Click Coding

Coding with Click Modular Router

Johan Bergs Jeremy Van den Eynde

University of Antwerp
imec - IDLab Research Group

October 2017

IDLab
INTERNET & DATA LAB

Outline

IDLab
INTERNET & DATA LAB

Storing own elements

All elements are stored in the `elements/` directory

- Yours should be put in `elements/local`
- Put the `.hh` and `.cc` files there

To make those elements available:

```
make elemlist
make
```

Notice new elements being compiled, solve any compilation problems and use your elements

Do it yourself

Let's make an example element

- 1 input, 1 output, Push
- Configure a packet size threshold, if larger: drop packet

Download the source code online to avoid copy errors at `https://github.com/mosaicresearch/click_modular_router_lessons/tree/master/examples/simpleelements`

IDLab
INTERNET & DATA LAB

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
const char *port_count() const
const char *processing() const
```

Element header

Necessary in the source file:

- Include `click/config.hh` **first**!
- `CLICK_DECLS` macro
- `CLICK_ENDDECLS` macro
- `EXPORT_ELEMENT` macro
- Implementation of the methods

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:
    uint32_t maxSize;
};
CLICK_ENDDECLS
#endif
```

IDLab
INTERNET & DATA LAB

simplepushelement.cc I

```
#include <click/config.h>
#include <click/args.hh>
#include <click/error.hh>
#include "simplepushelement.hh"

CLICK_DECLS
SimplePushElement::SimplePushElement() {}
SimplePushElement::~SimplePushElement() {}

int SimplePushElement::configure(Vector<String> &conf, ErrorHandler *errh) {
  if (Args(conf, this, errh).read_m("MAXPACKETSIZE", maxSize).complete() < 0) return
      -1;
  if (maxSize <= 0) return errh->error("maxsize should be larger than 0");
  return 0;
}

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)
```

IDLab
INTERNET & DATA LAB

What's in a name

To avoid confusion, we recommend to:

- Make the ElementName CamelCase
- Use that name in the `class_name` macro
- Use that name in lowercase for the header (.hh) and source (.cc) files
- Use that name in uppercase, with `CLICK_` prepended, for the include guards

IDLab
INTERNET & DATA LAB

simplepullelement

simplepullelement.hh:

```
class SimplePullElement: public Element {
  public: ...
    const char *processing() const { return PULL; }
    Packet* pull(int);
}
```

simplepullelement.cc:

```
Packet* SimplePullElement::pull(int) {
  Packet* p = input(0).pull();
  if(p == 0) return 0;
  click_chatter("Got a packet of size %d",p->length());
  if (p->length() > maxSize){
    p->kill();
    return 0;
  } else return p;
}
```

IDLab
INTERNET & DATA LAB

simpleagnosticelement

```
simpleagnosticelement.hh:
class SimpleAgnosticElement: public Element {
  public: ...
    const char *processing() const { return AGNOSTIC; }
    void push(int, Packet *);
    Packet* pull(int);
};

simpleagnosticelement.cc
void SimpleAgnosticElement::push(int, Packet *p) {
  // see push element
}
Packet* SimpleAgnosticElement::pull(int) {
  // see pull element
}
```

IDLab
INTERNET & DATA LAB

simpleagnosticelement11

simpleagnosticelement11.hh:

```
class SimpleAgnosticElement11: public Element {
  public: ...
    const char *processing() const { return AGNOSTIC; }
    const char *port_count() const { return "1/1"; }
    Packet *simple_action(Packet *);
};
```

simpleagnosticelement11.cc

```
Packet* SimpleAgnosticElement11::simple_action(Packet *p){
  click_chatter("Got a packet of size %d",p->length());
  if (p->length() > maxSize){
    p->kill();
    return 0;
  } else return p;
}
```

Port count

Defined by `const char *port_count() const`. Can return:

- `"1/1"`: one input port, one output port
- `"1/2"`: one input port, two output ports
- `"1-2/0"`: one or two input ports and zero output ports.
- `"1/-6"`: One input port and up to six output ports.
- `"2-/-"`: At least two input ports, any number of output ports.
- `"3"`: Exactly three input and output ports. (If no slash appears, the text is used for both input and output ranges.)
- `"1-/="`: At least one input port and the same number of output ports.
- `"1-/=+"`: At least one input port and one more output port than there are input ports.

Parsing configurations with Args I

Call this on

- the configuration (`conf`)
- the element (`this`)
- the errorhandler (`errh`)
- concatenate with variants of `read()`
- end with `complete()`

Check the return value (C-style):

- 0: all parsing went fine
- Negative: problems detected, configure should return -1

Parsing configurations with Args II

```
int MyElement::configure(Vector<String> &conf,
    ErrorHandler *errh) {
  String data;
  uint32_t limit = 0;
  bool stop = false;

  if (Args(conf, this, errh)
    .read_mp("DATA", data)
    .read_p("LIMIT",limit)
    .read("STOP", stop)
    .complete() < 0) return -1;
  ...
}
```

Arguments to read I

Argument name

- Type: `const char *`
- Example: "DATA".

Result storage: variable name of the variable that will hold the passed value

Variants of read

- `read`: default, no special requirements
- `read_m`: Mandatory argument
- `read_p`: Positionally specified argument
- `read_mp`: Mandatory, positionally specified argument

Args: example I

```
Args(conf, this, errh).read("P", p).read("ADDRESS",
    addr).complete()
```

Will this match:

- `P 5, ADDRESS 192.168.0.3`
- `ADDRESS 1.2.3.4, P 5`
- `P 5`
- `ADDRESS 192.168.0.3`
- `(nothing)`

IDLab
INTERNET & DATA LAB

Args: example II

How about

```
Args(conf, this, errh).read("P", p).read_m("ADDRESS",
    addr).complete()
```

Will this match:

- `P 5, ADDRESS 192.168.0.3`
- `ADDRESS 1.2.3.4, P 5`
- `P 5`
- `ADDRESS 192.168.0.3`
- `(nothing)`

IDLab
INTERNET & DATA LAB

Parsing elements I

Elements might need other elements

- Pass them in the configuration
- Check their name and type
- Calling public methods and accessing public members is possible

In Click script:

```
SimpleElement(UsedElement);
```

or

```
myElement:: UsedElement;
SimpleElement(myElement);
```

Add an element to the header:

Parsing elements II

```
#include "usedelement.hh"
class ElementUser: public Element {
 private:
   UsedElement* used;
}
```

Use the element in the C++ code

```
ElementUser::push(...) {
  used->doSomething(...);
}
```

Parsing elements III

Check and configure the element in the configure function:

```
int ElementUser::configure(Vector<String> &conf,
    ErrorHandler *errh) {
  UsedElement* tempUsedElement;
  int result = Args(conf, this, errh).read("ANELEMENT",
    ElementCastArg("UsedElement"),
    tempUsedElement).complete();

  if(result < 0) return result; // parsing failed

  used = tempUsedElement;
  return 0;
}
```

Click library functions

The C++ STL cannot be used in the kernel

- Click provides its own implementation, use it
- Equivalents to most STL datastructures available
- E.g. vector, hashmap, ...

Additional types: Timers and tasks to schedule actions, see later
Additional functions:

- Manipulate strings
- Manipulate packets
- E.g. `click_gettimeofday(struct timeval *tv)`

IDLab
INTERNET & DATA LAB

Click containers

Overview of the most important types

- Vector
- HashMap
- String

Click STL: vector I

Constructor: straightforward template

```
Vector<SomeThing> myvector;
```

Even better: typedef it for reuse

```
Typedef Vector<SomeThing> SomeThingVector;
```

Use macro magic for template instantiation

```
// generate Vector template instance
#include <click/vector.cc>
#if EXPLICIT_TEMPLATE_INSTANCES
template class Vector<SomeThing>;
#endif
```

IDLab
INTERNET & DATA LAB

Click STL: vector II

Add things to it: `myvector.push_back(some_thing);`

Use iterators to walk over it

```
for (SomeThingVector::const_iterator i =
    myvector.begin(); i!= myvector.end(); i++){
  doSomeThingWith(*i);
}
```

And remove things with iterators

```
myyvector.erase(i);
```

Or pop it as a stack/heap

```
myvector.pop_front(); myvector.pop_back();
```

IDLab
INTERNET & DATA LAB

Click STL: hashmap example I

```
#ifndef AODVSETRREPHEADERS_HH
#define AODVSETRREPHEADERS_HH
#include <click/element.hh>

CLICK_DECLS
typedef HashMap<Packet*, IPAddress*> DestinationMap;
class AODVSetRREPHeaders : public Element {
  public:
    virtual void push (int, Packet *);
    void addRREP(Packet*,IPAddress *);
  private:
    DestinationMap destinations;
};
CLICK_ENDDECLS
#endif
```

IDLab
INTERNET & DATA LAB

Click STL: hashmap example II

```
AODVSetRREPHeaders::AODVSetRREPHeaders():
  destinations(){}
void AODVSetRREPHeaders::push (int port, Packet * p){
  ...
  // packet should be in destinations
  DestinationMap::Pair * pair =
    destinations.find_pair(packet);
  assert(pair);
  IPAddress* destination = pair->value;
  ... // do something with destination
  delete pair->value; // free memory properly
  destinations.remove(packet); // then remove from map
  ...
}
```

IDLab
INTERNET & DATA LAB

Click STL: hashmap example III

```
void AODVSetRREPHeaders::addRREP(Packet* rrep,
    IPAddress * ip){
  destinations.insert(rrep,ip);
}

// macro magic to use bighashmap
#include <click/bighashmap.cc>
#if EXPLICIT_TEMPLATE_INSTANCES
template class HashMap<Packet*, IPAddress*>;
#endif
```

Click STL: string

Use it when manipulating C strings

```
String test = "mytest";
```

Use standard operators to modify it

```
test += "should say hello";
```

When used in click_chatter, convert it

```
click_chatter("my string is %s",test.c_str());
```

Packet formats

You want to make your own packets, here's how

Format closely mirrors RFCs

Use structs

- Fill them with signed/unsigned ints, in_addr, ...
- Easy packet manipulation
- Avoids dirty operations with chars and bytes
- Define those in shared headers for reuse

Create your packet format

```
struct MyPacketFormat {
  uint8_t type; // 8 bit = 1 byte
  uint32_t lifetime; // 32 bit = 4 bytes
  in_addr destination; // IP address
};
```

IDLab
INTERNET & DATA LAB

Click data types

Click already defines lots of data types for you, see `include/clicknet`:

- `click_ether`
- `click_ip`
- `click_udp`
- `click_tcp`
- etc.

IDLab
INTERNET & DATA LAB

Creating a packet

Provide headroom and tailroom:

```
int tailroom = 0;
int packetsize = sizeof(MyPacketFormat);
int headroom = sizeof(click_ip) + sizeof(click_udp) + sizeof(click_ether);
WritablePacket *packet = Packet::make(headroom, 0, packetsize, tailroom);
if (packet == 0) return click_chatter("cannot make packet!");
memset(packet->data(), 0, packet->length());
MyPacketFormat* format = (MyPacketFormat*) packet->data();
format->type = 0;
format->lifetime = htonl(counter);
format->destination = ip.in_addr();
```

Destroy with `packet->kill()`, only way to free your memory correctly!

IDLab
INTERNET & DATA LAB

Processing a packet I

Cast the packet data to the right format

```
// start with the first part
my_header* head = (my_header*)(packet->data());
// continue with later bytes
int offset = sizeof(my_header)
second_header* h2 = (my_second_header*)(my_header + 1);
```

Use the format to read from and write to

```
if (head->somefield == 2){
  head->otherfield = htons(38);
  ...
}
```

IDLab
INTERNET & DATA LAB

Processing a packet II

Only write to writable packets

```
WritablePacket *q = p->uniqueify();
// only use q now!
q->somefield = newvalue
```

Manipulating packet size

Add data with `push(unsigned len)`

- Inserts the data at the beginning of the packet
- Create enough headroom, otherwise expensive push!

Remove data with `pull(unsigned len)`

- Removes the data at the beginning of the packet
- Frees headroom

Equivalents at tail of packet: `put` and `take`

IDLab
INTERNET & DATA LAB

Manipulating packet annotations

Get IP header:

```
packet->ip_header();
```

Set IP header of length len:

```
packet->set_ip_header(const click_ip* header, unsigned
    len);
```

Similar operations exist for TCP and UDP headers
Both operations require header annotations, set by the `MarkIPHeader` element!

IDLab
INTERNET & DATA LAB

Simple timer I

Runs the run_timer function upon expiry

```
class MyElement: public Element {
  public:
    void run_timer(Timer*);
  private:
    Timer timer;
}

MyElement::MyElement(): timer(this){}
int MyElement::configure(Vector<String> &conf,
    ErrorHandler *errh){
  timer.initialize(this);
  timer.schedule_after_msec(1000);
  return 0;
}
```

Simple timer II

```
void MyElement::run_timer(Timer* t){
  click_chatter("we are now 1 second later");
  timer.schedule_after_msec(1000);
}
```

Advanced timer with extra data I

Run your callback function upon expiry with data, because you want to know some context information.
Code is a little bit harder:

```
class MyElement: public Element{
  private:
    struct TimerData{ // callback data
      MyElement* me;
      Something* s;
  }
  static void handleExpiry(Timer*, void *); // callback
    function
  void expire(const MyElement &, TimerData *);
}
```

IDLab
INTERNET & DATA LAB

Advanced timer with extra data II

```
void MyElement::someFunction(){
  TimerData* timerdata = new TimerData();
  timerdata->s = new Something();
  timerdata->me = this;
  Timer t = new
    Timer(&MyElement::handleExpiry,timerdata);
  t->initialize(this);
  t->schedule_after_msec(2500);
}
void MyElement::handleExpiry(Timer*, void * data){
  TimerData * timerdata = (TimerData*) data;
  assert(timerdata); // the cast must be good
  timerdata->me->expire(*timerdata->s,timerdata);
}
```

IDLab
INTERNET & DATA LAB

Advanced timer with extra data III

```
void MyElement::expire(const Something& s, TimerData*
    timerdata){
  // do things with Something
  // timerdata passed to free memory after timer expiry
}
```

Adding handlers

Add to element by overriding `add_handlers`

- Callback with function pointers
- Refer to static methods

Use `add_read_handler` and `add_write_handler`

Adding a write handler I

```
class WriteElement: public Element{
  public:
    static int handle(const String &conf, Element *e,
    void * thunk, ErrorHandler * errh);
    void add_handlers();
}

int WriteElement::handle(const String &conf, Element
    *e, void * thunk, ErrorHandler * errh){
  WriteElement * me = (WriteElement *) e;
  if(Args(conf, this, errh).read(...).complete() < 0)
    return -1;
  me->doSomethingWithParsed(...);
  return 0;
}
```

IDLab
INTERNET & DATA LAB

Adding a write handler II

```
void WriteElement::add_handlers(){
  add_write_handler("a_handle", &handle, (void *)0);
}
```

Adding a read handler

```
class ReadElement: public Element{
  public:
    static String handle(Element *e, void * thunk);
    void add_handlers();
}

String ReadElement::handle(Element *e, void * thunk){
  ReadElement * me = (ReadElement *) e;
  return me->giveSomeValue(...);
}
void ReadElement::add_handlers(){
  add_read_handler("a_handle", &handle, (void *)0);
}
```

IDLab
INTERNET & DATA LAB

References I

Click website: `http://www.read.cs.ucla.edu/click/`

- Element documentation (by name or category)
- Programming Concepts
- Doxygen documentation

Click thesis (online: publications, PhD thesis)

- Comprehensive documentation of every concept
- Interesting chapters for development:
  - Introduction
  - Architecture: elements, packets, connections, push and pull, packet storage, element implementation
  - Language: syntax, configuration strings, compound elements

Click source code

References II

- /elements/: dozens of elements, some more trivial than others
- /include/: the Click STL headers

Introduction

Click graphs can get large, sometimes you need visual checks
Helps you verify the situation
Tools available:

- click-flatten
- click-viz

click-flatten

Flattens out compound elements for click-viz, the resulting router will do exactly the same
Located in tools/click-flatten

IDLab
INTERNET & DATA LAB

click-flatten (continued)

```
tools/click-flatten/click-flatten ping-3.click
# 33 "ping-3.click"
AddressInfo@1 :: AddressInfo(senderaddr 10.0.0.1
  1A:7C:3E:90:78:41);
# 34 "ping-3.click"
AddressInfo@2 :: AddressInfo(receiveraddr 10.0.0.2
  1A:7C:3E:90:78:42);
# 40 "ping-3.click"
Null@5 :: Null;
# 43 "ping-3.click"
Null@6 :: Null;
# 4 "ping-3.click"
sender/ICMPPingSource@1 :: ICMPPingSource(senderaddr,
  receiveraddr);
...
```

IDLab
INTERNET & DATA LAB

click-flatten (continued)

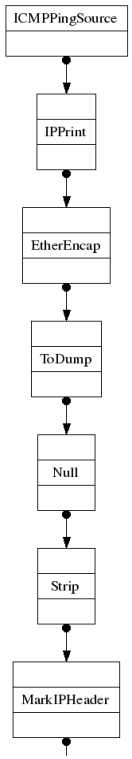
```
sender/ICMPPingSource@1 -> sender/IPPrint@2
  -> sender/EtherEncap@3
  -> sender/ToDump@4
  -> Null@5
  -> receiver/Strip@1
  -> receiver/MarkIPHeader@2
...
  -> sender/IPPrint@6
  -> sender/Discard@7;
sender/filter [1] -> sender/IPPrint@8
  -> sender/Discard@9;
```

IDLab
INTERNET & DATA LAB

click-viz

Basic visualization of Click scripts, renders dotty output (Graphviz software)
Usage:

```
tools/click-flatten/click-flatten
    ping-1.click |
    tools/click-viz/click-viz | dot
    -Tpng > ping-1.png
```



IDLab
INTERNET & DATA LAB

Gnu Debugger

A low-level, well known and very powerful debugger
Basics:

- `gdb userlevel/click`
- `run someclickscript.click`
- (wait for crash)
- `bt`
- `quit`

IDLab
INTERNET & DATA LAB

valgrind

A memory debugger, shows and debugs invalid memory access
Basic usage: `valgrind userlevel/click somescript.click`
Errors and warnings might come from glibc or Click elements, and might appear in other elements.