

An extension of the *ns* simulator for active network research

Guillermo Rodríguez, Pedro Merino*, María-del-Mar Gallardo

Departamento de Lenguajes y Ciencias de la Computación, University of Málaga, Complejo Tecnológico, Campus de Teatinos, 29071 Málaga, Spain

Received 21 December 2000; revised 14 May 2001; accepted 29 May 2001

Abstract

Active Networks (ANs) represent a new paradigm of computer network, which will enable new Internet applications and services and improve end-to-end performance of the existing ones. However, ANs being an emerging technology, there is still a significant lack of tools for the design and evaluation of active network protocols. In particular, network simulators have proven to be very valuable tools and they have been widely used in the Internet research community. In this paper, we present a novel extension for the well-known network simulator *ns* to incorporate AN support. Our solution is versatile yet powerful, providing a consistent framework for researchers to design and evaluate active protocols. © 2002 Elsevier Science B.V. All rights reserved.

Keywords: Active networks; Programmable networks; Protocol simulation

1. Introduction

Active Networks (ANs) [1,2] represent a major breakthrough since they introduce a new technology that allows users to inject customized programs into the network. By providing a *programmable* interface in network nodes, ANs offer the possibility of dynamically constructing and customizing new services and interposing application-specific processing between end systems. This new paradigm has recently proven to be a very valuable tool to study a number of common problems in today's Internet, such as multicast [3], network caching [4], congestion control [5], and many others.

ANs being an emerging technology, there is still a significant lack of tools for the design and evaluation of AN protocols. In particular, network simulators have been widely used for network research within the Internet community, mainly because networks that are large enough to be interesting are also expensive and difficult to control, and therefore are rarely available for experimental purposes. Multi-protocol network simulators can provide a rich environment for experimentation at low cost, while offering additional advantages like the possibility to study large-scale protocol interaction in a controlled environment, facilities for systematic generation of test cases across a wide range of topologies and traffic distributions, or improved data visualization capabilities.

The publicly available network simulator *ns* [6,7] is becoming more and more popular in the Internet research community. Developed as part of the VINT project, *ns* is a multi-protocol, object-oriented, programmable simulator which also includes a number of facilities for large-scale simulations and the capability to interface the simulator to a live network. Its architecture is designed in a way intended to promote extension by users.

The Network simulator *ns* has been used to investigate a number of protocols, including TCP behavior, router queuing policies, different multicast variations, multimedia, wireless and satellite networking and application-level protocols like web caching. As an example of *ns* popularity in the network community, and according to Ref. [6], it was the most commonly used simulator at SIGCOMM' 98.

In this paper we present a novel extension to the network simulator *ns*, which provides support for active networks. The extension is versatile and flexible, supporting networks with a mix of active and non-active nodes, as well as any number of simultaneous different protocols. It requires little modifications, which are compatible with further updates within the *ns* project. Our approach offers a powerful common framework for AN researchers to investigate new Internet services.

The remainder of this paper is organized as follows: in Section 2 we briefly describe the main architectural components of active networks, and then the software architecture of *ns*. Section 3 analyses the problem of adding active network support to *ns*, and describes our solution. In Section 4 we present a number of examples of how active protocols

* Corresponding author. Tel.: +34-952-132797; fax: +34-952-131397.
E-mail addresses: guille@iies.es (G. Rodríguez), pedro@lcc.uma.es (P. Merino), gallardo@lcc.uma.es (M.-M. Gallardo).

can be developed with *ns* and our extension. We review the related work in Section 5 and conclude in Section 6.

2. Preliminaries

In this section, we provide a brief outline of ANs and review the software architecture of the network simulator *ns*.

2.1. Active networks

As described in Ref. [8], three major architectural components comprise the functionality of each active node: the Node Operating System (NodeOS), the Execution Environments (EEs) and the Active Applications (AAs).

The *NodeOS* is responsible for managing the node's resources and mediating the demand for these resources, which include transmission (link bandwidth), processing (CPU cycles) and storage (memory consumption).

Each *Execution Environment* defines a virtual machine that interprets active packets that arrive at the node. This virtual machine can be programmed or controlled through a programming interface (API) exported by the EE. Thus, an EE acts like the “shell” program in a general-purpose operating system. Several EEs might be present on a single active node.

Users obtain services from the active network via *Active Applications*, which program the virtual machine provided by a given EE to implement an end-to-end service. AAs are often referred to as “Active Protocols”. As far as AA developers are concerned, EEs may be characterized by three key attributes:

1. *Degree of Programmability*. Different EEs define different virtual machines: some might implement a “universal” (Turing complete) VM while others might just allow to select one option from a predefined set of choices.
2. *Set of Abstractions* of node resources provided by the EE. For example, whether active packets may store custom information in active nodes and retrieve information installed by other packets, provided that they are authorized to access that information.
3. *Code Distribution System*. The exact mechanism used to load the AA code into the relevant nodes of the network is EE-dependant. The code may be carried in-band with the packet itself, or installed out-of-band. In the latter case, loading may happen during a separate signaling phase or on-demand, upon packet arrival; it may occur automatically or under explicit control.

2.2. Software architecture of *ns*

The network simulator *ns* has been designed in a way intended to promote extension by users. The model provided by the software architecture is “programmable

composability”. In this model, simulation configurations are expressed as *programs*, which compose objects dynamically into arbitrary configurations. The Network simulator *ns* exploits a *split programming model* where the simulation kernel, i.e. the core set of high performance primitives, is implemented in a compiled language (C++) while simulations are defined, configured and controlled by a simulation program written in the Tcl scripting language.

In this split programming model, fine-grained, core simulation objects implemented in C++ are combined through Tcl scripts to compose more powerful, higher level “macro-objects”. For example, a router is composed of demultiplexers, queues, packet schedulers and so on. By implementing each primitive in C++ and composing them using Tcl, a range of different routers can be simulated faithfully. This allows researchers to experiment with custom architectures. Also, this model has a significant effect on performance. Low-level operations like route lookups, packet processing and forwarding and the like are implemented in C++, while higher-level control operations like statistics collection, modeling of link failures and low-rate control protocols are implemented in Tcl.

This composable object model is naturally expressed using object-oriented design. Because of this, *ns* does not use regular Tcl, but an object-oriented version of Tcl (OTcl), developed at MIT. OTcl provides mechanisms for interaction with C++, which allows an object's implementation to be split across the two languages.

This architecture allows two levels of programming. Simple scripts, topology layout and parameter variation can often be done exclusively in OTcl, while more demanding users who want to fine-tune *ns* internals to suit their own needs or want to implement new core components can do it in C++, retaining the full power and flexibility of this language while achieving high performance.

Even when new objects may easily be added to *ns*, and existing ones modified at will, most simulation models are built from a set of predefined components:

- *Nodes* are macro-objects representing both routers and end systems. They are composed primarily of a set of *classifiers* (demultiplexers), which classify packets as they arrive, routing them through outgoing links or delivering them to local agents.
- *Links* are macro-objects composed primarily of a set of *connectors*. Unlike classifiers, connectors have only one target. When a connector receives a packet, performs some sort of processing on it and then forwards it to the next object. A simple link usually comprises an input queue, a delay and a TTL checker.
- *Agents* are low level objects which are typically attached to nodes to model protocol endpoints. Agents can be connected to *applications* and *traffic generators*, which are commonly used to inject traffic into the network or to receive incoming data. Examples of simulated

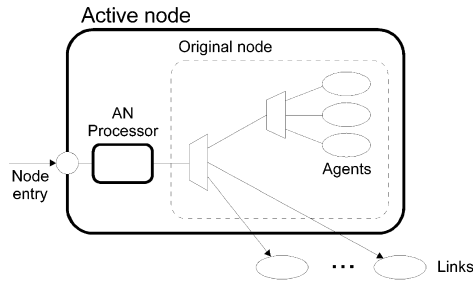


Fig. 1. Structure of a node after activation.

applications are telnet or ftp, while traffic generators include exponential, pareto or CBR sources.

- Other components provide support for LANs, mobile and satellite networking, error models, trace and monitoring support, abstraction techniques for large-scale simulation, and interfacing to live networks.

3. Building active network support into *ns*

In this section we tackle the challenge of incorporating active network support into the network simulator *ns*, and after analyzing the different issues involved, we propose a solution to this problem. In designing this solution, we have set the following goals:

1. It must be versatile and flexible, supporting networks with a mix of active and non-active nodes, as well as any number of simultaneous different protocols. Furthermore, it must be generic enough so as to enable the design and evaluation of active protocols, which will be later implemented for an arbitrary EE.
2. It must require little or no modifications to *ns* internals. The Network simulator *ns* being an evolving project; it is still undergoing continuous changes. Also, because it has been designed so as to promote extension and customization by users, a number of modified versions are being used in the research community. By avoiding unnecessary assumptions about *ns* internal implementation details, we maximize compatibility with future releases and user-modified versions.
3. It must be easy to use. Our primary goal when developing a common framework for AA design and evaluation is to avoid duplication of efforts by providing models for those components which are common to all active networks, like active packet routing and processing or active node storage. It is clear that if this approach is to be effective, ease of use is a must, as is a smooth learning curve.
4. Where possible, it must be kept simple and efficient, as we do not want to preclude the use of our system for large-scale simulations. Special attention will be paid to the scalability problem.

In order to build AN support into *ns*, we have to address

three different problems, which will be dealt with in the remaining of this section: (i) the design and implementation of the necessary infrastructure for active packet routing and processing, (ii) how AAs are to be modeled, and (iii) the design of the network API provided by our “simulated EE” to AAs.

Our system does not explicitly model the code distribution system. Rather, it assumes that the AA code has been already loaded at the relevant nodes. This is because although the code distribution system is a basic aspect for the design of EEs themselves, it has limited impact on AA behavior or performance. Moreover, for most systems this impact is always time-bounded to a short startup period after which all the code has been loaded at relevant nodes, and thus the overall effect on AA performance becomes negligible.

3.1. Basic infrastructure for AN support

First of all, we have to design and implement the basic infrastructure needed for active packet routing and processing.

Since deployment of active nodes in the Internet is expected to be done incrementally, ANs will typically contain both active and non-active nodes. For this reason, active routing protocols are typically expressed as variations on default (IP) forwarding. Specifically, this means that ANs may be seen as regular packet networks in which some nodes will be able to process active packets passing through them.

Our approach has been to rely on default end-to-end routing as implemented in *ns*. In order for intermediate active nodes to detect active packets passing through them, some kind of “router alert” mechanism has to be built into these nodes. With this mechanism in place, when active packets traverse an active node, they are automatically intercepted and evaluated, then forwarded towards their destination.

Node activation is achieved by inserting a new component at the node’s entry and linking it to the next object in that node, typically a classifier. This component is what we have called the *AN processor*, and it implements the “router alert” mechanism. Fig. 1 shows the internal structure of a regular node after activation. The AN processor monitors all packets arriving at the node. Non-active packets pass through transparently, reaching the next component. Active packets are intercepted and processed, and as a result of this processing one or more packets may be forwarded through the node. The AN processor also fulfils an additional role: it implements the “simulated EE” and provides the network API through which AAs access node services.

As active networks might also contain regular (non-active) routers, node activation is done on a per-node basis, by means of the Tcl procedure `enable-active`, which deals with the details described above.

3.2. Modeling active applications

The traditional approach to create a new protocol in *ns* is

Table 1
EE network API

Capsule manipulation	Control operations	Environment access/storage
GetSrc	sendto	getAddr
GetDst	discard	getTime
setDst		getNeighbors
getSize		get
setSize		put
setData		remove

the following. First, a new packet type must be defined, specifying header fields along with their positions and sizes. Then, a new agent must be created by subclassing the C++ virtual class `Agent`, which provides basic agent functionality such as sending and receiving packets through the network. When writing a new agent, a number of virtual methods might be overridden as needed. These methods will typically be called upon certain events, such as packet reception or expiration of timers. Agents also define an OTcl interface in order to allow access from simulation scripts. This interface typically includes operations for agent configuration, attachment of applications and traffic generators, etc.

If this process were to be applied to active application modeling, a new packet type and agent class would have to be defined for each new AA. C++ being a compiled language, this process involves rebuilding the simulator every time a new AA is added. Furthermore, *ns* would have to be rebuilt every time an existing AA undergoes a small change or modification; something that is expected to happen quite often in the development process. While this approach is perfectly valid for traditional networking, where the pace at which new protocols appear is fairly slow, it is clearly not suited to active networking. In such a dynamic field, having to go through an “edit-build-run” cycle repeatedly for each small change rapidly becomes a major hindrance to user productivity.

Instead, our approach has been the following. Rather than writing a new C++ class for each new active protocol, we have developed only one, generic, “active” agent. This agent sends packets carrying the name of a Tcl procedure, which is to be evaluated at each active node, along with a list of user-defined parameters, which represent packet state information. When the AN processor detects an active packet, it extracts the name of the procedure to run and the parameter list, and then invokes this procedure, passing a reference to itself as the last parameter. Through this reference, EE services can be accessed.

This solution has a number of advantages. Because AAs are modeled as Tcl procedures, there is no need to rebuild every time a new AA is introduced or every time something changes in existing AAs. Thus, the “edit-build-run” cycle turns into an “edit-run” one. Also, because Tcl is a higher-level language than C++, start up costs are much lower, and this in turn allows rapid prototyping in the early stages

of the development process. The ease of use of this approach can be readily seen in the compact listings corresponding to the example AAs shown in Section 4. Obviously, this approach also introduces a certain performance penalty, which is evaluated in Section 3.4.

3.3. The execution environment

Different EEs provide different facilities and impose different constraints, mostly related to the degree of programmability or expressive power and the ability to store and retrieve user data in active nodes. In order to allow simulation of active protocols, which might be later implemented for an arbitrary EE, our model must be completely flexible and versatile. The most generic, unconstrained model is that of a full-fledged EE which provides a Turing-complete language, with the ability to store and retrieve user information in active nodes and in which each active packet may invoke a potentially different user-defined forwarding routine when evaluated at active nodes. Other more restrictive EEs may always be considered as subsets of this unconstrained model and thus are also automatically supported.

The ANTS toolkit [9] implements such an EE. The ANTS architecture features the notion of *capsules*, which are active packets containing a byte-coded Java program as well as a payload of user data. The network API provided by ANTS consists of the Java Virtual Machine augmented with the ANTS classes, and thus it provides a Turing-complete language. ANTS also allows capsules to install state information and custom user data at active nodes. Because of its ease of use and flexibility, ANTS has become very popular in the active network community, and has been used in a number of projects.

Our simulated EE supports an API that mirrors that of ANTS. Although our system is by no means tied to ANTS, we have found this approach very useful because of two distinct reasons. First, many active network researchers are already familiar with the ANTS API, so this will simplify the learning process. Second, a number of active protocols have been already implemented for ANTS and their code made publicly available. By having a similar API, we can easily translate these protocols; this allows us to show how our system can be used for the design and evaluation of AAs, while at the same time represents an additional validation of our design.

The EE model also features a *soft-store*, as described in the ANTS literature [9]. This soft-store, often referred to as the node cache, is a repository in which AAs can store arbitrary data. Unlike reliable stores, soft-stores provide no guarantees regarding the persistence of stored data, which is thus defined to be *soft-state*. This enforces robust protocol design, as correct protocol operation must not depend on the availability of this data (although there will be an impact on performance). In the store, data items are cached under AA-defined keys and aged for

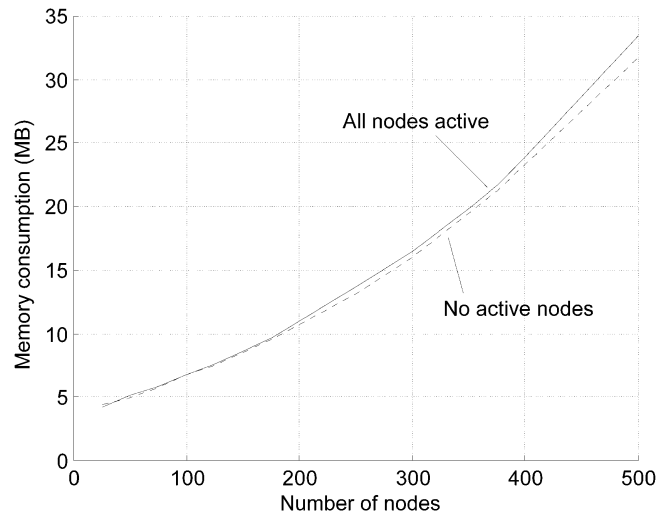


Fig. 2. Memory consumption.

AA-defined intervals, after which they are evicted if still present. Caching improves performance, while aging guarantees that stale state will not be retained within the network.

The API provided by our EE (actually, a subset of it) is shown in Table 1, where the similarities with the ANTS API can be readily seen. We have identified three main categories, which are described next. Most calls behave like their ANTS counterparts, although there are some differences, which might be worth pointing out.

1. *Capsule manipulation* calls allow AAs to access and modify packet header fields, such as source and destination addresses (the later having an immediate effect on capsule forwarding), packet attributes, such as its simulated size, and also, via the `setData` call, any AA-specific state carried by the capsule across the network.
2. *Control operations* calls, which lead to the creation or destruction of capsules within the network.
3. *Environment and storage* calls return information about the local node environment, or provide access to the node soft-store.

Probably, the main difference between our model and that of ANTS lies in capsule forwarding. ANTS requires capsules to explicitly forward themselves from each active node to the next one towards their destination. Also, a capsule willing to be delivered to a local application must explicitly request it. Specifically, if a capsule does not call `routeForNode` or `deliverToApp` on itself before its evaluation finishes, it will be silently discarded. On the contrary, our EE will not discard a capsule unless it is specifically asked to do so by the capsule itself. If the Tcl procedure returns and the `discard` method has not been called, the capsule is automatically routed towards its destination (which can be modified at any time with the `setDst` call) or delivered to its target agent.

3.4. Performance evaluation

Simulation experiments are typically constrained in complexity and scale by computer resource availability, mainly physical memory and CPU power. In order to evaluate the additional performance penalty introduced by our extensions, we conduct several experiments to measure memory consumption and total processing time. All reported results correspond to a standard PC machine used for development: an Intel Pentium II-266 MHz with 256 MB of RAM running ns-2.1b6 under Linux (kernel 2.2.5-15).

Memory consumption might grow due to the additional complexity of the active node infrastructure; thus, it will depend on the total number of active nodes in the topology. Our first set of experiments evaluate this cost by building simulation scenarios for both active and non-active topologies, with a number of nodes ranging from 25 to 500, and measuring memory usage. As can be seen in Fig. 2, the difference is negligible.

Total processing time is likely to increase due to per-node AA execution. This in turn depends on two different factors: (i) the number of active nodes in the *average* route traversed by active packets, and (ii) the complexity of the AA itself.

The second set of experiments evaluate the base overhead introduced by our model by simulating the transmission of a simple UDP-like active capsule through paths with a varying number of active nodes. This capsule is intended to be the equivalent of the “null RPC” that is often used to assess the costs of an implementation. The measured results are compared against values obtained from equivalent non-active experiments using regular UDP agents. As can be seen in Fig. 3, the overhead is reasonably small (about 30% on average). Plotted data sets correspond to scenarios in which a single source generates traffic at a rate of 10 packets/s, with a total simulation length (logical time) of 60 s.

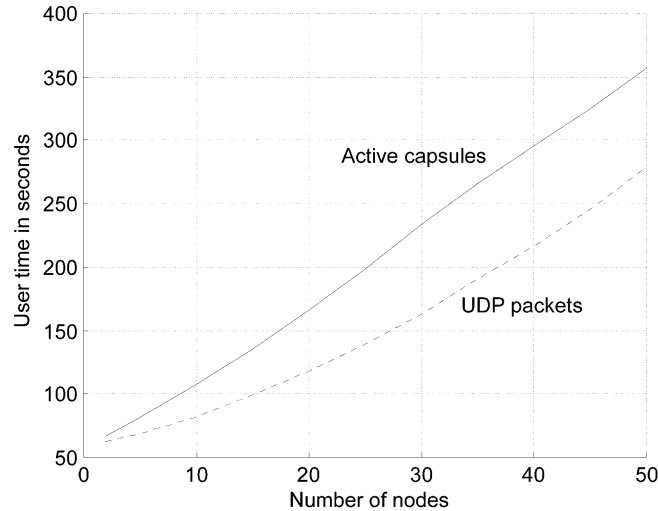


Fig. 3. Processing time.

The additional performance hit due to AA complexity depends on a number of factors and cannot be readily quantified, mainly due to the lack of an equivalent non-active model to use as a baseline for comparison. However, according to our experience, total processing time will remain within the same order of magnitude.

To sum up, this is a price that we are more than willing to pay, in view of the potential benefits and ease of use of our system.

4. Examples

In this section we first present four introductory examples, which aim to demonstrate how we intend our model to be applied for AA design and evaluation. Then we pass on to a real-world application which we have developed using *ns* and our extension.

4.1. Typical examples

The first two examples are ping and trace active protocols, corresponding to the ICMP echo request/reply functions and to the record route IP option. These simple AAs provide a foothold by laying out the basics on how our system is to be used. The code is shown in Fig. 4.

The next two examples are direct translations of the

```
# Ping capsule
#
proc ping { data ee } {
  if { [$see getAddr] == [$see getDst] } {
    $see setDst [$see getSrc]
  }
}

# Trace capsule
#
proc trace { route ee } {
  $see setData "$route [$see getAddr]"
}
```

Fig. 4. Ping and trace protocols.

mobile and multicast protocols presented in the ANTS literature [9]. They were chosen to show how non-trivial (although still admittedly simple) services could be constructed. Also, the similarities between our implementations and the original ANTS versions help us show how easily an AA modeled with our system could be translated into a real implementation.

```
# Mobile register capsule
#
# On entry, 'data' contains three arguments:
#   home = "home" node (HA)
#   forward = next address to be registered
#   mobileid = id of the mobile host
#
# Initially sent towards "foreign" node (FA).
#
proc register { data ee } {
  set home [lindex $data 0]
  set forward [lindex $data 1]
  set mobileid [lindex $data 2]

  # go to foreign, then home
  if { [$see getAddr] == [$see getDst] } {
    $see put $mobileid $forward $MOBILE_TTL

    # if we're at home, discard, else head for home
    if { [$see getAddr] == $home } {
      $see discard
    } else {
      # update the 'forward' parameter
      $see setData "$home [$see getAddr] $mobileid"
      $see setDst $home
    }
  }
}

# Mobile data capsule
#
# On entry:
#   mobileid = id of the mobile host
#
proc data { mobileid ee } {
  # look up forwarding record
  set f [$see get $mobileid]

  # if found, update our route
  if { $f != "" } {
    $see setDst $f
  }
}
```

Fig. 5. Mobile protocol.

```

# Multicast subscribe capsule
#
# On entry, 'data' contains three arguments:
#   group = mcast group
#   sender = mcast sender
#   reverse = last visited node
#
# Initially sent towards the mcast sender
#
proc subscribe { data ee } {
  set group [lindex $data 0]
  set sender [lindex $data 1]
  set reverse [lindex $data 2]

  # first, look up forwarding record
  set m [$see get [key $group $sender]]

  # or create one if necessary
  if { $m == "" } {
    # first element is the time of the last update;
    # set it to -1 here to force update of upstream
    # pointers.
    set m -1
    $see put [key $group $sender] $m $MCAST_TTL
  }

  # are we at an intermediate node?
  if { $reverse != "" } {
    # does the list contain our info?
    set nodes [lrange $m 1 [llength $m]]
    if { [lsearch $nodes $reverse] == -1 } {
      # no, add it
      lappend m $reverse
      $see put [key $group $sender] $m $MCAST_TTL
    }
  }
  set age [expr ([$see getTime] - [lindex $m 0])]

  # need to refresh upstream pointers?
  if { [lindex $m 0] != -1 && $age < $RATE } {
    $see discard
  } else {
    set m [lreplace $m 0 0 [$see getTime]]
    $see put [key $group $sender] $m $MCAST_TTL
    $see setData "$group $sender [$see getAddr]"
  }
}

# Multicast data capsule
#
# On entry, 'data' contains three arguments:
#   group = mcast group
#   sender = mcast sender
#   mdata = multicast data payload
#
proc data { data ee } {
  set group [lindex $data 0]
  set sender [lindex $data 1]

  # look up forwarding record
  set m [$see get [key $group $sender]]

  # must find it to continue
  if { $m != "" } {
    # send a copy every way, then die
    set nodes [lrange $m 1 [llength $m]]
    foreach n $nodes { $see sendto $n "data" $args }
  }
  $see discard
}

```

Fig. 6. Multicast protocol.

In what follows, we sketch a basic overview of how these protocols work. A detailed explanation is out of the scope of this paper, but it can be found in the original documentation.

The mobile protocol provides support for mobile hosts. It comprises two co-operating capsule types, which are shown in Fig. 5. Mobile hosts that are roaming periodically send register capsules to their “home” node, via a local “foreign” node. These capsules enter (or refresh) forwarding pointers in each of these two nodes. When the capsule

reaches the “home” node, it is discarded. The second capsule type is a data capsule, which makes use of this forwarding information. This capsule is first directed towards the base location of the mobile. If the mobile is roaming, the data capsule will be forwarded from the “home” node to the “foreign” node, and from there to the current mobile location. To facilitate shortcut routing, forwarding pointers can be entered at any node and chained together.

The multicast protocol implements a basic unreliable multicast service. It is composed of two co-operating capsule types, shown in Fig. 6. Applications that wish to receive data sent by a given sender to a given group, periodically direct subscribe capsules towards the sender. These capsules install or refresh forwarding pointers in each active node that they traverse; forwarding information sent by different receivers is merged to form a distribution tree. To multicast data to the group, the sender uses a data capsule that routes itself along the distribution tree.

4.2. A case study: the Active Reservation System (ARS) protocol

We have been actively using *ns* and our extension in the development of our ARS protocol. ARS is targeted at travel agencies, and provides mechanisms for client/server interaction in the form of data queries and responses, which are cached in the network using application-specific criteria.

The basic philosophy of ARS is similar to that of the stock quotes protocol developed at MIT [10]. An example case of ARS application is airflight status querying systems. Typically, users (travel agencies) request status information for a given combination of flights. In current systems, this information is assembled in a web page at the server and then sent to the requesting clients. Traditional web caching strategies cannot be applied to this problem, because (i) the data is dynamically changing all the time, and (ii) even when there might be very ‘popular’ data items, every client might request a different combination, so the cache hit ratio would be very low.

One solution, as proposed in Ref. [10], is to cache data with a per-item granularity, instead of caching entire web pages. This way popular items will yield a high hit ratio, no matter which combination is requested. Also, because data undergoes continuous changes, requests must specify a maximum acceptable *degree of staleness*, thus allowing clients to trade response time against currency of data. For example, a general, informative query of the table of flights between two specific airports may be several minutes old, while operations like reservation of seats in a flight might require up-to-date information.

To these basic ideas, ARS adds a number of features, the most remarkable of which is the implementation of a “modulo caching with lookaround” strategy, as described in Ref. [4]. Under the realistic assumption that the universe of different items will be much larger than the (per protocol)

available space for storage at active nodes, the *modulo caching* approach tries to avoid having the same reduced set of items stored in every cache. It only caches data items every n active nodes, doing it in such a way that cached items get uniformly distributed across the network topology. Also, every time a data item is cached, a broadcast message with limited range is sent to neighboring nodes, which annotate where the data has been stored. Thus, nodes that do not have a copy of the item, will at least probably know where to find it (*lookaround*). We are currently investigating other strategies, like self-organizing hierarchical caches.

Another useful feature is the notion of *datasets*, which are just sets of items with similar attributes. A dataset is in itself another kind of item, which contains descriptors for several other items. By specifying a dataset name and a *filter* object, clients may request information about all the items in the dataset that match the filter specification.

Although we do not include a complete model as well as detailed review here due to space concerns, the design and prototyping of ARS has benefited much from our extension to *ns*.

5. Related work

Several authors and researchers in the field of active networks, including Refs. [5,10,11], report the use of modified versions of the network simulator *ns* for their work. The lack of AN support in *ns* has forced these projects to make substantial modifications to the simulator ad hoc, instead of benefiting from a common facility.

Researchers from the *PANAMA* project [11] are also using *ns* to investigate scalable fair reliable multicast protocols in active networks. They have publicly released a package containing the changes and modifications they have introduced in *ns*, in order to help others using *ns* for AN research. Although this package makes a modest attempt at providing generic AN support, it is actually geared towards multicast protocols in general, and towards AER, their own active multicast protocol, in particular. It also has many of the problems discussed earlier. First, it provides just the basic infrastructure needed to route and process active packets, but does not implement EE functionality, so every new active protocol must be started from scratch. Second, it modifies some of the internal core components in *ns*. Third, new AAs are developed following the traditional approach, i.e. implementing new agents in derived C++ classes, which involve rebuilding the simulator for each small change or modification in the AAs under development.

Some AN-specific simulators are being developed. For example, researchers from the Georgia Institute of Technology have been using their locally developed AN simulator, *AN-Sim*, for several projects [4]. *AN-Sim* is specifically targeted at active networks and large topologies, and it seems to be a very interesting project. However, it is still

undergoing intensive development, and it lacks the large established user base that *ns* has.

6. Conclusions

In this paper, we presented an architecture intended to augment the well-known network simulator *ns* adding active network support. Our work addresses the lack of simulation tools for active networks.

We provide all common components needed for modeling and simulation of generic protocols for active networks. These components include the infrastructure for active packet routing and processing, a mechanism for AA modeling, and an EE, which provides an API through which AAs can access node services.

Our system is flexible and versatile. It can simulate hybrid networks with active and non-active nodes, and an arbitrary number of simultaneous active and non-active protocols. Assumptions about *ns* internal implementation details are avoided where possible, and no *ns* component need to be modified. This maximizes compatibility with both future releases and user-modified versions of the simulator. The system has been developed to keep a small memory footprint and low overhead. One major concern is ease of use. By modeling AAs as Tcl procedures, we have been able to turn the “edit-build-run” cycle of the traditional development model into an “edit-run” one. This will considerably increase user productivity, at the price of a modest performance hit.

By providing an API similar to that of ANTS, we have been able to translate a number of example AAs. This validates our approach and demonstrates how easily our system can be used to model and evaluate active network protocols. Our architecture, though, is by no means tied to ANTS, and AAs targeted to an arbitrary EE may be simulated.

Our experience with our system strongly suggests that it will encourage experimentation and hasten the development of new protocols in the exciting field of active networks. First results were presented in Ref. [12] and further progress will be made available at <http://www.lcc.uma.es/~gisum/active.html>.

Acknowledgements

This work has greatly benefited from the discussions in the *ns* mailing list. In particular, we wish to acknowledge Lidia Yamamoto, from the Université de Liège (ULg), Belgium, for her help and insightful comments.

References

- [1] D.L. Tennenhouse, D. Wetherall, Towards an Active Network Architecture, Proceedings of Multimedia Computing and Networking '96, MMCN '96 January 1996.
- [2] K.L. Calvert, S. Bhattacharjee, E. Zegura, J. Sterbenz, Directions in

- active network research, IEEE Communications Magazine 36 (10) (1998) 72–78.
- [3] M. Calderon, M. Sedano, A. Azcorra, C. Alonso, Active network support for multicast applications, IEEE Network 12 (3) (1998) 46–52.
- [4] S. Bhattacharjee, K.L. Calvert, E.W. Zegura, Self-Organizing Wide-Area Network Caches, IEEE INFOCOM '98 March 1998.
- [5] T. Faber, A.C.C.: using active networking to enhance feedback congestion control mechanisms, IEEE Network 12 (3) (1998) 61–65.
- [6] S. Bajaj, et al., Improving simulation for network research, USC Computer Science Department Technical Report 99-702b September 1999.
- [7] K. Fall, K. Varadhan, (Eds.), The *ns* Manual, June 2001 (<http://www-isi.edu/nsnam/ns>).
- [8] K. Calvert (Ed.), Architectural Framework for Active Networks, Draft AN architecture working group, July 1999.
- [9] D. Wetherall, J.V. Guttag, D.L. Tennenhouse, ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols, OPEN-ARCH '98 1998.
- [10] U. Legedza, D. Wetherall, J. Guttag, Improving the Performance of Distributed Applications Using Active Networks, Proceedings of IEEE INFOCOM '98 March 1998.
- [11] S.K. Kasera et al., Scalable fair reliable multicast using active services, IEEE Network Magazine Jan 2000.
- [12] G. Rodriguez, P. Merino, Modeling and Simulation of Active Network Protocols, Proceedings of IEEE International Conference On Distributed Computing Systems Workshop on Internet, Taipei (April 2000).