Advanced Topics in Operating Systems, CS262

Prof. Eric A. Brewer

Lecture Notes, 10/9/98

Active Messages
U-Net

I. Active Messages

Basic model of computation:

```
LOOP
    compute;
    communicate;
END
```

Observe the following:

- If compute and communicate phases don't overlap then can't get high processor utilization unless much more computation than communication is occurring. => Courser-grained sharing model required.
- Overlapping compute and communication phases imply that high processor utilization is achieved as long as processor and network costs/utilization balance. => Want asynchronous communications model.
- The shorter the communication phase is the finer-grained a sharing model is possible. => Want low latency communications.

Approach: Treat message sending as the critical path and get everything off the critical path that you can.

Two primary sources of slow-down:

- generalized buffering & resource allocation,
- allowing for blocking/delayed server activity.

Active message solution:

- No buffering (beyond that needed for data transport).
- Short user-level receive handlers that may not block: either generate a reply message right away or extract the received message from the network somewhere into user space and return.
- Head of a message packet contains the address of the receive handler to run.

Some ways to think about active messages:

- Interrupt-level RPC (when response to a message is generated immediately).
- "Link layer" communication facility: gets bits from A to B and does nothing else; everything else is the responsibility of the application.
- Exports a "raw hardware" model: asynchronous hand-off to network on sender side; interrupt handler on receiver side.

Potentially an order-of-magnitude faster than more generalized communication facilities.

Systems, such as Split-C, that employ simple communications abstractions like Put & Get, benefit from that speed-up.

Message-driven machine designs also benefit from the minimalist approach:

- Very fine-grained data model in which computation is driven by messages that contain a function designator and data.
- Frequently a message does not contain all the data needed to invoke a computation. => Have to block awaiting the arrival of the rest of the data.
- Active message approach implies that "simple" messages get processed quickly and resource allocation for and execution of multi-message functions gets handled in an application-specific manner; which allows for application-specific optimizations and batching.

3 key features about the paper:

- Try to improve utilization of massively parallel machines by focusing on the interaction between overlapping computation and communication.
- Provide an extremely lean communication facility, called active messages, that tries to remove as much processing as possible from the basic communications operation of getting a message from node A to node B.
- 2 primary sources of slow-down removed: generalized buffering for messages and support for blocking/delayed receiver activity.

Some flaws:

- The paper discusses lots of details that seem only semi-relevant to the Instructor's interpretation of what this paper is about. Maybe the paper is confused; maybe the Instructor is confused.
- The active message design pushes all the hard buffering and scheduling decisions into the application and observes that what's left over is simple and fast. For what fraction of various workloads will this end up merely being an "accounting trick"?

A lesson: Say it again, Sam: Optimize the critical path.

II. U-Net User-Level Network Interface

New issues to active messages:

- vitualize the network interface: each process has its own virtual NI
  - some direct, some emulated by the kernel
  - set up is always through the kernel
- protection between processes for network state
  - use VM hardware to enforce this
- authentication of messages (again for protection)
  - reliably tag the sending endpoint
  - reliably dispatch to the correct receiving endpoint
- conventional OS and hardware
- managing resources without the kernel in the (common!) path
- minimize copies
  - “zero” copy is really one copy from NI to correct place in the receiving process (“base” level U-Net)
  - true zero copy, NI puts data directly into process address space (like CM-5)