Packet Capturing with the JVM and Clojure
Yes, we can!

Ruediger Gad

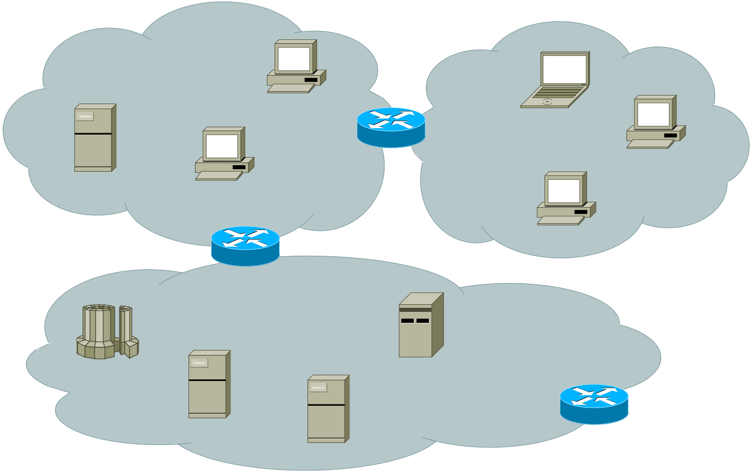
Terma GmbH, Space, Darmstadt, Germany

:clojureD
2017-02-25

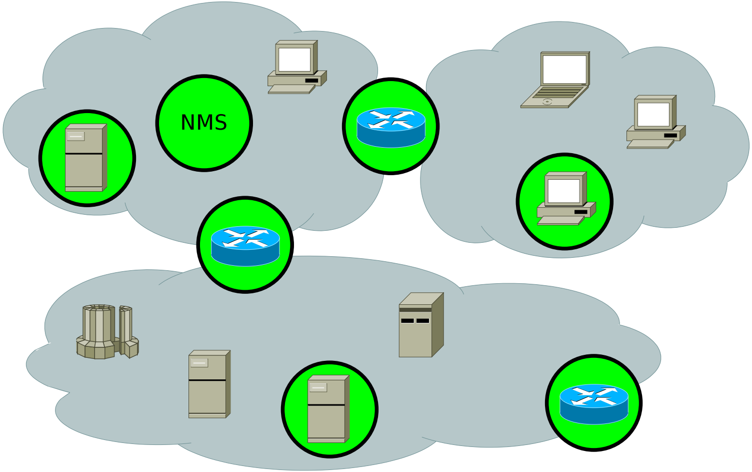
Outline

- Brief Introduction
- Packet Capturing & the JVM
- Get up to speed.
- Domain Specific Language (DSL) for Data Transformation
- Adding Dynamic Capabilities
- Dynamic Self-adaptive Adjustments

Computer Networks



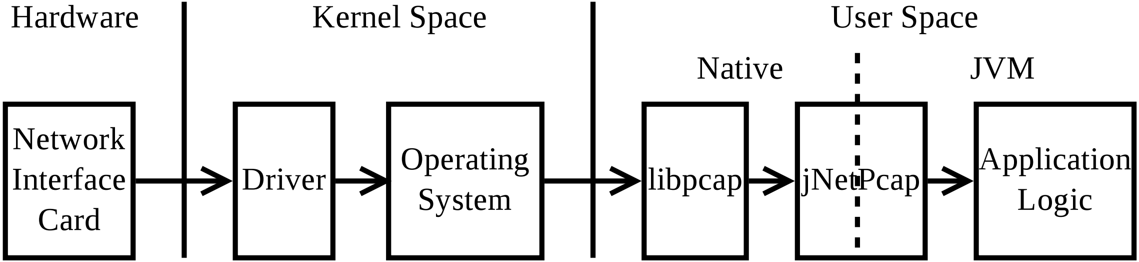
Network Monitoring



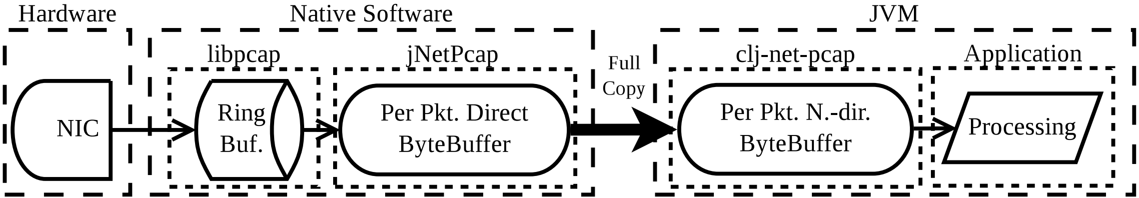
Network Monitoring Use Case Overview

- Requirements
  - Distribution, Flexibility, Data Analysis, ...
- JVM-based
  - Re-use Existing Libraries
    Communication Middleware, Data Processing, ...
- Clojure
  - Powerful, Dynamic, ...
- Packet Capturing as "Worst Case Scenario"
  - Data Throughput
  - Data Volume

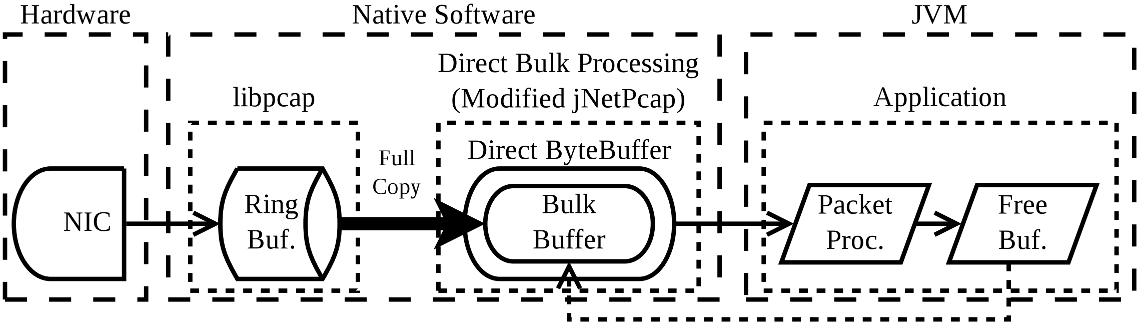
Packet Capturing (Pcap) & the JVM



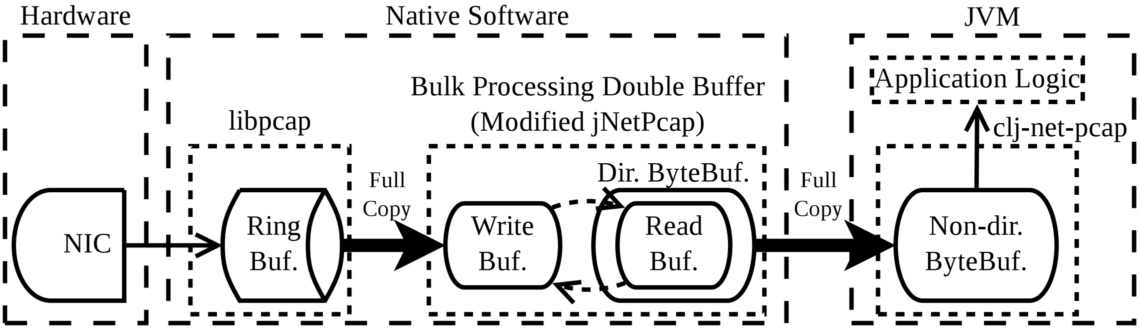
Pcap & the JVM, Per Packet Forwarding



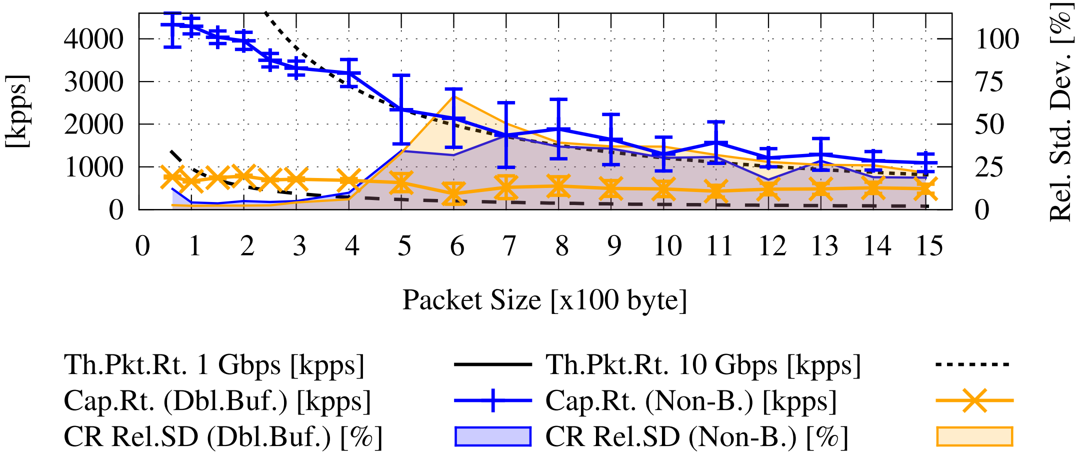
Pcap & the JVM, Packet Bulk Forwarding



Pcap & the JVM, Improved Packet Bulk Forwarding



Raw Pcap Performance Comparison



Making Sense of Raw Packet Data

- Raw Packet Data (Byte Arrays) to Java Types
- “Address Fields” (Offsets)
- Name Data
- Transform Data
  - Integer Values (4, 8, 16, 32 bit)
  - Timestamps
  - Addresses (IP, MAC)
- Output Data Type

Data Extraction DSL

Listing 1: Extraction DSL Expression Example

```
{:type :java-map
 :rules [[ts (timestamp 0)]
        [len (int32 12)]
        [ipDst (ipv4-address ipv4-dst)]
        [udpDst (int16 udp-dst)]]}
```

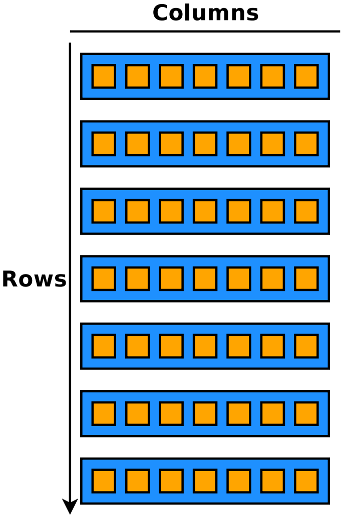
Listing 2: Extraction Function based on DSL

```
(fn [ba]
  (doto (java.util.HashMap.)
    (.put "ts" (timestamp ba 0))
    (.put "len" (int32 ba 12))
    (.put "ipDst" (ipv4-address ba 46))
    (.put "udpDst" (int16 ba 52))))
```

Data Extraction Throughput Comparison

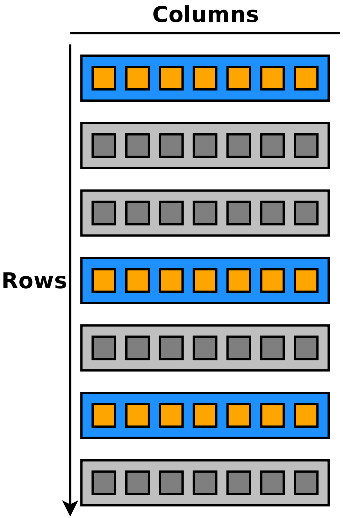
Method	Throughput $[\bar{x}]$	$[sd(x)]$
jNetPcap	265.7 kpps	10.4 kpps
DSL	612.2 kpps	8.8 kpps

What if throughput demands increase further?



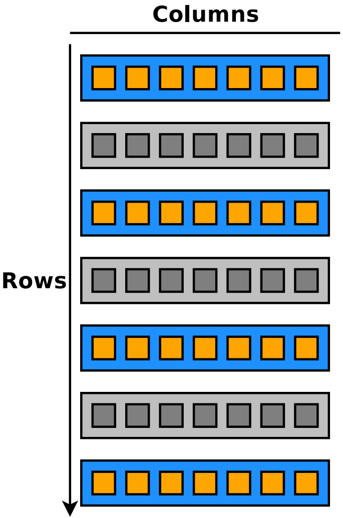
What if throughput demands increase further?

- Do nothing?
  → Random Drops of “Rows”



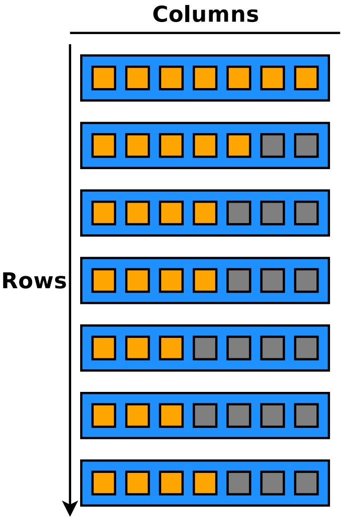
What if throughput demands increase further?

- Do nothing?
  → Random Drops of “Rows”
- Apply sampling?
  → More “Controlled” Drops of Rows



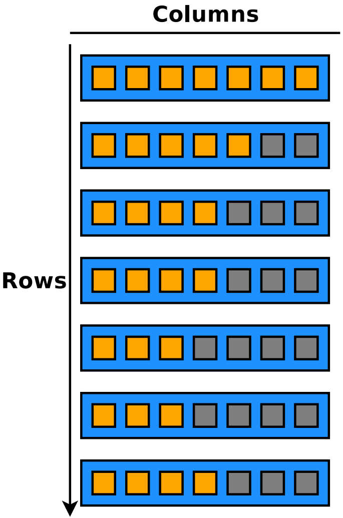
What if throughput demands increase further?

- Do nothing?
  → Random Drops of "Rows"
- Apply sampling?
  → More "Controlled" Drops of Rows
- Reduce extraction operations / extracted fields?
  → "Drop columns" in favor of rows.



What if throughput demands increase further?

- Do nothing?
  → Random Drops of "Rows"
- Apply sampling?
  → More "Controlled" Drops of Rows
- Reduce extraction operations / extracted fields?
  → "Drop columns" in favor of rows.
- → Adjust DSL expression rules.



Throughput for Varying DSL Expression Complexity

Method	Capture Rate [$\bar{x}$]	[$sd(x)$]
DSL 1	612.2 kpps	8.8 kpps
DSL 2	726.4 kpps	9.1 kpps
DSL 3	1114.8 kpps	46.4 kpps
DSL 4	1478.7 kpps	146.9 kpps

Throughput for Varying DSL Expression Complexity

Method	Capture Rate [$\bar{x}$]	[$sd(x)$]
DSL 1	612.2 kpps	8.8 kpps
DSL 2	726.4 kpps	9.1 kpps
DSL 3	1114.8 kpps	46.4 kpps
DSL 4	1478.7 kpps	146.9 kpps

- Do this dynamically.
- At Run-time

Defining a Clojure Function at Run-time

```
=> (def f1-str "(clojure.core/fn [] (clojure.core/println \"foo\"))")
#'user/f1-str
```

Defining a Clojure Function at Run-time

```
=> (def f1-str "(clojure.core/fn [] (clojure.core/println \"foo\"))")
#'user/f1-str
=> (def f1-list (binding [*read-eval* false] (read-string f1-str)))
#'user/f1-list
```

Defining a Clojure Function at Run-time

```
=> (def f1-str "(clojure.core/fn [] (clojure.core/println \"foo\"))")
#'user/f1-str
=> (def f1-list (binding [*read-eval* false] (read-string f1-str)))
#'user/f1-list
=> (type f1-list)
clojure.lang.PersistentList
```

Defining a Clojure Function at Run-time

```
=> (def f1-str "(clojure.core/fn [] (clojure.core/println \"foo\"))")
#'user/f1-str
=> (def f1-list (binding [*read-eval* false] (read-string f1-str)))
#'user/f1-list
=> (type f1-list)
clojure.lang.PersistentList
=> f1-list
(clojure.core/fn [] (clojure.core/println "foo"))
```

Defining a Clojure Function at Run-time

```
=> (def f1-str "(clojure.core/fn [] (clojure.core/println \"foo\"))")
#'user/f1-str
=> (def f1-list (binding [*read-eval* false] (read-string f1-str)))
#'user/f1-list
=> (type f1-list)
clojure.lang.PersistentList
=> f1-list
(clojure.core/fn [] (clojure.core/println "foo"))
=> (def f1 (eval f1-list))
#'user/f1
```

Defining a Clojure Function at Run-time

```
=> (def f1-str "(clojure.core/fn [] (clojure.core/println \"foo\"))")
#'user/f1-str
=> (def f1-list (binding [*read-eval* false] (read-string f1-str)))
#'user/f1-list
=> (type f1-list)
clojure.lang.PersistentList
=> f1-list
(clojure.core/fn [] (clojure.core/println "foo"))
=> (def f1 (eval f1-list))
#'user/f1
=> (f1)
foo
nil
```

Function in Atom for Dynamic Behaviour

```
=> (def f-atom (atom (fn [x] (inc x))))
#'user/f-atom
```

Function in Atom for Dynamic Behaviour

```
=> (def f-atom (atom (fn [x] (inc x))))
#'user/f-atom
=> (@f-atom 41)
42
```

Function in Atom for Dynamic Behaviour

```
=> (def f-atom (atom (fn [x] (inc x))))
#'user/f-atom
=> (@f-atom 41)
42
=> (reset! f-atom (fn [x] (dec x)))
#object[user$eval16$fn___17 0x45ac5f9b ...
```

Function in Atom for Dynamic Behaviour

```
=> (def f-atom (atom (fn [x] (inc x))))
#'user/f-atom
=> (@f-atom 41)
42
=> (reset! f-atom (fn [x] (dec x)))
#object[user$eval16$fn__17 0x45ac5f9b ...
=> (@f-atom 41)
40
```

Improving Dynamic Behaviour via Watch

Listing 3: Improving Dynamic Behaviour via Watch

```
...
=> (def f (atom (eval @f-list)))
#'user/f
```

Improving Dynamic Behaviour via Watch

Listing 4: Improving Dynamic Behaviour via Watch

```
...
=> (def f (atom (eval @f-list)))
#'user/f
=> (@f 41)
42
```

Improving Dynamic Behaviour via Watch

Listing 5: Improving Dynamic Behaviour via Watch

```
...
=> (def f (atom (eval @f-list)))
#'user/f
=> (@f 41)
42
=> (add-watch f-list :id (fn [k r o n-val] (reset! f (eval n-val))))
#object[clojure.lang.Atom 0xc7045b9 ...
```

Improving Dynamic Behaviour via Watch

Listing 6: Improving Dynamic Behaviour via Watch

```
...
=> (def f (atom (eval @f-list)))
#'user/f
=> (@f 41)
42
=> (add-watch f-list :id (fn [k r o n-val] (reset! f (eval n-val))))
#object[clojure.lang.Atom 0xc7045b9 ...
=> (reset! f-list `(fn [~x] (dec ~x)))
(clojure.core/fn [x-sym] (clojure.core/dec x-sym))
```

Improving Dynamic Behaviour via Watch

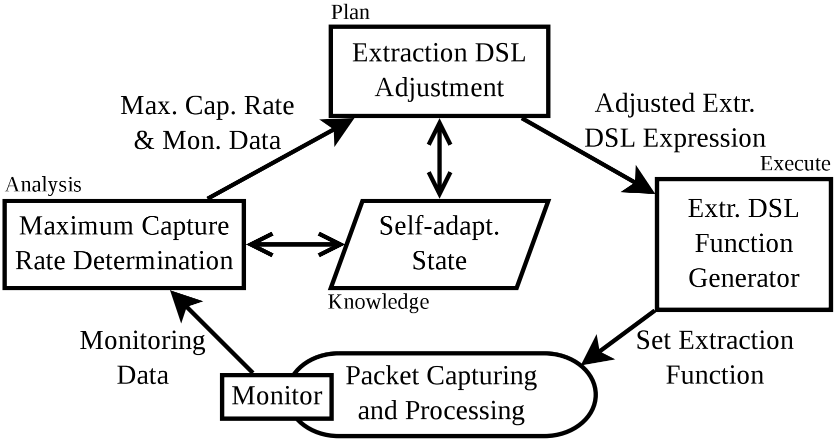
Listing 7: Improving Dynamic Behaviour via Watch

```
...
=> (def f (atom (eval @f-list)))
#'user/f
=> (@f 41)
42
=> (add-watch f-list :id (fn [k r o n-val] (reset! f (eval n-val))))
#object[clojure.lang.Atom 0xc7045b9 ...
=> (reset! f-list `(fn [~x] (dec ~x)))
(clojure.core/fn [x-sym] (clojure.core/dec x-sym))
=> (@f 41)
40
```

Where to go from here?

- Dynamic Adjustments
  → Work.
- Manual Adjustments?
  - Slow, Labour Intensive, Impossible(?!), . . .
- Automatic Dynamic Adjustments
  → Self-adaptive Adjustments

Self-adaptivity Feedback Loop ("MAPE-K")



Plan

Extraction DSL Adjustment

Max. Cap. Rate & Mon. Data

Adjusted Extr. DSL Expression

Execute

Analysis

Maximum Capture Rate Determination

Self-adapt. State

Knowledge

Extr. DSL Function Generator

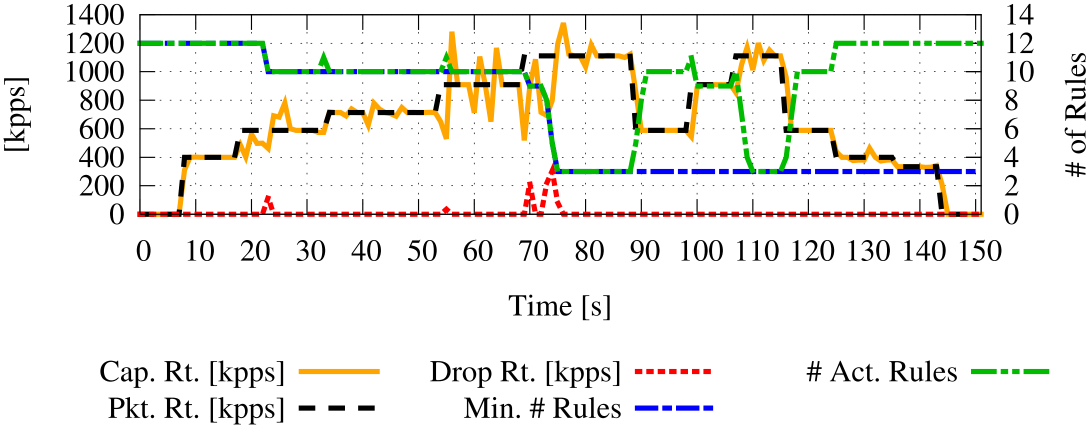
Monitoring Data

Set Extraction Function

Monitor

Packet Capturing and Processing

Self-adaptive Performance Adjustments



Summary

- Introduction
  - Computer Networks & Computer Network Monitoring
- Packet Capturing with the JVM
  - Hardware → Kernel Space → User Space → JVM
  - Per Packet vs. Bulk Data Forwarding
  - Different Memory Management Approaches
  - Improvement by about x5.6 (up to approximately 10 Gbps)
- Data Processing DSL
  - Dynamic Data Extraction
- Self-adaptive Performance-based Data Processing Adjustments

Summary continued

- DSL Abstraction Benefits
  - Extendibility, Maintainability, Flexibility, . . .
- Clojure-related Aspects
  - Homoiconic, Dynamic Capabilities, JVM-based, . . .
- Implementations: Open Source Software
  `https://github.com/ruedigergad/clj-net-pcap`
  `https://github.com/ruedigergad/dsbdp`

Summary continued

- DSL Abstraction Benefits
    - Extendibility, Maintainability, Flexibility, . . .
- Clojure-related Aspects
    - Homoiconic, Dynamic Capabilities, JVM-based, . . .
- Implementations: Open Source Software
  https://github.com/ruedigergad/clj-net-pcap
  https://github.com/ruedigergad/dsbdp

Packet Capturing with the JVM and Clojure?

Yes, we can!

End

Thank you very much for your attention!

Questions?

Ruediger Gad
Terma GmbH, Space
Darmstadt, Germany

ruga@terma.com
r.c.g@gmx.de
https://github.com/ruedigergad
https://ruedigergad.com