EEL-4736/5737 Principles of Computer System Design

Lecture Slides 6

Textbook Chapter 4

Client-service Organization – Enforcing Modularity

Introduction

- Modular design does not imply modularity is enforced
  - "Soft" modularity can be bypassed due to programming error, or malicious intent
- Let us study approaches that have been used to enforce modularity
  - Clients and services
  - Virtualization (ch 5)

An example

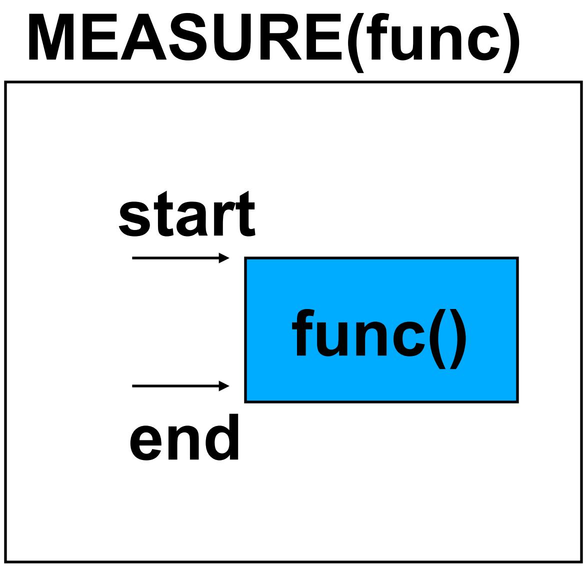
procedure MEASURE (*func*)

start <- GET_TIME (SECONDS)

func()

end <- GET_TIME (SECONDS)

return *end-start*



procedure GET_TIME (*units*)

time <- CLOCK

time <- CONVERT_TO_UNITS (*time*, *units*)

return *time*

Modularity

- *Why*?
  - From the perspective of measurement, you want to measure a difference in seconds, and support multiple systems
    - Which units? Which syscall API?
  - Split measurement from clock interactions
    - GET_TIME() can be different depending on hardware
- *How*?
  - Two procedures run in the same application process: same register set, same memory
  - Procedure call stack convention implemented by compiler

Stack discipline

- Typical way to support for procedure calls – stack memory abstraction
  - Push – add to top of stack
  - Pop – retrieve from top of stack
  - Different instruction sets implement this abstraction in different ways
    - PUSH/POP instructions
    - Stack pointers and loads/stores
    - Stack grow up or down
- At the end of invocation of a procedure
  - Should leave the stack as was found

An example

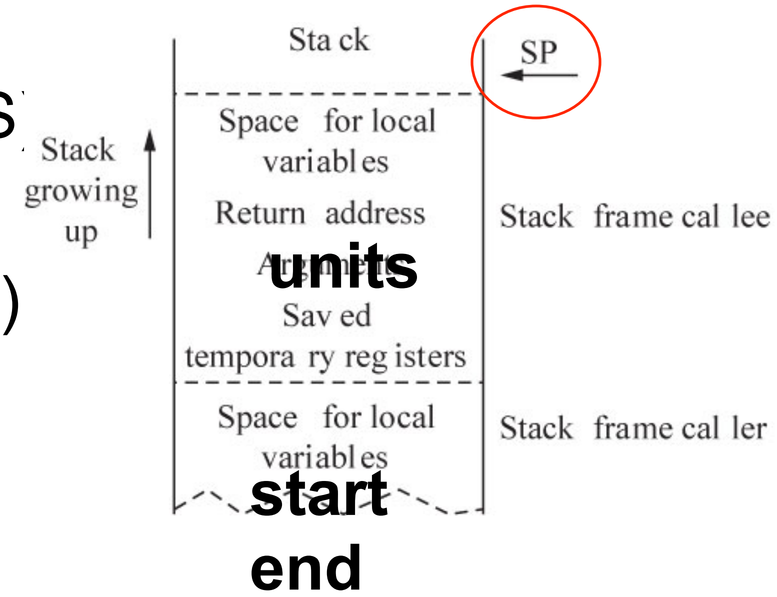
procedure MEASURE (*func*)

    start <- GET_TIME (SECONDS)

    func()

    end <- GET_TIME (SECONDS)

    return *end-start*



procedure GET_TIME (*units*)

    time <- CLOCK

    time <- CONVERT_TO_UNITS (*time*, *units*)

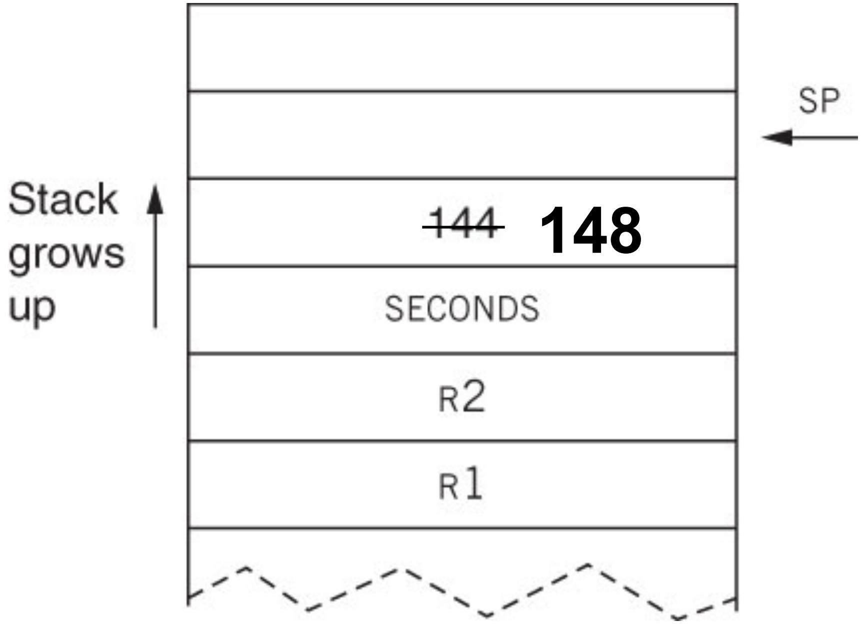
    return *time*

RISC-like machine code

MEASURE:

100:	STORE R1, SP	**Save registers that will use**
104:	ADD 4, SP	
108:	STORE R2, SP	
112:	ADD 4, SP	
116:	MOV SECONDS, R1	**Argument for callee**
120:	STORE R1, SP	
124:	ADD 4, SP	
128:	MOV 148, R1	**Return address**
132:	STORE R1, SP	
136:	ADD 4, SP	
140:	MOV 200, R1	**Call GET_TIME**
144:	JMP R1	
148:	…	

Peeking into the stack



RISC-like machine code

GET_TIME:

200:	MOV SP, R1	**Position R1 as index to read R2=argument**
204:	SUB 8, R1	
208:	LOAD R1, R2	
212:	… get_time body	
220:	MOV time, R0	**R0 = return value**
224:	MOV SP, R1	**Load return address**
228:	SUB 4, R1	
232:	LOAD R1, PC	

148: Back to MEASURE; adjust stack (sub 16), restore registers R1, R2, use R0

Stack convention

- This example assumes that:
  - Callee leaves stack as it was found
  - Callee returns where caller told it to
  - Callee places return value in R0
  - Caller places values of temporary registers in stack not expecting callee to change them
- This is a **convention**
  - The compiler follows this convention when generating the machine code
- Suppose now that GET_TIME is linked to an object file

Stack convention

- This example assumes that:
  - Callee leaves stack as it was found
    - A mistake in GET_TIME may set the wrong value for SP
      - Cascades to MEASURE getting incorrect values for R1, R2; other procedures which called MEASURE?
  - Callee returns where caller told it to
    - A malicious programmer can try to return somewhere else so to gain control of sensitive information
  - Caller places values of temporary registers in stack not expecting callee to change them
    - Mistake may cause callee to overwrite stack and register values restored
- Caller will fail if convention is not followed
  - Specification; without enforcement

Other opportunities

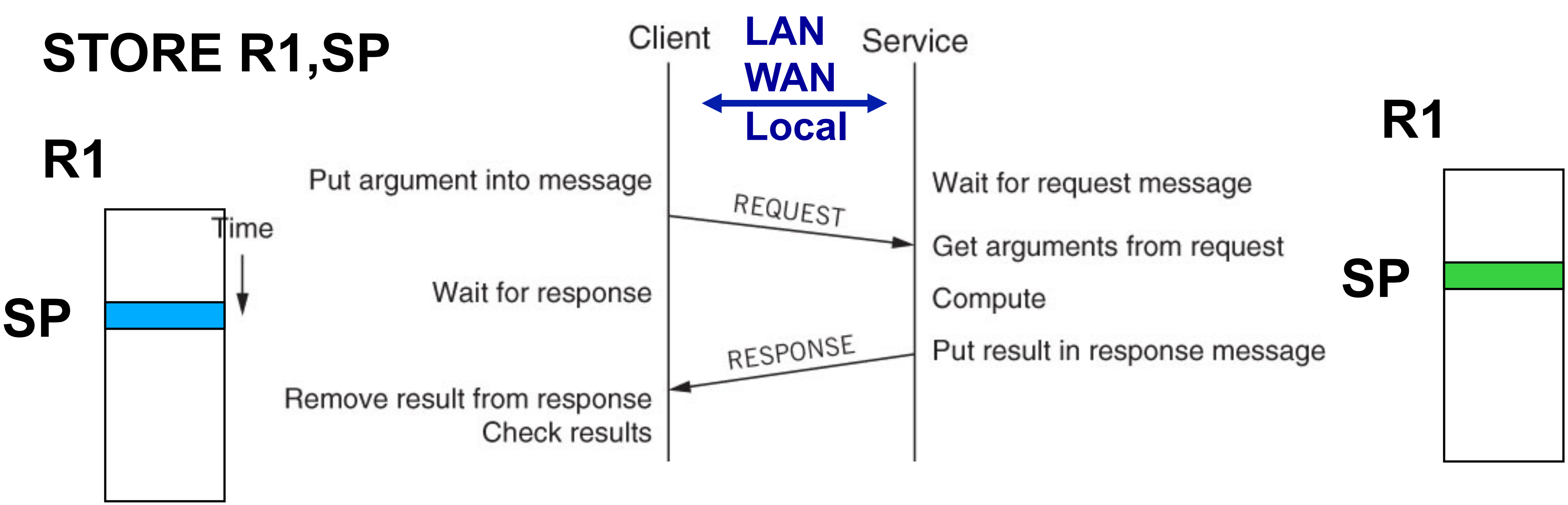
- Errors can propagate when memory is improperly modified
  - E.g. a procedure that is passed a pointer to a data structure
  - Global variables
- *Procedures share the same memory name space and the same register set*
  - The same context – a process

Language and run-time support

- Languages have evolved to support techniques to ‘beef up’ soft modularity
  – “Strongly typed” – Java and C#
  – Language and runtime restricts that programs write only to memory locations that correspond to variables managed by the language
    - In accordance with their type
    - E.g. cannot use an Integer variable as an address of a memory location
- Still, many applications or modules written in languages that don’t enforce modularity

Client/Service Organization

- Limit interaction among modules to explicit messages
  - Contrast with the previous example, where both procedures are in the same memory name space and share registers



Client/service organization

- Benefits
  - Isolation
  - Ability to run clients/services across two or more computers
    - Scalability, redundancy
- Disadvantage
  - Performance cost
    - Packing and sending messages has an overhead

Revisiting our example

procedure MEASURE (*func*)

SEND_MESSAGE (*NameForTimeService*, {“Get Time”, CONVERT2EXTERNAL(SECONDS)})

response <- RECEIVE_MESSAGE (*NameForClient*)

start <- CONVERT2INTERNAL (*response*)

func()

…

Revisiting our example

```
procedure TIME_SERVICE ()
    do forever
    request <- RECEIVE_MESSAGE (NameForTimeService)
    opcode <- GET_OPCODE (request)
    unit <- CONVERT2INTERNAL
    (GET_ARGUMENT(request))
    if opcode = "Get time" and (unit = SECONDS or unit =
    MINUTES) then
        time <- CONVERT_TO_UNITS (CLOCK, unit)
        response <- {"OK", CONVERT2EXTERNAL(time)}
    else
        response <- {"Bad request"}
    SEND_MESSAGE (NameForClient, response)
```

New things to worry about

- CONVERT2INTERNAL / EXTERNAL
  - Two computers may use different internal representations for data – big endian, little endian

Words	0				1			
Bytes	0	1	…	7	0	1	…	7
Bits	$2^0\ 2^1\ 2^2\ldots$			$\ldots 2^{63}$	$2^0\ 2^1\ 2^2\ldots$			$\ldots 2^{63}$

Words	n				$n-1$			
Bytes	7	…	1	0	7	…	1	0
Bits	$2^{63}\ldots$			$\ldots 2^2\ 2^1\ 2^0$	$2^{63}\ldots$			$\ldots 2^2\ 2^1\ 2^0$

Send 0xABCDh, receive 0xDCBAh

New things to worry about

- *Marshaling*
  - Conversion of all data to be sent in a message into an array of bytes
  - Suitable for transmission and proper conversion to an object by the receiver
- Need to name and discover client and service endpoints
- Need to name and discover which services are exposed and their interface

Enforcing modularity

- Client/service don't rely on *shared state*
  - E.g. a stack or global variables
  - Failure in one does not directly corrupt data in the other
- Many errors cannot propagate
  - Client does not need to trust the service to return to appropriate return address
    - The service does not control that
    - Arguments can be checked when unmarshalled

Enforcing modularity

- Client can recover from service failure
  - Service goes into infinite loop?
    - Client can time out on waiting for response
    - Client may initiate a recovery process – e.g. try a backup, redundant service
  - Setting appropriate timers is not trivial
  - Added complexity that needs to be incorporated in the program (MEASURE example does not have it)

Summary

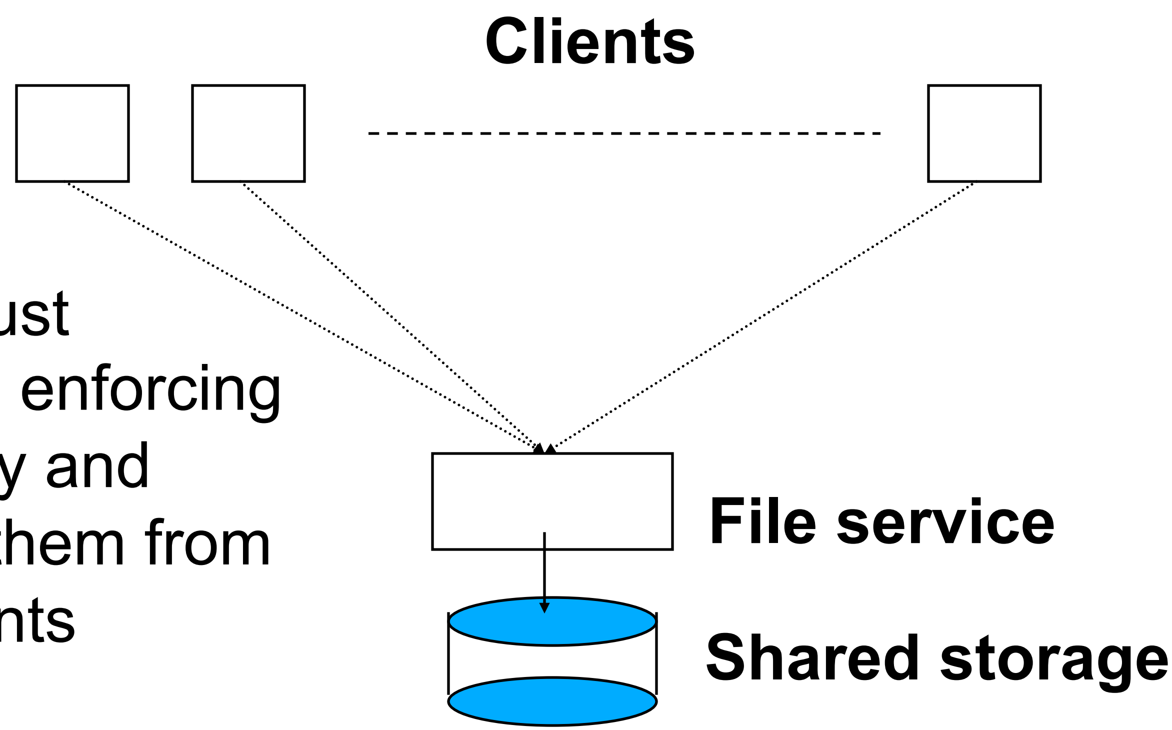
- Example of *sweeping simplification*
  - Remove all forms of interaction other than messages
- An improvement, but not a panacea
  - Service may allow messages to result in writes of value to any address in its space
- Trade-off
  - Shared data: ease of access within module
  - But, opportunity of error propagation
- Performance considerations put a bound on the size of modules

Additional flexibility

- One service can work for multiple clients
  - E.g. a printer or file server
- One client can use several services
  - E.g. a Web browser
- Single module can be both client and service
  - E.g. a proxy/cache

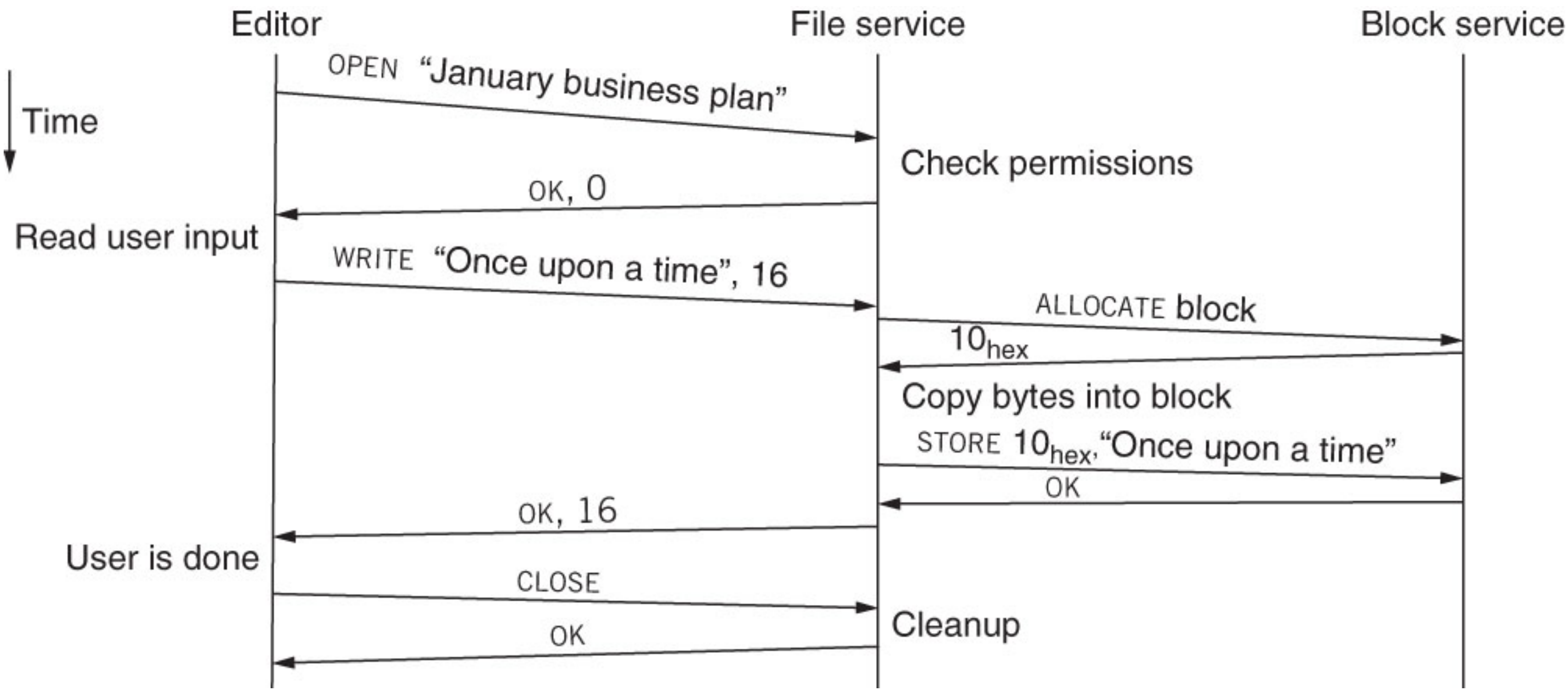
Trusted intermediaries

- Example: file service



Clients trust service in enforcing modularity and isolating them from other clients

File service



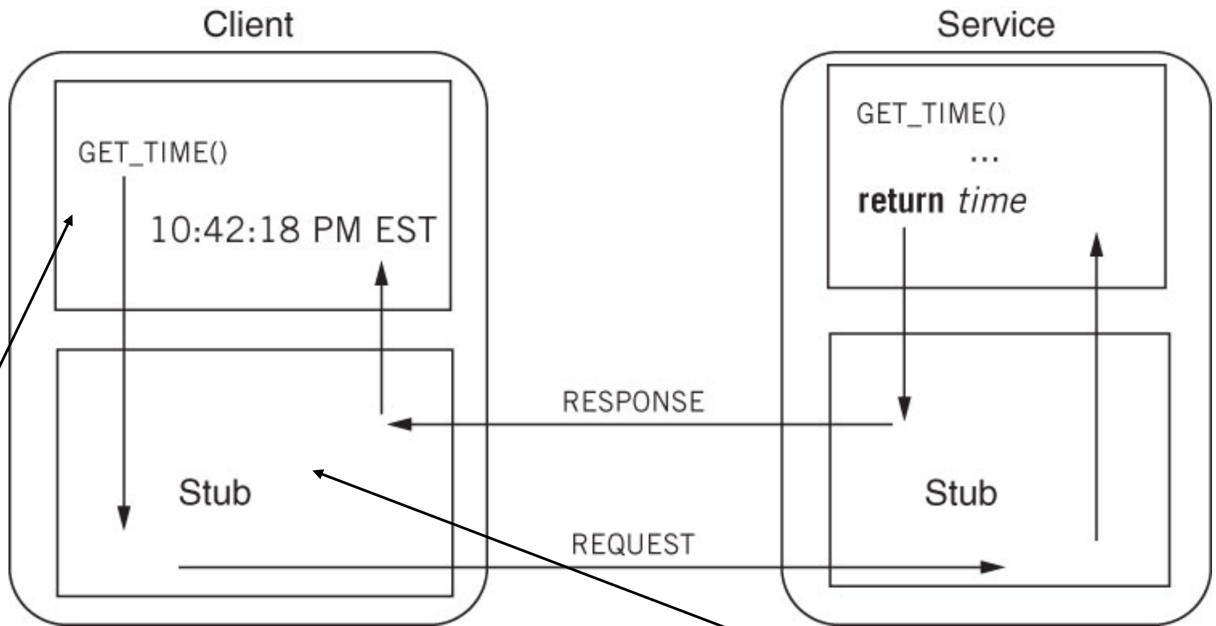
Trusted intermediaries

- There are general downsides
  - Vulnerability in the trusted intermediary can break the modularity enforced by the server
    - (Distributed) denial-of-service
      - Other clients bring service to a crawl
    - Vulnerability exploit
      - E.g. may gain access to other client's files
  - Must trust the intermediary
    - Eavesdropping? Privacy? Censorship?
- Peer-to-peer computing
  - Without intermediaries
  - Each peer a client and a service

Remote Procedure Call - RPC

- A typical client/service interaction has similar semantics to a procedure call
  - Client sends message to initiate a call to a named function with arguments
  - Service receives message, retrieves arguments, processes function
  - Service sends message with return value
  - Client receives message, retrieves return value
- Such a common pattern that it is useful to provide an abstraction and convenient APIs that hides implementation details

RPCs



procedure MEASURE (*func*)

start <- GET_TIME (SECONDS)

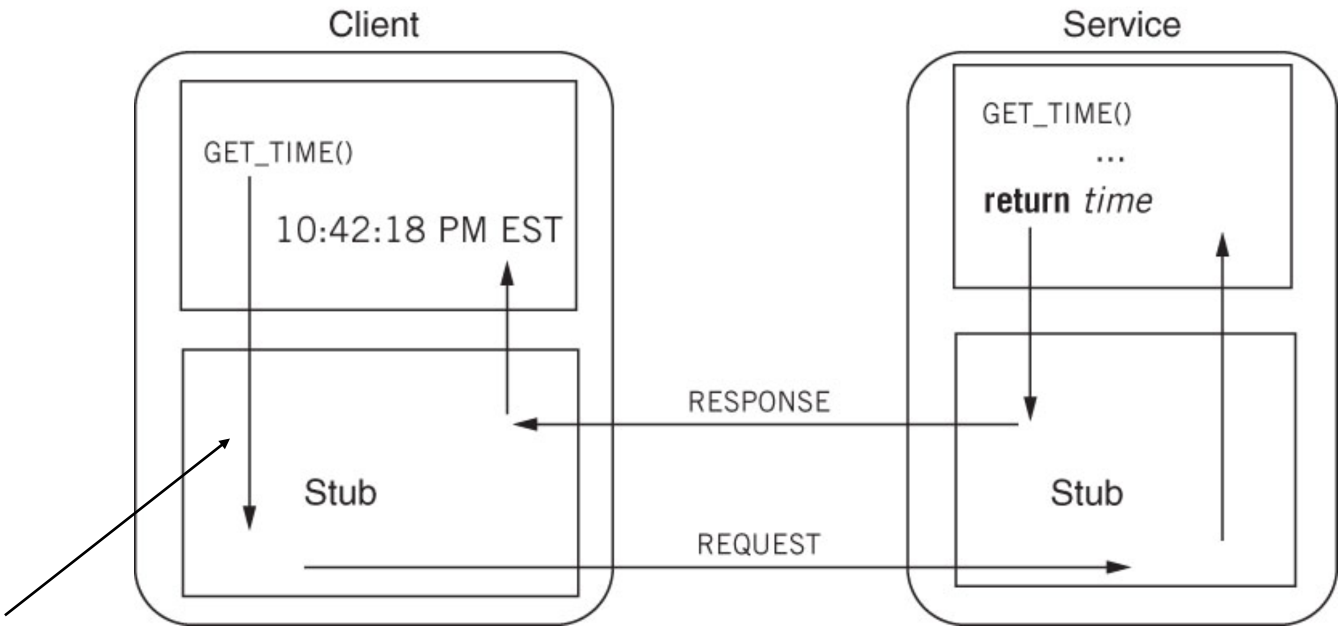
func()

end <- GET_TIME (SECONDS)

return *end-start*

Hides marshalling, communication details

RPCs



procedure GET_TIME (*unit*)

SEND_MESSAGE (NameForTimeService, {"Get Time",*unit*})

response <- RECEIVE_MESSAGE (*NameForClient*)

return *response*

Using RPCs

- Stubs can be generated automatically
- Need additional logic to handle faults

```
procedure MEASURE (func)
        try
                start <- GET_TIME(SECONDS)
        catch (signal servicefailed)
                return servicefailed
        func ()
        try
                end <- GET_TIME(SECONDS)
        catch (signal servicefailed)
                return servicefailed
        return end - start
```

RPC != PC

- RPCs take more time than PCs
- RPCs can reduce “fate sharing”
  - Callee failure causes caller to also fail
- RPCs introduce new failures that are not present in procedure calls
  - Service failure
  - *No response, timeout*
    - How to handle this? Not possible to determine if no response happened before or after service performed the action

Dealing with no-response

- At-least-once RPC:
  - Client stub re-sends as many times as needed until it gets a response
    - Client stub may still give up, after multiple re-tries – cannot provide guarantee implied by name
  - Service needs to be ready to execute many times the same request
  - Read? Write? Append?
    - Side-effect-free / idempotent operations

Dealing with no-response

- At-most-once RPC:
  - Client tries once; if no response, stub returns error to caller
  - More appropriate for requests with side effects
    - E.g. RPC to transfer funds from one account to another – at-most-once to avoid undesired transfers
  - Implementation needs to account for the underlying network – which may duplicate messages

Dealing with no-response

- Exactly-once RPC
  – Ideal semantics, but in principle impossible to guarantee if client and service are independent
    - What if service goes away forever?
  – Techniques can be used to implement an approximation of this semantics under certain assumptions

Reading

- Sections 4.2, 4.3, 4.5