

CS 518

Active Messages: Minimalist Communication

Randy Wang
Fall 2002
Princeton University

Outline

- Introduction
- Active Messages
- Common issues in RPC and AM

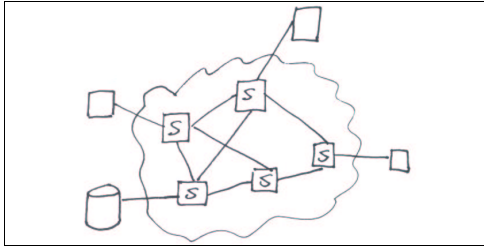
Outline

- Introduction
- Active Messages
- Common issues in RPC and AM

TCP, RPC, and AM

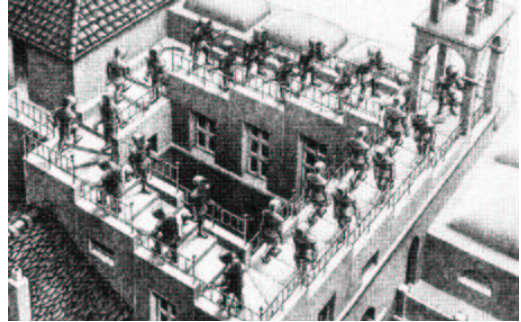
- Specialization
 - TCP to RPC: particular style of communication in RPC allows it to use a more efficient transport implementation
 - RPC to AM: direct application control of the network makes communication even more efficient
- Considerations
 - Ease of programming and semantics
 - Performance
 - Tension between the two

Hardware Trends



- Tremendous improvement of network performance
- Convergence of networking technology for SAN, LAN, and WAN
- Tremendous improvement of processors, but they are very bad at certain things:
 - Data copying
 - Context switches
- Emergence of “smart” devices: NIs, smart routers, SMPs, smart disks, ...
- May avoid unnecessary data movement

Wheel of Reincarnation



- Where to be on the wheel?
 - generality vs. efficiency, software vs. hardware, ...
- Current trend: moving applications closer to devices (VIA, extensible routers, ...)
- Re-thinking the communication interface
- Mutual influence between the structure of distributed systems and the style of communication

Differences of Traditional Parallel Programs and Distributed Systems

- Degree of cooperation and sharing
- Granularity and frequency of communication
- Amount of overlapping communication/communication/computation/IO
- Degree of trust
- Existence and adherence to standard interfaces
- Tolerance of heterogeneity (of both h/w and s/w)
- Fault isolation and tolerance
- Freedom of distributing data and computation
- The difference is narrowing
- What is the nature of the relationship between the pieces that run inside devices and various hosts?

Outline

- **Introduction**
- **Active Messages**
- Common issues in RPC and AM

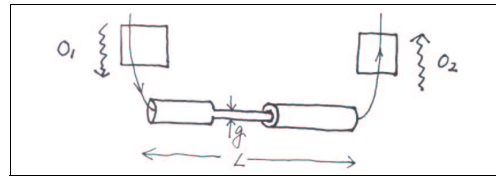
Initial Motivation for AM

```

for () {
  communicate();
  compute();
}
get(); /*initial items*/
for () {
  sync(); /*wait for previous item
  get(); /*next items */
  compute();
}
    
```

- Traditional:
 - If compute and communicate don't overlap,
 - Need a lot of compute to dwarf communicate
 - Communication is expensive
 - Need coarse-grained sharing
- Want:
 - If compute and communicate can overlap
 - Only need to "balance" compute and communicate
 - Cheap communication
 - Can have fine-grained sharing
- Need:
 - asynchronous communication and low-overhead communication
 - one makes the other even harder

"LogP" Model

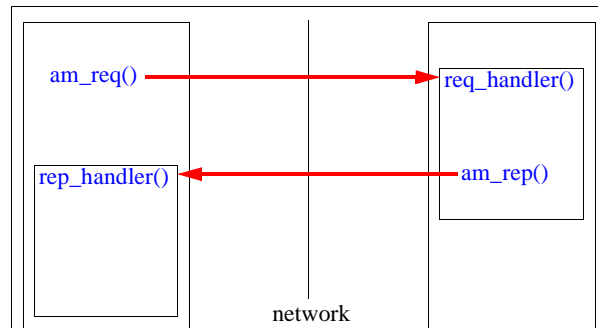


- Simple model for understanding small message performance
- LogP model:
 - o : cost of protocol processing on two end hosts
 - protocol design
 - host processor speed
 - L : time spent on the wire
 - speed of light
 - switching delays (if any)
 - g (gap): throughput of lots of small messages
 - bottleneck components, such as NI, n/w fabric (queues)
- Typical worse culprit: o

Reducing Overhead

- Two primary sources of slow-down
 - generalized buffering and resource allocation
 - allowing for blocking/delayed server activity
- Active Messages solution
 - No buffering (beyond that needed for data transport)
 - Short user-level receive handlers
 - can't block
 - either generate a reply message right away, or
 - extract the message from the network and put it into user space and return
 - not for generic computation
 - Head of message packet contains the address of the receive handler to run

Programming with Active Messages



- No buffering (beyond that needed for data transport)
- Short user-level receive handlers
 - can't block
 - either generate a reply message right away, or
 - extract the message from the network and put it into user space and return
 - not for generic computation
- Head of message packet contains the address of the receive handler to run

Example

```
get(); /*initial items*/
for () {
    sync(); /*wait for previous item */
    get(); /*next items */
    compute();
}
```

- Let's try to implement `get()` and `sync()` using AM
- This is so commonly needed that it generalizes to "Split-C"
 - AM becomes a communication assembly language
 - Compiler support to compile higher level communication models down to AM primitives

CS518

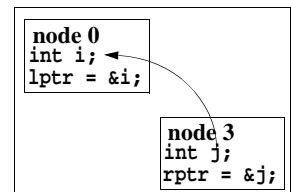
12

Randy Wang

Example (cont.)

```
main() {...iget(3,&i,&j); ... sync();}

int numGetOuts;
iget(int pid, int *lptr, int *rptr) {
    numGetOuts++;
    am_request_3(pid, iget_handler, lptr, rptr,
                &numGetOuts);
}
iget_handler(int pid, int *lptr, int *rptr,
             int *ctr) {
    am_reply_3(pid, iget_reply_handler, lptr,
              *rptr, ctr);
}
iget_reply_handler(int pid, int *lptr, int rval,
                  int *ctr) {
    *lptr=rval;
    (*ctr)--;
}
sync(void) {
    while (numGetOuts) {
        am_poll();
    }
}
```



CS518

13

Randy Wang

Some Questions

(dealt with later)

- When/how are active messages executed?
- What's the operating system's role?
- What if the process is not executing?
- What about big messages?

CS518

14

Randy Wang

Some Ways of Thinking about AM

- Interrupt-level RPC
- "Link layer" communication facility
 - gets bits from A to B
 - everything else is the responsibility of the application
- Exports the raw network hardware
 - asynchronous hand-off to network on sender
 - interrupt handler on receiver

CS518

15

Randy Wang

Summary

- Asynchrony
 - Overlap communication and computation
 - Better utilization of parallel machines
- Low overhead
 - No generic buffering and support for blocking/delayed server activity
 - Minimalism made possible by user control of communication

Outline

- Introduction
- Active Messages
- Common issues in RPC and AM

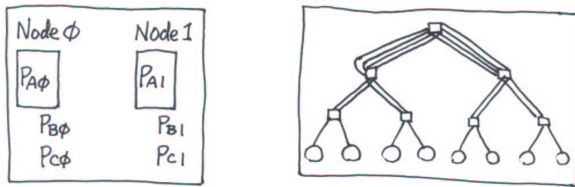
Language Support

- RPC
 - Meant for direct use for application programmers
 - Pretty close to Mesa procedure call semantics (even exceptions, but no pointers...)
 - Easy to use for client/server applications
 - Implementation: automatic stub generation by Mesa compiler
- AM
 - “Assembly language” for higher level communication layers
 - Possible (but not as easy) to be used directly by apps in an event driven fashion
 - Split-C:
 - example use of AM
 - implements asynchronous read and write with notification (and “global pointers”)

Binding and Location

- Definitions
 - Binding: which service requests map to which exported services
 - Location: on which machine are the services located
- RPC
 - Use the “Grapevine” name servers
 - Binding can be static or dynamic
 - Nice: exporters maintain no state
- AM
 - Original version simply used virtual addresses of handlers (for SPMD programs)
 - Newer version similar to modern RPC mechanisms
- Common:
 - Some trusted & authenticated name server
 - Need to distribute the name server and keep it highly available

Multiplexing and Security

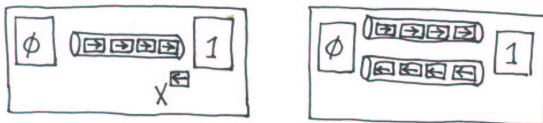


- RPC
 - In general, authentication and security orthogonal from the RPC mechanism
 - Authentication can be done at bind time
 - Rights checking at call time
- AM, initial version on CM5
 - Direct user access to network
 - No kernel enforcement
 - Co-scheduled SPMD
 - Protection by VM
 - Shoot-down and message reinjection for context switches

Multiplexing and Security (cont.)

- AM (U-net)
 - Virtualizes network interface (a special case of Exokernel philosophy)
 - Setup through kernel (slow path)
 - Fast checking during communication
 - no more kernel involvement
 - reliably tag the sending endpoint
 - reliably dispatch to the correct receiving endpoint
 - use VM hardware to enforce protection

Transport Issues for Small Calls



- RPC
 - synchronous: so minimum state per call
 - piggy-back acks on replies and new calls
- AM
 - Register-to-register transfer
 - asynchronous: deadlock avoidance can be tricky
 - requests fill network buffers, replies block, deadlock
 - Impose rules:
 - Handlers can't block
 - Reply handlers can't use network
 - Solutions
 - Separate request/reply networks
 - Result
 - Replies always drained, progress guaranteed

Polling vs. Interrupts

- How to signal control transfer in all the cases?
- Polling: busy waiting on flag or value
 - Per poll cost cheap
 - But burns cycles
 - Fragile performance: polling too often? not often enough? where to poll?
 - Potentially easier synchronization: non-preemptive scheduling
- Interrupt
 - Costly per interrupt
 - But multi-programming friendly
 - Robust behavior
 - Potentially harder synchronization: interrupt can happen any time
- What's the right amount of h/w support? Does it ever make sense to devote an SMP node to polling?