Implementation of an IPv6 Stack for NS-3

Sébastien Vincent
LSIIT (UMR CNRS 7005)
Louis Pasteur University
Strasbourg, France
vincent@lsiit.u-strasbg.fr

Julien Montavont
LSIIT (UMR CNRS 7005)
Louis Pasteur University
Strasbourg, France
montavont@lsiit.u-strasbg.fr

Nicolas Montavont
IT / Telecom Bretagne
Rennes, France
montavont@telecom-bretagne.eu

ABSTRACT

This paper presents the implementation of an IPv6 stack within the network simulator NS-3. IPv6 is currently being deployed in the world, and should be the Internet Protocol for at least the next fifty years. On another hand, NS-3 aims at being the reference for simulation of the Internet based communication and thus it is important that NS-3 proposes a framework for IPv6. In this paper, we present the main components of our implementation and how we tackle the new mechanisms introduced by IPv6. Finally, we provide some simulation scenarios and results to show that most IPv6 features are already working in our framework, such as the Neighbor Discovery protocol.

Keywords

IPv6, NS-3, multihoming, autoconfiguration

1. INTRODUCTION

The Internet, especially mobile Internet, is in the midst of transitioning to IPv6, yet many network simulation tools are deficient in modeling IPv6. This paper reports on an effort to add an IPv6 stack to the new NS-3 simulator, which presently is IPv4-based only [16].

Several network simulation tools are available today and may target different areas of work. Some of them are free and open-source, while others are commercial. It is not the aim of this paper to give an overview of the existing simulators, but what can be said is that all have advantages and drawbacks. For example Omnet++ [14, 13] is a well-known open-source simulator which offers a good simulation framework in term of ease of development. The INET framework proposes an implementation of IPv6, which consists in the network layer, and the ICMPv6 messages. The Opnet simulator [10] is another project which offers a large choice of protocols, including IPv6. The Opnet IPv6 module is quite complete, and provides implementation for the Neighbor Discovery protocol, ICMPv6, static and dynamic routing protocol or tunneling methods. However this is a commercial tool which requires a non-free license to be used.

Among all available network simulators, the Network Simulator (NS-2) [8] has been largely adopted by the research community. In 2006 an initiative to build a third version of NS was conducted. Today, NS-3 offers the basics of a network simulator, and is ready to welcome new implementation of protocols. However NS-3 focuses on IPv4 only, while IPv6 is being deployed in the Internet. In this paper, we propose an IPv6 framework for NS-3. We choose to contribute to NS-3 because it is an open-source project, and because we beleive that the possible interaction with real operating systems that NS-3 proposes will open new horizons for testing future protocols. This interaction will allow performing tests where some parts of an experimentation will be simulated, while other parts will be real (such as wireless transmission).

The next section presents an overview of the current framework of NS-3 while Section 3 describes the basics of IPv6 and the new mechanisms introduced by this new version of the IP protocol. Then we present in Section 4 our implementation of IPv6 in NS-3, including the Neighbor Discovery protocol which is the main effort that has been made. Section 5 illustrates how our implementation is working over four scenarios that show multihoming support, Neighbor Discovery operations and dual stack (IPv4 - IPv6) support. Finally, future directions for this work are given in Section 6.

2. OVERVIEW OF NS-3 ARCHITECTURE

This section provides a quick overview of the architecture of the NS-3 simulator. First of all, we briefly present its key features followed by a description of the main components that compose the simulator. Finally we analyze the procedures for sending and receiving data packets. A more complete documentation of the simulator architecture is available on the NS-3 website [9].

2.1 General Features

NS-3 is the new version of the well-known object-oriented discrete event network simulator NS-2 that is still one of the most popular network simulator. However, NS-2 has received several criticisms over the years about performance and design limits, such as the fact that a node can only have one IP address. In addition, the combination of C++ and oTCL languages makes new NS-2 models difficult to implement and debug. Theses various issues have motivated the development of a new simulator that would provide better modularity and performance - the NS-3 project

WNS2, October 23, 2008, Athens, Greece.

Digital Object Identifier: 10.4108/ICST.VALUETOOLS2008.4374
http://dx.doi.org/10.4108/ICST.VALUETOOLS2008.4374

was launched.

NS-3 is a discrete event-driven network simulator. The code is under the GNU GPLv2 license which encourages the open source community to participate in the development. The main concepts behind NS-3 are modularity, re-usability and extensibility. Furthermore, the architecture has been designed to support in a near future ambitious features such as distributed simulation, and the integration of native kernel stacks (i.e. from Linux or *BSD) or real applications (e.g. the Quagga Routing Software Suite) directly in the core of the simulator. These projects are currently under heavy development.

Unlike its predecessor, NS-3 is written in pure C++. Although this programming language is not the most accessible one, the developers include several useful mechanisms in order to ease the coding, such as the use of templates and smart pointers for the memory management. They also implement several design patterns as *Factory* or *Singleton* to make easier the interaction with the NS-3 core foundation. For example the *Factory* design pattern allows the creation of objects without specifying the exact class of the object that will be created (e.g. in case of subclassing).

In order to retrieve results from a simulation, most simulators generate text files. NS-3 has a powerful tracing system for decoupling the generation of trace events from the serialization of these ones to a trace file. NS-3 currently supports two serialization formats, plain text and PCAP. The PCAP format is used by the tcpdump and Wireshark tools to represent the exchange of frames over a live network. As these tools have been largely adopted by the network community (students, researchers and industries) especially for their rich set of features, it is very useful to benefit from these applications to analyze simulation results just as live network captures.

2.2 Main components

Now, we focus on the main components (or classes) of the current code of NS-3.

2.2.1 Node

The *Node* class is the representation of a network entity such as a personal computer, a router, etc. A *Node* can aggregate other *Object* as protocol stacks (e.g. IPv4), and therefore have capabilities to process packets. Note that a *Node* could have one or more network interfaces and applications that are respectively implemented by the *NetDevice* and *Application* classes. Further details on these two classes are given in the following subsections.

2.2.2 Topology

The topology part is composed of two classes. As previously mentioned, a network interface is represented by a *NetDevice* object. The medium used to communicate and therefore interconnecting neighbor nodes is represented by a *Channel* object. A *NetDevice* could been seen as the link between a *Node* and the *Channel*. As both of these classes are abstract, they have to be subclassed to match an existing technology: *CsmaNetDevice* and *CsmaChannel* for Ethernet links, *WifiNetDevice* and *WifiChannel* for Wi-Fi links, or *PointToPointNetDevice* and *PointToPointChannel* for point-to-point links. Note that the **NetDevice* and the corresponding **Channel* are linked together and can not be mixed with another type of device or channel (e.g. a *WifiNetDevice* instance can not be linked with a *CsmaChannel*).

For each communication medium, a *Helper* class is available to allow users to build topologies without manipulating these low-level classes. Moreover, these *Helpers* manage the assignment of layer 2 addresses on the *NetDevice* instances, making the creation of simulation scenarios easier for users.

2.2.3 Application

Data that transit on real networks are mainly generated by applications. Users interact with these applications in order to exchange information with one (or more) remote application(s) running on remote host. To represent an application, NS-3 implements the abstract class *Application*. This class mainly interacts with the network stack (for example IPv4 in the case of an *InternetNode*), and exchanges data with other nodes. The creation of a new application is achieved by the derivation of the *Application* class followed by an implementation of its behavior. Similarly to a real application, it uses sockets to communicate via a *Socket* object. Note that an *InternetNode* would respectively use the *TcpSocket* class for TCP flows and the *UdpSocket* class for UDP flows. Both classes are derived from the *Socket* class. Also, an application could use a *PacketSocket* object to send / receive packets dealing directly with a *NetDevice* without passing through the usual protocol stacks (similarly to the *PF_PACKET* socket in Linux).

2.2.4 IPv4 Stack

The current NS-3 implementation is based on the IP protocol version 4 (IPv4 [11]). The IPv4 stack of NS-3 is based on the following classes:

- Ipv4L3Protocol;
- Ipv4L4Protocol;
- Ipv4Interface;
- Ipv4L4Demux;
- Ipv4Route.

In the case of an Ethernet-like technology, a node should also support ARP (Address Resolution Protocol), which is implemented in NS-3 by the *ArpCache*, *ArpL3Protocol*, and *ArpIpv4Interface* classes. We can notice that the *AddInternetStack* function is used to add a fully functionnal IPv4 stack. In addition to the creation of a functional *IPv4L3Protocol* object, this function also adds and configures ARP features and all necessary transport protocols such as UDP or TCP.

The *Ipv4L3Protocol* class holds methods to add interfaces, routes, and assign IPv4 addresses. This class also has an *Ipv4L4Demux* instance which maintains a list of all available *Ipv4L4Protocol* implementations and acts as a demultiplexor to select the *Ipv4L4Protocol* instance that matches the protocol number included in the protocol field of an IP packet. *Ipv4L4Protocol* is an abstract class that corresponds to the layer 4 protocol (i.e. UDP or TCP).

The *Ipv4Route* class is the representation of a route entry in the IPv4 routing table of a node. It includes some fields such as the destination route, the interface index, and the gateway whenever it exists. The stack holds a static routing table, namely *Ipv4StaticRouting*. This class holds three lists,

Digital Object Identifier: 10.4108/ICST.VALUETOOLS2008.4374
http://dx.doi.org/10.4108/ICST.VALUETOOLS2008.4374

one for host destinations, one for unicast network destinations and the last for multicast destinations. It also includes one default route for unicast and one for multicast.

The *Ipv4Interface* class is an abstract class that contains an IPv4 address. The *ArpIpv4Interface* is a subclass of *IPv4Interface* and enables the address resolution through ARP when a node need to transmit packets. The *ArpL3-Protocol* class implements the logic of ARP (messages processing) and it interacts with the *ArpCache* instances, one per interface, which store the following states of an ARP entry: *ALIVE*, *WAIT*, *REPLY* or *DEAD*.

2.3 Writing Scenarios

By contrast to NS-2 in which a scenario is represented by TCL scripts, an NS-3 scenario is a C++ program that has to be compiled. The basic steps to create a simulation are (i) create the *Nodes*, (ii) install the network stack in *Nodes* (e.g. IPv4), (iii) create the *Channels* and configure their parameters (bitrate, delay, etc.), (iv) add *NetDevice(s)* to the *Nodes*, (v) assign network addresses, (vi) install the *Applications* and configure them for each *Node*, and finally (vii) run the simulation.

NS-3 has recently added a class called *Helper* for several components. These *Helper* classes are used to avoid manipulating low-level classes and to reduce the size of the scenario source code. For example, the *Ipv4AddressHelper* class generates IPv4 addresses valid for a specified network and increment the host part at each call. We can also mention the *CsmaHelper* class that helps creating a *CsmaChannel* and a user-specified number of *CsmaNetDevice* on this link (the same *Helper* classes exist for Wi-Fi and Point-to-Point medium).

Furthermore, we can mention that python bindings for the NS-3 API have been integrated in the main development tree. Python is a very popular and accessible programming language. The purpose is to enable the creation of python-based NS-3 scenarios which would decrease the complexity of building a NS-3 scenario for non-C++ users.

3. IPV6 SPECIFICITIES

The protocol used over the Internet for data delivery between hosts is currently the Internet Protocol version 4 (IPv4 [11]). This protocol, even though being nearly thirty years old, has been quite efficient during the past years. However, the increasing popularity of Internet has raised address shortage and route maintenance issues. The address shortage problem is partially tackled by the massive introduction of NATs (Network Address Translators) which allow associating a single public IP address to multiple hosts in a private network. Although such mechanism is encouraged by most of ISPs (e.g. NATs are largely externalized cost of ISPs), the usage of NAT raises several complications in communication between hosts (especially when considering incoming data packets) and may also have a performance impact. Another solution to resolve these issues and enhance the services offered by IP is to adopt the new version of IP known as IP version 6 (IPv6 [2]).

Given the experience acquired with IPv4, IPv6 has been defined to resolve most of the issues observed over the last thirty years in the Internet. First, it extends the address space from 2^{32} to 2^{128}. An IPv6 address is composed of 128 bits: the 64 first bits generally identify the network subnet (also known as subnet prefix) whereas the last 64 bits usually identify a host in this subnet. An IPv6 address is also associated to a scope which specifies the validity of an address: link-local, site-local or global. The validity of link-local addresses is limited to the link (hence the name), i.e. such addresses are only used for communication between direct neighbors. A link-local address is automatically configured on each enabled network interface by combining the `FE80::/64` prefix to the IEEE 802 EUI-64 of the interface. For a site which is not connected to the Internet or which remains private, the site-local scope has been introduced. Theses addresses are equivalent to private IPv4 addresses (e.g. 192.168.0.0). However, such addresses have been deprecated due to routing and site delimitation issues. They have been replaced by Unique Local Address [4]. At last, global IPv6 addresses are routable over the entire IPv6 Internet and therefore are used for communications between any two remote IPv6 hosts.

Note that all popular operating systems now support IPv6. However, IPv6 is not backward compatible with IPv4. As the network migration from IPv4 to IPv6 is progressive, transitional mechanisms have been defined such as dual stack nodes which support both IPv4 and IPv6.

3.1 Neighbor Discovery

IPv6 includes a new protocol known as Neighbor Discovery [6] which achieves various tasks such as router discovery, address resolution (mapping IPv6 addresses with link-layer addresses), address autoconfiguration, neighbor unreachability detection, next-hop determination and host redirection. All messages introduced by Neighbor Discovery are Internet Control Message Protocol for the Internet Protocol Version 6 (ICMPv6 [1]) messages.

Neighbor Discovery introduces a new address configuration procedure known as stateless address autoconfiguration mechanism [12]. A host may automatically configure a valid global IPv6 address upon receiving a Router Advertisement message that a local access router periodically broadcasts over an IPv6 link. A Router Advertisement generally provides the link prefix(es) for global address configuration in addition to the link-layer address of the local access router. Note that the IPv6 link-local address of the router is retrieved from the IPv6 header of the Router Advertisement, so no additional packet exchange is needed to forward IPv6 packets to this router. Upon receiving a Router Advertisement, a host configures a global IPv6 address for each prefix listed in the message by combining these prefixes with the IEEE 802 EUI-64 of the interface that received the message. Also, the host adds the relevant routes in its routing table (default route, route for on-link destination, etc.) and registers in the neighbor cache (equivalent to the ARP cache in IPv4) the mapping between the IPv6 and the link-layer addresses of the router. Furthermore, the access router provides a lifetime for each prefix announced in order to help hosts to know when an address is deprecated and should not be used to initiate new communication. Note that IPv6 also enables a stateful address autoconfiguration through Dynamic Host Configuration Protocol version 6 (DHCPv6 [3]) or manual address configuration.

Before assigning an unicast IPv6 address to an interface (regardless the scope and how the address has been obtained), a host must ensure that this address is unique by performing a Duplicate Address Detection (DAD) [12]. This procedure consists in multicasting a Neighbor Solicitation

Digital Object Identifier: 10.4108/ICST.VALUETOOLS2008.4374
http://dx.doi.org/10.4108/ICST.VALUETOOLS2008.4374

message to request a potential neighbor host which already uses the target address to reply with a Neighbor Advertisement. After a certain amount of time without reception of such a message, the address is determined to be unique and therefore is assigned to the interface by the originator of the DAD procedure. When the DAD fails (i.e. the target address is already in use by a neighbor), the address must not be assigned to the interface.

3.2 Multihoming

Multihoming refers to a situation where a host is reachable through multiple paths, either because this host has several network interfaces connected to various access networks, or because the subnet in which the host is located is multihomed itself. Whereas a host could only have one address per interface in IPv4, IPv6 allows a host to assign several IPv6 addresses (regardless the scopes are) to a single interface. A practical scenario which illustrates such configuration is a link provided with two access routers advertising one unique prefix each. A host would configure at least 3 addresses on the interface connected to this link: one link-local address and two global addresses (referred to as *IP1* and *IP2* in the following), each corresponding to a prefix. Thus, this host could either communicate with *IP1* or *IP2* only, or both *IP1* and *IP2* simultaneously.

3.3 Multicast

The IP multicast is the mechanism to deliver IP packets to multiple destinations simultaneously by sending each packet only once and creating a copy whenever the link to the destination split. Unlike IPv4, IP multicast is part of the base IPv6 specifications. For example, IPv4 broadcast feature is replaced by multicasting to the group representing all on-link hosts. The Multicast Listener Discovery [15] protocol (the equivalent of Internet Group Management Protocol in IPv4) is directly embedded in ICMPv6 messages.

3.4 Header Extension

In addition to a simplified IP header (the number of fields has been divided by 2 compared to the IPv4 header), IPv6 also makes easier the definition of new extensions directly embedded in the IP header. The IPv6 header includes a *next header* field which specifies either the protocol from the upper layer of the OSI model (e.g. TCP or UDP) or an IPv6 extension. Readers may refer to [2] for further details on the format and utility of the most common options.

4. IMPLEMENTATION

In this section we explain how we have implemented an IPv6 stack in NS-3. Our implementation is freely available online [16]. We decide to choose the development version of NS-3 for our contribution. We use it in order to always stay up-to-date with the code, even if we have to re-write some of our code because of changes in the NS-3 API. Basically, we followed the way IPv4 is implemented in order to respect the philosophy of NS-3. From the outset we try to avoid (or at least minimize) radical changes in existing classes in order to remain compliant with the main NS-3 branch. Figure 1 represents the main classes that we develop.

4.1 IPv6 basics

We first focus on supporting the basics of IPv6: address format, packet header and route entry representation. We

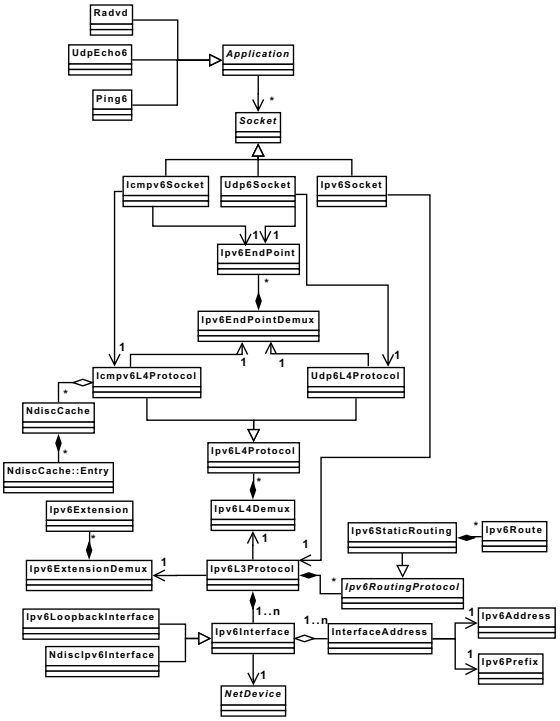


Figure 1: Main IPv6 classes diagram

develop the *Ipv6Interface* class which corresponds to an IPv6 network interface. We also introduce the *Ipv6Address* and *Ipv6Prefix* classes to manage IPv6 addresses (local and global). Note that *Ipv6Prefix* is a bitmask similarly to the *Ipv4Mask* class for IPv4. Such classes allow the distinction between the network part and the host part of an IP address. There are mainly used in the routing process to determine the route for a destination by performing binary AND between addresses and prefixes (i.e. the bitmask). As previously mentioned, a network interface may have several IPv6 addresses. Thus, an *Ipv6Interface* may handle multiple *Ipv6Address* objects. We choose to store each of them in a separate *InterfaceAddress* class as the latter contains information about the state of the associated address (as defined in the DAD procedure [12]).

The logic of IPv6 is done in the class *Ipv6L3Protocol* as represented in Figure 2. Like its IPv4 counterpart, *Ipv6L3Protocol* could contain one or more *Ipv6RoutingProtocol* (which deal with *Ipv6Route* objects) in addition to an instance of the *IPv6StaticRouting* class for the static routing tables. To support multihoming capabilities we add a special field in the route entry: the IPv6 address prefix to use for this destination in order to avoid ingress filtering. *Ipv6L3Protocol* holds also an *Ipv6L4Demux* object which maintains the list of all supported *Ipv6L4Protocol*.

Furthermore, we modify the *NetDevice* class. By now, we only focus on the *CsmaNetDevice* subclass in which we add the necessary methods to recognize the protocol number corresponding to IPv6 in the protocol field of Ethernet frames. We also enable the multicast Ethernet representation of the several well-known multicast IPv6 addresses that are mandatory to complete a functional IPv6

Digital Object Identifier: 10.4108/ICST.VALUETOOLS2008.4374
http://dx.doi.org/10.4108/ICST.VALUETOOLS2008.4374

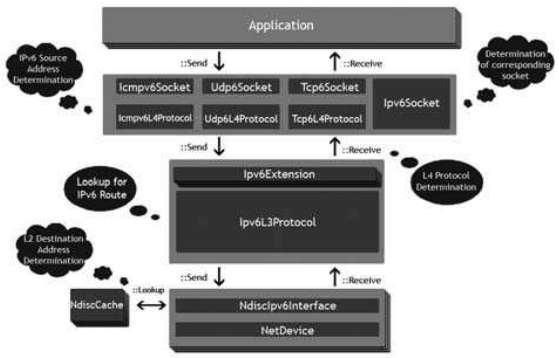


Figure 2: The IPv6 implementation overview

stack. These IPv6 multicast addresses are, for example, `FF02::1` (all link-local IPv6 nodes), `FF02::2` (all link-local IPv6 routers), `FF02::3` (all link-local IPv6 hosts) and the solicited-node multicast IPv6 addresses.

Similarly to IPv4, we also introduce a class called *InternetStackv6* which creates all the necessary objects to build a fully functional IPv6 stack that could be installed in a *Node*.

4.1.1 Internet Control Message Protocol version 6

The Internet Control Message Protocol version 6 (ICMPv6 [1]) is an important part of IPv6 that must be fully supported by every IPv6 stacks. ICMPv6 is a multi-purpose protocol and therefore provides various mechanisms to perform diagnostics (ping), report errors (e.g. destination unreachable) and multicast memberships [15]. For the moment, we support information-related messages (ICMPv6 Echo and Reply messages) and most of the error reporting messages. However, all abnormal behaviors or errors occurred when processing packets do not yet trigger the transmission of the corresponding ICMPv6 message. Note that NS-3 does not provide yet an implementation of the ICMP protocol for IPv4.

Among all features provided by ICMPv6, our first motivation to implement this protocol is to support Neighbor Discovery for which all messages are ICMPv6 messages. We start by implementing the ICMPv6 protocol by subclassing *Ipv6L4Protocol* in a new *Icmpv6L4Protocol* class. This class receives ICMPv6 packets directly from *Ipv6L3Protocol* like any other layer 4 protocol (see Section 4.4). Next, we define the *Icmpv6Header* and *Icmpv6OptionHeader* classes that are respectively the representation of the ICMPv6 header and the ICMPv6 options. Finally, *Icmpv6Header* and *Icmpv6OptionHeader* are subclassed to create various subclasses, each corresponding to a specific Neighbor Discovery message and option.

4.2 Neighbor Discovery

A significant work has been done to support the Neighbor Discovery protocol [6] which is one of the most important new features of IPv6.

4.2.1 Neighbor Solicitation and Neighbor Advertisement

Neighbor Solicitation and Neighbor Advertisement messages are mainly used to resolve the link-layer address associated to an IPv6 address (such as ARP in IPv4) and when a node performs DAD [12] to ensure the uniqueness of an IPv6 address. We have completed their implementation through the *Icmpv6NS* and *Icmpv6NA* classes. All Neighbor Discovery message options are implemented in separate classes in order to be as flexible as possible - certain options are not mandatory, other are shared between different messages, etc. Note that these messages are handled in the *Icmpv6L4Protocol* class. Similarly to ARP, the processing of Neighbor Solicitation and Neighbor Advertisement messages may update a cache, called the neighbor cache in IPv6. This cache stores various parameters such as the correspondence between the IPv6 and the layer 2 addresses. The neighbor cache is implemented by the *NdiscCache* class. An *NdiscCache* entry contains several fields such as an *Ipv6Address*, the status of the entry (*REACHABLE*, *STALE*, *PROBE*, *DELAY* or *INCOMPLETE*), the layer 2 address corresponding to the *Ipv6Address* (if known) and various other timers. The timers ensure that an entry status is always up-to-date [6].

Before sending a packet to the *NetDevice*, the *NdiscIpv6Interface* (*Ipv6Interface* specialization) instance performs a look-up in the *NdiscCache* object to verify whether the link-layer address of an IPv6 destination is known or not. In the latter case, it sends a Neighbor Solicitation message to this IPv6 address. The data packet will be stored in a queue, waiting for the reception of a Neighbor Advertisement resolving the requested link-layer address. If no responses are received after several Neighbor Solicitation retransmission, the destination is considered unreachable and the corresponding *NdiscCache* entry is deleted (as specified in [6]).

4.2.2 Router Solicitation and Router Advertisement

Router Solicitation and Router Advertisement are the messages that transport information on routers and prefixes for an IPv6 link. Access routers generate Router Advertisement in order to advertise their presence on the link and optionally the IPv6 prefix(es) they operate, their link-local address and the maximum recommended MTU for the link. Host may send Router Solicitation to explicitly request a Router Advertisement. These messages are implemented through *Icmpv6RA* and *Icmpv6RS* classes. As already mentioned, each option is implemented in a separate class.

Given the multihoming capabilities of IPv6, many configurations are possible for an IPv6 link as introduced in Section 3.2. An IPv6 link may be served by several access routers, and a single router may advertise more than one prefix. On the other hand, a host may have several network interfaces, and thus may be simultaneously connected to several IPv6 links. Therefore, in our implementation in NS-3, it is important to manage a list of prefixes / addresses per interface, and to keep track of the router that operates a given prefix.

Taking the above observations into account, we propose to store the IPv6 prefixes at the layer 3, in the *Ipv6L3Protocol* class. This class has a list of *RouterPrefix* objects. The *RouterPrefix* object contains an IPv6 prefix (usually received in Router Advertisement messages), a timer to indicate the validity of the prefix and the index of the interface from which the host received the Router Advertisement. The interface index is very important, as a prefix received via a given interface can not be used on any other interface. IPv6 addresses are managed at the interface level.

Digital Object Identifier: 10.4108/ICST.VALUETOOLS2008.4374
http://dx.doi.org/10.4108/ICST.VALUETOOLS2008.4374

On the router side, we design several classes for the Router Advertisement generation. Router Advertisements are generated by an *RadvdApplication* instance, as in the Linux operating system. A class namely *InterfaceConfig* is used to store the information that need to be sent in Router Advertisement. It contains a list of prefixes, the recommended size for the MTU, and the period at which Router Advertisement must be sent. A router operating several IPv6 prefixes will send all prefixes in the same Router Advertisement. Otherwise, we can configure a different period on different interfaces of a router (when a router is operating two distinct IPv6 links for example).

4.3 IPv6 Extensions

We also propose the implementation of the IPv6 extensions for NS-3. These extensions should be very useful to implement new protocols that use specific IPv6 extensions such as Mobile IPv6 [5]. These extensions are processed when receiving an IPv6 packet, before forwarding the packet to upper layers (whenever necessary). For the moment we have only two extensions that could trigger a special behavior in the stack. These are the Fragmentation and the Routing type 0 extensions. We currently support and process both of them. While the Routing type 0 extension is deprecated, we believe that it still could be useful for some case studies. Other extensions as Destination or Hop-by-Hop are also recognized but there are currently no special processing in the stack. Security extensions (Authentication Header and Encapsulating Security Payload) are not implemented yet. Some extensions could be followed by options, thus we include Padding, Jumbogram, Router Alert header options and processing.

4.4 Transport protocols

To provide a UDP implementation for IPv6, we have duplicated the various classes related to UDP in IPv4 and modified them according to IPv6 specificities. Although this may be not optimal, we have based our reflections on the current UDP implementation in the Linux kernel. Therefore, the *Udp6L4Protocol* and *Udp6Socket* classes are very similar to their IPv4 counterpart. When performing our implementation, NS-3 did not support the TCP transport protocol which is why we do not achieve a TCP implementation for IPv6. Now a TCP implementation for IPv4 is available in NS-3, so we plan to develop its IPv6 counterpart.

In order to experiment the behavior of our IPv6 stack, we have developed several IPv6 applications. First, we make an IPv6 port of the existing UDP Echo server and client application of NS-3, called respectively *UdpEcho6Client* and *UdpEcho6Server*. We also developed a *Ping6* application to enable ICMPv6 diagnostics between two hosts. Finally, the last application we made is an NS-3 equivalent of the Router Advertisement Daemon for Linux (radvd). This application is in charge of the transmission of Router Advertisement over the network and reception of Router Solicitation (see section 4.2.2).

4.5 Integration difficulties

The main difficulty of the work was to understand the existing NS-3 code, which consists in more than 500.000 lines of code. It was important to undestand the different features, like how the stack interacts with other elements such as NetDevice or the transport protocol. However this task was ease by the good design and documentation available for NS-3. Because we choose to have a non-intrusive approach to implement our IPv6 stack, there were not so much difficulties regarding the current NS-3 code.

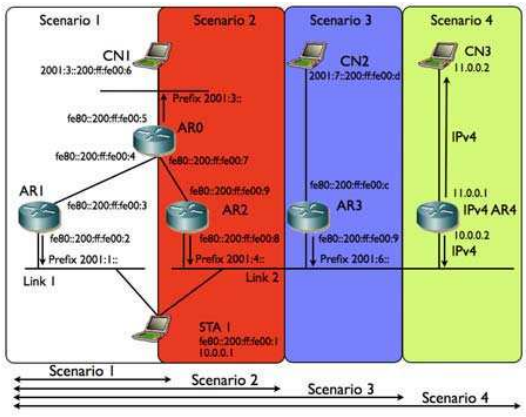


Figure 3: Topology of the first 4 trials

A significant part of the work has been to design and correctly implement the ICMPv6 features, because most parts of the IPv6 logics are implemented there (like Neighbor Discovery Protocol). Then we had some difficulties with multiple addresses management for source address selection. At the beginning on a *Ipv6Interface* object we had just one global IPv6 address and one link-local address. When we were able to have multiple addresses on an interface, we face an issue about source address determination at the socket level. We overcome this by re-designing the routing operations by specifying an extra parameter *prefix* which will be used to determine the correct route for that prefix.

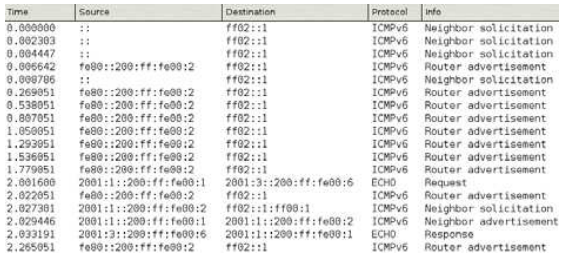
5. FIRST SIMULATIONS

In this section, we give a first set of results that show that our implementation (available at [16]) performs well in basic cases, and allows using IPv6 features in NS-3. The topology used for these simulations is depicted in Figure 3. This figure represents the four incremental scenarios that we are evaluating. Each scenario is made to demonstrate a set of IPv6 features, and each scenario is obtained by adding a new constraint / entity from the previous scenario. The first scenario involves the initiation of a communication between STA1 and CN1 after STA1 has performed IPv6 stateless autoconfiguration over link1. This scenario illustrates the basic features of IPv6. Scenario 2 introduces multihoming, where STA1 connects to an additional link through a second network interface. Scenario 3 extends scenario 2 by adding an access router (AR3) on link2. This illustrates the case of a multihomed link. Finally, Scenario 4 introduces the dual stack IPv4 / IPv6 feature, where STA1 communicates with CN1 and CN2 in IPv6 and with CN3 in IPv4. Note that all access links are Ethernet links.

5.1 Scenario 1: basic operations

The blank part (the first square on left) of Figure 3 represents the topology of this first scenario. We assume an end-host (STA1) which gets connected to an IPv6 link operated by access router AR1. AR1 periodically sends Router

Digital Object Identifier: 10.4108/ICST.VALUETOOLS2008.4374
http://dx.doi.org/10.4108/ICST.VALUETOOLS2008.4374



Time	Source	Destination	Protocol	Info
0.000000	::	ff02::1	ICMPv6	Neighbor solicitation
0.002303	::	ff02::1	ICMPv6	Neighbor solicitation
0.004447	::	ff02::1	ICMPv6	Neighbor solicitation
0.006642	fe80::200:ff:fe00:2	ff02::1	ICMPv6	Router advertisement
0.008786	::	ff02::1	ICMPv6	Neighbor solicitation
0.269051	fe80::200:ff:fe00:2	ff02::1	ICMPv6	Router advertisement
0.538051	fe80::200:ff:fe00:2	ff02::1	ICMPv6	Router advertisement
0.807051	fe80::200:ff:fe00:2	ff02::1	ICMPv6	Router advertisement
1.050051	fe80::200:ff:fe00:2	ff02::1	ICMPv6	Router advertisement
1.293051	fe80::200:ff:fe00:2	ff02::1	ICMPv6	Router advertisement
1.536051	fe80::200:ff:fe00:2	ff02::1	ICMPv6	Router advertisement
1.779051	fe80::200:ff:fe00:2	ff02::1	ICMPv6	Router advertisement
2.001600	2001:1::200:ff:fe00:1	2001:3::200:ff:fe00:6	ECHO	Request
2.022051	fe80::200:ff:fe00:2	ff02::1	ICMPv6	Router advertisement
2.027301	2001:1::200:ff:fe00:2	ff02::1:ff00:1	ICMPv6	Neighbor solicitation
2.029446	2001:1::200:ff:fe00:1	2001:1::200:ff:fe00:2	ICMPv6	Neighbor advertisement
2.033191	2001:3::200:ff:fe00:6	2001:1::200:ff:fe00:1	ECHO	Response
2.265051	fe80::200:ff:fe00:2	ff02::1	ICMPv6	Router advertisement

Figure 4: IPv6 stateless autoconfiguration (Scenario 1)

Advertisements with prefix `2001:1::/64` between 200 and $600ms$. AR1 is connected to another router AR0, which operates another link where is located the correspondent of STA1 called CN1. After the stateless autoconfiguration, STA1 initiates a communication with CN1.

Figure 4 is the tcpdump capture of link1 where STA1 is attached to. On this trace, we can see the different steps of the IPv6 stateless autoconfiguration: first STA1 sends a Neighbor Solicitation in order to perform DAD on its IPv6 link-local address, AR1 does also DAD on its link-local and global addresses. Then STA1 receives a Router Advertisement from AR1. This Router Advertisement is represented in Figure 5 where the prefix option is set to `2001:1::/64`. Upon the reception, STA1 performs DAD on its global address (created from the prefix `2001:1::/64`) by sending another Neighbor Solicitation. Around $2s$ of the simulated time, STA1 initiates a UDP flow (by using the UDP Echo server and client application) with CN1 (destination `2001:3::200:ff:fe00:6`). STA1 sends this data packet to its gateway (i.e. AR1) without requesting the link-layer address resolution, as the Router Advertisement messages sent by AR1 contain the link-layer option filled with the link-layer address of AR1. Then we see that CN1 responds, but AR1 first needs to resolve the link-layer address of STA1 before forwarding the packet to STA1. Thus AR1 sends a Neighbor Solicitation to which STA1 responds with a Neighbor Advertisement. Finally the data packet is forwarded to STA1.

This first scenario shows that the basic features of IPv6 work, especially Neighbor Discovery. Routers can be configured to send Router Advertisements, and nodes can perform IPv6 stateless autoconfiguration, DAD, and finally can resolve the link-layer address associated to an IPv6 address.

5.2 Scenario 2: multi-interfaced node

The second scenario is represented by the white and red parts (the two squares on the left) of Figure 3. We assume the same configuration as in Scenario 1, but we consider a second interface on STA1 which is connected to a new link operated by AR2. AR2 is sending Router Advertisements between 200 and $600ms$ with the IPv6 prefix `2001:4::/64`. In this scenario, we show one of the main features of multihoming, which is the failure recovery. We assume that at the simulated time $t = 5s$, AR1 will be shutdown (simulating a failure). As a consequence, STA1 will no longer receive Router Advertisements on the link1 and will change its default route to use AR2 on link2.

The routing table of STA1 is illustrated in Table 1 at three

```
Internet Protocol Version 6
Internet Control Message Protocol v6
  Type: 134 (Router advertisement)
  Code: 0
  Checksum: 0x0c09 [correct]
  Cur hop limit: 0
▷ Flags: 0x00
  Router lifetime: 1
  Reachable time: 1
  Retrans timer: 1
▽ ICMPv6 Option (MTU)
    Type: MTU (5)
    Length: 8
    MTU: 1280
▽ ICMPv6 Option (Prefix information)
    Type: Prefix information (3)
    Length: 32
    Prefix length: 64
  ▽ Flags: 0xe0
      1... .... = Onlink
      .1.. .... = Auto
      ..1. .... = Router Address
      ...0 .... = Not site prefix
    Valid lifetime: 5
    Preferred lifetime: 3
    Prefix: 2001:1::
▽ ICMPv6 Option (Source link-layer address)
    Type: Source link-layer address (1)
    Length: 8
    Link-layer address: 00:00:00:00:00:02
```

Figure 5: Router Advertisements sent by AR1 (Scenario 1)

different periods of time. The table at the top illustrates the entries when STA1 only receives Router Advertisements from AR1. The table in the middle illustrates the entries when STA1 receives Router Advertisements from both AR1 and AR2. We see that AR1 remains the default router for STA1, but a new entry has been added for destinations that are on link2. Finally the last table illustrates the entries when the entry for AR1 has expired (resulting from a lack of new Router Advertisement due to the shutdown of AR1). When the prefix advertised by AR1 expires, AR2 becomes the default router of STA1. So the UDP Echo Request messages are sent over the second interface (i2) and reach CN1 while AR1 is not available. We can note that in all the three routing tables, the entries for the link local destinations are always available (*fe80::*) because these do not depend on the prefixes that are advertised.

In conclusion, we can say that basic features of multihoming are already working in our implementation. When a host as two links and one of them fails, it automatically modifies its routing table to prefer the available link.

5.3 Scenario 3: multihoming on a single link

The third scenario is represented by the white, red and blue parts (the three squares on the left) of Figure 3. In addition to the previous configuration, we consider an additional router (AR3) on link2 which advertises a new prefix

Digital Object Identifier: 10.4108/ICST.VALUETOOLS2008.4374
http://dx.doi.org/10.4108/ICST.VALUETOOLS2008.4374

Table 1: STA1 Routing tables in Scenario 2

Routing table before a RA is received on i2		
Destination	Next Hop	Interface
::	fe80::0200:00ff:fe00:0002	1
::1	::	0
fe80::	::	1
fe80::	::	2
2001:1::	::	1

Routing table after a RA is received on i2		
Destination	Next Hop	Interface
::	fe80::0200:00ff:fe00:0002	1
::1	::	0
fe80::	::	1
fe80::	::	2
2001:1::	::	1
2001:4::	::	2

Routing table after AR1 has expired		
Destination	Next Hop	Interface
::	fe80::0200:00ff:fe00:0009	2
::1	::	0
fe80::	::	1
fe80::	::	2
2001:4::	::	2

`2001:6::/64`. Therefore, STA1 ends up with three global IPv6 addresses: one on the link1 with the prefix `2001:1::/64`, two on the link2 with prefixes `2001:4::/64` (from AR2) and `2001:6::/64` (from AR3). AR3 is connected to another end-host (CN2) to which STA1 initiates a new communication.

We assume that AR2 is the default router for STA1. Therefore, when STA1 initiates a communication with CN2, it uses AR2 as next hop for the UDP Echo Request as illustrated in Figure 6. However, AR2 does not have any route to reach CN2, and thus the packet loops between AR2 and AR0 (each of them using their default route). After the Time To Live field of the UDP Echo Request reaches 0, the packet is discarded and an ICMPv6 Error message is sent to STA1 entitled Time Exceeded. After this, we assume that the user configures a new static route for the prefix of CN2 (i.e. `2001:7::`) through AR3. Once this route is installed on STA1, packets are reaching CN2.

In this scenario we illustrate that two routers can be used on the same link. However, we still miss an automatic behavior which would automatically redirect unsuccessful packets from one router to another. An ICMPv6 redirect message should be used in this case. We are currently working on this issue.

5.4 Scenario 4: Dual stack configuration

In the last scenario, we consider the whole topology of Figure 3. We consider the same scenario as in the previous subsection, but we add an extra correspondent CN3 and an extra router AR4 which only implement IPv4. STA1 becomes a dual stack node with both IPv4 and IPv6 stacks.

Figure 7 represents a tcpdump capture of the messages exchanged over all interfaces of STA1. We can see that both IPv4 and IPv6 packets are exchanged well. This is a very interesting feature since dual mode nodes will certainly be studied in the next years in order to evaluate transition mechanisms that are needed in the real world. In addition,

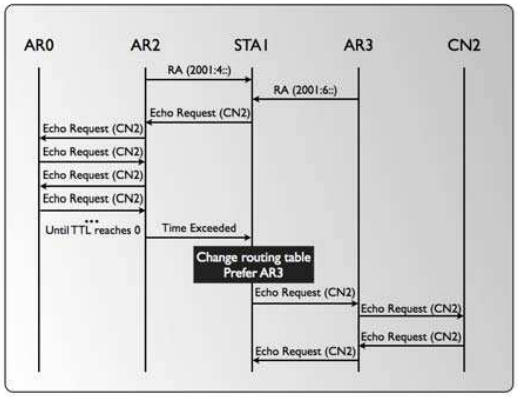


Figure 6: Message flows for Scenario 3

Time	Source	Destination	Protocol	Info
4.001600	2001:6::200:ff:fe00:7	2001:7::200:ff:fe00:e	ECHO	Request
4.012833	2001:7::200:ff:fe00:e	2001:6::200:ff:fe00:7	ECHO	Response
4.439051	fe80::200:ff:fe00:8	ff02::1	ICMPv6	Router advertisement
4.494051	fe80::200:ff:fe00:9	ff02::1	ICMPv6	Router advertisement
5.241051	fe80::200:ff:fe00:8	ff02::1	ICMPv6	Router advertisement
5.296051	fe80::200:ff:fe00:9	ff02::1	ICMPv6	Router advertisement
5.999929	00:00:00_00:00:07	Broadcast	ARP	Who has 10.0.0.2? Tell 10.0.0.1
6.002004	00:00:00_00:00:0a	00:00:00_00:00:07	ARP	10.0.0.2 is at 00:00:00:00:00:0a
6.005717	10.0.0.1	11.0.0.2	ECHO	Request
6.023514	00:00:00_00:00:0a	Broadcast	ARP	Who has 10.0.0.1? Tell 10.0.0.2
6.025588	00:00:00_00:00:07	00:00:00_00:00:0a	ARP	10.0.0.1 is at 00:00:00:00:00:07
6.029301	11.0.0.2	10.0.0.1	ECHO	Response
6.043051	fe80::200:ff:fe00:8	ff02::1	ICMPv6	Router advertisement
6.288051	fe80::200:ff:fe00:9	ff02::1	ICMPv6	Router advertisement
6.955051	fe80::200:ff:fe00:8	ff02::1	ICMPv6	Router advertisement
7.001568	10.0.0.1	11.0.0.2	ECHO	Request
7.012705	11.0.0.2	10.0.0.1	ECHO	Response

Figure 7: Scenario 4: IPv4 and IPv6 packets from and to STA1

this indicates that our implementation of an IPv6 stack in NS-3 does not break the IPv4 support.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented our implementation of an IPv6 stack in the new version of the Network Simulator namely NS-3. The project NS-3 started in 2006 and focused on the core of the simulator. Today, it already proposes an Internet framework with a light version of a UDP / TCP / IP stack. The NS-3 project is very ambitious, as it aims at becoming the reference in simulation of the Internet model. The true novelty in this new version is the interaction with real systems. The OSI model implementation of NS-3 is close to what a current operating system implements (e.g. Unix systems) which allows performing experimentation by interconnecting a real testbed with the simulator. For example, the same socket interface is implemented in NS-3 and FreeBSD.

Up to now, NS-3 only focuses on the fourth version of the Internet Protocol. However, the new version of IP, namely IPv6, is currently being deployed in the Internet, and should be the next IP protocol used for the next fifty years. IPv6 offers lot of advantages over IPv4 as it benefits from the experience of more than thirty years of the Internet usage. One of them is certainly the Neighbor Discovery protocol. Inspired from the ARP protocol for IPv4, it enables router discovery, host autoconfiguration, neighbor discovery, address resolution, and next hop determination. Moreover, IPv6 proposes a simplified header system which eases the development of extensions. IPv6 also offers several proto-

Digital Object Identifier: 10.4108/ICST.VALUETOOLS2008.4374
http://dx.doi.org/10.4108/ICST.VALUETOOLS2008.4374

cols for multihoming and mobility support. This support allows optimizing communications of mobile devices such as PDA or Personal Area Network connected to the Internet.

Therefore, it is important to include the support of IPv6 within NS-3 as we propose in this paper. Our implementation follows the same philosophy as the development of IPv4 in NS-3. Thus we design an IPv6 socket API and layers 3 and 4 for IPv6. In addition, we develop ICMPv6 in order to generate the Neighbor Discovery messages (i.e. Neighbor Advertisement, Neighbor Solicitation, Router Advertisement and Router Solicitation). The implementation of radvd allows routers to advertise IPv6 prefix(es) on the link(s) they operate similarly to GNU/Linux operating systems. We include in this development the multihoming support, where a host may be simultaneously connected to several links, and where several IPv6 prefixes may be sent over a single IPv6 link. In Section 5 we illustrate various scenarios in order to show the features that have been implemented.

The implementation presented in this paper does not pretend to be complete and is still an ongoing work. However, we believe that the main components of an IPv6 stack have been implemented, and can serve for further development. We expect to propose this implementation to be included in the core of NS-3.

Our future work is to propose an implementation of other IPv6 protocols, such as Mobile IPv6 [5] or SHIM6 [7]. Currently we are focusing on a complete support of ICMPv6 in addition to the integration of TCP over IPv6.

7. ACKNOWLEDGMENT

The authors would like to thank David Gross from LSIIT and Mehdi Benamor from Telecom Bretagne for their great help throughout the implementation process. The achievement of this work would not be possible without their invaluable assistance.

8. REFERENCES

[1] A. Conta, S. Deering, and M. Gupta. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification, Internet Engineering Task Force Request for Comments (RFC) 4443, March 2006.

[2] S. Deering and R. H. Postel. Internet Protocol, Version 6 (IPv6) Specification, Internet Engineering Task Force Request for Comments (RFC) 2460, December 1998.

[3] R. Droms, J. Bound, B. Volz, T. Lemon, C. Perkins, and M. Carney. Dynamic Host Configuration Protocol for IPv6 (DHCPv6), Internet Engineering Task Force Request for Comments (RFC) 3315, July 2003.

[4] R. Hinden and B. Haberman. Unique Local IPv6 Unicast Addresses, Internet Engineering Task Force Request for Comments (RFC) 4193, October 2005.

[5] D. Johnson, C. Perkins, and J. Arkko. Mobility Support in IPv6, Internet Engineering Task Force Request for Comments (RFC) 3775, June 2006.

[6] T. Narten, E. Nordmark, W. Simpson, and H. Soliman. Neighbor Discovery for IP version 6 (IPv6), Internet Engineering Task Force Request for Comments (RFC) 4861, September 2007.

[7] E. Nordmark and M. Bagnulo. Shim6: Level 3 Multihoming Shim Protocol for IPv6, Work in Progress, Internet Engineering Task Force draft-ietf-shim6-proto-10.txt, February 2008.

[8] The Network Simulator - ns-2 Home Page, http://www.isi.edu/nsnam/ns/.

[9] ns-3 Project Home Page, http://www.nsnam.org/.

[10] Opnet Technologies, Making Network and Applications perform, http://www.opnet.com/.

[11] J. Postel. Internet Protocol, Internet Engineering Task Force Request for Comments (RFC) 791, September 1981.

[12] S. Thomson, T. Narten, and T. Jinmei. IPv6 Stateless Address Autoconfiguration, Internet Engineering Task Force Request for Comments (RFC) 4862, September 2007.

[13] A. Vagra. OMNeT++ Home Page, http://www.omnetpp.org, 1997.

[14] A. Vagra. The OMNeT++ discrete event simulation system. *European Simulation Multiconference (ESM 2001)*, June 2001.

[15] R. Vida and L. Costa. Multicast Listener Discovery Version 2 (MLDv2) for IPv6, Internet Engineering Task Force Request for Comments (RFC) 3810, June 2004.

[16] S. Vincent, M. Benamor, J. Montavont, and N. Montavont. IPv6 for NS-3, https://ns3v6.enstb.fr, 2008.

Digital Object Identifier: 10.4108/ICST.VALUETOOLS2008.4374
http://dx.doi.org/10.4108/ICST.VALUETOOLS2008.4374