Archipelago
Measurement Infrastructure

Updates and Case Study

Young Hyun
CAIDA

ISMA 2010 AIMS Workshop
Feb 9, 2010

Outline

- Introduction
- Monitor Deployment
- Measurements & Collaborations
- Tools Development
- Case Study
- Future Work

Introduction

- Archipelago (Ark) is CAIDA's active measurement infrastructure
  - in production since Sep 2007
- focusing on
  - easy development and rapid prototyping
  - dynamic and coordinated measurements
  - measurement services (service-oriented architecture)
- please see AIMS'09 talk for greater details

Architecture

* measurement nodes ("monitors") located worldwide
  * standard rack-mounted servers
  * many thanks to the organizations hosting Ark boxes
  * special thanks for finding hosting sites:
    - Emile Aben (RIPE)
    - Sebastian Castro Avila (.nz Registry Services)
    - Hyunchul Kim (Seoul National University)

Monitor Deployment

* 41 monitors in 25 countries

	Continent
17	North America
2	South America
14	Europe
1	Africa
5	Asia
2	Oceania

	Organization	
21	academic	3/4 academic
10	research network	
5	network infrastructure	1/4 commercial
4	commercial network	
1	community network	

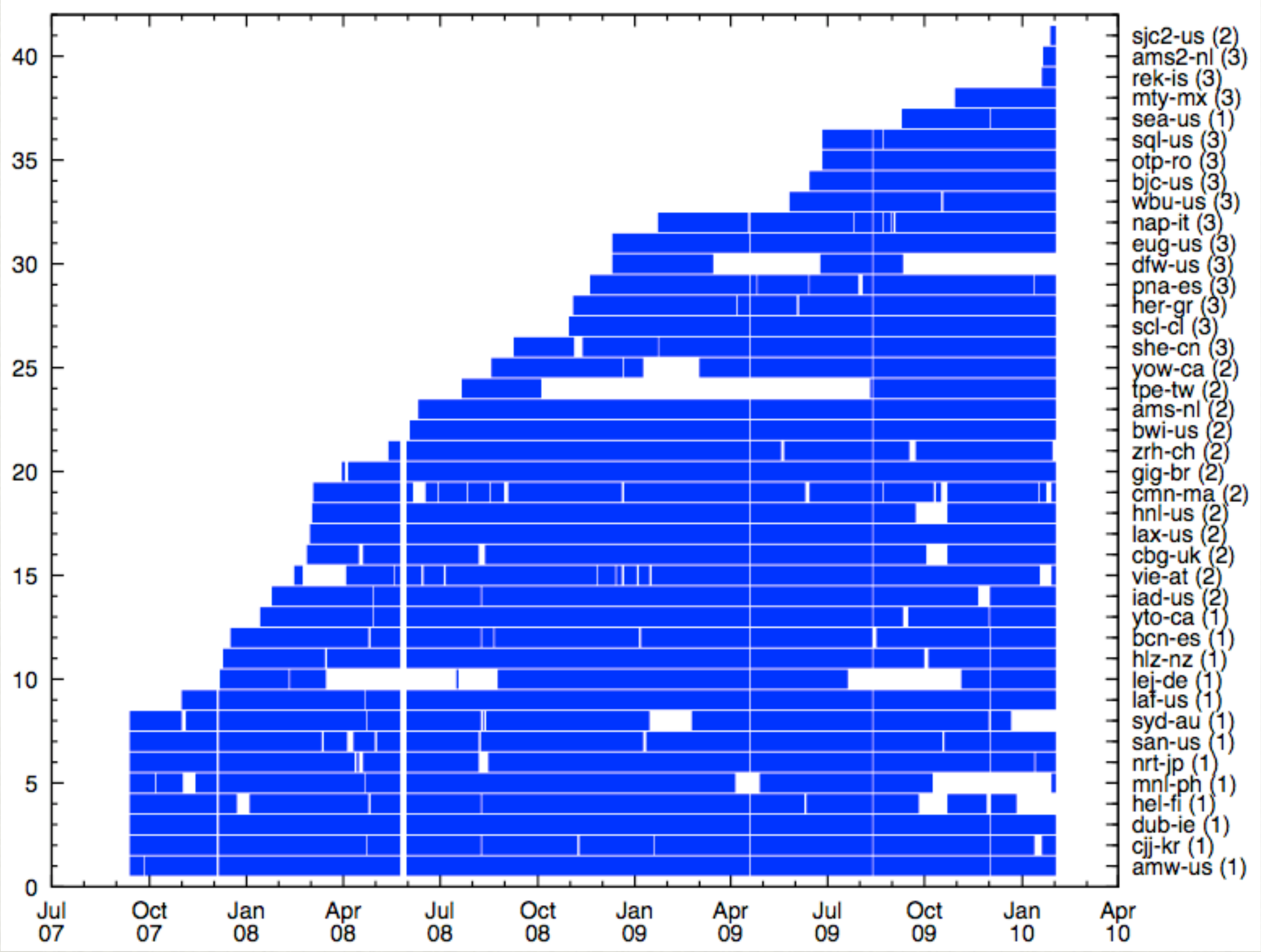
Measurements

* IPv4 Routed /24 Topology (and AS Links)
* IPv6 Topology
* DNS Names & Query/Response Traffic
* Alias Resolution

IPv4 Routed /24 Topology

- ongoing large-scale topology measurements
  - ICMP Paris traceroute to every routed /24 (8.25 million)
    - about 126 /8-equivalents of routed space (as of Oct 2009)
  - running *scamper*
    - written by Matthew Luckie of WAND, University of Waikato
- dynamically divide up the measurement work among members of monitor teams
  - 3 teams active
  - 13-member team probes every /24 in 2-3 days at 100pps
    - only one monitor probes each /24 per cycle (== one pass through all /24's)

IPv4 Routed /24 Topology



data availability per monitor (row)

IPv4 Routed /24 Topology

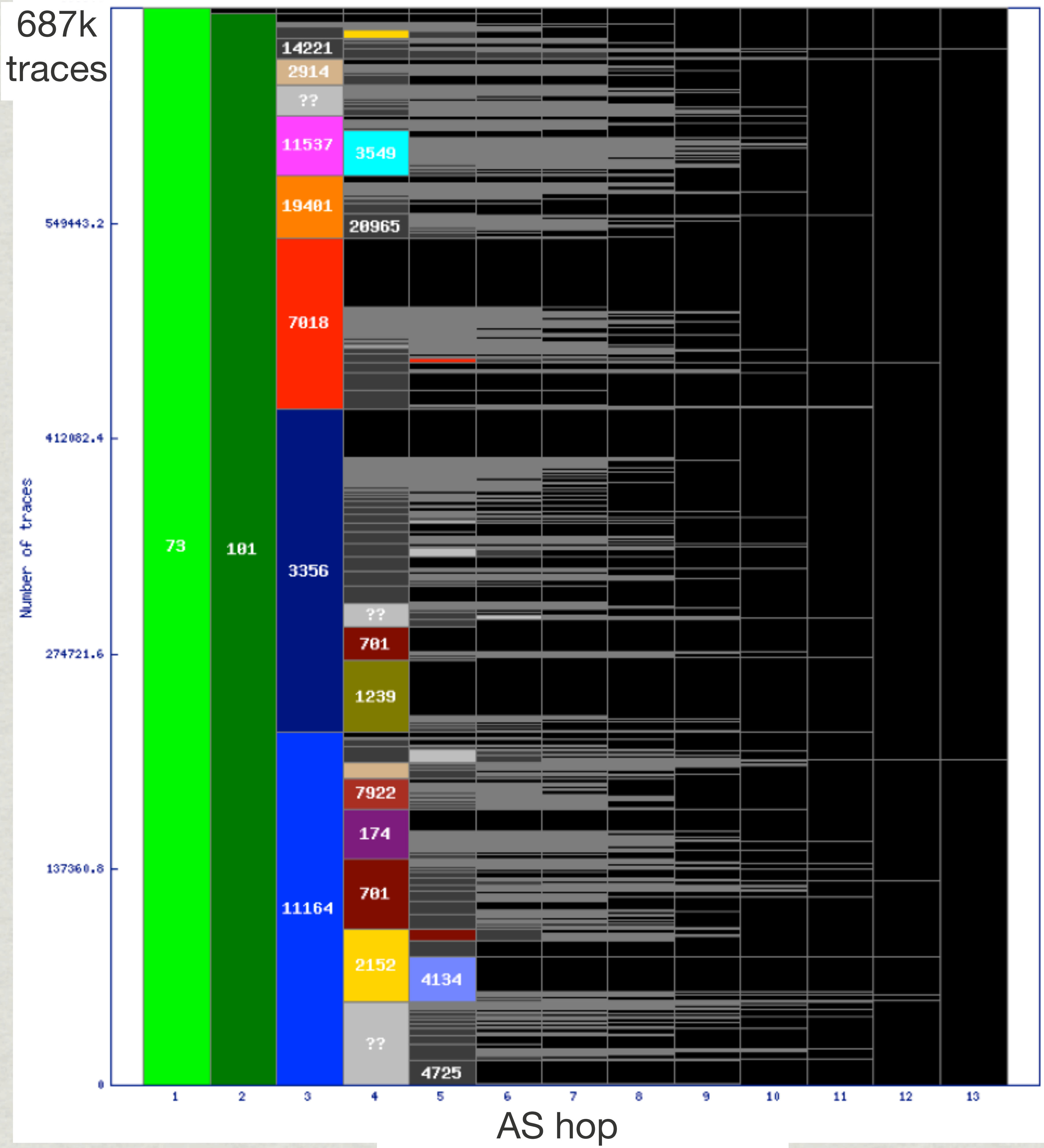
- collected from Sep 2007 to Jan 2010 (29 months):
  - 5.7 billion traceroutes; 2.3TB data
  - ~800 cycles
- collecting every month now:
  - ~290 million traceroutes; ~120 GB data
- IPv4 topology data is key input into other datasets
  - e.g., AS links and alias resolution

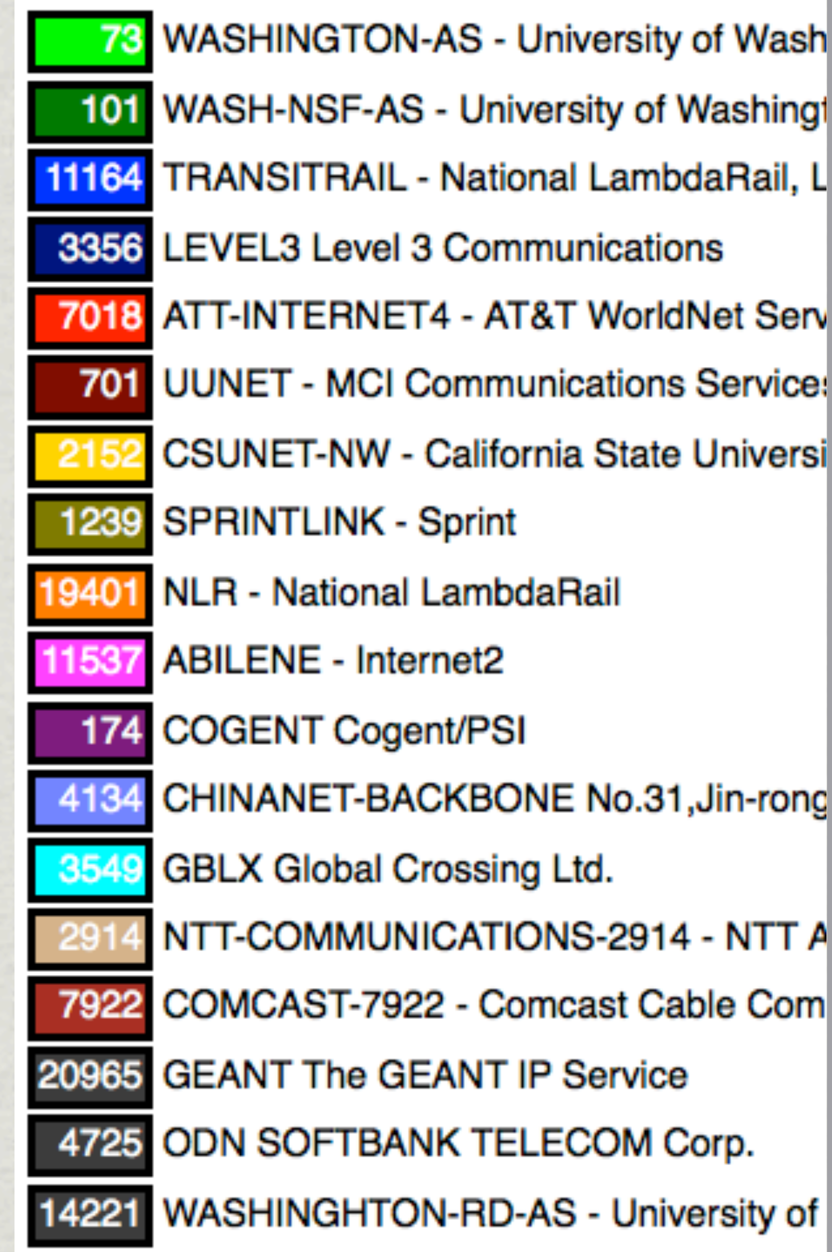
Statistics Pages

* per-monitor analysis of IPv4 topology data

www.caida.org/projects/ark/statistics

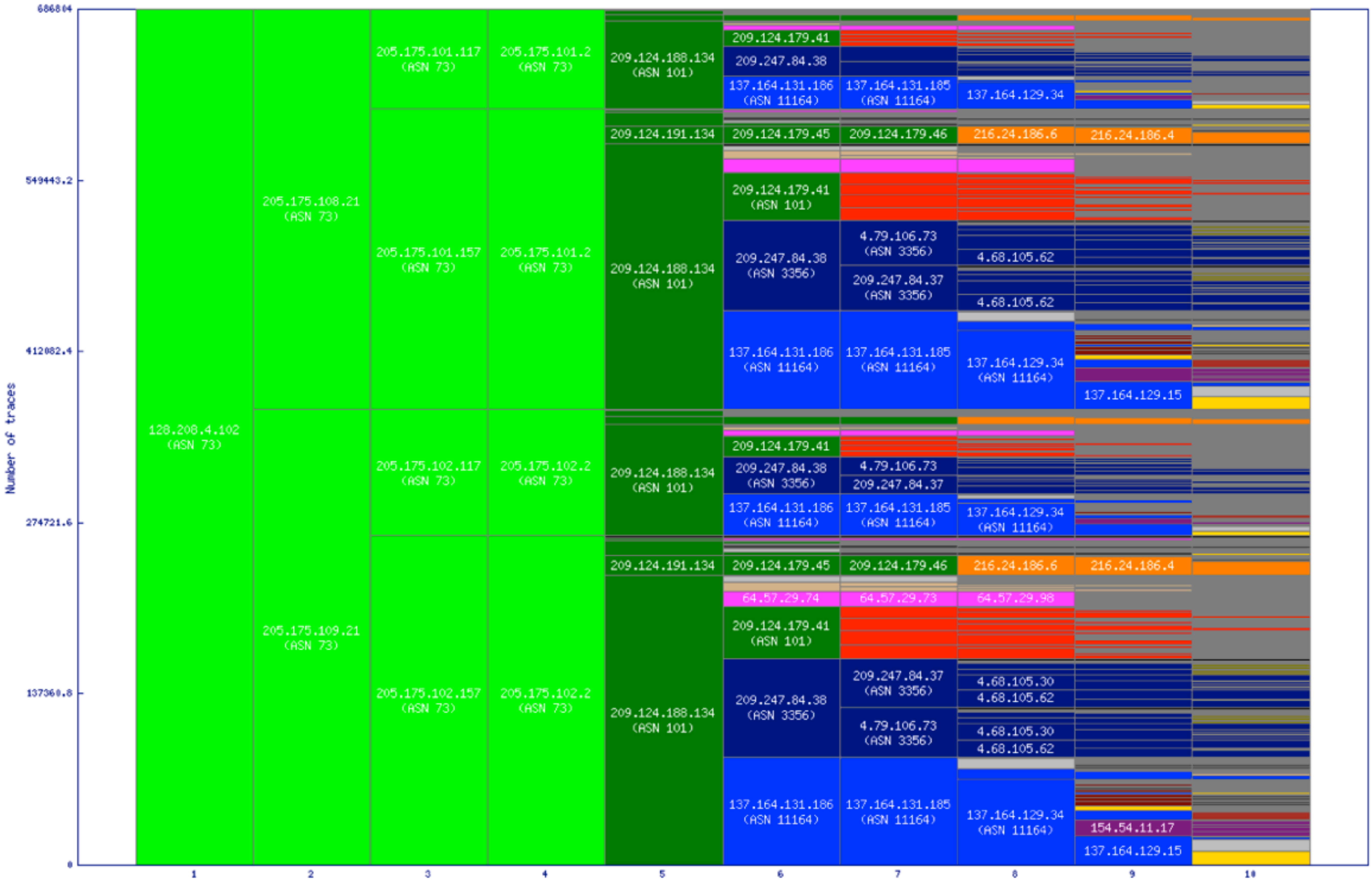
AS dispersion by AS hop





sea-us monitor

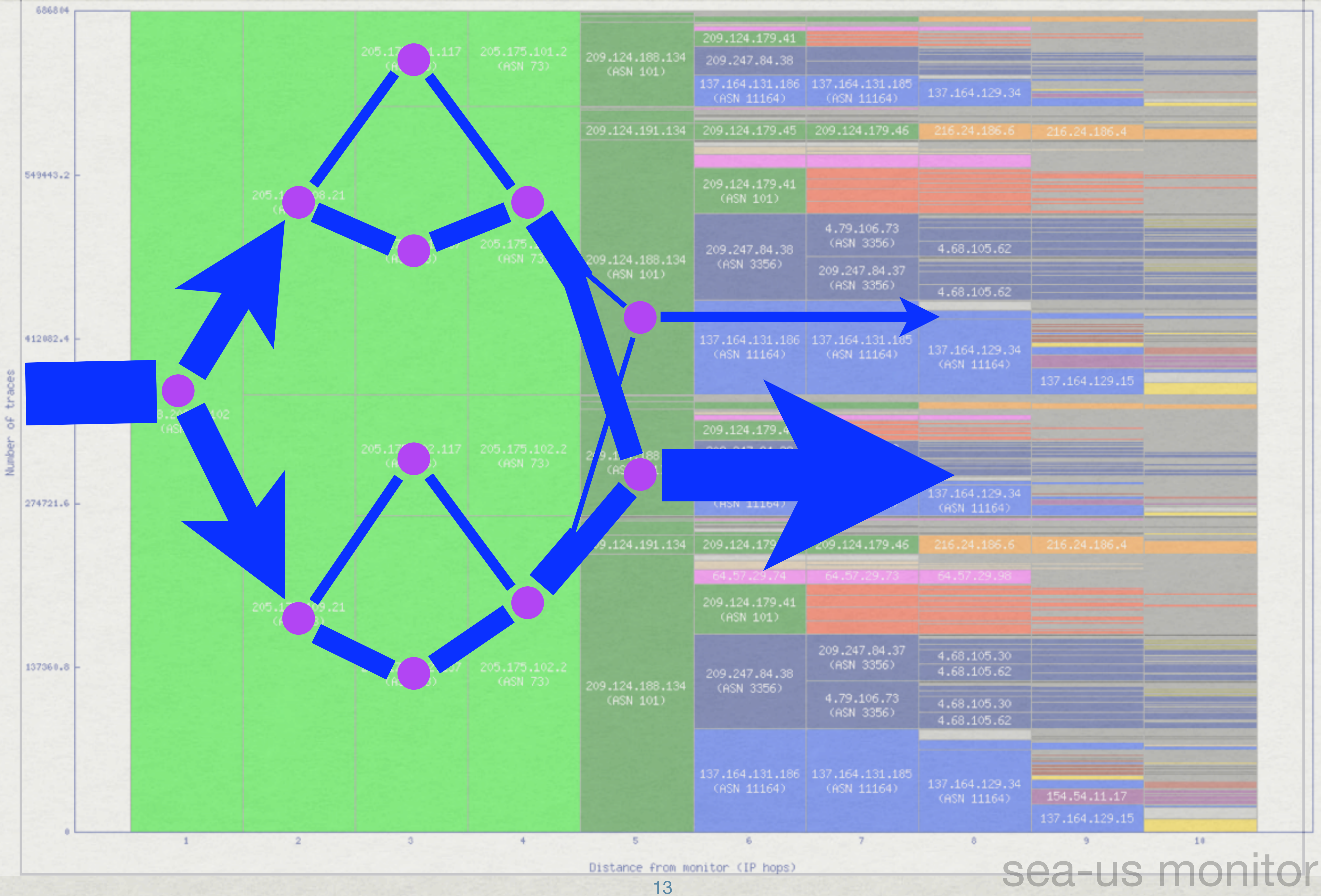
AS dispersion by IP hop



IP hop

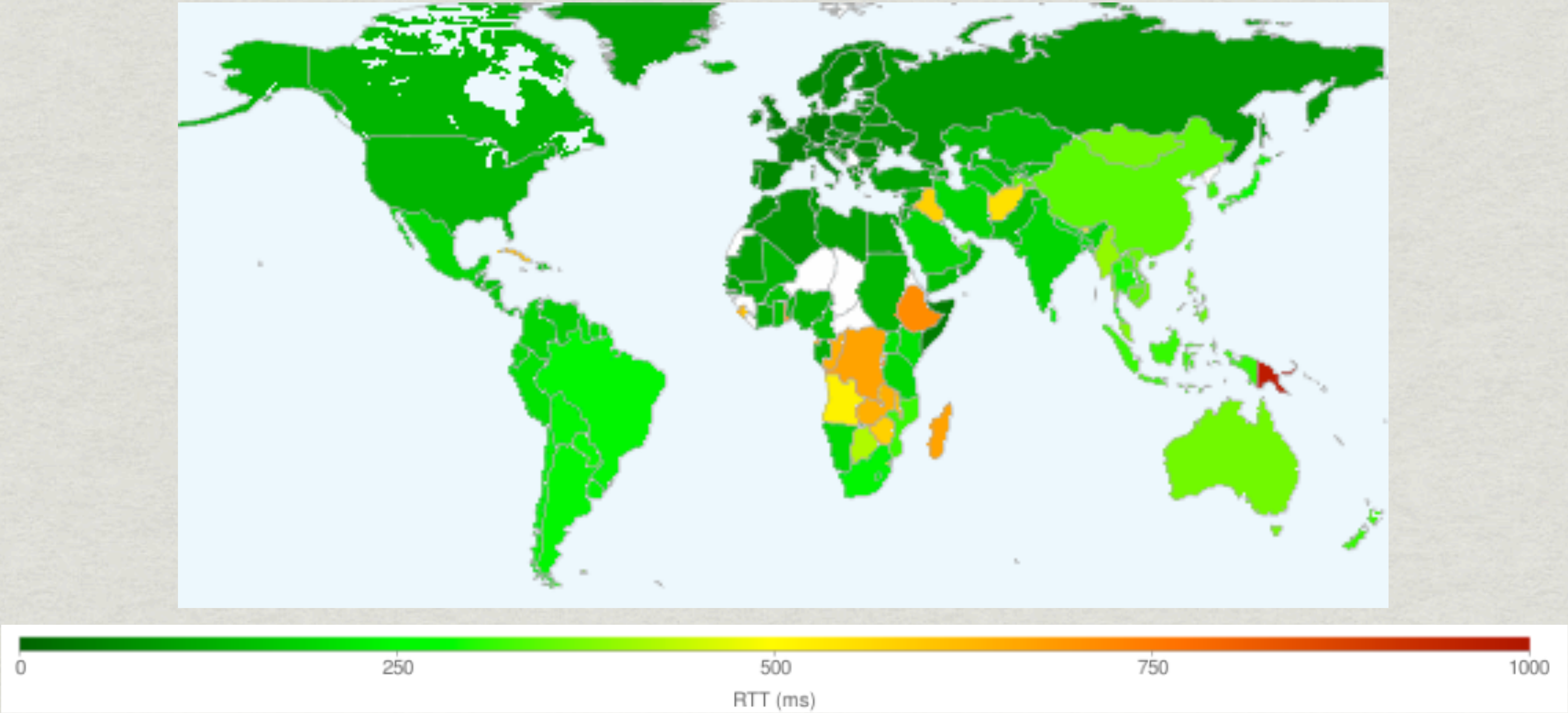
sea-us monitor

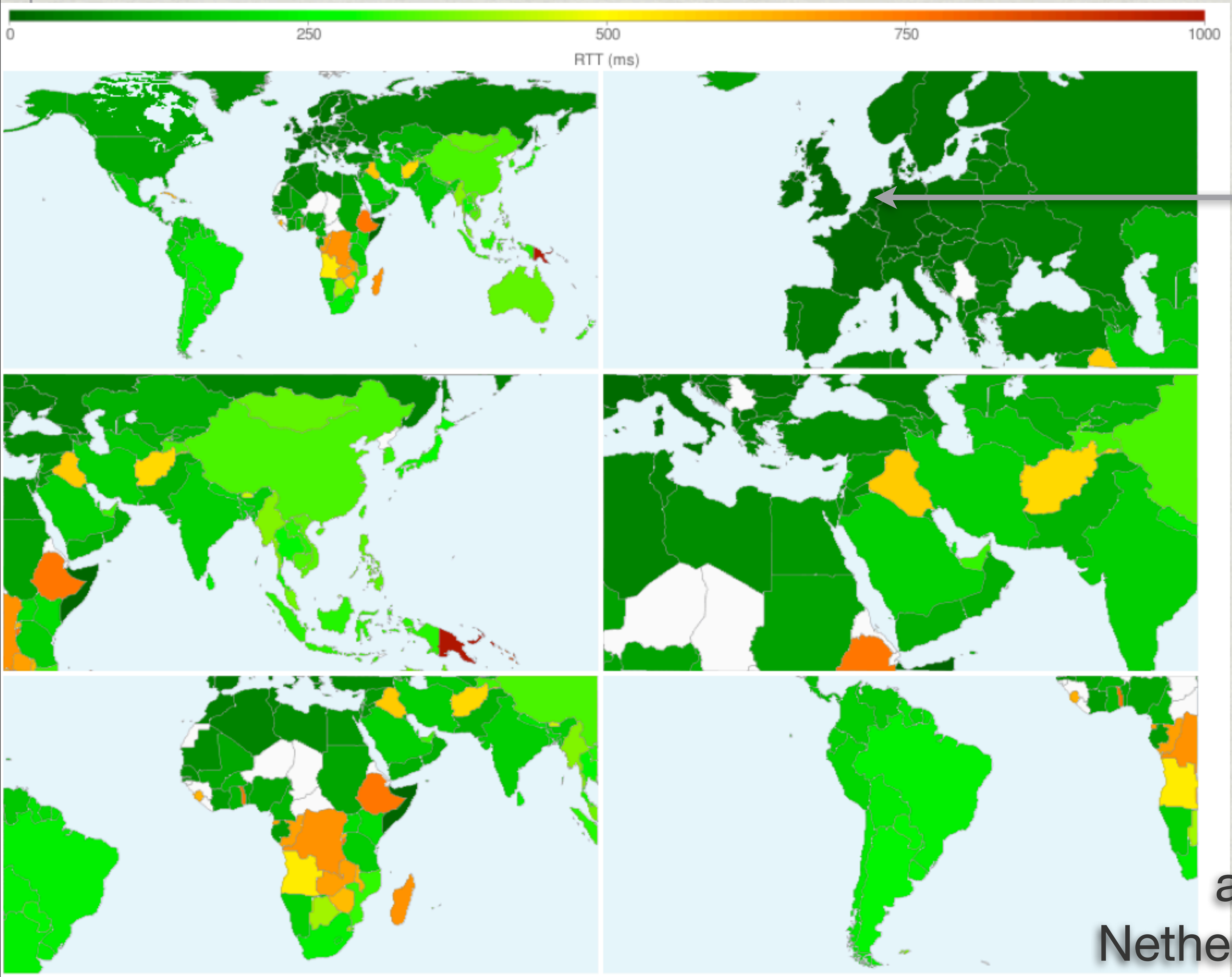
AS dispersion by IP hop: see load balancing



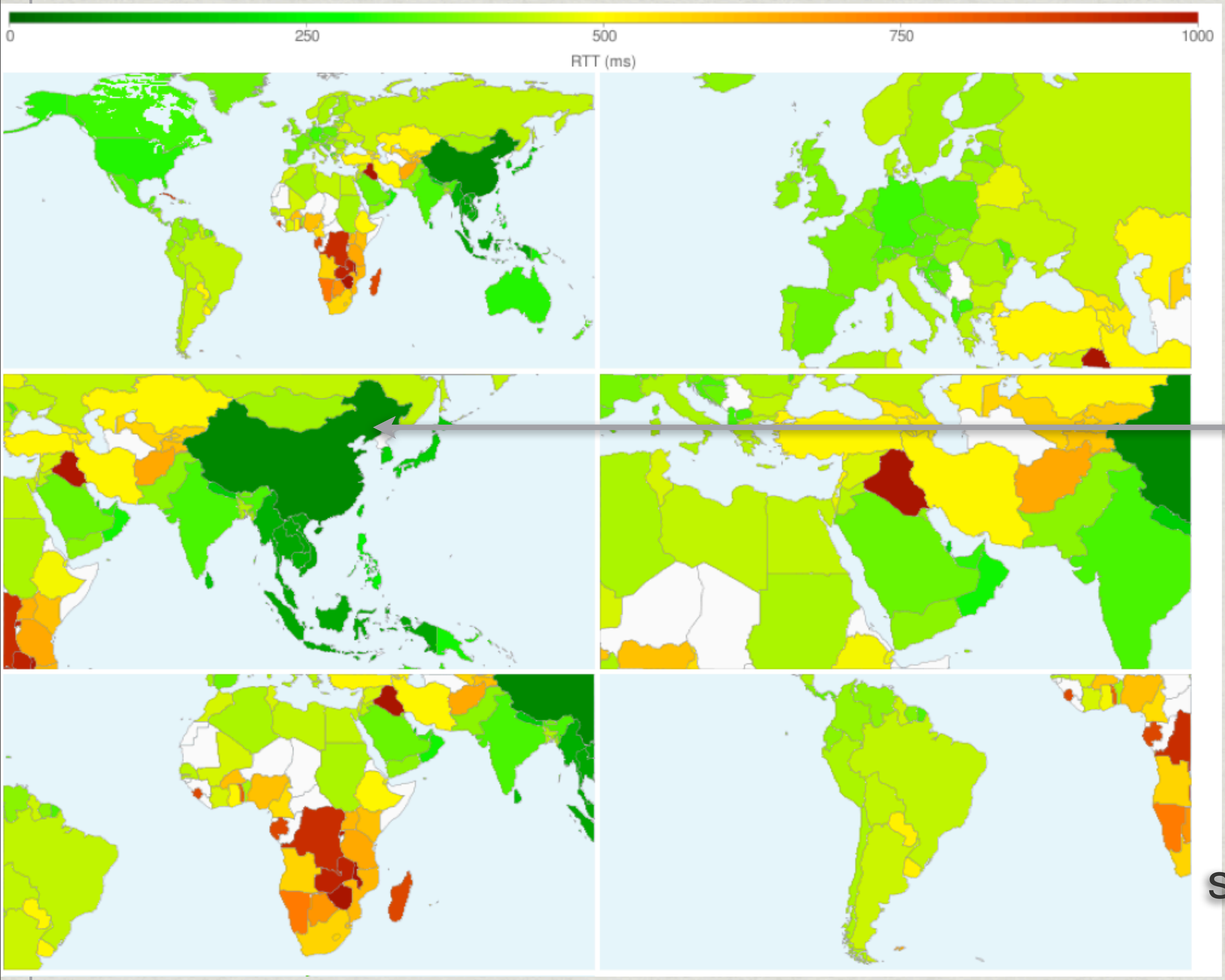
Statistics Pages

- work in progress: RTT plotted by country
  - geolocate destinations with NetAcuity
  - color each country by median RTT of destinations

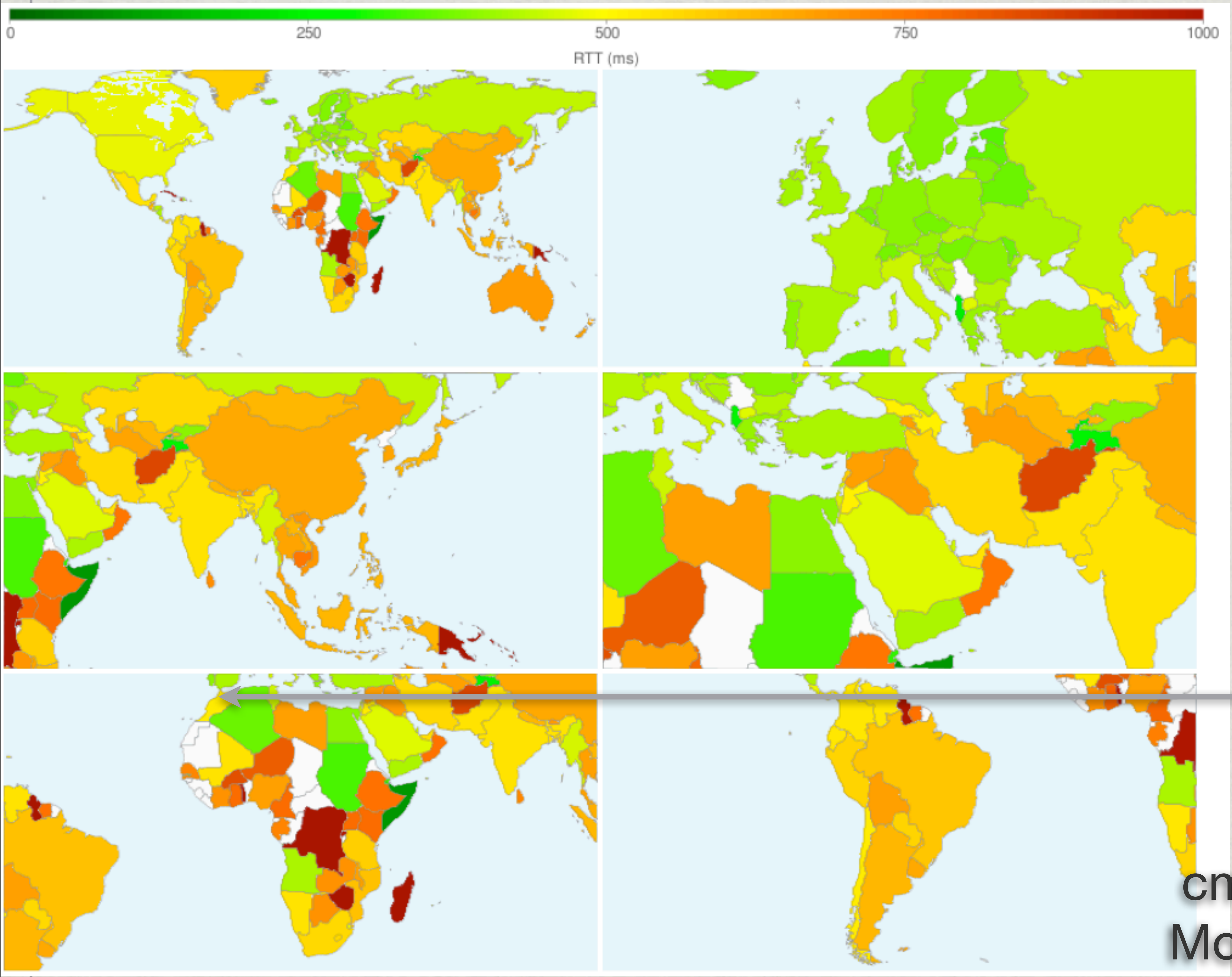




view
from
ams-nl
Netherlands



view from she-cn China



view
from
cmn-ma
Morocco

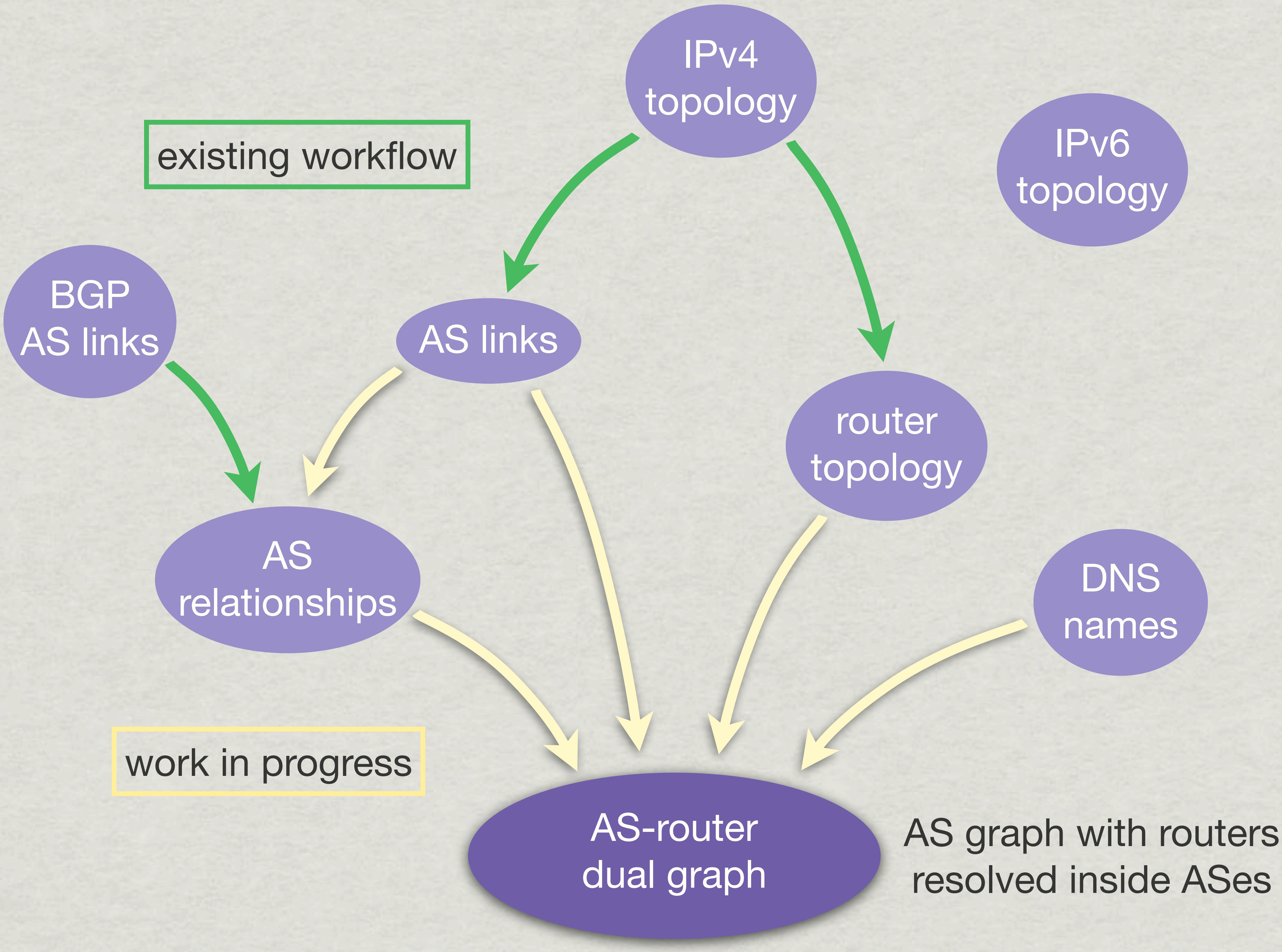
IPv6 Topology

- ongoing large-scale IPv6 measurements
  - 2.7 million traces since Dec 2008
- 11 monitors
  - 4 in US, 5 in Europe, 1 Asia, 1 Oceania
- ICMP Paris traceroute to every routed prefix
  - each monitor probes a random destination in every routed prefix in every cycle
    - 2,184 prefixes <= /48 (as of Oct 2009)
    - # prefixes increased 41% between Aug 2008 and Oct 2009
  - probing rate intentionally reduced to 2 days per cycle

Alias Resolution

- goal: collapse interfaces observed in traceroute paths into routers
  - toward a router-level map of the Internet
- earlier efforts at CAIDA:
  - iffinder (Mercator technique)
  - kapar (APAR)
- past year: MIDAR
  - RadarGun-like approach
    - probe targets to obtain IP ID samples
    - find targets that share an IP ID counter

Measurement Big Picture



Collaborations

- Rob Beverly and MIT Spoofer Project
  - how many networks allow packets with spoofed IP addresses to leave their network?
  - worked on adding IPv6
    - some work still to do before deployment
- Matthew Luckie
  - using Ark monitors for various topology measurements
  - Alistair King
    - masters student supervised by Matthew
    - implemented Doubletree using Marinda (tuple space)
      - Doubletree was one of the motivations for adopting the tuple space model of coordination in Ark

Tools Development

- *mper*
  - new probing engine
  - inspired by the probing engine of Scriptroute
    - but different needs & goals => different design & implementation
  - mper's goal:
    - make it easy to develop complex, distributed, and parallel measurements
      - to be clear: mper itself doesn't provide distributed measurements but provides features oriented towards it
        - clients use the Marinda tuple space for distributed measurements

Tools Development

- *mper*
  - based on the solid foundation of Matthew Luckie's *scamper*
    - uses the code from the backend of scamper
      - sending/receiving ICMP, UDP, TCP packets; IPv4 & IPv6
      - scheduling parallel probes, etc.
  - new control interface for use by client measurement programs
  - new probe-response matching techniques
  - fine control over probe spacing for dynamic feedback-based measurements
  - simulated probing
    - currently, simulated response delay

mper

* new probe-response matching techniques
  * guarantees no probe-response mismatches in any consecutive 65,536 packets (in worst case)
    * not just low probability; simply impossible
    * even with same (src, dest, proto, sport, dport) for all probes
    * multiple probers can run simultaneously without interference
  * preserves flow labels for load balancing
  * works for all probing methods (ICMP, UDP, and TCP) and all types of responses (e.g., TCP ACK)
    * TCP was especially susceptible to mismatches before
  * doesn’t rely on UDP checksums being preserved
    * older FreeBSD clobber the UDP checksum in responses (thanks to Matthew for fixing the FreeBSD kernel)
    * also problem in other older systems

mper

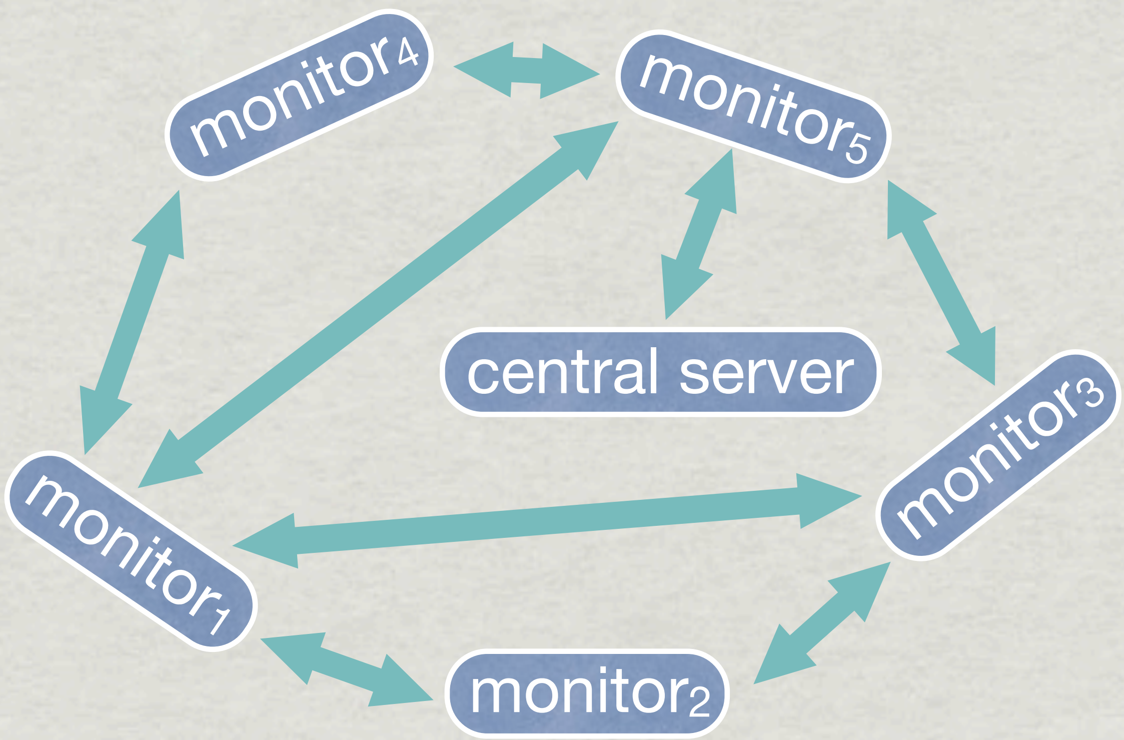
- mper client can be written in any language
- Ruby binding: *rb-mperio*

```ruby
require 'mperio'
class Prober
  def initialize
    @mperio = MperIO.new 8742
    @mperio.delegate = self
    @mperio.ping_icmp 1, "192.172.226.123"
    @mperio.start
  end

  def mperio_on_data(result)
    if result.responded?
      printf "%d %d\n", result.rx_sec, result.reply_ipid
    end
    @mperio.stop
  end
end
```

Tools Development

- *Marinda*
  - tuple space for decentralized communication, interaction, and coordination
    - *tuple*: array of values (strings, numbers, true/false, wildcard, nested arrays)
  - a distributed shared memory + easy-to-use operations
    - clients retrieve tuples by pattern matching



Case Study

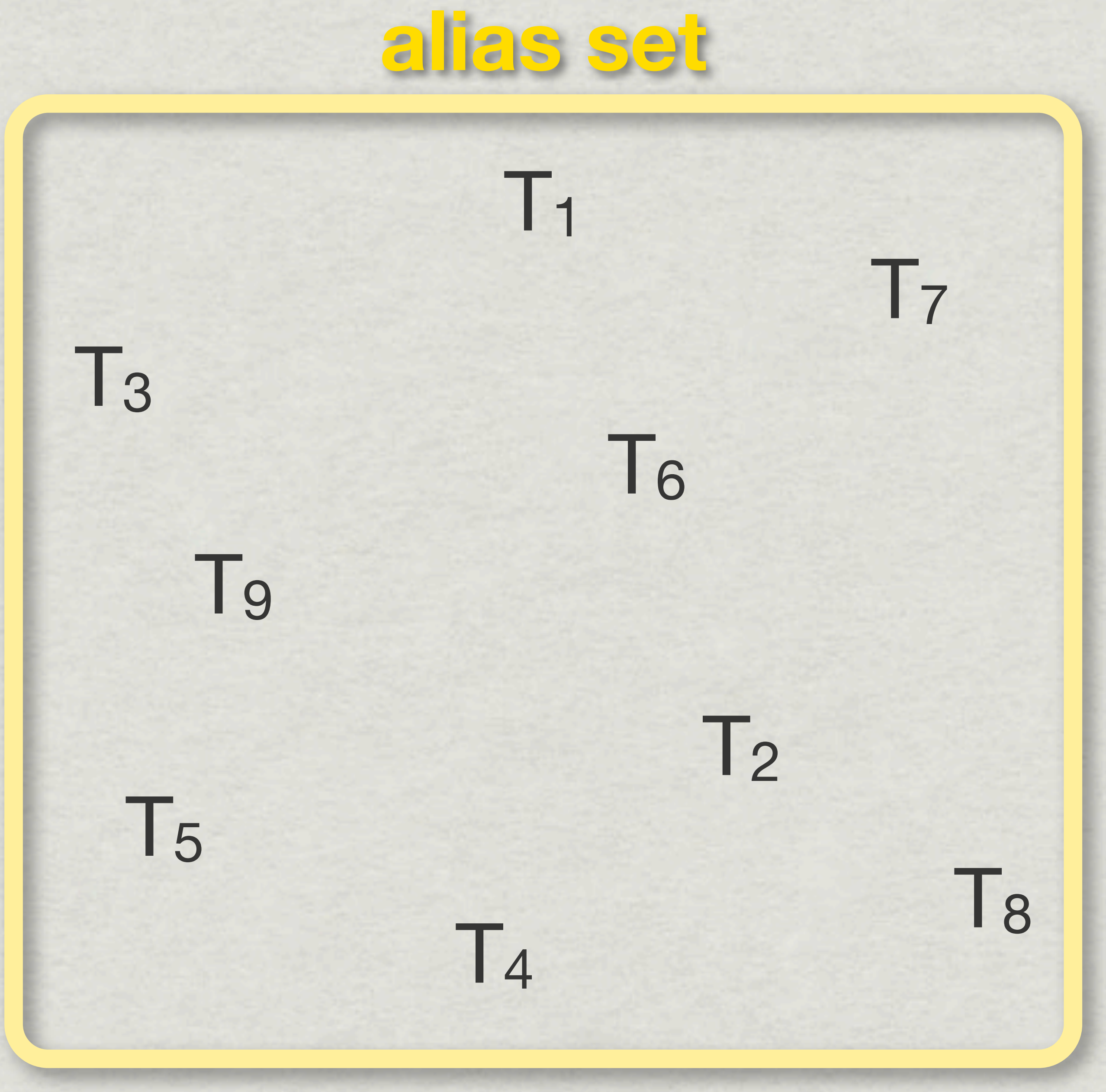
- example of distributed measurement with *mper* and *Marinda*
- case study: one part of MIDAR alias resolution
  - represents a common coordination pattern
  - demonstrates ease of implementation

Case Study

- problem:
  - probe the targets of an alias set to confirm (or *corroborate*) that they are aliases
- requirements:
  - probe targets in alias set one at a time
    - for details, see MIDAR talk later
  - some targets can only be probed from certain monitors because of probing method restrictions

Alias Set

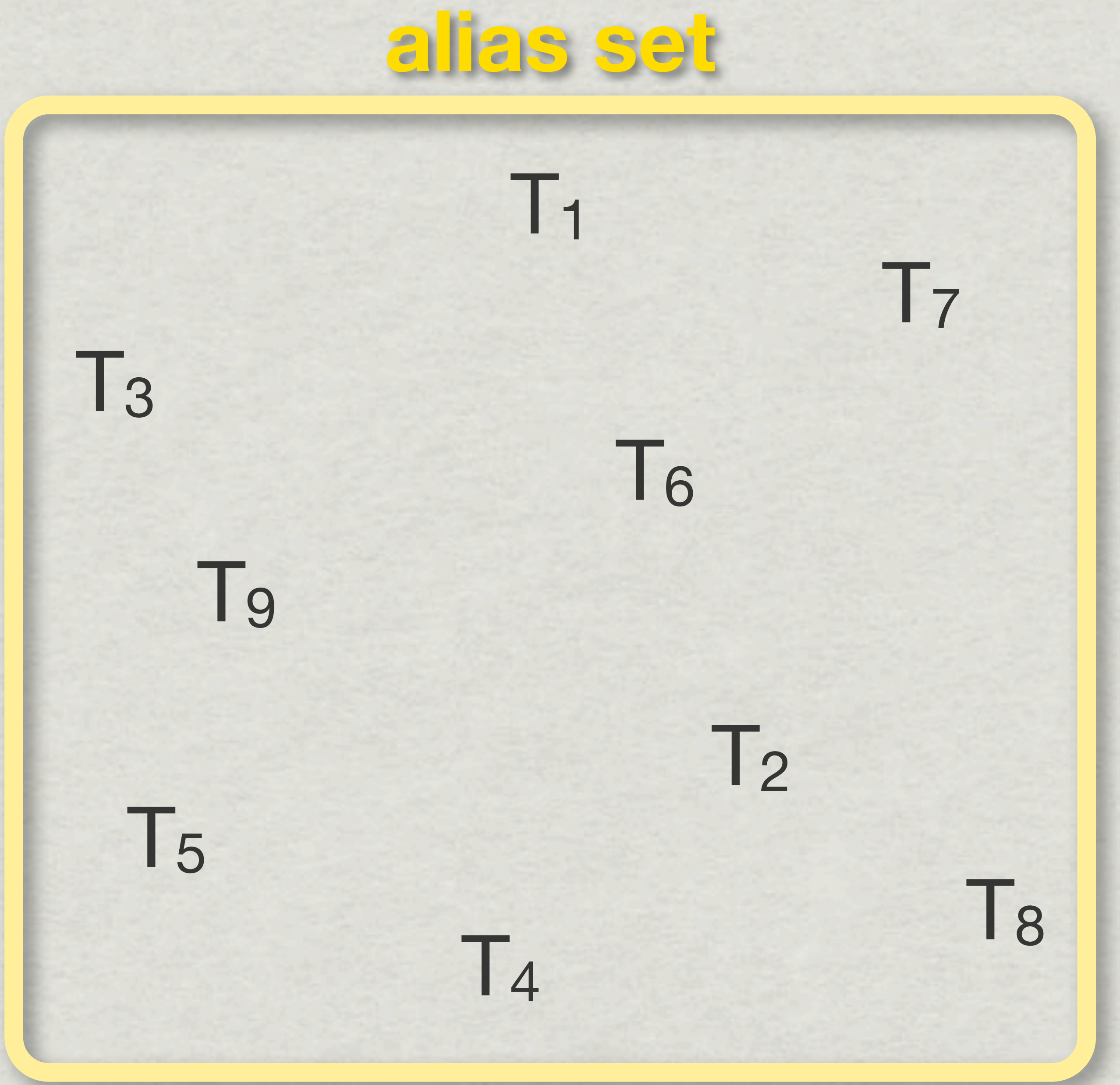
* *alias set:* the set of IP addresses belonging to the same router



T_i = interface *i* ("target *i*")

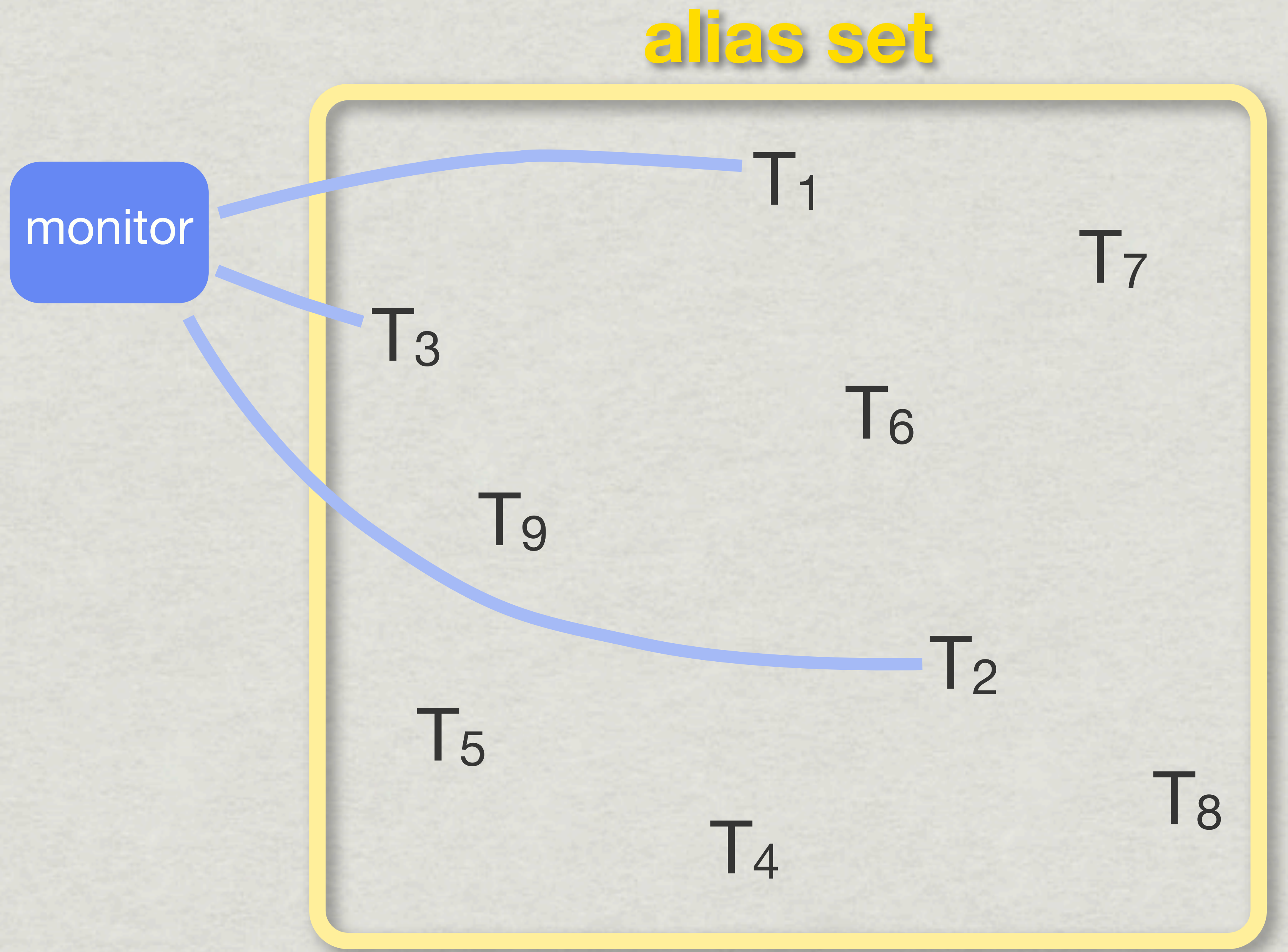
Alias Set

* assign each target to exactly one monitor



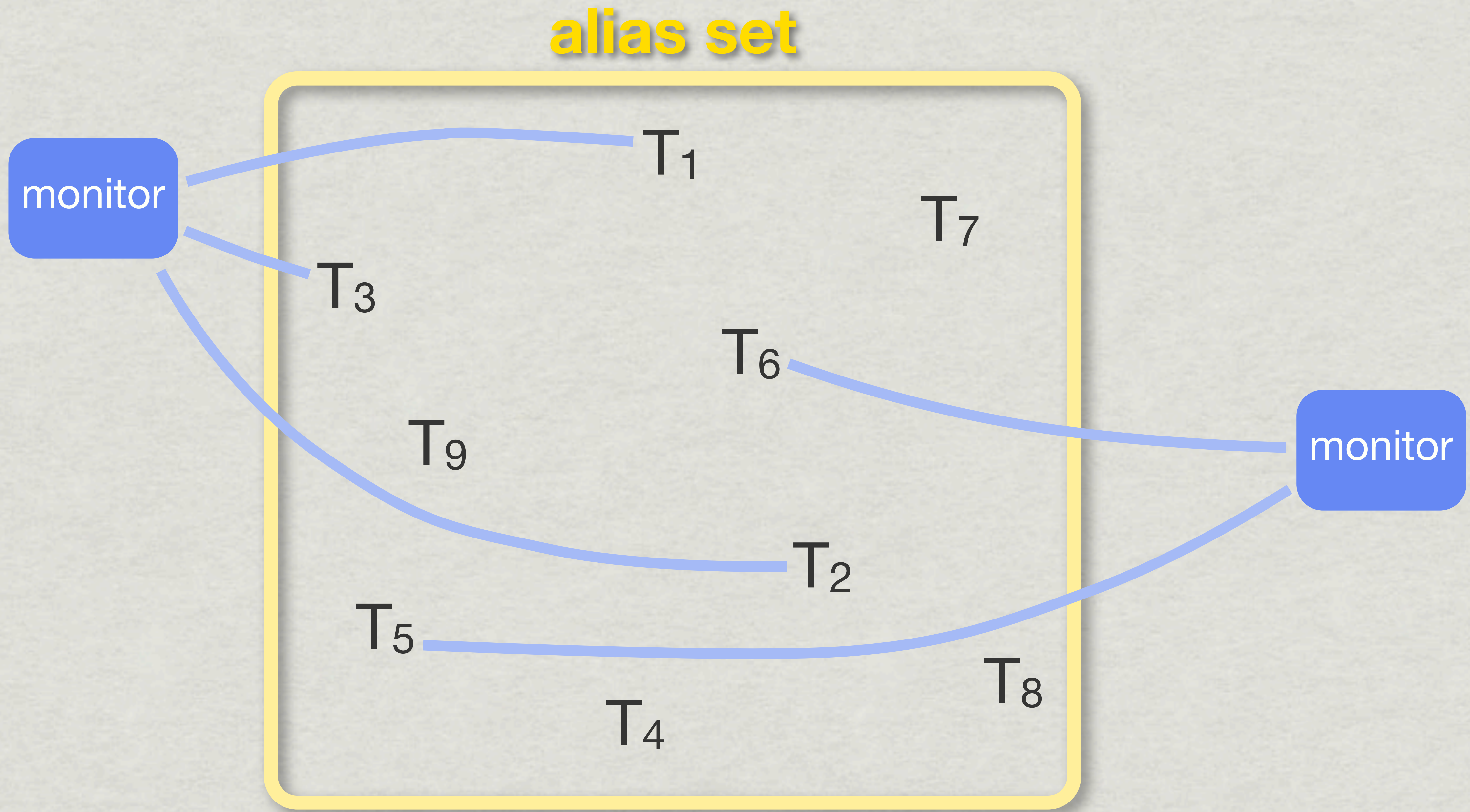
Alias Set

* assign each target to exactly one monitor



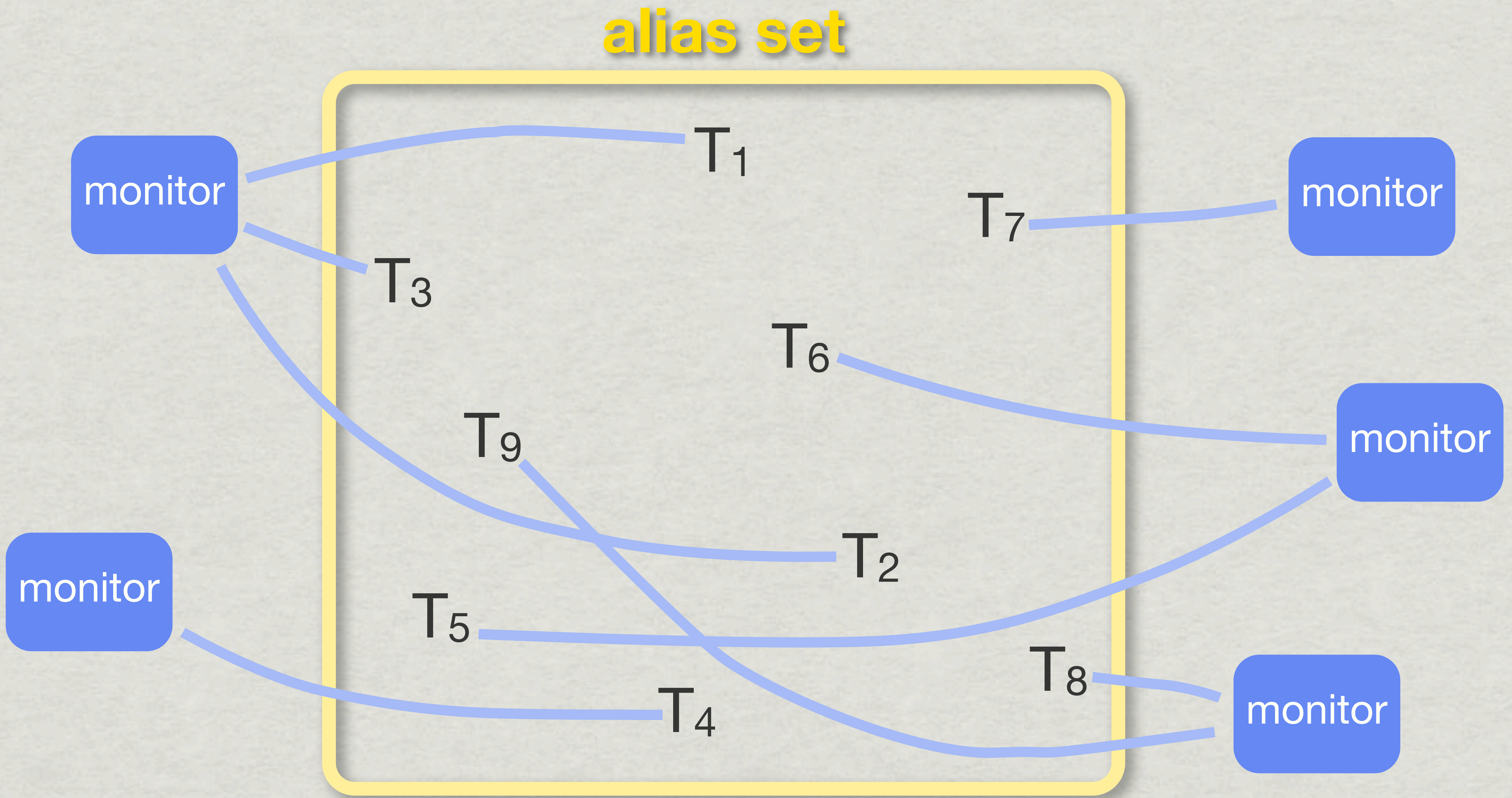
Alias Set

* assign each target to exactly one monitor



Alias Set

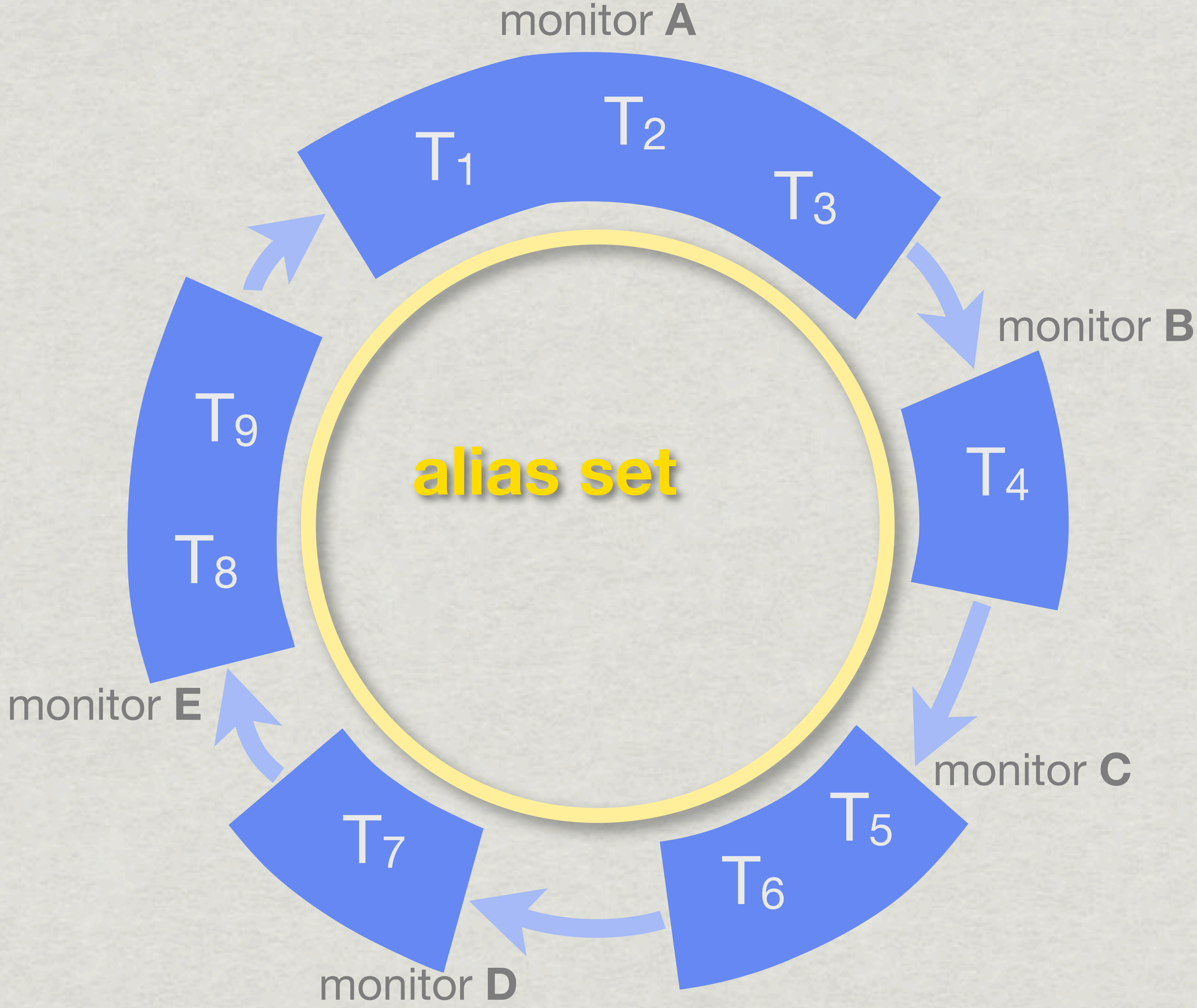
* assign each target to exactly one monitor



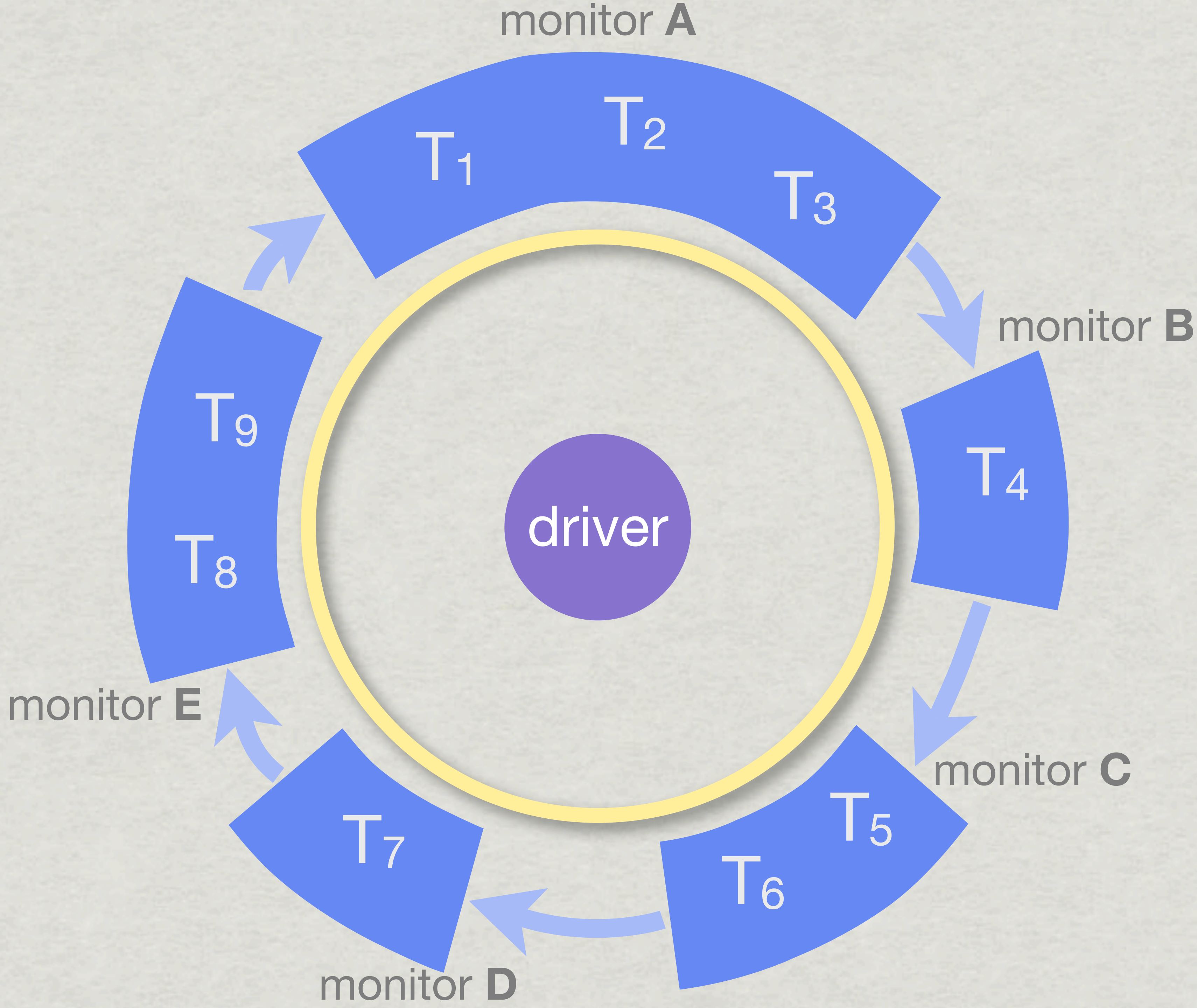
Implementation

- design:
  - a *driver* program running on the central server globally coordinates measurements
  - a *prober* program running on each monitor executes measurements
- probing requires coordination across monitors:
  - driver tells a monitor to probe a target
  - monitor notifies driver of completion after probing

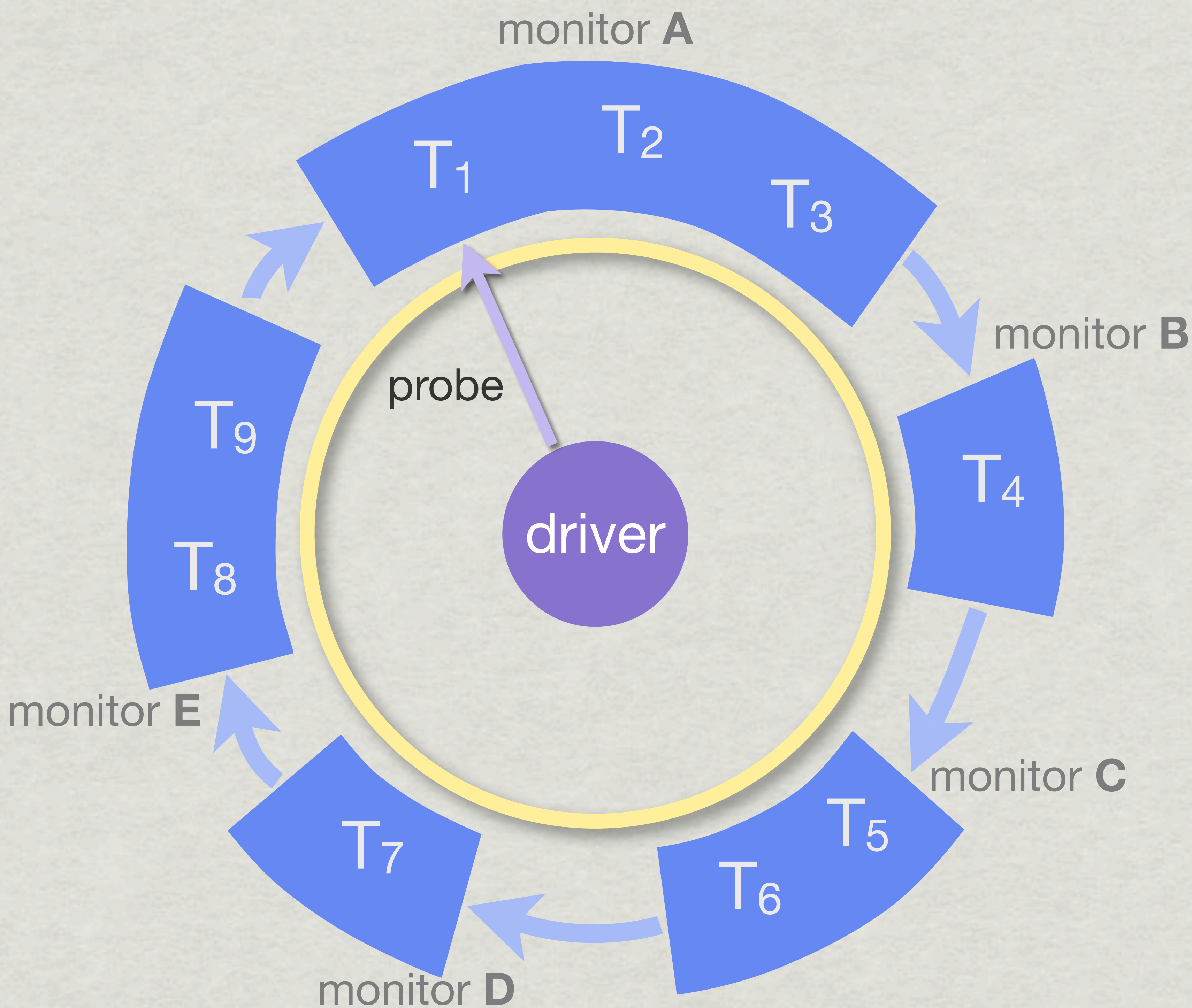
Probing


monitor A
T1
T2
T3
monitor B
T4
monitor C
T5
T6
monitor D
T7
monitor E
T8
T9
alias set

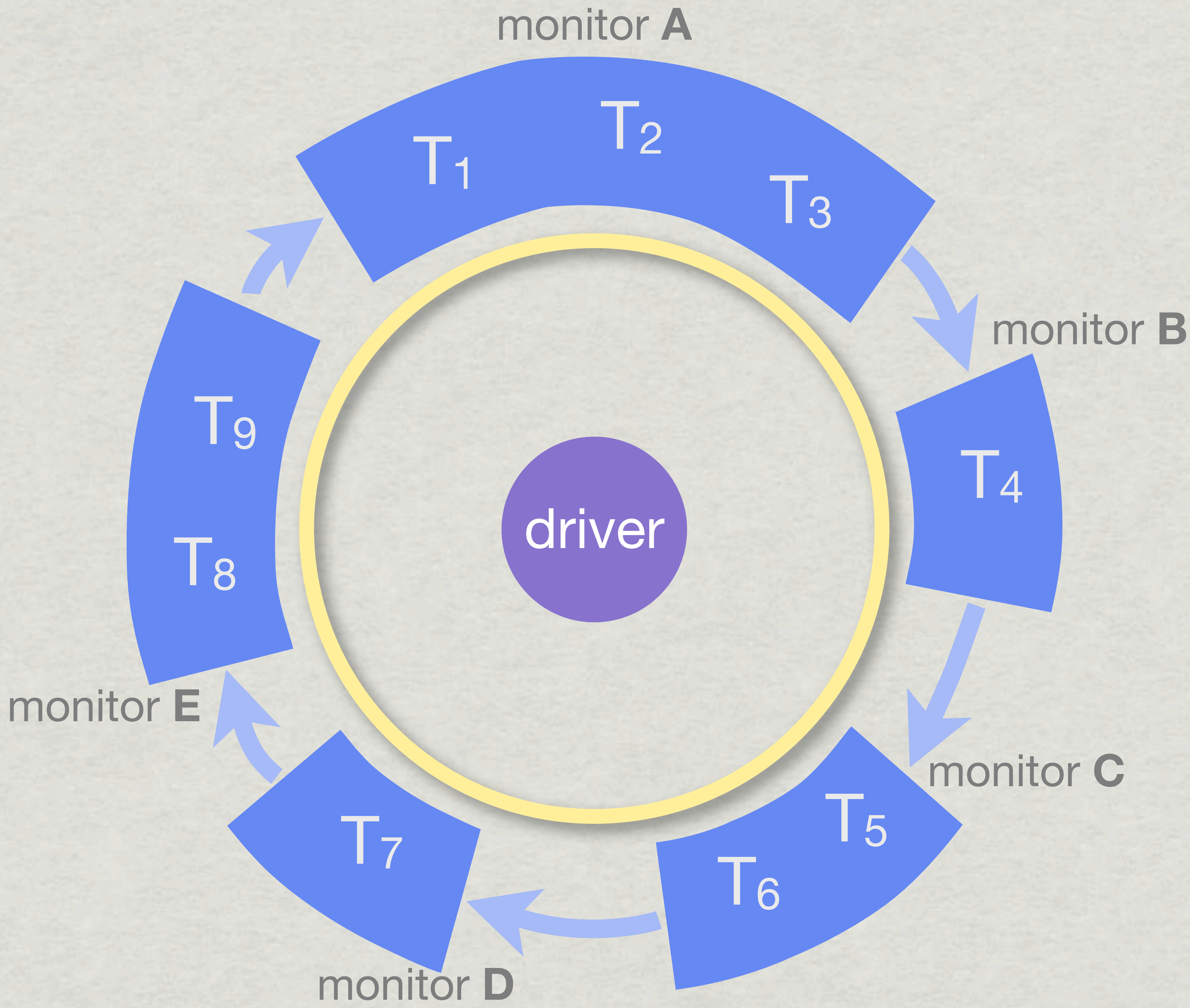
Probing


monitor **A**
T_1
T_2
T_3
monitor **B**
T_4
monitor **C**
T_5
T_6
monitor **D**
T_7
monitor **E**
T_8
T_9
driver

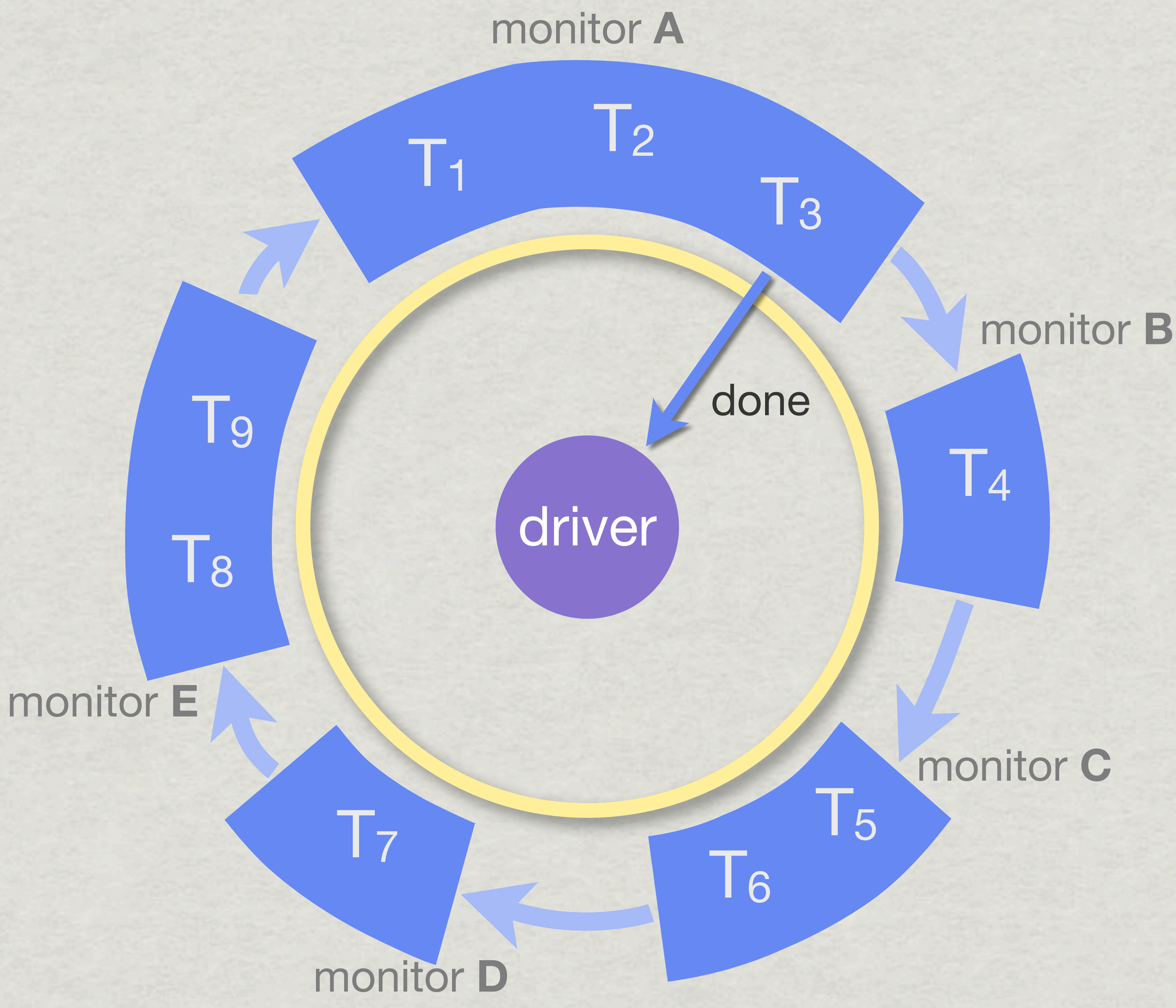
Probing


monitor A
T1
T2
T3
monitor B
T4
monitor C
T5
T6
monitor D
T7
monitor E
T8
T9
probe
driver

Probing

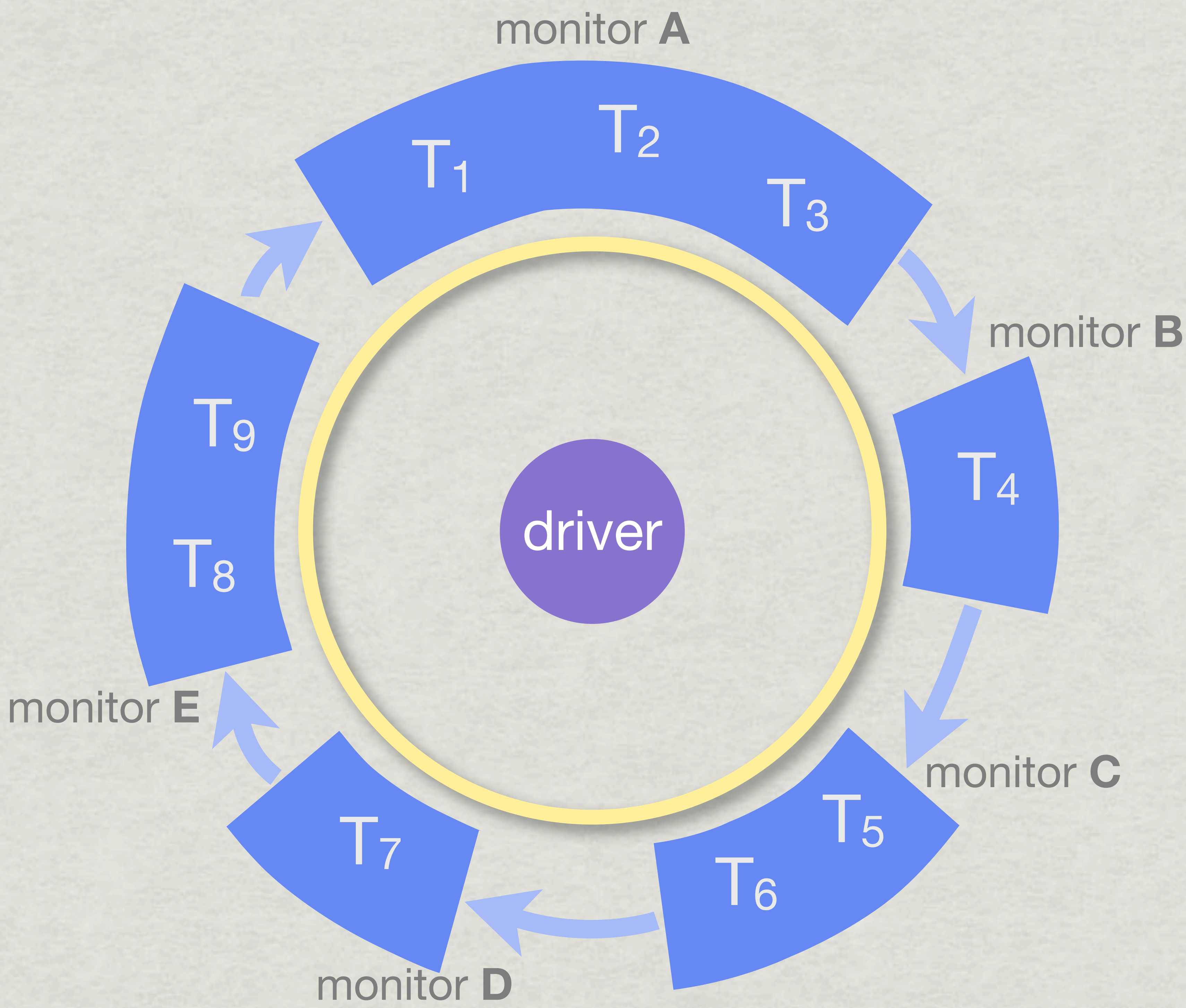

monitor **A**
T_1
T_2
T_3
monitor **B**
T_4
driver
monitor **C**
T_5
T_6
monitor **D**
T_7
monitor **E**
T_8
T_9

Probing

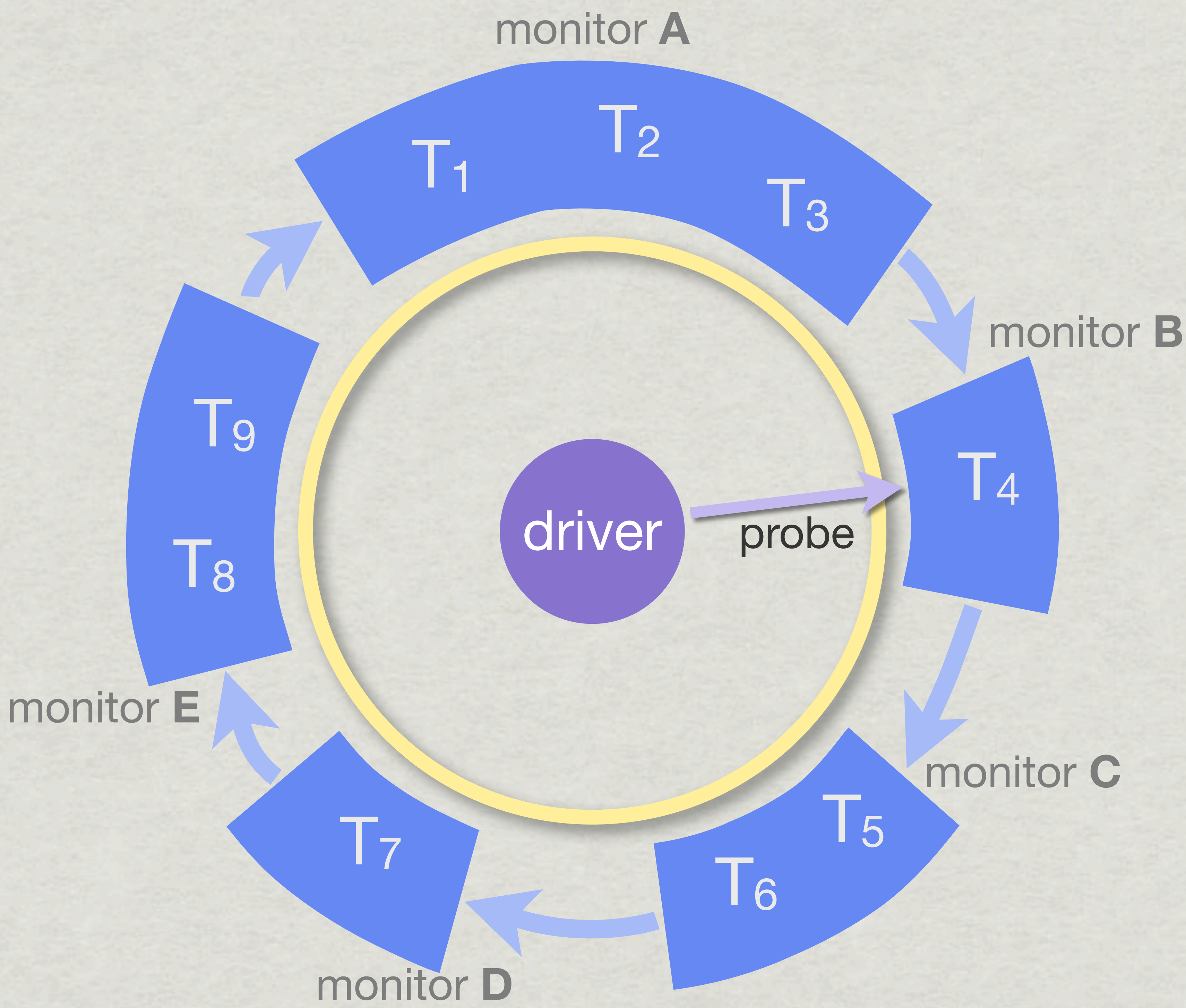


monitor **A**
T_1
T_2
T_3
monitor **B**
T_4
done
driver
monitor **C**
T_5
T_6
monitor **D**
T_7
monitor **E**
T_8
T_9

Probing


monitor A
T_1
T_2
T_3
monitor B
T_4
driver
T_9
T_8
monitor E
T_7
T_5
T_6
monitor C
monitor D

Probing


monitor A
T1
T2
T3
monitor B
T4
driver
probe
monitor C
T5
T6
monitor D
T7
monitor E
T8
T9

Probing

* in practice, probe multiple alias sets in parallel
  * two levels of parallelism:
    * *across* monitors
    * *within* a monitor
* recent MIDAR run:
  * 14,566 alias sets of varying sizes
    * counting only alias sets that must be probed from multiple monitors
  * took ~46 **minutes**; would take 115 **hours** without the parallelism (148x slower)

91.7%

set size	%
2	56.2
3	14.6
4	8.5
5	5.0
6	4.1
7	3.4
...	
148	max

Driver

- coordination is simple to implement
  - telling a monitor to probe a target:

```
$ts.write ["PROBE", monitor, set_id, iteration, target_index]
```

  - reacting to monitor notifications of completion:
    - the driver's main control loop
    - implicit loop using Ruby's block notation
    - handles notifications from all monitors

```
#  ["DONE", <monitor>, <alias-set-id>, <iteration>, <target-index>]
$ts.consume_stream(["DONE", nil, nil, nil, nil]) do |tuple|
  puts(tuple[1])   #  do something with tuple
end
```

Driver

```
def start
  prime_jobs()

  # ["DONE", <monitor>, <alias-set-id>, <iteration>, <target-index>]
  $ts.consume_stream(["DONE", nil, nil, nil, nil]) do |tuple|
    monitor, set_id, iteration, target_index = tuple[1..-1]
    set = @sets[set_id]

    unless submit_job(set)
      tuple = ["FINISHED", "set", set.set_id]
      $ts2.write tuple   # broadcast set completion
      $ts2.take tuple

      @active_count -= 1
      unblock_next_job(set)
      prime_jobs()
    end

    break if @active_count == 0
  end
end
```

Prober

- prober runs on each monitor
  - coordinates with the driver
  - executes measurements with mper and saves results

```
#  [PROBE, <monitor>, <alias-set-id>, <iteration>, <target-index>]
$ts1.consume_stream_async(["PROBE", $monitor, nil, nil, nil]) do
  |tuple|
  set_id, iteration, index = tuple.values_at 2, 3, 4
  set = find_set(set_id)
  set.schedule(iteration, index)
  if @more
    @more = false
    execute_measurement(set)
  else
    @deferred_measurements << set
  end
end
```

Prober

```
$ts2.monitor_stream_async(["FINISHED", nil, nil]) do |tuple|
  case tuple[1]
  when "set"
    set_id = tuple[2]
    @active_sets.delete(set_id)

  when "run"
    @drain = true
    if @active_measurements.empty? && @deferred_measurements.empty?
      @mperio.suspend
    end
  end
end
```

Prober

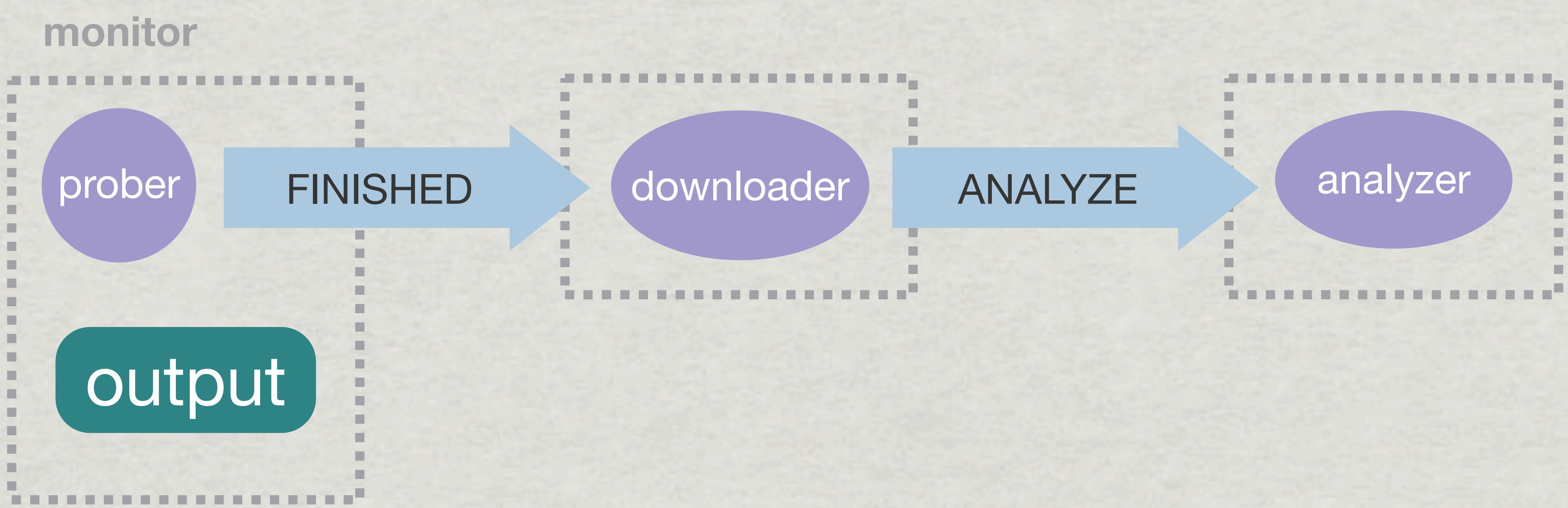
* notifying driver of completion of probe:

```
$ts.write ["DONE", $monitor, set_id, iteration, target_index]
```

* notifying *downloader* of completion of run:

```
$ts.write ["FINISHED", "prober", run_id, $monitor, out_path]
```

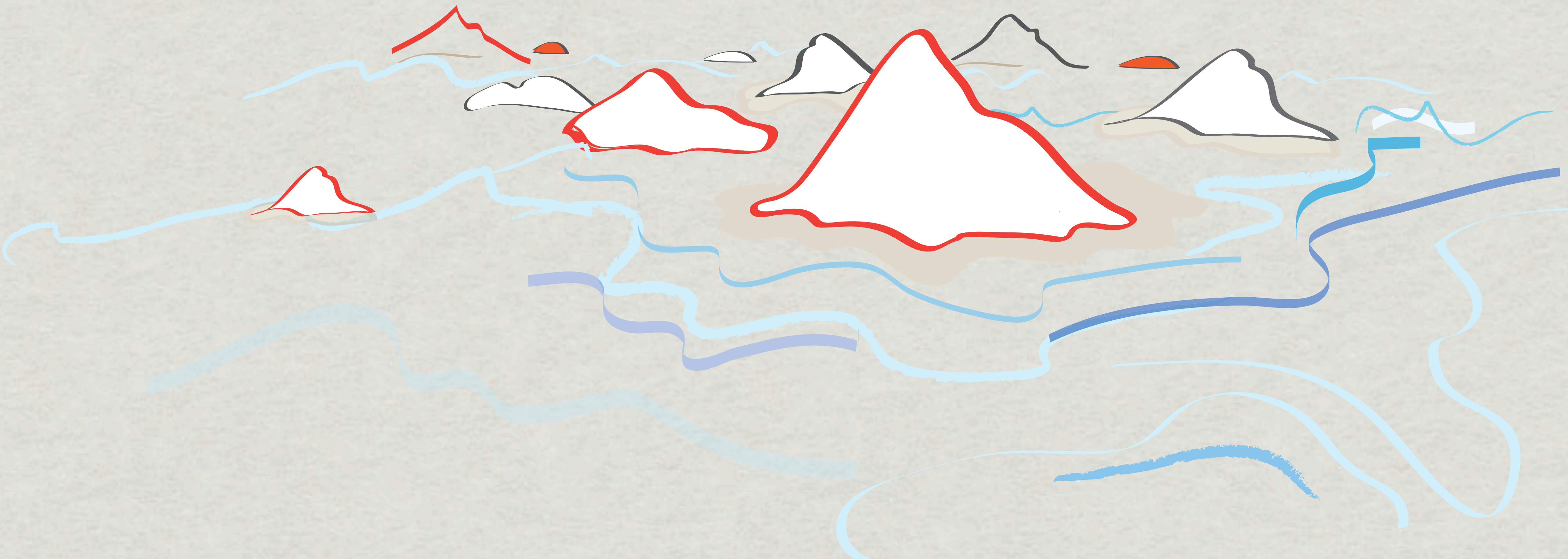
* working towards automating full system
  * coordinate stages on different machines with Marinda



Future Work

* release mper and Marinda under GPL
* create AS-router dual graph
* improve infrastructure to allow more collaborators to use Ark

Thanks!



For more information, or to request data:

www.caida.org/projects/ark

For questions, or to offer hosting: ark-info@caida.org