Tcl and OTcl for ns

Outline:

Introduction

ns is an IP-based network simulator. ns-2 has been designed such that a user can specify a simulation using a simple interpreted scripting language. The scripting language causes the simulation objects to be created and the appropriate links made between them. Once the scenario is defined using the scripting language, the simulation can be run without need for recourse to the slow scripting language, and the simulation can be run using C++.

Like many people I used Marc Greis' tutorial to learn the basics of ns. However, this does not describe Tcl/OTcl, nor is it the purpose of that tutorial. In my case, since I had no experience with Tcl, I didn't know anything about Tcl syntax and Tcl flow control. Consequently, my earlier simulation scripts were quite long and inefficient, requiring time to be programmed, and more bug prone. Knowledge of Tcl would have enabled me to program these simulations better. I suspect that I'm not the only one to find himself in that predicament - ns is a simulation tool that happens to be implemented using Tcl - I don't think that Tcl use would be very high amongst the research/networking simulation community and hence I would suspect that many new ns users are not familiar with it.

Here, I try to present some useful aspects of Tcl/OTcl that are not discussed in the Greis tutorial, but are certainly useful to those who wish to do anything useful in ns. This information is probably available elsewhere on the net. The scriptics site has a multitude of resources on Tcl, but I thought that it would be useful to provide an ns specific description of Tcl/OTcl and give some simple examples.

I guess that this would be useful to ns beginners who have no experience with Tcl/OTcl and have read and

understand the Greis tutorial.

The relationship between Tcl/OTcl and ns

Tcl is a scripting language that allows simple access to a set of library functions. One alternative to access these library functions would be write C programs (or any other language for that matter) to use them. This takes time and requires knowledge of C, compilers and linker. Tcl provides a much simpler interface to the functionality and since the syntax of Tcl is very simple, it is easy for someone to learn how to use Tcl to use the functionality available in the library.

When you compile Tcl on it's own, you end up with a shell called `tclsh`. This can be invoked interactively, and you can issue Tcl commands directly to the Tcl shell. This is moderately interesting, but the shell is mostly invoked in a non-interactive fashion to execute a set of Tcl commands.

Various extensions to Tcl exist - many of these simply extend the shell by increasing the amount of library functions that are available.

OTcl is an extension to the basic Tcl shell that provides object-oriented (OO) functionality. This enables OTcl classes to be defined and supports methods for classes and data members of classes. The relationship between Tcl and OTcl is similar to that between C and C++: OTcl is a superset of Tcl. It now appears that OTcl is being supported by the ns development team.

There is another component of the ns system called TclCl that is used to provide linkage between classes at the C++ level and their corresponding OTcl classes. I think that TclCl is something that is also specific to ns.

ns can be viewed as a particular Tcl shell: one that has OTcl additions as well as a library of networking components (links, queues etc.) that can be generated and manipulated. The use of OTcl enables networking components to be defined as OTcl classes. Since ns can be viewed as a Tcl shell, the basic syntax of the Tcl language is entirely applicable in ns scripts; this is something that took me a little while to comprehend. The same applies with OTcl - ns just uses OTcl to define the networking components, and hence an ns script must follow the syntax specified by OTcl.

ns is typically invoked with a script defining the simulation, but it is also possible to use ns in an interactive fashion in the same way that it is possible to use the Tcl shell interactively.

Basic Tcl constructs

Variables and arrays

Defining a variable in Tcl is very simple:

```
set var1 1
set var2 "Tcl Variable 2"
```

The variables can be referenced by prefixing the variable name with a $. For example to print the above variables, we can use

```
puts "var1=$var1, var2=$var2"
```

Any situation in which you require that the value of the variable be used is one in which the $ prefix should be added to the variable name. In some situations, it is necessary to use the variable name directly. For example

```
incr var1
```

can be used to increment var1. I guess you can think of it as the difference between call-by-reference and call-by-value: in the former case you use the variable name on its own, while in the latter you prefix it with a '$'.

An alternative is to assign the results of a function to a variable. This can be done as follows:

```
set var3 [expr 5*10]
```

This sets the variable var3 to the result of calling the `expr` function with the parameter `5*10`. The `expr` function attempts to evaluate the supplied parameter to derive a value. Tcl interprets the square brackets as delimiters for a nested command: it attempts to execute the command inside the square brackets and assigns the result to var3 in this case. The returned value will be 50. Hence, the value 50 will be assigned to var3.

In Tcl all variables are represented internally as strings. Whether that string can be viewed as an integer or a floating point number only matters when you use a function that requires numeric arguments.

Tcl also supports arrays. These are very useful in ns for storing, say, nodes. Tcl supports arrays that can be indexed by simple numeric arguments, as is standard in most languages, but Tcl also supports arrays that can be indexed by arbitrary strings. It is not necessary to declare the size of the array in advance. Here, two example of arrays are given

```
set n(0) [$ns node]
set n(1) [$ns node]

set opts(bottlenecklinkrate) 1Mb
set opts(ECN) "on"
```

In the first example the array is called n and the index is numeric. In the second, the array is called opts and the index is non-numeric.

For loops

For loops are very useful in ns and can be used in conjunction with arrays to easily create larger network topologies. To generate 100 nodes, the following code can be used:

```
for {set i 0}{$i < 100}{incr i} {
    set n($i) [$ns node]
}
```

While loops

These are very similar to for loops. The syntax is

```
set i 0
while {$i < 10} {
    set n($i) [new Node]
    incr i
}
```

If statements

If statements are very simple

```
if {$i < 10} {
    puts "i is less than 10"
}
if {$var2 == "Tcl Variable 2"} {
    puts "var2 = Tcl Variable 2"
}
```

Procedures

Procedures are an essential component of Tcl and can be used to make programming ns simpler. As in any functional programming language, procedures can be used for repetitive tasks, or simply to logically break down the tasks in the program.

Procedures are defined in Tcl as follows:

```
proc proc1 {} {
    puts "in procedure proc1"
}
```

This defines a procedure that takes no parameters and prints out "in procedure proc1". To call this procedure

```
proc1
```

can be used.

A procedure with parameters can be defined as follows:

```
proc proc2 {parameter1} {
    puts "the value of parameter1 is $parameter1"
}
```

This procedure can be invoked as follows:

```
proc2 10
```

A procedure that returns a value can be defined as follows:

```
proc proc3 {min max} {
    set randomvar [rand $min $max]
```

```
    return $randomvar
}
```

This procedure generates a random variable and returns it to the calling function. This can be invoked as follows

```
set randomvar [proc3 0 1]
```

to obtain a uniform random value between 0 and 1.

Sometimes it is necessary within a procedure to reference a variable that has global scope. This is the purpose of the `global` keyword. So, for example, in an ns script, the simulator object typically is called ns, and typically has global scope. So, it could be referenced in a procedure as follows:

```
proc proc4 {} {
    global ns
    $ns at 10.0 "exit 0"
}
```

A logical way to break down an ns script can be as follows:

```
set ns [new Simulator]
create_topology
create_agents
create_sources
create_recorders
$ns run
```

where `create_topology`, `create_agents`, `create_sources` and `create_recorders` are all procedures.

OTcl constructs

Here, I try to explain the OO concepts of Tcl. I am reasonably experienced in C++ programming, and I found that I instinctively tried to relate OTcl concepts to C++ concepts. This does not work some of the time, but when it doesn't I think that I highlight this fact.

OTcl classes and objects

OTcl has keywords to register a class name with OTcl and to define a parent class for it. This can be done as follows:

```
class MyNode -superclass Node
```

This defines a class called `MyNode` that is a child class of the class called `Node`. As in C++ this means that any member functions of `Node` will be available to `MyNode`. However, as is explained below, the set of data members of a class is not declared in advance: class data members can be created on demand. The parent/child relationship then really applies to the set of member functions available rather than the set of data members.

To create a MyNode object the new function can be called as follows:

```
set mynodevar [new MyNode]
```

I think that objects can be deleted aswell, but this is something that is rarely necessary in ns simulation scripts.

The `self` variable can be used within the classes member functions to refer to the particular object. It can be considered to be analogous to the `this` pointer in C++. In particular, the `self` variable must be used to call other member functions from within a member function.

Class member functions

A member function for a particular class can be declared as follows:

```
MyNode instproc memberfn1 {} {
    puts "in member function 1"
}
```

This can be called using the following code:

```
set myobj [new MyNode]
$myobj memberfn1
```

Class data

The data members of a specific class are only defined within the member functions of the class. This is unlike C++ in which the data members of the class are specfied in advance, and typically in header files. Class data members are created or references using the `instvar` directive. The following code illustrates the use of OTcl class data members.

```
MyNode instproc memberfn2 {} {
    $self instvar datamember1
    set datamember1 "data member - memberfn2"
}

MyNode instproc memberfn3 {} {
    $self instvar datamember1
    puts "datamember1 = $datamember1"
}

set myobj [new MyNode]
$myobj memberfn2
$myobj memberfn3
```

A new `MyNode` object is created and assigned to `myobj`. The `memberfn2` member function is called. This assigns a value to the data member `datamember1`. Then `memberfn3` is called and the value of `datamember1` is printed. The value that is assigned in `memberfn2` will be printed.

Object initialisation

C++ has special constructor functions that are called when an object is created. These can be used to initialise the data members for the object. OTcl has the same functionality. Like C++, a special member function is used. In OTcl the member function is called init (in C++ the member function has the same name as the class). The initialisation function for the OTcl class MyNode could be defined as follows:

```
MyNode instproc init {} {
    $self next
    $self instvar datamember1
    set datamember1 0
}
```

The function is quite clear apart from the `$self next` command. This has the purpose of calling the initialisation function for the parent class.

The constructor function can also take arguments. The parameters supplied to the init function are specified in the call to the new function as follows:

```
MyNode2 instproc init {arg1 arg2} {
    $self next
    $self instvar dm1 dm2
    set dm1 $arg1
    set dm2 $arg2
}

set mn2 [new MyNode2 1 2]
```

OTcl and C++ linkage

This is getting slightly under the bonnet - is it relevant here?

When OTcl objects are created in ns, a C++ object is also usually instantiated. The C++ is the one that is used in the simulation, but OTcl must provide an interface by which parameters of the C++ object can be assigned values. There must be *linkage* between the OTcl and C++ objects.

Miscellaneous extras

I had some idea for this section, but it escapes me now.

Conclusion

I have attempted to describe the relationship between Tcl, OTcl and ns. I have explained some Tcl and OTcl syntax and basic keywords that can be used to implement simulations more efficiently.