Automating Network Monitoring on Experimental Network Testbeds

Michael Golightly, Jack Brassil

Problem

- Experimenters can benefit from additional experiment-wide network monitoring
  - debugging aid for large-scale experiments
  - Malicious flow detection
  - Aids experiment ‘traffic engineering’

- Many monitoring tools require tool-specific expertise (often not found in the student’s toolkit)
- Deploying tools in large-scale experiments manual and tedious
  - Difficult to manage if experiment topologies vary or are dynamically modified
  - Difficult to configure/provision before running experiment

Our Solution Approach

- ***Automated,*** experiment-wide network monitoring tool deployment
- Develop an ***extensible*** deployment framework that can be used for a broad class of monitoring tools
- Give user ***flexible control***
  - monitoring resource consumption (cost)
  - Coverage
  - Data collection granularity
  - Impact on running experiment
- Similar in spirit to Emulab's *trace*, Orbit's Measurement Framework & Library (OML), etc.

NetFlowize

- A tool to deploy NetFlow probes and collectors on Emulab/DETER experiments
  - NetFlow widely used throughout both network systems and security communities
  - Most typically used testbed-wide by provider/operator rather than experiment-wide, e.g., PlanetFlow
  - Uses unmodified, open-source NetFlow components
  - Can be extended to collect data from infrastructure switches and routers (more later)
- Users only specify one of two deployment modes
  - Resource lightweight or heavyweight

Brief NetFlow Backgrounder

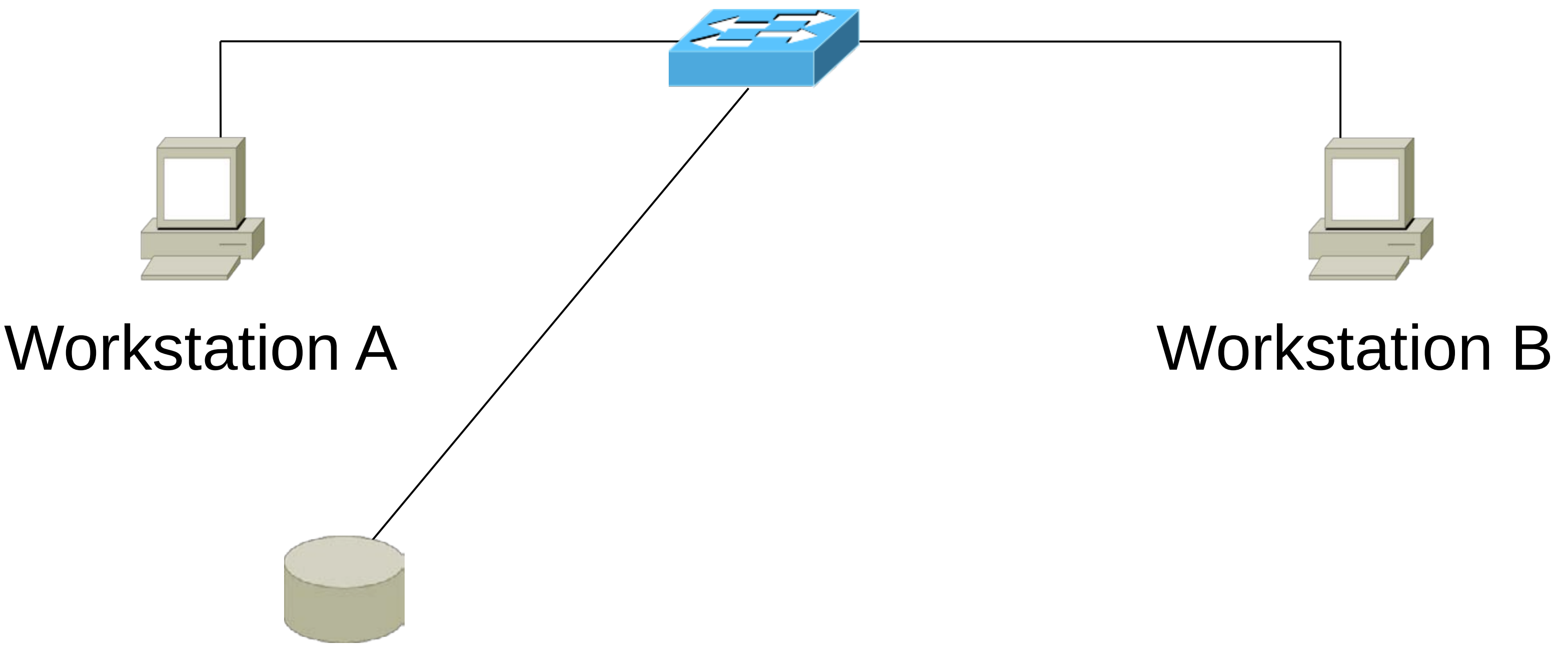
- Flow – unidirectional sequence of packets that are logically associated
  - headers match a specific n-tuple, e.g.

    <src IP, dst IP, src port, dst Port, protocol>
  - Creation and expiration policy – what conditions start and stop a flow

    TCP SYN, TCP FIN, timeouts
- NetFlow counters
  - packets, bytes, time

LABS hp

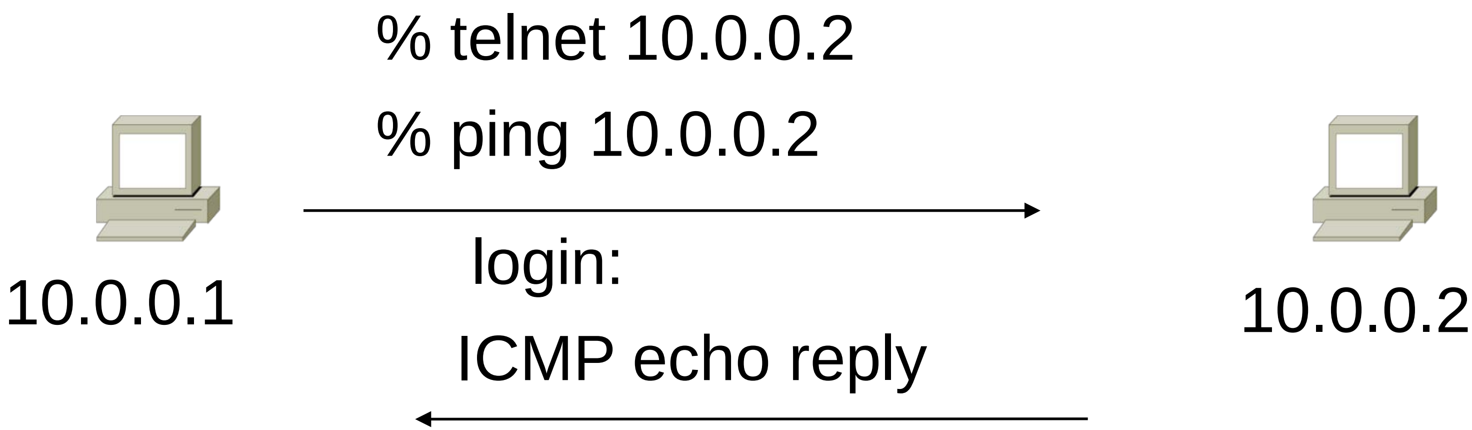
Passive Probe Collection



Flow probe connected

to switch port in “traffic mirror” mode

Simple Flow Report



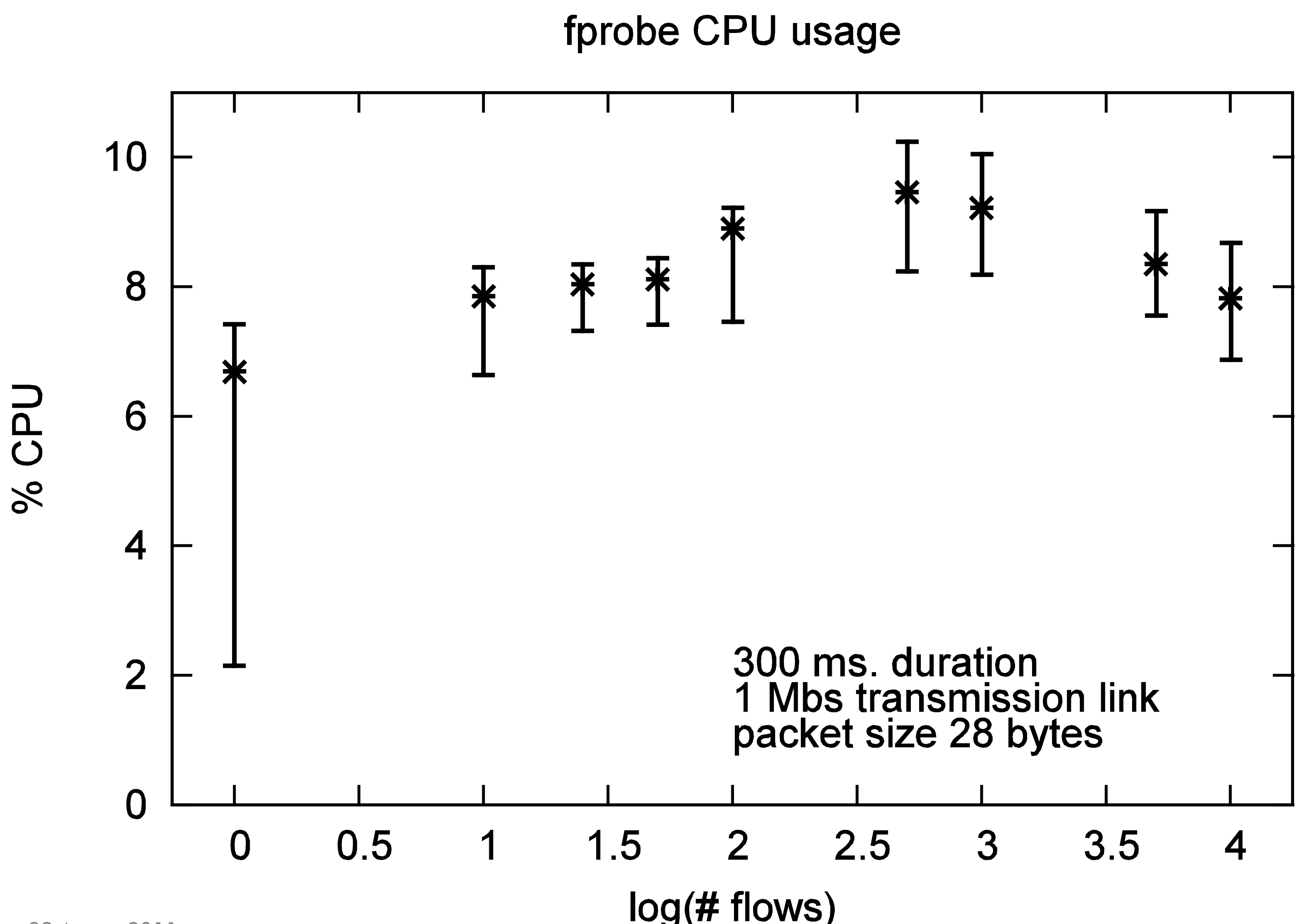
Active Flows

Flow	Source IP	Destination IP	prot	srcPort	dstPort
1	10.0.0.1	10.0.0.2	TCP	32000	23
2	10.0.0.2	10.0.0.1	TCP	23	32000
3	10.0.0.1	10.0.0.2	ICMP	0	0
4	10.0.0.2	10.0.0.1	ICMP	0	0

Monitoring Overhead

- client <-> monitor <-> server
- monitor acting as bridge between client and server
- client flooding 28 byte UDP packets to server
- Emulab PC850 machines
  - 850MHz Intel Pentium III processor.
  - 512MB PC133 ECC SDRAM.
  - Intel EtherExpress Pro 10/100Mbps NIC (10 Mbs)
- CPU overhead of building flow records

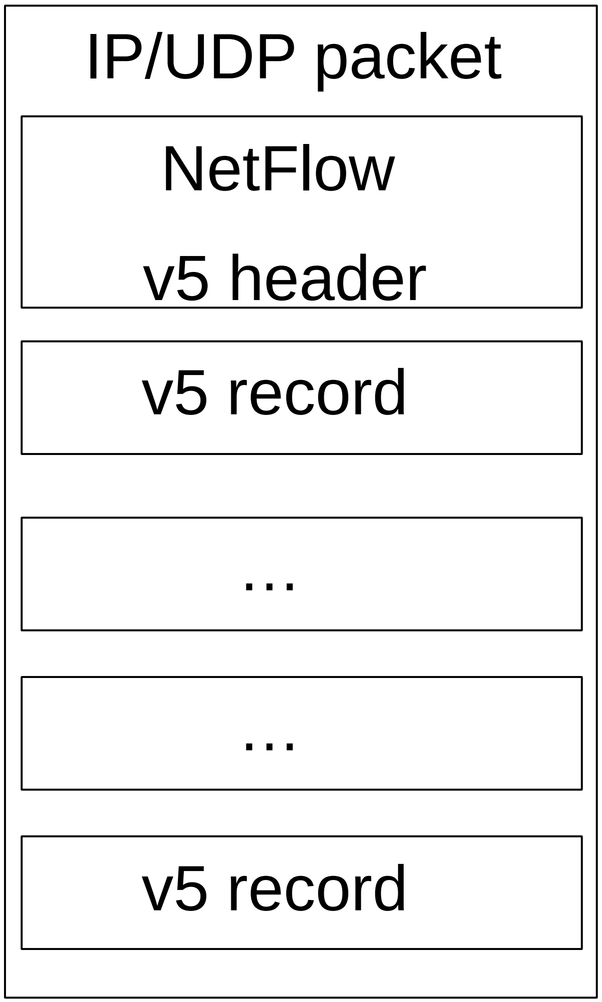
Fprobe CPU usage (PC850, 10 Mbs)



Working with Flows

- Building flow records from packets
  - Probes
    - Software: fprobe
    - Hardware: switches & routers
- Collecting and aggregating flow records
  - Collectors (Unix end hosts)
    - flow-tools, SiLK, …
- Analyzing flow records
  - flow-tools, SiLK, ntop, …
  - Traffic mix, DDoS attacks, port scans, …

NetFlow v5 Packet Example



- UDP packets
- 24 byte header
- 48 byte flow record
- 1-30 records in 1500 byte frame

NetFlow v5 Packet Header

```
struct ftpdu_v5 {
  /* 24 byte header */
  u_int16 version;          /* 5 */
  u_int16 count;            /* The number of records in the PDU */
  u_int32 sysUpTime;        /* Current time in millisecs since router booted */
  u_int32 unix_secs;        /* Current seconds since 0000 UTC 1970 */
  u_int32 unix_nsecs;       /* Residual nanoseconds since 0000 UTC 1970 */
  u_int32 flow_sequence;    /* Seq counter of total flows seen */
  u_int8  engine_type;      /* Type of flow switching engine (RP,VIP,etc.) */
  u_int8  engine_id;        /* Slot number of the flow switching engine */
  u_int16 reserved;
```

NetFlow v5 Record: Key Fields

```
/* 48 byte payload */
 struct ftrec_v5 {
   u_int32 srcaddr;    /* Source IP Address */
   u_int32 dstaddr;    /* Destination IP Address */
   u_int32 nexthop;    /* Next hop router's IP Address */
   u_int32 dPkts;      /* Packets sent in Duration */
   u_int32 dOctets;    /* Octets sent in Duration. */
   u_int16 srcport;    /* TCP/UDP source port number or equivalent */
   u_int16 dstport;    /* TCP/UDP destination port number or equiv */
   u_int8  tcp_flags;  /* Cumulative OR of tcp flags */
   u_int8  prot;       /* IP protocol, e.g., 6=TCP, 17=UDP, ... */
   u_int8  tos;        /* IP Type-of-Service */
   u_int16 drops;
         .
         .
         .
 } records[FT_PDU_V5_MAXFLOWS];
};
```

Experiment View by Protocol

# protocol	flows	octets	packets	duration
tcp	93.877	97.143	93.326	91.589
udp	4.257	2.466	5.932	8.286
icmp	1.337	0.368	0.576	0.117
gre	0.010	0.002	0.006	0.005
pim	0.012	0.002	0.004	0.001
ipv6	0.004	0.000	0.001	0.000
igmp	0.000	0.000	0.000	0.000
ospf	0.001	0.000	0.000	0.000
rsvp	0.000	0.000	0.000	0.000

Summary View of Experiment Run

```
Total Flows                           : 24236730
Total Octets                          : 71266806610
Total Packets                         : 109298006
Total Time (1/1000 secs) (flows): 289031186084
Duration of data  (realtime)          : 86400
Duration of data (1/1000 secs)        : 88352112
Average flow time (1/1000 secs)       : 11925.0000
Average packet size (octets)          : 652.0000
Average flow size (octets)            : 2940.0000
Average packets per flow              : 4.0000
Average flows / second (flow)         : 274.3201
Average flows / second (real)         : 280.5177
Average Kbits / second (flow)         : 6452.9880
Average Kbits / second (real)         : 6598.7781
```

Netflowize tool

- Automatically determines where to place Netflow probes and collectors
- Leverages underlying physical network topology
- Relies on persistent resource assignment across experiment swaps
- Configurable
  - *Lightweight*: Use existing experimental infrastructure
  - *Heavyweight*: Deploys monitoring infrastructure overlay using additional experimental resources

Naïve Approach to Overlay Creation

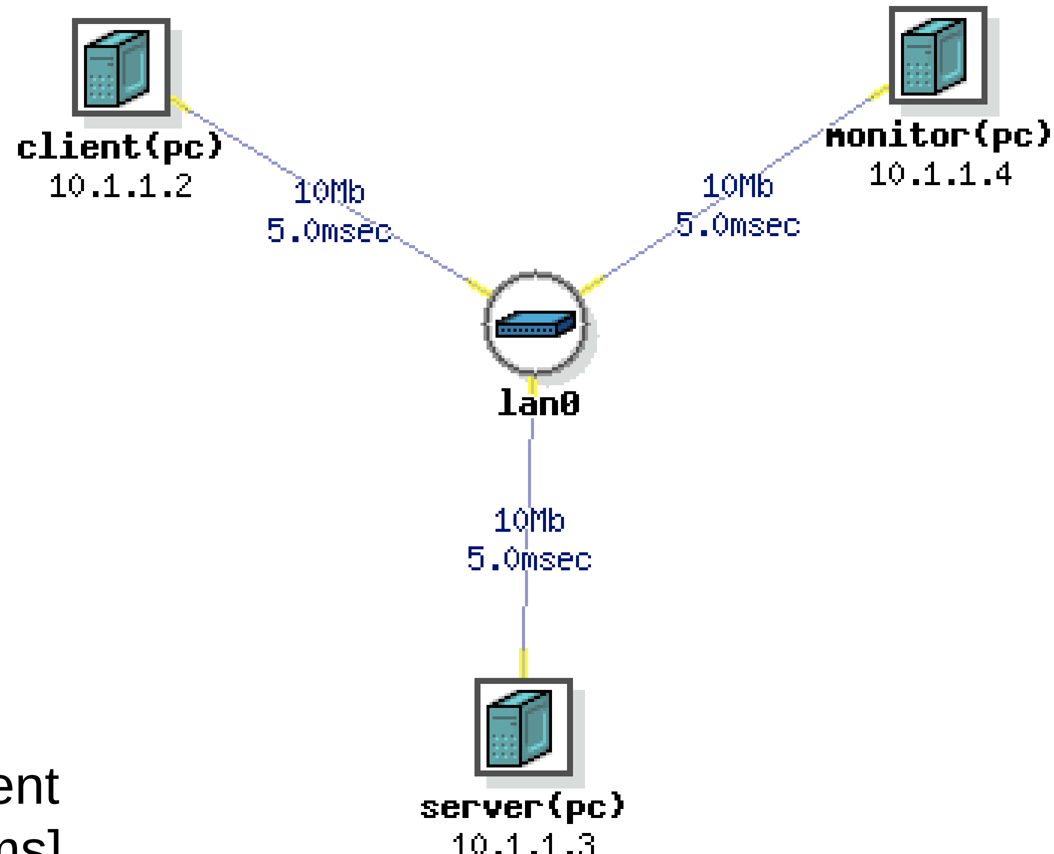
- Analyze ns topology description
- Modify toplogy description to add overlay nodes, links, and NetFlow software probes and collectors
- Swap experiment out and back in

Do this and watch bad things happen …

Example: 3 node experiment

```
set ns [new Simulator]
source tb_compat.tcl
# Create nodes
set client [$ns node]
set server [$ns node]
set monitor [$ns node]
# Create lan
set lan0 [$ns make-lan "$client $server $monitor" 10Mb 10ms]

$ns run
```



Logical view of topology

Physical Experiment Topology

Node ID	Name	Type	Default OSID	Node Status	Hours Idle[1]	Startup Status[2]	SSH	Console	Log
pc86	server	pc850	RHL90-STD	up	0.1	none			
pc93	tbdelay0	pc850	FBSD410-STD	up	0.13	0			
pc103	tbdelay2	pc850	FBSD410-STD	up	0.13	0			
pc152	monitor	pc850	RHL90-STD	up	0.1	none			
pc164	client	pc850	RHL90-STD	up	0.1	none			

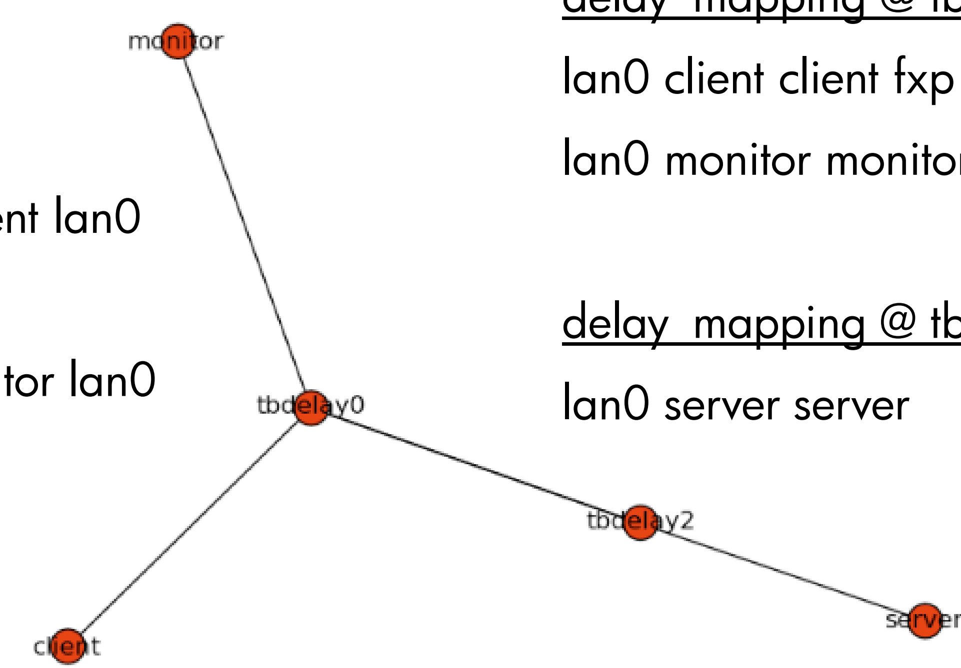
<u>ltp_map</u>

L client monitor lan0

L client server lan0L monitor client lan0

L monitor server lan0

L server client lan0L server monitor lan0



<u>delay_mapping @ tbdelay0</u>

lan0 client client fxp1 fxp4

lan0 monitor monitor fxp2 fxp3

<u>delay_mapping @ tbdelay2</u>

lan0 server server

NS Topology Description: Example 1

```
$ns duplex–link [ $ns node ] [ $ns node ]\\
    10Mb 0 ms DropTail
```

- Perfectly valid topology (just bad form)
- Emulab will fill in unspecified details
  - Create 2 nodes running the default operating system
  - assign the nodes' names (e.g., tbnode-n1, tbnode-n2)
  - name the connecting link (e.g., tblink-l3)

Difficult to parse and modify topology

NS Topology Description: Example 2

```
# create nodes
for { set i 0 } { $i < 2 } { incr i } {
  set node ( $i ) [ $ns node ]
  tb–set–node–os $node ( $i ) FBSD410–STD
}
# create link
set link0 [ $ns duplex–link $node ( 0 ) $node ( 1 )\\
          10Mb 0 ms DropTail
```

A more common form, still difficult to parse

Solution: Post-instantiation experiment modification

- Get exported physical topology details via XML-RPC
- Might be necessary to ssh into nodes for attached link details
- Construct physical topology graph

Much easier to parse and modify topology using the minimum number of resources

Overlay Construction

Lightweight mode:

- Probe Placement
  - 'set cover' type algorithm to identify minimum number of probes to deploy
- Collector Placement
  - pick a node at random (easy)
  - use control network for record distribution (ideally dedicated measurement network)

Overlay Construction

Heavyweight mode:

- Probe Placement
  - replace each *link* with LAN + node for probe
  - attach new dedicated node to *lossless LAN*
  - use existing nodes for *lossy LANs*
- Collector Placement
  - create a new dedicated node
  - use control network for record distribution

Tricks

- Lightweight mode favors putting probes on shaper (delay) nodes to minimize impact on experimental nodes
- Heavyweight mode takes advantage of Emulab's trace to deploy nodes
- Modifications tagged so they can be automatically stripped from experiment

Current Status

- ~700 lines of python
- Grab tool at

  http://66.92.233.103/netflowize-0.3.tar.bz2

Future Work

- Instrumented experiment should be checked for duplicates, unnecessary hardware resources, incomplete coverage
- Inadequate handling of infeasible requests
- More control knobs?
- Virtual node handling?
- Integrate more efficient probe
- Extensions beyond NetFlow
- Integration into existing workbenches
- Multi-tenant cloud monitoring?