DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

800 ANNI 1222·2022

UNIVERSITÀ DEGLI STUDI DI PADOVA

SIGNET

signet.dei.unipd.it

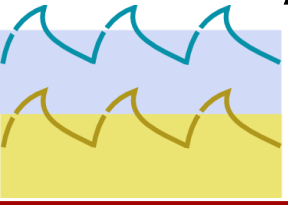
Schedulers and packers

UNWiS - Padova (Italy)

30th of January – 3rd of February 2023

Filippo Campagnaro, Roberto Francescon, Angela Soldà, Antonio Montanari, Michele Zorzi

filippo.campagnaro@unipd.it



Schedulers and clocks

The structure of events scheduling and timers and how to define your *handlers*

Basics of ns2

- DESERT directly inherits its simulator inner structure from *ns2*
- *ns2* is based on a linked list of Events
- The Event is the node (of this linked list) and possess also its *handler*: the *event consumer*

```
class Event {
public:
    Event* next_;  /* event list */
    Event* prev_;
    Handler* handler_;   /* handler to call when event ready */
    double time_;   /* time at which event is ready */
    scheduler_uid_t uid_;   /* unique ID */
    Event() : time_(0), uid_(0) {}
};
```

Event and handler

- An *Event* abstracts exactly what the name suggests
- It has:
  - a time of happening
  - the link to the previous and
  - the next Event
  - the Handler reference
- The *Handler* is a class that takes care of doing what the Event is supposed to perform: it *consumes* the event through the method *handle*

Scheduler

- Manager of Events, sort of clock or calendar
- An idea of what it does can be seen directly from declaration and its methods

```
class Scheduler : public TclObject {
public:
    static Scheduler& instance() {
        return (*instance_);     // general access to scheduler
    }
    void schedule(Handler*, Event*, double delay);   // sched later event
    virtual void run();        // execute the simulator
    virtual void cancel(Event*) = 0;    // cancel event
    virtual void insert(Event*) = 0;    // schedule event
    virtual Event* lookup(scheduler_uid_t uid) = 0;    // look for event
protected:
       void dispatch(Event*);    // execute an event
       void dispatch(Event*, double);    // exec event, set clock_
       double clock_;
...
```

Scheduling events

- The scheduler has its own time reference: `clock_`
- Scheduling an Event implies:
  - setting its Handler (what should be done)
  - increasing the ID count and assigning it to the Event
  - setting the Event's `time_`: the time at which the Handler must be called, and the Event consumed (`clock_ + delay`)
- `Scheduler::dispatch()` really just calls the Handler for the Event
  - the `clock_` is then updated to dispatch time
- The Scheduler runs until it is stopped

RealTime is different

- The process of dispatching Events in the Scheduler is the same in all schedulers
- Time management is what changes in RealTime scheduler
- The `clock_` variable of the RealTime scheduler is constantly set with the current time of the host machine: `clock_ = tod();`
  - as usual, after each consumed event
- And, as usual, the time of an Event is checked against `clock_`

Using schedulers in Tcl

- in a simulation, we are going to use a Calendar scheduler (default)
- for connecting DESERT to actual modems, we need to use the RealTime scheduler

```
$ns use-scheduler RealTime
```

If you want to use a different scheduler:

```
$ns use-scheduler <scheduler-name>
```

This is all that is required to use the different schedulers

Built-in scheduler and handlers

The simulation events processing has built-in scheduler and handlers to manage common tasks to all simulations

Packets inherits from `Event` and, when `Packet p` is created and sent down the stack, it is scheduled

```
Scheduler::instance().schedule(this, p, delay);
```

This is performed inside the instance inheriting from Handler, that manages packets traces along the stack

`delay` must be calculated based on the protocols

Defining handlers in C++

We can define handlers (not schedulers!) in our C++ code, that can help us in programming on event-based protocols

```
class MyModuleTimer : public TimerHandler
{
public:
    MyModuleTimer(MyModule *m)
      : TimerHandler()
    {
        assert(m != NULL);
        module = m;
    }

protected:
    virtual void expire(Event *e);
    MyModule *module;
}
```

Derive your new timer from *TimerHandler*: it is less error-prone and there are features already implemented

MyModule is the module inside which this new timer is declared and defined: you want to act on *MyModule*, ususally

This really is the function you need to define: here you specify what your protocol does in case something else happens

An example: TDMA

TDMA is a good example as it is a time-based protocol

```
void
UwTDMATimer::expire(Event *e) {
    ((UwTDMA *) module)->changeStatus();
}

UwTDMATimer tdma_timer;
```

Where the `changeStatus` sets the node time slot

```
if (slot_status == UW_TDMA_STATUS_MY_SLOT) {
        slot_status = UW_TDMA_STATUS_NOT_MY_SLOT;
        tdma_timer.resched(frame_duration - slot_duration + guard_time);
...
```

Packers

Going physical: what packers are and what purpose they serve

What packers are

- Packers are DESERT modules used to translate from virtual packets, used in simulations, to real packets, to be transmitted on the channel
- The `packer` class is defined in the A*daptation Layer (AL)* module folder and inherits from `TclObject`: provides the methods to pack and unpack virtual packets and their headers
- The AL class is derived from `MPhy` and provides methods to fragment and re-assemble the packets

```
class Uwal : public MPhy
{
    packer *pPacker;
}
```

What packers achieve

The packer module allows to move from the simulation to the real-field trial. It resides in the Adaptation Layer folder and is responsible for forging the header and the payload for the actual packet

It maps an NS-Miracle packet into a legal modem payload (i.e., a string of binary characters)

```
std::string packPayload(Packet *);
std::string packHdr(Packet *);
```

Packers: nuts and bolts

Packers are used by the Adaptation Layer module

```
packer *pPacker;          // uwal.h
pPacker->packHdr(p);      // uwal.cpp
pPacker->packPayload(p); // uwal.cpp
```

The bitstream is written directly inside the same packet for which translation was requested

```
unsigned char *buf = new unsigned char[hdr_length];
packMyHdr(p, buf, offset);
hdr_uwal *hal = HDR_UWAL(p);
memcpy(hal->binPkt(), buf, hdr_length);      // packer.cpp
```

The AL then proceeds to send down the translated packet

```
sendDown(p);      // uwal.cpp
```

Defining our own packers

If a new layer module is introduced, and its new header does not have a correct way of serializing, we need to define a new packer

Packer files are found in the Addons

Quite always, what you want is the correct way to serialize the new header introduced by some layer you developed:

- You don't need to rewrite payload packing
- You want to rewrite header packing/unpacking and utility methods (header fields printing)

Overwrite pack/unpack

The functions you usually need to redefine are:

- `init()`: initialization of the packer fields
- `packMyHdr(...)`: serialization functions that translates virtual fields to actual bits
- `unpackMyHdr(...)`: parser function that recovers the virtual packet fields from the bitstream

```
void init();
size_t packMyHdr(packet*, unsigned char*, size_t);
size_t unpackMyHdr(unsigned char*, size_t, Packet*);
void printMyHdrMap();
void printMyHdrFields(Packet*);
```

Example: CBR header

We push to `n_bits` the fields we need to correctly serialize and deserialize the header. They are assigned during simulation via Tcl binding

```
void packerUWCBR::init() {

    n_bits.clear();

    n_bits.push_back(SN_Bits);
    n_bits.push_back(RFTT_Bits);
    n_bits.push_back(RFTT_VALID_Bits);
    n_bits.push_back(TRAFFIC_TYPE_Bits);
}
```

Example: CBR header

In the following snippet we can see the serialization method of the CBR

```
size_t packerUWCBR::packMyHdr(Packet* p, unsigned char* buf, size_t offset)  {

    hdr_cmn* ch = HDR_CMN(p);
    hdr_uwcbr* uch = HDR_UWCBR(p);

    if ( ch->ptype() == PT_UWCBR ) {
        int field_idx = 0;

        offset += put(buf, offset, &(uch->sn_), n_bits[field_idx++]);
        offset += put(buf, offset, &(uch->rftt_), n_bits[field_idx++]);
        offset += put(buf, offset, &(uch->rftt_valid_), n_bits[field_idx++]);
        offset += put(buf, offset, &(uch->traffic_type_), n_bits[field_idx++]);

    }
    return offset;
}
```

How to use packers: setup

To correctly translates between virtual packets and bits, we need to configure and add all the packers for the various layers

```
set packer_ [new UW/AL/Packer]
set packer_payload0 [new NS2/COMMON/Packer]
set packer_payload1 [new UW/IP/Packer]
set packer_payload2 [new NS2/MAC/Packer]

$packer_ addPacker $packer_payload0
$packer_ addPacker $packer_payload1
$packer_ addPacker $packer_payload2

$uwal_ linkPacker $packer_
```

How to use packers: AL

To perform an emulation, using real modems, we need to use packers. It is sufficient to include the AL in the protocol stack

- instantiate an AL module
- add it to the protocol stack
- place it between the drivers layer and the layer above (usually a MAC layer)

```
set uwal  [new Module/UW/AL]
...
$node  addModule 2 $uwal 1 "UWAL"
...
$node  setConnection $mac $uwal
$node  setConnection $uwal $modem
```