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 Module

UNWiS - Padova (Italy)

30th of January – 3rd of February 2023

Filippo Campagnaro, Roberto Francescon, Angela Soldà, Michele Zorzi

filippo.campagnaro@unipd.it

DESERT structure

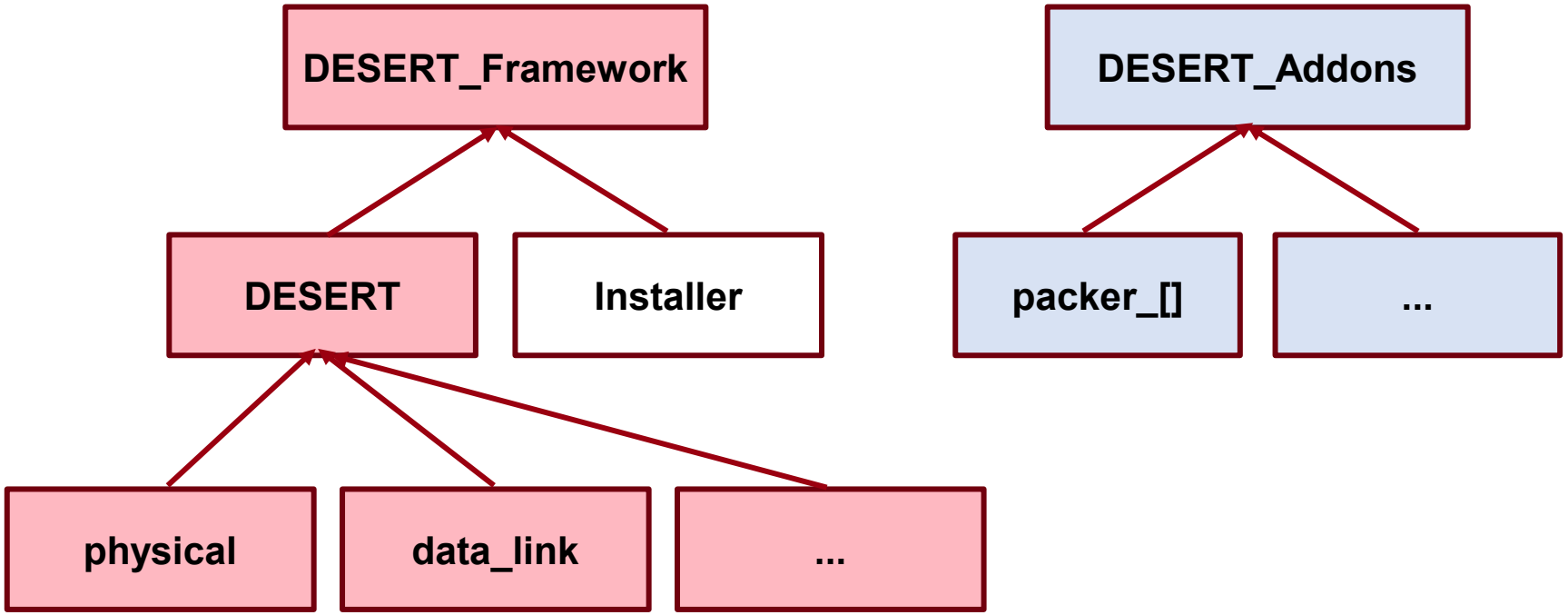
Architecture of the DESERT software stack and modules organization

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

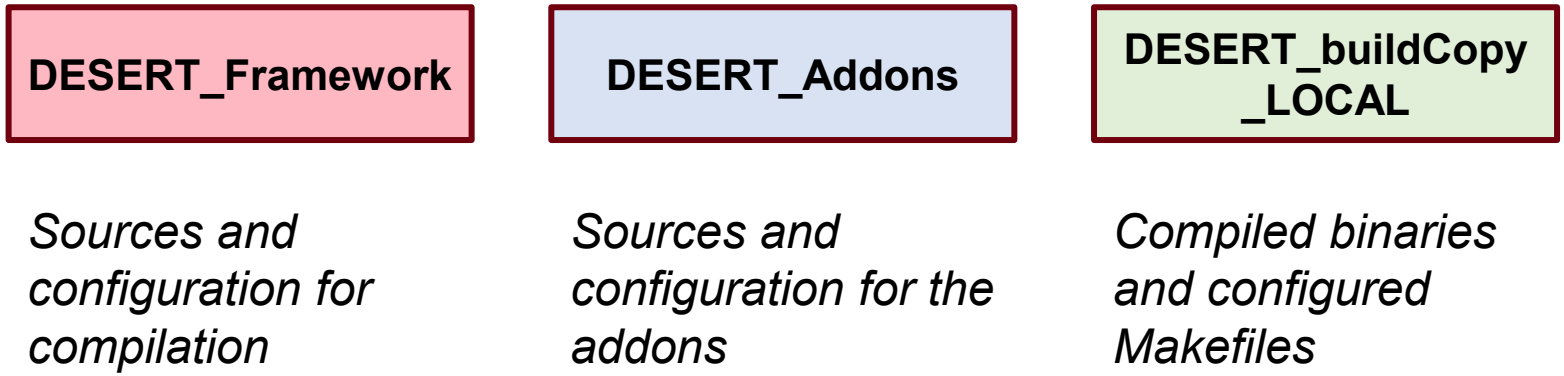
Folders organization

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 Modules

How to develop and add a new module in DESERT Underwater

Where to put them

- As seen, DESERT modules lie in `DESERT_Underwater/DESERT_Framework/DESERT`
- DESERT organizes modules according (when possible) to the ISO/OSI stack: understand and decide where your new module should be placed
- As an example, we want to implement a new physical layer module, called *Cool PHY*

Cool PHY basics

- We start by creating the folder to host our new PHY module: `uwcoolphy`

```
DESERT_Framework/DESERT/physical/uwcoolphy
```

- then, we proceed to create the 2 basic files needed by C++ to define (properly) a class, and so, an object: `uwcoolphy.h`, `uwcoolphy.cpp`
- add another important file, that will be explained later: `initlib.cpp`
- and add a `Makefile.am`, specifying the compilation of the module

The module class

- Our module, is a layer module (PHY), so it can be inserted in the communication stack of the nodes in our simulation
- It is implemented through a C++ class, that we call `UwCoolPhy`
- For this class to work in a DESERT simulation, some types of binding are necessary
  - Bound variable (`bind`)
  - Constructor connection with OTcl

Module Tcl binding [1]

- The first binding links variables for, e.g., setting parameters, via the function `bind()`

```
C++  |  bind("debug", (int *) &debug_);

Tcl  |  Module/UW/PHYSICAL set debug 1
```

- the string parameter is the Tcl variable that is bound (has the same value of) to the second parameter (a C++ declared variable)
- and is usually done in the C++ constructor

Module Tcl binding [2]

- The second binding is performed at the Constructor and allows to instantiate it in Tcl
- It is built, defining a class and its constructor

```
static class UwCoolPhyClass : public TclClass
{
public:
    UwCoolPhyClass()
      : TclClass("Module/UW/CoolPhy")
    {  }
    TclObject * create(int, const char *const *) {
        return (new UwCoolPhy);
    }
}
```

Name of the C++ class plus "Class"

Name of the module in the Tcl script

Name of the C++ class (contructor)

The *initlib.cpp* file

- to complete our *Cool PHY* module, we define all the methods implementing the features we want
- we need to fill in also the `initlib.cpp`, which makes the real connection between C++ and OTcl
  - define an extern variable (it represents an EmbeddedTcl object) with a specific name structure: `<ModuleNameInCamelCase><TclCode>`
  - define a function with fixed naming, which is the name of your module with the first capital letter and `_Init` suffix

```
#include <tclcl.h>
extern EmbeddedTcl UwCoolPhyTclCode;

extern "C" int
Uwcoolphy_Init()
{
    UwCoolPhyTclCode.load();
    return 0;
}
```

Initializing Tcl variables

- It is suggested to initialize all the Tcl variables you have *bind*ed in your C++ class
- To this end, we create a file ending in <>-*init.tcl*: `uwcoolphy-init.tcl`
- There, we can initialize the variables via Tcl code

```
Module/UW/CoolPhy set SN_Bits            32
Module/UW/CoolPhy set err_Bits           0
Module/UW/CoolPhy set Modulation_Bits    0
Module/UW/CoolPhy set Traffic_Type_Bits 0
Module/UW/CoolPhy set debug_             0
```

Make the module to compile [1]

To be able to compile our module, we need to fill in our `Makefile.am`

- add the definition of your library: name of the library must always be `lib` prefix + name of the folder hosting the module

```
lib_LTLIBRARIES = libuwcoolphy.la
```

- Specify the sources

```
libuwcoolphy_la_SOURCES = initlib.cpp \
          <TABS>          uwcoolphy.cpp
```

- Remember to define the CPPFLAGS, LDFLAGS and LIBADD as needed
- Define the Tcl files variables defaults like this

```
TCL_FILES =  uwcoolphy-init.tcl
```

Make the module to compile [2]

We also need to modify the top-level `configure.ac` and `Makefile.am`

configure.ac

Add the folder of your module to the CPPFLAGS

```
DESERT_CPPFLAGS="$DESERT_CPPFLAGS "'-I$(top_srcdir)/physical/uwcoolphy'
```

Add your module's Makefile to the AC_CONFIG_FILES

`physical/uwcoolphy/Makefile`

Makefile.am

Add module's folder to the list of SUBDIRS

`physical/uwcoolphy`

Using the module

- In a Tcl simulation script, we are going to use our module
  - first, by setting its parameters
  - then, instantiating an object
  - and connecting, when needed, or calling the necessary methods on it

```
Module/UW/CoolPhy  set BitRate_                    32500
Module/UW/CoolPhy  set AcquisitionThreshold_dB_    10.0

set  phy($id)   [new Module/UW/CoolPhy]

$node($id) addToChannel    $channel   $phy($id)   1
$phy($id)  setSpectralMask $data_mask
```

The *command* method [1]

There exists another way to communicate between C++ and Tcl scripts: the *command* method

This method parses strings, passed via Tcl, that are interpreted as commands, and can then call C++ functions (its binding is already defined and inherited)

It must be declared in this way, in your module class

```
virtual int command(int, const char *const *);
```

The *command* method [2]

Then, it should be defined in this way

```
UwCoolPhy::command(int argc, const char *const *argv) {
    if (argc == 2) {
        if (strcmp(argv[1], "Method1") == 0) {
            method1();
            return TCL_OK;
        }
    }
    if (argc == 3) { ...
    }
    return Module::command(argc, argv);
}
```

Process first based on number of parameters and then based on string comparison. Remember to return `TCL_OK` if string comparison returns 0, and to call the original (father) command at the end of your module's command:

```
return Module::command(argc, argv);
```