Exploiting Multi-Core Processors For Parallelizing Network Intrusion Prevention

Robin Sommer

International Computer Science Institute, &
Lawrence Berkeley National Laboratory

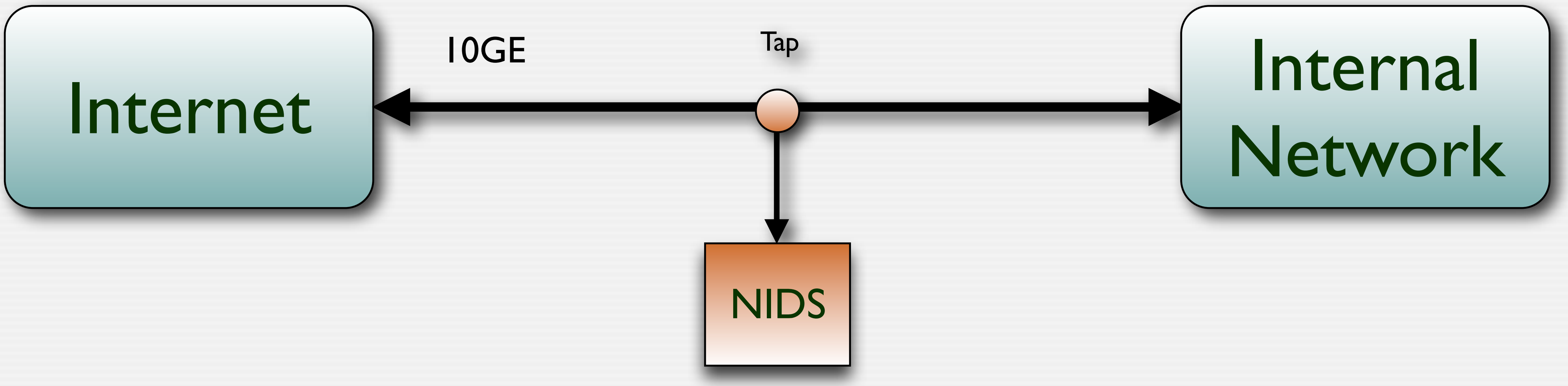
robin@icsi.berkeley.edu
http://www.icir.org

October 2009
Intel Research Berkeley

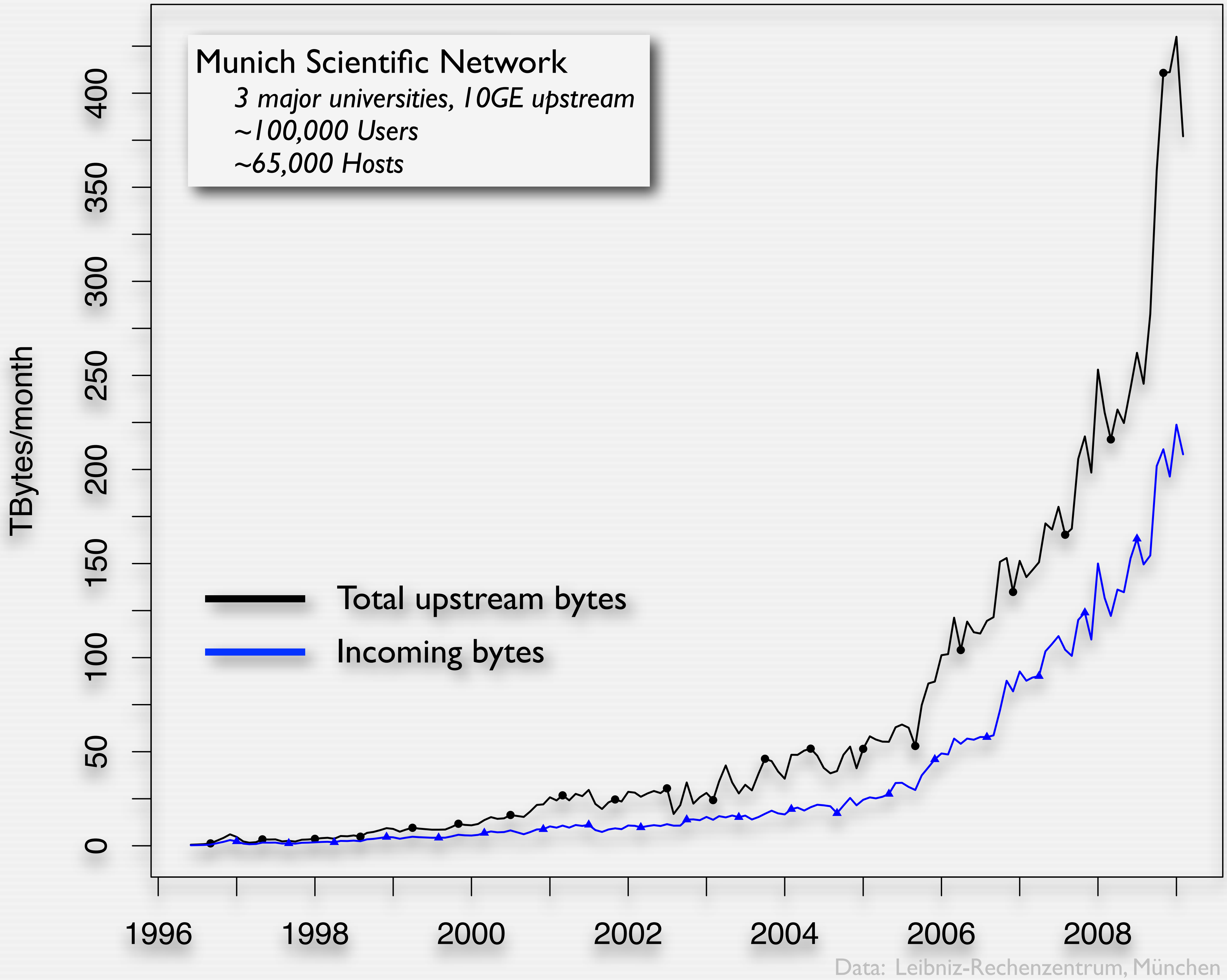
Network Intrusion Detection Systems

- NIDS are typically deployed at a site's upstream link
  - Monitor all external traffic, *packet by packet*
  - Follow the protocol dialogues closely
  - Alert on suspicious activity

  - Face stringent performance requirements due to volume and real-time demands



Development of Internet Traffic



Need for Performance

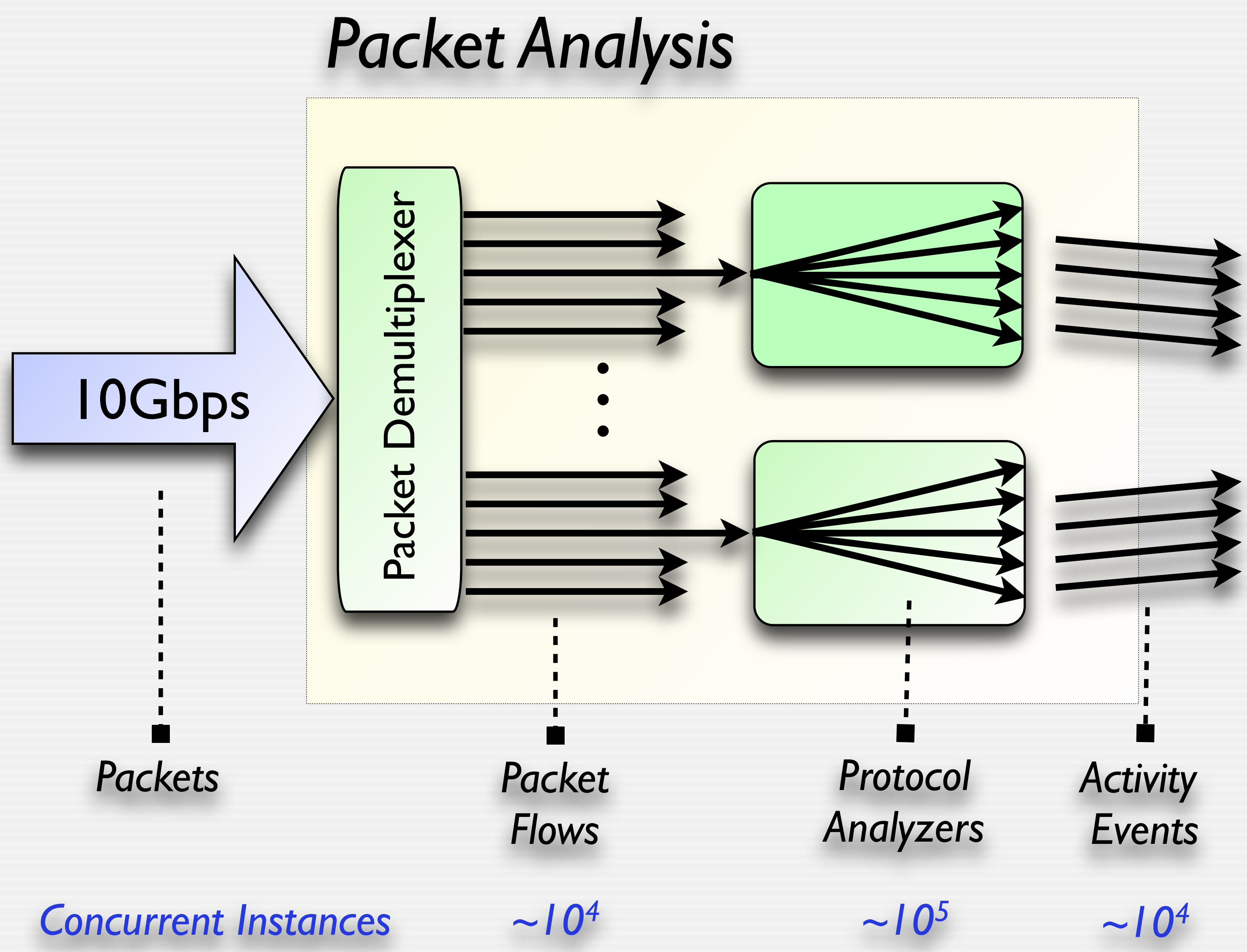
- Keep needing to do *more analysis* on *more data* at *higher speeds*
- NIDS used to run successfully on commodity hardware
  - In particular important for open-source NIDS (e.g., Snort, Bro)
- Not any more!
  - Moore's law doesn't hold for single-core performance anymore
  - Unfortunately, today's NIDS implementations are single-threaded and thus limited
- To overcome, we can
  - Significantly restrict the amount of analysis, *or*
  - Turn to expensive & inflexible custom hardware, *or*
  - *Parallelize processing to leverage commodity multi-core architectures*
- Parallelizing an application is inherently *domain-specific*
  - There's no generic approach to concurrency
  - Need examine carefully where the concurrency potential is that we can exploit

Outline

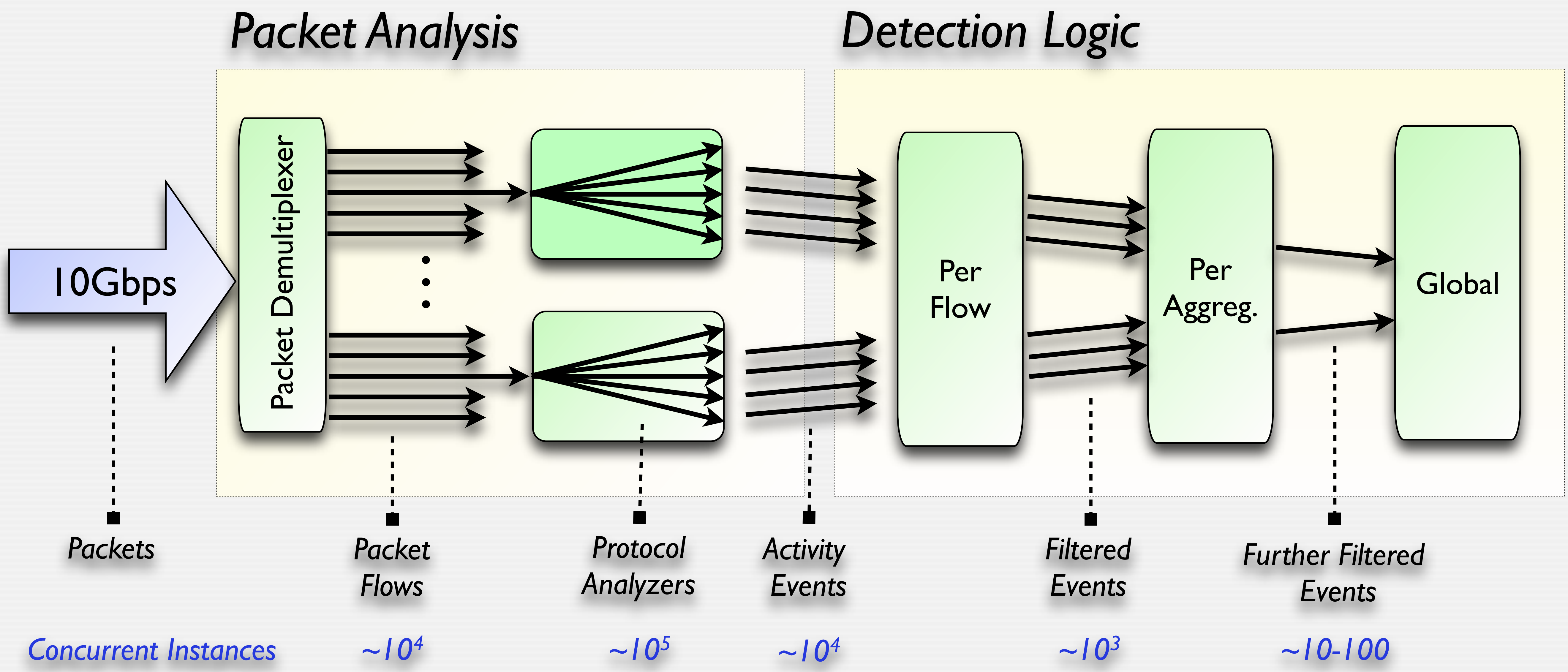
1. *Concurrency Potential in Network Traffic Analysis*
   - A pipeline of highly concurrent stages
2. *Coarse-grained Parallelism: The NIDS Cluster*
   - A load-balancing solution
3. *Fine-grained Parallelism: Building a multi-threaded NIDS*
   - Turning a traditional NIDS into a highly concurrent system
4. *Future Directions*

Concurrency Potential in Network Traffic Analysis

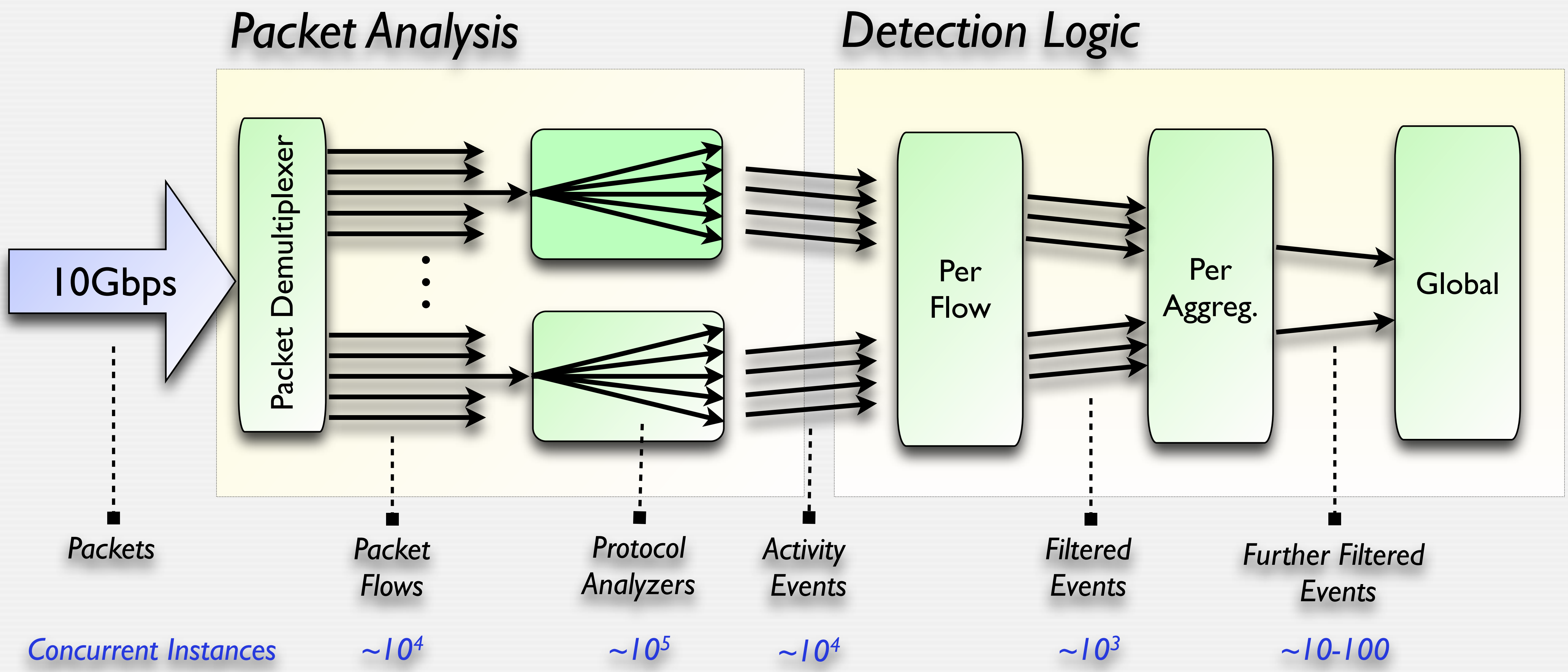
Traffic Analysis Pipeline



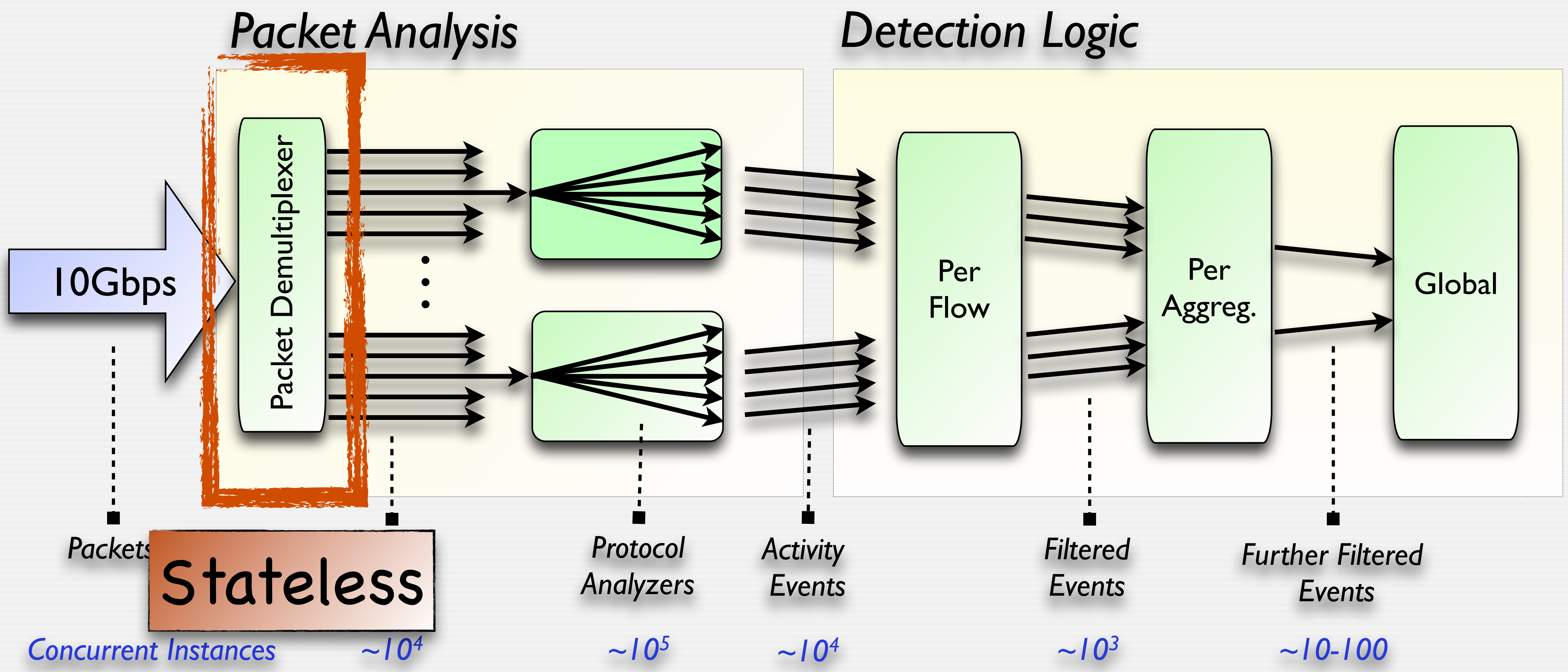
Traffic Analysis Pipeline



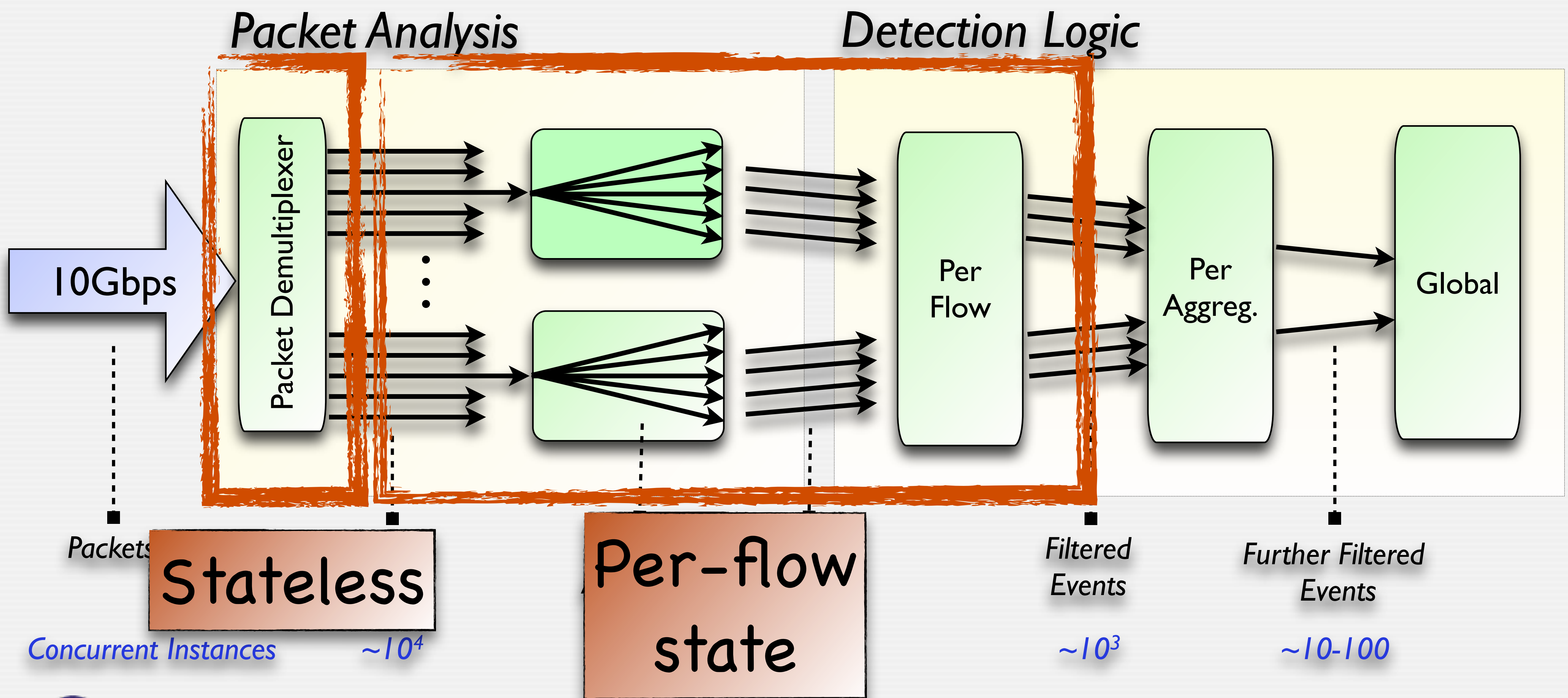
Traffic Analysis Pipeline



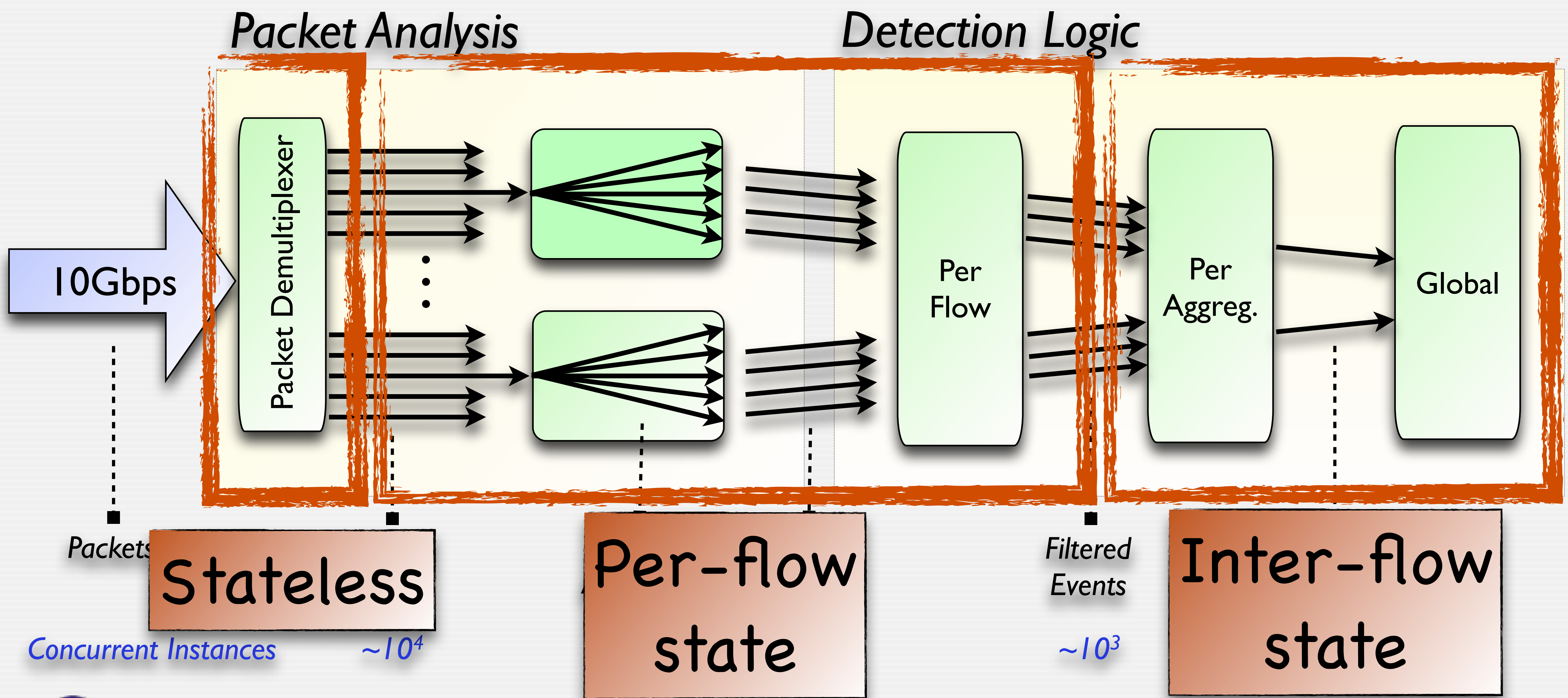
Traffic Analysis Pipeline



Traffic Analysis Pipeline



Traffic Analysis Pipeline

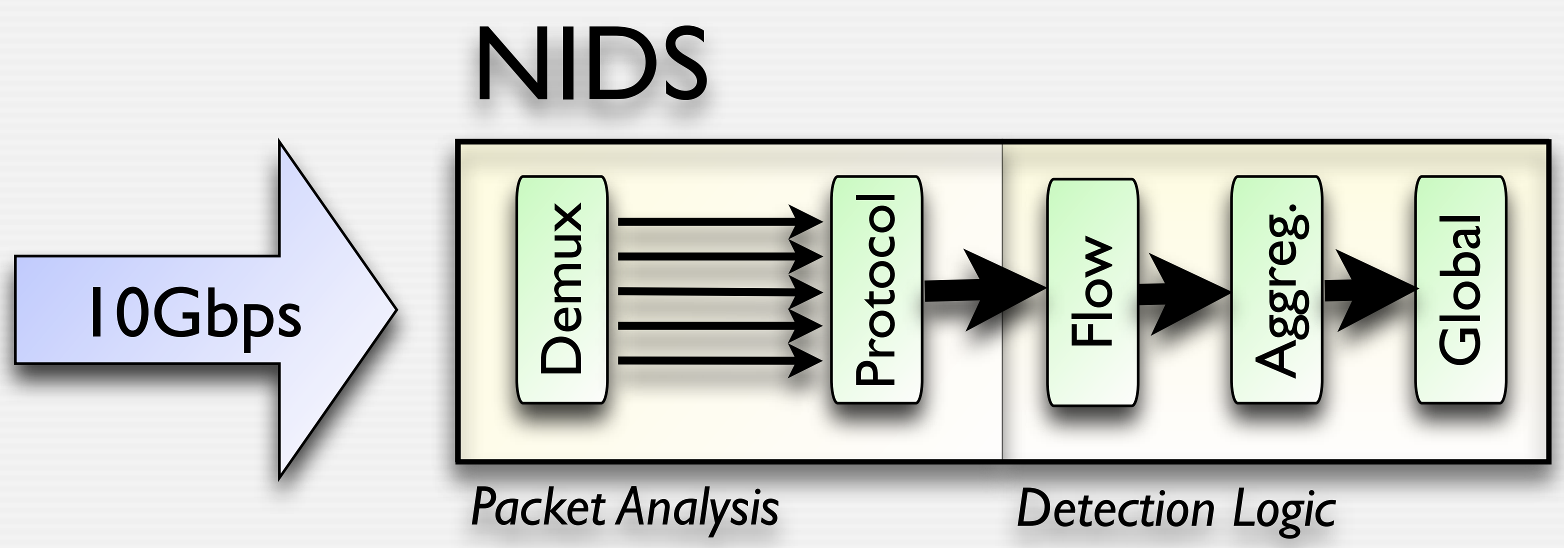


Building a Concurrent NIDS

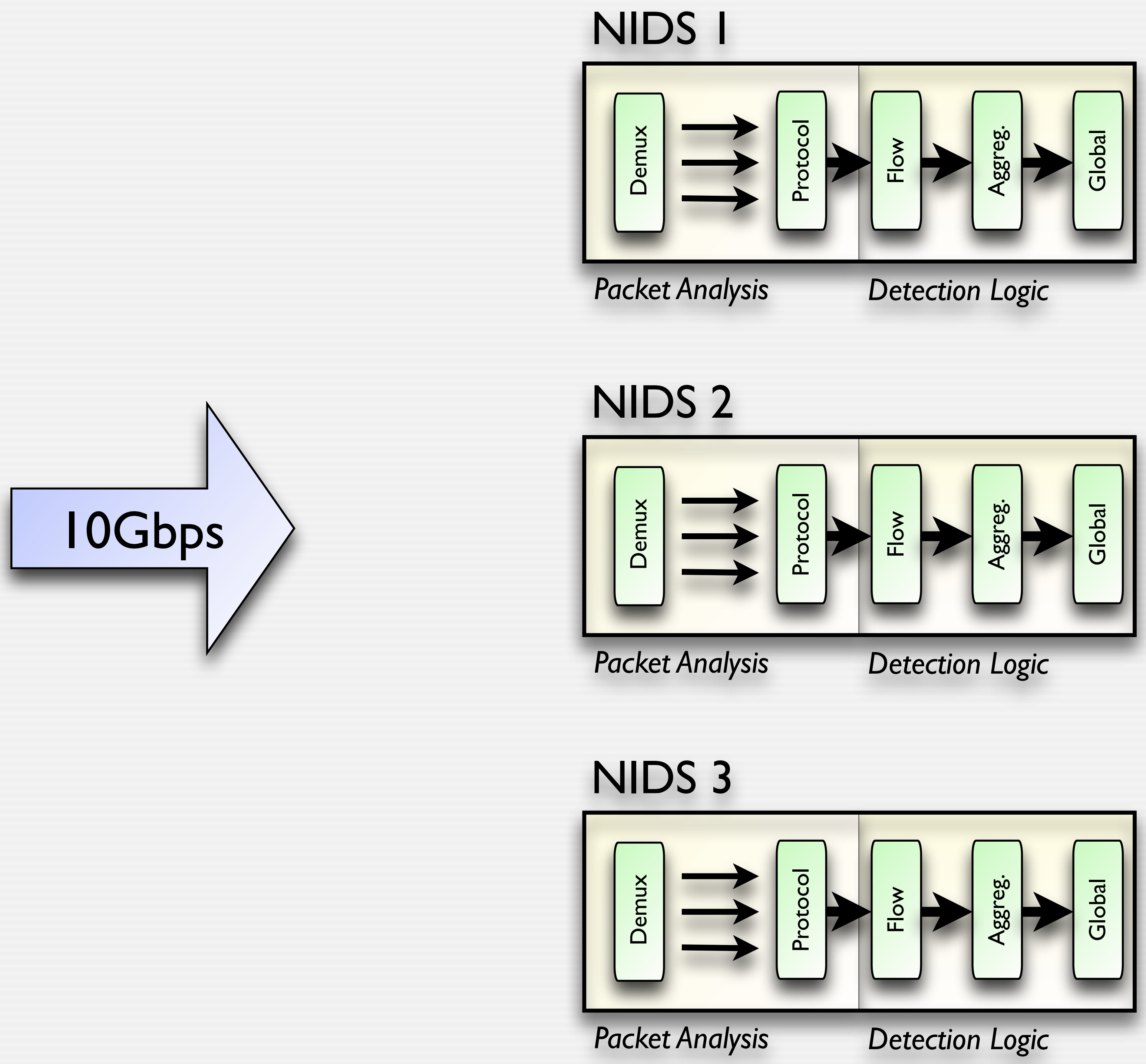
- Don’t really want to build a new NIDS from scratch ...
  - Can we parallelize an existing one?
- Our group at ICSI has been developing the Bro NIDS for more than a decade now.
  - Originally designed by Vern Paxson, who is still leading the project.
  - Open-source, with contributions from many external people.
  - Bro has been the corner-stone of LBNL’s operational security for >10 years.
  - It’s single-threaded however ...
- Can Bro exploit the concurrency of the pipeline?
  - We are talking about 160K lines of C++ code, plus another 25K lines of script code
- Two strategies:
  - Coarse-grained parallelism: The Bro Cluster
  - Fine-grained parallelism: Multi-core Bro

Coarse-grained Parallelism
The Bro Cluster

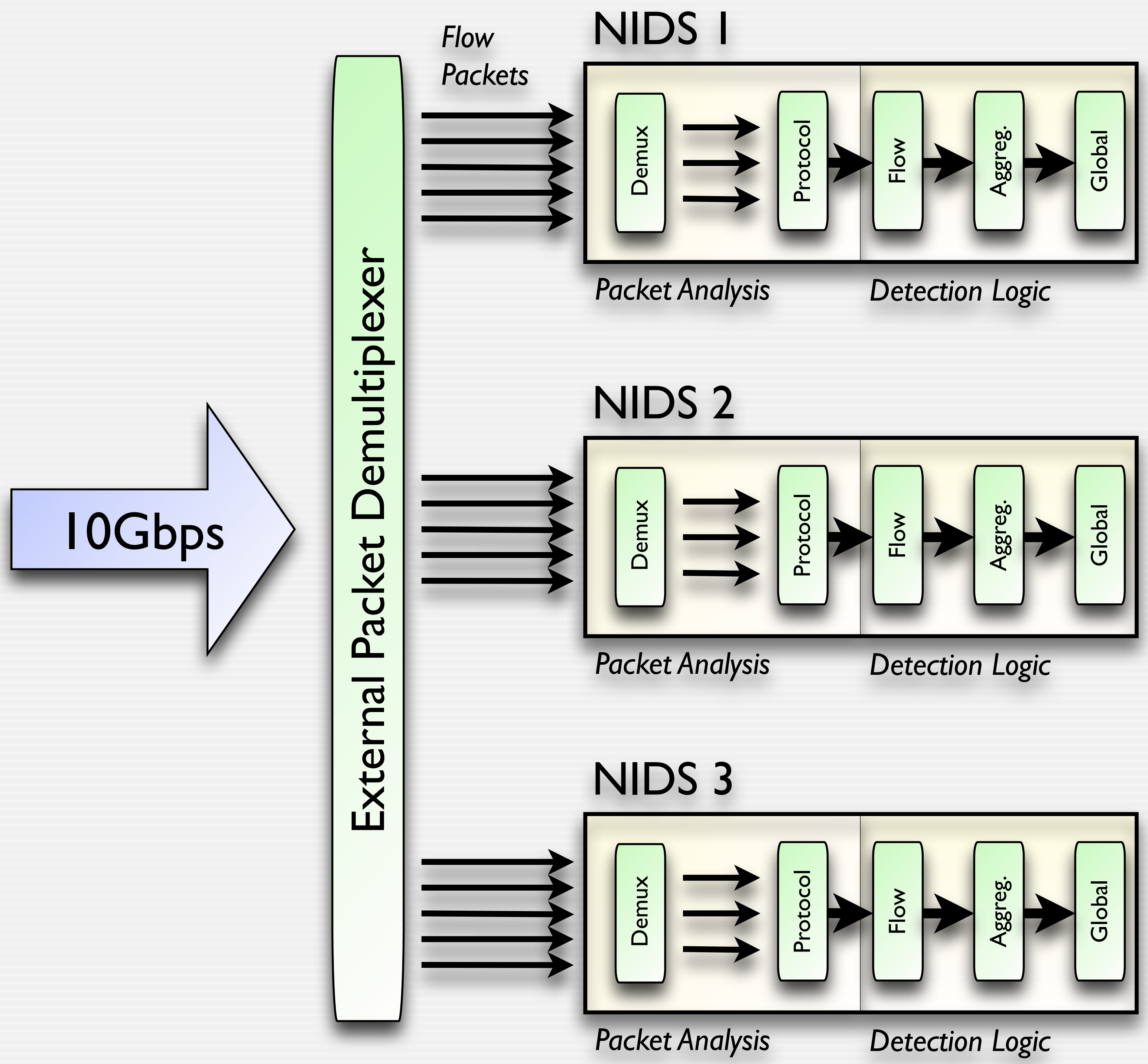
Load-Balancer Approach



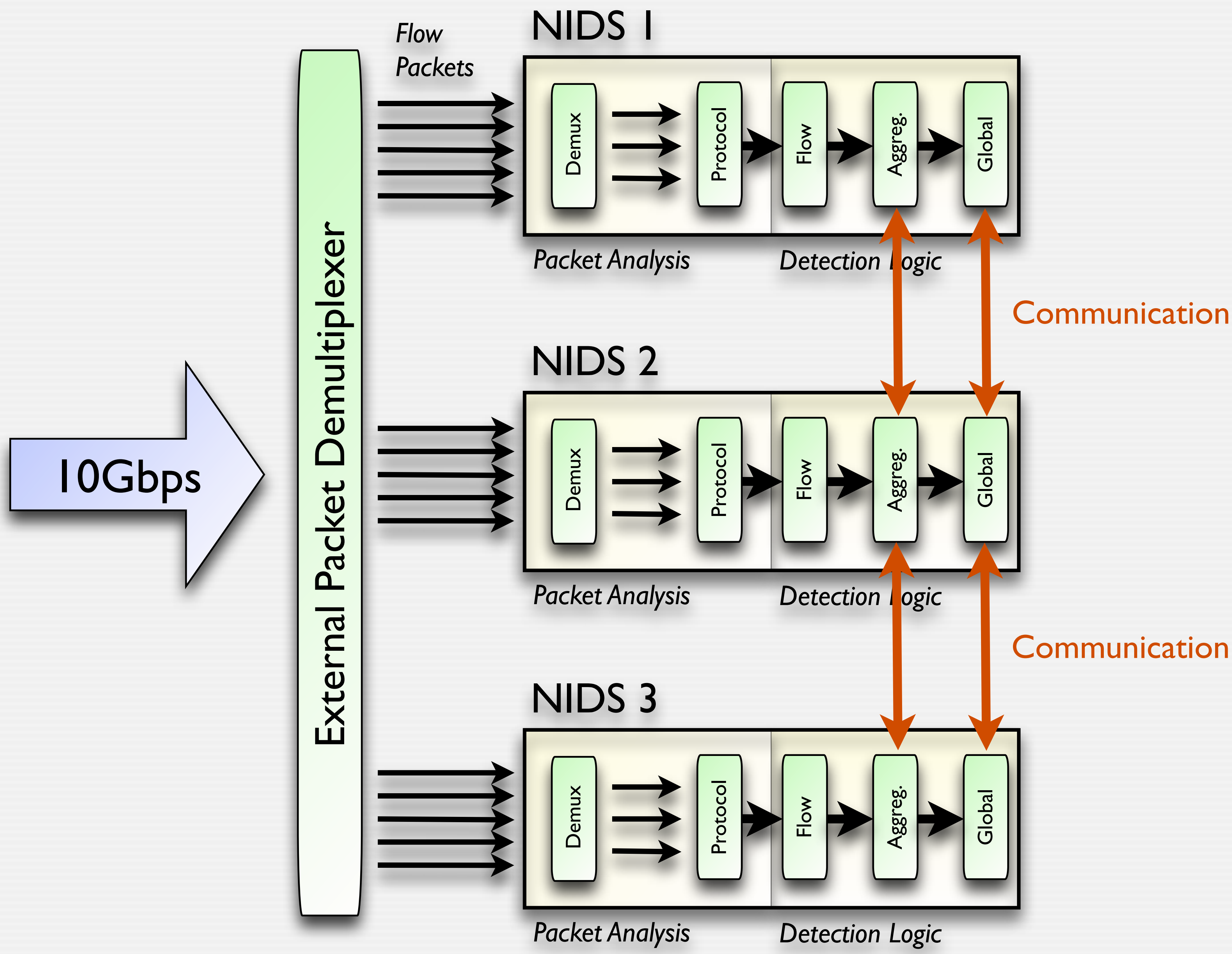
Load-Balancer Approach



Load-Balancer Approach



Load-Balancer Approach



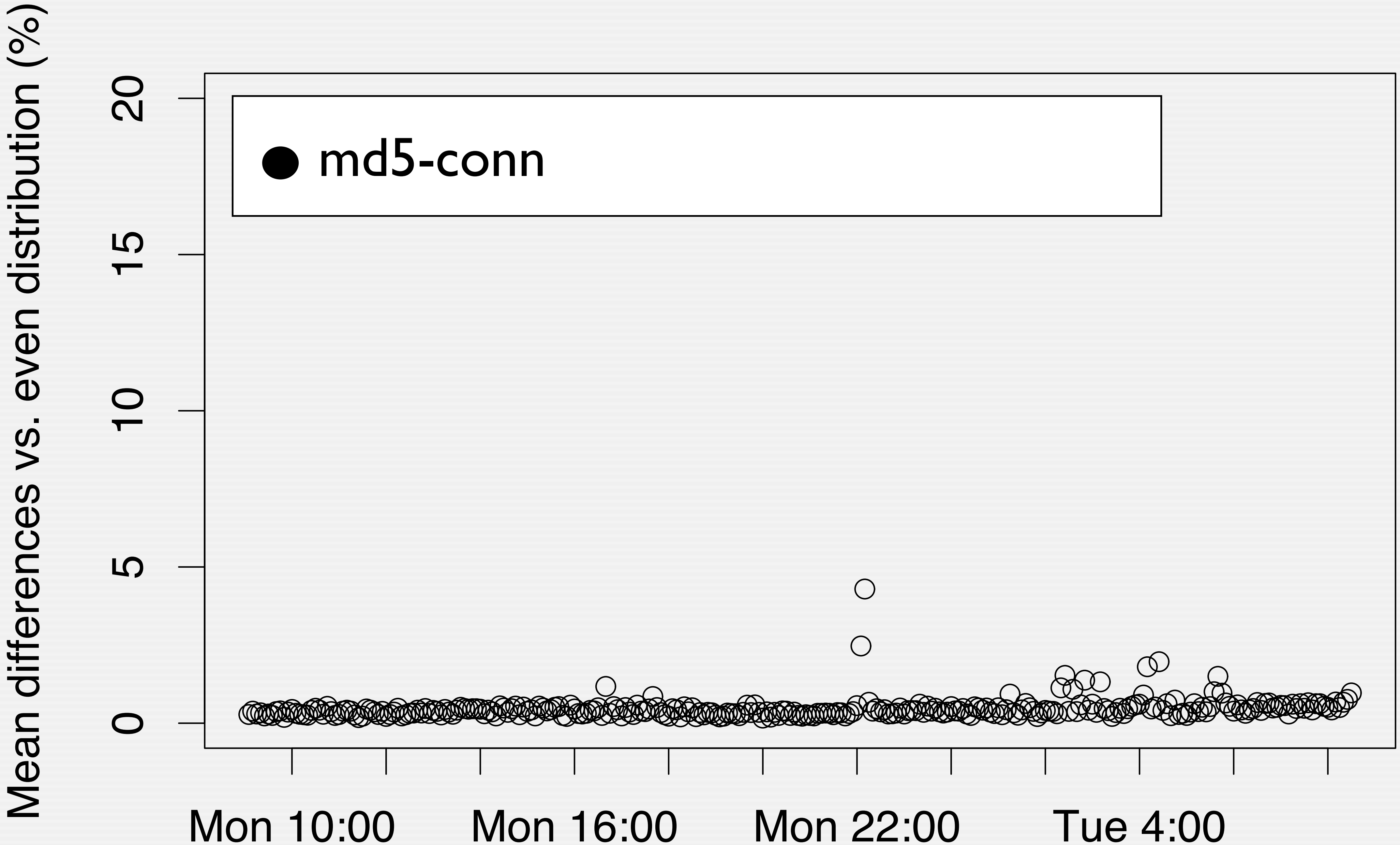
The Bro Cluster

- We built such a cluster using the Bro NIDS
- There are a number of practical challenges:
  - Communication capability required
    Fortunately, Bro has communication primitives built-in
  - External demultiplexer needs to operate line-rate
    Worked with a vendor to build an appliance implementing our dispatching scheme
  - Management of multi-machine setup is tedious
    Build a management interface transparently hiding the complexity for the operator
- Installations
  - Research cluster at LBNL w/ 10 NIDS machines (backends)
  - Operational cluster at LBNL w/ 15 NIDS, replacing Labs' security monitoring
  - Other sites are running, or planing to run, similar setups
  - Planing much larger research cluster on the Berkeley campus

External Packet Dispatcher

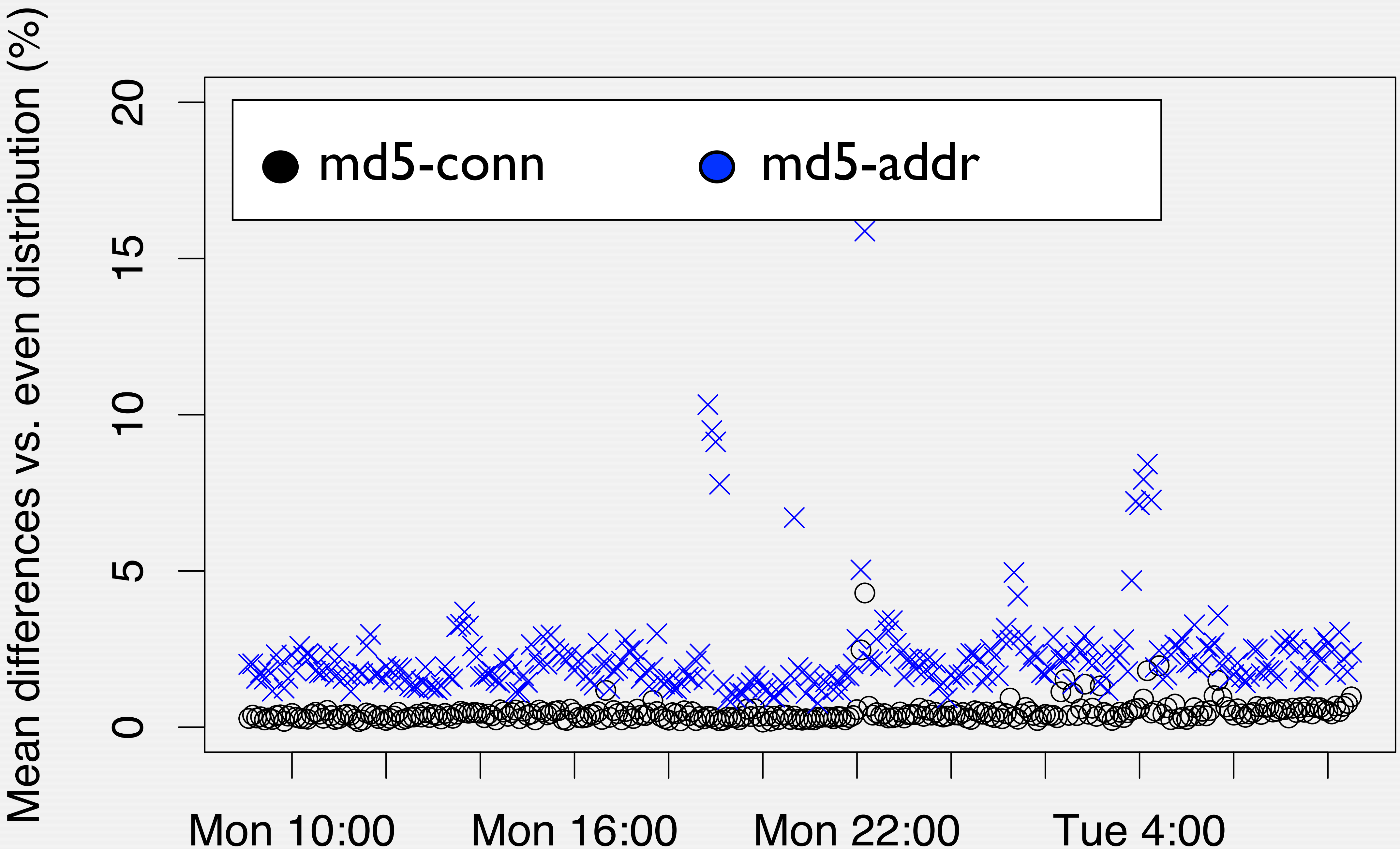
- How to decide where to send a packet?
- We want the dispatcher to
  - Keep flows together
  - Be simple and stateless for implementation in hardware
- Observation: Each packet contains a flow identifier
  - 4-tuple of IP addresses and TCP/UDP port numbers
- Dispatcher can calculate hash over the 4-tuple
  - `backend := hash(`*`tuple`*`) mod N`
- But how smooth a distribution does that yield?

Simulation of Packet Dispatcher



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

Simulation of Packet Dispatcher



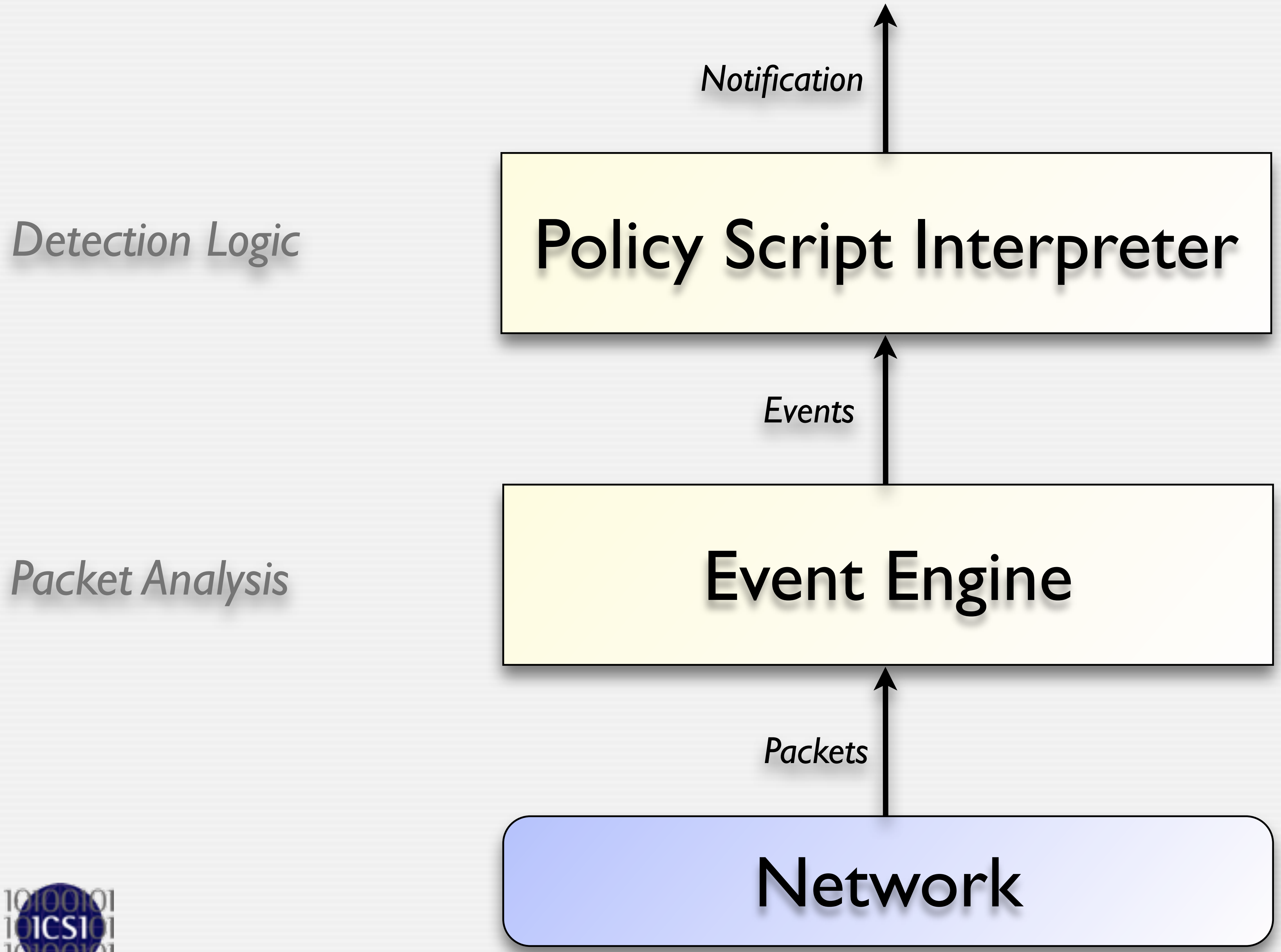
`1 day of UC Berkeley campus TCP traffic (231M connections), n = 10`

Fine-grained Parallelism
Building a Multi-Threaded NIDS

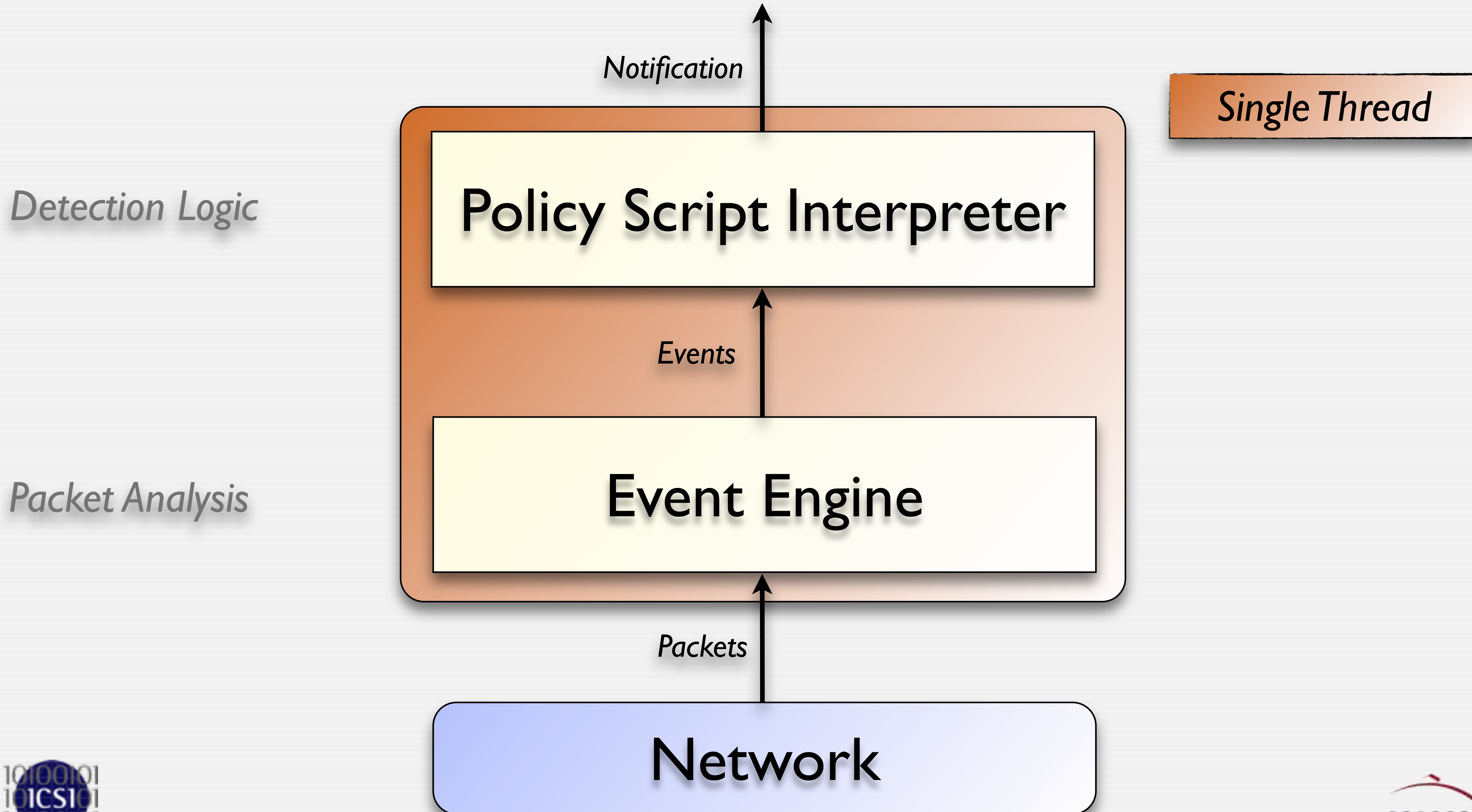
“Real” Multi-Core NIDS

- Cluster has short-comings:
  - Chances are that today’s backends have multiple cores, which will be wasted
  - State is unnecessarily duplicated across all backends
  - Communication introduces race-conditions
  - Setup requires quite a bit of effort (and money)
- What we *really* want is a multi-threaded NIDS
  - ... and we want it to scale well with increasing numbers of cores
- Still don’t want to write a new NIDS from scratch
  - Turn the traditional Bro into a multi-threaded application
- Main objective is doing that *transparently:*
  - Do not expose parallel processing to the operator
  - But parallelize internally “under the hood”

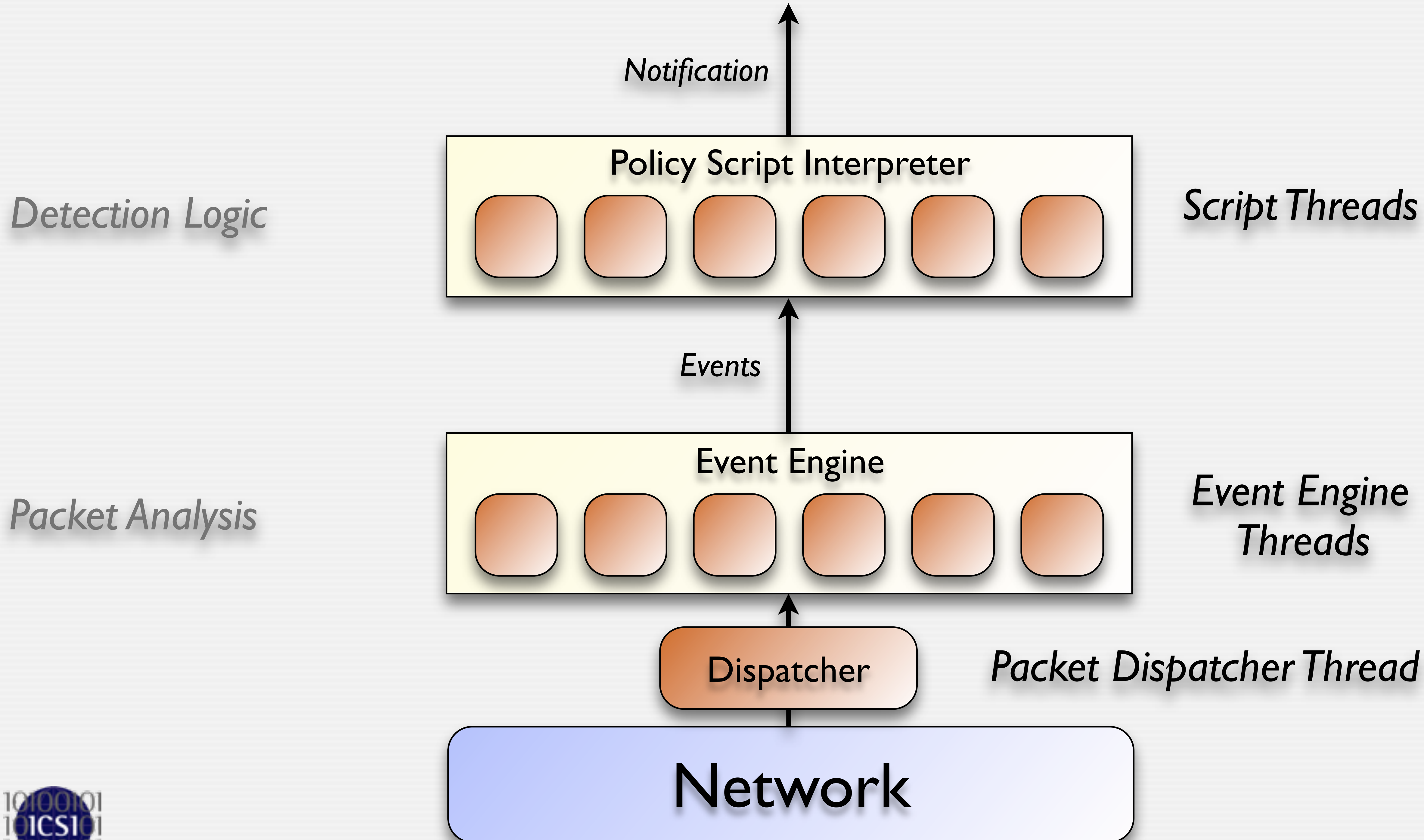
Bro's Architecture



Bro's Architecture



Bro's Architecture

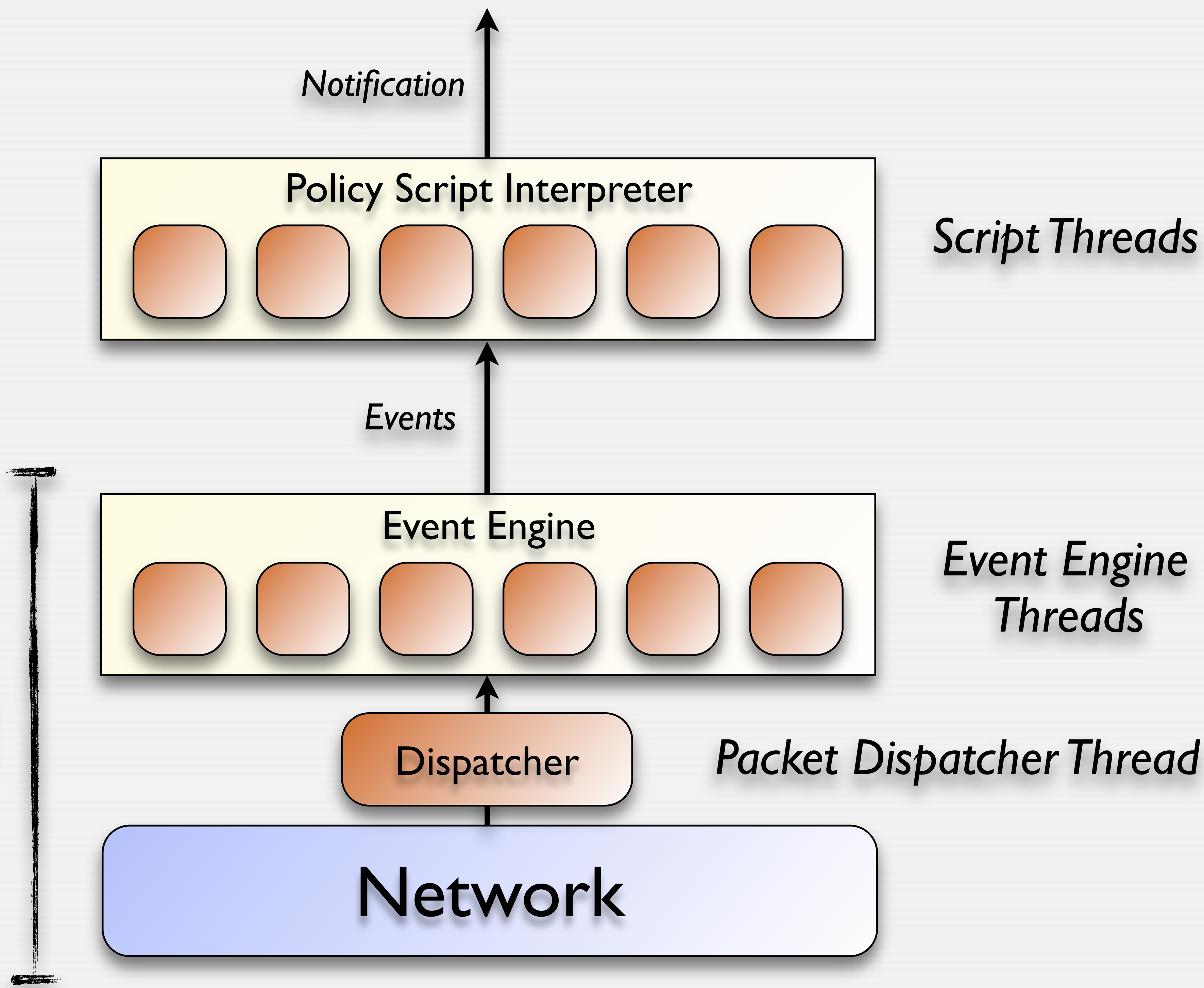


Bro's Architecture

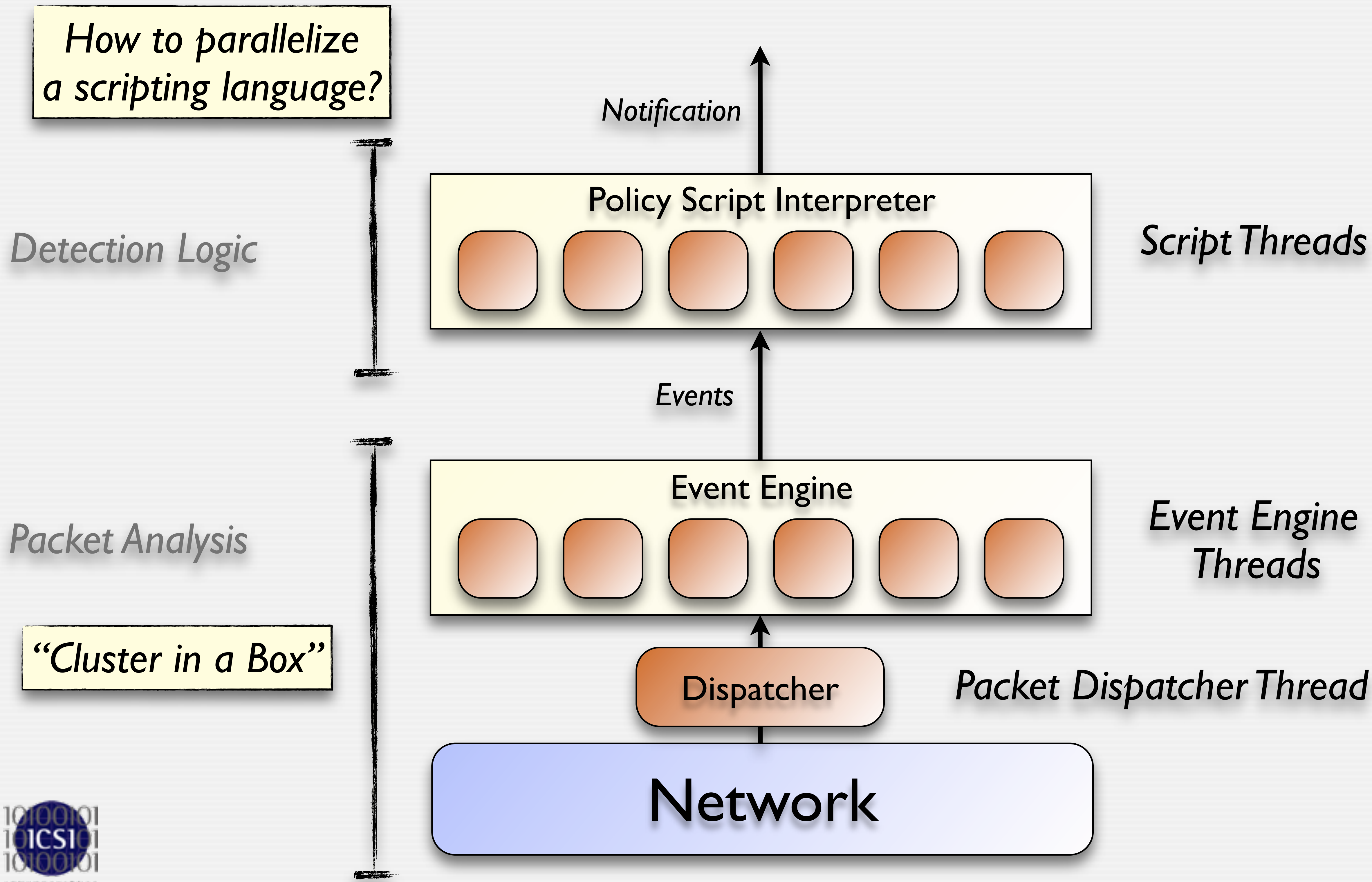
Detection Logic

Packet Analysis

"Cluster in a Box"



Bro's Architecture



Script Example: Matching URLs

Task: Report all Web requests for files called "`passwd`".

```
event http_request(c: connection, method: string, path: string)
{
    if ( method == "GET" && path == /.*passwd/ )
        NOTICE(SensitiveURL, c, path); # Alarm.
}
```

(Syntax simplified.)

Example: *`http_request(1.2.3.4/4321⇒5.6.7.8/80, "GET", "/index.html")`*

Script Example 2: Flow-based

Task: *Report all successful HTTP requests for files called "`passwd`".*

```
global potentially_sensitive: table[connection] of string;

event http_request(c: connection, method: string, path: string)
{
    if ( method == "GET" && path == /.*passwd/ )
        potentially_sensitive[c] = path; # Add to table.
}

event http_reply(c: connection, response: int, reason: string) )
{
    if ( response == OK && c in potentially_sensitive )
        NOTICE(SensitiveURL, c, potentially_sensitive[c]);
}
```

(Syntax simplified.)

Script Example 3: Aggregated

Task: *Count failed connection attempts per source address .*

```
global attempts: table[addr] of int &default=0;

event connection_rejected(c: connection)
{
    local source = c.orig_h;          # Get source address.
    local n = ++attempts[source];     # Increase counter.
    if ( n == SOME_THRESHOLD )        # Check for threshold.
        NOTICE(Scanner, source);      # If so, report.
}
```

(Syntax simplified.)

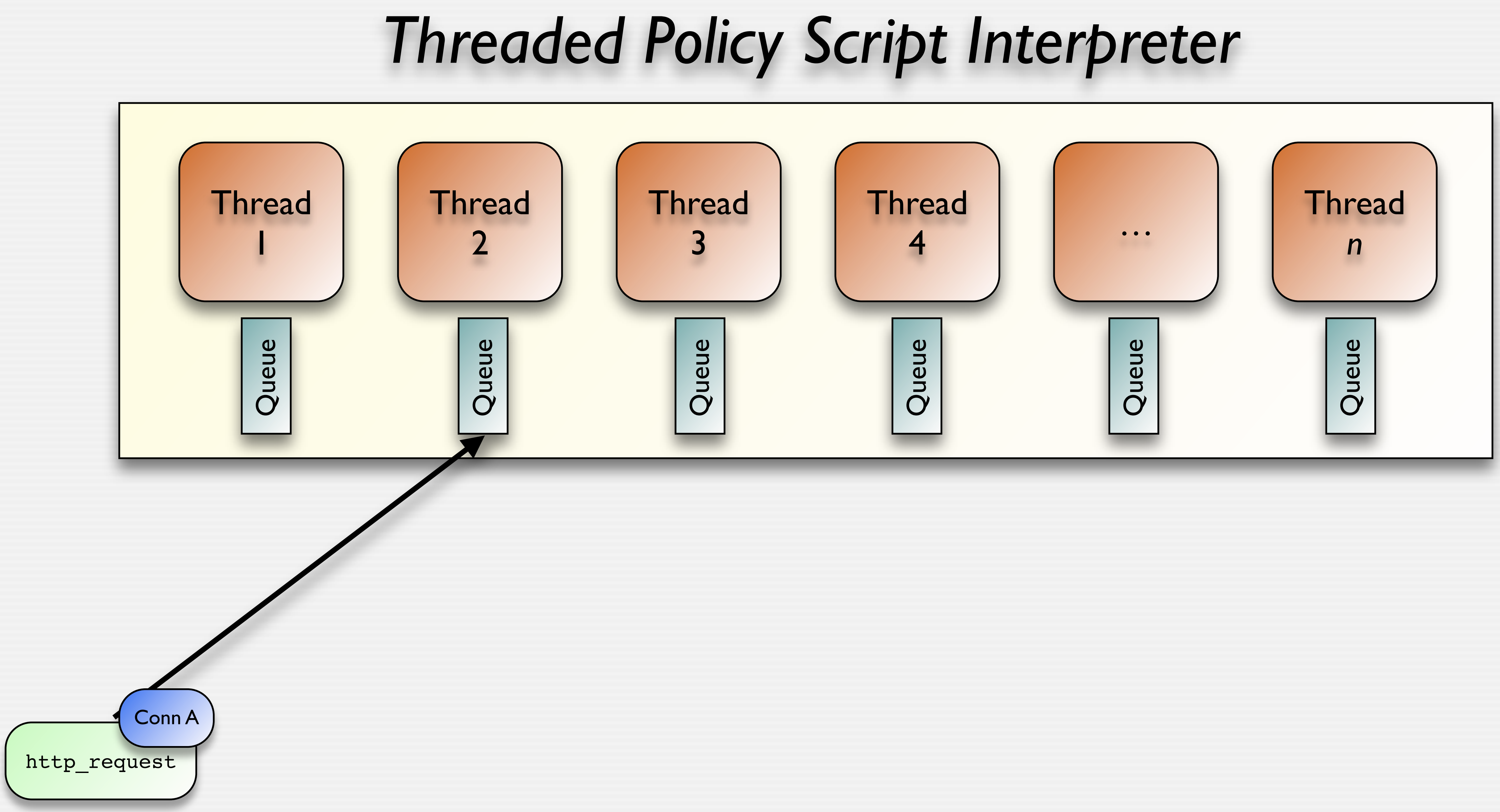
“Scheduling Scopes”

- Accessing a piece of state from only *one* thread buys us:
  - Lock-free memory accesses
  - Preservation of temporal order of event execution
- We add the concept of *scopes* to Bro’s script language:
  - For each variable, one specifies the semantic granularity of accesses (e.g., *connection, originator, responder, host pair*)
  - All accesses with the same underlying unit will come from the same thread.
  - Internally, we keep thread-local versions of each variable
  - For each event handler, Bro derives a scope based on which variables it accesses
  - When it is scheduled, the scope & current unit determine which thread it goes to

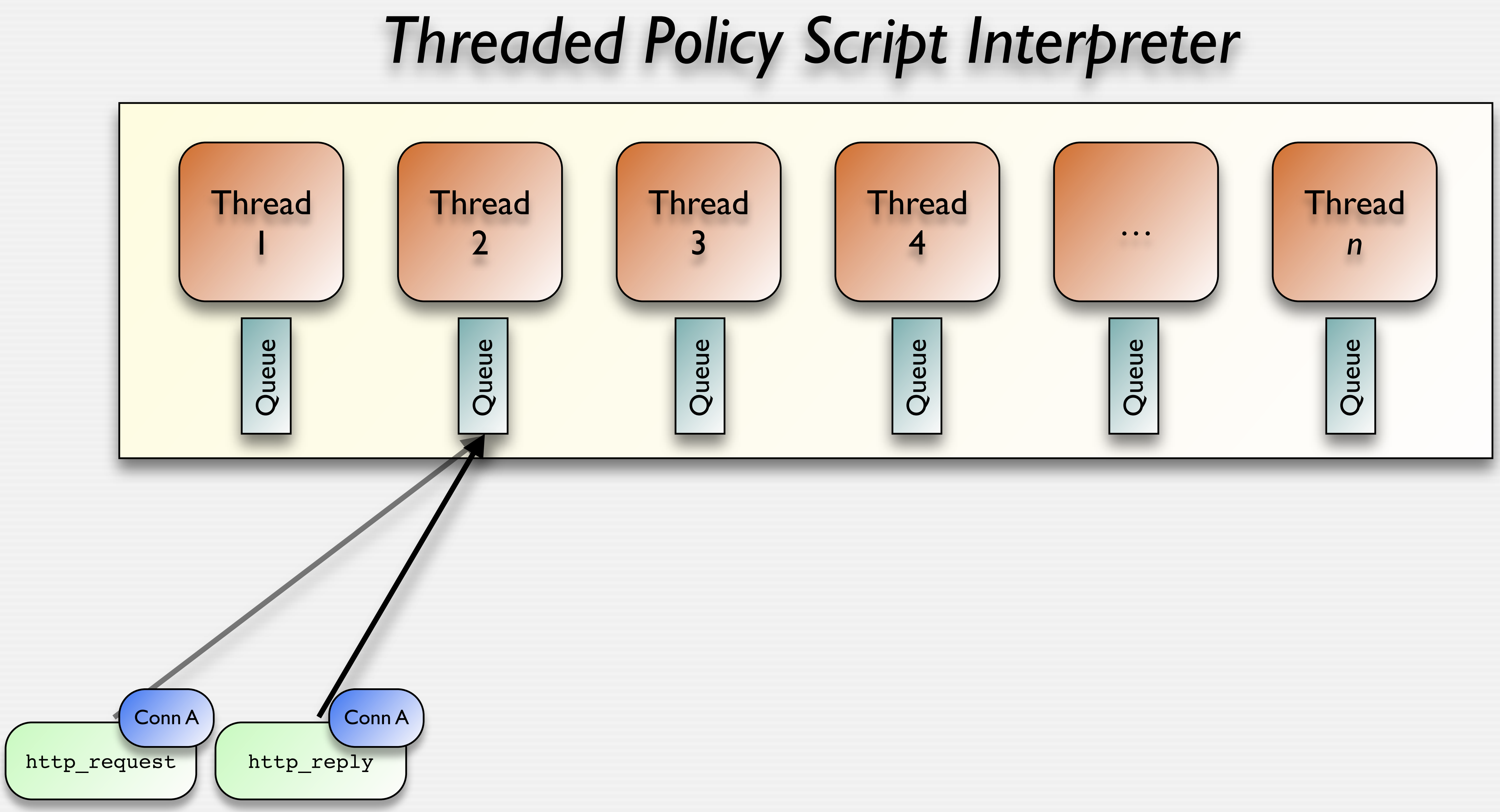
```
global potentially_sensitive: table[connection] of string
   &scope=connection;

global attempts: table[addr] of int &default=0
   &scope=originator;
```

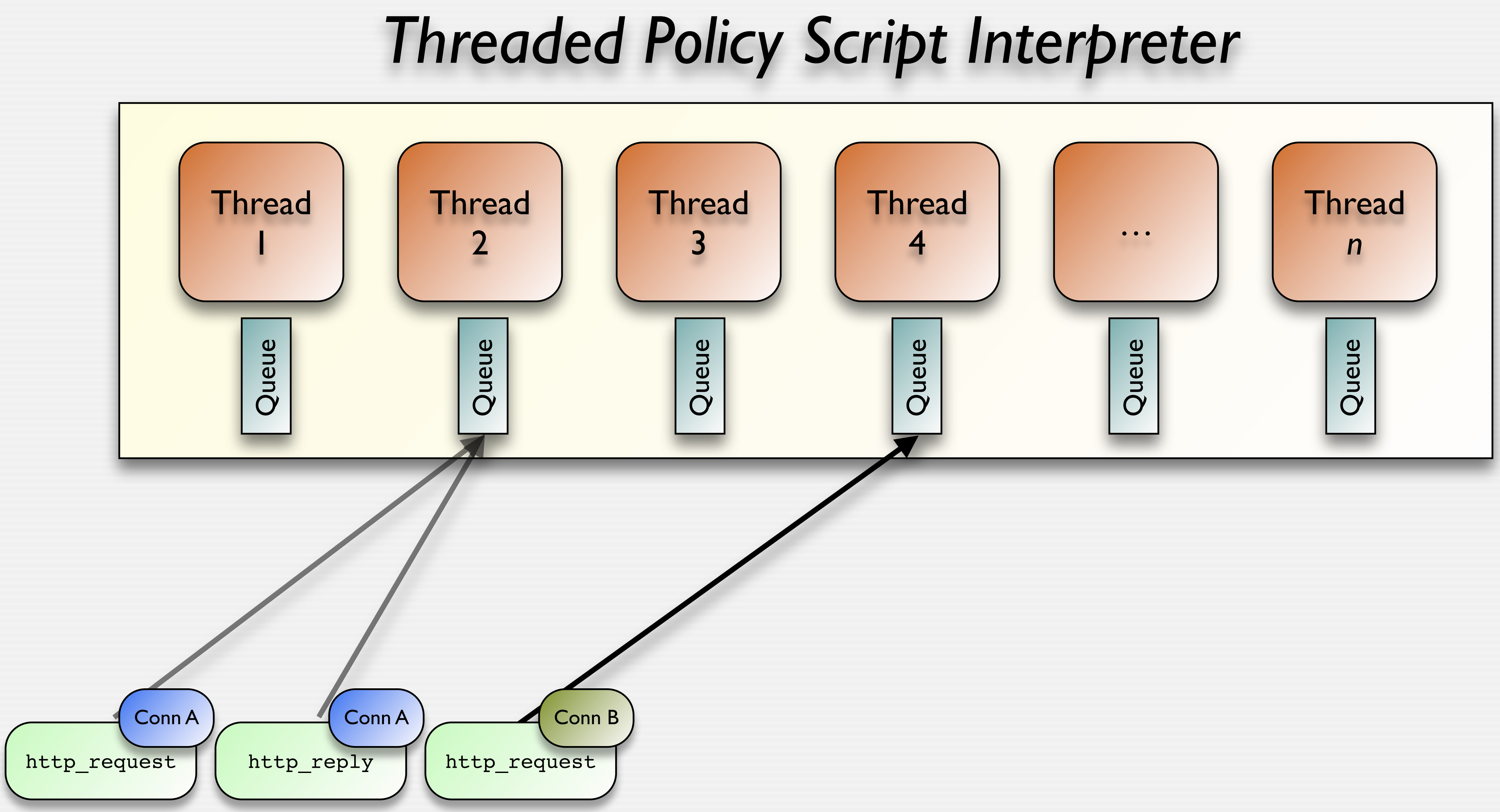
Parallel Event Scheduling



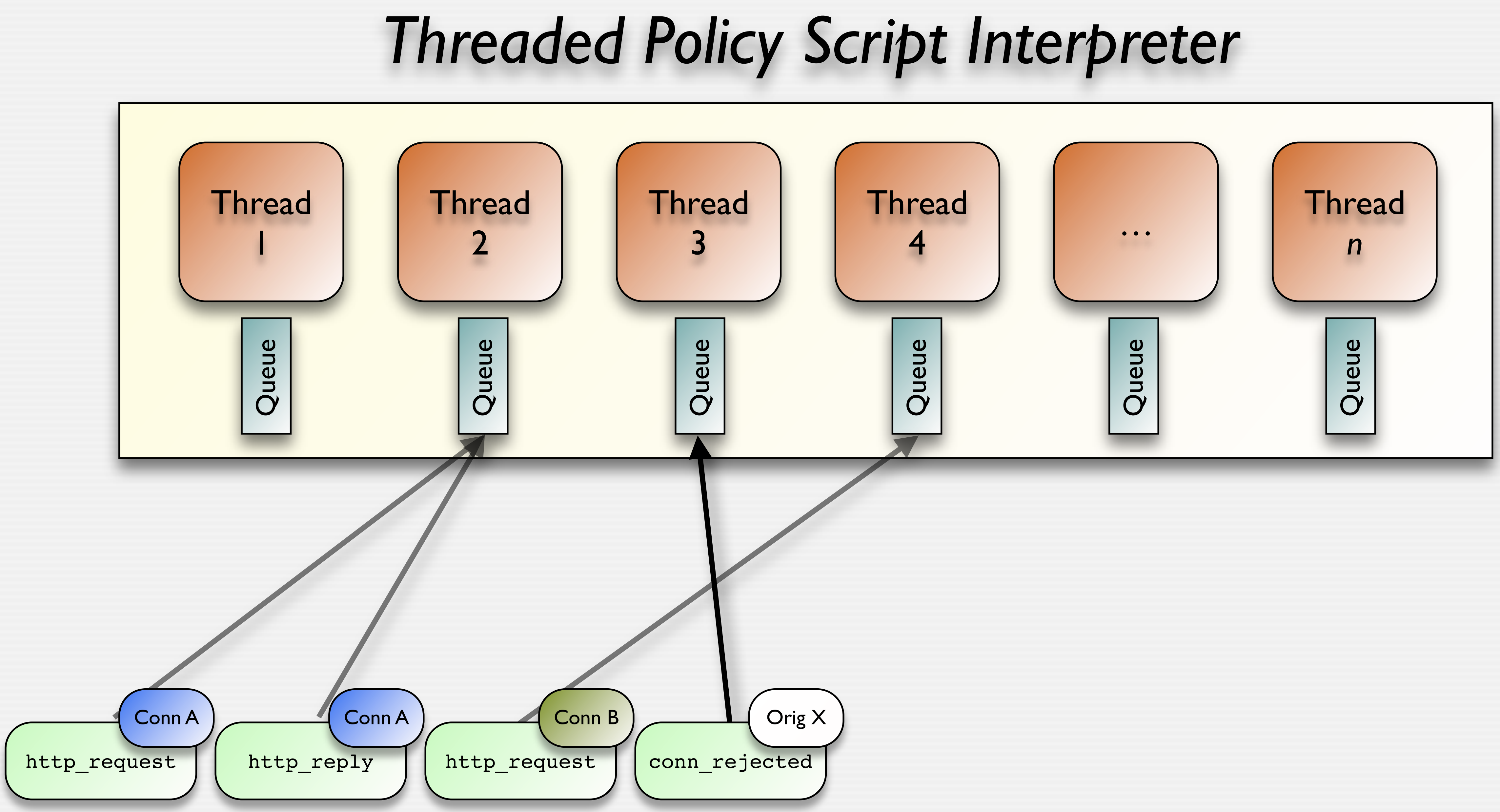
Parallel Event Scheduling



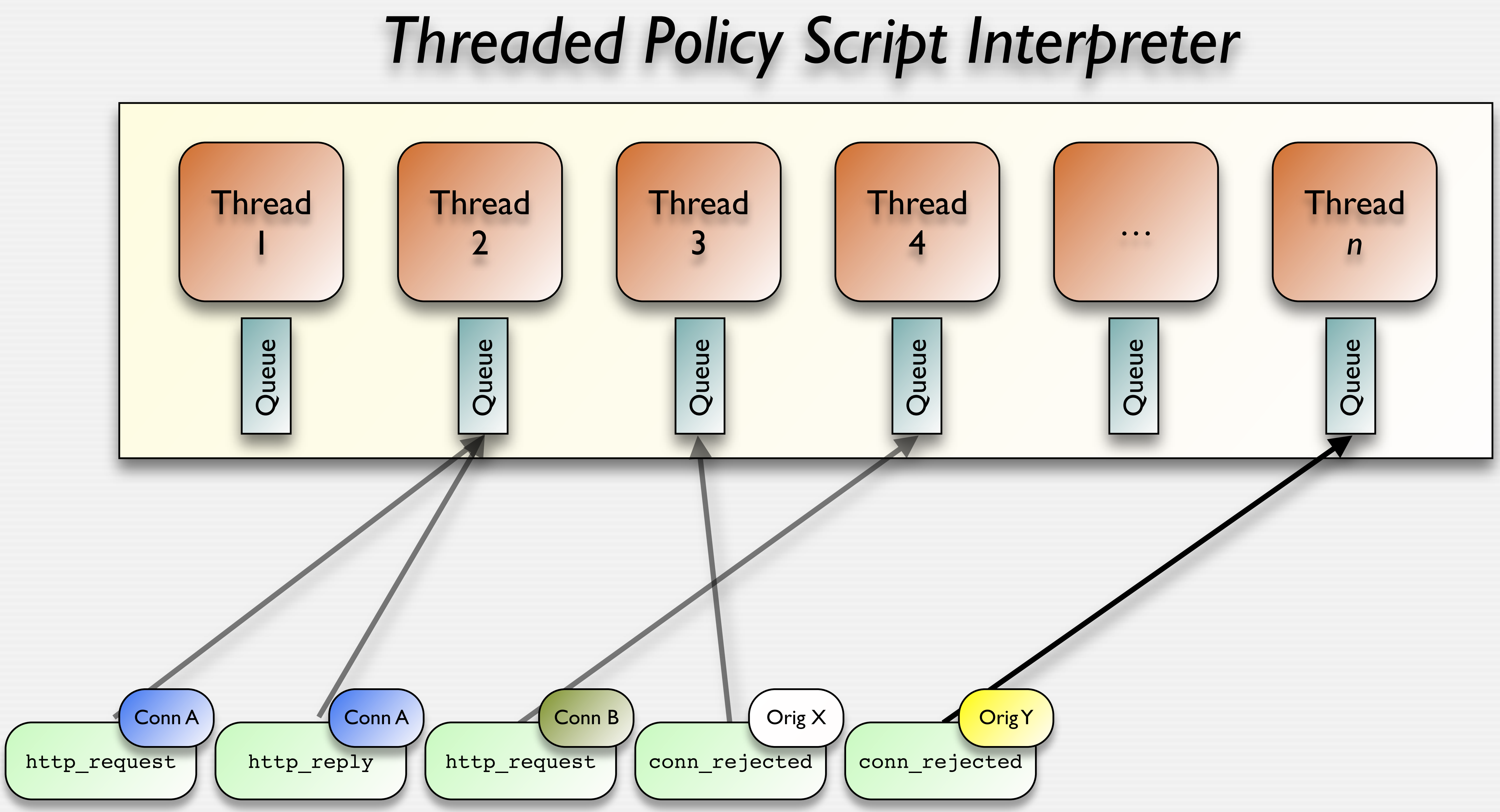
Parallel Event Scheduling



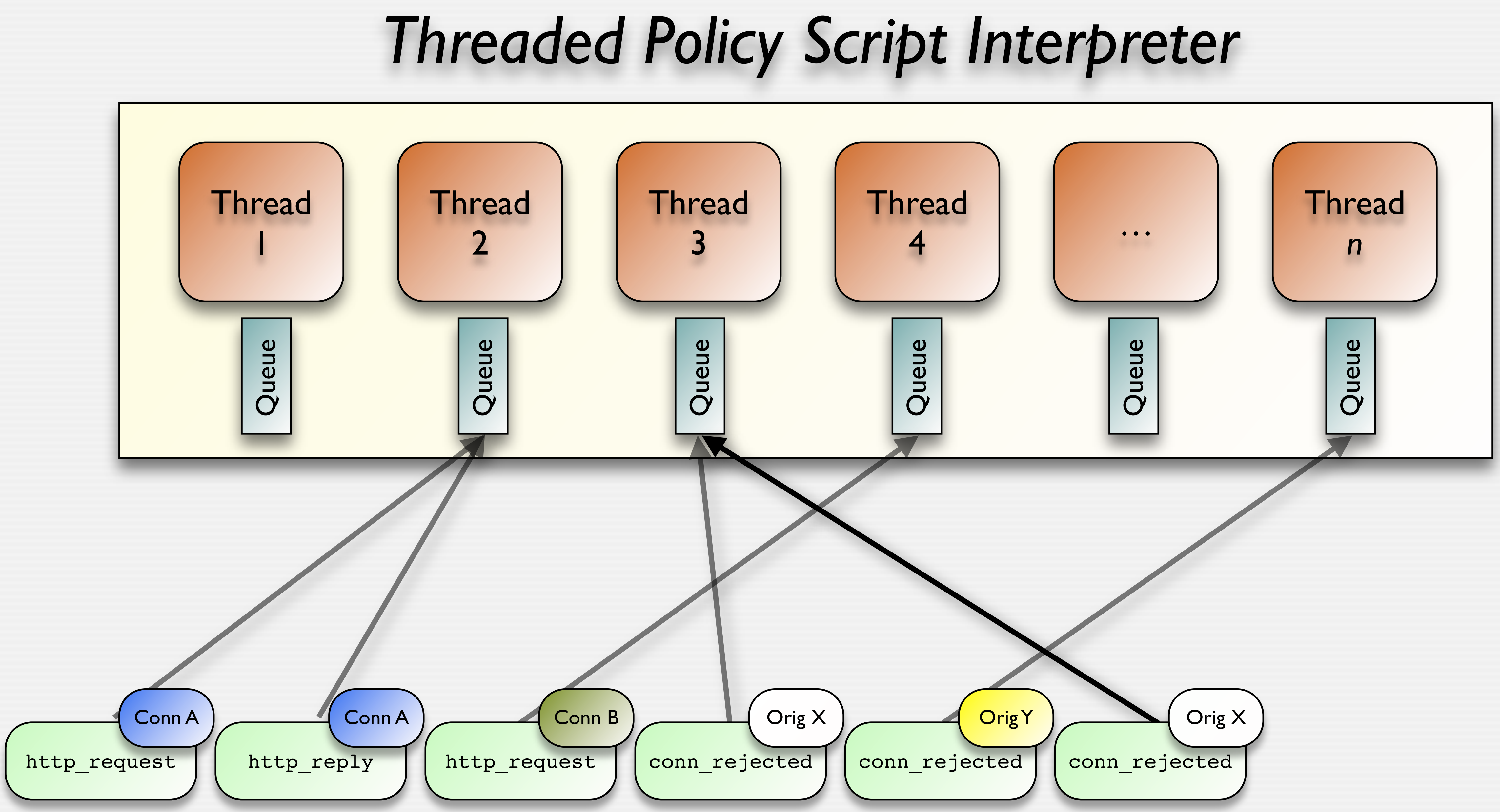
Parallel Event Scheduling



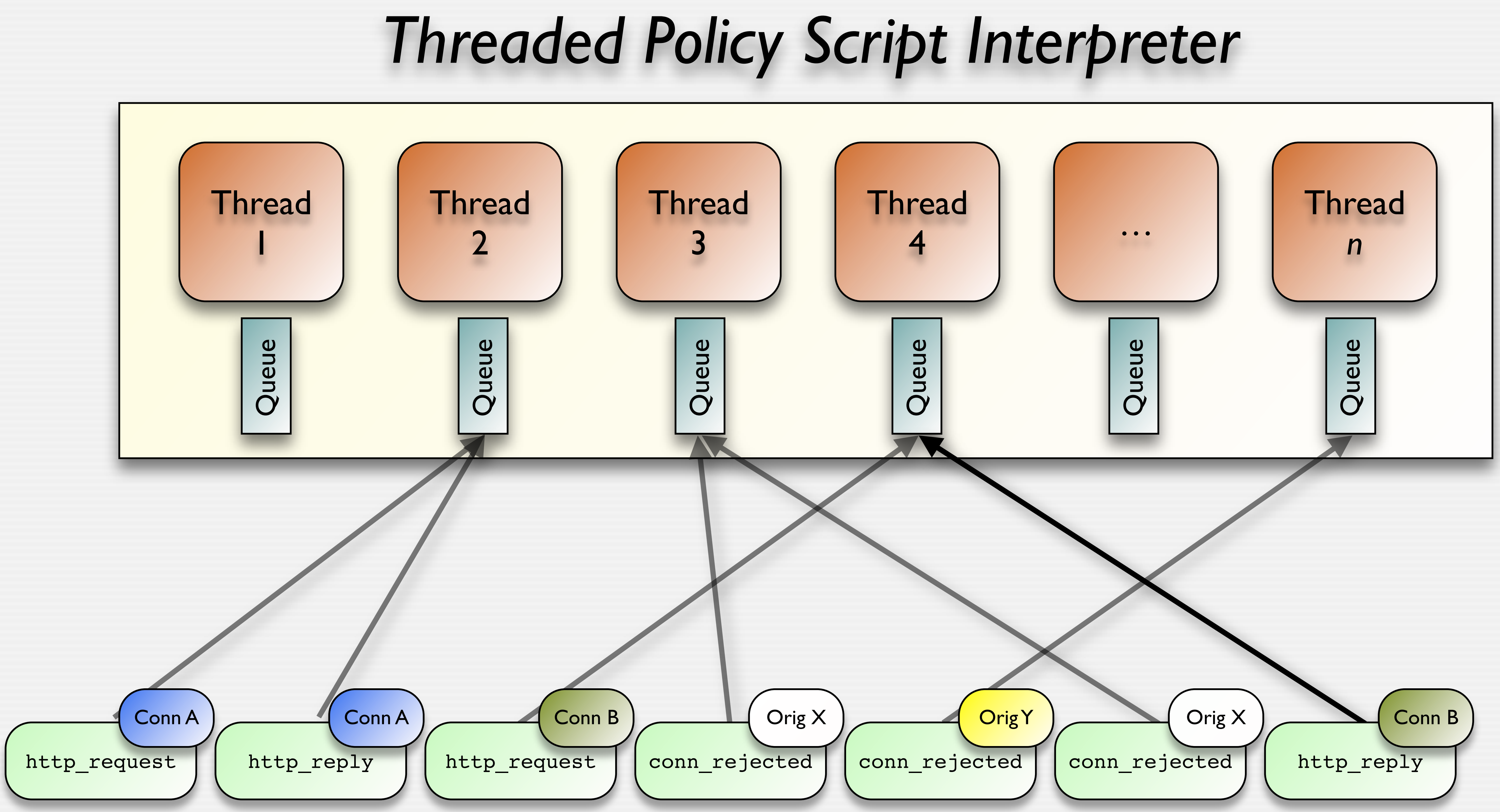
Parallel Event Scheduling



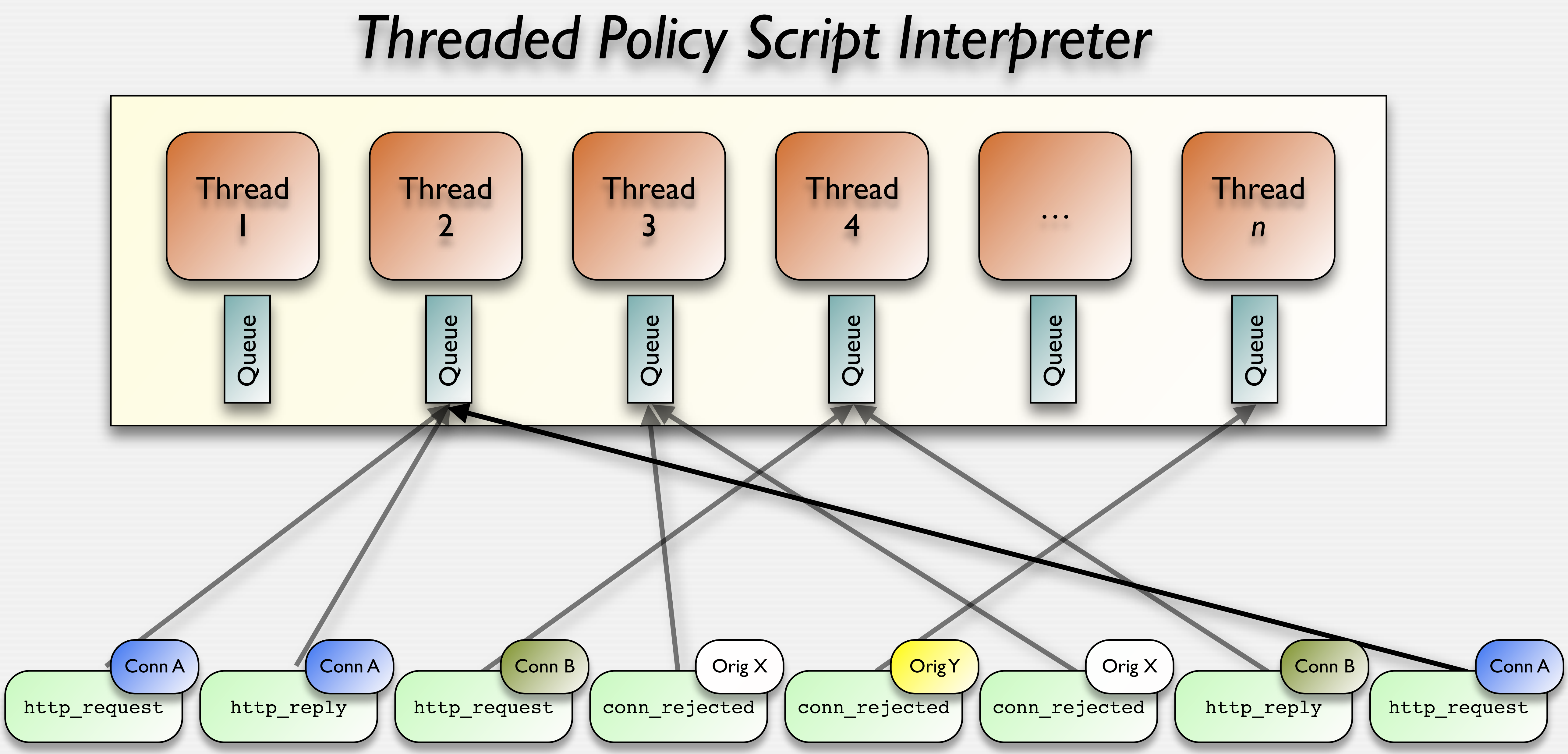
Parallel Event Scheduling



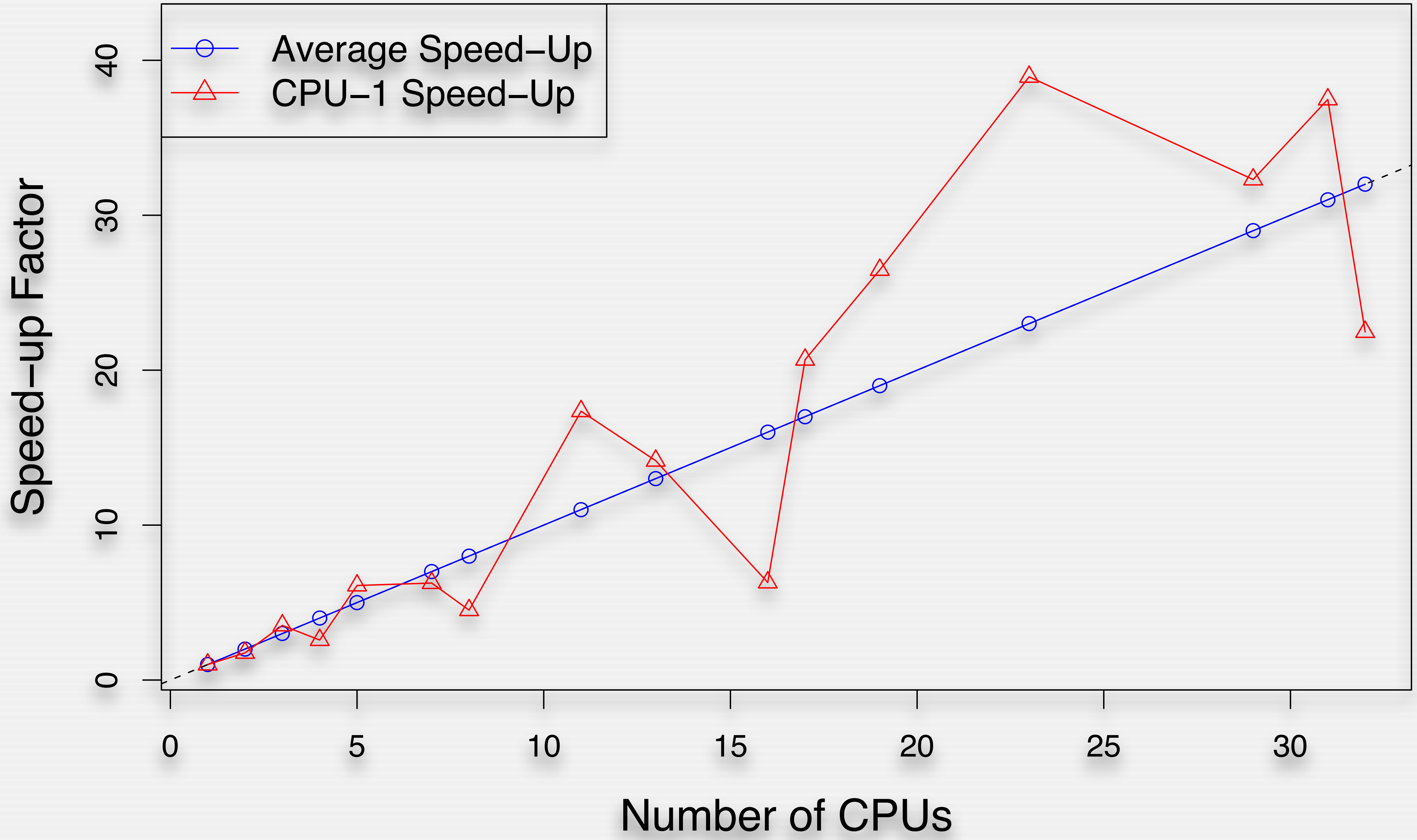
Parallel Event Scheduling



Parallel Event Scheduling



Does the Scoping Model Scale?

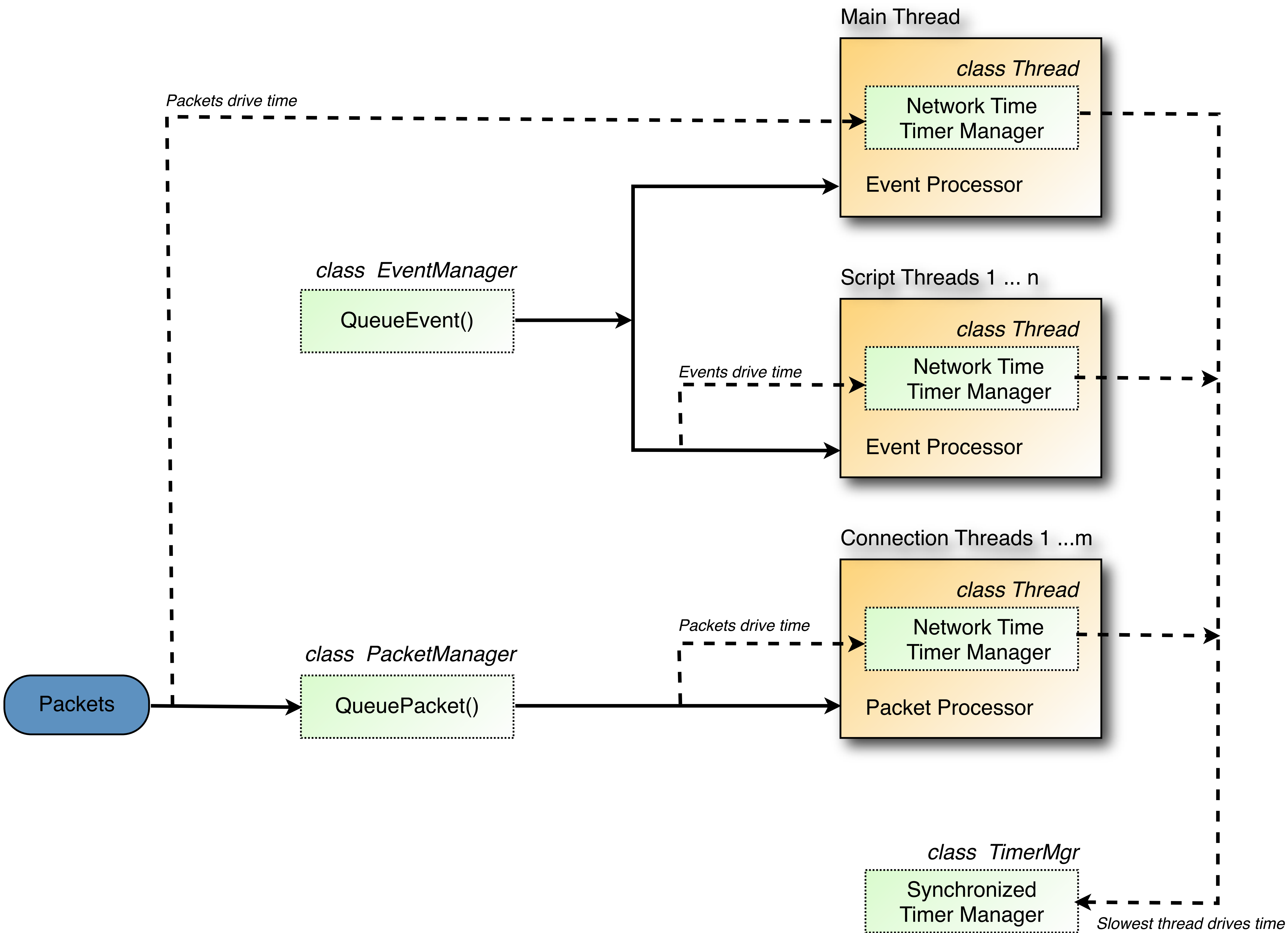


Simulation based on 15 minutes of LBNL traffic (24GB, 50M events)

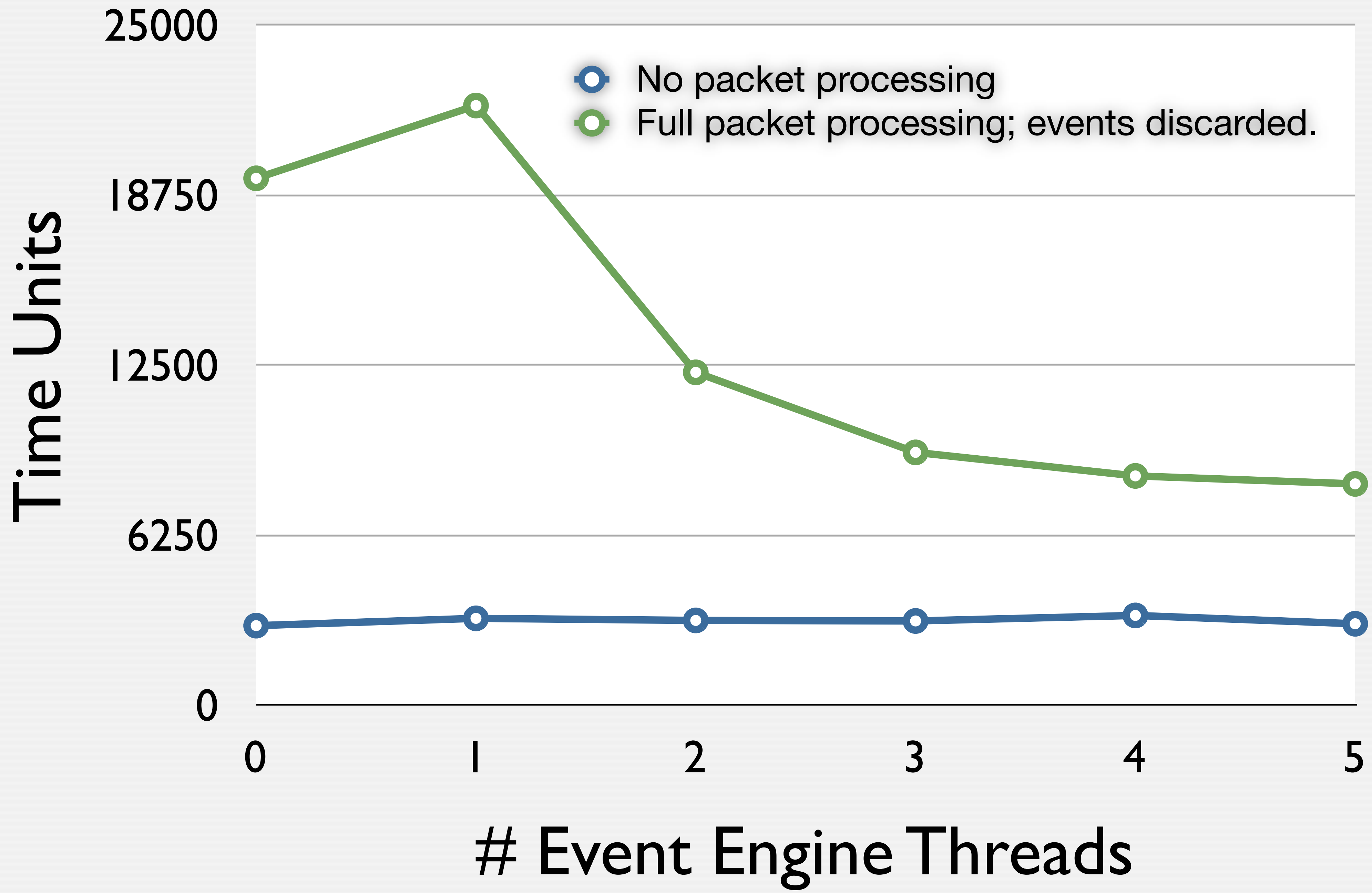
Implementation of Multi-Core Bro

- We have a prototype that we are now profiling further
  - Parallelization based Intel's Threading Building Blocks
  - We do not use on TBB's task concept; just a portable thread abstraction
- Spent a lot of time in making Bro's code thread-safe
  - Extensive use of globals and statics ...
  - Race conditions, e.g., in memory management
  - Not pretty ...
- Assigned scopes to the most important globals
  - Profiling showed which global variables are accessed the most (>100)
  - Surprisingly many are covered with a small set of scopes
  - Some minor script adaptations to observe scoping rules
  - "Real" globals are fully locked

Multi-Core Bro Data Flow

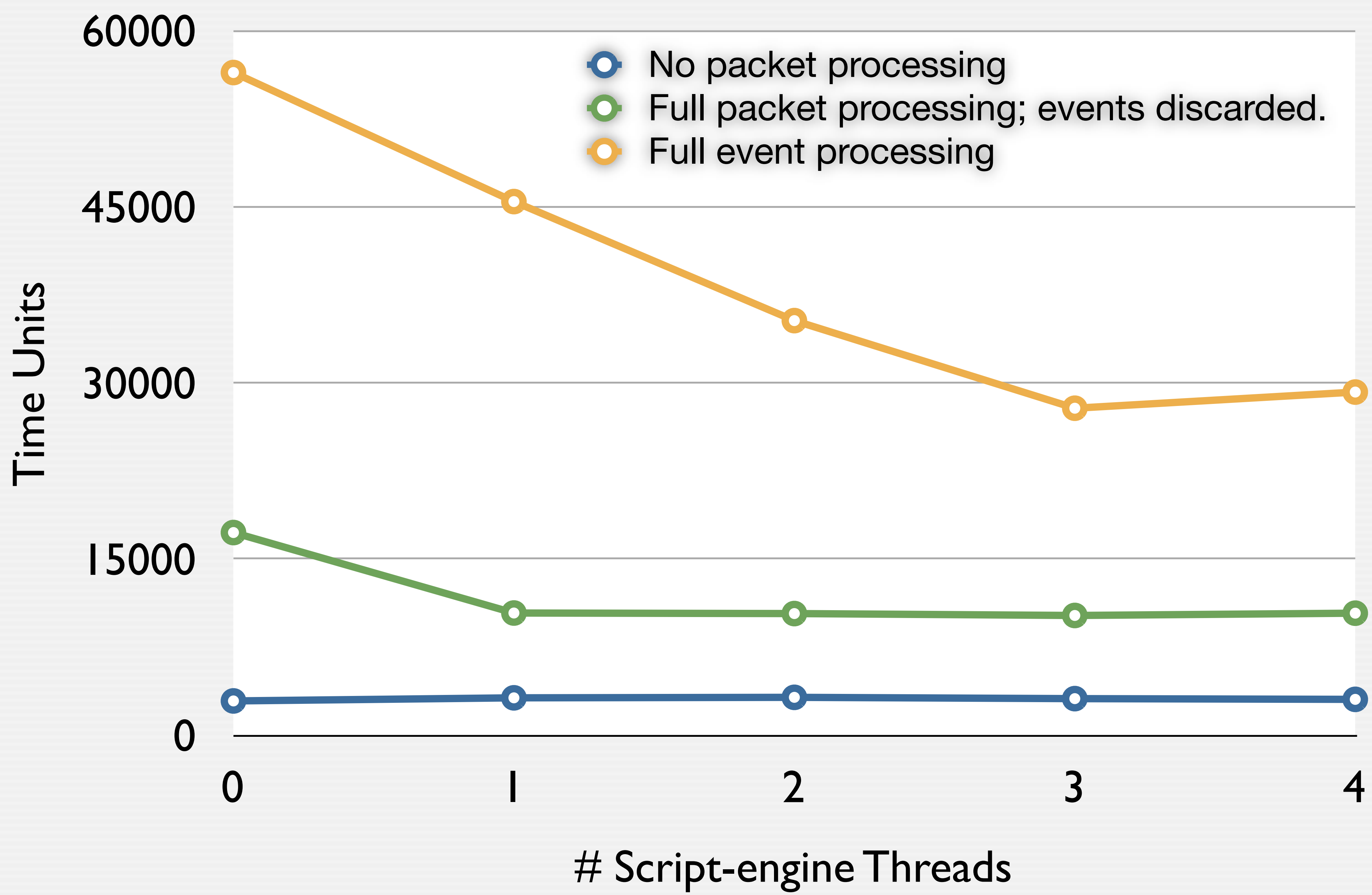


Event Engine Performance So Far ...



Dual quad-core Xeon system with 10min/11GB of LBL traffic
1 script-engine thread

Script-Engine Performance So Far ...



Dual quad-core Xeon system with 10min/11GB of LBL traffic
2 event-engine threads

Future Directions

Understanding the Bottlenecks

- Concurrent execution is just the start
  - “Just” parallelizing execution does not necessarily yield the expected speed-up
- Extensive profiling and optimization
  - CPU usage
  - Impact of the memory hierarchy
  - Explore different scheduling strategies

  - Use well-defined input traffic to understand the effects.
  - Offline vs. online
- Evaluation on different hardware platforms
  - Commodity systems with dual quad-core Xeons
  - 64-core Tilera platform
  - Simics simulator to explore a wide variety of options
- Parallelizing the packet dispatcher
  - Several vendors provides platforms that directly schedule packets to threads
  - Tilera can do that as well

Future: Automate the Scoping

- Scopes are assigned manually in our prototype
  - Not ideal, as it's not completely transparent to the user
- Bro should be able to infer scopes automatcially
  - Static and dynamic analysis of access patterns
- Scoping rules require minor script modifications
- Likewise, Bro could rewrite code internally
  - For example, auto-split event handlers

Future: An Abstract Machine

- Working on an abstract machine for traffic analysis
  - Instruction set with domain-specific support for typical operations
  - Compiler to turn it into highly efficient native code
- Will provide a concurrency abstraction
  - Will work well for the scoping model.
  - Eventually, Bro scripts will be compiled into this execution model

Summary

- We are building a highly-concurrent NIDS
  - Based on the open-source Bro NIDS
- Designed concurrency models for its main components
  - Packet analysis, based on pure per-flow analysis, no state correlation
  - Detection logic, based on scheduling scopes corresponding to processing units
  - Leveraging domain-specific knowledge for parallelization
- Simulations and the Cluster predict excellent performance
- Prototype ready for intensive profiling now
  - Analyzing real-world performance, in particular memory effects
- Optimistic that multi-core Bro will eventually be able to scale to a large number of cores in production environments

ICSI is hiring ...

We have an opening for a post-doc position to work on highly concurrent network traffic analysis.

http://www.icir.org/jobs.html

Thanks for your attention.

Robin Sommer
International Computer Science Institute, &
Lawrence Berkeley National Laboratory

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