Smart & Mobile Embedded Web Server

User Guide

Last revised: February 25, 2013

Contents

1 Introduction **2**
- 1.1 What is Smews? . . . 2
- 1.2 About this guide . . . 2

2 Getting Started **3**
- 2.1 Preparing your environment . . . 3
- 2.2 What's in the package? . . . 3
- 2.3 Compiling Smews . . . 4
  - 2.3.1 Short description . . . 4
  - 2.3.2 Exhaustive description . . . 5
- 2.4 Installing Smews . . . 7
- 2.5 Simple example, step by step . . . 7

3 Creating your own Web Application **8**
- 3.1 Static Content . . . 8
- 3.2 Dynamic Content . . . 9
  - 3.2.1 Defining handlers . . . 9
  - 3.2.2 Parsing URL arguments . . . 9
  - 3.2.3 Persistence of the generated data . . . 10
  - 3.2.4 Interaction mode . . . 11
  - 3.2.5 Timers . . . 12
- 3.3 Post request processing . . . 13
  - 3.3.1 Form . . . 13
  - 3.3.2 Received Post Files . . . 14
- 3.4 Web Application SConscript . . . 18
- 3.5 Synthesis of the features . . . 19

4 Creating an application for managing non-TCP packets **22**
- 4.1 Declaring GPIP generators . . . 22
- 4.2 Handlers . . . 23
  - 4.2.1 Packet information . . . 23
  - 4.2.2 `doPacketIn` handler . . . 23
  - 4.2.3 `doPacketOut` handler . . . 24
  - 4.2.4 Initiating packet generation . . . 25
- 4.3 Synthesis of the features . . . 25

5 Credits **27**
- 5.1 Contributors . . . 27
- 5.2 License . . . 27

Chapter 1

Introduction

1.1 What is Smews?

Smews stands for *Smart & Mobile Embedded Web Server*. This research prototype is designed for hardware-constrained devices like smart cards, sensor boards and other small devices. It is a stand-alone software, working without any underlaying OS. It acts as the OS by itself, dedicated to the support of Web applications. Its kernel includes device drivers, TCP/IP stack, Web server and Web applications container. It is based on a event-driven architecture and its implementation is full of cross-layer optimizations.

Web applications are pre-processed, compiled, linked with Smews then embedded in a device. Web applications are made of static and dynamic contents. Static contents are simple files; dynamic contents are linked to server-side code. During the pre-processing phase, plenty of optimizations are made on Web contents (pre-calculation of protocol headers, checksums, parsing. automatons). Smews supports Comet (server-pushed data) and provides an advanced typing of dynamic Web contents (persistent, idempotent and volatile contents). The post request processing is also implemented.

1.2 About this guide

This document contains the necessary information to get started with Smews, understand its philosophy and master its usage. It should be read by both users willing to use Smews and developers wanting to be introduced to Smews functionalities.

Chapter 2 contains information about Smews compilation, installation and execution, giving detailed features descriptions and concrete examples. Chapter 3 explains how to build your own Web application for Smews.

Chapter 2

Getting Started

2.1 Preparing your environment

You can download the latest Smews release from its Github repository `http://github.com/2xs/smews`

In order to work with Smews, you will need:

- a valid C compilation environment, depending on your platform the target:
  - *for target MicaZ and Funcard7* you will need: `gcc-avr` (4.3 or above) and `avr-libc` packages;
  - *for target MSP430* you will need the following packages: `binutils-msp430`, `msp430-libc`, `gcc-msp430`. These packages can be found in `deb http://wyper.ca/debian/ i686` repository;
  - *for target GBA* you will need: DevKitAdvance (`http://devkitadv.sourceforge.net/`);
  - *for target MBED* you will need: `arm-none-eabi-gcc`. This version of gcc can be built using the script available at (`https://github.com/esden/summon-arm-toolchain`). For the ethernet version of the mbed port, you will also need rflpc library. This library can be automatically downloaded and compiled by running the `summon-rflpc` script in the `targets/mbed_ethernet` folder.
- Python version 2.5 or above;
- SCons version 0.96 or above.

The Smews compilation process is based on SCons and Python. It has been tested on Linux, Windows (using MinGW) and Mac OS.

2.2 What's in the package?

In the Smews package you can find the following directories and files:

- `SConstruct, SConscript` – Smews general SCons files;
- `apps/` – examples of Web applications to be served by Smews;
- `core/` – Smews kernel C source code (portable files);

- `panManager/` – Personal Area Network manager (setting device-computer link);
- `tools/` – Python preprocessing tools (used by the SCons compilation chain);
- `targets/` – Smews ports to different devices.

Every port is made of a directory in `targets/`. Target-dependent code is located in `targets/$target/drivers`. The `targets/$target/SConscript` file defines target-specific compilation instructions. The `targets/$target/install` script is in charge of installing Smews on the target device. The following lines describe all existing Smews ports.

WSN430 The WSN430 sensor board is based on a 16-bits msp430 processor at 8 MHz. Communications are done over a serial line by using the SLIP protocol.

MicaZ The MicaZ sensor board is based on a 8-bits AVR processor at 8 MHz. Communications are done over a serial line by using the SLIP protocol.

Funcard7 The Funcard7 smart card is based on a 8-bits AVR processor at 4 MHz. Communications are done over a serial line using a custom "IP over APDU" protocol.

GBA The Game Boy Advance is based on a 32-bits Arm7 processor at 16 Mhz. Communications are done over a serial line by using the SLIP protocol.

MBED The MBED platform is a popular prototyping board based on the LPC1768 SoC by NXP. It includes an ARM Cortex-M3 32bits processor at 100Mhz as well as an ethernet mac 100Mbps full duplex. Smews as been ported to this platform using SLIP and ethernet. Note that before compiling for the `mbed_ethernet` target, you have to run the `targets/mbed_ethernet/summon-rflpc` script. In order to make the port work in IPv6, you should add the `icmpv6` application that will handle neighbor solicitation packets.

Linux This target has been written for debug and development purposes. It allows to execute Smews as a linux process, communicating *via* the TUN virtual interface.

Skeleton This target is an empty one. It is a starting point for someone wanting to port Smews to a new device.

2.3 Compiling Smews

Smews uses SCons for building which allows to design complex build mechanism and dependencies. The multiple pre-processing steps help Smews to be more efficient in terms of memory consumption and processing speed.

2.3.1 Short description

When building Smews you can specify multiple targets and Web applications to be embedded with it. For example to build Smews for a destination target A and B (directories in `targets/`) and embed in it Web Applications X and Y (directories in `apps/`) you can type:

```
$ scons target=A,B apps=X,Y
```

`target=targetA,targetB,...`	targets on which to compile Smews
`apps=[urlA:]appA,[urlB:]appB,...`	applications to be embedded with Smews
`ipaddr=ip`	set the Smews IP address
`debug=true|false`	enable or disable debug mode
`gzip=true|false`	enable or disable static Web contents compression
`chuncksNbits=number`	size of the chuncks used for pre-calculated checksums
`endian=little|big`	data endianness
`disable=[arguments|comet|retransmit|post|gpip]+`	disable internal functionalities

Table 2.1: Smews compilation options

Files are compiled in the `bin/$target` directory. This contains the Smews executable file. It also contains a `libsmews.a` file, allowing to embed Smews in other software (see `core/main.c` provided as a sample main C code linked with the library `libsmews.a`).

To clean built files type:

```
$ scons -c
```

For a complete description of available arguments, type:

```
$ scons -h
```

2.3.2 Exhaustive description

The Smews compilation chain has many arguments enabling different options. The only necessary argument for a build is `target`. All other arguments are optional and have a default value. Table 2.1 synthesizes the list of available arguments. We give here a precise description of every argument.

Option `target=targetA,targetB,...`

Set the target(s) on which Smews is going to be compiled. Example:

```
$ scons target=linux,WSN430
```

Option `apps=[urlA:]appA,[urlB:]appB,...`

Specifies the set of Web applications that will be linked and embedded with Smews. An application is a directory or sub-directory in the `apps` directory. Every application is made of simple files (static Web contents) and/or C files (dynamic Web contents). Contents are separated by comma and the name of the content can be preceded by a replacement URL. Default value: `apps=:welcome`. Example:

```
$ scons target=linux apps=contactsBook,mycalendar:calendar,:welcome
```

Resources extracted from directories `contactsBook`, `calendar` and `smews` will be respectively accessible at the following URLs:

```
http://$smewsip/contactsBook
http://$smewsip/mycalendar
http://$smewsip/
```

Option `ipaddr=ip`

Set the Smews IP address. Can be either a IPv4 or IPv6 address. Examples:

```
$ scons target=linux apps=:welcome ipaddr=192.168.1.2
$ scons target=linux apps=:welcome ipaddr=2001::2
```

Option `debug=true|false`

Allows to compile Smews in debug mode, without any optimization (`gcc -g -O0` options). Default value: `debug=false`. Example:

```
$ scons target=linux apps=:welcome debug=true
```

Option `gzip=true|false`

Allows to compress static files at compile time. This reduces the target footprint, but the client browser must be able to unpack the content. Default value: `gzip=true`. Example:

```
$ scons target=linux apps=:welcome gzip=false
```

Option `chuncksNbits=number`

In Smews, the checksums of static files are pre-calculated on chunks of data at compile time. The size of the chunks is computed as `1<<chuncksNbits`. The size of outgoing segments is limitated by this size since it has to be multiple of it. Default value: `chuncksNbits=5`.

```
$ scons target=linux apps=:welcome chunckNbits=5
```

Option `endian=little|big`

Forces data little or big endianness. Default value depends on the target (`ENDIANNESS` C macro defined in the `target/$target/target.h` file). Example:

```
$ scons target=linux apps=:welcome endian=little
```

Option `disable=[arguments|comet|retransmit]+`

Allows to disable some internal functionalities, making the binary smaller and the execution faster:

- `arguments` – URL arguments parsing will be removed from the source code;
- `comet` – removes comet support in the binary, which disallows the server to push data to the client;
- `retransmit` – disable TCP packets retransmission in case of lost segments. The generated web server will not be fully compliant with the TCP RFC.

Example:

```
$ scons target=linux apps=:welcome disable=comet,arguments
```

2.4 Installing Smews

Once Smews has been compiled, it can be embedded in the target device. This step can be done by executing the `target/$target/install` script.

Then, the panManager has to be launched. It enables the communication between a computer and a device, managing the link-layer protocol. When launching the panManager, two arguments are mandatory:

Plugin The first argument is the plugin, implementing the desired link-layer protocol. Currently, the two existing plugins are SLIP (Serial Line IP) and APDUIP (APDU supporting IP). For SLIP, you can specify the serial device using the -p option (/dev/ttyS0 by default).

IP configuration The second argument is used to configure the computer routing table. It describes an IP address and a mask size by using the "slash" notation. The panManager supports either IPv4 and IPv6 addresses.

Example: configuring a SLIP link with local address 192.168.1.1 and a 24-bits mask:

```
$ panManager slip 192.168.1.1/24
```

Example: configuring a APDUIP link with local address 2001::1 and a 126-bits mask:

```
$ panManager apduip 2001::1/126
```

After this step, you can reach Smews at its IP address by using any HTTP/1.1 compliant Web client.

2.5 Simple example, step by step

We synthesize the previous sections by giving a complete example where we compile, install and access Smews on a WSN430 sensor board. All commands are given from the main Smews directory.

Compilation, embedding the welcome page and sensor application, with IPv4 address 192.168.1.2:

```
$ scons target=WSN430 apps=:welcome,sensor ipaddr=192.168.1.2
```

Installation, *i.e.* copy of the code in the EEPROM of the WSN430 board (as root):

```
$ ./targets/WSN430/install
```

PanManager configuration (as root, in panManager folder):

```
$ bin/panManager slip 192.168.1.1/24
```

Chapter 3

Creating your own Web Application

Web applications are made of static (files) and dynamic (generated by the server at runtime) contents. Applications are identified by subdirectories of the `apps/` directory. The `apps/examples` contains application examples using various features of Smews. A folder identifies a particular Web Application or a set of Web applications which will be embedded in the web server (see Section 2.3 for more details). Every application is made of a set of files, of various types :

.c and .h files C and H files are used to embed server-side code, compiled and linked with Smews;

.c files with XML C files containing specific XML meta-data (as c comment) are associated to Web resources, *i.e.* they are in charge of generating dynamic Web contents in response to client requests;

other files Other files are considered as static. They will be embedded and served by Smews "as is";

.c.embed files The .embed extension is automatically removed. This allows to serve a C file as a static content.

3.1 Static Content

When embedding static file in Smews, the HTTP content-type field is automatically inferred from the file extension (the mappings list is contained in the file `tools/mimeRessources`). Furthermore, many optimizations occur off-line, including HTTP header and TCP checksums pre-calculation. As an example, create a new file containing the following code and save it in `apps/helloWorld/hello.html`:

```
<html>
    <head>
        <title>Hello World!</title>
    </head>
    <body>
        Hello World!
    </body>
</html>
```

Compile Smews setting the target to `linux` and the content to be embedded `helloWorld` as below:

```
$ scons target=linux apps=helloWorld ipaddr=192.168.1.2
```

Then run the ELF file from the `/bin/linux` directory. You can now access the server at `http://192.168.1.2/helloWorld/hello.html`. Keep in mind that the linux target is a particular case where Smews binary file is a process, that is why it can be launched directly. If tested on different target, refer to the instructions in Section 2.4.

3.2 Dynamic Content

Smews can serve content generated by native code, which we will refer to as *Dynamic Content*. Any C file containing XML specific code (as C comment) will considered as a Web resource and associated to an URL. The XML meta-data are interpreted in the pre-processing phase, thus generating some C code. We describe here the role of the XML markers available (a detailed list is given in Table 3.1, page 19).

3.2.1 Defining handlers

Web applications are scheduled by Smews *via* a pre-defined set of handler functions (a detailed list is given in Table 3.2, page 20):

- `init` – function executed during Smews initialization;
- `initGet` – function executed when a Get request is received ;
- `doGet` – function used to generate the HTTP response.

The XML meta-data allows to associate C functions to these handlers. As an example, create a new file containing the following code and save it in `apps/helloWorld2/hello.c`:

```
/*
<generator>
        <handlers doGet="do_hello"/>
</generator>
*/

/* simple contents generator */
static char do_hello(struct args_t *args) {
    out_str("Generated Hello World!");
    return 1;
}
```

You can now access the server at `http://192.168.1.2/helloWorld2/hello`. On the server side, the `doHello` function will be called by Smews when a the response to the client request has to be generated. The `out_str` function outputs a string as the HTTP response (a detailed list of the functions provided by Smews to Web applications is given in Table 3.3, page 21).

3.2.2 Parsing URL arguments

Smews allows to parse URL arguments before calling the `doGet` handler. This parsing is processed in the kernel, so it is quite efficient and requires few memory. The format of the arguments are defined statically. At compile-time, a parsing automaton is generated for every dynamic Web resources, allowing an efficient arguments parsing. Here is an example where URL arguments are parsed then sent as an HTTP response:

```
/*
<generator>
    <handlers doGet="output_args"/>
    <args>
        <arg name="i1" type="uint8" />
        <arg name="s" type="str" size="6" />
        <arg name="i2" type="uint16" />
    </args>
</generator>
*/

static char output_args(struct args_t *args) {
    if(args) {
        out_str("first int : ");
        out_uint(args->i1);
        out_str("\nstr : ");
        out_str(args->s);
        out_str("\nsecond int : ");
        out_uint(args->i2);
    } else {
        out_str("no args");
    }
    return 1;
}
```

The `args` parameter points to the values of the arguments that have been parsed by Smews. Every URL argument is directly accessible as a field of the `struct args_t` structure, with the name that has been described in the XML meta-data. The `out_uint` function is used to output an integer.

3.2.3 Persistence of the generated data

Smews provides an advanced typing of dynamic contents. The internal behavior of the Smews TCP/IP stack automatically adapts to this typing, allowing to output many simultaneous segments while keeping memory consumption as low as possible. Three types of persistence have been defined:

volatile – means that the content generated will not be stored in memory and will be regenerated in case of TCP NACK;

idempotent – means that the function will return the same value no matter when you might call it (it is deterministic and has no side-effect). In the current implementation, idempotent data are managed is the same manner than volatile data;

persistent (default option) – means that the output will be kept in memory and will be delivered from there in case of NACK.

Here is an example of a volatile Hello World. In case of TCP loss, the HTTP response may generated several times by multiple calls of the `doGet` function:

```
/*
<generator>
    <handlers doGet="do_hello_v"/>
    <properties persistence="volatile"/>
</generator>
```

```
*/

/* possible persitence are persistent (by default), idempotent and volatile */
static char do_hello_v(struct args_t *args) {
    out_str("Volatile Hello World!");
    return 1;
}
```

3.2.4 Interaction mode

Smews also supports Web push, often called *Comet*. In Smews, Comet is implemented *via* channels. A channel is a way of sending data asynchronously by pushing it from server to a set of registered clients. To see how can you accomplish this take a look at the next example:

```
/*
<generator>
    <handlers doGet="waitknock"/>
    <properties interaction="alert" channel="knockknock"/>
</generator>
*/

/* launched when knockknock is triggered */
static char waitknock(struct args_t *args) {
    out_str("somebody knocked");
    return 1;
}
```

The above example set the interaction tag to `alert`, and defines a channel named `knockknock`. In alert mode, HTTP client requests are not answered as soon as they are received. Instead, they are simply registered as listening to the `knockknock` channel. The request is pending until an event occurs.

Let's create a second Web resource as follows:

```
/*
<generator>
    <handlers doGet="triggerknock"/>
</generator>
*/

/* triggers the knockknock comet channel */
static char triggerknock(struct args_t *args) {
    server_push(&knockknock);
    return 1;
}
```

When a request targets this second resource, the `triggerknock` function is called. It triggers the `knockknock` channel. At this time, the `waitknock` is called, thus generating a HTTP response containing the string "somebody knocked". This response will be sent to all registered clients.

Three interaction mode are supported by Smews:

rest This is the default mode, corresponding to the classical HTTP request/response scheme;

alert Incoming requests are pending, they will be answered when the associated channel is triggered;

stream Incoming requests are pending. Every time the associated channel is triggered, a chunk of HTTP response is generated and sent to all listening clients. The HTTP response is possibly never-ending.

3.2.5 Timers

Any Web application in Smews can use timers in order to execute a function at a given interval. As an example, this feature may be used in a sensor board to periodically check the current temperature. When the sampled value reaches a threshold, a HTTP chunck containing the data is sent to all registered clients *via* the stream interaction modes:

```
/*
<generator>
        <handlers init="init_adc_timer"
                initGet="set_threshold" doGet="send_temperature"/>
        <properties persistence="volatile"
                  interaction="stream" channel="tempAlert" />
        <args> <arg name="threshold" type="uint16" /> </args>
</generator>
*/

static uint16_t threshold = 512;
static uint16_t curr_sample;

/* timer callback, checking the temperature */
static void check_temp() {
        uint16_t tmp_result = get_adc_val(ADC_TEMP);
        if(tmp_result >= threshold) {
                curr_sample = tmp_result;
                trigger_channel(&tempAlert);
        }
}

/* initializes ADC and timer */
static char init_adc_timer(void) {
        return init_adc(ADC_TEMP) && set_timer(&check_temp, 200);
}

/* called when a get request is received. initializes the threshold */
static char set_thresholds(struct args_t *args) {
        if(args != NULL) {
                threshold = args->threshold;
                return 1;
        } else return 0;
}

/* called to generate the HTTP response */
static char send_temperature(struct args_t *args) {
        out_uint(curr_sample);
        return 1;
}
```

The `init_adc_timer` function is called during Smews initialization. It associates the `check_temp` function to a timer, asking Smews to execute it every 200 ms. It also initializes the ADC (allowing to sample the temperature). When a client request is received, the `set_thresholds`

function is called, thus setting the current threshold. The periodic timer executed every 200 ms compares the current temperature to this threshold. If needed, it triggers the channel, thus sending the current time to every registered clients. This produces an infinite HTTP response: the Web browser receives new data whenever the threshold is reached.

3.3 Post request processing

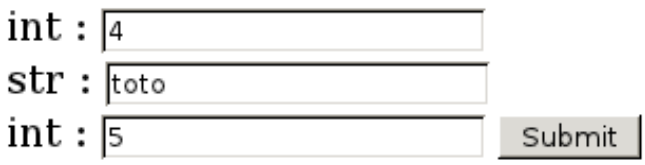
Smews supports post request processing, which are used to send lots of data. Smews considers two types of data : forms and files. The C files containing XML specific code (as C comment) like to generate dynamic content (see 3.2 section).

3.3.1 Form

Example of a HTML form :

```
<form action="post_test_1" method=post>
        <label>int :</label>
        <input type="text" name="i1" id="i1" value="4" />
        <br />
        <label>str :</label>
        <input type="text" name="s" id="s" value="toto" />
        <br />
        <label>int :</label>
        <input type="text" name="i2" id="i2" value="5" />
        <input type=submit value="Submit"/>
</form>
```

Display from a web browser :



SMEWS needs doPost function and arguments (like to parse URL arguments, see 3.2.2 section).

```
/*
<generator>
    <handlers doPost="output_args"/>
    <args>
        <arg name="i1" type="uint8" />
        <arg name="s" type="str" size="6" />
        <arg name="i2" type="uint16" />
    </args>
</generator>
*/

static char output_args(struct args_t *args) {
    if(args) {
        out_str("first int : ");
        out_uint(args->i1);
        out_str("\nstr : ");
        out_str(args->s);
```

```
        out_str("\nsecond int : ");
        out_uint(args->i2);
    } else {
        out_str("no args");
    }
    return 1;
}
```

The process is similar to get request processing with arguments.

3.3.2 Received Post Files

To send files to the server Smews, use multipart data with HTML. Here are two examples to manage one or more files, with an application that counts all characters of send files.

Example of a HTML form to send files :

```
With Single File
<form enctype=multipart/form-data action="app_counter_simple" method=post>
        <input type=file name=filename value="Choose file"/>
        <input type=submit value="Submit"/>
</form>
<br />
With Several Files
<form enctype=multipart/form-data action="app_counter_multi" method=post>
        <input type=file name=filename1 value="Choose file"/>
        <input type=file name=filename2 value="Choose file"/>
        <input type=file name=filename3 value="Choose file"/>
        <input type=submit value="Submit"/>
</form>
```

Display from a web browser :

With Single File

With Several Files

SMEWS needs doPostIn and doPostOut functions. The doPostIn function allows to receive data with help of the in function. Otherwise, doPostOut function is similar to doPost. Moreover, the content-types must be specified to define the file type, which will be accepted by the application (the possibles content-types are contained in the file tools/mimesListPost which can be modified). It is very important to understand that the function doPostIn will be called for each part of multipart request (each file) and the function doPostOut will be called only once at the end.

First example with one file

```
/*
<generator>
        <handlers doPostOut="doPostOut" doPostIn="doPostIn"/>
        <content-types>
                <content-type type="text/plain"/>
```

```
        </content-types>
</generator>
*/

/* file structure */
struct file_t {
        char *filename;
        uint16_t size;
};

/* dopostin function to operate post data */
static char doPostIn(uint8_t content_type, uint8_t call_number,
char *filename, void **post_data) {
        uint16_t i = 0;
        short value;

        /* no file */
        if(!filename)
                return 1;

        /* file already treat */
        if(*post_data)
                return 1;

        /* counting filename size */
        while(filename[i++] != '\0');

        /* allocating memory */
        struct file_t *file = mem_alloc(sizeof(struct file_t));
        if(!file)
                return 1;
        file->filename = mem_alloc(i * sizeof(char));
        if(!file->filename)
                return 1;

        /* copying filename */
        i = 0;
        do{
                file->filename[i] = filename[i];
        }while(filename[i++] != '\0');

        /* counting and saving characters */
        i = 0;
        while((value = in()) != -1)
                i++;
        file->size = i;

        /* saving adress memory in post data */
        *post_data = file;

        return 1;
}

static char doPostOut(uint8_t content_type, void *data) {
        uint16_t i;
```

```
        if(data){
                /* printing data */
                out_str("The file \"");
                out_str(((struct file_t*)data)->filename);
                out_str("\" contains ");
                out_uint(((struct file_t*)data)->size);
                out_str(" characters.");

                /* counting filename size */
                i = 0;
                while(((struct file_t*)data)->filename[i++] != '\0');

                /* cleaning memory */
                mem_free(((struct file_t*)data)->filename,i * sizeof(char));
                mem_free(data,sizeof(struct file_t));
        }
        else
                out_str("No data file");
        return 1;
}
```

The doPostIn function has 4 parameters :

- `uint8_t content_type` : the number of the content-type that have been parse by Smews. His number signification is available in the file core/defines.h after the compilation of SMEWS.
- `uint8_t part_number` : number of actual part with multipart data (0 if one part).
- `char *filename` : filename of current file.
- `void **post_data` : data flowing between dopostin and dopostout functions.

The doPostOut function has 2 parameters and they are similar to parameters of doPostIn function. In the XML, the content-types tag must be specified, but can be empty. In this case, all types accepted in the tools/mimesListPost can be receive by the application.
Warning : the application must manage the allocation and freeing memory of `post_data`. Moreover, if the user want to use `filename`, he must copy it and must not forget to free after use.

Second example with several files

```
/*
<generator>
        <handlers doPostOut="doPostOut" doPostIn="doPostIn"/>
        <content-types>
                <content-type type="text/plain"/>
        </content-types>
</generator>
*/

/* file structure */
struct file_t {
        char *filename;
        uint16_t size;
};
```

```
/* dopostin function to operate post data */
static char doPostIn(uint8_t content_type, uint8_t part_number,
char *filename, void **post_data) {
        uint16_t i = 0;
        short value;

        /* no file */
        if(!filename)
                return 1;

        /* too many files (only 3 managed) */
        if(part_number > 2)
                return 1;

        /* allocating files structure if necessary */
        if(!*post_data) /*post_data used like tab*/
                *post_data = mem_alloc(3 * sizeof(struct file_t));
        if(!*post_data)
                return 1;

        /* counting filename size */
        while(filename[i++] != '\0');

        /* allocating filename memory */
        ((struct file_t *)*post_data)[part_number].filename = mem_alloc(i * sizeof(char));
        if(!((struct file_t *)*post_data)[part_number].filename)
                return 1;

        /* copying filename */
        i = 0;
        do{
                ((struct file_t *)*post_data)[part_number].filename[i] = filename[i];
        }while(filename[i++] != '\0');

        /* counting and saving characters */
        i = 0;
        while((value = in()) != -1)
                i++;
        ((struct file_t *)*post_data)[part_number].size = i;

        return 1;
}

static char doPostOut(uint8_t content_type, void *post_data){
        if(post_data){
                uint8_t j;
                for(j = 0 ; j < 3 ; j++){
                        if(!((struct file_t *)post_data)[j].filename)
                                continue;
                        uint8_t i;
                        /* printing data */
                        out_str("\nThe file \"");
                        out_str(((struct file_t *)post_data)[j].filename);
                        out_str("\" contains ");
                        out_uint(((struct file_t *)post_data)[j].size);
```

```
                        out_str(" characters.");
                        /* cleaning filename */
                        i = 0;
                        while(((struct file_t *)post_data)[j].filename[i++] != '\0');
                        mem_free(((struct file_t *)post_data)[j].filename,i * sizeof(char));
                }
                /* cleaning tab */
                mem_free(post_data,3 * sizeof(struct file_t));
        }
        else
                out_str("No data file");
        return 1;
}
```

This example uses `part_number` to manipulate several files and data are saved in a tab, pointed by post_data.

Note : multipart is necessary if you use HTML form. But if you construct packets, Smews is able to process them (for example, text/plain like content-type and all data in request post data).

3.4 Web Application SConscript

Every Web application can provide a customized SConscript file, giving specific instruction for the pre-compilation phase. For a good example take a look in the `apps/welcome` folder, containing such a script. Here, we only detail the most interesting part of this file:

```
[...]
appListName = 'appList.js'
appListPath = os.path.join(genDir,tmpBase,appListName)
appListAbsPath = os.path.join(sconsBasePath,appListPath)

appList = open(appListAbsPath,'w')
if len(dirsMap.keys()) > 2:
    appList.write('var appList = new Array();\n')
    appsCount = 0
    for dir in dirsMap.keys():
        if dir != 'welcome' and dir != httpCodesDir:
                [.. Code Missing ..] # Get Web Content Files
                [.. Code Missing ..] # Write line in .js
    # write title or empty string
    if appsCount > 0:
        appList.write('var appTitle = \"' + target.capitalize() + '
app examples:\";\n')
    else:
        appList.write('var appTitle = \"\";\n')
else:
    appList.write('var appTitle = \"\";\n')
appList.close()

ret = {appListPath: os.path.join(dirsMap['smews'],appListName)}
Return('ret')
```

The above lines iterate over the dirsMap hash map which has been built by the main Smews SCons files. It contains the set of applications being compiled with Smews (to be more precise, a set of associations between application names and paths). This code creates a `appList.js` file

with code for adding links to the main page of the `welcome` application, links which point to other applications that were also compiled along with it. The generated `applist.js` file is embedded in `index.html` of the `welcome` application and then referenced when needed to display link to applications:

```
[...]
<script type="text/javascript" src="appList.js"> </script>
[...]
```

3.5 Synthesis of the features

We give here an synthetic and exhaustive list of the features available when writing a dynamic content generator for Smews:

- Table 3.1 details the XML markers available for sue in comment of C the files that need to be considered as a Web resource and associated to an URL;
- Table 3.2 gives the list of the callbacks a dynamic Web resource can implement;
- Table 3.3 lists the functions provided by Smews that can be called in any Web applicative code.

`<generator />`
root of the applicative meta-data
`<handlers init="<funcName>" initGet="<funcName>" doGet="<funcName>" />`
callback functions definition
`<properties persistence="persistent|idempotent|volatile" interaction="rest|alert|stream" channel="<channelNem>" />`
defines the properties of the Web resource
`<args />`
defines the possible URL arguments (set of `arg` markers)
`<arg name="<str>" type="uint8|uint16|uint32|str" size="<nBytes>" />`
defines one argument
`<content-types />`
defines the possible content-types
`<content-type type="<str>"/>`
defines one content-type

Table 3.1: XML markers for dynamic contents

`char (init_app_func_t)(void)`

initialization of the applications, returning 1 if ok, 0 if ko.

`char (initget_app_func_t)(struct args_t *)`

called as soon as a get request is received, returning 1 if ok, 0 if ko. The parameter targets the parsed URL arguments.

`char (doget_app_func_t)(struct args_t *)`

called when the HTTP response is ready to be sent, returning 1 if ok, 0 if ko. The parameter targets the parsed URL arguments. This function is in charge of generating the HTTP response *via* `out_*` calls.

`char (generator_dopost_in_func_t)(uint8_t,uint8_t,char *,void **)`

called when the HTTP request is ready to be processed, returning 1 if ok, 0 if ko. The parameter are the content-type, the number of part, the filename of current file and the post data. This function is in charge of collecting posta data of HTTP request *via* `in` calls.

`char (generator_dopost_out_func_t)(uint8_t,void *)`

called when the HTTP response is ready to be sent, returning 1 if ok, 0 if ko. The parameters are the content-type and the post data. This function is in charge of generating the HTTP response *via* `out_*` calls.

Table 3.2: Functions handlers provided by Web applications

`out_c(char c)`

to be used in the `doGet` callback, thus adding the `c` byte to the HTTP response that is currently generated

`out_uint(uint16_t i)`

to be used in the `doGet` callback, thus adding the `i` integer to the HTTP response that is currently generated

`out_str(const char str[])`

to be used in the `doGet` callback, thus adding the `str` string to the HTTP response that is currently generated

`short in()`

to be used in the `doPostIn` callback, thus receiving post data

`trigger_channel(const struct output_handler_t *handler)`

usable in any function, it triggers a Comet channel, useful for alert or stream interaction modes

`set_timer(timer_func_t callback, uint16_t period_millis)`

usable in any function, it allows to automatically call the `callback` function every `period_millis` milliseconds

Table 3.3: Functions provided by Smews

Chapter 4

Creating an application for managing non-TCP packets

Smews now includes a feature called *general purpose ip handler (GPIP)*. This features allows one to develop an application that will handle non-TCP packets.

This new feature emerged from a discussion on how to correctly implement the ICMPv6 Neighbor Solicitation packets to finish the IPv6 port on the mbed ethernet target. It has been clear to us that handling those messages should neither be in core nor in target specific code. Indeed, ICMPv6 should benefit more than just a target and is outside smews main core (as smews core is designed to handle HTTP). Thus, the decision was to implement a way for a developper to add a Smews application that has 2 handler functions and associate them to a IP protocol number (e.g. 1 for ICMPv4 or 58 for ICMPv6, 17 for UDP...). The functions are for processing input packets and generate an output.

GPIP applications are standard smews applications that can define a new type of generator. Thus, they can also integrates an associated web application, although it is not mandatory. To illustrate this feature, this section will describe an application for answering ICMPv4 echo request protocol.

4.1 Declaring GPIP generators

As for a web dynamic application, a GPIP application consists in at least one **.c** file with XML metadata to describe the application generator. The metadata must declare two handlers and a protocol number.

The two handlers are called when a packet is received and when smews is ready to generate a response. The XML metadata for our application will thus be:

```
/*
<generator>
        <handlers doPacketIn="icmp4_packet_in" doPacketOut="icmp4_packet_out"/>
        <properties protocol="1" />
</generator>
 */
```

The `doPacketIn` attribute declares the function that will be called by smews when a packet is received. The `doPacketOut` attribute declares the function that will be called when smews is ready to generate an answer. In our example, the handlers are the two C functions `icmp4_packet_in` and `icmp4_packet_out`.

The protocol number is given by the `protocol` property and is set to 1 which is the protocol number of ICMPv4. This value is the **Protocol** field of an IPv4 packet and the **Next Header** field of an IPv6 packet.

4.2 Handlers

The prototype of the handler functions is as follows:

```
char handler_func(const void *connection);
```

The `connection` parameter is a handle that will allow the developer to get some information on the incoming packet such as its remote IP address, its payload size...

4.2.1 Packet information

To get information on an incoming packet, the application can use the following functions. All the functions take the connection handle as parameter.

1. `get_local_ip` : returns the ip to which the packet was sent. Yet, smews has only one IP address so it returns the IP that was defined at compile time, but a smews version that can have multiple local IP (for example with multiple interfaces) may be developed later,
2. `get_remote_ip` : returns the ip from which the packet was sent,
3. `get_payload_size` : returns the payload size of the packet (it does not include IP header size),
4. `get_protocol` : returns the protocol that triggered the call to this functions. It allows one to have the same handler for multiple protocols
5. `get_send_code` : to be used in the `doPacketOut` handler. This returns the return value of the corresponding `doPacketIn` function. This is needed because due to the smews scheduler, the `doPacketOut` is not guaranteed to be called right after the `doPacketIn` that triggered it.

4.2.2 `doPacketIn` handler

The handler is called when a packet with the corresponding protocol number is received by smews. After decoding the IP header, smews called the `doPacketIn` handler. The handler can then call the `in()` function to get the content of the packet, byte per byte.

The function must return 0 if **no** reply has to be generated. Thus, in the `doPacketIn` function returns 0, then the `doPacketOut` will **not** be called.

If an answer needs to be generated, the function can return any non-zero value. It will be possible to get this return value back when generating the reply is generated in the `doPacketOut` function use the `get_send_code()` function.

Here is a sample code that interprets an ICMPv4 echo request packet (the `checksum_*` functions are provided by smews) :

```
char icmp4_packet_in(const void *connection_info)
{
        uint8_t tmp;
        uint16_t payload_size = get_payload_size(connection_info);
        int i;

        checksum_init();
```

```
        tmp = in(); /* type */

        if (tmp != ICMP_ECHO_REQUEST)
                return 0; /* drop packet and do not generate a reply */
        checksum_add(tmp);
        tmp = in(); /* code */
        checksum_add(tmp);

        checksum[S0] = in();
        checksum[S1] = in();
        checksum_add16(UI16(checksum));

        identifier[S0] = in();
        identifier[S1] = in();
        checksum_add16(UI16(identifier));

        sequence_number[S0] = in();
        sequence_number[S1] = in();
        checksum_add16(UI16(sequence_number));
        for (i = 0 ; i < payload_size - ICMP_HEADER_SIZE; ++i)
        {
                icmp_payload[i] = in();
                checksum_add(icmp_payload[i]);
        }
        buffer_size = payload_size - ICMP_HEADER_SIZE;
        checksum_end();
        if(UI16(current_checksum) != 0xffff)
                return 0; /* invalid checksum */
        return 1;
}
```

4.2.3 `doPacketOut` handler

This handler is called by Smews when it is ready to generate a reply. This function is quite similar to a `doGet` handler. It simply generate the response packet, byte per byte, using the `out_c()` function.

In the `doPacketOut`, the `get_*` functions described earlier can still be called to get information on the incoming packet that triggered the reply. Here is an example that implements the ICMPv4 reply associated to the previously described request :

```
char icmp4_packet_out(const void *connection_info)
{
        int i;
        /* generate reply */

        out_c(ICMP_ECHO_REPLY); /* type */
        out_c(0);  /* code */

        /* Generate checksum from request one, only the type byte has change,
         * so the value can be reused to accelerate the computation of the checksum
         */
        UI16(checksum) += (ICMP_ECHO_REQUEST << 8);
        if ((UI16(checksum) >> 8) < ICMP_ECHO_REQUEST) /* overflow, should add one */
                UI16(checksum)++;
```

```
        out_c(checksum[S0]);
        out_c(checksum[S1]);
        out_c(identifier[S0]);
        out_c(identifier[S1]);
        out_c(sequence_number[S0]);
        out_c(sequence_number[S1]);
        for (i = 0 ; i < buffer_size ; ++i)
        {
                out_c(icmp_payload[i]);
        }
        return 0;
}
```

4.2.4 Initiating packet generation

Sometimes, you need to write a GPIP application that must initiates the packet generation. That is, outputing a packet is not done after receiving one, but when the application needs it (at smews start, every x seconds, ...).

The right way to implement this is to call the request_packet_out_call function. This function takes two parameters: the protocol number and the destination ip address.

When you call this function Smews registers the `doPacketOut` and will call it when it fits its scheduling rules (which will often be immediatly). Your doPacketOut handler will then perform as usual, and you will be able to create your packet using out_c function. It will automatically be sent by smews to the IP specified in the request_packet_out_call call.

Note that you have to call request_packet_out_call for each packet.

4.3 Synthesis of the features

We give here an synthetic and exhaustive list of the features available when writing a GPIP generator in Smews:

- Table 4.1 details the XML markers available for sue in comment of C the files that need to be considered as a GPIP generator,
- Table 4.2 gives the list of the callbacks a GPIP generator can implement,
- Table 4.3 lists the functions provided by Smews that can be called in any GPIP application.

`<generator />`
root of the applicative meta-data
`<handlers init="<funcName>" doPacketIn="<funcName>" doPacketOut="<funcName>" />`
callback functions definition
`<properties protocol="<protocolNumber>" />`
defines the protocol number associated to the GPIP generator

Table 4.1: XML markers for GPIP application

`char (init_app_func_t)(void)`

initialization of the applications, returning 1 if ok, 0 if ko.

`char (do_packet_in_func_t)(struct args_t *)`

called as soon as a packet is received, returning non zero value if the `doPacketOut` callback has to be called.

`char (do_packet_out_func_t)(struct args_t *)`

called when the response is ready to be sent. This function is in charge of generating the response *via* `out_c` calls.

Table 4.2: Functions handlers provided by GPIP applications

`out_c(char c)`

to be used in the `doPacketOut` callback, thus adding the `c` byte to the response that is currently generated

`short in()`

to be used in the `doPacketIn` callback, thus receiving packet data

`unsigned char *get_local_ip(const void *connection, unsigned char *ip)`

Gets the local ip associated to a packet. Can be called in `doPacketIn` and `doPacketOut`.

`unsigned char *get_remote_ip(const void *connection, unsigned char *ip)`

Gets the remote ip associated to a packet. Can be called in `doPacketIn` and `doPacketOut`.

`unsigned char *get_payload_size(const void *connection)`

Gets size of the packet (without IP header). Can be called in `doPacketIn` and `doPacketOut`.

`char get_send_code(const void *connection)`

Gets the return value of the `doPacketIn` callback. Can be called in `doPacketOut`.

`void request_packet_out_call(unsigned char protocol, unsigned char *ip)`

request one call to `doPacketOut` callback to send a packet to `ip`.

Table 4.3: Functions provided by Smews

Chapter 5

Credits

5.1 Contributors

Simon Duquennoy is the main author of Smews

Thomas Soëte wrote the WSN430 port and the MBED SLIP port

Geoffroy Cogniaux wrote the FunCard7 port

Alex Negrea is the main author of this user guide and wrote the TLS implementation (only available in the svn repos)

Geoffrey Chavepeyer and **Fabien Duchêne** implemented the IPv6 support

Emilien Hidden and **Olivier Szika** implemented the support for POST requests

Thomas Vantroys wrote the Arduino port

Michaël Hauspie wrote the MBED ethernet port, the GPIP feature and the ICMPv4 and ICMPv6 applications

Jean-François Hren designed the Smew (bird) in the Smews logo ;)

Thanks to **Gilles Grimaud** for his kind supervision and his wise advices.

5.2 License

Smews is under CeCILL license (`http://www.cecill.info/`) compliant with the GPL licence of SOSSE and TUN-TAP. It also includes part of the softwares listed below:

- SOSSE: Matthias Brüstle – http://www.mbsks.franken.de/sosse/
- TUN-TAP: Maxim Krasnyansky – http://vtun.sourceforge.net/tun/