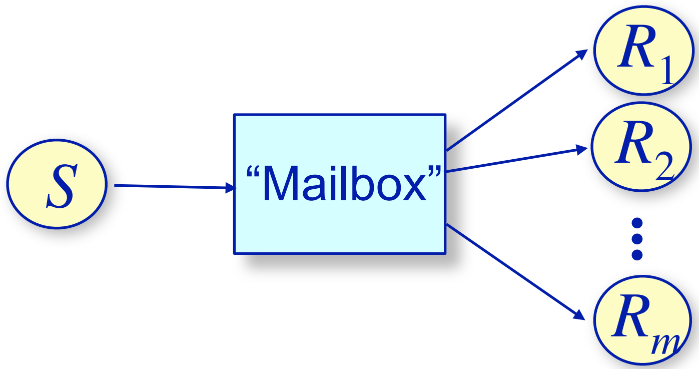
Message Passing
Naming communicants

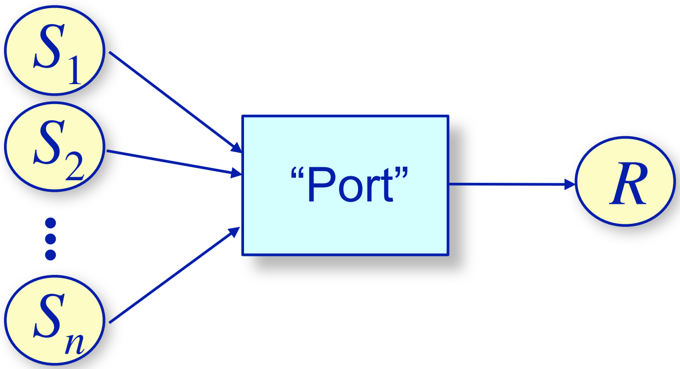
- How do processes refer to each other?
  - ➢ Does a sender explicitly name a receiver?



Can a message be sent to a group?

Can a receiver receive from a group? (a reduction operation)





Web requests conform to what model?

1. Many-to-one
2. One-to-one
3. One-to-many

Message Passing Issues

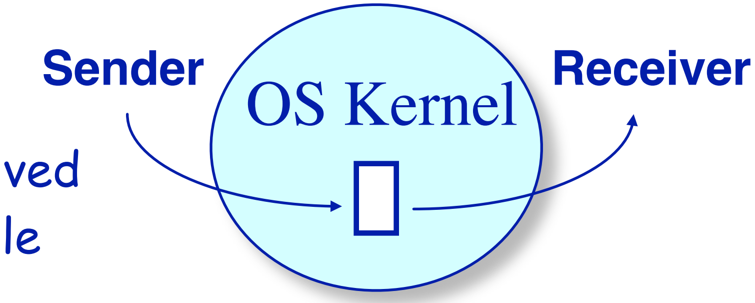
Synchronization semantics

- When does a *send*/*receive* operation terminate?

Blocking:

Sender waits until its message is received

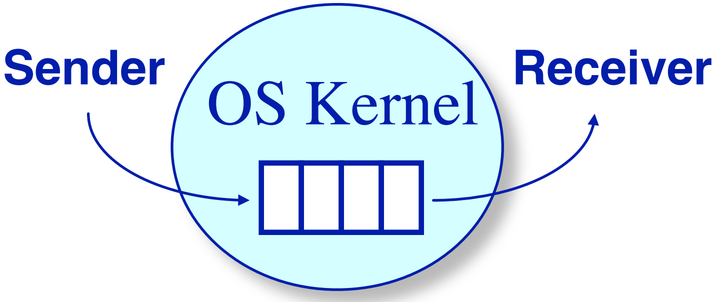
Receiver waits if no message is available



Non-blocking:

Send operation "immediately" returns

Receive operation returns if no message is available



Partially blocking/non-blocking:

send()/receive() with timeout

Semantics of Message Passing

send(*receiver*, *message*)

		Synchronization	
		Blocking	Non-blocking
Naming	Explicit (single)	Send message to receiver Wait until message is accepted.	Send message to receiver
	Implicit (group)	Broadcast message to all receivers. Wait until message is accepted by all	Broadcast message to all receivers

Semantics of Message Passing

receive(*sender*, *message*)

Synchronization

Naming

	Blocking	Non-blocking
Explicit (single)	Wait for a message from sender	If there is a message from sender then receive it, else continue
Implicit (group)	Wait for a message from any sender	If there is a message from any sender then receive it, else continue

Which do you think would be easier to program?

A. A message passing program that blocks.

B. A message passing program that does not block.

RPC is not message passing

- Regular client-server protocols involve sending data back and forth according to shared state

```
Client:                       Server:
HTTP/1.0 index.html GET
                              200 OK
                              Length: 2400
                              (file data)
HTTP/1.0 hello.gif GET
                              200 OK
                              Length: 81494
                              …
```

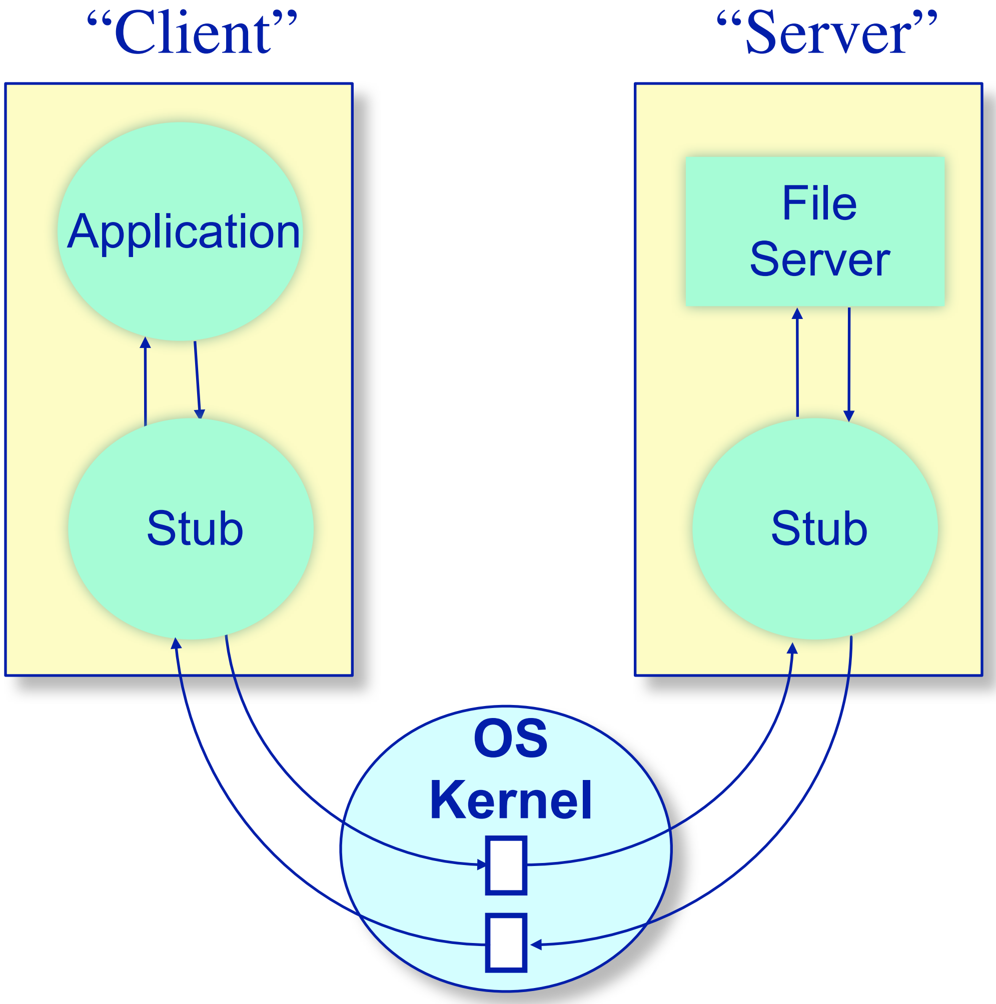
Remote Procedure Call

- RPC servers will call arbitrary functions in dll, exe, with arguments passed over the network, and return values back over network

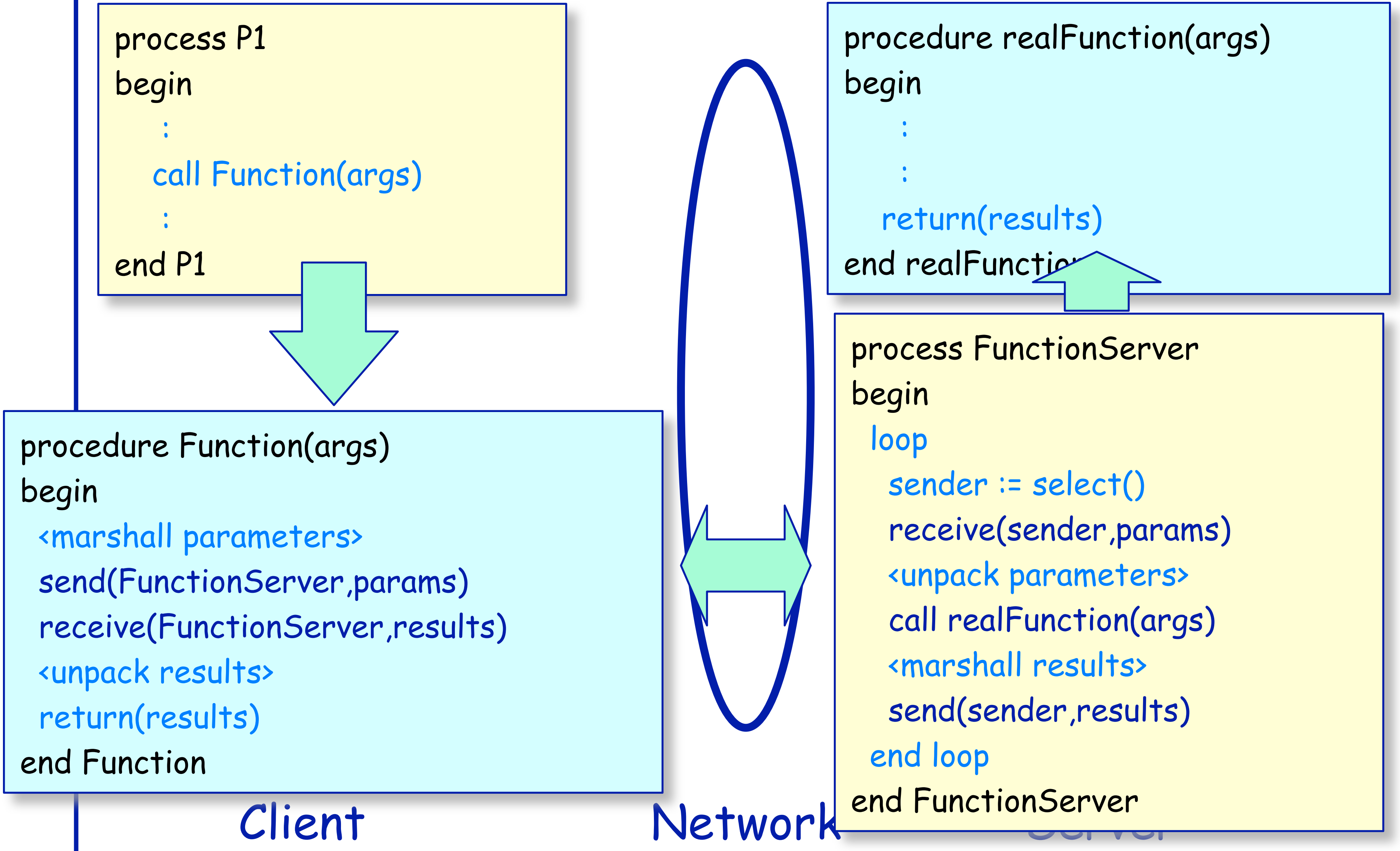
```
Client:                       Server:

foo.dll,bar(4, 10, “hello”)

                              “returned_string”

foo.dll,baz(42)

                              err: no such function

…
```

RPC: Message Passing Evolves

- Remote procedure calls abstract out the *send/ await-reply* paradigm into a “procedure call”
- Remote procedure calls can be made to look like “local” procedure calls by using a stub that hides the details of remote communication



Remote Procedure Call



```
process P1
begin
    :
    call Function(args)
    :
end P1
```

```
procedure Function(args)
begin
  <marshall parameters>
  send(FunctionServer,params)
  receive(FunctionServer,results)
  <unpack results>
  return(results)
end Function
```

```
procedure realFunction(args)
begin
      :
      :
   return(results)
end realFunction
```

```
process FunctionServer
begin
  loop
    sender := select()
    receive(sender,params)
    <unpack parameters>
    call realFunction(args)
    <marshall results>
    send(sender,results)
  end loop
end FunctionServer
```

RPC (Cont'd.)

- Similarities between procedure call and RPC
  - Parameters ↔ request message
  - Result ↔ reply message
  - Name of procedure ↔ passed in request message
  - Return address ↔ mailbox of the client

- Implementation issues:
  - Stub generation
    - Can be automated
    - Requires the signature of the procedure
  - How does a client locate a server? … Binding
    - Static – fixed at compile-time
    - Dynamic – determined at run-time with the help of a name service
  - Why run-time binding?
    - Automatic fail-over

Problems with RPC

- Failure handling
  - A program may hang because of
    - Failure of a remote machine; or
    - Failure of the server application on the remote machine
  - An inherent problem with distributed systems, not just RPC
    - Lamport: “A distributed system is one where you can’t do work because some machine that you have never heard of has crashed”
- Performance
  - Cost of procedure call << same machine RPC << network RPC

Java RMI (remote method invocation) is an example of an RPC system.

A. Yes
B. No

Why use RPC?

A. Programmer convenience
B. Improve performance
C. Simplify implementation
D. Simplify API

Network and Distributed File Systems

- Provide transparent access to files stored on remote disks

- Issues:
  - Naming: How do we locate a file?
  - Performance: How well does a distributed file system perform as compared to a local file system?
  - Failure handling: How do applications deal with remote server failures?
  - Consistency: How do we allow multiple remote clients to access the same files?

Naming Issues

- Two Approaches To File Naming
  - Explicit naming: <file server: file name >
    - E.g., windows file shares
    - //arrvindh-laptop/Users/arrvindh/Desktop
  - Implicit naming
    - Location transparency: file name does not include name of the server where the file is stored
- Server must be identified.
- Most common solution (e.g., NFS)
  - Static, location-transparent mapping
  - Example: NFS Mount protocol
    - Mount/attach remote directories as local directories
    - Maintain a mount table with directory → server mapping, e.g., mount zathras:/vol/vol0/users/arrvindh /home/arrvindh

Performance Issues: Simple Case

- Simple case: straightforward use of RPC
  - Use RPC to forward every file system request (e.g., open, seek, read, write, close, etc.) to the remote server
  - Remote server executes each operation as a local request
  - Remote server responds back with the result
- Advantage:
  - Server provides a consistent view of the file system to distributed clients. What does consistent mean?
- Disadvantage:
  - Poor performance

Solution: Caching

Why does turning every file system operation into an RPC to a server perform poorly?

1. Disk latency is larger than network latency
2. Network latency is larger than disk latency
3. No server-side cache
4. No client-side cache

Sun's Network File System (NFS)

- Cache data blocks, file headers, etc. both at client and server
  - Generally, caches are maintained in memory; client-side disk can also be used for caching
  - Cache update policy: write-back or write-through
- Advantage:
  - Read, Write, Stat etc. can be performed locally
    - Reduce network load and
    - Improve client performance
- Problem: How to deal with failures and cache consistency?
  - What if server crashes? Can client wait for the server to come back up and continue as before?
    - Data in server memory can be lost
    - Client state maintained at the server is lost (e.g., seek + read)
    - Messages may be retried
  - What if clients crash?
    - Loose modified data in client cache

NFS Protocol: Statelessness

- Stateful vs. stateless server architectures

- NFS uses a stateless protocol
  - Server maintains no state about clients or open files (except as hints to improve performance)
  - Each file request must provide complete information
    - Example: ReadAt(inode, position) rather than Read(inode)
  - When server crashes and restarts, it processes requests as if nothing has happened !

- Idempotent operations
  - All requests can be repeated without any adverse effects

- Result:
  - Server failures are (almost) transparent to clients
  - When server fails, clients hang until the server recovers or crash after a timeout

NFS Protocol: Consistency

- What if multiple clients share the same file?
  - Easy if both are reading files …
  - But what if one or more clients start modifying files?

- Client-initiated weak consistency protocol
  - Clients poll the server periodically to check if the file has changed
  - When a file changes at a client, server is notified
    - Generally, using a delayed write-back policy
  - Clients on detecting a new version of file at the server obtain a new version

- Consistency semantics determined by the cache update policy and the file-status polling frequency

- Other possibility: server-initiated consistency protocol

NFS: Summary

- Key features:
  - Location-transparent naming
  - Client-side and server-side caching for performance
  - Stateless, client-driven architecture
  - Weak consistency semantics
- Advantages:
  - Simple
  - Highly portable
- Disadvantages:
  - Inconsistency problems

Andrew File System (AFS): A Case Study

- Originally developed at CMU → later adapted to DFS by IBM

- Key features:
  - Callbacks: server maintains a list of who has which files
  - Write-through on file close
    - On receiving a new copy, server notifies all clients with a file copy
  - Consistency semantics:
    - Updates are visible only on file close
  - Caching:
    - Use local disk of clients as caches
    - Can store larger amount in cache → smaller server load
  - Handling server failures:
    - Loose all callback state → need a recovery protocol to rebuild state

- Pros and cons:
  - Use of local disk as a cache reduces server load
  - Callbacks → server is not involved in read-only files at all
  - Central server is still the bottleneck (for writes, failures, …)