Bro IDS

# The Bro Network Intrusion Detection System

Robin Sommer

*International Computer Science Institute &*
*Lawrence Berkeley National Laboratory*

robin@icsi.berkeley.edu
http://www.icir.org

RWTH Aachen - Dezember 2007

ICSI INTERNATIONAL COMPUTER SCIENCE INSTITUTE

BERKELEY LAB

# The Bro NIDS - Outline

- Overview
  - System Philosophy
  - Basic Architecture
  - Examples and Deployment
  - *Tomorrow: A more practical demonstration how to work with the system*
- Current Research with the Bro NIDS
  - Port-independent protocol analysis
  - Parallel Analysis
    - The NIDS Cluster
    - Strategies for a multi-threaded Bro

# Bro Overview

# System Philosophy

- Bro is being developed at ICSI & LBNL since 1996
  - LBNL has been using Bro operationally for >10 years
  - It is one of the main components of the lab's network security infrastructure
- Bro provides a real-time network analysis framework
  - Primary a network intrusion detection system (NIDS)
  - However it is also used for pure traffic analysis
- Focus is on
  - Application-level semantic analysis (rather than analyzing individual packets)
  - Tracking information over time
- Strong separation of mechanism and policy
  - The core of the system is policy-neutral (no notion of "good" or "bad")
  - User provides local site policy

# System Philosophy (2)

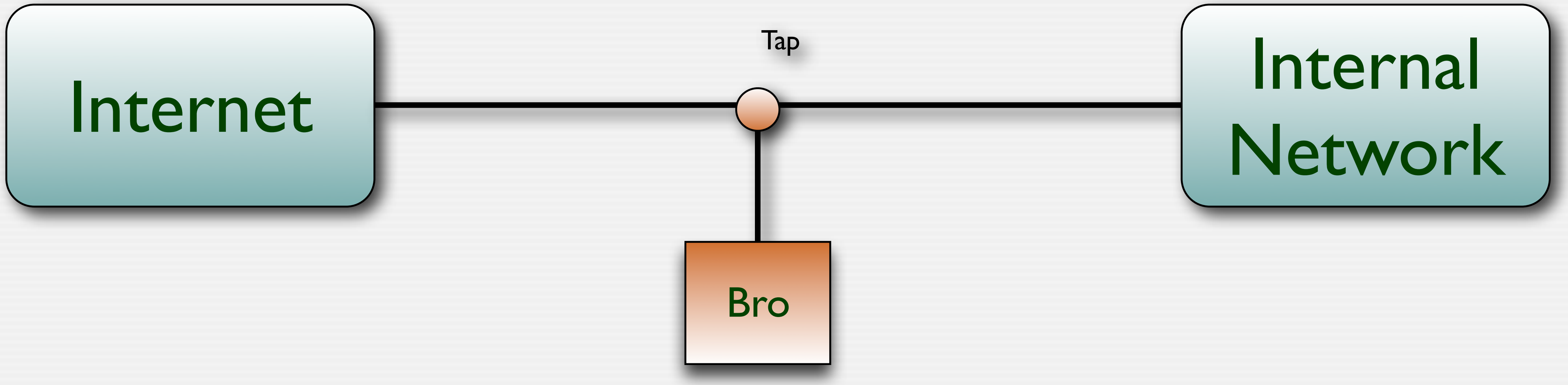
- Operators *program* their policy
  - Not really meaningful to talk about what Bro detects "by default"
- Bro is not restricted to any particular analysis model
- Most typical is the misuse-detection style
- Focus is *not* signature matching
  - Bro is fundamentally different from, e.g., Snort (though it *can* do signatures as well)
- Focus is *not* anomaly detection
  - Though it does support such approaches (and others) in principle
- System thoroughly logs all activity
  - It does not just alert

# Target Environments

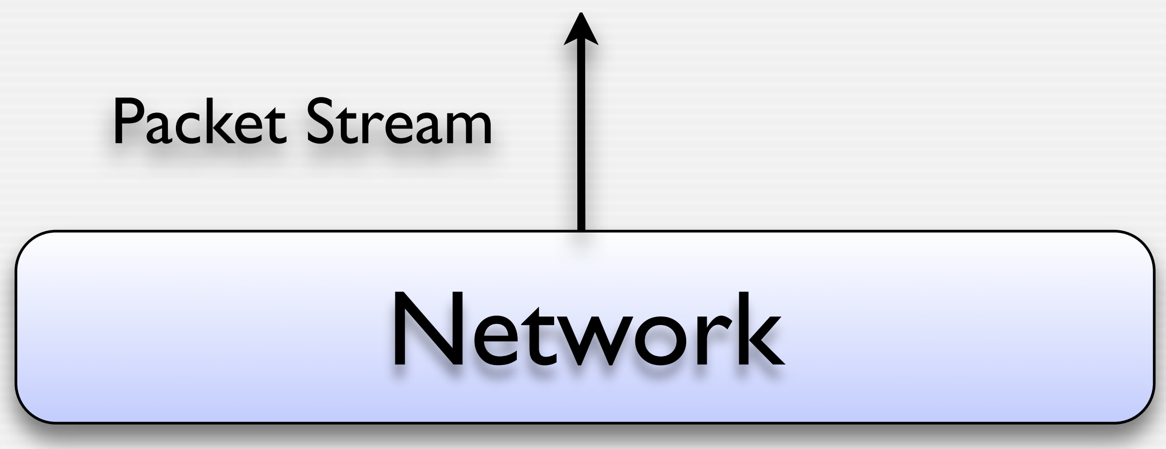
- Bro is specifically well-suited for scientific environments
  - Extremely useful in networks with liberal ("default allow") policies
  - Supports intrusion prevention schemes
  - High-performance on commodity hardware
  - Runs on Unix-based systems (e.g., Linux, FreeBSD, MacOS)
  - Open-source (BSD license)
- It does however require some effort to use effectively
  - Pretty complex, script-based system
  - Requires understanding of the network
  - No GUI, just ASCII logs
  - Only partially documented
  - Lacking resources to fully polish the system
- Development is primarily driven by *research*
  - However, our focus is operational use; we invest much time into "practical" issues
  - Want to bridge gap between research and operational deployment

# Bro Deployment

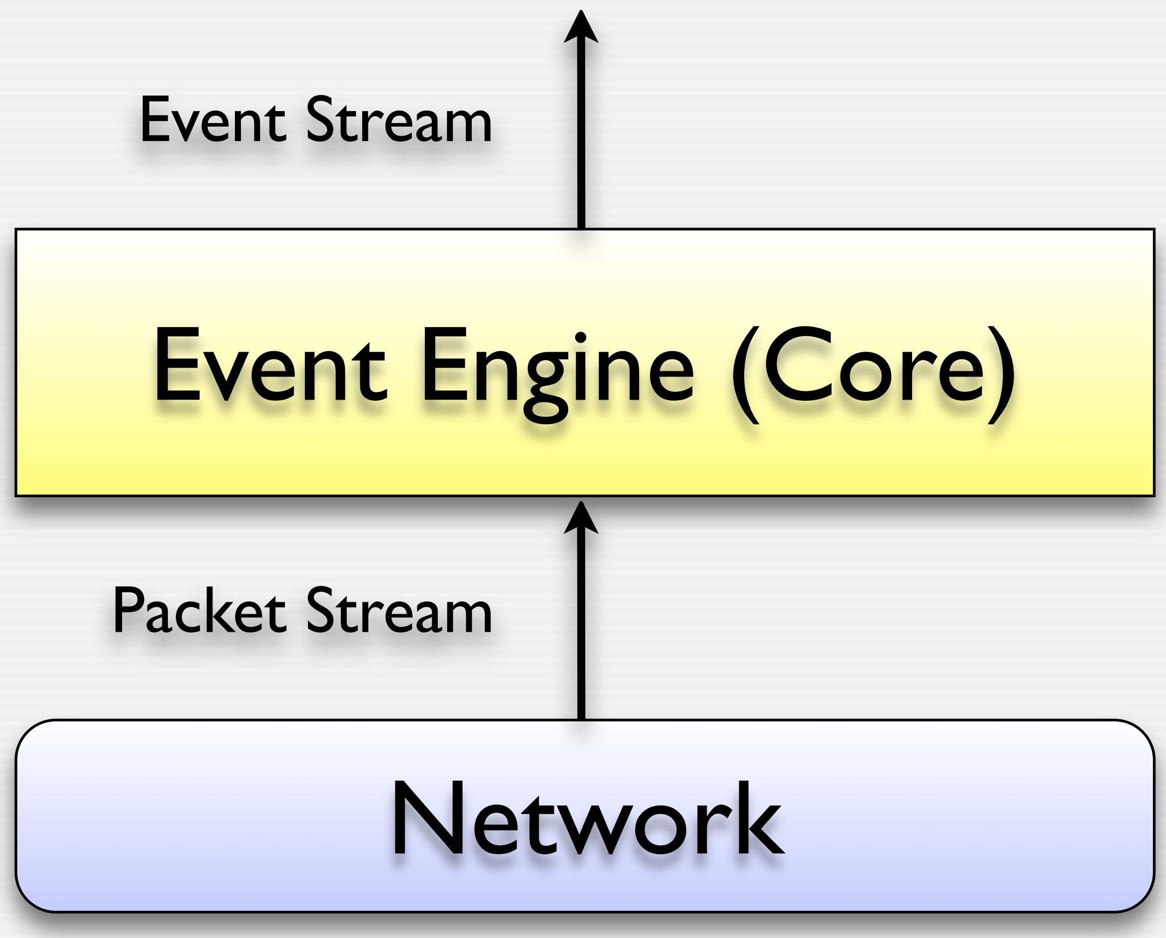
- Bro is typically deployed at a site's upstream link
  - Monitors all external packets coming in or going out
  - Deployment similar to other NIDS
  - By default, purely passive monitoring



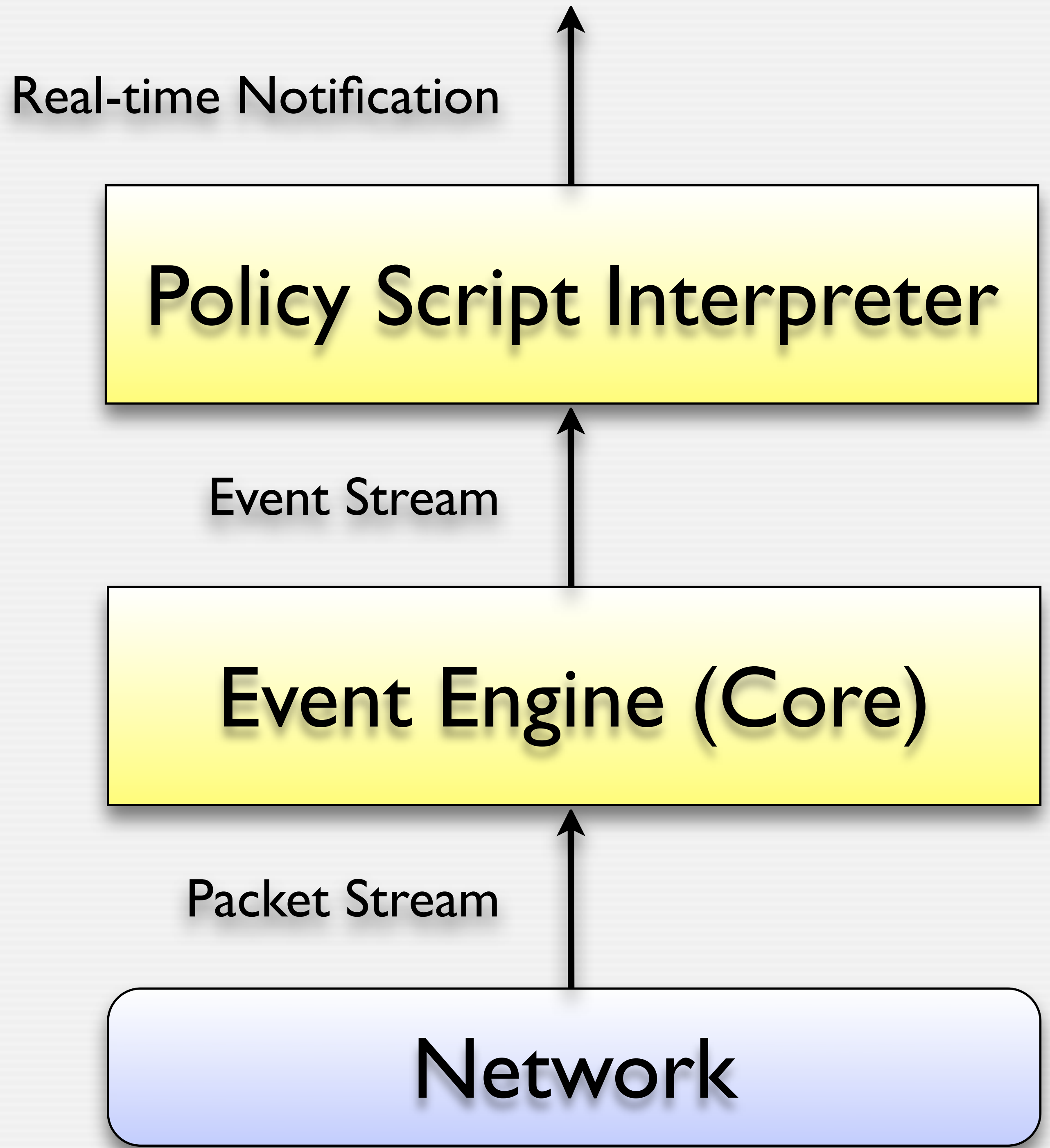
# Architecture



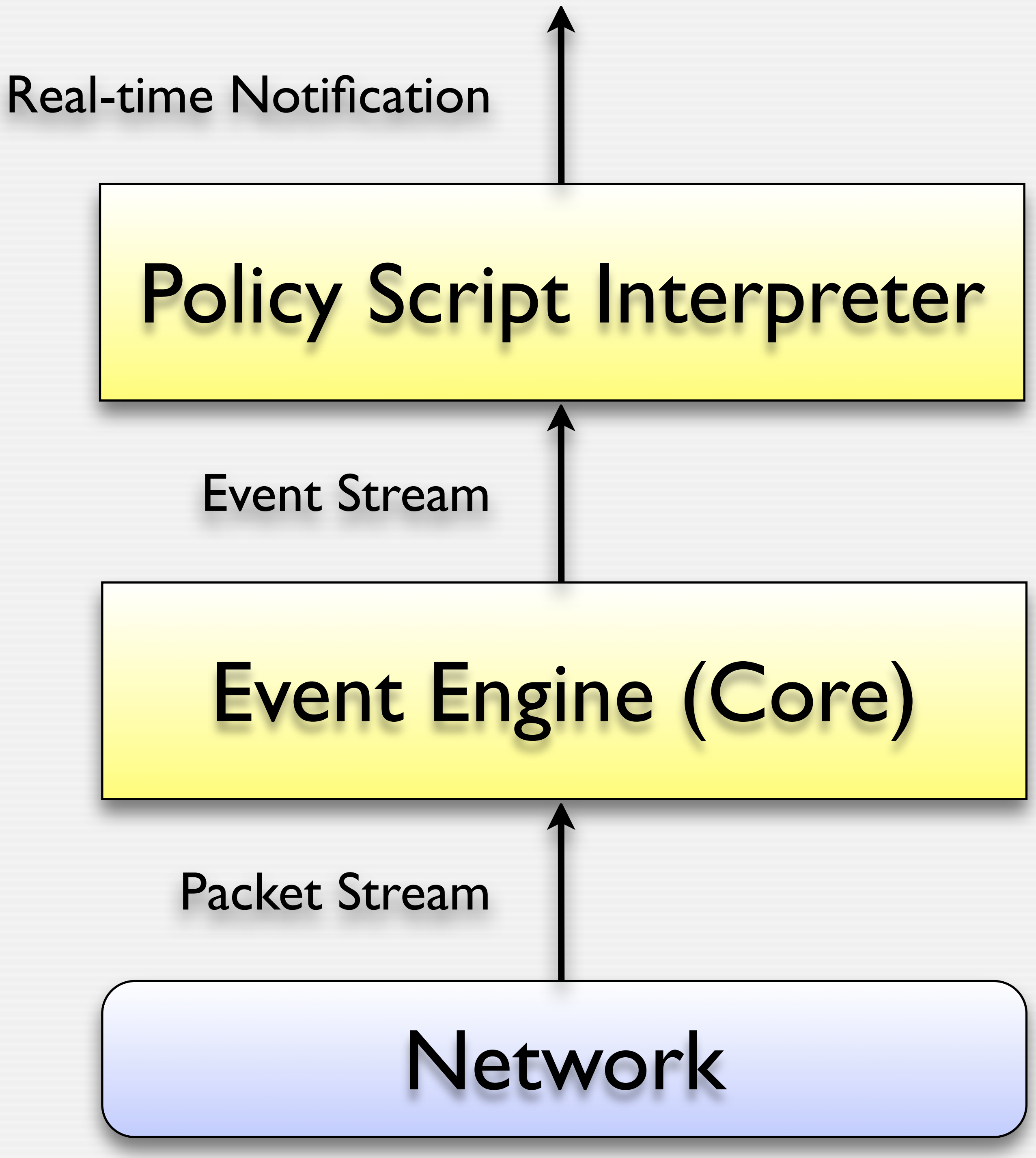
# Architecture



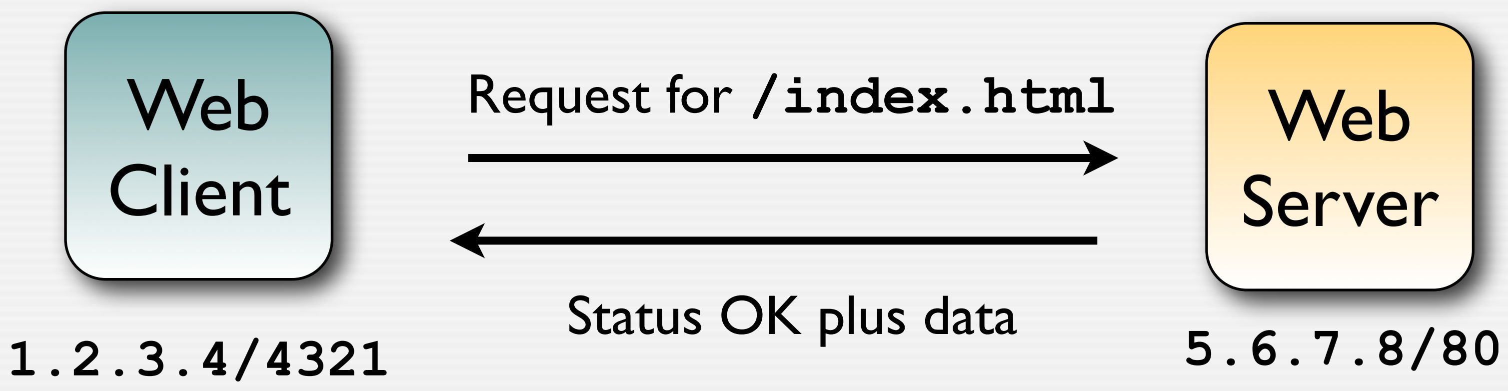
# Architecture

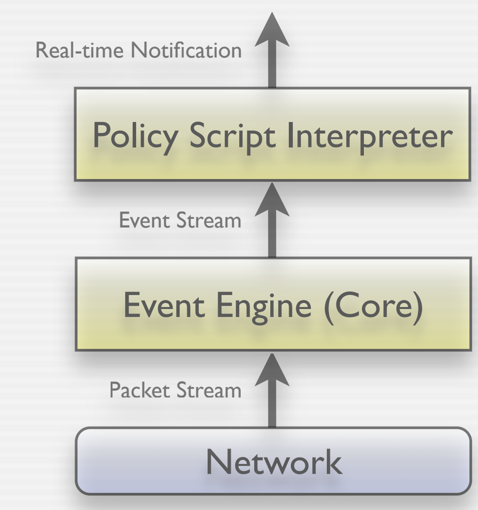


# Event Model - Example

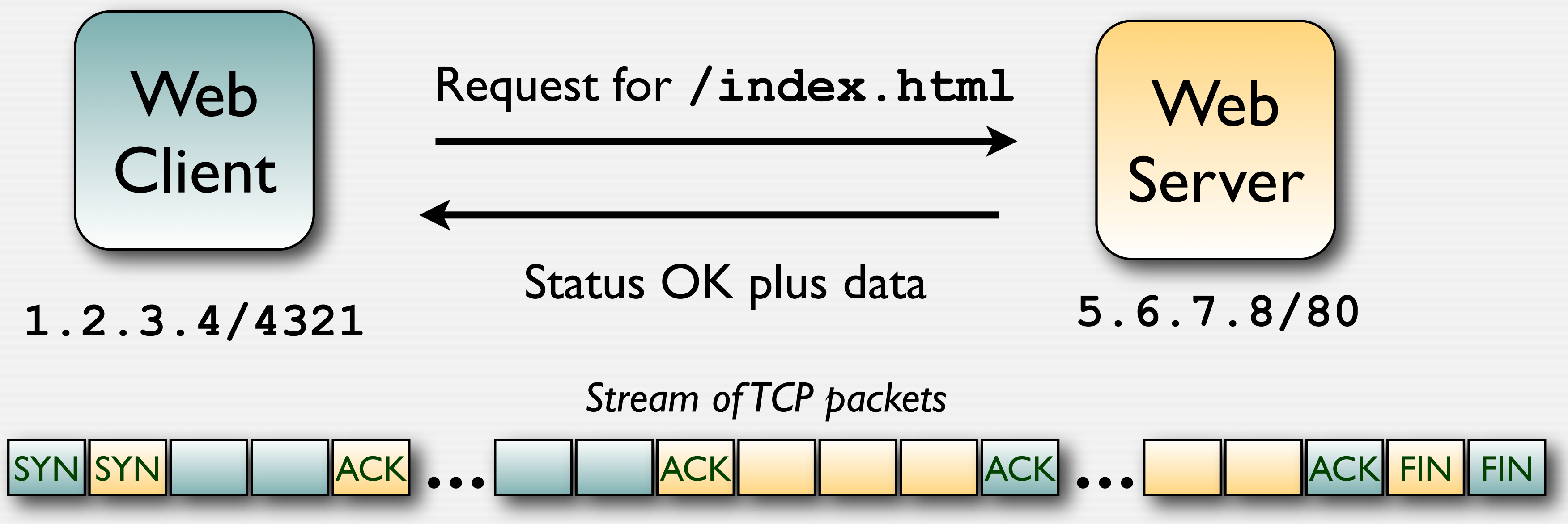


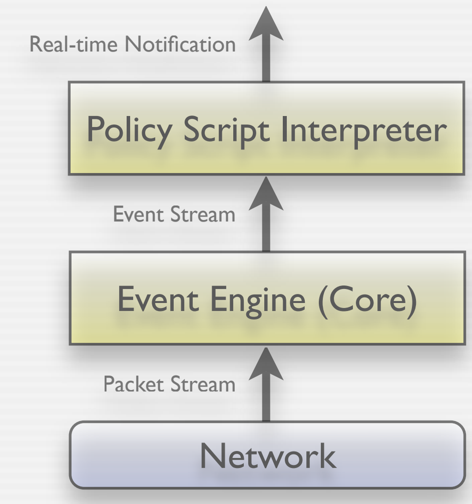
# Event Model - Example



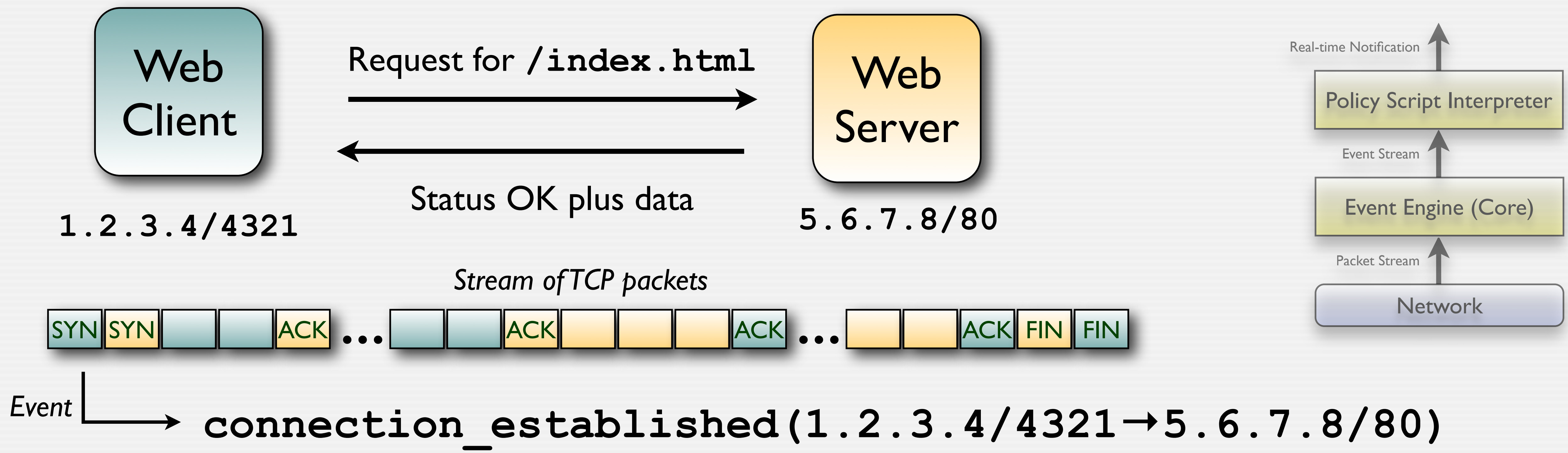



# Event Model - Example

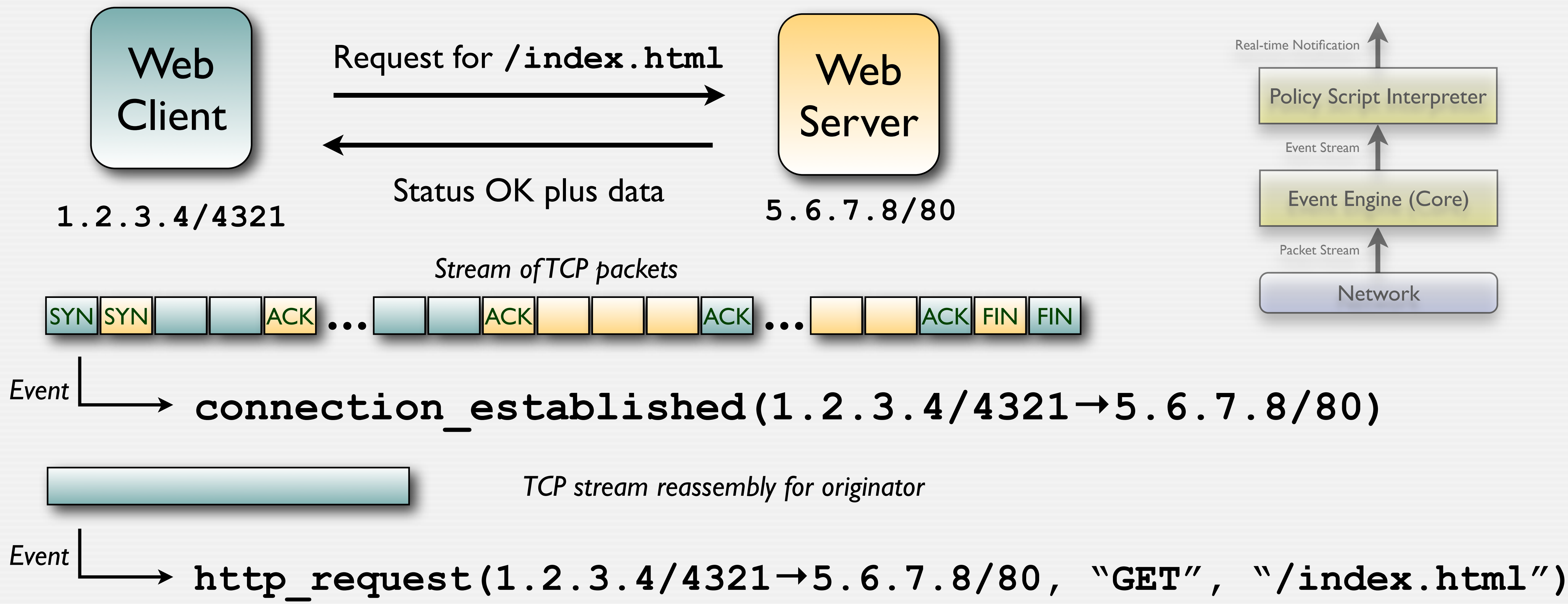


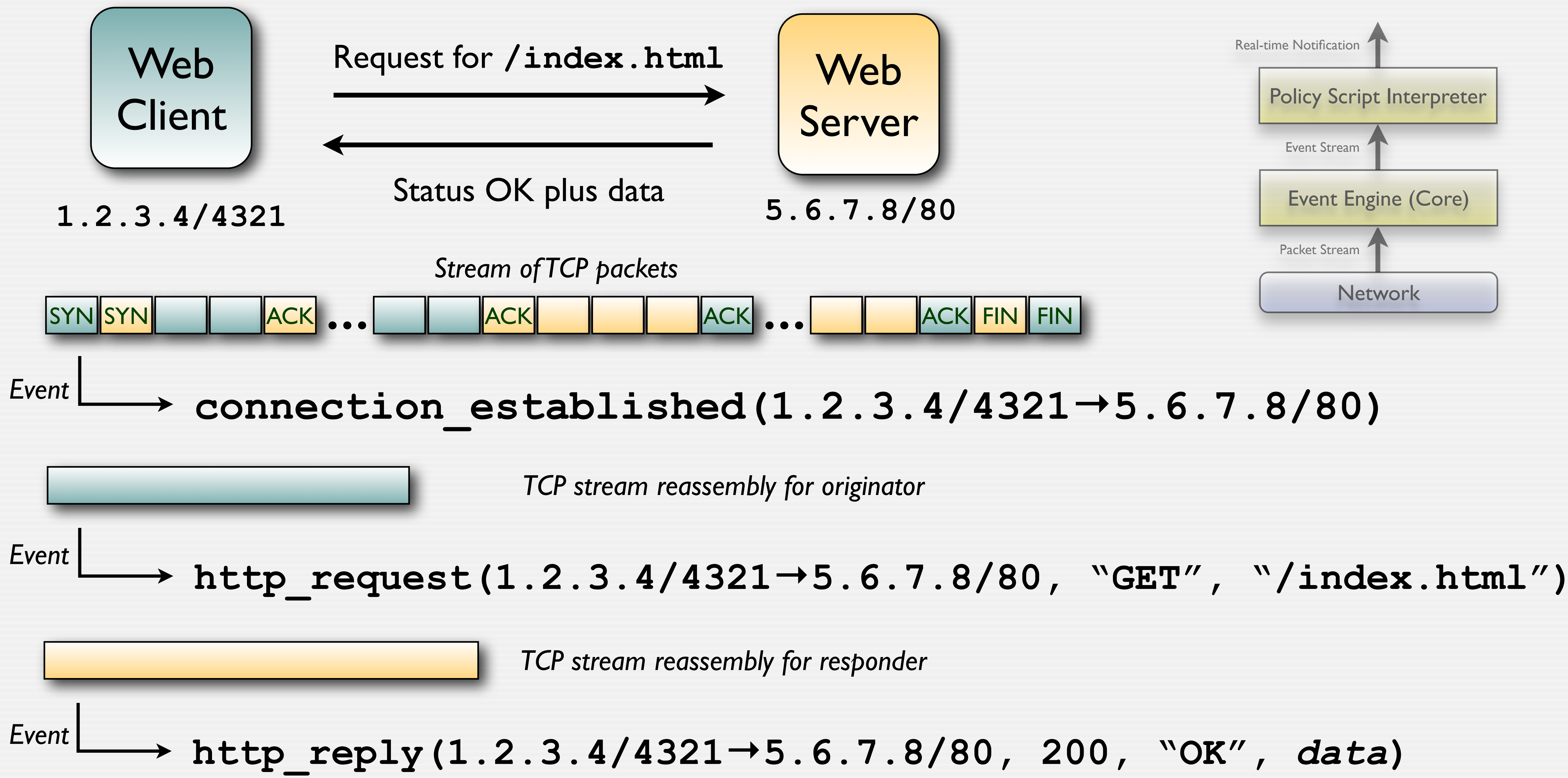
# Event Model - Example



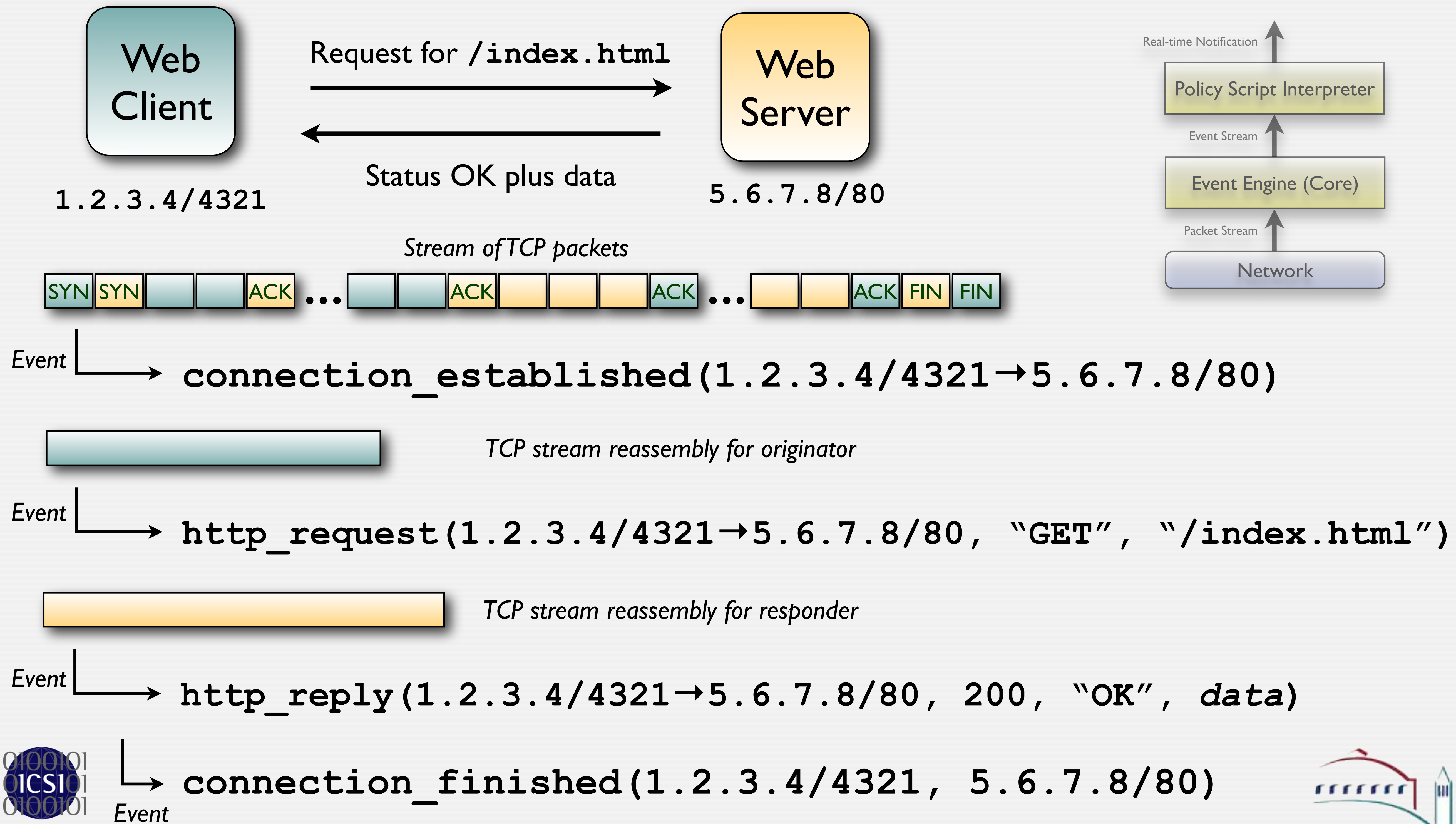
# Event Model - Example



# Event Model - Example



# Event Model - Example



# Event-Engine

- Performs *policy-neutral* analysis
  - Turns low-level activity into high-level events
  - Examples: `connection_established, http_request`
  - Events are annotated with context (e.g., IP addresses, URL)
- Event-engine is written in C++ for performance
  - Performs work *per packet*
- Contains *analyzers* for >30 protocols, including
  - ARP, IP, ICMP, TCP, UDP
  - DCE-RPC, DNS, FTP, Finger, Gnutella, HTTP, IRC, Ident, NCP, NFS, NTP, NetBIOS, POP3, Portmapper, RPC, Rsh, Rlogin, SMB, SMTP, SSH, SSL, SunRPC, Telnet
- Analyzers generate ~300 types of events

# Expressing Policy with Scripts

- Scripts are written in custom, domain-specific language
  - Bro ships with 20K+ lines of script code
  - Default scripts detect attacks & log activity extensively
- Scripts process event stream, incorporating ...
  - ... context from past events
  - ... site's local security policy
- Scripts take actions
  - Generating alerts via syslog or mail
  - Executing program as a form of response
  - Recording activity to disk

# Bro's Scripting Language

- Bro's scripting language is
  - Procedural
  - Event-based
  - Strongly typed
  - Rich in types
    - Usual script-language types, such as tables and sets
    - Domain-specific types, such as addresses, ports, subnets
  - Supporting state management (persistance, expiration, timers, etc.)
  - Supporting communication with other Bro instances

# Script Example: Matching URLs

```
event http_request(c: connection, method: string, path: string)
{
    if ( method == “GET” && path == “/etc/passwd” )
        NOTICE(SensitiveURL, c, path);
}
```

*http_request(1.2.3.4/4321→5.6.7.8/80, “GET”, “/index.html”)*

*Code simplified. See policy/http-request.bro.*

# Script Example: Tracking SSH Hosts

```
global ssh_hosts: set[addr];

event connection_established(c: connection)
    {
    local responder = c$id$resp_h; # Responder's address
    local service = c$id$resp_p;   # Responder's port

    if ( service != 22/tcp )
        return; # Not SSH.

    if ( responder in ssh_hosts )
        return; # We already know this one.

    add ssh_hosts[responder]; # Found a new host.
    print "New SSH host found", responder;
    }
```

ICSI INTERNATIONAL COMPUTER SCIENCE INSTITUTE

BERKELEY LAB

# Policy-neutral Logging

- Bro's default scripts perform two main tasks
  - Detecting malicious activity (mostly misuse-detection)
  - Logging activity comprehensively without any actual assessment
- In practice, the policy-neutral logs are often most useful
  - Typically we do not know in advance how the next attacks looks like
  - But when an incident occurred, we need to understand what exactly happened
- Typical questions asked
  - *"How did the attacker get in? What damage did he do? Did the guy access other hosts as well? How can we detect similar activity in the future?"*

# Example Log: Connection Summaries

- One-line summaries for all TCP connections
- Most basic, yet also one of the most useful analyzers

```
Time                     Duration   Source             Destination
1144876596.658302  1.206521   192.150.186.169   62.26.220.2 \
  http   53052     80         tcp      874          1841        SF        X
  Serv   SrcPort   DstPort    Proto    SrcBytes     DstBytes    State     Local
```

*LBNL has connection logs for every connection attempt since June 94!*

# Example Log: HTTP Session

```
1144876588.30 start 192.150.186.169:53041 > 195.71.11.67:80
1144876588.30 GET /index.html (200 "OK" [57634] www.spiegel.de)
1144876588.30 > HOST:  www.spiegel.de
1144876588.30 > USER-AGENT:  Mozilla/5.0 (Macintosh; PPC Mac OS ...
1144876588.30 > ACCEPT:  text/xml,application/xml,application/xhtml ...
1144876588.30 > ACCEPT-LANGUAGE:  en-us,en;q=0.7,de;q=0.3
[...]
1144876588.77 < SERVER:  Apache/1.3.26 (Unix) mod_fastcgi/2.2.12
1144876588.77 < CACHE-CONTROL:  max-age=120
1144876588.77 < EXPIRES:  Wed, 12 Apr 2006 21:18:28 GMT
[...]
1144876588.77 <= 1500 bytes: "<!-- Vignette StoryServer 5.0 Wed Apr..."
1144876588.78 <= 1500 bytes: "r "http://spiegel.ivwbox.de" r..."
1144876588.78 <= 1500 bytes: "icon.ico" type="image/ico">^M^J ..."
1144876588.94 <= 1500 bytes: "erver 5.0 Mon Mar 27 15:56:55 ..."
[...]
```

ICSI INTERNATIONAL COMPUTER SCIENCE INSTITUTE

BERKELEY LAB

# Deployment Example: Lawrence Berkeley National Lab

# Lawrence Berkeley National Lab

- Main site located on a 200-acre area in the Berkeley hills
- Close proximity to UC Berkeley

ICSI INTERNATIONAL COMPUTER SCIENCE INSTITUTE

BERKELEY LAB

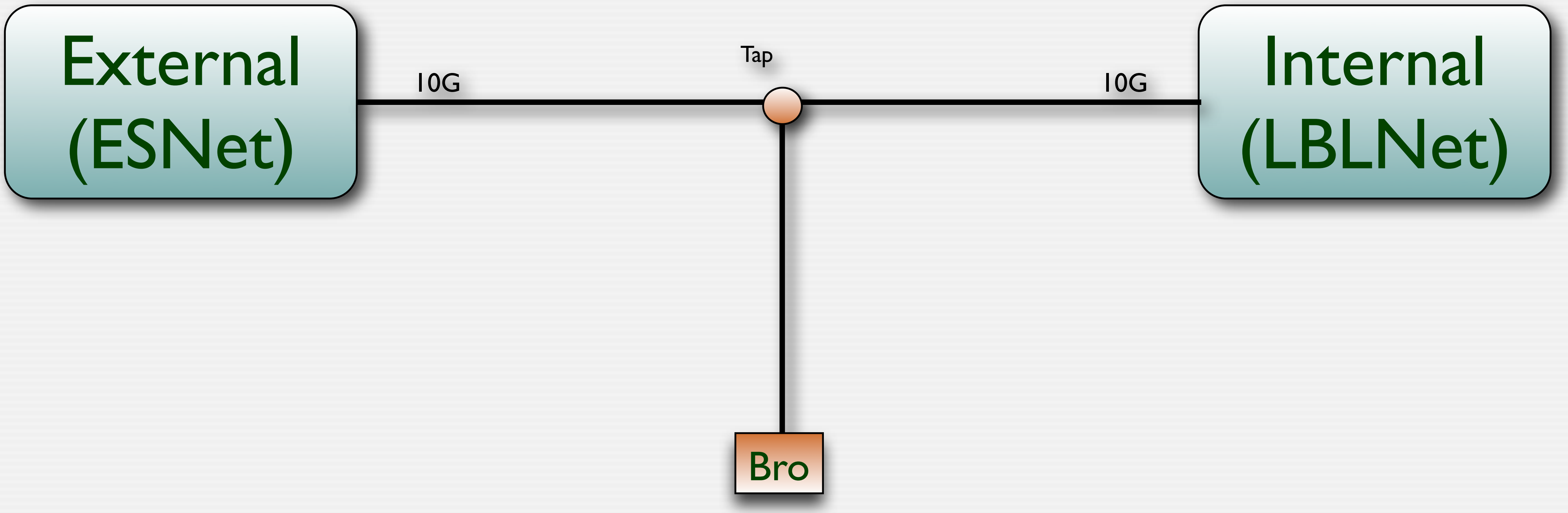
# Lawrence Berkeley National Lab

- Managed by UC for the U.S. Department of Energy
- Open, unclassified research
  - Research is freely shared
  - Collaborations around the world
- Diversity of research
  - Nanotechnology, Energy, Physics, Biology, Chemistry, Environmental, Computing
- Diverse user community
  - 3,800 employees
  - Scientific facilities used by researchers around the world
  - Many staff people have dual appointments with UC Berkeley
  - Many users are transient and not employees
- Very liberal, default-allow security policy
  - Characteristic for many research environments
  - Requires comprehensive approach to monitoring

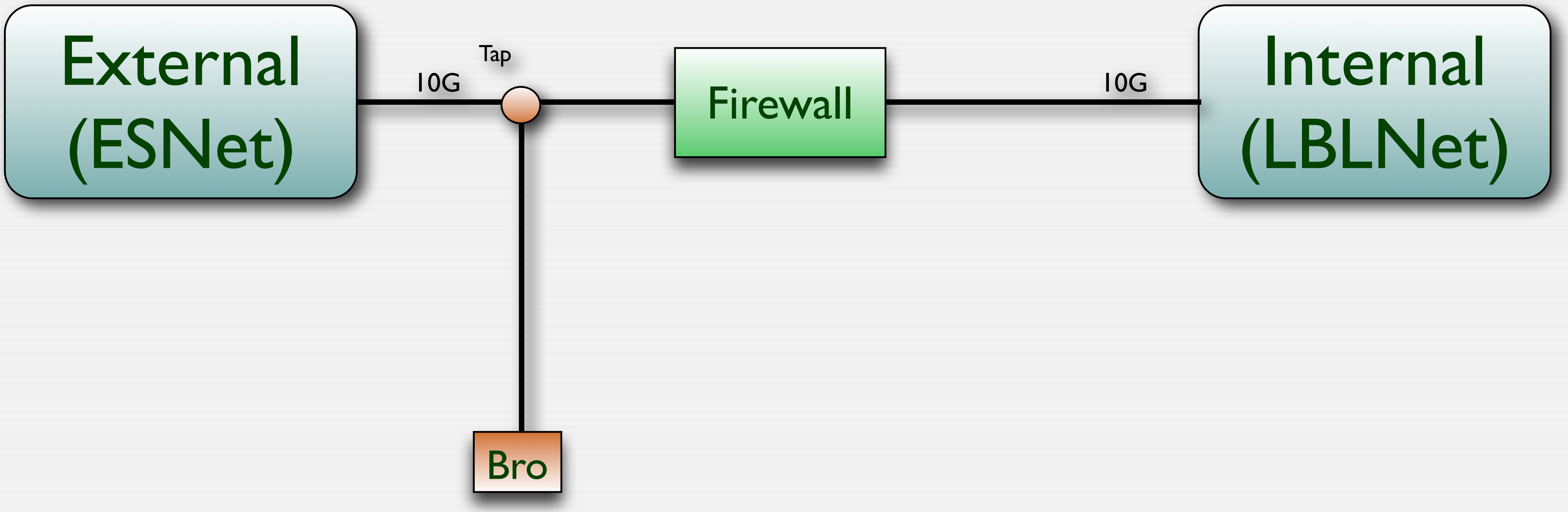
# Bro at the Lawrence Berkeley Lab

- Primary security threats
  - System compromises
  - Loss of personally identifying information (PII)
  - Credential theft (e.g., SSH keys)
  - Bad publicity
  - Auditors(!)
- LBNL has been using Bro for >10 years
  - Monitors the lab's 10 Gbps Internet uplink
  - Credited with numerous attack detections
- Bro is one of the main components of lab's security
  - Several Bro boxes for different tasks
  - Bro automatically *blocks* attackers

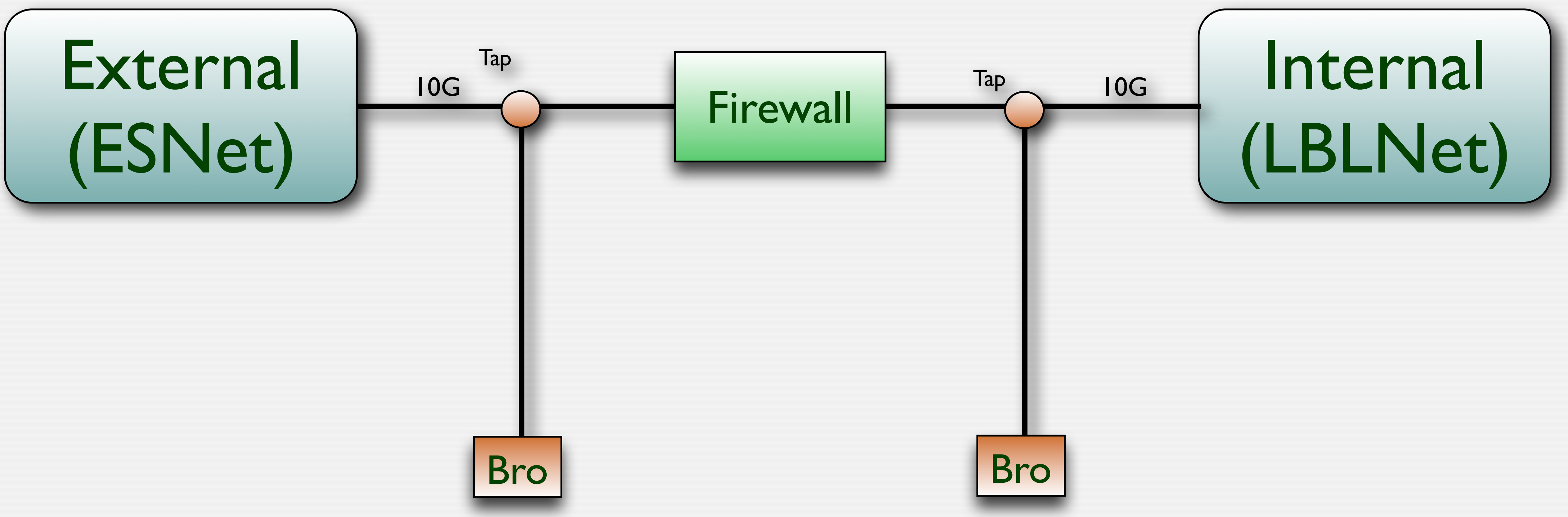
# LBNL's Bro Setup



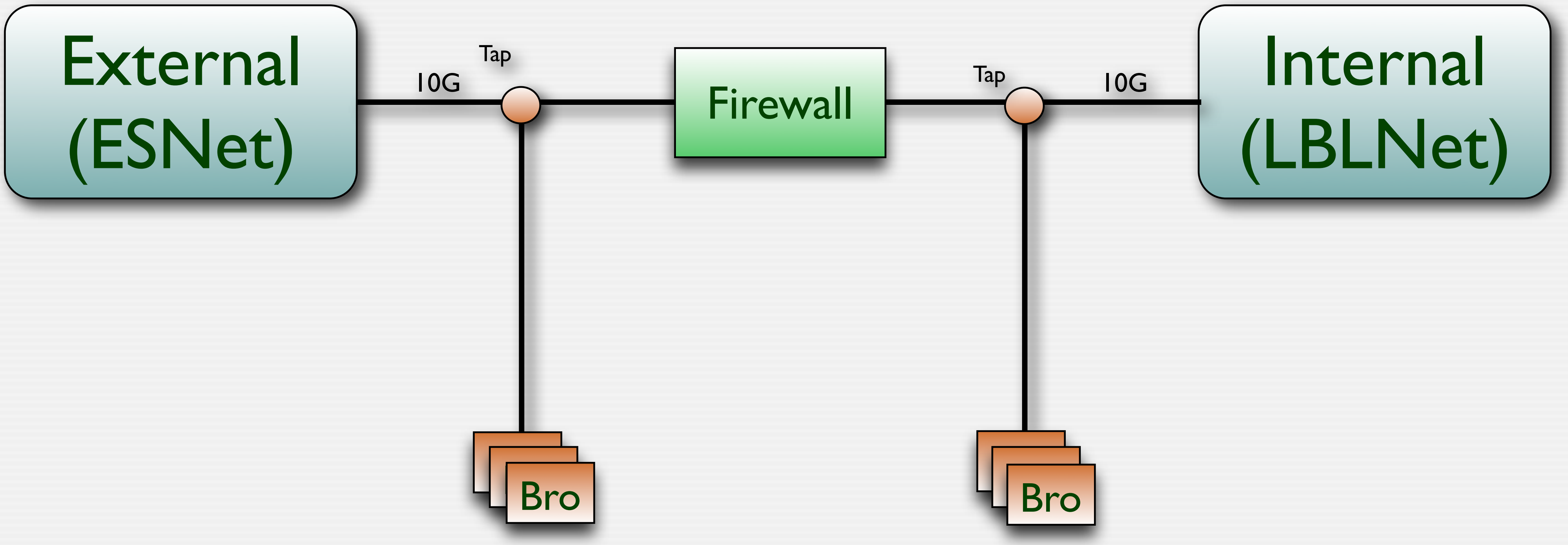
# LBNL's Bro Setup



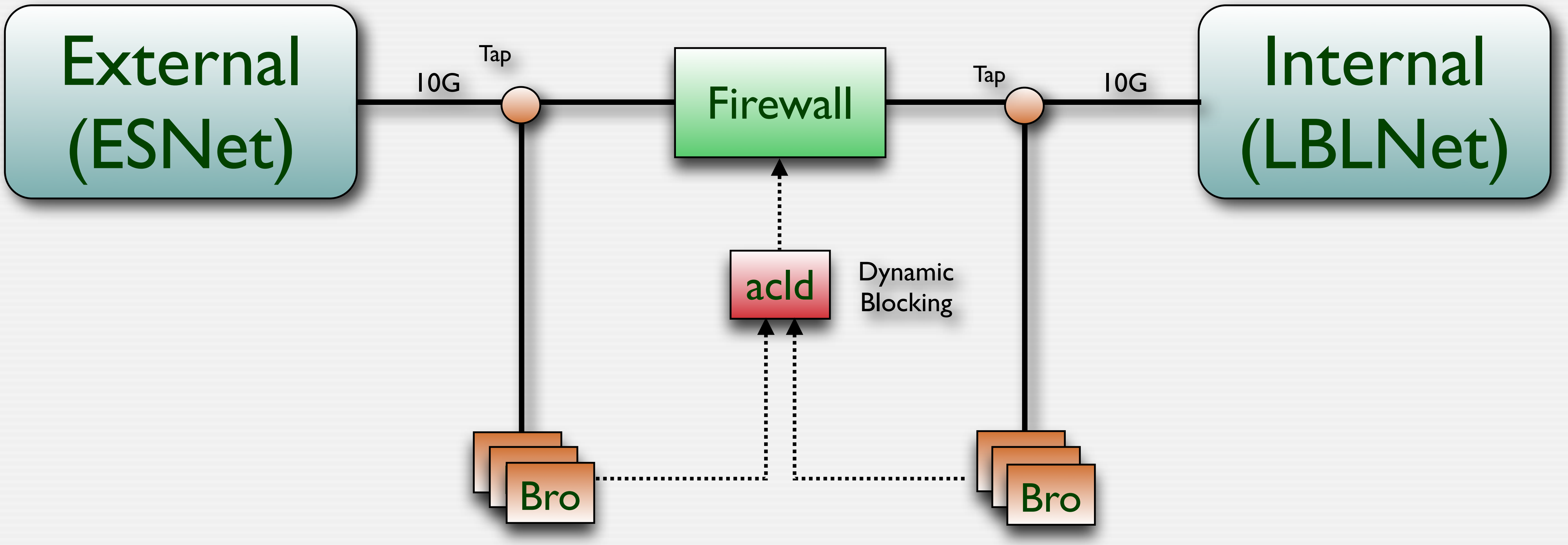
# LBNL's Bro Setup



# LBNL's Bro Setup



# LBNL's Bro Setup




Bro blocks more than 4000 addresses per day!

# Port-independent Protocol Analysis with Dynamic Protocol Detection (DPD)

# Port-based Protocol Analysis

- Bro has lots of application-layer analyzers
- But which protocol does a connection use?
- Traditionally NIDS rely on ports
  - Port 80? Oh, that's HTTP.
- Obviously deficient in two ways
  - There's non-HTTP traffic on port 80 (firewalls tend to open this port...)
  - There's HTTP on ports other than port 80
- Particularly problematic for security monitoring
  - Want to know if somebody avoids the well-known port

# Port-independent Analysis

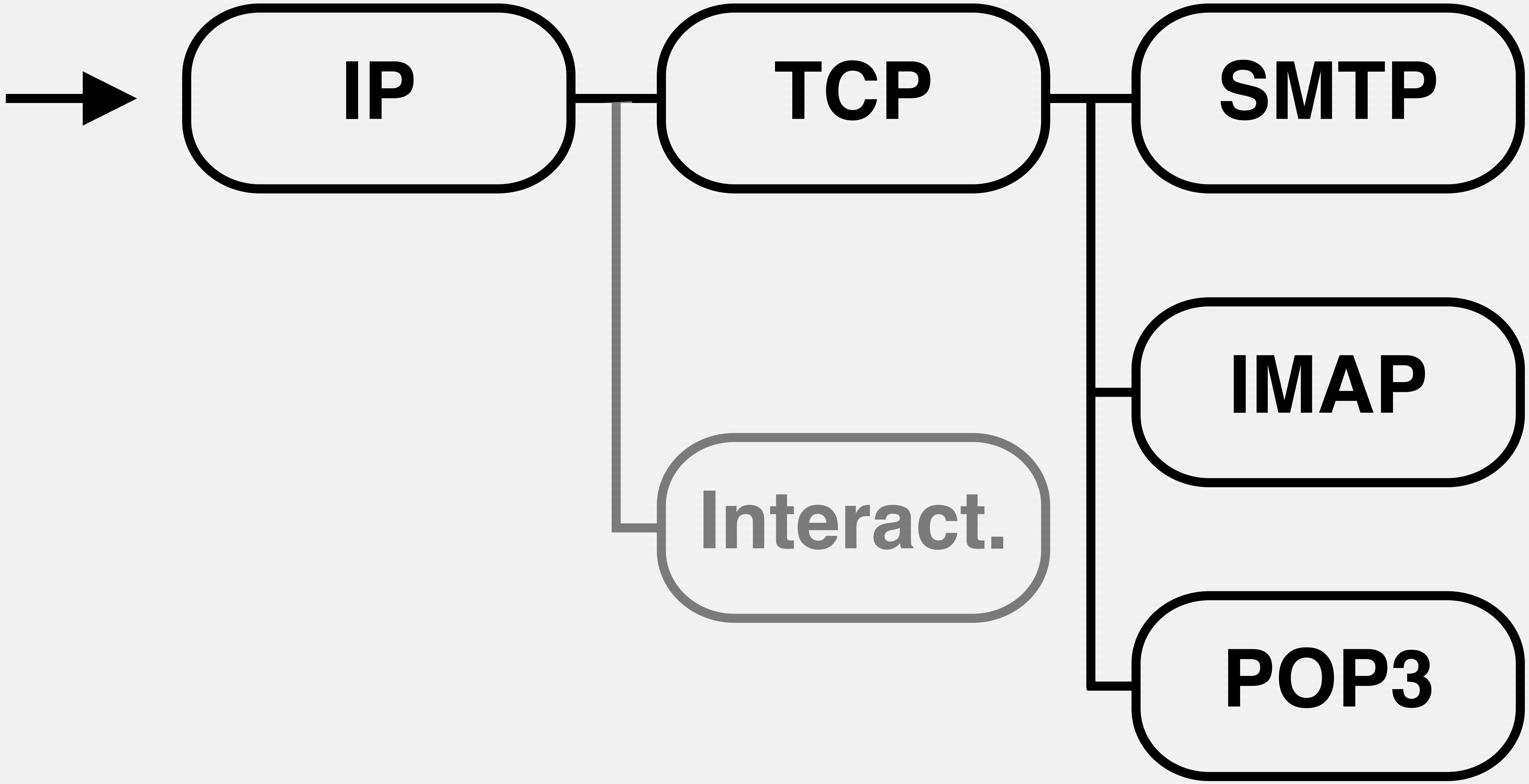
- Look at the *payload* to see what is, e.g., HTTP
- Analyzers already know how a protocol looks like
  - Leverage existing protocol analyzers
  - Let each analyzer *try to parse* the payload
    - If it succeeds, great!
    - If not, then it's actually another protocol
- Ideal setting: *for every connection, try all analyzers*
- However, performance is prohibitive
  - Can't parse 10000s of connections in parallel with all analyzers

# Making it realistic ...

- Bro uses byte patterns to *prefilter* connections
  - An HTTP signature looks for *potential* uses of HTTP
  - Then the HTTP analyzer verifies by trying to parse the payload
  - Signatures can be loose because false positives are inexpensive (no alerts!)
- Other NIDS often ship with protocol signatures
  - These directly generate alerts (imagine reporting all non-80 HTTP conns!)
  - These do not trigger protocol-layer semantic analysis (e.g., extracting URLs)
- In Bro, a match triggers further analysis
- Main internal concept: analyzer trees
  - Each connection is associated with an analyzer tree

# Example: Analyzer Tree

A connection looks like mail, but what is it?



# Application Example: FTP Data

- FTP data sessions can't be analyzed by port-based NIDSs
- Bro's DPD has a notion of "expected connections"
  - Can be told in advance which analyzer to use for an upcoming connection
- Bro also has a *File Analyzer*
  - Determines file-type (via libmagic)
  - Checks for malware (via libclamav)
- FTP analysis combines these
  - Parses control connection to learn about upcoming FTP data
  - Calls `expect_connection(conn_id, FileAnalyzer)`
  - File Analyzer is inserted into analyzer tree when connection is seen

# Application Example: FTP Data (2)

```
xxx.xxx.xxx.xxx/2373 > xxx.xxx.xxx.xxx/5560 start
response (220 Rooted Moron Version 1.00 4 WinSock ready...)
USER ops (logged in)
SYST (215 UNIX Type: L8)
[...]
LIST -al (complete)
TYPE I (ok)
SIZE stargate.atl.s02e18.hdtv.xvid-tvd.avi (unavail)
PORT xxx,xxx,xxx,xxx,xxx,xxx (ok)
STOR stargate.atl.s02e18.hdtv.xvid-tvd.avi, NOOP (ok)
ftp-data video/x-msvideo `RIFF (little-endian) data, AVI'
[...]
response (226 Transfer complete.)
[...]
QUIT (closed)
```

# Application Example: Finding Bots

- IRC-based bots are a prevalent problem
  - Infected client machines accept commands from their "master"
  - Often IRC-based but not on port 6667
- Just detecting IRC connections not sufficient
  - Often there is legitimate IRC on ports other than 6667
- DPD allows to analyze all IRC sessions *semantically*
  - Looks for typical patterns in NICK and TOPIC
  - Reports if it finds IRC sessions showing both such NICKs and TOPICs
- Very reliable detection of bots
  - Munich universities use it to actively block internal bots automatically

# Application Example: Finding Bots (2)

```
Detected bot-servers:
IP1 - ports 9009,6556,5552 password(s) <none> last 18:01:56
 channel #vec:
 topic ".asc pnp 30 5 999 -b -s|.wksescan 10 5 999 -b -s|[...]"
 channel #hv:
 topic ".update http://XXX/image1.pif f"
[...]
Detected bots:
IP2 - server IP1 usr 2K-8006 nick [P00|DEU|59228]
IP4 - server IP1 usr XP-3883 nick [P00|DEU|88820]
[...]
```

# DPD: Summary & Outlook

- Port-independent protocol analysis
  - Idea is straight-forward, but Bro is the only system which does it
- Bro now has a very generic analyzer framework
  - Allows arbitrary changes to analyzer setup during lifetime of connection
  - Is not restricted to any particular approach for protocol detection
- Main performance impact: need to examine *all* packets
  - Well, that's pretty hard to avoid
- Potential extensions
  - More protocol-detection heuristics (e.g., statistical approaches)
  - Analyze tunnels by pipelining analyzers (e.g., to look inside SSL)
  - Hardware support for pre-filtering (e.g., on-NIC filtering)

# Parallel Network Intrusion Detection

# Motivation

- NIDSs have reached their limits on commodity hardware
  - Keep needing to do *more analysis* on *more data* at *higher speeds*
  - However,  CPU performance is not growing anymore the way it used to
  - Single NIDS instance (e.g., Snort, Bro) cannot cope with Gbps links
- To overcome, we must either
  - Restrict the amount of analysis, or
  - Turn to expensive,custom hardware, or
  - Employ some form of parallelization of the processing across
    (a) machines, or
    (b) CPUs

# Orthogonal Approaches

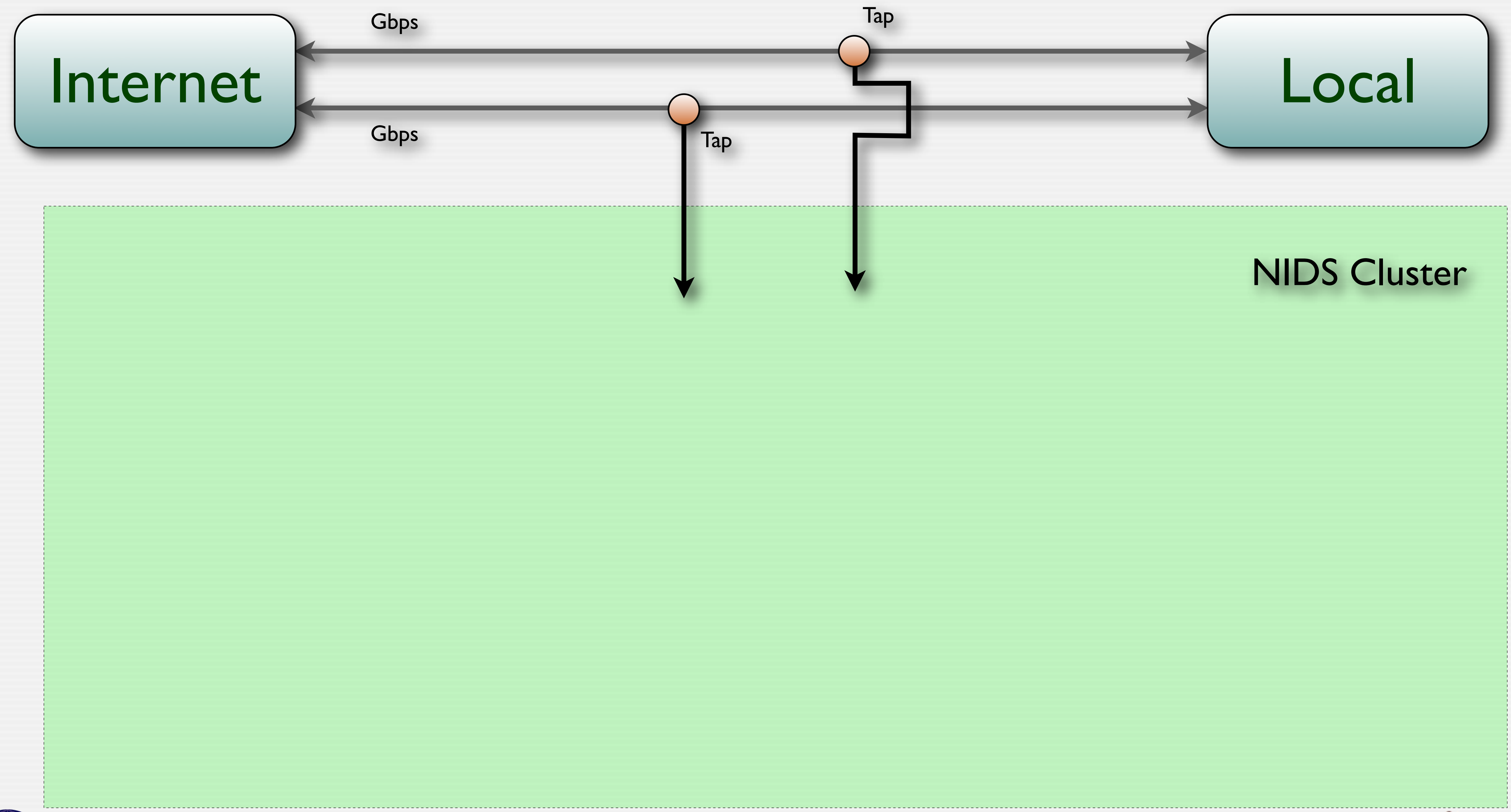
- The NIDS Cluster
  - Many PCs instead of one
  - Communication and central user interface creates the impression of one system
  - First installations up and running
- Parallel operation within a single NIDS instance
  - In software: multi-threaded analysis on multi-CPU/multi-core systems
  - In hardware: compile analysis into a parallel execution model (e.g., on FPGAs)
  - Work in progress

# The NIDS Cluster

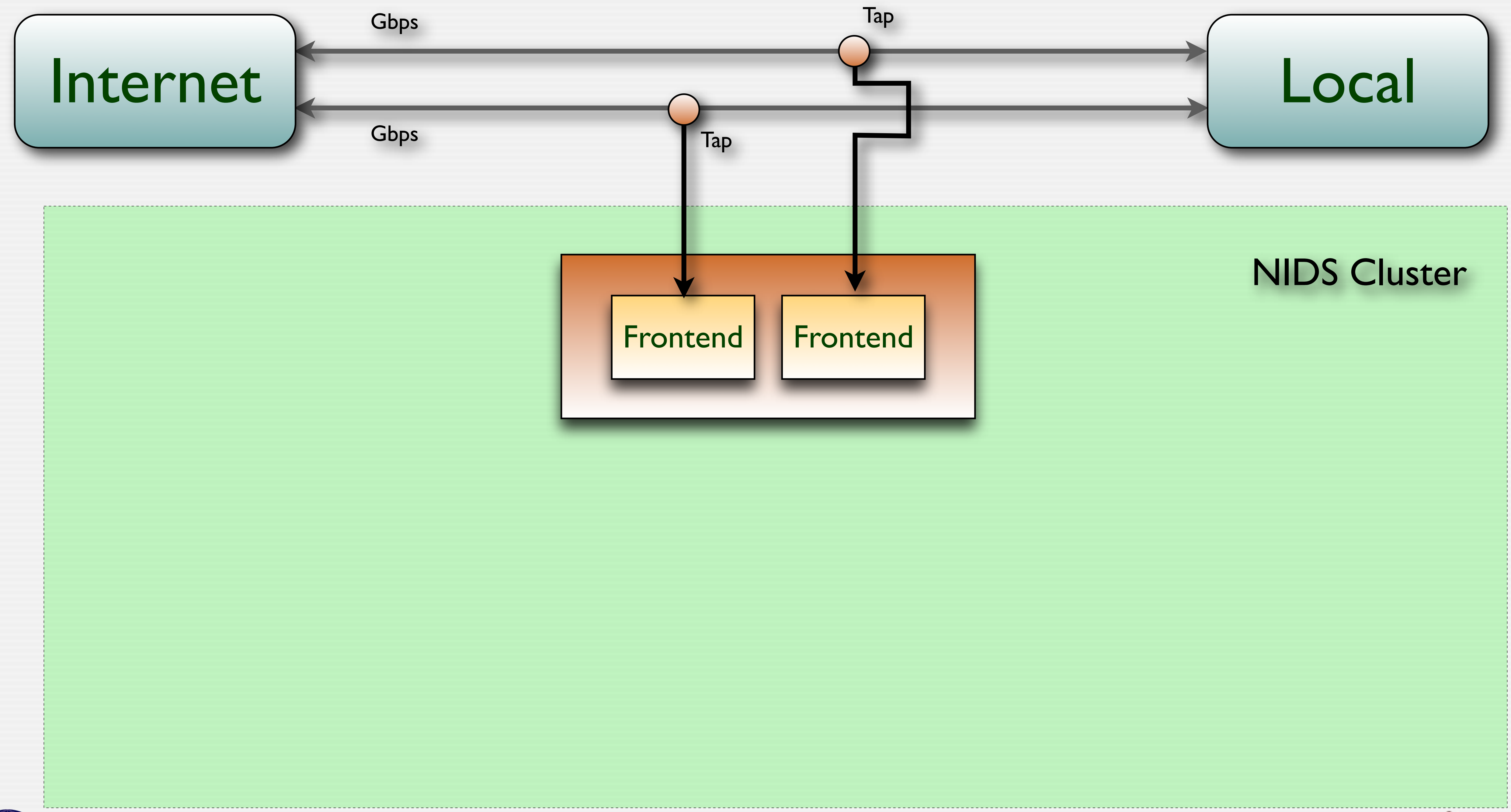
# Overview

- We do load-balancing with the "*NIDS Cluster*"
  - Use many boxes instead of one
  - Every box works on a slice of traffic
  - Correlate analysis to create the impression of a single system
- Most NIDS provide support for multi-system setups
- However, instances tend to work independent
  - Central manager collects alerts of independent NIDS instances
  - *Aggregates* results instead of *correlating* analysis
- NIDS cluster works *transparently* like a single NIDS
  - Gives same results as single NIDS would if it could analyze all traffic
  - Does not sacrifice detection accuracy

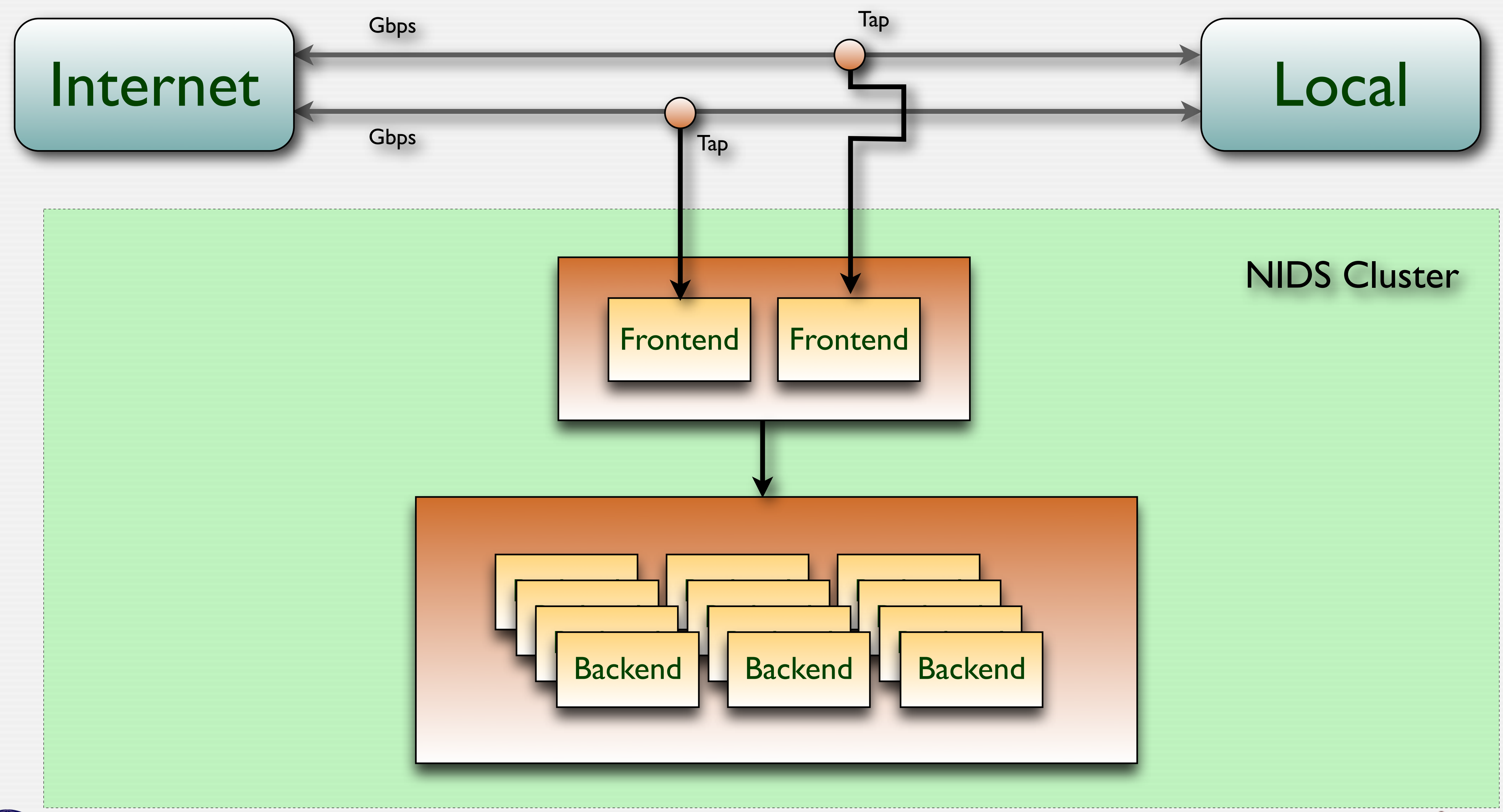
# Architecture



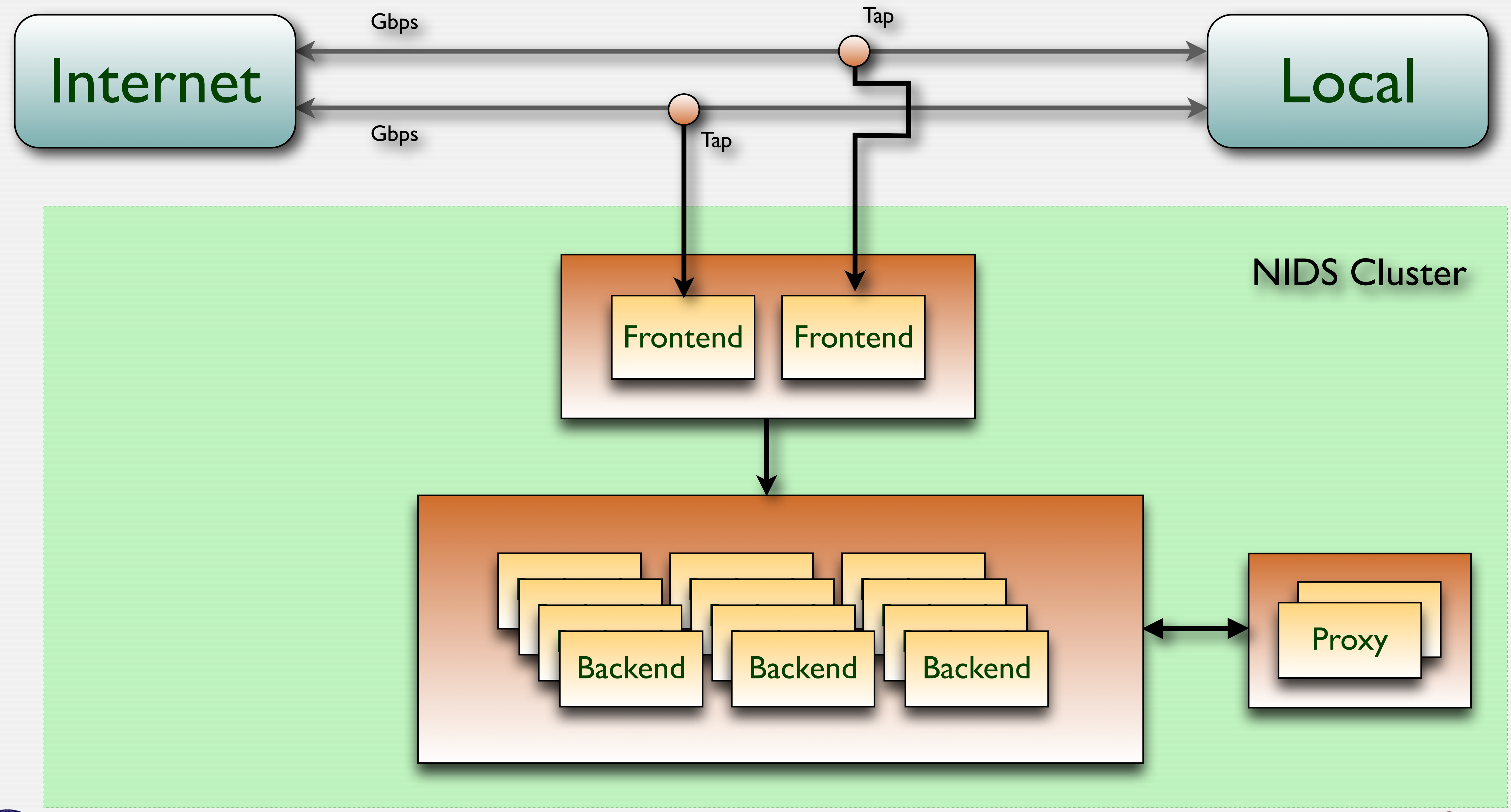
# Architecture



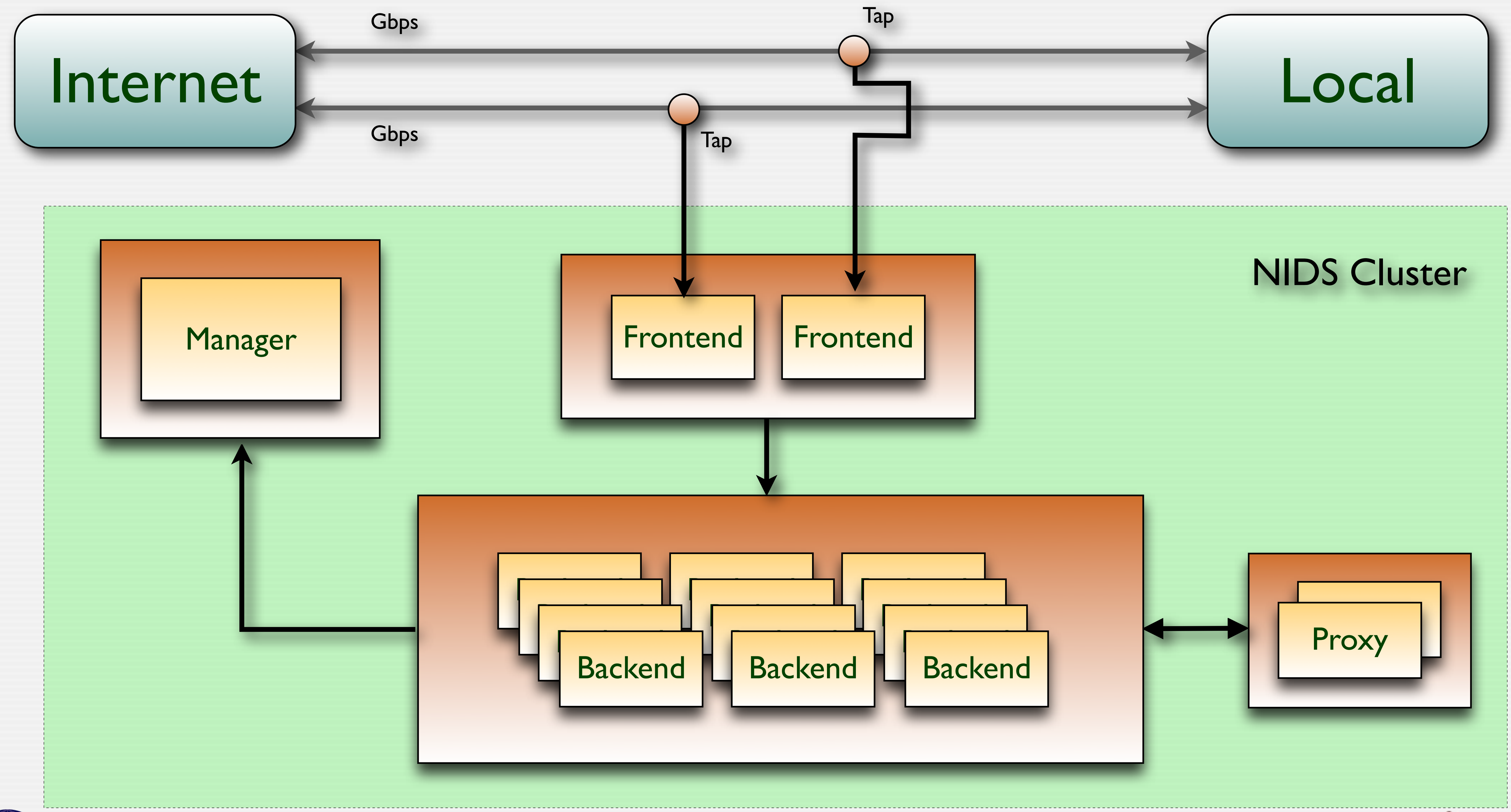
# Architecture



# Architecture



# Architecture



# Environments

- Initial target environment:
  Lawrence Berkeley National Laboratory
  - LBNL monitors 10 Gbps upstream link with the Bro NIDS
  - Setup evolved into many boxes running Bro independently for sub-tasks
  - Cluster prototype now running at LBNL with 1 frontend & 10 backends
- Further prototypes
  - University of California, Berkeley
    2 x 1 Gbps uplink, 2 frontends / 6 backends for 50% of the traffic
  - Ohio State University
    450 Mbps uplink, 1 frontend / 12 backends
  - IEEE Supercomputing Conference 2007
    Conference's 1 Gbps backbone / 10 Gbps "High Speed Bandwidth Challenge" netwo
- Goal: Replace operational security monitoring

# Challenges

*Main challenges when building the NIDS Cluster*

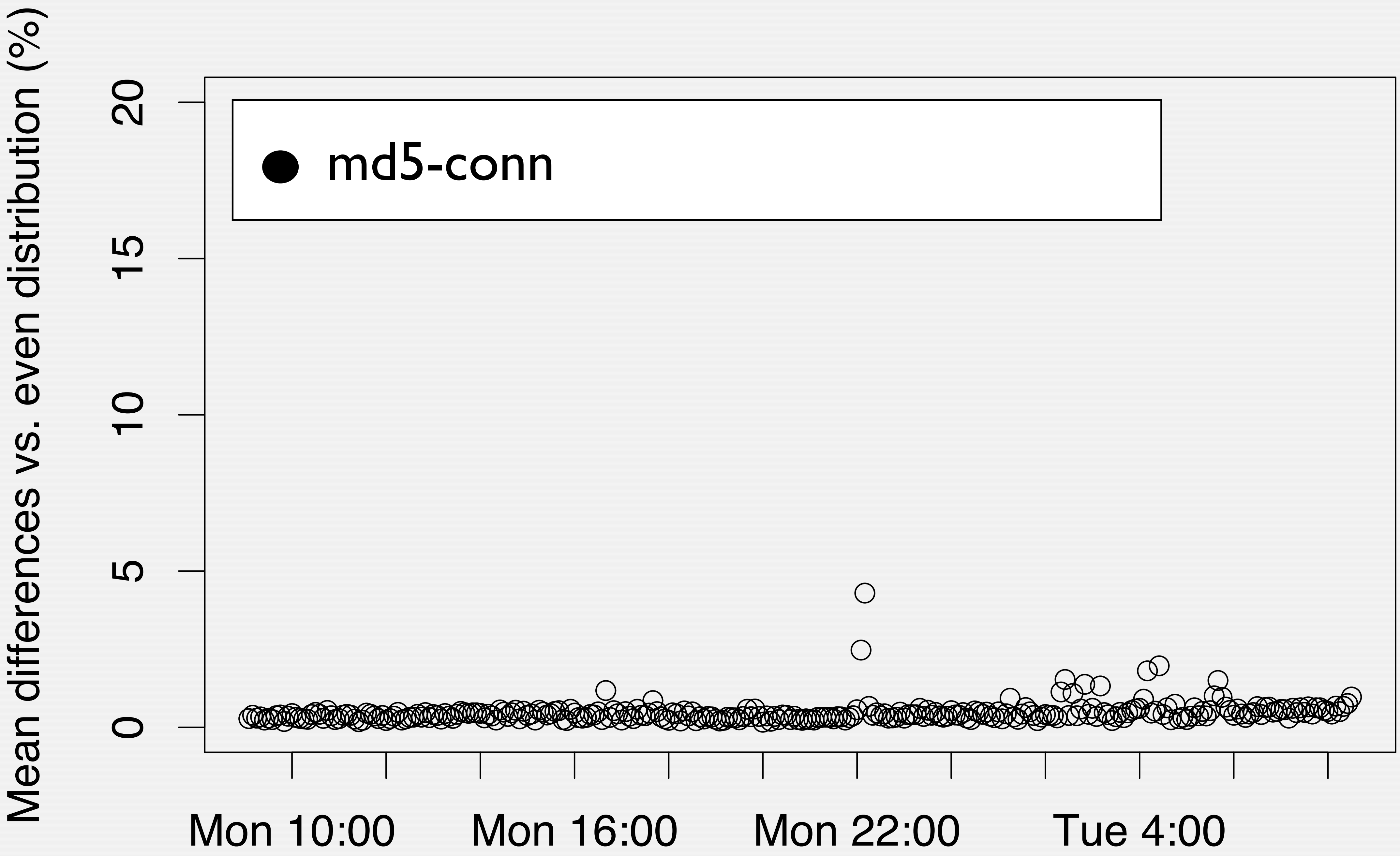
1. Distributing the traffic evenly while minimizing need for communication
2. Adapting the NIDS operation on the backend to correlate analysis with peers
3. Validating that the cluster produces sound results

# Distributing Load

# Distribution Schemes

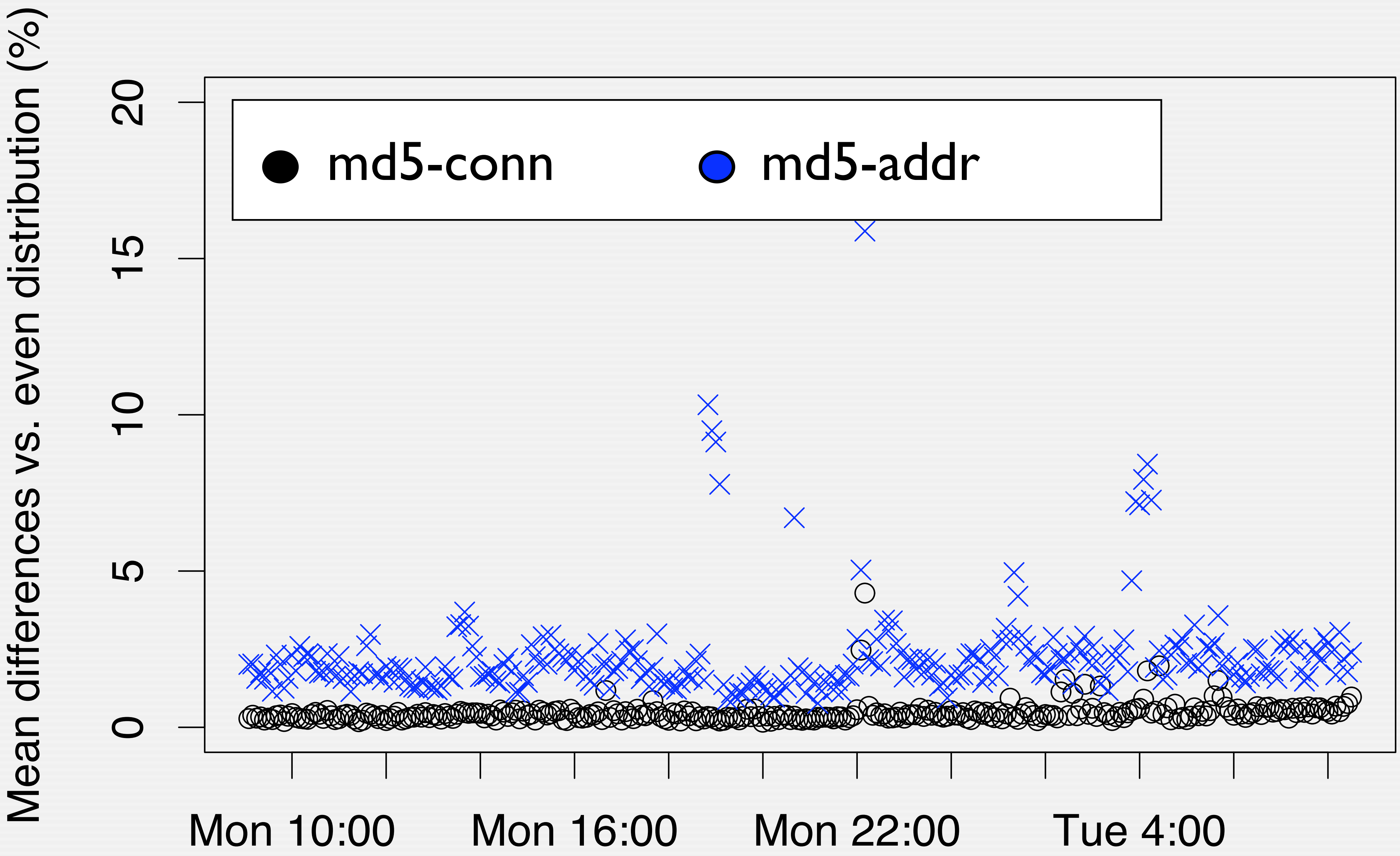
- Frontends need to pick a backend as destination
- Option 1: Route packets individually
  - Simple example: round-robin
  - Too expensive due to communication overhead (NIDS keep per-flow state)
- Option 2: Flow-based schemes
  - Send all packets belonging to the same flow to the same backend
  - Needs communication only for inter-flow analysis
- Simple approach: hashing flow identifiers
  - E.g., `md5(src-addr + src-port + dst-addr + dst-port) mod n`
  - Even simpler: `md5(src-addr + dst-addr) mod n`
  - Hashing is *state-less,* which reduces complexity and increases robustness
- But how well does hashing distribute the load?

# Simulation of Hashing Schemes




1 day of UC Berkeley campus TCP traffic (231M connections), n = 10

# Simulation of Hashing Schemes




1 day of UC Berkeley campus TCP traffic (231M connections), n = 10

# Cluster Frontends

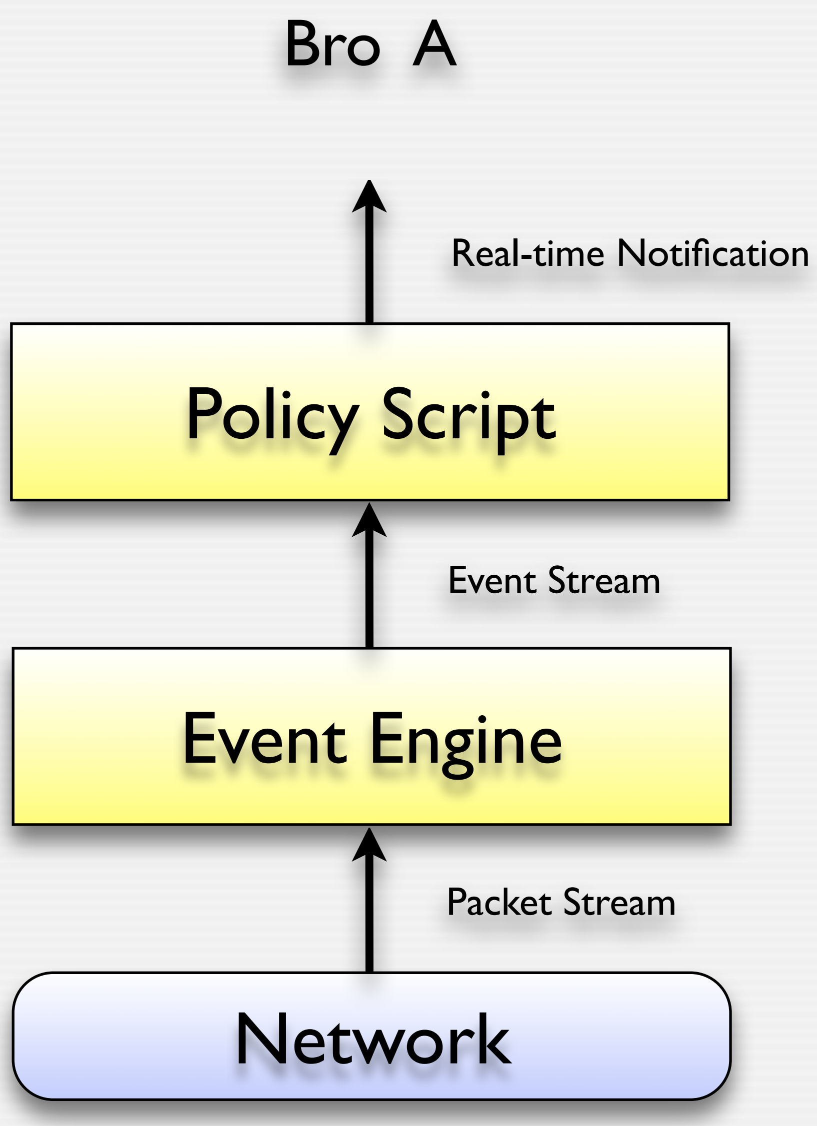
- We chose the address-based hash
  - Ports not always available (e.g., ICMP, fragments) & more complex to extract
  - Even with a perfect distribution, load is hard to predict
- Frontends rewrite MAC addresses according to hash
- Two alternative frontend implementations
  - In software with Click (SHA1)
  - In hardware with a prototype of Force-10's P10 appliance (XOR)
  - Working on cheaper hardware solutions

# Adapting the NIDS

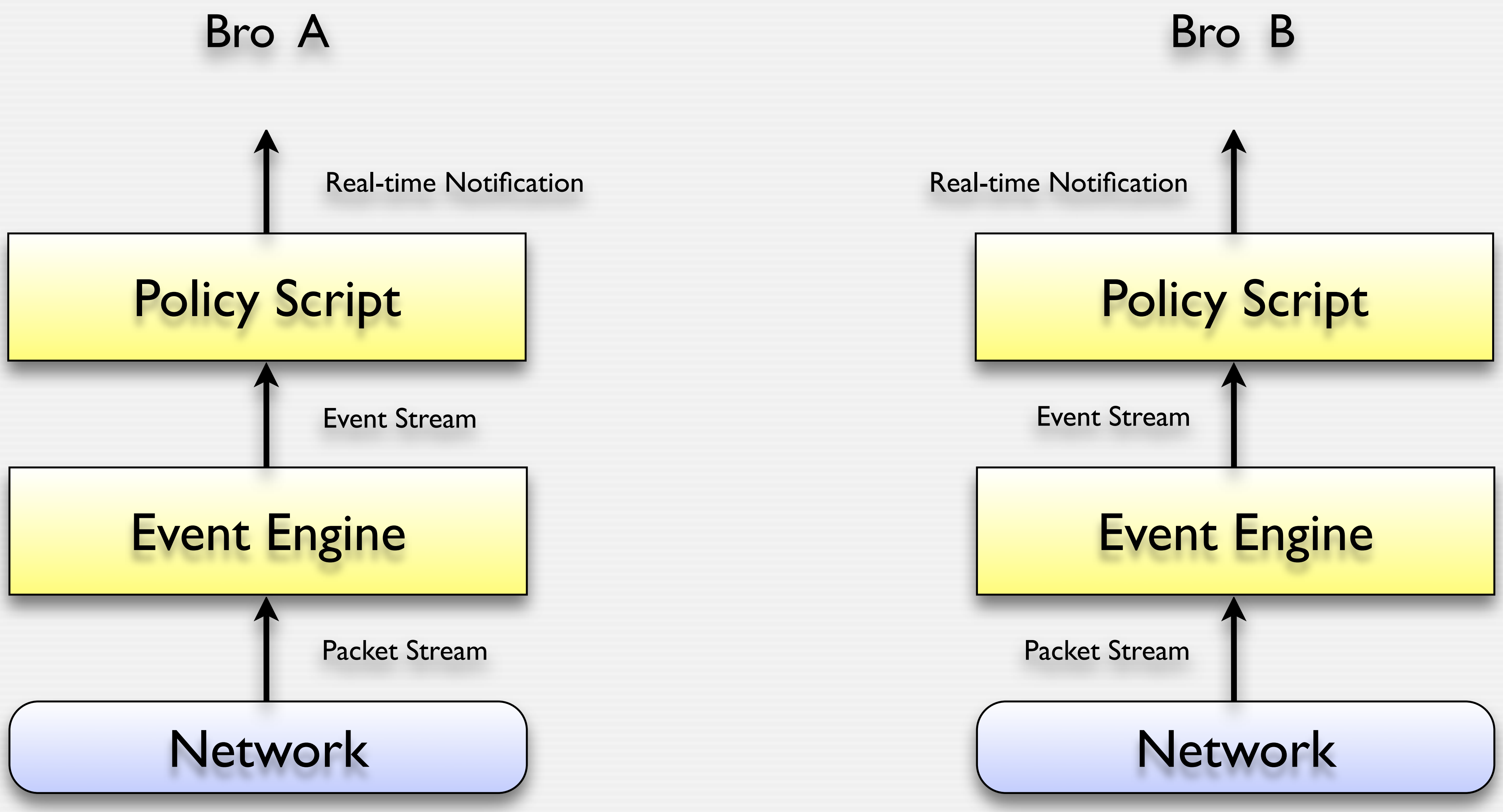
# Cluster Backends

- On the backends, we run the Bro NIDS
  - Bro is the NIDS used in our primary target environment LBNL
  - Bro already provides extensive, low-level communication facilities
- Bro consists of two layers
  - Core: Low-level, high-performance protocol analysis
  - Event-engine: Executes scripts which implement the detection analysis
- Observation: Core keeps only per-flow state
  - No need for correlation across backends
- Event-engine does all inter-flow analysis
  - The scripts needs to be adapted to the cluster setting

# Detour: Bro's Communication



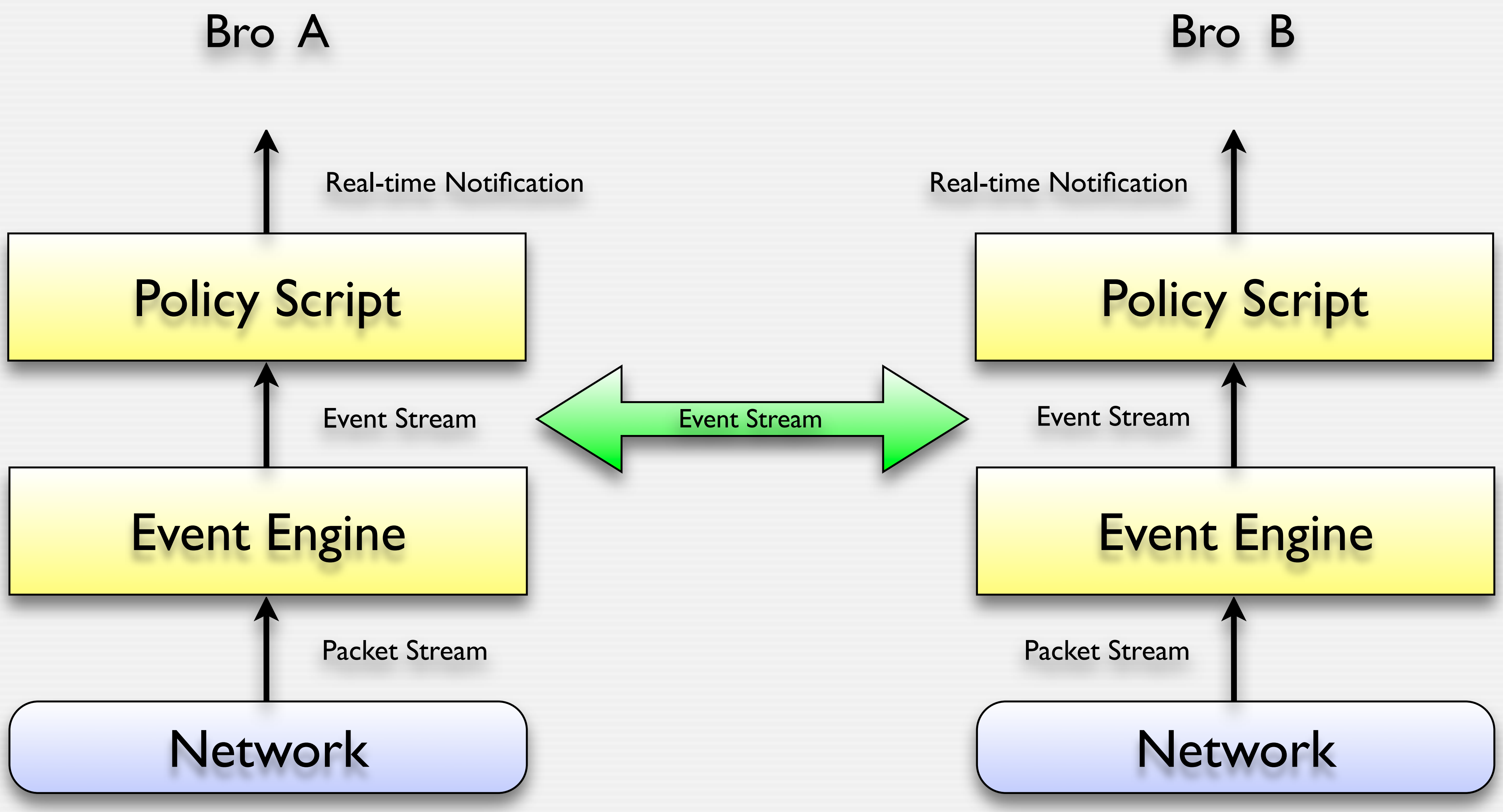
# Detour: Bro's Communication



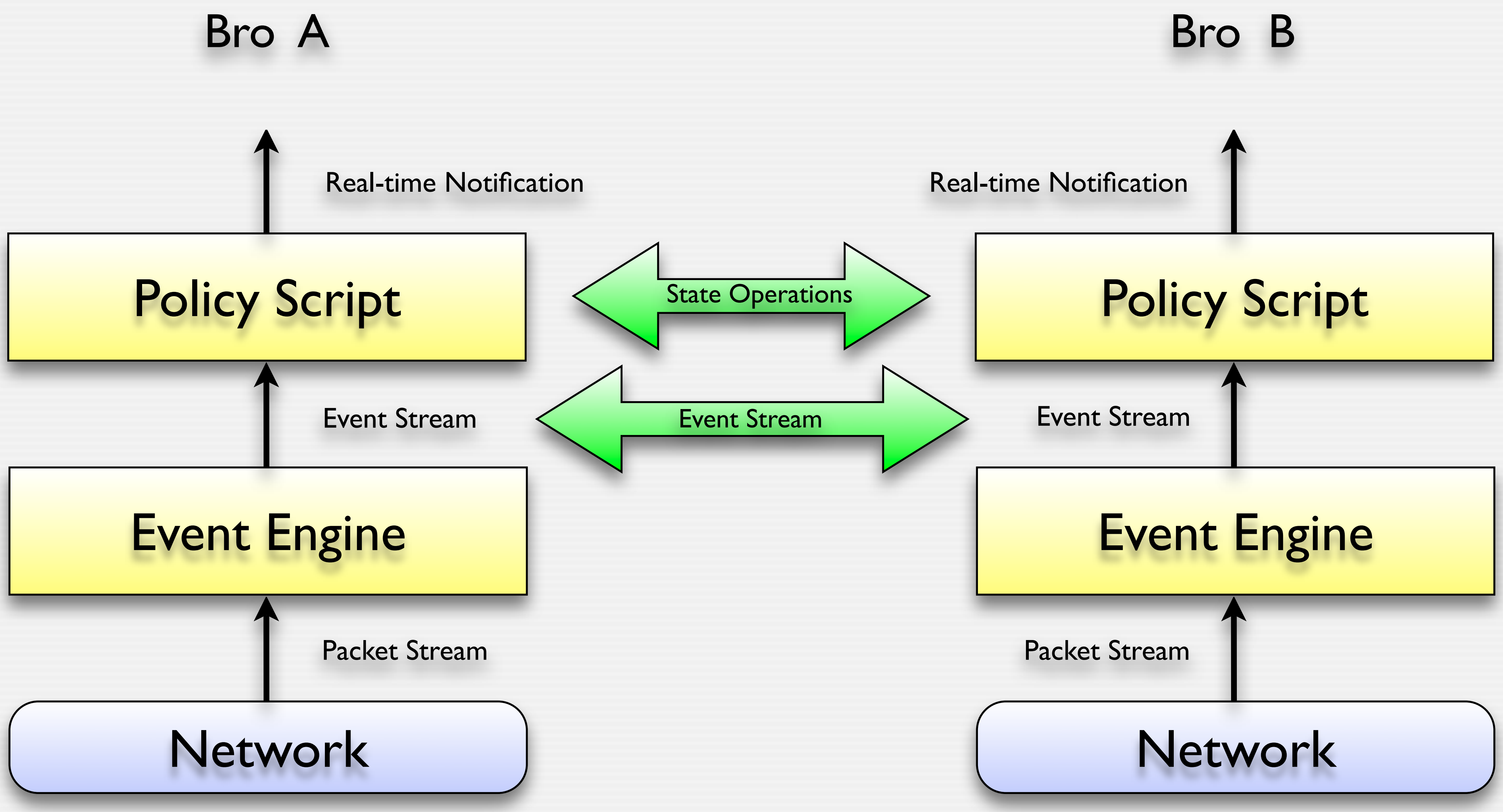

INTERNATIONAL COMPUTER SCIENCE INSTITUTE

BERKELEY LAB

# Detour: Bro's Communication



# Detour: Bro's Communication



# Adapting the Scripts ...

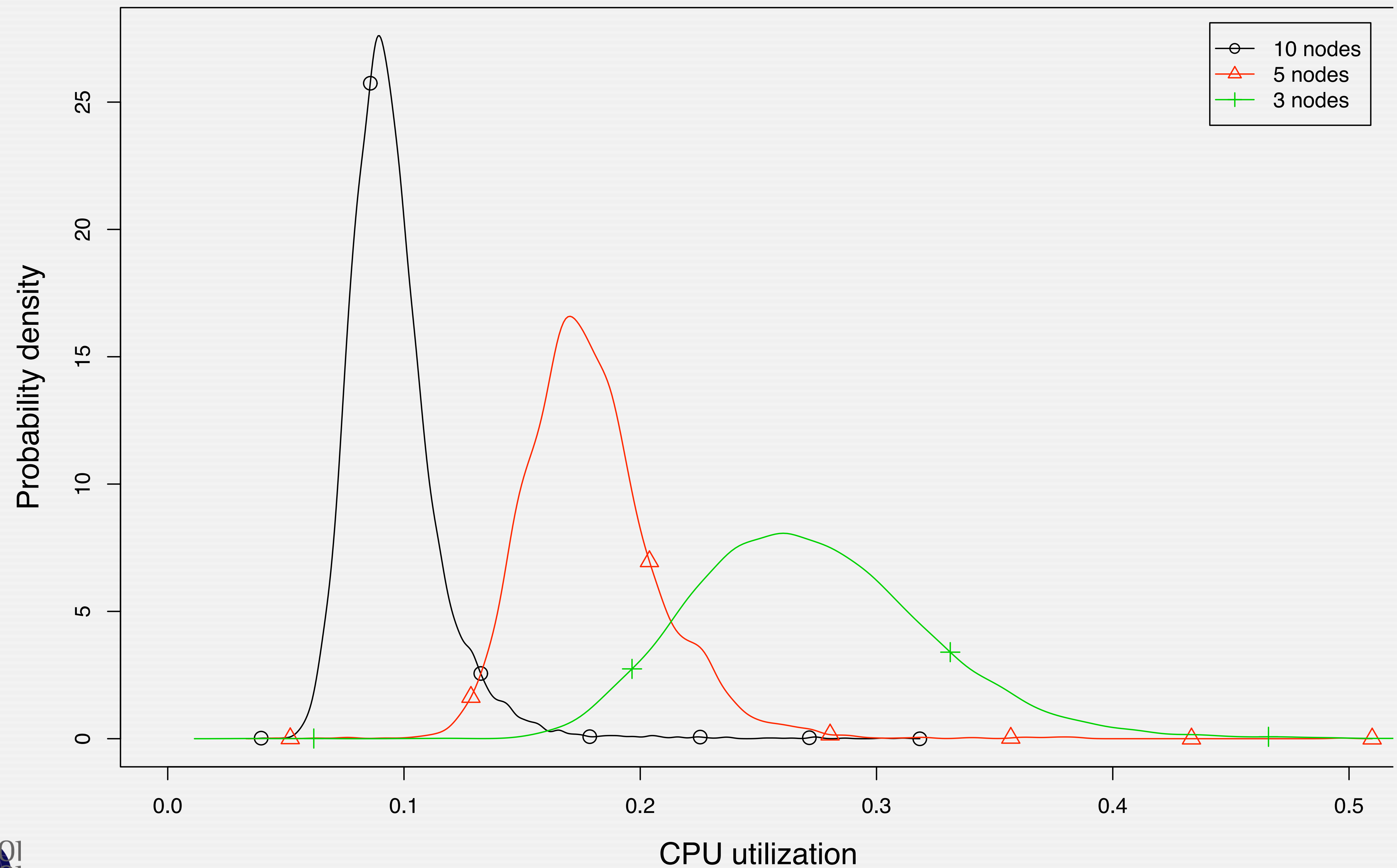
- Script language provides primitives to share state
  - Almost all state is kept in tables, which can easily be synchronized across peers
- Main task was identifying state related to inter-flow analysis
  - A bit cumbersome with 20K+ lines of script code ...
- Actually it was a bit more tricky ...
  - Some programming idioms do not work well in the cluster setting and needed to be fixed
  - Some trade-offs between gain & overhead exists are hard to assess
  - Bro's "loose synchronisation" introduces inconsistencies (which can be mitigated)
- Many changes to scripts and few to the core

# Validating the Cluster

# Accuracy

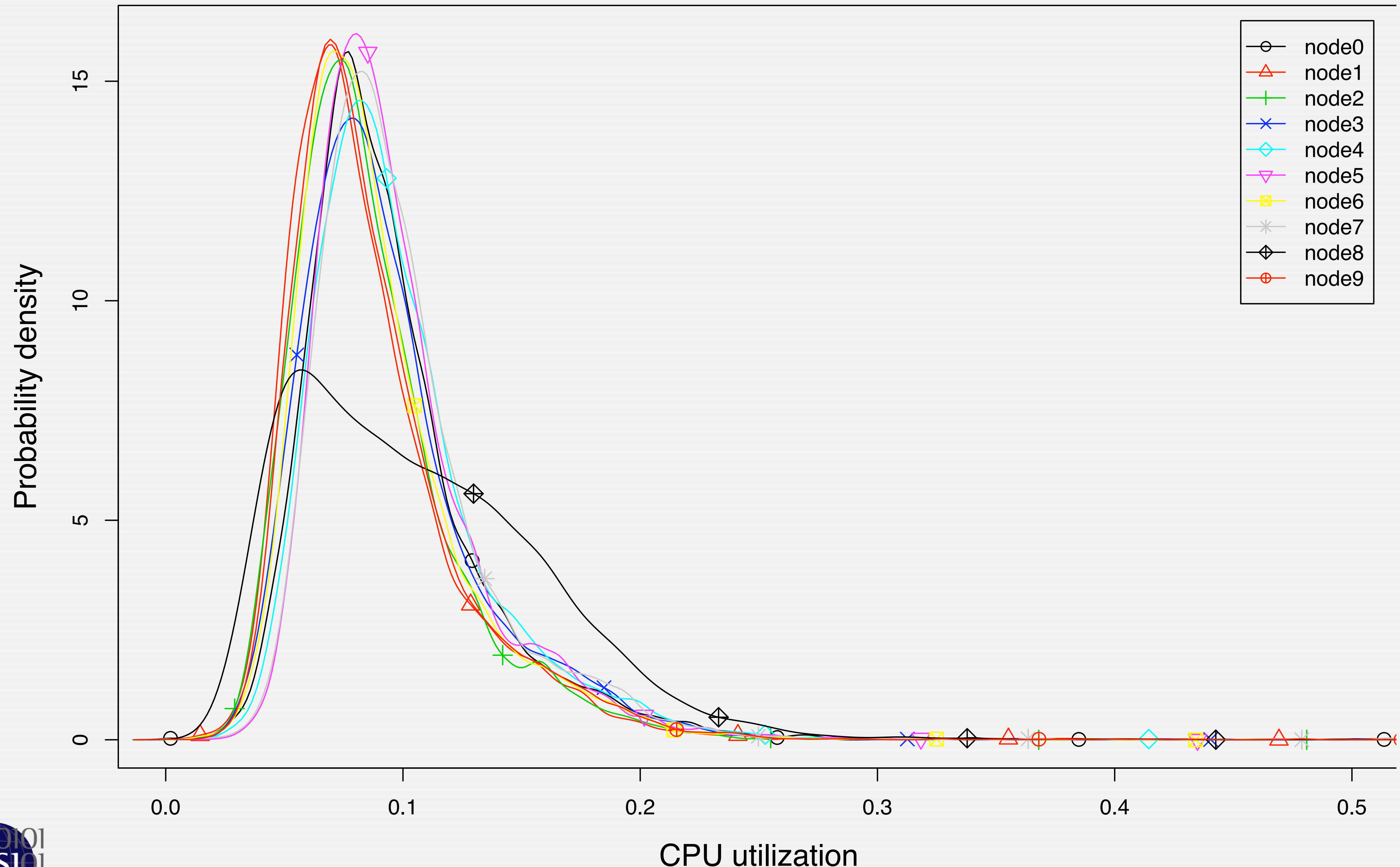
- Goal:  Cluster produces same result as a single system
- Compared the results of cluster vs. stand-alone setup
  - Captured a 2 hour trace at LBNL's uplink (~97GB, 134M pkts, 1.5 M host pairs)
  - Splitted the trace into slices and copied them to the cluster nodes
  - Setup the cluster to examine the slices just as if it would process live traffic
  - Compared output of the manager with the output of a single Bro instance on the trace
- Found excellent match for the alarms & logs
  - Cluster reported all alarms of the single instance as well
  - Slight differences in timing & context due to latency and synchronization semantics
  - Some artifacts of the off-line measurement setup

# Scaling of CPU



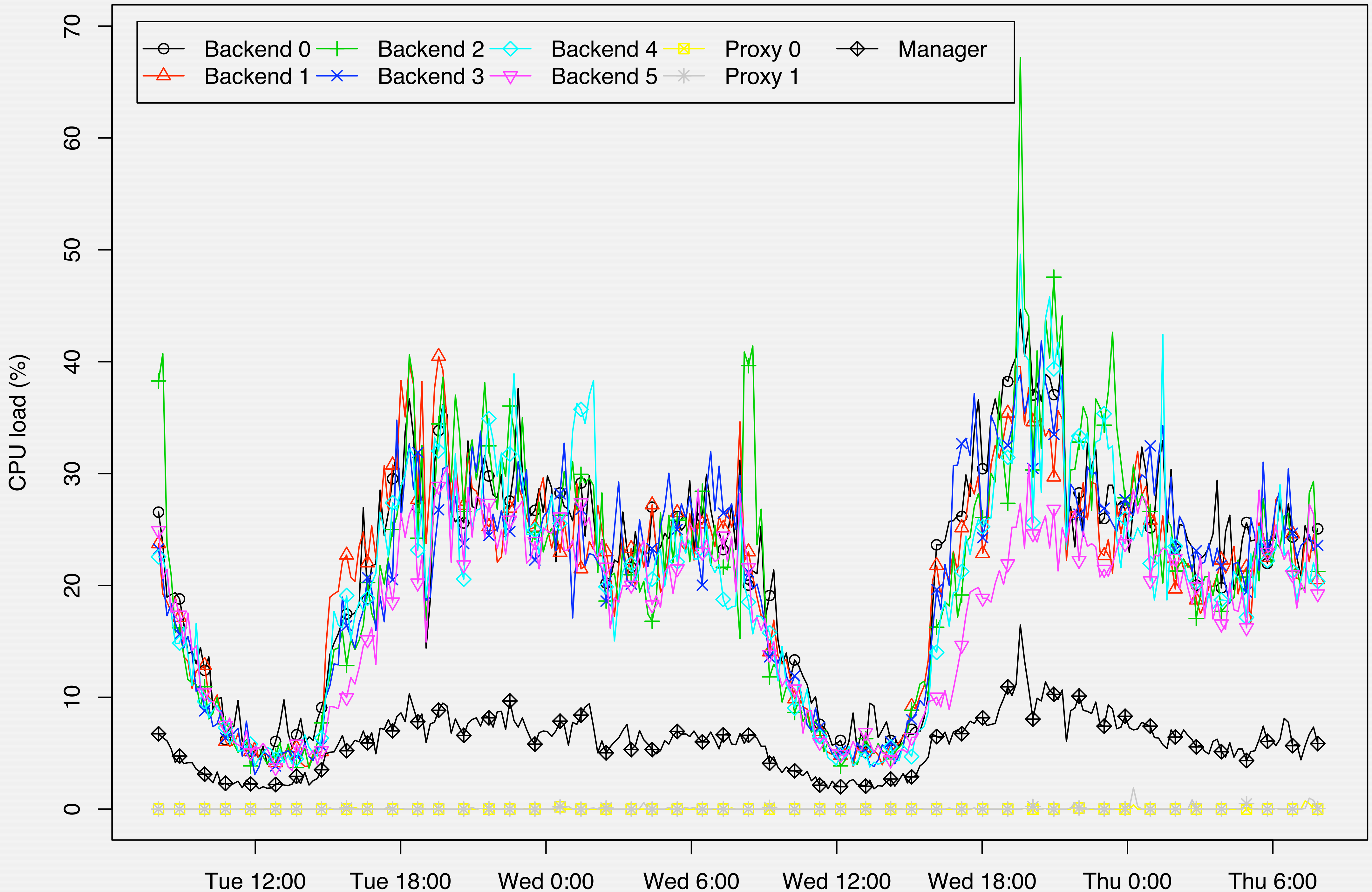

`ext. LBNL config, 2hr full trace, (~97GB, 134M pkts)`

# CPU Load per Node




10 backends, ext. LBNL config, 2hr full trace, (~97GB, 134M pkts)

# Load on Berkeley Campus




With 1 frontend = 50% of the total traffic

# Cluster Summary

- Cluster monitors Gbps networks on commodity hardware
  - Provides high-performance, stateful network intrusion detection
  - Correlates analysis across its nodes rather than just aggregating results
- When building the cluster we
  - Examined different load distribution schemes
  - Adapted an open-source NIDS to the cluster setting
  - Evaluated correctness & performance in a real-world setting
- Challenge was to build something which works
  - Less to lead into fundamentally new research directions
- Now in the process of making it production quality
- We will soon release the *Cluster Shell*

# The Cluster Shell

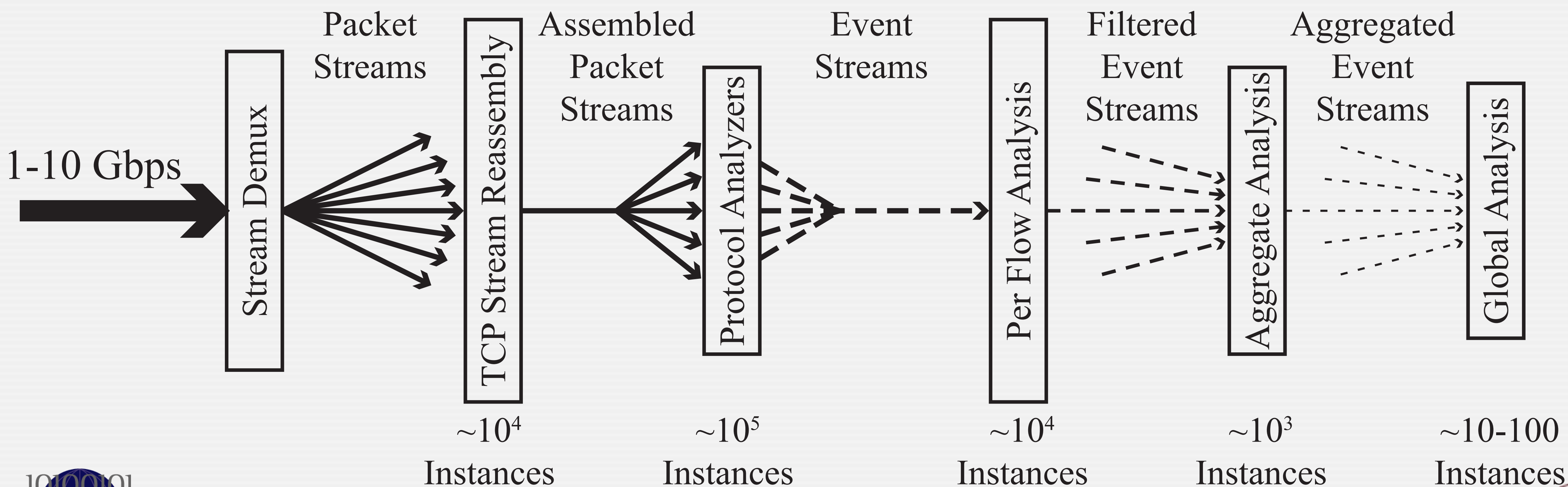
homer ~

```
robin@homer:~>cluster

Welcome to BroCluster 0.1

Type "help" for help.

[BroCluster] > status
Name         Type        Status      Host       Pid      Peers    Started
manager      manager     homer       running    3743     9        07 Oct 16:49:53
proxy-1      proxy       homer       running    3781     9        07 Oct 16:50:02
worker-2a    worker      lisa        running    86072    2        07 Oct 16:11:18
worker-2b    worker      lisa        running    86110    2        07 Oct 16:11:19
worker-3a    worker      bart        running    93591    2        07 Oct 16:11:21
worker-3b    worker      bart        running    93629    2        07 Oct 16:11:23
worker-4a    worker      maggie      running    92713    2        07 Oct 16:11:24
worker-4b    worker      maggie      running    92751    2        07 Oct 16:11:26
worker-5a    worker      abraham     running    17416    2        07 Oct 16:11:27
worker-5b    worker      abraham     running    17453    2        07 Oct 16:11:29
[BroCluster] > capstats
Host           mbps        kpps        (10s avg)
192.168.1.5    113.1       20.4
192.168.1.4    186.0       27.1
192.168.1.3    131.4       30.7
192.168.1.6    114.5       21.4
[BroCluster] > analysis
            dns is enabled    -  DNS analysis
            ftp is enabled    -  FTP analysis
      http-body is enabled    -  Analysis of HTTP bodies
    http-header is disabled   -  Analysis of HTTP headers
     http-reply is enabled    -  Server-side HTTP analysis
   http-request is enabled    -  Client-side HTTP analysis
           scan is enabled    -  Scan detection
           smtp is enabled    -  SMTP analysis
[BroCluster] >
```

# Parallel Analysis Inside *One* Box

# Potential

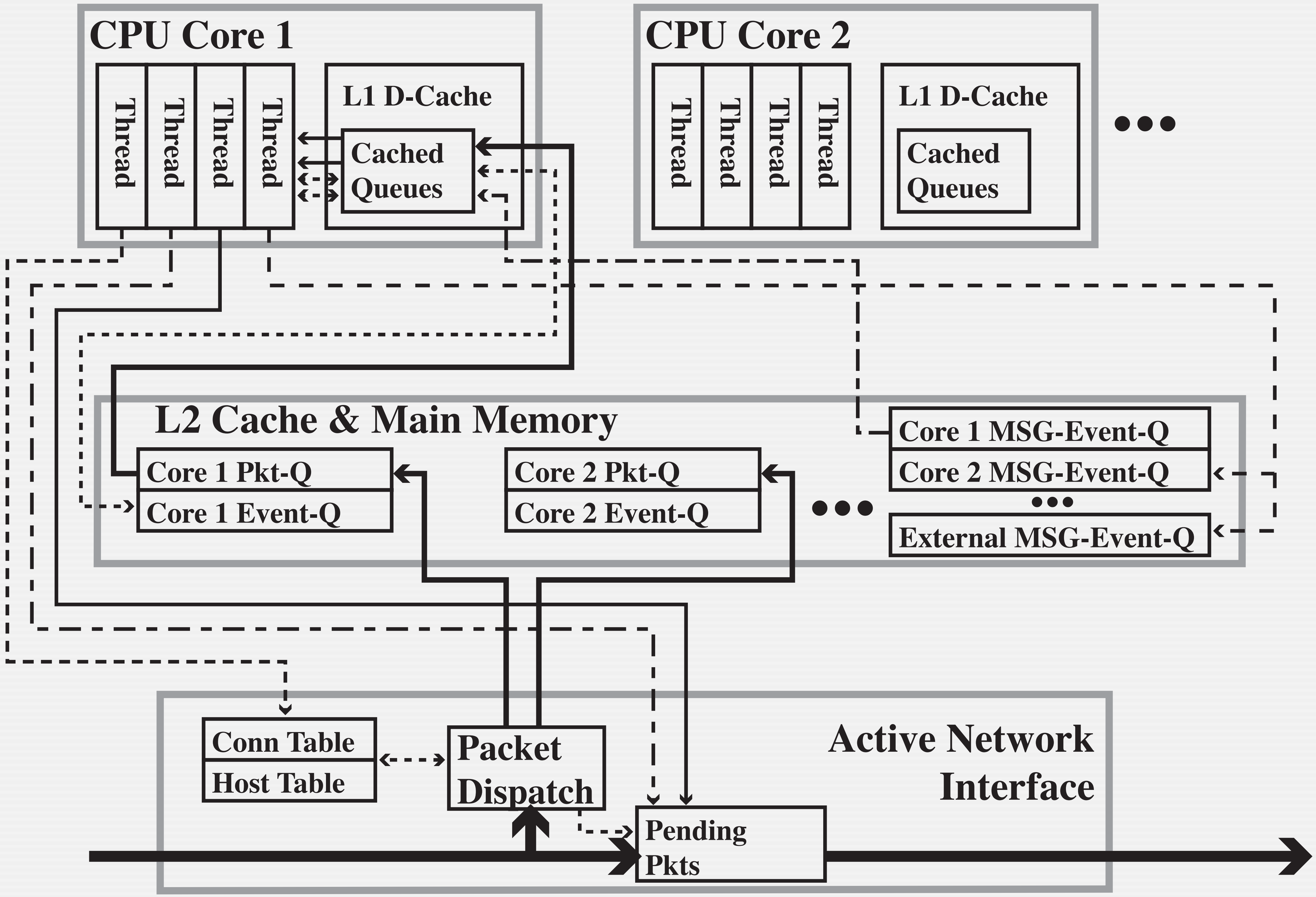
- Observation
  - Much of the processing of a typical NIDS instance can be done in parallel
  - However, existing systems do not exploit the potential
- Example:  Bro NIDS



# Commodity Hardware

- Multi-thread/multi-core CPU provide necessary power
  - Inexpensive commodity hardware
  - *Aggregated* throughput does in fact still follow Moore's law
- Need to structure applications in highly parallel fashion
  - Do not get the performance gain out of the box
  - Need to structure processing into separate low-level threads
- Work in progress; we want to address
  - Intrusion *prevention* functionality
  - Exchange of state between threads for global analysis
  - Yet minimize inter-thread communication
  - Factor in memory locality (within one core / across several cores)
  - Provide performance *debugging* tools

# Proposed Architecture



# Active Network Interface

- Only non-commodity components currently
  - Prototype to be based on NetFPGA platform ($2000)
  - Commodity hardware might actually be suitable later
    (E.g., Sun's Niagara 2 has 8 CPU cores plus 2 directly attached 10GE controller!)
- Thread-aware Routing
  - ANI copies packet directly into thread's memory (cache)
  - ANI keeps per-flow table of routing decisions
  - Dispatcher thread takes initial routing decision per flow
- Selective packet forwarding
  - ANI holds packets until it gets the clearance (might use caching per e.g. flow/ip)
- Normalization

# Parallelized Network Analysis

- Architecturally-aware Threading
  - Need to identify the right granularity for threads
  - Protocol analysis consists of fixed blocks of functionality
  - Event processing needs to preserve *temporal* order
    → Multiple independent event queues (e.g., one per core)
- Scalable Inter-thread Communication
  - Can use shared memory
  - Need to consider nonuniformities in system's cache hierarchy
  - Potentially restructure detection algorithms to minimize communication (e.g., loosing semantics via probabilistic algorithms)
- Prevention Functionality
  - Only forward packet once all events are processed
- Evaluation, profiling & debugging
  - Race conditions & memory access patterns

ICSI INTERNATIONAL COMPUTER SCIENCE INSTITUTE

BERKELEY LAB

# Going Further: Custom Hardware

- Goal: *custom platform for highly parallel, stateful network analysis*
- Custom hardware (e.g., FPGAs) is ideal for parallel tasks
- Expose the parallelism and map it to hardware
- We can identify three types of functionality in Bro
  - Fixed function blocks →Handcraft (e.g., robust reassembly)
  - Protocol analyzers → Use BinPAC with new backend
  - Policy scripts → Compile into parallell computation model
- Envision using MIT's *Transactor* model
  - Many small self-contained units communicating via message queues
- Ambitious but highly promising
  - Generic network analysis beyond network intrusion detection

# Summary & Outlook

# The Bro NIDS

- Bro is one of the most powerful NIDS available
  - Open-source and runs on commodity hardware
  - While primarily a research system, it is well suited for operational use
  - Deployed at large universities & labs
- Working a various extensions
  - Interactive Cluster Shell for easy installation/operation of a Bro Cluster
  - New analyzers for
    NetFlow, BitTorrent, SIP, XML w/ XQuery support, SSL (rewritten)
  - Time Machine interface (see http://www.net.t-labs.tu-berlin.de/research/tm)
- Current Work
  - Turning cluster prototype into production
  - Multi-core support
  - Inter-site Data sharing

# Thanks for your attention!

Robin Sommer

*International Computer Science Institute &*
*Lawrence Berkeley National Laboratory*

robin@icsi.berkeley.edu
http://www.icir.org

# DPD

# Advanced DPD Applications

- Turning off analyzers if it's not "their" protocol
  - Fundamental question: when to decide it's not "theirs"?
  - Analyzers report `ProtocolViolation` if they can't parse basic structure
  - Policy script can then decide whether to indeed disable analyzer
- Reporting protocols found on non-standard ports
  - Reports `ProtocolFound` and `ServerFound`
  - Further identify applications on top of HTTP (e.g., Gnutella, SOAP, Squid)
  - Easy to extend by adding more patterns

# Analyzing FTP Data Connections

- FTP data cannot be analyzed by port-based NIDSs
- Bro has a *File Analyzer*
  - Determines file-type (via libmagic)
  - Checks for malware (via libclamav)
- With DPD, FTP can use the File Analyzer
  - Parses control connection to learn about upcoming FTP data
  - File Analyzer is inserted into analyzer tree when connection is seen

# Example: FTP Data Analysis

```
xxx.xxx.xxx.xxx/2373 > xxx.xxx.xxx.xxx/5560 start
response (220 Rooted Moron Version 1.00 4 WinSock ready...)
USER ops (logged in)
SYST (215 UNIX Type: L8)
[...]
LIST -al (complete)
TYPE I (ok)
SIZE stargate.atl.s02e18.hdtv.xvid-tvd.avi (unavail)
PORT xxx,xxx,xxx,xxx,xxx,xxx (ok)
STOR stargate.atl.s02e18.hdtv.xvid-tvd.avi, NOOP (ok)
ftp-data video/x-msvideo `RIFF (little-endian) data, AVI'
[...]
response (226 Transfer complete.)
[...]
QUIT (closed)
```

# Example: Bots

```
Detected bot-servers:
IP1 - ports 9009,6556,5552 password(s) <none> last 18:01:56
 channel #vec:
 topic ".asc pnp 30 5 999 -b -s|.wksescan 10 5 999 -b -s|
[...]"
 channel #hv:
 topic ".update http://XXX/image1.pif f'', password(s) XXX"
[...]
Detected bots:
IP2 - server IP1 usr 2K-8006 nick [P00|DEU|59228]
IP4 - server IP1 usr XP-3883 nick [P00|DEU|88820]
[...]
```

# Example: Protocol Signature

```
signature dpd_http_client {
  ip-proto == tcp
  payload /^[[:space:]]*(GET|HEAD|POST)[[:space:]]*/
  tcp-state originator
}

signature dpd_http_server {
  ip-proto == tcp
  payload /^HTTP\/[0-9]/
  tcp-state responder
  requires-reverse-signature dpd_http_client
  enable "http"
}
```

# *Recent Developments (3)* binpac: A "yacc" for Writing Application Protocol Parsers

# Writing Analyzers Manually

- Protocol analyzers are central to any NIDS
- Writing such an analyzer appears straight-forward
  - Take the protocol-specification and code a parser in your favorite language
- However, in practice this is *really* tedious
  - Protocols are complex (e.g., HTTP has pipelining, chunking, MIME, etc.)
  - Protocol specifications are incomplete
  - Analyzer must robust (abundant "crud"; attacker can craft traffic)
  - Analyzer must be efficient (handling 10000s of connections in real-time)
  - Analyzer cannot reused (tends to be tightly coupled to app environment)
- Proof: severe vulnerabilities in existing analyzers
  - Witty propagated through 12,000 deployments of ISS security software

# The yacc Approach

- Problems caused by significant lack of abstraction
  - In the programming language community, nobody write parsers manually
  - Parser generators turn grammar-plus-semantics into low-level code
- `binpac:` a *yacc* for network protocols
  - Declarative language and its compiler
  - Translates protocol specification into C++ code for parsing
- Primary goals
  - Relieve user from low-level details
  - Generate parsers which are as efficient as manually coded ones
  - Support reuse of analyzers across applications

# Why not just use *yacc*?

- Network protocols are not programming languages
- Syntax
  - Variable-length arrays (e.g., `Content-length: 42`)
  - Selection among grammar rules (e.g., DNS types for differerent RRs)
  - Byte encoding (e.g., byte-order)
- Input model
  - Analyzers require incremental, in-parallel processing
- Robustness
  - Analyzers must detect and recover from parsing errors

# Small Example - HTTP Excerpt

```
type HTTP_Request = record {
  request:HTTP_RequestLine;
  msg:    HTTP_Message(BODY_MAYBE);
};

type HTTP_RequestLine = record {
  method:   HTTP_TOKEN;
  :         HTTP_WS;
  uri:      HTTP_URI;
  :         HTTP_WS;
  version:  HTTP_Version;
} &oneline;
```

```
type HTTP_Message(b: ExpectBody)= record {
  headers:        HTTP_Headers;
  body_or_not:    case b of {
      BODY_NOT_EXPECTED -> none: empty;
      default -> body: HTTP_Body(b);
  };
};

type HTTP_Headers = HTTP_Header[]
  &until($input.length() == 0);

type HTTP_HEADER_NAME = RE/|([^: \t]+:)/;
type HTTP_Header = record {
  name:             HTTP_HEADER_NAME;
  :                 HTTP_WS;
  value:            bytestring &restofdata;
} &oneline;
```

# (Almost) Full HTTP Analyzer

```
analyzer HTTP withcontext {  # members of $context
    connection: HTTP_Conn;
    flow:       HTTP_Flow;
};
enum DeliveryMode {
    UNKNOWN_DELIVERY_MODE,
    CONTENT_LENGTH,
    CHUNKED,
};
# Regular expression patterns
type HTTP_TOKEN = RE/[^()<>@,;:\\"\/\[\]?={} \t]+/;
type HTTP_WS    = RE/[ \t]*/;
extern type BroConn;
extern type HTTP_HeaderInfo;
%header{
    // Between %.*{ and %} is embedded C++ header/
code
    class HTTP_HeaderInfo {
    public:
        HTTP_HeaderInfo(HTTP_Headers *headers) {
          delivery_mode = UNKNOWN_DELIVERY_MODE;
          for ( int i = 0; i < headers->length(); +
+i ) {
            HTTP_Header *h = (*headers)[i];
            if ( h->name() == "CONTENT-LENGTH" ) {
              delivery_mode = CONTENT_LENGTH;
              content_length = to_int(h->value());
            } else if ( h->name() == "TRANSFER-
ENCODING"
                    && has_prefix(h->value(),
"CHUNKED") ) {
              delivery_mode = CHUNKED;
            }
          }
        }
        DeliveryMode delivery_mode;
        int content_length;
    };
%}

# Connection and flow
connection HTTP_Conn(bro_conn: BroConn) {
    upflow = HTTP_Flow(true);  downflow = HTTP_Flow
(false);
};
```

```
flow HTTP_Flow(is_orig: bool) {
    flowunit = HTTP_PDU(is_orig)
                     withcontext(connection, this);
};


# Types
type HTTP_PDU(is_orig: bool) = case is_orig of {
    true  -> request: HTTP_Request;
    false -> reply:   HTTP_Reply;
};
type HTTP_Request = record {
    request:    HTTP_RequestLine;
    msg:        HTTP_Message;
};
type HTTP_Reply = record {
    reply:      HTTP_ReplyLine;
    msg:        HTTP_Message;
};


type HTTP_RequestLine = record {
    method:     HTTP_TOKEN;
    :           HTTP_WS;    # an anonymous field has
no name
    uri:        RE/[[:alnum:][:punct:]]+/;
    :           HTTP_WS;
    version:    HTTP_Version;
} &oneline, &let {
    bro_gen_req: bool = bro_event_http_request(
        $context.connection.bro_conn,
        method, uri, version.vers_str);
};
type HTTP_ReplyLine = record {
    version:    HTTP_Version;
    :           HTTP_WS;
    status:     RE/[0-9]{3}/;
    :           HTTP_WS;
    reason:     bytestring &restofdata;
} &oneline, &let {
    bro_gen_resp: bool = bro_event_http_reply(
        $context.connection.bro_conn,
        version.vers_str, to_int(status), reason);
};
```

```
type HTTP_Version = record {
    :           "HTTP/";
    vers_str:   \RE/[0-9]+\.[0-9]+/;
};

type HTTP_Message = record {
    headers:    HTTP_Headers;
    body:       HTTP_Body(HTTP_HeaderInfo(headers));
};
type HTTP_Headers = HTTP_Header[] &until
($input.length() == 0);
type HTTP_Header = record {
    name:       HTTP_TOKEN;
    :           ":";
    :           HTTP_WS;
    value:      bytestring &restofdata;
} &oneline, &let {
    bro_gen_hdr: bool = bro_event_http_header(
        $context.connection.bro_conn,
        $context.flow.is_orig, name, value);
};
type HTTP_Body(hdrinfo: HTTP_HeaderInfo) =
        case hdrinfo.delivery_mode of {
    CONTENT_LENGTH -> body: bytestring &chunked,
                        &length =
hdrinfo.content_length;
    CHUNKED        -> chunks: HTTP_Chunks;
    default        -> other: HTTP_UnknownBody;
};
type HTTP_Chunks = record {
    chunks:     HTTP_Chunk[] &until
($element.chunk_length == 0);
    headers:    HTTP_Headers;
};
type HTTP_Chunk = record {
    len_line:   bytestring &oneline;
    data:       bytestring &chunked, &length =
chunk_length;
    opt_crlf:   case chunk_length of {
        0       -> none: empty;
        default -> crlf: bytestring &oneline;
    };
} &let {
    chunk_length: int = to_int(len_line, 16);  # in
hexadecimal
};
```

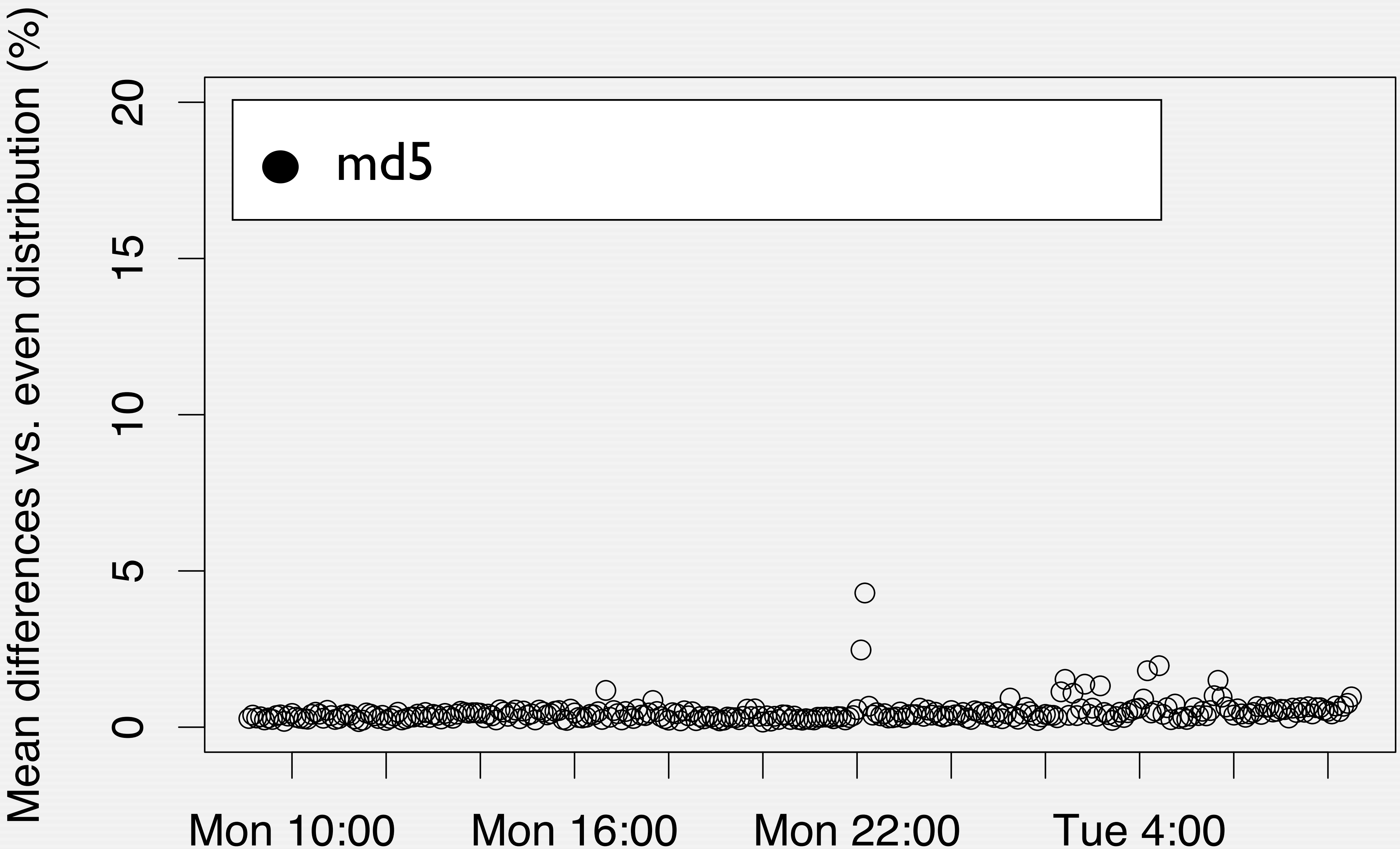
*(Excludes MIME formatting and escape sequences.)*

# `binpac` in Bro 1.2

- `binpac` ships as part of the Bro distribution
- Includes `binpac` analyzers for several protocols
  - HTTP, DNS, SUN/RPC, RPC Portmapper, CIFS, DCE/RPC, NCP
  - `bro --use-binpac` enables `binpac` version for existing analyzers
- `binpac` will be default choice for new analyzers
- Analyzers already begin to be reused

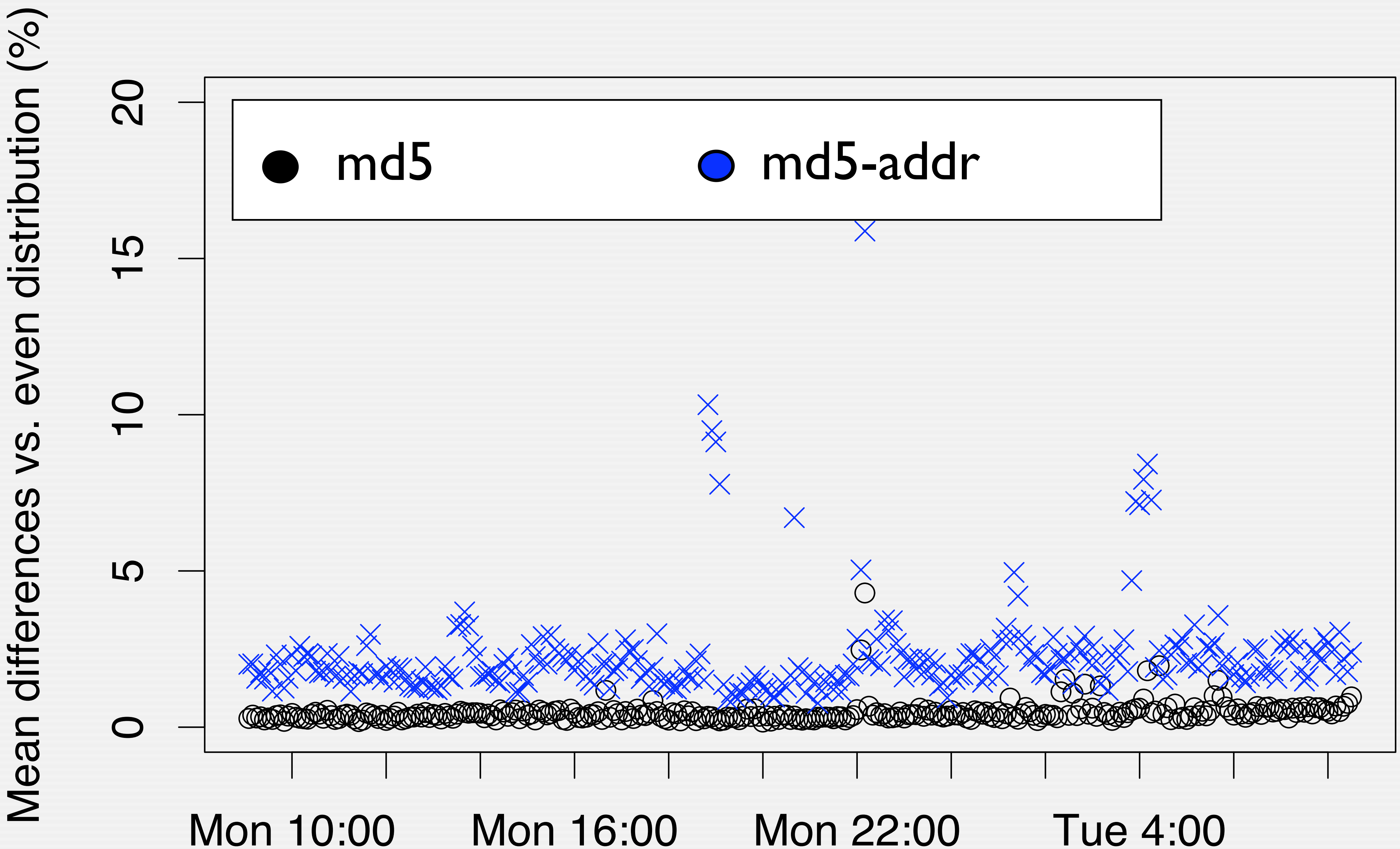
# Cluster

# Simulation of Hashing Schemes



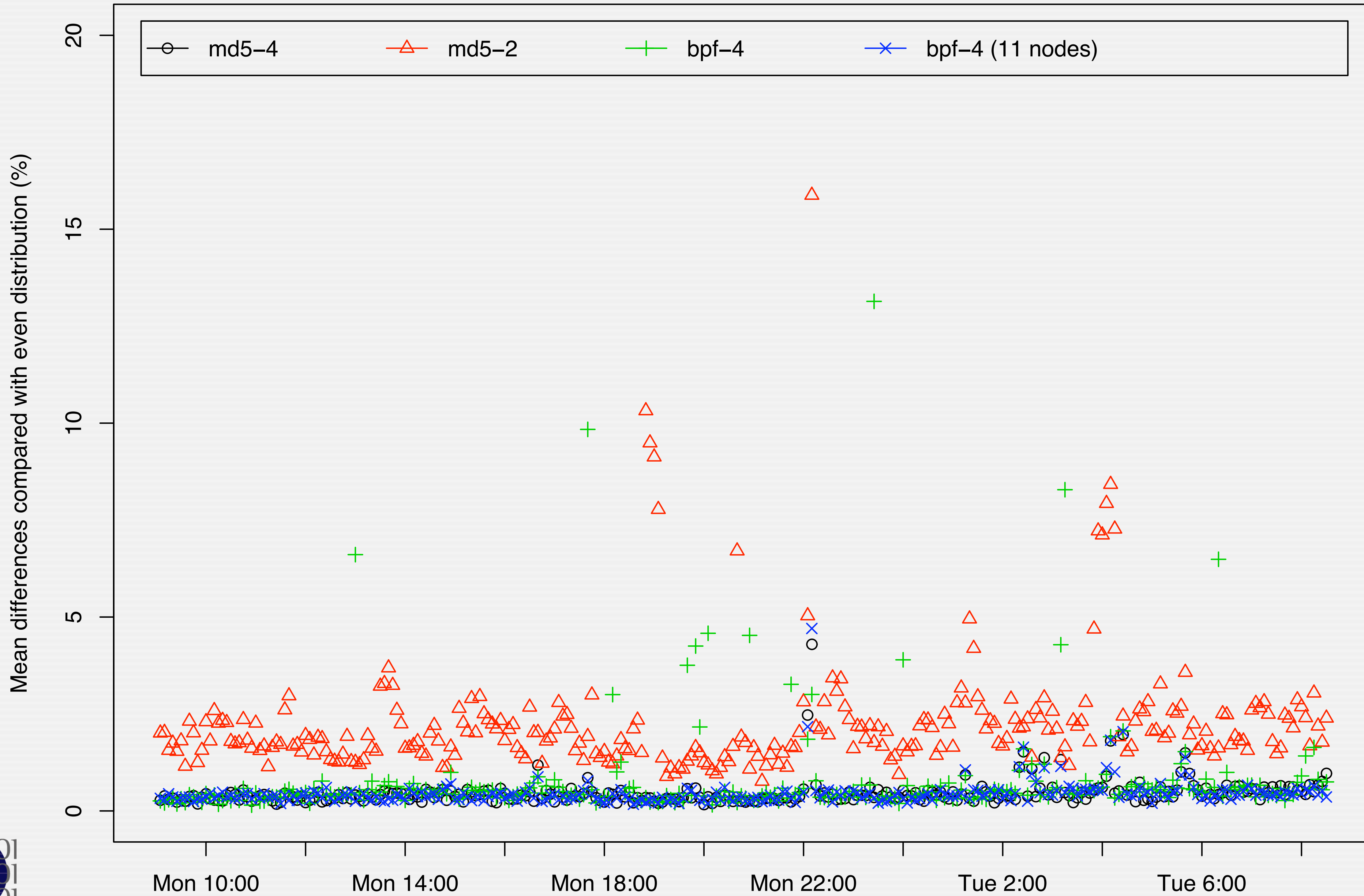

1 day of UC Berkeley campus TCP traffic (231M connections), n = 10

# Simulation of Hashing Schemes

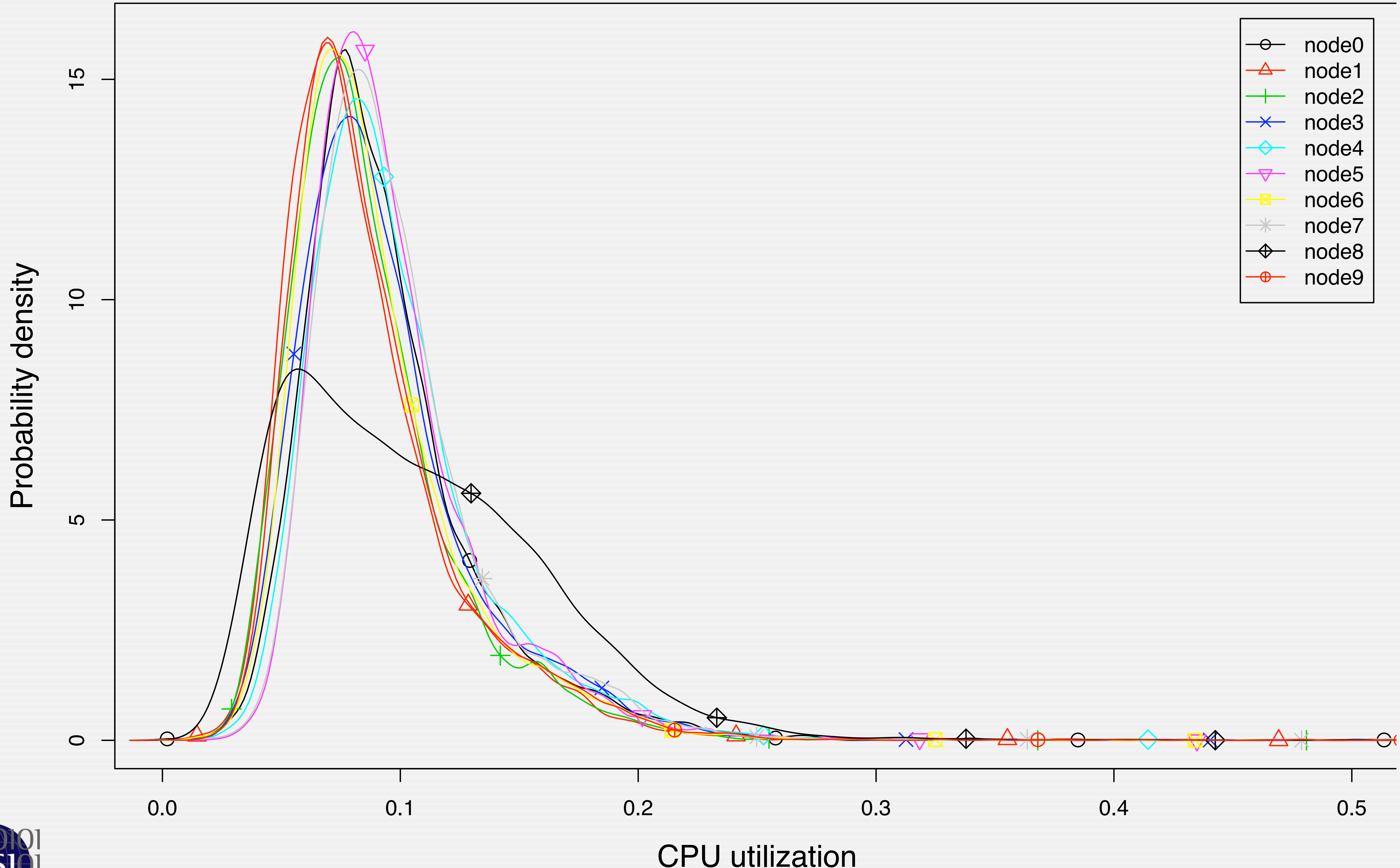



1 day of UC Berkeley campus TCP traffic (231M connections), n = 10

# Simulation of Hashing Schemes

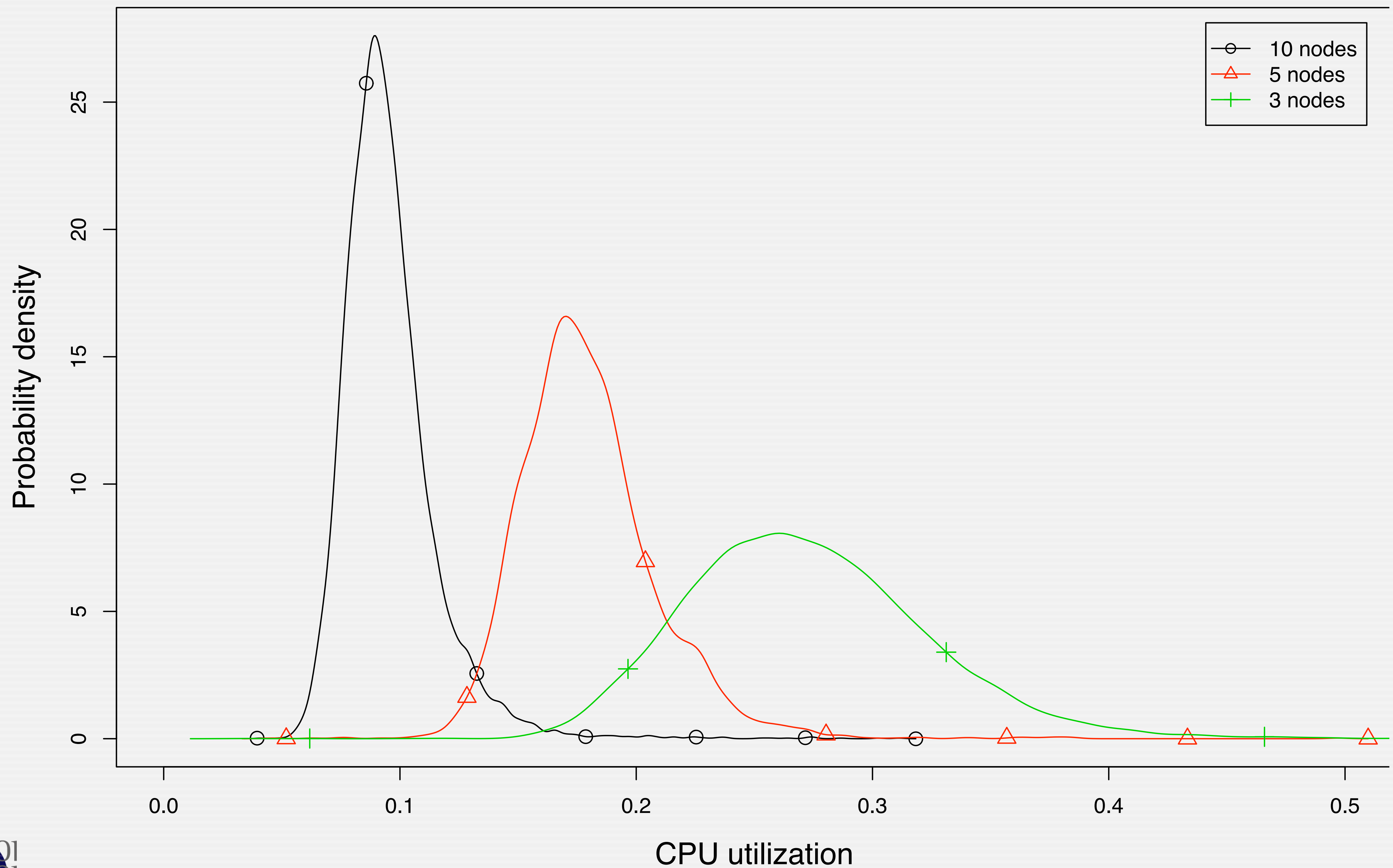


# CPU Load per Node



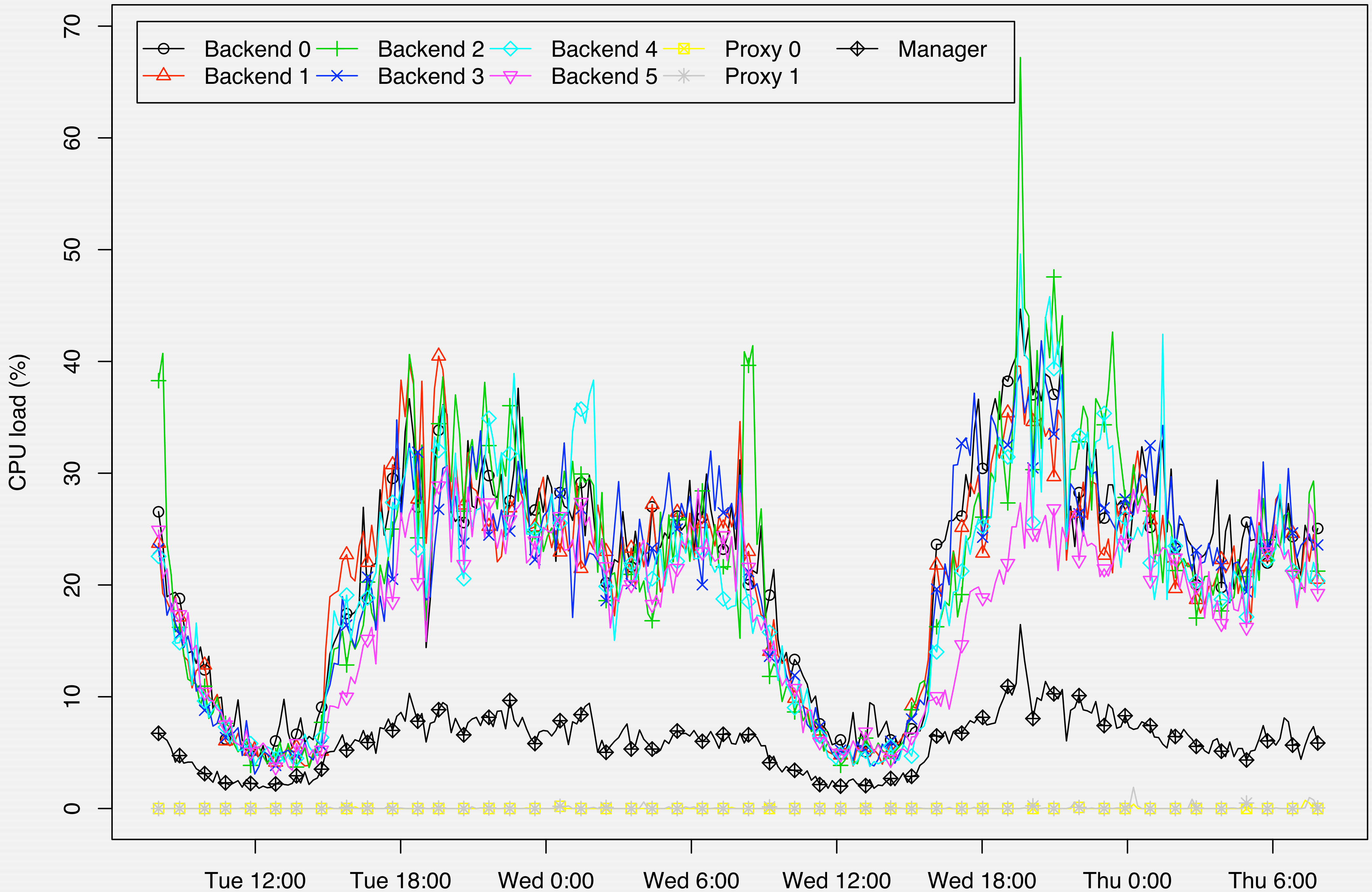

10 backends, ext. LBNL config, 2hr full trace, (~97GB, 134M pkts)

# Scaling of CPU




ext. LBNL config, 2hr full trace, (~97GB, 134M pkts)

# Load on Berkeley Campus




With 1 frontend = 50% of the total traffic

# LBNL Infrastructure

# 10Gbps Tap Setup