RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

Recitation 3

Project 3

Alireza Naghizadeh

Recap

- Click router is assembled from packet processing modules called elements
  - Individual elements implement simple router functions such as encapsulation, annotation, queueing etc.
- Click configuration is a directed graph of elements where packets follow the edges of the graph
- Configurations are written in a declarative language that supports user-defined abstractions
- Click configurations can be compounded and elements can added (modular)

Project Goals

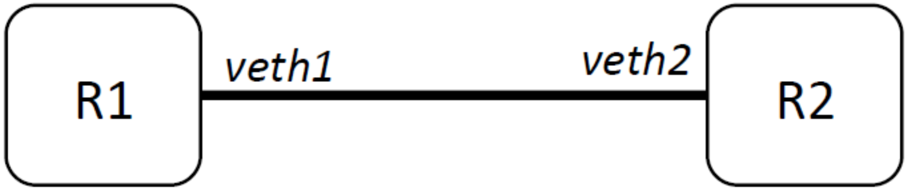
- Understanding compound elements
- Passing multiple arguments to configuration files
  - Configuration using files
- Writing custom elements
- Design and implement basic networking protocols

Compound Elements

- Group elements in larger element
- Configuration with variables
- Pass configuration to the internal elements
    - can be anything (constant, integer, elements, IP address, …)
- Motivates reuse (same as modules in a project)

Hands on with our framework

- To simplify your life, we will provide you an abstracted concept of router port
- This will allow you to implement your own protocols on top of the click framework
- You already got briefly introduced to some of these tools:
  - Remember createNet1 script?
- This creates a pair of linked interfaces (veth1 and veth2)

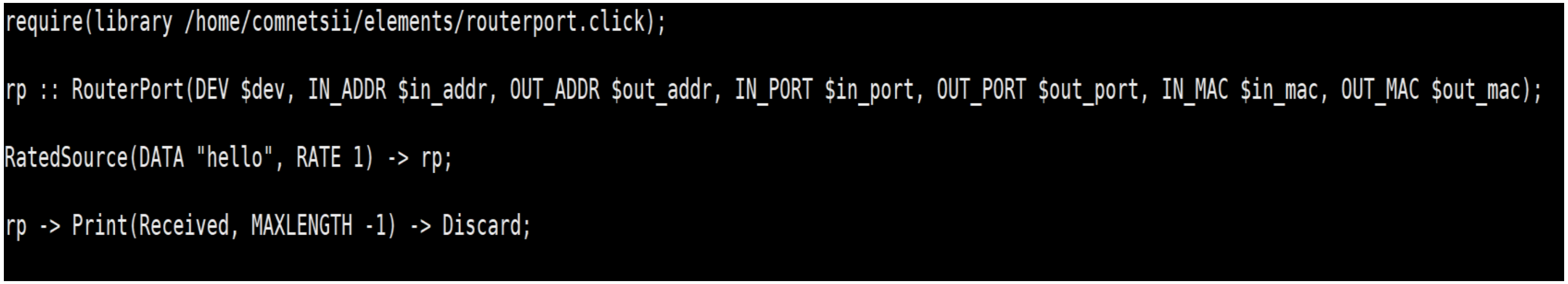


Hands on with our framework

- Port abstraction: defines one end of a link
- Everything that gets into veth1 arrives unchanged to veth2
- Abstraction obtained through the provided element *routerport.click*
- $ elements/routerport.click
- At the beginning of your configuration file:
  - require(library /home/comnetsii/elements/routerport.click)
- RouterPort is a push element with one input and one output port

Hands on with our framework

- RouterPort takes up to 7 parameters: device name, local ip, remote ip, local port, remote port, local mac, remote mac
- Example (printer.click):
  - Element that sends every one second a hello message into the port
  - Prints all packets received and discard them

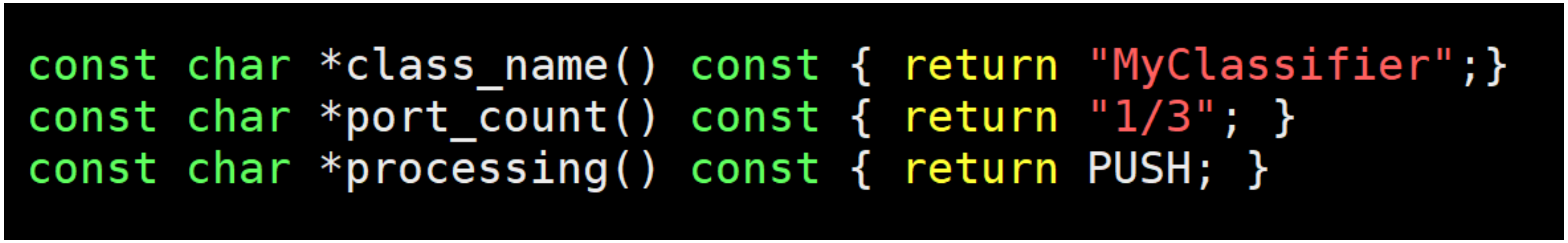


```
require(library /home/comnetsii/elements/routerport.click);

rp :: RouterPort(DEV $dev, IN_ADDR $in_addr, OUT_ADDR $out_addr, IN_PORT $in_port, OUT_PORT $out_port, IN_MAC $in_mac, OUT_MAC $out_mac);

RatedSource(DATA "hello", RATE 1) -> rp;

rp -> Print(Received, MAXLENGTH -1) -> Discard;
```

- Note: all our scripts generate pair of interfaces belonging to the same subnet
- Connect only interfaces on the same subnet
- Feel free to look into examples and elements directory.

Writing custom elements: Element Header

- Necessary in the header:
  - Include-guard macros
  - Click element macros
  - Include click/element.hh
  - The class declaration containing 3 special methods:



```
const char *class_name() const { return "MyClassifier";}
const char *port_count() const { return "1/3"; }
const char *processing() const { return PUSH; }
```

Writing custom elements: Element Source

- Necessary in the source file:
  - Include click/config.hh
  - CLICK_DECLS macro
  - CLICK_ENDDECLS macro
  - EXPORT_ELEMENT macro
  - Implementation of the methods

- Recommended:
  - Understand the element MyClassifier header and source file at (.hh and .cc):

```
comnetsii@comnetsII:~/click/elements/local$
```

Writing custom elements: SimplePushElement.hh

```
#ifndef CLICK_SIMPLEPUSHELEMENT_HH
#define CLICK_SIMPLEPUSHELEMENT_HH
#include <click/element.hh>
CLICK_DECLS
class SimplePushElement : public Element {
public:
  SimplePushElement();
  ~SimplePushElement();
   const char *class_name() const{ return “SimplePushElement”;}
   const char *port_count() const{ return “1/1”;}
   const char *processing() const{ return PUSH;}
   int configure(Vector<String>&, ErrorHandler*);
   void push(int, Packet *);
private: unit32_t maxSize; };
CLICK ENDDECLS
#endif
```

Writing custom elements: SimplePushElement.cc

```
#include <click/config.h>

#include <click/confparse.hh>

#include <click/error.hh>

#include “SimplePushElement.hh”

CLICK_DECLS

SimplePushElement::SimplePushElement(){}

SimplePushElement::~SimplePushElement(){}


int SimplePushElement::configure(Vector<String> &conf, ErrorHandler *errh)

{

   if(cp_va_kparse(conf, this, errh, “MAXPACKETSIZE”, cpkM, cpInteger,
&maxSize, cpEnd) < 0) return -1;

   if(maxSize <= 0) return errh->error(“maxsize should be larger than 0”);

   return 0; SimplePushElement::SimplePushElement(){}

}
```

Writing custom elements: SimplePushElement.cc

```
void SimplePushElement::push(int, Packet *p){
   click_chatter(“Got a packet of size %d”, p->length());
   if(p->length() > maxSize) p->kill();
   else output(0).push(p);
}
CLICK_ENDDECLS
EXPORT_ELEMENT(SimplePushElement)
```

Find these files at: click/elements/local
simplepushelem.hh
simplepushelem.cc

Try in configuration file:
RatedSource("Hello")->SimplePushElement(MAXPACKETSIZE 4)->Print->Discard;

Try changing the arguments!

Writing custom elements

- Similarly you can define pull element
  - Needs to implement pull operation
- And agnostic element
  - Needs to implement both push and pull elements
- const char *port_count() const has to return the number of ports your elements will have (it can be a flexible number, see examples)

Packet Formats

- Packet formats == structs
  - structs are a typical C concept, very low level
  - tempting to improve this by wrapping the packets in objects
  - attractive to create packet factories
- Do not do this (large overhead):
  - In terms of memory and computation (allocate objects, create and delete objects)
- Use the plain structs
  - Requires getting used to
  - Straightforward: most packet manipulation is low-level anyway

Packet Formats Example

- **Define the packet header**

```
struct MyPacketFormat{
        unit8_t type; //8 bit = 1 byte
        unit32_t lifetime; // 32 bit = 4 bytes
        in_addr destination; //IP address
};
```

- **Cast a packet to access the header**

```
MyPacketFormat* format = (MyPacketFormat*)packet->data();
format->type = 0;
format->lifetime = htonl(counter);
format->destination = ip.in_addr();
```

Compile the New Elements

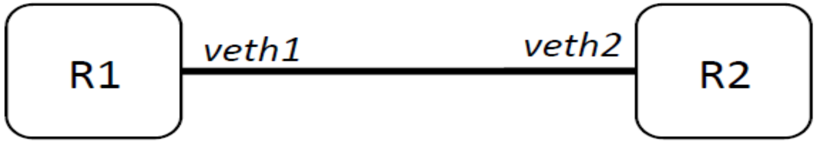
- All elements are stored in $click/elements/ directory
  - Yours should be kept in $click/elements/local/
  - Put the .hh and .cc files there
- Go to the base click folder $click
- To make those elements available:
  - $make elemlist
  - $make
- Notice new elements being compiled
- Solve any compilation problems
- Your elements are ready to be used in the configuration files! ☺

Project 3 - Part 1

Exercise 1

- Generate a small network of two routers:

  comnetsii@comnetsII:~/tools$ sudo createNet1



- Exchange packets between routers:
  - Start two click instances (two terminals) using the example found in:
    - Examples/router/printer.click
  - Make sure to set the parameters (arguments to configuration file) appropriately.

  Example – for first click:

```
comnetsii@comnetsII:~/examples/router$ sudo ~/click/userlevel/click
printer.click dev=veth2 in_addr=192.168.1.1 out_addr=192.168.1.2
in_port=10001 out_port=10002 in_mac=fe:c6:3b:4e:ce:90
out_mac=da:0f:42:8a:6e:8c
```

  - Similarly modify the command for the second click instance.

Exercise 1

- At this point, you will get an error message for inability of RouterPort to handle the arguments passed.
- Modify routerport.click without modifying printer.click (no make or compile required)
- Run both the click instances and you should see packets being received at both the terminals.

Exercise 1

- A SimplePullElement pulls the packets from the previous element, prints the content and discards it
- Create a SimplePullElement similar to SimplePushElement by writing simplepullelem.hh and simplepullelem.cc files
- Compile your code and solve the errors (if any)
- At this point, your element is ready to be used.

- For your own satisfaction, write a simple configuration file and test if your element works as desired (no submission required for this).

General Info

Include in the archive the modified routerport.click file, pull element header and source files also:

- Make sure your code is readable – use commenting!
- If you have written a test configuration file, feel free to send
- Write a README file if there is anything you think we should know
- Do not submit the old and unmodified files.
- Do not include the whole click resources

- For coming exercises, explore source code of elements in the directory: $click/elements/
- Read:
  - https://pdos.csail.mit.edu/papers/click:kohler-phd/thesis.pdf

RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

Project 3 - Part 2

Recap

- Configurations are written in a declarative language that supports user-defined abstractions
- Click router configuration is modular and can be used as a library
    - Click configurations can be compounded and elements can added
- Writing new elements in click is simple
    - Just follow the defined syntax
    - Debug when necessary
    - Use the custom elements in the configurations – simple to plug-in

Project Goals

- Understanding compound elements
- Passing multiple arguments to configuration files
  - Configuration using files
- **Writing custom elements**
- **Design and implement basic networking protocols**

Packet Formats

- Packet formats == structs
  - structs are a typical C concept, very low level
  - tempting to improve this by wrapping the packets in objects
  - attractive to create packet factories
- Do not do this (large overhead):
  - In terms of memory and computation (allocate objects, create and delete objects)
- Use the plain structs
  - Requires getting used to
  - Straightforward: most packet manipulation is low-level anyway

Packet Formats Example

- Define the packet header

```
struct MyPacketFormat{
        unit8_t type; //8 bit = 1 byte
        unit32_t lifetime; // 32 bit = 4 bytes
        in_addr destination; //IP address
};
```

- Cast a packet to access the header

```
MyPacketFormat* format = (MyPacketFormat*)packet->data();
format->type = 0;
format->lifetime = htonl(counter);
format->destination = ip.in_addr();
```

Exercise 2

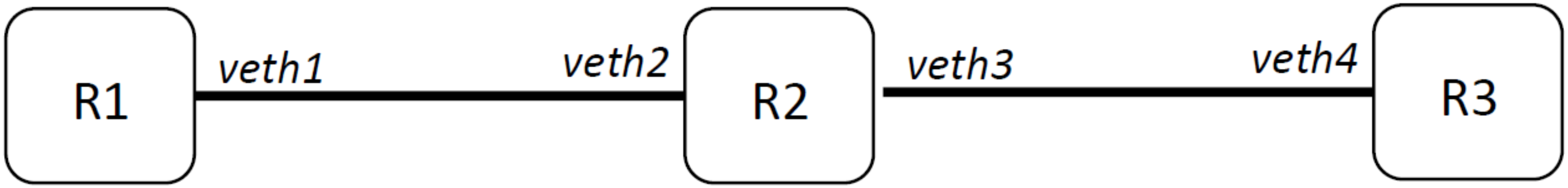
- Write an element that changes the content of every packet into another
- The new content should be configurable from the configuration script i.e. changeContent(“Newcontent”);
- Feel free to add more functionalities to your elements like maxLength, TYPE etc.
- Use the RouterPort (modified) element to build your network

Hints:

- Think of push/pull port requirements. Can only push or only pull work?
- Refer to existing elements’ implementations in click/elements/ directory

Exercise 2

- Use provided script to create 4 virtual interfaces
- Run: $ createNet2
- Obtained topology



- Tips:
  - You can reuse previous exercises to implement R1 and R3
  - At R2, you would use your implemented element to change the content of the packet sent by R1

Hand on with our framework

- Writing end-host applications
- Download package:

http://www.winlab.rutgers.edu/comnet2/Projects/downloads/project2.zip

- The package provides a simple port concept for developing at the application layer

- Examples on how to use it:

http://www.winlab.rutgers.edu/comnet2/Projects/example1.html
http://www.winlab.rutgers.edu/comnet2/Projects/example2.html
http://www.winlab.rutgers.edu/comnet2/Projects/example3.html
http://www.winlab.rutgers.edu/comnet2/Projects/example4.html

General Info

- Submission instructions for

Project 3-exercise 1 & exercise 2:

  - Deadline is 8/5.
  - Submit a single archive (zip or tar.gz)
  - Include in the archive all the source files, configuration files and README file describing your implemented protocol
  - Make sure your code is readable – use commenting!
  - Do not submit the old and unmodified files.
  - Do not include the whole click resources
  - This is a hard deadline no way for extension.

RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

Project 3 - Part 3

Exercise 3: ARQ

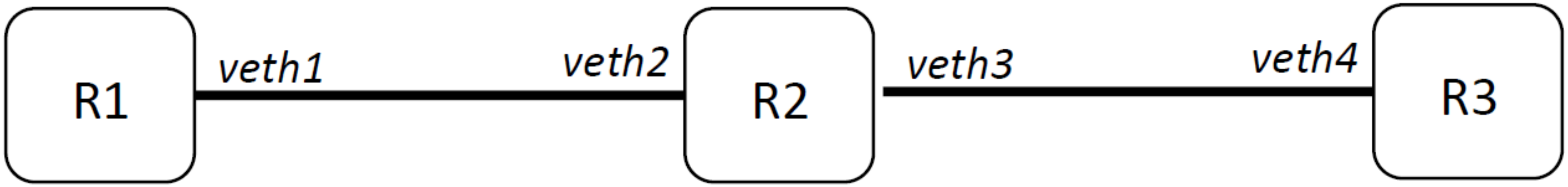
- Design a protocol to transport a file from one end of a communication link to the other over an unreliable link
- Making project with “fragmentation and reassembly” has extra credits but is not restrictly necessary.
- ARQ protocol (stop & wait)
    - Use *diff* command to compare sent and received file
- Requirements:
    - The file must be transferred from sender to receiver without any distortion/damange ➔Content of file at receiver == at sender
    - Before the transfer of the file, the name of the file has to be sent to the receiver in a packet ahead
    - You must design a packet header to carry some signaling information. It should at least include:
        - The sender’s own identification/name (can use chars or numbers)
        - Message type (DATA, ACK, NACK)
        - Sequence number

Exercise 3: ARQ

- Requirements:
  - The packet header cannot exceed 256 bytes
  - Each packet's size (including header) cannot exceed 1466 bytes
  - The filename of the transferred file cannot be hard-coded in your program. You must put the filename as a command line argument of the sender program.
  - The receiver will store the filename and should also be a command line argument at the receiver
  - When the transfer is comple, at least of the two programs should exit gracefully (not hanging)
- Where to start
  - In the package you downloaded, you will find text2.dat. This is the file you need to transfer.
  - This requires "fragmentation and reassembly", you can create and send a file with 500 bytes to remove this requirement.
  - In src/ directory, there are two files: common.h and common.cpp, which provide some basic classes and functions.
  - In doc/ directory you will find the complete documentation

Exercise 3: ARQ

- Use provided script to create 4 virtual interfaces
- Run: $ createNet2
- Obtained topology



- Tips:
  - R only needs to forward packets i.e. you will need 2 elements, the 2 RouterPort

Exercise 3: ARQ

- Use the application layer package to implement the end host applications
- Implement one sender and one receiver
- How to test
  - If running the program using the two application clients, setup the used ports with the following characteristics:
    - The packet over the link is either lost or received completely
    - Loss probability is 0.2
    - No partial information is heard and no need for a CRC check

Hints:

- Refer example1, 2 and 3 (you can reuse most of the code)
- Run each program in a separate console for easy debugging
- The sender need send to "*localhost*" (the destination address of the sending port). The receiver also needs to send to "localhost" if it sends back some feedback signaling.

Exercise 3: Tips

- One timer associated with each sending port.
- If you think your protocol programming need more timers, you can design a new class like SendingPort class, with more timers.
- Please place your design in a separate file. Then derive your working class from that class.
- If you want to combine the functions of a sending port and a receiving port in a new class, e.g. "new_port", and use it in both sender and receiver side, go ahead to design it in a separate file.
- You may wonder why the sender and receiver have different protocols (programs). They need not to be different and shall be same. But as we mixed application (a uni-directional file transfer) with a link layer protocol in a same C++ program, it turns out to be two different ones.

- Use gdb for debugging purpose.
- The 256 byte for packet header size is only an upper bound. You can choose what any small number you want for header size. The PacketHdr class does not hard code any format. You can use its API to design your format.

General Info

- Submission instructions for

Project 3-exercise 3 (ARQ):

  - Deadline is 8/14.
  - Submit a single archive (zip or tar.gz)
  - Include in the archive all the source files, configuration files and README file describing your implemented protocol
  - Make sure your code is readable – use commenting!
  - Do not submit the old and unmodified files.
  - Do not include the whole click resources
  - This is a hard deadline no way for extension.