

Kernel Sockets

Anders Persson
anders.persson@sun.com

April 23, 2007

1 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.

The rest of this document describes a lightweight interface for creating socket within the kernel.

2 Basic Interface

A kernel socket is represented by an opaque data type (`ksocket_t`), which should only be manipulated by the kernel socket functions (Table 1.) Most of the userland socket functions have a kernel socket equivalent¹. For example, creating a new socket is done using `ksock_socket()`, and a typical usage scenario would be:

```
    :  
if ((error = ksock_socket(&ks, AF_INET, SOCK_STREAM)) != 0)  
    goto fail;  
    :  
    :
```

¹The usage of `AF_UNIX` sockets have not yet been evaluated, so a `socketpair(3SOCKET)` equivalent does not yet exist

```

int ksock_socket(ksocket_t *s, int domain, int type, int protocol)
int ksock_bind(ksocket_t s, struct sockaddr *name, int namelen)
int ksock_listen(ksocket_t s, int backlog)
int ksock_accept(ksocket_t s, ksocket_t *ns, struct sockaddr *name,
                 socklen_t *namelen, ksock_callback_t *cb, void *cbarg)
int ksock_connect(ksocket_t s, const struct sockaddr *name,
                 int namelen)
int ksock_send(ksocket_t s, void *msg, size_t msgsize, int flags,
              size_t *sent)
int ksock_sendto(ksocket_t s, void *msg, size_t msgsize, int flags,
                const struct sockaddr *to, int tolen, size_t *sent)
int ksock_sendmsg(ksocket_t s, const struct msghdr *msg, size_t *sent)
int ksock_recv(ksocket_t s, void *msg, size_t msgsize, int flags,
              size_t *recvd)
int ksock_recvfrom(ksocket_t s, void *msg, size_t msgsize, int flags,
                  struct sockaddr *from, int *fromlen, size_t *recvd)
int ksock_recvmsg(ksocket_t s, struct msghdr *msg, int flags,
                 size_t *recvd)
int ksock_shutdown(ksocket_t s, int how)
int ksock_close(ksocket_t s)
int ksock_setsockopt(ksocket_t s, int level, int optname,
                    const void *optval, int optlen)
int ksock_getsockopt(ksocket_t s, int level, int optname,
                    void *optval, int *optlen)
int ksock_getpeername(ksocket_t s, struct sockaddr *name,
                     socklen_t *namelen)
int ksock_getsockname(ksocket_t s, struct sockaddr *name,
                     socklen_t *namelen)
int ksock_setnonblocking(ksocket_t s, boolean_t nonblocking)

```

Table 1: Kernel Socket Interface

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

Another difference lies in how blocking behavior is controlled and out-of-band mark information is provided. Instead of using an `ioctl(2)/fcntl(2)` like interface, there are explicit functions to handle each operation. For example, to control the blocking behavior of the socket `ksock_setnonblocking()` would be used instead of setting `FNDELAY` with `fcntl(2)`.

<i>Event</i>	<i>Occurs when...</i>
Connected	The kernel socket is connected
Disconnected	The kernel socket is disconnected
New Connection	A new connection is placed on the accept queue
Have Data	Data is received
Can Send	A non-blocking send should succeed
Exception	There is an exception (e.g., TCP OOB data)

Table 2: Kernel Socket Events

3 Event Notification

Although the basic kernel socket interface is reminiscent of the userland API, event notification is handled quite differently. For userland 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.

Six different events have been defined (Table 2), and for each event a callback function can be registered. The defined callback type (`ksock_callback_t`, Table 3) have a function pointer for each event. To be notified about a particular event the user would assign a user specific event handler, and similarly, if a certain event should be ignored, then its function pointer should be set to `NULL`. Enabling callbacks for a particular kernel socket is done through `ksock_callback()` (Table 3), which takes the callbacks and a user value as arguments. The user supplied value is passed in as one of the arguments to the callback functions.

Most events only take the kernel socket and user value as arguments, however, both the *HaveData*, and *Exception* events provide an additional argument. In the case of *HaveData*, the callback function provides the amount of data received. The additional argument for the *Exception* event is used to distinguish between different exception cases.

When a new connection is accepted, it does not inherit the callback functions or user argument from the listener. However, the user can pass in a callback and an object when calling `ksock_accept()`, and they will be registered with the new connection. If a callback is supplied by the user, then it is possible that an event notification occurs *before* `ksock_accept()` returns. A `NULL` callback argument indicates that the user does not want to install a callback.

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:

```

typedef struct ksock_callback_s {
    void (*ksock_cb_connected)(ksocket_t, void *),
    void (*ksock_cb_disconnected)(ksocket_t, void *),
    void (*ksock_cb_newconn)(ksocket_t, void *),
    void (*ksock_cb_havedata)(ksocket_t, void *, ssize_t),
    void (*ksock_cb_cansend)(ksocket_t, void *),
    void (*ksock_cb_exception)(ksocket_t, void *, uint32_t)
} ksock_callback_t

int ksock_callback(ksocket_t s, ksock_callback_t *cb, void *arg)

```

Table 3: Callback Interface

- Perform any blocking operations
- Use any kernel socket functions

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.

4 Limitations

The interface provides the required functionality to create `AF_INET` and `AF_INET6` sockets. Support for `AF_UNIX` sockets is still being evaluated. In addition, there is no services for resolving host names (i.e., `getaddrinfo(3SOCKET)`, `gethostbyname(3NSL)`).