Simulation

Faking reality:
How?

1) Pretending the causes
2) Faking the effects
3) Trying to obtain the same output (red card)

Overview

- Observing reality
- Modeling
- Implementation
- Scenario Creation

a simple scenario

a VERY simple scenario

•Processing Time

•Discard Probability

•Delay

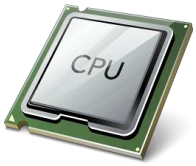
•Packet Loss

Lets consider a very simple scenario composed of two nodes. Each node receives a packet from its peer, processes it and sends it back.
The nodes are connected via a generic network, with an associated delay and packet loss.

Entities

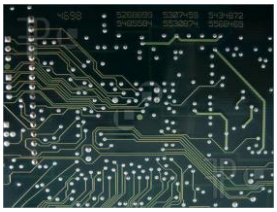
- Communicating elements



- Connections

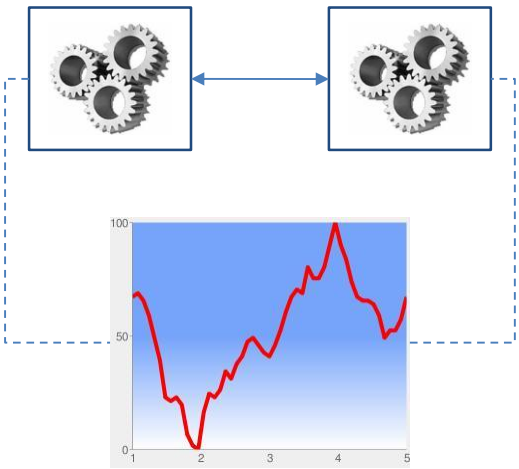




Even more complex scenarios are generally composed of communicating elements of various kind, connected through connections (single or multiple) with very heterogeneous characteristics.
Bear in mind that we are not necessarily talking about networking systems…

Modeling

- Elements
- Connections
- Operations
- Output



Our first goal is to define a model of the scenario.

- Elements: which part of our system are actually interesting **from our point of view**. (Different people/situation may decide different modeling strategies).
- Interaction between elements: interfaces, infrastructures, information exchanges
- Operations: behavior, reactions
- How we can observe the system

Example with a PC: model the CPU if you are interested in analyzing computational power; model network interfaces is you need to analyze communications.

How?

- Modular
- Event Driven
- Object Oriented
- Open Source

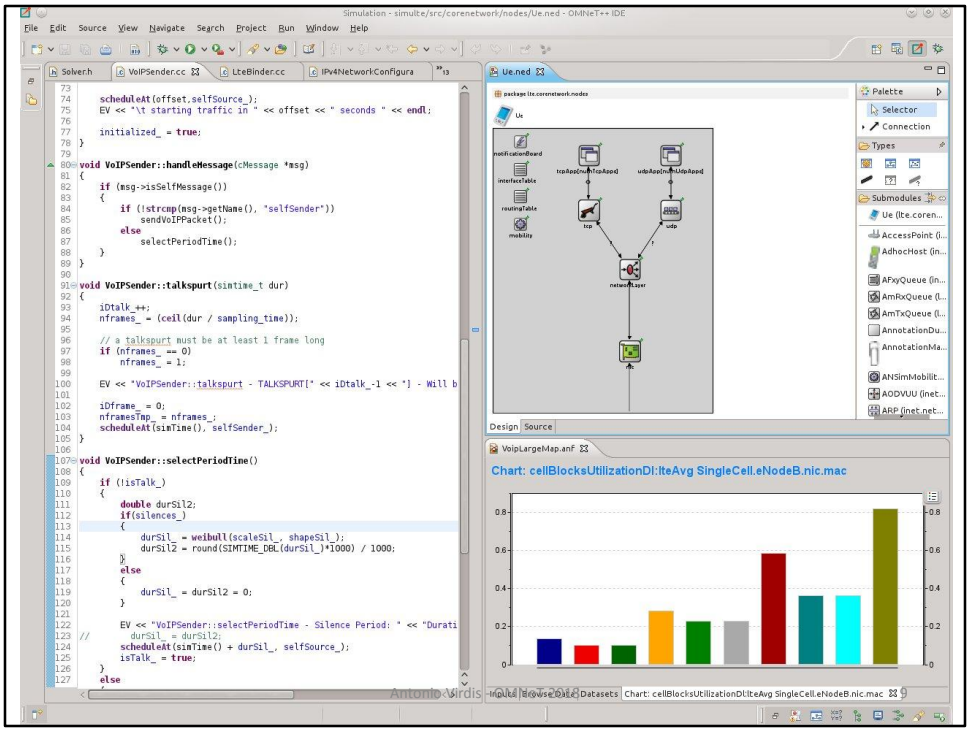
Actually omnet 5 is out now

Omnet++

”OMNeT++ is an extensible, modular, component-based C++ simulation library and framework”

- Eclipse-based IDE
- User interface
  - Graphical
  - Command Line
- Extensions available

It is very interesting example of good programming



As for Excel I'll let you "discover" the IDE by yourselves.

- Coding Part
- Structure representation
- Statistics

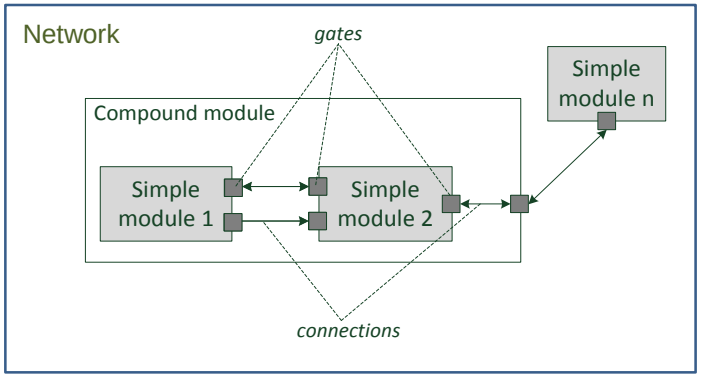
Implementation

- Structure [NED]
- Messages [C++]
- Behavior [C++]
- System definition [NED]

Our model will be defined via different files

- Structure: elements and connections between them (similarly to HTML)

Entities and Connections



- Simple Modules
- Compound Modules
- Gates
- Connections
- Network

Simple modules are the basic building blocks in the omnet architecture.
Simple modules can be composed together to build *compound modules*.
Both simple and compound modules may have input/output interfaces, called *gates*.
Connections can be build between gates, allowing modules to exchange messages.
Al these elements can be used together to build a *network*.

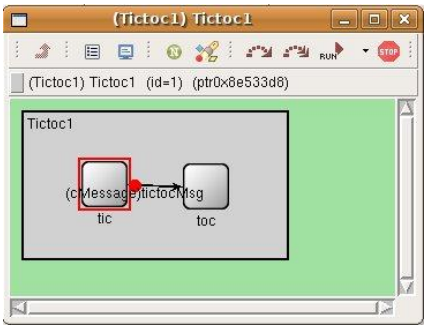
Simple Module [NED]

```
simple Txc
{
      gates:
          input in;
          output out;
}
```

The structure of a simple module is defined via NEDfiles. NED stands for NEtwork Description language.
It is very similar to declarative languages (e.g. HTML) as it defines *what* something is, rather than *how* it *behaves*.
In this case we are defining a simple node called Txc, with one input and one output interfaces.
As we will see, OMNeT follows the paradigms of Modularity and Object Oriented also from the “structure” point-of-view

Network definition

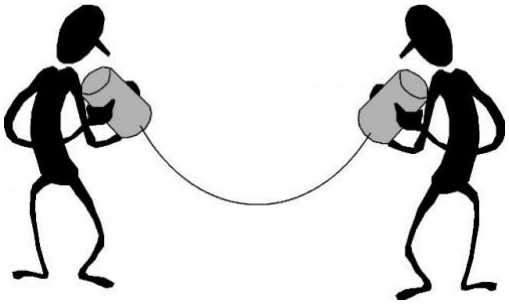
```
network Tictoc
{
    submodules:
        tic: Txc;
        toc: Txc;
    connections:
        tic.out --> { delay = 100ms; } --> toc.in;
        tic.in <-- { delay = 100ms; } <-- toc.out;
}
```

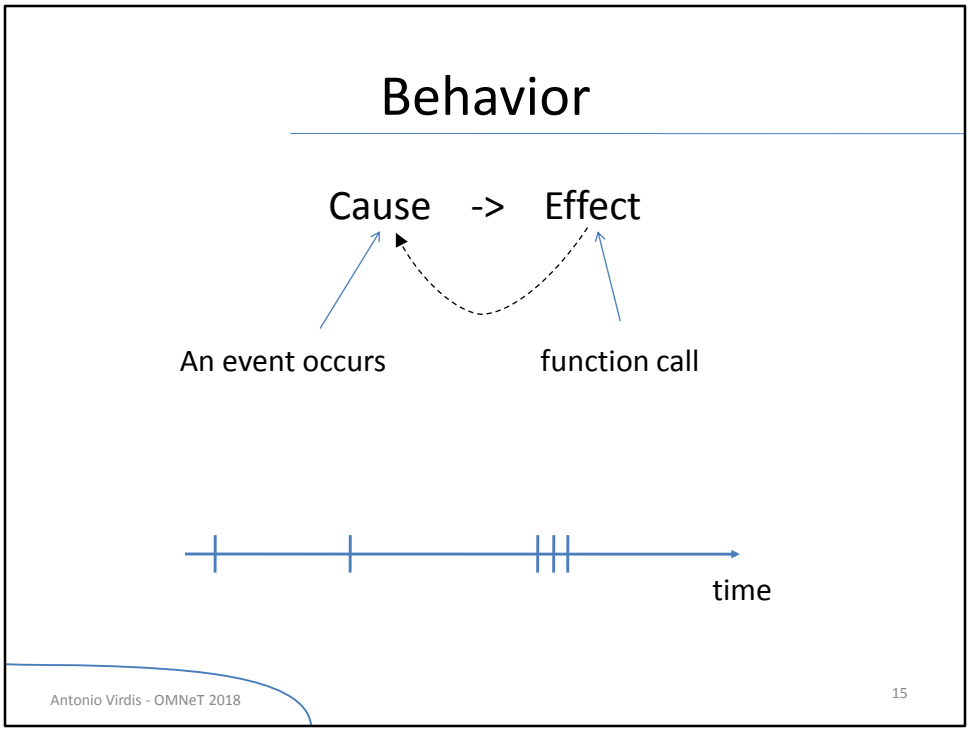


We can create two instances of the simple module Txc in order to create a network. This is done again using NED language.
1) First we define the name of the network.
2) Then we specify the modules that will compose the network itself.
3) Finally we connect the gates of the modules, specifying their connections, and possibly their characteristics (e.g. delay)

At this point, we can start our simulation and obtain a nice structure that… does nothing.
It is just like we created our two football players that are standing still in the middle of the field.
We need to add the **behavior**

Behavior

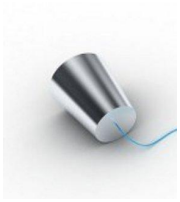




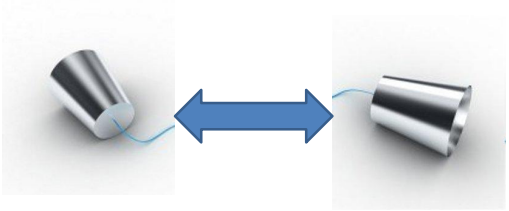
As we said, OMNeT++ is an event driven simulator.
The evolution of the system follows a cause-effect paradigm.
Events are represented as **messages** (almost always).
The behavior of modules is defined via C++ files.
Each simple module is associated with a C++ class.
Compound modules do not have associated classes. Their behavior is the summation of their composing modules
Events are placed in a timeline. After an event is managed, the next nearest event is executed.
Each event can create one or more new events: each event can add a new event to the timeline

Behavior

- Start -> Tic talks first



- Event -> answer



Example.

Behavior: events

- `void initialize()`

- `void handleMessage()`

- `void finish()`

We can have three types of events: node creation, message reception, simulation finish (from the point of view of a module).
Events are managed by event handlers. (similarly to what happens with DOM events in HTML, e.g. onLoad, onClick, etc.)
The *initialize* function is called automatically after a module is created and is generally used to perform setup operations on a module.
The `handleMessage` function is called whenever a message is received by a simple module. It is generally used to identify the message type, manage it, and create a new one.
The *finish* function is called automatically at the end of the simulation on each module. It is generally used to record statistics.

Behavior: functions

- send()

- scheduleAt()

- cancelEvent()

How can we create events?

The *send* function is used for sending out messages through an output gate
ScheduleAt and *cancelEvent* are used for managing **timers**.

Behavior [C++]

```
class Txc : public cSimpleModule {
  protected:
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};

// Module class needs to be registered with OMNeT++
Define_Module(Txc);
```


How can we realize the intended behavior?
How can we manage events?
How we can implement the cause->effect relation?

This is done automatically by the OMNeT environment.
Each module-related class has to derive at least from the cSimpleModule class.
If we use the OMNeT++ IDE commands, this is done automatically.

What is the difference between the class constructor and the initialize function? They work at two different logical levels.
The constructor deals with operations related to the C++ class, as variable initializations, creation of the data structures, etc.
The initialize function deals with OMNeT stuff and operations related to the module, as message creation and first transmission, reading parameter, etc.
This is the first example of the double nature of the module: a C++ class on one side and NED module on the other.

Initialize()

```
void Txc::initialize()
{
    if (strcmp("tic", getName()) == 0)
    {
        cMessage *msg = new cMessage("tictocMsg");

        send(msg, "out");
    }
}
```


In our example we use the initialize function to create the first event.
getName is a member function inherited from the cSimpleModule class. It returns the name of the current module.
A new object of type *cMessage* is first instantiated and then it is send through the gate *"out".*
NOTE: *out* is the name of the gate as defined in NED.
The send function will generate a new event at a point that depends on the connection parameters…

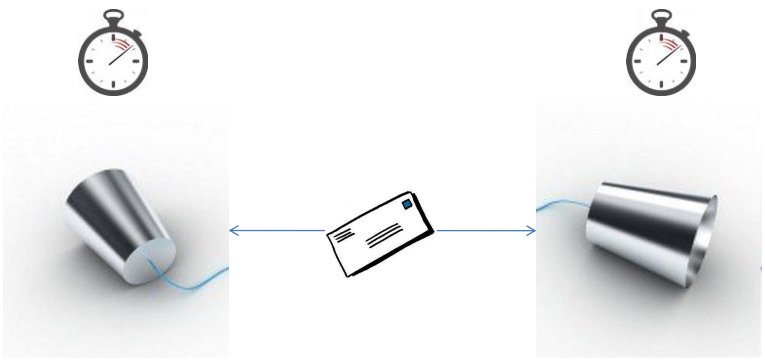
Example with a 0 delay connection.

handleMessage()

```
void Txc::handleMessage(cMessage *msg)
{
    // check message type
    […]
    // send message out
    send(msg, "out");
}
```

Timers

Timers



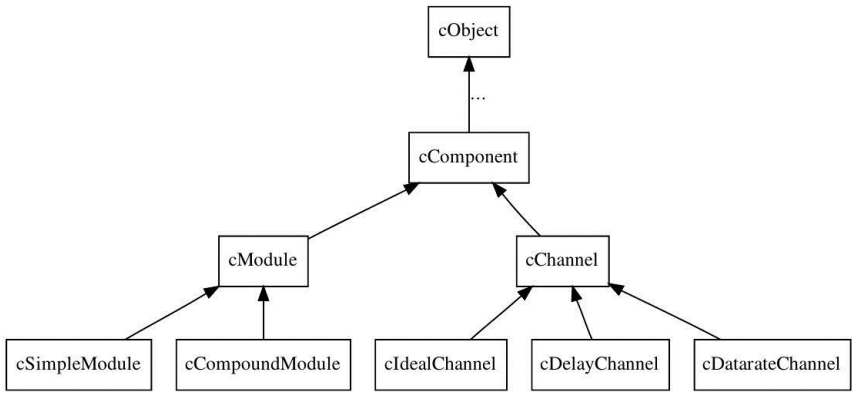
We may need to model events that will occur periodically or after a variable amount of time. Those type of events can be realized via timers.
Timers are event that are thrown on a simple module after a certain amount of time.
How can we create a timer using the available events?
A simple module can create a timer by sending a message to itself.

Timer

```
class Txc : public cSimpleModule {
  private:
       cMessage *beep;
       cMessage *tictocMsg;
...
//in handleMessage()
if (msg->isSelfMessage()) //same as msg==beep
       send(tictocMsg, "out");
else if( strcmp(msg->getName(),"tictocMsg") == 0 )
       scheduleAt( simTime() + 10, beep );
       … // store msg somewhere
```


The handleMessage function has *cMessage * msg* as actual parameter.
We have to check for the type of it as we have only one *handleMessage()* function to deal with all kind of messages.
isSelfMessage is a member function of the class cMessage which returns true if the given message was scheduled via the *scheduleAt* function.
getName is a member function of the class cMessage which returns a string containing the name of a message.
scheduleAt(time,msg) sends the message msg to this module (i.e. the same module that is calling it). The message will be received at time *time*.
A possible full implementation of the code above would require each Txc module to create its own instance of beep message, e.g. into the initialize function.
We recall that only the *tic* module has to create the "tictocMsg", thus only one instance of it will be available during execution.
Note that when the tictocMsg is recived, a pointer to it has to be stored into the tictocMsg member variable, in order to keep a reference to it and send it out in after the timer beep "expires", i.e. the handleMessage function is called with msg==beep.
What happens to that pointer *after* we send the message?

Element Hierarchy



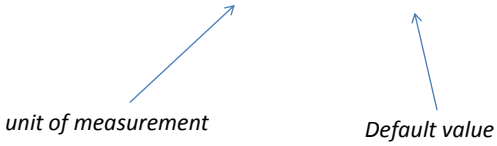
Common properties

Defining **Parameters**

```
simple Txc
{
  parameters:
      double procTime = default(10);
}
```

Defining **Parameters**

```
simple Txc
{
  parameters:
      double procTime @unit(s)= default(10s);
}
```



Specifying Parameters

[NED]

```
network Tictoc
{
    submodules:

        tic: Txc
        {
            procTime = 1s;
        }

        toc: Txc;

}
```

Reading parameters

```
//in handleMessage()
if (msg==beep)
      send(tictocMsg, "out");
else
{
     scheduleAt( simTime() + 10 , beep );
}
```


Always avoid writing values (of any kind) directly into the C++ code.

1) Hard to modify
2) Hidden default values

Reading parameters

```
//in handleMessage()
if (msg==beep)
      send(tictocMsg, "out");
else
{
      simtime_t time = par("procTime");
      scheduleAt( simTime() + time , beep );
}
```

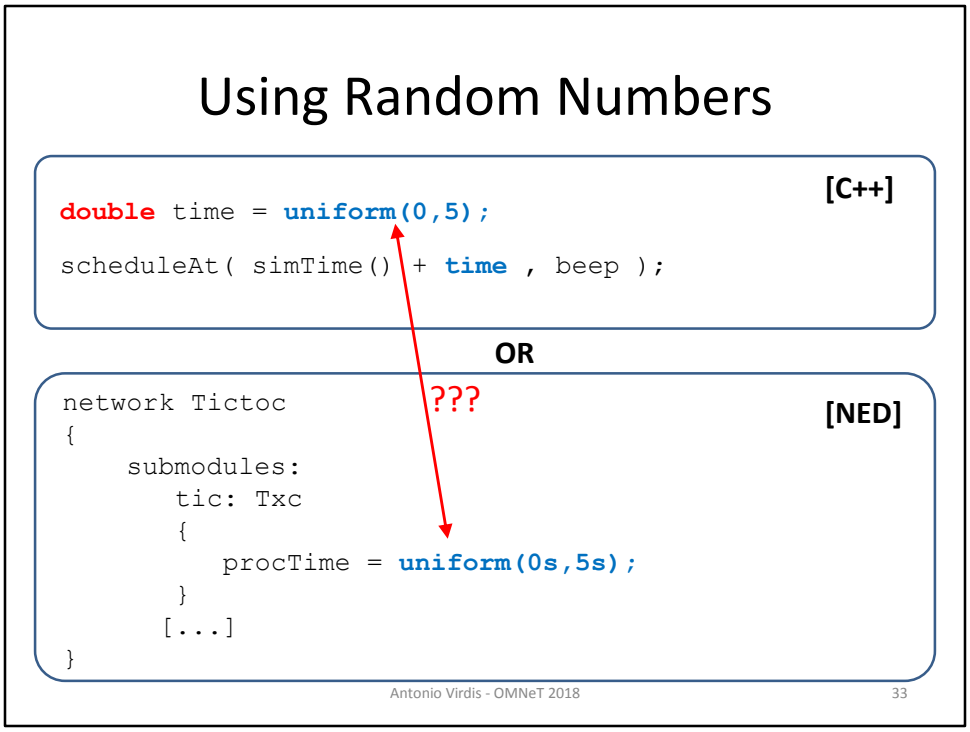

The value of a parameter can be read with the member function *par(name)*.

Random Numbers

Random Numbers

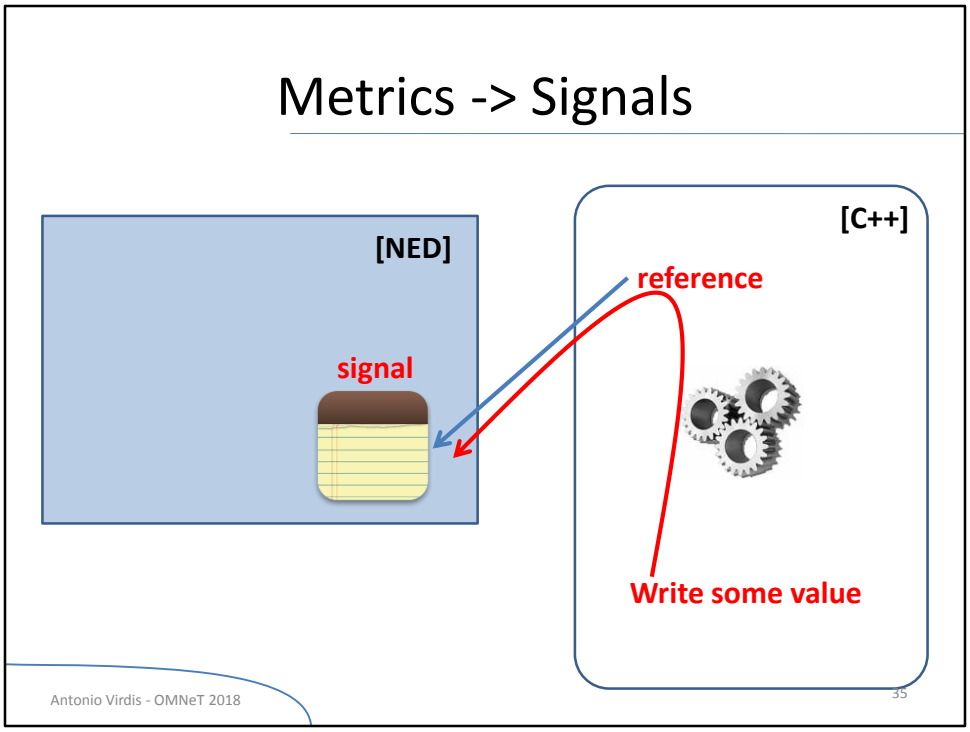
- Mersenne Twister
- Various distributions:
  - `uniform() ;`
  - `exponential() ;`
  - `normal();`
  - ….
- Importance of the **SEED**

OMNeT++ gives us various random number generators.
The default one is the well known Mersenne Twister.



We can generate random numbers from both C++ and NED code.
NOTE: the value contained in the variable procTime is set only once. Thus, consecutives calls of the function par() on said variable will always give the same value.

Metrics



The output of the system can be observed with *signals*. Each signal is a container for the values assumed by a certain metric during the simulation.
Signals can be of various kind, depending on the way the values are stored.
In this lesson we will consider only *scalar* signals, i.e. signals that returns only one value at the end of the simulation.
Signals are defined in NED files as part of modules. They can be referred to in C++ code in order to use them for storing values.

Signal usage

- Define a ***signal + statistic*** **[NED]**

- define variable with type simsignal_t **[C++]**
- Register signal via registerSignal()
- Use the signal via emit(name,value)

Signal usage

[NED]

```
parameters:
  @signal[delay](type=long);
  @statistic[delayStat](source="delay";record=mean;);
```

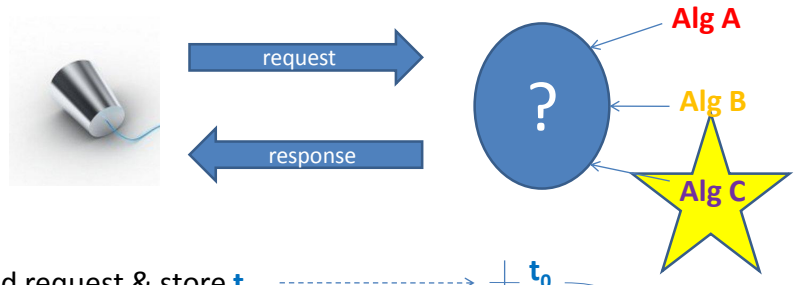
[C++]

```
private:
  simsignal_t delaySignal;
```

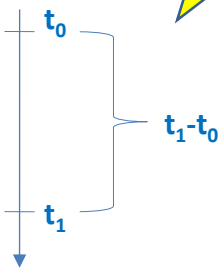
```
//initialize
  registerSignal("delay");
```

```
//somewhere
  emit(delaySignal, value);
```

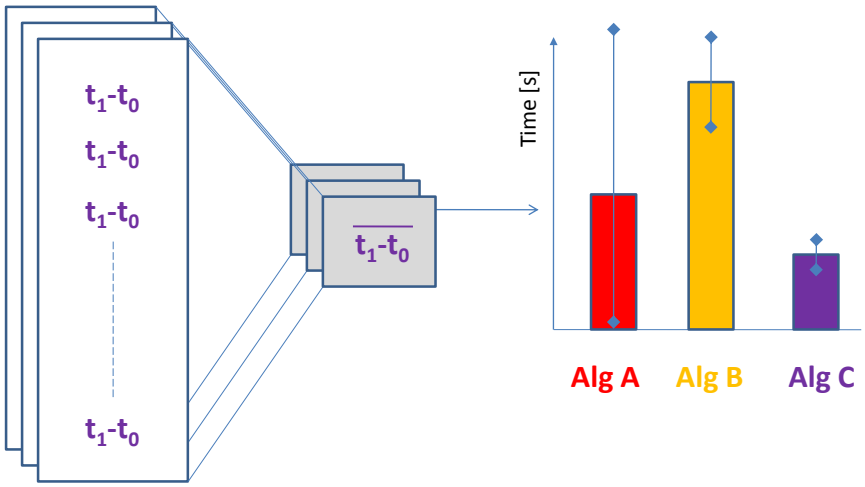
Example: response time



- Send request & store t_0
- […]
- Recv response & store t_1
- Emit value t_1-t_0



Example: response time



Managing parameters

Configuration Files

- Managed via **.ini** files

```
[General]
network = Tictoc
sim-time-limit = 100s
…
[Config Test1]
description = "first TicToc campaign"
Tictoc.t*c.procTime = 50ms
```


All the parameters we defined for the modules can be changed via .ini configuration files. We can specify simulation-specific parameters in the same way.

.ini files are organized as follows:

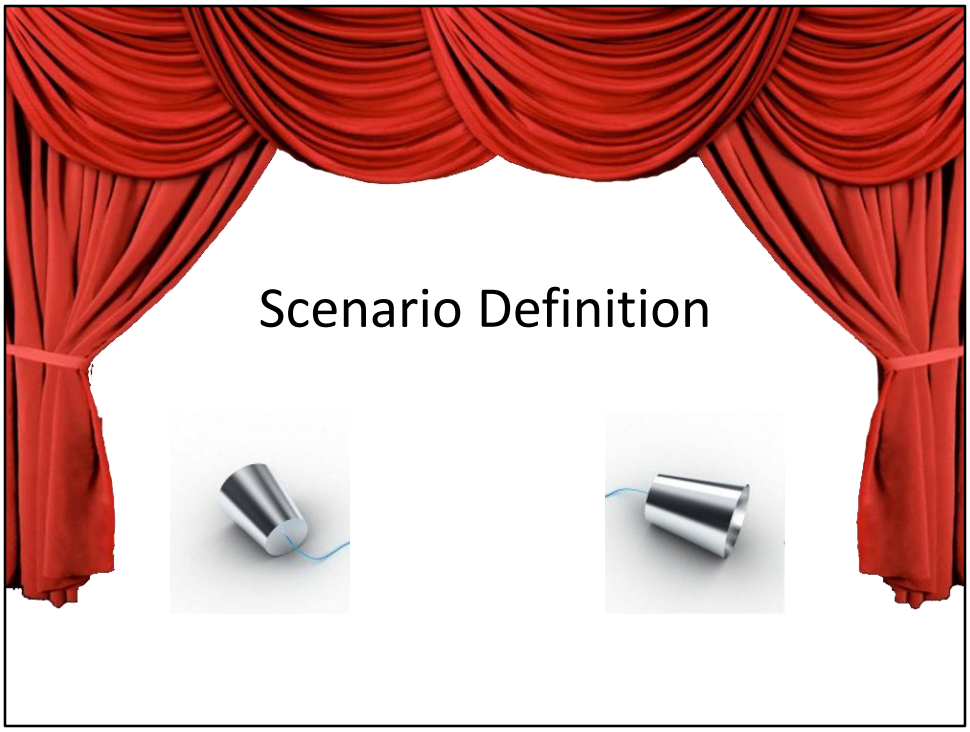
1) A *General* configuration is first defined. The value contained here will be common to all the configurations in the file.
2) One or more configuration follows, each one inheriting the parameters of the general one, and possibly overwriting them

Inheriting Configurations

```
[Config Test1]
description = "first TicToc campaign"
Tictoc.t*c.procTime = 50ms

[Config specificTest]
extends Test1
sim-time-limit = 5s
```


Inheritance of configurations can be achieved via the *extends* keyword. This way we can organize a configuration file hierarchically.



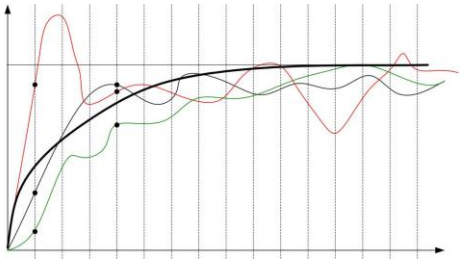
The structure and the behavior are separated from the definition of the scenario
This allows us to change system a lot without re-compiling the whole project

Scenario Definition

- Can be done via .ini file
- Two Main Parts:
  - Parameters
  - Factors

Once our module is ready and implemented, we have to define our simulation scenario by specifying both simulation parameters and model factors.
This can be done with .ini files.

Simulation Parameters

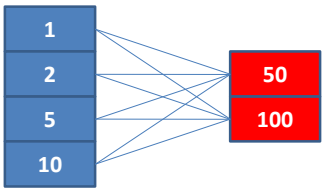


- Duration: `sim-time-limit`
- Warm Up: `warmup-period`
- Repetitions: `repeat`
- Define Seed: `seed-set = ${repetition}`

Factors range

- Which parameters?
- Which values?

```
**.t*c.delayTime    =  ${1, 2, 5, 10}
**.t*c.packetSize   =  ${50 , 100}
```



First of all we have to decide which characteristics of our system will be explored during the simulation, thus we have to decide the *factors.*
Then we have to decide the values that will be assumed by the factors.
We have two main solutions:

1) create an .ini configuration for each value assumed by a factor. What happens if we have more than one factor?
2) define a set of values for each factor

In the second case the simulation environment automatically computes the Cartesian product between all the sets of values for each parameter.

Omnetpp

- Download
  - www.omnetpp.org/omnetpp/cat_view/17-downloads/1-omnet-releases
- Install Guide
  - http://omnetpp.org/doc/omnetpp/InstallGuide.pdf
- Tutorial
  - http://www.omnetpp.org/doc/omnetpp/tictoc-tutorial/