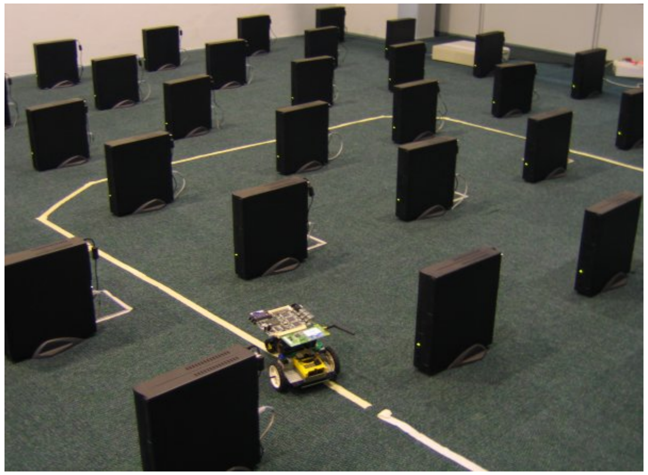
meraka INSTITUTE
African Advanced Institute for Information & Communications Technology

Meraka Wireless Mesh Testbed



Workshop on building experiments

Day 2: Click modular router

Based on presentation by Bart Braem of the PATS group at University of Antwerp

Presented by David Johnson

Contents of Talk

- **Visualizing the Click graph with Clicky**
- **Building your own elements**
  - How to write your own element
  - Push, pull and agnostic version
  - Parsing its configuration
  - The Click STL
  - Packet manipulation
  - Timers
  - Handlers
  - References
- **Tutorials**

Why click

- **Developed by M.I.T. To simplify the process of building packet processors**
  - + Could theoretically replace any linux tool which involves packet processing
    - – Iptables, tc, all routing protocols …
- **Designed in 1999 and published in thesis by Eric Kohler in 2000**
- **All of M.I.T. Work on roofnet project using Srcr mesh protocols was developed using Click**

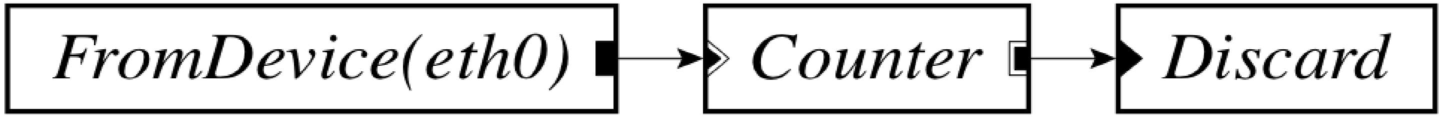
General concept

- **Create state machine for processing packets**
- **Architecture centered on elements**
  - Small building blocks
  - Perform simple operations like queuing
  - Written in C++
- **Click routers**
  - Directed graphs of elements
  - Text files
- **Open source**
  - Runs on Linux, Mac OS X and BSD

Click graph

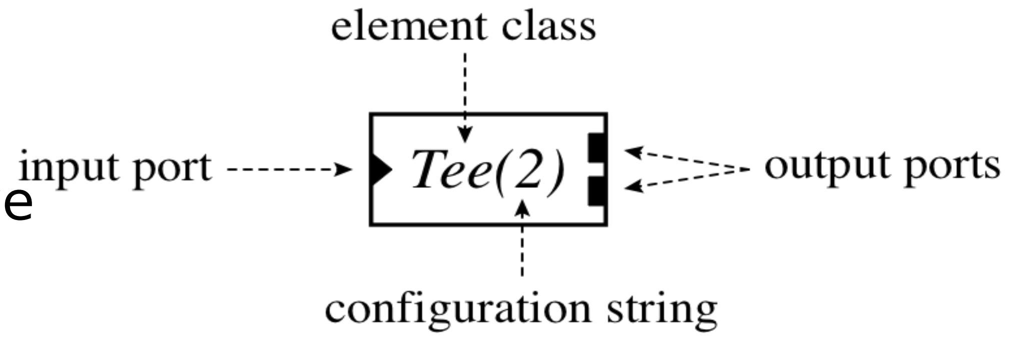
- **Elements connected by edges**
  - + Output ports to input ports
- **Describes possible packet flows**

```
• FromDevice(eth0)
      -> Counter
      -> Discard;
```



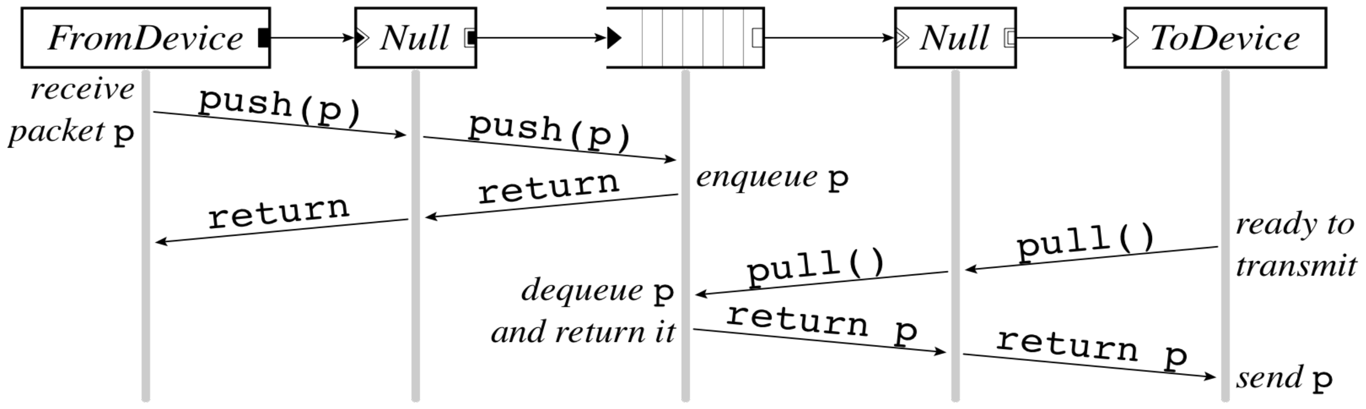
Elements

- **Class**
  - + element type (reuse!)
- **Configuration string**
  - + initializes this instance
- **Input port(s)**
  - + Interface where packets arrive
  - + Triangles
- **Output port(s)**
  - + Interface where packets leave
  - + Squares
- **Instances can be named**
  - + myTee :: Tee



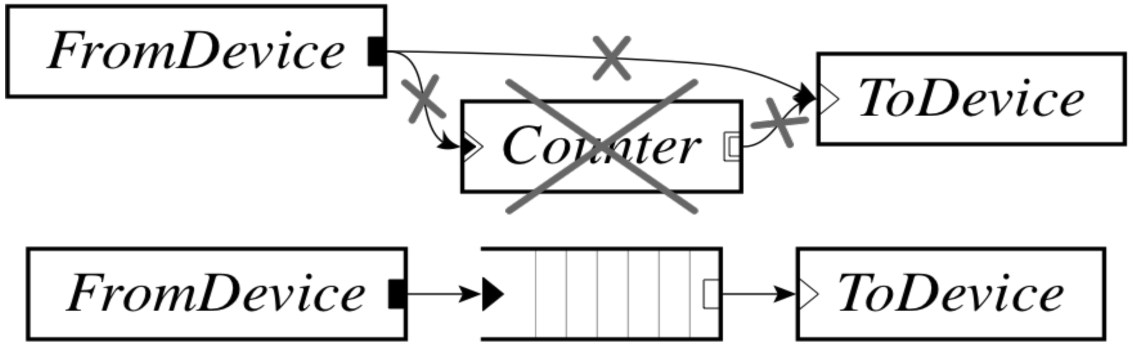
Push and pull ports

- **Push port**
  - Filled square or triangle
  - Source initiates packet transfer
  - Event based packet flow
- **Pull port**
  - Empty square or triangle
  - Destination initiates packet transfer
  - Used with polling, scheduling, …
- **Agnostic port**
  - Square-in-square or triangle-in-triangle
  - Becomes push or pull (inner square or triangle filled or empty)



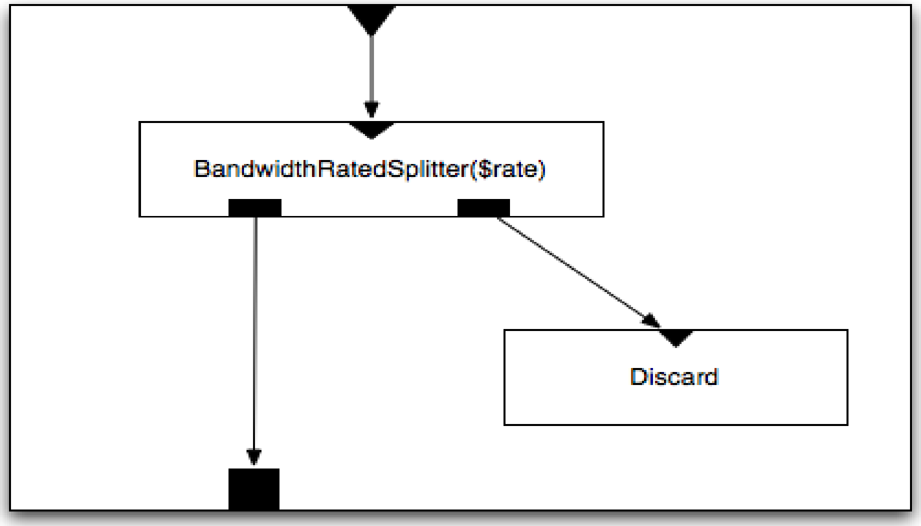
Push and pull ports violations

- **Push port**
  - Has to be connected to push or agnostic port
  - Conversion from push to pull
    - With push-to-pull element
    - E.g. queue
- **Pull port**
  - Has to be connected to pull or agnostic port
  - Conversion from pull to push
    - With pull-to-push element
    - E.g. unqueue



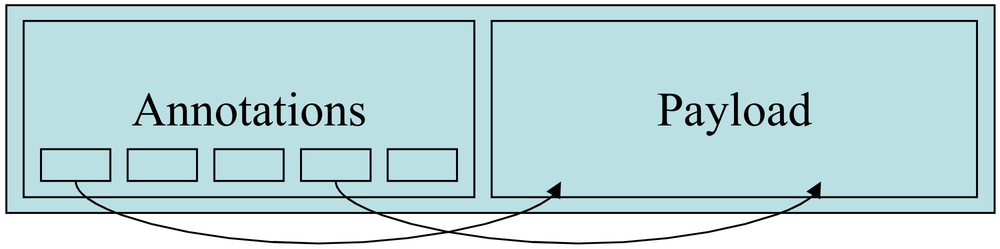
Compound elements

- **Group elements in larger elements**
- **Configuration with variables**
  - Pass configuration to the internal elements
  - Can be anything (constant, integer, elements, IP address, ...)
  - Motivates reuse



Packets

- **Packet consists of**
  - Payload
    - char*
    - Access with struct*
  - Annotations (metadata to simplify processing)
    - "post-it"
    - IP header information
    - TCP header information
    - Paint annotations
    - User defined annotations



Click scripts (1)

- Text files describing the Click graph
  - Elements with their configurations
  - Define instances of objects
  - Connections

-
  ```
  src :: FromDevice(eth0);
  ctr :: Counter;
  sink :: Discard;
  src -> ctr;
  ctr -> sink;
  ```

-
  ```
  FromDevice(eth0)
      -> Counter
      -> Discard;
  ```

Click scripts (2)

- **Input and output ports identified by number (0,1,..)**
  - Input port: `-> [nr1]Element ->`
  - Output port: `-> Element[nr2] ->`
  - Both: `-> [nr1]Element[nr2] ->`
  - Only one port: number can be omitted (default port 0)
  - Motivates instance naming

```
mypackets::IPClassifier(dst host $myaddr,-);
FromDevice(eth0)
    -> mypackets;
mypackets[0]
    -> Print(mine)
    -> [0]Discard;
mypackets[1]
    -> Print("the others")
    -> Discard;
```

Compound elements in Click Scripts

-
  ```
  elementclass DumbRouter {
      $myaddr |
      mypackets :: IPClassifier(dst host $myaddr,-);
      input[0]
            -> mypackets;
      mypackets[0]
            -> [1]output;
      mypackets[1]
            -> [0]output;
  }
  u :: DumbRouter(1.2.3.4);
  FromDevice(eth0)
      -> u;
  u[0]
      -> Discard;
  u[1]
      -> ToDevice(eth0);
  ```

* *Also see Classifier element*

Element configurations

- **Listed in click script**
  - First required arguments
  - Then optional arguments
  - Then arguments by keyword (after keyword)
- **Lots of types supported**
  - Integers
  - Strings e.g. “data”
  - IP addresses 143.129.77.30
  - Elements

Parsing configurations: examples

```
SimpleElement(“data”)
SimpleElement(“data”,ACTIVE false)
SimpleElement(“moredata”,800)
SimpleElement(“data”,800,DATASIZE 67, SOURCE
  1.2.3.4)
```

Running Click

+ Kernel module
  - ***click-install sample.click***
  - Completely overrides Linux routing
  - High speed, requires root permissions
  - Crashing Click = crashing kernel = crashing system
+ Userlevel
  - ***click sample.click***
  - Runs as a daemon on a Linux system
  - Easy to install and still fast
  - Recommended for development
+ nsclick
  - Runs as a routing agent within the ns-2 network simulator
  - Multiple routers on 1 system
  - Difficult to install but less hardware needed

Userlevel

- Send packets to and receive packets from the network device
  - + Connects to the network interface
  - + Delivered to Click at FromDevice(ethX)
    - – You get all traffic on this interface
  - + Sent to system by ToDevice(ethX)

Userlevel

- **Send packets to and receive packets from the Linux protocol stack**
    - – Note Linux routing still working
  - + With a virtual tun/tap interface
  - + Set the default route to the tun/tap device to make sure traffic passes through the Click route
  - + Deliver to Linux stack - Use ToHost(ethX)
  - + Delivered to Click – Use FromHost(ethX)
    - – You only get this if its for you
  - + Warning: Linux routing is still working
  - + Addresses (IP and MAC) must be correct
  - + Make sure IPs from Click are different from system IP's

Wireless with click

- **To use click with wireless card first put it in raw mode**
  - rmmod ath_pci
  - modprobe ath_pci autocreate=monitor
  - ifconfig ath0 up
- **Many wireless elements have been written to allow you to build**
  - Access point from scratch
  - Station mode from scratch
    - See WiFi, Wireless AccessPoint and WirelessStation in element docs
- **Can now redefine 802.11 standard by changing the interchange of management frames**

Wireless with click

- **Access point example**

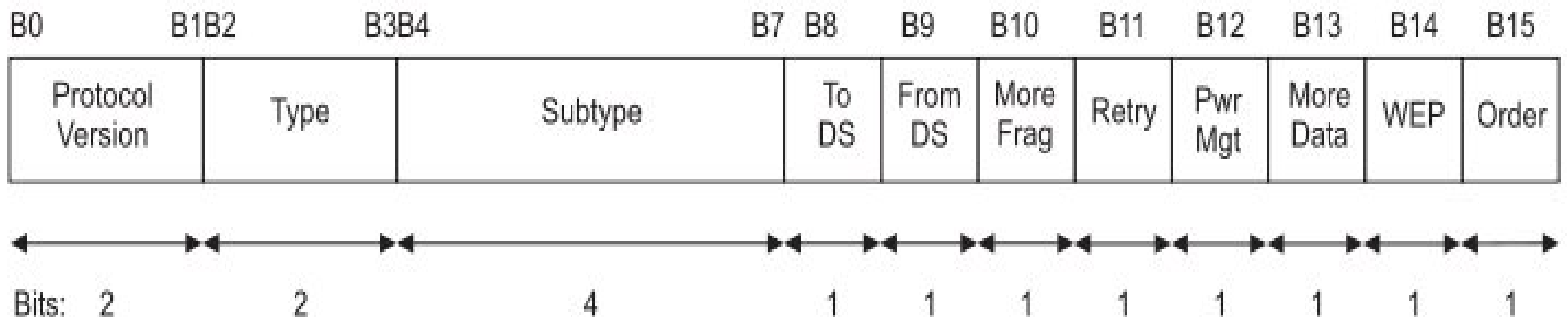
```
from_dev :: FromDevice(ath0, PROMISC true)
-> RadiotapDecap()
-> extra_decap :: ExtraDecap()
-> FilterPhyErr()
-> tx_filter :: FilterTX()
-> HostEtherFilter(station_address, OFFSET 4)
-> dupe :: WifiDupeFilter()
-> wep_decap :: WepDecap()
-> wifi_cl :: Classifier(0/00%0c, //mgt
      0/08%0c, //data
      );

// management
wifi_cl [0] -> management_cl :: Classifier(0/00%f0, //assoc req
            0/10%f0, //assoc resp
            0/80%f0, //beacon
            0/a0%f0, //disassoc
            0/50%f0, //probe resp
            0/b0%f0, //auth
            );
```

See: http://read.cs.ucla.edu/click/elements/classifier

Wireless with click – 802.11 specification

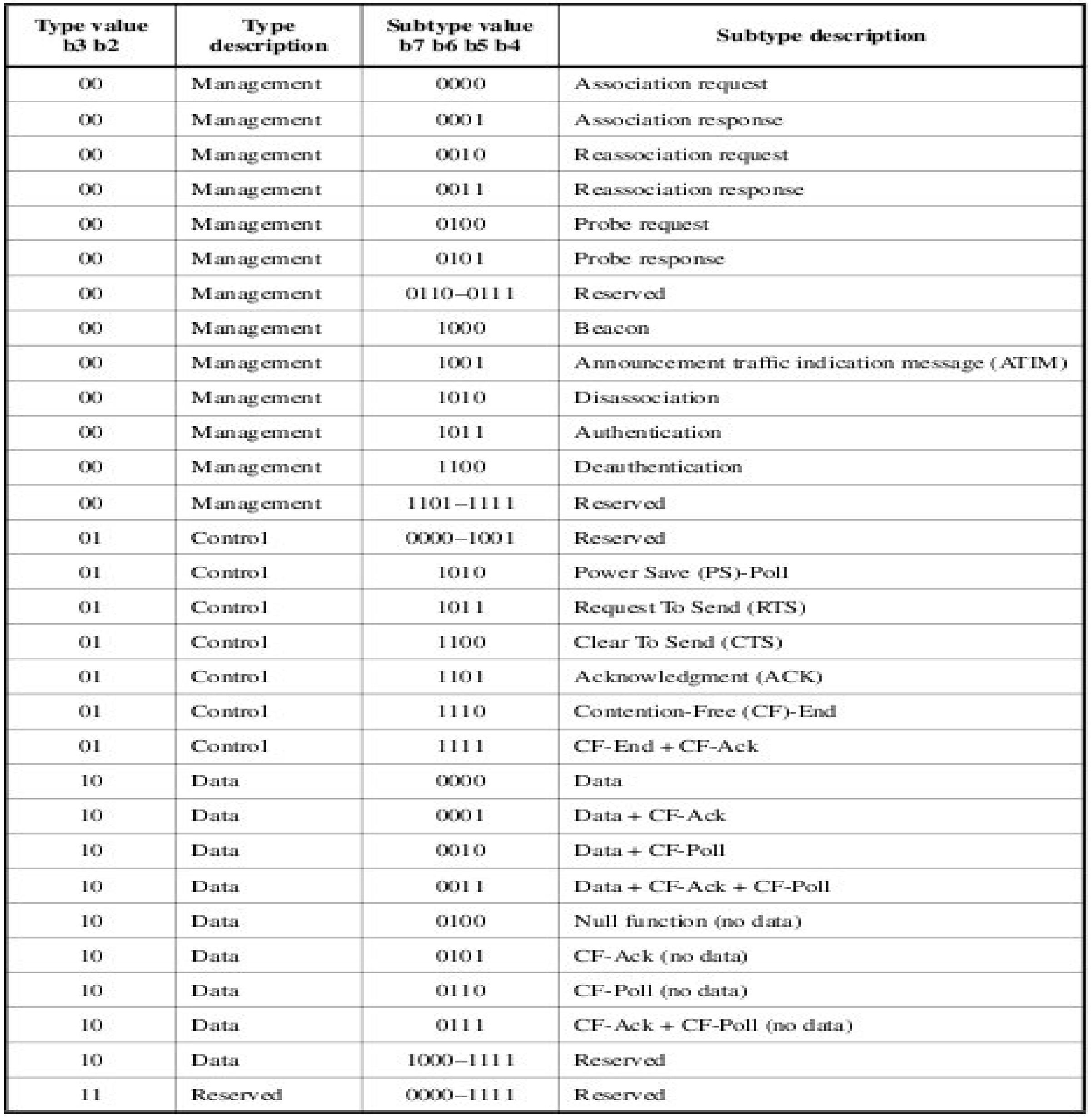
Page 35 of 802.11 specification

The Frame Control field consists of the following subfields: Protocol Version, Type, Subtype, To DS, From DS, More Fragments, Retry, Power Management, More Data, Wired Equivalent Privacy (WEP), and Order. The format of the Frame Control field is illustrated in Figure 13.



Wireless with click – 802.11 specification

Page 36 of 802.11 specification



Type value b3 b2	Type description	Subtype value b7 b6 b5 b4	Subtype description
00	Management	0000	Association request
00	Management	0001	Association response
00	Management	0010	Reassociation request
00	Management	0011	Reassociation response
00	Management	0100	Probe request
00	Management	0101	Probe response
00	Management	0110–0111	Reserved
00	Management	1000	Beacon
00	Management	1001	Announcement traffic indication message (ATIM)
00	Management	1010	Disassociation
00	Management	1011	Authentication
00	Management	1100	Deauthentication
00	Management	1101–1111	Reserved
01	Control	0000–1001	Reserved
01	Control	1010	Power Save (PS)-Poll
01	Control	1011	Request To Send (RTS)
01	Control	1100	Clear To Send (CTS)
01	Control	1101	Acknowledgment (ACK)
01	Control	1110	Contention-Free (CF)-End
01	Control	1111	CF-End + CF-Ack
10	Data	0000	Data
10	Data	0001	Data + CF-Ack
10	Data	0010	Data + CF-Poll
10	Data	0011	Data + CF-Ack + CF-Poll
10	Data	0100	Null function (no data)
10	Data	0101	CF-Ack (no data)
10	Data	0110	CF-Poll (no data)
10	Data	0111	CF-Ack + CF-Poll (no data)
10	Data	1000–1111	Reserved
11	Reserved	0000–1111	Reserved

Clicky to visualise your network

- Start click with a port of your choosing
  - click -p 8080 test-wifi.click
- From a local machine you can connect clicky to this port
  - Clicky -p 8080
- From a remote machine you can connect clicky to this port
  - Clicky -p 172.20.1.11:8080
- Now you can see visually what's happening in your click graph
- Hint: Use the counter element throughput your code and then make use of clicky to debug by checking the count of packets through your graph

How to write your own element

- **Will create a simple example element**
  - 1 input, 1 output, Push
  - Configures a packet size threshold
  - If Larger: drop packet

Writing your own element header

- **Necessary in the header:**
  - Macros:
    - Include-guards
    - Click element macros
  - Include click/element.hh
  - The class declaration...
  - Contains 3 special methods
    - `const char *class_name() const`
    - `const char *port_count() const`
    - `const char *processing() const`

Writing your own element source

- **Necessary in the source file:**
  - Include click/config.hh first!!
  - Macros:
    - CLICK_DECLS
    - CLICK_ENDDECLS
    - EXPORT_ELEMENT
  - Implementations of the methods

Writing your own element source

- **To compile your element**
  - Better to create a new folder in elements
    - e.g. mkdir /home/click-git/elements/davidstuff
  - Add to click-git/configure

```
# Check whether --enable-davidstuff was given.
if test "${enable_davidstuff+set}" = set; then
  enableval=$enable_davidstuff;
else
  enable_davidstuff=NO
fi
test "x$enable_all_elements" = xyes -a "x$enable_davidstuff" = xNO && enable_davidstuff=yes
if test "x$enable_auction" = xyes; then
    :
fi
```

  - Now put all code in elements/davidstuff
  - Compile with
    - ./configure --enable-wifi --enable-local --enable-radio –enable-davidstuff
    - make elemlist
    - make

Writing your own element source

- **Userland click will now be in**
  - click-git/userland/click
- **To test your click .click file use**
  - click-git/userland/click yourfile.click
- **When you are finally happy with your click element**
  - Make install
    - This will move click to /usr/local/sbin and you won't have to supply a path anymore you can simply type
      - Click yourfile.click

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
```

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;
}

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)
```

What’s in a name

- **To avoid confusion, recommend:**
  - 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

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;
}
```

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
}
```

Think of operator overloading

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;
   }
}
```

Portcount

- **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
- **Or more complicated:**
  - + "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 and 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: cp_va_kparse

- **Call this function on**
  - the configuration (conf)
  - the element (this)
  - the errorhandler (errh)
  - an argument list
  - a closing mark (cpEnd)
- **Check the return value**
  - 0: all parsing went fine
  - Negative: problems detected, configure should return -1
- **E.g:**

```
int MyElement::configure(Vector<String> &conf, ErrorHandler *errh) {
      String data; uint32_t limit = 0; bool stop = false;
      if (cp_va_kparse(conf, this, errh,
                "DATA", cpkP+cpkM, cpString, &data,
                "LIMIT", cpkP, cpUnsigned, &limit,
                "STOP", 0, cpBool, &stop,
                cpEnd) < 0)   // always terminated by cpEnd
          return -1;
      ... }
```

Cp_va_kparse: the argument list

- **Argument name**
  - Type: const char *
  - Example: "DATA".
- **Parse flags**
  - Type: int
  - Zero or sum of cpkP, cpkM, and cpkC.
- **If the parse flags contain cpkC, then a confirmation flag comes next**
  - Type: bool *
  - This flag is set to true if an argument successfully matched the item and false if not.
- **Argument type: Defines the type of argument read from the configuration string**
  - Type: CpVaParseCmd
  - Example: cpString, cpIPAddress, cpInteger
- **Optional parse parameters**
  - Determined by the argument type
  - For example, cpUnsignedReal2 takes a parse parameter that defines how many bits of fraction are needed.
- **Result storage:**
  - Determined by the argument type

Cp_va_kparse: parse flags

- cpkN (=0): default, no special requirements
- cpkM: Mandatory argument
- cpkP: Positionally specified argument
- cpkC: Confirmation of presence needed
- cpkD: Deprecated argument
- To combine just sum them
  - cpkD+cpkC

Cp_va_kparse: example

-
  ```
  int MyElement2::configure(Vector<String>
  &conf, ErrorHandler *errh) {
      bool p_given; uint32_t p = 0x10000;
      IPAddress addr, mask;
      if (cp_va_kparse(conf, this, errh,
        "P", cpkC, &p_given, cpUnsignedReal2,
                          16, &p,
        "NETWORK", 0, cpIPPrefix, &addr,
                     &mask,
        cpEnd) < 0)
              return -1;
         ... }
  ```

Cp_va_kparse: example (2)

- `cp_va_kparse(conf, this, errh,`
  `"P", cpkC, &p_given, cpUnsigned, &p,`
  `"NETWORK", 0, cpIPAddress, &addr, &mask,`
  `cpEnd)`
- *Will it match*
  - *P 5, NETWORK 192.168.0.3*
  - *NETWORK 1.2.3.4, P5*
  - *P 5*
  - *NETWORK 192.168.0.3*
  - *(nothing)*
- *How about*

```
cp_va_kparse(conf, this, errh,
 "P", cpkC, &p_given, cpUnsigned, &p,
 "NETWORK", cpkM, cpIPAddress, &addr, &mask,
 cpEnd)
```

Parsing configurations: elements

- Elements might need other elements
  - Pass them in the configuration
  - Check their name and type
  - Calling public methods and accessing public members is possible
- Click script
  - `SimpleElement(IPRouteTable);`
    or

```
myIpRouteTable::IPRouteTable;
SimpleElement(myIpRouteTable);
```

Parsing configurations: elements (2)

- Add an element to the header

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

- Use the element in the C++ code

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

Parsing configurations: elements (3)

Check and configure the element in the configure function:

```
int ElementUser::configure(Vector<String> &conf,
  ErrorHandler *errh){

  Element* tempUsedElement;
  int res = cp_va_kparse(conf, this, errh, “ELEMENT”, 0,
  cpElement, &tempUsedElement, cpEnd);
  if(res < 0) return res; // parsing failed

  if (!(used =(UsedElement *) tempUsedElement-
  >cast(“UsedElement"))){
  return errh->error("Supplied element is not a valid
  UsedElement element");
  }

}
```

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 (schedule actions)
- **Additional functions**
  - Manipulate strings
  - Manipulate packets
  - E.g. click_gettimeofday(struct timeval * tv)

The Click STL

- **Overview of the most important types**
  - Vector
  - HashMap (will become HashContainer)
  - String

Click STL: Vector

- 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
```

Click STL: Vector (2)

- Add things to it:
  - `myvector.push_back(some_thing);`
- Use iterators to walk over it

```
for (SomeThingVector::const_iterator i =
  myvector.begin(); i.live(); i++) {
   doSomeThingWith(i.value());
}
```

- And remove things with iterators
  - `myyvector.erase(i);`
- Or pop it as a stack/heap
  - `myvector.pop_front(); myvector.pop_back();`

Click STL: example for HashMap

```
#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
```

Click STL: example for HashMap (2)

```
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
    …
}
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());`

Creating and destroying packets

- 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
};
```

Creating and destroying packets (2)

- **Provide headroom and tailroom to Packet::make(unsigned headroom, const unsigned char* data, unsigned len, unsigned tailroom):**

```
int tailroom = 0;
int packetsize = sizeof(MyPacketFormat);
int headroom = sizeof(click_ip)+sizeof(click_udp)
+sizeof(click_eth);
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()**
  - Frees your memory!

Processing packets

- 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 *) (packet->data()+offset);
```

- Use the format to read from and write to

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

- Only write to writable packets

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

Manipulating packets

- **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**

Packet headers

- **Get IP header:**
  - `packet->ip_header();`
- **Set IP header of length `len`:**
  - `packet->set_ip_header(const click_ip* header, unsigned len);`
- **Both operations require header annotations, set by the MarkIPHeader element!**
- **Similar operations exist for TCP and UDP headers**

Timers (simple, without extra data)

- 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_ms(1000);
     return 0;
}
void MyElement ::run_timer(Timer* timer){
     click_chatter("we are now 1 second later");
     timer.schedule_after_ms(1000);
}
```

Handlers: basics

- Like function calls to an element
- ReadHandler
  - Request a value from an element
- Or WriteHandler
  - Pass a string to an element
  - You can parse the string with cp_va_parse!
- There is no ReadWriteHandler
  - You can’t call a ReadHandler with arguments
- Can be called from other elements or through socket
  - Take a look at Pokehandlers!

Handlers: calling

- **Start click**
  - `click -p <port_nr> <click_script>`
  - `click -p 10000 somescript.click`
- **Connect to click with telnet**
  - `telnet localhost 10000`
  - `read <elementname>.<handlername>`
    - `read rt.table`
  - `write <elementname>.<handlername> <values>`
    - `write arptable.insert 00:50:BA:85:84:B1 10.0.1.2`

Handlers: Write implementation

-
  ```
  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(cp_va_kparse(conf, me, errh, …, cpEnd) < 0) return -1;
      me->doSomethingWithParsed(…);
      return 0;
  }
  ```

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

Handlers: Read implementation

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

Sources of documentation

- **Click website: http://www.read.cs.ucla.edu/click/**
  - + Element documentation (by name or category)
  - + Programming Concepts
  - + Doxygen documentation (‘Internals documentation’)
- **Click thesis Eric Kohler**
  - + Comprehensive documentation of every concept
    - – Introduction
    - – Architecture: elements, packets, connections, push and pull, element implementation
    - – Language: syntax, configuration strings, compound elements
- **Click mailinglist: https://amsterdam.lcs.mit.edu/mailman/listinfo/click**
  - + Large source of information, mainly for Click developers

Tutorial 1

- **Write a simple click script which:**
  - Reflects all icmp traffic on any of its interfaces
  - Forwards all icmp traffic from the incoming interface to all other interfaces

Tutorial 2

- **Write a simple script which using a raw wireless interface**
  - Generates 802.11 Beacons every 5 seconds
  - Receives 802.11 Beacons and sends a packet out the Ethernet interface containing the Beacon information

Tutorial 3

- **Build a wireless scanner which scans all channels and reports the SSID's that were seen.**
  - You may or may not need a new element

Tutorial 4

- **Write an element which reads the date on one node and sends the date information in a custom packet to another node.**
  - You will need to create your own packet structure
  - Allow the time zone (GMT+x hours) to be sent to the element using a handler – then change the time based on the time zone