DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

800 ANNI 1222·2022

UNIVERSITÀ DEGLI STUDI DI PADOVA

W&M

SIGNET

signet.dei.unipd.it

Development of a DESERT Addon and packet header

UNWiS - Padova (Italy)

30th of January – 3rd of February 2023

Filippo Campagnaro, Roberto Francescon, Angela Soldà, Michele Zorzi

filippo.campagnaro@unipd.it

DESERT structure

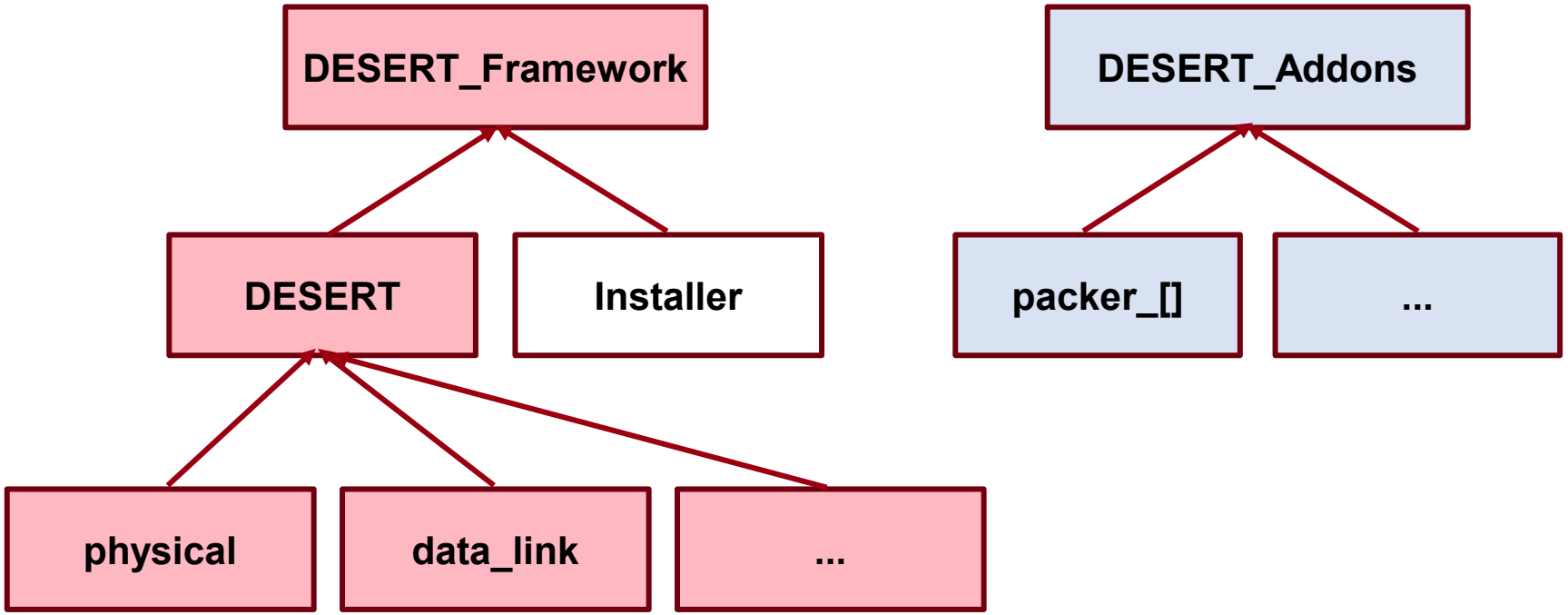
Architecture of the DESERT software stack and modules and addons organization

NS-MIRACLE inheritance

- DESERT inherits the modular structure of NS-MIRACLE
- A general `Module` is implemented, along with most basic layers (`MMac`, `MPhy`), inheriting from this `Module`
- Before delving into development, check for basic functionalities: they could be already implemented

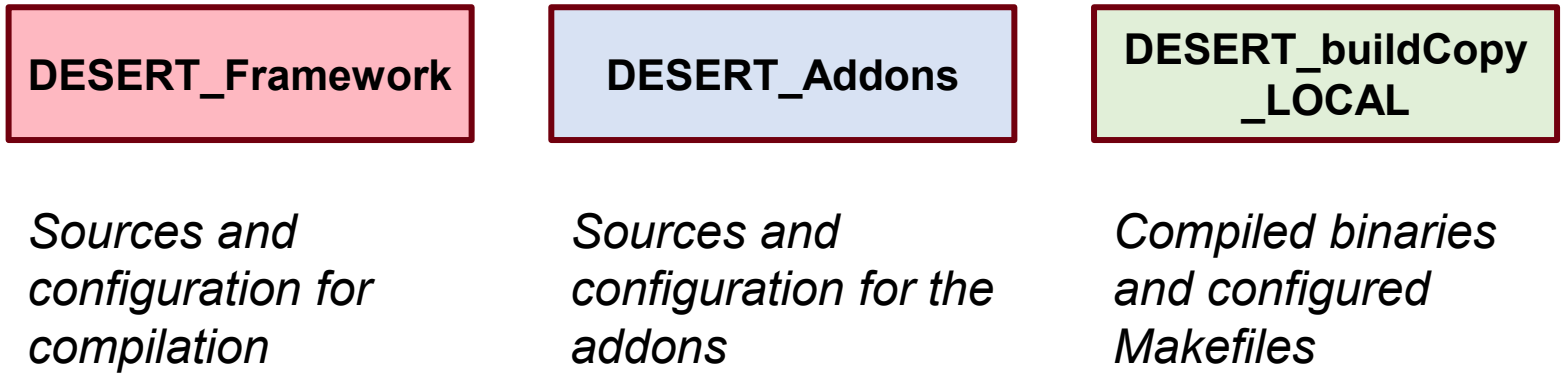
Folders organization

DESERT, as it is downloaded from the main repo, is organized in the following folders



Folders organization

- Once compiled, DESERT will create another folder named `DESERT_buildCopy_LOCAL` at the root of the DESERT repo
- This folder contains the compiled elements and the generated `Makefiles`: from here it is possible to compile a single module



DESERT Addons

Features that do not fit in the core DESERT structure

What are Addons?

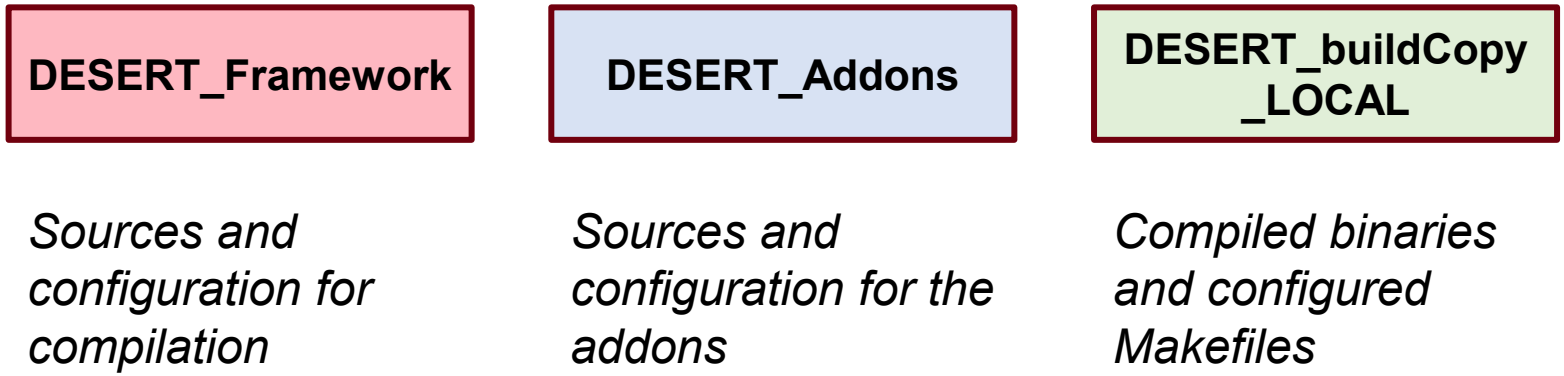
- An addon is a module that does not fit into the core DESERT structure
- an addon is an external *autotools* project that depends on DESERT, NS-MIRACLE and, possibly, WOSS
- we need to create the addon internal `Makefile.am` and a dedicated `configure.ac` with `m4` folder to resolve the dependencies
- we don't need to modify the DESERT top-level `Makefile.am` nor `configure.ac`

Addon-module relationship

- A module will depend on some other modules or classes of DESERT and will be compiled with each DESERT installation
- An addon relies on some existing module or classes, too, but its installation can be skipped, if not needed
  - for example, a common case is that of an addon defining a required new packer for a protocol
  - during installation, DESERT will ask you which addons you want to install

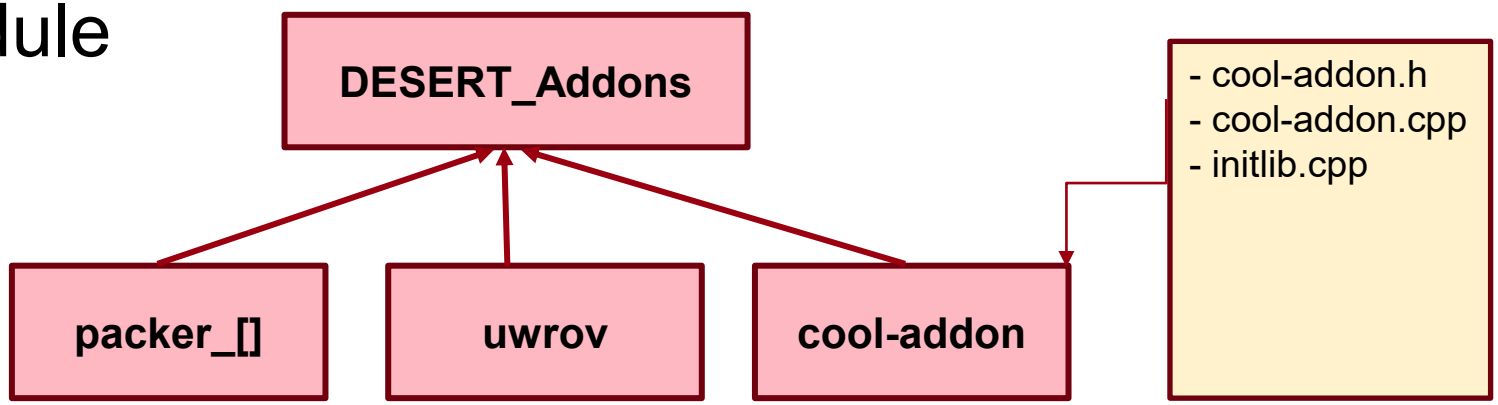
Where addons live

- The space where addons live is the `DESERT_Addons` folder, seen previously
- At the top-level of DESERT folders hierarchy



Basic addon structure

- Inside the `DESERT_Addons` folder, create your addon folder, say `cool-addon`
- Create, inside this folder, the basics files for a properly C++ module: `cool-addon.h` and `cool-addon.cpp`
- Create also the `initlib.cpp` file, as for the module



Additional files

- To make the addon compilable and complete we need to add few more files
  - a `Makefile.am`
  - an initialization file for the Tcl: `cool-addon-init.tcl`
  - the `autogen.sh` script
  - The `initlib.cpp` file
  - a `configure.ac` config file
  - the `m4` folder

These files will be explained in the next slides

The files for *autotools*

- The `Makefile.am` can be taken from another addon or module and modified according to your addon needs, similarly to the module
- For the `configure.ac` we follow a more structured process
  - We take one from another addon and copy into our folder
  - Modify the AC_INIT with the name of your addon: `AC_INIT(cool-addon, 1.0.0)`
  - In case of WOSS, it is better to take the `configure.ac` from another WOSS-based addon

The *initlib*

- For the initlib.cpp file, we proceed as with a module

```
#include <tclcl.h>

extern EmbeddedTcl CoolAddonTclCode;

extern "C" int
Cooladdon_Init()
{
    CoolAddonTclCode.load();
    return 0;
}
```

The Tcl initialization script

- To get rid of the Tcl compiler warnings is useful to have a Tcl variable initialization script:

  `cool-addon-init.tcl`

- Add all the variables that you have *bind*ed in your addon class

```
UW/PHY/CoolAddon set SN_Bits             32
UW/PHY/CoolAddon set err_Bits            0
UW/PHY/CoolAddon set Modulation_Bits     0
UW/PHY/CoolAddon set Traffic_Type_Bits 0
UW/PHY/CoolAddon set debug_              0
```

The *autogen* script

- The `autogen.sh` file is needed to compile your addon
  - copy an `autogen.sh` from another addon: it is the fastest and safest way
  - it does not need further modification

The *m4* folder

- It is easier to copy an m4 folder from another addon: it contains the `desert.m4` file
- we need to modify it, specifying the dependencies we need for our addon
  - list in `dir` and in `lib` the directories and libraries your addon uses

```
for dir in   \
      physical/uw-al    \
      network/uwip    \
      transport/uwudp   \
      application/uwcbr
do
```

```
for lib in  \
      uwal   \
      uwip   \
      uwudp  \
      uwcbr
do
```

Make DESERT aware

To have our addon visible during DESERT installation, so that the Makefile installation flags are setup by the DESERT installer, we modify an installation script: `DESERT_Framework/.addon.list`

We just need to add the name (`cool-addon`) of the folder to the list

Set up and ready

These are the basic steps needed to create a new working addon in DESERT.

After you laid out these basic building blocks, go on with an **installation of** the entire **DESERT** and **mark the new addon for installation**: this will allow the creation of all the files needed in compilation.

After this step, you can reach

```
DESERT_buildCopy_LOCAL/.buildHost/DESERT_ADDON/cool-addon
```

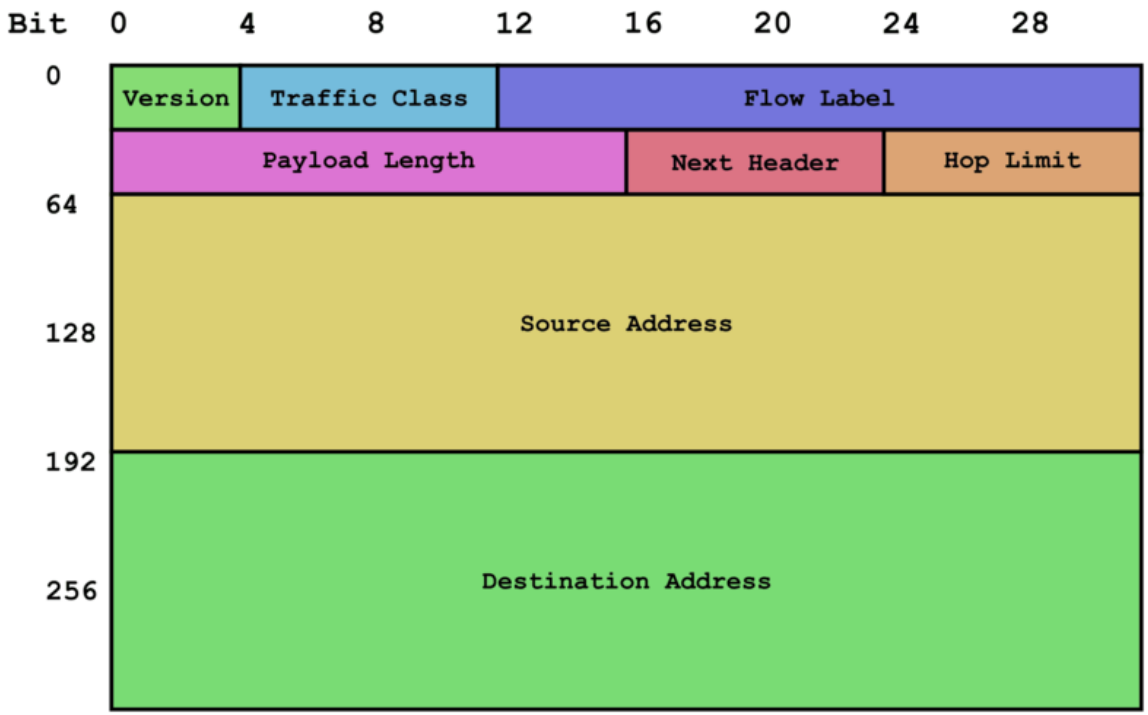
and recompile the single addon, when needed.

Creating a new header

How to define and implement a new header for your packets

Headers

DESERT is sending **packets** to-and-fro. **Packets** have **headers**. **Headers** are powerful and critical tools for managing communications **networks**.



The header structure

A header in DESERT can be defined as one prefers through a class or a `struct`: quite always, a `struct` is enough

```
#include <packet.h>

typedef struct hdr_uwmyprotocol
{
  uint8_t param1_; /**< Parameter used by the protocol */

  static int offset_; /**< Required by the PacketHeaderManager */

  uint8_t getParam(){return param1_;}
  ...

} hdr_uwmyprotocol_instance;
```

myproto-hdr.h

The offset field [1]

The offset field, along with its getter method, is necessary for the header manager so it knows how many bits to skip to reach the correct header

```
#include <packet.h>

typedef struct hdr_uwmyprotocol
{
    ...
    inline static struct hdr_uwmyprotocol *
    access(const Packet *p)
    {
        return (struct hdr_uwmyprotocol *) p->access(offset_);
    }
...
} hdr_uwmyprotocol_instance;
```

myproto-hdr.h

The offset field [2]

This steps are necessary to allow easy integration with preexisting structures: now every piece of DESERT will know your new header

```
#include <tclcl.h>
#include <myproto-hdr.h>

static class MyProtoHeaderClass : public PacketHeaderClass
{
public:
    MyProtoHeaderClass()
        : PacketHeaderClass("PacketHeader/MYPROTO",
                            sizeof(hdr_uwmyprotocol))
    {
        this->bind();
        bind_offset(&hdr_uwmyprotocol::offset_);
    }
}
```

Inherits from PacketHeaderClass

Here the size of your header is set and binded with offset

initlib.cpp **OR** myproto.cpp

The header type

Now what is left to do is to define the type, *at the beginning* of the header `.h` file

- A type for each new header defined

```
extern packet_t MY_PACKET_TYPE;
```

- these header files are usually put in the same folder as the module
- our new header will be linked al visible throughout ns2 (and, so, DESERT)

Access macro

It is requested also to define the function (via `macros`) that allows to access the various fields of the new header

```
#define HDR_UWMYPROTOCOL(p) (hdr_uwmyprotocol::access(p))
```

myproto-hdr.h

to be put at the beginning of the header file, along with the pre-processor directives

Putting everything together

- After having declared and defined the header struct, implementing everything that is needed
  - declare your new packet type (`extern packet_t`)
  - define the access macros
  - define fields and methods of the header
  - link with `PacketHeaderClass` and bind offset

Then it is possible to use the new header

```
hdr_uwmyprotocol *myh = HDR_UWMYPROTOCOL(p);

uint8_t param = myh->getParam();
myh->param1 = calcParam();
```

Summarizing

```
#define HDR_UWMYPROTOCOL(p) (hdr_uwmyprotocol::access(p))

extern packet_t MY_PACKET_TYPE;

typedef struct hdr_uwmyprotocol
{
    /** rememeber to manage the offset_ field */
}
```

myproto-hdr.h

```
static class MyProtoHeaderClass : public PacketHeaderClass
{
  /** remember to manage inheritance from PacketHeaderClass*/
}
```

initlib.cpp