Click Under the Hood

The following page explains how Click actually works under the hood. The explanation is initially based on single-threaded, user-level click; we discuss multi-threaded and kernel mode Click in later sections of this document. Also note that line numbers might differ slightly depending on which source version you have.

This document assumes you have read the Click SOSP paper or the Click thesis. Three dots in code excerpts ("...") denote that irrelevant lines of code have been omitted. Numbers beginning with a plus ("+") sign denote line numbers.

1. Overview

This section gives a brief introduction to the internals of Click that is necessary in order to understand the sections that follow it.

1.1 Overall Structure

A Click router (where a router could be a firewall, NAT, or any other packet processing unit) contains a class called Master (lib/master.cc and include/click/master.hh. This master contains, among other things, two other classes. The first of these is Router (lib/router.cc and include/router.hh) which holds information about the actual configuration once it has been parsed from the .click file. The second is a set of RouterThread instances (lib/routerthread.cc and include/click/routerthread.hh. As their name suggests, these constitute the running threads of a Click router (in the case of single-threaded Click only a single thread exists).

Click schedules the router's CPU with a task queue (depending on the values of some #DEFINEs the queue of tasks may be implemented as a heap or a linked list). Each router thread runs a loop that processes the task queue one element at a time. The task queue is scheduled with the flexible and lightweight stride scheduling algorithm. Tasks are simply elements that would like special access to CPU time. Thus, elements are Click's unit of CPU scheduling as well as its unit of packet processing. An element should be on the task queue if it frequently initiates push or pull requests without receiving a corresponding request. For example, an element that polls a device driver should be placed on the task queue; when run, it would remove packets from the driver and push them into the configuration. However, most elements are never placed on the task queue. They are implicitly scheduled when their push or pull methods are called. Once an element is scheduled, either explicitly or implicitly, it can initiate an arbitrary sequence of push and pull requests, thus implicitly scheduling other elements.

Finally, it is worth pointing out that the basic unit of communication between elements is the Packet, defined in include/click/packet.hh. This is the object that most element execution methods expect to receive and/or output.

1.2 Element Scheduling

In this section we give a more detailed introduction to how elements are scheduled to run in Click. As mentioned, Click uses tasks (essentially elements) as its basic unit of scheduling; tasks are defined in include/click/task.hh. Not all elements need to be scheduled: many of them will be implicitly scheduled when a scheduled element calls them. Exactly which elements are scheduled and which are implicitly called depends on the elements themselves. To make the discussion more concrete, imagine the following simple Click

configuration file:

```
FromDevice("eth0")
        -> Strip(14)
        -> CheckIPHeader(BADSRC 18.26.4.255 2.255.255.255 1.255.255.255)
        -> Queue
        -> ToDevice("eth0");
```

In this case, FromDevice's initialize method uses the ScheduleInfo class to periodically schedule itself. Strip, CheckIPHeader and Queue do not do so, as they are all implicitly called when FromDevice executes. In greater detail, when FromDevice is scheduled to run, its selected method is called which in turn makes a call to output(0).push(p). Since in our case this element is connected to Strip, this call essentially causes the simple_action callback of Strip to run. This method's returning of a Packet object triggers execution of the callback of the next element down the line in the configuration, in this case CheckIPHeader. This element's callback is also simple_action, which once again causes execution of Queue's callback method (enq) to run when it returns a packet.

At this point the execution chain finishes and control returns to the scheduler. At the other end of the configuration, ToDevice also schedules itself to run periodically. When its callback run_task runs, it pulls a packet from the queue, sends it out a physical interface and returns control back to the scheduler. Periodic scheduling is not the only way for elements to schedule themselves: InfiniteSource, for instance, schedules itself once during initialization, and re-schedules itself each time its callback is executed (up to the number of packets it's supposed to generate). As can be seen, different elements have different callbacks (although there's a limited number of them); we will explain how these get used later on in this document.

1.3 Source Tree Structure

The listing below gives an overview of the Click source tree. Note that the listing is not complete, limiting itself to showing directories relevant to this document's discussion.

```
.
|-- conf                    // various Click configuration files
|-- drivers                 // drivers with Click extensions (for polling)
|-- elements                // the actual Click elements
|-- etc                     // patch and other miscellaneous files
|-- include
|   |-- click               // base Click header files
|-- lib                     // base Click cpp files
|-- linuxmodule             // Click linux kernel mode files
`-- userlevel               // Click user level files
```

2. When Click Runs

The process begins when the install binary is executed (click for user-level, click-install for the kernel-based version). The source for this is found in userlevel/click.cc, and the main is found in line 421; here's an excerpt:

```
static Router *router;

...
```

```
int
main(int argc, char **argv)
{
  click_static_initialize();

  ...

  router = parse_configuration(router_file, file_is_expr, false, errh);

  ...

  router->master()->thread(0)->driver();
}
```

In this abbreviated excerpt we can see that running a Click router consists of doing a bit of initialization, parsing the given router configuration, and running the router thread (the 0 in thread(0) is because we're running single-threaded Click; the multi-threaded version would look slightly different). We will now explain each of these stages in greater detail.

2.1 Initialization

The click_static_initialize function, found in userlevel/click.cc +442 is simple:

```
    ...

    cp_va_static_initialize();

    ...

    Router::static_initialize();

    ...

    click_export_elements();
```

The first function initializes element parameter types and the second one router parameters. The third call warrants a closer look, as it takes care of populating a data structure with the names and other information about all available elements. The function is automatically defined during the build process in the file userlevel/elements.cc:

```
#include <click/config.h>
#include <click/package.hh>
#include "../elements/standard/delayshaper.hh"
#include "../elements/threads/spinlockrelease.hh"
#include "../elements/test/listtest.hh"

 ... (many other elements) ...

beetlemonkey(uintptr_t heywood)
{
  switch (heywood) {
   case 0: return new AdjustTimestamp;
   case 1: return new AggregateCounter;
```

```
   case 2: return new AggregatePacketCounter;

   ... (many other elements) ...

   case 259: return new ToRawSocket;
   case 260: return new ToSocket;
   case 261: return new UMLSwitch;
   default: return 0;
  }
}

void
click_export_elements()
{
 (void) click_add_element_type_stable("AdjustTimestamp",beetlemonkey,0);

 ... (many other elements) ...

 (void) click_add_element_type_stable("ToRawSocket", beetlemonkey, 259);
 (void) click_add_element_type_stable("ToSocket", beetlemonkey, 260);
 (void) click_add_element_type_stable("UMLSwitch", beetlemonkey, 261);
 CLICK_DMALLOC_REG("nXXX");
}
```

As shown, the file defines the elements that are available, includes their header files and assigns numbers to each of the elements. In turn, the function click_add_element_type_stable is called; this function is once again defined in lib/driver.cc, +399:

```
extern "C" int
click_add_element_type_stable(const char *ename,
                              Element *(*func)(uintptr_t),
                              uintptr_t thunk)
{
    assert(ename);
    if (Lexer *l = click_lexer())
        return l->add_element_type(String::make_stable(ename),
                                   func,
                                   thunk);
    else
        return -99;
}
```

This function in turn adds the element to the lexer (used to parse Click configuration files) by calling add_element_type with the element's name, a pointer to its class, and the element code "type" (an integer). The function is itself defined in lib/lexer.cc +836:

```
int
Lexer::add_element_type(const String &name, ElementFactory factory, uintptr_t thunk,
                        bool scoped)
{
  int tid = _element_types.size();
  _element_types.push_back(ElementType());
  _element_types[tid].factory = factory;
  _element_types[tid].thunk = thunk;
  _element_types[tid].name = name;
  _element_types[tid].next = _last_element_type |
                             (scoped ? (int)ET_SCOPED : 0);
```

```
  if (name)
    _element_type_map.set(name, tid);
  _last_element_type = tid;
  return tid;
}
```

where the relevant data structures are defined in include/click/lexer.hh, +153, as:

```
struct ElementType {
  ElementFactory factory;
  uintptr_t thunk;
  String name;
  int next;
};

HashTable<String, int> _element_type_map;

Vector<ElementType> _element_types;
```

At the end of all of these calls the lexer has a data structure called _element_types containing element names, (integer) types and factories. In addition, the lexer also keeps a separate hash _element_types mapping element names to their integer types.

2.2 Configuration Parsing

The process begins once again in the main of userlevel/click.cc (+588) with a call to parse_configuration (+303):

```
static Router *
parse_configuration(const String &text, bool text_is_expr, bool hotswap,
                    ErrorHandler *errh)
{
  Master *master = (router ? router->master() : new Master(nthreads));
  Router *r = click_read_router(text, text_is_expr, errh, false, master);
  if (!r)
    return 0;

  ...

  return r;
```

The function is straightforward, receiving a Click configuration in the form of a string and delegating most of the work to click_read_router, defined in lib/driver.cc +481:

```
Router *
click_read_router(String filename,
                  bool is_expr,
                  ErrorHandler *errh,
                  bool initialize,
                  Master *master)
{
    ...
```

```
    // read file
    String config_str;
    if (is_expr) {
        config_str = filename;
        filename = "config";
    } else {
        config_str = file_string(filename, errh);
        if (!filename || filename == "-")
            filename = "<stdin>";
    }

    ...

    RequireLexerExtra lextra(&archive);
    int cookie = l->begin_parse(config_str, filename, &lextra, errh);
    while (l->ystatement())
        /* do nothing */;
    Router *router = l->create_router(master ? master : new Master(1));
    l->end_parse(cookie);

    ...

    if (initialize)
        if (errh->nerrors() > before || router->initialize(errh) < 0) {
            delete router;
            return 0;
        }

    return router;
```

The function calls begin_parse on the lexer to initialize some basic variables and then calls ystatement (defined in lib/lexer.cc +1798), which calls yconnection (lib/lexer.cc +1435) and which, in turn, subsequently calls yelement (lib/lexer.cc +1163) on each of the elements:

```
bool
Lexer::yelement(Vector<int> &result, bool in_allowed)
{
  ...

  *resp = get_element(e->name,
                      e->type >= 0 ? e->type : e->decl_type,
                      e->configuration,
                      e->filename,
                      e->lineno);

  ...
```

We omit almost all of this function's implementation since its details are largely unimportant for our purposes, What is important is the call to get_element, whose implementation is in lib/lexer.cc +993:

```
int
Lexer::get_element(String name, int etype, const String &conf,
                   const String &filename, unsigned lineno)
{
  ...
```

```
  _c->_element_names.push_back(name);
  _c->_element_configurations.push_back(conf);
  if (!filename && !lineno) {
      _c->_element_filenames.push_back(_file._filename);
      _c->_element_linenos.push_back(_file._lineno);
  } else {
      _c->_element_filenames.push_back(filename);
      _c->_element_linenos.push_back(lineno);
  }
  _c->_elements.push_back(etype);
  _c->_element_nports[0].push_back(0);
  _c->_element_nports[1].push_back(0);
  return eid;
}
```

The related data structures are defined in include/click/lexer.hh, +170:

```
    HashTable<String, int> _element_map;
    Compound *_c;
```

with Compound being defined in lib/lexer.cc +117:

```
class Lexer::Compound : public Element { public:
  ...

 private:
  Vector<int> _elements;
  Vector<String> _element_names;
  Vector<String> _element_configurations;
  Vector<String> _element_filenames;
  Vector<unsigned> _element_linenos;
  Vector<int> _element_nports[2];
  Vector<Router::Connection> _conn;

  friend class Lexer;
}
```

As shown, get_element populates _c and _element_map with information about the actual elements in the configuration file. Finally, recall from the listing of click_read_router the call to create_router, defined in lib/lexer.cc +1933:

```
Router *
Lexer::create_router(Master *master)
{
  Router *router = new Router(_file._big_string, master);
  if (!router)
    return 0;

  ...

  // add elements to router
  Vector<int> router_id;
  for (int i = 0; i < _c->_elements.size(); i++) {
    int etype = _c->_elements[i];
    if (etype == TUNNEL_TYPE)
```

```
      router_id.push_back(-1);
    else if (Element *e = (*_element_types[etype].factory)
                          (_element_types[etype].thunk)) {
      int ei = router->add_element(e,
                                   _c->_element_names[i],
                                   _c->_element_configurations[i],
                                   _c->_element_filenames[i],
                                   _c->_element_linenos[i]);
      router_id.push_back(ei);
    } else {
      _errh->lerror(_c->element_landmark(i),
                    "failed to create element %<%s%>",
                    _c->_element_names[i].c_str());
      router_id.push_back(-1);
    }
  }

  ...

  // expand connections to router
  int pre_expanded_nc = _c->_conn.size();
  for (int i = 0; i < pre_expanded_nc; i++) {
    int fromi = router_id[ _c->_conn[i][1].idx ];
    int toi = router_id[ _c->_conn[i][0].idx ];
    if (fromi < 0 || toi < 0)
      add_router_connections(i, router_id);
  }

  // use router element numbers
  for (Connection *cp = _c->_conn.begin(); cp != _c->_conn.end(); ++cp) {
    (*cp)[0].idx = router_id[(*cp)[0].idx];
    (*cp)[1].idx = router_id[(*cp)[1].idx];
  }

  // sort and add connections to router
  click_qsort(_c->_conn.begin(), _c->_conn.size());
  for (Connection *cp = _c->_conn.begin(); cp != _c->_conn.end(); ++cp)
    if ((*cp)[0].idx >= 0 && (*cp)[1].idx >= 0)
      router->add_connection((*cp)[1].idx,
                             (*cp)[1].port,
                             (*cp)[0].idx,
                             (*cp)[0].port);

  // add requirements to router
  for (int i = 0; i < _requirements.size(); i += 2)
      router->add_requirement(_requirements[i], _requirements[i+1]);

  return router;
}
```

The main thing to take away from this listing is that a new Router variable is instantiated and initialized with the string of the configuration file and the Master instance given to the function (recall that Master contains a Router instance and potentially several RouterThread instances). Another equally important point is that it is here that elements are actually instantiated, specifically in the call:

```
Element *e =(*_element_types[etype].factory)(_element_types[etype].thunk)
```

The first part of the call, (*_element_types[etype].factory), resolves to the function beetlemonkey (recall listing for userlevel/elements.cc), to which we essentially pass as a parameter an integer representing the element type (this corresponds to (_element_types[etype].thunk)). The function beetlemonkey then returns a new element depending on the type given as a parameter. Going back to create_router, the function also places calls to add_element and add_connection, which populate private data member variables of Router regarding the router's configuration (the following listing is from include/click/router.hh, +227):

```
Vector<Element*> _elements;
Vector<String> _element_names;
Vector<String> _element_configurations;
Vector<String> _element_landmarks;
Vector<Connection> _conn;
Vector<String> _requirements;
```

The last step before actually running the router is to initialize the elements themselves. To do so, recall the bottom of the listing click_read_router: one of the last lines makes a call to router->initialize, defined in lib/router.cc +1043:

```
int
Router::initialize(ErrorHandler *errh)
{
  ...

  for (int ord = 0; ord < _elements.size(); ord++) {
    if ((r = _elements[i]->configure(conf, &cerrh)) < 0) {
      element_stage[i] = Element::CLEANUP_CONFIGURE_FAILED;
      all_ok = false;
      if (cerrh.nerrors() == before) {
        if (r == -ENOMEM)
          cerrh.error("out of memory");
        else
          cerrh.error("unspecified error");
      }
    }
    else
      element_stage[i] = Element::CLEANUP_CONFIGURED;
  }

  ...

  if (all_ok) {
    _state = ROUTER_PREINITIALIZE;
    initialize_handlers(true, true);
    for (int ord = 0; all_ok && ord < _elements.size(); ord++) {

    ...

      if (_elements[i]->initialize(&cerrh) >= 0)
        element_stage[i] = Element::CLEANUP_INITIALIZED;
      else {
        if (cerrh.nerrors() == before && !_elements[i]->cast("Error"))
          cerrh.error("unspecified error");
        element_stage[i] = Element::CLEANUP_INITIALIZE_FAILED;
        all_ok = false;
      }
    }
```

```
}
```

The reason this code fragment is important is because it takes care of calling each element's configure and initialize methods, in that order. Elements use the first method to receive configuration information (i.e., the parameters given to an element in a Click configuration file). The second is used to initialize the element, including whether it should schedule itself to run or not (recall the section on scheduling at the beginning of this document). With all of this in place, the Master object, and the Router object it contains, have all the information needed to start running the router.

2.3 Router Execution

The process begins in userlevel/click.cc's main with a call to router->master()->thread(0)->driver() (in the case of single-threaded Click), which is implemented in lib/routerthread.cc +513:

```
void
RouterThread::driver()
{
  const volatile int * const stopper = _master->stopper_ptr();
  int iter = 0;

  ...

  driver_loop:

    if (*stopper == 0) {
        // run occasional tasks: timers, select, etc.
        iter++;

        _master->run_signals(this);

        if ((iter % _iters_per_os) == 0)
            run_os();

        bool run_timers = (iter % _master->timer_stride()) == 0;
        if (run_timers) {
            _master->run_timers(this);
        }
    }

    // run task requests (1)
    if (_pending_head)    // uintptr_t, from include/master.hh +103
        process_pending();

    run_tasks(_tasks_per_iter); // _tasks_per_item set to 128 on linux

    // check to see if driver is stopped
    if (*stopper > 0) {
        driver_unlock_tasks();
        bool b = _master->check_driver();
        driver_lock_tasks();
        if (!b)
            goto finish_driver;
    }
```

```
    goto driver_loop;

  finish_driver:
    driver_unlock_tasks();
}
```

The basic idea of this function is simple enough: infinitely stay in the driver_loop until the function _master->check_driver(), which returns whether there are any routers running in the master, returns false. Ignoring calls to yield execution to the OS (run_os) or running timers (_master->run_timers(this)), most of the work is carried out by the run_tasks call, defined in lib/routerthread.cc + 342:

```
inline void
RouterThread::run_tasks(int ntasks)
{
  ...

  Task *t;
  for (; ntasks >= 0; --ntasks) {
    t = task_begin();
    if (t == this)
      break;

      t->fast_remove_from_scheduled_list();

      ...


      // 21.May.2007: Always set the current thread's pass to the current
      // task's pass, to avoid problems when fast_reschedule() interacts
      // with fast_schedule() (passes got out of sync).
      _pass = t->_pass;

      t->_status.is_scheduled = false;
      t->fire();

      ...
  }
}
```

This function runs a set number of tasks (set to 128 on linux). In each iteration, it removes the first task from the list, and sets (in fast_remove_from_scheduled_list) the next task as the current task to run.It then calls the task's fire method, defined in include/click /task.hh +554 :

```
inline void
Task::fire()
{
  ...

  if (!_hook)
    (void) ((Element*)_thunk)->run_task(this);
  else
    (void) _hook(this, _thunk);

  ...
}
```

with the various variables shown having the following definitions in include/click/task.hh:

```
  // defines TaskCallback as a pointer to a function returning bool
  // and taking Task *, void * as parameters
  typedef bool (*TaskCallback)(Task *, void *);
  TaskCallback _hook;
  void *_thunk;
```

In short, if an element's callback is run_task, fire directly calls that method, passing the element (the task) as a parameter. If, on the other hand, the element uses a different callback, the function pointer _thunk, representing the callback, is called with the task and the _thunk pointer as parameters.

How are these callbacks initialized? This depends on the element in question, but just to give an example consider InfiniteSource, whose definition can be found in elements/standard/infinitesource.cc. Its constructor (+30) is defined as follows:

```
InfiniteSource::InfiniteSource()
  : _packet(0), _task(this)
{
}
```

The line _task(this) causes a call to one of the Task constructors found in include/click/task.hh +289:

```
inline
Task::Task(Element* e)
    : _prev(0), _next(0),
      _should_be_scheduled(false), _should_be_strong_unscheduled(false),
      _pass(0), _stride(0), _tickets(-1),
      _hook(0), _thunk(e),
      _thread(0), _home_thread_id(-1),
      _owner(0), _pending_nextptr(0)
{
}
```

The important part here is _thunk(e): this void * pointer gets set equal to the element, and it's precisely this pointer which will be used to run the element's run_task callback when this task/element is scheduled (recall the listing of the fire method above).

So far we have explained how schedulable elements schedule themselves and how their callbacks are called. We have yet to describe how implicitly-scheduled elements (those than run when scheduled elements call them) are actually executed. The details can vary depending on the Click configuration in use and the elements in it, but the basic concepts should be the same irrespectively. Given this, it helps to use a sample Click configuration to drive the rest of the explanation (this listing comes from conf/test.click):

```
InfiniteSource(DATA \<00 00 c0 ae 67 ef  00 00 00 00 00 00  08 00
45 00 00 28  00 00 00 00  40 11 77 c3  01 00 00 01
02 00 00 02  13 69 13 69  00 14 d6 41  55 44 50 20
70 61 63 6b  65 74 21 0a>, LIMIT 5, STOP true)
        -> Strip(14)
        -> Align(4, 0)    // in case we're not on x86
        -> CheckIPHeader(BADSRC 18.26.4.255 2.255.255.255 1.255.255.255)
        -> Print(ok)
```

```
        -> Discard;
```

The actual parameters passed to the various elements are irrelevant for our purposes; this configuration simply generates 5 packets, prints their contents and discards them. Note that for the explanation below all listings for a particular element refer to the .cc file found under the elements directory and its sub-directories.

Before we begin, it is worth pointing out that by and large Click uses three callbacks for implicitly-scheduled elements, including default implementations; these are defined in lib/element.cc:

```
void
Element::push(int port, Packet *p)
{
    p = simple_action(p);
    if (p)
        output(port).push(p);
}

Packet *
Element::pull(int port)
{
    Packet *p = input(port).pull();
    if (p)
        p = simple_action(p);
    return p;
}

Packet *
Element::simple_action(Packet *p)
{
    return p;
}
```

The process begins as explained before, with InfiniteSource using ScheduleInfo to periodically schedule itself. When it is scheduled, its run_task method runs (+109):

```
bool
InfiniteSource::run_task(Task *)
{
    if (!_active || !_nonfull_signal)
        return false;
    int n = _burstsize;
    if (_limit >= 0 && _count + n >= (ucounter_t) _limit)
        n = (_count > (ucounter_t) _limit ? 0 : _limit - _count);
    for (int i = 0; i < n; i++) {
        Packet *p = _packet->clone();
        if (_timestamp)
            p->timestamp_anno().assign_now();
        output(0).push(p);
    }
    _count += n;
    if (n > 0)
        _task.fast_reschedule();
    else if (_stop && _limit >= 0 && _count >= (ucounter_t) _limit)
        router()->please_stop_driver();
```

```
    return n > 0;
}
```

As can be seen, this method calls output(0).push(p). In this case, the next element down the line is Strip. Since its class doesn't have a push method implemented, the default method in the Element superclass is called instead, which in turns calls Strip::simple_action and output(port).push(p).

The next element is Align. This time, Align does implement a push method, so it is called:

```
void
Align::push(int, Packet *p)
{
  output(0).push(smaction(p));
}
```

Ignoring the details of smaction, this element then calls CheckIPHeader's push method. Since it doesn't define one, once again the default provided by Element is used. The next element is Print, which once again relies on the default push method. Finally, the Discard element provides its own push method:

```
void
Discard::push(int, Packet *p)
{
    _count++;
    p->kill();
}
```

This method simply kills the packet and releases any memory allocated to it. At this point the execution chain started by InfiniteSource's output(0).push(p) is finished, and the process will repeat 4 more times corresponding to the remaining packets to generate (see for loop in the listing of the element's run_task method).

Once the remaining packets are generated and processed, run_task returns control to the scheduler. In most configurations the scheduler would then schedule the next task. In this particular one, no more tasks are left to run, so the router process finishes and quits.

3. Kernel Mode

In this section we describe how Click works when run in kernel mode. The process begins with a different executable called click-install. This command can take a number of arguments, but in the simplest case takes the name of a file containing a Click configuration, as in the user-level case. The command is defined in tools/click-install/click-install.cc, and its main looks as follows (+301):

```
int
main(int argc, char **argv)
{
  click_static_initialize();

  ...

  RouterT *r = read_router(router_file, file_is_expr, nop_errh);
```

```
...

// install Click module if required
if (access(clickfs_packages.c_str(), F_OK) < 0) {
  // find and install proclikefs.o
  StringMap modules(-1);
  if (read_active_modules(modules, errh) && modules["proclikefs"] < 0) {
    String proclikefs_o =
      clickpath_find_file("proclikefs.ko", "lib", CLICK_LIBDIR, errh);
    install_module(proclikefs_o, String(), errh);
  }
  String click_o =
    clickpath_find_file("click.ko", "lib", CLICK_LIBDIR, errh);

  // install it in the kernel
  String options;
  if (threads > 1)
    options += "threads=" + String(threads);
  if (greedy)
      options += " greedy=1";
  if (!accessible)
      options += " accessible=0";
  if (uid != 0)
      options += " uid=" + String(uid);
  if (gid != 0)
      options += " gid=" + String(gid);
  if (cpu != -1)
      options += " cpu=" + String(cpu);
  install_module(click_o, options, errh);

  // make clickfs_prefix directory if required
  if (access(clickfs_dir.c_str(), F_OK) < 0 && errno == ENOENT) {
    if (mkdir(clickfs_dir.c_str(), 0777) < 0)
      errh->fatal("cannot make directory %s: %s",
                  clickfs_dir.c_str(),
                  strerror(errno));
  }

  // mount Click file system
  int mount_retval = mount("none", clickfs_dir.c_str(), "click", 0, 0);

  ...

String clickfs_config = clickfs_prefix + String("/config");

// write flattened configuration to CLICKFS/config
int exit_status = 0;
{
  String config_place = (hotswap ? clickfs_hotconfig : clickfs_config);
  int fd = open(config_place.c_str(), O_WRONLY | O_TRUNC);
  String config = r->configuration_string();
  int pos = 0;
  while (pos < config.length()) {
    ssize_t written = write(fd,
                            config.data() + pos,
                            config.length() - pos);
    if (written >= 0)
      pos += written;
```

```
      else if (errno != EAGAIN && errno != EINTR)
        errh->fatal("%s: %s", config_place.c_str(), strerror(errno));
    }
    int retval = close(fd);
  }
}
```

As with userlevel Click, this function begins by calling the click_static_initialize method that initializes some basic variables such as parameter types, and a list of which elements are available. Also like in user-level Click, the function then parses the router configuration. Here's where things start to differ. In order to run in kernel mode, Click needs to be loaded as a kernel module. In addition, Click in kernel mode uses entries in the Linux /proc filesystem in order to, among others, implement the elements' read and write handlers. This system is implemented as another kernel module called proclikefs. Starting a router in kernel mode consists, then, of inserting the proclikefs kernel module (via the install_module method), mounting that filesystem (see call to mount above) and inserting the actual Click kernel module (again, via the install_module method). The install_module method is found at +196:

```
static void
install_module(const String &filename, const String &options,
               ErrorHandler *errh)
{
  String cmdline = "/sbin/insmod ";
  if (output_map)
    cmdline += "-m ";
  cmdline += filename;
  if (options)
    cmdline += " " + options;
  int retval = system(cmdline.c_str());
  if (retval != 0)
    errh->fatal("'%s' failed", cmdline.c_str());
}
```

Calling this method on our setup results in the system calls /sbin/insmod /usr/local/lib/proclikefs.ko and /sbin/insmod /usr/local /lib/click.ko. The proclikefs.ko module is defined in linuxmodule/proclikefs.c, and the actual Click filesystem in linuxmodule/clickfs.cc, though we will skip any detailed explanation of how it works. The Linux kernel module main program is defined in linuxmodule/module.cc, with execution beginning, as with all Linux kernel modules, with the init_module function (+279):

```
extern "C" int
init_module()
{
    // C++ static initializers
    NameInfo::static_initialize();
    cp_va_static_initialize();

    // default provisions
    Router::static_initialize();
    NotifierSignal::static_initialize();

    // thread manager, sk_buff manager, config manager
    click_init_sched(ErrorHandler::default_handler());
    skbmgr_init();
    click_init_config();

    // global handlers
```

```
    Router::add_read_handler(0,
                             "packages",
                             read_global,
                             (void *) (intptr_t) h_packages);
    Router::add_read_handler(0,
                             "meminfo",
                              read_global,
                             (void *) (intptr_t) h_meminfo);
    Router::add_read_handler(0,
                             "cycles",
                             read_global,
                             (void *) (intptr_t) h_cycles);
    Router::add_read_handler(0,
                             "errors",
                             read_errors,
                             0,
                             HANDLER_DIRECT);
    Router::add_read_handler(0,
                             "messages",
                             read_messages,
                             0,
                             HANDLER_DIRECT);

    ...

    init_clickfs();

    return 0;
}
```

Here once again the process starts with cp_va_static_initialize. Note that this call had been done at the beginning of this section already. The difference is that the first time it was done by the click-install tool, while this time it is being run by the Click kernel module. The next line worth pointing out is the call to click_init_sched, defined in linuxmodule/sched.cc +340:

```
void
click_init_sched(ErrorHandler *errh)
{
    ...

    click_master = new Master(1);

    for (int i = 0; i < click_master->nthreads(); i++) {
        RouterThread *thread = click_master->thread(i);
        pid_t pid = kernel_thread
            (click_sched, thread, CLONE_FS | CLONE_FILES | CLONE_SIGHAND);
        ...
    }

  ...

}
```

As shown, this function creates a new Master object and adds RouterThreads to it (only one in our case, since we have single-threaded Click). For each of the threads it creates a kernel thread by calling kernel_thread and setting click_sched as its callback function; this function is defined in linuxmodule/sched.cc +81:

```
static int
click_sched(void *thunk)
{
    RouterThread *rt = (RouterThread *)thunk;

    ...

    printk("<1>click: starting router thread pid %d (%p)\n",
           current->pid,
           rt);

    // add pid to thread list
    SOFT_SPIN_LOCK(&click_thread_lock);
    if (click_thread_tasks)
        click_thread_tasks->push_back(current);
    SPIN_UNLOCK(&click_thread_lock);

    // driver loop; does not return for a while
    rt->driver();

    // release master (preserved in click_init_sched)
    click_master->unuse();

    // remove pid from thread list
    SOFT_SPIN_LOCK(&click_thread_lock);
    if (click_thread_tasks)
        for (int i = 0; i < click_thread_tasks->size(); i++) {
            if ((*click_thread_tasks)[i] == current) {
                (*click_thread_tasks)[i] = click_thread_tasks->back();
                click_thread_tasks->pop_back();
                break;
            }
        }
    printk("<1>click: stopping router thread pid %d\n", current->pid);
    SPIN_UNLOCK(&click_thread_lock);

    return 0;
}
```

This function essentially invokes the thread's router and calls its driver function which actually runs the router (an explanation of the driver function appears in the user-level Click section above). When the router finishes, control returns and click_sched performs a bit of clean-up before returning.

So far we installed the click.ko and proclikefs modules, set up kernel threads and set them running. However, we have not yet told the router to actually install a particular Click configuration. For this we need to go back to the listing of click-install's main method. After the modules are inserted, the click-install tool takes care of writing the string representing the Click configuration to /proc/click/config.

When the entry is written to, its read handler gets called. Where is this handler set-up? The call to click_init_config in the listing of init_module above starts off the process; click_init_config is defined in linuxmodule/config.cc +171:

```
void
click_init_config()
{
    lexer = new Lexer;
```

```
    Router::add_read_handler(0, "classes", read_classes, 0);
    Router::add_write_handler(0, "config", write_config, 0);
    Router::add_write_handler(0, "hotconfig", write_config, (void *)1);
    Router::set_handler_flags(0, "config",
                              HANDLER_WRITE_UNLIMITED |
                              Handler::RAW |
                              Handler::NONEXCLUSIVE);
    Router::set_handler_flags(0, "hotconfig",
                              HANDLER_WRITE_UNLIMITED |
                              Handler::RAW |
                              Handler::NONEXCLUSIVE);

    click_config_generation = 1;
    current_config = new String;

    click_export_elements();
}
```

Of particular interest to this discussion is the line related to the "config" entry, for which the function write_config is set as its handler. This function is found in linuxmodule/config.cc +157:

```
static int
write_config(const String &s, Element *, void *thunk, ErrorHandler *)
{
    click_clear_error_log();
    int retval = (thunk ? hotswap_config(s) : swap_config(s));
    return retval;
}
```

As in this case thunk is 0 (see call to add_write_handler above), swap_config is called (linuxmodule/config.cc +118):

```
static int
swap_config(const String &s)
{
    kill_router();
    if (Router *router = parse_router(s)) {
        if (router->initialize(click_logged_errh) >= 0)
            router->activate(click_logged_errh);
        install_router(s, router);
        return (router->initialized() ? 0 : -EINVAL);
    } else {
        install_router(s, 0);
        return -EINVAL;
    }
}
```

Just as in the userlevel Click case, this function calls parse_router in order to parse the router configuration and return a Router object. The function then installs the router by calling install_router(s, router), found in linuxmodule/config.cc +94:

```
static void
install_router(const String &config, Router *r)
{
```

```
    click_config_generation++;
    click_router = r;
    if (click_router)
        click_router->use();
    *current_config = config;
}
```

with click_router defined in linuxmodule/module.cc, +37 as follows:

```
Router *click_router = 0;
```

If all of these operations were successful, the click-install tool finishes and exits, leaving the kernel threads to run the actual Click router. The click-uninstall tool can then be used to remove these Click threads from the kernel.

4. Multi-threaded Click

Note that this section refers to kernel-mode multi-threaded Click, multi-threading in user level is currently marked as experimental.

To enable multi-threading, the first thing that's needed is to add an option to the configure script before building Click:

```
./configure --enable-multithread[=N]
```

When running Click configurations, an extra parameter needs to be given to the click-install tool

```
click-install -t 8 test.click
```

where t specifies the number of desired threads. The relevant bits in linuxmodule/click-install.cc are shown below:

```
#define THREADS_OPT             309

static const Clp_Option options[] = {
#if FOR_LINUXMODULE
  { "threads", 't', THREADS_OPT, Clp_ValUnsigned, 0 },
#endif
};

int
main(int argc, char **argv)
{
  int threads = 1;

while (1) {
    int opt = Clp_Next(clp);
    switch (opt) {
#if FOR_LINUXMODULE
     case THREADS_OPT:
      threads = clp->val.u;
      if (threads < 1) {
        errh->error("must have at least one thread");
```

```
        goto bad_option;
      }
      break;
#endif
    }
}

#if FOR_LINUXMODULE
    String options;
    if (threads > 1)
      options += "threads=" + String(threads);
    install_module(click_o, options, errh);
}
```

The main function parses the "t" command-line parameter and adds it to a string of options which is passed to the Click kernel module when it is installed (essentially the install_module function generates a string consisting of /sbin/insmod /usr/local/lib/click.ko threads=N. From the discussion in the previous section we know that control will now pass to linuxmodule/module.cc's init_module, which will in turn call linuxmodule/sched.cc's click_init_sched, +340:

```
void
click_init_sched(ErrorHandler *errh)
{
#if HAVE_MULTITHREAD
    click_master = new Master(click_parm(CLICKPARM_THREADS));

    ...

    for (int i = 0; i < click_master->nthreads(); i++) {
        click_master->use();
        RouterThread *thread = click_master->thread(i);
        thread->set_greedy(greedy);
        pid_t pid = kernel_thread
            (click_sched, thread, CLONE_FS | CLONE_FILES | CLONE_SIGHAND);
        if (pid < 0) {
          errh->error("cannot create kernel thread for Click thread %i!",
                      i);
          click_master->unuse();
        }
    }
}
```

Note that only the parts relevant to multi-threading are shown above. This part is pretty simple, and limits itself to using the CLICKPARM_THREADS macro (defined in linuxmodule/moduleparm.h) to retrieve the number of threads given to the module as a parameter and using it as an argument to one of the Master constructors (lib/master.cc +62):

```
Vector<RouterThread*> _threads; // <-- from include/click/master.hh

Master::Master(int nthreads)
    : _routers(0)
{
    for (int tid = -1; tid < nthreads; tid++)
        _threads.push_back(new RouterThread(this, tid));
}
```

The Master object now has N RouterThreads, each with its own thread id. Going back to the listing of click_init_sched, the function now runs through the threads in Master and launches a kernel thread for each of them, setting click_sched as their callback function. This function is defined at +81 as:

```
static int
click_sched(void *thunk)
{
    RouterThread *rt = (RouterThread *)thunk;
#ifdef CONFIG_SMP
    int mycpu = click_parm(CLICKPARM_CPU);
    if (mycpu >= 0) {
        mycpu += rt->thread_id();
        if (mycpu < num_possible_cpus() && cpu_online(mycpu))
            set_cpus_allowed(current, cpumask_of_cpu(mycpu));
        else
            printk("<1>click: warning: cpu %d for thread %d offline\n",
                   mycpu,
                   rt->thread_id());
    }
# endif

    printk("<1>click: starting router thread pid %d (%p)\n",
           current->pid,
           rt);

    // add pid to thread list
    SOFT_SPIN_LOCK(&click_thread_lock);
    if (click_thread_tasks)
        click_thread_tasks->push_back(current);
    SPIN_UNLOCK(&click_thread_lock);

    // driver loop; does not return for a while
    rt->driver();

    // release master (preserved in click_init_sched)
    click_master->unuse();

    // remove pid from thread list
    SOFT_SPIN_LOCK(&click_thread_lock);
    if (click_thread_tasks)
        for (int i = 0; i < click_thread_tasks->size(); i++) {
            if ((*click_thread_tasks)[i] == current) {
                (*click_thread_tasks)[i] = click_thread_tasks->back();
                click_thread_tasks->pop_back();
                break;
            }
        }
    printk("<1>click: stopping router thread pid %d\n", current->pid);
    SPIN_UNLOCK(&click_thread_lock);

    return 0;
}
```

The first lines of code are there to pin (i.e., assign) the thread to a CPU. The CLICKPARM_CPU is a parameter that can be given to click-install to specify which CPU a configuration should run on. The function calls set_cpus_allowed to pin the current task to the mycpu cpu (we're omitting a lot of details about these calls, including the variable current which is a struct task_struct * of the current task, to

improve readability). The function then calls the thread's driver() function in order for the Click configuration to run. After it finishes some clean-up is done and the router thread finishes.

As described so far, when a thread runs it pulls a task (i.e., a schedulable Click element) and runs it; another thread pulls the next task and so forth. This means that we don't have much control over which threads run which parts of a Click configuration. This is especially important when we start making performance experiments and pinning particular threads to CPU cores. In fact, Click provides an element called StaticThreadSched that allows us to assign specific schedulable elements to specific threads. Here's how one instance of such an element might look like in a Click configuration file:

```
StaticThreadSched(fd0 0, td0 0,
                  fd1 1, td0 1,
                  fd2 2, td3 2,
                  fd3 3, td2 3);
```

This configuration has four network interfaces (split into fd=FromDevice? and td=ToDevice? elements). A pair of FromDevice/ToDevice? is assigned to a thread id (the second argument in each pair), for a total of four threads. This allows us to later on, for instance, assign each of the four threads to different CPU cores in order to improve performance.

How does this element actually do its work? To see this, it is worth having a look at its implementation in elements/threads /staticthreadsched.cc:

```
int
StaticThreadSched::configure(Vector<String> &conf, ErrorHandler *errh)
{
    Element *e;
    int preference;
    for (int i = 0; i < conf.size(); i++)
    {
      if (cp_va_space_kparse(conf[i], this, errh,
                         "ELEMENT", cpkP+cpkM, cpElement, &e,
                         "THREAD", cpkP+cpkM, cpInteger, &preference,
                         cpEnd) < 0)
            return -1;

          ...

          _thread_preferences[e->eindex()] = preference;
    }
    _next_thread_sched = router()->thread_sched();
    router()->set_thread_sched(this);
    return 0;
}

int
StaticThreadSched::initial_home_thread_id(Element *owner, Task *task,
                                          bool scheduled)
{
    int eidx = owner->eindex();
    if (eidx >= 0 && eidx < _thread_preferences.size()
        && _thread_preferences[eidx] != THREAD_UNKNOWN)
        return _thread_preferences[eidx];
    if (_next_thread_sched)
```

```
        return _next_thread_sched->initial_home_thread_id(owner,
                                                          task,
                                                          scheduled);
    else
        return THREAD_UNKNOWN;
}
```

Before going further it is worth pointing out that StaticThreadSched inherits not only from Element, but also from ThreadSched (include/click/standard/threadsched.hh) which is essentially an empty "interface" class mandating the implementation of the initial_home_thread_id function above. As shown in the listing, configure reads each [ELEMENT THREAD_ID] pair, populating the _thread_preferences data structure (of type vector<int>). The eindex() function returns the element's index within its router; this acts as an id. The configure function finishes by setting the element's router's scheduler to this (this is possible because, as mentioned, StaticThreadSched inherits from ThreadSched).

So far we've explained the configure function. In order to show how initial_home_thread_id is used we need to start at a schedulable element, for instance InfiniteSource, whose initialize method is as follows:

```
int
InfiniteSource::initialize(ErrorHandler *errh)
{
  if (output_is_push(0)) {
    ScheduleInfo::initialize_task(this, &_task, errh);
    _nonfull_signal = Notifier::downstream_full_signal(this, 0, &_task);
  }
  return 0;
}
```

This method calls ScheduleInfo's initialize_task, defined in elements/standard/scheduleinfo.cc +117:

```
void
ScheduleInfo::initialize_task(Element *e, Task *task, bool schedule,
                              ErrorHandler *errh)
{
    int tickets = query(e, errh);
    if (tickets > 0) {
        task->initialize(e, schedule);
        task->set_tickets(tickets);
    }
}
```

This function calls Task's initialize method, defined in lib/task.cc +191:

```
void
Task::initialize(Element *owner, bool schedule)
{
    Router *router = owner->router();
    int tid = router->initial_home_thread_id(owner, this, schedule);
    _thread = router->master()->thread(tid);
    _owner = owner;

    ...
```

```
    _status.home_thread_id = _thread->thread_id();
    _status.is_scheduled = schedule;
    if (schedule)
        add_pending();
}
```

This is where the actual work is done. In particular, the function first retrieves the schedulable element's router, calling initial_home_thread_id on it. Since we had changed the router's scheduler to be the StaticThreadSched element, this call will resolve to StaticThreadSched::initial_home_thread_id, which will return the thread id corresponding to the owner element (i.e., the assignment we had given as parameter to the StaticThreadSched element in the Click configuration file). The function then retrieves the thread corresponding to the thread id and sets it as the running thread.

Lastly, we haven't talked about setting CPU affinities, i.e., how threads are assigned to CPU cores. Click largely leaves this up to the OS. Under Linux, users can install a Click configuration and then set affinities afterwards, either through command-line tools like taskset or by calling the sched_setaffinity and sched_getaffinity functions. If no affinities are set, the OS will decide the assignment. Another way would be to write a new Click element to configure this; no such element exists yet.

5. Poll-based Click

In order to run Click in polling mode, a modified network driver is needed, which generally limits things to an Intel card. Patched versions of the Intel drivers are provided within the Click source tree under the drivers sub-directory. The first thing to do is to insert the modified driver's kernel module. Once this is done, create a Click configuration that uses the element PollDevice instead of the usual FromDevice. That's it!

To do: perhaps a description of which modifications are needed for a driver to work with Click in polling mode, as well as a detailed description of the PollDevice element.

6. Memory Allocation

The Click functions and definitions related to memory allocation operations are found in include/click/glue.hh and lib/glue.cc. glue.hh defines the following:

```
#  include <linux/malloc.h>
#  include <linux/vmalloc.h>

# define CLICK_LALLOC(size)     (click_lalloc((size)))
# define CLICK_LFREE(p, size)   (click_lfree((p), (size)))
extern "C" {
void *click_lalloc(size_t size);
void click_lfree(volatile void *p, size_t size);
}
```

This file includes the standard Linux headers for malloc and vmalloc and sets up a couple of macro definitions: CLICK_LALLOC resolves to glue.cc's click_lalloc function and CLICK_LFREE to its click_lfree, both covered below. Most of the interesting bits are in glue.cc, which

begins by defining some variables and macros:

```
uint32_t click_dmalloc_where = 0x3F3F3F3F;
size_t click_dmalloc_curnew = 0;
size_t click_dmalloc_totalnew = 0;
size_t click_dmalloc_failnew = 0;

struct task_struct *clickfs_task;
#  define CLICK_ALLOC(size)     kmalloc((size), (current == clickfs_task ?
                                          GFP_KERNEL : GFP_ATOMIC))
#  define CLICK_FREE(ptr)       kfree((ptr))
```

This code sets up some simple variables to keep track of memory allocation and potential leaks. For instance, click_dmalloc_curnew is used to report any potential leaks when the Click Linux kernel module exits (linuxmodule/linux.cc +333):

```
extern "C" void
cleanup_module()
{
    extern size_t click_dmalloc_curnew; /* glue.cc */

    ...

    if (click_dmalloc_curnew) {
        printk("<1>click error: %d outstanding news\n",
               click_dmalloc_curnew);
        click_dmalloc_cleanup();
    }
}
```

The macros shown simply replace calls to CLICK_ALLOC and CLICK_FREE with calls to Linux's kmalloc and kfree, respectively. Further down the same file we find definitions for the new and delete operators:

```
void *
operator new(size_t sz) throw ()
{
  click_dmalloc_totalnew++;
  if (void *v = CLICK_ALLOC(sz)) {
    click_dmalloc_curnew++;
    return v;
  } else {
    click_dmalloc_failnew++;
    return 0;
  }
}

void *
operator new[](size_t sz) throw ()
{
  click_dmalloc_totalnew++;
  if (void *v = CLICK_ALLOC(sz)) {
    click_dmalloc_curnew++;
    return v;
  } else {
    click_dmalloc_failnew++;
```

```
    return 0;
  }
}

void
operator delete(void *addr)
{
  if (addr) {
    click_dmalloc_curnew--;
    CLICK_FREE(addr);
  }
}

void
operator delete[](void *addr)
{
  if (addr) {
    click_dmalloc_curnew--;
    CLICK_FREE(addr);
  }
}
```

Since kmalloc and its corresponding CLICK_ALLOC should only be used to allocate relatively small amounts of memory, Click provides alternate functions:

```
extern "C" {

# define CLICK_LALLOC_MAX_SMALL 131072

void *
click_lalloc(size_t size)
{
    void *v;
    click_dmalloc_totalnew++;
    if (size > CLICK_LALLOC_MAX_SMALL)
        v = vmalloc(size);
    else
        v = CLICK_ALLOC(size);
    if (v) {
        click_dmalloc_curnew++;
    } else
        click_dmalloc_failnew++;
    return v;
}

void
click_lfree(volatile void *p, size_t size)
{
    if (p) {
        if (size > CLICK_LALLOC_MAX_SMALL)
            vfree((void *) p);
        else
            kfree((void *) p);
        click_dmalloc_curnew--;
    }
}
}
```

If the amount of memory requested is less than CLICK_LALLOC_MAX_SMALL, CLICK_ALLOC and kfree are used as previously. If the amount is greater, on the other hand, the functions make calls to vmalloc and the corresponding vfree.

A. Acknowledgment

The creation of this document was partly funded by the EU FP7 CHANGE (257422) Project.

http://www.change-project.eu/ [http://www.change-project.eu/]