



# Scalable, Resilient System Software

Barney Maccabe  
Interim CIO, Professor  
University of New Mexico

DOE ASCR PI Meeting  
Denver, CO



March 31, 2008

# NCAA Western Regional

Notre Dame 7  
New Hampshire 3

Notre Dame 3  
Michigan State 1

Michigan State 3  
Colorado College 1



Michigan v Notre Dame  
North Dakota v Miami

April 10 & 12  
Denver, CO



Colorado Springs, CO  
March 28 & 29



# Scalable Systems Software

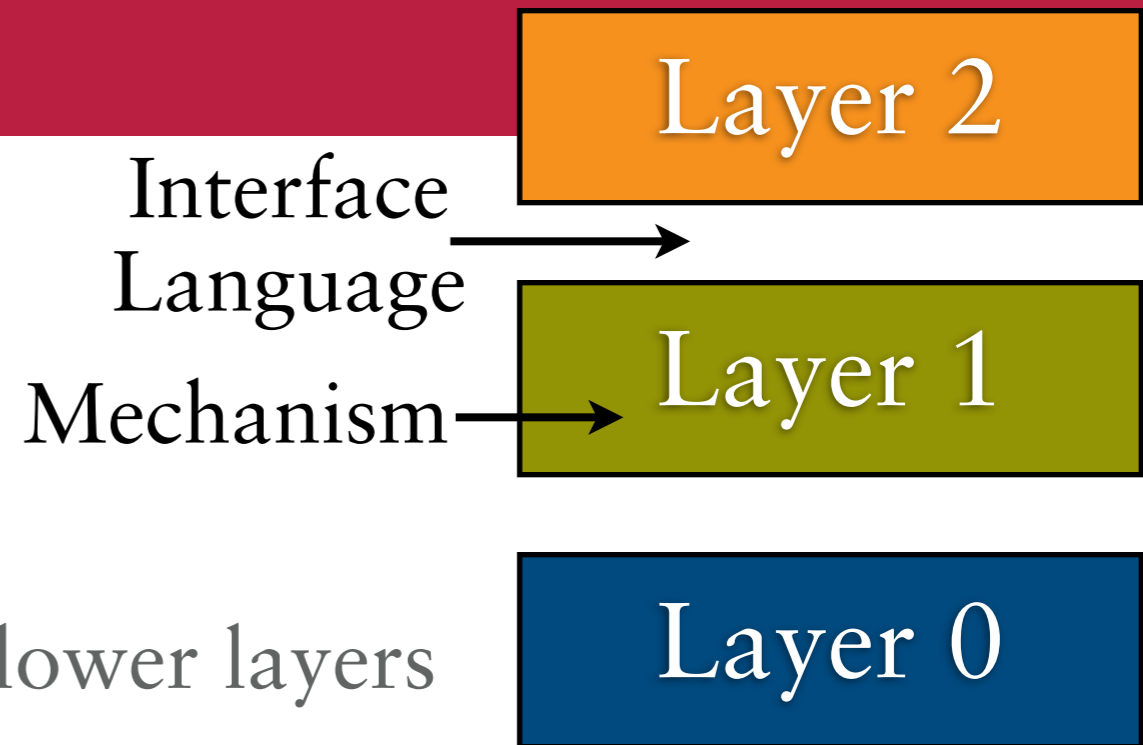
- Figure out which problems you won't solve.
  - doing nothing is better than doing the wrong thing
- Define languages, provide mechanisms and tools
- Don't try to provide solutions

“Perfection is reached, not when there is no longer anything to add, but when there is no longer anything to take away.”

A. Saint-Exupery

# Layers

- Layers define languages
  - Abstraction
    - Hide unimportant details of lower layers
  - Reflect costs
    - Programming is resource management
- Mechanisms implement the language for a layer
- Tools are orthogonal to layers, e.g., a transaction service



# Open Architecture

- User may choose to
  - accept,
  - reject, or
  - replace
- any service provided provided by a layer
- e.g., selectively replacing the translation service provided by a compiler

# Layer Examples

- Virtual Memory
  - Seymour Cray's quote
  - Bad page replacement decisions are very expensive
  - It's probably cheaper to buy more memory
  - Explicit, "out of core," management can work well for many applications
- Virtual Memory is magic

# Layer Examples

- Synchronized communication
  - Ada rendezvous
  - CSP (Communicating Sequential Processes)
    - MPI synchronized communication
  - Self synchronizing applications don't need synchronized communication
  - Flow control
    - many to one—better to fail sooner

# Layer Examples

- Consistent & coherent filesystem metadata
  - metadata updates in the critical path
- Checkpoint
  - Initiate a lightweight transaction on node 0, broadcast transaction id to all nodes
  - Each node creates unique data object & dumps data
    - no (global) metadata, no consistency, no coherency
  - All nodes send their data object id to node 0
  - Node zero builds an “index object” and commits the transaction (could be done “off line” by a meta-bot)

# General Principles

- Simplicity (Butler Lampson)
  - Make it fast rather than general or powerful
  - Don't hide power
  - Leave it to the client
- Transparency (Nicolas Wirth)
  - Programming is resource management
  - Features should reflect uniform costs (uniform layer of abstraction)

# More Principles

- Make layers thin
  - The less you do, the less you can do wrong!
  - Easy to add new capabilities
    - virtual node mode in Puma
    - direct mapping across multiple cores
- Focus on enabling good things
  - don't start by preventing bad things
  - it's easier to prevent bad things in higher layers

# Dealing with Flakiness

- Graceful degradation
  - Use everything when things are good,
  - Degraded performance when components unavailable
- Hold in spare replacement
  - Keep spare parts available
  - Migrate to spares when components unavailable
- Run-through failure
- Timeliness

# Graceful Degradation

“Graceful degradation means that your Web site continues to operate even when viewed with less-than-optimal software...”

# Words

- Fault tolerant (durability, as in *duration*)
- Fault resilient (availability)
- Fault oblivious—don't interfere with these applications
- Fault masking (reliability)
  - avoidance, prevention, and recovery
  - property of a layer, not the system/application
  - make the right tradeoff

# More words

- Old DEC terms
  - exception—inappropriate use of a resource (divide by zero); correct
  - fault—temporary unavailable resource (page fault); provide resource and reissue instruction
  - interrupt—external request for service (disk request completed); resume instruction
  - trap—request for service (OS trap); provide service and resume after the issuing instruction

# Context

- DEC Terms define the context
  - What could go wrong (what doesn't go wrong)
  - Casual relations (synchronous/asynchronous)
  - How to respond
- What is the context for component failures?
  - What can fail?
  - What can you rely on when a component fails?
  - How can you respond to a component failure?

# Example

- Vehicle tracking with flaky sensors
  - an array of “smart” sensors
  - periodic centroid calculation among detecting sensors
- How does a user express the computation?
  - temporal framework: all activated sensors at time  $T$ 
    - synchronized clocks
  - (un)reliable communication
  - Is it good enough?

# This is a conversation

- Don't look for solutions
  - languages,
  - mechanisms, and
  - tools
- We need to understand the vocabulary first
  - what is observable?
  - what is knowable?
  - what are the options for response?

# Thanks

“The unavoidable price of reliability is simplicity.”  
C.A.R Hoare

“Fools ignore complexity;  
pragmatists suffer it;  
experts avoid it;  
geniuses remove it.”  
Alan J. Perlis