HWL Documentation

Release 1.0

Robert Sombrutzki

May 24, 2012

CONTENTS

Inhalt:

CHAPTER

ONE

CLICK

1.1 Installation

1.1.1 Vorbereitungen

1. Benötigte Software installieren: gcc, g++, autoconf, libx11-dev, libxt-dev, libxmu-dev, flex, bison, git

   gcc und g++ müssen zur Zeit in Version 4.4 vorliegen. Zum Übersetzen der Software sollten also die Pakete gcc-4.4 und g++-4.4 installiert werden und temporär die symbolischen Links in /usr/bin auf die entsprechenden binaries gesetzt werden:

   ```
   sudo rm /usr/bin/gcc
   sudo rm /usr/bin/g++
   sudo ln -s /usr/bin/gcc-4.4 /usr/bin/gcc
   sudo ln -s /usr/bin/g++-4.4 /usr/bin/g++
   ```

2. Account auf gitsar bei Robert beantragen
3. Folgende Einträge in der .ssh/config vornehmen:

   ```
   Host gruenau
       User username
       HostName gruenau.informatik.hu-berlin.de

   Host gitsar
       User username
       ProxyCommand ssh -q gruenau netcat sar 2222
   ```

Am komfortabelsten funktioniert die Verbindung beim Einsatz von SSH-Keys und dem ssh-agent, da man dann nur einmal die Passphrase für den SSH Schlüssel eingeben muss.

1.1.2 Software auschecken

1. In einem separaten Terminal SSH Verbindung zu gruenau herstellen (und geöffnet halten):

   ```
   ssh gruenau
   ```

2. click-brn auschecken:

   ```
   git clone ssh://gitsar/home/sombrutz/repository/click-brn/.git
   ```

3. brn-tools.sh ausführen:

```
cp click-brn/elements/brn2/tools/brn-tools.sh .
chmod a+x brn-tools.sh
./brn-tools.sh
```

4. Umgebungsvariablen setzen. Entweder per `source /tmp/./brn-tools.bashrc` oder durch das Kopieren der Einträge in `brn-tools.bashrc` in die eigene `.bashrc`

1.1.3 Testen

Zum Testen der Installation kann die *simple_flow* (oder eine andere) Simulation ausgeführt werden:

```
cd click-brn-scripts/003-simple_flow
run_sim.sh
```

Die Ausgabe der Simulation sollte dabei folgendermaßen aussehen:

```
sim is ns
Send 26 packet, received 26 packets and 26 echo packets are received. OK !
```

Alternativ können über das Shellscript test.sh alle Simulationen angestoßen werden. `test.sh` erzeugt dabei ein pdf namens `testbed.pdf`, welches die Resultate aller Simulationen enthält.

1.1.4 Troubleshooting

- **Wenn die Simulation mit dem Fehler `*** buffer overflow detected ***: ./ns terminated`** abbricht liegt das vermutlich daran, dass die falsche TCL Version verwendet wird. Abhilfe schafft entweder die Deinstallation von TCL auf dem System oder aber das Sicherstellen, dass ns2/bin/tcl vor den Systembinaries im Pfad liegt.
- **Falls das Ausführen von test.sh Warnungen und/oder Fehler erzeugt liegt dies unter Ubuntu** Systemen evtl. daran, dass die dash als System Shell verwendet wird. Eine mögliche Lösung besteht darin, eine andere System Shell mittels `sudo dpkg-reconfigure dash` festzulegen.

1.2 Simulationen

Im Ordner `simulation/click-brn-scripts/` liegen Simulationsscripte für verschiedene Experimente. Beim Ausführen der Scripte (siehe *run_sim.sh*) werden verschiedene die Ausgaben der verschiedenen Knoten gesammelt. Anschließend werden diese Ausgaben analysiert und ausgewertet und die Resultate im Ordner `simulation/click-brn-scripts/<SCRIPT>/<NUMBER_OF_EXPERIMENT>` gespeichert.

Eine Simulation wird durch die folgenden beiden Dateien definiert:

- Mes-files beinhalten alle Geräte, die für das Experiment vorbereitet werden.
- Des-Files beinhalten eine grobe Beschreibung vom Experiment. Z.b.: Die Dauer des Experiments; das Verzeichnis für die log-files; Netzwerk-Topologie; etc. ...

1.2.1 run_sim.sh

Um eine Simulation auszuführen wird das *run_sim.sh* Script verwendet, welches sich im Verzeichnis */helper/simulation/bin/* befindet. Das Script nimmt als Parameter den zu verwendenden Simulator (*ns* oder *jist*) und den Pfad zur *des* Datei der Simulation entgegen:

```
run_sim.sh ns <des-File>
run_sim.sh jist <des-File>
```

1.3 Experimente im Testbed

1.3.1 clickctrl.sh

Mit dem *clickctrl.sh* Script (zu Finden im Ordner */helper/host/bin/* kann man einzelne *Handler* von einzelnen Knoten im Netzwerk abfragen oder schreiben. Ein Anwendungsfall ist beispielsweise das Abfragen von Statistiken zur Laufzeit eines Experiments. Das Skript wird folgendermaßen aufgerufen:

```
clickctrl.sh read address port element handler
clickctrl.sh write address port element handler "arguments of element"
```

Das Skript verwendet intern das ControlSocket Element.

1.4 Weitere Dokumentation

- Search click documentation: http://read.cs.ucla.edu/click/docs
- Publications about click and stuff that uses click: http://read.cs.ucla.edu/click/publications
- Manual how to program click elements: http://read.cs.ucla.edu/click/doxygen/class_element.html
- Information about click elements: http://www.read.cs.ucla.edu/click/elements
- Network Simulator 2 (NS2) Docu: http://isi.edu/nsnam/ns/

CHAPTER

TWO

CLICK-ELEMENTE BAUEN

2.1 Handler

2.1.1 Handler bauen

Handler sind Zugangspunkte, mittels derer die Nutzer mit einzelnen Elementen zur Laufzeit des Click-Routers interagieren können. Man unterscheidet dabei zwischen read und write handlern, die sich jeweils wie Dateien in einem Dateisystem verhalten. Im folgenden wird dies anhand eines Beispiels erläutert.

1. Falls nicht vorhanden, füge eine Klassen-Methode mit dem Namen add_handlers() zum Element hinzu. Diese Funktion dient als Sammelstelle für die Registrierung aller Element-Handler:

   ```
   void
   MyElement::add_handlers()
   {
           //todo
   }
   ```

2. Als nächstes folgt die read/write Handler im einzelnen. Dazu werden die Methoden *add_read_handler* und *add_write_handler* verwendet. Das Argument *name* definiert den Namen des Handlers, *func* steht für die aufzurufende Funktion und *thunk* für zusätzliche Parameter. Vollständige Signatur und Beispiel:

   ```
   void add_read_handler (const String &name, ReadHandler func, void *thunk)
   void add_write_handler (const String &name, WriteHandler func, void *thunk)

   // Beispiel
   add_read_handler("info", read_my_param, (void *) H_READ);
   add_write_handler("debug", write_my_param, (void *) H_DEBUG);
   ```

3. Nun müssen die Funktionen der neu eingeführten Handler definiert werden. (Die Besonderheit ist hier das Schlüsselwort static. Dies hat tiefergehende Gründe, die hier nicht weiter diskutiert werden.)

   ```
   #include <click/straccum.hh>
   ...
   static String
   read_my_param(Element *e, void *thunk) {
           StringAccum sa;
           // do foo hier
           return String();
   }

   static int
   write_my_param(const String &string, Element *e, void *vparam, ErrorHandler *errh) {
   ```

```
        MyElement *e = (MyElement *)e; //cast
        e->cmd = string;
}
```

Quelle: http://www.read.cs.ucla.edu/click/element?s[]=handler

2.1.2 Handler verwenden

Handler können auf verschiedenste Weisen verwendet werden.

- In einem Click-Script: Dazu wird meist ein *Script*-Bereich am Ende eines Click-Scripts eingeführt.

  ```
  ...

  Script(
  read MyElement.info,
  write MyElement.debug 3
  );
  ```

Quelle: http://www.read.cs.ucla.edu/click/elements/script?s[]=handler

- ControlSocket

Quelle: http://www.read.cs.ucla.edu/click/elements/controlsocket?s[]=handler

2.1.3 Weiterführende Links

-

2.2 OpenSSL

Bei der Verwendung der Funktionen aus der SSL-Library ist es notwendig, dem Compiler mitzuteilen, dass er die ssl-lib mitlinken soll. Dies geschieht durch das Hinzufügen der folgenden Zeile an das Ende der Quelldatei eines Elementes, in dem die SSL-Includes zur Anwendung kommen:

```
ELEMENT_LIBS(-lssl)
```

CHAPTER

THREE

CLICK-SKRIPTE BAUEN

Vorwissen: Click-Paper

Bevor man beginnt ein Click-Skript zu entwickeln, sollte man erst einmal einen Entwurf machen, in dem man festhält, welches Ziel man in der Simulation verfolgt. Das Skript, welches man am Ende entwickelt, liefert den Netzwerkknoten, auf dem es läuft, einen vollständigen Netzwerkstack. Dementsprechend müssen Netzwerkgeräte, Routing-Protokolle, u. Ä. in den Entwurft mit einbezogen werden. Die Vorüberlegungen beinhalten: * Welche Elemente oder Elementklassen dafür benötigt werden und * wie diese Elemente miteinander verbunden sind.

Höchst wahrscheinlich benötigt man ein Wifi-Device, welches stellvertretend für die physikalische Schicht steht. Sodann wird ein Routing-Protokoll benötigt, stellvertretend für den Vermittlungsschicht.

3.1 Elemente zum Senden und Empfangen im BRN2-Netzwerk

Jedes zu übertragene Paket benötigt einen Header, der Netzwerkinformationen enthält, die z. B. für das Routing von Packeten wichtig sind. Um die Packete mit den entsprechenden Headern korrekt auszustatten, gibt es verschiedene Elemente.

- BRN2EtherEncap(), BRN2EtherDecap()

Hinzufügen und Enternen der Ethernet-Header im BRN-Netz.

- BRN2Encap(), BRNDecap()

Hinzufügen und Entfernen der BRN-Header. *Achtung*: Das BRN2Encap() ist ein Element ohne In- und Outputs und ist daher als Fluss-Element nicht zu gebrauchen. Es stellt hauptsächlich eine Funktion namens “add_brn_header()” zur Verfügung, die von anderen Click-Elementen verwendet werden kann und sollte. Dies hat einen konzeptionellen Grund, nämlich eine größere Kontrolle über die zu verschickenden Packete zu erhalten. Hier ein Beispiel:

```
tls
        // -> BRN2Decap() /* Diese Funktion wird vom "tls"-Element implizit übernommen*/
        -> BRN2EtherEncap(USEANNO true)
        -> [1]device_wifi;
```

In diesem Beispiel enthält das tls nicht nur die Packet-Generierungsfunktion *Packet::make()* sondern auch *BRNPacketAnno::set_ether_anno()* und *BRNProtocol::add_brn_header()*.

- EtherEncap(), EtherDecap()

Hinzufügen und Entfernen der standerd Ethernet-Header. Diese sind allerdings für das BRN-Netzwerk wenig brauchbar.

3.2 Besonderheiten in der BRN-Architektur

Das Click-Skript wird beim Aufruf mit run_sim.sh vorbereitet. Dabei werden einige Variablen durch spezifische Informationen über das Netzwerkgerät, den Knoten, etz. ersetzt. Die wichtigsten Informationen stehen in der mes-Datei (oder auch Nodetable genannt).

3.3 Simulation & Debugging

Wer eine genauere Analyse des Netzwerkverkehrs machen möchte, der sollte sich die dumps anschauen. Das dumping muss jedoch zuvor aktiviert werden im Click-Skript. Dazu fügt man diese beiden Zeilen an forderster Stelle im Click-Skript ein:

```
#define RAWDUMP
#define RAWDEV_DEBUG
```

CHAPTER

FOUR

INTEGRATION VON NS2 UND CLICK

4.1 PollingDevice

4.1.1 Original

`FromDevice` scheduled sich solange selbst bis von oben kein Packet mehr kommt oder das device belegt ist (not ready). Im Simulator wird anhand der `ClickQueue` entschieden, ob das Gerät frei ist oder nicht. Ist kein Paket in der queue (max. 1 Paket (`QueueSize == 1`) ist das Gerät frei:

```
int LLExt::ready() {
    ClickQueue* pcq = (ClickQueue*) ifq_;
    if (pcq) {
        return (!packetpending_ && pcq->ready());
    }

    // No ClickQueue? Then we're always ready.
    return 1;
}
```

dequeue (ClickQueue) nimmt ein Paket aus der Queue, wenn die Uebertragung beendet wurde) und gibt es noch oben (click):

```
Packet* ClickQueue::deque()
{
    Scheduler& s = Scheduler::instance();
    double dcurtime = s.clock();
    fprintf(stderr,"Time: %f", dcurtime);
    Packet* retval = pq_->deque();
    return retval;
}
```

Das Gerät ist frei, sobald `ClickQueue::deque()` aufgerufen wurde.

4.1.2 Verbesserung

`TXFeedback` signalisiert, dass das Geraet wieder bereit ist. Maximal ist 1 Paket im Simulator

4.1.3 Problem

Wenn die Uebertargung abgeschlossen wurde und das Paket aus der ClickQueue, wird es nach oben gegeben. Das jedoch nicht (unbedingt) sofort, sondern erst nach einer Verzögerung. Im tcl file lässt sich diese Verzögerung einstellen:

set netifq	Queue/ClickQueue
set netll	LL/Ext
#LL set `delay_`	1ms
LL set `delay_`	0ms

Durch die 1 ms Verzoegerung gibt es einen unterschied zwischen Pollen und Nicht-pollen: Nicht pollen (txfeedback basiert) erkennt das frei-werden des Geraetes erst 1ms spaeter (nach dem `delay_`) Die alte Version hatte diese Verzoegerung nicht. Dadurch gibt es unterschiede in der Geschwindigkeit.

loesung (mehr Workaround): Verzoegerung auf 0ms setzten.