1. Architectural Specification – Offload Sockets Framework and Sockets Direct Protocol (SDP)

1.1 Introduction

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 stream sockets (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.

While 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, SDP facilitates the direct mapping of stream connections to InfiniBand reliable connections (or virtual circuits). The IPoIB specification is currently being created and published by an IETF working group. The IPoIB specification will define the packet level format of IP packets on the IBA fabric, and describe the 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 (and partition) on the host system and these devices are 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.

Sockets Direct Protocol only deals with stream sockets, and if installed in a system, allows bypassing the OS resident TCP stack for stream 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 link drivers. The IPoIB stack has no dependency on the SDP stack; however, the SDP stack depends on IPoIB drivers for local IP assignments and for IP address resolution.

1.2 Requirements for Offload Sockets Framework in Linux

This section lists a set of requirements and goals for offload sockets framework 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 can be applied (socket.c) 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 Asynchronous I/O (AIO) being added to Linux. AIO support 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. Makes sure the added 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 the data copies and using RDMA support in SANs 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.

?? select() should work across AF_INET socket file descriptors (fd) 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 protocol (TCP/IP) stack should be directed to the kernel 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 will use only the OS network stack.

?? Offload sockets framework will support offloading of stream sessions both within local subnet and outside local 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 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. 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.

?? Since the socket extensions defined by the Interconnect Software Consortium (ICSC) in the open group are 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. 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.

1.3 System Structural Overview

Sockets (BSD definition) are the most common API used by applications for accessing the network stack on most modern operating systems (including Linux). Most implementations of Sockets such as in Linux also support File I/O APIs (such as read, write) to operate over sockets. Offload Sockets Framework allow 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.

1.3.1 Existing Sockets Architecture in Linux

Currently, Linux implements the Sockets and associated networking protocol stack as a series of connected layers of software modules, all kernel resident. These layers are initialized and bound to each other during kernel start up. Figure 1 shows Linux networking architecture.

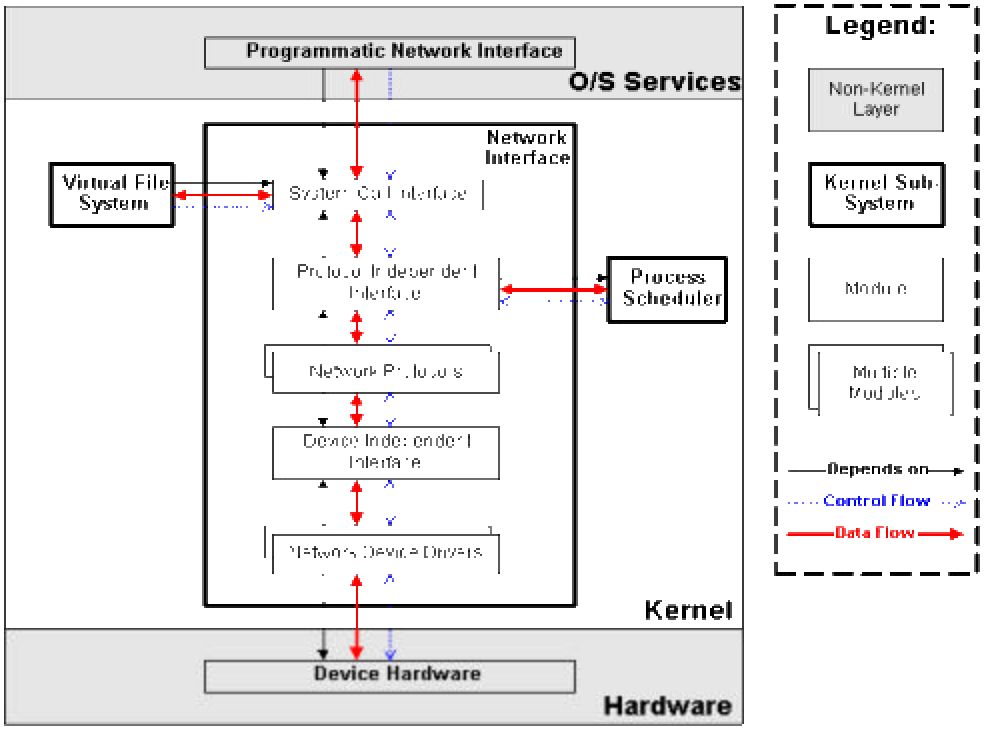

Figure 1 Linux Networking Architecture

Each network object is represented as a socket. Sockets are associated with processes in the same way that i-nodes are associated; sockets can be shared between processes by having its process data structures pointing to the same socket data structure.

Linux also associates a VFS inode and file descriptor, for each socket allocation, to facilitate the normal file operations over the socket handle. At kernel initialization time, the address families built into the kernel register themselves with the BSD socket interface. Later on, as applications create and use BSD sockets, an association is made between the BSD socket and its supporting address family.

The Key features of the existing Linux network architecture are:

1. Network device drivers communicate with the hardware devices. There is one device driver module for each possible hardware device.
2. The device independent interface module provides a consistent view of all of the hardware devices so that higher levels in the subsystem don't need specific knowledge of the hardware in use.
3. The network protocol modules are responsible for implementing each of the possible network transport protocols.
4. The protocol independent interface (BSD Socket Layer) module provides an interface that is independent of hardware devices and network protocol. This is the interface module that is used by other kernel subsystems to access the network without having a dependency on particular protocols or hardware.

1.3.2 Limitations in Existing Linux Sockets Architecture

The current AF_INET sockets architecture in Linux is hard-wired to the in-kernel TCP/IP network stack; making it impossible to hook into the socket API (and file I/O API) calls in user-space without standard C-library code modifications (or load time hooks).

In the current network architecture the protocol layers are tightly coupled to each other and pre-initialized at kernel initialization, making it impossible to load and select between protocol providers that possibly support the same address family (such as AF_INET).

Kernel coding shortcuts do not follow a strict protocol dispatch model within the AF_INET protocol family. Without kernel modifications, introducing new AF_INET transport protocols is impossible without providing a completely new address family.

1.3.3 Evaluations of Alternative Architectures

This section briefly surveys the related work done in this area and explores the solution space for application-transparent high performance I/O architecture (including user-mode I/O) in Linux.

Some of the previous work in this space has traded off transparency for performance. These solutions typically define a custom API that fits the underlying hardware architecture and requires applications to be recoded. An example for this approach is VI architecture that specifies its own VIPL API.

For complete application transparency, the user-mode I/O architecture needs to fit underneath the existing standard socket API. Survey of related work shows the following as some of the possible solutions for transparently supporting high-performance I/O underneath the socket and File I/O APIs:

a) User-mode implementation of high-performance sockets with direct mapping of stream connections to NIC hardware resources from user-mode. This would involve modifying the user-mode library that exports the socket and file I/O APIs (e.g. glibc), such that the socket and file I/O APIs are abstracted out into a separate offload user-mode I/O library that gets demand loaded by glibc. The I/O library will provide a bottom edge interface that can be used by specific protocol libraries (such as one for SDP) to register with it. This solution also requires changes to the socket driver in kernel for supporting operations such as select(), fork(), dup() etc. The user-mode I/O library provides two code paths: If there are no offload user-mode I/O libraries (socket provider for network I/O, file system provider for file I/O) installed, the code is same as it is currently in glibc (i.e. makes a syscall to kernel components). If user-mode providers exist, they are transparently loaded. Another option (mostly cosmetic) is to encapsulate (contain) the standard glibc library within another library. This new library exposes the same set of interfaces as the standard glibc library, and decides if a specific API call needs to be handled by it or forwarded to the standard glibc. While this solution might provide the most optimal performance for speed path operations, the major drawbacks with this approach is that it requires extensive changes to standard C-libraries and kernel socket driver to support direct user-mode mapping of stream sockets transparently to applications.

b) Define a new address family (AF_INET_OFFLOAD) for enabling any offload protocols/transports and modify the socket driver (socket.c) to transparently re-direct AF_INET sockets to AF_INET_OFFLOAD. The major difference between

this option and option (a) is the additional overhead of traps to the kernel (syscalls) in speed path operations. On IA-32 architecture, the syscall overhead was measured at 805 nanoseconds on a 933 MHz processor; IA-64 performance is sub 947 nanoseconds, which implies the syscall overhead to trap to kernel is not the most significant component in today's TCP/IP stack code path. This solution restricts the kernel modifications to the socket driver to enable application transparency by re-directing sockets created under AF_INET address family to the AF_INET_OFFLOAD address family., and provides better modularity to fit into the Linux OS.

c) Changing the Linux loader to transparently hook into the socket and file I/O calls from user-mode. This requires modifying the standard Linux loader to dynamically load the user-mode I/O library into an application's address space and mangle the in-memory function tables (created when glibc was loaded) so that the socket and file I/O functions point to the equivalent functions in the new library (method often used by debuggers). The dynamic loading and function pointer mangling can be done at the time of loading the glibc library, or could be delayed until the first socket or file I/O API call makes the syscall and enters the kernel. Some of the related work shows this solution has been successfully applied for transparent performance analysis of libraries. However, the biggest drawback with this solution is that it requires changes to the loader, and makes it harder to debug.

d) Use the call intercept methods employed in strace(1) and ltrace(1) to modify the behavior of network and file I/O calls transparently to application. This approach is very similar to solution(c) and suffers the same drawbacks.

The current implementation of sockets stack in Linux does not leave room for adding high-performance network I/O support easily. Similar limitation also exists in the file system stack. Based on initial research, it appears that the kernel-mode solution (explained as option (b) above) offers almost the same performance benefits as the user-mode solution, with far less complexity and better architectural modularity to fit into the Linux OS.

1.3.4 Software Component View

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 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 2.

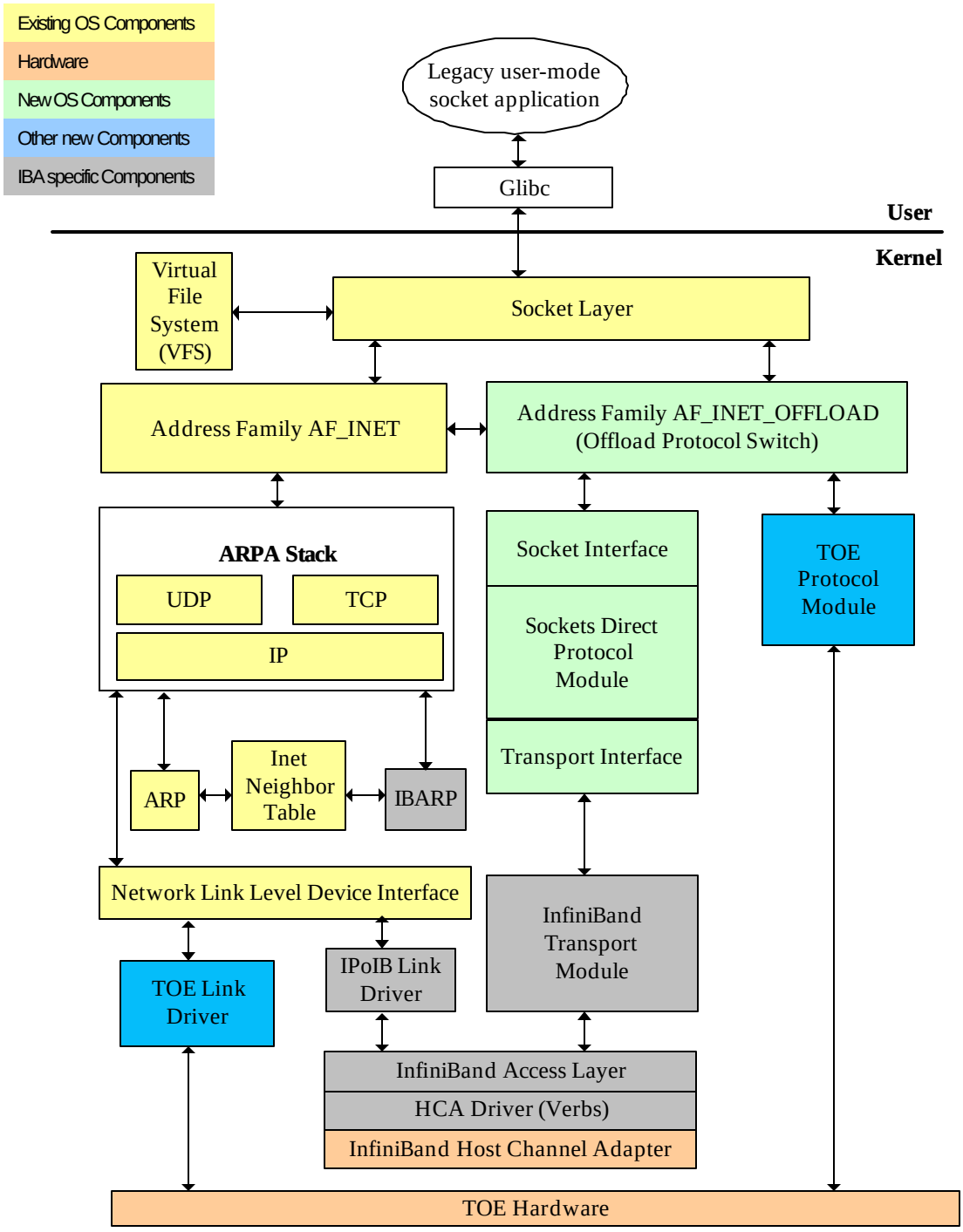

Figure 2 Offload Sockets framwork in the kernel

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

1.3.5 Component Details:

1.3.5.1 Offload Protocol Switch

Component Type	**Offload Protocol Switch**: New kernel module supporting AF_INET_OFFLOAD.
Purpose	The OPS module exposes a new address family (AF_INET_OFFLOAD) and preserves the protocol operation semantics with the kernel socket driver at the top while providing a new offload sock structure, interface, and binding to offload protocols and transports underneath to support hardware devices capable of offloading all reliable transport features.
Functionality	The OPS module provides the standard socket call interface for AF_INET_OFFLOAD and switches socket traffic across multiple protocol providers underneath. Provides new binding interface for new offload protocol modules to register network interface information along with a new offload proto operations. Attempts to switch sock_stream connections to offload protocols and falls back to standard stack if failures occur. Switches to standard stack at the AF_INET interface level requiring no changes to the AF_INET stack.
Externally resources required	Offload protocol and transport modules.
External interfaces	All external interfaces to AF_INET are defined by the Linux socket calls in *include/linux/net.h and include/linux/sock.h.* There is a new offload protocol structure and interface used for offload protocols that register with the AF_INET_OFFLOAD address family.
Internal interfaces	The OPS module is a thin veneer between the socket driver and the offload and non-offload INET protocols. Internal interfaces will be defined to provide routing information for the offload protocol modules and transport interfaces.
Dependencies and Inter-relationships with other components	The OPS is dependent on standard socket driver interface at the top and a new offload protocol structure and interface at the bottom.
Requirements and constraints	The OPS must be built to support a completely different address family while at the same time be designed to support switching across the standard AF_INET address family. This will enable Linux distributors the freedom to decide whether or not to support legacy IP socket applications under AF_INET or simply support offloading only under the new AF_INET_OFFLOAD address family. Modifications to the Linux kernel for transparent

	AF_INET support must be kept to a minimum.
Development Verification/Validation/Analysis/Test strategies	Standard socket applications test suites.

1.3.5.2 Offload Protocol Module - Sockets Direct

Component Type	**Offload Protocol Module – Socket Direct Protocol**: A new loadable Linux kernel module.
Purpose	The OP module provides standard socket session management across reliable transports. SDP is a standard wire protocol that maps TCP_STREAMS to reliable networks like InfiniBand. Preserves the standard socket abstraction at the top and manages these sessions across a new offload transport interface at the bottom.
Functionality	The OP module provides standard socket semantics via protocol operations exported up to OPS. Manages sockets across multiple transport providers underneath. Supports new binding interface for new offload transport modules and the new binding to OPS. Attempts to switch sock_stream connections to offload transports beneath.
Externally resources required	OPS (AF_INET_OFFLOAD address family) module and the IB Transport Module.
External interfaces	All external interfaces are defined by the new offload protocol structure and interface at the top and the new offload transport operations for the bottom half.
Internal interfaces	The OP module includes a socket/session management at the top, a Sockets Direct wire protocol engine in the middle and a transport management and interface at the bottom that is capable of supporting multiple transport modules.
Dependencies and Inter-relationships with other major components	The OP is dependent on OPS definitions at the top, SDP specification at the middleware layer, and the OTI definitions at the bottom. All of these are new interfaces with the exception of the *proto_ops* that is already defined.
Requirements and constraints	The OP must preserve the socket semantics at the top and support SDP specification as defined in the IBTA. Required to support InfiniBand transports.
Development Verification/Validation/Analysis/Test strategies	Standard socket applications test suites.

1.3.5.3 Offload Transport Module - InfiniBand Transport

Component Type	**Offload Transport Module**: A new loadable Linux kernel module.
Purpose	Provides abstraction of the underlying IBA Access Interface.
Functionality	Maps IBA access interfaces to standard OTI operations for socket based offload protocol modules. Transport services exported include: transport registration, IP to IB name services, IB connection services, memory mapping, and RDMA and data transfer.
Externally resources required	The IB-OT requires OTI definitions and registration mechanisms and InfiniBand Access Layer.
Externally visible attributes	OTI defined operations.
External interfaces	OTI defined operations and interface mechanism.
Internal interfaces	Interface with IPoIB device driver for address resolution.
Dependencies and Inter-relationships with other major components	The OTI dependent on new definitions of OTI operations interface and structure
Requirements and constraints	Requirements of these transports include memory registration, memory window binding, message sends and receives, RDMA write and reads, connect request/accept/reject/listen, and more (TBD)
Development Verification/Validation/Analysis/Test strategies	IB_AT developer unit tests. Standard socket applications test suites.

1.4 Theory of Operation

1.4.1 Socket Driver and Offload Protocol Switch 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.

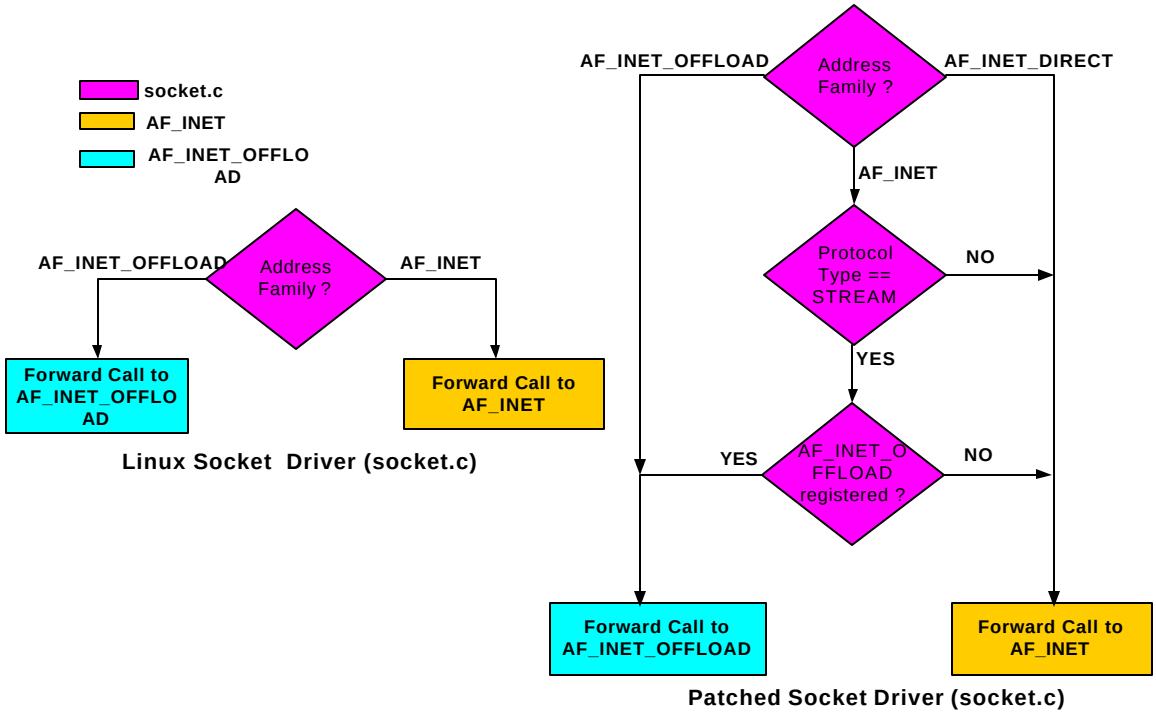

Figure 3: Proposed Patch to Linux Socket Driver

Figure 4 shows a high level overview of the protocol switching logic during socket creation by the offload protocol switch implemented under the AF_INET_OFFLOAD address family.

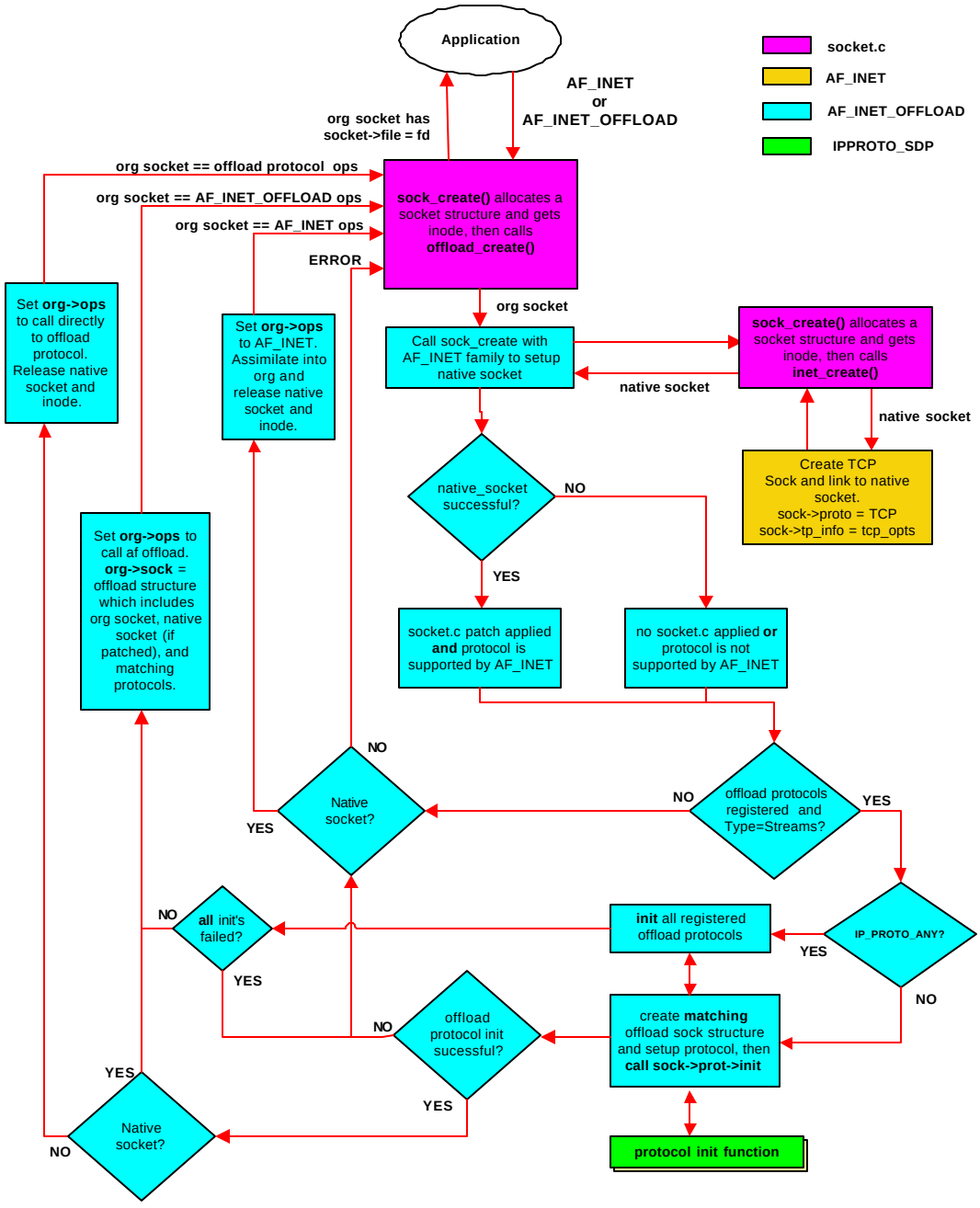

Figure 4: Protocol Switching Logic in AF_INET_OFFLOAD for Socket Creation

1.4.2 Offload Protocol Switch 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 *ops_proto_register ()* 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 APS module by calling o*ps_inet_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).

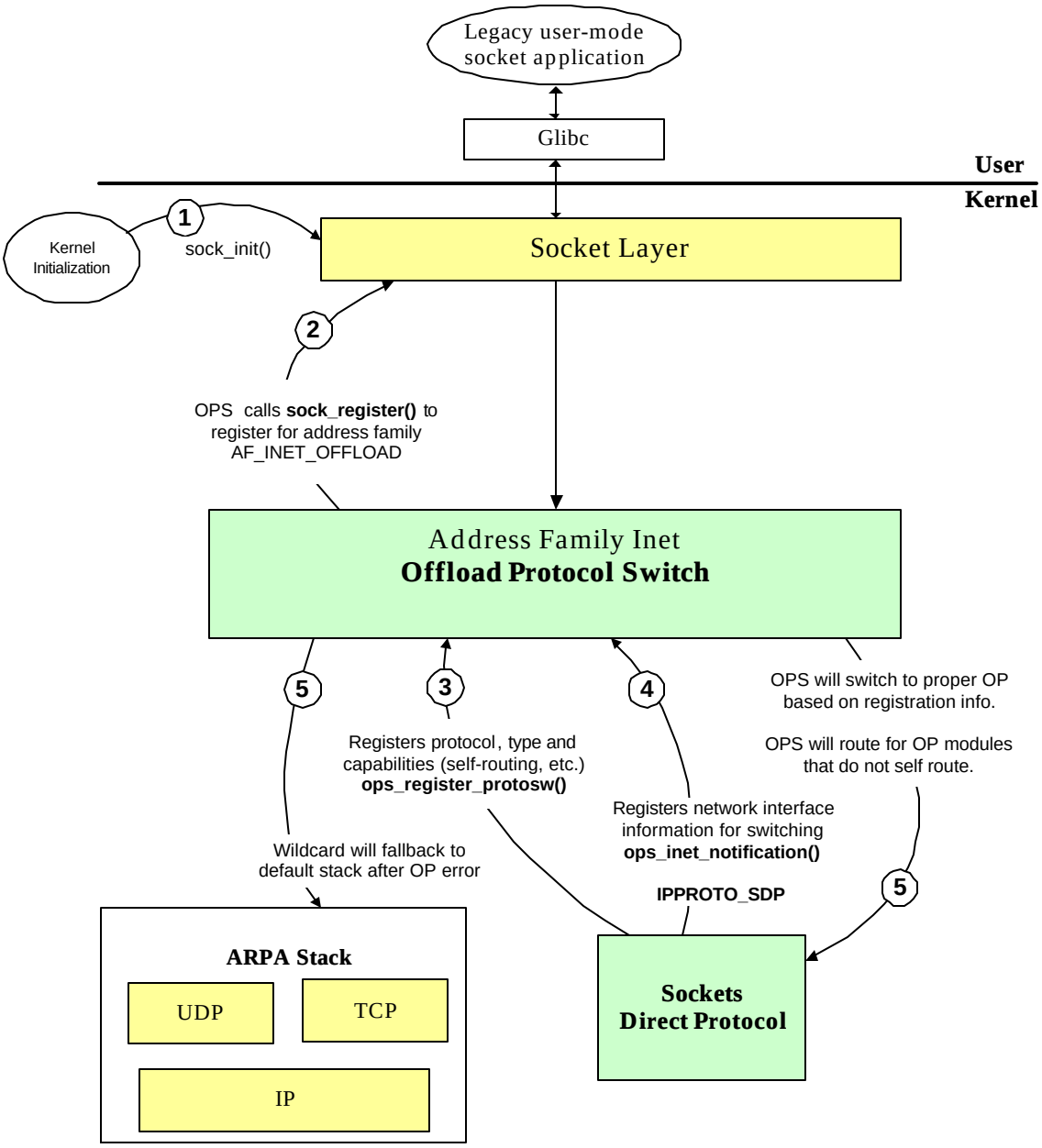

Figure 5 OPS and SDP interaction

1.4.3 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.

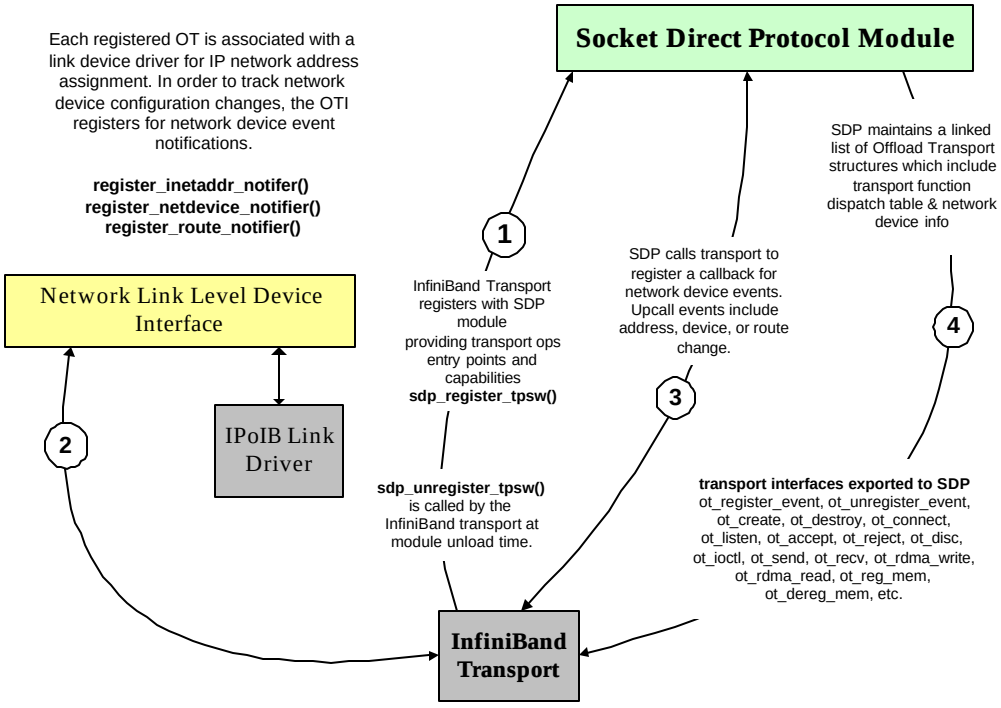

Figure 6 SDP and InfiniBand Transport

1.5 References

InfiniBand Architecture Specification Volume 1, Release 1.0a

Annex B Sockets Direct Protocol (SDP), Release 1.0.a

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

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

Memory Mgmt. in User Level Network Interfaces reference:

http://ww.cs.berkeley.edu/~mdw/papers