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

Intel Golang Team

Gregory Shimansky

Legal Information

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

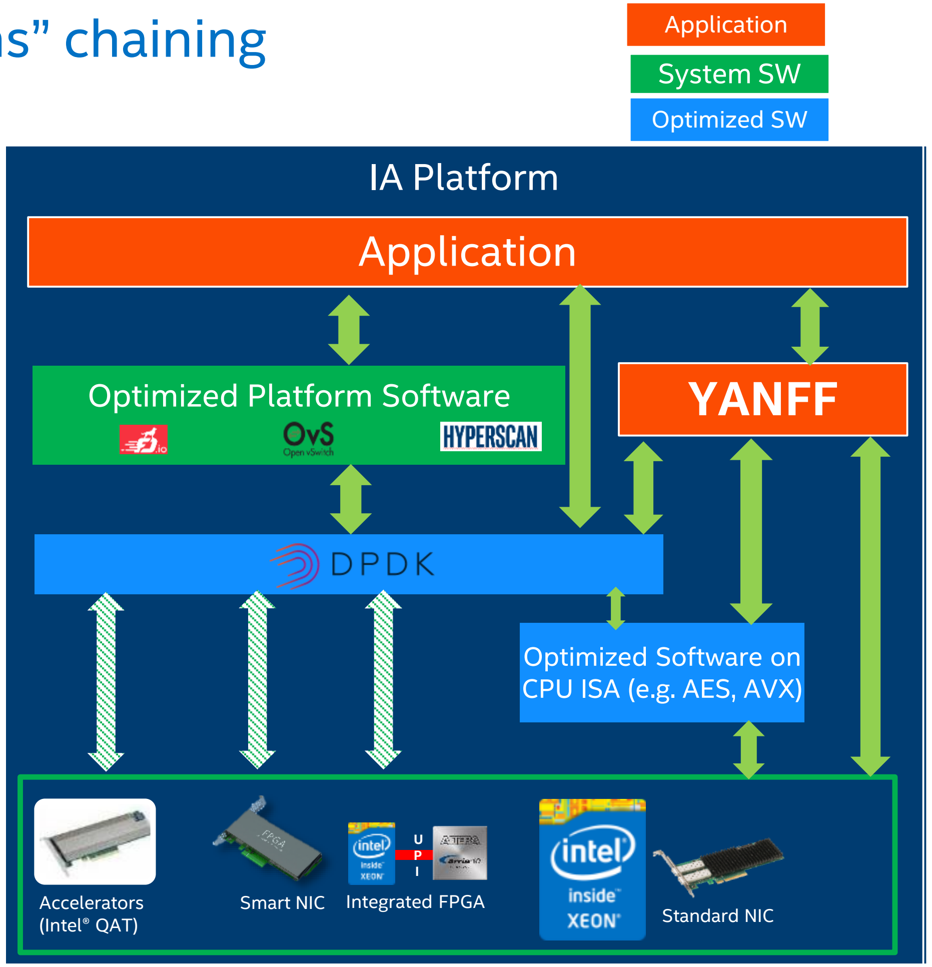
- Developers need framework to shorten development cycle of VNFs
  - Currently VNFs are monolithic - “virtual appliances” instead of network functions
  - Significant part of VNF is about plumbing. Plumbing VNFs to CommSPs network is an art. Should be abstracted from VNFs
- Lack of stable and unified 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

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

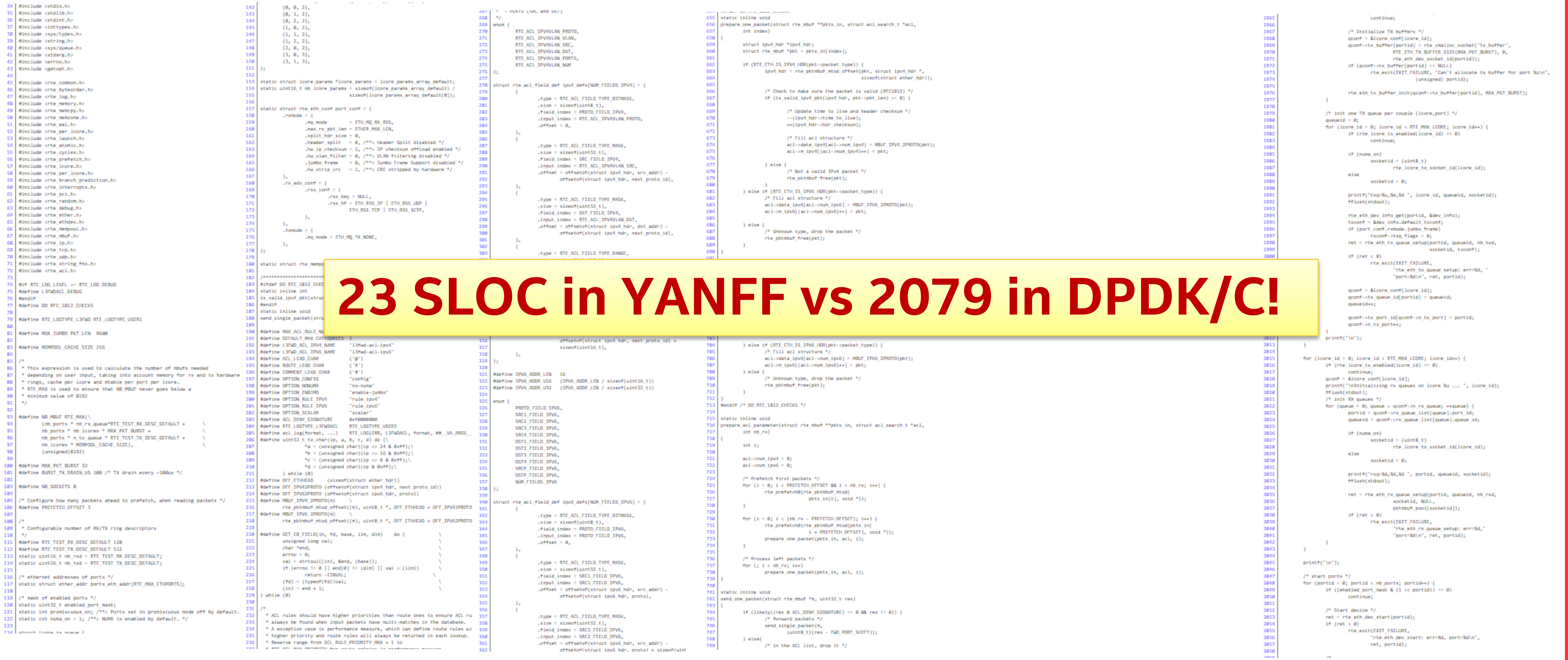
```
var L3Rules *rules.L3Rules

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)
}
```

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

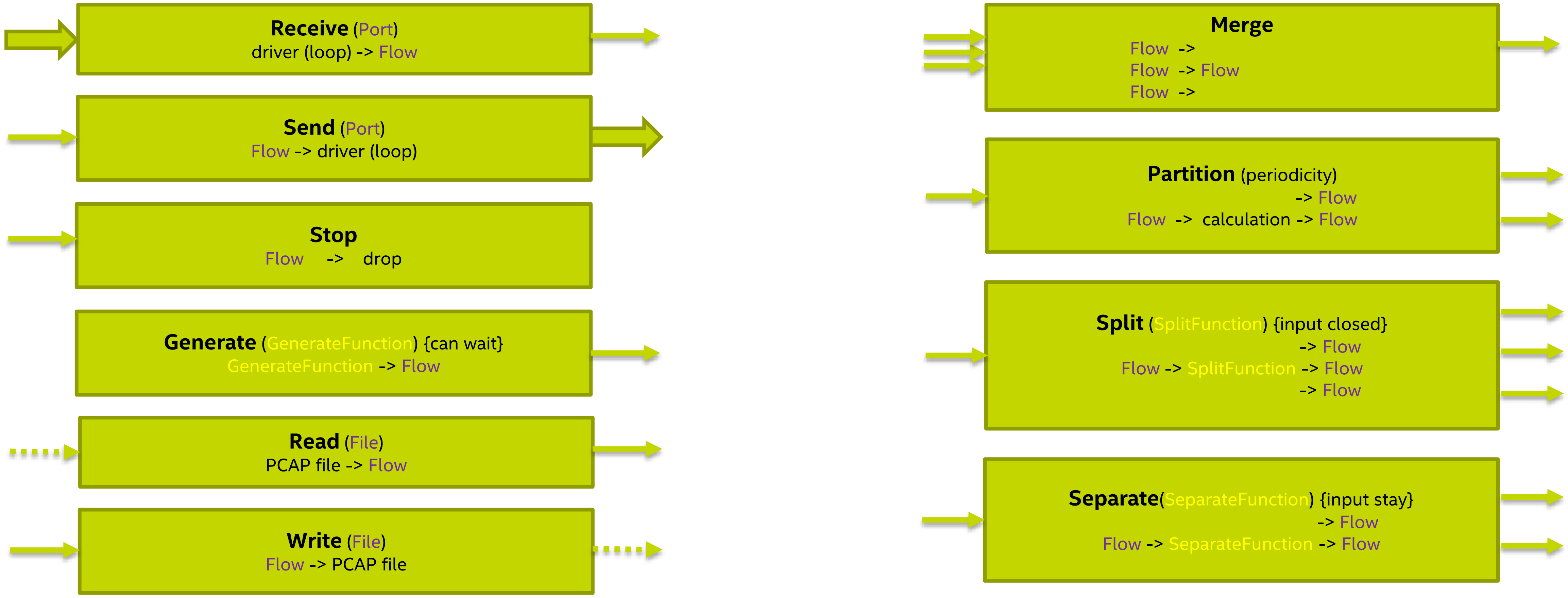
Flow
Abstraction without public fields, which is used for pointing connections between Flow functions.
Opened by Receive / Split / Separate / Counter / Generate.
Closed by Send / Merge / Stop.

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.

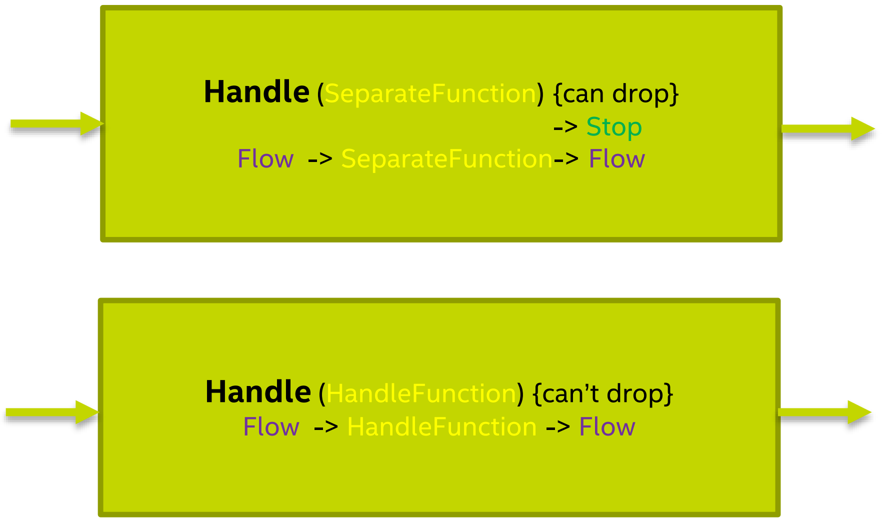
Port
Network door, used in Receive, Send.

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

Building Processing Graph



Packet modification functions


Handle (SeparateFunction) {can drop}
-> Stop
Flow -> SeparateFunction-> Flow
Handle (HandleFunction) {can't drop}
Flow -> HandleFunction -> Flow

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

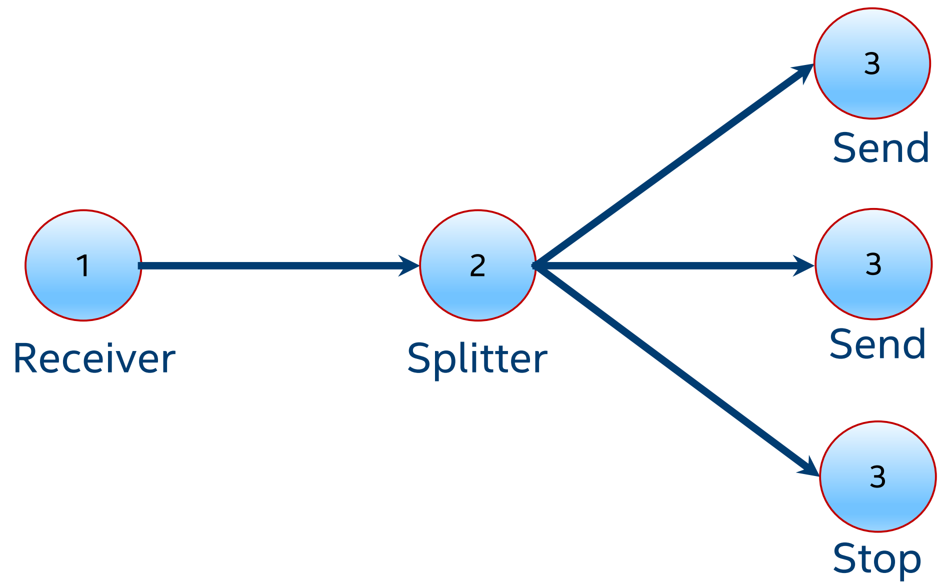
Encapsulate / Decapsulate

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 nff-go/vagrant
   $ vagrant up
   ```
2. Open two terminal windows
3. cd to vagrant directory below
4. run “vagrant ssh nff-go-”***VM_number”*** to connect to pktgen VM and target VM, e.g.

```
$ vagrant ssh nff-go-1     # NFF-Go test program host
yanff-1$ bindports         # if ports not bound yet

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

Let's try (01 of 11)

Flow graph:

```
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())
}
```

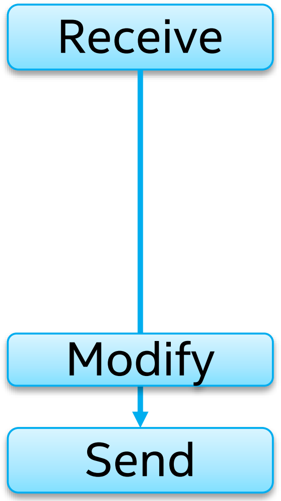
```
nff-go-1$ cd $YANFF/examples/tutorial
nff-go-1$ sudo ./step01
```

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

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

Optimization Notice

Let’s try (02 of 11)

Flow graph:



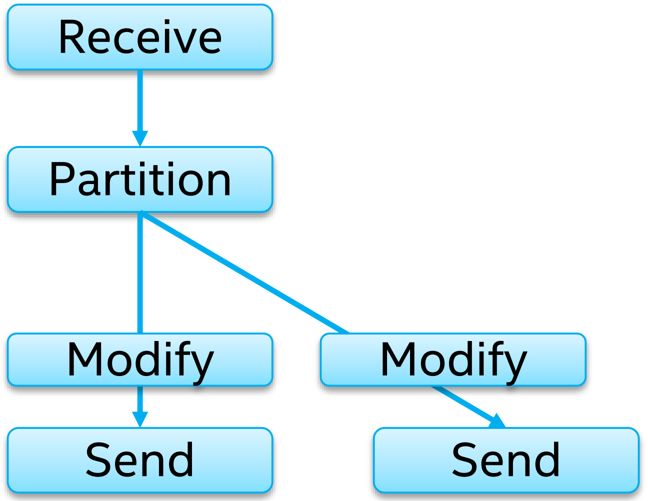
```
nff-go-1$ sudo ./step02
```

```
nff-go-0$ ./runpktgen.sh
Pktgen:/> load step02.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())
}
```

Let’s try (03 of 11)

Flow graph:



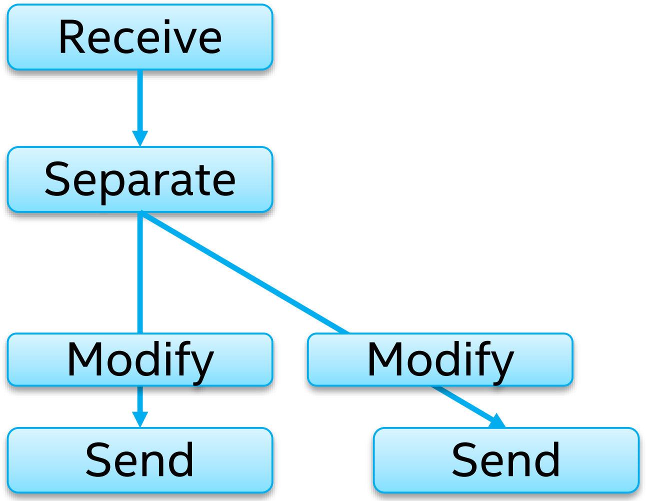
```
nff-go-1$ sudo ./step03
```

```
nff-go-0$ ./runpktgen.sh
Pktgen:/> load step03.pg
Pktgen:/> start 0
…
Pktgen:/> quit
```

```
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())
}
```

Let's try (04 of 11)

Flow graph:



```
nff-go-1$ sudo ./step04
```

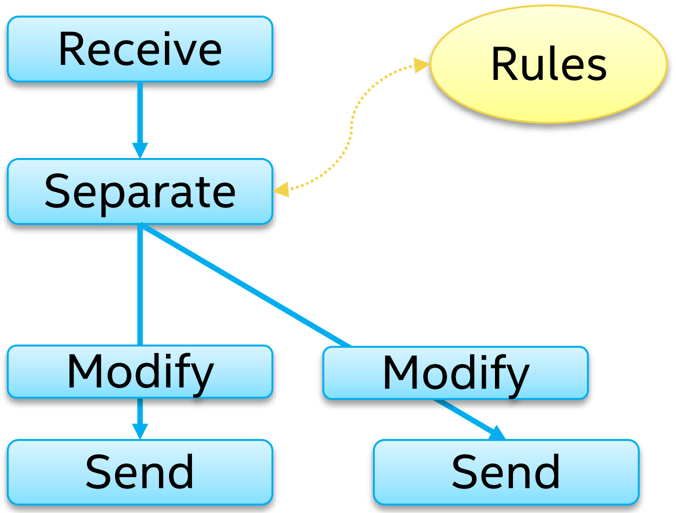
```
nff-go-0$ ./runpktgen.sh
Pktgen:/> load step04.pg
Pktgen:/> start 0
…
Pktgen:/> quit
```

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

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

	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 {
	cur.ParseL3()
	if cur.GetIPv4() != nil {
		cur.ParseL4ForIPv4()
		if cur.GetTCPForIPv4() != nil &&
packet.SwapBytesUint16(cur.GetTCPForIPv4().DstPort) == 53 {
			return false
		}
	}
	return true
}
```

Optimization Notice

Let's try (05 of 11)

Flow graph:



```
nff-go-1$ sudo ./step05
```

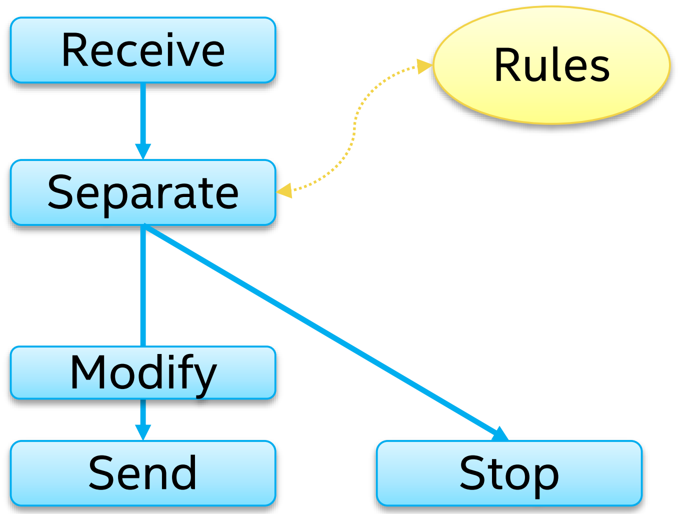
```
nff-go-0$ ./runpktgen.sh
Pktgen:/> load step05.pg
Pktgen:/> start 0
…
Pktgen:/> quit
```

```
… … …
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

Let’s try (06 of 11)

Flow graph:



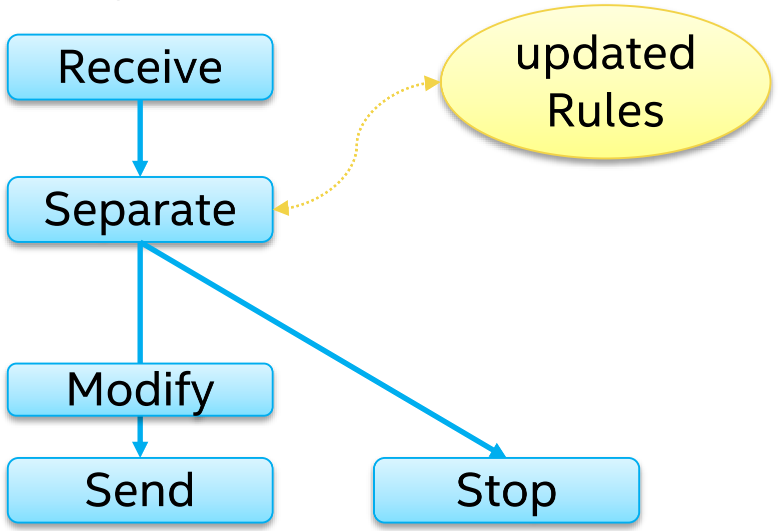
```
nff-go-1$ sudo ./step06
```

```
nff-go-0$ ./runpktgen.sh
Pktgen:/> load step06.pg
Pktgen:/> start 0
…
Pktgen:/> quit
```

```
… … …
func main() {
        var err error
        config := flow.Config{}
        checkFatal(flow.SystemInit(&config))
        L3Rules = rules.GetL3RulesFromORIG(“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.SetSender(firstFlow, 0))
        checkFatal(flow.SetStopper(secondFlow))
        checkFatal(flow.SystemStart())
}

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

Let’s try (07 of 11)

Flow graph:



```
nff-go-1$ sudo ./step07
```

```
nff-go-0$ ./runpktgen.sh
Pktgen:/> load step07.pg
Pktgen:/> start 0
…
Pktgen:/> quit
```

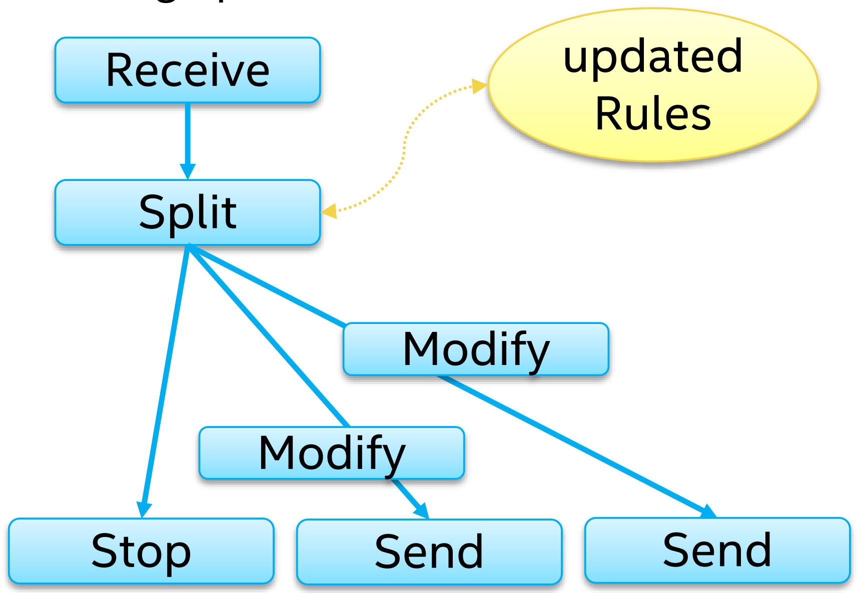
```
… … …
import “time”
var rulesp unsafe.Pointer
… … …
    l3Rules, err := packet.GetL3ACLFromORIG("rules1.conf")
    checkFatal(err)
    rulesp = unsafe.Pointer(&l3Rules)
    go updateSeparateRules()
… … …

func MySeparator(cur *packet.Packet, ctx flow.UserContext) bool {
    localL3Rules := (*packet.L3Rules)(atomic.LoadPointer(&rulesp))
    return cur.L3ACLPermit(localL3Rules)
}

func updateSeparateRules() {
    for {
        time.Sleep(time.Second * 5)
        locall3Rules, err := packet.GetL3ACLFromORIG("rules1.conf")
        checkFatal(err)
        atomic.StorePointer(&rulesp, unsafe.Pointer(locall3Rules))
    }
}
```

To make changes in rules1.conf file it is necessary to connect to target VM in another window or run YANFF executable in screen terminal multiplexer.

Optimization Notice

Let’s try (08 of 11)

Flow graph:



```
… … …
const flowN = 3
… … …
        firstFlow, err := flow.SetReceiver(0)
        checkFatal(err)
        outputFlows, err := flow.SetSplitter(firstFlow, mySplitter, flowN, nil)
        checkFatal(err)
        checkFatal(flow.SetStopper(outputFlows[0]))
        for i := uint8(1); i < flowN; i++ {
                checkFatal(flow.SetHandler(outputFlows[i], modifyPacket[i-1], nil))
                checkFatal(flow.SetSender(outputFlows[i], i-1))
        }
… … …

func mySplitter(cur *packet.Packet, ctx flow.UserContext) uint {
        localL3Rules := L3Rules
        return cur.L3ACLPort(localL3Rules)
}
… … …
```

```
nff-go-1$ sudo ./step08
```

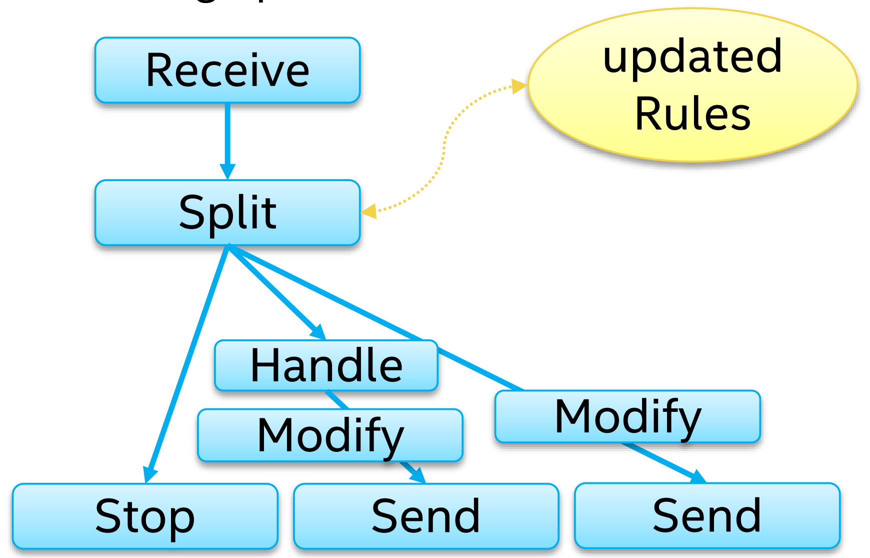
```
nff-go-0$ ./runpktgen.sh
Pktgen:/> load step08.pg
Pktgen:/> start 0
…
Pktgen:/> quit
```

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

Let’s try (09 of 11)

Flow graph:



```
… … …
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
}
```

```
nff-go-1$ sudo ./step09
```

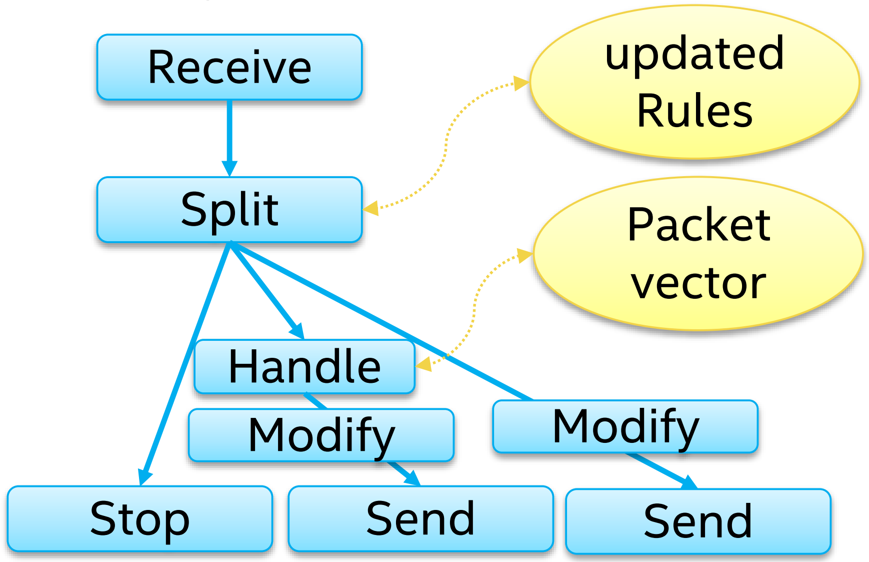
```
nff-go-0$ ./runpktgen.sh
Pktgen:/> load step09.pg
Pktgen:/> start 0
…
Pktgen:/> quit
```

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

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
	}
}
```

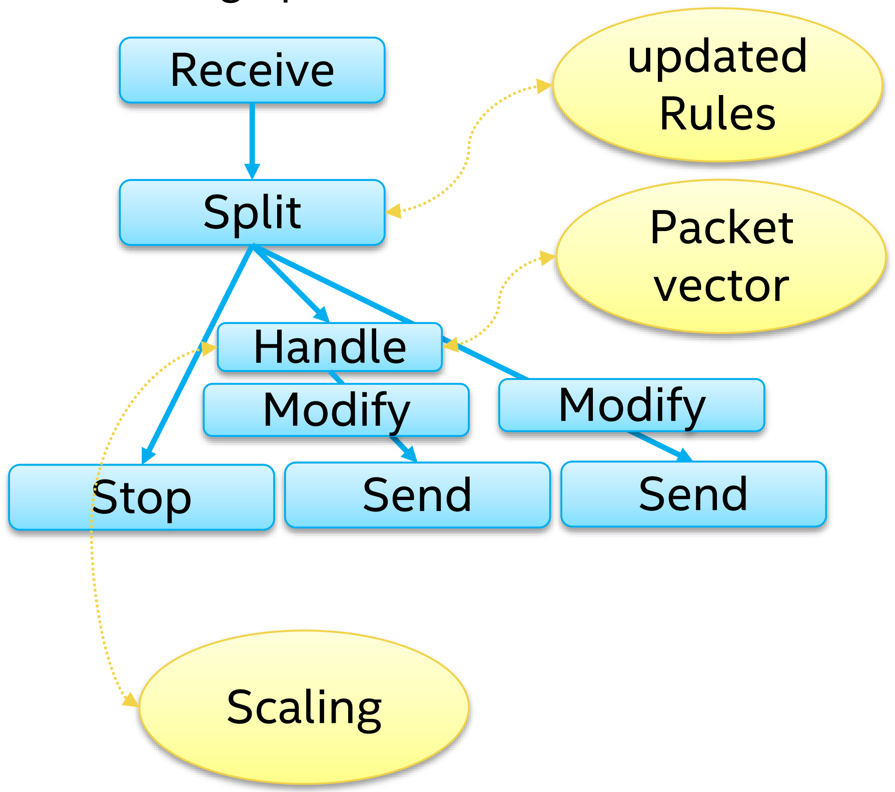
```
nff-go-1$ sudo ./step10
```

```
nff-go-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.

Let's try (11 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
	}
	// Some heavy computational code
	heavyCode()
}
```

```
nff-go-1$ sudo ./step11
```

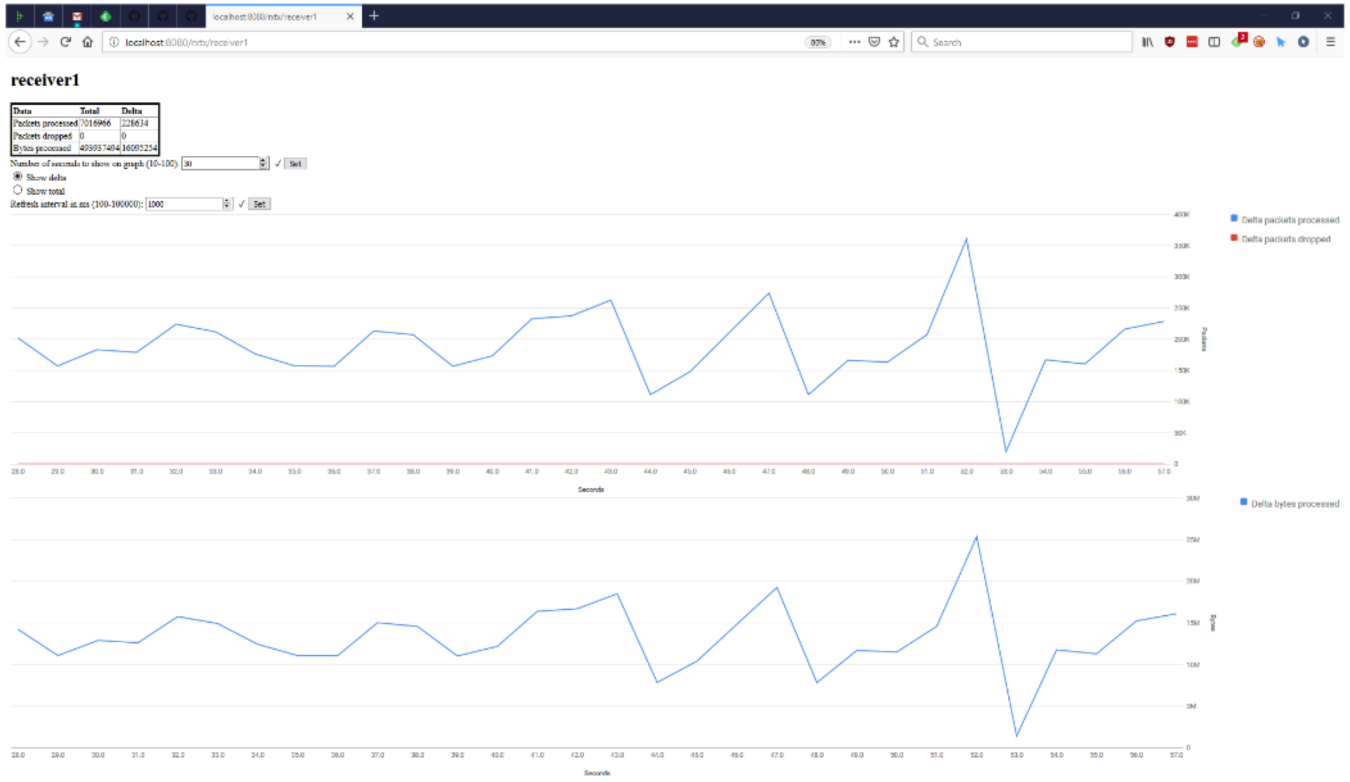
```
nff-go-0$ ./runpktgen.sh
Pktgen:/> load step11.pg
Pktgen:/> start 0
…
Pktgen:/> quit
```

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.

Alternative network packet IO

- KNI interfaces (examples/kni.go)
- Linux raw sockets (examples/OSforwarding.go)
- PCAP files (examples/clonablePcapDumper.go)
- Linux XDP (coming soon)

Statistic counters



localhost:8080/nxtx/receiver1

receiver1

Data	Total	Delta
Packets processed	7016966	228634
Packets dropped	0	0
Bytes processed	493937494	16095254

Number of seconds to show on graph (10-100): 30 ✓ Set

● Show delta

○ Show total

Refresh interval in ms (100-100000): 1000 ✓ Set

```
… … …

        // Set up address for stats web server
        statsServerAddres = &net.TCPAddr{
                Port: 8080,
        }

        config := flow.Config{
                StatsHTTPAddress: statsServerAddres,
        }

… … …
```

Optimization Notice

Finally: NAT

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

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

Q & A ?

Optimization Notice

Optimization Notice

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

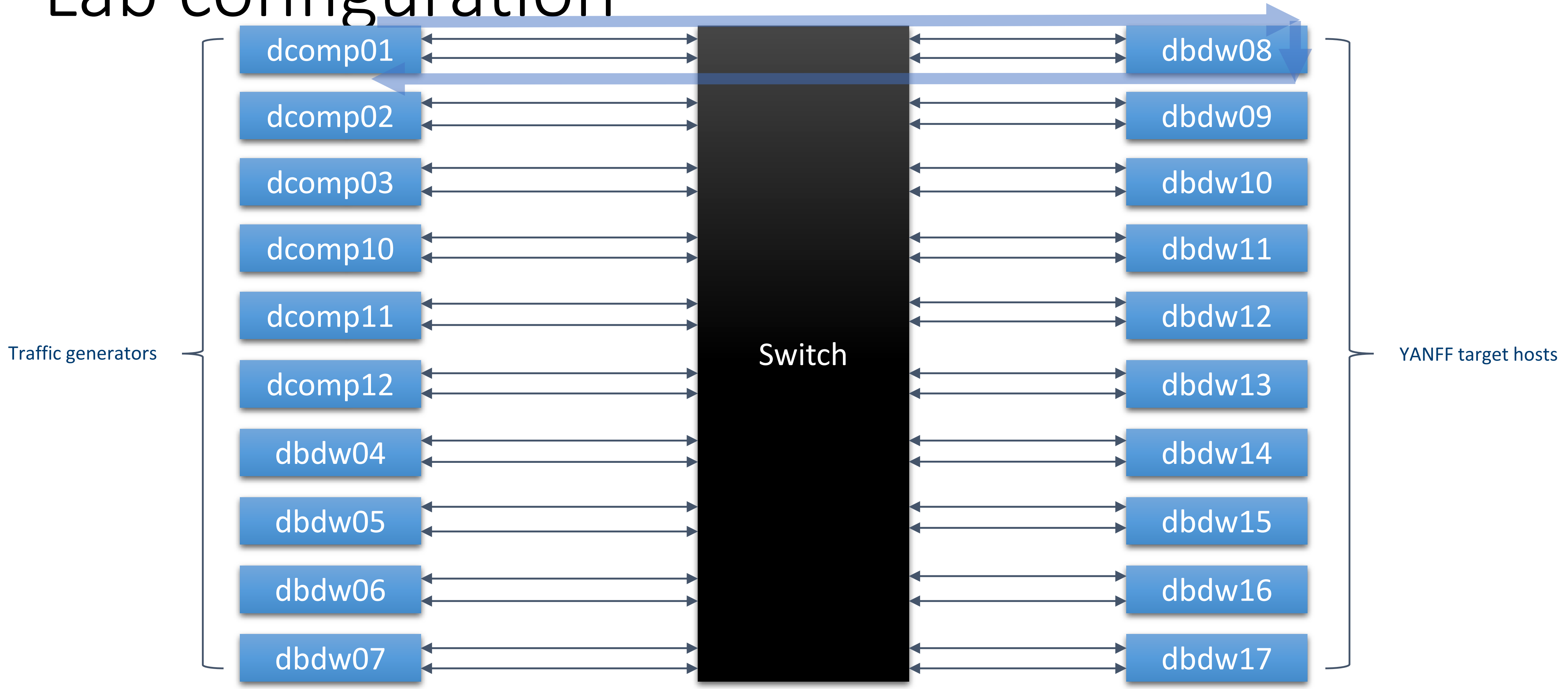
Library External Components

- 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: , Login: gashiman, Password:

Finally (2 of 2): ipsec

- Showing ipsec example