Pedro Vale Estrela FAQ / Assorted NS2 Tips

This page contains several assorted NS2 tips, detailing some things that I've learned in "the hard way" in NS2.

Others are frequently asked questions (FAQ) that Ive replied multiple times to NS2 users.

Files and Patches (contains the patches mentioned in these pages)

Contact: pedro.estrela@inesc.pt

Index

How to modify existing procedures from the standard TCL library (/tcl/lib/) without having to recompile NS2 at each step.

If you are modifying the standard procedures that are present in /tcl/lib/ns-lib.tcl, or in any of its sourced files, then you are supposed to recompile NS2 to re-generate the standard NS2 tcl library and make your changes permanent.

However, if you are testing with a single script, its much easier to notice that TCL enables to replace all procedures in run-time; thus, to avoid modifying the ns2 core files, you should clone the procedure that you are modifying to the top of your script, and its sufficient to use your own modified, private copy of it. When

its perfect, only then modify the original core file in /tcl/lib to make your changes permanent.

More info: http://inesc-0.tagus.ist.utl.pt/~pmsrve/ns2/ns2_tips.html#_Toc121320478

Also check and step 4 of http://inesc-0.tagus.ist.utl.pt/~pmsrve/ns2/ns2_debugging3.html

How to know values at run-time which are "easy" to store at configuration time

One common FAQ is to know specific configuration details at run-time in C++ modules like classifiers and agents of the current node handle

and addresses. Many variations of this FAQ exist, like this example: http://mailman.isi.edu/pipermail/ns-users/2006-January/053450.html

For this, at configuration time, normally is fairly easy to store a reference to the corresponding node immediately after the components creation.

At the time of creation, add to it an instance variable called mynode, eg

```
$agent set mynode_handle $node
$agent set mynode_id [$node id]
```

then, at run time just check the variable:

```
$agent set mynode_handle
$agent set mynode_id
```

In C++, its simply a case of correctly using "tcl binding", where otcl instance variables have always the

same values of C++ variables, eg:

```
contructor()
{
```

```
   ...

   bind("mynode_id", mynode_id);       // but don't forget to add "Class set mynode_id 0"
to /tcl/lib/ns-default.tcl !!!

   ...

}
```

or just calling the same tcl code from C++, which is useful for more situations (see below)

```
tcl.eval("%s set mynode_id", name()); // equivelent to "$self set mynode_id"

assert(*tcl.result());

int mynode_id = atoi(tcl.result());
```

Another way, not discussed here, but recommended for more serious programming in NS2, is to modify C++ code to add references to fairly used variables directly in the C++ objects. This is essential to enable performance enouncements to components that deal with all the data packets, like classifiers, queues, etc.

How to trigger procedures in other modules at run-time.

Picture this: you have an module, say a classifier (C), which resides on each node (N) and listens to all packets; on certain run-time conditions, you want to trigger something at another component residing on that the node (N), namely an agent (A). However, in the place (C) where you are aware of that special condition (eg, where the triggering "if" is) you don't hold any variable to either (N) or (A) that could be used to trigger what you want!

The first part of this FAQ is done by simply storing a reference on C to both N and A at configuration time (see above)

The second part of this FAQ is done by calling otcl methods from C++ code (see below)

Then, simply call on C the appropriate otcl method of A using this mechanism:

```
tcl.eval("%s set myagent_handle", name());

assert(*tcl.result());

int myagent_handle = tcl.result();
```

```
tcl.eval("%s call_special_instproc %d, myagent_handle, some_int_parameter);
```

Again, this is the easiest way to do such things, and is only recommended for immediate results; for more serious programming in NS2, one should add appropriate pointers to objects in C++ and respect the Object-Oriented rules.

How to call oTCL code from C++ counterpart - an illustrated example

Original Ideia: http://www.cse.msu.edu/~wangbo1/nshowto5.html

```
The following code is an example on how to call otcl code in a C++ object:


//////////////////////////

AODV* aodvagent;

// set i to be the node id


char command[256];

sprintf(command, "foreach aodvagent [Agent/AODV info instances]\ {\nif

{ [$aodvagent id] == %d} {\nset i $aodvagent}}\nset t $i\n", i);


Tcl& tcl = Tcl::instance();

tcl.eval(command);

const char* ref = tcl.result();


aodvagent = (AODV*)tcl.lookup(ref);


//////////////////////////
```

For tutorial purposes, let me explain what this code does, as it is a common example of C++ / oTCL

integration:

- suppose i == 99. It contains the ID of a node that you are interested

- in C++, a script is prepared to run, where the c++ variable id is inserted in the "%d". Thus, the script results as this (comments are mine):

```
// cycle through all AODV nodes
foreach aodvagent [Agent/AODV info instances]

        // if the current node has the id we want, save it
        if { [$aodvagent id] ==  99 } {
                set i $aodvagent}
        }

        // dummy set, to make the function return the value $i.
        // could also been just "return $i"
        set t $i
}
```

- in C++, the tcl.eval command executes the script

- the result of it saved in tcl.result() (eg, the otcl object _oXX that we're looking for)

- converts the _oXX otcl object to c++ object with tcl.lookup()

some additional examples:

a) This calls a otcl instproc of the corresponding otcl object of the AODV weve found in the previous step.

```
      sprintf(command, "%s <some AODV Agent instproc> %d %s,

         aodvagent->name(), some_integer_parameter, a_string);

      tcl.eval(command);

   ... use tcl.result() ...
```

```
b) This calls a otcl instproc of the corresponding otcl object of THIS c++ object.
```

```
      TCL_EVALFr(%s <some instproc of the current otcl object that must return something>
      %d %s,

         this->name(), some_integer_parameter, a_string);
```

```
- notice as that C++ this->name() returns the same value as $self in otcl, EG, the handle
of the current object).

- Also notice my printf-style version of tcl.evalf, named TCL_EVALFr, which accepts
variable number of arguments and asserts that the function returns something. (check
```
utils_ns.cc for that and other C++ helper functions)

How to known which Agents are implemented in NS2 / which protocols are implemented in NS2

A very common question is to know what protocols are implemented in NS2. For example, NS2 is well respected by having very good implementations of the different flavors/options of TCP. However, to know exactly what agents implement said protocols, one has two options:
a) look into the NS2 manual AFAIK, it does not have the description of every single thing implemented;

b) search directly into the source code, using grep d recurse Agent and related methods.

Alternative 1

Using the otcl MASH object inspector (installation described here), this task is made trivial; one just has to check the otcl classes under Agent.

For this, simply create a MashInspector object, and check the class named Class. On the instances column youll check all the NS2 classes, which in my version of NS are 595 of them.

Thus, for the TCP variants are present in Agent/TCP/*. (screenshot)

Alternative 2

An alternative would be to simply use the info introspection functions of NS2.

Eg, Agent info subclass

However, note that this will only give you the first level of subclasses under Agent/*. Thus, Agent/TCP /NewReno and its other friends will NOT appear.

Alternative 3

For addressing this, one can use the subclass method that I propose in ns2_shared_procs.tcl, and is explained below.

Eg. Agent subclass

This results in this complete list of Agent subclasses.

Agent/SRM Agent/rtProto/Session Agent/HttpInval Agent/SatRoute Agent/TCPSink Agent/rtProto/LS Agent/LMS/Receiver Agent/MIPBS Agent/rtProto/Algorithmic Agent/Mcast Agent/OmniMcast Agent/SRM/Adaptive Agent/VatRcvr Agent/TCP/Vegas Agent/DSRAgent Agent/TCP/Newreno/FS Agent/AckReconsClass Agent/TCP/Reno Agent/TCP/FullTcp/Newreno Agent/ConsRcvr AckReconsClass Agent Agent/TCPSink/QS Agent/RAP Agent/TCP/BayFullTcp/Sack Agent/AbsTCP/RenoAck Agent/SRM/SSM Agent/TIMIPMH Agent/Diffusion /ProbGradient Agent/rtProto/DV Agent/Message Agent/TCP/FullTcp/Tahoe Agent/PGM/Sender Agent/IMEP Agent/RTP Agent/AbsTCPSink Agent/SRAgent Agent/BayTcpApp/FtpServer Agent/NOAH Agent/CtrMcast/Encap Agent/RTCP

Agent/CBR/RTP Agent/LossMonitor/PLM Agent/LMS/Sender Agent/TCP/FullTcp/Sack Agent/AbsTCP/RenoDelAck Agent/CBR Agent/TORA Agent/Ping Agent/LMS Agent/CIPMH PLMLossTrace Agent/AbsTCP/TahoeDelAck Agent/DumbAgent Agent/CBR/UDP Agent/Diffusion Agent/TCP/Newreno/Asym/FS Agent/Encapsulator Agent/TCP/SimpleTcp Agent/MFTP/Rcv/Stat Agent/QSAgent Agent/TCP/SackRH Agent/TCP/Session Agent/TCPSink/DelAck Agent/GAF Agent/DSRProto Agent/rtProto/Static Agent/Timip Agent/TCP/FullTcp Agent/PGM /Receiver Agent/CtrMcast Agent/Watchdog_Timip Agent/SAack Agent/TCP/Vegas/RBP Agent/IVS Agent/Flooding Agent/IVS/Source Agent/TCP/Reno/RBP Agent/TIMIPBS Agent/Hawaii Agent/TCPSink/Asym Agent/Diffusion/RateGradient Agent/MFTP/Snd Agent/TCP /RFC793edu Agent/CBR/UDP/SA Agent/TCP/Newreno/Asym Agent/CIPBS Agent/Mcast/Control Agent/TCP/BayFullTcp/Tahoe Agent/LDP Agent/timip_v0 Agent/TCPSink/Sack1/DelAck Agent/TCPSink/Sack1 Agent/Pushback Agent/TFRCSink Agent/TIMIP_V0MH Agent/TCP/FS Agent/DiffusionApp Agent/Watchdog Agent/Watchdog_timip_v0 Agent/TIMIP_V0BS Agent/TCP/BayFullTcp/Newreno Agent/TCP/Asym Agent/AODV Agent/TCP/Newreno Agent/TCP/Newreno/QS Agent/MFTP Agent/BayTcpApp Agent/TCP/Sack1 Agent/CtrMcast/Decap Agent/TCP/Reno/Asym Agent/LossMonitor Agent/DSDV Agent/rtProto/Direct Agent/IVS/Receiver Agent/SA Agent/MFTP/Rcv Agent/TCP/Fack Agent/Decapsulator Agent/SRM/Probabilistic Agent/MIPMH Agent/TCP/Int Agent/rtProto/Dummy Agent/SRM/Deterministic Agent/PGM Agent/AbsTCPSink/DelAck Agent/TCP/BayFullTcp

Agent/UDP Agent/SRM/Fixed Agent/rtProto/Manual Agent/TCP/Reno/FS Agent/DiffusionRouting Agent/TCP Agent/TFRC Agent/Diff_Sink Agent/Null LossTrace Agent/BayTcpApp/FtpClient Agent/AbsTCP Agent/AbsTCP/TahoeAck Agent/rtProto

What info options are available in otcl classes and objects?

The otcl info procs are very powerful introspection methods that mirror the corresponding info commands in regular TCL.

The table below shows the most useful: (sources: object.html / class.html).

CLASSES:

Sample Invocation	Description
<class> info superclass	returns the superclass list of the object
<class> info heritage	returns the inheritance precedence list (eg, the complete heritage chain, starting from the parent Class)
<class> info subclass	returns the subclass list of the object, of the DIRECT heritage subclasses only.
<class> info instances	returns a list of the instance objects of the class, of the DIRECT heritage subclasses only.
<class> info instprocs	Returns a list of the names of instproc methods defined on the class
<class> info instcommands	returns a list of the names of both Tcl and C instproc methods defined on the class

OBJECTS:

Sample Invocation	Description
_oXX info class	returns the class of the object
_oXX info procs	returns a list of the names of proc methods defined on the object
_oXX info commands	returns a list of the names of both Tcl and C proc methods defined on the object
_oXX info vars	returns a list of the names of instance variables defined on the object

These tables shows other info commands useful only for reverse engineering the procs: (sources: object.html / class.html). If you want to understand these facilities, study the Object proc retrieve code in object.html.

CLASSES:

Sample Invocation	Description
<class> info instargs	query the argument list of a Tcl instproc method.
<class> info instbody	query the body of a Tcl instproc method.
<class> info instdefault	query the default value of an argument of a Tcl instproc method

OBJECTS:

Sample Invocation	Description
_oXX info args <proc>	query the argument list of a Tcl proc method
_oXX info body <proc>	query the body of a Tcl proc method
_oXX info default <proc> <argment> <variable>	query the default value of an argument of a Tcl proc method

Finally, note that the most useful class introspection commands are limited that it only return the names of the DIRECT subclasses.

For example, this sequence shows this problem:

```
49 % new Simulator                                  ;# create a simulator object

_o4

50 % new Agent/TCP/FullTcp                          ;# create a FULLTCP object

_o11

51 % new Agent/TCP/FullTcp                          ;# create another FULLTCP object

_o12

51 % Agent info instances                   ;# lets search for them

<eg, returns nothing>

52 % Agent/TCP info instances                        ;# lets search for them one lever above

<eg, also doesnt return nothing>

54 % Agent/TCP/FullTcp info instances ;# They only appear on the exact class

_o12 _o11                                           ;# here are them

55 % Agent  instances                                ;# This proc returns all direct and indirect instances
of a class

_o12 _o11 _o7                               ;# here are them
```

To address this, Ive created these additional procs, present in my ns2_shared_procs.tcl, which are subclass-aware.

Sample Invocation	Description
<class> instances	Returns the direct and indirect instances of a class
<class> subclass	Returns all direct and indirect subclasses of a class

How to check the syntax of .tcl files

When modifying TCL files, generally, the source code files fall under 2 types:
files which only have procs, without actually executing anything (eg, almost all of the files in the ns/tcl

hierarchy of directories);
and files with control statements, namely the scripts that are used to perform simulations.

It is normally (very) difficult to get a 100% syntax-correct .tcl file. For instance, a brace inside a comment IS CONSIDERED for brace matching, and thus is very difficult to make then match evenly. (as this behavior is quite different for all other languages that I know).

However, if the files are of the first type (eg, defining only procs), then you can test the syntax of it in a very simple manner:

For this, simply start ns, and manually source the file:
> source your_file.tcl

If the file has correct syntax, it will load ok. In this case, you can then load it subsequently again, as it will redefine the existing procs with the new versions.

Example: in \tcl\lib, do "source ns-lib.tcl". this file will load the complete ns2 tcl library recursively, as it contains multiple source commands within.

UPDATE: also check these fine links for TCL syntax info:

Why can I not place unmatched braces in Tcl comments?

http://phaseit.net/claird/comp.lang.tcl/fmm.html
http://mini.net/tcl/1669.html

How to get a trace of the oTcl procedures:

One of the most important things to debug deeply nested otcl instprocs is to have an idea of what instprocs, and with arguments, on what objects are being executed.

For this, one should use the "dputs" and "dputsl" functions that I propose here. These are supercharged puts-like functions, with special trace information for otcl purposes:
These functions give information on:
- The current simulator time
- The current object (eg, _oXXX)
- The Class and instproc information (e.g., to know where to look in the source code.)
- The current function's argument list (in name/value pairs) (for "dputsl" only)
- Any user defined string

Thus, by inserting appropriate dputsl " " commands in interesting instprocs, one gets a great otcl trace, which is very useful for debug run-time otcl errors.

Which linux distro NOT to use:

Quite simply, one should use a linux distribution in which the "install" script runs without any error.

This is NOT the case in certain linux distributions - especially "Mandrake" - that feature the latest and most recent versions of all core oprating system components. For the NS2 case, the most relevant is the Gcc compiler; however, you should also aim for 100% stable versions of the kernel itself, the drivers, the X windows etc etc etc. In 99.9% of the cases, you'll simply never actually use the latest features of every piece of software, and the stability that you'll gain will be WELL worth of it.

Thus, having a clean NS2 installation with the "install" script will result in much more time in actually using and modifying NS, instead of trying to install it!

I suggest you to use a debian distro. It has a very interesting policy of stable software, where the software is automatically updated ONLY when it has passed the "test of time", and the new versions are know to be stable. The distro has a simple image download to make the initial installation, and then everything else comes from the internet afterwards automatically.

PS: on a related note, this idea also works backwards; If you are trying to compile old code for old versions of NS (NS-1b6 and friends), You'll get a much better chance if you use earlier versions of GCC, like gcc-2.95.

Check the files, patches, etc in this directory

Go back to my NS2 page

Contact: pedro.estrela@inesc.pt
