# The Challenges of Dynamic Network Interfaces

Brooks Davis
*The FreeBSD Project*
*Seattle, WA*
`brooks@{aero,FreeBSD}.org`

## Abstract

On early BSD systems, network interfaces were static objects created at kernel compile time. Today the situation has changed dramatically. PC Card, USB, and other removable buses allow hardware interfaces to arrive and depart at run time. Pseudo-device cloning also allows pseudo-devices to be created dynamically. Additionally, in FreeBSD and Dragonfly, interfaces can be renamed by the administrator. With these changes, interfaces are now dynamic objects which may appear, change, or disappear at any time. This dynamism invalidates a number of assumptions that have been made in the kernel, in external programs, and even in standards such as SNMP. This paper explores the history of the transition of network interfaces from static to dynamic. Issues raised by these changes are discussed and possible solutions suggested.

## 1 Introduction

In the early days of UNIX, network interfaces were static. The drivers were compiled into the kernel along with their hardware addresses. The set of devices on each machine changed only when the administrator modified the kernel. Those days are long gone. Today devices, hardware and virtual, may come and go at any time. This dynamism creates a number of problems for both kernel and application developers.

This paper discusses how the current dynamism came about in FreeBSD, documents the problems it causes, and proposes solutions to some of those problems. The following section details the history of dynamic devices from the era of purely static devices to the modern age of near complete dynamism. Following this history, the problems caused by this dynamism are discussed in detail. Then solutions to some of these problems are proposed and analyzed, and advice to implementers of userland applications is given. Finally, the issues are summarized and future work is discussed.

Published in the Proceedings of EuroBSDCon 2004

## 2 History

In early versions of UNIX, the exact set of devices on the system had to be compiled in to the kernel. If the administrator attempted to use a device which was compiled in, but not installed, a panic or hang was nearly certain. This system was easy to program and efficient to execute. Unfortunately, it was not convenient to administer as the set of available interfaces grew.

4.1BSD, released June, 1981, included a solution to this problem called autoconfiguration [McKusick2]. Autoconfiguration is a process by which devices are detected at boot time [McKusick1]. Under autoconfiguration, each system bus is probed for devices and those devices that have drivers in the system are enabled. The process of identifying devices is called probing and devices which are found in probing are attached. The procedure used to probe devices varies by bus. On some buses, such as non-PnP ISA, compiled-in addresses are probed and if they respond as expected the device is assumed to be there. With more advanced buses such as PCI or SCSI, devices are self-identifying. PCI devices are identified by an ID number composed of a vendor portion and a vendor allocated product portion. SCSI devices are identified by a device class and a free form string. With autoconfiguration, all devices to be used still had to be compiled in to the kernel, but a super set could be used enabling one kernel to work with multiple system configurations.

In FreeBSD 2.0, the LKM (Loadable Kernel Modules) system modeled after the facility in SunOS 4.1.3 was implemented by Terry Lambert [man4-2]. The LKM system freed administrators from the requirement that all device drivers be compiled into a single static kernel. Now devices could be loaded at run time. This enabled support for a number of new features. Devices manufactured after the kernel was first built could be supported without a full rebuild. Pseudo devices could be added on the fly. And some development testing could be done without a reboot[1]. With LKM came the possibility of devices coming and going during run time. Generalized support for detaching interfaces was not implemented until Doug Rabson replaced LKM with the dynamic kernel linker (KLD) and newbus in FreeBSD 3.0. As part of this process, he implemented a primitive version of `if_detach()`. The KLD interface is an enhanced module system based on dynamic linking of ELF binaries.

While modules introduced the possibility of dynamic interfaces, dynamic interfaces were first used by ordinary people with the introduction of PC Card (PCMCIA) devices. The PC Card standard supports hot insertion and removal of cards. Through use of short pins, a few milliseconds of warning that a device is departing are provided, but otherwise, they come and go at will. In the 2.x time frame, PAO[2] was developed to provide PC Card support to FreeBSD [PAO]. Fairly functional support was available based on the work in PAO in FreeBSD 3. With the release of FreeBSD 4, PAO development ceased because all major changes had been incorporated into the FreeBSD tree. CardBus support was added in FreeBSD 5.0 and is currently fairly mature (roughly speaking, CardBus is to PCI what PC Card is to ISA.)

In FreeBSD 4.0, support for USB Ethernet devices was added. Like PC Card devices, USB devices may be attached and detached at will.

Since the introduction of USB networking devices, a number of new types of removable networking devices have appeared. The fwe(4) [man4], Ethernet over FireWire driver appeared in 5.0 and was later merged in to 4.8. The fwip(4), IP over FireWire interface, implements RFC 2734 [RFC2734], IPv4 over IEEE1394, and RFC 3146 [RFC3146], IPv6 over IEEE1394; it was introduced in 5.3. Bluetooth support was introduced in FreeBSD 5.2. Being wireless, Bluetooth devices are inherently dynamic. While not currently supported, hot plug PCI, Compact PCI, PCI Express, and Express Card all support some form of hot insertion and removal. Compact PCI is of particular interest because it provides the administrator with a button to press to indicate their intention to remove a device and a light the OS can use to notify the administrator that the device in inactive and removal is now safe.

I anticipate that as bus standards evolve, an increasing number will support hot plug devices. Further, I expect that in the not too distant future, hot plug interfaces will be the norm with the exception of integrated interfaces on motherboards.

In early autoconfiguration implementations, a finite number of units was statically allocated when the kernel or module was compiled. Devices implementing these sorts of preallocations are referred to as count devices due to the kernel configuration directive used to declare them. Today, this sort of hard coding is frowned upon unless there can only be a small, fixed number of devices (usually one). In FreeBSD 6.x, support for count devices will be removed from the config program. This will require that device counts either be fully dynamic or specified at boot time. With physical devices, new driver instances are allocated and destroyed as hardware is added and removed so the removal of count devices is no hardship.

In FreeBSD 3.4 the netgraph system was introduced-netgraph(4). Netgraph allows dynamically configured nodes implementing networking functions to be configured into arbitrary graphs. One of the standard nodes is ng_iface(4) which appears as a virtual network interface. Since netgraph nodes are configured dynamically using ngctl(8) [man8], this is another source of dynamic interfaces.

For pseudo-devices such as the loopback interface, lo(4), most devices are created by the network interface cloning code, referred to as if_clone. Network interface cloning was introduced in FreeBSD 4.4. Typically, a cloned device is created with a command like "`ifconfig vlan create`" which creates a new vlan interface and prints its name. This creates a number of interesting opportunities such as an IPv6 tunnel server that creates gif(4) devices on demand. The initial cloning code in FreeBSD was obtained from NetBSD and has since been extended to allow cloned devices to match more com-

[1]Assuming that the developer was fortunate enough to make an error that did not result in a kernel panic.

[2]According to the PAO FAQ, "PAO stands for nomads."

plex names in the ifconfig(8) create request. Initially, cloned devices could only be created with a command of the form "`ifconfig <drivername> [<unit>] create`". With enhanced cloning support, devices may support more complex names such as `<ethernet_interface>.<vlan_tag>` for vlan(4).

A new feature of FreeBSD 5.2 and DragonFly BSD is the ability to rename network interfaces. This can be useful to allow an administrator to hide the details of interface types or to easily identify the purpose of a dynamically created interface. Returning to the example of a gif(4) based tunnel server, tunnels could be named after registered users; so instead of `gif42`, the interface could be named `gif-<user>-<host>`, a much more meaningful name. Another way this feature can be useful is to give logical names to physical devices, allowing them to be upgraded or replaced without changing most system configuration. Eventually, devd(8) will support the ability to make decisions based on attributes such as slot number which could allow interfaces to be named based entirely on their location in the system.

As I have shown in this section, significant interface dynamism is present in today's network stack. With hot-pluggable and wireless hardware, kernel modules, netgraph interfaces, pseudo interface cloning, and interface renaming, virtually no device can safely be assumed to be static.

## 3 Problems

With network interface dynamism come a variety of problems of two major types. First are those having to do with userland, typically network management or monitoring applications. Most of the userland issues revolve around the fact that applications have not adapted to modern dynamic systems. In some cases, the applications have been partially adapted, but trip over problems such as the concept of a *different* interface. Second are problems in the kernel which are typically hardware race conditions or stale references to freed data.

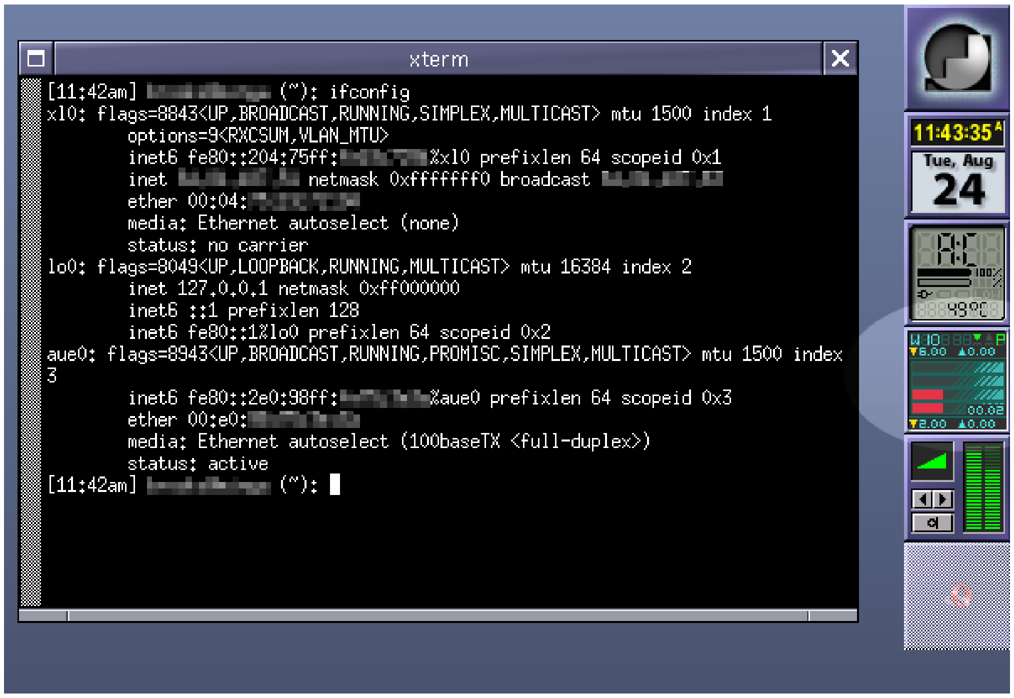
To facilitate this discussion of the problems caused by dynamic network interfaces, I will present three example systems and the problems they face:

1. A laptop with removable wired and wireless interfaces.
2. A server with hot swappable network interfaces.
3. An IPv6 tunnel server.

These systems are sufficient to expose most of the issues of dynamic network interfaces.

The laptop I will consider has a USB Ethernet interface supported by the aue(4) driver and a PC Card wireless device supported by the wi(4) driver. This situation presents several challenges. These challenges derive from the fact that these interfaces may come and go at arbitrary times. From the user's perspective there will be three significant problems. First, most of the simple network monitoring tools used in this type of environment do not detect the arrival or departure of interfaces. This causes them to only show interfaces that were attached when they were started. Second, because devices may come and go in arbitrary order, the indexes of those devices may not be consistent. Since many monitoring applications assume that the kernel index of an interface is unique for an instance of the application, they may confuse the wired and wireless interfaces with each other. Figure 1 shows an example of conflicting ifconfig(8) and `wmnd` [WMND] output. While the index is *the* valid handle to an interface, the life of that index is only the life of the driver attachment; when the device is detached, the index may be freely reused by another interface. Thus, it is not safe to assume the index will remain associated with the same interface unless the interface is being monitored for departure. The third and final userland problem is caused by interface renaming. Since interfaces may be renamed at any time, the name must not be used as a handle for an interface unless the interface list is somehow monitored for changes or some external assurance is provided that the name will not change. Automated monitoring systems should therefore assume that interfaces may change their names, but it is perfectly reasonable to use names for start up configuration or as part of an ifconfig(8) command line.

In addition to these userland problems, there are two classes of kernel issues. The first is stale pointers to the `struct ifnet`. When an interface is removed, its `struct ifnet` is destroyed, but sometimes references to the interface in the form of pointers to that structure remain. The `struct ifnet` is the device independent interface to a network interface within the kernel. A number of structures including `struct mbuf` may contain a pointer to the `struct`



```
[11:42am] [redacted] (~): ifconfig
xl0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500 index 1
        options=9<RXCSUM,VLAN_MTU>
        inet6 fe80::204:75ff:[redacted]%xl0 prefixlen 64 scopeid 0x1
        inet [redacted] netmask 0xfffffff0 broadcast [redacted]
        ether 00:04:[redacted]
        media: Ethernet autoselect (none)
        status: no carrier
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384 index 2
        inet 127.0.0.1 netmask 0xff000000
        inet6 ::1 prefixlen 128
        inet6 fe80::1%lo0 prefixlen 64 scopeid 0x2
aue0: flags=8943<UP,BROADCAST,RUNNING,PROMISC,SIMPLEX,MULTICAST> mtu 1500 index
3
        inet6 fe80::2e0:98ff:[redacted]%aue0 prefixlen 64 scopeid 0x3
        ether 00:e0:[redacted]
        media: Ethernet autoselect (100baseTX <full-duplex>)
        status: active
[11:42am] [redacted] (~): █
```

Figure 1: The xterm shows the correct interface listing, but `wmnd` shows `wi0` which has been removed and replaced with `aue0`.

`ifnet` of an interface. Since the `struct ifnet` is destroyed immediately when the interface is detached, using these references may result in accessing random data or, worse, an unmapped page. Today little is done to prevent this race from occurring. Fortunately, since the interface is marked as down and the queues are drained early in the detach process, the system will generally not retain stale references when the `struct ifnet` is destroyed. There are however some situations that will prolong this race making it more likely it will be lost in a way that causes a kernel panic. The main one is use of the dummynet(4) system. Dummynet is a "traffic shaper, bandwidth manager, and delay emulator." While dummynet is holding packets, it currently stores a pointer to the destination interface which is used to send the packet once the desired delay has occurred. This increases the race window to the point that it will almost certainly be triggered if a significant delay is configured. This generally will not affect a typical laptop user, but could easily affect a server user.

The second kernel issue is hardware races on eject. These occur when a function that manipulates the hardware runs during or after the physical removal of a device. With devices such as PC Cards, drivers manipulate the hardware in ways that may cause system hangs or panics if the device is removed, due to issues such as corrupted reads or writes to nowhere. Complete solutions to these problems often require hardware modifications. These problems can be avoided by powering down the device in an orderly manner before removing it. My understanding is that an eventual result of modernization of the device code will be the addition of a devcontrol(8) program that allows such operations on all devices that support them. In addition to this solution, there are some workarounds to reduce the risk, but I will leave their discussion to others.

Having discussed the laptop case, I will move on to the case of a server with hot swappable interfaces. The server case has most of the problems of the laptop with two major differences. First, if hot-plug PCI or compact PCI devices are used, they close the hardware removal race by providing a mechanism for the administrator to notify the OS that the hardware is going to be detached. In theory this mechanism could also close the reference races. Unfortunately, these technologies are not currently supported by FreeBSD.

The second way the server scenario differs from the laptop scenario is that servers are often monitored by SNMP [RFC1157]. Dynamic interfaces present problems when dealing with SNMP. Today, most

SNMP agents use the kernel interface index as the `ifIndex` variable. In MIB-II (RFC 1213 [RFC1213]) the `ifIndex` variable for each interface is defined to be:

> A unique value for each interface. Its value ranges between 1 and the value of `ifNumber`. The value for each interface must remain constant at least from one re-initialization of the entity's network management system to the next re-initialization.

MIB-II defines, `ifNumber` to be:

> The number of network interfaces (regardless of their current state) present on this system.

This means that interfaces may not have sparse indexes in SNMP. This in turn will not work if interfaces are dynamic. In RFC 2233 [RFC2233], "The Interfaces Group MIB using SMIv2", section 3.1.5 attempts to revise the interface numbering constraints to allow for dynamic interfaces. They do so by removing the constraint that `ifIndex` be less than or equal to `ifNumber` which allows the index space to be sparse, and by adding the constraint that the same `ifIndex` may not be reused by a *different* dynamic interface.

Unfortunately, the concept of a different interface is complicated and application specific. RFC 2233 simply states that the following constraints must be observed:

1. a previously-unused value of `ifIndex` must be assigned to a dynamically added interface if an agent has no knowledge of whether the interface is the "same" or "different" to a previously incarnated interface.
2. a management station, not noticing that an interface has gone away and another has come into existence, must not be confused when calculating the difference between the counter values retrieved on successive polls for a particular `ifIndex` value.

In the simplest case of the server with hot-plug interfaces, the current system mostly works because interfaces are typically added but not removed except to be replaced by a different device serving the same function. However, the second constraint above may not be handled correctly in this case because the counters are attached to the interface and will be reset. A slight modification to the agent to allow detection of this case and setting the `ifCounterDiscontinuityTime` object for the interface when its removal is detected would correct this issue.

The more complex case of a server with frequent interface arrivals and departures is typified by the IPv6 tunnel server scenario. This tunnel server has hardware similar to that in the previous scenario, but has a vastly different mode of operation. Registered users may request a tunnel for one or more hosts. When requested, a gif(4) interface is created using cloning. When the user requests that the tunnel be torn down or a specified timeout passed, the interface is destroyed. Because interfaces are created on demand, the automatically assigned kernel interface indexes should not be used for SNMP `ifIndex` values as is. The problem is that the only tunnel interfaces that may be considered the *same* are those which share the same user, host[3] pair. Thus, since kernel interface indexes will be allocated in a manner which attempts to limit the sparseness of the index space, kernel indexes will frequently reference *different* interfaces once a few interfaces have been destroyed. Ideally, the tunnel server should allocate `ifIndex` values and inform the SNMP agent when interfaces are created, but this is easier said than done, as most agents simply assume that the kernel index is the correct value for `ifIndex`. Since the user and host are not known to the kernel, there is no current mechanism for the kernel to choose a correct value for `ifIndex`. Additionally, there is no easy way to control the index from userland.

The problems posed by these three example systems cover most of the issues caused by dynamic network interfaces. Kernel and hardware races present challenges for kernel developers, and the complexity of maintaining consistent references to interfaces causes problems in userland.

[3]By host we mean the machine itself, not the IP address in most cases, e.g. a laptop might move about, but would be the same host.

## 4 Solutions

In this section I propose and evaluate solutions to some of the problems presented in the previous section. In particular, I discuss two possible solutions to the problem of stale `struct ifnet` references, as well as the kernel framework for a partial solution to the problems of inconsistent indexes to the *same* interface.

At first glance, the problem of stale `struct ifnet` references would seem to be solvable through the simple addition of reference counts. After all, the problem is that references to the interface's `struct ifnet` are still held when the structure is freed. Unfortunately, there are significant problems with this approach. The first is simply that reference counts are expensive to maintain. Incrementing or decrementing a reference count requires either obtaining a lock or using another atomic operation. This is especially problematic when code is in the fast path since ever moment counts and many atomic operations take over a hundred cycles to complete. Since the `struct ifnet` references in dummynet and `struct mbuf` are used in the fast path, atomic or mutex operations should be avoided there if possible. The second problem is that the `struct ifnet` is part of the softc of physical interfaces which is destroyed when the device is detached. This means that a reference count might not prevent the destruction of the `struct ifnet`. The `struct ifnet` could be moved to separate storage to be managed by the networking system, but doing so would required modifications to virtually every one of the approximately 100 network drivers in the system plus all the externally maintained ones. Not only would this be difficult, but it would fail to resolve the hardware races, so the effort is unlikely to be worthwhile. Due to these problems, reference counting `struct ifnet` is unlikely to work.

There is a second possible solution, which is referencing the interface by index instead of a direct pointer to `struct ifnet`. In this case, each long lived `struct ifnet` pointer would be replaced with the interface's index. The pointer dereferences would be replaced with `ifnet_byindex()` called. To avoid null-pointer dereferences in this case, `ifnet_byindex()` would be modified to return a pointer to a special `dead_if` interface which has no-op functions in place of driver specific ones. Where possible, the `dead_if struct ifnet` would be filled with values that will not provoke panics. In general, kernel code should be modified to check the return value of `ifnet_byindex()` for `dead_if` and abort processing unless the check is more expensive than completing the operation and the expense matters. This solution avoids the need for an explicit (and expensive) atomic operation because assignment to pointers is atomic on all architectures supported by FreeBSD. If the modifications to `ifnet_byindex()` are done by insuring that the array used to implement it has all empty entries filled with pointers to `dead_if`, there will be no performance impact on `ifnet_byindex()`. There will be some performance impact on code that previously referenced `struct ifnet` directly since an additional look up will be needed. Since `ifnet_byindex()` is simply a macro as shown in Figure 2, this should be relatively cheap, but performance testing will be needed to precisely quantify the extent of the impact. If the performance impact is deemed too high, it may be possible to use macros to choose between these solutions at compile time so that environments such as dummynet systems with dynamic interfaces could optionally enable this extra level of indirection. It is worth noting that while using indirect references to `struct ifnet` will shrink the race, it will not completely close it.

```
struct ifindex_entry {
    struct ifnet *ife_ifnet;
    struct ifaddr *ife_ifnet_addr;
    struct cdev *ife_dev;
};

#define ifnet_byindex(idx) \
    ifindex_table[(idx)].ife_ifnet
#define ifaddr_byindex(idx) \
    ifindex_table[(idx)].ife_ifnet_addr
#define ifdev_byindex(idx) \
    ifindex_table[(idx)].ife_dev
```

Figure 2: `ifnet_byindex()` and related macros from `sys/net/if_var.h`

The problem of SNMP agents assuming that the kernel interface index is a good value for `ifIndex` is difficult to solve, short of rewriting the agent to remove this assumption and forcing the agent to manage its own application specific `ifIndex` space. The kernel will assign the same indexes to interfaces across reboots and some effort is made to preserve indexes across module reloads, but since the allocator attempts to avoid sparse allocations, the indexes are inherently unsuited to the requirements of SNMP agents in applications such as an IPv6 tunnel server. One possible solution to this problem is to enable userland programs to set the kernel index of interfaces.

I propose an implementation of this functionality as follows: setting the index will only be allowed when the interface is not in the `IFF_UP` state. The actual change will take place by detaching the interface, changing the index, clearing the interface statistics, and reattaching the interface. From the perspective of userland applications, the interface will be destroyed and a new interface will be created with the desired index. A tunnel server controller process could use this functionality to create interfaces with userland managed indexes, thus allowing SNMP agents to work with fewer modifications. The agent will need to set `ifCounterDiscontinuityTime` appropriately. To aid in setting it, it may be useful to add a new per-interface variable indicating the epoch of the interface. The epoch would be reset any time the interface statistics were reset.

There are a few potential issues with this approach. First, the `if_index` variable in `struct ifnet` is a signed short so the useful range of index values is 1 to 32767 ($2^{15}-1$) which is not very large for some applications. This could be solved by increasing `if_index` to an int or long, but that would raise other issues. Specifically, there are some arrays that are currently required to be at least as long as the highest index. If the index is allowed to grow to `INT_MAX`, these arrays would be larger than the system address space of a 32-bit system. As such, these interfaces would have to be modified to use more complex structures such as trees or hashes. This would cause some operations such as `ifnet_byindex()` to change from constant time to $O(logn)$. This could severely impact system performance if indirect references were used in the fast path as suggested earlier in this section.

The second issue with expanding `if_index` is related to the problem of sparse indexes. With the current limit on the maximum index, storage concerns are not insoluble, but there are efficiency concerns. In the kernel this should not be a major issue as there is no reason to search for interfaces one index at a time when the interface list can be walked directly. In userland things are more difficult. The correct way to access interface information is via sysctl(3), but sysctl(3) does not provide the equivalent of SNMP's `GetNext` functionality. This means that walking the list of interfaces by index could take 32767 syscalls with the existing implementation. This is probably not acceptable overhead for each update of a monitoring interface. Even without expanding `if_index`, it will probably be necessary to provide better sparse access support to userland. Some options for this include adding a `GetNext` equivalent to sysctl(3), adding a next interface pointer to the sysctl(3) output, or publishing a list or bitmap of allocated indexes. A `GetNext` equivalent for sysctl(3) or the addition of a next interface pointer would allow applications to only make syscalls for information that actually exists. A list would be easy to produce and cheap to process in userland, but a bitmap would be smaller and could be maintained at virtually no cost. A bitmap is probably the easiest option.

I have presented possible solutions to two of the problems of dynamic interfaces. The solution of adding a layer of indirection to long-lived, stored references to `struct ifnet` shows some promise if performance is acceptable. Allowing userland applications to control kernel interface index allocation may or may not be useful in practice. It would allow tunnel servers to work with more or less unmodified SNMP agents, but it would not provide a full solution. A full solution will probably require application specific agents or better generalization of generic agents to allow application specific `ifIndex` management.

## 5 Advice to Application Implementers

Other than the problems with SNMP agents and indexes, most userland issues with dynamic network interfaces are problems of application design. Most simple interface monitoring tools such as `wmnd` are written with the assumption that once the application is started, the set of interfaces will remain constant. Since this is not the case with modern versions of FreeBSD, these applications behave in unexpected (though generally harmless) ways.

To prevent this problem, applications should use appropriate APIs to access interface data, and should use those APIs in ways that allow detection of changes to the list of interfaces. In particular, applications need to detect the arrival, departure, and renaming of interfaces. In this section, three possible ways to do so are presented. The first way is to periodically rescan the entire list of interfaces. In environments with few interfaces this may be done for every application refresh or it may be done less frequently if scanning the whole list is too expensive and delayed detection of changes in the list is acceptable. Another method is to monitor the `/dev/net` directory for the comings and goings of device nodes. This can be accomplished with the kqueue(2) [man2] mechanism or by scanning the directory with readdir(3) [man3]. A third approach (applicable only to programs that run as root outside a jail) is to monitor the routing socket for arrival and departure notifications.

There are two related complications with the third approach. If an interface is destroyed and then replaced between update cycles, the application needs to detect this some way. This isn't an issue with routing sockets or kqueues on `/dev/net` because notices will be sent for for both arrival and departure, but since the list is monitored via sysctl or by using opendir on `/dev/net`, there may be continuity problems where an interface appears to still exist, but in fact has been replaced with another. In the case of the routing socket there is an issue that a rename is modeled as a detach and attach which means a application may need a heuristic to detect this situation. To aid in solving this problem, I have added an interface epoch variable to FreeBSD. The epoch helps with both the problem of detecting interfaces that replace removed ones in the same cycle and interfaces that were renamed rather then removed. In this context, an interface is the same if and only if both its index and epoch are the same. In the routing socket case, replacing the current detach and attach notifications with a rename notification would be the ideal solution.

Once applications have been modified or written to notice new interfaces, the author may wish to consider ways to bring these new interfaces to the user's or administrator's attention. Exactly how this should be done is application specific. For example, in a WindowMaker dock application on a laptop, bringing new interfaces to the front may be the best approach, but that certainly wouldn't be appropriate to a tunnel server.

In addition to monitoring for added or removed interfaces, application designers should avoid the following two practices. First, many current applications refer to interfaces by name internally. Since interfaces can now be renamed at any time, this is no longer considered good practice. Instead, applications should refer to interfaces by index and convert that to a name when needed. Second, many monitoring or status applications currently obtain interface information via the kvm(3) interface which provides direct access to kernel memory. This is bad practice for a number of reasons. First, requiring that applications be suid kmem is dangerous from a security perspective as it is nearly always possible to leverage kmem access to obtain root access in the case of a programming error. Second, since the `ifnet_list` is now dynamic, walking it without a lock is not reliable. Third, perfectly good sysctl interfaces exist to access this information, so there is no actual need to put up with the first two drawbacks.

## 6 Conclusions and Future Work

As I have shown above, dynamic network interfaces present a number of challenges to developers of network device drivers and network monitoring and management applications. In the kernel these challenges are divided between hardware races which may be reduced by careful programming, but may only be eliminated with external signaling mechanisms, and races involving freeing of `struct ifnet` instances before all references to them have been removed. The problem of stale `struct ifnet` references may be reduced by replacing long lived references to `struct ifnet` with interface indexes, allowing a special no-op interface to be substituted when the interface is removed. Further exploration of this idea is needed before it can be put into common use. Performance impacts will need to be quantified and it will be necessary to determine whether or not the solution reduces the race sufficiently to warrant the overhead.

In userland the challenges are generally issues with outdated assumptions in userland applications. The most common problem today is network monitoring applications that assume the set of network interfaces is static. Solving this problem requires modifying the applications to monitor for changes in the interface list such that attaches, detaches, and renames are all correctly detected. The addition of an epoch variable on each interface should help detection of some of these cases.

A secondary userland problem is specific to SNMP agents. SNMP agents need to maintain an `ifIndex` which is unique for each *different* interface. Prior to the introduction of dynamic interfaces, agents were able to use the kernel interface index for `ifIndex`. This no longer works because allocation of kernel indexes is done in a manner which minimizes sparse allocation and RFC 2233 requires that allocations be sparse in a dynamic system. Allowing userland applications to control kernel indexes may provide a workaround in some circumstances, but enhancement of SNMP agents to allow external management of `ifIndex` variables for applications such as tunnel servers will probably ultimately be necessary.

Going forward, I intend to implement a sample network interface monitoring application demonstrating best practices in this area. Blind copy-and-paste from outdated applications is probably the single most significant cause of monitoring tools that do not correctly handle network interface dynamism. An up-to-date example should help this situation significantly.

Today nearly all network interfaces are potentially dynamic and in the future I believe dynamic interfaces will be the rule rather then the exception. Give this state of affairs, future kernel and application programmers should keep the dynamic nature of network interfaces in mind when they write interface related code. In fact, programmers dealing with kernel or application level management of any hardware or virtual devices should keep dynamism in mind to avoid the sort of problems we see with network interfaces today.

## References

[McKusick1] K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*, Addison-Wesley, Boston, MA, 1996.

[McKusick2] K. McKusick. *Twenty Years of Berkeley Unix: From AT&T-Owned to Freely Redistributable*, Open Sources, O'Reilly and Associates, January 1999. `http://www.oreilly.com/catalog/opensources/book/kirkmck.html`

[man2] The FreeBSD Project, *FreeBSD System Calls Manual*, FreeBSD 5.3, 2004.

[man3] The FreeBSD Project, *FreeBSD Library Functions Manual*, FreeBSD 5.3, 2004.

[man4-2] The FreeBSD Project, *FreeBSD Kernel Interfaces Manual*, FreeBSD 2.0, 1995. `http://www.freebsd.org/cgi/man.cgi?query=lkm&manpath=FreeBSD+2.0-RELEASE`

[man4] The FreeBSD Project, *FreeBSD Kernel Interfaces Manual*, FreeBSD 5.3, 2004.

[man8] The FreeBSD Project, *FreeBSD System Manager's Manual*, FreeBSD 5.3, 2004.

[PAO] `http://www.jp.freebsd.org/PAO/`

[RFC1157] J. Case, M. Fedor, M. Schoffstall, and J. Davin, *A Simple Network Management Protocol (SNMP)*, RFC1157, IETF Network Working Group, May 1990.

[RFC1213] K. McCloghrie and M. Rose, editors, *Management Information Base for Network Management of TCP/IP-based internets: MIB-II*, RFC1213, IETF Network Working Group, March 1991.

[RFC2233] K. McCloghrie and F. Kastenholz, *The Interfaces Group MIB using SMIv2*, RFC2233, IETF Network Working Group, November 1997.

[RFC2734] P. Johansson, *IPv4 over IEEE 1394*, RFC2734, IETF Network Working Group, December 1999.

[RFC3146] K. Fujisawa and A. Onoe, *Transmission of IPv6 Packets over IEEE 1394 Networks*, RFC3146, IETF Network Working Group, October 2001.

[WMND] WindowMaker Network Devices. `http://www.yuv.info/wmnd/`