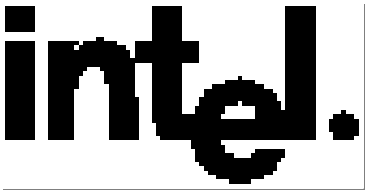


# -----Offload Sockets Framework and Sockets Direct Protocol High Level Design

**Draft 2**

**June 2002**

# *Revision History and Disclaimers*

| Rev. | Date | Notes |
|---|---|---|
| Draft 1 | March 2002 | First cut. |
| Draft 2 | June 2002 | Updated with new OFFLOAD address family architecture and naming.<br>Combined Socket Framework and SDP/IBT HLD's into one document..<br>All content included and reviewed by internal design team. |

# *Approval*

| Role | Signature | Date |
|---|---|---|
| Responsible Engineer | | |
| Engineering Group Leader | | |
| Software Engineering Manager | | |

# *Abstract*

The Offload Sockets Framework (OSF) enables network applications to utilize Linux sockets and File I/O APIs to communicate with remote endpoints across a system area network (SAN), while bypassing the kernel resident TCP/IP protocol stack. The offload sockets framework is completely transport and protocol independent and can be used to support multiple offload technologies. For the rest of this document, as an application of the Offload Sockets Framework, Sockets Direct Protocol (SDP) is used as the target protocol and InfiniBand as the target transport. However, other transport technologies (such as TCP Offload Engines - TOE) and protocols (such as iSCSI) can easily make use of the offload sockets framework.

The Sockets Direct Protocol (SDP) is an InfiniBand specific Protocol defined by the Software Working Group (SWG) of the InfiniBand Trade Association (IBTA). It defines a standard wire protocol over IBA fabric to support user-mode stream sockets-compatible (SOCK_STREAM) networking over IBA. SDP utilizes various InfiniBand features (such as remote DMA (RDMA), memory windows, solicited events etc.) for high-performance zero-copy data transfers. SDP is a pure wire-protocol level specification and does not go into any socket API or implementation specifics. Work is under way to separate the SDP wire-protocol, which is transport and O/S agnostic, from the IBA transport dependencies. This separation will enable transports other then IBA fabrics.

While SDP facilitates direct mapping of stream connections to InfiniBand reliable connections (or virtual circuits), IP-Over-IB (IPoIB) specifies a mapping of IP (both v4 & v6) protocols over IBA fabric and treats the IBA fabric simply as the link layer. IPoIB specification is currently being worked on by IETF working group and will be published as an IETF RFC. The IPoIB RFC will define the packet level format of IP packets on the IBA fabric, and also describes an InfiniBand address resolution protocol (IBARP). Conceptually, the IPoIB driver in Linux will look like a network driver and will plug-in underneath the IP stack as any standard Linux network device. The IPoIB driver exposes a network device per IBA port on the host system, and these devices could be used to assign (statically or dynamically - using protocols such as DHCP) IP addresses. The SDP stack simply makes use of these IP assignments for endpoint identifications.

SDP only deals with stream sockets (TCP), and if installed in a system, allow bypassing the OS resident TCP stack for all TCP connections between any endpoints on the IBA fabric. All other socket types (such as datagram, raw, packet etc.) are supported by the Linux IP stack and operate over the IPoIB drivers. The IPoIB stack has no dependency on the SDP stack; however, the SDP stack depends on IPoIB drivers for learning about local IP assignments and for address resolution.

The Linux implementation of SDP includes many interdependent pieces that are referred to as Offload Sockets Framework (OSF). OSF enables applications to use these same standard sockets and File I/O APIs to transparently communicate with remote endpoints/nodes across a system area network (SAN), bypassing the kernel resident TCP/IP protocol stack. This framework will also allow applications to bypass the resident TCP/IP protocol stack while using the default address family of AF_INET.

OSF includes an Offload Protocol Switch (OPS) module (Ipv4 internet protocol switch) that allows integration of Offload Protocol (OP) modules along side the existing TCP/IP stack, an OP module that supports SDP wire protocol, an Offload Transport Interface that provides a common interface to Offload Transport (OT) module that supports InfiniBand hardware. This design document covers the specifics of all three components.

# Contents

# Figures

# 1. Introduction

## 1.1 Purpose and Scope

This HLD defines the implementation of all offload components described in the *"Offload Sockets Framework and Sockets Direct Protocol Architecture Specification"*, including inter-component dependencies, and provides sufficient design detail that will satisfy the product requirements as specified.

## 1.2 Audience

Anyone interested in understanding this implementation of the Architecture Specification should read this document, including:

- Software developers who are integrating the separate modules into their own software projects
- Hardware developers who need an understanding of the software behavior to optimize their designs
- Evaluation engineers who are developing tests for InfiniBand-compliant devices
- Others in similar roles who need more than a basic understanding of the software

## 1.3 Acronyms and Terms

| | |
|---|---|
| OSF: | Offload Sockets Framework (Software components that enables protocol offloading) |
| OPS: | Offload Protocol Switch (A logical software module that performs protocol switching) |
| OT: | Offload Transport (An entity that exports reliable transport semantics) |
| OTI: | Offload Transport Interface |
| OP: | Offload Protocol (Any upper layer protocol run over OTs. E.g. Sockets Direct Protocol) |
| SDP: | Sockets Direct Protocol (A Socket emulation protocol specified for InfiniBand) |
| TOE: | TCP Offload Engine (Hardware that supports offloading TCP/IP protocol from host) |
| IBA: | InfiniBand Architecture |
| IPoIB: | IP-over-InfiniBand (and IETF defined RFC to send IP packets on InfiniBand fabric) |

## 1.4 References

**InfiniBand**

InfiniBand Architecture Specification, Version 1.0a, http://www.infinibandta.org/

IP over IB IETF draft: http://www.ietf.org/ids.by.wg/ipoib.html

InfiniBand Specification Annex A4 - Sockets Direct Protocol (SDP), Release 1.0.a

**Sockets**

Fast Sockets reference: http://www.cs.purdue.edu/homes/yau/cs690y/fastsocket.ps

Stream Socket on Shrimp reference: http://www.cs.princeton.edu/shrimp/Papers/canpc97SS.ps

**Device Drivers**

Rubini, Alessandro and Corbet, Johathan. Linux Device Drivers Book, 2nd Edition: O'reilly, June 2001. ISBN: 0-59600-008-1. http://www.xml.com/ldd/chapter/book/

# 1.5 Conventions

This document uses the following typographical conventions and icons:

| | |
|---|---|
| *Italic* | is used for book titles, manual titles, URLs, and new terms. |
| **Bold** | is used for user input (in the Installation section). |
| `Fixed width` | is used for code definitions, data structures, function definitions, and system console output. Fixed width text is always in Courier font. |

**NOTE**

Is used to alert you to an item of special interest.

**DESIGN ISSUE**

Is used to alert you to unresolved design issues that may impact the module's design, function, or usage.

# 1.6 Before You Begin

Please note the following:

This document assumes that you are familiar with the *InfiniBand Architecture Specification*, which is available from the InfiniBand Trade Association at http://www.infinibandta.org.

# 2. Features

This section lists a set of features and goals for Offload Sockets Framework (OSF) support in Linux. The items listed in this section are by no means complete and may need to be further refined before finalizing on the best solution.

- All offload protocols/transports need to have a standard Linux network driver. This allows network administrators to use standard tools (like ipconfig) to configure and manage the network interfaces and assign IP addresses using static or dynamic methods.
- The offload sockets framework should work with and without kernel patches. To this effect, the offload protocols and transports will reside under a new offload address family (AF_INET_OFFLOAD) module. Applications will be able to create socket instances over this new address family directly. However, for complete application transparency, an optional minimal patch to the Linux kernel (socket driver) can be applied to allow re-direction of AF_INET sockets to the new AF_INET_OFFLOAD address family. The AF_INET_OFFLOAD module will work as a protocol switch and interact with the AF_INET address family. The patch also defines a new address family called AF_INET_DIRECT for applications that want to be strictly using the OS network stack. This kernel patch can be optional based on distributor and/or customer requirements.
- All standard socket APIs and File I/O APIs that are supported over the OS resident network stack should be supported over offload sockets.
- Support for native Asynchronous I/O (AIO) is being worked in Linux community. The offload framework should utilize this to support newer protocol and transports that are natively asynchronous. (For example, SDP stack could utilize the AIO support to support PIPELINED mode in SDP)
- Architecture should support a layered design so as to easily support multiple offload technologies, and not just SDP. This insures the offload sockets framework is useful for multiple offload technologies.
- The proposed architecture should support implementations optimized for zero-copy data transfer modes between application buffers across the connection. High performance can be achieved by avoiding data copies and using RDMA support in modern interconnects to do zero copy transfers. This mode is typically useful for large data transfers where the overhead of setting up RDMA is negligible compared to the buffer copying costs.
- The proposed architecture should support implementations optimized for low latency small data transfer operations. Use of send/receive operations incurs lower latency than RDMA operations that needs explicit setup.
- Behavior with signals should be exactly same as with existing standard sockets.
- `listen()` on sockets bound to multiple local interfaces (with IPADDR_ANY) on a AF_INET socket should listen for connections on all available IP network interfaces in the system (both offloaded, and non-offloaded). This requires the `listen()` call from application with IPADDR_ANY to be replicated across all protocol providers including the in-kernel TCP stack.
- Multiplexed I/O operations using API's such as `select()` and `poll()` should work across AF_INET socket file descriptors supported by different protocol/transport providers including the

in-kernel IP stack. This guarantees complete transparency at the socket layer irrespective of which protocol/transport provider is bound to a socket. .

- Operations over socket connections bound to the in-kernel O/S protocol (TCP/IP) stack should be directed to the TCP/IP stack with minimum overhead. Application bound to kernel network stack should see negligible performance impact because of offload sockets support.
- Ability to fallback to kernel TCP/IP stack dynamically in case of operation/connection failure in direct mapping of stream connections to offloaded protocols/transports. Connection requests for AF_INET sockets that fail over offload stack is automatically retried with the kernel TCP/IP stack. Once a direct mapped connection is established, it cannot be failed back to the TCP stack, and any subsequent failures are reported to application as typical socket operation failures.
- Offload Socket framework enables sockets of type STREAMS only. Other socket types (such as RAW, DATAGRAMS, PACKET etc.) will use only the OS network stack.
- Offload sockets framework will support offloading of stream sessions both within local IP subnet and outside local IP subnet that needs routing. Offload protocols/transports will have the ability to specify if they do self-routing or need routing assistance. Ability to offload stream sessions to remote IP subnet will be useful for TOE vendors in general and for IBA edge router vendors who map SDP sessions on IBA fabric to TCP sessions outside fabric. For protocols/transports that do self-routing, the offload sockets framework simply forwards the requests to them. For protocols/transports that need routing support (such as SDP), the framework utilizes the OS route tables and applies its configurable policies before forwarding requests to offload transports. This enables the use of O/S managed route tables to configure both offload and non-offload stacks.
- Since the socket API extensions defined by the Interconnect Software Consortium (ICSC) in the open group is work-in-progress at this time, the offload sockets framework will not attempt to address them in this phase. This could be attempted at a later phase.
- Offload sockets framework should not affect any existing Linux application designs that uses standard OS abstractions and features (such as `fork(), exec(), dup(), clone(),` etc.). Transparency to applications should be maintained.
- Offload sockets framework should support both user-mode and kernel-mode socket clients and maintain the existing socket semantics for existing user mode or kernel mode clients.
- The offload sockets framework currently deals with only IPv4 address family. Even though the same offload concepts can be equally applied to offload IPv6 family, it is deferred for later stages of the project.

# 3. Design Assumptions & Rules

Design is based on Linux 2.4.x kernel feature set. Kernel modifications will be limited as much as possible and will be isolated with compile time switches. Asynchronous I/O kernel support that is currently under development in the Linux community will be monitored and will be supported as part of this design when it becomes available. Both IA32 and IA64 environments will be supported. This project is specific for IPv4 address family but will design with future IPv6 in mind.

# 4. Design Overview

The Offload Sockets Framework architecture implements high-performance socket support in the kernel by providing a new Offload Protocol Switch (OPS) via a new address family (AF_INET_OFFLOAD) module. This new address family module will support dynamic binding of offload protocols and transports under the offload address family. The offload protocols will register with the offload family and the transport modules will register with the offload protocol modules. The offload sockets architecture is shown in Figure 1.

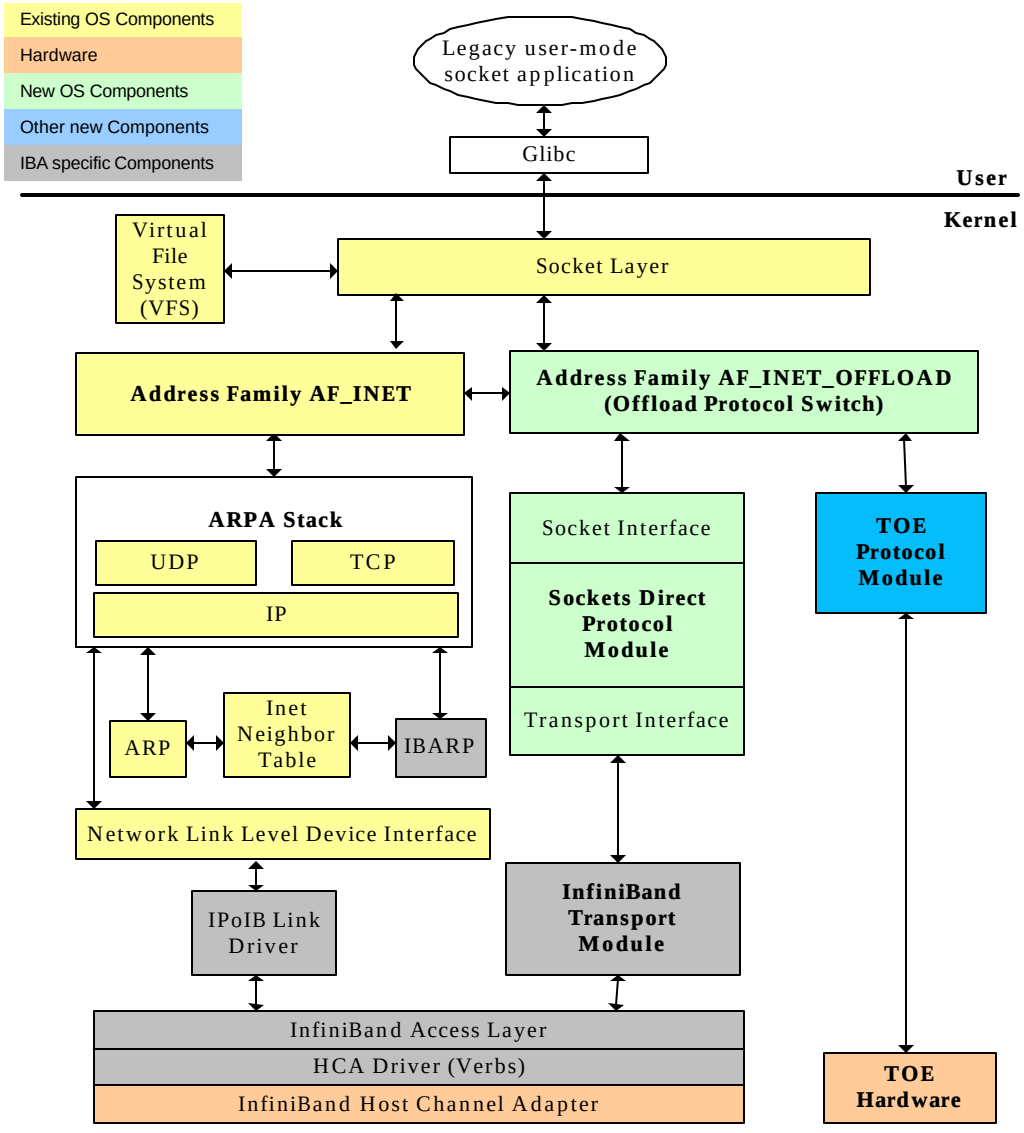



**Figure 1 High Performance Sockets in the kernel**

There are multiple major components to this, such as the Offload Protocol Switch (OPS) module, the SDP offload protocol (OP) module (which include the socket and transport Interface), and the InfiniBand offload transport (OT) module.

# 5. Design Details

## 5.1 Offload Protocol Switch Module

The Offload Protocol Switch (OPS) module exposes a new address family (AF_INET_OFFLOAD) and preserves the socket operation semantics with the kernel socket driver at the top. At the bottom it provides a new offload protocol structure and interface for offload protocols. These offload protocols will register with the AF_INET_OFFLOAD address family during initialization using an exported registration call.

The OPS module is logically a thin veneer between the socket driver and the INET offload protocols. OPS internal interfaces will be defined to provide routing information for the offload protocol modules and transport interfaces. All external interfaces to AF_INET_OFFLOAD are defined by the Linux socket calls in *`include/linux/net.h`* and *`include/linux/sock.h`*.

This module is capable of switching socket traffic across multiple offload protocol modules underneath via supplied protocol address information. The module will also be capable of failing back to standard AF_INET stack if failures occur during connection initiation over a offload protocol stack. This switching to the standard stack requires no changes to the AF_INET stack.

### 5.1.1 Linux Kernel Modifications

The OPS module will be coded as a new Linux AF_INET_OFFLOAD address family. This requires new definitions, AF_INET_OFFLOAD and AF_INET_DIRECT, in the kernel `linux/include/linux/in.h` include file.

### 5.1.2 Initialization

The OPS kernel module will be demand loaded during system initialization. The OPS module will have a dependency on the standard Linux socket driver. Actual OPS initialization is performed in accordance with the standard Linux kernel module load procedure; the '`init_module()`' function is called once the OPS module has been successfully loaded into the kernel. See section 7.1 for OPS initialization process and interaction with the offload protocol modules.

### 5.1.3 Shutdown

OPS shutdown can be initiated in one of two ways:

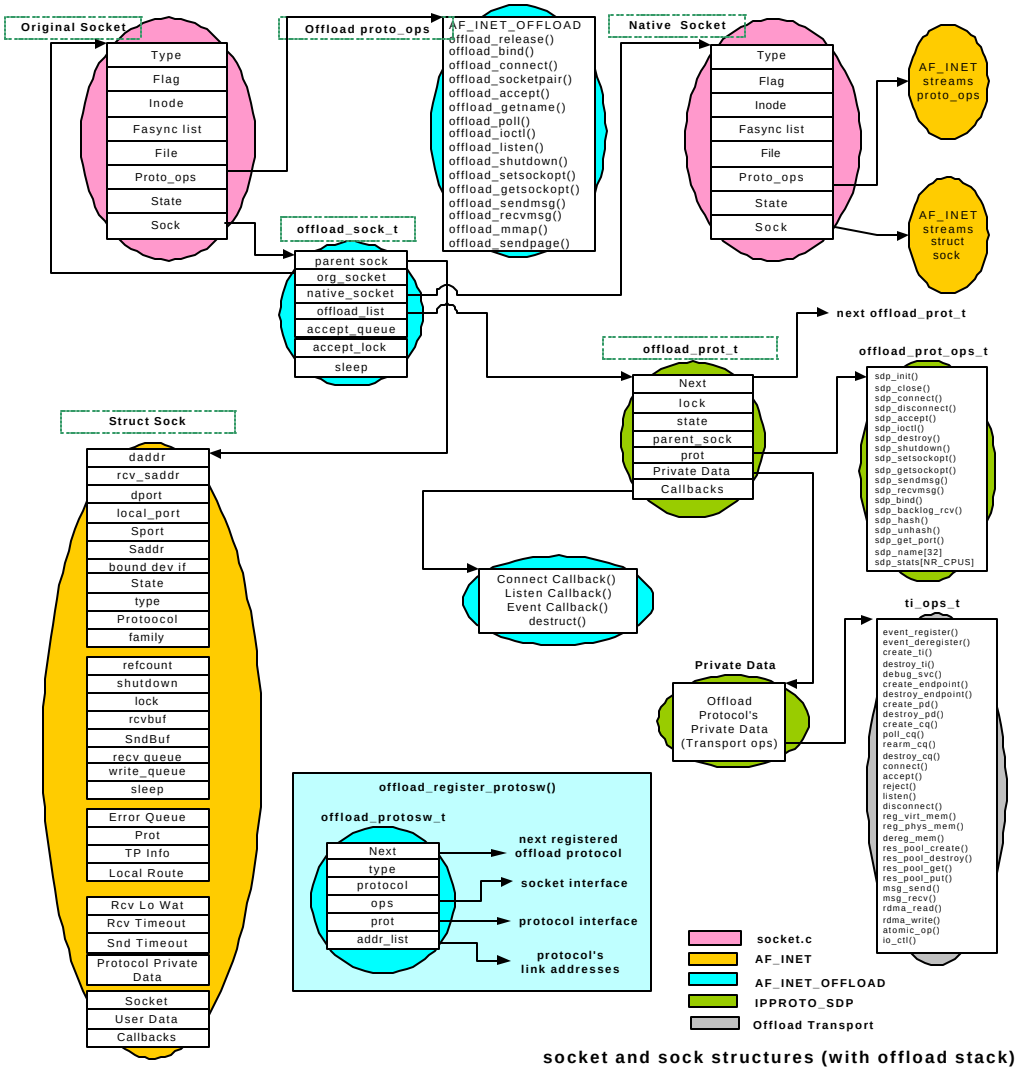
- During normal system shutdown procedures.
- The OPS module is forcibly removed using the Linux system administration command 'rmmod' (remove module).

With either method, the OPS module unload function '`cleanup_module()`' is called per the standard Linux kernel module unload procedure. The OPS '`cleanup_module()`' function will notify the protocol modules of the pending OPS module shutdown. It is expected that all offload protocol modules will shutdown by releasing allocated system resources, halting and unloading. See section 7.1 for OPS shutdown process and interaction with offload protocol modules.

## 5.1.4 Socket driver modifications

This section describes the Linux kernel patch envisioned to support the offload protocol switch (OPS). The OPS is packaged as a separate address family (AF_INET_OFFLOAD) module. This allows OPS to be used without any Linux kernel patches, as long as applications explicitly specified the AF_INET_OFFLOAD address family during socket creation. However, this solution is not suitable for applications that need binary/legacy compatibility (uses only AF_INET family sockets) and flexibility to run over offloaded or non-offloaded stacks without modifications. A minor Linux socket driver patch achieves the binary backward compatibility of applications.

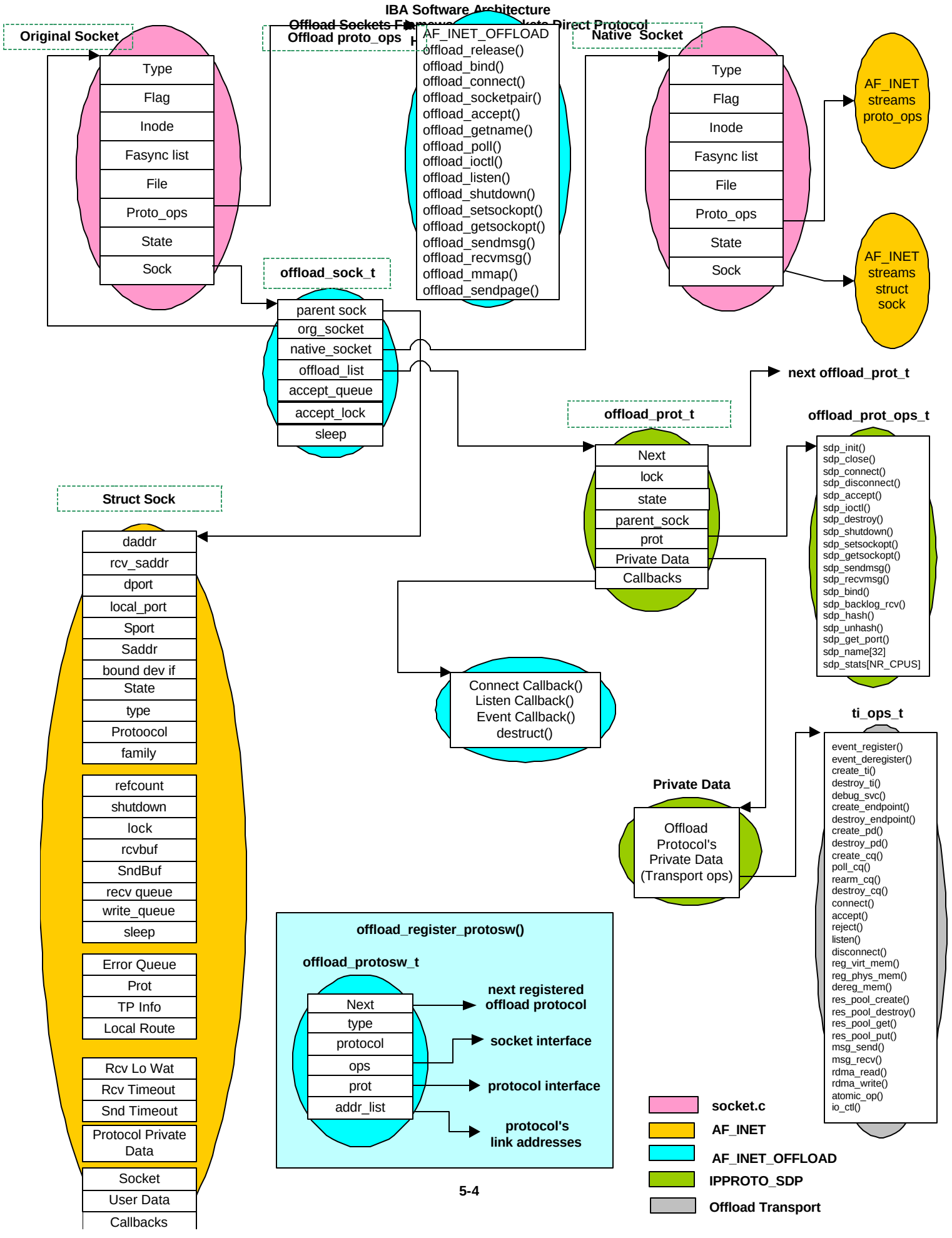
The current Linux socket driver address family switching logic is shown below first, followed by the patched socket driver logic. With the socket driver patch, whenever a AF_INET socket is created by an




**socket and sock structures (with offload stack)**

application, if it is of type SOCK_STREAM and if the OPS module is loaded and registered with the socket driver, the socket creation calls are forwarded to the OPS module (or AF_INET_OFFLOAD) family. The patch also exposes as new address family called AF_INET_DIRECT if specific applications wanted to specify during socket creation not to attempt offloading them. Section 5.1.6 explains another reason to expose the AF_INET_DIRECT address family to avoid endless loop in the socket driver.

### 5.1.5 Offload data structures


IBA Software Architecture
Original Socket
Type
Flag
Inode
Fasync list
File
Proto_ops
State
Sock
AF_INET_OFFLOAD
offload_release()
offload_bind()
offload_connect()
offload_socketpair()
offload_accept()
offload_getname()
offload_poll()
offload_ioctl()
offload_listen()
offload_shutdown()
offload_setsockopt()
offload_getsockopt()
offload_sendmsg()
offload_recvmsg()
offload_mmap()
offload_sendpage()
Native Socket
AF_INET streams proto_ops
AF_INET streams struct sock
offload_sock_t
parent sock
org_socket
native_socket
offload_list
accept_queue
accept_lock
sleep
next offload_prot_t
offload_prot_t
Next
lock
state
parent_sock
prot
Private Data
Callbacks
offload_prot_ops_t
sdp_init()
sdp_close()
sdp_connect()
sdp_disconnect()
sdp_accept()
sdp_ioctl()
sdp_destroy()
sdp_shutdown()
sdp_setsockopt()
sdp_getsockopt()
sdp_sendmsg()
sdp_recvmsg()
sdp_bind()
sdp_backlog_rcv()
sdp_hash()
sdp_unhash()
sdp_get_port()
sdp_name[32]
sdp_stats[NR_CPUS]
Struct Sock
daddr
rcv_saddr
dport
local_port
Sport
Saddr
bound dev if
State
type
Protoocol
family
refcount
shutdown
lock
rcvbuf
SndBuf
recv queue
write_queue
sleep
Error Queue
Prot
TP Info
Local Route
Rcv Lo Wat
Rcv Timeout
Snd Timeout
Protocol Private Data
Socket
User Data
Callbacks
Connect Callback()
Listen Callback()
Event Callback()
destruct()
Private Data
Offload Protocol's Private Data (Transport ops)
ti_ops_t
event_register()
event_deregister()
create_ti()
destroy_ti()
debug_svc()
create_endpoint()
destroy_endpoint()
create_pd()
destroy_pd()
create_cq()
poll_cq()
rearm_cq()
destroy_cq()
connect()
accept()
reject()
listen()
disconnect()
reg_virt_mem()
reg_phys_mem()
dereg_mem()
res_pool_create()
res_pool_destroy()
res_pool_get()
res_pool_put()
msg_send()
msg_recv()
rdma_read()
rdma_write()
atomic_op()
io_ctl()
offload_register_protosw()
offload_protosw_t
Next
type
protocol
ops
prot
addr_list
next registered offload protocol
socket interface
protocol interface
protocol's link addresses
socket.c
AF_INET
AF_INET_OFFLOAD
IPPROTO_SDP
Offload Transport

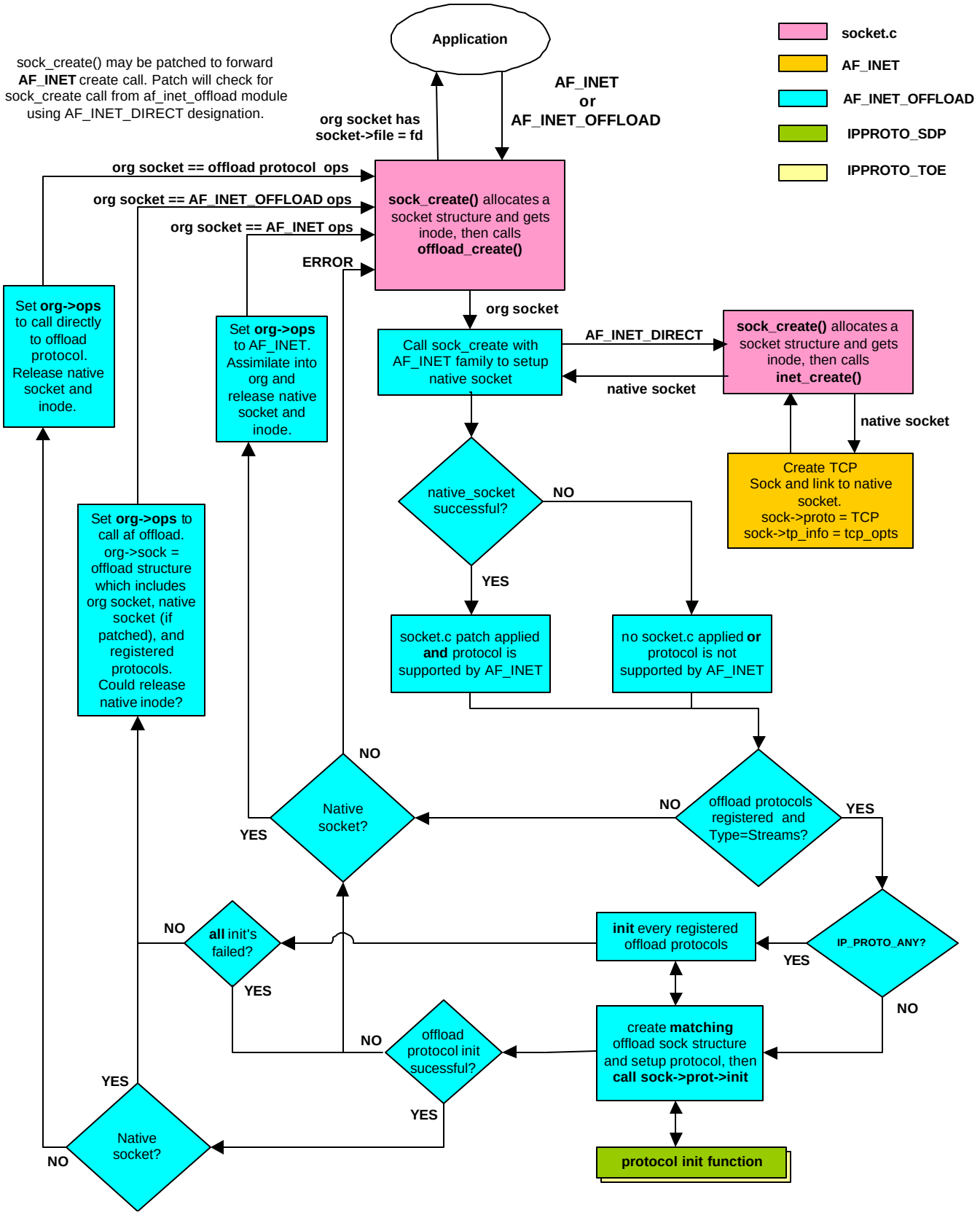
This section attempts to describe the offload data structures and how they relate with the existing Linux `socket` and `sock` data structures. Linux networking stack utilizes the `socket` and `sock` data structures for all socket management functions across the various layers. The `socket` structure is typically created by the socket driver and the `sock` structure is created by the address family driver.

Typically a socket structure has a single sock pointer in it which points to the address family and protocol specific operations on that socket. The primary reason for this is that during socket creation the address family and protocols are well specified which allows the socket to bind itself during socket creation to a particular protocol. However with offload socket framework, to deal with IPADDR_ANY and IPPROTO_ANY conditions where a specific protocol or local endpoint is not specified upfront, there is need for multiple offload protocols to be linked to the same socket until enough information comes (during bind or connect) to bind the socket to a specific protocol. To fulfill this requirement the `socket` structure is transparently modified (by adding more members to the end of the buffer pointed by the `sock` member element). Thus a socket created by calling the OPS module has the `socket->sock` field point to an `offload_sock_t` structure. The `offload_sock_t` structure starts with a sock element followed by other elements. The `offload_sock_t` contains a `offload_list` which points to a linked list of `offload_prot_t` structures each representing a offload protocol module registered with the OPS. The `offload_prot_t` structures keep all the data and state needed by the offload protocols and transports.

Since the sockets created by the OPS can span over multiple protocols (including the AF_INET TCP protocol), a higher level accept queue is maintained at the `offload_sock_t` level. The `offload_sock_t` also points to a 'Native socket' which points to a fully qualified socket that is created using the same credentials as the 'Original socket' but over the standard Linux AF_INET address family.

For specific details of the `offload_sock_t` structure, please refer the accompanying data structure definitions in section 12.

## 5.1.6 Socket create

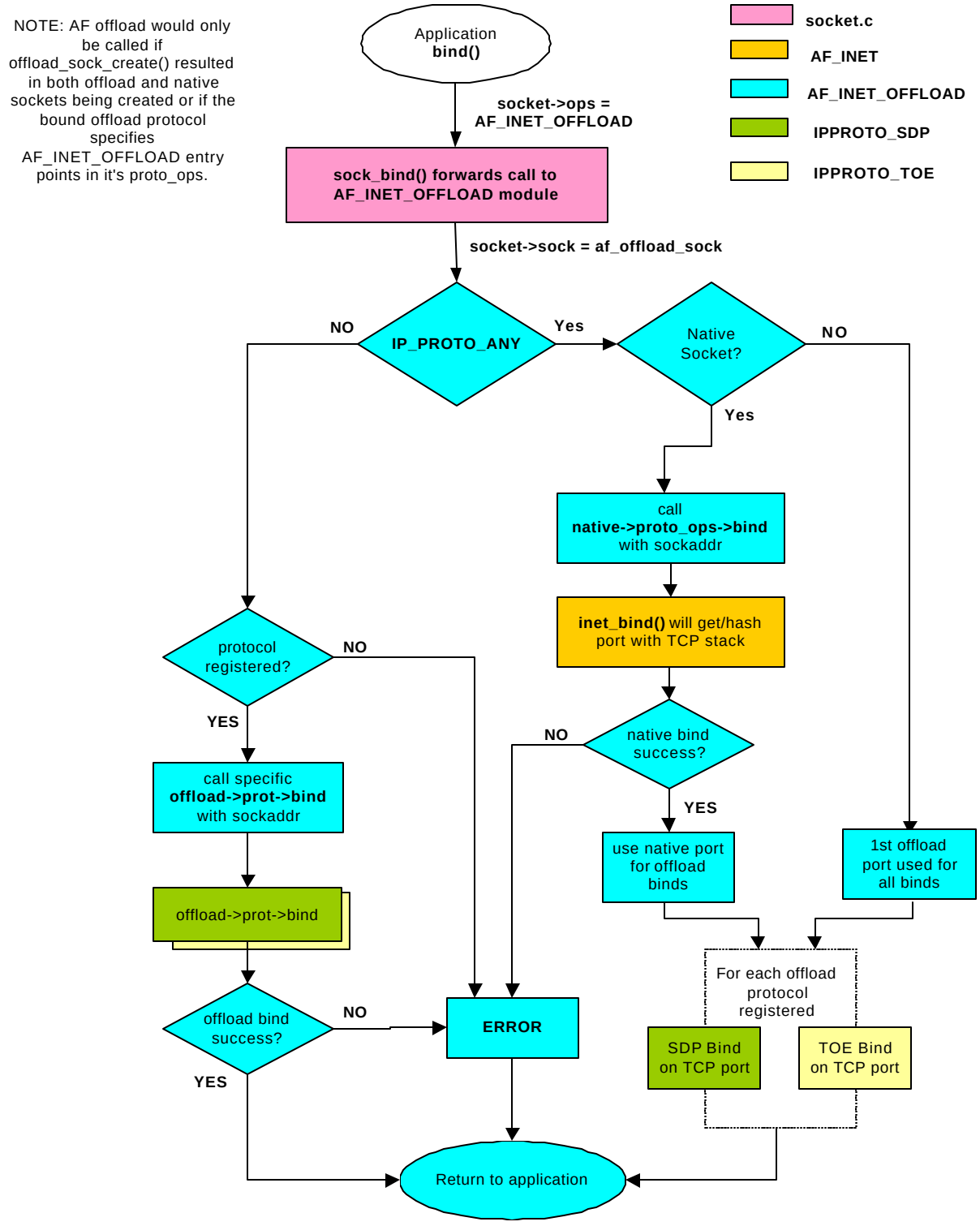


This section illustrates the socket creation process using the offload sockets framework. The socket driver `sock_create()` function creates the 'Original socket' structure and forwards the call to the OPS module by calling its `offload_create()` entry point in the `proto_ops` table of the socket. The OPS module calls back the socket driver's `sock_create()` function with the AF_INET_DIRECT address family to create a 'Native socket' over the AF_INET stack. If the native socket creation was successful, it implies that the socket driver patch is running and the application specified protocol is something supported by the OS AF_INET stack.

OPS checks if the application specified protocol in the `socket()` calls is a offload protocol registered, and also if the type of socket specified by application is STREAM (since OPS only offloads STREAM sockets). If a match is not found, the specified protocol is not offload able and the native socket is copied to the original socket to fulfill the socket creation request. If the native socket was not available, then the socket creation call is failed and returned. If a match is found, OPS checks if the protocol specified is a wildcard (IP_PROTO_ANY). If it is a wildcard protocol, then the socket context initialization function needs to be called on all the registered offload protocol modules one after the other. If a specific protocol is specified by application (like say IPPROTO_SDP or IPPROTO_TCP), then the specific protocol module's socket initialization function is called. If the protocol module initialization fails, then the OPS attempts a fail back to the AF_INET protocol stack if a native socket was available.

Based on the above conditions, the `offload_create()` call from the socket driver could return with either `original_socket->proto_ops` pointing to protocol operation call table of OPS, or protocol operations of a specific offload protocol (such as SDP), or protocol operations of AF_INET, or a NULL indicating error. In any case, this is completely transparent to the socket driver and the application above it.

## 5.1.7 Socket bind

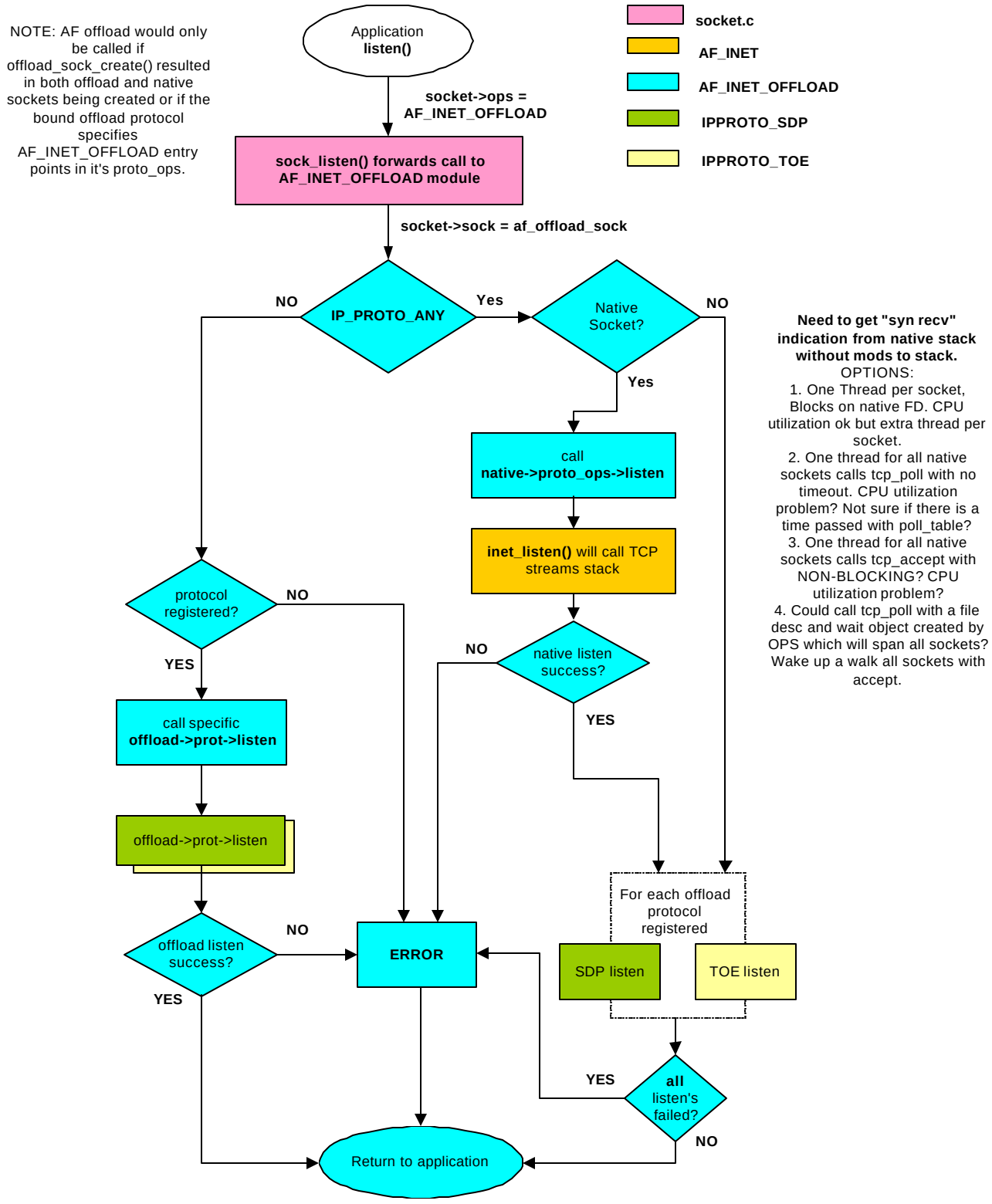


This section illustrates the program flow in the OPS for a socket `bind()` operation. Since the socket creation request was forwarded to the OPS module by the socket driver (as described in the previous section), the `socket->proto_ops` at the time of bind points to the function call table of the AF_INET_OFFLOAD address family as exposed by the OPS driver. The socket driver simply forwards the `bind()` operation to the AF_INET_OFFLOAD address families `bind()` entry point as exposed in the `socket->proto_ops` member.

Once the OPS module's bind entry point gets called by the socket driver, it first checks for if the protocol specified on the socket is a wildcard (IP_PROTO_ANY) protocol. If it is not a wildcard protocol, the bind entry point of the appropriate offload protocol module that supports the specified protocol is called. If the protocol modules bind fails, an error is returned.

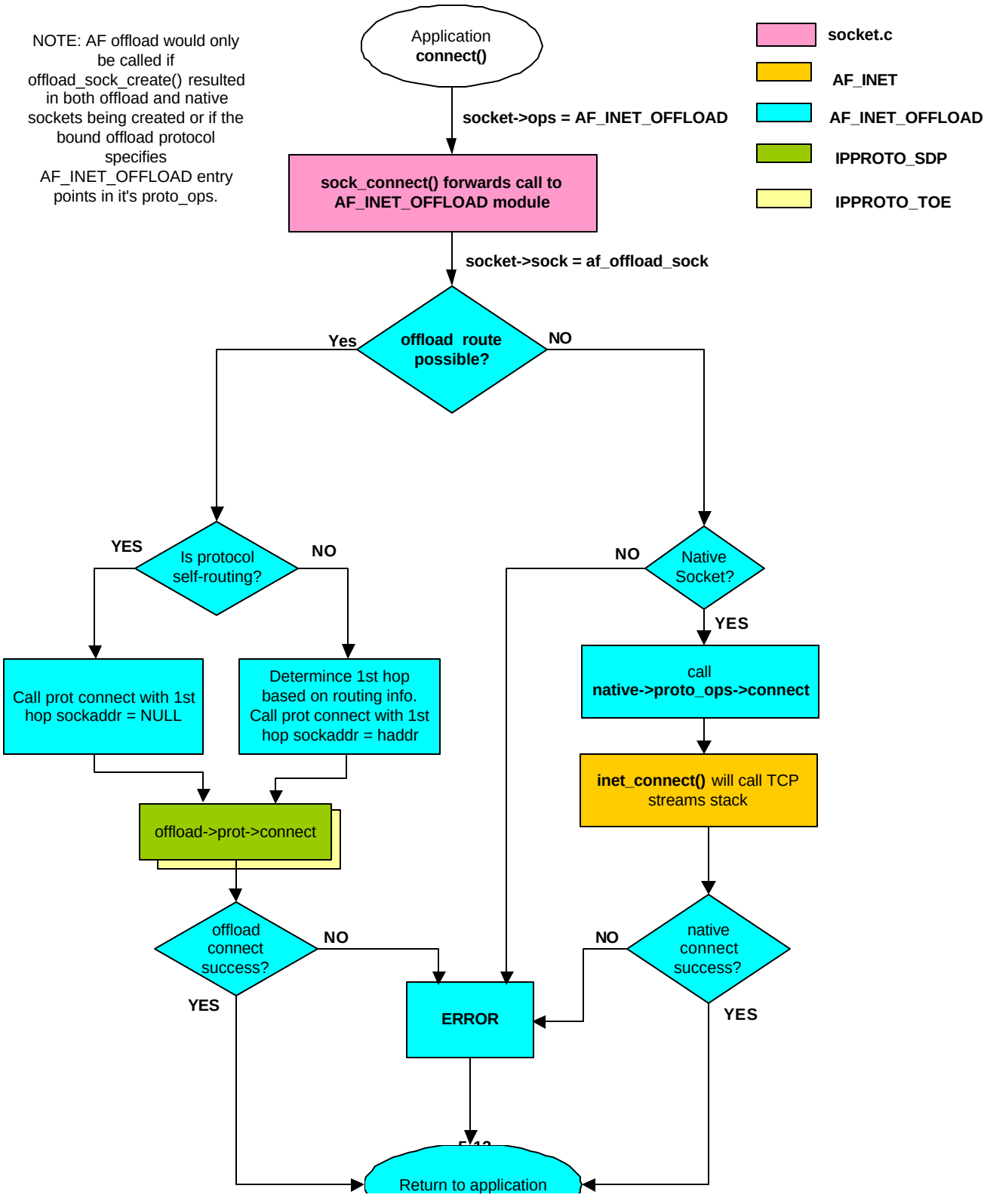
If a wildcard protocol was specified during the 'Original socket' creation (as explained in previous section), the OPS attempts to make sure the same bind port is used for all the protocol modules. If a bind port (well known port) is specified by the application, the specified port value is used on binds to all the registered protocol modules. If a dynamic port is specified by the application (port = 0), then a dynamic bind is done over the first offload protocol, and the port value returned is used to do the binds on rest of the protocols. Also, in case of wildcard socket binds, the 'Original socket' is first checked to see if a native socket handle (bound over the AF_INET stack) exists within the 'Original socket'. If a native socket is not present, the bind is first attempted over the 'Native socket' (causing a TCP bind). If the 'Native socket' bind returns in error, the bind fails immediately. If the 'Native bind' succeeds, then the binds on all other offload protocol modules are attempted. This guarantees that for dynamic ports, the port value returned by TCP can be used for binds on all other protocols, thereby unifying the port space for the application.

## 5.1.8 Socket listen



OPS handles socket listen very similar to how bind is handled. Listens on sockets that are bound to a specific protocol are forwarded to the protocol module handing the specific protocol. For wildcard protocol bound sockets, the listens are posted on all the available protocol modules bound to the socket, including TCP if a 'Native socket' is attached to the 'Original socket'. Since TCP interface uses blocking semantics for operations such as accept(), poll() etc., the OPS module utilizes a worker thread to process these operations. Also, a global accept queue is constructed at the OPS module to process incoming connection requests through any of the protocol modules, including TCP. The accept/poll section describes details on these operations.

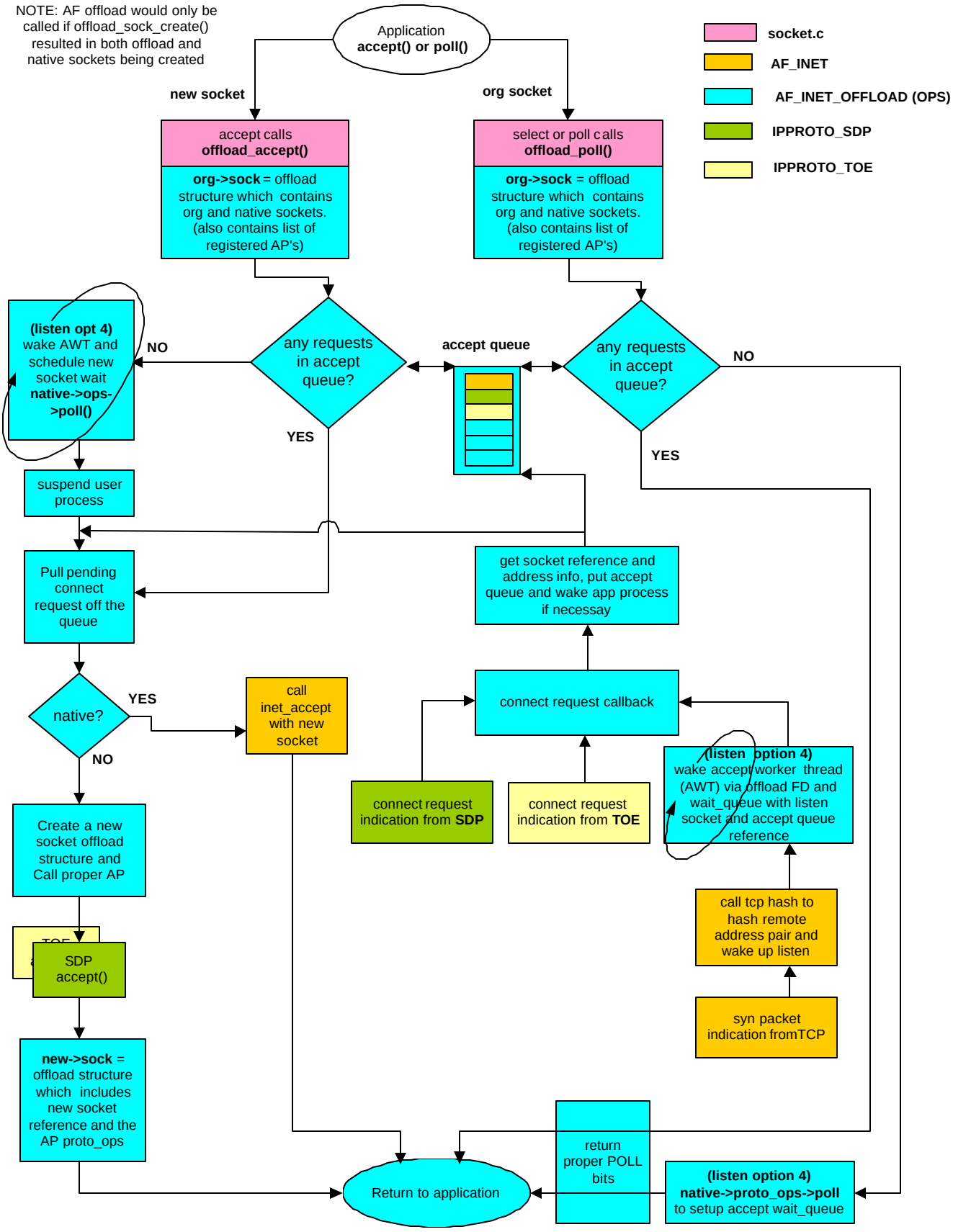
## 5.1.9 Socket connect



This section illustrates the program flow in the OPS for a socket `connect()` operation. Since the socket creation request was forwarded to the OPS module by the socket driver simply forwards the `connect()` operation to the AF_INET_OFFLOAD address families `connect()` entry point as exposed in the `socket->proto_ops` member.

Once the OPS module's connect entry point gets called by the socket driver, it checks it's current routing table and based on the sockaddr passed with the connect, routes to the proper offload protocol. If no routes are found via registered offload protocol providers then the OPS module will forward the connect request to the native stack. Otherwise, it will check to see if the protocol provider needs first hop information and will pass the appropriate 1st hop and final destination address information via the protocols connect entry point. Any errors returned by the offload protocol stack or by the native protocol stack will be returned to the application.

## 5.1.10 Socket accept and poll on a listen FD



This section illustrates the program flow in the OPS for a socket `accept()and poll()` operation. Since the socket creation request was forwarded to the OPS module by the socket driver, the `socket->proto_ops` simply forwards the `accept()or poll()` operation to the AF_INET_OFFLOAD address families `accept()or poll()` member respectively.

The OPS poll entry point is called as a result of either a select or poll at the application level. The kernel do_select is called to handle any timeout or multiple listen FD's specified by the application. This diagram covers the details of listen FD's only. The OPS will check for listen state and for any pending requests on the accept queue. If there are accept requests pending, it will return the proper POLL bits to the application to indicate the pending connection. Otherwise, it will wake up the Accept Working Thread (AWT) to setup the native accept wait_queue to process native connections (SYN). Any errors returned by the native protocol stack will be returned to the application.

The OPS accept entry point is called as a result of a application accept on a listen FD after a poll or select has indicated a connection request event on a previous listen. The OPS will check for listen state and for any pending requests on the accept queue. If there are accept requests pending, it will pull the accept information off of the accept queue. Otherwise, it will wake up the Accept Working Thread (AWT) to setup the native accept wait_queue and will suspend the user process.

If the accept request is a native request, then the native accept is called with the new socket. Otherwise, if the accept request is an offload protocol request, the OPS will create a new socket offload structure and call the appropriate offload protocol accept. Once the offload protocol `accept()` returns, the OPS will setup the proto_ops and the new socket reference and return to the application. Any errors returned by the offload protocol stack or by the native protocol stack will be returned to the application. .

### 5.1.11 Socket operations on a connected FD

Once a socket transitions into a connected state, it is also bound to a specific protocol that fulfilled the connection setup. Once a socket is connected the operations on this socket are forwarded blindly to the protocol module it is bound to through the proto operations exposed by the protocol modules. The OPS module could also try to remove itself from the function call path of a connected socket by pointing the `proto_ops` member of the socket structure to directly point to the `offload_proto_ops_t` structure of the bound protocol underneath at the time of connection setup completion. Any data transfer operations (send/recv/poll/select) are handed exactly same as how Linux stack handles them currently. If the socket is connected through the Linux AF_INET TCP stack, the OPS module makes the original socket resemble a native socket with the `proto_ops` structures pointing to the AF_INET/TCP protocol operation call tables. With this model, the socket driver and application above it always sees a transparent interface.

# 5.2 Sockets Direct Protocol Module

The Linux implementation of SDP includes many interdependent components that are referred to as Offload Sockets Framework (OSF). This framework will allow applications to bypass the resident TCP/IP protocol stack while using the default address family of AF_INET.

Offload Sockets Framework includes an Offload Protocol Switch (OPS) module (Ipv4 internet protocol switch) that allows integration of Offload Protocol (OP) modules along side the existing TCP/IP stack, an OP module that supports SDP on the wire, and an InfiniBand Offload Transport (OT) that provides the transport abstraction to an InfiniBand fabric.

There are 3 major internal components of the SDP module, socket interface layer at the top, the SDP state machine in the middle, and the transport interface at the bottom. The SDP module exposes the standard `proto_ops` call interface at top. In addition to the standard `proto_ops` call interface at the top, SDP also supports the new extended proto operations, `offload_ops`, as defined by the OPS interface. These new interfaces include a new connect call that supports a first hop IP address and a new sock structure that defines specific offload protocol and transport details. This section covers the design details of the Socket Direct Protocol Module.

## 5.2.1 Linux Kernel Modifications

The SDP module will be coded as a new Linux AF_INET network protocol type. This requires a new definition, `IPPROTO_SDP`, in the kernel linux/include/linux/in.h include file. In addition to the new SDP protocol type, the sock structure in linux/include/net/sock.h will need to be modified to include a new `sdp_opt` structure qualified with "`#if defined (CONFIG_SDP)`"

## 5.2.2 Initialization

The SDP kernel module will be demand loaded after the initialization of OPS, normally during system initialization. The SDP module will have a dependency on OPS. Offload transports will have a dependency on SDP.

Actual SDP initialization is performed in accordance with the standard Linux kernel module load procedure; the '`init_module()`' function is called once the SDP module has been successfully loaded into the kernel. See section 7.1 for SDP initialization process and interaction with the OPS and the OT.

## 5.2.3 Shutdown

SDP shutdown can be initiated in one of two ways:

- During normal system shutdown procedures.
- The SDP module is forcibly removed using the Linux system administration command 'rmmod' (remove module).

With either method, the SDP module unload function "`cleanup_module()`" is called per the standard Linux kernel module unload procedure.

The SDP "`cleanup_module()`" function will notify the OPS and the offload transports of the pending SDP module shutdown. It is expected that SDP will shutdown by releasing allocated system resources,

halting and unloading itself. This will result in a cleanup of all INET addresses in use by the OPSI on behalf of the SDP module. See section 7.1 for SDP shutdown process and interaction with the OPS and the OTI.

### 5.2.4 Buffer Strategy

SDP supports several data transfer mechanisms designed to take advantage of reliable transports with RDMA capabilities. Four types of transfer modes are supported with SDP; buffered, combined, write zero copy, and read zero copy. Depending on the application workload and buffer availability, SDP will use the most optimal mode for transferring the data.

Small transfers, determined by a data copy size threshold, will use buffered mode to transfer the data. This mode requires a pre-allocated buffer pool that the application data is copied to/from during each data transfer. This buffer pool is provided by the offload transports but will be sized and managed by SDP on a per connection basis. The transport must provide a mechanism to created/destroy these pools with buffer size, count, and data segments. A free pool queue will be provided by the transport, with a low overhead get/put interface, to retrieve and replace buffer resources. The transport must also provide interfaces, to send buffers, to pre-post receive buffers, and to poll or get callback indications when buffer transfers are complete.

Large transfers, greater then the data copy size threshold, that have available read and write buffers on each end of the connection will use the zero copy read or write mechanism to transfer the data. This zero copy data transfer mode requires a mechanism to initialize RDMA operations with the transport. The transport must provide a mechanism to pre-allocate RDMA transport descriptors that can be used for the data operation to avoid unnecessary overhead in the transfer mode. The transport must also provide interfaces to register and un-register user memory, to initiate RDMA reads/writes, and to poll or get callback indications when transfers are complete.

Figure 2 illustrates the buffered mode and the zero copy paths using a reliable transport:

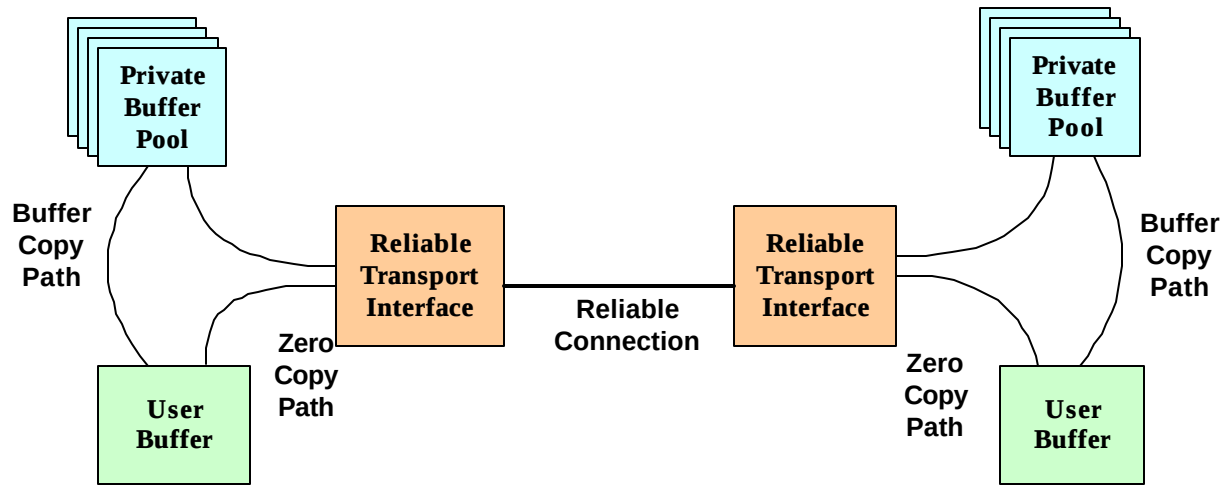



**Figure 2 SDP Buffer Mode Overview**

Refer to the Data Transfer Mechanism section in the InfiniBand annex A4 Socket Direct Protocol Specification for more details.

### 5.2.5 Connection Services

Sockets Direct wire protocol defines a standard message protocol that includes a Base Socket Direct Header (BSDH) with all messages. Each message is identified with a Message Identifier (MID) and in addition to BSDH may contain specific message information and actual upper level protocol (ULP) data. For connection establishment the Hello Message (HM) and HelloAck Message are used and to disconnect or abort the DisConn and the AbortConn messages are used. Refer to the SDP Message Formats section in the InfiniBand annex A4 Socket Direct Protocol Specification for more details.

The SDP module is totally transport independent and expects the transport to provide the address resolution and port mapping based on its underlying link. The offload transport interface will provide a mechanism to resolve the endpoints based on a standard IP address/port pairs and return an opaque endpoint data handle back to SDP that can be used for either the listen or connect request.

The transport is also required to provide a mechanism to pass private data (up to 76 bytes) with a connection, listen, accept, and reject calls. These private data buffers are used by SDP to setup connection attributes as defined by the Hello Message (HM) and the HelloAck Message (HAH). The connection reject codes (consumer reject) are exported to the SDP module as specified by the InfiniBand annex A4 Socket Direct Protocol Specification.

InfiniBand's IP over IB mapping and CM REQ does not provide enough granularity to support a listen that multiplexes based on a link address, network address, and a port. It only provides a mechanism to listen on the link address (GID) and the service identifier (mapped IP port). Therefore, the transport interface must supply additional multiplexing fields in the listen call that includes an offset to private data and a size of private data so that SDP can provide the local IP address as the additional multiplexing field. Without this feature, SDP would be required to provide additional multiplexing on top of the listen if there was a GID/SID conflict due to IP aliasing features of an O/S.

### 5.2.6 Completion Model

The transport provides a completion queue model that allows the SDP module to either poll for completions or to get an indication of a completion via a callback. It is assumed that the underlying transport and link interface provides the best possible de-serialization and scale-up with the completion callbacks based on the hardware characteristics and workloads running on the system. The SDP module will be capable of running at any priority including interrupt level.

SDP will initialize the CQ and size it according to the maximum work requests expected for sends, receives, and RDMA transfers based on the initial socket creation and socket option settings. The SDP module will setup one CQ for send messages and RDMA requests, and another one for receive messages. This architecture allows for development of independent SDP send and receive completion engines that can work independently.

### 5.2.7 Data Transfer Models

SDP defines four data transfer mechanisms:

- Bcopy – transfer of user data from send buffers into receive private buffers.

- Read Zcopy – transfer of user data through RDMA reads, preferably directly to and from user buffers.
- Write Zcopy – transfer of user data through RDMA writes, preferably directly to and from user buffers.
- Transaction – an optimized user data transfer for transactions that piggy-backs user data transfers using private buffers on the top of the Write Zcopy mechanism used to transfer the data on the opposite half-connection.

The SDP module will be compliant with all data transfer modes as specified by the InfiniBand annex A4 Socket Direct Protocol Specification.

**NOTE**

All models will be supported at the final release of this product, however each mode will be implemented in phases in the following order: Bcopy, Read Zcopy and Write Zcopy, and finally transaction. Since Bcopy is the only required model to be compliant with the specification all phases will be compliant with SDP protocol.

### 5.2.8 Locking and Threading Model

The SDP thread safe design is based on the premise that the SDP module will be driven by external events and thus will not require any dedicated threads of control, which are exclusive to the SDP module. External agents, such as an user application thread, an OPSI initialization thread, or system event notification will make calls into the SDP module thus providing threads of control. Access to internal SDP data structures will be serialized by the use of simple locks.

The offload transport may have dedicated threads of control that can invoke up-calls into SDP module. One such case would be the invocation of an IO completion callback routine. The SDP module is designed to limit the amount of processing in the IO completion callback thread if the transport is calling in a high-priority thread context such as interrupt level. All up-calls will be checked for priority levels and processed accordingly.

# 5.3 InfiniBand Offload Transport Module

The InfiniBand Offload Transport is designed to abstract reliable hardware transport interface specifics in such a fashion that an offload protocol (OP) module, Sockets Direct Protocol (SDP) in our case, can interface to a consistent transport API over a potentially wide range of differing reliable hardware transports. The InfiniBand Transport module will support any hardware interface that utilizes the InfiniBand Access Layer (IAL).

The InfiniBand OT has at least one Linux network link device associated with it, represented by the Linux `struct netdevice`. Once the InfiniBand OT registers with the SDP module, it will provide link device configuration events to the SDP module. In this manner, Internet link device configuration changes (IP address assignment, address change or device shutdown) and IP network route change events will be propagated to the offload SDP agent.

InfiniBand OT service's defines the Internet Protocol Address format as its base level addressing mechanism. Offload protocol modules establish connections or send/receive datagrams based on IPv4 or IPv6 addressing formats. Initially, only the IPv4 address format will be supported. Offload transports are expected to convert an IP address into the transport/link specific address format required for transmission or reception. The InfiniBand OT module is designed to abstract the InfiniBand transport/link into common OT definitions.

The InfiniBand OT module provides four groups of basic services. Each service includes a collection of OT components that provide an implementation instance of that service. That is to say, each component has a well-defined offload transport service interface with a behavior implemented by an offload transport.

Figure 3 shows the OTI components that this specification defines.

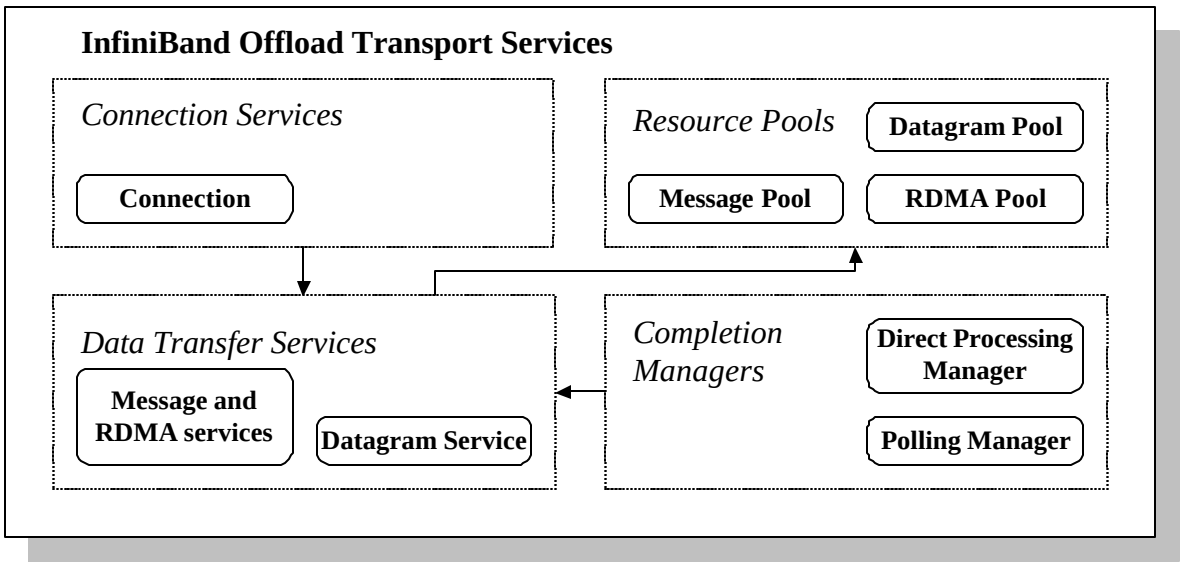



**Figure 3 InfiniBand offload Transport Service Components**

The transport architecture is flexible enough to allow the creation of additional components for each type of service within the transport. However, the interfaces between internal services and components not shown in Figure 3 are outside the scope of this document.

### 5.3.1 Linux Kernel Modifications

The InfiniBand OT is designed as a loadable kernel module. Modifications to provide notifications of route changes via events will require changes to the Linux kernel.

### 5.3.2 Initialization

The InfiniBand OT kernel module will be demand loaded after the discovery of offload capable hardware, normally during system initialization. Actual InfiniBand OT initialization is performed in accordance with the standard Linux kernel module load procedure; the 'init_module()' function is called once the InfiniBand OT module has been successfully loaded into the kernel.

### 5.3.3 Shutdown

The InfiniBand OT shutdown can be initiated in one of two ways:

- A network link device, which the InfiniBand OT is monitoring, is shutdown via normal system shutdown procedures. System shutdown results in a link device event notification (system shutdown) being delivered to the InfiniBand OT.
- The InfiniBand OT module is forcibly removed using the Linux system administration command 'rmmod' (remove module).

With either method, the InfiniBand OT module unload function 'cleanup_module()' is called per the standard Linux kernel module unload procedure.

The InfiniBand OT 'cleanup_module()' function will notify the SDP module. The SDP module will notify the OPSI (Offload Protocol Switching Interface) of the interface shutdown, which results in the shutdown of INET addresses in use by the OPSI.

### 5.3.4 Connection Services

Connection services abstract and simplify the details of using a specific connection protocol. A connection service is responsible for creating data transfer service's, authenticating their data transfers with remote endpoints, and presenting configured data transfer services to the user. The library provides a single channel connection service that is used to establish point-to-point communication over the transport link medium to a remote transport agent or endpoint.

Communication endpoints are identified by Internet Protocol addresses. Initially only IPv4 (Internet Protocol Version 4) format addresses are supported, with provisions to support IPv6 into the near future.

### 5.3.5 Buffer Strategy

Resource pools manage the allocation and distribution of data buffers, work requests, and transfer elements. Transfer elements are used to specify data transfer operations to the library. Work requests are required internally by the library to perform data transfers over the fabric interconnection hardware. There are three types of resource pools: datagram pools, message pools, and Remote Direct Memory Access (RDMA) pools.

#### 5.3.5.1 Datagram Buffer Pool

The datagram pool administers datagram elements; queues pending requests for asynchronous operation; and, at the user's option, allocates and manages the distribution of data buffers used for connectionless message-passing operations. Datagram elements are used to request data transfers using connectionless communication and support Segmentation and Reassembly (SAR) of connectionless messages.

#### 5.3.5.2 Message Buffer Pool

The message pool is similar to the datagram pool, but message elements are used to perform data transfers over connected channels. The message pool administers message elements; queues pending requests for asynchronous operation; and, at the user's option, allocates and manages the distribution of data buffers used for message-passing operations.

#### 5.3.5.3 RDMA Resource Pool

The RDMA pool manages data transfer request elements and, optionally, RDMA data buffers. The data transfer request elements are used to signal the library to perform RDMA operations over connected channels. The RDMA pool provides asynchronous operation for requesting resources.

### 5.3.6 Data Transfer Services

Data transfer services are responsible for posting the following types of requests: sends, receives, RDMA reads, and RDMA writes. These services provide work queue management, perform message-passing flow control and manage scatter-gather data transfer requests. Connection services are used to create and configure data transfer services.

#### 5.3.6.1 Message and RDMA Service

A channel provides a connected communication path between two transport endpoints. Channels support both RDMA and message-passing data transfers and may optionally provide message-level flow control.

#### 5.3.6.2 Datagram Service

A datagram service provides connectionless communication to remote endpoints. Multiple endpoints may be reached through a single datagram service; however, datagram services support only connectionless message data transfers.

### 5.3.7 Completion Services

Completion services check for and process completed data transfer operations. Completion services are responsible for invoking any necessary post-processing, including user callbacks, and for resuming stalled data transfers.

#### 5.3.7.1 Direct Processing Manager

The direct processing manager processes completions of data transfer requests within the context of the completion callback that delivered the completion notification to the library. This provides the lowest possible latency for processing a single completion by avoiding a thread context switch.

#### 5.3.7.2 Polling Manager

A polling manager is a completion service without active threads. A polling manager manages completions across multiple data transfer services, but it relies on the user to poll for completed requests.

### 5.3.8 Locking and Threading Model

The InfiniBand OT thread safe design is based on the premise that it will be driven by external events and thus will not require any dedicated threads of control. External agents, such as the SDP module, the InfiniBand Access Layer, or system event notification will make calls into the InfiniBand OT thus providing threads of control through the InfiniBand OT. Access to internal InfiniBand OT data structures will be serialized by the use of simple locks.

# 6. System Resource Usage

## 6.1 Sockets Direct Protocol Module

Memory requirements are dependent on the socket allocation size. Private buffers will be allocated based on the socket buffer size as specified by either the default O/S setting or by the setsockopt call from the application. Additional overhead on a per socket basis includes the `struct offload_ops_t` (~128bytes).

## 6.2 InfiniBand Transport Module

The IB Offload Transport will consume additional memory on a per socket basis based on the connection attributes (maximum send and receive depths). Refer to the IB HCA hardware for CQ and QP overhead.

# 7. Internal Compatibility

## 7.1 Interaction with Other Components

### 7.1.1 Socket Driver and OPS Module Interaction

The Offload Protocol Switch (OPS) module provides a dynamic binding facility for offload protocols modules. It is a loadable module that registers dynamically with the Linux socket driver (socket.c). It exposes a new address family (AF_INET_OFFLOAD) but at the same time interacts with the AF_INET address family so that IPPROTO_ANY traffic can be directed to both the offload protocols under AF_INET_OFFLOAD and to the standard AF_INET protocols. Applications can also directly create sockets on AF_INET_OFFLOAD address family and bind to any offload protocols registered with this offload address family, without requiring any kernel patches.

In order to seamlessly support sockets created by applications with AF_INET address family a small patch must be applied to the socket driver (socket.c) sock_create() code to direct AF_INET address traffic to the offload address family module (AF_INET_OFFLOAD) exposed by the OPS module. The OPS module will switch sockets appropriately based on family, protocol type, and protocol. No modifications are needed in the AF_INET stack since the OPS module will interact with the standard AF_INET stack via the address family socket interface. Figure 3 shows the original Linux socket driver address family switching logic and the changes in the proposed kernel patch.

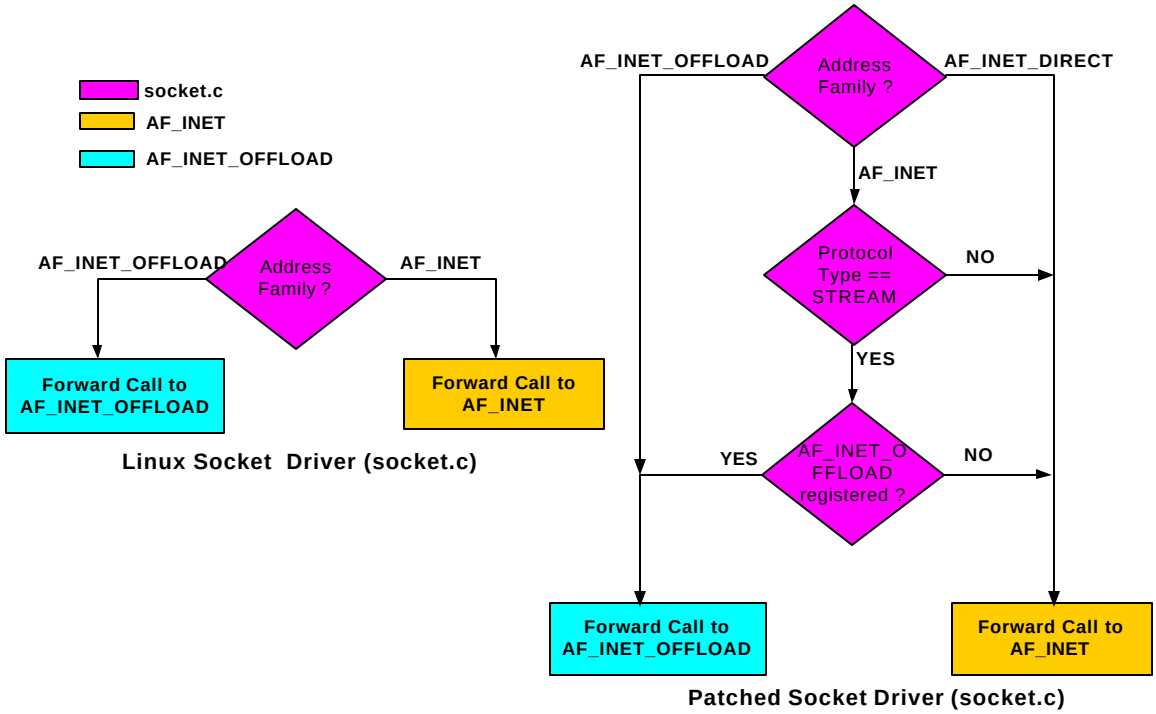



**Figure 4 Proposed Patch to Linux Socket Driver**

### 7.1.2 OPS and SDP Module Interaction

Figure 5 shows the various steps involved in the OPS and offload protocol modules initialization and operation

1. The socket.c sock_create() code is modified to forward all AF_INET sock_create calls to AF_INET_OFFLOAD module. The AF_INET_OFFLOAD module will direct the create call based on the offload protocols that have registered. If the create is forwarded back to the AF_INET, the OPS will use AF_INET_DIRECT so that the sock_create() code will know to switch the create directly to the AF_INET path thus sending all subsequent socket calls directly to the correct family.
2. The AF_INET_OFFLOAD module, as part of its initialization, registers its address family AF_INET_OFFLOAD to the socket layer by calling sock_register () call back to the socket layer. All future socket calls, with address family AF_INET_OFFLOAD will be directed by the socket layer to the OPS layer.
3. When Offload Protocol modules get loaded, the offload_register_protosw() is called to register their entry points and capabilities like self-routing, rdma etc to the AF_INET_OFFLOAD module.
4. When a network interface configuration changes (address assignment, address change, route changes, etc.), the Offload protocol module which is bound to this network interface notifies the OPS module by calling offload_notification().
5. Depending on the incoming request, the OPS module will then switch to proper protocol module. The switching policy depends on the protocol capabilities, binding priority set by the user. The AF_INET stack is the default stack, and if none of the offload protocol modules are loaded or if none of the module capabilities matches the incoming request, the OPS will forward the request to the AF_INET stack. For protocol modules that do not support self-routing, the AF_INET_OFFLOAD driver will handle the routing issues.

Once an appropriate protocol module is chosen, it is up to the protocol stack to handle the request. For example if SDP protocol module is chosen to service an request, then it is up to the SDP module to establish a session with its peer, pre register buffer element and decide on mode of data transfer (Send vs. RDMA Write for example).

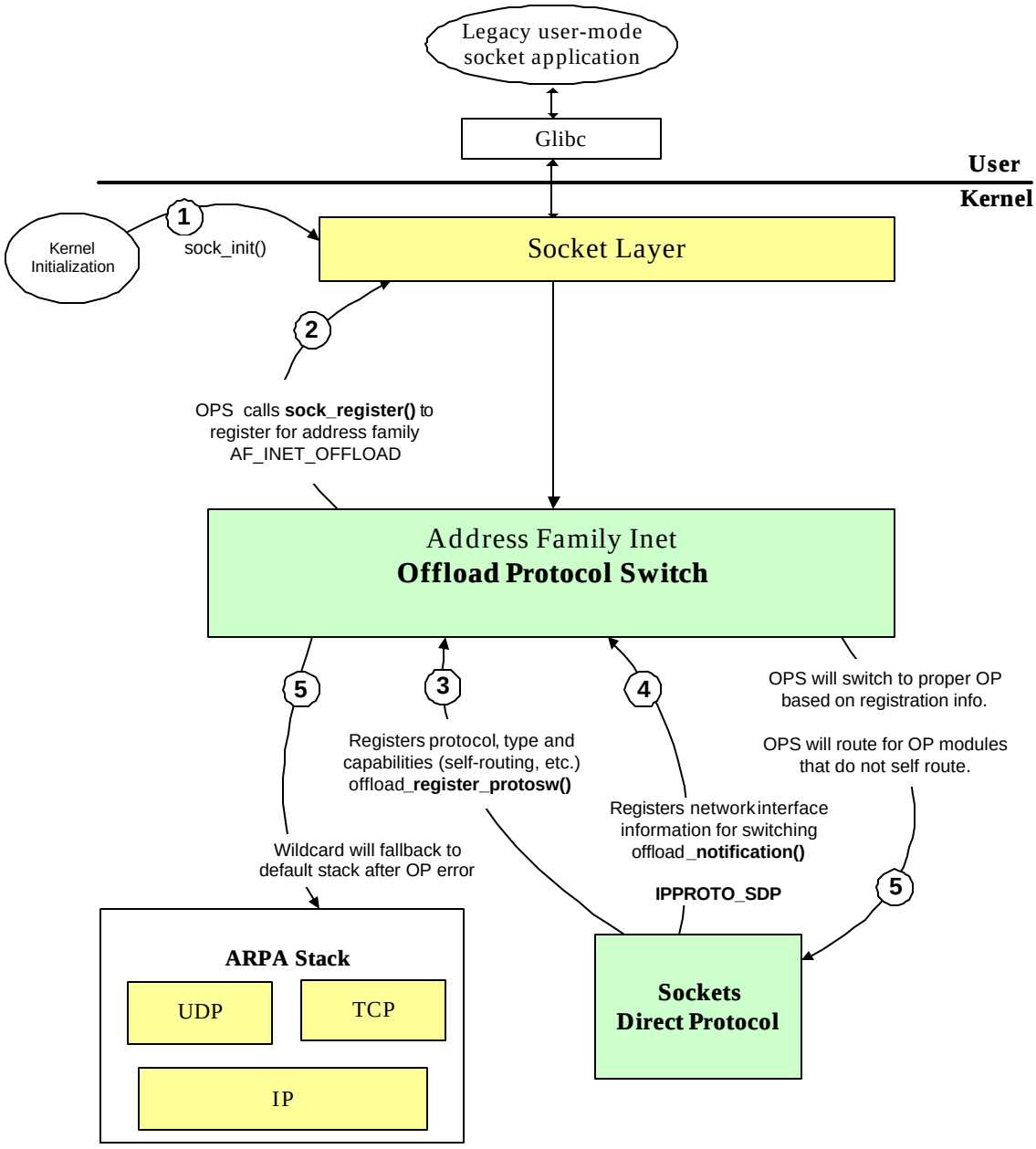



**Figure 5 OPS and SDP interaction**

## 7.1.3 SDP and InfiniBand Transport Interaction

Offload Transports (OTs) implement the offload transport interface abstractions. The InfiniBand transport is designed to be a kernel loadable module, which provides an InfiniBand hardware transport specific implementation of the offload transport abstraction defined by the offload transport interface.

Figure 6 shows the various steps involved in the OT module initialization and operation.

1. The InfiniBand (IB) Transport module will register with the SDP module and provide the transport operations interface and capabilities.
2. The InfiniBand Transport module obtains kernel notification services for network devices, IP configuration, and IP routing information using the device name provided by the link driver. When an IP configuration notification occurs for this net device the transport module will forward the via the event notification upcall to SDP, if SDP has registered for events.
3. The SDP module, using the transport register event call, will register for address, net device, and route changes.
4. The SDP module maintains a list of transports, with address information, and will switch to appropriate transport based on address information during a socket bind.

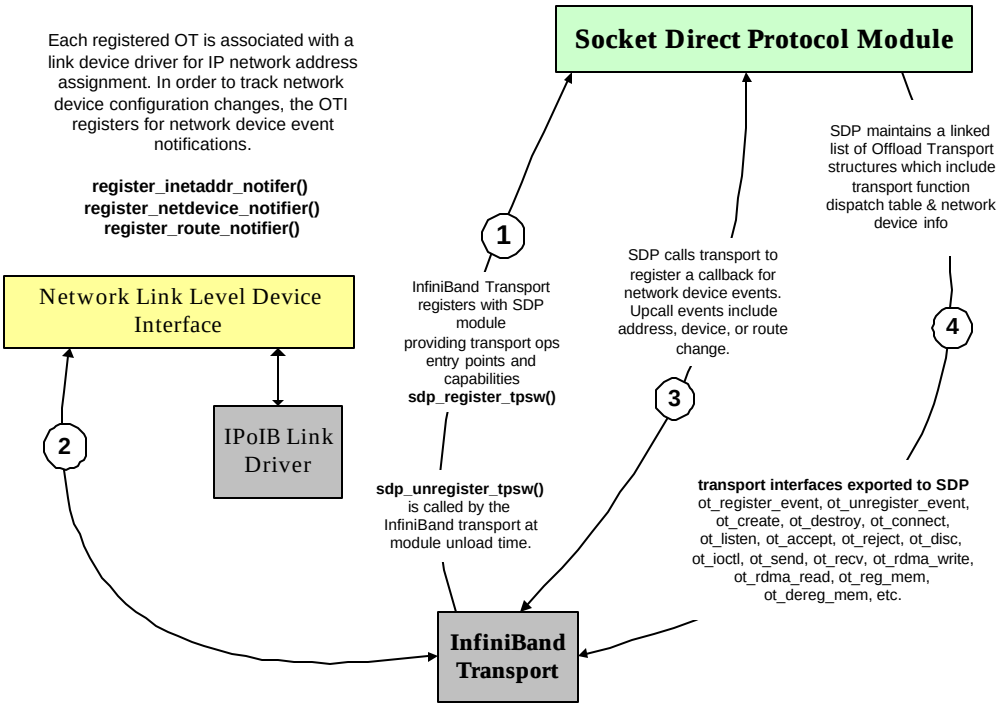



**Figure 6 SDP and InfiniBand Transport**

# 8. External Compatibility

## 8.1 Standards

### 8.1.1 Sockets Direct Protocol Module

The SDP module and state machine is designed to support all compliance statements highlighted in the InfiniBand Annex A4 Socket Direct Protocol Specification.

### 8.1.2 InfiniBand Offload Transport Module

The InfiniBand Transport module is designed to support the new Offload Transport Interface (OTI) at the top and interface with the standard InfiniBand Access Layer at the bottom.

# 9. Other Dependencies

## 9.1 Offload Protocol Switch Module

## 9.2 Sockets Direct Protocol Module

SDP has the following dependencies on an IB specific Offload Transport:

- Offload Transport must provide a listen interface that supports all 76 bytes of IB CM REQ private data and will multiplex based on the IP address/port pair provided in the endpoint handle and the specified private data. The transport shall not byte swap any of the private data. This private data will include, in big-indian format, the SDP Hello and the HelloAck messages.
- Offload Transport must support a ”consumer reject” error code during connection callbacks and reject calls. IB transports are expected to map this consumer reject code to IB code 28.
- Connect shall be non-blocking.
- Calls that may block include accept, create_endpoint, res_pool_get,
- Accept and Reject calls MUST not be restricted to the connection callback thread.
- IB transports are expected to map the IP port to a SID as detailed in the IBTA “Annex A0 Applications Specific Identifiers” specifications.
- The transport interface cannot abstract the hardware’s R_KEY when registering memory for RDMA transfers. The actual 32-bit R_KEY from the IB interface must be provided to the SDP module. This key will be provided to the remote SDP entity via the SDP SinkAvail and the SrcAvail messages.

## 9.3 InfiniBand Transport Module

No other dependencies at this time.

# 10. Installation and Configuration

## 10.1 Offload Protocol Switch Module

TBD: update after coding.

### 10.1.1 Installation

+

### 10.1.2 Configuration

## 10.2 Sockets Direct Protocol Module

TBD: update after coding.

### 10.2.1 Installing

+

### 10.2.2 Configuring

## 10.3 InfiniBand Transport Module

TBD: update after coding.

### 10.3.1 Installing

+

### 10.3.2 Configuring

# 11. Unresolved Issues

## 11.1 Offload Protocol Switch Module

No unresolved issues at this time.

## 11.2 Sockets Direct Protocol Module

No unresolved issues at this time.

## 11.3 InfiniBand Transport Module

No unresolved issues at this time.

# 12. Data Structures and APIs

## 12.1 Offload Protocol Switch Definitions

To view the data structures and APIs associated with this component, click on here.

## 12.2 SDP Definitions

To view the data structures and APIs associated with this component, click on here

## 12.3 InfiniBand OT Definitions

To view the data structures and APIs associated with this component, click on here