Managing complexity: the key difficulty in (computer) system building

Tools to cope with complexity

Simplifying insights / experience / successful designs

Modularity

- **Split up system, consider small pieces separately**

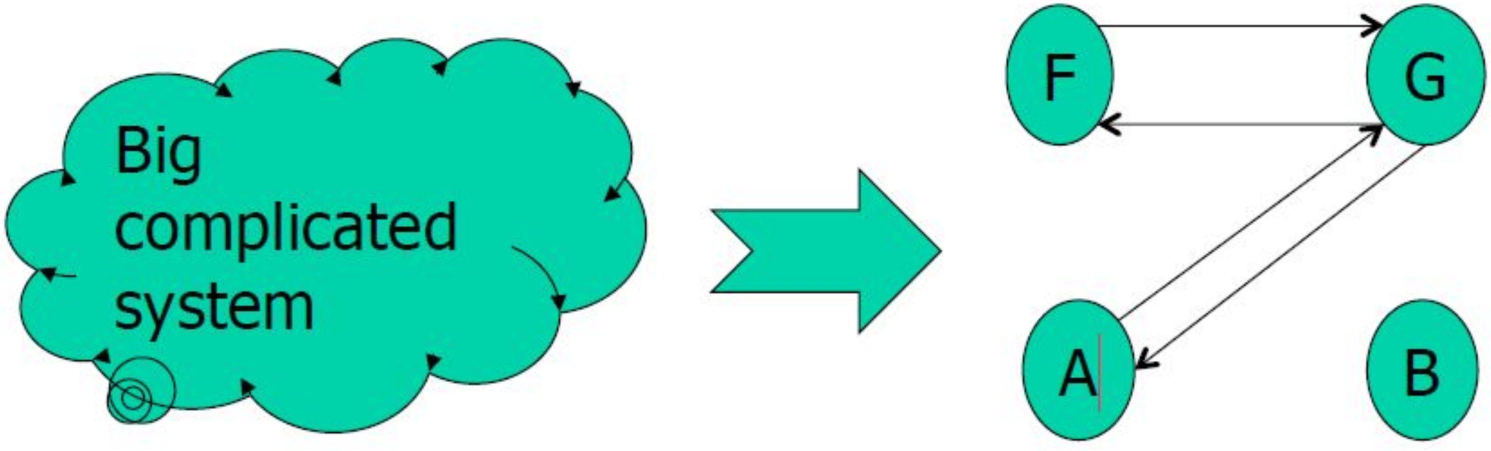
Abstraction

- **Hide implementation internals / hide usage details.**

Hierarchy -- Reduce connections / regular layouts

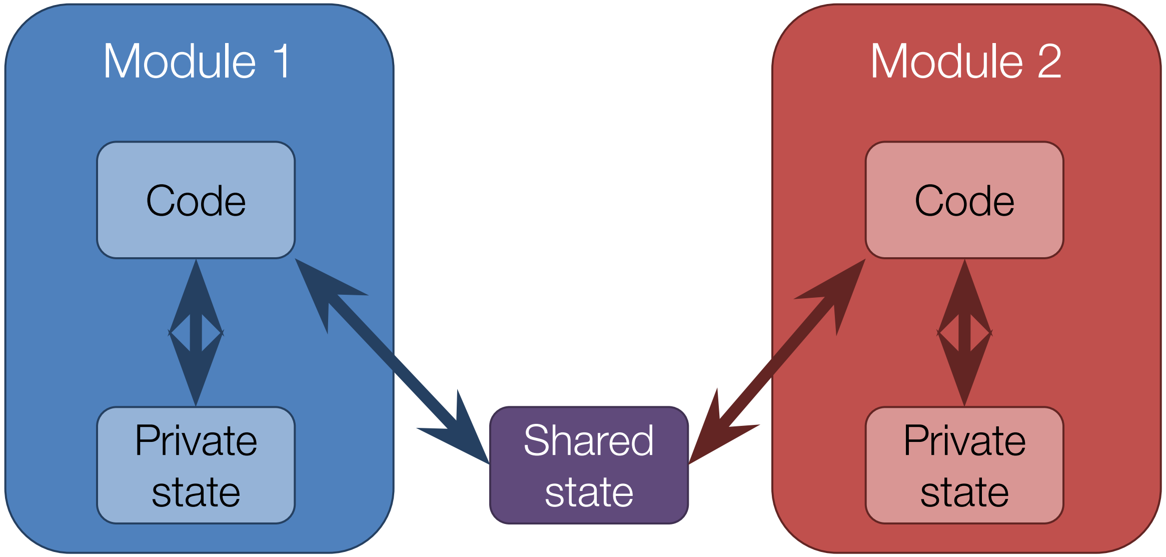
Layering -- Gradually build up capabilities. Reduce connections

A modularity tool: the procedure call



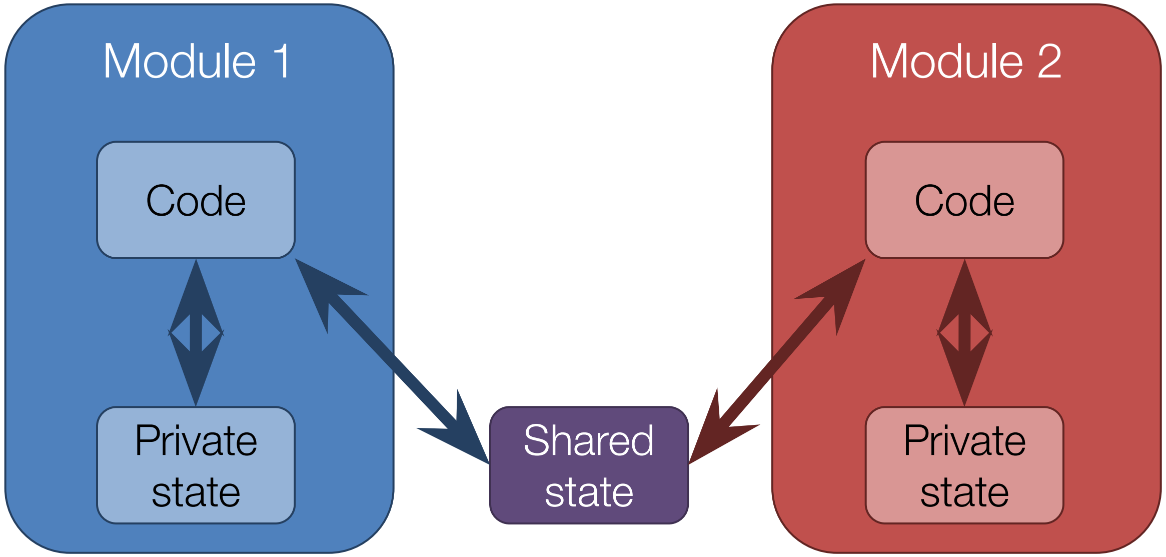
Procedures as modules

- What is private and what is shared between procedures?
  - Local variables are private
  - Stack, heap, global variables are shared



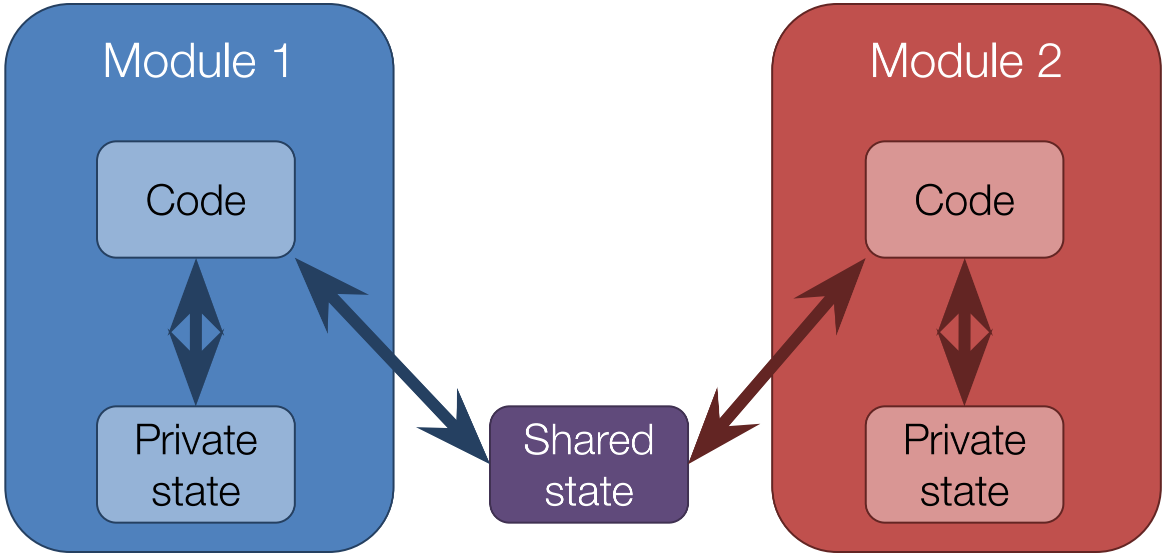
Procedures as modules

- How is control transferred between procedures?
  - Caller adds arguments and RA to stack, jumps into callee code
  - Callee sets up local variables, runs code, jumps to RA



Procedures as modules

- Is modularity between procedures *enforced*?
  - No, either module can corrupt the other
  - No guarantee that callee will return to caller either
    - callee may crash (and crash the application – aka “fate sharing” or hang)

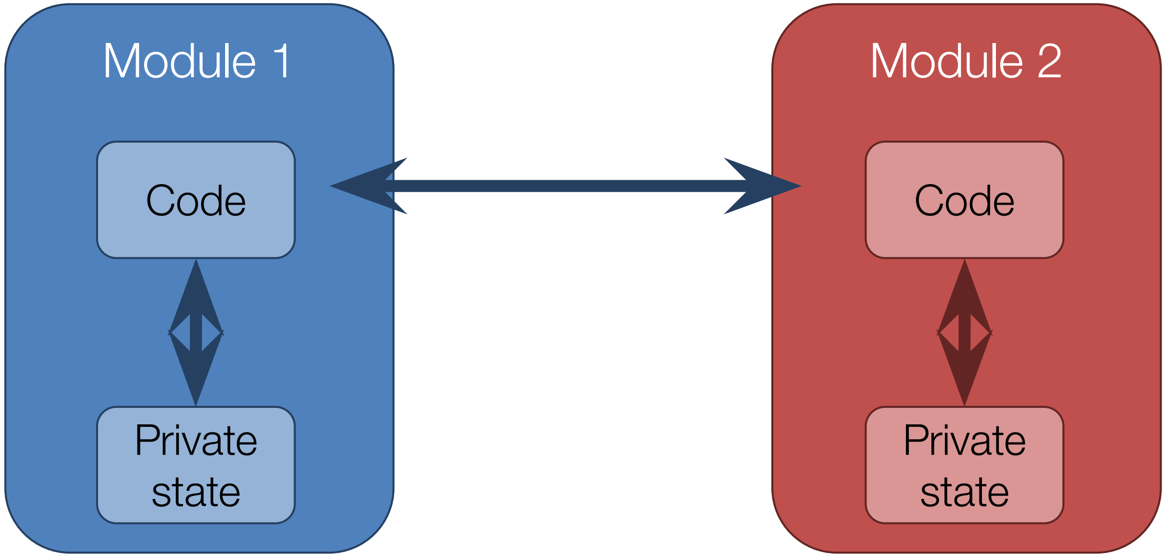


Soft modularity!

Can we do better?

enforce modularity by (among various options)

- deploying on different machines, and
- making communication explicit through messages



Do we still have the same problems?

- Can either module corrupt the other?
- Can guarantee that caller will eventually continue?

(regardless of what calle does (e.g., hang, crash)

[Another way to think about RPCs]

The process of building a distributed application

Design from scratch

Communication-oriented design

- Start with communication protocol;
  - Design message format and syntax
- Design components (clients, servers) by specifying how they react to incoming messages

Where it gets hard

- Protocol design problems
- Complexity: specify components as finite state machines
- Focus on communication instead of on the application

Design based on existing single-box application

Application-oriented design

- Start with application
  - Design, build, test conventional (single-box) application
- Partition your application using middleware support [e.g.RPC]

Where it gets hard

- Preserving semantics when partitioning program (using remote resources)
- Masking failures
- Concurrency

Remote Procedure Calls (RPC) and Java Remote Method Invocation (RMI) are meant to offer support with this approach

So far:

- Procedure Calls: an *[imperfect]* tool for modularity

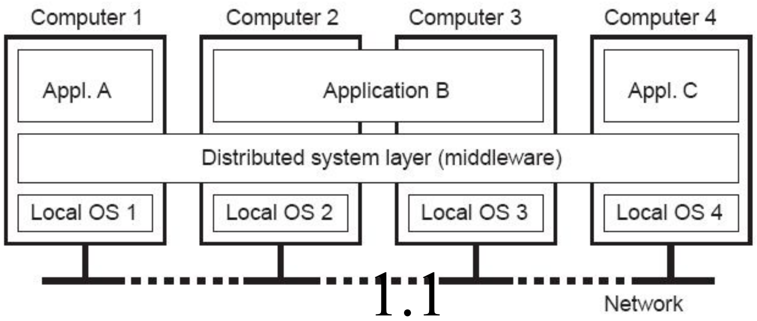
  "soft modularity" – contract not enforced

- Enforced/hard modularity: enforce the contract
  - E.g., separate caller and callee on different machines
    - One could build using request/reply only … but impractical (compared to procedure)

Next: Remote Procedure Calls:

- Make calling a procedure hosted remotely similar to calling a local procedure
- Enforced modularity + transparency

Distributed System: A collection of *independent computers* that appears to its users as a *single coherent system*



- *Independent* hardware installations
- *Uniform software layer* (middleware)

Key goals

- Connect users and resources
- ***Distribution transparency***
- Openness
- Scalability

Goal II: Transparency

Dimension	**Description**
Access	Hide differences in *data representation and resource access*
Location	Hide where a resource is *located*
Migration	Hide that a *resource may move* to another location
Relocation	Hide that a *resource may migrate while in use*
Replication	Hide that a *resource may have multiple copies*
Concurrency	Hide that a *resource may be shared by several competing users*
Failure	Hide the failure and recovery of a resource

Note: transparency may be set as a goal, but achieving it is a different story.

Today

- Transparency …
  - implementation mechanisms, limitations
- … in a specific context Remote Procedure Calls

Remote Procedure Calls (RPC)

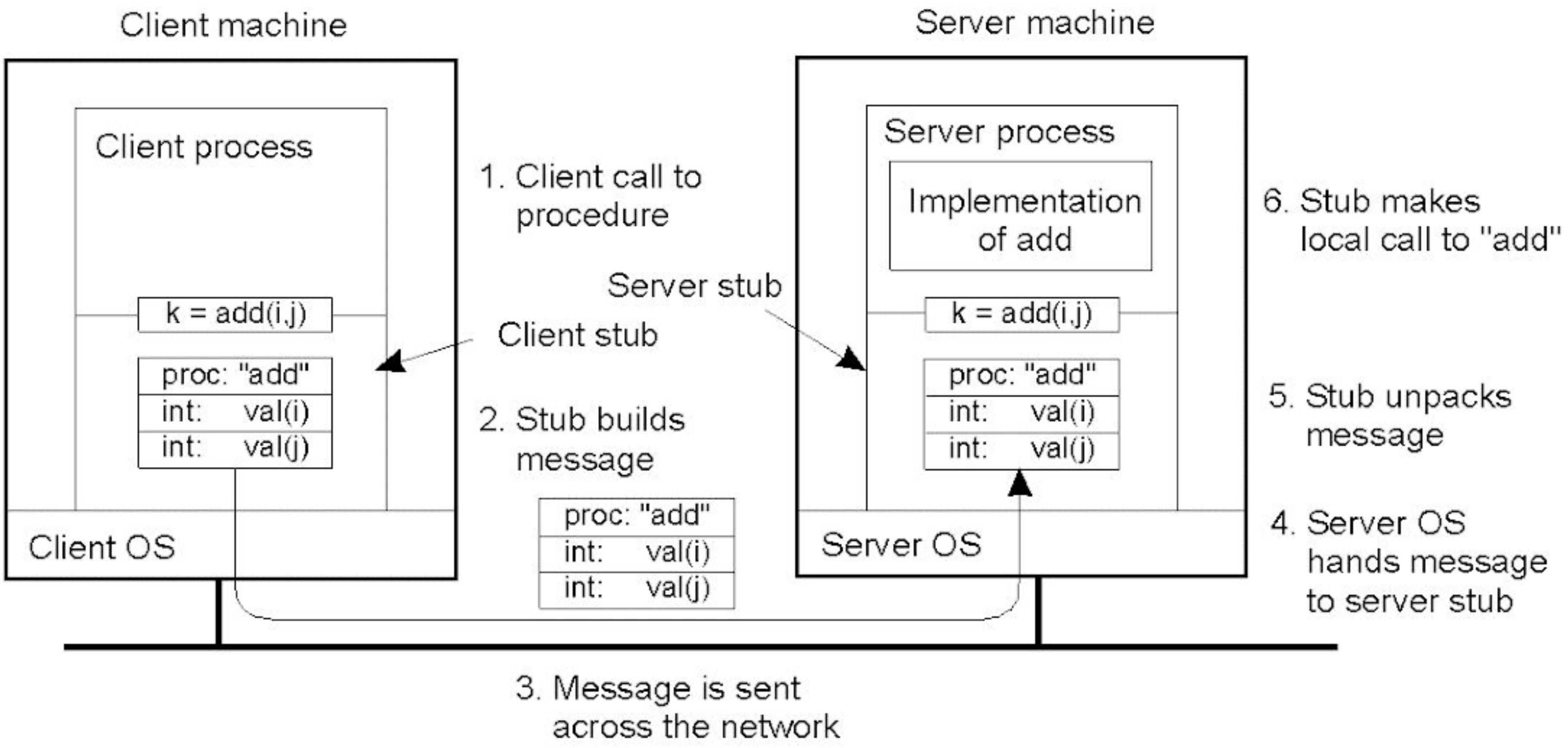
- *Goal*: Make calling a procedure hosted remotely similar to calling a local procedure
  - Info is passed as messages in procedure arguments and results
- *Key issue*: Provide transparency
  - (access transparency) mask differences in data representation
  - (location, failure transparency) handle failures
  - (location transparency) handle different address spaces
  - (provide migration transparency, replication transparency)
  - Security!

The context is important! The implementation details are specific to the programming language

Outline

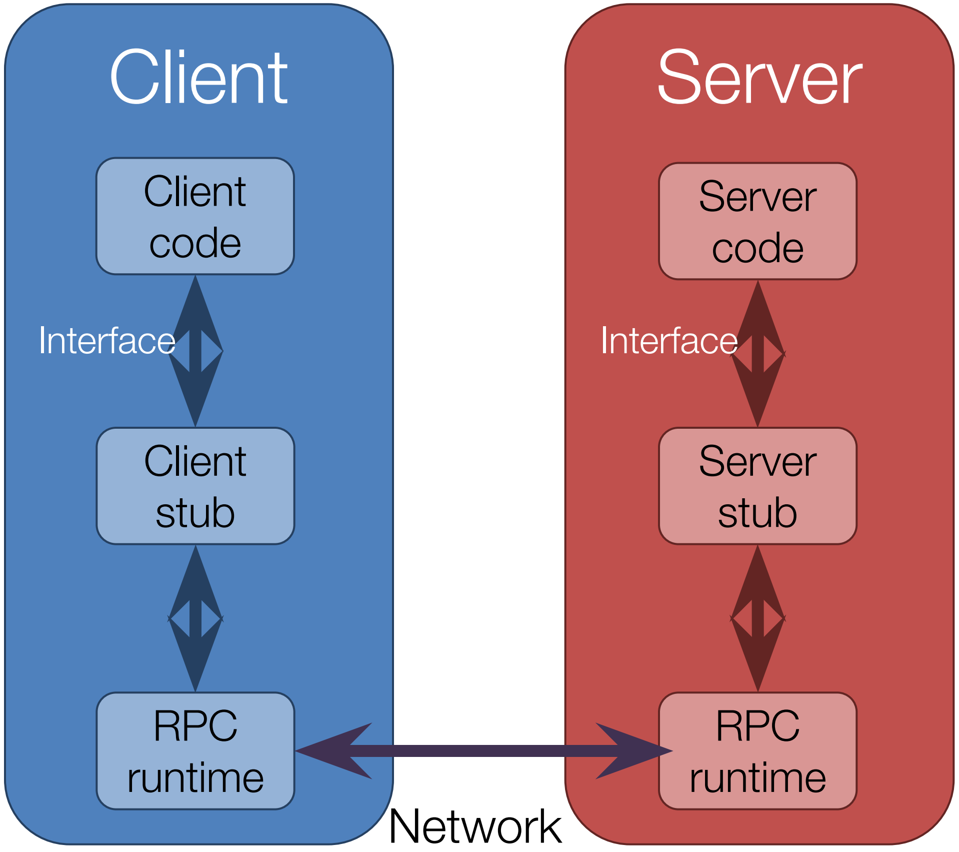
- Mechanics.
  - How does it actually work …
  - … and limitations
- RPC in practice.
  - Case study: The Network File System

- RPC makes request/response ‘look’ local
  - Provides the illusion of a function call
- RPC isn’t a really a function call
  - In a normal call, the PC jumps to the function
  - Function then jumps back to caller
- This is *similar* to request/response though
  - Stream of control goes from client to server
  - And then returns back to the client



- Note: parameters/result are passed by value

RPC architecture

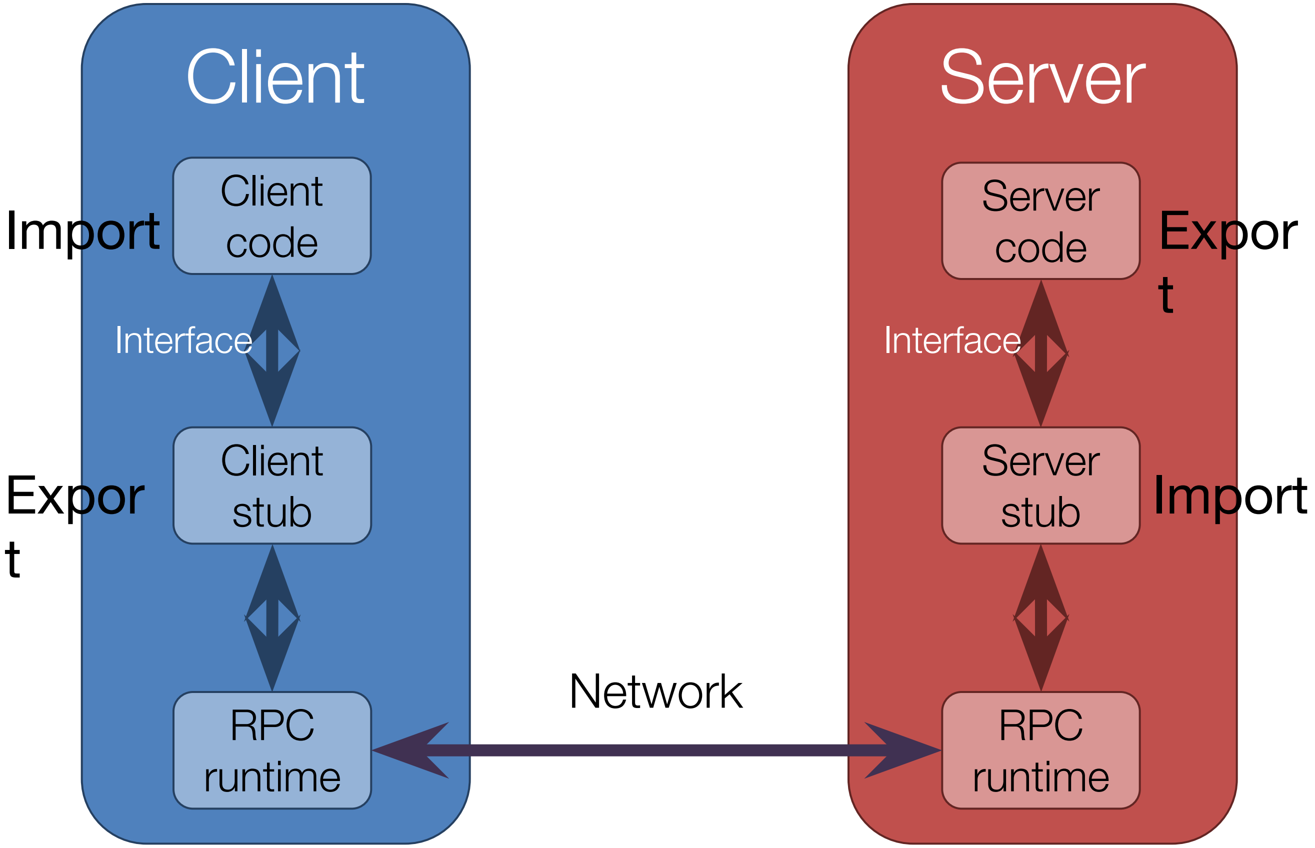


- User (unmodified) code
- Generated code to map user defined functions to generic runtime
- Thread control
- **Request/reply protocol**
- **Serialization (e.g., ProtocolBuffers**

How is everything we've looks at so far relevant?

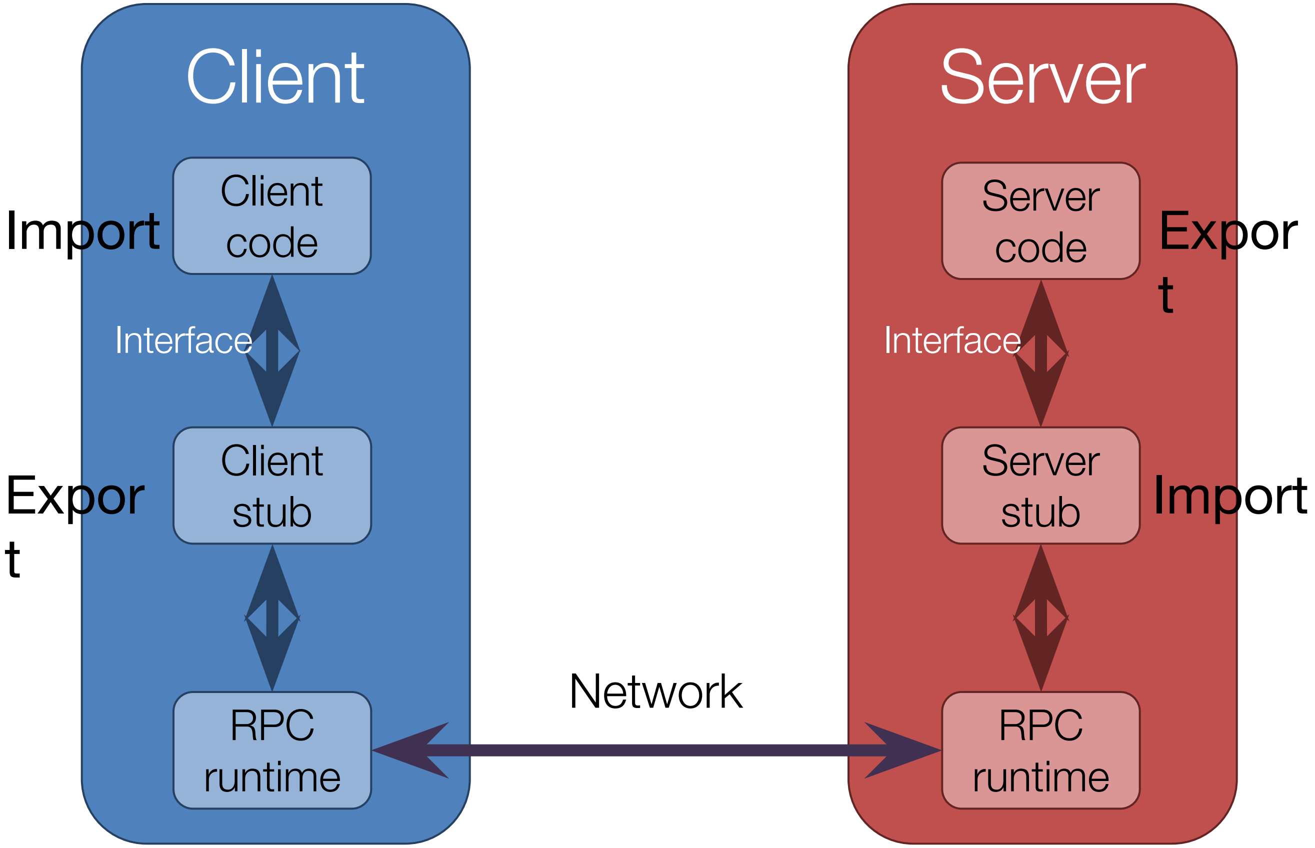
- Request/reply protocol
  - Two generals: discussion about impossibility to guarantee agreement
    - All the discussion about call semantics (e.g., at-most-once semantics)
    - Design discussion around handling network faults and server crashes
      - Retries
- Common wire protocol

RPC architecture



Who imports and who exports the interface?

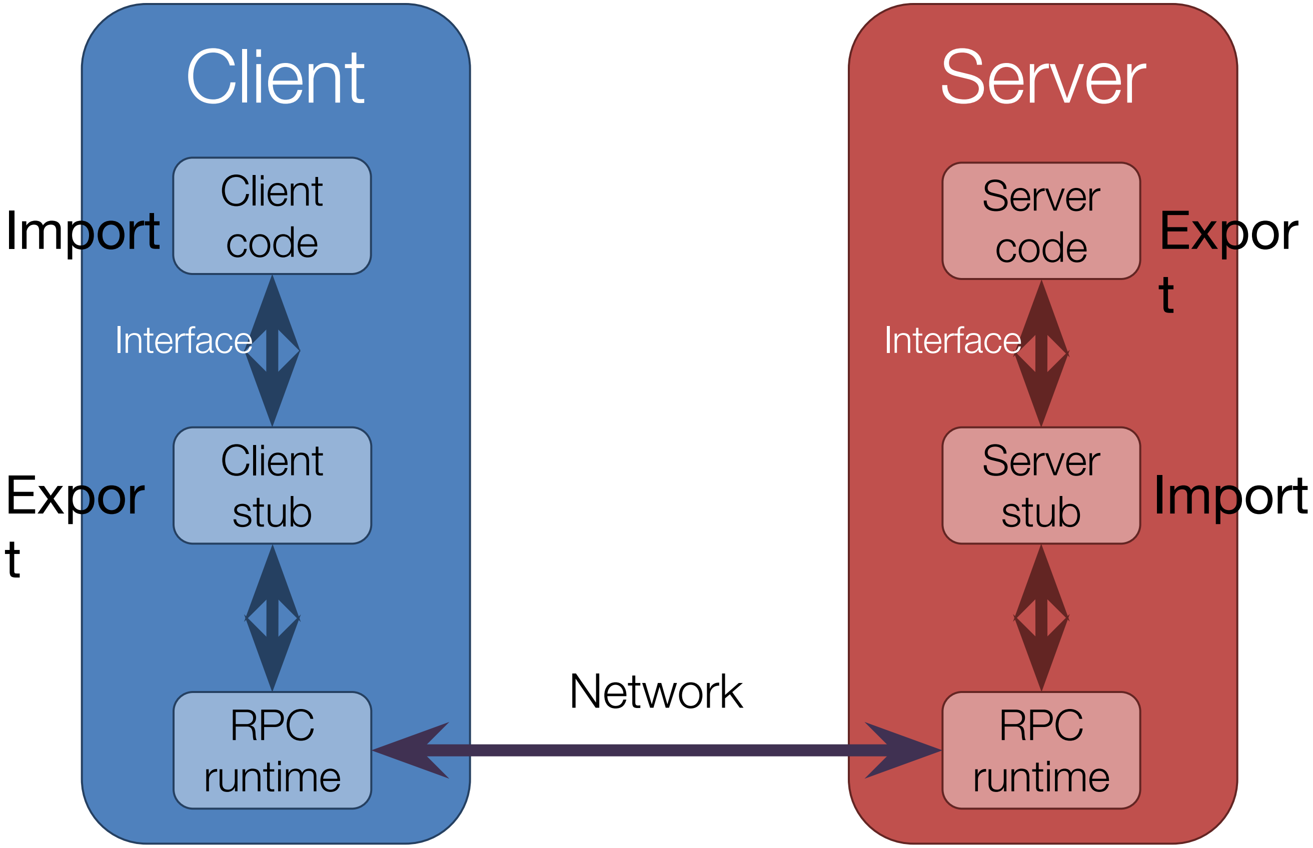
RPC architecture



Who defines the interface?

The programmer

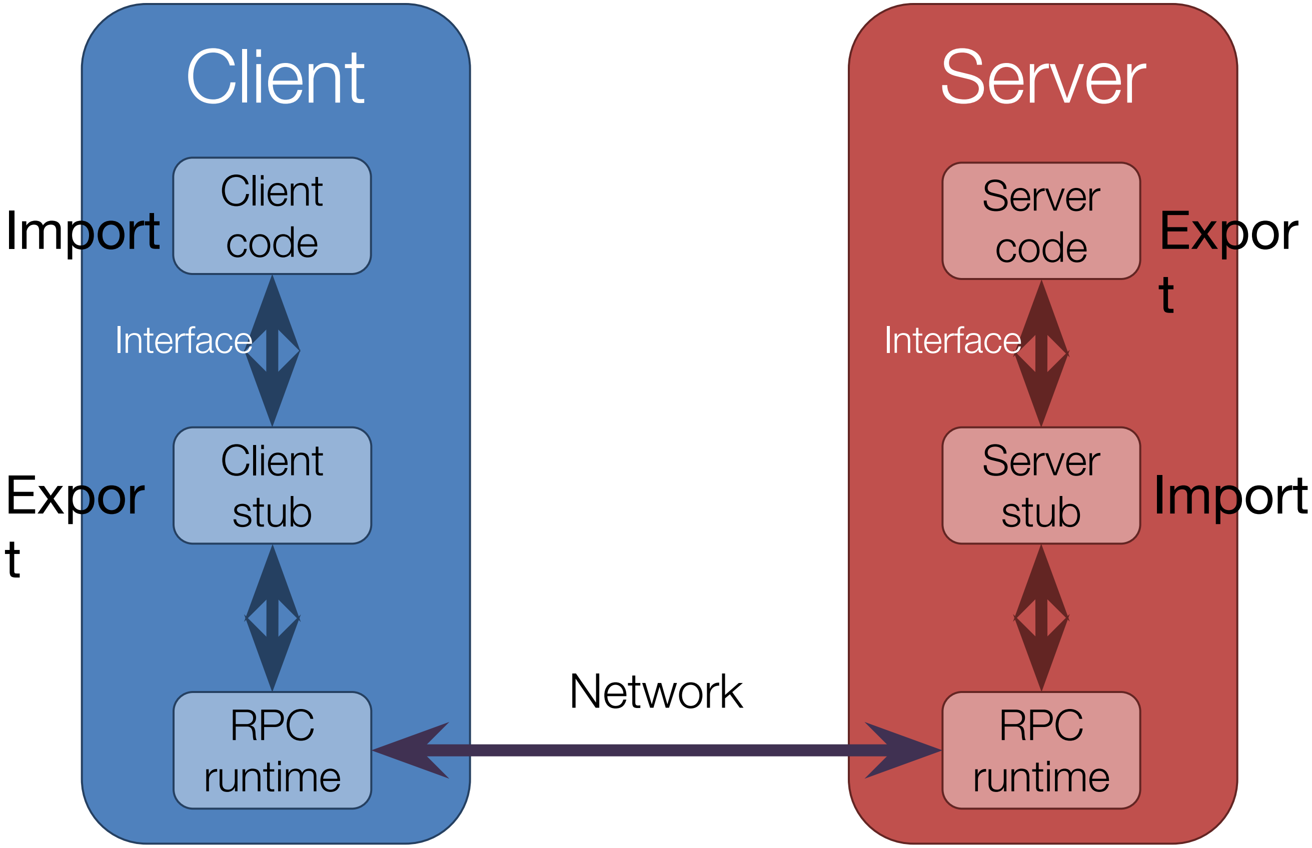
RPC architecture



Who writes the client and server code?

The programmer

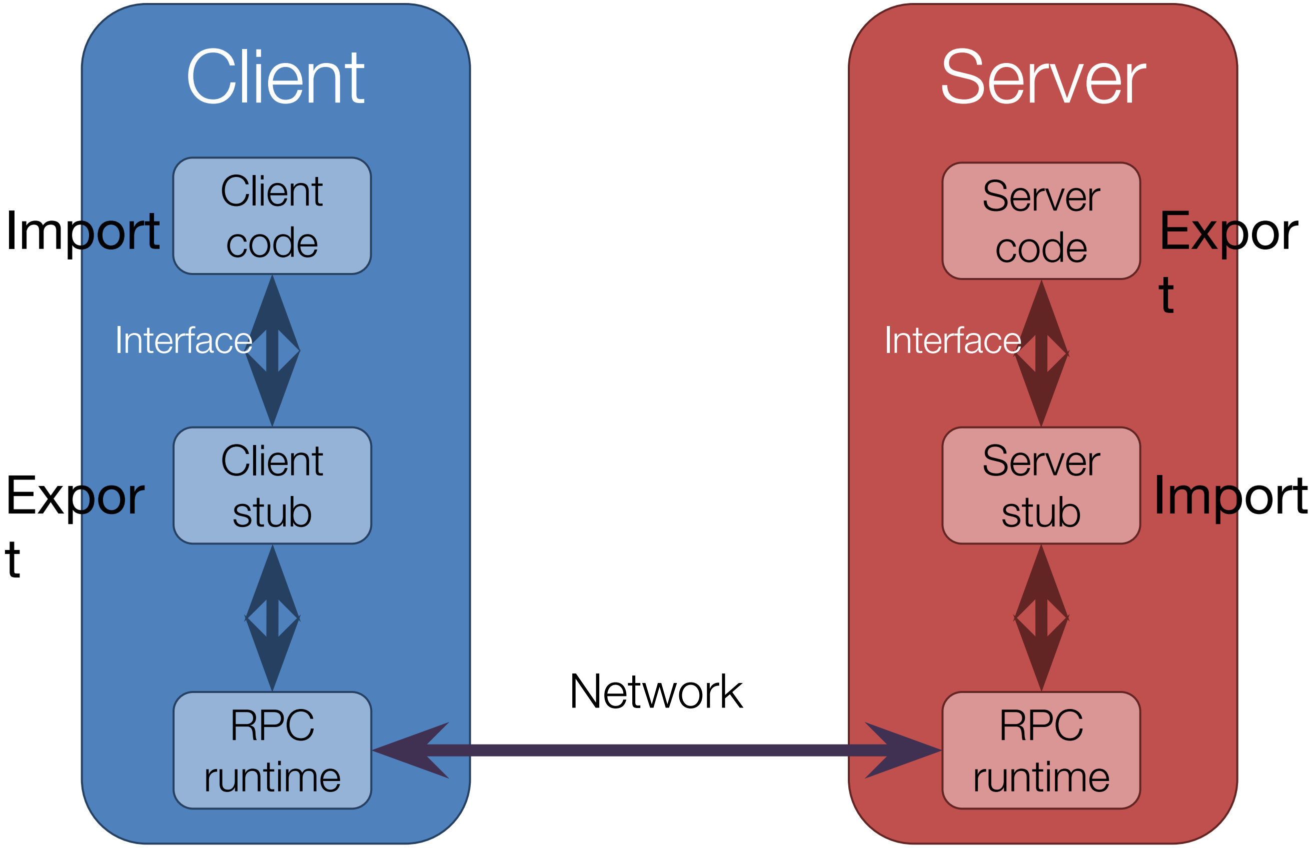
RPC architecture



Who writes the stub code?

An automated stub generator (rmic in Java)

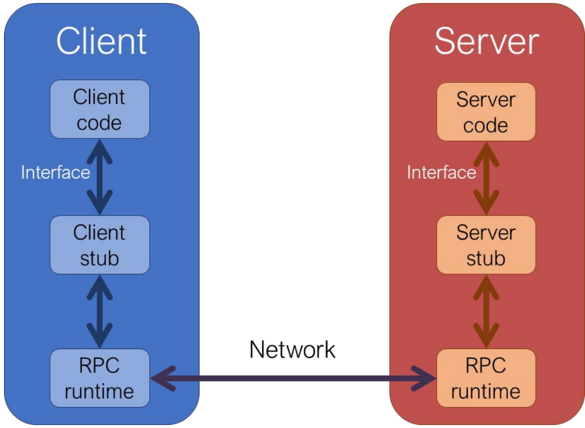
RPC architecture



Why can stub code be generated automatically?

Interface precisely defines
what data comes in, what is returned

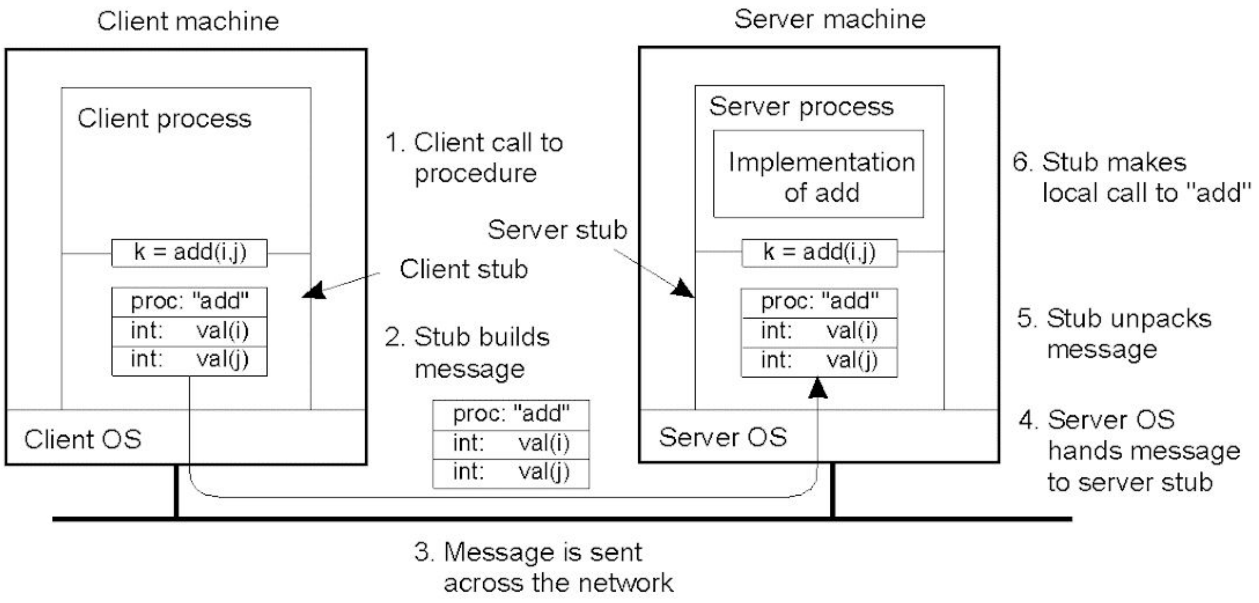
RPC stub functions

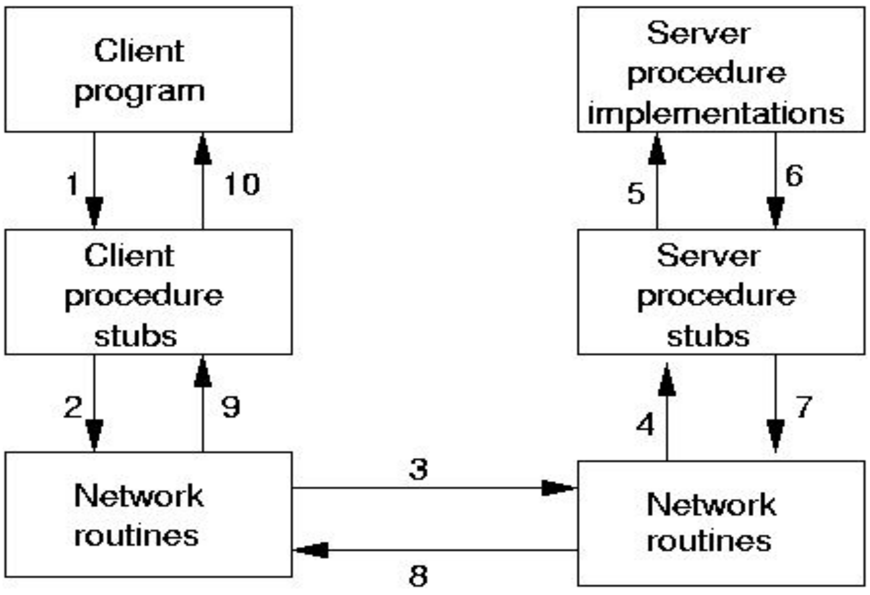


- Client stub
  1) Builds request message with server function name and parameters
  2) Sends request message to server stub
     - Transfer control to server stub: clients-side code is paused
  8) Receives response message from server stub
  9) Returns response value to client
- Server stub
  3) Receives request message
  4) Calls the right server function with the specified parameters
  5) Waits for the server function to return
  6) Builds a response message with the return value
  7) Sends response message to client stub

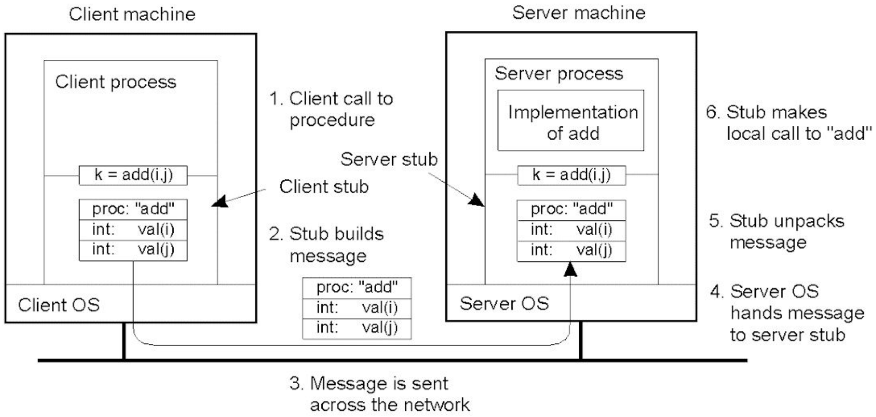
Steps of a Remote Procedure Call.

1. Client procedure calls client stub in normal way
2. Client stub builds message calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client





RPC mechanics



Objective: developer concentrates only the client- and server-specific code

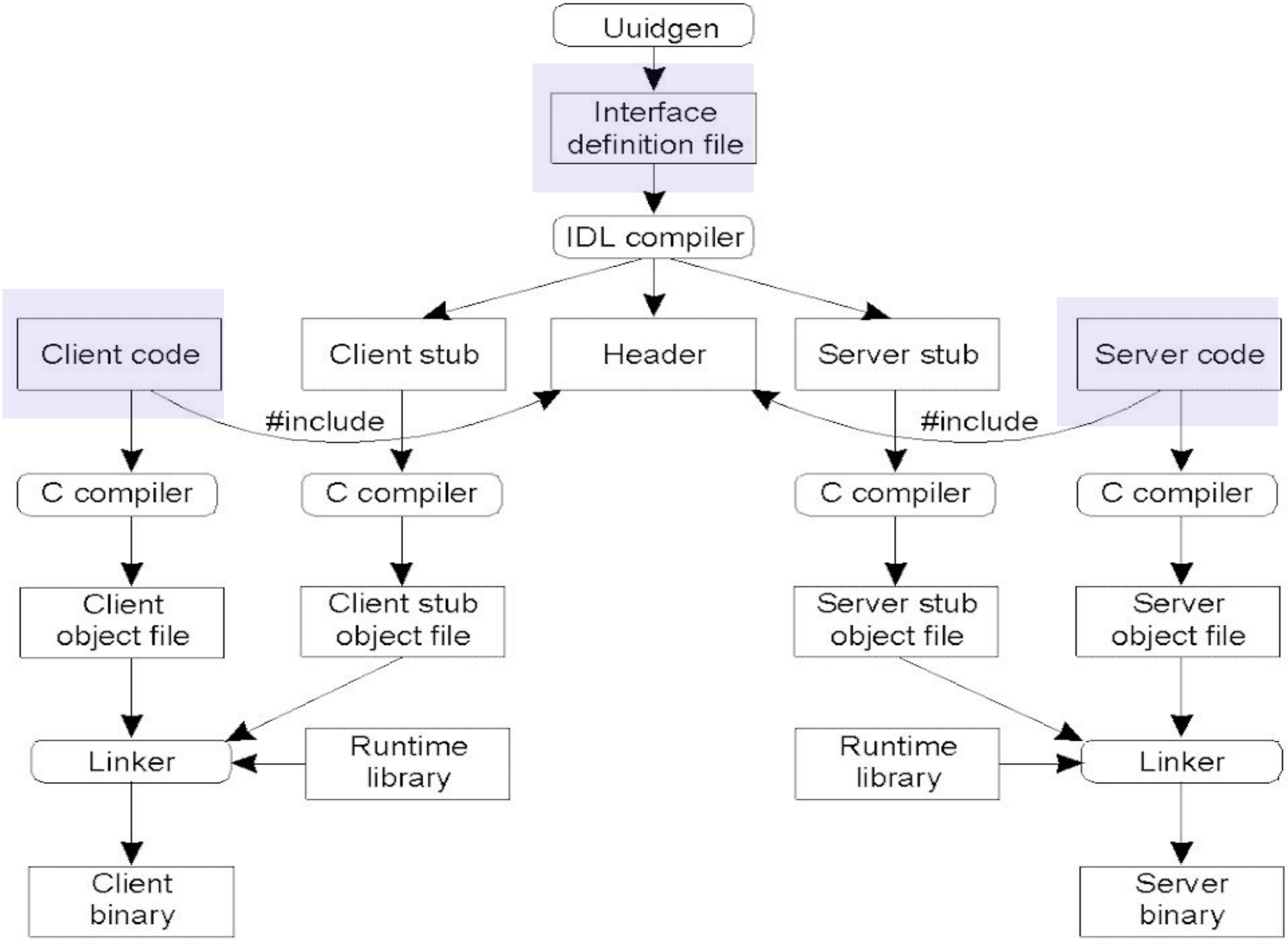
- the RPC ‘system’ (i.e., code generators and libraries) does the rest.

Components of an RPC system

- *Standards* for wire format of RPC msgs and data types. (e.g. Sun’s XDR)
- *Library* of routines to marshal / unmarshal data.
- *Stub generator* (aka “RPC compiler”) to produce "stubs".
  - For client: marshal arguments, call, wait, unmarshal reply.
  - For server: unmarshal arguments, call real fn, marshal reply.
- *Server framework*: Dispatch each call message to correct server stub, thread management, etc
- *Client framework*: Give each reply to the correct waiting thread.
- *Binding mechanisms*: how does the client find the right server?

Q: Poll: Where would Google’s Protocol Buffers fit above?

RPC in practice: Writing a Client and a Server



Outline

- Mechanics. How does it actually work …
  - … and limitations
- Discussion: do we achieve transparency?
  - Generic RPC framework
  - Case study: The Network File System

Issue (I): Does RPC achieve transparency?

Yes:

- Hides wire format and marshal / unmarshal leg-work.
- Hides details of send / receive APIs
- Hides details of transport protocol (TCP vs. UDP)
- Hides who the client is.

No:

- Latency, performance impact may be visible
- [Depending on the language one aims to support …] Limits on what data can be passed using RPCs. (example in next slide)
  - Typing info needed.
  - Breaks if context is wider than arguments (e.g., lobal variables, pointers)
- Concurrency
- Failures

You are to implement the RPC support for C.

C language has a construct called *union* where the same memory location can hold one of several alternative data types.

Example: The following piece of code declares a new union type *union_def*. Variables of this type can hold either one of an integer, a float or a character. Then in the variable is assigned a float then an integer.

```
union union_def { int a; float b; char c;} ;  // define the type
union union_def union_var;          // define the variable
union_var.b=99.99; or               // initialize the variable
union_var.a=34;
```

Can your RPC implementation transparently support 'union' type?   Explain your answer.

Issues (II): Should the IDL* break transparency?

- Original take (Sun RPC): attempt to provide *total* transparency

- Today (JavaRMI): same interface *but force programmer to handle exceptions for remote calls*
  - … the programmer has to acknowledge (s)he is making a remote call
  - Separate network errors from errors in the server code.

*IDL = interface definition language

Outline

- Mechanics. How does it actually work …
  - … and limitations
- War stories: Does one achieve transparency?
  - Case study: The Network File System

Case study: Network File System (NFS)

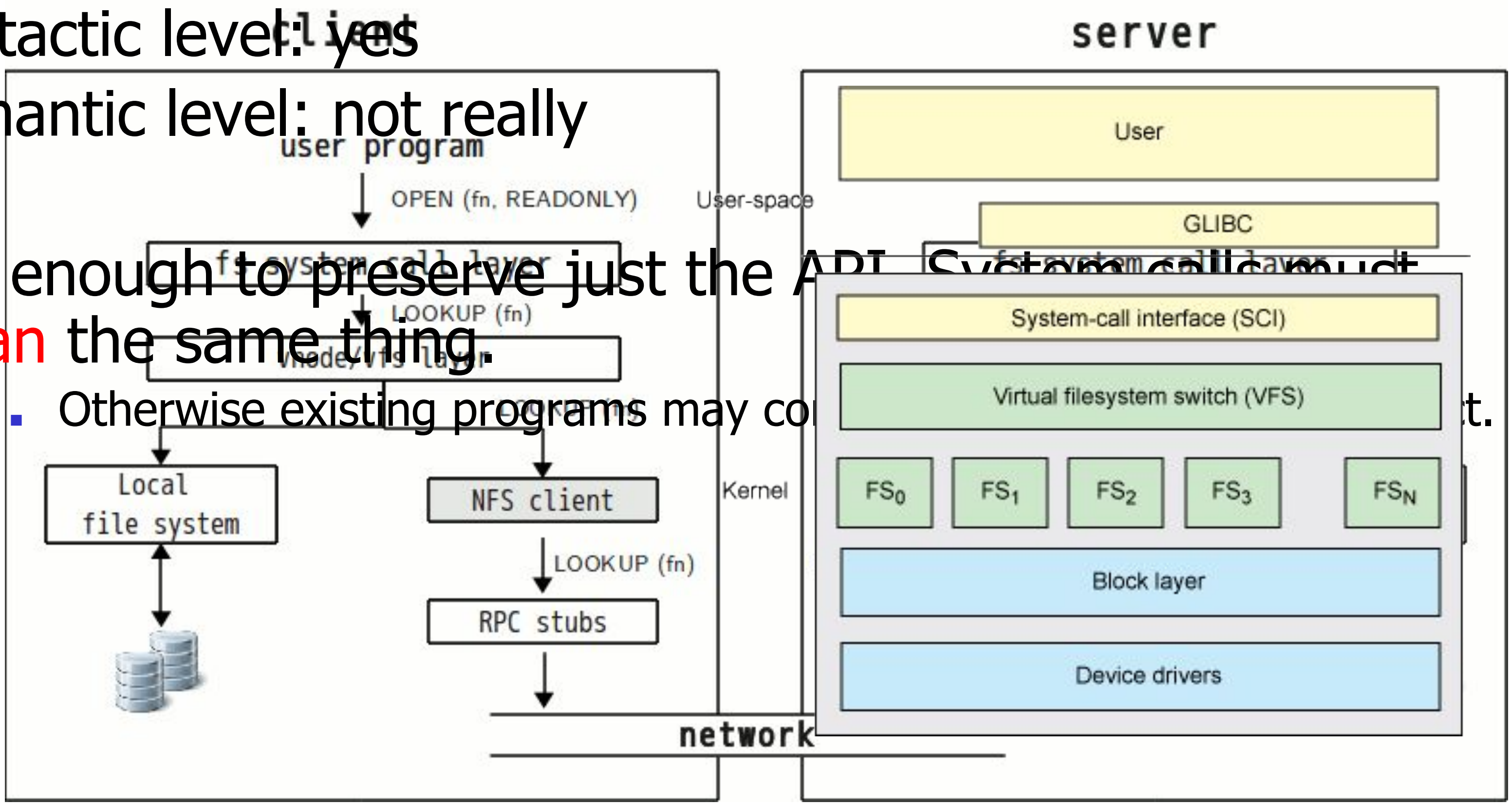
- What does the RPC split up in this case?
  - app calls, syscalls, kernel file system, local disk?

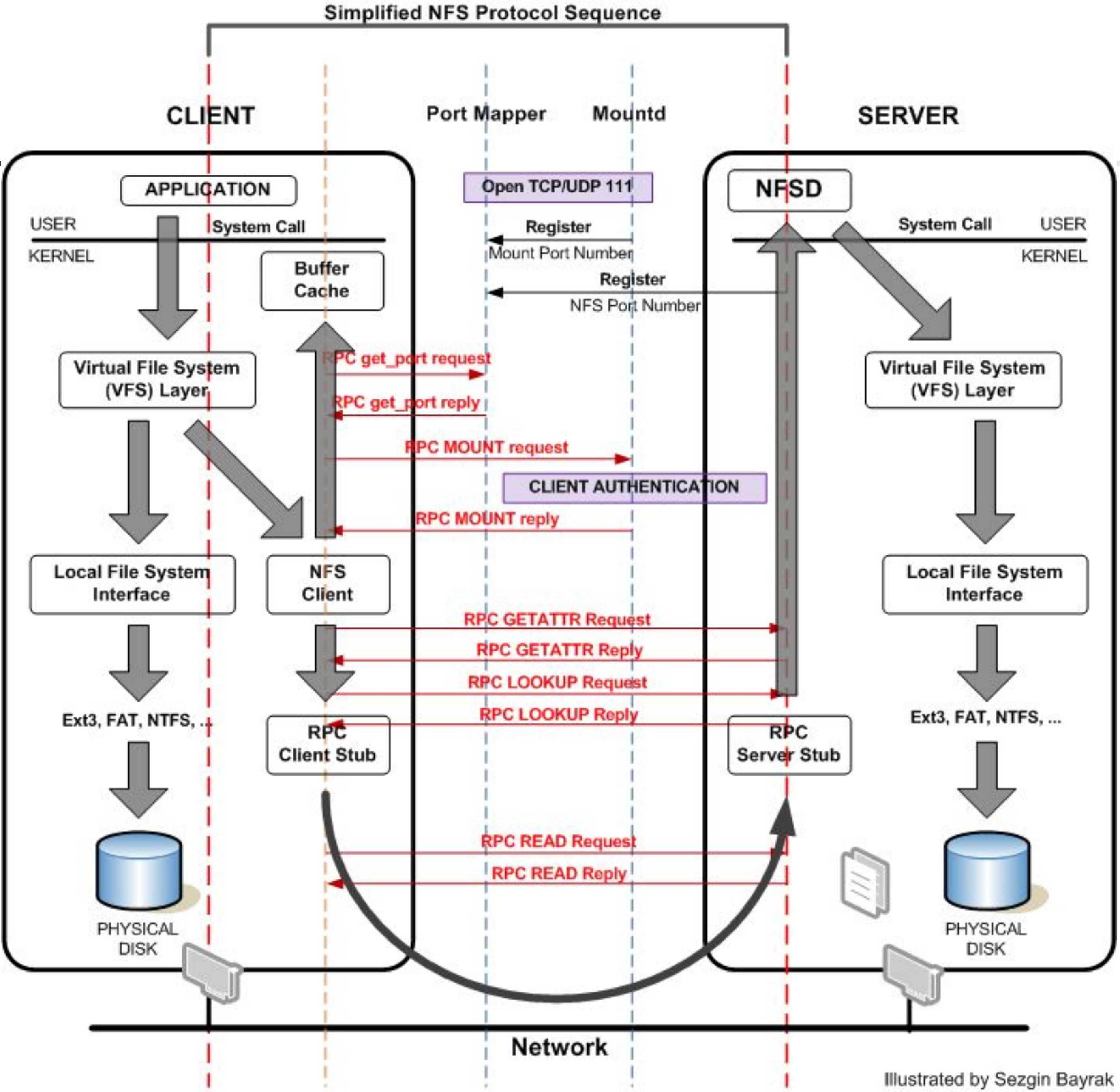
Is transparency preserved?

- Syntactic level: yes
- Semantic level: not really

Not enough to preserve just the API. System calls must **mean** the same thing.

- Otherwise existing programs may co t.





Simplified NFS Protocol Sequence

CLIENT

Port Mapper

Mountd

SERVER

APPLICATION

USER

System Call

KERNEL

Buffer Cache

Virtual File System (VFS) Layer

Local File System Interface

NFS Client

Ext3, FAT, NTFS, ...

RPC Client Stub

PHYSICAL DISK

Open TCP/UDP 111

Register

Mount Port Number

Register

NFS Port Number

RPC get_port request

RPC get_port reply

RPC MOUNT request

CLIENT AUTHENTICATION

RPC MOUNT reply

RPC GETATTR Request

RPC GETATTR Reply

RPC LOOKUP Request

RPC LOOKUP Reply

RPC READ Request

RPC READ Reply

NFSD

System Call

USER

KERNEL

Virtual File System (VFS) Layer

Local File System Interface

Ext3, FAT, NTFS, ...

RPC Server Stub

PHYSICAL DISK

Network

Illustrated by Sezgin Bayrak

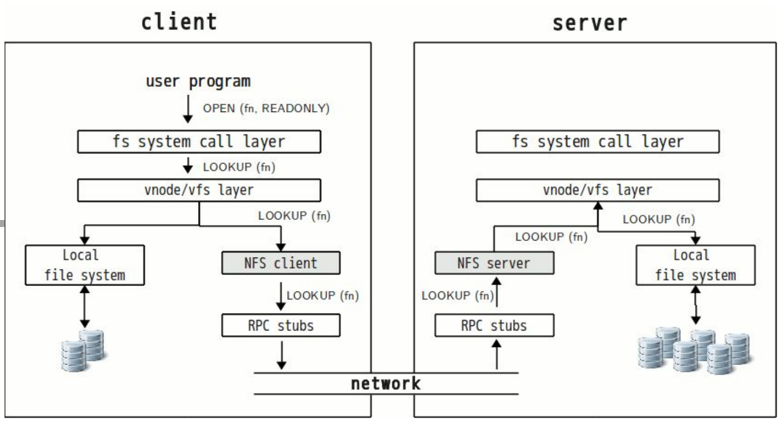
Does NFS preserve the semantics of file system operations?

New semantics: *Open()* system call:

- Originally (i.e., no RPC), *open()* only failed if file didn't exist
- Now open (and all others) can fail for other reasons (e.g., if server has died, network is down).
  - Obs: Apps have to know to retry or fail gracefully.
  - Obs: Think of process coordination through FS

This is fundamental, not an NFS quirk.

NSF: New Semantics (II)



New semantics: close() system call

- Originally client only waits for disk in *write()*
  - *close()* never returned an error for local file system.

- Now: *close()* might fail if server disk out of space.
  - So apps have to check *close()* for 'out-of-space' errors
    - as well as *write()*.
  - Why? Caused by additional optimizations: NFS trying to hide latency by batching.
    - Side effect of async write RPCs in client, for efficiency.
    - They could have made write() synchronous (and much slower!).

NSF: New Semantics … (III)

New semantics: deletion of open files

Scenario: I open a file for reading. Some other process deletes it while I have it open.

- Old behavior:
  - my reads still work!

- New behavior: my reads fail.
  - Side-effect of NFS's **statelessness**.
    - NFS server does not remembers all operations it has performed for each specific client.

How would one attempt to fix this?

NSF: New Semantics examples … (IV)

- Scenario:
  - rename("a", "b")
  - Suppose server performs rename, crashes before sending reply.

- NFS client re-sends rename(). But now "a" doesn't exist
  - (NFSv2: retries not identified) produces an error. This never used to happen.
  - v3 and after – mechanisms to identify retries

NFS: security

Security is totally different

- On local system: UNIX enforces read/write protections:
  Can't read my files w/o my password

- On [older versions of] NFS: Server believes whatever UID appears in NFS request
  - Anyone on the with network access can put whatever they like in the request

  Why aren't NFS servers ridiculously vulnerable?
  - Hard to guess correct file handles.
  - This is fixable (SFS, AFS, even some NFS variants do it)
    - Require clients to authenticate themselves cryptographically.
      - Hard to reconcile with statelessness.

RPC / NFS case-study summary

Areas of RPC non-transparency

- 1. Partial failure, network failure
- 2. Latency
- 3. Efficiency/semantics tradeoff
- 4. Security. You can rarely deal with it transparently.
- 5. Pointers. Write-sharing.
- 6. Concurrency (if multiple clients)

However, it turns out none of these issues prevented NFS from being useful.

- People fix their programs to handle new semantics.
- … install firewalls for security.
- And get most advantages of ‘transparent’ client/server.

Design stories

Environment: Unreliable communication, nodes may crash

- Context1 – (generic) RPC / network file system stories
  - Takeaway: not 100% 'transparent'. Pay attention to semantics
- Context2 – (specialization) Distributed garbage collection
  - Keep track of number of active [remote] references to an object (reference counting style)
  - Take away: Light-weight solutions are possible in certain cases.
- Context3: one-to-many communication

Environment: Unreliable communication, nodes may crash

Goal: Keep track of number of active ‘users’ of a [remote] entity

We’ve seen [somewhat similar] versions of this problem already

- [NFS] How many remote clients have this file open?
- [Request/Reply Protocol] Is the client done with ‘this’ request?
- [new context: garbage collection for a distributed object oriented language]
  How many remote clients have a reference to this object?

Questions to consider:

- How will creating the remote object and a reference to it work?
- How will passing a remote reference work?
  - P^1 attempts to pass to P^2 a remote reference to O
- What happens if a client crashes (while holding a remote reference)

Solution 0: Mark and Sweep? (Non-Distributed GC)

Solution 1: Increment/Decrement on top of a request/reply protocol

Solution 1: Increment/Decrement on top of a request/reply protocol

- Overview
  - Reference Counting

- Questions to consider:
- How will creating the remote object and a reference to it work?
  - Create object on remote server. Creation method returns reference.
- How will passing a remote reference work?
  - P^1 attempts to pass to P^2 a remote reference to O
- What happens if the client crashes (while holding a remote reference)
  - Handling network / client / server failures?

W4-Q2: GC on top of increment/decrement RPC

Assume you implement a reference counting garbage collection mechanism on top of an increment/decrement RPC.

Further assume the following fault scenarios: the network can lose packets, and nodes on which client code is deployed may fail. However, the nodes on which the remote objects are deployed will never fail (e.g., they use some form of hardware redundancy). [assuming, of course, a correct implementation]

Q1: Are there scenarios that lead to a resource leak (i.e., objects are never used anymore but they are never collected)?

□ Q1: YES - resource leak is possible

□ Q1: NO - no resource leak possible

Q2: Are there scenarios where the system will behave incorrectly: the garbage collection mechanism will inadvertently collect an object to which there are still active clients holding references

□ Q2: YES - the system may inadvertently garbage collect live objects

□ Q2: NO - the system will never collect objects that are live

Goal: Keep track of number of active ‘users’ of a [remote] entity

[context: garbage collection for a distributed object oriented language]

- How many remote clients have a reference to this object?

Questions to consider:

- How will creating the remote object and a reference to it work?
- How will passing a remote reference work?
  - P^1 attempts to pass to P^2 a remote reference to O
- What happens if the client crashes (while holding a remote reference)

Solution 0: Mark and Sweep? (Non-Distributed GC)

Solution 1: Increment/Decrement on top of a request/reply protocol

Solution 2: Reference Listing

Solution 2: Reference Listing (Java RMI's solution)

- Overview
  - Object 'stub' maintains **<u>a list</u>** of remote clients
  - Each remote client regularly sends a reminder 'I'm alive!'
  - 'stub' deletes entry from list if it's not refreshed for a certain time

- Questions to consider:
- How will creating the remote object and a reference to it work?
  - Create object on remote server. Creation method returns reference.
- How will passing a remote reference work?
  - P^1 attempts to pass to P^2 a remote reference to O
- What happens if the client crashes (while holding a remote reference)
  - Handling network / client / server failures?

W4-Q3: GC on top of reference listing

Assume you implement a reference counting garbage collection mechanism on top of reference listing

Assume the following failure scenarios: *(i)* the network can lose packets, and *(ii)* nodes on which client code is deployed may fail.

Q1: Are there scenarios that lead to a resource leak (i.e., objects are never used anymore but they are never collected)?

□ Q1: YES - resource leak is possible

□ Q1: NO - no resource leak possible

Q2: Are there scenarios where the system will behave incorrectly: i.e., the garbage collection mechanism will inadvertently collect an object to which there are still active clients holding references

□ Q2: YES - the system may inadvertently garbage collect live objects

□ Q2: NO - the system will never collect objects that are live

Reference Listing (Java RMI's solution)

- Advantages:
  - Simple protocol
  - Client failures handled gracefully. No explicit client failure detection
- Drawbacks
  - overheads/scalability – the list of client proxies can grow large
  - unannounced client failures may lead to ***temporary*** resource leak
  - possible early garbage collection if network partitions

No need to detect duplicates. Why does this work?

- List entry 'refresh' operations only.
- 'Refresh' operations are *idempotent (i.e.,* duplicate operations have no effect)
  - *… and* (in this case) relative ordering is not important.

A larger category: ‘Soft-state’ mechanisms or Leases

- *Mechanism*: (soft-state / leases)
  - Client holding a “lease” regularly sends lease renewal message to server
  - Server keep list of lease holders and associated timeouts. Deletes if no updat

The Altern

Object

Advantages:

(1) Decouples state producer and consumer

No explicit state removal messages

No explicit failure detection (of consumer)

Simpler server fail-over

Consequence: ‘Eventual’ state

(2) Lightweight. Fairly easy to tune

There are a number of hybrid possibilities that combine the two

A full analysis: *A Comparison of Hardstate and Softstate Signaling Protocols*, Ping Ji et al., SIGCOMM 2003

Design stories

- Context1 – generic: RPC / network file system (NFS) stories
  - Takeaway: not 100% 'transparent'. Pay attention to deviations in semantics

- Context2 – specialization: distributed garbage collection
  - Keep track of number of active [remote] references to an object (reference counting style)
  - Take away: Light-weight solutions are possible in certain cases.

- Context3: one-to-many communication

Environment: Unreliable communication, nodes may crash