

Building Reliable High-Perf Comm Systems from Components

Xiaoming Liu, Christoph Kreitz,
Robbert van Renesse, Jason Hickey,
Mark Hayden, Kenneth Birman, Robert
Constable, Cornell

Component based systems pros & cons

- Pros:
 - Modularity, extensibility, maintenance, ...
- Cons:
 - Overhead, lack of locality, redundant code, (in)ability to optimize across components, configuration management, verification of compositions...
- Objectives in paper:
 - Component based systems can be “efficient” - deliver performance, optimization techniques can be applied across components, and configuration can be “hardened” – automated and verified.

Paper overview

- Originally Horus component-based communication architecture, C
- Components – micro-protocols
 - micro-protocols implement single property or functionality
 - can select set of (stack) of micro-protocols to impl comm service
 - should be certain that composition is correct
 - should also be able to optimize performance for common case
- Ensemble in OCaml
 - functional languages more suitable for formal analysis and verification
 - turned out it performed well too
- I/O Automata (IOA) – spec of high-level micro-protocol components
- Nuprl – automating the reasoning process

Specification

- From abstract behavioral specification to concrete one, to implementation...

```
Specification FifoNetwork()
Variables
  in-transit: queue of <Address, Message>
Actions
  Send(dst : Address; msg : Message)
  condition: true
  { in-transit.append(<dst, msg>); }

  Deliver(dst : Address; msg : Message)
  condition: in-transit.head() == <dst, msg>
  { in-transit.dequeue(); }
```

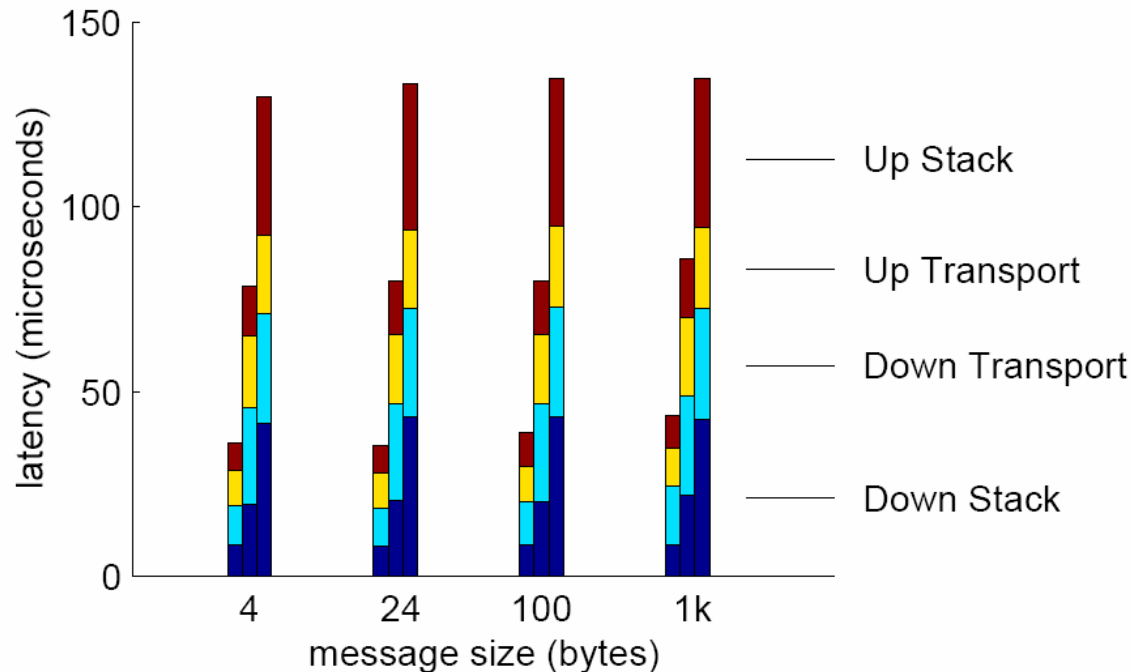
```
Specification FifoProtocol(p : Address)
Variables
  send-window, rcv-window, ...
Actions
  Above.Send(dst : Address; msg : Message)
  Below.Send(dst : Address; <hdr, msg> : <Header, Message>)
  Below.Deliver(dst : Address; <hdr, msg> : <Header, Message>)
  Above.Deliver(dst : Address; msg : Message)
  Timer()
```

Optimization

- Goals
 - Identify common case and simplify, localize, skip unnecessary conditionals or code blocks
 - Important to extract *Common Case Predicate*
 - Automate or programmer interaction
- Static – a priori, per micro-block
 - Create *bypass* code (and a demux to eval CCP)
 - To extract CCP, automate check for typical cases, plus let programmer add some
- Dynamic – at protocol stack composition time
 - Compose bypass code for entire/set of stack
 - Composition theorems
- At runtime evaluate CCP
- Other common techniques: scatter gather, header compression, avoid garbage collection, what should/n't be on critical path...

Does this work?

- Compare Imperative, Functional, Hand-optimized, Machine-optimized



- Hand best, but hard to do for larger stacks; IMP better than FUNC, but hard to apply reasoning tools
- Optimizations -> better perf, few TLB misses, smaller code size