



# Erlang based OS for Internet of Everything

Sandhya Narayan, Ph.D.

2/18/2015

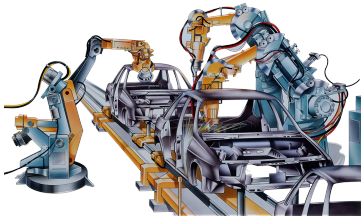
# Outline

- The connected cow



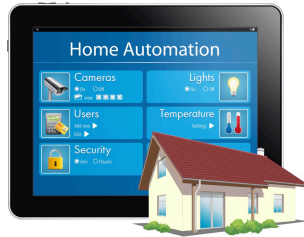
- In a world with trillions of devices often in places hard to reach, is there is any software framework or platform suitable for IoE?
- Can such a platform be Erlang-based?
- This talk will provide some motivation and an example implementation.

# loE examples



## Industrial Automation

WirelessHART, 802.15.4  
6tsch, RPL  
IEEE/IIC/IETF



## Home Area Networks

ZigBee, Z-Wave  
6lowpan, RPL  
IETF/ZigBee/private



## Personal Area Networks

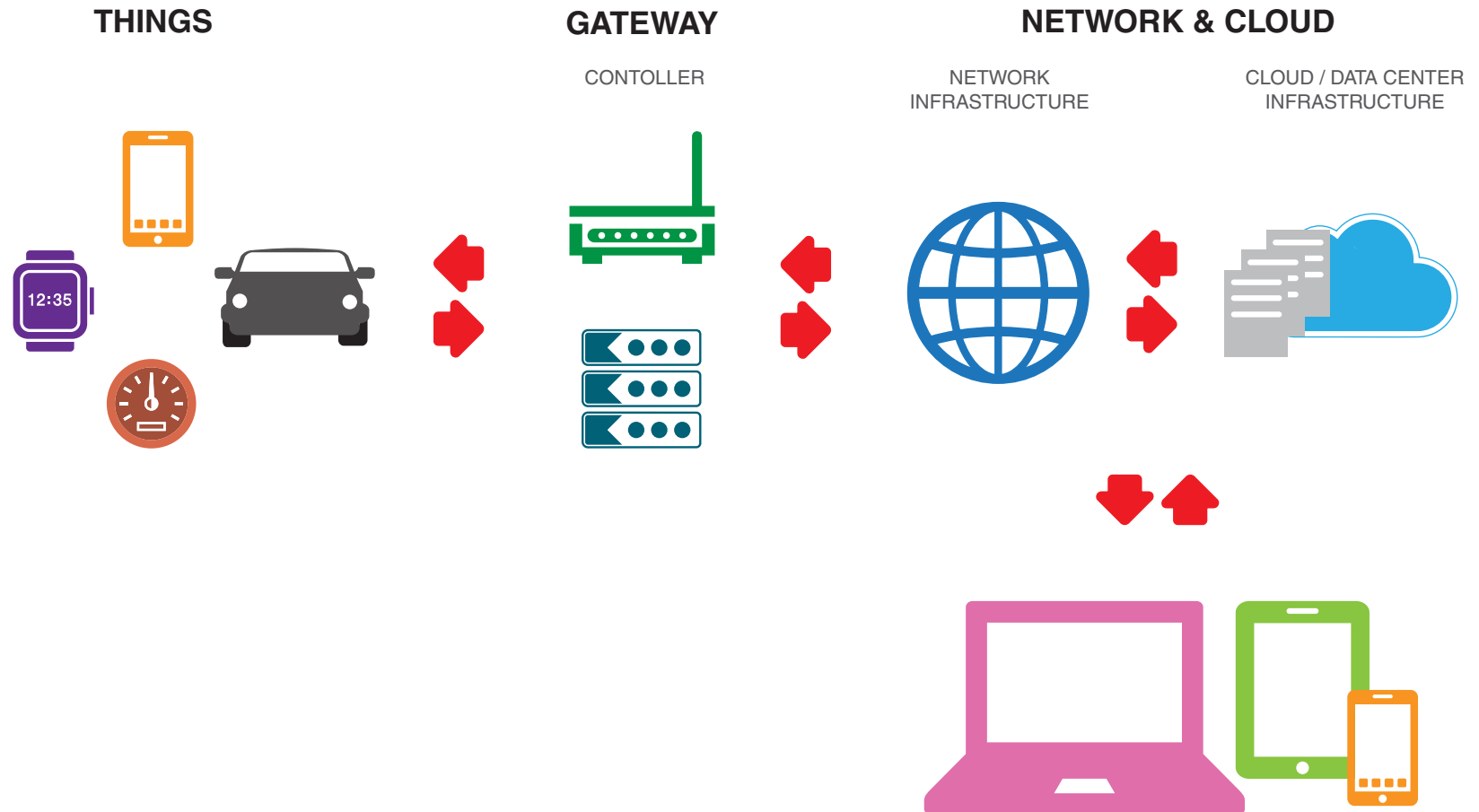
Bluetooth, BLE  
3G/LTE  
3GPP/IEEE



## Networked Devices

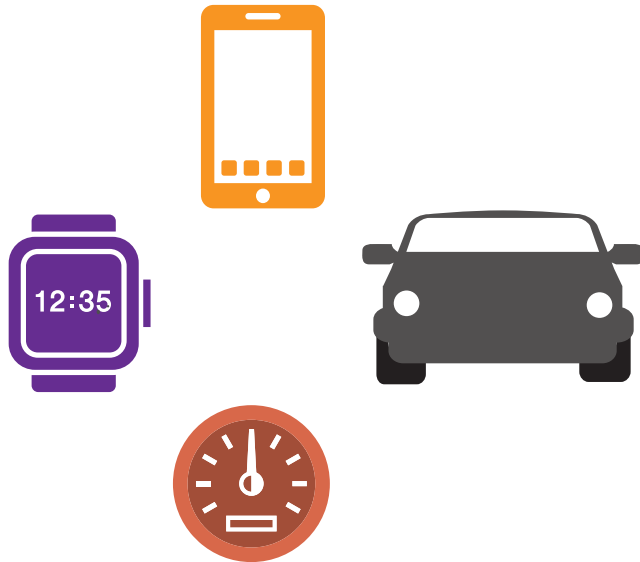
WiFi/802.11  
TCP/IP  
IEEE/IETF

# Typical IoE Architecture





# Diverse languages and programming environments



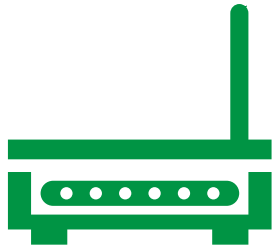
## THINGS

---

Embedded C, ASICs  
FPGA

---

# Diverse languages and programming environments

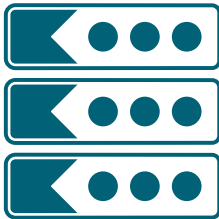


GATEWAY

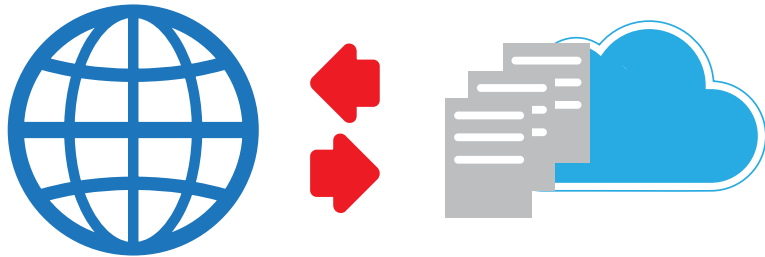
---

C, Java, ASICs

---



# Diverse languages and programming environments



## NETWORK & CLOUD

---

Ruby/Rails, Java,  
Node.js, Python,  
PHP, Scala, Erlang,  
Hadoop, Spark, R

---

# Diverse languages and programming environments

- Development in silos
- Need teams with varied skill-sets
- Not effective and efficient enough for specialization, scale and rapid change
- Focus is on complexity not on the application
- May support real time data collection, but can there be timely analysis and action?

# Taming Complexity

- Trillions of devices
- Hard to reach and replace
- Millions of apps

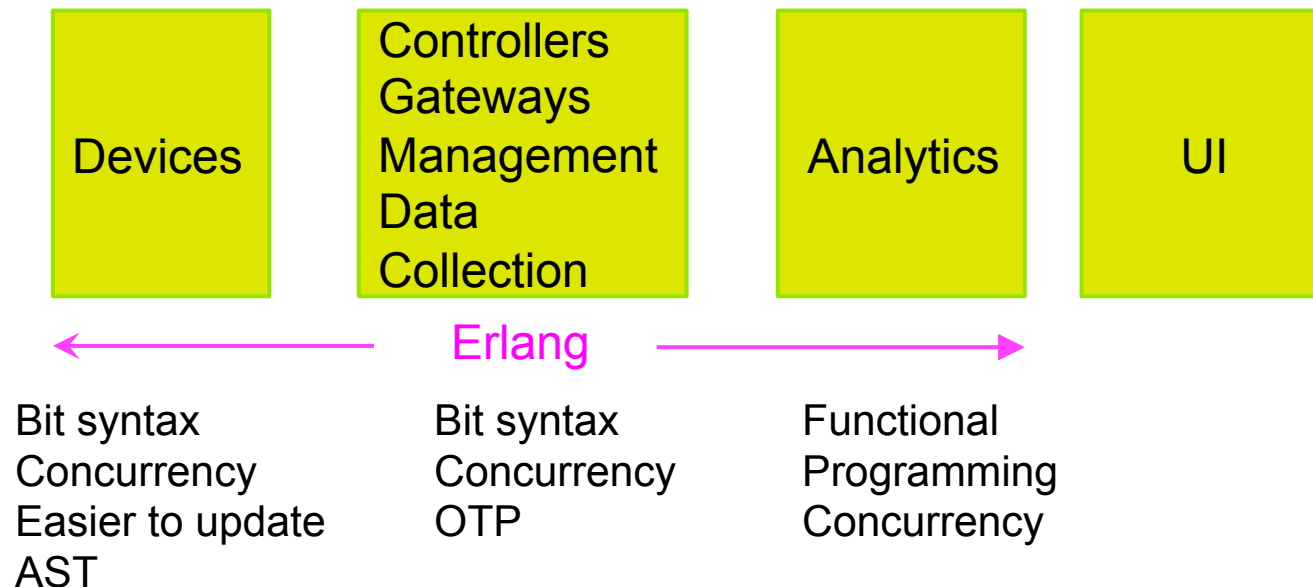
Many interesting questions

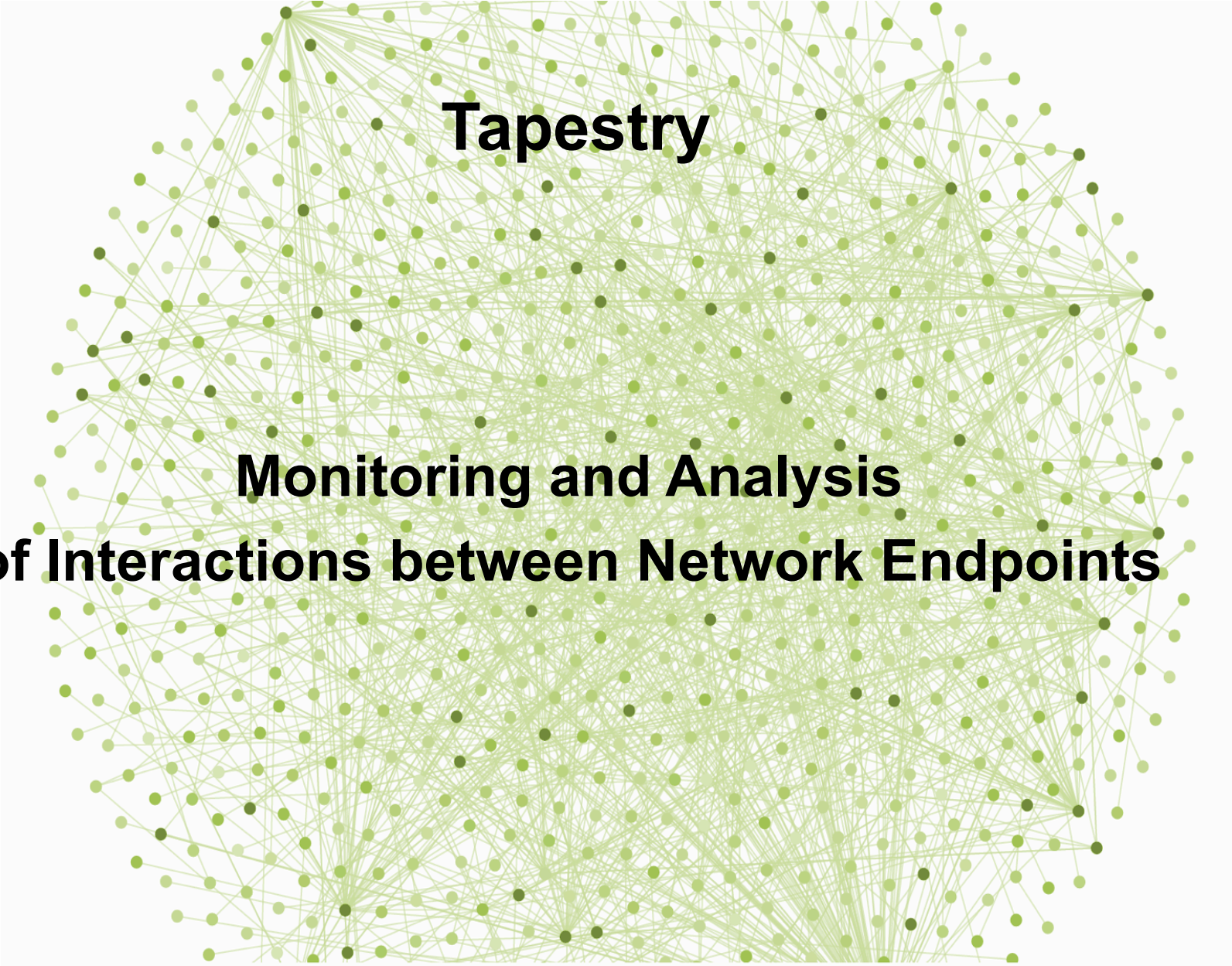
- Do we have that many developers?
- Mechanisms to update code, OS, protocol?
- Methods to make it secure and reliable?



# Unified approach?

Unified development focusing on the application possible?  
For some cases would it be better than trying to use a different toolkit for each component?



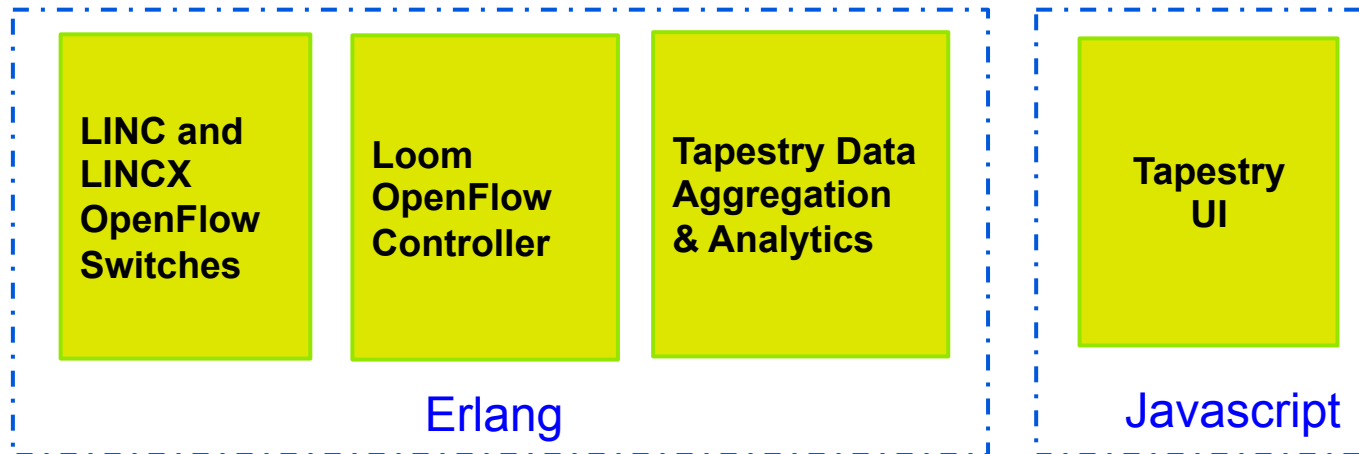


# **Tapestry**

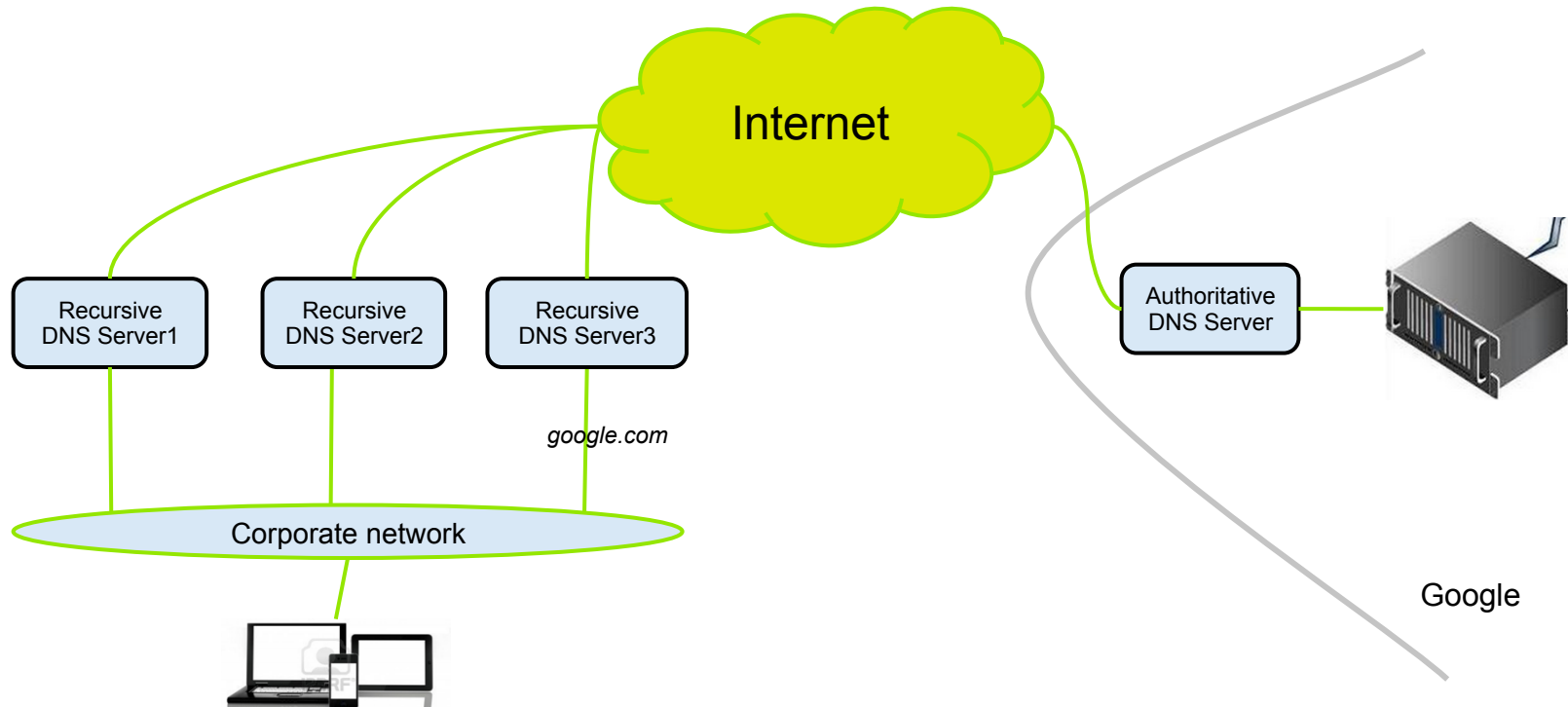
## **Monitoring and Analysis of Interactions between Network Endpoints**

# Tapestry

- Is a general scalable platform for control, telemetry and analytics of network devices
- Speaks a special protocol: Openflow
- All components except UI are in Erlang
- Implements embedded machine learning with no specialized analytics platform (e.g. Spark, Hadoop)
- Developed as a PoC by a small team



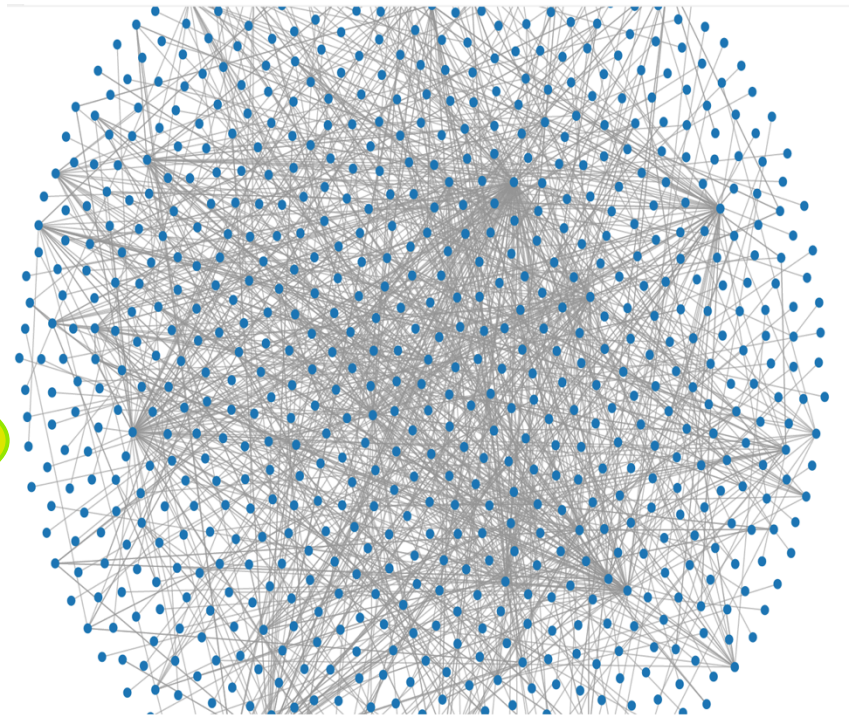
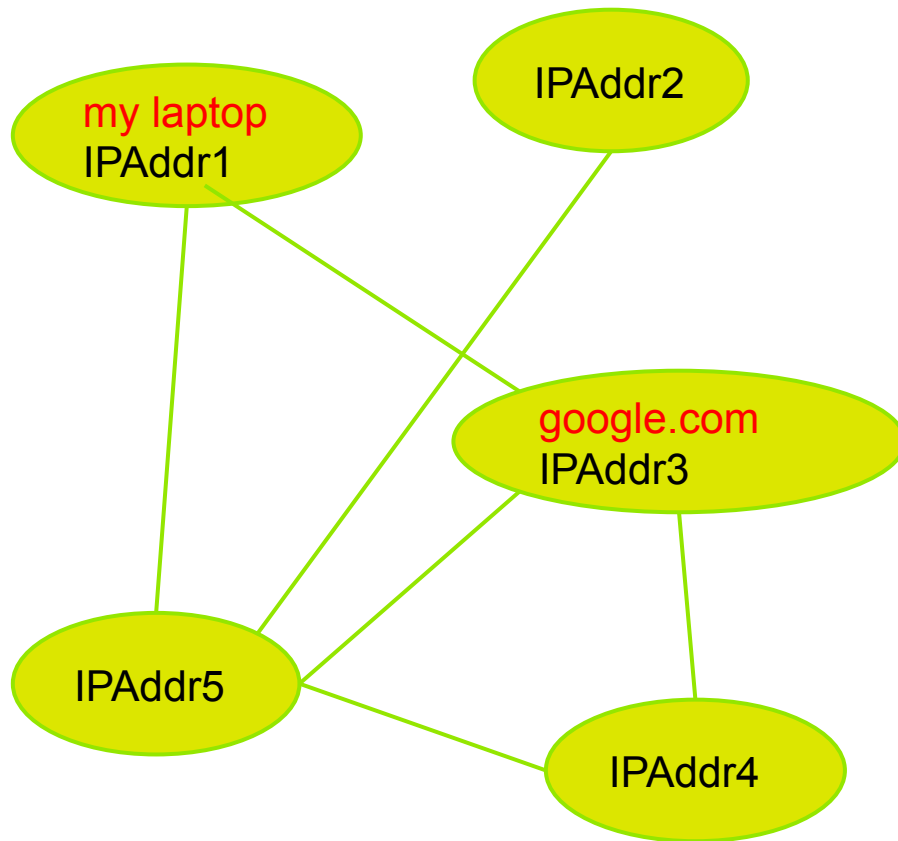
# DNS: An indicator of endpoint interactions



DNS response → {**Requesting IP address,**  
**Resolved IP address**}



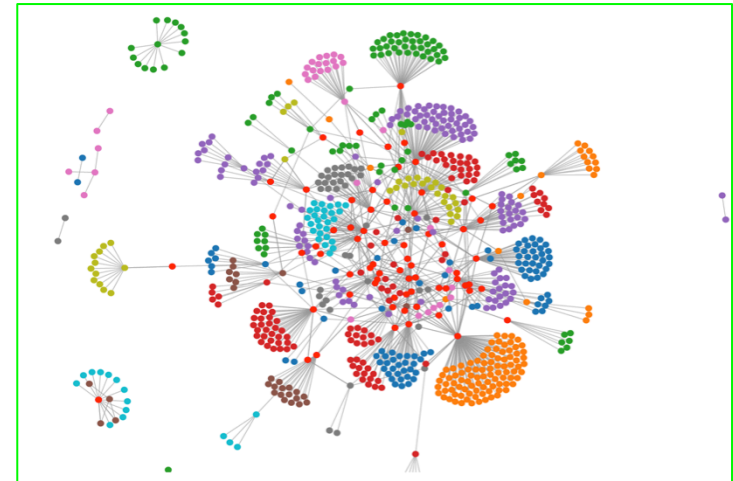
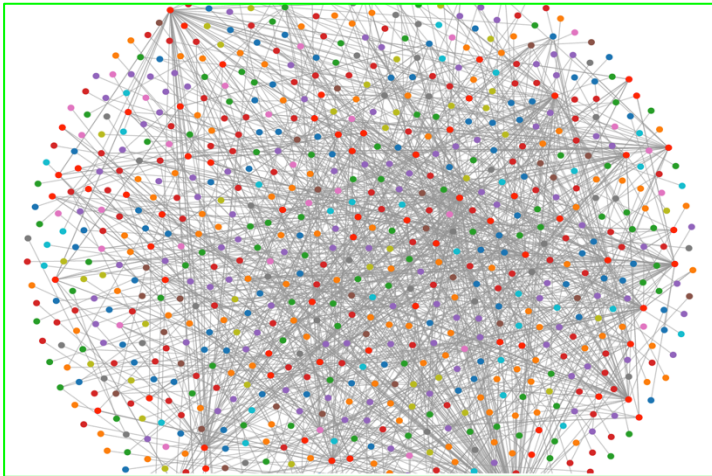
# Endpoint interactions



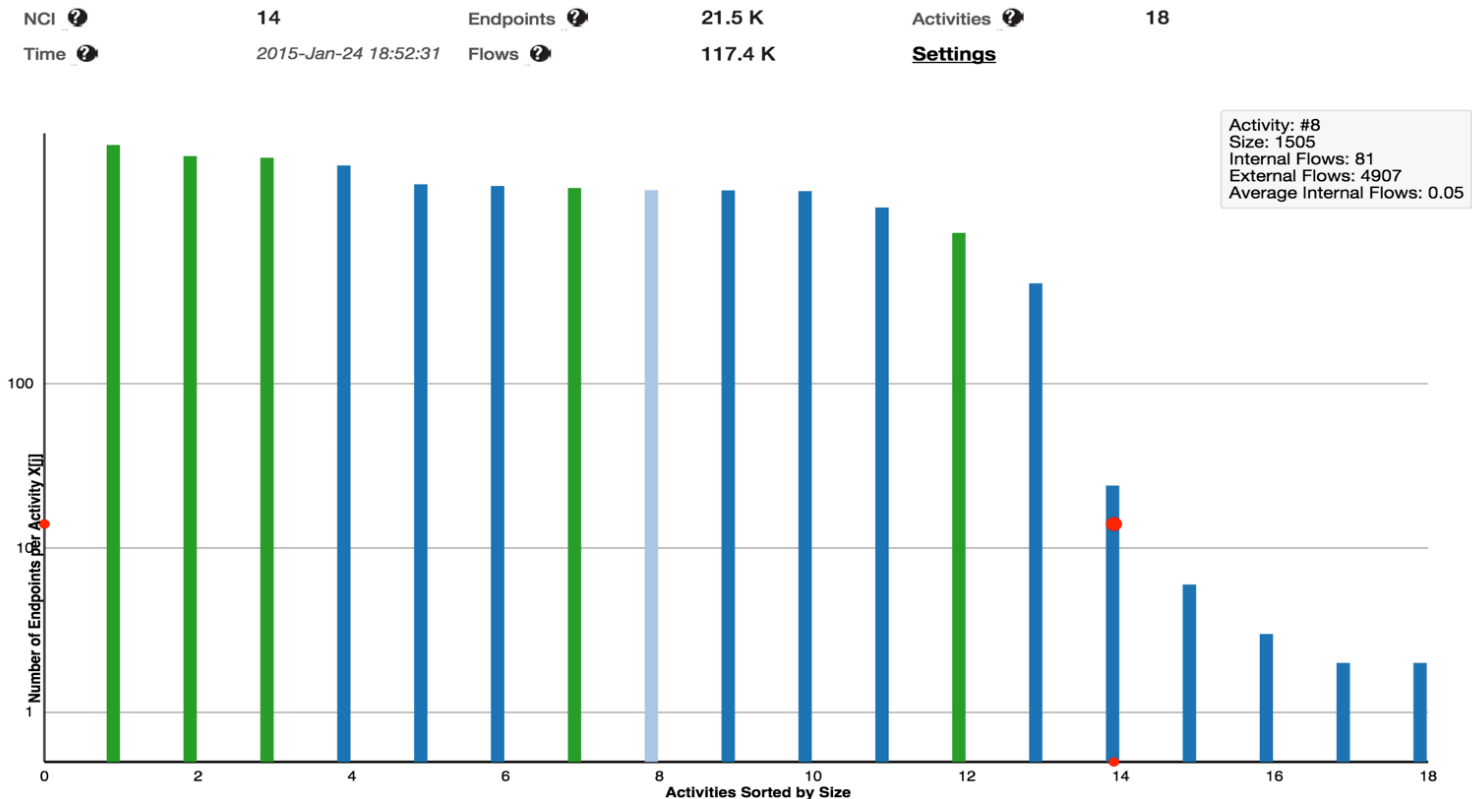


# Tapestry Analytics: Community Detection

- A network is said to have community structure if the nodes of the network can be easily grouped into sets of nodes such that each set of nodes is densely connected internally and more sparsely connected outside the group.
- The number of communities in a network and their sizes are not known beforehand and they are established by the community detection algorithm.
- Two community detection algorithm implemented: Label Propagation Algorithm and Louvain Method.



# Histogram of activities (communities)



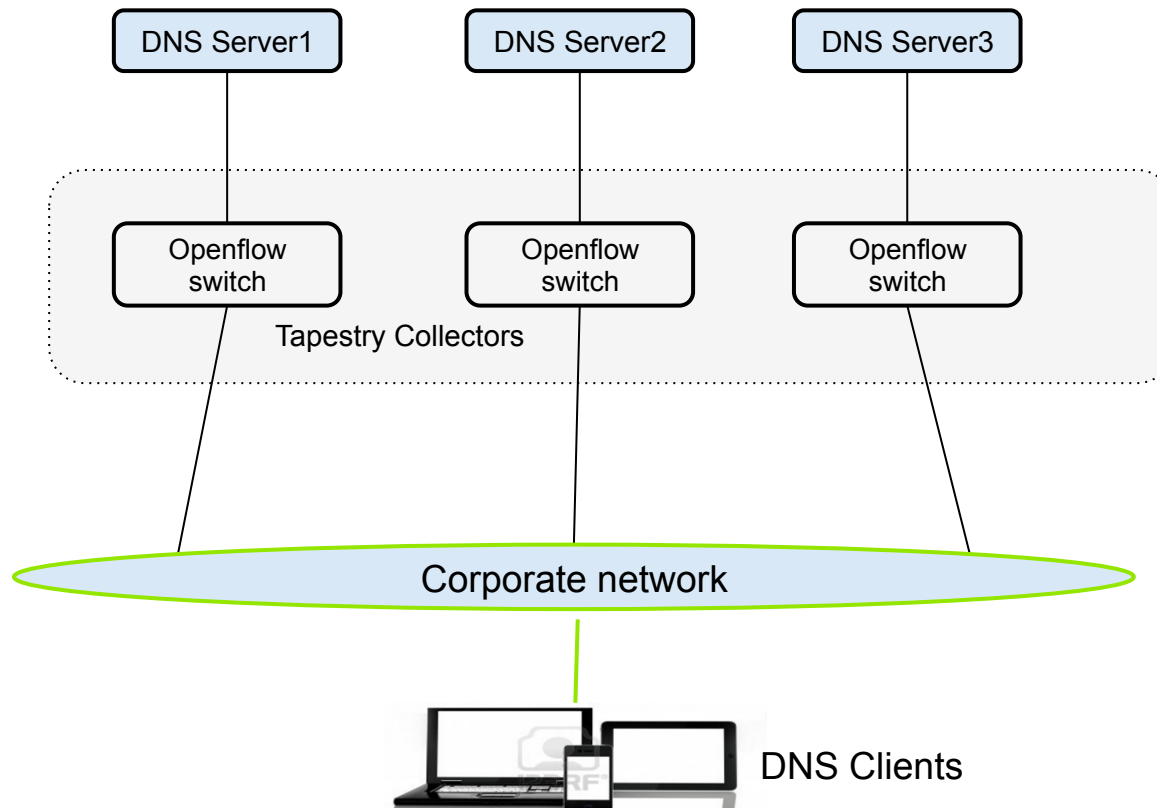
NCI is defined as follows:

$$NCI(N) = \text{Max } j, X[j] \geq j$$

where  $NCI(N)$  is the Network Complexity Index of network  $N$  and  $X[j]$  is the number of endpoints engaged in an activity.

# Tapestry demo video

# Tapestry Openflow Collectors



# Openflow Switch



## OpenFlow-enabled Network Device

*Flow Table comparable to an instruction set*

MAC src	MAC dst	IP Src	IP Dst	TCP dport	...	Action	Count
*	10:20:..	*	*	*	*	port 1	250
*	*	*	5.6.7.8	*	*	port 2	300
*	*	*	*	25	*	drop	892
*	*	*	192.*	*	*	local	120
*	*	*	*	*	*	controller	11

### Many Flavors

- Versions 1.0-1.5
- pure Openflow / Hybrid
- Hardware/ FPGA/ Software

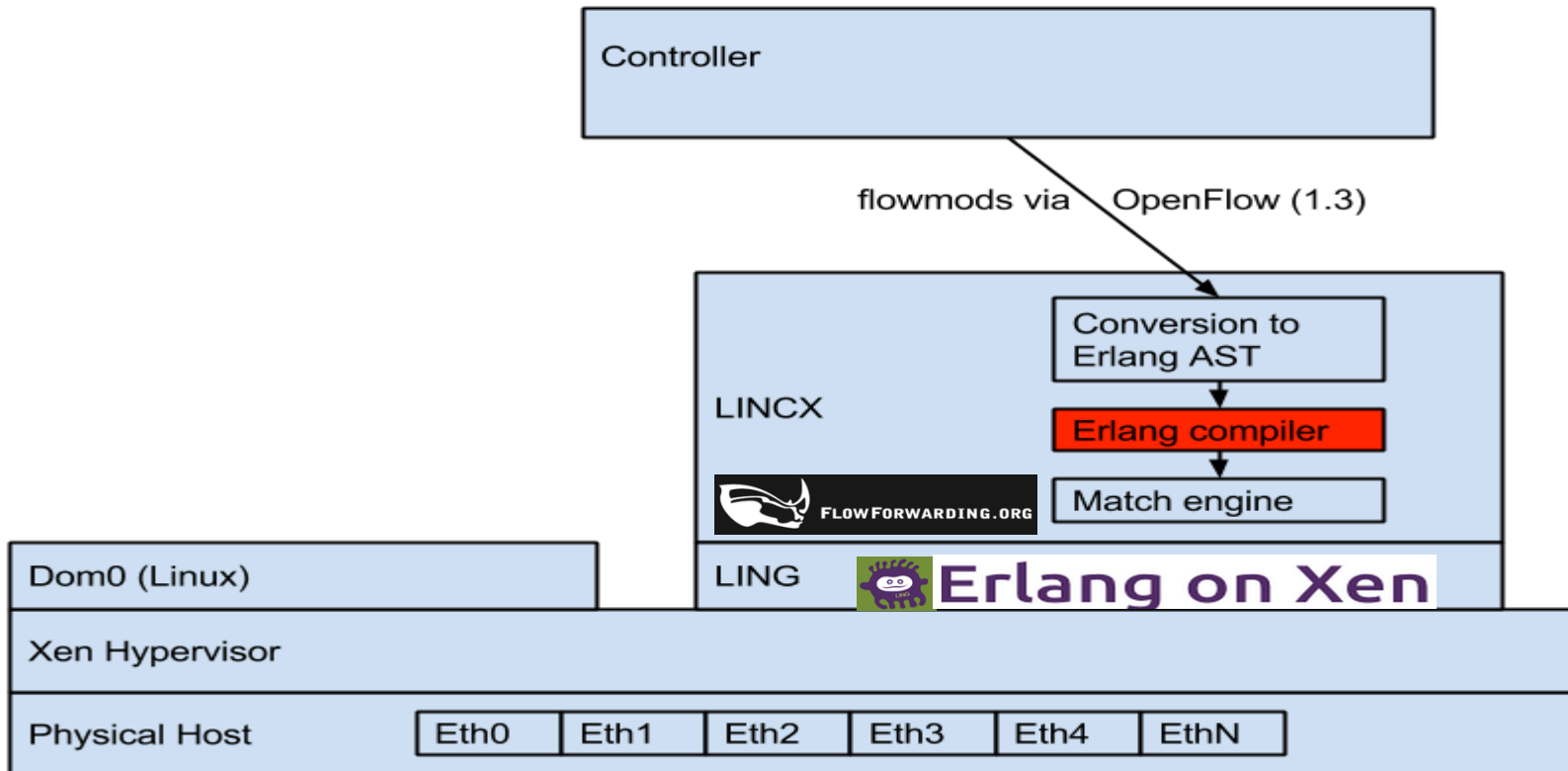
### FlowForwarding.org

Openflow switches in Erlang

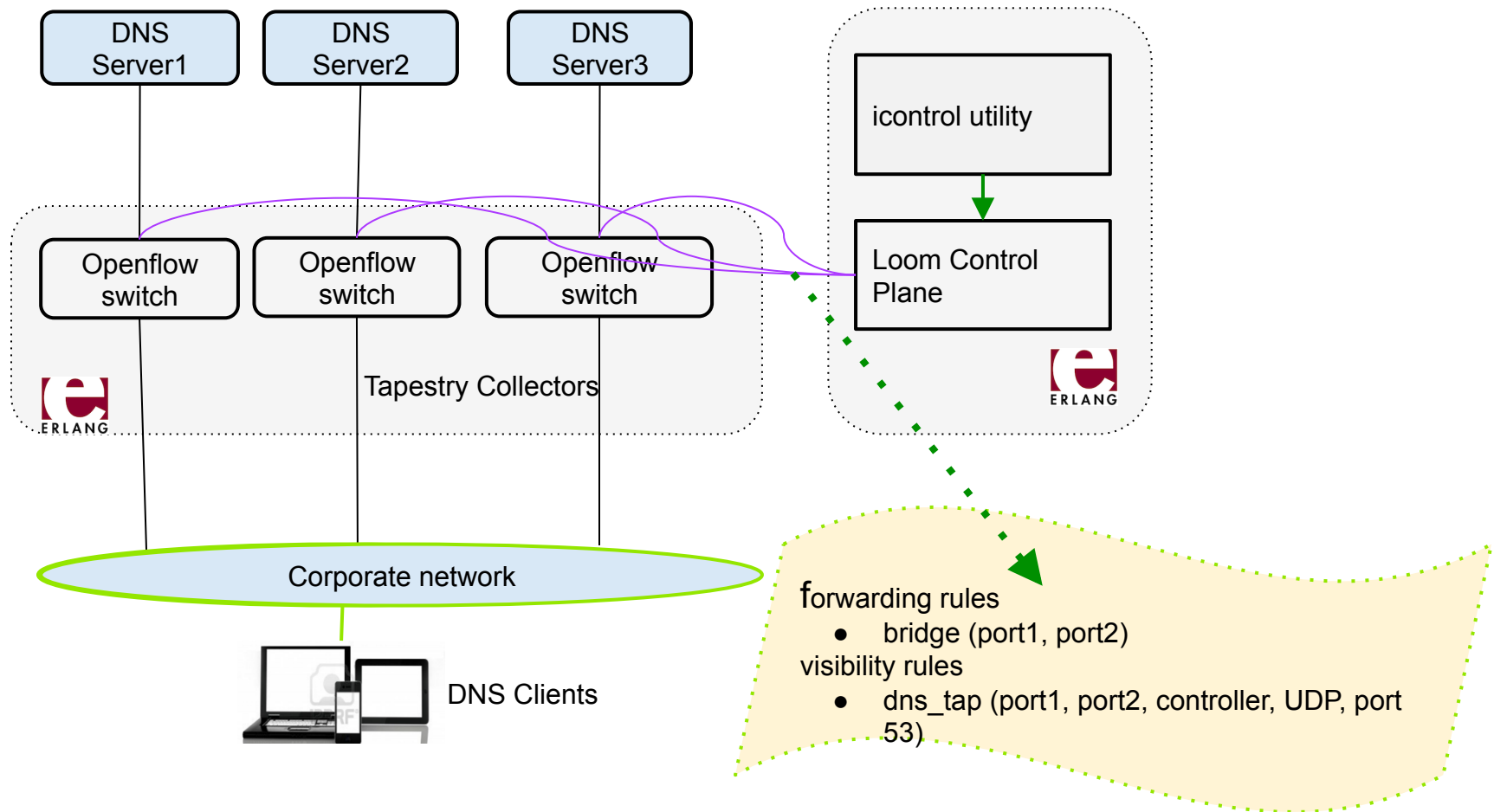
- LINC
- LINCX
- LINC-OE



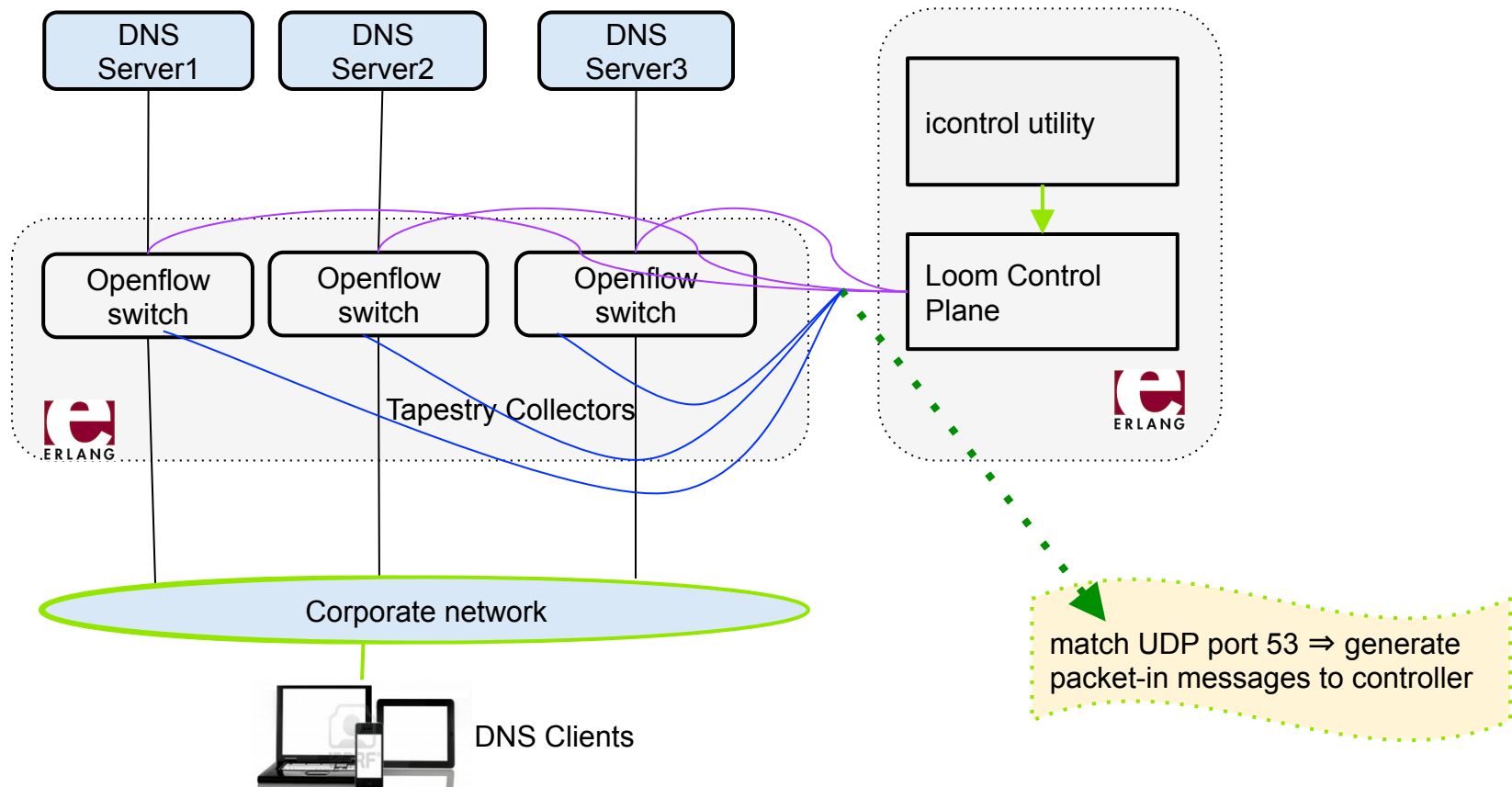
# LINCX - Fast CPU based Ethernet Switch



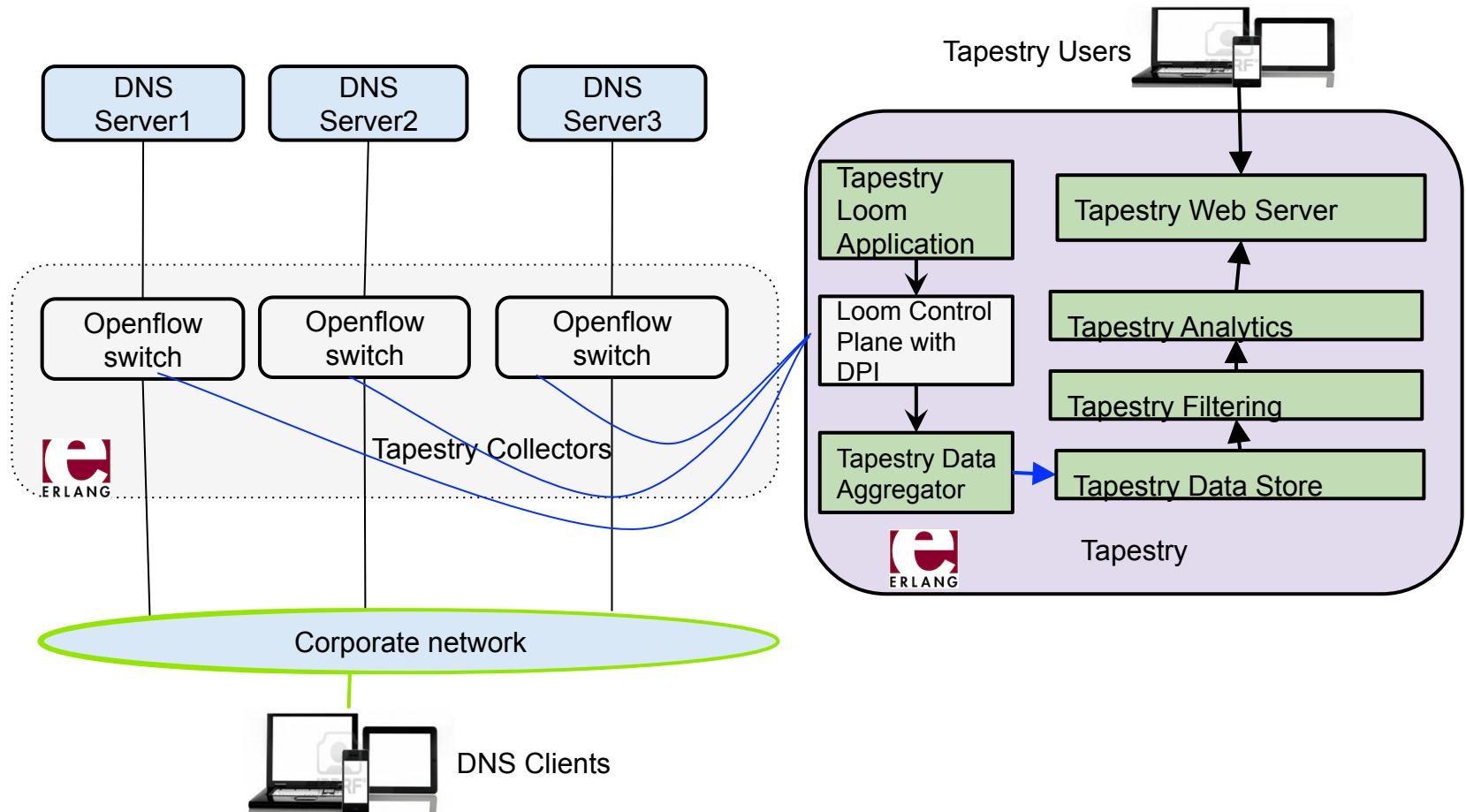
# Setup flow rules in switches



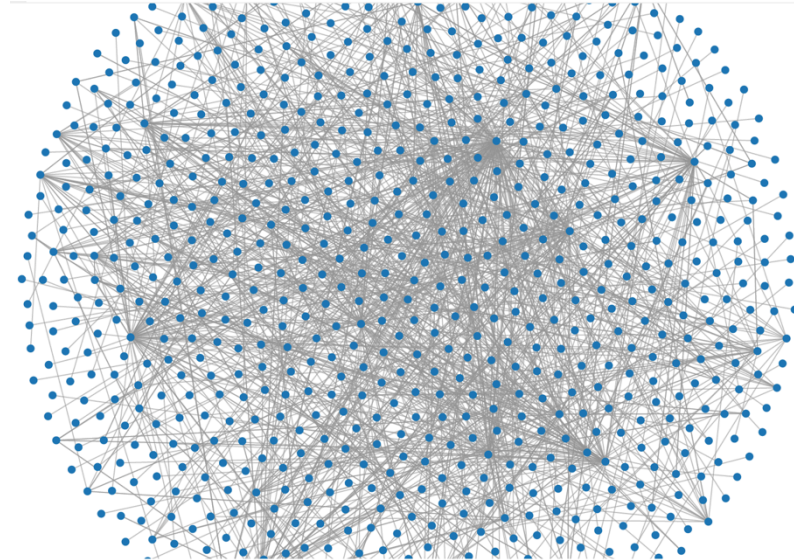
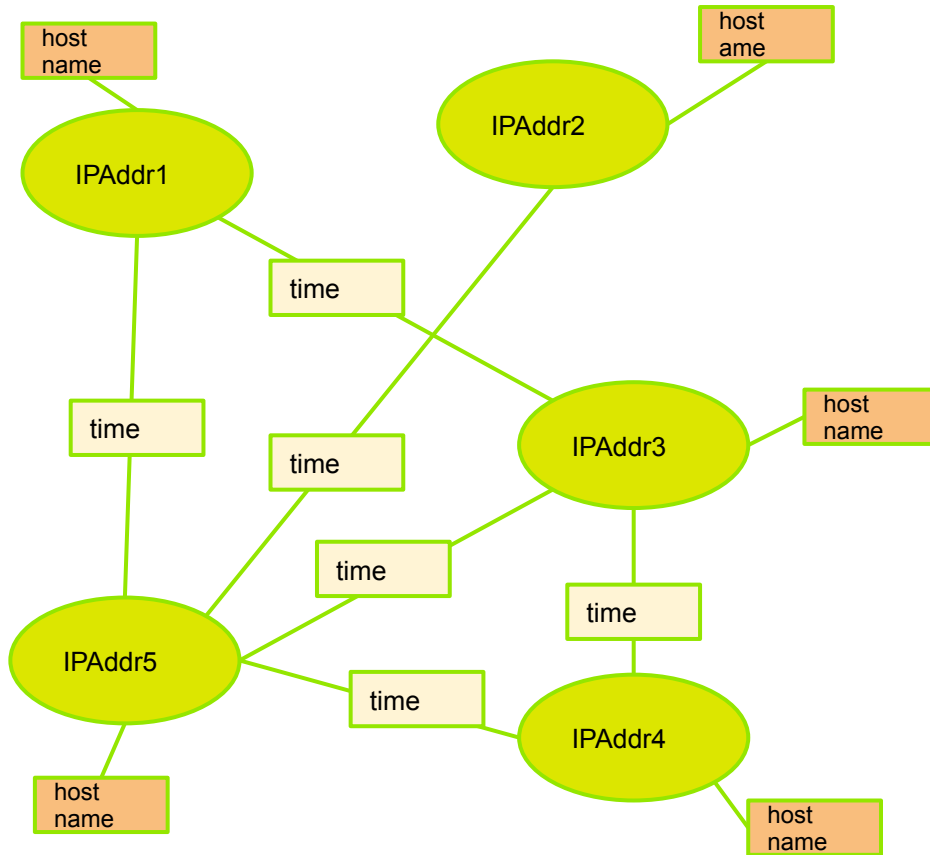
# Identify and collect DNS responses



# Tapestry: A real time SDN Analytics Application



# Tapestry Data Store



## Uses Digraph

- available in Erlang
- supports labels on vertices and edges
- digraph utilities



# Tapestry Filtering

## Default Settings

```
% capture all requester ip addresses
    {requester_whitelist,[{"0.0.0.0",0},
{":::",0}]},
    {requester_blacklist,[]},
% accept all resolved ip addresses
    {resolved_whitelist,[{"0.0.0.0",0},{":::",
0}]},
    {resolved_blacklist,[]},
% query filters (regular expressions)
    {query_whitelist,[""]},
    {query_blacklist,[]},
```

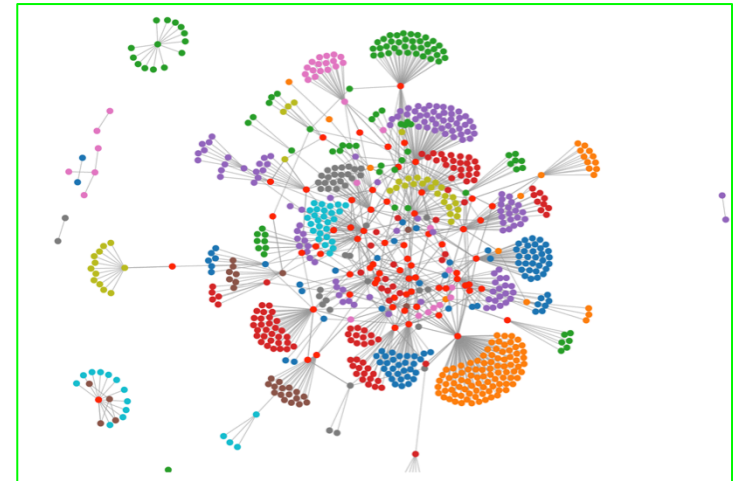
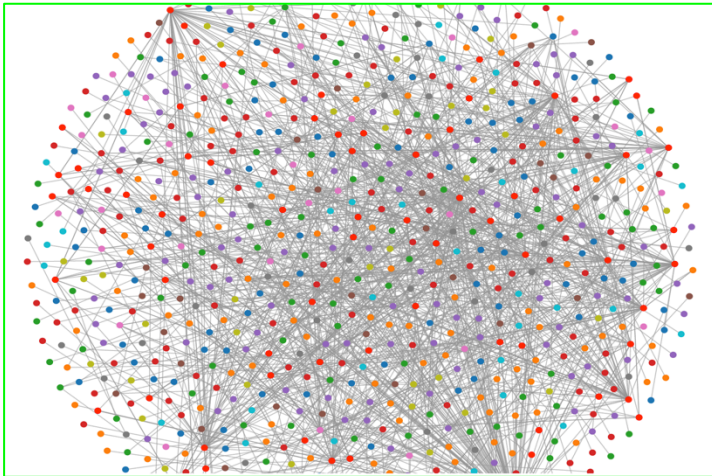
## Settings in deployment

```
{requester_blacklist,[{"10.102.3.50",32}]}.  
{resolved_blacklist,[{"10.102.3.50",32}]}.  
{query_blacklist,["^google.com$"  
"\.google.com$"]}.  

```

# Tapestry Analytics: Community Detection

- A network is said to have community structure if the nodes of the network can be easily grouped into sets of nodes such that each set of nodes is densely connected internally and more sparsely connected outside the group.
- The number of communities in a network and their sizes are not known beforehand and they are established by the community detection algorithm.
- Two community detection algorithm implemented: Label Propagation Algorithm and Louvain Method.



# Tapestry Analytics: Community Detection

- Each node is initialized with a unique label and at every iteration of the algorithm each node adopts a label that a maximum number of its neighbors have with ties broken uniformly randomly.
- As the labels propagate through the network in this manner, densely connected groups of nodes form a consensus on their labels.
- At the end of the algorithm, nodes having the same labels are grouped together as communities.
- The advantage of this algorithm over the other methods is its simplicity and time efficiency. The algorithm uses the network structure to guide its progress and does not optimize any specific chosen measure of community strengths.

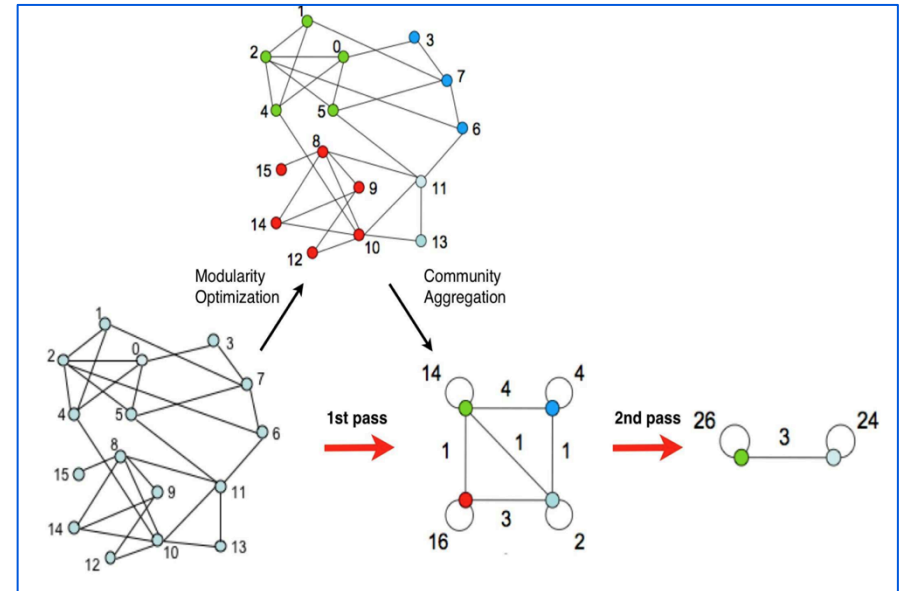
```
prop_labels(G)->
  random:seed(),
  Vertices = digraph:vertices(G),
  SplitValue = random:uniform(length(Vertices)),
  {V1,V2} = lists:split(SplitValue,Vertices),
  V = V2 ++ V1,
  RunCond = lists:foldl(fun(Vertex,Acc)->
    {StopCount,GoCount,G2} = Acc,
    Result = label_vertex(G,Vertex)
    case Result of
    go ->
      {StopCount,GoCount+1,G2};
    stop ->
      {StopCount+1,GoCount,G2}
    end
  end,{0,0,G},V),
  {NewStopCount,NewGoCount,G3} = RunCond,
  case NewGoCount > 0 of
  true ->
    prop_labels(G3);
  false ->
    G3
  end.
```

# Community Detection using Louvain Method

The algorithm is divided in two phases that are repeated iteratively.

Assume that we start with a weighted network of  $N$  nodes. First, we assign a different community to each node of the network. Then, for each node  $i$  we consider the neighbours  $j$  of  $i$  and we evaluate the gain of modularity that would take place by removing  $i$  from its community and by placing it in the community of  $j$ . The node  $i$  is then placed in the community for which this gain is maximum but only if this gain is positive. If no positive gain is possible,  $i$  stays in its original community. This process is applied repeatedly and sequentially for all nodes until no further improvement can be achieved and the first phase is then complete.

The second phase of the algorithm consists in building a new network whose nodes are now the communities found during the first phase.



Modularity is a benefit function that measures the quality of a particular division of a network into communities. Networks with high modularity have dense connections between the nodes within modules but sparse connections between nodes in different modules.

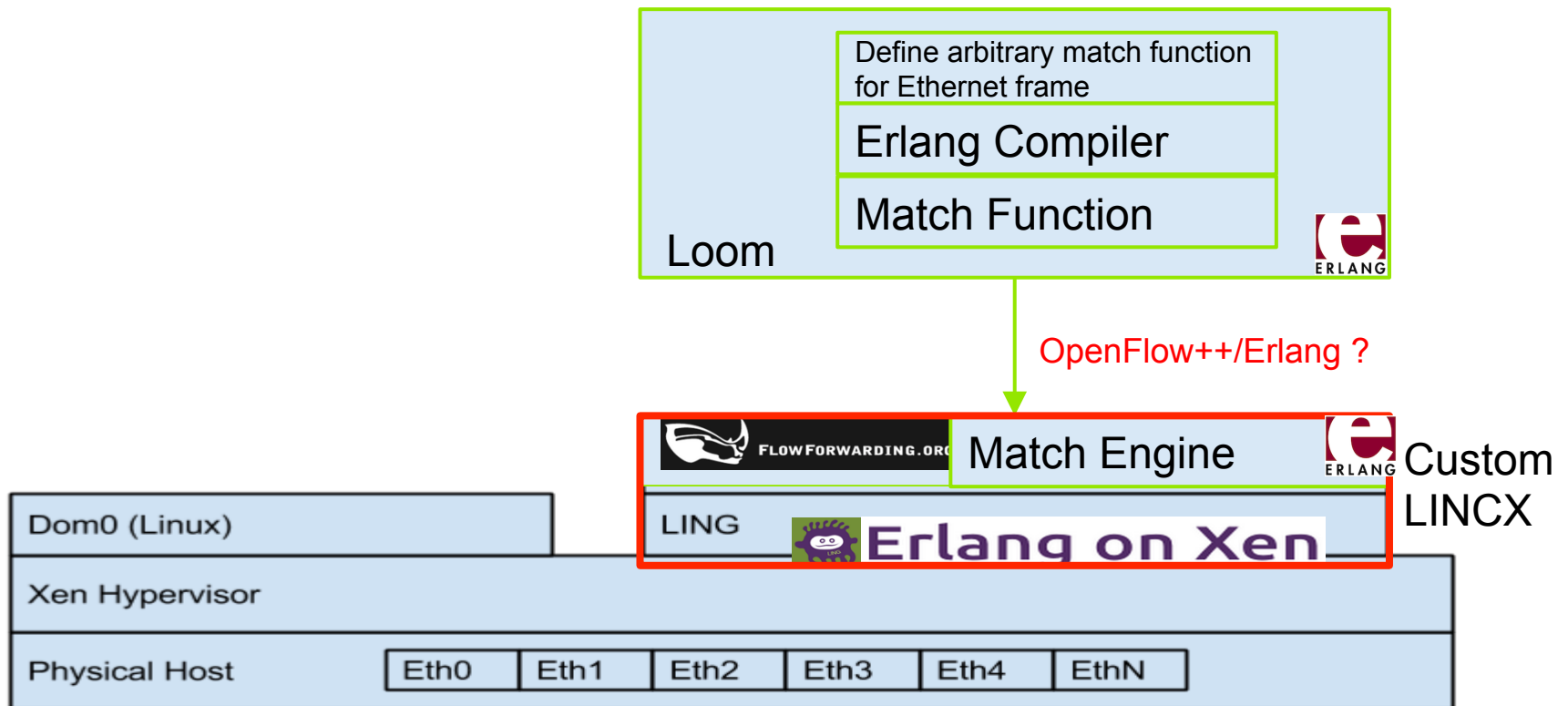
# DPI for DNS positive responses

- Code segment uses Erlang's pattern matching syntax and networking libraries
- Uses DNS request/response parsing function available in Erlang kernel
- Identifies DNS responses that have valid A records in them.
- DPI done at the controller

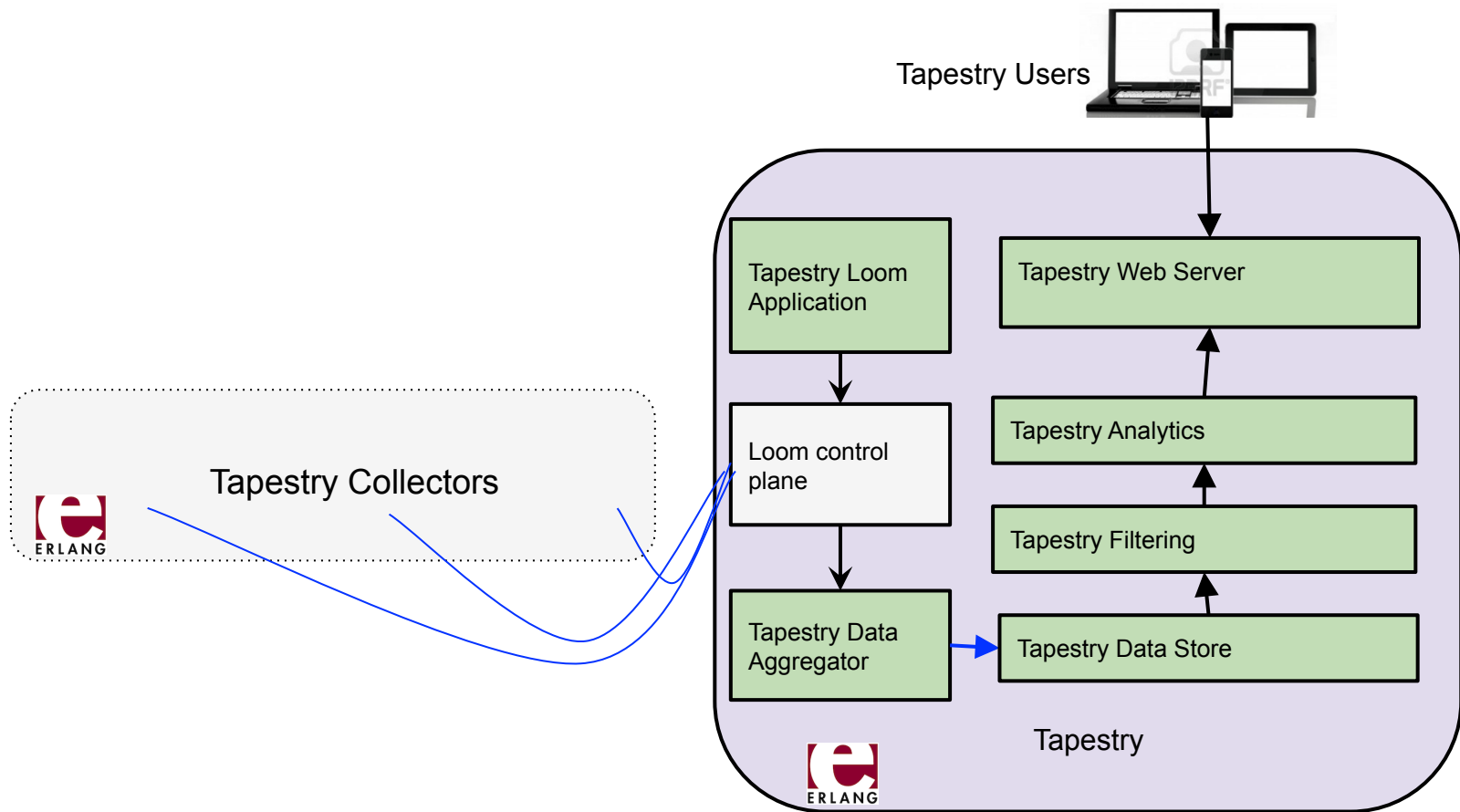
```
Result = case (Type1 == ipv4) and (Type2 == udp) of
    true ->
        inet_dns:decode(Payload);
    _ -> unknown
end,
case Result of
    {ok, DnsRec} ->
        Match = match_reply(DnsRec),
        .....
end
```

```
match_reply({dns_rec, {dns_header, _, true, _, _, _,
_, _, _, _},
    [{dns_query, Query, a, in}], RRSets, _, _}) ->
    Record = lists:keyfind(a, 3, RRSets),
    case Record of
        false ->
            {error, no_a_record};
        {dns_rr, _, a, _, _, _, ID, _, _, _} ->
            {ok, ID, Query}
    end;
match_reply(_) ->
    {error, bad_response}.
```

# Custom LINCX - match anything in Ethernet frame



# Tapestry: A platform for IoE





# Thank you



**[www.flowforwarding.org](http://www.flowforwarding.org)**