Toward Scalable Docker-Based Emulations of Blockchain Networks

Diego Pennino and Maurizio Pizzonia

26 May 2023, Bologna

5^{th} Distributed Ledger Technology Workshop
DLT 2023

Blockchain Network Testing

Performing realistic experiments for blockchain networks is notoriously hard.

Blockchain Network Testing

Performing realistic experiments for blockchain networks is notoriously hard.

The complexity:

Blockchain Network Testing

Performing realistic experiments for blockchain networks is notoriously hard.

The complexity:

- Large number of nodes,

Blockchain Network Testing

Performing realistic experiments for blockchain networks is notoriously hard.

The complexity:

- Large number of nodes,
- Software complexity,

Blockchain Network Testing

Performing realistic experiments for blockchain networks is notoriously hard.

The complexity:

- Large number of nodes,
- Software complexity,
- Communications among nodes (properties of transport protocols),

Blockchain Network Testing

Performing realistic experiments for blockchain networks is notoriously hard.

The complexity:

- Large number of nodes,
- Software complexity,
- Communications among nodes (properties of transport protocols),
- Nodes are spread over the internet (delay and packet loss)

Blockchain Network Testing

Performing realistic experiments for blockchain networks is notoriously hard.

The complexity:

- Large number of nodes,
- Software complexity,
- Communications among nodes (properties of transport protocols),
- Nodes are spread over the internet (delay and packet loss)

Emulation or **Simulation**???

Blockchain Network Testing

Performing realistic experiments for blockchain networks is notoriously hard.

The complexity:

- Large number of nodes,
- Software complexity,
- Communications among nodes (properties of transport protocols),
- Nodes are spread over the internet (delay and packet loss)

Simulation

Emulation or **Simulation**???

Blockchain Network Testing

Performing realistic experiments for blockchain networks is notoriously hard.

The complexity:

• Large number of nodes,	**Simulation**
• Software complexity,	**Emulation**
• Communications among nodes (properties of transport protocols),	**Emulation**
• Nodes are spread over the internet (delay and packet loss)	**Emulation**

Emulation or **Simulation**???

Blockchain Network Testing

Performing realistic experiments for blockchain networks is notoriously hard.

The complexity:

• Large number of nodes,	**Simulation**
• Software complexity,	**Emulation**
• Communications among nodes (properties of transport protocols),	**Emulation**
• Nodes are spread over the internet (delay and packet loss)	**Emulation**

Emulation or **Simulation**???

Emulation: two choices

Emulation: two choices

PRO:

- Simple to handle
- Simple to modify

CONS:

- Small amount of nodes (few hundreds)
- Fake Networks

Local

Emulation: two choices

PRO:

- Huge amount of nodes
- “Real” Network environment

CONS:

- Hard to handle
- Slow to modify
- Clusters

Distributed

PRO:

- Simple to handle
- Simple to modify

CONS:

- Small amount of nodes (few hundreds)
- Fake Networks

Local

Emulation: Our Solution

PRO:

- Huge amount of nodes
- "Real" Network environment

CONS:

- Hard to handle
- Slow to modify
- Clusters

Distributed

PRO:

- Simple to handle
- Simple to modify

CONS:

- Small amount of nodes (few hundreds)
- Fake Networks

Local

Emulation: Our Solution

PRO:

- Huge amount of nodes
- "Real" Network environment

CONS:

- Hard to handle
- Slow to modify
- Clusters

Boosted Hardware on premises

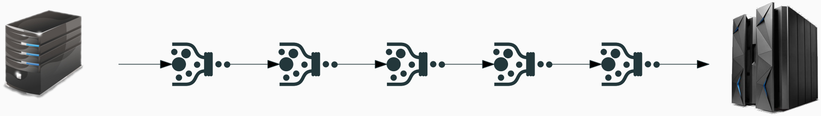
PRO:

- Simple to handle
- Simple to modify

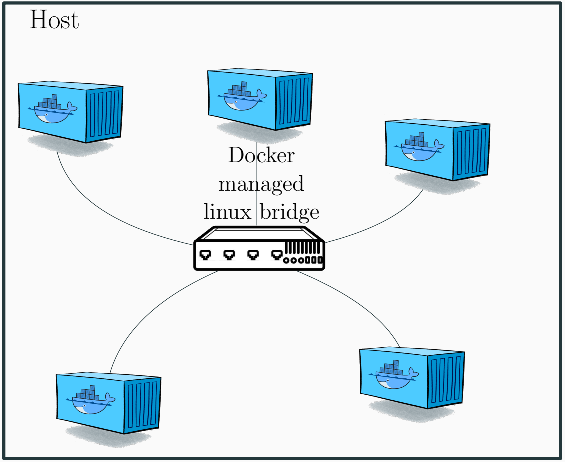
CONS:

- Small amount of nodes (few hundreds)
- Fake Networks

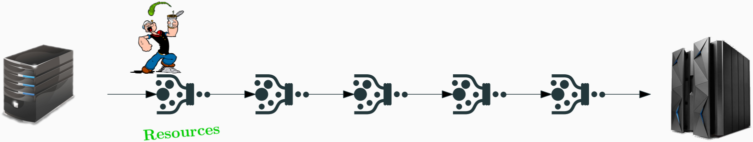
Roadmap



The typical Docker scenario



1^{st} bottleneck: Resources usage



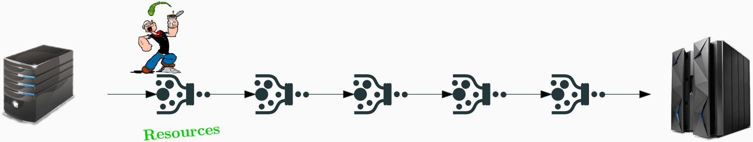
Security limits

`/etc/security/limits.conf`

Kernel parameters

`/etc/sysctl.conf`

1^{st} bottleneck: Resources usage

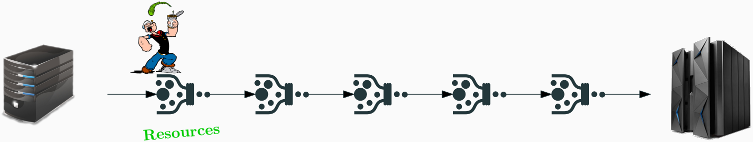


Security limits
`/etc/security/limits.conf`

- **nofile** number of open files
- **nproc** maximum number of processes

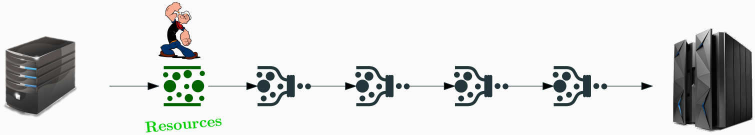
Kernel parameters
`/etc/sysctl.conf`

1^{st} bottleneck: Resources usage



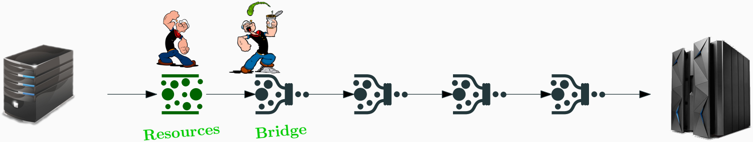
Security limits /etc/security/limits.conf	**nofile**	number of open files	
	nproc	maximum number of processes	
Kernel parameters /etc/sysctl.conf	**pty**	maximum number of pseudo-terminal	def:4096
	gc_thresh1	garbage collector ARP entries	def:128
	gc_thresh2	garbage collector ARP entries	def:512
	gc_thresh3	garbage collector ARP entries	def:1024

1^{st} bottleneck: Resources usage

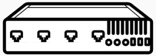


Security limits /etc/security/limits.conf	**nofile**	number of open files	
	nproc	maximum number of processes	
Kernel parameters /etc/sysctl.conf	**pty**	maximum number of pseudo-terminal	def:4096
	gc_thresh1	garbage collector ARP entries	def:128
	gc_thresh2	garbage collector ARP entries	def:512
	gc_thresh3	garbage collector ARP entries	def:1024

2^{nd} bottleneck: The Bridge

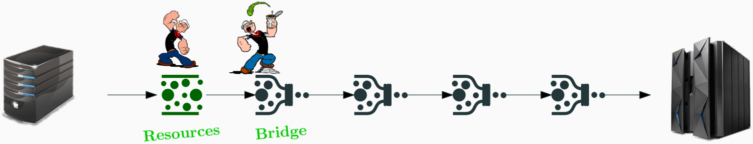


Default Linux Bridge

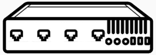


$2^{10} = 1024$ ports

2^{nd} bottleneck: The Bridge

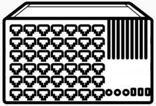


Default Linux Bridge



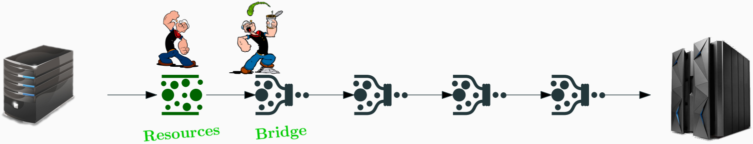
$2^{10} = 1024$ ports

Our Linux Bridge

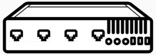


$2^{17} = 131{,}072$ ports

2^{nd} bottleneck: The Bridge

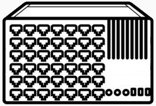


Default Linux Bridge



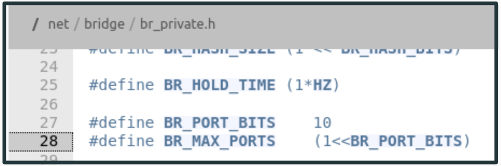
$2^{10} = 1024$ ports

Our Linux Bridge

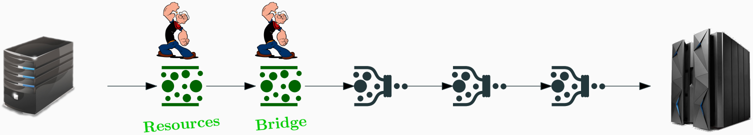


$2^{17} = 131{,}072$ ports

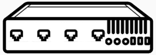
Kernel



2^{nd} bottleneck: The Bridge

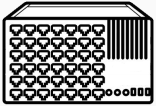


Default Linux Bridge



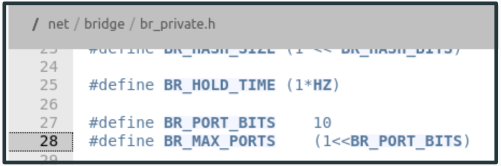
$2^{10} = 1024$ ports

Our Linux Bridge



$2^{17} = 131{,}072$ ports

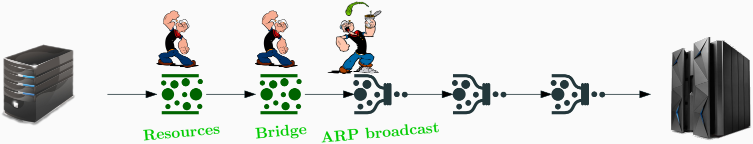
Kernel



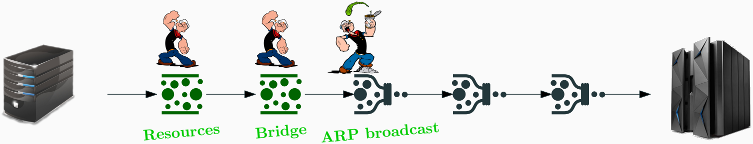
/ net / bridge / br_private.h

```
#define BR_HOLD_TIME (1*HZ)

#define BR_PORT_BITS    10
#define BR_MAX_PORTS    (1<<BR_PORT_BITS)
```

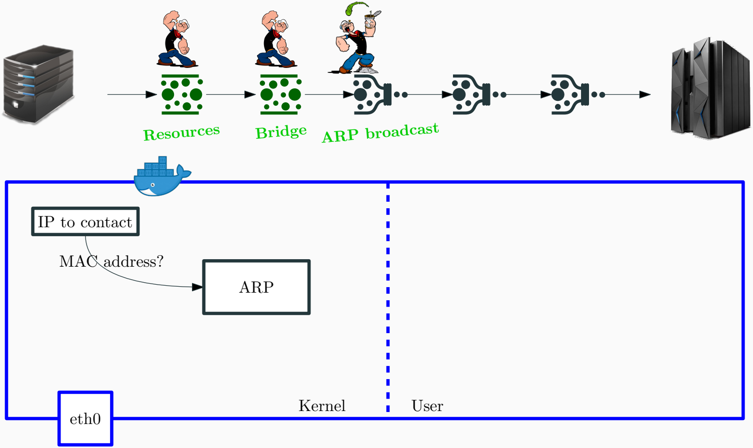
3^{rt} bottleneck: ARP Broadcast



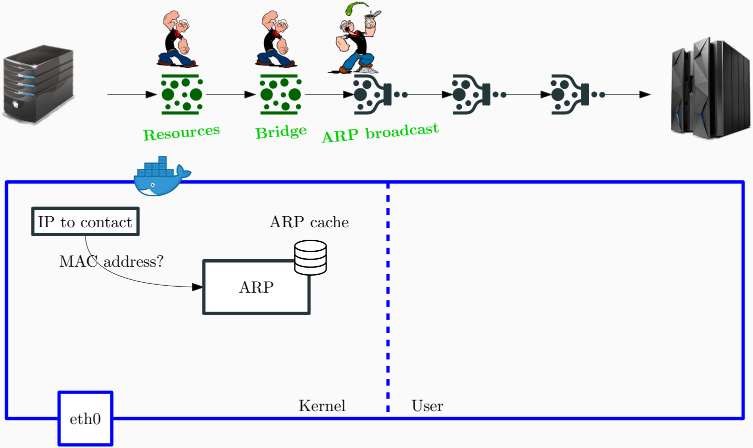
3^{rt} bottleneck: ARP Broadcast



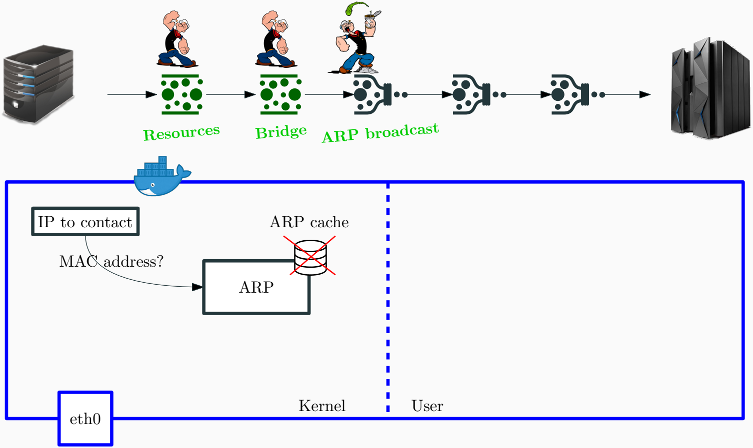
3^{rt} bottleneck: ARP Broadcast



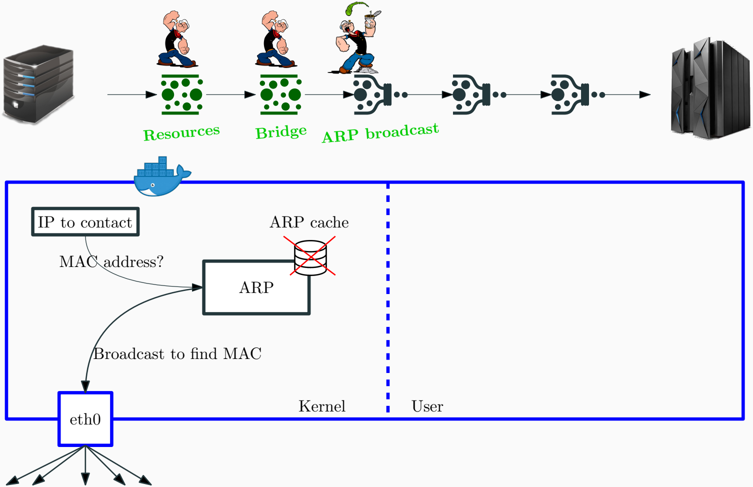
3^{rt} bottleneck: ARP Broadcast



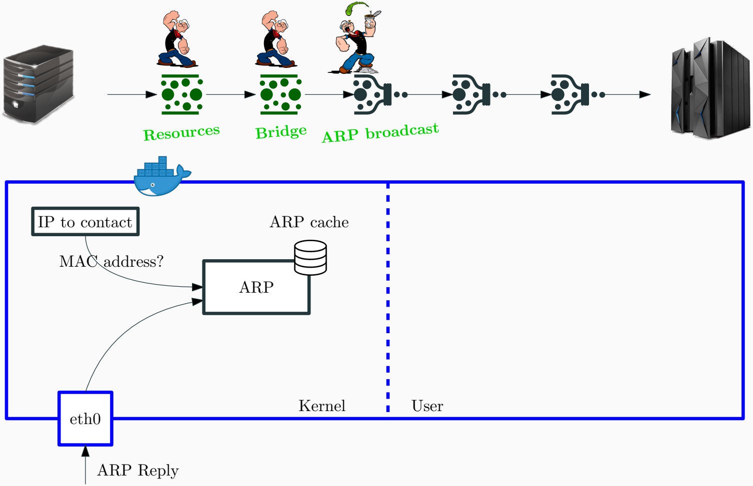
3^{rt} bottleneck: ARP Broadcast



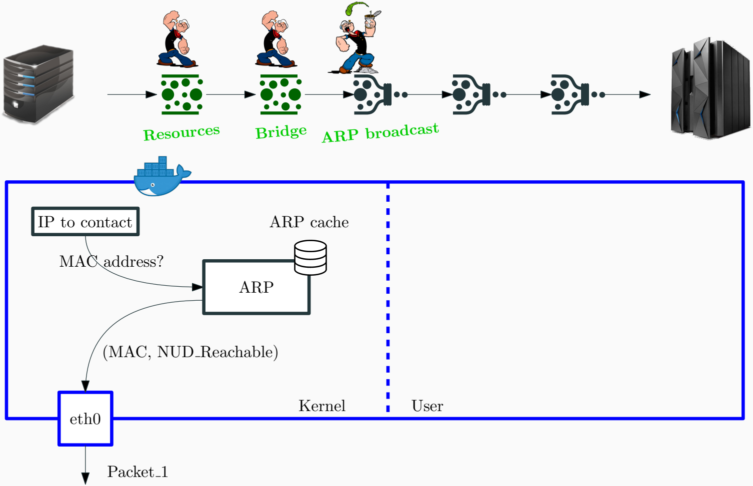
3^{rt} bottleneck: ARP Broadcast



3^{rt} bottleneck: ARP Broadcast

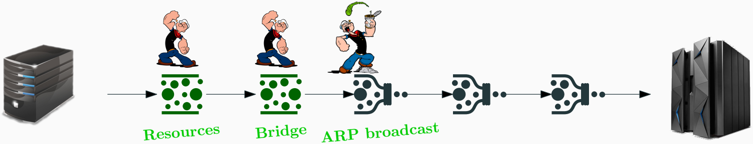


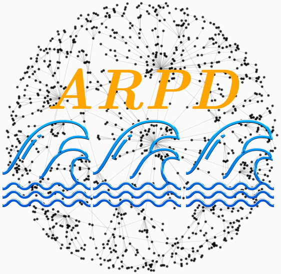
3^{rt} bottleneck: ARP Broadcast



Neighbour Unreachability Detection(NUD): *Reachable*= Valid entry recently used

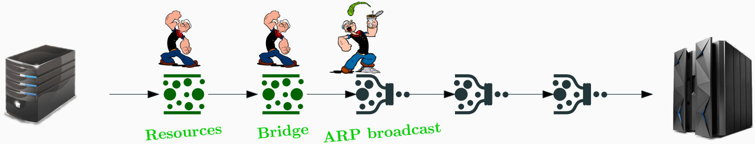
3^{rt} bottleneck: ARP Broadcast

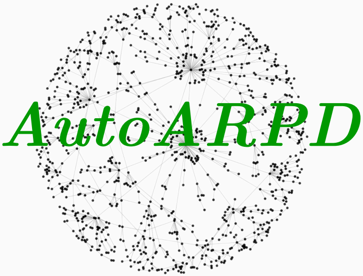




Arpd. https://github.com/shemminger/iproute2/blob/main/misc/arpd.c

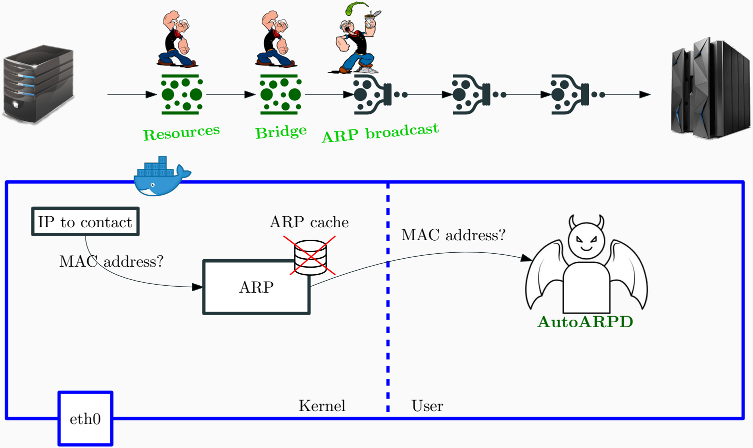
3^{rt} bottleneck: ARP Broadcast





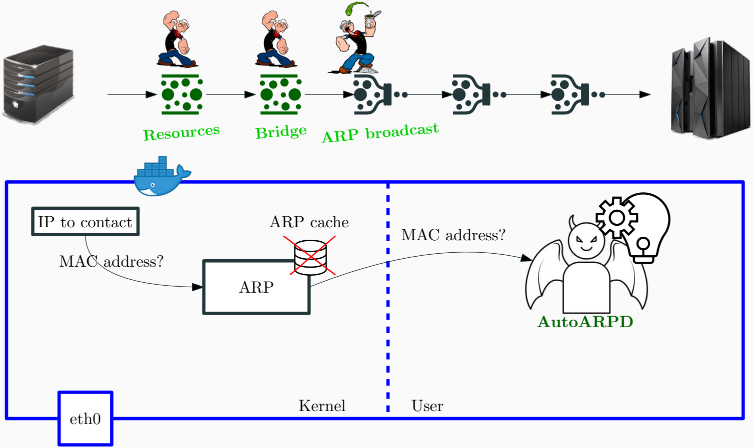
AutoArpd. https://gitlab.com/uniroma3/compunet/networks/AutoARPD

3^{rt} bottleneck: ARP Broadcast



AutoArpd. https://gitlab.com/uniroma3/compunet/networks/AutoARPD

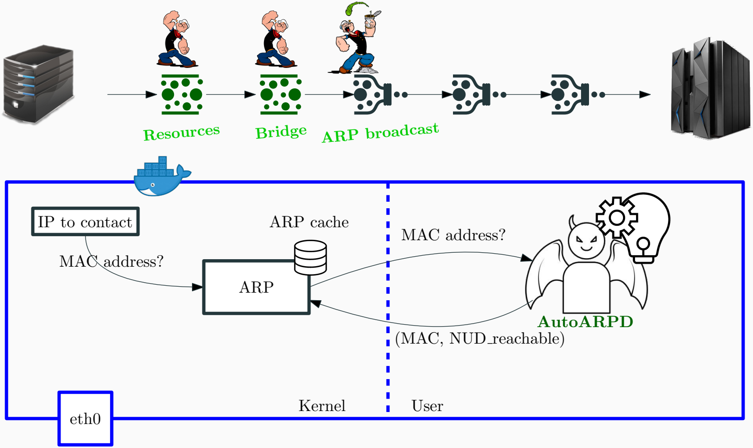
3^{rt} bottleneck: ARP Broadcast



Usage: `autoarpd <RULE> [ interfaces ]` **E.g**: `autoarpd 02:42:ip1:ip2:ip3:ip4 eth0`

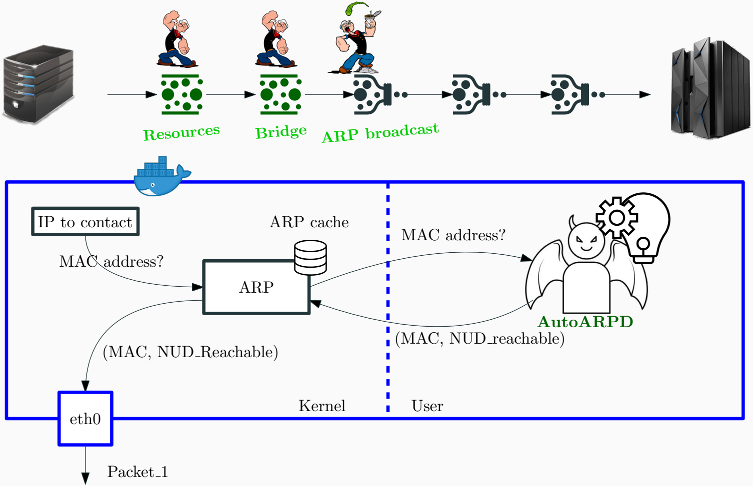
AutoArpd. https://gitlab.com/uniroma3/compunet/networks/AutoARPD

3^{rt} bottleneck: ARP Broadcast



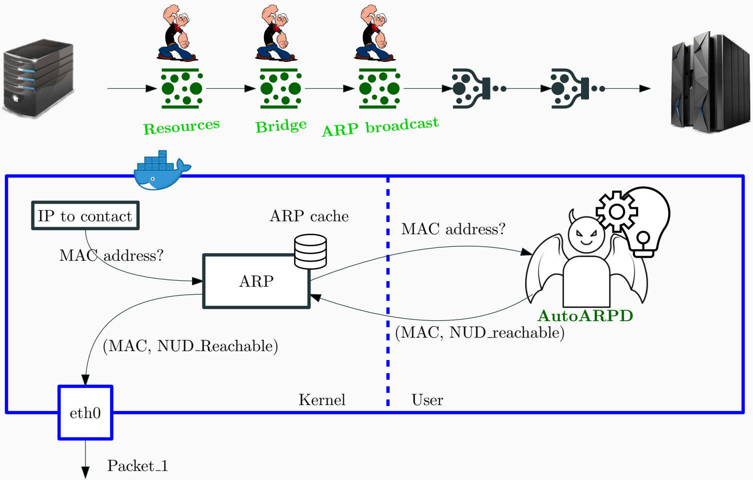
AutoArpd. https://gitlab.com/uniroma3/compunet/networks/AutoARPD

3^{rt} bottleneck: ARP Broadcast



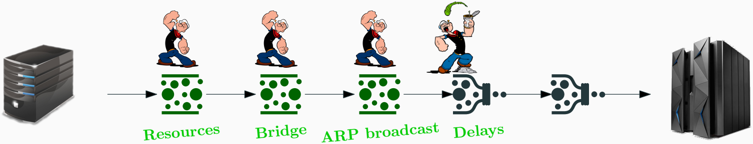
AutoArpd. https://gitlab.com/uniroma3/compunet/networks/AutoARPD

3^{rt} bottleneck: ARP Broadcast

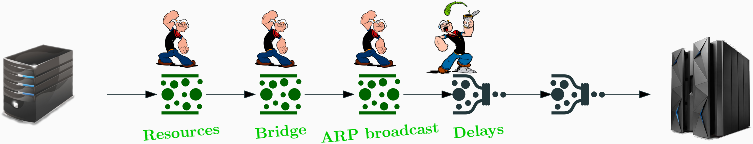


AutoArpd. https://gitlab.com/uniroma3/compunet/networks/AutoARPD

4^{th} bottleneck: Emulating Realistic Internet Delays



4^{th} bottleneck: Emulating Realistic Internet Delays




IEEE/ACM TRANSACTIONS ON NETWORKING, VOL. 18, NO. 1, FEBRUARY 2010

Measurement-Based Analysis, Modeling, and Synthesis of the Internet Delay Space

Bo Zhang, Tze Sing Eugene Ng, Animesh Nandi, Rudolf H. Riedi, Peter Druschel, and Guohui Wang

***Abstract*—Understanding the characteristics of the Internet delay space (i.e., the all-pairs set of static round-trip propagation delays among edge networks in the Internet) is important for the design of global-scale distributed systems. For instance, algorithms used in overlay networks are often sensitive to violations of the triangle inequality and to the growth properties within the Internet delay space. Since designers of distributed systems often rely on simulation and emulation to study design alternatives, they need a realistic model of the Internet delay space. In this paper, we analyze measured delay spaces among thousands of Internet edge networks and quantify key properties that are important for distributed system design. Our analysis shows that existing delay space models do not adequately capture these important properties of the Internet delay space. Furthermore, we derive a simple model of the Internet delay space based on our analytical findings. This model preserves the relevant metrics far better than existing models, allows for a compact representation, and can be used to synthesize delay data for simulations and emulations at a scale where direct measurement and storage are impractical. We present the design of a publicly available delay space synthesizer tool called DS^2 and demonstrate its effectiveness.**

***Index Terms*—Analysis, distributed system, Internet delay space, measurement, modeling, simulation, synthesis.**

I. INTRODUCTION

DESIGNERS of large-scale distributed systems rely on simulation and network emulation to study design alternatives and evaluate prototype systems at scale and prior to deployment. To obtain accurate results, such simulations or emulations must include an adequate model of the *Internet delay space*, the all-pairs set of static round-trip propagation delays among edge networks. Such a model must accurately reflect those characteristics of real Internet delays that influence system performance. For example, having realistic clustering properties is important because they can influence the load balance of delay-optimized overlay networks and the effectiveness of server placement policies and caching strategies. Having realistic growth characteristics [16] is equally important because the effectiveness of certain distributed algorithms depends on them. Many distributed systems are also sensitive to the inefficiency of IP routing with respect to delay. Such inefficiency manifests itself as triangle inequality violations in the delay space and must be reflected in a model as well.

Currently, two approaches are used to obtain a delay model. The first approach, adopted for instance by the p2psim simulator [24], is to collect actual delay measurements using a tool such as King [14]. However, due to limitations of the measurement methodology and the quadratic time requirement for measuring a delay matrix, measured data tend to be incomplete and there are limits to the size of a delay matrix that can be measured in practice. To its credit, p2psim provides a 1740 × 1740 delay space matrix, which is not a trivial amount of data to obtain.

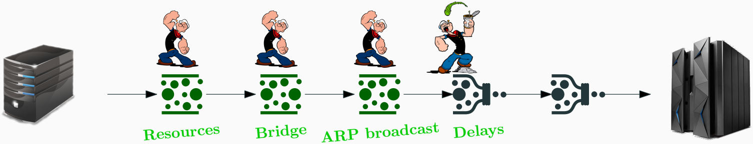
The second approach is to start with a statistical network topology model (e.g., [8], [10], [18], [44], and [47]) and assign artificial link delays to the topology. The delay space is then modeled by the all-pair shortest-path delays within the topology. The properties of such delay models, however, tend to differ dramatically from the actual Internet delay space. This is because these models do not adequately capture rich features in the Internet delay space, such as those caused by geographic constraints, variation in node concentrations, and routing inefficiency.

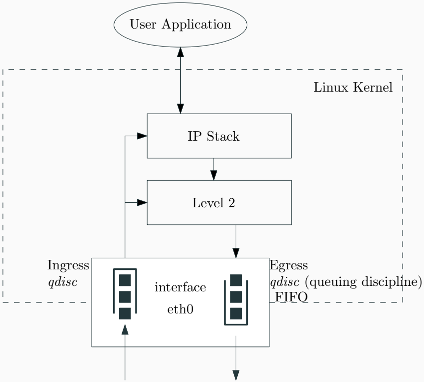
A delay space model suitable for large-scale simulations must adequately capture the relevant characteristics of the Internet delay space. In addition, the model must have a compact representation since large-scale simulations tend to be memory-bound. The naive approach of storing 16-bit delay values for

Delay

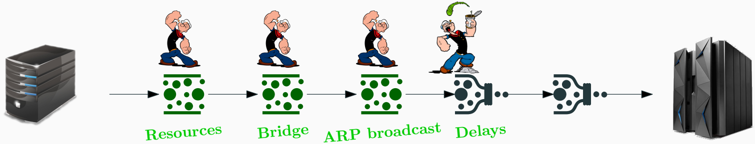
Rice University, Internet delay space synthesizer `https://www.cs.rice.edu/~eugeneng/research/ds2/`

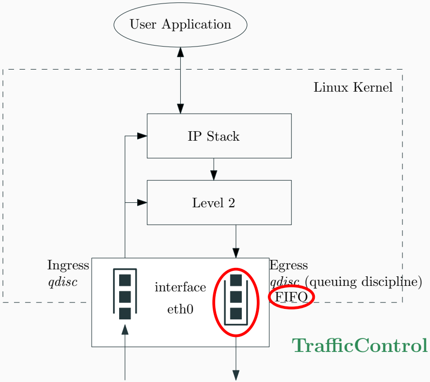
4^{th} bottleneck: Emulating Realistic Internet Delays





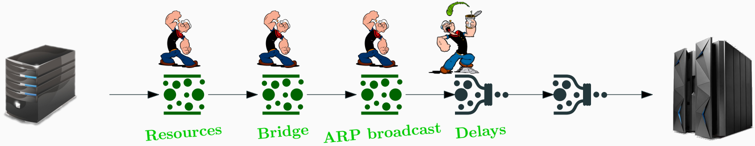
4^{th} bottleneck: Emulating Realistic Internet Delays

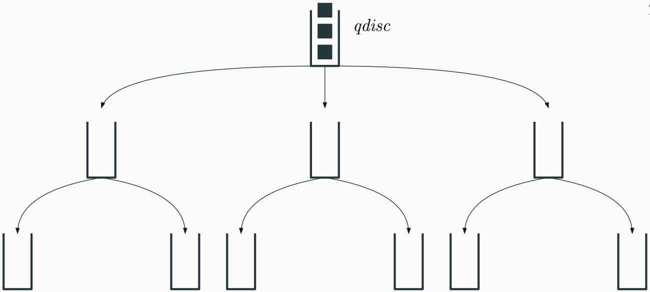




TrafficControl (TC)

4^{th} bottleneck: Emulating Realistic Internet Delays

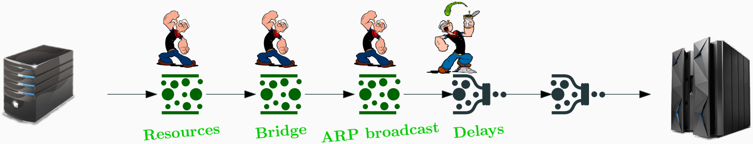


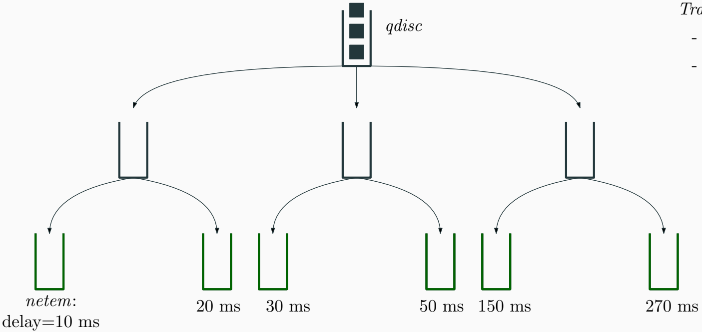


TrafficControl (TC)

- qdiscs structure

4^{th} bottleneck: Emulating Realistic Internet Delays

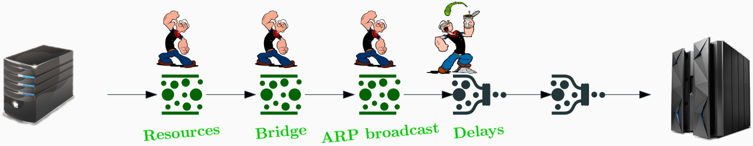


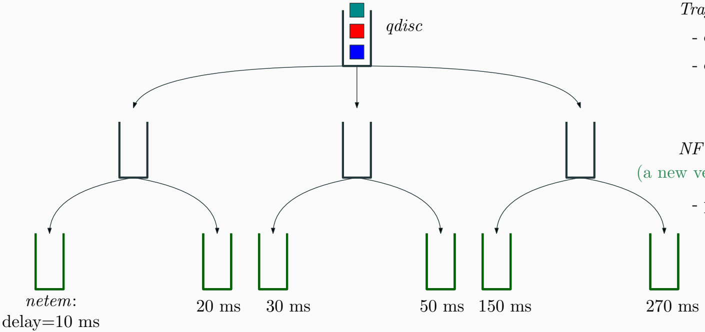


TrafficControl (TC)

- qdiscs structure
- qdiscs types

4^{th} bottleneck: Emulating Realistic Internet Delays





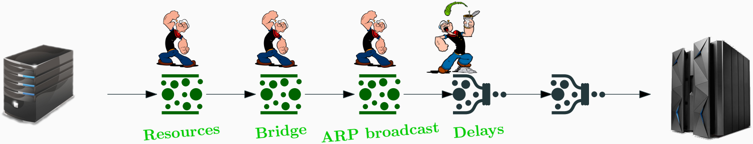
TrafficControl (TC)

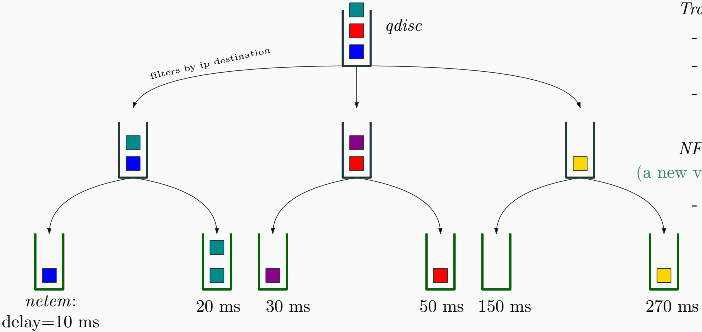
- qdiscs structure
- qdiscs types

NFTables (NFT)
(a new version of ip tables)

- packets coloring

4^{th} bottleneck: Emulating Realistic Internet Delays





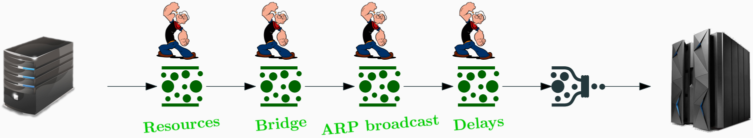
TrafficControl (TC)

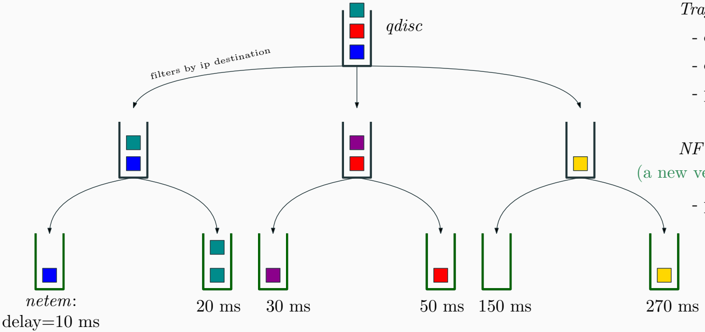
- qdiscs structure
- qdiscs types
- packets filters

NFTables (NFT)
(a new version of ip tables)

- packets coloring

4^{th} bottleneck: Emulating Realistic Internet Delays





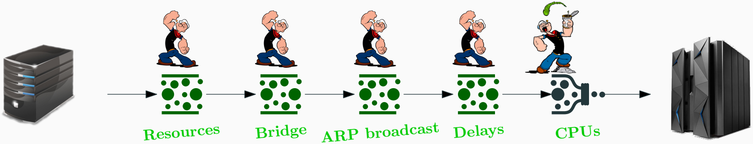
TrafficControl (TC)

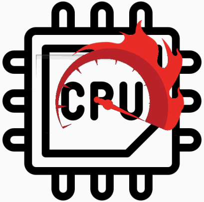
- qdiscs structure
- qdiscs types
- packets filters

NFTables (NFT)
(a new version of ip tables)

- packets coloring

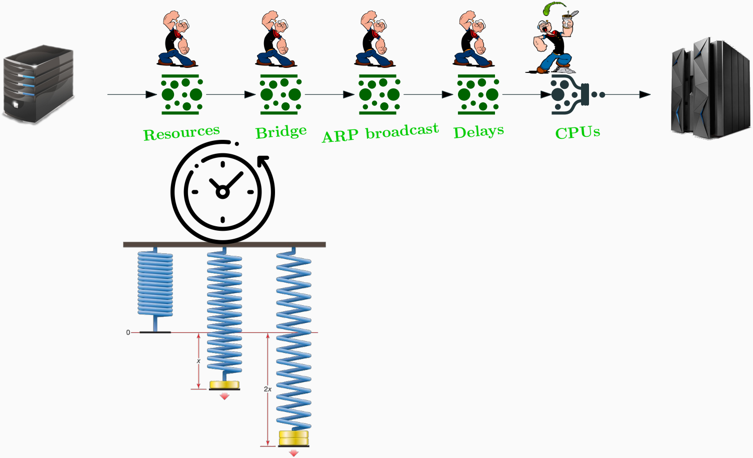
5^{th} bottleneck: CPUs workload





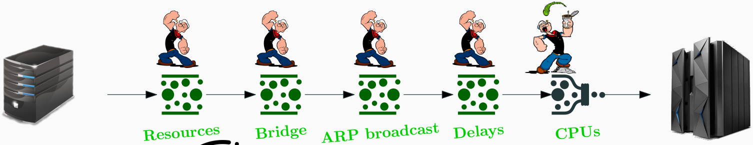
CPUs workload

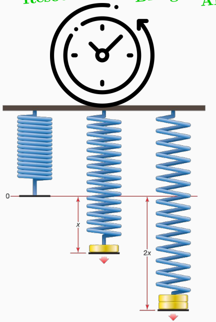
5^{th} bottleneck: CPUs workload



Time *inflation*

5^{th} bottleneck: CPUs workload

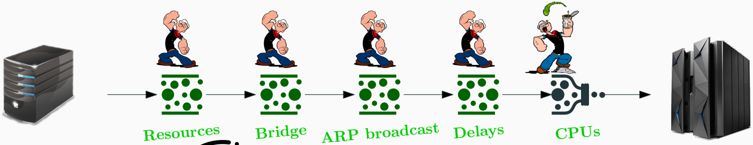


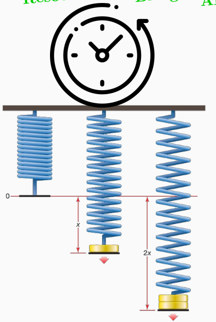


Time *inflation*

- Network *delay*

5^{th} bottleneck: CPUs workload

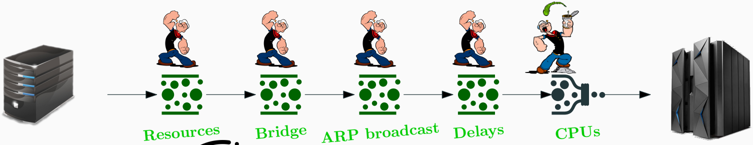


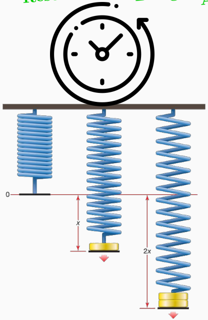


Time *inflation*

- Network *delay*
- Blockchain *software*

5^{th} bottleneck: CPUs workload

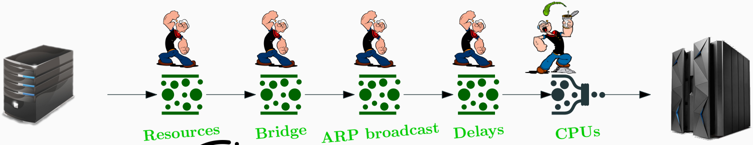


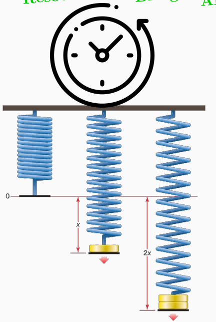


Time *inflation*

- Network *delay*
- Blockchain *software*
- Transaction *load*

5^{th} bottleneck: CPUs workload

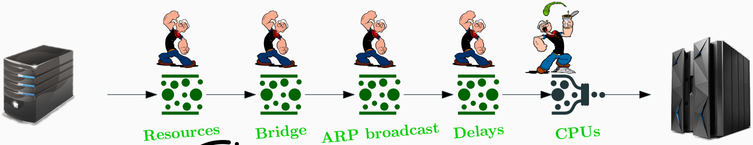


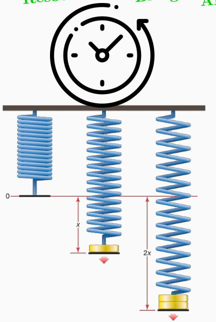


Time *inflation*

- Network *delay*
- Blockchain *software*
- Transaction *load*
- Kernel TCP *protocol*

5^{th} bottleneck: CPUs workload

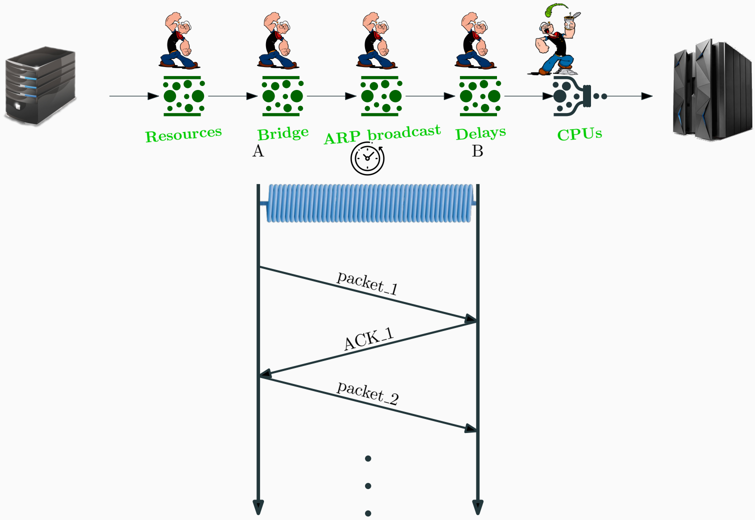




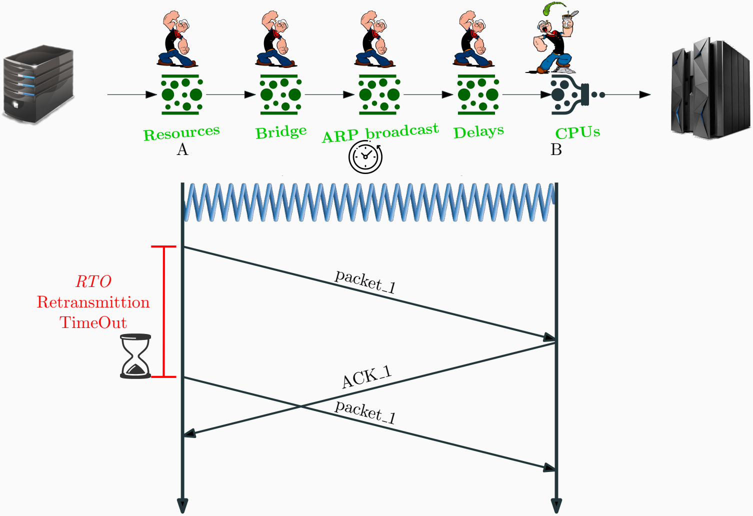
Time *inflation*

- Network *delay*
- Blockchain *software*
- Transaction *load*
- Kernel TCP *protocol*

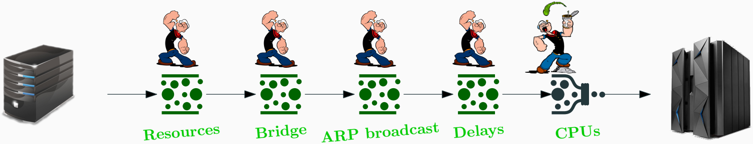
5^{th} bottleneck: CPUs workload



5^{th} bottleneck: CPUs workload



5^{th} bottleneck: CPUs workload



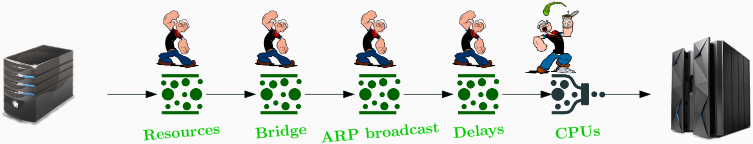
Berkeley Packet Filter *(BPF)* code

kernel

/ include / net / tcp.h

```c
static inline u32 tcp_timeout_init(struct sock *sk)
{
        int timeout;

        timeout = tcp_call_bpf(sk, BPF_SOCK_OPS_TIMEOUT_INIT, 0, NULL);

        if (timeout <= 0)
                timeout = TCP_TIMEOUT_INIT;
        return timeout;
}
```

5th bottleneck: CPUs workload



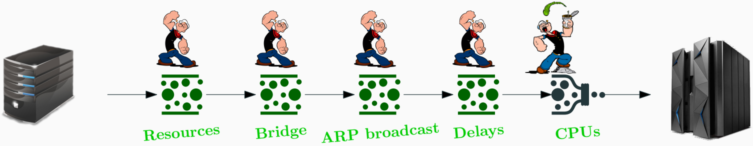
Berkeley Packet Filter (*BPF*) code

kernel

/ include / net / tcp.h

```
static inline u32 tcp_timeout_init(struct sock *sk)
{
        int timeout;

        timeout = tcp_call_bpf(sk, BPF_SOCK_OPS_TIMEOUT_INIT, 0, NULL);

        if (timeout <= 0)
                timeout = TCP_TIMEOUT_INIT;
        return timeout;
}
```

5th bottleneck: CPUs workload



Berkeley Packet Filter (*BPF*) code

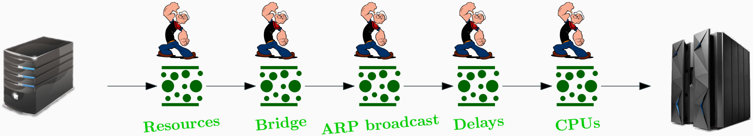
kernel

/ include / net / tcp.h

```c
static inline u32 tcp_timeout_init(struct sock *sk)
{
        int timeout;

        timeout = tcp_call_bpf(sk, BPF_SOCK_OPS_TIMEOUT_INIT, 0, NULL);

        if (timeout <= 0)
                timeout = TCP_TIMEOUT_INIT;
        return timeout;
}
```

```c
__section("sockops")
int set_initial_rto(struct bpf_sock_ops *skops)
{
        const int timeout = 3; // initial RTO timout in seconds
        const int hz = 250;  // this value has to match the HZ value of the system

        int op = (int) skops->op;
        if (op == BPF_SOCK_OPS_TIMEOUT_INIT) {
                skops->reply =  timeout * hz;
                return 1;
        }

        return 1;
}
```

5^{th} bottleneck: CPUs workload



Berkeley Packet Filter *(BPF)* code

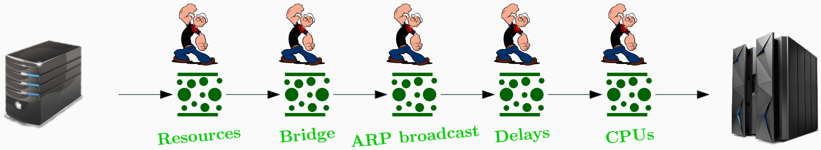
kernel

/ include / net / tcp.h

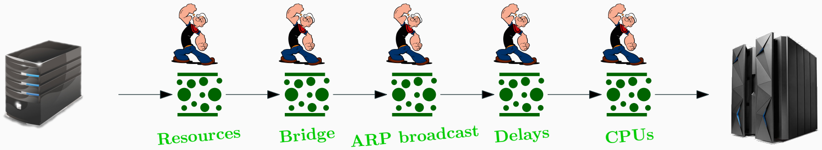
```c
static inline u32 tcp_timeout_init(struct sock *sk)
{
        int timeout;

        timeout = tcp_call_bpf(sk, BPF_SOCK_OPS_TIMEOUT_INIT, 0, NULL);

        if (timeout <= 0)
                timeout = TCP_TIMEOUT_INIT;
        return timeout;
}
```

```c
__section("sockops")
int set_initial_rto(struct bpf_sock_ops *skops)
{
        const int timeout = 3; // initial RTO timout in seconds
        const int hz = 250;  // this value has to match the HZ value of the system

        int op = (int) skops->op;
        if (op == BPF_SOCK_OPS_TIMEOUT_INIT) {
                skops->reply =  timeout * hz;
                return 1;
        }

        return 1;
}
```

Our Solution



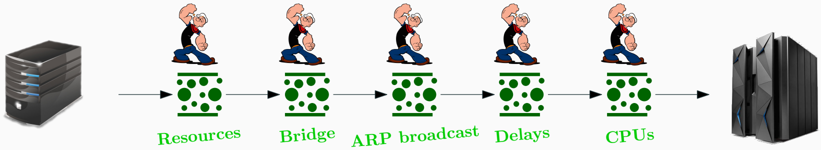
Our Solution



OUR Emulation:

- Huge amount of nodes ⟶ 3500 containers in 400GB RAM
- “Real” Network env. ⟶ end-to-end realistic internet delays, 8000 TCP-based and 64000 UDP-based connections
- Simple to handle ⟶ Makefile and Python scripts
- Simple to modify ⟶ Python scripts

Our Solution



Future works:

- ⋆ Simplify the setup
- ⋆ Multiple host (kubernetes)
- ⋆ Real software of a blockchain node
- ⋆ Create a library to create transaction load
- ⋆ Create a library to support data gathering

The End

Thank you!

diego.pennino@unitus.it, diego.pennino@uniroma3.it, pizzonia@ing.uniroma3.it