Volo

The New Socket Framework

Design Document

volo-iteam@sun.com

Table of Contents

List of Tables

Chapter - 1 Introduction

1.0 Introduction

This document describes the design of Project Volo which delivers a framework to dynamically add function call based, low latency, high performance sockets. It also describes the changes made to Solaris networking code to move native Solaris sockets to the new socket framework. Volo also delivers a kernel socket API.

1.1 Motivation

Beginning with FireEngine (PSARC 2002/433) Solaris networking started to move away from streams. FireEngine removed streams (only) from the fast data path. The current Solaris socket implementation is a hybrid implementation of streams and function calls. Function calls are used for the fast data path and streams are used for plumbing, control path and non fast path.

This complex hybrid implementation results in code complexity and poor performance in benchmarks that for some reason or another do not use the fast path. Furthermore, there is no framework available for ISV's to use this hybrid mechanism to dynamically add new sockets.

The lack of non streams socket framework has resulted in new socket implementations such as SCTP (PSARC 2003/586) and SDP (PSARC 2003/064) implementing their own function call based mechanisms. This has resulted in code duplication. As new sockets are added more code duplication will occur, so a framework is needed that facilitates sharing of common code and allows dynamic addition of new socket types.

There have also been several requests from internal and external customers looking for a kernel socket API in Solaris. Unfortunately, unlike other modern operating systems, Solaris does not have one.

Chapter - 2 Project Volo

2.0 Project Volo

Project Volo provides a framework to dynamically add non streams based sockets in Solaris. It also provides a kernel socket programming API.

Following are the main components of Project Volo.

2.1 Socket Configuration Framework

When Solaris boots an in-kernel table is populated with the information on what socket types are supported and how to create a certain socket type. The information is passed to the kernel when /sbin/soconfig runs from /etc/inittab. soconfig reads and parses the information in /etc/sock2path and passes it into the kernel.

Volo has enhanced the syntax of /etc/sock2path to include a module name that is loaded for creating non streams based sockets. Unlike the current Solaris behavior where socket modules remain loaded whether a socket of that type is active or not, Volo socket modules are only loaded when a socket of that type is created and modules are unloaded when there are no remaining socket types.

Chapter 3 provides details of this mechanism.

2.2 Socket Implementation Framework

A common socket implementation framework that is based on up and down function calls is provided to ease the implementation of new socket types and allow sharing of code. Since one design objective of Volo is to move as much state into the protocol as possible the up and down calls do very little and can be shared between different socket types.

No locks should be held when making a down call (calling the protocol). It is not recommended to hold any locks while making an up call. An up call/down call should never result in a up/down call in the same thread context. It is expected that simultaneous up/down calls will occur but with different thread contexts.

For tcp/udp/icmp sockets backward compatibility is provided by falling back to a streams based implementation when a streams ioctl is issued.

Chapter 4 provides details of this framework.

2.3 IP Helper Streams

One of the initial goals of Volo was to implement non streams sockets for IP based protocols. During the development it became clear that IP was heavily entrenched in streams queues and to implement non streams sockets for IP based protocols IP would have to be rewritten. This would have broadened the scope of the project beyond providing the infrastructure for creating non streams based sockets. After consulting with other engineers and experimenting with other ideas, including using a single queue per protocol, we had to settle on using a helper stream per socket because IP relies heavily on having a one to one relationship between conn_t and queue_t. Therefore sockets for IP based protocols create an IP stream that is used to pass data and issue ioctls and get/set socket opts to IP. An IP helper stream is created using the layered driver interface and a layered ioctl interface is used to pass ioctl's which the protocol can not handle to IP.

It is expected that a future project will remove IP's dependence on stream queues and

helper streams will no longer be needed. IP helper streams are described in Chapter 5.

2.4 Streams Support

Streams support in IP based sockets has been retained to support opening of a protocol device such as /dev/tcp. TLI/XTI sockets open protocol devices. The code has been factored out such that there are separate entry points for function calls and streams. These functions call into common code. This is described in Chapter 6.

2.5 Synchronous Squeue Enter and Changes to IP

Current IP implementation in Solaris is streams based which uses message passing and is asynchronous. With function call based sockets operations are synchronous. To support synchronous semantics, TCP squeues were enhanced to support synchronous enter and changes were made to IP bind/connect. These changes are described in Chapters 5 and 6.

2.6 Kernel API

Unlike other modern operating systems currently Solaris does not provide a kernel socket API for use by kernel modules. There is a TLI based interface, but it is very difficult to use. As part of project Volo we are providing a callback based kernel socket API, which closely resembles the user level socket API. This API is built on top of non streams based sockets. The details of the API are provided in Chapter 6. The stability level of kernel socket API interfaces is evolving.

2.7 Filtering Hooks

With streams removed ISV's need a mechanism to filter/modify messages flowing through the sockets. Volo was planned to provide hooks for 3rd party modules to register and filter messages, Filtering hooks will now be provided as a separate follow on project.

Chapter - 3 Socket Configuration and Lookup framework

3.0 Socket Configuration Framework

This chapter describes Volo socket configuration, lookup and creation. This framework allows dynamic addition of new socket types via socket modules.

3.1 Current Solaris Socket Configuration and Creation

3.1.1 Socket Configuration

In Solaris, socket configuration is stored in sock2path(4) which looks like

```
#   Family  Type  Protocol  Path
    2       2     0         /dev/tcp
    2       2     6         /dev/tcp

    26      2     0         /dev/tcp6
    26      2     6         /dev/tcp6
```

At boot time soconfig(1M) is run from inittab(4) which uses entries in sock2path(4) to populate the in-kernel socket table. Modules associated with the devices are loaded and remain part of the kernel even if no socket of that type is ever opened.

3.1.2 Socket Lookup and Creation

A socket(3SOCKET) call results in a lookup of the in-kernel socket table. The lookup is based on domain/type/protocol. The device associated with the socket type is opened to create a stream, a vnode is allocated and the stream is assigned to the vnode which is wrapped in a socket specific structure called sonode.

3.2 Volo Socket Configuration and Creation

3.2.1 Socket Configuration

Volo sockets are implemented as a new module type – sockmod. sock2path(4) syntax has been enhanced to ~~include an optional socket module name as follows~~ accept a device name or a module name. A device is identified by a leading "/dev"

Family type protocol path | module ~~[module]~~

soconfig(1M) and kernel socket configuration code has been modified to support the new syntax. As before soconfig(1M) runs from inittab(4) at boot time and reads sock2path(4) to populate the in-kernel table. ~~If no module name is found, the behavior is the same as described in section 3.1.1 and the resulting socket will be streams based.~~ Socket modules are expected to be placed in kernel/socketmod

~~If a module name is present, checks are made to ensure that the module exists and can be loaded. Any error will result in the entry not being added to the in-kernel table. The socket module must exist in kernel/sockmods relative to the search path.~~

~~A module entry takes precedence over the device entry and the resulting socket will be non-streams based. Currently, if there is an error in creating the socket using the module~~

~~entry, an error is returned and no attempt is made to create a socket using the device entry. This could be a future enhancement.~~

soconfig(1M) can be used to add or delete an entry but can not be used to update an entry. To update an entry, the entry must first be deleted. An entry can only be deleted if no sockets of that type are currently active. ~~Providing a device or module name to soconfig is optional. Only one is really needed.~~ The following examples illustrate the syntax of soconfig

To add an entry ~~without module~~ that uses device ~~name~~

#/sbin/soconfig 2 2 0 /dev/tcp

To add an entry ~~without device~~ that uses module ~~name~~

#/sbin/soconfig 2 2 0 ~~—~~socktcp

~~To add an entry with both device and module name~~

~~#/sbin/soconfig 2 2 0 /dev/tcp socktcp~~

To delete an entry

#/sbin/soconfig 2 2 0

3.3 New Loadable Module Type – sockmod

A new loadable module type sockmod has been introduced. The protocol writer registers a single create function that is called as part of socket creation to pass upper handle and other information to the protocol. Socket module writer is not required to keep track of open sockets, the framework keeps track of open sockets and fails unloading of the module if any sockets are active. The code below shows how a module registers it's create function.

~~Sockmod facilitates registration of create and destroy functions, which are called at create and close time. The code segment below shows how the functions are registered when module's _init() is called. A socket module writer is not required to keep track of open sockets. The _mod_remove() call will fail with EBUSY if there are any active sockets using the module.~~

```
#include <sys/strsubr.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/modctl.h>
#include <sys/cmn_err.h>
#include <sys/socketvar.h>

extern struct sonode *sotcp_create(vnode_t *, int, int, int, int,
    struct sonode *, int, int *);
extern void sotcp_destroy(struct sonode *);

static smod_reg_t sinfo = {
        SOCKMOD_VERSION,
        "socktcp",
        sotcp_create,
        sotcp_destroy
};
static smod_reg_t sinfo= {
    SOCKMOD_VERSION,
    “NewProto”,
    SOCK_UC_VERSION,
    SOCK_DC_VERSION,
    *new_proto_create,
    NULL
};
/*
 * Module linkage information for the kernel.
 */
static struct modlsockmod modlsockmod = {
        &mod_sockmodops, "TCP New Proto socket module", &sinfo
};
```

```
static struct modlinkage modlinkage = {
        MODREV_1,
        &modlsockmod,
        NULL
};

int
_init(void)
{
        return (mod_install(&modlinkage));
}
int
_fini(void)
{
        return (mod_remove(&modlinkage));
}

int
_info(struct modinfo *modinfop)
{
        return (mod_info(&modlinkage, modinfop));
}
```

3.4 In-kernel Lookup Tables

3.4.1 Socket Lookup Table

When soconfig is run to configure the sockets, a link list of struct sockparams headed by sphead is created in the kernel.

```
/*
 * sockparams
 *
 * Used for mapping family/type/protocol to module
 */
struct sockparams {
        /*
         * The family, type, protocol, and sdev_info and smod_name are
         * set when the entry is created, and they will never change
         * thereafter.
         */
        int             sp_family;
        int             sp_type;
        int             sp_protocol;

        sdev_info_t     sp_sdev_info;   /* STREAM device */
        char            *sp_smod_name;  /* socket module name */
        smod_info_t     *sp_smod_info;  /* socket module */

        kmutex_t        sp_lock;        /* lock for refcnt */
        uint64_t        sp_refcnt;      /* entry reference count */

        /*
         * The entries below are only modified while holding
         * splist_lock as a writer.
         */
        int             sp_flags;       /* see below */
        struct sockparams *sp_next;
};
```

It should be noted that when a sockparms is created sp_smod_info is NULL, only sp_mod_name is populated.

3.4.2 Module Lookup Table

Module information is maintained in a separate table headed by smod_list_head, which is a linked list of smod_info_t structure's.

```
/* Socket module information */
typedef struct smod_info {
        int             smod_version;
        char            *smod_name;
        kmutex_t        smod_lock;      /* lock for smod_refcnt */
        int             smod_refcnt;    /* # of entries */
        so_create_func_t        smod_create_func;
        so_destroy_func_t       smod_destroy_func;
        struct smod_info        *smod_next;
} smod_info_t;
/*
 * Socket module information
 */
typedef struct smod_info {
        int             smod_version;
        char            *smod_name;
        uint_t          smod_refcnt;            /* # of entries */
        size_t          smod_uc_version;        /* upcall version */
        size_t          smod_dc_version;        /* down call version */
        so_proto_create_func_t  smod_proto_create_func;
        so_proto_fallback_func_t smod_proto_fallback_func;
        so_create_func_t        smod_sock_create_func;
        so_destroy_func_t       smod_sock_destroy_func;
        list_node_t     smod_node;
} smod_info_t;
```

When a module is loaded, its _init() function calls mod_load() which registers using a smod_reg_t (Table 1), and which this results in a smod_info structure being created for that module. smod_info is also which added to the linked in the module info list. The smod_reg_t is defined as:

```
/* Socket module register information */
typedef struct smod_reg_s {
        int             smod_version;
        char            *smod_name;
        so_create_func_t        smod_create_func;
        so_destroy_func_t       smod_destroy_func;
} smod_reg_t;
/*
 * Socket module register information
 */
typedef struct smod_reg_s {
        int             smod_version;
        char            *smod_name;
        size_t          smod_uc_version;
        size_t          smod_dc_version;
        so_proto_create_func_t  smod_proto_create_func;

        /* __smod_priv_data must be NULL */
          smod_priv_t   *__smod_priv;
} smod_reg_t;
```

Table 1: smod_reg_t

3.5 Socket Creation

A socket(3SOSCKET) system call results in a lookup of the sockparams structures list. If a match is found and sp_smod_info is NULL, a search of the smod_info structures is done based on the module name. If a match is found, the module reference count is incremented and a pointer to the entry is returned. This pointer is stored in sockparams for future use.

If the module is not found in the list, the module is loaded, This results in the registration and creation of the entry, which is returned as described above.

The create function that the module registers is called to create the socketsetup the newly created socket.

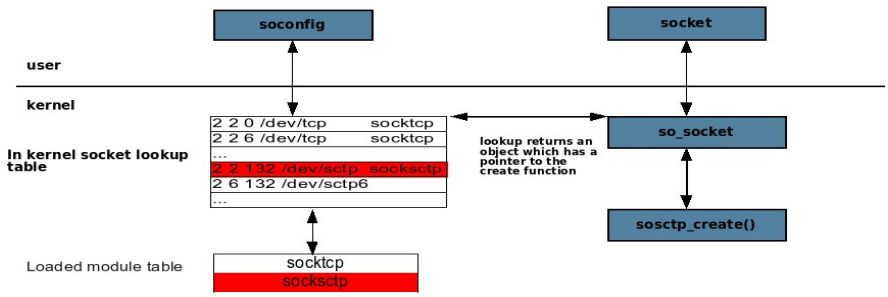
3.6 Socket Close

As part of socket close, the destroy function that the module registered is called, the reference count on sockparams is decremented and if zero, the reference count on sockmod is decremented. If sockmod reference count becomes zero the module is unloaded and the sockmod info pointer in sockparams is set to NULL.

No locks are held when the module is being unloaded. If a new request results in the module lookup before it has been unloaded the unload will fail because it checks for reference counts on smod_info structure.

3.7 Socket Creation and Lookup in a Nutshell

Following is a graphical representation of socket creation. The only change is that the in-kernel table has either a device entry or a module entry not both.



Chapter - 4 Socket implementation Framework

4.0 Socket Implementation Framework

This chapter describes the socket implementation framework that new socket developers will use to implement new socket types.

4.1 Current Sockfs

Historically IP based protocols in Solaris have been streams based and used TPI messages for communication with other system conponents. Sockfs (PSARC/1995/045) is an access layer that provides socket semantics while internally communicating to the protocols via TPI messages.

Due to performance reason starting with FireEngine (PSARC 2002/433) Solaris networking began moving away from a TPI based interface to a function call based interface. FireEngine merged TCP and IP and created function call based fast paths for send and accept calls, however creation of socket and control operations were left to use streams framework. Similar changes were made to UDP by Yosemite(PSARC 2005/082).

Newer protocols (SCTP/SDP) have been implemented as function call based. Their socket implementation is also function call based, however these implementations are stand alone implementations with no framework to share common code.

Currently Solaris sockets are a hybrid implementation of streams and direct function call based fast paths. The mixed use of TPI and direct function calls has made an already complex stack even more difficult to understand, which is a mitigating factor as we try to get community members interested in our networking stack. Moreover ISV's have no way to dynamically add function call based or hybrid sockets. ISV's can only add streams based sockets.

4.2 Volo Socket Framework

Volo introduces a socket framework that provides a complete set of generic socket operations and a function call based interface between sockfs and the transport protocol. This framework allows dynamic addition of new socket types to a running Solaris kernel, as well as sharing of common code.

Sockets for the core set of protocols (TCP/UDP/ICMP) have been moved to use the Volo framework.

There are features such as Kernel SSL Proxy(KSSL) and the Network Layer Cache (NL7C) that have been integrated into the TPI code path. Volo will detect these types of sockets and fallback to using streams based sockets. Volo will also fallback to streams based socket if an I_PUSH or I_POP operation is issued on the socket.

4.3 Sonode

Currently sonode switch (PSARC 2003/729) is the main data structure of the socket and it is wrapped in vnode (v_data). Sonode maintains, among other things, the socket state and a function pointer table for socket operations. This pointer is initialized at socket creation time based on the type of socket and allows using the same sonode structure for different protocol types.

Volo continues to use sonode, however the current sonode is very TPI centric so Volo has modified it to better reflect the new design.

4.3.1 Changes to the sonode

Since TPI did not follow socket semantics, the sonode had to contain state, options and addresses, as well as specific TPI information. In some cases the information was cached due to performance reasons, and in other cases it had to be maintained because the transport protocol lost it, or did not even keep it at all. For the majority of sockets, that information is no longer needed, and has therefore been factored out of the basic sonode.

Socket type specific information is stored by allocating an extended socket. The only requirement being that the first object be the generic sonode. In the sonode so_priv points to the type specific information.

Sonode is listed in section 8.1

4.3.2 TPI Extended Socket

In Volo, a TPI socket module has been introduced that allocates TPI specific socket objects. The TPI information has been factored out into a separate structure (Table 2), which is allocated when TPI based sockets are created, or when function call based sockets fall back to TPI (more in Section 4.3.4). The structure is accessible via so_priv.

```
/*
 * TPI specific information; part of sotpi_sonode_t, or allocated when a socket
 * falls back to TPI.
 */
typedef struct sotpi_info {
     ...
} sotpi_info_t;

/*
 * TPI specific sonode; only allocated by the TPI socket module.
 */
typedef sotpi_sonode {
     struct sonode     tpi_sonode;
     sotpi_info_t      tpi_info;
} sotpi_sonode_t;
```

Table 2: TPI specific sonode structures

TPI based sockets relied on STREAMS and the STREAM head to provide much of the needed functionality, including the receive buffer, flow control and signal generation. Since there is no STREAM head for function based sockets, the sonode will take on the responsibility of providing the missing functionality.

4.3.3 TPI Fast Paths, KSSL and NL7C

As mentioned earlier, the TPI socket operations contains fast-paths, which were introduced by FireEngine (PSARC/2002/433) and Yosemite (PSARC/2005/082). The fast-paths are only there for send and accept, and they are enabled for TCP and UDP. These fast-paths will not be removed by Volo for multiple reasons:

1. It minimizes the risk of the Volo put back, as users can revert to the TPI code path, without a loss in performance
2. More importantly, KSSL, (PSARC/2002/557, PSARC/2002/625) and NL7C (PSARC/2005/038) both rely on these fast-paths

Once the functionality provided by KSSL and NL7C is natively available for function call based sockets, then the fast-paths can be safely removed. Until the functionality is available however, applications that try to use the features will fall back to using TPI based sockets.

Determining whether either of the features should be used can be done when a bind

operation is issued. Both KSSL and NL7C are only available for TCP, so we have introduced a TCP specific bind operation that calls *nl7c_lookup_addr()* and *kssl_check_proxy()* to determine if a fall back needs to happen. Apart from the bind operation taking slightly longer, all critical paths are unaffected.

4.3.4 Fallback to TPI

TPI based sockets allowed the application to issue *I_POP* and *I_PUSH* ioctls to insert or remove STREAM modules. Although in limited use, telnet and rlogin are still using the functionality. Another STREAM ioctl that may be observed is *I_LOOK*, which was used by rpcgen to determine the type of socket. The behavior was captured in the following CR:

1181685 rpc applications and rpcgen should not reference "sockmod" (and "timod?)

I_LOOK can be handled without falling back to TPI, however, if an I_POP or I_PUSH is issued, fall back must happen. Although fall back is supported, it is *not* a generic solution that apply to any previously TPI based transport. In fact, fall back will only be supported by TCP, UDP and ICMP.

The fallback sequence is started when an application issues either an *I_PUSH* or *I_POP* ioctl. The following steps are then taken to convert the function call based socket into a TPI based one:

1. The fallback thread waits until it has exclusive access to the sonode. This involves waiting for pending operations to complete, and the fallback thread will make sure to wake up any thread that is blocked in a operation
2. The TPI structure (sotpi_info_t) is allocated and initialized, and then the generic TPI setup is issues which creates the STREAM head
3. The transport protocol is then informed that a fallback is underway, and is informed about the STREAM head (see Section 4.5.1.8).
4. Once within the perimeter of the protocol, and there is a guarantee that no new data or connections can arrive, the protocol notify the sonode (see Section 4.5.2.13), which will then move all data to the STREAM head. If there are any pending connections, then the protocol is responsible for sending connection indications to the STREAM head.
5. Once the fallback thread has left the protocol's perimeter, and control is back in the sonode, any non-TPI information is freed.
6. The socket operations are changed into the TPI operations
7. The fallback thread drops the exclusive access

4.4 Socket Framework

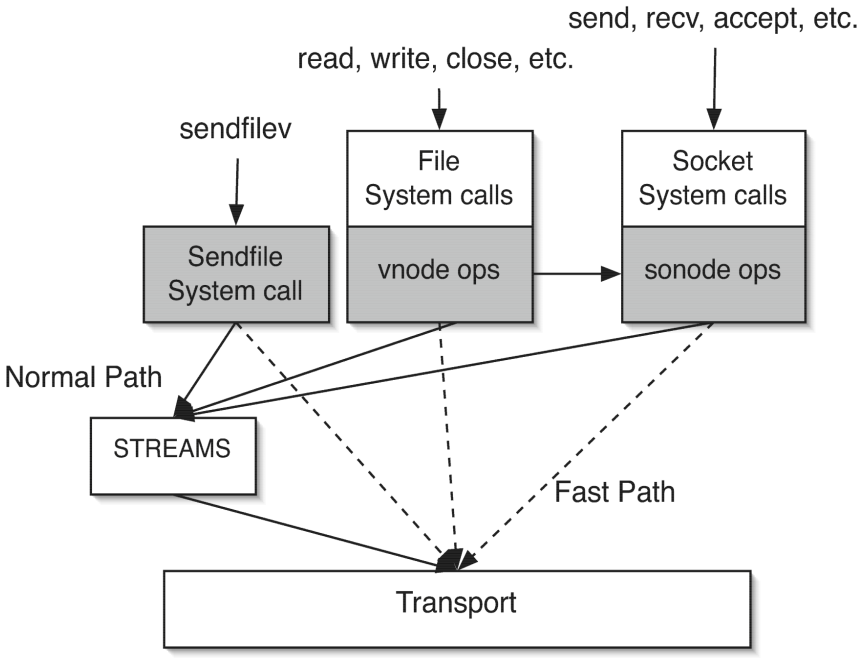


Figure 1: Code flow for TPI sockets. The gray boxes indicate areas where TPI specific code is used. The code flow for SCTP and SDP is similar, but without the path through STREAMS.

Part of the complexity of adding a new socket type is the existence of multiple entry points into the socket. Most operations come in via the vnode ops and sonode switch, the exceptions are sendfilev(3EXT) and socketpair(). Figure 1 show the call flow for a TPI based socket. Part of the socket framework is to establish a single operation vector which will handle all request. The operation vector is an extension of the already existing sonode switch.

Another source of complexity is the lack of generic socket operations. Unlike TPI, which had a specification describing the message exchange, there is no such interface that is function call based. As a result, the existing transport protocols that use function calls (SCTP and SDP) each implemented their own socket and vnode operations, as well as adding the code indicating no support for socketpair(3socket) and sendfilev(3ext). Although not a lot of code, having each transport protocol going through the same exercise is error prone and hard to maintain. The framework will provide a set of generic socket operations to be used in the sonode ops structure, and the transport protocol should only have to provide its' own operations if something exceptional needs to be done.

To support generic operations, we define a function call based interface that will be used to handle communication between sockfs and the transport protocol. The interface defines a set of up- and downcalls. Figure 2 below shows how the pieces of the framework fit together.

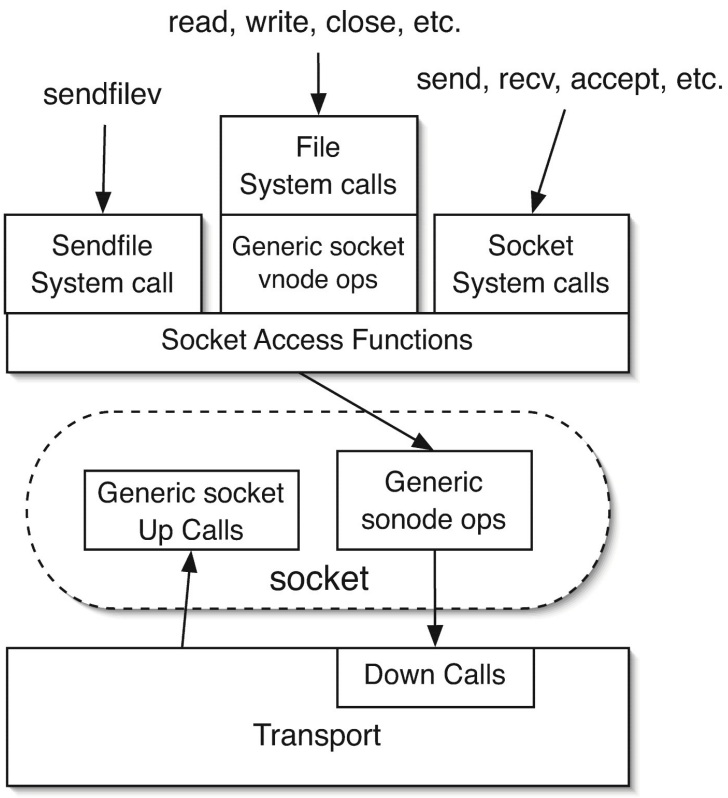


Figure 2: Call flow in Volo.

4.4.1 Changes to the sonode switch

A number of socket operations are generated by file related system calls, including read(2), write(2), ioctl(2), poll(2) and close(2). In sockfs today, each socket type implements their own vnode ops. To reduce the number of operation vectors each protocol has to handle. The socket framework will extend the sonode switch to contain:

```
int     (*sop_ioctl)(struct sonode *, int, intptr_t, int, struct cred *,
                int32_t *);
int     (*sop_poll)(struct sonode *, short, int, short *, struct pollhead **);
int     (*sop_close)(struct sonode *, struct cred *, int);
```

No additional operations are needed for read() and write(), as those map directly to recv() and send() and can therefore use the existing operations. The call flow when issuing a write(2) call would be:

write(2) → fop_write() → socket_vop_write() → sop_send()

It is not expected that the extra level of indirection will lead to any significant overhead. It should be noted that socket modules are not locked into using the generic vnode ops, as they are free the specify their own vnode ops when the socket is created.

Another source of operation requests is the sendfilev(3EXT) system call, which interacted directly with the sonode without going through the sonode switch. The goal is to move towards protocol agnostic socket operations, and so we introduce the following addition to the sonode switch:

```
int (*sop_sendmblk)(struct sonode *, struct nmsghdr *, int, struct cred *,
      mblk_t **mpp);
```

The requirements for sendfilev(3EXT) and its use of sop_sendmblk() will be discussed in section 4.6.

4.4.2 Generic Socket Operations

The sonode ops for TPI type sockets are “generic”, as they handle all transport protocols that provide a TPI interface. However, the TPI specification does not match socket semantics, and to enable proper semantics there was a need to maintain socket state within the sonode. So for a given socket, every operation resulted in a multiple set of state check, the first one was needed to verify proper socket semantics, which was done in sockfs. Once the request reached the transport protocol, it would have to verify its' own state. Similarly, the sonode had to cache information either because the operation of querying the protocol was expensive, or due to the information not being available in the protocol when needed.

The interface that is defined by Volo follow socket semantics, which allows us to do the following:

1. In general, sockfs should not do any state checks for requests that will go down to the transport protocol, since the protocol have to do state checks once within its perimeter
2. Information that is maintained in the protocol should not be duplicated, or cached, in the sonode

As a result of the above, most socket operations will do very little work before being sent down to the protocol. The main tasks that are handled within sockfs are the following:

1. Copy user data in and out
2. Handle poll/select
3. Wait for socket events, for example, data to arrive, connection to be established, etc.

The operations that do the most work are receive, accept and send. The data for receive and connections for accept, are all stored in the socket, however, there is still a need to determine whether the operation are enabled, e.g., doing an accept() if the socket is not listening for connections should generate an error. TPI addresses this issue by maintaining complete socket state, which we want to avoid. Instead we considered two approaches; (1) maintain no state in the protocol, and rely on downcalls to verify the state with the protocol, and (2) maintain operation specific state.

There were concerns that adding an additional downcall per operation would result in unnecessary overhead. Locking could also become an issue with the first approach, as we sometimes have to hold locks while checking state. With the second approach we need to keep track of which of the three above mentioned operations have been enabled, it has none of the concerns of the first approach. We decided to go with the second approach.

Locking semantics used when interacting with the protocol is therefore simple; no locks are ever held when a down call is made, while not recommended the protocol may hold locks while making an upcall. The complete set of up- and down calls used by the generic socket operations are shown in Sections 4.4.3 and 4.5.2.

4.4.3 Downcalls

These are the downcalls used by the generic operations, and they follow socket semantics. No locks can be held when a down call is made, as the transport protocol may generate an up call in response. The protocol control block is identified using an opaque handle, which is defined as:

4.5

```
typedef void * sock_lower_handle_t;

typedef struct sock_downcalls_s {
        sock_lower_handle_t (*sd_create)(int, int, cred_t *, sock_upper_handle_t,
                        struct sock_upcalls_s *, int *);
        int     (*sd_accepted)(sock_lower_handle_t, sock_lower_handle_t,
                        sock_upper_handle_t, cred_t *);
        int     (*sd_bind)(sock_lower_handle_t, struct sockaddr *, socklen_t,
                        cred_t *);
        int     (*sd_unbind)(sock_lower_handle_t, cred_t *);
        int     (*sd_listen)(sock_lower_handle_t, int, cred_t *);
        int     (*sd_connect)(sock_lower_handle_t, const struct sockaddr *,
                        socklen_t, cred_t *, sock_gen_t *);
        int     (*sd_disconnect)(sock_lower_handle_t, cred_t *, sock_gen_t *);
        void    (*sd_fallback)(sock_lower_handle_t, queue_t *);
        int     (*sd_getpeername)(sock_lower_handle_t, struct sockaddr *,
                        socklen_t *, cred_t *);
        int     (*sd_getsockname)(sock_lower_handle_t, struct sockaddr *,
                        socklen_t *, cred_t *);
        int     (*sd_getsockopt)(sock_lower_handle_t, int, int, void *,
                        socklen_t *, cred_t *);
        int     (*sd_setsockopt)(sock_lower_handle_t, int, int, const void *,
                        socklen_t, cred_t *);
        int     (*sd_send)(sock_lower_handle_t, mblk_t *, struct nmsghdr *msg,
                        int flags, cred_t *);
        int     (*sd_shutdown)(sock_lower_handle_t, int, cred_t *);
        void    (*sd_setflowctrl)(sock_lower_handle_t, boolean_t);
        int     (*sd_ioctl)(sock_lower_handle_t, int, intptr_t, int,
                        cred_t *, int32_t *);
        int     (*sd_close)(sock_lower_handle_t, int, cred_t *);
} sock_downcalls_t;

typedef struct __sock_lower_handle *sock_lower_handle_t;

typedef struct sock_downcalls_s {
        void    (*sd_activate)(sock_lower_handle_t, sock_upper_handle_t,
                        struct sock_upcalls_s *, int, cred_t *);
        int     (*sd_accept)(sock_lower_handle_t, sock_lower_handle_t,
                        sock_upper_handle_t, cred_t *);
        int     (*sd_bind)(sock_lower_handle_t, struct sockaddr *, socklen_t,
                        cred_t *);
        int     (*sd_listen)(sock_lower_handle_t, int, cred_t *);
        int     (*sd_connect)(sock_lower_handle_t, const struct sockaddr *,
                        socklen_t, sock_connid_t *, cred_t *);
        int     (*sd_getpeername)(sock_lower_handle_t, struct sockaddr *,
                        socklen_t *, cred_t *);
        int     (*sd_getsockname)(sock_lower_handle_t, struct sockaddr *,
                        socklen_t *, cred_t *);
        int     (*sd_getsockopt)(sock_lower_handle_t, int, int, void *,
                        socklen_t *, cred_t *);
        int     (*sd_setsockopt)(sock_lower_handle_t, int, int, const void *,
                        socklen_t, cred_t *);
        int     (*sd_send)(sock_lower_handle_t, mblk_t *, struct nmsghdr *,
                        cred_t *);
        int     (*sd_send_uio)(sock_lower_handle_t, uio_t *, struct nmsghdr *,
                        cred_t *);
        int     (*sd_recv_uio)(sock_lower_handle_t, uio_t *, struct nmsghdr *,
                        cred_t *);
        short   (*sd_poll)(sock_lower_handle_t, short, int, cred_t *);
        int     (*sd_shutdown)(sock_lower_handle_t, int, cred_t *);
        void   (*sd_clr_flowctrl)(sock_lower_handle_t);
        int    (*sd_ioctl)(sock_lower_handle_t, int, intptr_t, int,
                        int32_t *, cred_t *);
        int    (*sd_close)(sock_lower_handle_t, int, cred_t *);
} sock_downcalls_t;
```

4.5.1.1 sd_activate~~ereate~~()

The sd_~~ereate~~activate() downcall is defined as:

```
sock_lower_handle_t (*sd_create)(int family, int type, cred_t *cr,
        sock_upper_handle_t uh, sock_upcalls_t *uc, int *errorp);
void    (*sd_activate)(sock_lower_handle_t, sock_upper_handle_t,
                struct sock_upcalls_s *, int, cred_t *);
```

This call is only used by the active open to pass to the protocol the upper handle and the upcalls.

And it is used to request a new lower handle. The downcall is used only for active opens, and the protocol is passed in the address family and socket type, as well info regarding how the socket can be notified. Upon success, a new lower handle should be allocated and returned. Failure should result in NULL being returned, and the *errorp* argument must be set to indicate the type of error that occurred. The error code returned in *errorp* should follow the convention of socket(3xnet).

4.5.1.2 sd_acceptpted()

sd_acceptted() is defined as:

```
int     (*sd_accepted)(sock_lower_handle_t llh, sock_lower_handle_t alh,
                sock_upper_handle_t auh, cred_t *cr);
```

sd_accepted() is used to notify the protocol that a connection has been taken off of the accept queue, as a result of accept() being called. If for some reason the protocol cannot allow the operation to complete, then an error following the convention in accept(3xnet) should be returned. Upon error, the socket is freed.

4.5.1.3 sd_bind()

The sd_bind() downcall is defined as:

```
int     (*sd_bind)(sock_lower_handle_t lh, struct sockaddr *addr,
            socklen_t addrlen, cred_t *cr);
```

sd_bind() is used to send a bind request to the protocol. There are no restrictions on the value of the address, and it could be NULL. A NULL address is normally interpreted as an unbind request, but whether that is supported is left to the protocol. If the bind request was processed successfully, then 0 should be returned, otherwise a valid *bind(3xnet)* errno value should be returned.

4.5.1.4 sd_unbind()

sd_unbind() is defined as

```
int     (*sd_unbind)(sock_lower_handle_t, cred_t *);
```

It is used to unbind. The error semantics of sd_unbind() is same as sd_bind() when the address passed in is NULL.

4.5.1.5 sd_listen()

sd_listen() is defined as:

```
int     (*sd_listen)(sock_lower_handle_t lh, int backlog, cred_t *cr);
```

And it is used to request the protocol to start listening for incoming connections. If the down call can not be completed, then an error code that satisfies *listen(3xnet)* should be returned. Otherwise 0 should be returned.

4.5.1.6 sd_connect()

sd_connect() is defined as:

```
int     (*sd_connect)(sock_lower_handle_t lh, const struct sockaddr *addr,
            socklen_t addrlen, sock_connid_t *id, cred_t *cr, sock_gen_t *gen);
```

where *sock_gen_t* is:

```
typedef uint64_t sock_gen_t sock_connid_t;
```

sd_connect() is used to issue a connection request with the protocol. In addition to the destination address, the caller passes in a pointer to a generation variable. Upon success, 0 is returned and the generation is set. The generation number should be used to compare with the generation numbers passed up by the 'connected' and 'disconnected' up calls to determine whether to wakeup the thread. The generation variable could be used to identify multiple simultaneous connection requests, if such behavior is provided by the protocol.

Failure to issue the request should be reported by returning an error code corresponding to *connect(3xnet)*.

4.5.1.7 sd_disconnect()

It is defined as

```
int     (*sd_disconnect)(sock_lower_handle_t, cred_t *, sock_gen_t *);
```

It is used to disconneet. It returns zero on success. If the socket is not connected -1 is returned and errno is set to EALREADY.

4.5.1.8 sd_fallback()

sd_fallback() is defined as:

```
void    (*sd_fallback)(sock_lower_handle_t lh, queue_t *q);
```

And it is used to notify the protocol that socket is falling back to TPI. The protocol is passed a pointer to the queue that will be used for the TPI communication.

4.5.1.9 sd_getpeername()

sd_getpeername() is defined as:

```
int     (*sd_getpeername)(sock_lower_handle_t lh, struct sockaddr *addr,
                socklen_t *addrlen, cred_t *cr);
```

sd_getpeername() is used to obtain the peer address of a connected socket. Upon success the address should be filled in and 0 returned. Upon failure an error corresponding to one in *getpeername(3xnet)* should be returned.

4.5.1.10 sd_getsockname()

sd_getsockname() is defined as:

```
int     (*sd_getsockname)(sock_lower_handle_t lh, struct sockaddr *addr,
                socklen_t *addrlen, cred_t *cr);
```

sd_getsockname() is used to obtain the local address of the socket. Upon success the address should be filled in and 0 returned. Upon failure an error corresponding to one in *getsockname(3xnet)* should be returned.

4.5.1.11 sd_getsockopt()

sd_getsockopt() is defined as:

```
int     (*sd_getsockopt)(sock_lower_handle_t lh, int level, int optname,
```

```
                void *optval, socklen_t *optlen, cred_t *cr);
```

And it is used to retrieve a socket options. Upon success 0 should be returned, otherwise an error corresponding to *getsockopt(3xnet)* should be returned.

4.5.1.12 sd_setsockopt()

sd_setsockopt() is defined as:

```
int     (*sd_setsockopt)(sock_lower_handle_t lh, int level, int optname,
                const void *optval, socklen_t optlen, cred_t *cr);
```

And it is used to set a socket option. Upon success 0 should be returned, otherwise an error corresponding to *setsockopt(3xnet)* should be returned.

4.5.1.13 sd_send()

sd_send() is defined as:

```
int     (*sd_send)(sock_lower_handle_t lh, mblk_t *mp, struct nmsghdr *msg,
                    int flags, cred_t *cr);
```

sd_send() is used to submit data to the transport protocol for transmission. The down call should not be made if the protocol has turned on flow control. The protocol should accept all data unless it is in a state where send path has been shutdown.

If the data cannot be accepted, then the protocol is responsible for freeing it, and an error code corresponding to *sendmsg(3xnet)* should be returned. The protocol should *not* discard data if flow controlled is enabled, or becomes enabled as a result of the downcall. However, the protocol can stop additional data from being sent down by notifying the socket via an up call (see Section 4.5.2.7).

sd_send is required for sendfile. If both sd_send and sd_send_uio are defined sd_send_uio is used.

4.5.1.14 sd_send_uio()

```
int     (*sd_send_uio)(sock_lower_handle_t lh, uio_t *uiop, struct nmsghdr *msg,
            cred_t *cr);

    sd_send_uio allows protocol to directly manage uio's. sd_send_uio has precedence
over sd_send.
```

4.5.1.15 sd_recv_uio ()

```
int     (*sd_recv_uio)(sock_lower_handle_t, uio_t *, struct nmsghdr *,
            cred_t *);

sd_recv_uio is used to get data directly frpm the protocol. When sd_recv_uio is non
null the ptotocol stores the data as opposed to su_recv where socket framework stores
the data. Only sd_recv_uio or su_recv can be defined.
```

4.5.1.16 sd_poll ()

```
short   (*sd_poll)(sock_lower_handle_t, short, int, cred_t *);

sd_poll is used by the framework to poll the protocol for POLLIN, POLLRDNORM and
POLLRDBAND.
```

4.5.1.17 sd_shutdown()

sd_shutdown() is defined as:

```
int     (*sd_shutdown)(sock_lower_handle_t lh, int how, cred_t *cr);
```

sd_shutdown() is used to shutdown either outgoing or incoming communication channels. The 'how' argument is guaranteed to be one of the following: SHUT_RD, SHUT_WR, or SHUT_RDWR. Success is indicated by returning 0, while error should return a value corresponding to *shutdown(3xnet)*.

4.5.1.18 sd_clr_setflowctrl()

sd_setflowctrl() is defined as:

```
void    (*sd_setclrflowctrl)(sock_lower_handle_t, boolean_t);
```

And iIt is used to notify the protocol that there is enough buffer space to remove the receive flow control. The downcall would normally be triggered by a *recv()* system call.

4.5.1.19 sd_ioctl()

sd_ioctl() is defined as:

```
int     (*sd_ioctl)(sock_lower_handle_t lh, int cmd, intptr_t arg, int mode,
                    cred_t *cr, int32_t *rvalp);
```

sd_ioctl() is used to issue ioctls to the protocol. Unlike the send and receive operations, the protocol is responsible for copying in user data.

4.5.1.20 sd_close()

sd_close() is defined as:

```
int     (*sd_close)(sock_lower_handle_t lh, int flag, cred_t *cr);
```

sd_close() is used to close down the end point. Once the protocol has processed the close request, no new up calls are allowed to be made.

4.5.2 Upcalls

The upcalls are used by the transport protocol to notify a socket about events that impact the operation of the socket. Although not recommended the protocol may hold locks while performing an upcall. The transport protocol uses an opaque handle to reference the socket, which is defined as follows:

```
typedef void * sock_upper_handle_t;
typedef struct __sock_upper_handle *sock_upper_handle_t;
```

And the upcalls are defined as:

```
/*
 * Socket up calls; invoked by the transport protocol to notify the socket
 * about events.
 */
typedef struct sock_upcalls_s {
        sock_upper_handle_t (*su_newconn)(sock_upper_handle_t,
            sock_lower_handle_t, sock_downcalls_t *);
        void    (*su_connected)(sock_upper_handle_t, sock_gen_t);
        int     (*su_disconnected)(sock_upper_handle_t, sock_gen_t, int);
        void    (*su_opctl)(sock_upper_handle_t, sock_opctl_op_t,
                    sock_opctl_action_t);
        int     (*su_recv)(sock_upper_handle_t, mblk_t *, size_t, int);
        void    (*su_set_prop)(sock_upper_handle_t, struct sock_options *);
        void    (*su_txq_full)(sock_upper_handle_t, boolean_t);
        int     (*su_recv_space)(sock_upper_handle_t);
        void    (*su_signal_oob)(sock_upper_handle_t, size_t);
        void    (*su_recv_oob)(sock_upper_handle_t, mblk_t *, size_t);
        void    (*su_move_msgs)(sock_upper_handle_t, queue_t *);
        void    (*su_set_error)(sock_upper_handle_t, int);
```

```
} sock_upcalls_t;
```

```
typedef struct sock_upcalls_s {
        sock_upper_handle_t (*su_newconn)(sock_upper_handle_t,
            sock_lower_handle_t, sock_downcalls_t *, cred_t *, pid_t);
        void    (*su_connected)(sock_upper_handle_t, sock_connid_t, cred_t *,
            pid_t);
        int     (*su_disconnected)(sock_upper_handle_t, sock_connid_t, int);
        void    (*su_opctl)(sock_upper_handle_t, sock_opctl_action_t,
            uintptr_t);
        ssize_t (*su_recv)(sock_upper_handle_t, mblk_t *, size_t, int,
                    int *, boolean_t *);
        void    (*su_set_proto_props)(sock_upper_handle_t,
                    struct sock_proto_props *);
        void    (*su_txq_full)(sock_upper_handle_t, boolean_t);
        void    (*su_signal_oob)(sock_upper_handle_t, ssize_t);
        void    (*su_zcopy_notify)(sock_upper_handle_t);
        void    (*su_set_error)(sock_upper_handle_t, int);
} sock_upcalls_t;
```

4.5.2.1 su_newconn()

su_newconn() is defined as:

```
sock_upper_handle_t (*su_newconn)(sock_upper_handle_t uh,
            sock_lower_handle_t lh, sock_downcalls_t *dc, struct cred *peer_cred,
            pid_t peer_cpid);
```

su_newconn() is used by the protocol to notify the socket about an incoming connection. The protocol will pass up a lower handle and set of down calls that should be used when sending requests to the new connection. If the upcall is handled successfully, then a new upper handle should be returned to the protocol, which is used to identify the socket of the new connection. The new connection should inherit the upcalls from the listening socket.

4.5.2.2 su_connected()

su_connected() is defined as:

```
void    (*su_connected)(sock_upper_handle_t uh, sock_gen_t gen sock_connid_t id,
                        cred_t * peer_cred, pid_t peer_cpid);
```

And it is issued by the protocol to indicate that a connection has been successfully established. The protocol will pass up a ~~generation~~connection id number that can be used to identify the connection request.

4.5.2.3 su_disconnected()

su_disconnected() is defined as:

```
int     (*su_disconnected)(sock_upper_handle_t uh, sock_gen_t gen sock_connid_t id ,
                           int error);
```

su_disconnected() is issued by the protocol to indicate that either a connection attempt failed, or that an already connected socket has been disconnected. The protocol will pass up a generation number so that a connection attempt can be identified. The protocol will also pass up an error code. If the error code is 0, then it is assumed that the connection was torn down in an orderly fashion. A non-zero value indicate to the protocol that it is safe to release the control block associated with the connection, without waiting for a close downcall.

4.5.2.4 su_opctl()

su_optctl() is defined as:

```
void    (*su_opctl)(sock_upper_handle_t,
                    sock_opctl_op_t op sock_opctl_action_t action,
                    sock_opctl_action_t action); uintptr_t arg);
```

su_opctl is used by the protocol to control the status of specific operations. The protocol can specify an operation and the action that should be taken. The types are defined as:

```
    typedef enum sock_opctl_op {
        SOCK_OPCTL_ACCEPT = 0,
        SOCK_OPCTL_RECV,
        SOCK_OPCTL_SEND
    } sock_opctl_op_t;

    typedef enum sock_opctl_action {
        SOCK_OPCTL_ENABLE = 0,
        SOCK_OPCTL_DISABLE,
        SOCK_OPCTL_SHUTDOWN
    } sock_opctl_action_t;
```

su_opctl is used by the protocol to control the status of specific operations. Operations supported are

```
/*
 * su_opctl() actions
 */
typedef enum sock_opctl_action {
        SOCK_OPCTL_ENAB_ACCEPT = 0,
        SOCK_OPCTL_SHUT_SEND,
        SOCK_OPCTL_SHUT_RECV
} sock_opctl_action_t;
```

4.5.2.5 su_recv()

su_recv() is defined as:

```
int     (*su_recv)(sock_upper_handle_t uh, mblk_t *mp, size_t sz, int flag);
ssize_t so_queue_msg(sock_upper_handle_t sock_handle, mblk_t *mp,
    size_t msg_size, int flags, int *errorp,  boolean_t *force_pushp)
```

su_recv() is used by the protocol to enqueue received data into the socket's receive buffer. The return value is the amount of free buffer space.

4.5.2.6 su_set_proto_props()

su_set_prop() is defined as:

```
void    (*su_set_proto_props)(sock_upper_handle_t,
            struct sock_options sock_proto_props *);
```

su_set_prop() is used by the protocol to notify the socket about property changes. The available properties are:

```
/* Option structure for socket */
struct sock_options {
        uint_t so_flags;                /* options to set */
        ushort_t so_wroff;              /* write offset */
        ssize_t so_rxhiwat;             /* recv high water mark */
        ssize_t so_rxlowat;             /* recv low water mark */
        ssize_t so_txhiwat;             /* transmit high water mark */
        ssize_t so_txlowat;             /* transmit low water mark */
        ssize_t so_maxblk;              /* maximum message block size */
        ssize_t so_maxpsz;              /* maximum packet size */
        uint_t  so_zcopyflag;           /* zero copy flag */
};
```

```
/* Socket protocol properties */
struct sock_proto_props {
        uint_t sopp_flags;              /* options to set */
        ushort_t sopp_wroff;            /* write offset */
        ssize_t sopp_txhiwat;           /* tx hi water mark */
        ssize_t sopp_txlowat;           /* tx lo water mark */
        ssize_t sopp_rxhiwat;           /* recv high water mark */
        ssize_t sopp_rxlowat;           /* recv low water mark */
        ssize_t sopp_maxblk;            /* maximum message block size */
        ssize_t sopp_maxpsz;            /* maximum packet size */
        ushort_t sopp_tail;             /* space available at the end */
        uint_t  sopp_zcopyflag;         /* zero copy flag */
        boolean_t sopp_oobinline;       /* OOB inline */
        uint_t sopp_rcvtimer;           /* delayed recv notification (time) */
        uint32_t sopp_rcvthresh;        /* delayed recv notification (bytes) */
        socklen_t sopp_maxaddrlen;      /* maximum size of protocol address */
};
```

Where sopp_flags is used to indicate which properties are being updated.

```
/* flags for stream options set message */
#define SOCKOPT_WROFF           0x0001  /* set write offset */
#define SOCKOPT_RCVHIWAT        0x0002  /* set read side high water */
#define SOCKOPT_RCVLOWAT        0x0004  /* set read side high water */
#define SOCKOPT_MAXBLK          0x0008  /* set maximum message block size */
#define SOCKOPT_TAIL            0x0010  /* set the extra allocated space */
#define SOCKOPT_ZCOPY           0x0020  /* set/unset zero copy for sendfile */
#define SOCKOPT_MAXPSZ          0x0040  /* set maxpsz for protocols */

/* flags to determine which socket options are set */
#define SOCKOPT_WROFF           0x0001  /* set write offset */
#define SOCKOPT_RCVHIWAT        0x0002  /* set read side high water */
#define SOCKOPT_RCVLOWAT        0x0004  /* set read side high water */
#define SOCKOPT_MAXBLK          0x0008  /* set maximum message block size */
#define SOCKOPT_TAIL            0x0010  /* set the extra allocated space */
#define SOCKOPT_ZCOPY           0x0020  /* set/unset zero copy for sendfile */
#define SOCKOPT_MAXPSZ          0x0040  /* set maxpsz for protocols */
#define SOCKOPT_OOBINLINE       0x0080  /* set oob inline processing */
#define SOCKOPT_RCVTIMER        0x0100
#define SOCKOPT_RCVTHRESH       0x0200
#define SOCKOPT_MAXADDRLEN      0x0400  /* set max address length */
```

4.5.2.7 su_txq_full()

su_txq_full() is defined as:

```
void    (*su_txq_full)(sock_upper_handle_t uh, boolean_t full);
```

su_txq_full() is used by the protocol to notify that flow control on the send side has been set or cleared. Nothing is returned by this call.

4.5.2.8 ~~su_recv_space()~~

~~su_recv_space() is defined as:~~

```
int     (*su_recv_space)(sock_upper_handle_t);
```

~~And it returns the amount of space is available in the receive buffer.~~

4.5.2.9 su_signal_oob()

su_signal_oob() is defined as:

```
void    (*su_signal_oob)(sock_upper_handle_t uh, ssize_t offset);
```

su_signal_oob() is used by the protocol to notify the socket that out-of-band data is pending. An offset into the data stream where the data will be found is given. Nothing is returned.

4.5.2.10 ~~sd_recv_oob()~~

~~su_recv_oob() is defined as:~~

```
~~void    (*su_recv_oob)(sock_upper_handle_t uh, mblk_t *mp, size_t sz);~~
```

~~su_recv_oob() is used by the protocol to enqueue out-of-band data into the socket's receive buffer. Nothing is returned.~~

4.5.2.11 su_zcopy_notify()

su_zcopy_notify() is defined as:

```
void su_zcopy_notify(sock_upper_handle_t sock_handle);

    This is used by the protocol to signal a thread waiting in so_zcopy_wait() that
it's done with the buffer.
```

4.5.2.12 su_set_error()

su_set_error() is defined as:

```
void    (*su_set_error)(sock_upper_handle_t, int);
```

su_set_error() is used by the protocol to indicate that a transient error has occurred. Nothing is returned by this call.

4.5.2.13 ~~su_move_msgs()~~

~~su_move_msgs() is defined as:~~

```
~~void    (*su_move_msgs)(sock_upper_handle_t uh, queue_t *q);~~
```

~~And it is used by the protocol during fallback to notify the socket that the is protocol quiescent, and it is therefore safe to move data over to the provided queue. Nothing is returned.~~

4.5.3 ~~Registration of up and down calls~~

~~When a socket implementation uses the standard framework to create a socket its create function calls socket_create_common() to create the socket.~~

```
~~struct sonode *
socket_create_common(vnode_t *accessvp, int domain, int type, int protocol,
    int version, struct sonode *tso, struct vnodeops *socket_vnodeops,
    struct sonodeops *sonodeops, sock_upcalls_t *sock_upcalls,
    sock_downcalls_t *sock_downcalls, int sflags, int *errorp)~~
```

~~socket_sop_create() takes as input a pointer to up call functions (sock_upcalls) and a pointer to down call functions (sock_downcalls). The framework stores the down call pointer in the socket and passes the upcall pointer to the protocol via the create function specified in the down calls.~~

~~In most cases, the generic implementation of socket_vnodeops, sonodeops and sock_upcalls that will be provided should be sufficient, however an implementation can use its own implementation for any or all of these function pointers.~~

~~Currently there is no mechanism to override a specific function within the set.~~

4.5.4 How the structures connect

The following figure shows the relationship of various structures for TCP .

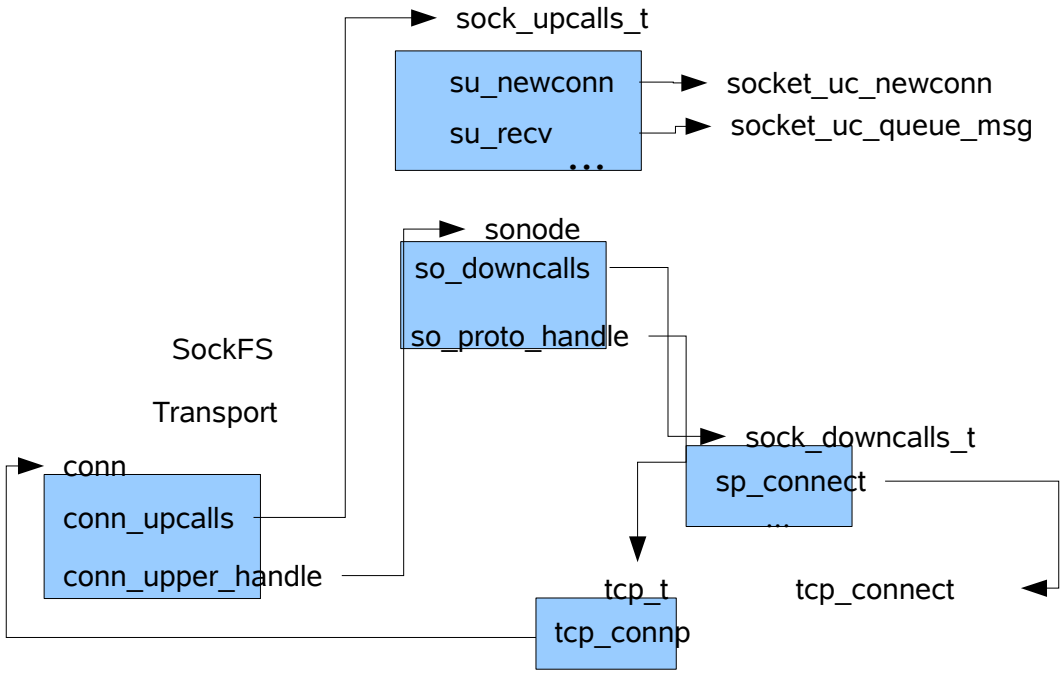


Figure 3: How the structures connect

4.6 Zero-Copy Interface

In the context of this document, zero-copy refers only to the ability of submitting data buffers to a protocol for transmission without having the data copied from those buffers into the protocol (i.e., the interface will not do any VM mapping, which is left to the consumer). The intension is that higher level functionality (e.g., *sendfilev(3EXT)*) will be built on top of the interface.

4.6.1 Background Information

sendfilev(3EXT) was introduced in PSARC/2000/279 and provide applications with a method for doing bulk data transmit without incurring the cost of a user-to-kernel copy. s*endfilev(3EXT)* takes care of all the data set up, as it must, but it also verifies socket state Having *sendfilev(3EXT)* take care of the whole zero-copy procedure worked before as it was the only consumer of the functionality, and TCP was the only supported protocol. However, with the introduction of Volo, there is now a need for a generic socket interface for supporting zero-copy for two reasons:

(1) A kernel socket API will be delivered and some modules would greatly benefit from having access to a zero-copy interface.

(2) Socket modules will allow 3rd party developers to add non-STREAMS protocols, some of which may provide zero-copy functionality.

4.6.2 Details of sendfilev(3EXT)

sendfilev(3EXT) is currently the only interface that utilize zero-copy, and would initially be the main consumer of the interface. The intent is to have *sendfilev(3EXT)* only handle the buffer allocation/setup strategy, and have use the zero-copy interface to submit the buffers to the protocol for transmission.

The current buffer allocation/setup strategy of *sendfilev(3EXT)* is based upon the size of the file transfer. When the file size is smaller than a certain threshold, buffers are allocated and data is read in from the file system into those buffers. For files that are larger than the threshold the data is segmapped. In both cases mblks (*msgb(9S)*) are used. In the case of small files, the size of the mblk being allocated is based upon the maximum mblk value, which would normally correspond to the MSS of the interface in use. In addition, enough space is left to insert any necessary headers and trailers.

In the case of large files, the data is normally segmapped, and *desballoc(9F)* is used to allocate a mblk that wraps around the segmapped buffer. A callback function is passed to *desballoc(9F)*, which releases the mapping when the mblk is being freed. Since the file is locked while the mapping is in place, segmap is only used if there is a guarantee that mblks will be freed in a timely manner. The mblks that have such requirements are marked with a *STRUIO_ZC* flag before being sent to the protocol. Freeing the buffers could normally be delayed due to the protocol waiting for acknowledgments, or the driver could be doing a lazy reclaim. *sendfilev(3EXT)* use a private socket option (*SO_SND_COPYAVOID)* to verify that buffers will be freed in a timely manner.

In addition to the callback provided by *desballoc(9F)*, *sendfilev(3EXT)* marks the last mblk with a flag (*STRUIO_ZCNOTIFY)* indicating to TCP that when the mblk if freed, it should signal a condition variable stored in the STREAM head. The *sendfilev(3EXT)* use the condition variable to ensure that the operation does not return to the user until all mapped pages have been released.

4.6.3 Interface

For *sendfilev(3EXT)* to continue the strategy described in the previous section, we have the following requirements:

(1) The interface must provide a way of granting the socket ownership of a buffer, with the intent that the data is queued for transmission without any additional copying.

(2) The consumer of the interface should be able to allocate an ideal sized buffer.

(3) When VM mapped buffers, or some other “limited” resource is submitted for transmission, the protocol must ensure that the buffers are not held on to for a long time. If the protocol cannot make such a guarantee, then it is expected to copy and free the original buffers.

(4) The interface must provide a method of querying the protocol whether "limited” resource buffers are likely to be freed in a timely manner.

The first requirement is handle by a new function call (Table 3), which the consumer use to submit a mblk chain for transmission. The already existing flag (*STRUIO_ZC)* is used to indicate that the mblk chain must be freed in a timely manner. Either all mblks are marked with the zero-copy flag, or none. There is no support for mixed buffer types. The consumer is expected to use *desballoc(*9F) or *esballoc(9F)* to get notified about buffers being freed. Information regarding how the mblks should be sized is extracted via a new socket option (Table 4).

```
int socket_sendmblk(struct sonode *so, struct nmsghdr *msg, int fflag,
       struct cred *cr, mblk_t *mpp);
```

Table 3: Function to submit mblks for transmission

```
#define SO_SND_BUFINFO 0x1000

struct so_snd_bufinfo {
        ushort_t sbi_wroff; /* write offset */
        ssize_t sbi_maxblk; /* max size of a single mblk */
        ssize_t sbi_maxpsz; /* max total size of a mblk chain */
        ushort_t sbi_tail;  /* extra space available at the end */
};
```

Table 4: Socket option to retrieve buffer info

The consumer can determine whether a protocol can handle zero-copy marked buffers without having to immediately free the mblk by using the existing *SO_SND_COPYAVOID* socket option.

As mentioned in the previous section, *sendfilev(3EXT)* use a separate flag (*STRUIO_ZCNOTIFY)* to get notified when the last mblk is freed. Although the technique works well with TCP, as result of cumulative acknowledgments, it might not apply to all protocols. That is, it might not be the case that if buffer *X+1* is freed, then *X* must have been freed as well.

So instead of placing a hard requirement on the protocol, it is up to the consumer to build a way of determining the completion of an operation. For example, the callback associated with each mblk can take as an argument a global state, which maintain a count of the number of mblks sent as part of the operation, and which is decremented each time a callback is made. When the count reaches zero, the operation has completed and a condition variable of the consumers' choice can be signaled.

The above mentioned interfaces are used by the kernel socket API to provide zero-copy functionality. More details is given in Section 8.3.

Chapter - 5 Interface Evolution

5.0 Interface Evolution

This chapter describes how Volo allows the interface to evolve while providing backward compatibility.

5.1 Restrictions

~~In order to provide backward compatibility module writers should not allocate sonode locally, socket_create_common() should be used instead,. There should be no code that is dependent on the sizeof() the sonode. This allows changing the size of sonode.~~

~~Any module specific information is stored in a module specific structure a pointer to which is stored in so_priv. It's up to the module writer to provide backward compatibility for this structure.~~

Module writers should not depend on the value of sizeof() sonode.

5.2 Version number

~~When registering a module a version number is supplied by the module. This version number will be used to determine which upcalls, downcalls and sonodeops combination the module supports. Sockfs will internally switch to using the appropriate functions.~~

~~Volo allows a module writer to have full control by providing it's own upcalls, downcalls amd sonodeops. In such an implementation the module writer is responsible for providing backward compatibility. However since Volo's sockfs layer is very thin we expect that in most cases system default implementation will be used.~~

When a socket module registers it supplies three versions in smod_reg_t. smod_version is the major number and should be initialized via the macro SOCKMOD_VERSION. smod_uc_version and smod_dc_version represent the version of the upcalls and downcalls that the module is coded to support. These should be initialized via macros SOCK_UC_VERSION and SOCK_DC_VERSION.

Based on this information the socket framework can provide backward compatability and if it can not the registeration will fail.

```
/*
 * Socket module register information
 */
typedef struct smod_reg_s {
int             smod_version;
char            *smod_name;
size_t          smod_uc_version;
size_t          smod_dc_version;
so_proto_create_func_t  smod_proto_create_func;

/*  smod_priv_data must be NULL */
smod_priv_t   *  smod_priv;
} smod_reg_t;
```

5.3 Current Status

Even though the design allows for backward compatibility, This functionality has not been coded.

Chapter - 6 IP Related Changes

6.0 IP Related Changes

This chapter describes IP related changes made to Solaris in order to support non streams based sockets.

6.1 IP Helper Streams

As pointed out in section 2.3, one of the goals of Project Volo was to implement non streams based sockets for IP based protocols. During the development it was realized that this goal can not be achieved without rewriting IP because IP is heavily dependent on a one to one relationship of stream queues and conn_t and it uses one to get to the other. Furthermore, request/packet processing in IP is asynchronous and there is no simple way to provide synchronous semantics for Volo sockets than what streams currently provides. Rewriting IP was out of the scope of Project Volo, so the only solution that was architecturally clean and resolved all issues was to use streams. Hence helper streams are used to access IP.

IP helper streams are used in the following cases

- Sending data to IP: ip_output_options() assumes that it can access conn_t via the queue passed in and vice versa, so a single queue per protocol could not be used.
- Passing ioctls to IP: Ioctls that can not be handled by the protocol are passed to IP using the layered driver interface ldi_ioctl(9F). Use of helper stream is necessary as lookup of ILL/IPIF 's may result in the request being queued for later processing, so streams framework is used to match synchronous request with asynchronous processing.
- Passing socket options request to IP: Certain socket options requests require processing in IP. These requests are passed to IP using two new interfaces ip_set_options() and ip_get_options(). These interfaces use layered interfaces ldi_putmsg(9F) and ldi_getmsg(9F) on the helper stream. The reason for using helper streams is same as explained above for ioctl's.

IP helper streams are created for IP based protocol sockets at socket creation time by calling

```
int ip_create_helper_stream(conn_t *connp, ldi_ident_t li);
```

This creates the helper stream and sets q_ptr's to point to conn_t and conn_rq and conn_wq to point to the appropriate queues. A new open flag IP_HELPER_STR flag is passed in to ip_open() to identify a helper stream open. This results in certain specific actions, for example conn_t is not allocated.

In the case of TCP accept, a helper stream is created when the connection is accepted not when the SYN comes in. This allows sharing of conn_t from tcp_free_list between streams based and non-streams based sockets.

IP helper streams are closed at socket close by calling

```
int ip_close_helper_stream(conn_t *connp);
```

6.2 IP Bind and Connect Changes

Currently IP based protocols call the following IP functions to perform bind and connect operations

```
void ip_bind_v4(queut_t *q, mblk_t *mp, conn_t *connp);

void ip_bind_v6(queue_t *q, mblk_t *mp, conn_t *connp,
       ip6_pkt_t *ipp);
```

Even though these functions are called directly they operate on TPI messages. Volo sockets do not pass TPI messages, so these functions have been replaced by

- For bind operations

```
int ip_proto_bind_laddr_v4(conn_t *connp,
          mblk_t **ire_mpp, uint8_t protocol,
          ipaddr_t src_addr, uint16_t lport,
          boolean_t fanout_insert);

int ip_proto_bind_laddr_v6(conn_t *, mblk_t **,
           uint8_t, const in6_addr_t *, uint16_t,
           boolean_t);
```

- For connect operations

```
int ip_proto_bind_connect_v4(conn_t *, mblk_t **,
          uint8_t, ipaddr_t *, uint16_t, ipaddr_t,
             uint16_t, boolean_t, boolean_t);

int ip_proto_bind_connected_v6(conn_t *, mblk_t **,
                uint8_t, in6_addr_t *, uint16_t,
                const in6_addr_t *, ip6_pkt_t *, uint16_t,
                boolean_t, boolean_t);
```

Upon success these functions return zero. Since these functions are used by both TLI and non-TLI protocol code the error returned must differentiate between Unix and TLI error code. Unix errors are returned as positive integers while TLI errors are returned as negative integers. TLI errors can be converted to unix by passing the absolute TLI error to the following function.

```
int mi_tlitosyserr(int terr);
```

The address supplied for binding has to be a legitimate address hosted on the system. The address could be hosted on a down interface. To check for addresses hosted on down interfaces Solaris currently walks the ill/ipif list. If an ipif can not be looked up the request is queued and EINPROGRESS is returned. This is not an option for Volo as operations in Volo must be synchronous.

Bind semantics are such that it is ok to use the ipif address for verification even if the ipif is marked IPIF_CHANGING or IPIF_CONDEMNED. The following new functions have been introduced to do this comparison. They return B_TRUE if there is a match.

```
boolean_t ipif_lookup_addr_exists(ipaddr_t addr,
     zoneid_t zoneid, ipstack_t *ipst);

boolean_t ipif_lookup_addr_exists_v6(const in6_addr_t addr,
    zoneid_t zoneid, ipstack_t *ipst);
```

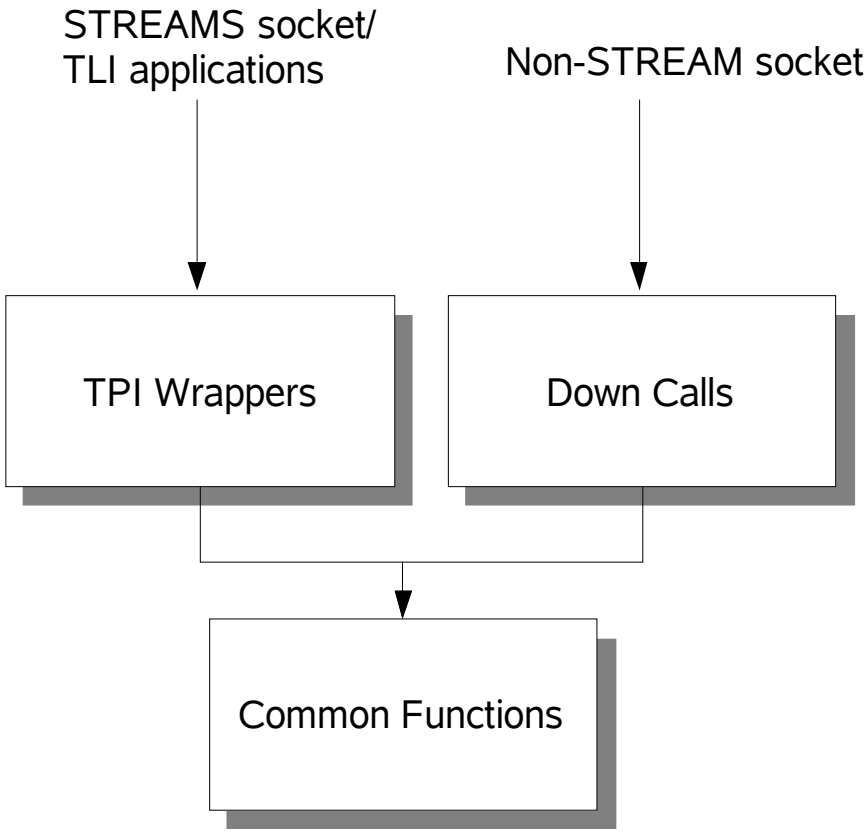
Chapter - 7 Protocol Layer Changes

7.0 Protocol Layer Changes

This chapter describes the changes made to protocols in order to support both TPI and non TPI based sockets.

7.1 Accessor Functions

Protocol code has been factored out such that for every socket related operation there is a common function that does the actual work. The common function is accessed via an accessor function that deals with the details of how the protocol is being accessed. For example in the case of TPI, the accessor function extracts the data from the message and calls the common function. The result is packaged in the right message format and returned. The following figure gives a graphical representation of the layout.



7.2 Socket Data Caching

In the current implementation, Solaris socket layer caches some data such as socket state, socket options and local and remote addresses.

One of the design goals of Project Volo was to cache as little information as possible in the socket. While some socket state is still cached in the socket, socket options are now completely stored in the protocol only, and set/get socket option operations are performed in the protocol. Local and remote addresses are also no longer cached in the socket and getpeername()/getsockname() get the address info from the protocol.

7.3 Synchronous Access to Squeue

Squeues are used by TCP to provide exclusive access to TCP data structures. Currently entering an squeue is asynchronous. If a thread tries to enter an squeue and cannot, the operation is queued and NULL is returned. If the operation will result in a reply, the calling thread may block waiting for the reply message, which is sent up when the operation is performed by some thread draining the squeue. Volo sockets are synchronous and do not wait for a reply message, so for Volo sockets squeue enter has to be synchronous.

Volo has introduced the following synchronous access function for squeues

```
int squeue_synch_enter(squeue_t *sqp, void *arg,
  uint8_t tag);
```

Given an squeue pointer (sqp) this function guarantees exclusive access to the squeue if the return value is zero.

The function checks the ownership of the squeue. If the squeue is available, it sets the calling thread to be the owner and returns. However if another thread owns the squeue a message is enqueued and the thread trying to enter the squeue is blocked using cv_wait()[1]. When the enqueued message is processed by the thread draining the squeue, the draining thread is suspended and the synchronous enter thread that was blocked is woken up.

When a thread that entered the squeue synchronously is finished using the squeue, it should call the following function to release the ownership of the squeue

```
void squeue_synch_exit(squeue_t *sqp, void *arg);
```

If a draining thread was suspended to give control to the synchronous thread, that thread is woken up and continues to drain the squeue. If there was no draining thread but some messages were queued while the synchronous thread was using the squeue, the squeue worker thread is signaled to drain the squeue.

7.4 All Received Data is Queued in the Socket

In the current Solaris socket implementation the received data is stored in different places depending upon the implementation of that particular socket. For example, UDP stores received data in the protocol if the socket is a direct socket or else in the stream head. TCP enqueues data in two different places. First it queues data in the tcp_rcv_list and once a threshold is reached or the push bit is set the data is pushed to the stream head.

In Volo all socket types store received data in the socket. The only exception to this is when the data arrives before accept finishes. In this case the data is stored in the protocol because a socket has not yet been assigned to the TCP structure. As part of accept completion the data is pushed to the socket.

In our testing we observed that for TCP making the data available as soon as it arrives results in (MTU – header overhead) byte reads. These small reads are inefficient and result in low throughput. To achieve better performance Volo framework applies current TCP rules of delaying received data indication so multiple data segments can be combined in one read. The only change made to this approach is that the read is satisfied as soon as the data queued can fill the read buffer.

1sowaitprim() also uses cv_wait()

Chapter - 8 Kernel Socket API

8.0 Introduction

There is currently no standard interface for creating sockets within the kernel. However, it is necessary for some kernel modules to create network connections (e.g., network file systems) and that is handled today through private interfaces and contracts that provide direct access to (unstable) kernel functions. Without a kernel socket interface it will remain challenging for developers to introduce new technologies into Solaris/OpenSolaris that depend upon the ability of doing some networking within the kernel. In fact, there have already been several requests from the OpenSolaris community for a kernel sockets interface. Many developers already have expectations that a kernel sockets interface is part of Solaris, because most other operating systems (e.g., FreeBSD, Windows) already provide similar interfaces.

8.1 Base Interface

A kernel socket is represented by an opaque data type (*ksocket_t*), which should only be manipulated by the kernel socket functions (Table 5). Volo provides a set of standard API for kernel socket manipulation. Most of the user-land socket functions have a kernel socket equivalent. For example, creating a new socket is done using *ksocket_socket()*, and a typical usage scenario would be:

```
ksocket_t ksock;

if ((error = ksocket_socket(AF_INET, SOCK_STREAM , 0,
 KSOCKET_SLEEP, credp, &ksock)) != 0)
    goto fail;
 ...
```

The above code fragment highlights one of the main differences from user-land interface: the return value is only used to report whether the operation was successful. If operation failed, the functions returns an *errno* value indicating what caused the failure, otherwise 0 is returned. For those user-land functions that return more than a success/failure indication (such as, *socket(3SOCKET)*, *recv(3SOCKET),* etc.), the kernel socket equivalent take an additional result parameter.

Another difference from the user-land interface is that every kernel socket function take a *cred_t* pointer argument. Passing in the credentials allows the consumer to multiplex requests with different credentials over a single kernel socket. The consumer can also temporarily increase the access rights to allow, for example, binding to a privileged port.

Besides the *cred_t* argument, *kernel_recvmsg()* and *kernel_sendmsg()* also take an additional argument which is a *uio* data structure pointer. So if the kernel socket consumer has a complete *uio* structure, obtain for example from *read(9E)* or *write(9E)*, it can be passed in directly. The *uio* argument can also be set to *null,* it which case the *msghdr* structure must contain a pointer to a valid *iovec* structure.

When a kernel socket is created, via *ksocket_create*() or *ksocket_accept*(), there is a single reference placed on the socket. A call to *ksocket_close*() will normally disconnect and free the kernel socket. However, if multiple threads will be accessing the kernel socket, the consumer should place an additional reference per thread to ensure that kernel socket is not freed while still in use. Adding a references is done via *ksocket_hold(),* and released via *ksocket_rele()*. A call to *ksocket_close()* will interrupt any pending operations and cause future operations to fail, but the kernel socket is not freed until the last reference is dropped. The consumer can register a callback function to get notified when the socket is about to be freed. More details regarding callback

functions will be given in the next section.

```
int      ksocket_socket(ksocket_t *s, int domain, int type, int protocol,
             int flags, struct cred *cr);
int      ksocket_bind(ksocket_t s, struct sockaddr *addr, socklen_t addrlen,
             struct cred *cr);
int      ksocket_listen(ksocket_t s, int backlog, struct cred *cr);
int      ksocket_accept(ksocket_t s, struct sockaddr *addr,socklen_t *addrlen,
             struct cred *cr, ksocket_t *ns);
int      ksocket_connect(ksocket_t s, const struct sockaddr *addr,
             socklen_t addrlen, struct cred *cr);
int      ksocket_send(ksocket_t s, void *msg, size_t msgsize, int flags,
             struct cred *cr, size_t *sent);
int      ksocket_sendto(ksocket_t ks, void *msg, size_t msglen, int flags,
             struct sockaddr *name, socklen_t namelen, struct cred *cr,
             size_t *sent);
int      ksocket_sendmsg(ksocket_t s, struct nmsghdr *msg, struct uio *auio,
             int flags, struct cred *cr);
int      ksocket_recv(ksocket_t s, void *msg, size_t msgsize, int flags,
             struct cred *cr, size_t *recv);
int      ksocket_recvfrom(ksocket_t s, void *msg, size_t msglen, int flags,
             struct sockaddr *name, socklen_t *namelen, struct cred *cr,
             size_t *recv);
int      ksocket_recvmsg(ksocket_t s, struct nmsghdr *msg, struct uio *auio,
             int flags, struct cred *cr);
int      ksocket_shutdown(ksocket_t s, int how, struct cred *cr);
int      ksocket_setsockopt(ksocket_t s, int level, int optname,
              const void *optval, int optlen, struct cred *cr);
int      ksocket_getsockopt(ksocket_t s, int level, int optname, void *optval,
              int *optlen, struct cred *cr);
int      ksocket_getpeername(ksocket_t s, struct sockaddr *addr,
              socklen_t *addrlen, struct cred *cr);
int      ksocket_getsockname(ksocket_t s, struct sockaddr *addr,
              socklen_t *addrlen, struct cred *cr);
int      ksocket_ioctl(ksocket_t s, int cmd, intptr_t arg, struct cred *cr,
              int *rvalp);
int      ksocket_close(ksocket_t s, struct cred *);

int      ksocket_setcallbacks(ksocket_t s, ksocket_callbacks_t *cb,
             void *arg, struct cred *cr);

int      ksocket_hold(ksocket_t s);
int      ksocket_rele(ksocket_t s);
```

Table 5: Kernel Socket Interface

8.2 Event Notification

Although the basic kernel socket interface is reminiscent of user-land API, event notification is handled quite differently. For user-land applications there are multiple general purpose systems that can be used to determine whether an event has occurred on a socket (e.g., *poll(2), port_get(3C)*, etc.) However, there exists no such general purpose notification system in the kernel. Instead, kernel socket events are reported asynchronously via callback functions.

8.2.1 Data Structures

Eleven different events have been defined, and for each event a callback function can be registered. An enum type is used to represent these events (Table 6). All the callback functions have the same prototype (Table 7), where the first argument is the kernel socket on which the event took place, and the second argument identifies the event. The third argument is a user cookie. Depending on the event, the fourth argument may carry some additional information. For *KSOCKET_EV_NEWDATA* the argument specifies how many bytes are queued, and for *KSOCKET_EV_DISCONNECTED* the argument indicates the reason why the socket was disconnected.

```
typedef enum ksocket_event {
        KSOCKET_EV_CONNECTED,
        KSOCKET_EV_DISCONNECTED,
        KSOCKET_EV_OOBDATA,
        KSOCKET_EV_NEWDATA,
        KSOCKET_EV_NEWCONN,
        KSOCKET_EV_CANSEND,
        KSOCKET_EV_CANTSENDMORE,
        KSOCKET_EV_CANTRECVMORE,
        KSOCKET_EV_ERROR,
        KSOCKET_EV_PROPCHANGED
        KSOCKET_EV_RELEASED
} ksocket_callback_event_t;
```

Table 6: Kernel Socket Events (ksocket_callback_event_t)

```
typedef void (*ksocket_callback_t)(ksocket_t, ksocket_callback_event_t, void *,
                uintptr_t);
```

Table 7: Kernel Socket callback function (ksocket_callback_t)

We defined a data type (*ksocket_callbacks_t*) to manage the callback functions (Table 8). In the structure there exists one *ksocket_callback_t* entry per event. Besides the function pointers, there is also a flag, which is used to indicate which callback functions are valid (Table 9).

```
typedef struct ksocket_callbacks {
        uint32_t                ksock_cb_flags;
        ksocket_callback_t      ksock_cb_connected;
        ksocket_callback_t      ksock_cb_disconnected;
        ksocket_callback_t      ksock_cb_newdata;
        ksocket_callback_t      ksock_cb_newconn;
        ksocket_callback_t      ksock_cb_cansend;
        ksocket_callback_t      ksock_cb_oobdata;
        ksocket_callback_t      ksock_cb_cantsendmore;
        ksocket_callback_t      ksock_cb_cantrecvmore;
        ksocket_callback_t      ksock_cb_error;
        ksocket_callback_t      ksock_cb_propchanged;
} ksocket_callbacks_t;
```

Table 8: Kernel Socket callback table (ksocket_callbacks_t)

```
#define KSOCKET_CB_CONNECTED            0x00000001
#define KSOCKET_CB_DISCONNECTED         0x00000002
#define KSOCKET_CB_NEWDATA              0x00000004
#define KSOCKET_CB_NEWCONN              0x00000008
#define KSOCKET_CB_CANSEND              0x00000010
#define KSOCKET_CB_OOBDATA              0x00000020
#define KSOCKET_CB_CANTSENDMORE         0x00000040
#define KSOCKET_CB_CANTRECVMORE         0x00000080
#define KSOCKET_CB_ERROR                0x00000100
#define KSOCKET_CB_PROPCHANGED          0x00000200
```

Table 9: Bit values for ksock_cb_flags

For a particular kernel socket, the definition of *ksocket_callbacks_t* is in the *ksocket_t* data structure. Moreover, there is an additional variable called *so_ksock_cb_arg* in *ksocket_t* data structure. This variable is a void type pointer, which is used to store the user cookie. Figure 4 shows the relationship of these data structures.

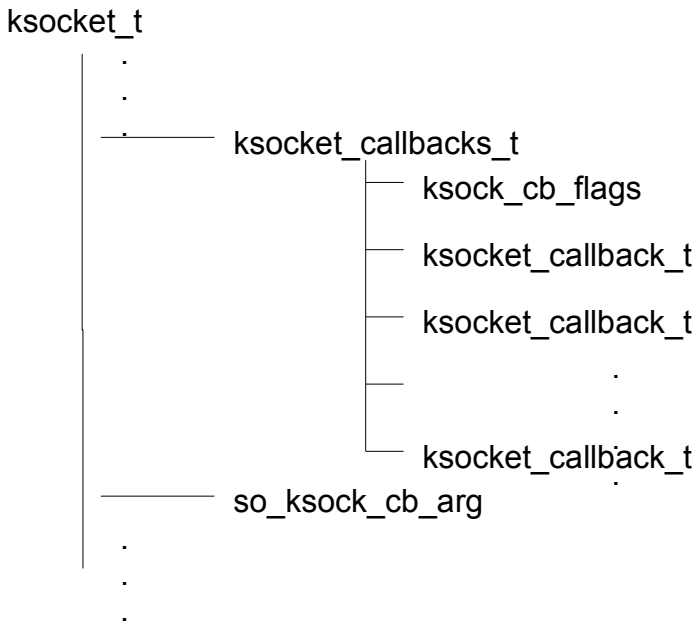


Figure 4: Relationship of Data Structures

8.2.2 Operations

ksocket_setcallbacks() (see Table 5) is used to set callback functions. The second argument, *cb,* contains the callback function information which the user want to set. If *cb* is *null,* it means clear all the callback functions for the specified kernel socket. Otherwise, set the callback function in the kernel socket according to the information in *ksock_cb_flags*. Specifically, for each bit (Table 9) that is set, update the kernel socket callback with the corresponding function in *cb*. The mechanisms allows the consumer to control a subset of the available callbacks. The third argument, *arg,* is a user cookie and will be passed as an argument when a callback function is triggered.

8.2.3 Usage of Callback Functions

The callback functions are expected to do a minimal amount of processing, for example, it could signal another thread or dispatch a task queue to do intensive processing. Callback functions are explicitly not allowed to:

- Perform any blocking operations
- Use any kernel socket functions (except for *ksocket_hold()* and *ksocket_rele())*

The above constraints could have been avoided if the callback function was directly dispatch on a task queue. However, such a design would not work well in certain scenarios. For example, if the callback function is only responsible for waking another thread, then the design would introduce additional latency. Another issue is the additional complexity that would be introduced just to handle errors. Many errors could also be seen as policy based, for example, if event function could not be dispatch, should the event be discarded? The current design gives the consumer more flexibility, without much additional work.

When a new connection is accepted, it does not inherit the callback functions or user cookie from the listener. The main motivation for having new connections inheriting the callbacks would be to avoid missing any events, however, events could occur at a stage where callbacks cannot be generated (e.g., before the kernel socket has been accepted). So instead the consumer must install callbacks manually after *ksocket_accept()* returns. Once callbacks have been installed, the consumer can check for “missed” events by

doing a *ksocket_recv()*.

8.3 Zero-copy Interface

The zero-copy interface for kernel socket is similar to the one defined in Section 4.6. That is, the *SO_SND_COPYAVOID* socket option is used to determine zero-copy support, and *SO_SND_BUFINFO* is used to extract buffer sizing information. It is possible to get notifications when either the zero-copy capabilities or buffer info change by enabling the *KSOCKET_EV_PROPCHANGED* callback. The callback function indicate which property changed by passing in the option (e.g., *SO_SND_COPYAVOID)*.

The consumer is expected to allocate mblks and submit the buffers using the kernel socket zero-copy send function (Table 10). The main difference is that the caller explicitly state whether the mblk chain must be released in a timely manner.

```
int ksocket_sendmblk(ksocket_t ks, struct nmsghdr *msg, struct cred *cr,
    boolean_t segmapped, mblk_t *mpp);
```

Table 10: ksocket_sendmblk

8.4 Limitations

The interface provides the required functionality to create AF_INET and AF_INET6 sockets. AF_UNIX sockets are not supported. In addition, there is no service for resolving host names (i.e., *getaddrinfo(3SOCKET), gethostbyname(3NSL)*).

Chapter - 9 Appendix

9.0 Exported Interfaces

9.0.1 Sockfs Related Interfaces

Socket Related Interfaces Exported		
Interface	Classification	Comments
socket_create	Project Private	See [1] below
socket_newconn	Project Private	See [1] below
socket_bind	Project Private	See [1] below
socket_listen	Project Private	See [1] below
socket_accept	Project Private	See [1] below
socket_connect	Project Private	See [1] below
socket_sendmsg	Project Private	See [1] below
socket_sendmblk	Project Private	See [1] below
socket_recvmsg	Project Private	See [1] below
socket_getpeername	Project Private	See [1] below
socket_getsockname	Project Private	See [1] below
socket_getsockopt	Project Private	See [1] below
socket_setsockopt	Project Private	See [1] below
socket_ioctl	Project Private	See [1] below
socket_close	Project Private	See [1] below
socket_destroy	Project Private	See [1] below
socket_poll	Project Private	See [1] below
socket_shutdown	Project Private	See [1] below
SOCKET_SLEEP	Project Private	See [1] below
SOCKET_NOSLEEP	Project Private	See [1] below
sock_lower_handle_t	Consolidation Private	<sys/socket_proto.h>
sock_upper_handle_t	Consolidation Private	<sys/socket_proto.h>
sock_upcalls_t	Consolidation Private	<sys/socket_proto.h>

| sock_downcalls_t | Consolidation Private | <sys/socket_proto.h>
|
| struct sock_options | Consolidation Private | <sys/socket_proto.h>
|
| sock_gen_t | Consolidation Private | <sys/socket_proto.h>
|
| SOCK_GEN_INIT | Consolidation Private | <sys/socket_proto.h>
|
| SOCK_GEN_BUMP | Consolidation Private | <sys/socket_proto.h>
|
| SOCK_GEN_LT | Consolidation Private | <sys/socket_proto.h>
|
| sock_opctl_op_t | Consolidation Private | <sys/socket_proto.h>
|
| sock_opctl_action_t | Consolidation Private | <sys/socket_proto.h>
|
| so_sonodeops | Consolidation Private | <sys/socket_proto.h>
|
| so_upcalls | Consolidation Private | See [1] below
|
| so_vnodeops | Consolidation Private | See [1] below
|
| socket_create_common | Consolidation Private | See [1] below
|
| socket_destroy_common | Consolidation Private | See [1] below
|
| smod_reg_t | Consolidation Private | See [1] below
|

--

Notes:
[1] <fs/sockfs/sockcommon.h>

9.0.2 Kernel Socket Related Interfaces

--
| Kernel Socket Related Interfaces Exported
|

--
| Interface | Classification | Comments
|

--
| ksocket_socket | Consolidation Private | <sys/ksocket.h>
|
| ksocket_bind | Consolidation Private | <sys/ksocket.h>
|
| ksocket_listen | Consolidation Private | <sys/ksocket.h>
|
| ksocket_accept | Consolidation Private | <sys/ksocket.h>
|
| ksocket_connect | Consolidation Private | <sys/ksocket.h>
|
| ksocket_send | Consolidation Private | <sys/ksocket.h>
|
| ksocket_sendto | Consolidation Private | <sys/ksocket.h>
|
| ksocket_sendmsg | Consolidation Private | <sys/ksocket.h>
|
| ksocket_recv | Consolidation Private | <sys/ksocket.h>
|

ksocket_recvfrom	Consolidation Private	<sys/ksocket.h>
ksocket_recvmsg	Consolidation Private	<sys/ksocket.h>
ksocket_getpeername	Consolidation Private	<sys/ksocket.h>
ksocket_getsockname	Consolidation Private	<sys/ksocket.h>
ksocket_getsockopt	Consolidation Private	<sys/ksocket.h>
ksocket_setsockopt	Consolidation Private	<sys/ksocket.h>
ksocket_ioctl	Consolidation Private	<sys/ksocket.h>
ksocket_close	Consolidation Private	<sys/ksocket.h>
ksocket_setcallbacks	Consolidation Private	<sys/ksocket.h>
ksocket_hold	Consolidation Private	<sys/ksocket.h>
ksocket_rele	Consolidation Private	<sys/ksocket.h>
ksocket_t	Consolidation Private	<sys/ksocket.h>
ksocket_callbacks_t	Consolidation Private	<sys/ksocket.h>
ksocket_event_t	Consolidation Private	<sys/ksocket.h>
KSOCKET_CB_CONNECTED	Consolidation Private	<sys/ksocket.h>
KSOCKET_CB_DISCONNECTED	Consolidation Private	<sys/ksocket.h>
KSOCKET_CB_NEWDATA	Consolidation Private	<sys/ksocket.h>
KSOCKET_CB_NEWCONN	Consolidation Private	<sys/ksocket.h>
KSOCKET_CB_OOBDATA	Consolidation Private	<sys/ksocket.h>
KSOCKET_CB_CANSEND	Consolidation Private	<sys/ksocket.h>
KSOCKET_CB_CANTSENDMORE	Consolidation Private	<sys/ksocket.h>
KSOCKET_CB_CANTRECVMORE	Consolidation Private	<sys/ksocket.h>
KSOCKET_CB_ERROR	Consolidation Private	<sys/ksocket.h>
KSOCKET_CB_PROPCHANGED	Consolidation Private	<sys/ksocket.h>
KSOCKET_CB_RELEASED	Consolidation Private	<sys/ksocket.h>
KSOCKET_SLEEP	Consolidation Private	<sys/ksocket.h>
KSOCKET_NOSLEEP	Consolidation Private	<sys/ksocket.h>

9.1 Sonode

```
struct sonode {
        struct  vnode   *so_vnode;      /* vnode associated with this sonode */

        sonodeops_t     *so_ops;        /* operations vector for this sonode */
        krwlock_t       so_fallback_lock;

        kmutex_t        so_lock;        /* protects sonode fields */
        kcondvar_t      so_state_cv;    /* synchronize state changes */
        kcondvar_t      so_connind_cv;  /* wait for T_CONN_IND */
        kcondvar_t      so_want_cv;     /* wait due to SOLOCKED */

        /* These fields are protected by so_lock */
        uint_t  so_state;               /* internal state flags SS_*, below */
        uint_t  so_mode;                /* characteristics on socket. SM_* */

        sock_gen_t      so_proto_gen;   /* protocol generation number */

        /* Accept queue */
        /* XXX-anders; tmp */
#define so_acceptq_len  so_rcv_queued
/*      int             so_acceptq_len; */
        /* XXX-anders move conn_ind_{head,tail} to sotpi_info_t */
        mblk_t          *so_conn_ind_head;      /* b_next list of T_CONN_IND */
        mblk_t          *so_conn_ind_tail;
        /* XXX-anders; tmp */
#define so_acceptq_cv   so_connind_cv
        struct sonode   *so_acceptq_head;
        struct sonode   **so_acceptq_tail;
        struct sonode   *so_acceptq_next;       /* acceptq list */

        ushort_t so_flag;               /* flags, see below */
        /* XXX remove fsid; it can be filled in by vnode op */
        dev_t   so_fsid;                /* file system identifier */
        int     so_count;               /* count of opened references */

        /* Needed to recreate the same socket for accept */
        short   so_family;
        short   so_type;
        short   so_protocol;
        short   so_version;             /* From so_socket call */

        /* Options */
        short   so_options;             /* From socket call, see socket.h */
        struct linger   so_linger;      /* SO_LINGER value */
#define so_sndbuf       so_proto_settings.so_txhiwat /* SO_SNDBUF value */
#define so_sndlowat     so_proto_settings.so_txlowat /* tx low water mark */
#define so_rcvbuf       so_proto_settings.so_rxhiwat /* SO_RCVBUF value */
#define so_rcvlowat     so_proto_settings.so_rxlowat /* rx low water mark */

#ifdef notyet
        int     so_sndtimeo;            /* Not yet implemented */
        int     so_rcvtimeo;            /* Not yet implemented */
#endif /* notyet */
        ushort_t so_error;              /* error affecting connection */
        /* XXX-anders remove delayed_error and backlog */
        ushort_t so_delayed_error;      /* From T_uderror_ind */
        int     so_backlog;             /* Listen backlog */

        mblk_t  *so_oobmsg;             /* outofline oob data */
        uint_t  so_oobsigcnt;           /* Number of SIGURG generated */
        uint_t  so_oobcnt;              /* Number of T_EXDATA_IND queued */
        size_t  so_oobmark;             /* offset of the oob data */

        pid_t   so_pgrp;                /* pgrp for signals */

        mblk_t          *so_eaddr_mp;   /* for so_delayed_error */

                                        /* put here for delayed processing  */
        void            *so_priv;       /* sonode private data */
        cred_t          *so_peercred;   /* connected socket peer cred */
        pid_t           so_cpid;        /* connected socket peer cached pid */
```

```
        zoneid_t        so_zoneid;      /* opener's zoneid */

        /* Generic socket data */
        struct pollhead so_poll_list;   /* common pollhead */

        /* Receive */
        volatile unsigned int   so_rcv_queued;
        mblk_t          *so_rcv_q_head;
        mblk_t          *so_rcv_q_last_head;
        mblk_t          *so_rcv_q_last_tail;
        mblk_t          *so_rcv_head;           /* 1st mblk in the list */
        mblk_t          *so_rcv_last_head;      /* curr. mblk in b_next chain */
        mblk_t          *so_rcv_last_tail;      /* last mblk in b_cont chain */
        kcondvar_t      so_rcv_cv;
        uint_t          so_rcv_wanted;  /* # of bytes wanted by app */
        uint_t          so_rcv_timer_interval;
        uint32_t        so_rcv_timestamp;
        uint32_t        so_rcv_thresh;

        /* Send */
        boolean_t       so_snd_qfull;   /* Transmit full */
        kcondvar_t      so_snd_cv;

        boolean_t so_rcv_wakeup;
        boolean_t so_snd_wakeup;

        /* Communication channel with protocol */
        sock_lower_handle_t     so_proto_handle;
        sock_downcalls_t        *so_downcalls;

        struct sockparams *so_sockparams;       /* vnode or socket module */
        struct sock_options so_proto_settings;  /* Change the name of struct */
        boolean_t               so_flowctrld;   /* Flow controlled */
        uint_t                  so_copyflag;    /* Copy related flag */
        kcondvar_t              so_copy_cv;     /* Copy cond variable */
        int                     so_copy_error;  /* Copy error, currently zero */

        /* kernel sockets */
        ksocket_callbacks_t     so_ksock_callbacks;
        void                    *so_ksock_cb_arg;       /* callback argument */
};

/* Socket network operations switch */
struct sonodeops {
        int     (*sop_init)(struct sonode *, struct sonode *, cred_t *,
                    int);
        int     (*sop_accept)(struct sonode *, int, cred_t *, struct sonode **);
        int     (*sop_bind)(struct sonode *, struct sockaddr *, socklen_t,
                    int, cred_t *);
        int     (*sop_listen)(struct sonode *, int, cred_t *);
        int     (*sop_connect)(struct sonode *, const struct sockaddr *,
                    socklen_t, int, int, cred_t *);
        int     (*sop_recvmsg)(struct sonode *, struct msghdr *,
                    struct uio *, cred_t *);
        int     (*sop_sendmsg)(struct sonode *, struct msghdr *,
                    struct uio *, cred_t *);
        int     (*sop_sendmblk)(struct sonode *, struct msghdr *, mblk_t *,
                    int, cred_t *);
        int     (*sop_getpeername)(struct sonode *, struct sockaddr *,
                    socklen_t *, cred_t *);
        int     (*sop_getsockname)(struct sonode *, struct sockaddr *,
                    socklen_t *, cred_t *);
        int     (*sop_shutdown)(struct sonode *, int, cred_t *);
        int     (*sop_getsockopt)(struct sonode *, int, int, void *,
                    socklen_t *, int, cred_t *);
        int     (*sop_setsockopt)(struct sonode *, int, int, const void *,
                    socklen_t, cred_t *);
        int     (*sop_ioctl)(struct sonode *, int, intptr_t, int,
                    cred_t *, int32_t *);
        int     (*sop_poll)(struct sonode *, short, int, short *,
                    struct pollhead **);
        int     (*sop_disconnect)(struct sonode *, int, cred_t *);
        void    (*sop_fini)(struct sonode *, cred_t *);
```

```
};
```