Chapter 3
Linkage Between OTcl and C++ in NS2

NS2 is an object-oriented simulator written in OTcl and C++ languages.[1] While OTcl acts as the frontend (i.e., user interface), C++ acts as the backend running the actual simulation (Fig. 2.1). From Fig. 3.1 class hierarchies of both languages can be either standalone or linked together using an OTcl/C++ interface called TclCL [17]. The OTcl and C++ classes which are linked together are referred to as *the interpreted hierarchy* and *the compiled hierarchy*, respectively.

Object construction in NS2 proceeds as follows. A programmer creates an object from an OTcl class in the interpreted hierarchy. Then, NS2 (or more precisely TclCL) automatically creates a so-called shadow object from a C++ class in the compiled hierarchy. It is important to note that no shadow object would be created when a programmer creates an object from a class in both compiled and standalone OTcl hierarchies.

Written in C++, TclCL consists of the following six main classes. First, class `TclClass` maps class names in the compiled hierarchy to class names in the interpreted hierarchy. Second, class `InstVar` binds member variables in both the hierarchies together. Third, class `TclCommand` allows the Tcl interpreter to execute non-OOP C++ statements. Fourth, class `TclObject` is the base class for all C++ simulation objects in the compiled hierarchy. Fifth, class `Tcl` provides methods to access the interpreted hierarchy from the compiled hierarchy. Finally, class `EmbeddedTcl` translates OTcl scripts into C++ codes. The details of the above classes are located in files *~tclcl/*tclcl.h, *~tclcl/*Tcl.cc, and *~tclcl/*tclAppInit.cc.

This chapter focuses on using TclCL in the following meaningful ways:

- Section 3.1 presents the motivation of having two languages in NS2.
- Section 3.2 explains class binding which maps C++ class names to OTcl class names.

[1] Refer to [16] for the C++ programming language.

T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*,
DOI 10.1007/978-1-4614-1406-3_3,

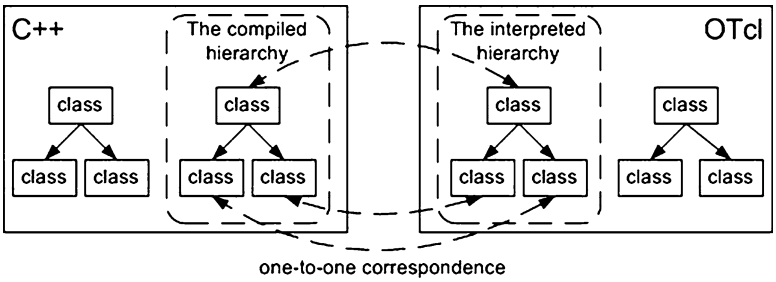

Fig. 3.1 Two language structure of NS2 [14]. Class hierarchies in both the languages may be standalone or linked together. OTcl and C++ class hierarchies which are linked together are called *the interpreted hierarchy* and *the compiled hierarchy*, respectively

- Section 3.3 discusses how NS2 binds a pair of member variables of two bound classes so that a change in one variable will be automatically reflected in the other.
- Section 3.4 shows a method to execute C++ statements from the OTcl domain.
- Section 3.5 walks through the shadow object construction process.
- Section 3.6 discusses various functionalities to access the Tcl interpreter from the C++ domain: Tcl statement execution, result passing between both the domains, and the TclObject reference retrieval.
- Section 3.7 briefly outlines how the OTcl codes are translated into the C++ code.

3.1 The Two-Language Concept in NS2

3.1.1 The Natures of OTcl and C++ Programming Languages

Why two languages? Loosely speaking, NS2 uses OTcl to create and configure a network (i.e., user frontend), and C++ to run simulation (i.e., internal mechanism). All C++ codes need to be compiled and linked to create an executable file. Since the body of NS2 is fairly large, the compilation time is not negligible. A typical Macbook Pro computer requires few seconds (long enough to annoy most programmers) to compile and link the codes with a small change such as including a C++ statement "`int i=0;`" into the program. OTcl, on the other hand, is an interpreted programming language, not a compiled one. Any change in a OTcl file can be executed without compilation. Since OTcl does not convert the codes into machine language, each line needs more execution time.[2]

[2]Although OTcl is an interpreted programming language, NS2 translates most of its OTcl codes into C++ using class `EmbeddedTcl` (see Sect. 3.7) to speed up the simulation. As a result, most change in OTcl also requires compilation.

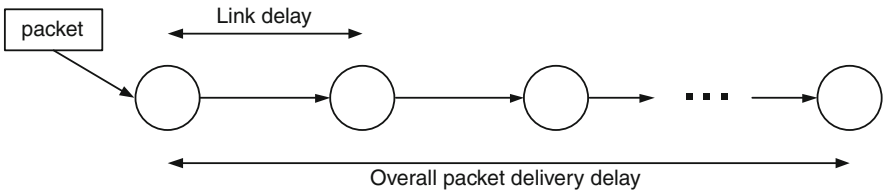

Fig. 3.2 A chain topology for network simulation

In summary, C++ is fast to run but slow to change. It is suitable for running a large simulation. OTcl, on the other hand, is slow to run but fast to change. It is suitable as a parameter configurator. NS2 is constructed by combining the advantages of these two languages.

3.1.2 C++ Programming Styles and Its Application in NS2

The motivation can be better understood by considering three following C++ programming styles.

3.1.2.1 Basic C++ Programming

This is the simplest form and involves basic C++ instructions only. This style has a flexibility problem, since any change in system parameters requires a compilation (which takes non-negligible time) of the entire program.

Example 3.1. Consider the network topology in Fig. 3.2. Define overall packet delivery delay as the time needed to carry a packet from the leftmost node to the rightmost node, where delay in link "`i`" is "`d_i`" and total number of nodes is "`num_nodes`." We would like to measure the overall packet delivery delay and show the result on the screen.

Suppose that every link has the same delay of 1 s (i.e., "`d_i = 1`" second for all "`i`"), and the number of nodes is 11 (`num_nodes = 11`). Program 3.1 shows a C++ program written in this style (the filename is "`sim.cc`"). Since the link delay is fixed, we simply increase "`delay`" for `num_nodes-1` times (Lines 4 and 5). After compiling and linking the file `sim.cc`, we obtain an executable file "`sim`." By executing "`./sim`" at the command prompt, we will see the following result on the screen:

```
>>./sim
Overall Packet Delay is 10.0 seconds.
```

Despite its simplicity, this programming style has a flexibility problem. Suppose link delay and the number of nodes are changed to 2 s and 5 nodes, respectively.

Program 3.1 A basic C++ program that simulates Example 3.1, where the delay for each of the links is 1 unit and the number of nodes is 11

```
    //sim.cc
1   main(){
2       float delay = 0, d_i = 1;
3       int i, num_nodes = 11;
4       for(i = 1; i < num_nodes; i++)
5           delay += d_i;
6       printf("Overall Packet Delay is %2.1f seconds.\n",
    delay);
7   }
```

Then, we need to modify, compile, and link the file `sim.cc` to create a new executable file "`sim`." After that, we can run "`./sim`" to generate another result (for `d_i` = 2 and `num_nodes` = 5). □

3.1.2.2 C++ Programming with Input Arguments

Addressing the flexibility problem, this programming style takes the system parameters (i.e., `argv, argc`) as input arguments [16]. As the system parameters change, we can simply change the input arguments, and do not need to recompile the entire program.

Example 3.2. Consider Example 3.1. We can avoid the above need for recompilation and relinking by feeding system parameters as input arguments of the program. Program 3.2 shows a program that feeds link delay and the number of nodes as the first and the second arguments, respectively. Line 1 specifies that the program takes input arguments. The variable `argc` is the number of input arguments. The variable `argv` is an argument vector that contains all input arguments provided by the caller (See the details on C++ programming with input arguments in [16]).

With this style, we only need to compile and link the program once. After obtaining an executable file "`sim`," we can change the simulation parameters as desired. For example,

```
>> ./sim 1 11
Overall Packet Delay is 10.0 seconds.
>> ./sim 2 5
Overall Packet Delay is 8.0 seconds.
```

Although this programming style solves the flexibility problem, it becomes increasingly inconvenient when the number of input arguments increases. For example, if delays in all the links in Example 3.1 are different, we will have to type in all the values of link delay every time we run the program. □

Program 3.2 A C++ program with input arguments: A C++ program which simulate Example 3.2. The first and second arguments are link delay and the number of nodes, respectively

```
    //sim.cc
1   int main(int argc, char* argv[]) {
2       float delay = 0, d_i = atof(argv[0]);
3       int i, num_nodes = atoi(argv[1]);
4       for(i = 1; i < num_nodes; i++)
5           delay += d_i;
6       printf("Overall Packet Delay is %2.1f seconds\n",
    delay);
7   }
```

3.1.2.3 C++ Programming with Configuration Files

This last programming style puts all system parameters in a configuration file, and the C++ program reads the system parameters from the configuration files. This style does not have the flexibility problem, and it facilitates program invocation. To change system parameters, we can simply change the content of the configuration file. In fact, this is the style from which NS2 develops.

Example 3.3. Program 3.3 applies the last C++ programming style (i.e., with configuration files). The program takes only one input argument: The configuration file name (See C++ file input/output in [16]). Function `readArgFromFile(fp,d)` reads the configuration file associated with a file pointer "`fp`," and sets variables "`num_node`" and "`d`" accordingly (the details are not shown here). In this case, the configuration file (`config.txt`) is shown in Lines 10 and 11. When invoking "`./sim config.txt`," the screen will show the following result:

```
>>./sim config.txt
Overall Packet Delay is 55.0 seconds.
```

To change the system parameters, we can simply modify the file "`config.txt`" and run "`./sim`." Clearly, this programming style removes the necessity for compiling the entire program and the lengthy invocation process. □

Recall from Sect. 2.5 that we write and feed a *Tcl simulation script* as an input argument to NS2 when running a simulation (e.g., executing "ns `myfirst_ns.tcl`"). Here, "`ns`" is a C++ executable file obtained from the compilation, while "`myfirst_ns.tcl`" is an input configuration file specifying system parameters and configuration such as nodes, link, and how they are connected. Analogous to reading a configuration file through C++, NS2 reads the system configuration from the Tcl simulation script. When we would like to change the parameters in the simulation, all we have to do is to modify the Tcl simulation script and rerun the simulation.

Program 3.3 C++ programming style with configuration files: A C++ program in file `sim.cc` (Lines 1–9) and a configuration file `config.txt` (Lines 10 and 11)

```
    //sim.cc
1   int main(int argc, char* argv[]) {
2       float delay = 0, d[10];
3       FILE* fp = fopen(argv[1],"w");
4       int i, num_nodes = readArgFromFile(fp,d);
5       for(i = 1; i < num_nodes; i++)
6           delay += d[i-1];
7       printf("Overall Packet Delay is %2.1f seconds\n",
    delay);
8       fclose(fp);
9   }

    //config.txt
10  Number of node = 11
11  Link delay = 1 2 3 4 5 6 7 8 9 10
```

3.2 Class Binding

Class binding maps C++ classes to OTcl classes. When a programmer creates an object from the interpreted hierarchy, NS2 determines the compiled class from which a shadow object should be instantiated by looking up the class binding map. As an example, an OTcl class `Agent/Tcp` is bound to the C++ class `TcpAgent`. A programmer can create an `Agent/TCP` object using the following OTcl statement

```
new Agent/TCP
```

In response, NS2 automatically creates a compiled shadow `TcpAgent` object.

3.2.1 Class Binding Process

A class binding process involves four following components:

- A C++ class (e.g., class `TcpAgent`)
- An OTcl class (e.g., class `Agent/TCP`)
- A mapping class (e.g., class `TcpClass`): A C++ class which maps a C++ class to an OTcl class
- A mapping variable (e.g., `class_tcp`): A static variable instantiated from the above mapping class to perform class binding functionalities.

Class binding is carried out by defining the mapping class as a part of the C++ file. Program 3.4 shows a mapping class `TcpClass` that binds the OTcl class `Agent/TCP` to the C++ class `TcpAgent`.

Program 3.4 Class `TcpClass` which binds the OTcl `Agent/TCP` to the C++ class `TcpAgent`

```
    //~ns/tcp/tcp.cc
1   static class TcpClass : public TclClass {
2   public:
3       TcpClass() : TclClass("Agent/TCP") {}
4       TclObject* create(int , const char*const*) {
5           return (new TcpAgent());
6       }
7   } class_tcp;
```

Every mapping class derives from the C++ class `TclClass` where all the binding functionalities are defined. Each mapping class contains only two functions: The constructor and function `create(...)`. In Line 3, the constructor feeds the OTcl class name `Agent/TCP` as an input argument to the constructor of its base class (i.e., `TclClass`).[3] In Lines 4–6, the function `create(...)` creates a shadow compiled object whose class is `TcpAgent`. This function is automatically executed when a programmer creates an `Agent/TCP` object from the interpreted hierarchy. We shall discuss the details of the shadow object construction process later in Sect. 3.5.

Note that a C++ class by itself cannot perform any operation. To bind classes, we need to instantiate a mapping object from the mapping class. In Line 7, such an object is the variable `class_tcp`. Since every class binding is unique, the mapping variable `class_tcp` is declared as `static` to avoid class binding duplication.

3.2.2 *Defining Your Own Class Binding*

The following steps bind an OTcl class to a C++ class:

1. Specify an OTcl class (e.g., `Agent/TCP`) and a C++ class (e.g., `TcpAgent`) which shall be bound together.
2. Derive a mapping class (e.g., `TcpClass`) from class `TclClass`.
3. Define the constructor of the mapping class (e.g., Line 3 in Program 3.4). Feed the OTcl class name (e.g., `Agent/TCP`) as an input argument to the constructor of the class `TclClass` (i.e., the base class).
4. Define function `create(...)` to construct a shadow compiled object. Invoke "`new`" to create a shadow compiled object and return the created object to the caller (e.g., `return (new TcpAgent())` in Line 5 of Program 3.4).
5. Declare a static mapping variable (e.g., `class_tcp`).

[3] A C++ operator "`:`" indicates what to be done before the execution of what is enclosed within the following curly braces [16].

Table 3.1 Examples of naming convention for mapping classes and variables

C++ class	OTcl class	Mapping class	Mapping variable
`TcpAgent`	`Agent/TCP`	`TcpClass`	`class_tcp`
`RenoTcpAgent`	`Agent/TCP/Reno`	`RenoTcpClass`	`class_reno`
`DropTail`	`Queue/DropTail`	`DropTailClass`	`class_drop_tail`

3.2.3 *Naming Convention for Class `TclClass`*

The convention to name mapping classes and mapping variables are as follows. First, every class derives directly from class `TclClass`, irrespective of its class hierarchy. For example, class `RenoTcpAgent` derives from class `TcpAgent`. However, their mapping classes `RenoTcpClass` and `TcpClass` derive from class `TclClass`.

Second, the naming convention is very similar to the C++ variable naming convention. In most cases, we simply name the mapping class by attaching the word "`Class`" to the C++ class name. Mapping variables are named with the prefix "`class_`" attached to the front. Table 3.1 shows few examples of the naming convention.

3.3 Variable Binding

Class binding, discussed in the previous section, creates connections between OTcl and C++ class names. By default, the bound classes have their own variables which are not related in any way. Variable binding is a tool that allows programmers to bind one variable in the C++ class to another variable in the bound OTcl class such that a change in one variable will be automatically reflected in the other.

3.3.1 *Variable Binding Methodology*

An OTcl variable, `iname`, can be bound to a C++ variable, `cname`, by including one of the following statements in the C++ class constructor:

- `bind("iname",&cname)` binds integer and real variables.[4]
- `bind_bw("iname",&cname)` binds a bandwidth variable.
- `bind_time("iname",&cname)` bind a time variable.
- `bind_bool("iname",&cname)` bind a boolean variable.

[4] In Sect. 3.3.3, we shall discuss the five following data types in NS2: Integer, real, bandwidth, time, and boolean.

Example 3.4. Let a C++ class `MyObject` be bound to an OTcl class `MyOTclObject`. Let `icount_`, `idelay_`, `ispeed_`, `idown_time_`, `iis_active_` be OTcl class variables whose types are integer, real, bandwidth, time, and boolean, respectively. The following C++ program binds the above variables:

```
class MyObject {
    public:
    int count_;
    double delay_,down_time_,speed_;
    bool is_active_;
    MyObject() {
        bind("icount_",&count_);
        bind("idelay_",&delay_);
        bind_bw("ispeed_",&speed_);
        bind_time("idown_time_",&down_time_);
        bind_bool("iis_active_",&is_active_);
    };
};
```

□

3.3.2 *Setting the Default Values*

NS2 sets the default values of bound OTcl class variables in the file *~ns*/tcl/lib/ns-default.tcl. The syntax for setting a default value is similar to the value assignment syntax. That is,

```
<className> set <instvar> <def_value>
```

which sets the default value of the instvar `<instvar>` of class `<className>` to be `<def_value>`. As an example, a part of file *~ns*/tcl/lib/ns-default.tcl is shown in Program 3.5.

In regards to default values, there are two important notes here. First, if no default value is provided for a bound variable, the instproc `warn-instvar {...}` of class `SplitObject` will show the following a warning message on the screen:

```
warning: no class variable <C++ class name>::<OTcl
variable name> see tcl-object.tcl in tclcl for
info about this warning.
```

The second note is that the default value setting in the OTcl domain takes precedence over that in the C++ domain. In C++, the default values are usually set in the constructor. But the values would be overwritten by those specified in the file *~ns*/tcl/lib/ns-default.tcl.

Program 3.5 Examples for default value assignment

```
    //~ns/tcl/lib/ns-default.tcl
1   ErrorModel set enable_ 1
2   ErrorModel set markecn_ false
3   ErrorModel set delay_pkt_ false
4   ErrorModel set delay_ 0
5   ErrorModel set rate_ 0
6   ErrorModel set bandwidth_ 2Mb
7   ErrorModel set debug_ false
    ...
8   Classifier set offset_ 0
9   Classifier set shift_ 0
10  Classifier set mask_ 0xffffffff
11  Classifier set debug_ false
```

3.3.3 *NS2 Data Types*

NS2 defines the following five data types in the OTcl domain: real, integer, bandwidth, time, and boolean.

3.3.3.1 Real and Integer Variables

These two NS2 data types are specified as `double`-valued and `int`-valued, respectively, in the C++ domain. In the OTcl domain, we can use "`e<x>`" as "$\times 10^{\texttt{<x>}}$", where `<x>` denotes the value stored in the variable `x`.

Example 3.5. Let `realvar` and `intvar` be a real instvar and an integer instvar, respectively, of an OTcl object "`obj`". The following shows various ways to set[5] `realvar` and `intvar` to be 1200:

```
$obj set realvar 1.2e3
$obj set realvar 1200
$obj set intvar 1200
```

□

3.3.3.2 Bandwidth

Bandwidth is specified as `double`-valued in the C++ domain. By default, the unit of bandwidth is bits per second (bps). In the OTcl domain we can add the following suffixes to facilitate bandwidth setting.

[5] See the OTcl value assignment in Appendix A.2.4.

- "k" or "K" means kilo or $\times 10^3$,
- "m" or "M" means mega or $\times 10^6$, and
- "B" changes the unit from bits to bytes.

NS2 only considers leading character of valid suffixes. Therefore, the suffixes "M" and "Mbps" are the same to NS2.

Example 3.6. Let bwvar be a bandwidth instvar of an OTcl object "obj." The different ways to set bwvar to be 8 Mbps (megabits per second) are as follows:

```
$obj set bwvar 8000000
$obj set bwvar 8m
$obj set bwvar 8Mbps
$obj set bwvar 8000k
$obj set bwvar 1MB
```
□

3.3.3.3 Time

Time is specified as double-valued in the C++ domain. By default, the unit of time is second. Optionally, we can add the following suffixes to change the unit.

- "m" means milli or $\times 10^{-3}$,
- "n" means nano or $\times 10^{-9}$, and
- "p" means pico or $\times 10^{-12}$.

NS2 only reads the leading character of valid suffixes. Therefore, the suffixes "p" and "ps" are the same to NS2.

Example 3.7. Let timevar also be a time instvar of an OTcl object "obj." The different ways to set timevar to 2 ms are as follows:

```
$obj set timevar 2m
$obj set timevar 2e-3
$obj set timevar 2e6n
$obj set timevar 2e9ps
```
□

3.3.3.4 Boolean

Boolean is specified as either true (or a positive number) or false (or a zero) in the C++ domain. A boolean variable will be true if the first letter of the value is greater than 0, or "t," or "T." Otherwise, the variable will be false.

Example 3.8. Let boolvar be a boolean instvar of an OTcl object "obj." The different ways to set boolvar to be true and false are as follows:

```
# set boolvar to be TRUE
$obj set boolvar 1
```

```
$obj set boolvar T
$obj set boolvar true
$obj set boolvar tasty
$obj set boolvar 20
$obj set boolvar 3.37
$obj set boolvar 4xxx

# set boolvar to be FALSE
$obj set boolvar 0
$obj set boolvar f
$obj set boolvar false
$obj set boolvar something
$obj set boolvar 0.9
$obj set boolvar -5.29
```

□

Again, NS2 ignores all letters except for the first one. As can be seen from Example 3.8, there are several strange ways for setting a boolean variable (e.g., `tasty`, `something`, `-5.29`).

Example 3.9. The following program segment shows how NS2 performs value assignment to instvars `debug_` and `rate_` of class `ErrorModel`.

```
# Create a Simulator instance
set ns [new Simulator]

# Create an error model object
set err [new ErrorModel]

# Set values for class variables
$err set debug_ something
$err set rate_ 12e3

# Show the results
puts "debug_(bool) is [$err set debug_]"
puts "rate_(double) is [$err set rate_]"
```

The results of execution of the above program are as follows:

```
>>debug_(bool) is 0
>>rate_(double) is 12000
```

□

During the conversion, the parameter suffixes are converted (e.g., "`M`" is converted by multiplying 10^6 to the value). For boolean data type, NS2 retrieves the first character in the string and throws away all other characters. If the retrieved character is a positive integer, "`t`," or "`T`," NS2 will assign a `true` value to the bound C++ variable. Otherwise, the variable will be set to `false`.

Table 3.2 NS2 data types, C++ mapping classes, and the corresponding C++ data types

NS2 data type	C++ mapping class	C++ data type
Integer	`InstVarInt`	`int`
Real	`InstVarReal`	`double`
Bandwidth	`InstVarBandwidth`	`double`
Time	`InstVarTime`	`double`
Boolean	`InstVarBool`	`bool`

3.3.4 Class Instvar

Class `Instvar` is a C++ class which binds member variables of OTcl and C++ classes together. It has five derived classes, each for one of the NS2 data types defined in Sect. 3.3.3. These five classes and their mapping class are shown in Table 3.2.

3.4 Execution of C++ Statements from the OTcl Domain

This section focuses on "method binding," which makes an *OTcl commands* available in the C++ domain. There are two types of method binding: OOP binding and non-OOP binding. The OOP binding, binds the method using the C++ function `command(...)` associated with a C++ class. In this case, the command string is called an "OTcl command." For non-OOP binding, the string – called "Tcl command" – is not bound to any class, and can be executed globally. It is not advisable to extensively use Tcl commands, since they violate the OOP principle.

3.4.1 OTcl Commands in a Nutshell

3.4.1.1 OTcl Command Invocation

The invocation of an OTcl command is similar to that of an instproc:

```
$obj <cmd_name> [<args>]
```

where `$obj` is a TclObject, `<cmd_name>` is the OTcl command string associated with `$obj`, and `<args>` is an optional list of input arguments.

Program 3.6 Function `command` of class `TcpAgent`

```
    //~ns/tcp/tcp.cc
1   int TcpAgent::command(int argc, const char*const* argv)
2   {
3       ...
4       if (argc == 3) {
5           if (strcmp(argv[1], "eventtrace") == 0) {
6               et_ = (EventTrace *)TclObject::lookup(argv[2]);
7               return (TCL_OK);
8           }
9           ...
10      }
11      ...
12      return (Agent::command(argc, argv));
13  }
```

3.4.1.2 C++ Definition of OTcl Commands

OTcl commands are defined in the function `command(argc,argv)` of C++ classes in the compiled hierarchy. This function takes two input parameters: "`argc`" and "`argv`," which are the number of input parameters and an array containing the input parameters, respectively.

Example 3.10. Consider an OTcl command `eventtrace{et}` associated with an OTcl class `Agent/TCP`, where "`et`" is an event tracing object. Program 3.6 shows the details of this OTcl command. Suppose `$tcp` and `$obj` are an `Agent/TCP` object and an event tracing object, respectively. The execution of OTcl command `eventtrace` is as follows:

```
$tcp eventtrace $obj
```

The OTcl command `eventtrace` and the event tracing object `$obj` are stored in `argv[1]` and `argv[2]`, respectively.[6] Line 5 returns `true`, and Line 6 stores the input parameter `argv[2]` in the class variable `et_`. □

3.4.1.3 Creating Your Own OTcl Commands

Here are the main steps in defining an OTcl command:

1. Pick a name, the number of input arguments, and the OTcl class for an OTcl command.

[6] As we shall see, `argv[0]` always contains the string "`cmd`."

2. Define a C++ function `command(argc,argv)` for the shadow C++ class in the compiled hierarchy.
3. Within the function `command(...)`, create two conditions which compare the number of input of arguments with `argc` and the name of the OTcl command with `argv[1]`. Specify the desired C++ statements, if both the conditions match. Return `TCL_OK` after the C++ statement execution.
4. Specify the default return statement in the case that no criterion matches with the input OTcl command. For example, Line 12 in Program 3.6 executes the function `command(...)` attributed to the base class `Agent`, and returns the execution result to the caller.

3.4.2 The Internal Mechanism of OTcl Commands

3.4.2.1 OTcl Command Invocation Mechanism

As discussed in Sect. 3.4.1, the syntax for the OTcl command invocation is similar to that of instproc invocation. Therefore, the internal process is similar to the instproc invocation mechanism discussed in Appendix A.2.4.

The process begins by executing the following OTcl statement:

```
$obj <cmd_name> [<args>]
```

If the class corresponding to `$obj` contains, either the instproc `<cmd_name>` or the instproc "`unknown`", it will execute the instproc. Otherwise, the process would move to the base class and repeat itself. In case of OTcl commands, the top-level class is class `SplitObject`. The function `unknown` of class `SplitObject` is specified in Program 3.7. The key statement in this instproc is "`$self cmd $args`" in Line 2, which according to the above OTcl command syntax can be written as follows:

```
$obj cmd <cmd_name> <args>
```

Program 3.7 Instproc `unknown` of class `SplitObject`

```
   //~tcl/tcl-object.tcl
1  SplitObject instproc unknown args {
2      if [catch "$self cmd $args" ret] {
3          set cls [$self info class]
4          global errorInfo
5          set savedInfo $errorInfo
6          error "error when calling class $cls: $args" $
           savedInfo
7      }
8      return $ret
9  }
```

Table 3.3 Description of elements of array "argv" of function command

Index (i)	Element (argv[i])
0	cmd
1	The command name (<cmd_name>)
2	The first input argument in <args>
3	The second input argument in <args>
⋮	⋮

The string "cmd" is the gateway to the C++ domain. Here, the string "cmd $args" (i.e., "cmd <cmd_name> [<args>]") is passed as an input argument vector (argv) to the function "command(argc,argv)" of the shadow class (eg., TcpAgent) as specified in Table 3.3.[7]

Next, the function command(argc,argv) compares the number of arguments and the OTcl command name with argc and argv[1], respectively. If both match, it takes the desired actions and returns TCL_OK (e.g., see Program 3.6).

3.4.2.2 OTcl Default Returning Structure

Owing to its OOP nature, NS2 allows OTcl commands to propagate up the hierarchy. That is, an OTcl command of a certain OTcl class can be specified in the function command(...) of the shadow class or in the function command() of any of its parent classes. If the input OTcl command (i.e., argv[1]) does not match with the string specified in the shadow class, function command(...) will skip to execute the default returning statement in the last line (e.g., Line 12 in Program 3.6).

The default returning statement first passes the same set of input arguments (i.e., (argc,argv)) to the function command(...) of the base class. Therefore, the same process of comparing the OTcl command and C++ statement execution will repeat in the base class. If the OTcl command does not match, the default returning process will be carried out recursively, until the top-level compiled class (i.e., class TclObject) is reached. Here, the function command(...) of class TclObject will report an error (e.g., no such method, requires additional args) and return TCL_ERROR (see file *~tclcl/*Tcl.cc).

3.4.2.3 Interpretation of the Returned Values

In file nsallinone-2.35/tcl8.5.8/generic/tcl.h, NS2 defines five following return values (as 0–5 in Program 3.8), which inform the interpreter of the OTcl command invocation result.

[7] Here, the argc is the number of nonempty element in argv.

Program 3.8 Return values in NS2

```
    //nsallinone-2.35/tcl8.5.8/generic/tcl.h
1   #define TCL_OK          0
2   #define TCL_ERROR       1
3   #define TCL_RETURN      2
4   #define TCL_BREAK       3
5   #define TCL_CONTINUE    4
```

- `TCL_OK`: The command completes successfully.
- `TCL_ERROR`: The command does not complete successfully. The interpreter will explain the reason for the error.
- `TCL_RETURN`: After returning from C++, the interpreter will exit (or return from) the current instproc without performing the rest of instproc.
- `TCL_BREAK`: After returning from C++, the interpreter will break the current loop. This is similar to executing the C++ keyword `break`, but the results prevail to the OTcl domain.
- `TCL_CONTINUE`: After returning from C++, the interpreter will immediately restart the loop. This is similar to executing the C++ keyword `continue`, but the results prevail to the OTcl domain.

Among these five types, `TCL_OK` and `TCL_ERROR` are the most common ones. If C++ returns `TCL_OK`, the interpreter may read the value passed from the C++ domain (see Sect. 3.6.3).

If an OTcl command returns `TCL_ERROR`, on the other hand, the interpreter will invoke procedure `tkerror` (defined in file `~tclcl/tcl-object.tcl`), which shows an error on the screen, and exits the program.

Example 3.11. Consider invocation of an OTcl command associated with an `Agent/TCP` object, `$tcp`. The process proceeds as follows (see also Fig. 3.3):

1. Execute an OTcl statement "`$tcp <cmd_name> <args>`" (position `(1)` in Fig. 3.3).
2. Look for an instproc `<cmd_name>` in the OTcl class `Agent/TCP`. If found, execute the instproc and complete the process. Otherwise, proceed to the next step.
3. Look for an instproc `unknown{...}` in the OTcl class `Agent/TCP`. If found, execute the instproc `unknown{...}` and complete the process. Otherwise, proceed to the next step.
4. Repeat steps (2) and (3) up the hierarchy until reaching the top level class in the interpreted hierarchy (i.e., `SplitObject`).
5. The main statement of the instproc `unknown{...}` of class `SpiltObject` in Program 3.7 is "`$self cmd $args`" in Line 2, which interpolates to

```
$tcp cmd <cmd_name> [<cmd_args>]
```

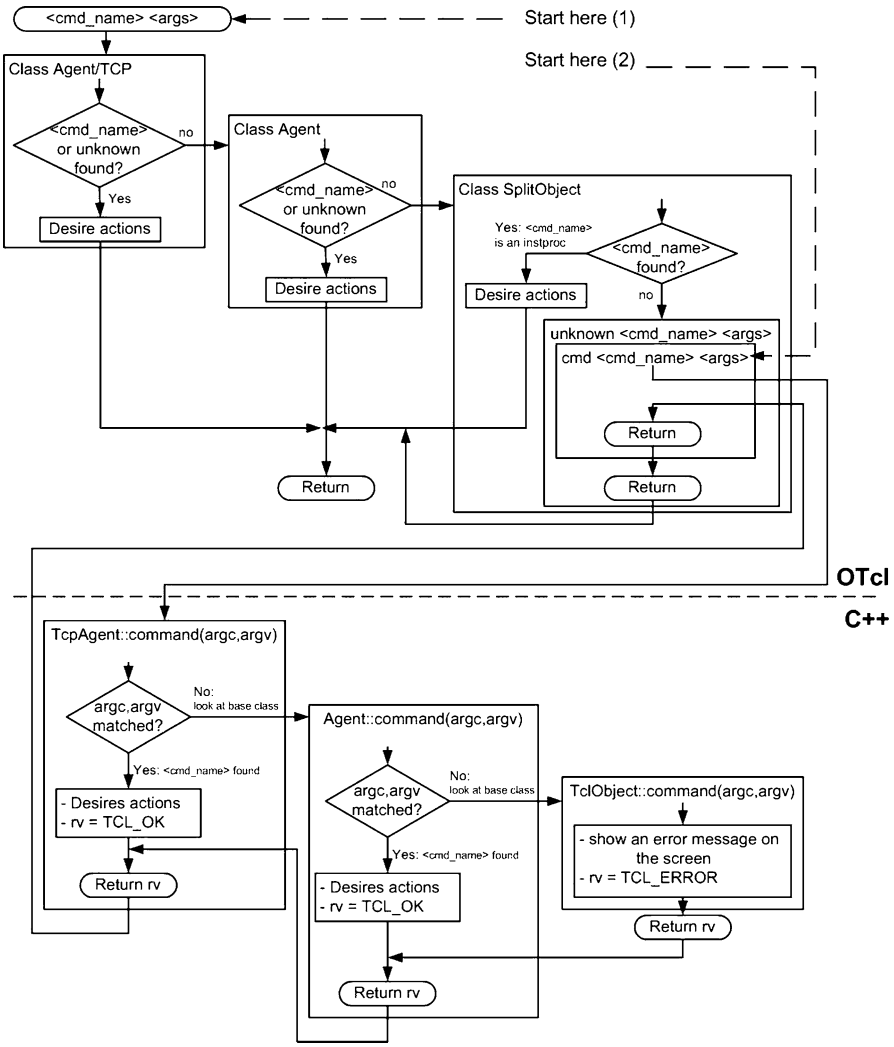

Fig. 3.3 The internal process of OTcl command invocation

6. Execute the function `command(argc,argv)` of class `TcpAgent`, where `argv[0]` and `argv[1]` are `cmd` and `<cmd_name>`, respectively.
7. Look for the matching number of arguments and OTcl command name. If found, execute the desired actions (e.g., Lines 6 and 7 in Program 3.6) and return `TCL_OK`.

8. If no criterion matches with (`argc`, `argv`), skip to the default returning statement (e.g., Line 12 in Program 3.6), moving up the hierarchy and executing the function `command(...)`.
9. Repeat steps (6)–(8) up the compiled hierarchy until the criterion is matched. Regardless of (`argc,argv`), the top-level class `TclObject` in the compiled hierarchy set the return value to be `Tcl-Error`, indicating that no criteria along the entire hierarchical tree matches with (`argc,argv`).
10. Return down the compiled hierarchy. When reaching C++ class `TcpAgent`, return to the OTcl domain with a return value (e.g., `TCL_OK` or `TCL_ERROR`). Move down the interpreted hierarchy and carry the returned value to the caller. □

3.4.3 An Alternative for OTcl Command Invocation

In general, we invoke an OTcl command by executing

```
$obj <cmd_name> <args>
```

which starts from position `(1)` in Fig. 3.3. Alternatively, we can also invoke a the command using the following syntax:

```
$obj cmd <cmd_name> <args>
```

which starts from position `(2)` in Fig. 3.3. This method explicitly tells NS2 that `<cmd_name>` is an OTcl command, not an instproc. This helps avoid the ambiguity when OTcl defines an instproc whose name is the same as an OTcl command name.

3.4.4 Non-OOP Tcl Command

Non-OOP Tcl commands are very similar to OOP Tcl command (i.e.) *OTcl commands*, discussed in the previous section. However, non-OOP Tcl command, also known as *Tcl commands*, are not bound to any class.

3.4.5 Invoking a TclCommand

A TclCommand can be invoked as if it is a global Tcl procedure. Consider the TclCommands `ns-version` and `ns-random`, specified in file *~ns*/common/misc.cc.

- TclCommand `ns-version` takes no argument and returns the NS2 version.
- TclCommand `ns-random` returns a random number uniformly distributed in $[0, 2^{31} - 1]$ when no argument is specified. If an input argument is given, it will be used to set the seed of the random number generator.

Program 3.9 Declaration and function `command(...)` of class `Random-Command`

```
    //~ns/common/misc.cc
1   class RandomCommand : public TclCommand {
2   public:
3       RandomCommand() : TclCommand("ns-random") { }
4       virtual int command(int argc, const char*const* argv);
5   };

6   int RandomCommand::command(int argc, const char*const* argv)
7   {
8       Tcl& tcl = Tcl::instance();
9       if (argc == 1) {
10          sprintf(tcl.buffer(), "%u", Random::random());
11          tcl.result(tcl.buffer());
12      } else if (argc == 2) {
13          int seed = atoi(argv[1]);
14          if (seed == 0)
15              seed = Random::seed_heuristically();
16          else
17              Random::seed(seed);
18          tcl.resultf("%d", seed);
19      }
20      return (TCL_OK);
21  }
```

These two TclCommands can be invoked globally. For example,

```
>>ns-version
2.34
>>ns-random
729236
>>ns-random
1193744747
```

By executing `ns-version`, the version (2.34) of NS2 is shown on the screen. TclCommand `ns-random` with no argument returns a random number.

3.4.5.1 Creating a TclCommand

A TclCommand creation process consists of declaration and implementation. The declaration is similar to that of a TclClass. A TclCommand is declared as a derived class of class `TclCommand`. The name of a TclCommand is provided as an input argument of class `TclCommand` (see Line 3 in Program 3.9).

Similar to that of OTcl commands, the implementation of Tcl commands is defined in function `command(argc,argv)` as shown in Lines 6–21 of Program 3.9. Here, we only need to compare the number of input arguments (e.g., Line 9), since the name of the Tcl command was declared earlier.

Program 3.10 Function `misc_init`, which instantiates of TclCommands

```
    //~ns/common/misc.cc
1   void init_misc(void)
2   {
3       (void)new VersionCommand;
4       (void)new RandomCommand;
5       ...
6   }
```

In addition to the above declaration and implementation, we need to specify active TclCommands in the function `init_misc()` as shown in Program 3.10. Here, each active TclCommands is instantiated by the C++ statement "`(void) new <TclCommand>`." At the startup time, NS2 invokes the function `init_misc(...)` from within the file *~tclcl*/tclAppInit.cc to instantiate all active TclCommands.

3.4.5.2 Defining Your Own TclCommand

To create a TclCommand, you need to

1. Pick a name, the number of input arguments, and the class name for your TclCommand.
2. Derive a TclCommand class directly from class `TclCommand`,
3. Feed the Tcl command name to the constructor of class `TclCommand`,
4. Provide implementation (i.e., desired actions) in the function `command(...)`, and
5. Add an object instantiation statement in the function `init_misc(...)`.

Example 3.12. Let the TclCommand `print-all-args` show all input arguments on the screen. We can implement this TclCommand by including the following codes to the file *~ns*/common/misc.cc:

```
class PrintAllArgsCommand : public TclCommand {
    public:
    PrintAllArgsCommand():TclCommand("print-all-args")
    {};
    int command(int argc, const char*const* argv);
}

int PrintAllArgsCommand::command(int argc,
                        const char*const* argv) {
    cout << "Input arguments: "
    for (int i = 1; i < argc; i++) {
        count << argv[i];
    }
```

```
    return (TCL_OK);
}

void init_misc(void)
{   ...
    (void)new PrintAllArgsCommand;
    ...
}
```

□

3.5 Shadow Object Construction Process

NS2 automatically constructs a shadow compiled object when an OTcl object is created from the interpreted hierarchy. This section demonstrates the shadow object construction process. The process is defined in the top level classes in both the hierarchies – namely classes `TclObject` and `SplitObject`. However, throughout this book, we shall refer to the objects instantiated from these two hierarchies as TclObjects. The term SplitObject shall be used when a clear differentiation for both the hierarchies is needed.

3.5.1 A Handle of a TclObject

A handle is a reference to an object. As a compiler, C++ directly accesses the memory space allocated for a certain object (e.g., `0xd6f9c0`). A handler in C++ is a pointer or a reference variable to the object. OTcl, on the other hand, uses a string (e.g., `_o10`) as a reference to the object. By convention, the name string of a SplitObject is of format `_<NNN>`, where `<NNN>` is a number uniquely generated for each SplitObject.

Example 3.13. Let variables `c_obj` and `otcl_obj` contains C++ and OTcl objects, respectively. Table 3.4 shows examples of the reference value of C++ and OTcl objects.

Table 3.4 Examples of reference to (or handle of) TclObjects

Domain	Variable name	Handle
C++	`c_object`	`0xd6f9c0`
OTcl	`otcl_object`	`_o10`

We can see the value of an OTcl object stored in an OTcl variable by running the following codes:

```
//test.tcl
set ns [new Simulator]
set tcp [new Agent/TCP]
puts "The value of tcp is $tcp"
```

which show the following line on the screen:

```
>>ns test.tcl
The value of tcp is _o10
```

□

3.5.2 *TclObjects Construction Process*

In general, an OTcl object can be created and stored in a variable `$var` using the following syntax:

```
<classname> create $var [<args>]
```

where `<classname>` (mandatory) and `<args>` (optional) are the class name and the list of input arguments for the class constructor, respectively.

This general OTcl object construction approach is not widely used in NS2, since it does not create shadow objects. NS2 uses the following statement to create an object from an interpreted hierarchy:

```
new <classname> <args>
```

This section focuses on how the global procedure "`new`" automatically creates a shadow object. We shall use the OTcl class `Agent/TCP`, bound to the C++ class `TcpAgent` in the C++ compiled hierarchy, as an example to facilitate the explanation.

The TclObject construction process consists of two main parts:

Part I [OTcl Domain]: SplitObject Construction

The main steps in this part are to execute the following OTcl statements and instprocs in sequence (see also Fig. 3.4):

I.1. The OTcl statement `new <classname> [<args>]`
I.2. The OTcl statement `$classname create $o $args`
I.3. The instproc `alloc` of the OTcl class `<classname>`
I.4. The instproc `init` of the OTcl class `<classname>`
I.5. The instproc `init` of the OTcl class `SplitObject`
I.6. The OTcl statement `$self create-shadow $args`

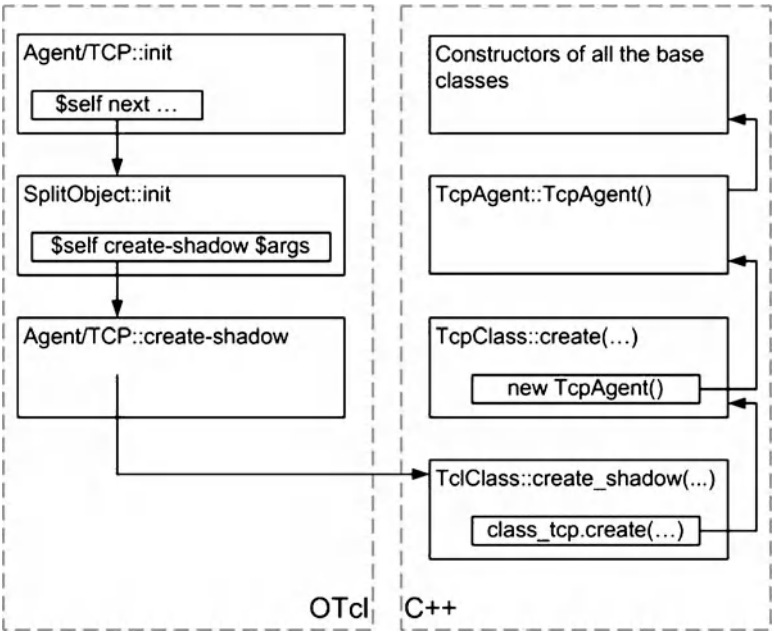

Fig. 3.4 An example of the shadow object construction process: Main steps in the constructor of class `Agent/TCP`

After these steps are complete, the constructed TclObject are returned to the caller. In most cases, the returned object is stored in a local variable (e.g., `$var`).

The details of the above six main steps are as follows. Consider Program 3.11 for the global procedure "`new{className args}`" (Step I.1). Line 2 retrieves the reference string for a SplitObject using the instproc `getid{}` of class `SplitObject`. The string is then stored in the variable "`$o`." Line 3 creates an object whose OTcl class is `$className` and associates the created object with the string stored in "`$o`" (Step I.2). Finally, if the object is successfully created, Line 11 returns the reference string "`$o`" to the caller.[8] Otherwise, an error message (Line 9) will be shown on the screen.

As discussed in Sect. A.2.4, the instproc `create` in Line 3 invokes the instproc `alloc{...}` (Step I.3) to allocate a memory space for an object of class `className`, and the instproc `init{...}` (Step I.4) to initialize the object.

The final two steps are explained through class `Agent/TCP`. Program 3.12 shows the details of constructors of OTcl classes `Agent/TCP` and `SplitObject`. The instproc `next{...}` in Line 2 invokes the instproc with the same name (i.e., `init` in this case) of the parent class. The invocation of instproc `init` therefore keeps moving up the hierarchy until it reaches the top-level class `SplitObject`

[8]Note that Line 11 returns a reference string stored in `$o`, not the variable `$o`.

Program 3.11 Global instance procedures `new` and `delete`

```
    //~tclcl/tcl-object.tcl
 1  proc new { className args } {
 2      set o [SplitObject getid]
 3      if [catch "$className create $o $args" msg] {
 4          if [string match "__FAILED_SHADOW_OBJECT_" $msg] {
 5              delete $o
 6              return ""
 7          }
 8          global errorInfo
 9          error "class $className: constructor failed:
                                                  $msg" $errorInfo
10      }
11      return $o
12  }

13  proc delete o {
14      $o delete_tkvar
15      $o destroy
16  }
```

(see Lines 6–11 in Program 3.12). Here, the instproc `create-shadow` in Line 8 marks the beginning of the C++ shadow object construction process, which will be discussed in Part II. After constructing the shadow object, the process returns down the hierarchy tree, performs the rest of the initialization in instprocs `init` (e.g., of class `Agent/TCP`), and returns the constructed object to the caller.

Program 3.12 The constructor of OTcl classes `Agent/TCP` and `SplitObject`

```
    //~ns/tcl/lib/ns-agent.tcl
 1  Agent/TCP instproc init {} {
 2      eval $self next
 3      set ns [Simulator instance]
 4      $ns create-eventtrace Event $self
 5  }

    //~tclcl/tcl-object.tcl
 6  SplitObject instproc init args {
 7      $self next
 8      if [catch "$self create-shadow $args"] {
 9          error "__FAILED_SHADOW_OBJECT_" ""
10      }
11  }
```

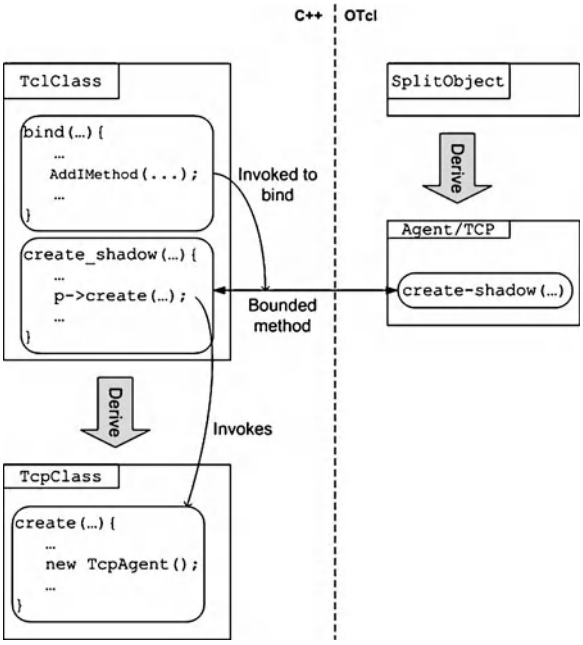

Fig. 3.5 The shadow object creation process: Moving from the OTcl domain to the C++ domain through the OTcl command `create-shadow` of class `Agent/TCP`

Part II [C++ Domain]: Shadow Object Construction

Continuing from Step 6 in Part I, the main steps in this part are to execute the following C++ statements and functions in sequence (see also Fig. 3.4):

II.1. Step 6 in Part I
II.2. The C++ function `create-shadow(...)` of class `TclClass`
II.3. The C++ function `create(...)` of class `TcpClass`
II.4. The C++ statement `new TcpAgent()`
II.5. All related C++ constructors of the class `TcpAgent`

After all the above steps, the constructed shadow object is returned to the caller in the OTcl domain.

The details of Part II are as follows. The first step in Part II is to execute the following OTcl statement from within the instproc `init` of the OTcl class `SplitObject`:

```
$self create-shadow $args
```

where `$self` is an OTcl object whose class is `Agent/TCP`.

It is here where NS2 moves from the OTcl domain to the C++ domain. From Fig. 3.5, the OTcl command `create-shadow` (Step II.1) of class `Agent/TCP` is bound to the C++ function `create_shadow()` of class `TclClass`

(Step II.2). From within the function `create_shadow()`, the statement "`p->create(...)`" is executed, where "`p`" is a pointer to a `TcpClass` object (Step II.3). From Lines 4–6 of Program 3.4, the function `create(...)` executes the C++ statement `new TcpAgent()` to create a shadow `TcpAgent` object (Step II.4). Here, all the constructors along the class hierarchy are invoked (Step II.5). Finally, the created object is returned down the hierarchy back to the caller.

3.5.3 TclObjects Destruction Process

TclObject destruction is the reverse of the TclObject construction. It destroys objects in both the C++ and OTcl domains, and returns the memory allocated to the objects to the system, using a global procedure `delete{...}`. From Program 3.11, Lines 14 and 15 destroy the objects in the OTcl and C++ domains, respectively.

In most cases, we do not need to explicitly destroy objects, since they are automatically destroyed when the simulation terminates. However, object destruction is a good practice to prevent memory leak. We destroy object when it is no longer in need. For example, the instproc `use-scheduler` of the OTcl class `Simulator` executes "`delete $schduler_` before creating a new one (see file ~*ns*/tcl/lib/ns-lib.tcl).

3.6 Access the OTcl Domain from the C++ Domain

This section discusses the following main operations for accessing the OTcl domain from the C++ domain[9]:

1. Obtain the reference to the Tcl interpreter (using the C++ function `instance()`),
2. Execute the OTcl statements from within the C++ domain (using the C++ functions `eval(...)`, `evalc(...)`, and `evalf(...)`),
3. Pass or receive results to/from the OTcl domain (using the C++ functions `result(...)` and `resultf(...)`), and
4. Retrieve the reference to TclObjects (using the C++ functions `enter(...)`, `delete(...)`, and `lookup(...)`).

[9] See the details in the file ~`tclcl/Tcl.cc`.

3.6.1 Obtain a Reference to the Tcl Interpreter

In OOP, a programmer would ask an object to take actions. Without an object, no action shall be taken. In this section, we shall demonstrate how NS2 asks the *Tcl interpreter* (i.e., object) to take actions.

NS2 obtains a C++ object which represents the Tcl interpreter using the following C++ statement:

```
Tcl& tcl = Tcl::instance();
```

where the function `instance()` of class `Tcl` returns the variable `instance_` of class `Tcl` which is a reference to the Tcl interpreter. After executing the above statement, we perform the above operations through the obtained reference (e.g., `tcl.eval(...)`, `tcl.result(...)`).

3.6.2 Execution of Tcl Statements

Class `Tcl` provides the following four functions to execute Tcl statements:

- `Tcl::eval(char* str)`: Executes the string stored in a variable "`str`" in the OTcl domain.
- `Tcl::evalc(const char* str)` : Executes the string "`str`" in the OTcl domain.
- `Tcl::eval()`: Executes the string which has already been stored in the internal variable `bp_`.
- Instproc `Tcl::evalf(const char* fmt,...)`: Uses the format "`fmt`" of `printf(...)` in C++ to formulate a string, and executes the formulated string.

Example 3.14. The followings show various ways in C++, which tell the Tcl interpreter to print out "`Overall Packet Delay is 10.0 seconds`" on the screen.

```
Tcl& tcl = Tcl::instance();

// Using eval(...)
tcl.eval("puts [Overall Packet Delay is 10.0
seconds]");

// Using evalc(...)
char s[128];
strcpy(s,"puts [Overall Packet Delay is 10.0
seconds]");
tcl.evalc(s);
// Using eval()
```

```
char s[128];
sprintf(tcl.buffer(),"puts [Overall
                  Packet Delay is 10.0 seconds]");
tcl.eval();

// Using evalf(...)
float delay = 10.0;
tcl.evalf("puts [Overall
               Packet Delay is %2.1f seconds]",
               delay);
```

Note that `tcl::buffer()` returns the internal variable `bp_`. □

3.6.3 *Pass or Receive Results to/from the Interpreter*

3.6.3.1 Passing Results to the OTcl Domain

Class `Tcl` provides two functions to *pass* results *to* the OTcl domain:

- `Tcl::result(const char* str)`: Passes the string `<str>` as the result to the interpreter.
- `Tcl::resultf(const char* fmt,...)`: Uses the format "`fmt`"of `printf (...)` in C++ to formulate a string, and passes the formulated string to the interpreter.

Example 3.15. Let an OTcl command `return10` of class `MyObject` returns the value "10" to the interpreter. The implementation of the OTcl command `return10` is given below:

```
int MyObject::command(int argc, const char*const*
argv)
{
    Tcl& tcl = Tcl::instance();
    if (argc == 2) {
        if (strcmp(argv[1], "return10") == 0) {
            Tcl& tcl=Tcl::instance();
            tcl.result("10");
            return TCL_OK;
        }
    }
    return (NsObject::command(argc, argv));
}
```

From OTcl, the following statement stores the value returned from the OTcl command "return10 of the C++ object whose class is MyObject in the OTcl variable "val".

```
set obj [new MyObject]
set val [$obj return10]
```

□

Example 3.16. Let an OTcl command returnVal of class MyObject return the value stored in the C++ variable "value" to the interpreter. The implementation of the OTcl command returnVal is given below:

```
int MyObject::command(int argc, const char*const*
argv)
{
    Tcl& tcl = Tcl::instance();
    if (argc == 2) {
        if (strcmp(argv[1], "returnVal") == 0) {
            tcl.resultf("%1.1f",value);
            Tcl& tcl=Tcl::instance();
            return TCL_OK;
        }
    }
    return (NsObject::command(argc, argv));
}
```

The OTcl command returnVal can be invoked as follows:

```
set obj [new MyObject]
set val [$obj returnVal]
```

□

3.6.3.2 Retrieving Results from the OTcl Domain

Class Tcl provides one function to *receive* results *from* the OTcl domain:

- Tcl::result(void): Retrieves the result from the interpreter as a string.

Example 3.17. The following statements stores the value of the OTcl variable "val" in the C++ variable "value":

```
Tcl& tcl=Tcl::instance();
tcl.evalc("set val");
char* value = tcl.result();
```

□

Class Tcl uses a private member variable "tcl_->result" to pass results between the two hierarchies, where "tcl_" is a pointer to a Tcl_Interp object. When passing a result value to the OTcl domain, the function result(...) stores

the result in this variable. The Tcl interpreter is responsible for reading the result value from the variable "`tcl_->result`." On the other hand, when passing a result value to the C++ domain, the interpreter stores the value in the same C++ variable, and the function `result()` reads the value stored in this variable.

3.6.4 TclObject Reference Retrieval

Recall that an object in the interpreted hierarchy always has a shadow compiled object. NS2 records an association of a pair of objects as an entry in its hash table.[10] Class `Tcl` provides the following three functions to deal with the hash table:

- Function `enter(TclObject* o)`: Inserts a TclObject "`*o`" into the hash table, and associates "`*o`" with the OTcl name string stored in the variable "`name_`" of the TclObject "`*o`." This function is invoked by function `TclClass:create_shadow(...)` when a TclObject is created.
- Function `delete(TclObject* o)`: Deletes the entry associated with the TclObject "`*o`" from the hash table. This function is invoked by function `TclClass:delete_shadow(...)` when a TclObject is destroyed.
- Function `lookup(char* str)`: Returns a pointer to the TclObject whose name is "`str`."

Example 3.18. Consider the OTcl command `target{...}` of the C++ class `Connector` in Program 3.13. This OTcl command sets the input argument as the forwarding NsObject (see the details in Sect. 5.3).

Program 3.13 Function `command(...)` of the C++ class `Connector`

```
    //~ns/common/connector.cc
1   int Connector::command(int argc, const char*const* argv)
2   {
3       Tcl& tcl = Tcl::instance();
4       ...
5       if (argc == 3) {
6           if (strcmp(argv[1], "target") == 0) {
7               ...
8               target_ = (NsObject*)TclObject::lookup(argv[2]);
9               ...
10          }
11      ...
12      }
13      return (NsObject::command(argc, argv));
14  }
```

[10] The definition of a hash table is given in Sect. 6.2.3.

Here, "argv[2]" is the forwarding object passed from the OTcl domain. Line 8 executes TclObject::lookup(argv[2]) to retrieve the shadow compiled object pointer corresponding to the OTcl object "argv[2]." The retrieved pointer is converted to a pointer to an object whose type is NsObject and stored in the variable "target_." □

3.7 Translation of Tcl Code

In general, Tcl is an interpreted programming language. It does not require compilation before execution. However, during the compilation, NS2 translates all built-in Tcl modules (e.g., all the script files in directory *~ns*/tcl/lib) into the C++ language using class EmbeddedTcl, to speed up the simulation.

The compilation process is carried out according to the file descriptor, Make file (see Sect. 2.7). The statement related to Tcl translation is

```
$(TCLSH) bin/tcl-expand.tcl tcl/lib/ns-lib.tcl $(NS_
  TCL_LIB_STL) \
| $(TCL2C) et_ns_lib > gen/ns_tcl.cc
```

where $(TCLSH) is the executable file which invokes the Tcl interperter, $(TCL 2C) is a Tcl script which translates Tcl codes to C++ codes, and $(NS_TCL_LIB _STL) are the list of Tcl files which will be translated to C++ programs.

The above statement has two parts, each divided by a pipeline "|" operator.[11]

First Part: Expansion

The first part is shown below:

```
$(TCLSH) bin/tcl-expand.tcl tcl/lib/ns-lib.tcl
 $(NS_TCL_LIB_STL)
```

This part asks the Tcl to interpret (i.e., run) the Tcl script file bin/tcl-expand.tcl, with two input arguments: *~ns*/tcl/lib/ns-lib.tcl and $(NS_TCL_ LIB_STL). It expands the content of all the files specified in the input arguments.

A part of the expansion is to *source* the Tcl files specified in the input files. By sourcing a Tcl file, we mean to replace a source statement with the following syntax

```
source <filename>
```

[11]The pipeline operator captures the screen output (i.e., stdout) resulting from the execution of what ahead of it. The captured string is then fed as a keyboard input (i.e., stdin) for the execution of what following it.

with the content in the file `<filename>`. Example source statements are in the file ~*ns*/tcl/lib/ns-lib.tcl:

```
source ns-autoconf.tcl
source ns-address.tcl
source ns-node.tcl
source ns-rtmodule.tcl
...
```

which tell NS2 to incorporate these files into the translation process.

Second Part: Translation

The second part is located behind the pipe operator ("|"), i.e.,

```
$(TCL2C) et_ns_lib > gen/ns_tcl.cc
```

The statement "`$(TCL2C) et_ns_lib`" translates the OTcl scripts from the former part into C++ programs using an `EmbeddedTcl` object "`et_ns_lib`." The output of the second part is the C++ programs, which are redirected using a redirection operator (i.e., ">"), to the C++ file `gen/ns_tcl.cc`.

3.8 Chapter Summary

NS2 is a network simulator tool consisting of OTcl and C++ programming languages. The main operations of NS2 (e.g., packet passing) are carried out using the C++ language, while the network configuration process (e.g., creating and connecting nodes) is carried out using the OTcl language. In most cases, programmers create object from the OTcl domain, and NS2 automatically creates *shadow* object in the C++ domain. The connection between the interpreted and compiled hierarchies is established through TclCL. In this chapter, we have discussed the main functionalities of TclCL – including class binding, variable binding, method binding, shadow object construction process, Tcl access mechanism, and Tcl code translation.

3.9 Exercises

1. Rewrite the program in Sect. 1.5 using three C++ programming styles discussed in Sect. 3.1.
2. The class binding process consists of four major components.

a. What are those components? What are their definitions?
b. Look into the NS2 codes. What are the components corresponding to the following classes: `RenoTcpAgent`, `Agent`, `Connector`.
c. Bind a C++ class `MyObject` to an OTcl class `MyOTclClass`. What are the main class binding components? Compile and run the codes. Verify the binding by printing out a string from the constructor of the C++ class. Discuss the results.

3. Consider Exercise 2.

a. Create an OTcl command named "`print-count`" which print out the value of `count_` on the screen. Write a Tcl simulation script to test the OTcl command.
b. Create an instproc "`print-count`" for the class `MyOTclObject`. If you execute "`print-count`," which body of "`print-count`" will NS2 execute (i.e., the OTcl command or the instproc)? Design an experiment to test your answer. Can you make NS2 to execute the other body? If so, how?

4. Consider Exercise 2.

a. Declare a variable `my_c_var_` and an instvar `my_otcl_var_` in classes `MyObject` and `MyOTclObject`. Bind them together.
b. Design an experiment to show that a change in one variable automatically updates the value of the other variable.
c. How would you define the default value of a variable in C++ and OTcl domain? If the default values are different, which one would be taken during run time? Design an experiment to prove your answer.

5. What are the major differences among classes `TclObject`, `TclClass`, and `InstVar`? Explain their roles during an object creation process.
6. What are the differences among a C++ function, an OTcl instproc, and an OTcl command?
7. What are the differences between functions `eval(...)` and `evalc(...)`?
8. Show a C++ statement to retrieve a `Tcl` interpreter.
9. The C++ class `TcpAgent` is bound to the OTcl class `Agent/TCP`. The C++ variable "`cwnd_`" bound to the OTcl "`cwnd_`" instvar are used to store the congestion window value.

a. Demonstrate how to set the default congestion window of TCP in the C++ domain to be 20 and in the OTcl domain to be 30. If you set the default value in both the C++ and OTcl domains, what would be the actual default value at run time?
b. Show different ways in the OTcl domain to change the congestion window value of an `Agent/Tcp` object `$tcp` to 40.

10. Can you perform the following actions? If not, what are the conditions under which you can perform such actions.

a. Bind ANY C++ variable to ANY OTcl variable.
b. Call ANY C++ statement from the OTcl domain.
c. Call ANY OTcl statement from the C++ domain.

11. What are the top-level classes in the C++ domain and in the OTcl domain?