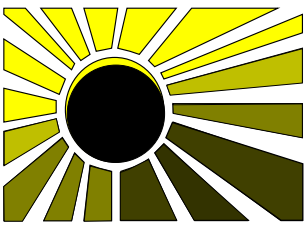
EPCC-SSP / summer 1999

Intersim Project

Benoît Mordelet

Abstract

The Intersim project aims to simulate Internet in order to understand scaling properties of protocols and routing algorithms. However, networks are growing in size and detailed modelling of modern IP networks can be computationally intensive. Efficient design of tomorrow's communications networks represents a crucial challenge for telecommunications companies world-wide. With the continued explosive growth of the Internet it is clear that existing simulation technology and hardware testbed systems cannot keep pace.



Contents

1 Introduction

- What is Intersim project ? In a few words, Intersim is an EPCC project which is developing techniques for carrying out large scale simulation of Internet traffic.
- The Intersim project aims to understand scaling properties of protocols and routing algorithms. However, networks are growing in size and detailed modelling of modern IP networks can be computationally intensive. Efficient design of tomorrow's communications networks represents a crucial challenge for telecommunications companies world-wide. With the continued explosive growth of the Internet it is clear that existing simulation technology and hardware testbed systems cannot keep pace.
- This project involved understanding the source code of ns and modifying it in order to make it run in parallel on a single specific example, FTP traffic. ns is serial code which was originally written at Berkeley University (USA). Writing a parallel scheduler for ns (see further to know what a scheduler is in ns) is one of the Intersim's goals. The example was chosen for its simplicity.
- My contribution to the project is aimed at showing that a parallelisation of ns is possible. We are attempting a new strategy aimed at SMP systems (Shared Memory Platform ?).
- This report is then naturally divided into two main parts : what I have understood of ns-2, and a documentation on the modifications I have done to it. This second part is a complement to the comments I have put in my C++ code, which I tried to write as clearly as possible. It is aimed at explaining in details the work I have done, so that somebody can easily work more on my code if necessary. Notice that every included source code sample is accompanied with the name of the file which contains it. "..." between two code lines means that some code has been truncated. I tried to choose key files to extract code samples.

2 About ns-2 implementation

This first main part describes a bit of ns-2 implementation, the only part I needed to understand and sometimes modify to run ns in parallel. It deals with the mutual calls of C++ and oTcl mechanisms, and how the virtual topology is built and use to simulate data traffic.

Notice that all what is explained in this first part is also "explained" in the official "*ns* Notes and Documentation" book. But since I didn't helped me much during the understanding phase of my project, I'll explain it as I would have like to be told. This part describes almost all that I guessed while analysing the obscure ns-2 source code.

2.1 The source code hierarchy tree

Three directories are particularly interesting in ns-2 source code tree. The tclcl directory contains all the code used two allow C++ / oTcl communications, objects shadowing and variables binding. The ns-2 directory contains all the C++ code of ns which isn't in tclcl; that means the simulator itself code. Finally, the ns-2/tcl/lib directory (and a few files elsewhere) contains the oTcl code of ns, which is incorporated into the C++ executable (you can find a list of all of them in the beginning of the ns-lib.tcl file), i.e. defined in a C++ char[], and given to the interpreter by the C++ code at ns initialisation.

2.2 C++ / oTcl mutual calls mechanisms

Since ns is written in both C++ and oTcl languages, it needs some mechanisms to allow them to communicate, i.e. calling objects instance procedures and returning results. Here is a description of the use of those mechanisms. It is useful mainly when you are trying to avoid oTcl calls in C++ code. Because I don't need it I won't explain in detail the implementation of object shadowing in ns, only the way to call them. You must notice that while being called from C++, oTcl might (and often does) call C++, and vice versa.

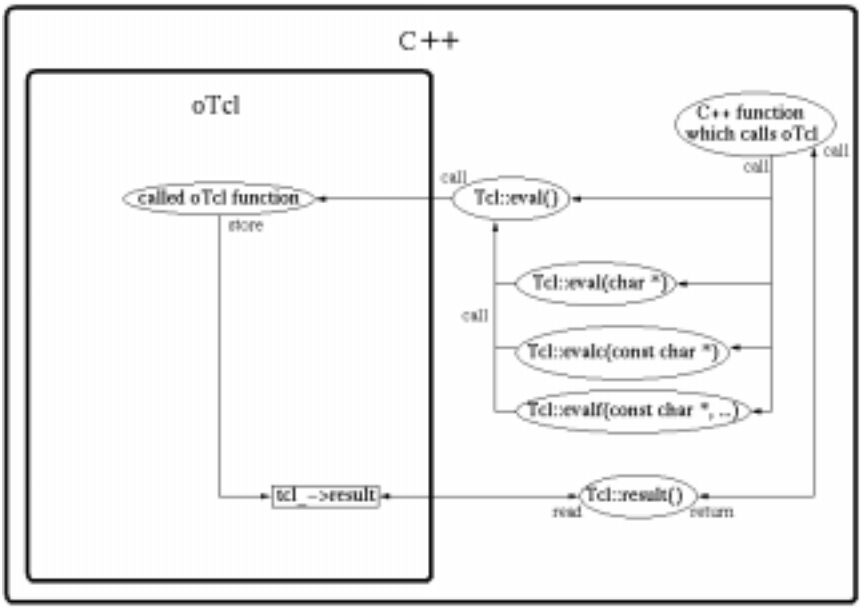


Figure 1: Mechanism to call oTcl at C++ level

2.2.1 Calling oTcl at C++ level

The oTcl interpreter as a C++ object The oTcl interpreter is seen from C++ as an object of the class `Tcl` (figure 1). It is instantiated at ns initialisation and can be accessed like this :

(ns-2/scheduler.cc)

```
void AtHandler::handle(Event* e)
{
        AtEvent* at = (AtEvent*)e;
        Tcl::instance().eval(at->proc_);  // <- here
        delete[] at->proc_;
        delete at;
}
```

`Tcl::instance()` returns a reference (`Tcl &`) to the interpreter and `eval(at->proc_)` will evaluate the oTcl command stored in `at->proc_`.

The eval* functions There are four forms of such eval function :

(tclcl/tclcl.h public part of class Tcl declaration)

```
        /*
         * Hooks for invoking the tcl interpreter:
         *  eval(char*) - when string is in writable store
         *  evalc() - when string is in read-only store (e.g., string consts)
         *  evalf() - printf style formatting of command
         * Or, write into the buffer returned by buffer() and
```

```
         * then call eval(void).
         */
        void eval(char* s);
        void evalc(const char* s);
        void eval();
        char* buffer() { return (bp_); }
        /*
         * This routine used to be inlined, but SGI's C++ compiler
         * can't hack stdarg inlining.  No big deal here.
         */
        void evalf(const char* fmt, ...);
```

So `eval(char* s)` just evaluates the oTcl command stored in s. Since every other eval* function finally calls it, it's the only one we really have to take care about. Notice that `evalf` is by far the most called of those functions; all the calls to the other eval* function could be counted on one hand.

If you expect a result from the interpreter, it can be read using the `result()` instance procedure of the interpreter, like this :

(ns-2/app.cc in the command procedure of the Application class)

```
        Tcl& tcl = Tcl::instance();
        ...
        char result[1024];
        sprintf(result, " %s ", tcl.result());
```

2.2.2 Calling C++ at oTcl level

oTcl code to use a C++ object oTcl doesn't have to declare anything to use C++ objects (figure 2). All the shadowing code is done in C++ (I mean the declaration of a new class, variable, ...). just type the name of the instance you want to use with its arguments on the same line, space separated (self is the oTcl equivalent of the C++ this pointer) :

(ns-2/tcl/lib/ns-link.tcl)

```
SimpleLink instproc ttl-drop-trace args {
        $self instvar ttl_
        if ![info exists ttl_] return
        if {[llength $args] != 0} {
                $ttl_ drop-target [lindex $args 0]
        } else {
                $self instvar drophead_        # <- here
                $ttl_ drop-target $drophead_   # <- here
        }
}
```

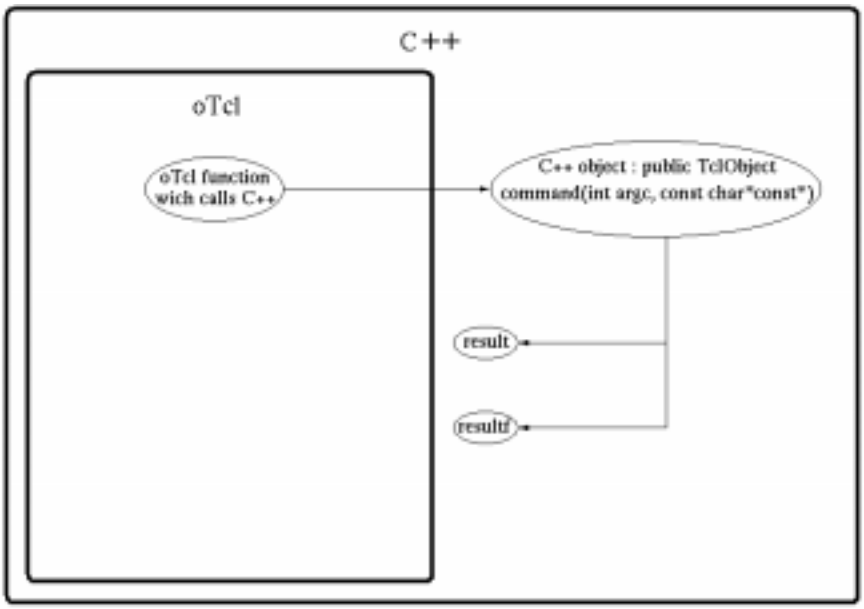


Figure 2: Mechanism to call C++ at oTcl level

The class TclObject and its "command" procedure The class TclObject is the C++ base class of all the shadowed objects. It provides several procedures to bind instance variables and the most important one : `int command(int argc, const char*const* argv)`. this is the function called when oTcl calls the services of that C++ object. The arguments are given like a program command line. `argc` is always above or equal to 2 because `argv[0]` = "cmd", `argv[1]` = command name, and with i>=2 `argv[i]` = command argument. The `command` procedure is supposed to return either `TCL_OK` when everything has gone without any problem, or `TCL_ERROR`.

The structure of a `command` function is always the same :

(ns-2/connector.cc)

```
int Connector::command(int argc, const char*const* argv)
{
        Tcl& tcl = Tcl::instance();
        /*XXX*/
        if (argc == 2) {
                if (strcmp(argv[1], "target") == 0) {
                        if (target_ != 0)
                                tcl.result(target_->name());
                        return (TCL_OK);
                }
                ...
                }
        }

        else if (argc == 3) {
```

```
                if (strcmp(argv[1], "target") == 0) {
                        if (*argv[2] == '0') {
                                target_ = 0;
                                return (TCL_OK);
                        }
                        target_ = (NsObject*)TclObject::lookup(argv[2]);
                        if (target_ == 0) {
                                tcl.resultf("no such object %s", argv[2]);
                                return (TCL_ERROR);
                        }
                        return (TCL_OK);
                }
        ...
        }
        return (NsObject::command(argc, argv));
}
```

The procedure starts parsing the argc/v parameters to know if the command is one of its own and execute it, or return the return code of the parent class command function. Such a structure allows fast and easy addition of commands to a C++ object.

This piece of code also shows the way to give a result string to the interpreter, using `Tcl::result(const char*)` or `Tcl::resultf(const char*, ...)`.

2.2.3 Making C++ objects visible from oTcl

The class TclClass So we now know how to use a C++ object at oTcl level. But we still don't know how to tell the interpreter that a particular class has been defined in C++ and can be used by oTcl. This is done via the `TclClass` C++ class. Just derive that class and instanciate it as a global variable in order to call the constructor once, like this :

(ns-2/scheduler.cc)

```
static class CalendarSchedulerClass : public TclClass {
public:
        CalendarSchedulerClass() : TclClass("Scheduler/Calendar") {}
        TclObject* create(int /* argc */, const char*const* /* argv */) {
                return (new CalendarScheduler);
        }
} class_calendar_sched;
```

This makes the new class usable. It is called `Scheduler/Calendar` and so is a child class of Scheduler; this is a name convention used in ns. This also provides a method to create new instances of that class.

Sharing instance variables This is enough to instanciate a new object at oTcl level, but not to modify an instance variable of that object directly in oTcl. Fortunately, as a derived class of `TclObject` your new oTcl-visible object can also share some of its instance variable with the interpreter. This is done by the bind* functions of the class `TclObject` :

(tclcl/tclcl.h in the public part of class TclObject declaration)

```
        void bind(const char* var, TracedInt* val);
        void bind(const char* var, TracedDouble* val);
        void bind(const char* var, double* val);
        void bind_bw(const char* var, double* val);
        void bind_time(const char* var, double* val);
        void bind(const char* var, int* val);
        void bind_bool(const char* var, int* val);
```

You can that way bind integers, doubles and their traced equivalent, but as well, bandwidth (bw), time duration or boolean :

(ns-2/tcl/lib/ns-default.tcl)

```
Agent/TCP set seqno_ 0
```

(ns-2/tcp.cc in TcpAgent constructor)

```
bind("seqno_", &curseq_);
```

oTcl is here told that `Agent/TCP` has an instance variable called `seqno_`, which default value is 0, and that can be directly read or modified. And C++ defines (in the bind function) the way oTcl will use to perform such accesses.

Defining an object half in C++ and half in oTcl With all these ns-2 features we are able to define some object partly in oTcl and partly in C++. The class `Agent` is defined like this :

(ns-2/agent.h)

```
class Agent : public Connector {
 public:
        Agent(int pktType);
        virtual ~Agent();
        void recv(Packet*, Handler*);
        void send(Packet* p, Handler* h) { target_->recv(p, h); }

        ...

        int command(int argc, const char*const* argv);
 protected:
        ...
        nsaddr_t addr_;                      // address of this agent
        nsaddr_t dst_;                      // destination address for pkt flo
        int size_;                      // fixed packet size
```

```
        int type_;                      // type to place in packet header
        int fid_;                      // for IPv6 flow id field
        int prio_;                      // for IPv6 prio field
        int flags_;                      // for experiments (see ip.h)
        int defttl_;                      // default ttl for outgoing pkts

#ifdef notdef
        int seqno_;             /* current seqno */
        int class_;             /* class to place in packet header */
#endif

        static int uidcnt_;
        int off_ip_;

        ...
};
```

(ns-2/agent.cc)

```
static class AgentClass : public TclClass {
public:
        AgentClass() : TclClass("Agent") {}
        TclObject* create(int, const char*const*) {
                return (new Agent(-1));
        }
} class_agent;

...

Agent::Agent(int pkttype) :
        size_(0), type_(pkttype),
        channel_(0), traceName_(NULL),
        oldValueList_(NULL), app_(0)
{
        off_ip_ = hdr_ip::offset();
#if defined(TCLCL_CLASSINSTVAR)
#else /* ! TCLCL_CLASSINSTVAR */
        /*
         * the following is a workaround to allow
         * older scripts that use "class_" instead of
         * flowid to work -K
         */
        bind("class_", (int*)&fid_);

//        memset(pending_, 0, sizeof(pending_));
        // this is really an IP agent, so set up
        // for generating the appropriate IP fields...
        bind("addr_", (int*)&addr_);
        bind("dst_", (int*)&dst_);
```

```
        bind("fid_", (int*)&fid_);
        bind("prio_", (int*)&prio_);
        bind("flags_", (int*)&flags_);
        bind("ttl_", &defttl_);

#ifdef OFF_HDR
        bind("off_ip_", &off_ip_);
#endif
#endif /* TCLCL_CLASSINSTVAR */
}
```

(ns-2/tcl/lib/ns-agent.tcl)

```
#
# Lower 8 bits of dst_ are portID_.  this proc supports setting the interval
# for delayed acks
#
Agent instproc dst-port {} {
        $self instvar dst_
        return [expr $dst_%256]
}

#
# Add source of type s_type to agent and return the source
# Source objects are obsolete; use attach-app instead
#
Agent instproc attach-source {s_type} {
        set source [new Source/$s_type]
        $source attach $self
        $self set type_ $s_type
        return $source
}
```

If you want to define such mixed class, you have to declare it as usual in C++, make the class visible from oTcl, and bind all the instance variables you want to use in oTcl. Then you can add some procedures in oTcl, but of course you have to use the explicit C++_calls_oTcl mechanism in order to use it from C++.

2.3 The network topology implementation

This second half of the first part describes how the network topology is implemented, and where the different events are inserted to be simulated. It is useful mainly when you are trying to build a conflict checking procedure for the parallel scheduler. We consider here only unicast nodes and simple links.

2.3.1 What oTcl does

oTcl has to build the whole topology at ns initialisation. All the topology objects are C++ object and so are instantiated using the overloaded create function of the class `TclClass`. But oTcl links them (i.e. sets the pointers between them) and chooses the different ingredients : what type of classifier, queue, delay, ... it also initialises the different transport agents (i.e. protocols, I have only used `Agent/TCP`) and Applications (FTP (the only one I have used), Telnet, ...) and attach them together and to the specified nodes. It is the part of oTcl we want to keep because of the facilities it gives to the user to define the topology.

2.3.2 A graph of "connectors"

The base class for all components of the physical network (i.e. nodes and links, not agents or applications) is the `Connector` class. So the network topology can be seen as a graph of Connectors with one-way links between them.

Root classes of a Connector The class `Connector` is derived from the class `NsObject`, which is derived from `TclObject` (which allows object shadowing) and `Handler`. To be of type `Handler` means that it can treat an given event to simulate it. The classes `NsObject` and `Connector` also add their own procedures and variables.

Main instance variables and procedures The `NsObject` class adds the `recv` procedure which will define an object behaviour when it receives a packet. The `handle` procedure is overloaded and simply gives the packet of an event to the `recv` function. The class `Connector` main addition is the `target_` instance variable which allows connector linkage to build the graph.

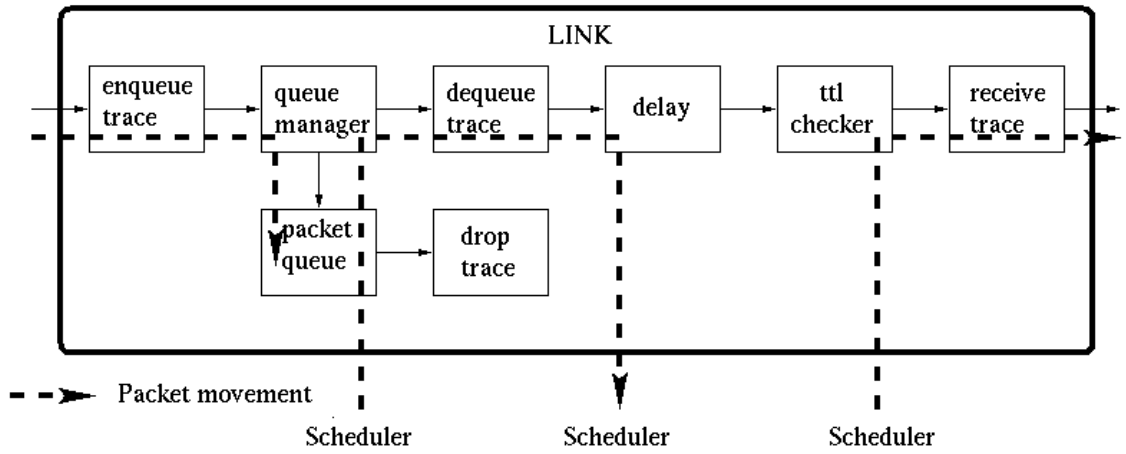


Figure 3: Anatomy of a link

2.3.3 Links anatomy

A little drawing Here is a description of what a link contains. I've added the paths that packets can follow in the link (figure 3).

Description of the connectors The connectors are linked together via their `target_` pointer. They have two main possible behaviour :

1. drop a packet.
2. send the packet to the next connector, after having modified some fields of the packet headers (or not).

In both cases the connector can produce new events and tell the scheduler to insert it in the event queue.

There are eight connectors in a simple link. they are :

1. the enqueue trace. All trace connectors aim at producing one line of the output nam file. This one trace the arriving of packets in the link.
2. the queue manager. It mainly forwards the packets from the enqueue trace to the packet queue, or from the packet queue to the dequeue trace.
3. the packet queue. It is an instance variable of the previous one. It stores the waiting packets and drop some packets when it is full.
4. the drop trace. It traces the packets that are dropped.
5. the dequeue trace. It traces the packets that are dequeued and will continue to the delay connector.
6. the delay. Given a packet, the delay connector schedule a new event which will occur in the ttl checker at current time + delay of the link.

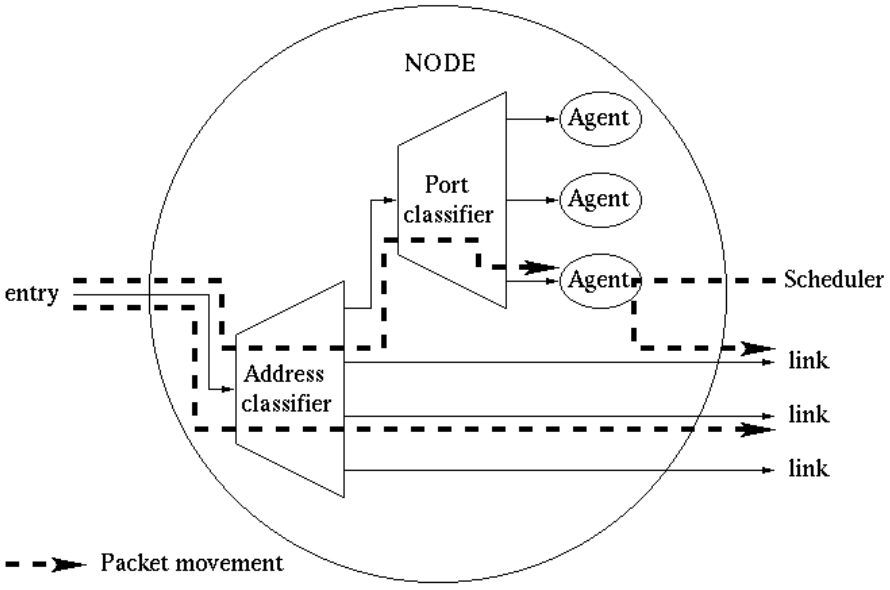


Figure 4: Anatomy of a node

7. the ttl checker. Every packets have a time-to-live variable. Every time the packet goes through a link it is decreased. If it comes to be null, the packet is dropped. That prevents packets to travel for eternity in a network.
8. the receive trace. It just traces the arrival of a packet in a node.

All connectors are potential event handlers, but few are really active. Only the packet queue, the delay connector and the ttl checker could really be used like this. Actually, the delay connector doesn't seem to be used in the simulations I ran.

2.3.4 Nodes anatomy

An other little drawing Here is a description of what a node contains. I've added the paths that packets can follow in the node (figure 4.

Description of the componnents A basic node is constituted of two classifiers. They do the routing in the network topology. They are a bit different than the connectors because their "`target_`" is replace by an array of slots where it can store pointers to another classifier, links, agents, etc...

(ns-2/classifier.h)

```
class Classifier : public NsObject {
 public:
        Classifier();
        ~Classifier();
        void recv(Packet* p, Handler* h);
        int maxslot() const { return maxslot_; }
```

```
        inline NsObject* slot(int slot) {
                if ((slot >= 0) || (slot < nslot_))
                        return slot_[slot];
                return 0;
        }
        int mshift(int val) { return ((val >> shift_) & mask_); }
        NsObject* find(Packet*);
        virtual int classify(Packet *const);

 protected:
        void install(int slot, NsObject*);
        void clear(int slot);
        int getnxt(NsObject *);
        virtual int command(int argc, const char*const* argv);
        void alloc(int);
        NsObject** slot_;        /* table that maps slot number to a NsObject */
        int nslot_;
        int maxslot_;
        int offset_;               // offset for Packet::access()
        int shift_;
        int mask_;
};
```

The routing function for static routing is quite short (and fast) :

(ns-2/classifier.cc)

```
int Classifier::classify(Packet *const p)
{
        return (mshift(*((int*) p->access(offset_))));
}
```

The classifier simply shifts the destination address in the virtual network before masking some bits to get a slot number. I don't know precisely how `shift_` and `mask_` are chosen, but I know that the route computation is done at ns initialisation. The address contains both a node number and a port number which is in fact an agent ID number. The first classifier (address classifier) forwards the packet to the right link if the current node isn't the destination or to the second classifier (port classifier) which forwards it to the right agent.

3 My modifications

This second part is a detailed explanation of what I have modified in ns. It is aimed at helping people who might use my code after I leave EPCC, by preventing them from wondering for hours what I can have tried to do ! The code samples that I have included in this part may of course be the modified code, and may not match with the original ns-2 code. You will have to ask for the EPCC modified code hierarchy if you want to examine it in detail.

There are three main sections :

1. How I manage to avoid oTcl calls from C++ (because they can't be parallelised).
2. How I make ns build a matrix at initialisation to check for events conflict.
3. The problems I had (and still have) with the parallel scheduler itself.

3.1 Avoiding oTcl calls from C++

Because oTcl can't be parallelised, I first had to avoid oTcl calls from C++ during the simulation. It is actually quite simple to do. After having a look at those calls I understood that the at-events are the only ones which require such a call. The at-events are defined in the user's script to describe the traffic sources activity :

(ns-2/tcl/ex/tcp-int.tcl)

```
        set ftp($i) [new Application/FTP]
        $ftp($i) attach-agent $tcp($i)
        $ns at 0.$i "$ftp($i) start"      # <- here
        ...
$ns at 5.0 "finish"                       # <- and here too
```

Since we want to simulate FTP traffic, and since the FTP application is written in oTcl we have to :

1. rewrite the FTP application in C++.
2. send the calls destined to the oTcl FTP application to the C++ one.

Actually, an other type of oTcl call still remains : when stopping, an FTP application receives the "done" order via oTcl. I don't know exactly the role of that function and where it is defined, and since I've been asked to start another part of ns modification after rewriting the FTP application, I still don't know, but I guess it mustn't be that difficult to avoid it.

3.1.1 Adding a new application to ns

To rewrite the FTP application in C++ :

1. “translate” all the oTcl procedures in C++, as instance procedures of a new class derived from class Application.
2. take care of writing a command procedure which match with the exact current oTcl procedures names !
3. make it visible from oTcl.

This application naturally sends orders to the TCP agent it is attached to. The underlying agent is pointed by the `agent_` instance variable of the `Application` class. You can build a command string that way and send it to `agent_->command(...)`. I could have called the right procedure directly of the `TcpAgent` class (with rather dirty code), but it would have failed if somebody one day decide to use an other implementation of the TCP agent. So I have just emulated what oTcl would have done. My code is stored in the files ns-2/myFTP.h and ns-2/myFTP.cc (the oTcl version is in ns-2/tcl/ns-source.tcl). The application can be used exactly like the oTcl version by naming it `Application/myFTP` in an ns script.

3.1.2 Modifying the “eval” function

You can now use the `myFTP` application to simulate FTP traffic, but oTcl is still called... and immediately calls the new C++ object `command` procedure. As I said in the first part, it is done via the `Tcl::eval(const char*)` function. I had to modify that function slightly to call directly the `myFTP` application :

(tclcl/Tcl.cc)

```
static int cut(char *s, char **argv)
{
  int i=0;
  char *p;
  char *st = new char[strlen(s)+1];

  strcpy(st,s);
  p=strtok(st," ");
  argv[0] = new char[strlen(p)+1];
  strcpy(argv[i++],p);
  while ( ((p=strtok(NULL," ")) != NULL) && (i<MAX_TOKENS) )
    {
      argv[i] = new char[strlen(p)+1];
      strcpy(argv[i++],p);
    }
  return i;
}

void Tcl::eval(char* s)
```

```
{
  TclObject *dest;
  int st;
  int argc;
  char *argv[MAX_TOKENS];

  argc=cut(s,argv);

  // new Tcl::eval function
  dest = Tcl::instance().lookup(argv[0]);
  if ( tclobjects.get(dest) == PTR_IN)
    {
      printf("shortcut : \"%s\"",s);
      st = dest->command(argc,argv);
      if (st != TCL_OK)
        {
          printf(" failed - calling OTCL\n");
          st = Tcl_GlobalEval(tcl_,s);
        }
      else printf(" OK\n");
      for (int i=0; i<MAX_TOKENS; i++) delete []argv;
    }
  else
    {
      //      printf("OTCL call : %s\n",s);
      st = Tcl_GlobalEval(tcl_, s);
    }
  if (st != TCL_OK) {
    int n = strlen(application_) + strlen(s);
    char* wrk = new char[n + 80];
    sprintf(wrk, "tkerror {%s: %s}", application_, s);
    if (Tcl_GlobalEval(tcl_, wrk) != TCL_OK) {
      fprintf(stderr, "%s: tcl error on eval of: %s\n",
              application_, s);
      exit(1);
    }
    delete[] wrk;
    //exit(1);
  }

  // old Tcl::eval function
  ...
}
```

A few words of explanation might be necessary (and sorry for not having commented that function, I realise it now !). The `cut` function simply separate the different tokens (space separated) of a given string. Now the `eval` function itself :

1. The parameter string match the following pattern : "_o<number> <command> [<argument> [<argument> ...]]". _o<number> is the name of the destination of the command (an oTcl object) in the oTcl namespace. <command> is the command itself : "start", "stop", "detach-agent", ...
2. So `argv[0]` stores the oTcl name of the destination object. A pointer to the C++ shadowed object is retrieved with the `dest = Tcl::instance().lookup(argv[0]);` line.
3. After testing if the call can be directly sent to C++ (please see the next section for more detail), the `command` function of the C++ object is called, and in case of failure (i.e. if it doesn't return `TCL_OK`) the interpreter is called.

3.2 Maintaining a list of possible shortcuts

I maintain a list of those objects which can be called directly. Those objects have to "register" while being instantiated (i.e. in their constructors), and "unregister" while being destroyed (i.e. in their destructor). This is done using the class `ptrlist`. Since I've modified this class please see the details of the implementation in the paragraph "The ptrlist class" of the next section. It has been modified to extend its possibilities, but is still compatible with this first use. A global instance of that class, called `tclobjects` is declared in ns-2/ptrlist.cc. The definition of that class is stored in the files ns-2/ptrlist.h and ns-2/ptrlist.cc.

I'll describe here the way to use it :

1. To register just use the put procedure.
   (ns-2/myFTP.cc)

```
myFTP::myFTP() : Application()
{
  // may receive requests destinated to oTcl
  tclobjects.put(this);
}
```

2. To unregister just use the del procedure.
   (ns-2/myFTP.cc)

```
myFTP::~myFTP()
{
  // delete that object from the list to free space and accelerate searches
  tclobjects.del(this);
}
```

3. To check if an object is in the list just call the get function. it return PTR_IN if the given pointer is in the list or PTR_NOT_IN else.
   (tclcl/Tcl.cc)

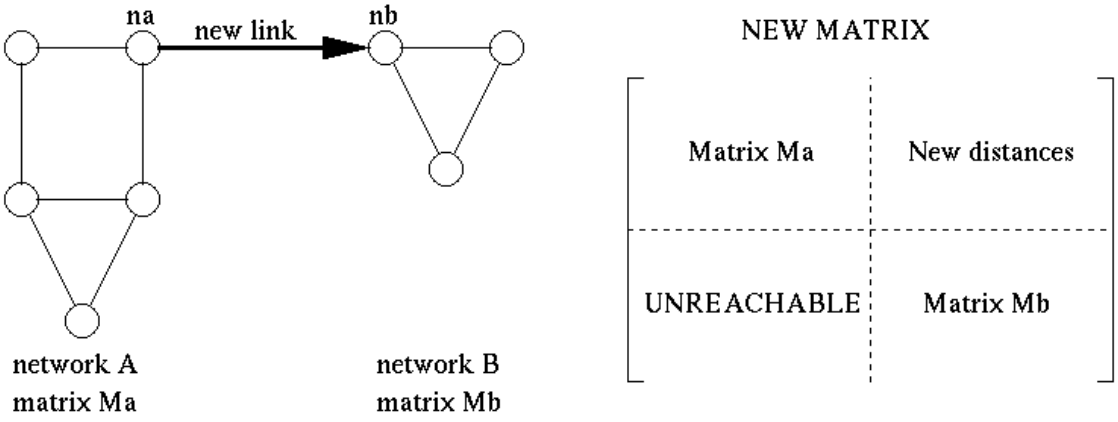


Figure 5: Connecting two networks

```
if ( tclobjects.get(dest) == PTR_IN)
  {
  ...
  }
```

3.3 Building a matrix of temporal distances in a network topology

We can now start thinking about the parallel scheduler itself. It is based on conflict checking of the different events in the queue. We are sure that two events won't conflict if their temporal distance (i.e. difference between their simulation execution time) is shorter than the distance between the nodes where they take place in the network. We need to compute those distances at ns initialisation.

3.3.1 The general case

I wrote the code to compute those distances in the general case, and then derived the new classes to use them in the particular case of ns.

The tempdist_matrix class This class stores the graph of one connected network, and gives a way to connect two of them or add a new path in the graph. This is where the computations of the values in the matrix are really implemented. Two different, but both very simple, algorithms are used to compute the distances between the nodes :

1. We join two disconnected graphs (figure 5) Considering node n1 in graph A and node n2 in graph B we have :

$$dist(n1, n2) = dist(n1, na) + \texttt{delay of the new link} + dist(nb, n2) \quad (1)$$

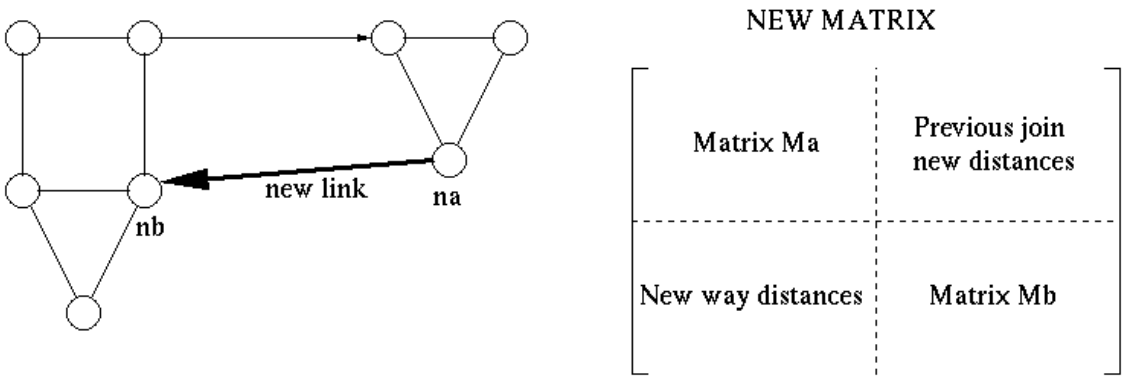


Figure 6: Adding a new path in a connected network

2. We only add a new possible path in the graph (figure 6) Considering nodes n1 and n2 the new distance between them is the minimum of the old distance and the new one calculated as previously.

There are three important procedure in that class : `get`, `connect` and `newway` (please have a look at ns-2/tempdist.cc for details). The `index` instance variable is a list that stores couples (pointer, index in the matrix) because all those three functions consider that the nodes of the graph are C++ pointers. You can so get the distance between two C++ objects, say obj1 and obj2, using `get(&obj1,&obj2)`. The `index` variable is a `ptrlist` like the one I use in the modified `Tcl::eval` function. please have a look at the second next paragraph for implementation details of the class `ptrlist`.

(ns-2/tempdist.h)

```
// "distance" between two unreachable nodes in a graph
#define UNREACHABLE -3.0

// store a connex graph and distances between its nodes
class tempdist_matrix
{
 public:

  tempdist_matrix() {dim=1; matrix = new double[1]; matrix[0]=0.0;}
  tempdist_matrix(void *addr) {dim=1; matrix = new double[1]; matrix[0]=0.0; ind
  ~tempdist_matrix() {delete []matrix;}

  // return the (temporal) distance between two nodes of the graph
  inline double get(void *from, void *to) {return matrix[index.get(from)*dim+ind

  // compute the new matrix when a new link connect two distinct networks.
  // "from" is supposed to be in this matrix, and "to" in the param matrix, else
  void connect(tempdist_matrix m, void *from, void *to, double delay);

  // update the matrix when a new link just add an other possible way for packet
```

```
  void newway(void *from, void *to, double delay);

  // for debugging
  void print(void);


  //private:

  int dim;        // dimension of the matrix
  double *matrix; // the matrix itself
  ptrlist index;  // the bindings between entries in the matrix and pointers ("f
};
```

The tm_set class This class only stores a set of `tempdist_matrix`, and so can store a non-connected graph. Its `connect` function (please have a look at ns-2/tempdist.cc for details) distinguishes if the two given nodes are in the same connected subgraph or not (i.e. in the same `tempdist_matrix`), and so call the right procedure of its root class (`connect` or `newway`). The `get` function does the same and if the two nodes are in the same subgraph return the distance in that graph, or `UNREACHABLE` else. The `list` instance variable stores all the disconnected parts of the graph. The `newnode` function creates a new disconnected part of the graph, with one node and no link in it.

(ns-2/tempdist.h)

```
// store a (possibly) non connex graph
class tm_set
{
  public :

    tm_set() {next_entry=0;}
    ~tm_set() {list.desfree();}

    // add a new 1-matrix to the list.
    void newnode(void *node);

    // compute the new matrix, when adding a new simple link.
    void connect(void *n1, void *n2, double delay);

    // get the distance between two nodes.
    double get(void *from, void *to);

    // for debugging
    void print(void);

    //private:

    // return the number of the matrix  in list where node is.
    int getnb(void *node);
```

```
    ptrlist list;          // store all the connex parts of the graph here
    int next_entry;        // next matrix number that can be binded to a matrix
};
```

The ptrlist class That class is a list of elements like this :

(ns-2/ptrlist.h)

```
// the elements of the list
struct cell
{
  cell * next;   // next element in the list
  void * datap;  // pointer data |
  int    datai;  // integer data | we store couples (pointer,integer)
};
```

It is used as a very simple database of couples (pointer, integer). It has two main uses :

1. In the modified `Tcl::eval` procedure, to store what C++ objects can directly receive calls destined to oTcl. In that case only the pointer field of the elements is important. The integer field (which didn't exist in the first version of that class) is simply set to `PTR_IN` : *(ns-2/ptrlist.cc)*

   ```
   // add a new object in the data base.
   // beware : you can really add several times the same couple, and you'll hav
   //          the del function to really erase it.
   // add the couple (addr,PTR_IN)
   void ptrlist::put(void * addr, char lock)
   {
     put(addr,PTR_IN,lock);
   }
   ```

2. In the computation and consultation of temporal distances matrices, it is used to store the raw number of a given C++ object. (pointer to the object, raw number) is stored.

In both cases all the function have a final argument `lock`, which default is `true`, which specify if the mutex locking must be used for thread safety. That is useful to avoid deadlocks when an instance procedure calls an other one.

The use of the `ptrlist` class is quite easy :

1. Use one of the `put` procedure to add a couple to the database. Just give a pointer and (pointer, PTR_IN) will be added or a full couple.
2. Use one of the `del` procedure to remove a couple from the database. Just give a pointer or an integer and the first matching couple in the list will be removed.

3. Use one of the `get` procedure to retrieve a couple from the database. Just give a pointer or an integer and you'll get the corresponding other field of the first matching couple in the list. If no couple matches, the function returns `PTR_NOT_IN`.

three other functions are defined. `destroy` and `desfree` will empty the list, and the second one will `delete` what is pointed by the `datap` field of each element. The `addcat` function adds a constant integer to the `datai` field of every element in the given list and concatenates the two lists. This last function is used only while computing temporal distances matrices, when joining two disconnected networks.

3.3.2 Building the matrix during ns initialisation

The possible handlers We compute the distances between the nodes of the network, but most of the events happen elsewhere (i.e. somewhere in the links). Since the distance between a part of the link and the nearest node is always null or undefined (you can't know how long the packets will wait in the queue). All the active handlers are associated to the nearest node and have a pointer to it (i.e. to its entry, the address classifier) :

(ns-2/scheduler.h)

```
/*
 * The base class for all event handlers.  When an event's scheduled
 * time arrives, it is passed to handle which must consume it.
 * i.e., if it needs to be freed it, it must be freed by the handler.
 */
class Handler {
 public:
        virtual void handle(Event* event) = 0;

        //***** ben *****
        Handler() {graph_node_p = NULL;}
        void ** graph_node_p;
        //***** neb *****
};
```

The oTcl code I have added a few lines to the oTcl code to build the matrix at ns initialisation, without modifying the user's script. I use the `node` procedure of the `Simulator` class to call the `newnode` procedure of the `node_dist` C++ class, the `init` procedure of the `SimpleLink` class and the `simplex-link` procedure of the `Simulator` class to call the `connect` procedure of the `node_dist` C++ class, and the `attach-agent` procedure of the `Simulator` class to set an agent pointer to its node :

(ns-2/tcl/lib/ns-lib.tcl)

```
Simulator instproc init args {
```

```
        $self create_packetformat
        $self use-scheduler Calendar
        $self set nullAgent_ [new Agent/Null]
        $self set-address-format def

        #***** ben *****
        $self create_dist-matrix
        #***** neb *****

        eval $self next $args
}

#***** ben *****
Simulator instproc create_dist-matrix args {
        $self instvar dist_matrix_
        set dist_matrix_ [new DistNode]
}

Simulator instproc print_mutex_times args {
        $self instvar scheduler_
        $scheduler_ print
}
#***** neb *****
...
# Default behavior is changed: consider nam as not initialized if
# no shape OR color parameter is given
Simulator instproc node args {
        $self instvar Node_ dist_matrix_
...
        #***** ben *****
        $dist_matrix_ newnode [$node entry]
        #***** neb *****

        return $node
}
...
Simulator instproc simplex-link { n1 n2 bw delay qtype args } {
        $self instvar link_ queueMap_ nullAgent_ dist_matrix_

        #***** ben *****
        $dist_matrix_ set delay_ $delay
        $dist_matrix_ connect [$n1 entry] [$n2 entry]
        #***** neb *****
...
}
...
Simulator instproc attach-agent { node agent } {
        $node attach $agent
```

```
        #***** ben *****
        $agent set-graph_node [$node entry]
        #***** neb *****
}
```

(ns-2/tcl/lib/ns-link.tcl)

```
SimpleLink instproc init { src dst bw delay q {lltype "DelayLink"} } {
        $self next $src $dst
        $self instvar link_ queue_ head_ toNode_ ttl_
        $self instvar drophead_
...
    #***** ben *****
    $queue_ set-graph_node [$src entry]
    $link_  set-graph_node [$dst entry]
    $ttl_   set-graph_node [$dst entry]
    #***** neb *****
}
```

3.3.3 Consulting the matrix

The `get` function of the `tm_set` class can be used to retrieve the distance between to handlers. It verifies that they are in the same subgraph and return their distance stored in the corresponding matrix, or `UNREACHABLE` else.

Finally we can check for conflicts like this :

(ns-2/pscheduler.cc)

```
//
// check if events e_1 and e_2 conflict
// 0 - no conflict 1 - conflict
//
int
pScheduler::check_conflict(Event* e_1,double t_1, Event* e_2,double t_2){

  void ** p1 = e_1->handler_->graph_node_p;
  void ** p2 = e_2->handler_->graph_node_p;
  double dist;
  double t = fabs(t_2 - t_1);

  //return 1;
 if ((p1==NULL)||(p2==NULL))
   {
     //fprintf(stdout,"check_conflict : homeless event h1=%d h2=%d\n",e_1->handl
     return 1;
   }
 //fprintf(stdout,"normal event\n");
```

```
 dist = node_dist::instance->get(*p1,*p2);
 if (dist == PTR_NOT_IN) {fprintf(stdout,"pb : not in list\n"); return 1;}
 if (dist == UNREACHABLE) {fprintf(stdout,"no conflict found - UNREACHABLE\n");
 if (t < dist) return 0; //{fprintf(stdout,"no conflict found\n"); return 0;}
 return 1;
 // return (t < dist ? 0 : 1);

//  printf("t_1:%f t_2:%f",t_1, t_2);
//  if ((t_2 - t_1) <  0.0001){printf("\n");return 0;}
//  printf("*\n");
//  return 1;
}
```

3.4 Making the parallel scheduler to be one of the multiple schedulers of ns

When you write a script for ns you are allowed to choose what scheduler you want to use. The default is `Scheduler/Calendar`, but adding the line `$ns use-scheduler Heap` would select the heap scheduler. I wrote a few lines to the C++ code to make the parallel scheduler one of those ones. It is the simplest add to ns you can make ! Just make it visible from oTcl :

(ns-2/pscheduler.cc)

```
static class pSchedulerClass : public TclClass {
public:
  pSchedulerClass() : TclClass("Scheduler/pScheduler") {}
  TclObject* create(int, const char*const*) {
return(new pScheduler);
}
} class_psched;
```

3.5 Debugging the parallel scheduler

Now here is the real problem that remains with ns : the parallel scheduler still doesn't work properly. I have spent three weeks debugging it. I have corrected some bugs but at least one remains. It must come from the threads safety, or from the conflict checking as well, but I didn't even manage to find the cause in the two weeks I spent on that bug. The problem is that I can't reproduce it exactly every time, it is a non-deterministic bug (so there is obviously at least a thread safety bug somewhere).

3.5.1 Writing thread safe code

Since the parallel scheduler uses a taskfarm algorithm, with one master thread and several worker threads, the accesses to the event queue must be protected (the worker threads may insert or cancel events in the queue). This job is performed by mutex locking when accessing the queue.

One of the first bug I've corrected was caused by those safety precautions. There was a deadlock while accessing the queue because those functions where calling other mutex protected functions. I solved the problem by writing some non locking procedures only for internal purpose, but I prefer the way I've protected the `ptrlist` class, but I don't have enough time to change it now...

3.5.2 Problems encountered

There were some other problems with the scheduler :

1. The `Scheduler` class defines a function called `double clock()` which returns the current time of simulation. I had to rewrite the declaration to make it overload-able, and to overload it. It now (in the parallel scheduler only of course) returns the current `time` of the thread that calls the function.
2. There is a problem when resizing the event queue. I "solved" it by initialising it large enough for the whole simulation but it doesn't really properly (and efficiently) solve the problem.
3. I changed a few other details because of the new compiler : cc complains where gcc doesn't.

I also wrote a few lines in the parallel scheduler procedures to measure its timings. I measured the total time of simulation (that excludes initialisation), the total times spent by each thread while it locked the event queue, and the total time spent by each thread while waiting to lock the event queue.

4 Conclusion

4.1 The results

I've got a few results about data accesses bottlenecks in the parallel scheduler (there are no communication costs since they are done via global data modifications) : every thread spent in average 5% to 15% of their execution time waiting for the event queue to be available. What is a bit strange is that it doesn't seem to depend on the number of threads involved or the network size. I guess this is because most of this time is spent while waiting for the master thread to free the event queue mutex. This is because the master thread main procedure (`pscheduler::run`) has a waiting loop which always lock the queue - read the queue to check for conflicts - free the queue. I think it would be better to check for conflict only when inserting or cancelling an event in the queue, or when a worker thread has just finished its task.

Anyway, we must be careful about those results since the parallel scheduler isn't fully working yet (I took care of the timings when the results where quite close to the correct output). There are also several `fprintf` statements that might last quite a long time and confuse those results. And I would add that I didn't have access to the bench batch queues on lomond, so other people's jobs may have confuse the results as well.

4.2 About the SSP

I would say that it was a really good experience for me since I had never worked on such a big project before, and since I've learned a lot about parallel computing at EPCC. So I thank my supervisors Martin and Geoff and other intersimers Doug and Kostas for having helped me when I was in trouble and for having given me the opportunity to work on such an interesting project. I thank also all the EPCC employees and all the other SSP students for having spent a good stay in Edinburgh with me.