NFF-GO (YANFF) — YET ANOTHER NETWORK FUNCTION FRAMEWORK LABS

Intel Golang Team

Gregory Shimansky

Legal Information

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer.

No computer system can be absolutely secure.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.

Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit http://www.intel.com/performance.

Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase. For more complete information about performance and benchmark results, visit http://www.intel.com/performance.

Cost reduction scenarios described are intended as examples of how a given Intel- based product, in the specified circumstances and configurations, may affect future costs and provide cost savings. Circumstances will vary. Intel does not guarantee any costs or cost reduction.

Results have been estimated or simulated using internal Intel analysis or architecture simulation or modeling, and provided to you for informational purposes. Any differences in your system hardware, software or configuration may affect your actual performance.

Intel does not control or audit third-party benchmark data or the web sites referenced in this document. You should visit the referenced web site and confirm whether referenced data are accurate.

Intel, the Intel logo and others are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

© 2017 Intel Corporation.

Optimization Notice

YANFF - Yet Another Network Function Framework

Framework for building performant native network functions

- Open-source project
- Higher level abstractions than DPDK
- Go language: productivity, performance, concurrency, safety
- Network functions are application programs and not virtual machines

Benefits:

- Easily leverage IA HW capabilities: multi-cores, AES-NI, CAT, QAT, DPDK
- 10x reduction lines of code
- No need to be expert network system programmer
- Similar performance with C
- Take advantage of cloud native deployment: continuous delivery, micro-services, containers

https://github.com/intel-go/yanff

Optimization Notice

Technical Motivation

- Developers need framework to shorten development cycle of VNFs
  - Currently VNFs are monolithic - “virtual appliances” instead of network functions
  - Significant part of VNF is about plumbing. Plumbing VNFs to CommSPs network is an art. Should be abstracted from VNFs
- Lack of stable and unified APIs for VNF control and data plane
- Challenges with access to HW Accelerators in cloud environment.
- Cloud-friendly APIs and designs needed.

Accelerating transition to from rule-based networking to *imperative networking*

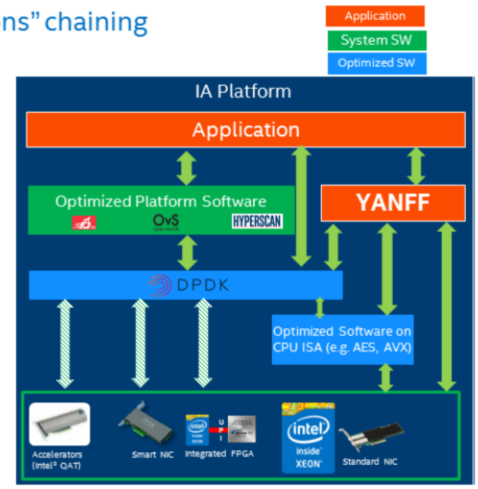
https://github.com/intel-go/yanff

Optimization Notice

VMs have relatively high overhead and memory footprint

YANFF: Yet Another Network Function Framework

- Simple but powerful abstractions:
  - Flow, Packet
- User builds packet processing graph using "flow functions" chaining
  - SetReceiver -> SetHandler -> SetSender
  - Several predefined possibilities of adding user processing inside packet processing graph
    - Split, Separate, Generate, Handle
- Can leverage predefined functions which parse packets, check ACL rules, etc.
- Run to completion – NFs can be expressed in the flow functions and natural chaining
- Auto-scaling, ease of development
- Zero-copy between NFs
- Flexible incoming flow handling – sources can be anything: network port, memory buffer, remote procedure call, etc.



https://github.com/intel-go/yanff

Optimization Notice

L3 Simple Forwarding Example

```
var L3Rules *rules.L3Rules

func main() {
	flow.SystemInit(16)
	L3Rules = rules.GetL3RulesFromORIG("Forwarding.conf")
	inputFlow := flow.SetReceiver(0)
	outputFlows := flow.SetSplitter(inputFlow, L3Splitter, uint(3))
	flow.SetStopper(outputFlows[0])
	for i := 1; i < 3; i++ {
		flow.SetSender(outputFlows[i], uint8(i-1))
	}
	flow.SystemStart()
}

// User defined function for splitting packets
func L3Splitter(currentPacket *packet.Packet) uint {
	currentPacket.ParseL4()
	return rules.L3_ACL_port(currentPacket, L3Rules)
}
```

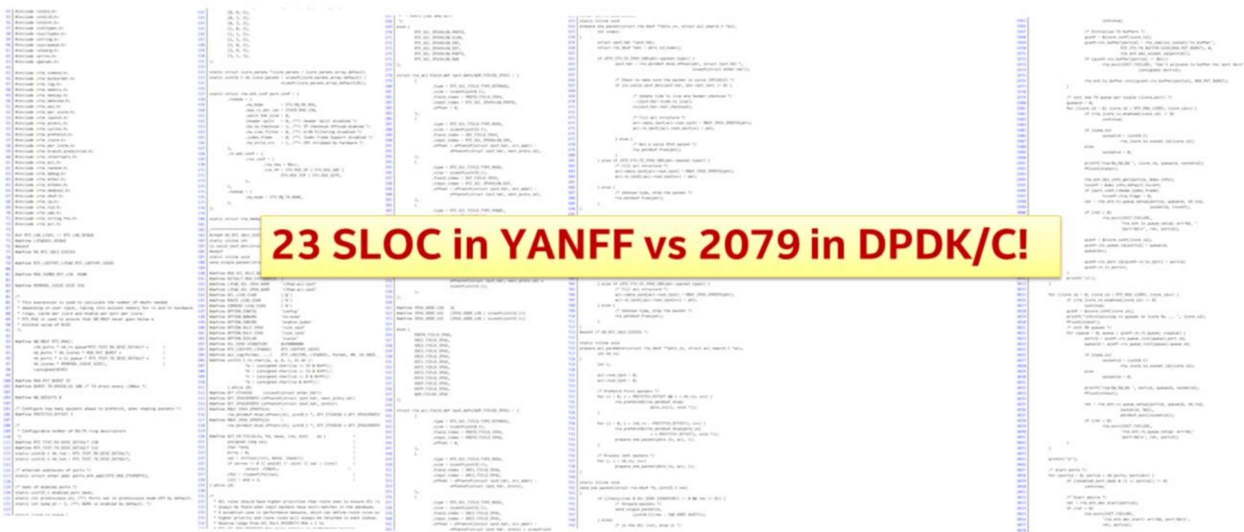

Optimization Notice

Configuration file for Forwarding

```
# Source address, Destination address, L4 protocol ID, Source port, Destination port, Output port
  111.2.0.0/31     ANY     tcp     ANY          ANY        1
  111.2.0.2/32     ANY     tcp     ANY          ANY        Reject
  ANY              ANY     udp     3078:3964    56:61020   2
```


Optimization Notice

Exactly The Same Example in DPDK/C



23 SLOC in YANFF vs 2079 in DPDK/C!

Optimization Notice

YANFF – Main Architectural Concepts

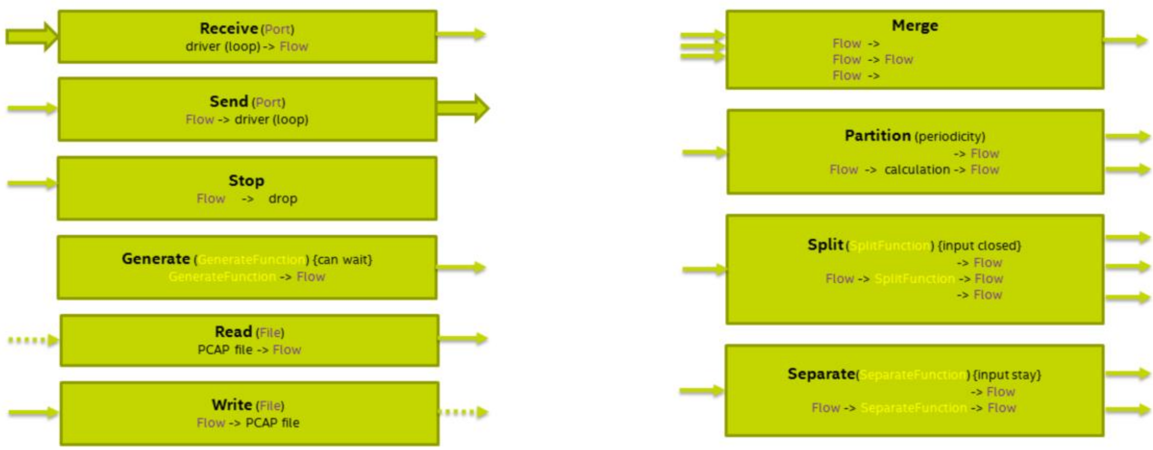
Flow
Abstraction without public fields, which is used for pointing connections between Flow functions.
Opened by Receive / Split / Separate / Counter / Generate.
Closed by Send / Merge / Stop.

Packet
High-level representation of network packet. Private field is *mbuf, public fields are mac / ip / data /etc: pointers to mbuf with offsets (zero copy).
Is extracted before any User defined function. Can be filled after user request by Packet functions. Can be checked by Rule functions.

Port
Network door, used in Receive, Send.

Rule
Set of checking rules, used in User defined functions.

Building Processing Graph



Packet modification functions

Packet functions

Parsing packet fields
Parse L2 or/and L3 or/and L4 levels

Initializing packet fields
Initialize L2 or/and L3 or/and L4 levels

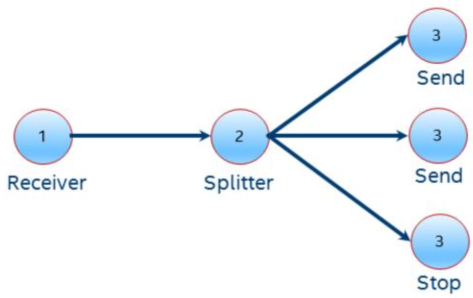
Encapsulate / Decapsulate

Rule functions

Create rule
Create checking rule from json / config

Checking packet fields by rule
Check L2 or/and L3 or/and L4 levels

Flow Graph Example - Forwarding



Let's build some functions!

Create test VMs

1.Create and provision two test VMs:

```
$ cd $GOPATH/src/github.com/intel-go/yanff/vagrant
$ vagrant up
```

2.Open two terminal windows

3.cd to vagrant directory below

4.run "vagrant ssh yanf-"***VM_number"*** to connect to pktgen VM and target VM, e.g.

```
$ vagrant ssh yanff-1      # YANFF test program host
yanff-1$ bindports         # if ports not bound yet

$ vagrant ssh yanff-0      # pktgen host
yanff-0$ bindports         # if ports not bound yet
```

Let's try (1 of 11)

Flow graph:

```
yanff-1$ cd $YANFF/examples/tutorial
yanff-1$ sudo ./step1
```

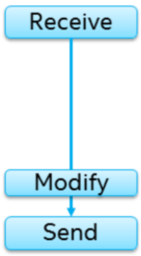
```
yanff-0$ cd $YANFF/examples/tutorial
yanff-0$ ./genscripts
yanff-0$ ./runpktgen.sh

Pktgen:/> start 0
......
Pktgen:/> quit
```

```
package main
import "github.com/intel-go/yanff/flow"

func main() {
     // Init YANFF system
     config := flow.Config{}
     checkFatal(flow.SystemInit(&config))

     initCommonState()

     checkFatal(flow.SystemStart())
}
```


Optimization Notice

Let's try (2 of 11)

Flow graph:




```
yanff-1$ sudo ./step2
```

```
yanff-0$ ./runpktgen.sh
Pktgen:/> load step2.pg
Pktgen:/> start 0
…
Pktgen:/> quit
```

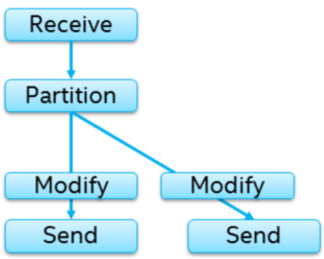
```go
package main

import "github.com/intel-go/yanff/flow"

func main() {
    config := flow.Config{}
    checkFatal(flow.SystemInit(&config))

    initCommonState()

    firstFlow, err := flow.SetReceiver(0)
    checkFatal(err)
    checkFatal(flow.SetHandler(firstFlow, modifyPacket[0], nil))
    checkFatal(flow.SetSender(firstFlow, 0))

    checkFatal(flow.SystemStart())
}
```


Optimization Notice

Example program receives packets from port 0 and sends them back to port 0. After “start 0” command pktgen should show that packets are not only sent on interface 0, but also the same number of packets is received back.

Let's try (3 of 11)

Flow graph:



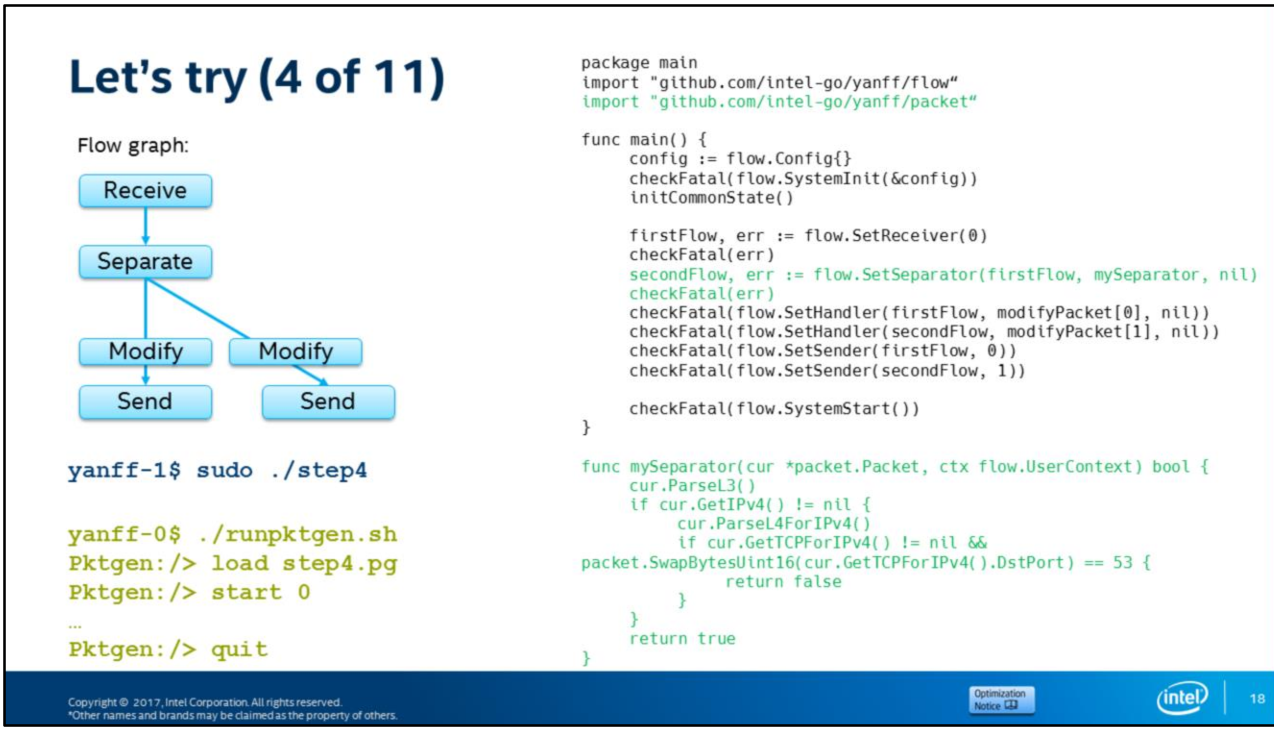

```
yanff-1$ sudo ./step3
```

```
yanff-0$ ./runpktgen.sh
Pktgen:/> load step3.pg
Pktgen:/> start 0
…
Pktgen:/> quit
```

```go
package main
import "github.com/intel-go/yanff/flow"

func main() {
	config := flow.Config{}
	checkFatal(flow.SystemInit(&config))

	initCommonState()

	firstFlow, err := flow.SetReceiver(0)
	checkFatal(err)
	secondFlow, err := flow.SetPartitioner(firstFlow, 300, 300)
	checkFatal(err)
	checkFatal(flow.SetHandler(firstFlow, modifyPacket[0], nil))
	checkFatal(flow.SetHandler(secondFlow, modifyPacket[1], nil))
	checkFatal(flow.SetSender(firstFlow, 0))
	checkFatal(flow.SetSender(secondFlow, 1))

	checkFatal(flow.SystemStart())
}
```


Optimization Notice

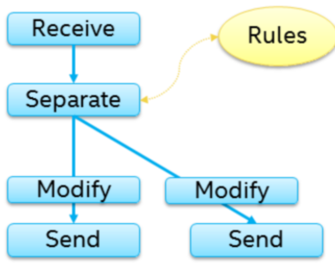
Example program receives packets from port 0 and sends them back half to port 0 and half to port 1. After “start 0” command pktgen should show that packets are sent on interface 0, and half of packets is received on interface 0, half on interface 1.



Example program receives packets from port 0 and sends back packets with port 53 to interface 1, all other packets to interface 0. Since step4.pg script generates packets with 11 different port numbers, 10/11 of packets should be received on port 0, and 1/11 should be received on port 1.

Let's try (5 of 11)

Flow graph:



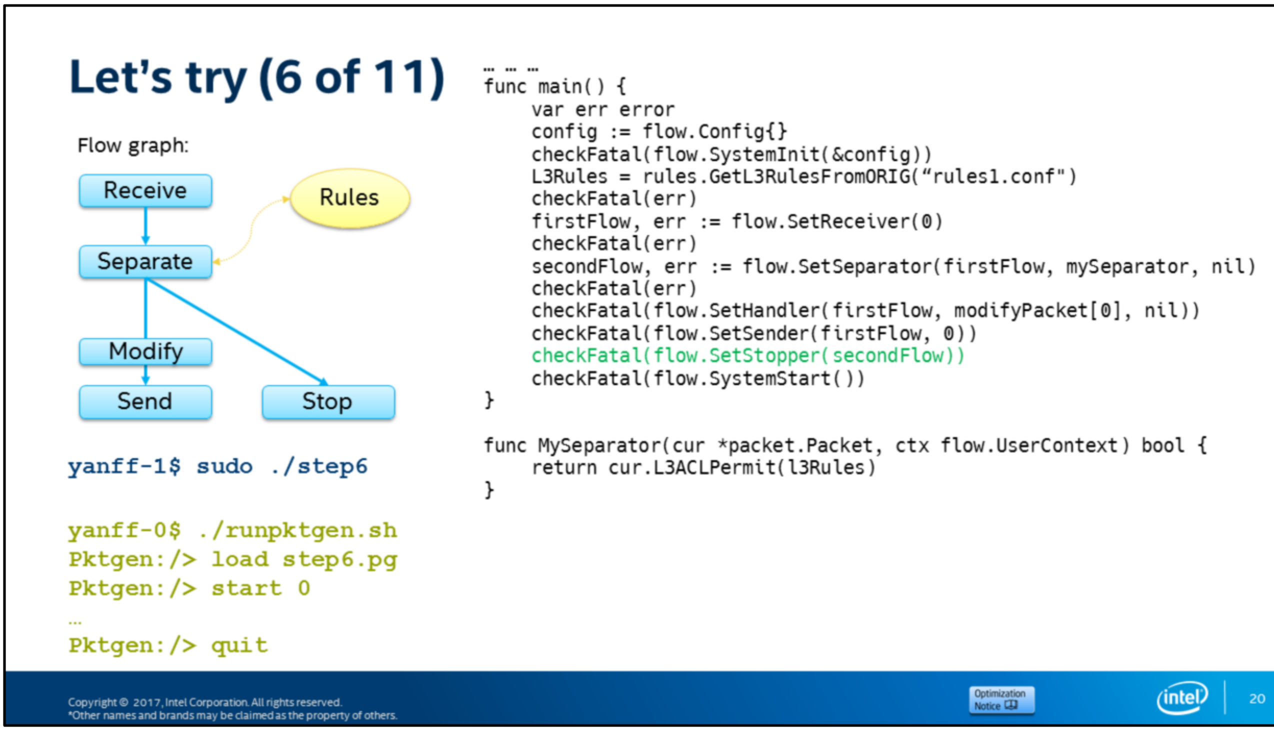

```
yanff-1$ sudo ./step5

yanff-0$ ./runpktgen.sh
Pktgen:/> load step5.pg
Pktgen:/> start 0
…
Pktgen:/> quit
```

```
… … …
import "github.com/intel-go/yanff/rules"
var L3Rules *rules.L3Rules

func main() {
    var err error
    config := flow.Config{}
    checkFatal(flow.SystemInit(&config))
    initCommonState()

    l3Rules, err = packet.GetL3ACLFromORIG("rules1.conf")
    checkFatal(err)

    firstFlow, err := flow.SetReceiver(0)
    checkFatal(err)
    secondFlow, err := flow.SetSeparator(firstFlow, mySeparator, nil)
    checkFatal(err)
    checkFatal(flow.SetHandler(firstFlow, modifyPacket[0], nil))
    checkFatal(flow.SetHandler(secondFlow, modifyPacket[1], nil))
    checkFatal(flow.SetSender(firstFlow, 0))
    checkFatal(flow.SetSender(secondFlow, 1))
    checkFatal(flow.SystemStart())
}

func MySeparator(cur *packet.Packet, ctx flow.UserContext) bool {
    return cur.L3ACLPermit(l3Rules)
}
```

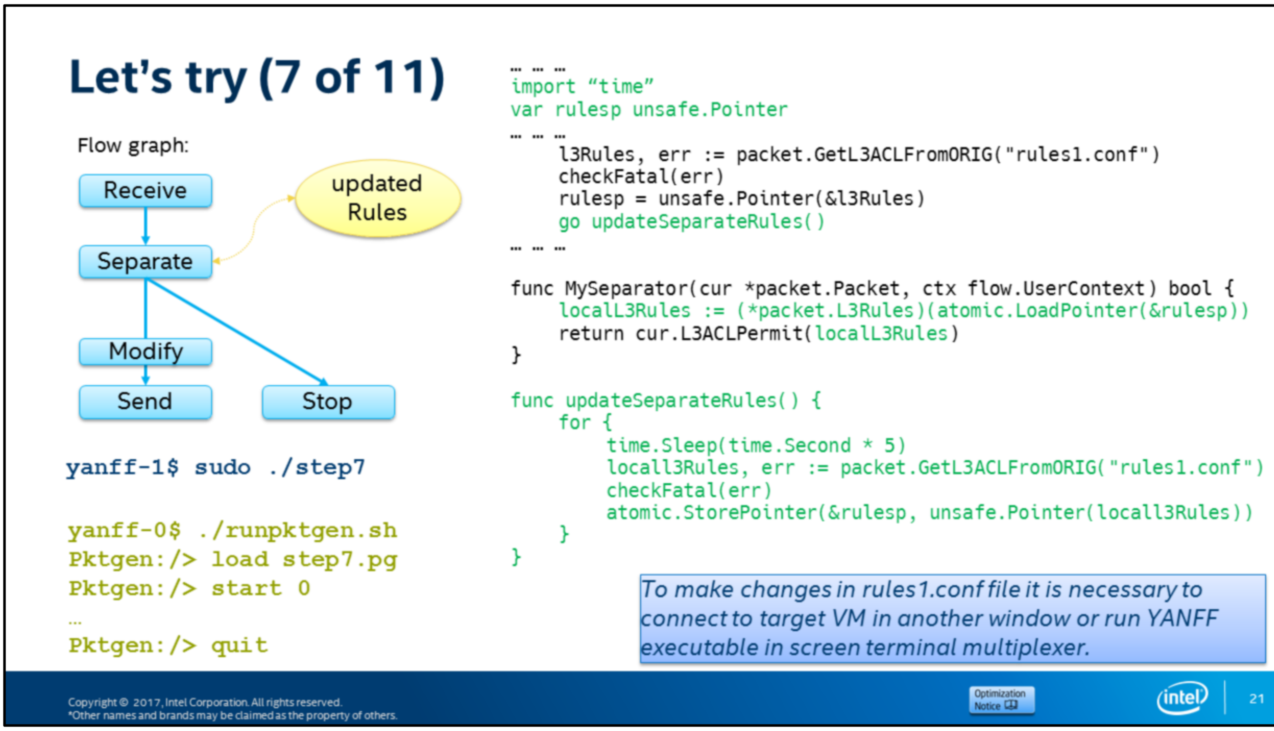

Optimization Notice

intel 19

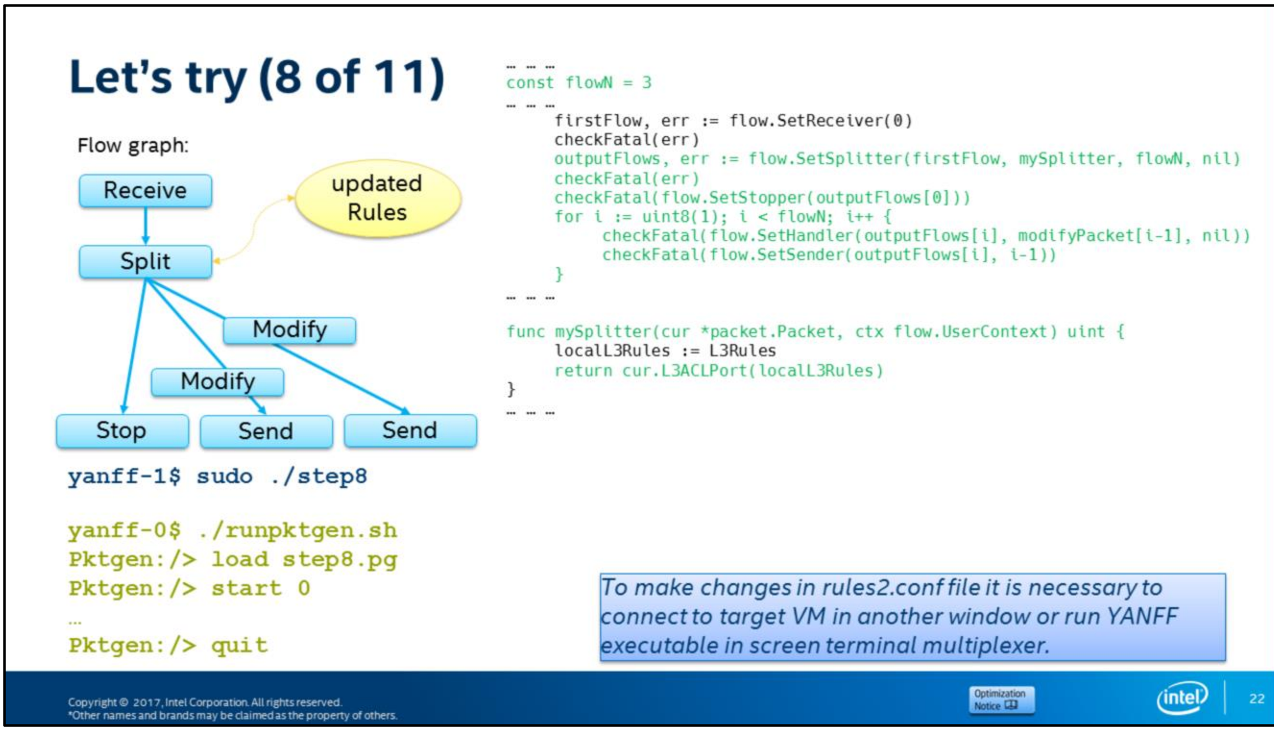
Example program receives packets from port 0 and sends back packets based on rules written in rules1.conf file. This rules file contents is written so that ¾ of packets should be received on port 0 and ¼ of packets should be received on port 1.



Example program receives packets from port 0 and sends back packets based on rules written in rules1.conf file. Only accepted packets are sent back to interface 0, rejected packets are dropped in this example. Rules file contents is written so that ¾ of packets should be received on port 0 and nothing should be received on interface 1.



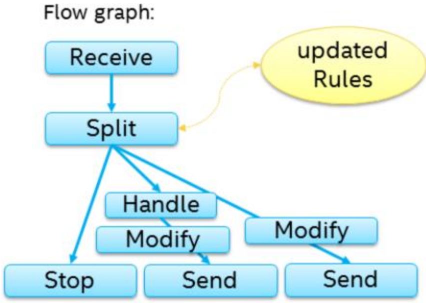
Example program receives packets from port 0 and sends back packets based on rules written in rules1.conf file. Only accepted packets are sent back to interface 0, rejected packets are dropped in this example. Rules file original contents is written so that ¾ of packets should be received on port 0 and nothing should be received on interface 1. But this example allows on the fly modification of rules file, so for example changing Reject rule to "111.2.0.2/31" should change number of received packets on interface 0 to ½.



Example program receives packets from port 0 and sends back packets based on rules written in rules2.conf file. This rules file specifies to which output port the program should send a packet, port 0 meaning to drop packet. Rules file original contents is written so that ½ of packets should be received on port 0 and ¼ should be received on port 1. But this example allows on the fly modification of rules file, so for example changing port 1 rule to “111.2.0.0/32” should change number of received packets on interface 0 to ¼ and received packets on interface 1 to ½.

Let's try (9 of 11)

Flow graph:




```
yanff-1$ sudo ./step9
```

```
yanff-0$ ./runpktgen.sh
Pktgen:/> load step9.pg
Pktgen:/> start 0
…
Pktgen:/> quit
```

```
… … …
import "github.com/intel-go/yanff/common"
… … …
     firstFlow, err := flow.SetReceiver(0)
     checkFatal(err)
     outputFlows, err := flow.SetSplitter(firstFlow, mySplitter, flowN, nil)
     checkFatal(err)
     checkFatal(flow.SetStopper(outputFlows[0]))
     checkFatal(flow.SetHandler(outputFlows[1], myHandler, nil))
     for i := uint8(1); i < flowN; i++ {
          checkFatal(flow.SetHandler(outputFlows[i], modifyPacket[i-1], nil))
          checkFatal(flow.SetSender(outputFlows[i], i-1))
     }
… … …
func myHandler(cur *packet.Packet, ctx flow.UserContext) {
     cur.EncapsulateHead(common.EtherLen, common.IPv4MinLen)
     cur.ParseL3()
     cur.GetIPv4NoCheck().SrcAddr = packet.BytesToIPv4(111, 22, 3, 0)
     cur.GetIPv4NoCheck().DstAddr = packet.BytesToIPv4(3, 22, 111, 0)
     cur.GetIPv4NoCheck().VersionIhl = 0x45
     cur.GetIPv4NoCheck().NextProtoID = 0x04
}
```

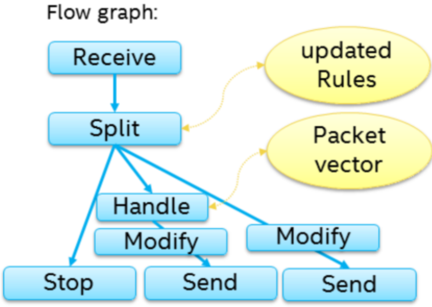
To make changes in rules2.conf file it is necessary to connect to target VM in another window or run YANFF executable in screen terminal multiplexer.

Optimization Notice

Example program receives packets from port 0 and sends back packets based on rules written in rules2.conf file. Difference with previous example is that port 1 flow also encapsulates a packet into IPv4 packet. Therefore received traffic on port 0 should be slightly higher than ½ of packets. If for example port 0 (drop) rule is changed to "111.2.1.2/32" (no packet will match) ½ of packets should be received on port 0 and ½ of packets should be received on port 1, but since on port 0 they are encapsulated, received traffic should be slightly higher on port 0.

Let's try (10 of 11)

Flow graph:



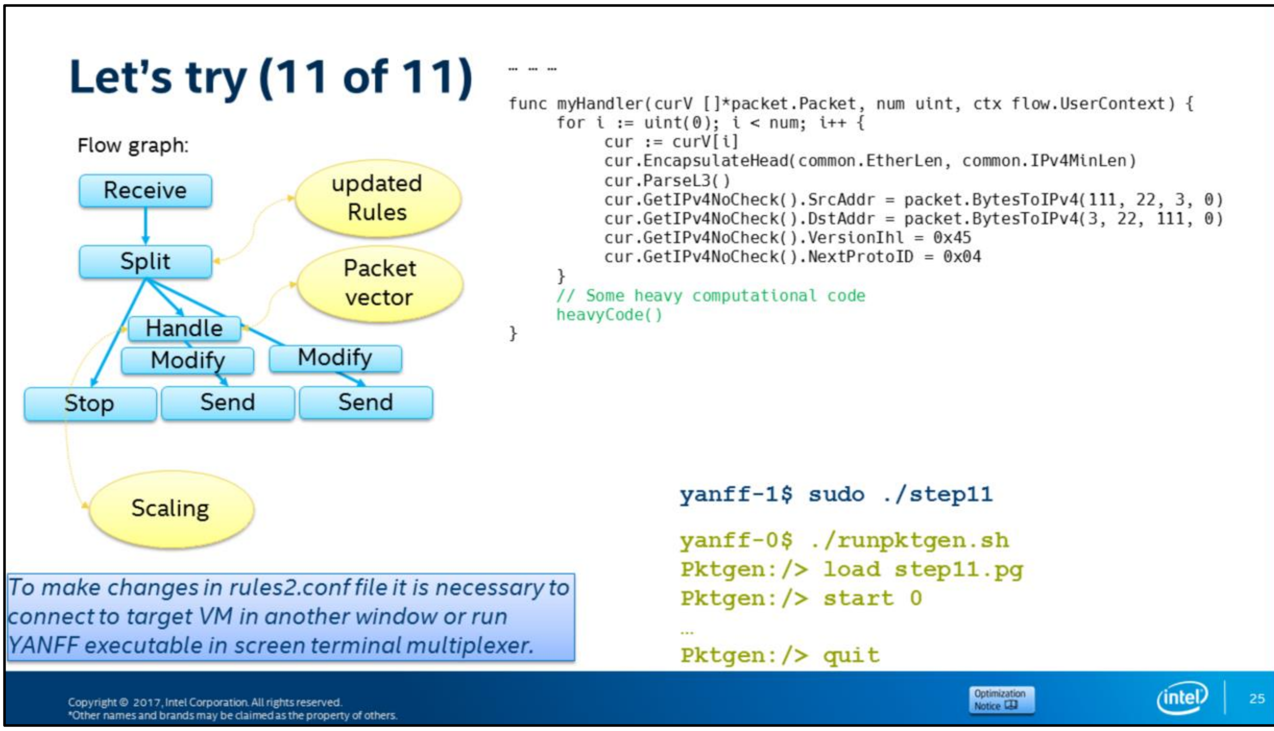

```
… … …
func myHandler(curV []*packet.Packet, num uint, ctx flow.UserContext) {
    for i := uint(0); i < num; i++ {
        cur := curV[i]
        cur.EncapsulateHead(common.EtherLen, common.IPv4MinLen)
        cur.ParseL3()
        cur.GetIPv4NoCheck().SrcAddr = packet.BytesToIPv4(111, 22, 3, 0)
        cur.GetIPv4NoCheck().DstAddr = packet.BytesToIPv4(3, 22, 111, 0)
        cur.GetIPv4NoCheck().VersionIhl = 0x45
        cur.GetIPv4NoCheck().NextProtoID = 0x04
    }
}
```

```
yanff-1$ sudo ./step10

yanff-0$ ./runpktgen.sh
Pktgen:/> load step10.pg
Pktgen:/> start 0
…
Pktgen:/> quit
```

To make changes in rules2.conf file it is necessary to connect to target VM in another window or run YANFF executable in screen terminal multiplexer.

Optimization Notice

Example program receives packets from port 0 and sends back packets based on rules written in rules2.conf file. Difference with previous example is that handler function accepts a packet vector, everything else is the same.



Example program receives packets from port 0 and sends back packets based on rules written in rules2.conf file. Difference with previous example is that vector handler function executes some heavy code to demonstrate automatic scaling. On systems with many CPU cores the example program should try to use more than in step 10 example.

Finally: NAT

```
yanff-1$ ./genscripts -pktgen direct
yanff-1$ sudo ../nat/main/nat -config nat.json
```

```
yanff-0$ ./runpktgen.sh
Pktgen:/> load nat.pg
Pktgen:/> start 0
Pktgen:/> start 1
…
Pktgen:/> quit
```


Example demonstrates using NAT example with pktgen. Sample configuration for NAT configures port 0 as private network port and port 1 as public network port. Addresses are translated from 192.168.1.1/24 subnet to 10.1.1.1/24 subnet with NAT IP on public interface 10.1.1.1. Sample pktgen script implements 11 client addresses that try to access one server on public network and server sending packets back to the clients. When both “start 0” and “start 1” commands are executed pktgen should show the same traffic received on interface 0 and interface 1 which means that address translation works in both directions.

Q & A ?

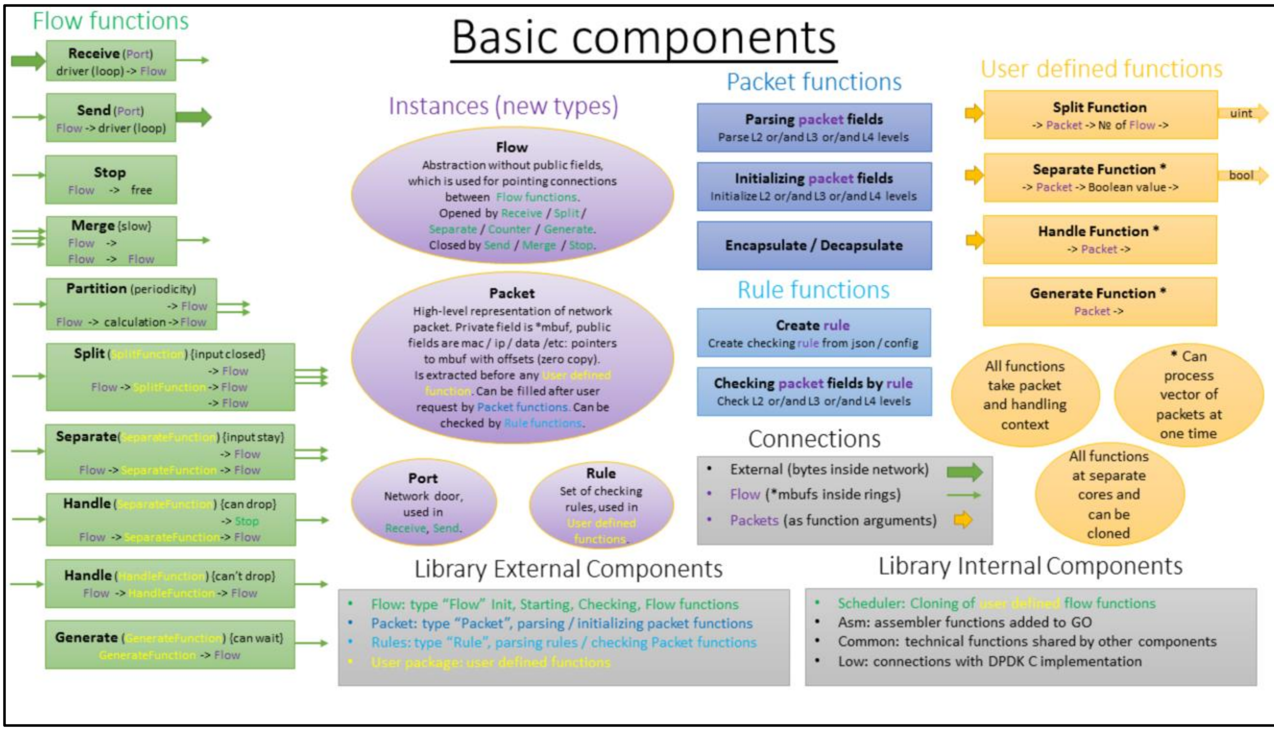
Optimization Notice

Optimization Notice

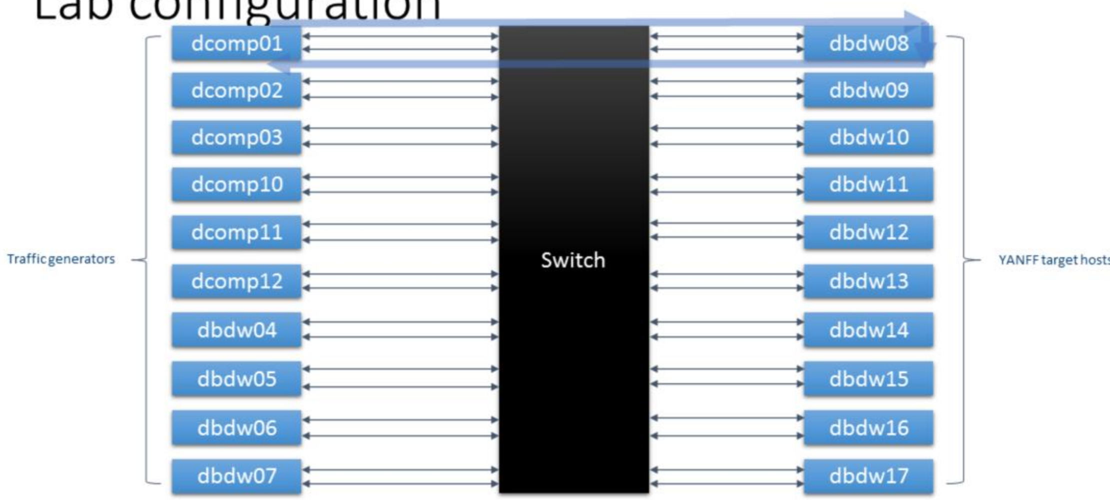
Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2®, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804


Basic components
Flow functions
Receive (Port)
driver (loop) -> Flow
Send (Port)
Flow -> driver (loop)
Stop
Flow -> free
Merge {slow}
Flow ->
Flow -> Flow
Partition (periodicity)
-> Flow
Flow -> calculation -> Flow
Split (SplitFunction) {input closed}
-> Flow
Flow -> SplitFunction -> Flow
-> Flow
Separate(SeparateFunction) {input stay}
-> Flow
Flow -> SeparateFunction -> Flow
Handle (SeparateFunction) {can drop}
-> Stop
Flow -> SeparateFunction-> Flow
Handle (HandleFunction) {can't drop}
Flow -> HandleFunction -> Flow
Generate (GenerateFunction) {can wait}
GenerateFunction -> Flow
Instances (new types)
Flow
Abstraction without public fields, which is used for pointing connections between Flow functions. Opened by Receive / Split / Separate / Counter / Generate. Closed by Send / Merge / Stop.
Packet
High-level representation of network packet. Private field is *mbuf, public fields are mac / ip / data /etc: pointers to mbuf with offsets (zero copy). Is extracted before any User defined function. Can be filled after user request by Packet functions. Can be checked by Rule functions.
Port
Network door, used in Receive, Send.
Rule
Set of checking rules, used in User defined functions.
Packet functions
Parsing packet fields
Parse L2 or/and L3 or/and L4 levels
Initializing packet fields
Initialize L2 or/and L3 or/and L4 levels
Encapsulate / Decapsulate
Rule functions
Create rule
Create checking rule from json / config
Checking packet fields by rule
Check L2 or/and L3 or/and L4 levels
Connections
External (bytes inside network)
Flow (*mbufs inside rings)
Packets (as function arguments)
User defined functions
Split Function
-> Packet -> № of Flow ->
uint
Separate Function *
-> Packet -> Boolean value ->
bool
Handle Function *
-> Packet ->
Generate Function *
Packet ->
All functions take packet and handling context
* Can process vector of packets at one time
All functions at separate cores and can be cloned
Library External Components
Flow: type "Flow" Init, Starting, Checking, Flow functions
Packet: type "Packet", parsing / initializing packet functions
Rules: type "Rule", parsing rules / checking Packet functions
User package: user defined functions
Library Internal Components
Scheduler: Cloning of user defined flow functions
Asm: assembler functions added to GO
Common: technical functions shared by other components
Low: connections with DPDK C implementation

Lab configuration



Jump host 207.108.8.161, Login: gashiman, Password: YanffLab

Finally (2 of 2): ipsec

- Showing ipsec example