CS533 - Concepts of Operating Systems, Winter 2012

The Structuring of Systems Using Upcalls

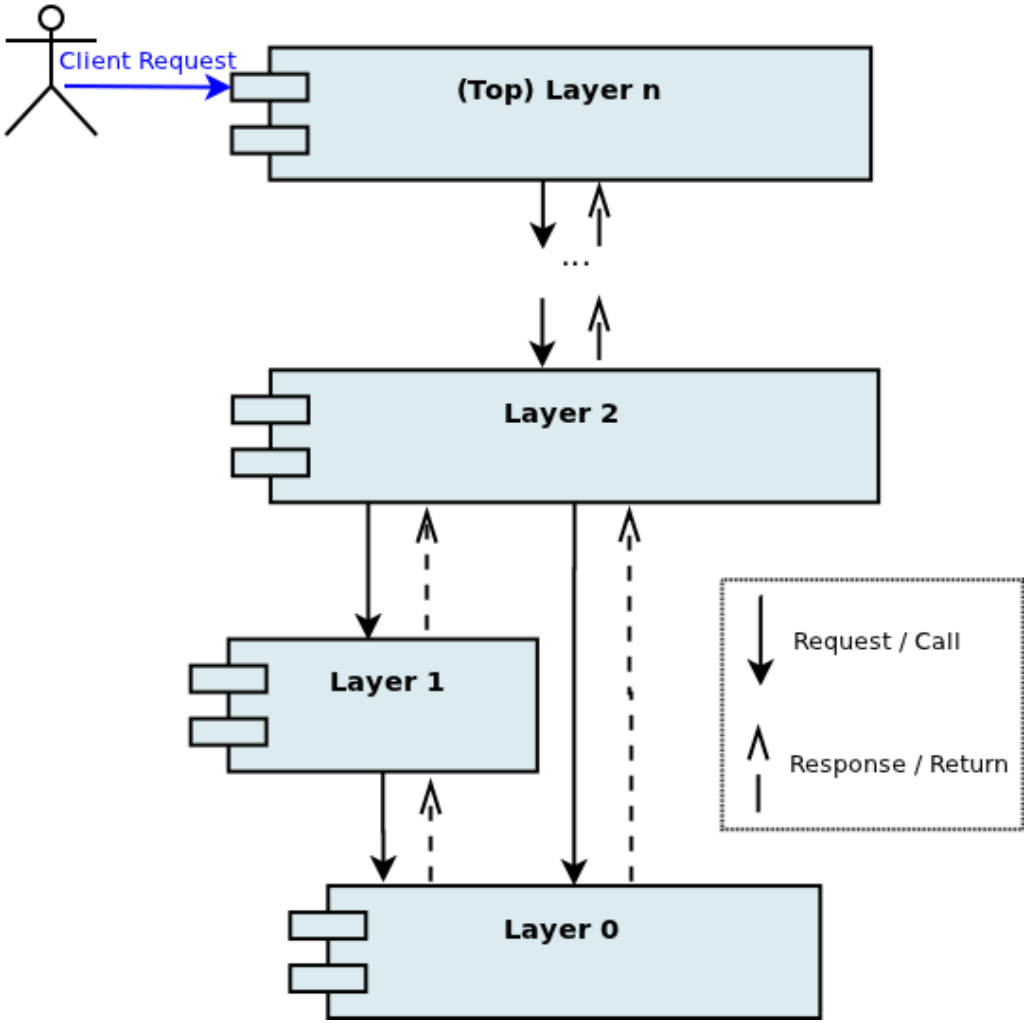
David D. Clark

Presented by: **Peter Banda**

Agenda

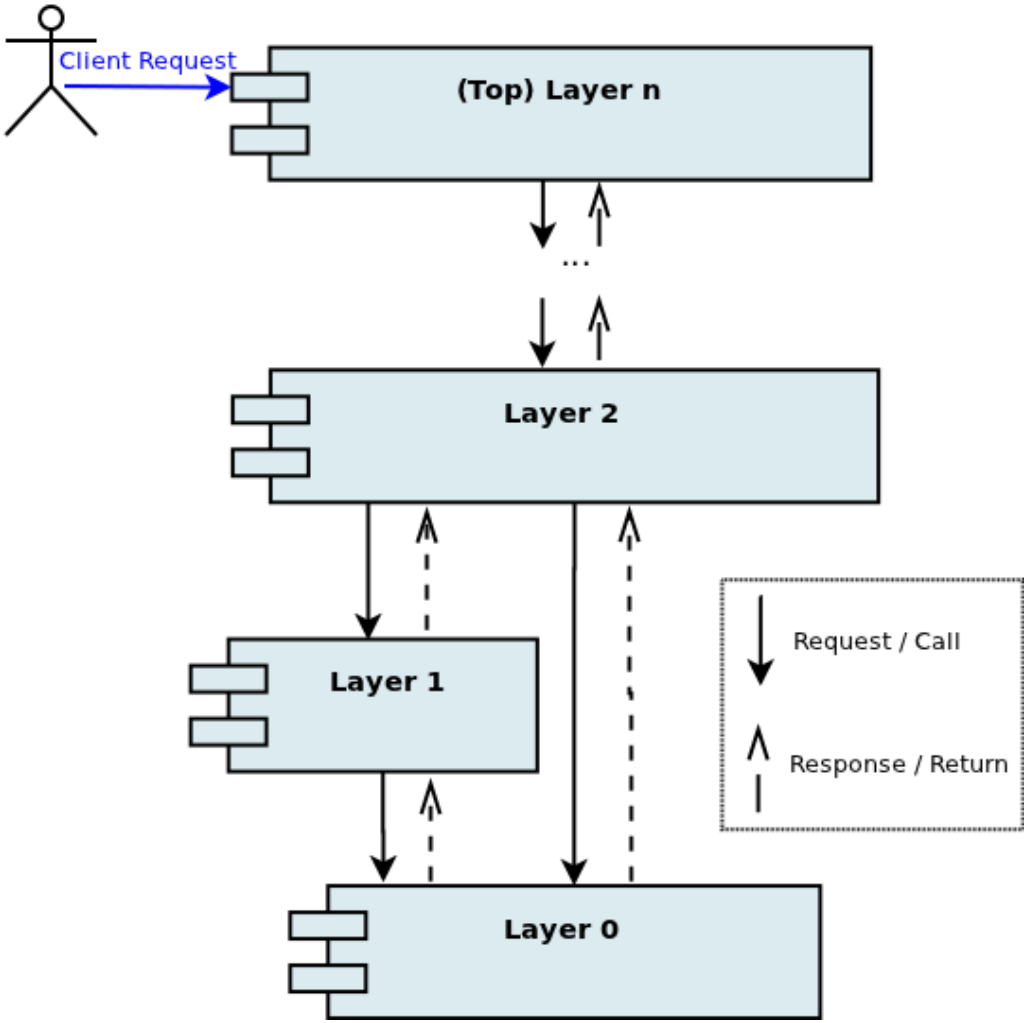
- Layers and Upcalls
- Example
- Multi-task Modules
- Problems with Upcalls
- Swift OS
- Conclusion and Discussion

Layered Architecture



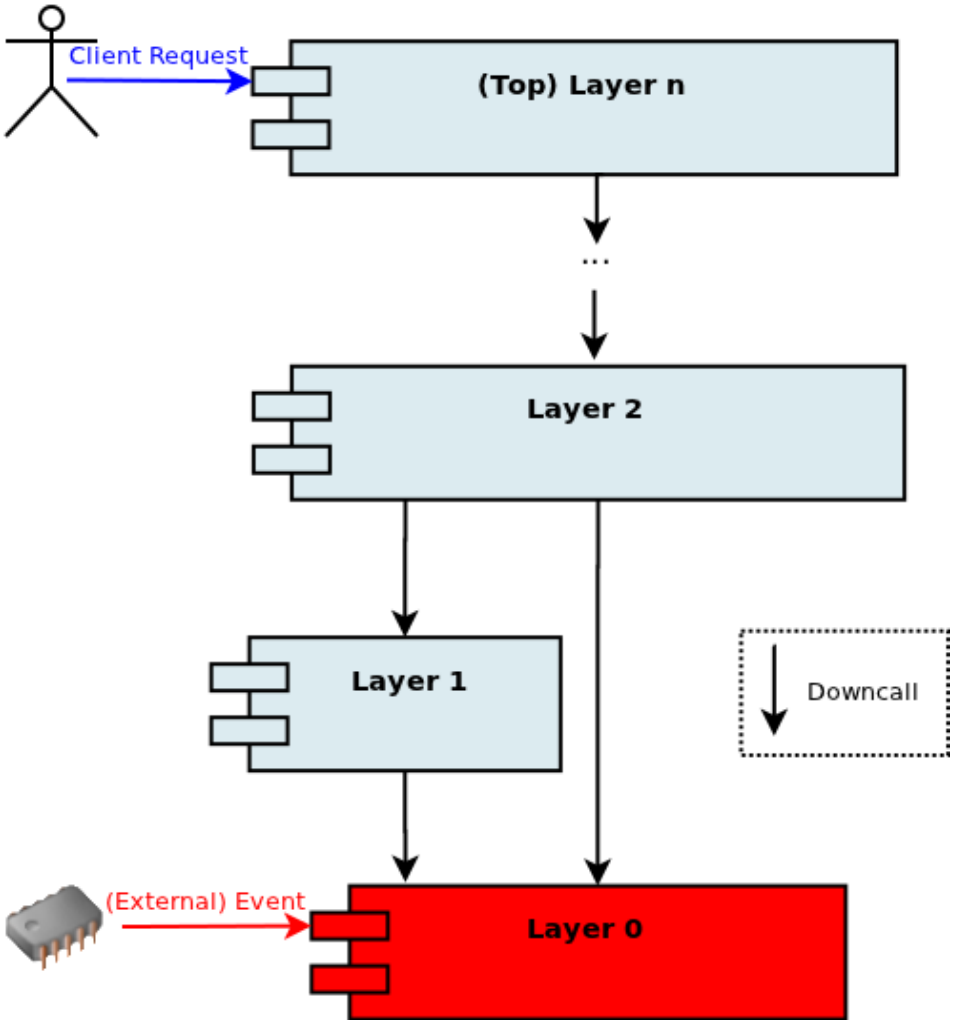
- Layer
  - a module providing services to the layer above (client layer)
  - abstract view of the functions
  - ordered according to "using", or "depending" relation in acyclic fashion
- Calls
  - a client invokes the lower layer (a subroutine call, RPC)
  - always top down: *downcall*
  - initiated by top client request

Layered Architecture (Cont.)



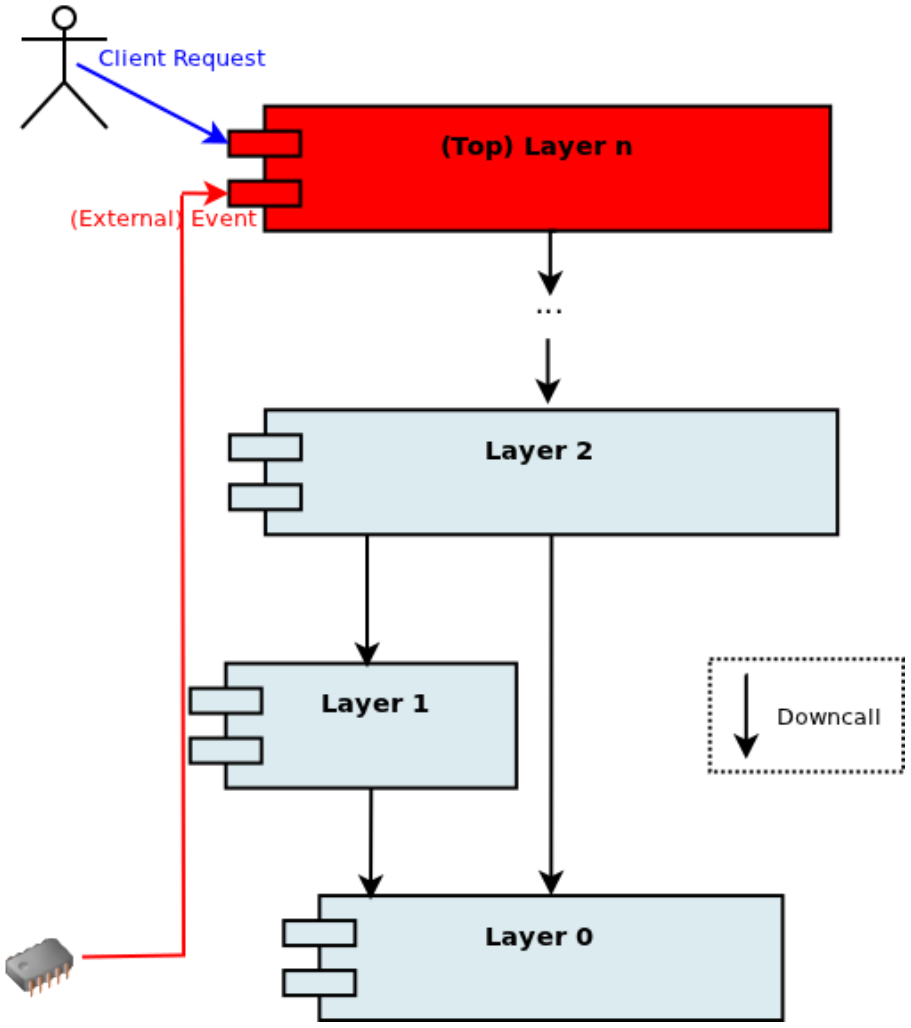
- Benefits
  - simplifies a verification, and failure localization
  - encapsulation; easier maintainability of code
  - strict organization, modularity
- Example
  - seven layer reference model of ISO
- Sometimes too strict...

Layered Architecture With Events



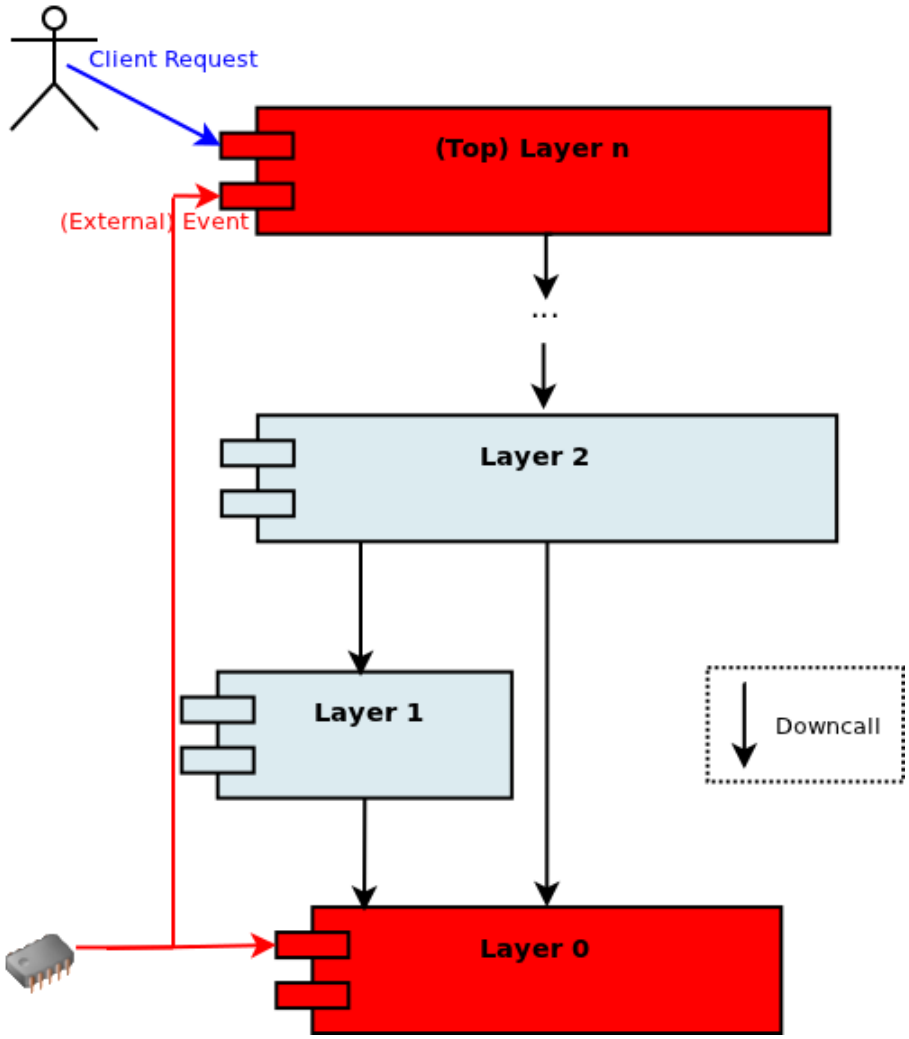
- Complication
  - client request is not only source of activity
  - Layer 0 handles (wraps) the external resource that produces events
- Layer 0
  - either takes a (silent) action based on its own info
  - or it's beyond its competency / cannot decide alone
- Who should handle event?

Layered Architecture With Events (Cont.)



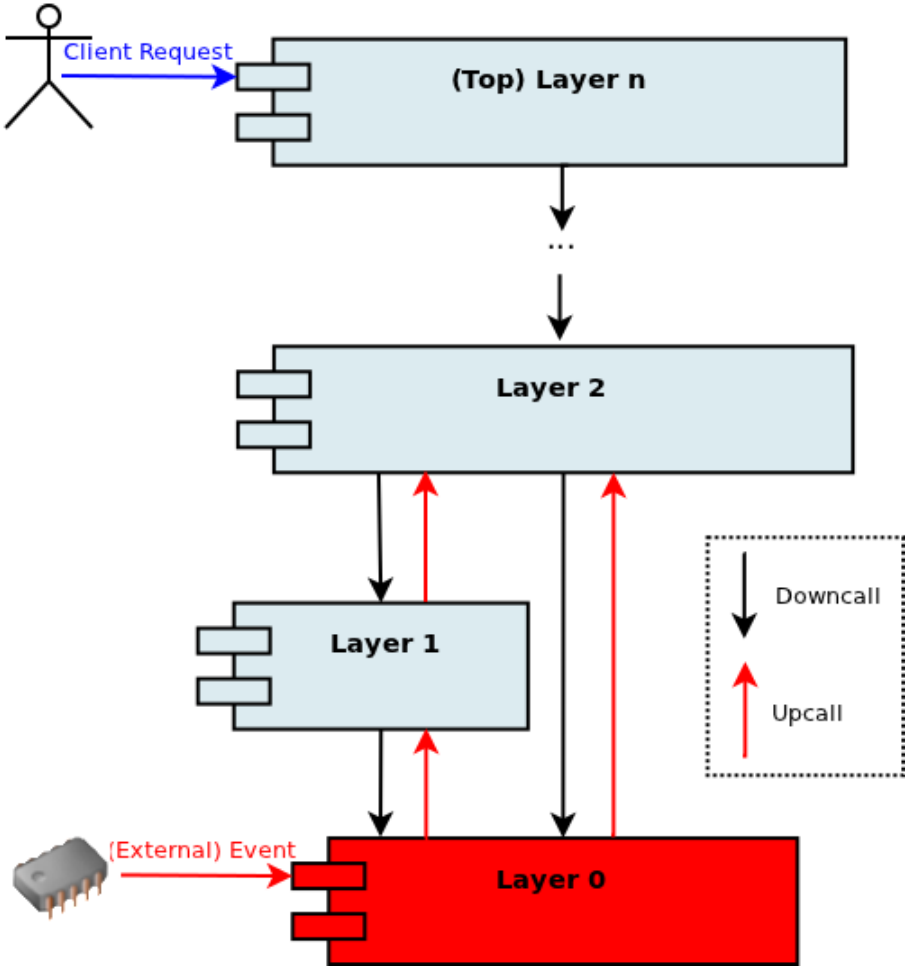
- Top layer handles the event
  - downcall structure is fully maintained
  - top layer cannot directly access a resource (why?); needs to register as event handler down at the Layer 0
  - pollutes top layer with raw info / exposes resources that should be hidden
  - in most cases top layer just delegates a call (requests sink down)
- It's only example...
  - e.g. Layer 2 can be event handler instead if it's sufficient

Layered Architecture With Events (Cont.)



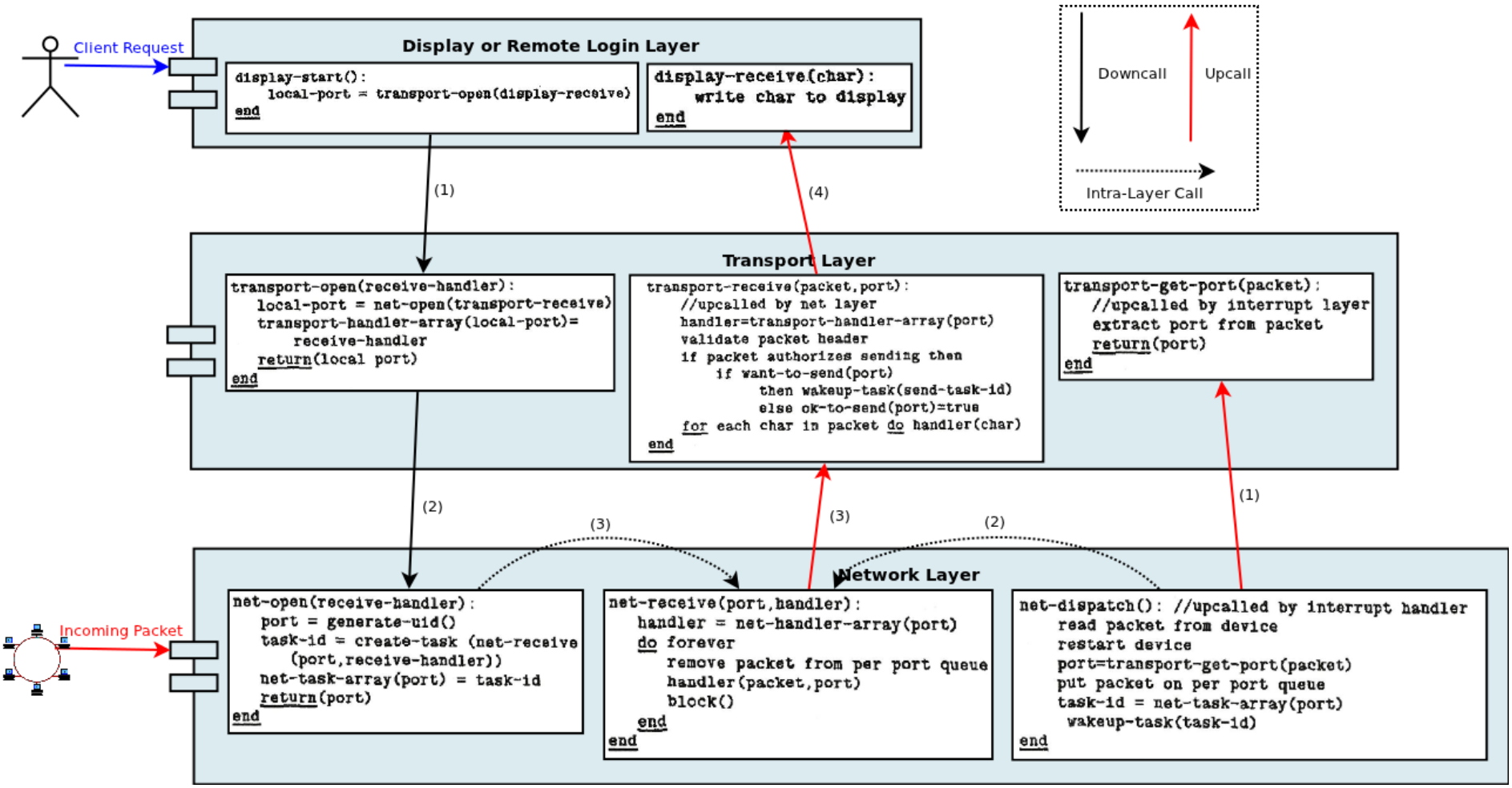
- Events are handled by the Top layer and Layer 0
  - careful not to process an event twice (discard events)
  - still, top layer has to understand raw data...
- Fix
  - Layer 0 handles external event, interprets it, and triggers internal event for Top layer

Layered Architecture With Upcalls



- Layer 0 should handle events produced by its resource!
- Procedure flow map on the natural flow of control in the program whether that is up, down, (or sideways?)
- Upcall
  - a lower level calls a higher level
  - still follows layering; 2-layer abstraction
  - notification to client layer; expects action from client
  - cannot be done without the client entry point → the client has to do downcall first (init.)

Example—Create and Receive Task



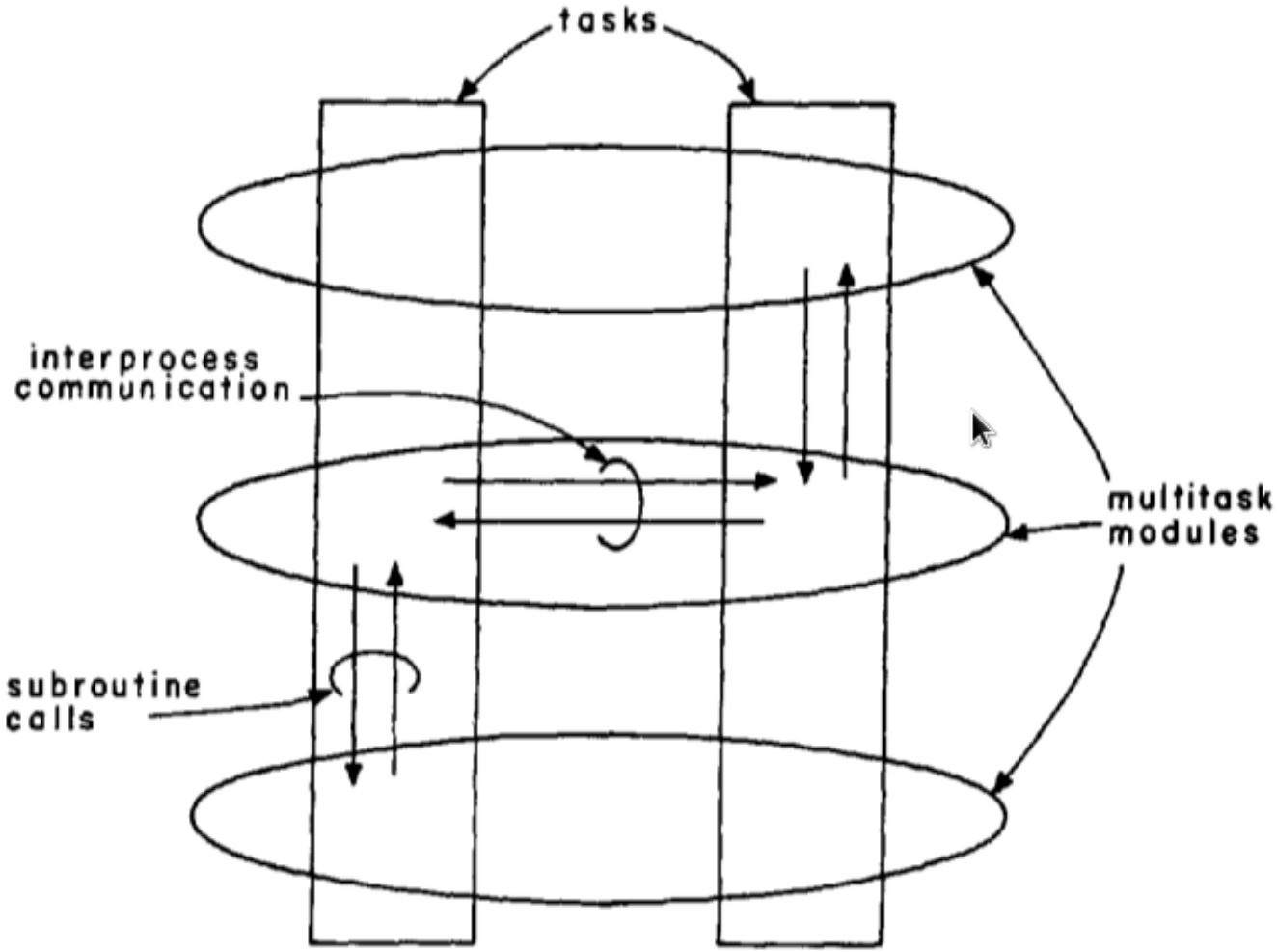
Special Upcalls and Piggybacking

- *Ask advice* call
  - feed subroutines just with “mandatory” attributes
  - if subroutine finds out it needs more information from client layer it asks for advice
  - upcall followed by downcall
  - results in a service with reasonable generality for a variety of clients
- *Arming* call
  - just notifies lower layer that an action should be taken (no serious processing)
  - always returns immediately (never blocking)
  - the resulting upcall executes whenever the flow control would permit
- *Piggybacking*
  - information from various layers are combined into a single outgoing packet efficiently
  - occurs naturally when using upcalls

Multi-task Modules

- Multi-task module
  - a layer is organized as the group of collaborating subroutines from different tasks
  - flow of control between layers is achieved through subroutine calls (synchronous), both up and down
  - subroutine interfaces are exported / imported
- Individual task
  - realizes a single thread of activity, usually on behalf of a single client or external event
  - move up and down between the layers as the natural flow of control indicates
  - intertask communication only occurs within the same module, which contains a collection of state variables, accessible using shared memory managed by monitor lock
  - intertask communication interface is private (no import / export)

Multi-task Modules



Advantages of (Synchronous) Subroutine Calls

- Asynchronous IPC
  - a layer implemented as a task or process (with a dispatcher)
  - communicating with lower layers by an interprocess signal (asynchronous messaging)
  - needs a buffering mechanism
- Synchronous Subroutine Call
  - more efficient / cheaper; the authors achieved a factor of five to ten speedup
  - no buffering
  - possible because of a single address space
  - subroutine interfaces are easier than interprocess communication interfaces
  - does not need to handle the format or usage of an message

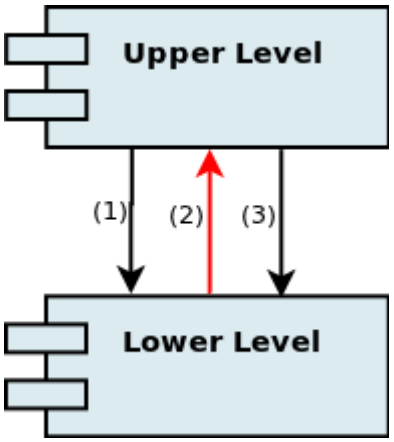
System Design

- Steps
  - form layers of abstraction (multi-task modules)
  - determine various events which trigger actions within the system
  - design the flow of control for each of these actions
- Decision about how the tasks are used need not be made until late in the design

Common Problems With Upcalls

- Propagation of failure
  - if a lower layer upcalls a upper layer, but the upper layer fails, the lower layer may be left in an inconsistent state, which may destroy not only the lower layer, but every other client of that layer
  - make sure that shared data used by (upcalled clients) is consistent and unlocked before any upcall is made
- Failed upcall handling
  - the task executing the failed upcall must be recovered or terminated
  - must use separate tasks for each of upcalling clients
  - if state beyond recovery, terminates just one vertical strip (task)
  - the system upcalls the layer-specific cleanup procedures with client id

Indirect Recursive Call



- Problem
  - when control returns back from the upcall, lower layer may find that its state has changed
- Solutions
  - all variables of the lower layer must be in consistent state, and then re-evaluated on return
  - recursive downcalls are prohibited
  - queue downcall work requests for later execution by the task holding the lock
  - restrict downcalls to do nothing but set flags that are checked at known times by other tasks, including the task making the upcalls
  - for upcalls triggering the same downcall, replace the downcall by extra return arguments, or another upcall to query the client

Swift Operating System

- OS embracing the ideas of upcalls and multi-task modules
- Scheduling
  - dynamic-priority task scheduling (priority=deadline)
  - to prevent priority inversion, a standard priority adjustment (deadline promotion) is used
- Address Space Management
  - Single shared address space → efficient passing of data from one task to another
  - Monitor locks guard access to shared memory of multiple tasks (in one layer)
  - Strong checking at compile and run time to insure that the address space is not corrupted (by high level language *CLU*)
  - Garbage collection used to prevent “dangling pointer”

Conclusion & Discussion

- Pure downcall flow is not sufficient
- Upcalls and multi-task modules provide simpler mapping of control flow
- New set of problems emerge
  - propagation of failure, task termination, indirect recursive call
- Paper overlaps two “orthogonal” aspects of a system specification: structure and communication mechanism
  - e.g., we can have a system with upcalls using asynchronous IPC (messaging), or
  - system with downcalls only, but using synchronous procedure calls
- Overuse of upcalls is dangerous
  - binds between client and lower layers are too strong → marriage relationship
  - in extreme case we cannot separate two modules anymore

Conclusion & Discussion

- Ask Advice issue
  - lower layer upcalls client layer just to get advice for an execution initiated by client's downcall
  - adhoc requests for extra parameters introduce more interlayer dependencies for sake of a simplification of subroutine interface
- One address space (one protection domain)
  - faster, but vulnerable to corruption
  - modules must have boundaries!
  - better to use large-grained protection with, e.g. URPC, or LRPC, for cross-domain calls

Thank you for your attention