Low-level Programming Model for NOW: Active Message Perspective

ASPLOS NOW Workshop
Oct 4, 1994

David E. Culler
Computer Science Division
University of California, Berkeley
culler@cs.berkeley.edu

Outline

- Background
- Two "telling" questions
  - How do you build an N → 1 receive queue?
  - How do you receive while you try to send?

What's this panel about?

Low-level Programming Model for NOW

Assembly Language (support for) **Communication***

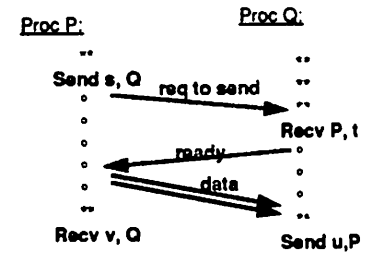
* on fast, reasonable, but not perfect networks with little or no changes to existing workstations

Requirements

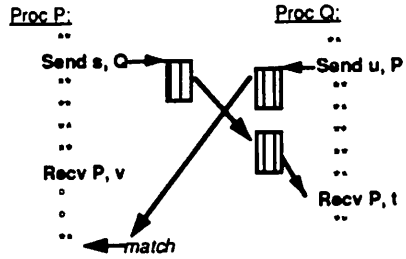
- General purpose
  - support all popular programming models
    - » shared memory
    - » message passing
    - » data parallel
    - » concurrent objects
    - » sockets, pipes and files
- Efficient
  - operate very close to the network

Protocols realize programming models

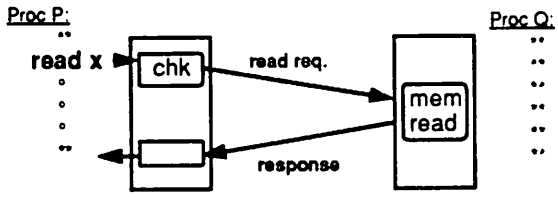
Sync. Message Passing



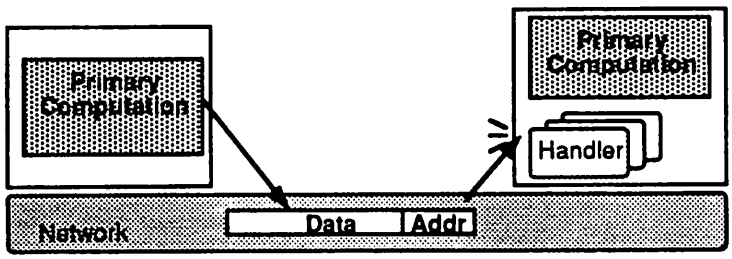
Async. Message Passing



Dist. Shared Memory



Active Messages are Primitives for Protocols

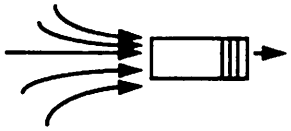


- Sender injects the message directly into network
  - handle while trying to send
- Small small user-level handler with each message
  - pull message out of the network and integrate into the ongoing computation, or reply (as determined by high level programming model)
- No buffering, parsing, or allocation (beyond transport).

Where's the beef?

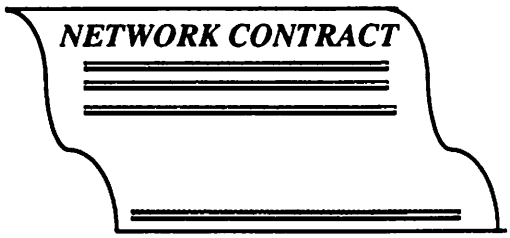
- Exposed as user-level protected primitives what was hidden in everybody's message layer, kernel, or hardware implementation of a high level programming model
- Demonstrated performance on real machines
  - 3 µs overhead + 3 µs latency on CM-5
  - 4 µs overhead + 12 µs latency on Paragon
  - 8 µs overhead + 8 µs latency on HP735 Medusa (FDDI)
    - » even with blechy timeout & retry
- Provided a clean framework for architectural advance
  - see Flash/Magic, Typhoon/Tempest, . . .
    - » specialized handler co-processor
    - » optimized send

Point 1: Receive queue is key



- Active Messages abstract the queue to operate directly out of hardware structures
  - queue remains for inactive processes
- Reflectiv memory => scan N input areas for request
- Shared Memory => lock, modify ptr, store, unlock
  - magic is powerful enough to build good primitives, but user is not allowed to do so.
  - tempest allows you to, but would like you to express these in terms of coherency protocols

Operating Close to the Network

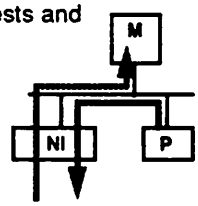


- Network is deadlock free (reasonably robust) only if always drained
- Sends stall due to contention

=> **Must receive while trying to send**

Point 2: Send Sequence

- Active Message
  - stores + load (status) => costs cycles, not hardware
- Distributed Shared Memory
  - load => instruction stalled after MMU check, but NI must be able to get into memory to receive (handle requests and responses)
    - » soldering iron in DASH
    - » ECC error in WWT
    - » dual ports to Magic
    - » take over bus arbitration in Tempest



  - microprocessors to date don't allow memory system to renege on valid load
- Shrimp
  - only stores => NI issues interrupt to processor when things start to look bad (???)

Our initial steps

- Calibrate and learn all we can out of existing hardware via generic active message layer
  - HP with stateful NI on graphics bus (FDDI)
    - simple, fast buffer and retry
  - ATM
  - Paragon, Meiko
  - Myrinet
- Construction of a variety of higher level programming models
  - Compiler-based shared memory
  - Message passing
  - Sockets
- = = > NI design, OS mods, Net req., (bus spec) as a package