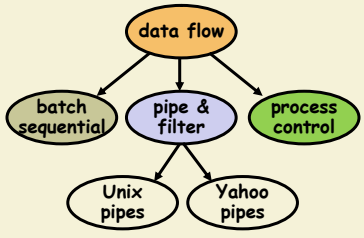
Software Architecture

Lecture 3
Call-Return Systems

João Pedro Sousa

George Mason University

last class
data flow styles



- process control
  - looping structure to control environment variables
- batch sequential
  - sequential processing steps, run to completion
  - typical of early MIS applications
- pipe & filter
  - incremental transformation of streams
  - Unix pipes and Yahoo pipes are special cases (sub-styles)

today
call-return styles

data flow
- batch sequential
- dataflow network (pipe & filter)
  - acyclic, fan-out, pipeline, Unix
- closed loop control

call-and-return
- main program/subroutines
- information hiding – objects
- stateless client-server
- SOA

interacting processes
- communicating processes
- event systems
  - implicit invocation
  - publish-subscribe

data-oriented repository
- transactional databases
  - stateful client-server
- blackboard
- modern compiler

data-sharing
- compound documents
- hypertext
- Fortran COMMON
- LW processes

hierarchical
- tiers
  - interpreter
  - N-tiered client-server

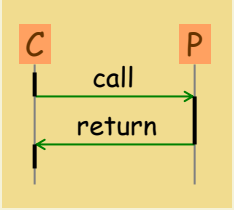
today's outline

- call-return styles
- single process flavors
  - main-subroutine, layers, modules, objects
- distributed flavors
  - components, tiers, cloud
  - implementing distributed call-return
- large-scale distributed, open-system flavors
  - SOA: later in the course

Acknowledgment
some of the material presented in this course is adapted from 17655, taught to the MSE at CMU by David Garlan and Tony Lattanze

in call-return styles

consumer components invoke functionality in provider components



- usually the caller waits until an invoked service completes and returns results before continuing
- components depend on invoked functionality to get their own work done
- the correctness of each component may depend on the correctness of the functionality it invokes

call-return has had many flavors throughout history

- subroutines
  - decomposition of main program into processing steps
- functional modules
  - aggregation of processing steps into modules
- abstract data types [Parnas]
  - bundle operations and data, hide representations and other decisions
- objects
  - sub-typing, polymorphism, dynamic binding of methods
- client-server
  - distribution, tiers
- components
  - multiple interfaces, advanced middleware services
- services
  - late binding of providers

in call-return styles
system topology comes in different flavors

- client-server
  - star: clients call servers
  - clients may register callbacks
- tiered
  - hierarchical star: layer n calls layer n+1
- peer-to-peer
  - no restriction
  - components define one or more interfaces
    - provides: a set of functionality that is offered
    - requires: a set of functionality that others must provide

in call-return styles
connectors piggyback control & data

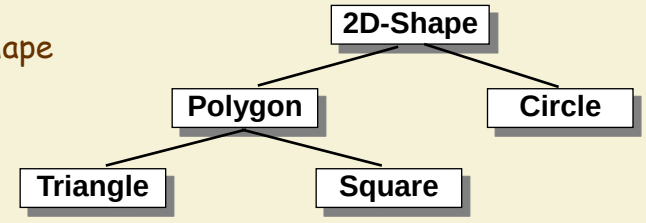
- consumers know the identity of providers upon which they rely
- control
  - building block: call
    - consumer blocks until a request is serviced
  - complex protocols may be built
    - example: client initializes server, starts a session, makes repeated requests, closes the session
- data flows in 2 ways
  - parameters: from consumer to provider
  - return values: from provider to consumer

in call-return styles
functional correctness is hierarchical

- the correctness of each component may depend on the correctness of the functionality it invokes
- leads to a pre/post-condition style of specification
  - pre = conditions under which a service may be requested
  - post = the result of having made a service request
- binding time of a provider may vary
  - static = at compile time
    - example: traditional compilers & ADTs
  - dynamic = at run time
    - example: OO method dispatch for class hierarchies
  - brokered (also at run time)
    - use broker to find components/service providers

example
dynamic binding

- object 2D-Shape



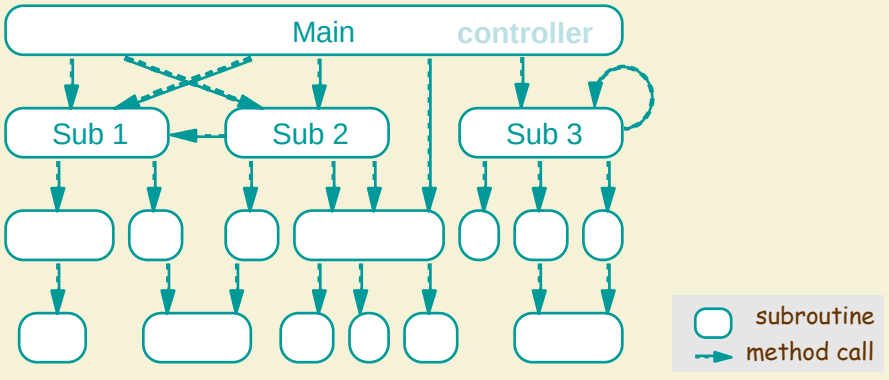
- 2D-Shape defines method Perimeter(): Real
  - same operation, but implemented differently for Triangle, Square, and Circle
  - if an object A is of any subtype of 2D-Shape a consumer may call A.Perimeter(), and the appropriate method will be chosen at run-time

outline

- call-return styles
- single process flavors
  - main-subroutine, layers, modules, objects
- distributed flavors
  - components, tiers, cloud
  - implementing distributed call-return

main-subroutine flavor

fairly unrestricted topology

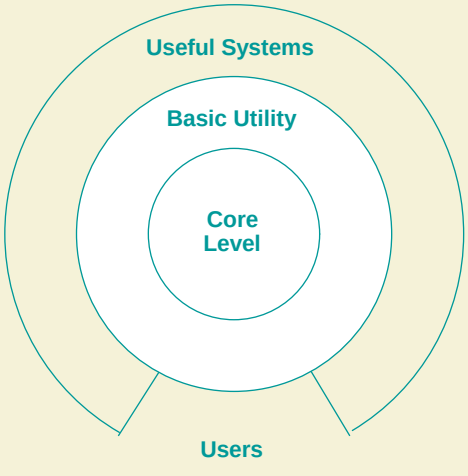


- frequently single thread of control
  - one component in the C&C view

main-subroutine flavor
subroutines indicate code structure

- "hierarchical" decomposition
  - based on definition-use relationship
- hierarchical reasoning
  - correctness of a subroutine depends on the correctness of the subroutines it calls
- subsystem structure implicit in the hierarchy
  - in large systems, "hierarchy" may become spaghetti

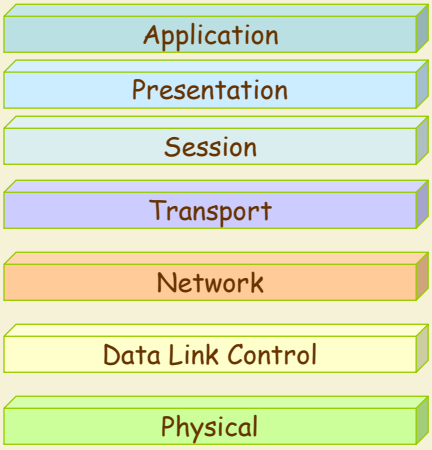
layers
enforce hierarchy



each layer

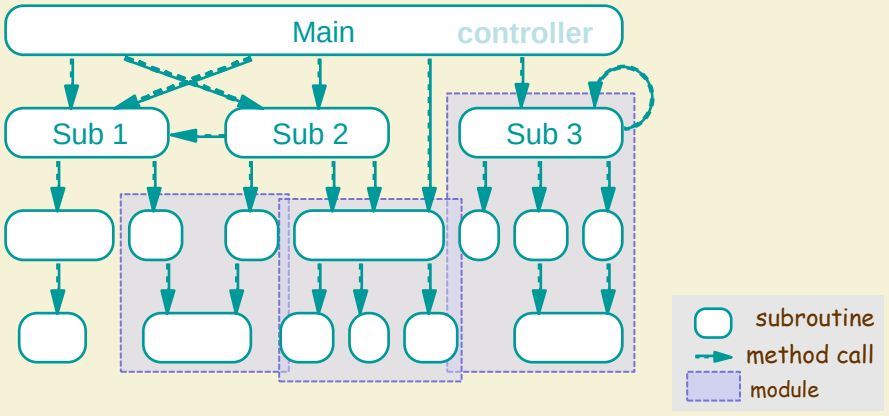
- provides a set of services to the layers above
- encapsulates a set of implementations and lower-level services
- relies on services from the layers below

example of layers
network protocol stack



more about this later in the course

modules
alternative way to rein in complexity



modules are:

- (naïve) a piece of code
- compilation unit, including interface declarations
- (Parnas) a place to encapsulate decisions

modules are good for:

- *management*: partition overall development effort
  - divide and conquer
- *understanding*
  - divide and conquer
- *evolution*: separation of concerns
  - changes to one module are isolated from others

in the 70's, modules turned into Abstract Data Types (ADTs)

- good programmers shared an intuition
  - if you get the manipulation of data structures right, the rest of the program is much simpler.
- ADTs' contributions:
  - *structure* representation bundled with operators
  - *rules for combining types* declarations
  - *language support* modules, scope, user-defined types
  - *specifications* abstract models, algebraic axioms
  - *integrity constraints* invariants of data structures
  - *information hiding* protect internal structure
- this is routine practice now part of O-O

in the 80's, ADTs turned into objects

- inheritance
  - share definitions of functionality
- polymorphism/dynamic binding:
  - determine actual operation to call at run-time
- capture families of related designs
  - inheritance hierarchy
- natural mapping to real world or domain
  - if we understand the domain then we are led to a natural system structure based on the domain
  - improve understandability, maintainability

objects have limitations:

- **scalability** on the number of object classes
  - vast numbers of classes requires additional structuring
  - hierarchical design suggested by Booch and Parnas
- **scalability** on the number of interactions
  - single interface can be limiting & unwieldy
  - push to permit multiple interfaces led to components
- overall system behavior hard to understand
  - distributed responsibility for behavior
  - interaction diagrams now used in design
- hard to capture similarities at the system level
  - object classes are fine grain
  - push led to design patterns & product lines

objects were not the end of the story

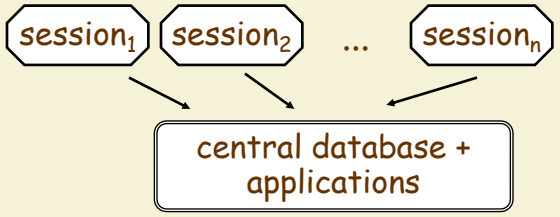
outline

- call-return styles
- single process flavors
  - main-subroutine, layers, modules, objects
- distribution
  - components, tiers, cloud
  - implementing distributed call-return

more complex, distributed applications led to enhanced call-return flavors

- client-server
  - objects are processes
  - asymmetric: client knows about servers, but not vice versa
- tiered
  - elaboration on client-server
  - aggregation into run-time layers (tiers)
  - usually small number of tiers
- components (ex.?)
  - multiple interfaces
  - special infrastructure supports finding and communicating
- compound documents (ex. ?)
  - document contains a set of embedded objects

early information systems predecessors of client-server

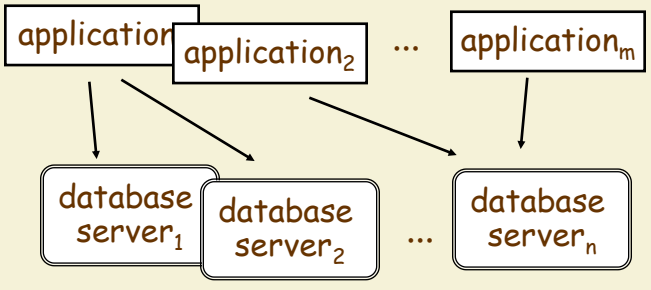


terminals or dumb PC

mainframe or server

- main features
  - no processing on terminals
  - centralized, secure, monolithic system (server)
  - single vendor for HW and SW
- forces for change
  - cheap PCs, bottlenecks in central unit

early client-server systems have two tiers

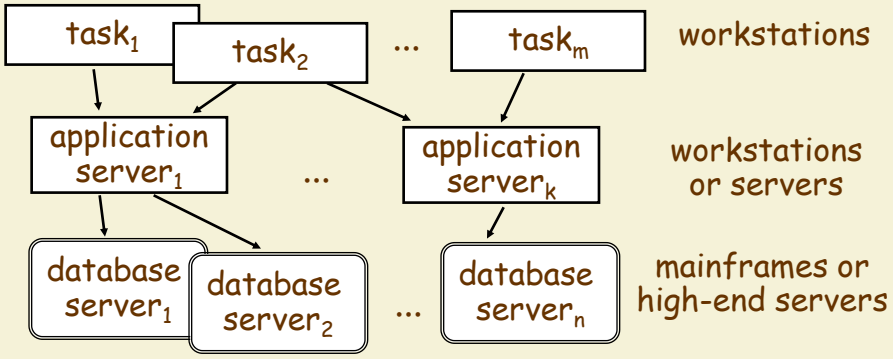


workstations

mainframes or servers

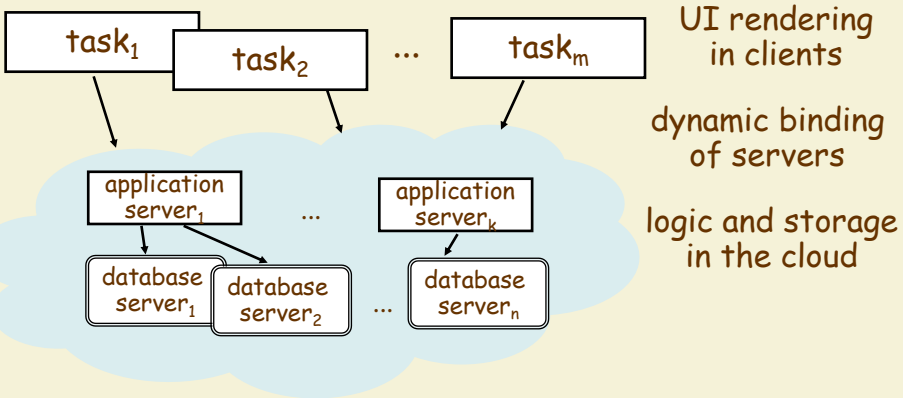
- main features
  - some processing moved out to workstations
  - applications/data are distributed on secure, specialized servers
- forces for change
  - increased numbers of clients
  - application complexity
  - server bottlenecks

many modern systems have several tiers



- main features
  - increased server specialization
  - client tasks invoke many different remote applications

the cloud
rely on servers for storage and processing



- drivers: mobility, reliability, scalability…
- challenges: privacy and security trusted to 3rd parties

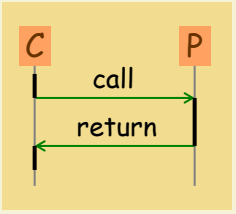
take 5

outline

- call-return styles
- single process flavors
  - main-subroutine, layers, modules, objects
- distribution
  - components, tiers, cloud
  - implementing distributed call-return

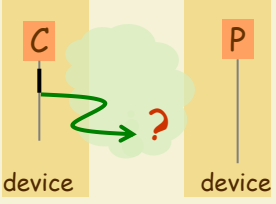
in early main-subroutine

consumer and provider in the same address space



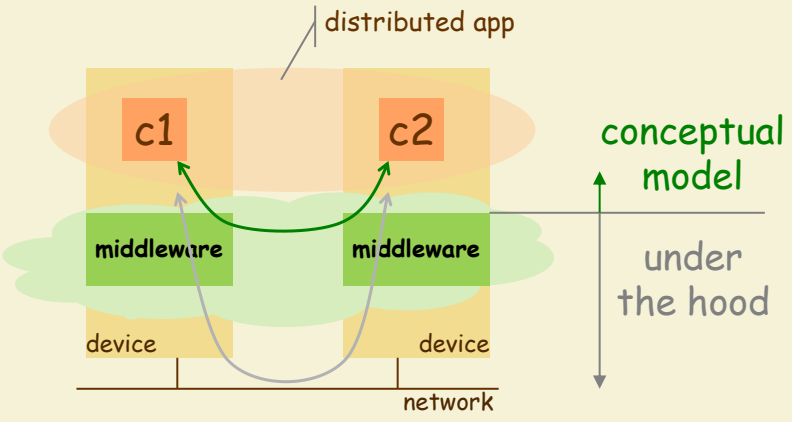
- compiler resolves address of provider
- parameter values and return results passed on the call stack
- consumer and provider may access shared variables, e.g. by passing parameters by reference

same implementation mechanisms do not apply across process/machine boundaries

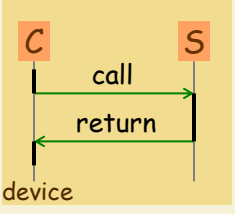


- compiler cannot resolve address of provider on a different address space/machine
- how to pass parameters & results to a different process/machine?

call-return conceptual model supported by middleware

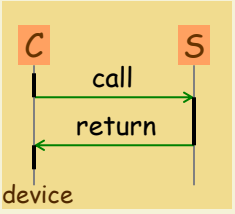


lifting the hood on RPC

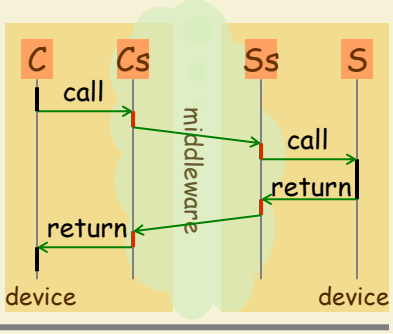


- putting the R in Procedure Calling
- how to pass parameters?
- how about shared memory?
- handling limitations in practice

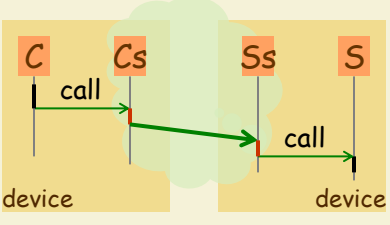
idea: stubs hide communication

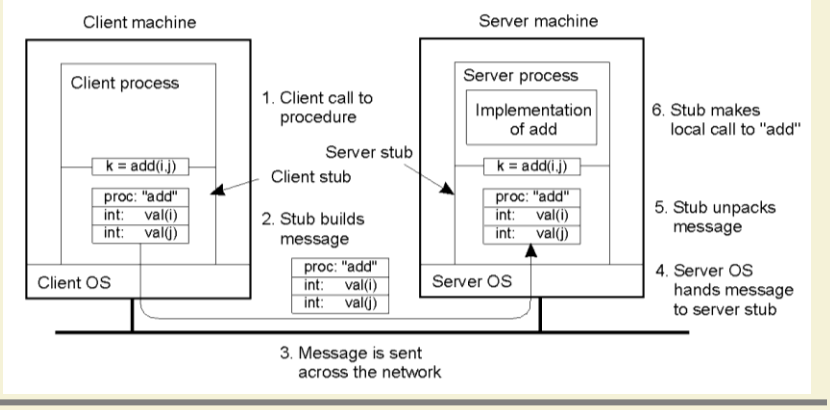


- client stub, aka server proxy, appears to C like a server running on the client device
- server stub, aka client skeleton, appears to S like a client running on the server device

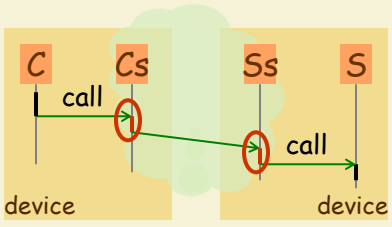


RPC is implemented by sending messages



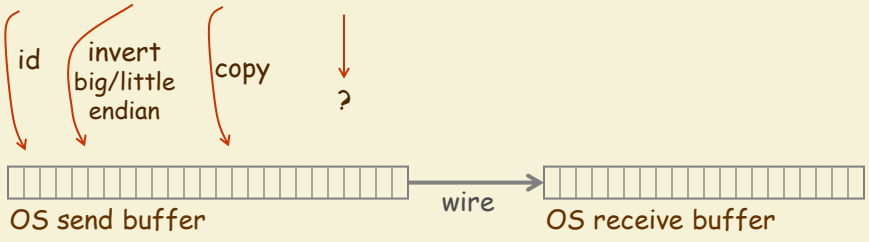


marshalling parameters is type-specific and platform-specific

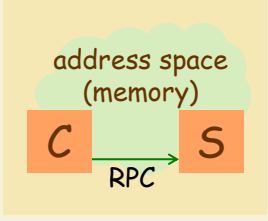


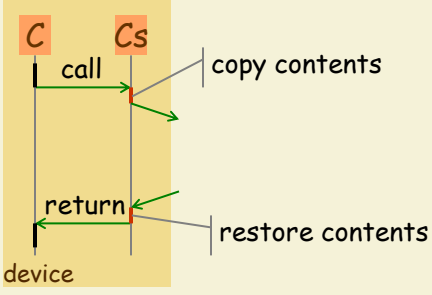

```
char *myString;
…
someProc(257,"Fred",myString);
```

```
void someProc(int d,
              char *n,
              char *m){…
```



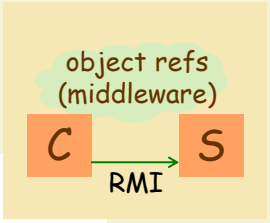
simulate shared address space to some extent

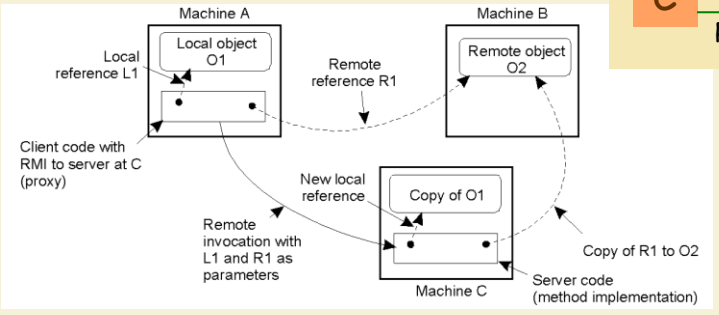




- references to simple, small structures resolved by copy/restore
- complex data structures not supported (structure contains pointers, e.g., linked lists)

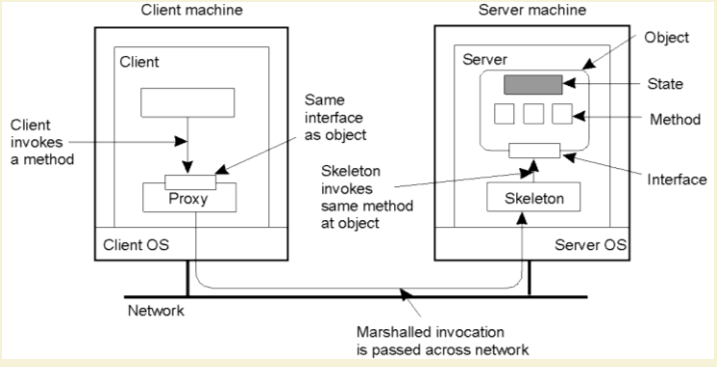
solution: increase granularity from bytes to objects





- both local objects and references to remote objects are passed by value (serialization)
- the result of the called method is also serialized and passed back to the caller

RMI uses similar ideas to RPC



- communication facilitated by local stubs (proxy/skeleton)
- stubs define/support an interface for method calling
- calling and return implemented by message passing
- separate mechanisms for dynamic binding (object registry)

RMI is different from RPC
in a number of ways

- doesn't try to hide distribution in the language: remote objects are declared "remote"
- marshalling is simplified
  - by passing by value only (object references can be used in nested RMIs)
  - (in Java) by having JVMs hide platform dependencies in data representation
- serialization could be much heavier by having to pass the code for the objects with every call, but that can be avoided by passing URLs for downloading the code, rather than the code itself

RMI example
calculate PI

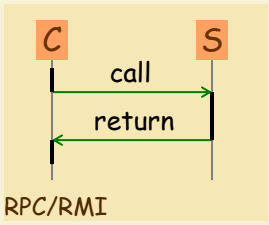
- suppose you have a computationally intensive task that you want to
  - define it in the client
  - invoke it in the client
  - execute it on a powerful server…
- look at the example code at http://download.oracle.com/javase/tutorial/rmi/overview.html
- if you're not familiar with generic types in Java, you may refer to http://download.oracle.com/javase/tutorial/java/generics/index.html

RMI example: code view

RMI example: run-time view

in summary
select a call-return style when:

- task is dominated by single thread of control
- caller knows and cares about the identity of server
- low volume of data is transferred



RPC/RMI

in distributed systems:

- it is fine to block the caller waiting for a reply
- the server is ready to process each request
- components and network are mostly reliable