PAN: A High-Performance Active Network Node Supporting Multiple Mobile Code Systems

Erik Nygren, Stephen Garland, and M. Frans Kaashoek
MIT Laboratory for Computer Science

IEEE OpenArch 1999
New York, New York
March 27, 1999

`http://www.pdos.lcs.mit.edu/~nygren/pan/`

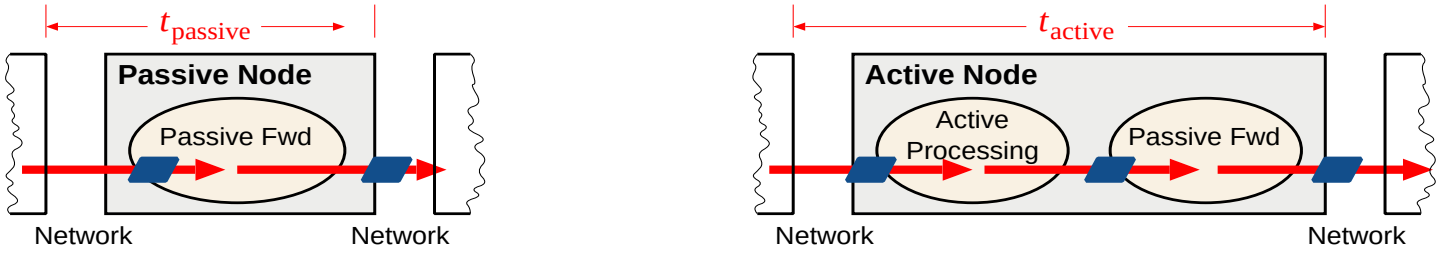
Active Networks

- Traditional *passive* networks forward packets based on packet headers
- *Active networks* process *capsules* containing both code and data
  - Code tells the node what to do with the capsule
  - Allow new network protocols to be dynamically deployed
  - Critical issues:
    performance, safety, security, resource management, interoperability
- *PAN* is a high-performance active network node
  - Inspired by Wetherall and Tennenhouse's *ANTS* system
  - Designed for performance and to allow experimentation
    with active network implementation issues

The Active Overhead

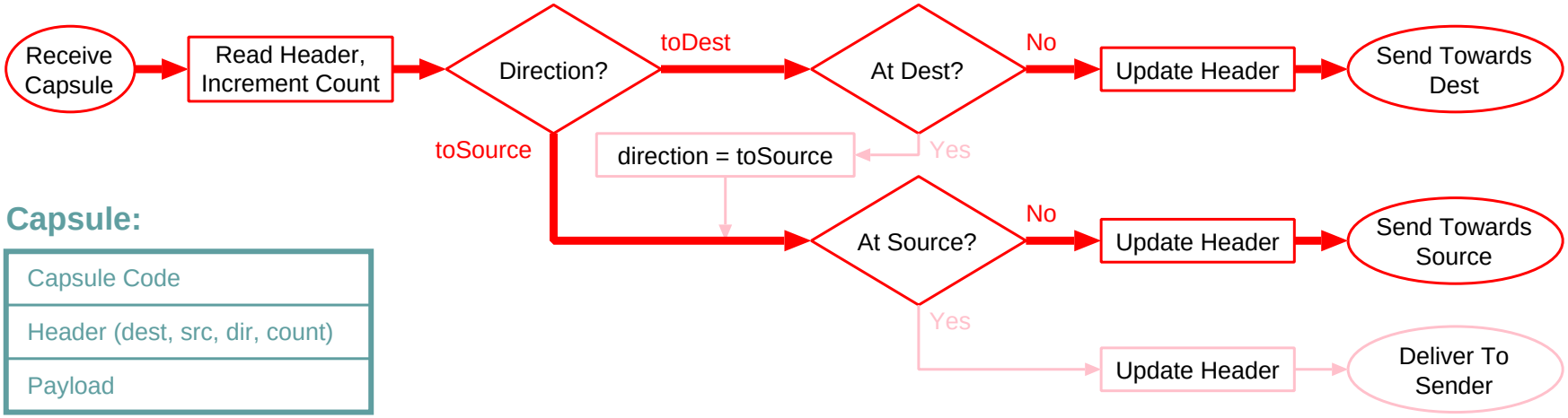
- Prototype active nodes written in Java and running in user-space tell little about potential performance
- Goal: demonstrate that an active node can obtain high performance with a low *active overhead*
- *Active overhead* is the percentage increase in processing time between passive and active forwarding:

$$\text{ActiveOverhead} = 100\% \cdot \left(\frac{t_{\text{active}} - t_{\text{passive}}}{t_{\text{passive}}}\right)$$



The Baseline Performance Hypothesis

- **Baseline Case:** a simple "ping" capsule which heads towards a destination then returns towards a source:



  - Functionality can be added incrementally to the baseline with an incremental performance cost

- **Performance Hypothesis:** In the baseline case, an active node can be made to process capsules with performance comparable to a traditional passive node while incurring only a small active overhead.

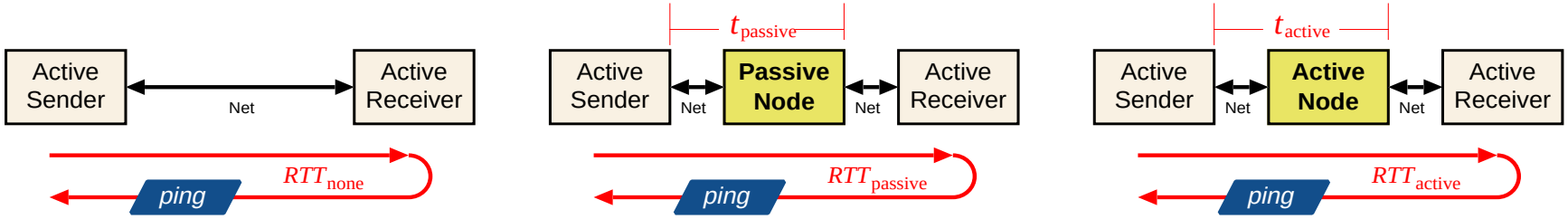
- Note: comparing against UNIX software router, not specialized hardware

Obtaining High Performance

- Approach:
  - Look at the active processing *critical path* in the baseline case
  - Eliminate potential sources of overhead
- Major sources of overhead:
  - Memory copies (bring capsule data into cache)
  - Code interpretation, loading, or translation in critical path
  - User / kernel boundary crossings
- These overheads can be eliminated through design choices
- Experiments:
  - Measure the cost of overhead sources
  - Show that overhead sources can be overcome
- **PAN points the way towards bridging the performance gap between a research prototype and a system with practical performance**

Measuring Performance

- Testbed: Three Intel PentiumPro 200's running Linux 2.0.32 with DEC Tulip-based 100 MBps Ethernet cards
- Network configurations:



$t_{\text{passive}} = \frac{1}{2} \cdot (RTT_{\text{passive}} - RTT_{\text{none}})$ For testbed, $t_{\text{passive}} = 157\mu s$

$t_{\text{active}} = \frac{1}{2} \cdot (RTT_{\text{active}} - RTT_{\text{none}})$

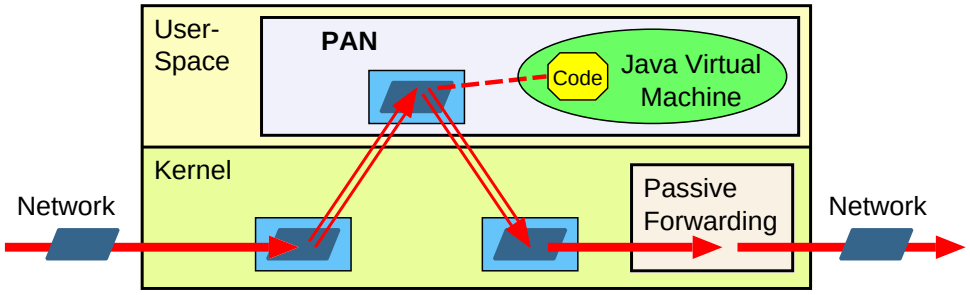
- Experiments:
  - Vary characteristics of *Active Node* and observe effect on ActiveOverhead
  - ActiveOverhead measured using 128 through 1500 byte packets

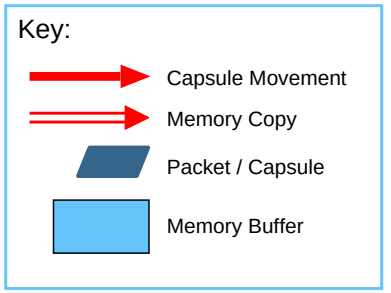
Bridging the Performance Gap

- ANTS: prototype designed without performance in mind
- PAN: can be configured with a wide range of performance characteristics

	ANTS	**PAN**
Written in...	Java	C
Mobile code	Java	Java *or* Intel native ix86 object code
Runs in...	user-space	user-space *or* kernel
Capsule data...	is copied	can be processed in-kernel without copies

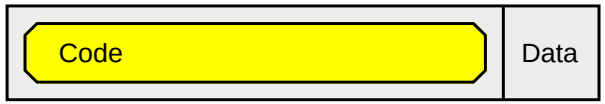
- Starting with low-performance PAN configuration:



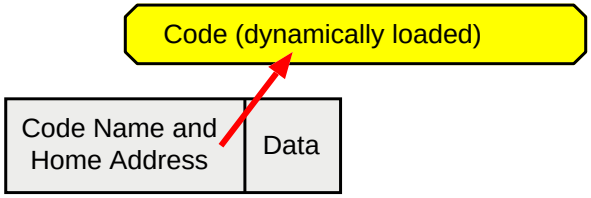


Step 1: Dynamic Code Loading and Code Naming

- Rather than having capsules *contain* code and data, capsules *name* code and contain data (done by ANTS and PAN):

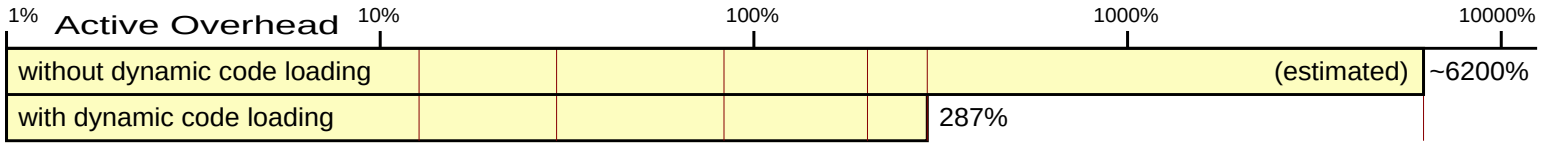


Capsule without dynamic loading



Capsule with dynamic loading

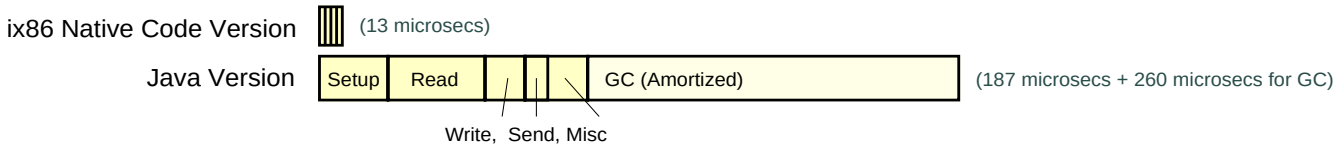
- Code is named by a crypto hash of the code, resulting in a unique name
- Code is dynamically loaded over network



(measurements using 1500 byte packets)

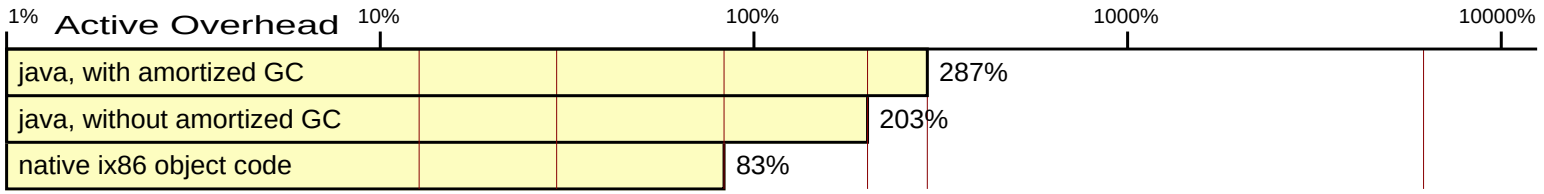
Step 2A: Importance of a Good Mobile Code System

- Current Java “Just-In-Time” (JIT) translators generate code that is much slower than comparable native code
  - Largest costs due to *garbage collection* and *object creation*
  - Some of this may be intrinsic to the design of the JavaVM
  - Comparison of baseline code run time between native ix86 code generated by gcc and Java code running in Kaffe OpenVM:



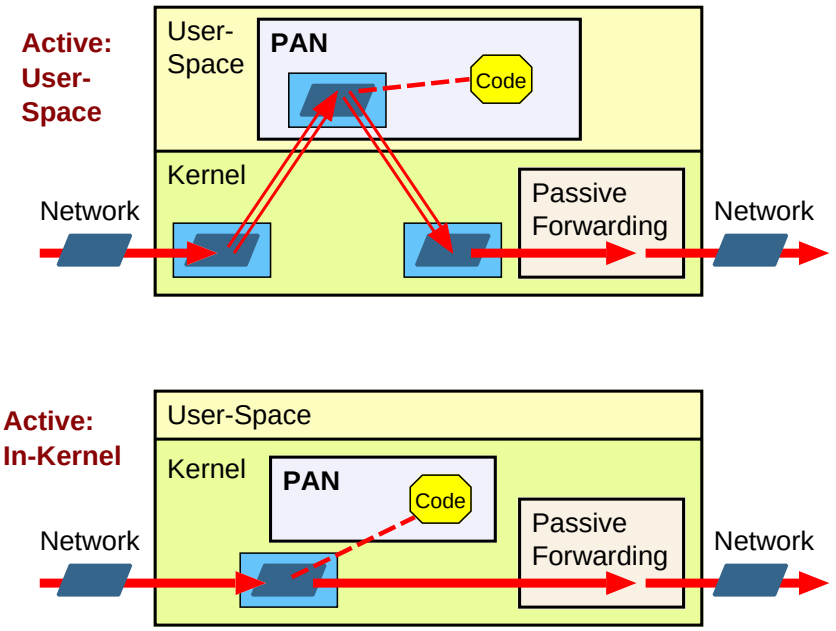
Step 2B: Using Native Code

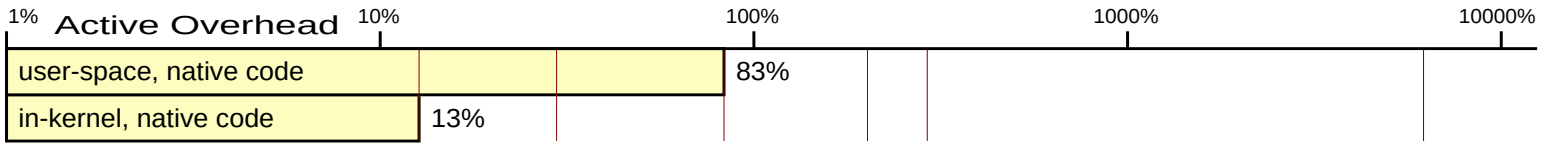
- Native code demonstrates performance that may be obtained as safe mobile code systems improve
- **Native object code doesn't provide safety, security, or interoperability!**



Step 3: Moving into the Kernel

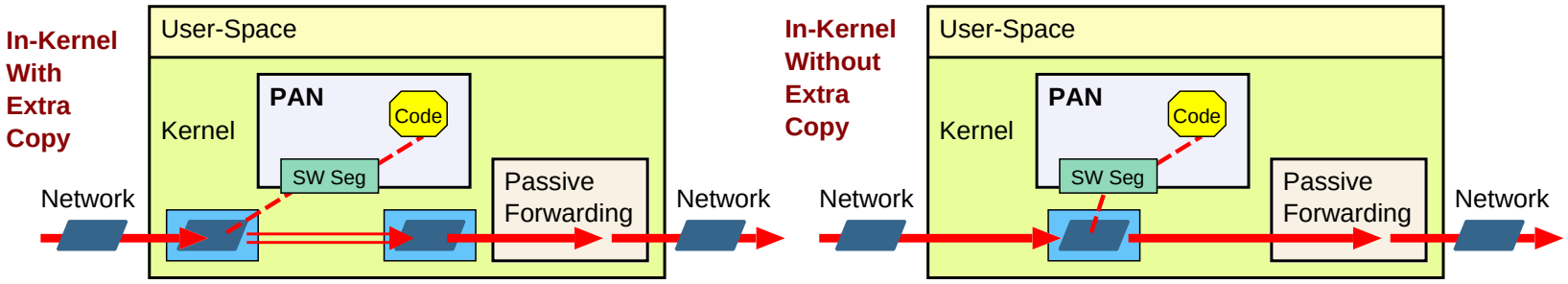
- Eliminates data copies and context switches to / from user-space

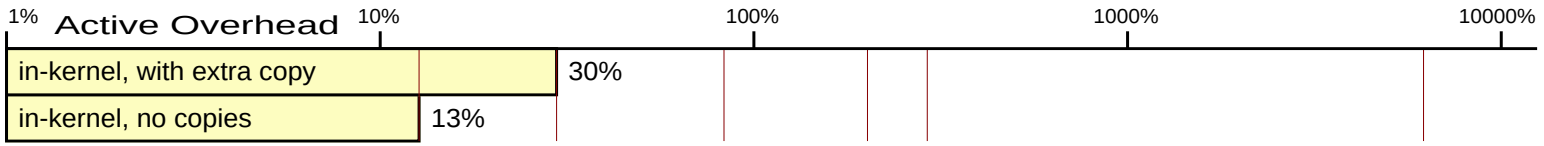




Step 4: Eliminating Data Copies With Software Segments

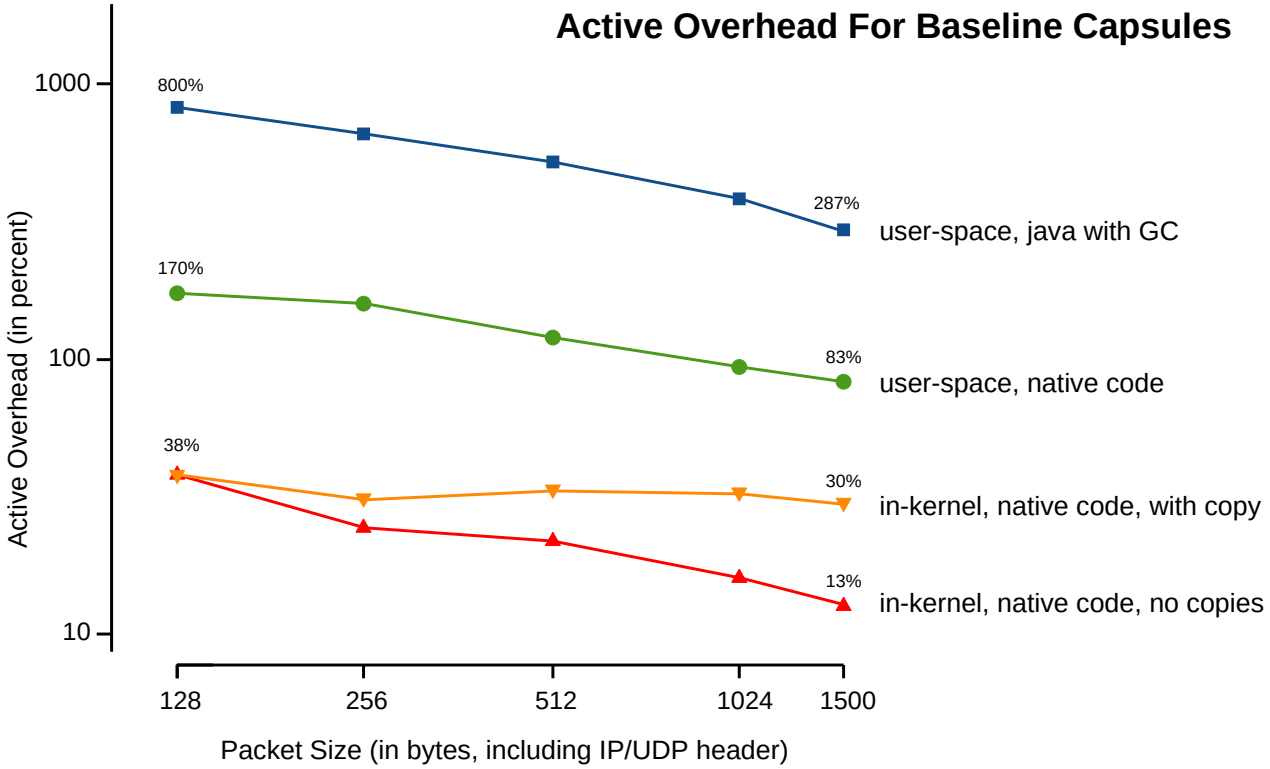
- Interoperability requires OS-independent memory format
- Naive approach: copy data into buffer for capsules to use
- *Software segment* abstraction wraps buffers
- Baseline can process capsules in-kernel without any copies!
  Results: $t_{\text{active}} - t_{\text{passive}} = 20\mu s$ regardless of capsule size
- Experiment:





Baseline Performance Hypothesis Verified

- Active overhead decreases with packet size:



Performance Explanation

- Low (13% to 38%) active overhead for in-kernel processing
- Very little on remaining critical path:
  - Capsule environment created and code looked up in code cache
  - PAN calls into capsule code
  - Capsule code looks at capsule header and calls API to forward capsule

Conclusions

- Contributions of this work:
  - Demonstrates that high performance is obtainable
  - Shows ways to get from a prototype to a high-performance active node
  - Experimental active network node for implementation research
- Future research:
  - Low active overheads with safety and security
  - By reducing overall bandwidth consumption, active protocols may generate overall gains in network performance