Introducing IPv6 Tokenised Interface Identifiers into the Linux Kernel

Mark K. Thompson
mkt@ecs.soton.ac.uk
School of Electronics and Computer Science
University of Southampton, UK
ECS Technical Report, number ECSTR-IAM05-006

5th July, 2005

Abstract

IPv6 Stateless Address Auto-configuration (SLAAC) enables network administrators to deploy devices in a network and have those devices automatically generate global addresses without any administrative intervention, and without the need for any stateful configuration service such as DHCPv6.

However, certain services — such as HTTP, SMTP and IMAP — may better benefit from having "well known" identifiers that do not depend on the physical hardware address of the server's network interface card.

Tokenised addresses offer a facility for administrators to specify the bottom 64 bits of an IPv6 address for a node whilst allowing the top 64 bits (the network prefix) to be automatically configured from router advertisements.

This report documents the approach taken and experience gained from introducing tokenised interface identifiers into the Linux 2.6.11 kernel, as shipped with Redhat Fedora Core 4.

This proof of concept work demonstrates the relative ease of introducing this useful utility for network node deployment, and further motivates wider deployment of the semi-automatic configuration approach.

1 Introduction

This report details changes to the Linux 2.6.11 kernel made to enable tokenised IPv6 interface identifiers as an alternative to EUI64-based interface identifiers when a node partakes in stateless address auto-configuration (SLAAC).

The intended use of tokenised identifiers is in server systems whose IIDs need to be well-known, but where stateful configuration services, e.g. DHCPv6, are not deployed.

The development machine used as a reference point is a Redhat Fedora Core 4 linux box, running a stock kernel version 2.6.11-1.1369_FC4. The driving principle of this development is that it should be a simple, clean patch to the kernel that sites can roll-out without impeding normal network use.

The default node behaviour regarding IPv6 address auto-configuration should be unaffected with the tokenised identifier option being a post-installation configuration option on a per-node basis.

This report details the build process, the changes to the stock kernel made to realise tokenised identifiers, and an exemplar user-space tool that interfaces with the kernel to query and configure interface tokens.

Terminology

FC4 Redhat linux, Fedora Core distribution release 4[1]

DHCPv6 Dynamic Host Configuration Protocol for IPv6 [3])

IID Interface Identifier, per s 2.5.1 of [4], these are 64-bit identifiers used to distinguish nodes in a subnet

EUI-64 64-bit global identifier used in stateless address auto-configuration of nodes as an IID [1]

SLAAC Stateless Address Auto-configuration [2]

2 Strategy

Having prepared the development box for work (below), a cursory analysis of the `net/ipv6` code suggests that an appropriate strategy for enabling tokenised identifiers would be to offer an `ioctl` interface for specification and retrieval of configured interface token on a per-interface basis.

The generation of new identifiers for an interface is triggered by the receipt of a router advertisement with a prefix information field that has the 'A' (Autonomous configuration) bit set. We augment the address auto-configuration code (`addrconf.c`) such that, if a token is specified for the interface, that token is used for the IID rather than an EUI64-based identifier.

The approach we propose differs from that adopted by Sun Microsystems in recent versions (2.9+) of their Solaris operating system. Where they permit flexible prefix lengths for tokens (also configured per-interface by `ioctls`), we follow the mandate of RFC3513 and only consider 64-bit IIDs.

3 Development steps

The following steps were taken to get the stock install to a state in which development could be undertaken.

3.1 Prepare development environment

The FC4 install was a workstation install with the developer tools selected. (Principally, `gcc-4.0` and kernel development headers). Since Redhat have stopped shipping pre-patched kernel source trees, the following steps were

[1] `http://www.redhat.com/fedora/`

necessary to acquire kernel sources that mirror that used to build the stock kernel:

1. Install kernel development package
   `rpm -Uvh kernel-devel-2.6.11-1.1369_FC4.i686.rpm`
2. Acquire kernel source tree with Fedora patchset
   `up2date --get-source kernel`
3. Install Fedora-packaged kernel sources and patches
   `rpm -Uvh kernel-2.6.11-1.1369_FC4.src.rpm`
4. Build the source tree
   `cd /usr/src/redhat/SPECS; rpm -bp --target i686 kernel-2.6.spec`
5. Move the source tree to a more familiar and useful place
   `mv /usr/src/redhat/BUILD/linux-2.6.11 /usr/src; cd !$`
   `ln -s ./linux-2.6.11 linux; cd /usr/src/linux`
6. Prep the kernel build with values used to build live kernel
   `make mrproper; cp configs/kernel-2.6.11-i686-prep-v6dev.config .config; make oldconfig`
7. Tag the kernel as custom
   `echo '-v6dev' > localversion-v6dev`
8. Build the kernel to speed-up later builds
   `make; make install; make modules_install`
9. Confirm `grub.conf` boot target
   `vim /boot/grub/grub.conf` and check our new kernel is the boot target

At this point, the kernel source tree reflects the live kernel, is tagged as a customised kernel, and is in a state where network code can be extended. Note that the `mrproper` target differs from FC4 release notes. This is due to the symbol mis-match errors that otherwise ensue when the kernel will try to load modified code in the `net/ipv6` module following installation.

3.1.1 Caveat

On two successive builds, the `initrd` image (the RAM disk that is loaded by the boot loader before the kernel is started) was missing sufficient code to load the LVM2-based root partition, resulting in a kernel 'Oops' message and a hung machine.

Building the ramdisk image after having installed the kernel and modules in the build process seemed to have fixed the problem, although it is not evident from examining the kernel Makefile and install scripts why the build intermittently fails.

The command to build and install the image is: `/sbin/mkinitrd -v -f /boot/initrd-2.6.11-prep-v6dev.img 2.6.11-prep-v6dev`

4 Code changes

The following changes are to support the linux kernel build process and do not provide functionality for tokenised interface identifiers.

Configuration directives

`configs/kernel-2.6.1-i686-prep-v6dev.config` This file is a copy of the stock `configs/kernel-2.6.1-i686.config`, but with a directive added so that tokenised address code is included in a default build of this source tree.

```
--- linux-2.6.11/configs/kernel-2.6.11-i686.config
+++ linux-2.6.11-prep-v6dev/configs/kernel-2.6.11-i686-prep-v6dev.config
@@ -850,6 +849,7 @@
 #
 CONFIG_IP_VS_FTP=m
 CONFIG_IPV6=m
+CONFIG_IPV6_IID_TOKENS=y
 CONFIG_IPV6_PRIVACY=y
 CONFIG_INET6_AH=m
 CONFIG_INET6_ESP=m
```

`net/ipv6/Kconfig` This file is consulted during the kernel build configuration process. The `IPV6_IID_TOKENS` option needs to be enabled to include tokenised IID support in the kernel.

```
--- linux-2.6.11/net/ipv6/Kconfig
+++ linux-2.6.11-prep-v6dev/net/ipv6/Kconfig
@@ -1,6 +1,15 @@
 #
 # IPv6 configuration
 #
+config IPV6_IID_TOKENS
+ bool "IPv6: Tokenised interface identifier support"
+ depends on IPV6
+ ---help---
+   Tokenised IIDs enable administrators to assign well-known
+   host-part addresses to nodes whilst still obtaining global
+   network prefix from Router Advertisements.
+   Configured through SIOCSIFTOKEN ioctls, an example token
+   on an interface might be ::53 for well-known nameservers.
+
 config IPV6_PRIVACY
  bool "IPv6: Privacy Extensions (RFC 3041) support"
  depends on IPV6
```

Header files

Two header files require an update for this modification. One to define the new `ioctl` interface from user-land, and an extension to the network interface configuration structure to maintain a per-interface token.

`include/linux/sockios.h` Whereas the Linux kernel documentation suggests that new tokens should be declared using the macros provided, the existing set of controls relevant to network configuration do not, and so our two new tokens do not.

We add two controls, one to query the current interface token for a device, and one to set.

`SIOCGIFTOKEN` `ioctl` for querying the current IID token set on an interface. From user-space, a pointer to a partially grounded instance of a `struct if6_addr` is passed to the kernel with `ifindex` set to the device index of the interface being queried. `ioctl SIOCGIFINDEX` should be used to determine the value of `ifindex`.

`SIOCSIFTOKEN` Similarly, the interface token is set by passing a pointer to fully-grounded instance of a `struct if6_addr` structure. `prefixlen` *must* be set to 64 to reflect the fact that the token is 64-bits long, specifically the bottom 64-bits of the address. The `prefix` field (essentially mislabelled for our purpose) contains the full 128-bit address, where the top 64-bits are expected[2] to be zero.

```
--- linux-2.6.11/include/linux/sockios.h
+++ linux-2.6.11-prep-v6dev/include/linux/sockios.h
@@ -82,6 +82,10 @@
 #define SIOCSMIIREG 0x8949 /* Write MII PHY register. */

 #define SIOCWANDEV 0x894A /* get/set netdev parameters */
+#ifdef CONFIG_IPV6_IID_TOKENS
+#define SIOCGIFTOKEN 0x894B /* get IID token in use         */
+#define SIOCSIFTOKEN 0x894C /* set IID token in use         */
+#endif /* def CONFIG_IPV6_IID_TOKENS */

 /* ARP cache control calls. */
      /*  0x8950 - 0x8952  * obsolete calls, don't re-use */
```

`include/net/if_net6.h` We add a token field to the `inet6_dev` structure. Note that this is a full 128-bit structure as per any other IPv6 address. Only the bottom 64 bits are consulted when configuring tokens against interfaces, however.

```
--- linux-2.6.11/include/net/if_inet6.h
+++ linux-2.6.11-prep-v6dev/include/net/if_inet6.h
@@ -185,11 +184,16 @@
  __u8 work_digest[16];
 #endif

+#ifdef CONFIG_IPV6_IID_TOKENS
+ struct in6_addr token;
+#endif /* CONFIG_IPV6_IID_TOKENS */
+
  struct neigh_parms *nd_parms;
```

[2] 'Expected', but not verified in the kernel. The bottom 64-bits are copied into the device's information structure

```
  struct inet6_dev *next;
  struct ipv6_devconf cnf;
  struct ipv6_devstat stats;
  unsigned long tstamp; /* ipv6InterfaceTable update timestamp */
+
 };

 extern struct ipv6_devconf ipv6_devconf;
```

IPv6 module changes

Two components of the IPv6 module are updated to achieve tokenised interface identifiers.

`net/ipv6/af_inet6.c` A small change to `af_inet6.c` is required to hook in the new user-land `ioctl` directives.

```
--- linux-2.6.11/net/ipv6/af_inet6.c
+++ linux-2.6.11-prep-v6dev/net/ipv6/af_inet6.c
@@ -455,6 +455,12 @@
  return addrconf_del_ifaddr((void __user *) arg);
  case SIOCSIFDSTADDR:
  return addrconf_set_dstaddr((void __user *) arg);
+#ifdef CONFIG_IPV6_IID_TOKENS
+ case SIOCGIFTOKEN:
+ return addrconf_get_iftoken((void __user *) arg);
+ case SIOCSIFTOKEN:
+ return addrconf_set_iftoken((void __user *) arg);
+#endif /* CONFIG_IPv6_IID_TOKENS */
  default:
  if (!sk->sk_prot->ioctl ||
      (err = sk->sk_prot->ioctl(sk, cmd, arg)) == -ENOIOCTLCMD)
```

`net/ipv6/addrconf.c` The bulk of the implementation task is in the address configuration component, `addrconf.c`.

In `ipv6_add_dev()`, invoked when a new IPv6-capable device is configured, we initialise the token to zero, suggesting that EUI-64 IIDs should be used where the interface is subject to SLAAC.

```
--- linux-2.6.11/net/ipv6/addrconf.c
+++ linux-2.6.11-prep-v6dev/net/ipv6/addrconf.c
@@ -396,6 +396,12 @@
        NULL);
  addrconf_sysctl_register(ndev, &ndev->cnf);
 #endif
+
+#ifdef CONFIG_IPV6_IID_TOKENS
+ /* 0 => no token, use EUI64 */
+ memset(ndev->token.s6_addr, 0, 16);
+#endif /* CONFIG_IPV6_IID_TOKENS */
  }
  return ndev;
 }
```

When an interface receives a router advertisement with a prefix information option set, `addrconf_prefix_rcv()` is invoked. Where the interface's token parameter is zero (i.e. `ipv6_addr_any()` holds true), we follow the original code and use EUI-64 addresses. However, when a token is specified, we generate an address concatenating the prefix from the received advertisement with the token, which is then subject to the usual hashing and duplicate address detection.

```
--- linux-2.6.11/net/ipv6/addrconf.c
+++ linux-2.6.11-prep-v6dev/net/ipv6/addrconf.c
@@ -1489,11 +1495,30 @@

  if (pinfo->prefix_len == 64) {
  memcpy(&addr, &pinfo->prefix, 8);
-if (ipv6_generate_eui64(addr.s6_addr + 8, dev) &&
-    ipv6_inherit_eui64(addr.s6_addr + 8, in6_dev)) {
-in6_dev_put(in6_dev);
-return;
+#ifdef CONFIG_IPV6_IID_TOKENS
+ /* check if in6_dev->token is not wildcard */
+ if(!ipv6_addr_any(&in6_dev->token)) {
+ ADBG((KERN_DEBUG
+       "%s - will use token %04x:%04x:%04x:%04x:%04x:%04x:%04x:%04x on %s\n",
+       __FUNCTION__, NIP6(in6_dev->token), dev->name));
+ /* Strictly /64, so grab the bottom 8 octets from token */
+ memcpy(addr.s6_addr+8, in6_dev->token.s6_addr+8, 8);
+ ADBG((KERN_DEBUG
+       "%s - which makes %04x:%04x:%04x:%04x:%04x:%04x:%04x:%04x\n",
+       __FUNCTION__, NIP6(addr)));
+ } else {
+ ADBG((KERN_DEBUG
+       "%s - not using tokens. Using EUI64 on %s\n",
+       __FUNCTION__, dev->name));
+#endif /* CONFIG_IPV6_IID_TOKENS */
+ if (ipv6_generate_eui64(addr.s6_addr + 8, dev) &&
+     ipv6_inherit_eui64(addr.s6_addr + 8, in6_dev)) {
+ in6_dev_put(in6_dev);
+ return;
+ }
+#ifdef CONFIG_IPV6_IID_TOKENS
  }
+#endif /* CONFIG_IPV6_IID_TOKENS */
  goto ok;
  }
  if (net_ratelimit())
```

The following four functions implement the kernel–user-land interface. The functions `inet6_token_get()` and `inet6_token_set()` manipulate the interface device structure in the kernel, and so require that the network device be locked so that no other kernel task can modify its contents whilst we work on it.

`addrconf_get_iftoken()` and `addrconf_set_iftoken()` are invoked by the `ioctl` handler in `af_inet6.c`, and are wrappers around the device manipulation code.

We immediately deprecate all non-linklocal address on that interface and solicit for an unscheduled router advertisement when the kernel receives a request from userland to update the token set on an interface. This is in part because we only accept one token per interface, and also because the tokenised identifier is deemed to be the canonical identifier for the node.

This is 'impure' from a functionality perspective in the sense that we do not set RS_SENT interface flag, nor do we obey local solicitation timers. However, a token set can be deemed orthogonal to typical operation and the router advertisement receipt code does correctly handle advertisements that it did not 'knowingly' solicit.

The decision to deprecate but not invalidate all other addresses forces the kernel to prefer the new tokenised address as the source address for future communication without disrupting existing connections.

```
--- linux-2.6.11/net/ipv6/addrconf.c
+++ linux-2.6.11-prep-v6dev/net/ipv6/addrconf.c
@@ -1739,6 +1766,142 @@
 }


+#ifdef CONFIG_IPV6_IID_TOKENS
+static int inet6_token_get(int ifindex, struct in6_addr *tok)
+{
+ struct inet6_dev *idev;
+ struct net_device *dev;
+
+ ASSERT_RTNL();
+
+ if ((dev = __dev_get_by_index(ifindex)) == NULL)
+ return -ENODEV;
+
+ if (!(dev->flags&IFF_UP))
+ return -ENETDOWN;
+
+ if ((idev = __in6_dev_get(dev)) == NULL)
+ return -ENXIO;
+
+ ADBG((KERN_DEBUG
+       "%s - token currently %04x:%04x:%04x:%04x:%04x:%04x:%04x:%04x on %d\n",
+       __FUNCTION__, NIP6(idev->token), ifindex));
+
+ memcpy(&(tok->s6_addr), &(idev->token.s6_addr), 16);
+ return 0;
+}
+
+static int inet6_token_set(int ifindex, struct in6_addr *tok)
+{
+ struct inet6_dev *idev;
+ struct net_device *dev;
+ struct inet6_ifaddr *ia;
+
+ ASSERT_RTNL();
+
```

```
+ if ((dev = __dev_get_by_index(ifindex)) == NULL)
+ return -ENODEV;
+
+ if (!(dev->flags&IFF_UP))
+ return -ENETDOWN;
+
+ if ((idev = __in6_dev_get(dev)) == NULL)
+ return -ENXIO;
+
+ ADBG((KERN_DEBUG
+       "%s - token out %04x:%04x:%04x:%04x:%04x:%04x:%04x:%04x;"
+   "in %04x:%04x:%04x:%04x:%04x:%04x:%04x:%04x\n",
+       __FUNCTION__,
+       NIP6(idev->token),
+       NIP6(*tok)));
+
+ memcpy(&(idev->token.s6_addr), &(tok->s6_addr), 16);
+
+ /* This is probably naughty - not considering rtsol timers or i/f RS_SENT flag
+  * At the same time, though, this isn't a solicit in response to a new interface
+  * but a solicit orthogonal to normal operation (i.e. please bebejee don't send
+  * me down to that dark, hot, firey place for what i'm about to do)
+  */
+        if (idev->cnf.forwarding == 0 && // mustn't be a router
+            idev->cnf.rtr_solicits > 0 && // must do router solicits
+     idev->cnf.autoconf > 0 && // must be set to perform SLAAC
+            (dev->flags&IFF_LOOPBACK) == 0) { // must not be loop-back
+                struct in6_addr all_routers;
+ struct in6_addr link_local;
+
+ ipv6_get_lladdr(dev, &link_local);
+                ipv6_addr_all_routers(&all_routers);
+ /* rtr sol src addr should be linklocal */
+                ndisc_send_rs(dev, &link_local, &all_routers);
+
+ /* Similarly evil, deprecate all other globals on this i/f */
+ for(ia=idev->addr_list;ia;ia=ia->if_next) {
+ if(!(ipv6_addr_type(&ia->addr) & IPV6_ADDR_LINKLOCAL)) {
+ ia->prefered_lft = 0; // ho ho i'm going to hell
+ }
+ }
+        }
+
+ return 0;
+}
+
+/* set current token used for IIDs - arg is in6_ifreq */
+int addrconf_set_iftoken(void __user *arg)
+{
+ struct in6_ifreq ireq;
+ int err;
+
+ ADBG((KERN_DEBUG
+       "%s - invoked\n",
```

```
+       __FUNCTION__));
+ if (!capable(CAP_NET_ADMIN))
+ return -EPERM;
+
+ if (copy_from_user(&ireq, arg, sizeof(struct in6_ifreq)))
+ return -EFAULT;
+
+ if (ireq.ifr6_prefixlen != 64)
+ return -EINVAL;
+
+ ADBG((KERN_DEBUG
+       "%s - ... saw set request for "
+   "%04x:%04x:%04x:%04x:%04x:%04x:%04x:%04x/%d, on index %d\n",
+       __FUNCTION__, NIP6(ireq.ifr6_addr),
+   ireq.ifr6_prefixlen, ireq.ifr6_ifindex));
+
+ rtnl_lock();
+ err = inet6_token_set(ireq.ifr6_ifindex, &ireq.ifr6_addr);
+ rtnl_unlock();
+ return err;
+}
+
+/* get current token used for IIDs - arg is
+ * in6_ifreq to be matched against where ifindex is looked at */
+int addrconf_get_iftoken(void __user *arg)
+{
+ struct in6_ifreq ireq;
+ int err;
+
+ ADBG((KERN_DEBUG
+       "%s - invoked\n",
+       __FUNCTION__));
+ if (!capable(CAP_NET_ADMIN))
+ return -EPERM;
+
+ if (copy_from_user(&ireq, arg, sizeof(struct in6_ifreq)))
+ return -EFAULT;
+
+ rtnl_lock();
+ err = inet6_token_get(ireq.ifr6_ifindex, &ireq.ifr6_addr);
+ rtnl_unlock();
+
+ if(err)
+ return err;
+
+ /* now, ireq.ifr6_addr should be the token value
+    * which we now need to get back into userland
+    */
+ if (copy_to_user(arg, &ireq, sizeof(struct in6_ifreq)))
+ return -EFAULT;
+
+ return 0;
+}
+#endif /* CONFIG_IPV6_IID_TOKENS */
```

```
+
 int addrconf_add_ifaddr(void __user *arg)
 {
  struct in6_ifreq ireq;
```

5 `ip6token` - Exemplar userland tool

`ip6token` is an example tool that manipulates interface tokens through the `ioctl` directives defined in the kernel patch above.

When invoked without any parameters, it iterates through the available IPv6 devices in the node, querying each for their current token status.

Invoked with one parameter, an interface name, the code queries the current token for that interface, e.g. `ip6token eth0`.

When invoked with two arguments, interface name and token literal, the tool attempts to set the token for that interface, e.g. `ip6token eth0 ::53` for a node acting as a name-server on a subnet.

```
/*
 * IPv6 Address [auto]configuration
 *  Tokenised Interface Identifier Utility
 *
 *  Developed as a PoC for the Cisco/6NET project:
 *  Operational Experience with IPv6 Network Renumbering
 *
 * Authors:
 * Mark Thompson <mkt@ecs.soton.ac.uk>
 *
 * This program is free software; you can redistribute it and/or
 *  modify it under the terms of the GNU General Public License
 *  as published by the Free Software Foundation; either version
 *  2 of the License, or (at your option) any later version.
 */

// vim: expandtab sw=4 ts=4 sts=4:
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <net/if.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <linux/sockios.h>
#include <sys/types.h>
#include <netdb.h>
#include <errno.h>

/* libc5 doesn't provide POSIX types */
#include <asm/types.h>

#ifndef ASSERT
#define ASSERT(X)                                \
do {                                             \
```

```
    if (!(X)) {                                 \
        fprintf(stderr, "ASSERT %s: %s (%d)\n", \
                #X, __FILE__, __LINE__);        \
                abort();                        \
    }                                           \
} while(0)
#endif

#ifndef _LINUX_IN6_H
#warning Specifying own struct in6_ifreq
/* This is in linux/include/net/ipv6.h but not exposed anywhere
 * else, it would seem
 */
struct in6_ifreq {
    struct in6_addr ifr6_addr;
    __u32 ifr6_prefixlen;
    unsigned int ifr6_ifindex;
};
#endif

#define X_SUCCESS   0
#define X_ERROR     1
#define X_NOTIMPL   2
#define X_USAGE     3

static inline int ipv6_addr_any(const struct in6_addr *a)
{
        return ((a->s6_addr32[0] | a->s6_addr32[1] |
                 a->s6_addr32[2] | a->s6_addr32[3] ) == 0);
}

int sock = 0; /* socket to kernel for ioctls - shared throughout utility */

void usage() {
    fprintf(stderr, "Usage:\n    token <interface> [<token>]\n");
    fprintf(stderr, "[GS]et IPv6 IID token for given interface\n");
    fprintf(stderr, "  where <interface> e.g. eth0\n");
    fprintf(stderr, "  and   <token> e.g. ::b00b:babe\n");
    fprintf(stderr, "\nToken '::' - resume using EUI64-based IIDs\n");
    if(sock!=0) close(sock);
    exit(X_USAGE);
}

/* go away, tidy up */
void pdie(const char *x) {
    perror(x);
    if(sock!=0) close(sock);
    exit(X_ERROR);
}

/* interface name -> device index in kernel */
int get_ifindex(const char *intf) {
    struct ifreq ifr;
    ASSERT(sock!=0);
```

```
    strncpy(ifr.ifr_name, intf, IFNAMSIZ);
    if(ioctl(sock, SIOCGIFINDEX, &ifr) < 0) pdie("SIOCGIFINDEX");

    return ifr.ifr_ifindex;
}

/* v6 literal -> correctly beefed ai_addr field of a sockaddr_in6 */
int tok2sockaddr(const char *token, struct sockaddr_in6 *sap) {
    struct addrinfo hints, *ai;
    int s;
    ASSERT(sock!=0);

    /* We only want IPv6 address */
    memset (&hints, '\0', sizeof hints);
    hints.ai_family = AF_INET6;

    /* Borrow getaddrinfo's glue for resolving stringified
     * literals into sockaddr_in6
     */
    if ((s = getaddrinfo(token, NULL, &hints, &ai))) {
        fprintf(stderr, "getaddrinfo: %s: %d\n", token, s);
        return -1;
    }

    /* The only bit of interest is the address... */
    memcpy(sap, ai->ai_addr, sizeof(struct sockaddr_in6));

    /* All good children clean their rooms */
    freeaddrinfo(ai);

    return 0;
}

void show_token(const char *intf) {
    struct in6_ifreq ifr6;
    ASSERT(sock!=0);

    /* Load the if index into the query structure, null the rest */
    memset(&ifr6, 0, sizeof(struct in6_ifreq));
    ifr6.ifr6_ifindex = get_ifindex(intf);

    /* Tease the kernel. Make it cough up the token */
    if (ioctl(sock, SIOCGIFTOKEN, &ifr6) < 0) pdie("SIOCSIFTOKEN");

    /* What's a neater way of dumping v6 addresses? */
    if(ipv6_addr_any(&ifr6.ifr6_addr))
        printf("No token on %s - will generate EUI64 IIDs\n", intf);
    else
        printf("Token on %s is ::%04x:%04x:%04x:%04x\n", intf,
        ntohs(ifr6.ifr6_addr.s6_addr16[4]), ntohs(ifr6.ifr6_addr.s6_addr16[5]),
        ntohs(ifr6.ifr6_addr.s6_addr16[6]), ntohs(ifr6.ifr6_addr.s6_addr16[7]));
}
```

```
void set_token(const char *intf, const char *token) {
    struct sockaddr_in6 sa;
    struct in6_ifreq ifr6;
    ASSERT(sock!=0);

    /* Convert string-token into in6_addr structure */
    if(tok2sockaddr(token, &sa)<0) {
        herror(token);
        pdie(NULL);
    }

    /* Prepare token-set ioctl */
    memcpy((char *) &ifr6.ifr6_addr, (char *) &sa.sin6_addr,
           sizeof(struct in6_addr));
    ifr6.ifr6_ifindex = get_ifindex(intf);
    ifr6.ifr6_prefixlen = 64; /* Tokens MUST be bottom 64 */
    if (ioctl(sock, SIOCSIFTOKEN, &ifr6) < 0) pdie("SIOCSIFTOKEN");

    printf("On next RA received on %s, the IID will be %s\n", intf, token);
}

void dump_all_interfaces() {
    struct ifconf ifc;
    struct ifreq *ifr;
    int mxreq=10;
    int n;
    ASSERT(sock!=0);

    ifc.ifc_buf = NULL; /* just be sure... */
    for(;;) {
        ifc.ifc_len = sizeof(struct ifreq) * mxreq;
        if((ifc.ifc_buf = realloc(ifc.ifc_buf, ifc.ifc_len)) == NULL)
           pdie("Out of memory");

        if(ioctl(sock,SIOCGIFCONF, &ifc) <0) {
            free(ifc.ifc_buf);
            pdie("SIOCGIFCONF");
        }

        /* 10 wasn't enough, so if we're full, assume overflow and run again */
        if(ifc.ifc_len == sizeof(struct ifreq) * mxreq) {
            mxreq += 10;
            continue;
        }
        break;
    }

    /* Walk through the interfaces */
    ifr = ifc.ifc_req;
    for (n = 0; n < ifc.ifc_len; n += sizeof(struct ifreq)) {
        show_token(ifr->ifr_name);
        ifr++;
    }
}
```

```
int main(int argc, char **argv) {
    /* Open a socket to the kernel */
    if((sock = socket(PF_INET6, SOCK_DGRAM, 0)) < 0) pdie("sock");

    switch(argc) {
        case 1:
            dump_all_interfaces();
            break;
        case 2:
            show_token(argv[1]);
            break;
        case 3:
            set_token(argv[1], argv[2]);
            break;
        default:
            usage();
    }

    (void) close(sock);
return X_SUCCESS;
}
```

6 Testing

The data below is taken from two sources captured whilst testing the different aspects of this implementation: the kernel debug log and the `tcpdump` packet capture utility, watching specifically for router solicitations and router advertisements. (Some superfluous data has been removed for formatting).

1. Initial post-boot stage, no token specified and router advertisements being received on device `eth0` (kernel interface number 2). Router advertisement received on interface 2 with prefix information option set containing network prefix for local link, which is marked as for autonomous auto-configuration. No token in device configuration, therefore EUI64 address correctly used in SLAAC.

```
17:59:27 fe80::280:c8ff:feb9:a8b9 > ff02::1: [icmp6 sum ok] \
   icmp6: router advertisement \
   (chlim=64, pref=medium, router_ltime=1800, reachable_time=0, \
    retrans_time=0) \
   (src lladdr: 00:80:c8:b9:a8:ba) \
   (prefix info: LA valid_ltime=3600,preferred_ltime=1800, \
    prefix=2001:630:d0:f111::/64) \
   (len 88, hlim 255)

17:59:27 addrconf_prefix_rcv - not using tokens. Using EUI64 on eth0
17:59:27 addrconf_prefix_rcv - address not in hash \
   (prefix rx:  2001:0630:00d0:f111:0000:0000:0000:0000, \
    candidate: 2001:0630:00d0:f111:0230:1bff:feb3:33dd)
```

At this point, the addresses configured on the interface are thus:

```
[mkt@apu ~]% ip -f inet6 addr show eth0
2: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qlen 1000
    inet6 2001:630:d0:f111:230:1bff:feb3:33dd/64 scope global dynamic
       valid_lft 3593sec preferred_lft 1793sec
```

2. User-land tool invoked with parameters that request configuration of `eth0` device to have a token of `::beef`. Router solicitation sent and advertisement received almost immediately in response, resulting in new interface address configured using a tokenised identifier.

```
18:00:20 addrconf_set_iftoken - invoked
18:00:20 addrconf_set_iftoken - ... saw set request for \
    0000:0000:0000:0000:0000:0000:0000:beef/64, on index 2
18:00:20 inet6_token_set - token \
    out 0000:0000:0000:0000:0000:0000:0000:0000; \
    in 0000:0000:0000:0000:0000:0000:0000:beef

18:00:20 fe80::230:1bff:feb3:33dd > ff02::2: [icmp6 sum ok] \
   icmp6: router solicitation \
   (src lladdr: 00:30:1b:b3:33:dd) (len 16, hlim 255)
18:00:20 fe80::280:c8ff:feb9:a8b9 > ff02::1: [icmp6 sum ok] \
   icmp6: router advertisement \
   (chlim=64, pref=medium, router_ltime=1800, reachable_time=0, \
    retrans_time=0)\
   (src lladdr: 00:80:c8:b9:a8:ba)\
   (prefix info: LA valid_ltime=3600,preferred_ltime=1800, \
    prefix=2001:630:d0:f111::/64)\
   (len 88, hlim 255)

18:00:20 addrconf_prefix_rcv - will use token \
    0000:0000:0000:0000:0000:0000:0000:beef on eth0
18:00:20 addrconf_prefix_rcv - which makes \
    2001:0630:00d0:f111:0000:0000:0000:beef
18:00:20 addrconf_prefix_rcv - address not in hash (prefix rx: \
    2001:0630:00d0:f111:0000:0000:0000:0000, candidate: \
    2001:0630:00d0:f111:0000:0000:0000:beef)
```

At this point, the addresses configured on the interface show the new tokenised address is valid and preferred, and the old EUI64 derived address is valid but deprecated. The old preferred timer for the old address counts down from zero at the point of the `ioctl` setting the new token instigated and the validity timer continuing toward zero, at which point the kernel will remove the address.

```
[mkt@apu ~]% ip -f inet6 addr show eth0
2: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qlen 1000
    inet6 2001:630:d0:f111::beef/64 scope global dynamic
       valid_lft 3592sec preferred_lft 1792sec
    inet6 2001:630:d0:f111:230:1bff:feb3:33dd/64 scope global \
       deprecated dynamic
       valid_lft 3527sec preferred_lft -8 sec
```

3. User-land tool invoked without any parameters. Tool iterates through available devices, querying for currently configured token.

```
18:00:25 addrconf_get_iftoken - invoked
18:00:25 inet6_token_get - token currently \
    0000:0000:0000:0000:0000:0000:0000:0000 on 1
18:00:25 addrconf_get_iftoken - invoked
18:00:25 inet6_token_get - token currently \
    0000:0000:0000:0000:0000:0000:0000:beef on 2
```

4. User-land tool invoked with parameters requesting a change in token to `::babe`. Router solicitation and concordant advertisement sent and received as before.

```
18:00:39 addrconf_set_iftoken - invoked
18:00:39 addrconf_set_iftoken - ... saw set request for \
    0000:0000:0000:0000:0000:0000:0000:babe/64, on index 2
18:00:39 inet6_token_set - token \
    out 0000:0000:0000:0000:0000:0000:0000:beef; \
    in 0000:0000:0000:0000:0000:0000:0000:babe

18:00:39 fe80::230:1bff:feb3:33dd > ff02::2: [icmp6 sum ok] \
   icmp6: router solicitation \
   (src lladdr: 00:30:1b:b3:33:dd) (len 16, hlim 255)
18:00:39 fe80::280:c8ff:feb9:a8b9 > ff02::1: [icmp6 sum ok] \
   icmp6: router advertisement \
   (chlim=64, pref=medium, router_ltime=1800, reachable_time=0, \
    retrans_time=0)\
   (src lladdr: 00:80:c8:b9:a8:ba)\
   (prefix info: LA valid_ltime=3600,preferred_ltime=1800, \
    prefix=2001:630:d0:f111::/64)\
   (len 88, hlim 255)

18:00:40 addrconf_prefix_rcv - will use token \
    0000:0000:0000:0000:0000:0000:0000:babe on eth0
18:00:40 addrconf_prefix_rcv - which makes \
    2001:0630:00d0:f111:0000:0000:0000:babe
18:00:40 addrconf_prefix_rcv - address not in hash (prefix rx: \
    2001:0630:00d0:f111:0000:0000:0000:0000, candidate: \
    2001:0630:00d0:f111:0000:0000:0000:babe)
```

At this point, the addresses configured on the interface show the new-new tokenised address is valid and preferred, the former tokenised address valid but deprecated, and the old EUI64 derived address also valid but still deprecated with no 'refresh' of its timer data performed by the kernel.

```
[mkt@apu ~]% ip -f inet6 addr show eth0
2: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qlen 1000
    inet6 2001:630:d0:f111::babe/64 scope global dynamic
       valid_lft 3597sec preferred_lft 1797sec
    inet6 2001:630:d0:f111::beef/64 scope global deprecated dynamic
```

```
      valid_lft 3569sec preferred_lft -3 sec
   inet6 2001:630:d0:f111:230:1bff:feb3:33dd/64 scope global \
      deprecated dynamic \
      valid_lft 3504sec preferred_lft -31 sec
```

5. No further administrator requests from user-land. Regular router advertisements received and correct address candidate used as per stable operational state.

```
18:11:56 fe80::280:c8ff:feb9:a8b9 > ff02::1: [icmp6 sum ok] \
   icmp6: router advertisement \
   (chlim=64, pref=medium, router_ltime=1800, reachable_time=0, \
    retrans_time=0) \
   (src lladdr: 00:80:c8:b9:a8:ba) \
   (prefix info: LA valid_ltime=3600,preferred_ltime=1800, \
    prefix=2001:630:d0:f111::/64) \
   (len 88, hlim 255)

18:07:18 addrconf_prefix_rcv - will use token \
    0000:0000:0000:0000:0000:0000:0000:babe on eth0
18:07:18 addrconf_prefix_rcv - which makes \
    2001:0630:00d0:f111:0000:0000:0000:babe

18:11:56 fe80::280:c8ff:feb9:a8b9 > ff02::1: [icmp6 sum ok] \
   icmp6: router advertisement \
   (chlim=64, pref=medium, router_ltime=1800, reachable_time=0, \
    retrans_time=0) \
   (src lladdr: 00:80:c8:b9:a8:ba) \
   (prefix info: LA valid_ltime=3600,preferred_ltime=1800, \
    prefix=2001:630:d0:f111::/64) \
   (len 88, hlim 255)

18:11:56 addrconf_prefix_rcv - will use token \
    0000:0000:0000:0000:0000:0000:0000:babe on eth0
18:11:56 addrconf_prefix_rcv - which makes \
    2001:0630:00d0:f111:0000:0000:0000:babe
```

7 Summary

The implementation has demonstrated to be a successful implementation of tokenised interface identifiers in the linux kernel.

The `ioctl` interface is consistent in principle with the approach taken by Sun Microsystems in the Solaris implementation of this non-standardised feature, with the exception that our implementation insists on tokens being 64 bits long.

Note that the behaviour should Duplicate Address Detection (DAD) fail is not specified in this implementation. The manual configuration of nodes on the same subnet to have the same interface identifier is considered a configuration error. The deployment of tokenised identifiers is intended to be strictly limited to those well-known services on server subnets, typically in small enterprises

or Small Office-Home Office environments – i.e. those environments in which a full DHC solution is often not desired.

Should operational experience demand, DAD failure handling can be added in a future revision of this (proof of concept) implementation.

Acknowledgements

This work was supported in part by the Operational Experience with IPv6 Network Renumbering project, a collaboration between Cisco Systems, Westfälsiche Wilhelms-Universität Münster, the University of Southampton and the 6NET Consortium.

References

[1] IEEE, "Guidelines for 64-bit Global Identifier (EUI-64) Registration Authority" White Paper. On-line at *http://standards.ieee.org/regauth/oui/tutorials/EUI64.html*.

[2] Thomson, S., Narten, T., "IPv6 Stateless Address Autoconfiguration" *RFC2462*, December 1998.

[3] Droms, R., Bound, J., Volz, B., Lemon, T., Perkins, C., Carney, M., "Dynamic Host Configuration Protocol for IPv6 (DHCPv6)" *RFC3315*, July 2003.

[4] Hinden, R., Deering, S., "Internet Protocol Version 6 (IPv6) Addressing Architecture" *RFC3513*, April 2003.