```c
#include <sys/types.h>
#include <sys/un.h>
#include <inttypes.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <string.h>
#include <errno.h>
#include <limits.h>
#include "cnxman-socket.h"
#include "libcman.h"

/* List of saved messages */
struct saved_message
{
        struct sock_header *msg;
        struct saved_message *next;
};

struct cman_handle
{
        int magic;
        int fd;
        int zero_fd;
        void *privdata;
        int want_reply;
        cman_callback_t event_callback;
        cman_datacallback_t data_callback;
        cman_confchgcallback_t confchg_callback;

        void *reply_buffer;
        int reply_buflen;
        int reply_status;

        struct saved_message *saved_data_msg;
        struct saved_message *saved_event_msg;
        struct saved_message *saved_reply_msg;
};

#define VALIDATE_HANDLE(h) do {if (!(h) || (h)->magic != CMAN_MAGIC) {errno = EINVAL; return -1;}} while (0)

/*
 * Wait for an command/request reply.
 * Data/event messages will be queued.
 *
 */
static int wait_for_reply(struct cman_handle *h, void *msg, int max_len)
{
        int ret;

        h->want_reply = 1;
        h->reply_buffer = msg;
        h->reply_buflen = max_len;

        do
        {
                ret = cman_dispatch(h, CMAN_DISPATCH_BLOCKING | CMAN_DISPATCH_IGNORE_EVENT | CMAN_DISPATCH_IGNORE_DATA);

        } while (h->want_reply == 1 && ret >= 0);

        h->reply_buffer = NULL;
        h->reply_buflen = 0;

        /* Error in local comms */
        if (ret < 0) {
                return -1;
        }
        /* cnxman daemon returns -ve errno values on error */
        if (h->reply_status < 0) {
                errno = -h->reply_status;
                return -1;
        }
        else {
                return h->reply_status;
        }
}


static void copy_node(cman_node_t *unode, struct cl_cluster_node *knode)
{
        unode->cn_nodeid = knode->node_id;
        unode->cn_member = knode->state == NODESTATE_MEMBER?1:0;
        strncpy(unode->cn_name, knode->name, sizeof(unode->cn_name) - 1);
        unode->cn_incarnation = knode->incarnation;
        unode->cn_jointime = knode->jointime;

        memset(&unode->cn_address, 0, sizeof(unode->cn_address));
        memcpy(&unode->cn_address.cna_address, knode->addr, knode->addrlen);
        unode->cn_address.cna_addrlen = knode->addrlen;
}

/* Add to a list. saved_message *m is the head of the list in the cman_handle */
static void add_to_waitlist(struct saved_message **m, struct sock_header *msg)
{
        struct saved_message *next = *m;
        struct saved_message *last = *m;
        struct saved_message *this;

        this = malloc(sizeof(struct saved_message));
        if (!this)
                return;

        this->msg = malloc(msg->length);
        if (!this->msg)
        {
                free(this);
                return;
        }

        memcpy(this->msg, msg, msg->length);
        this->next = NULL;

        if (!next)
        {
                *m = this;
                return;
        }

        for (; next; next = next->next)
        {
                last = next;
        }
        last->next = this;
}

static int process_cman_message(struct cman_handle *h, int flags, struct sock_header *msg)
{
        /* Data for us */
        if ((msg->command & CMAN_CMDMASK_CMD) == CMAN_CMD_DATA)
        {
                struct sock_data_header *dmsg = (struct sock_data_header *)msg;
                char *buf = (char *)msg;

                if (flags & CMAN_DISPATCH_IGNORE_DATA)
                {
                        add_to_waitlist(&h->saved_data_msg, msg);
                }
                else
                {
                        if (h->data_callback)
                                h->data_callback(h, h->privdata,
                                                 buf+sizeof(*dmsg), msg->length-sizeof(*dmsg),
                                                 dmsg->port, dmsg->nodeid);
                }
                return 0;
        }

        /* Got a reply to a previous information request */
        if ((msg->command & CMAN_CMDFLAG_REPLY) && h->want_reply)
        {
                char *replybuf = (char *)msg;
                int replylen = msg->length - sizeof(struct sock_reply_header);
                struct sock_reply_header *reply = (struct sock_reply_header *)msg;

                if (flags & CMAN_DISPATCH_IGNORE_REPLY)
                {
                        add_to_waitlist(&h->saved_reply_msg, msg);
                        return 0;
                }

                replybuf += sizeof(struct sock_reply_header);
                if (replylen <= h->reply_buflen)
                {
                        memcpy(h->reply_buffer, replybuf, replylen);
                }
                h->want_reply = 0;
                h->reply_status = reply->status;

                return 1;
        }

        /* OOB event */
        if (msg->command == CMAN_CMD_EVENT || msg->command == CMAN_CMD_CONFCHG)
        {
                if (flags & CMAN_DISPATCH_IGNORE_EVENT)
                {
                        add_to_waitlist(&h->saved_event_msg, msg);
                }
                else
                {
                        if (msg->command == CMAN_CMD_EVENT && h->event_callback) {
                                struct sock_event_message *emsg = (struct sock_event_message *)msg;
                                h->event_callback(h, h->privdata, emsg->reason, emsg->arg);
                        }

                        if (msg->command == CMAN_CMD_CONFCHG && h->confchg_callback)
                        {
                                struct sock_confchg_message *cmsg = (struct sock_confchg_message *)msg;

                                h->confchg_callback(h, h->privdata,
                                                    cmsg->entries,cmsg->member_entries,
                                                    &cmsg->entries[cmsg->member_entries], cmsg->left_entries,
                                                    &cmsg->entries[cmsg->member_entries+cmsg->left_entries], cmsg->joined_
```

```
				}
			}
		}

		return 0;
	}

	static int loopy_writev(int fd, struct iovec *iovptr, size_t iovlen)
	{
		size_t byte_cnt=0;
		int len;

		while (iovlen > 0)
		{
			len = writev(fd, iovptr, iovlen);
			if (len <= 0)
				return len;

			byte_cnt += len;
			while (len >= iovptr->iov_len)
			{
				len -= iovptr->iov_len;
				iovptr++;
				iovlen--;
			}

			if ((ssize_t)iovlen <=0 )
				break;

			iovptr->iov_base = (char *)iovptr->iov_base + len;
			iovptr->iov_len -= len;
		}
		return byte_cnt;
	}


	static int send_message(struct cman_handle *h, int msgtype, const void *inbuf, int inlen)
	{
		struct sock_header header;
		int len;
		struct iovec iov[2];
		size_t iovlen = 1;

		header.magic = CMAN_MAGIC;
		header.version = CMAN_VERSION;
		header.command = msgtype;
		header.flags = 0;
		header.length = sizeof(header) + inlen;

		iov[0].iov_len = sizeof(header);
		iov[0].iov_base = &header;
		if (inbuf)
		{
			iov[1].iov_len = inlen;
			iov[1].iov_base = (void *) inbuf;
			iovlen++;
		}

		len = loopy_writev(h->fd, iov, iovlen);
		if (len < 0)
			return len;
		return 0;
	}

	/* Does something similar to the ioctl calls */
	static int info_call(struct cman_handle *h, int msgtype, const void *inbuf, int inlen, void *outbuf, int outlen)
	{
		if (send_message(h, msgtype, inbuf, inlen))
			return -1;

		return wait_for_reply(h, outbuf, outlen);
	}

	static cman_handle_t open_socket(const char *name, int namelen, void *privdata)
	{
		struct cman_handle *h;
		struct sockaddr_un sockaddr;

		h = malloc(sizeof(struct cman_handle));
		if (!h)
			return NULL;

		h->magic = CMAN_MAGIC;
		h->privdata = privdata;
		h->event_callback = NULL;
		h->data_callback = NULL;
		h->confchg_callback = NULL;
		h->want_reply = 0;
		h->saved_data_msg = NULL;
		h->saved_event_msg = NULL;
		h->saved_reply_msg = NULL;

		h->fd = socket(PF_UNIX, SOCK_STREAM, 0);
		if (h->fd == -1)
		{
			int saved_errno = errno;
			free(h);
			errno = saved_errno;
```

```
                return NULL;
        }

        fcntl(h->fd, F_SETFD, 1); /* Set close-on-exec */
        memset(&sockaddr, 0, sizeof(sockaddr));
        memcpy(sockaddr.sun_path, name, namelen);
        sockaddr.sun_family = AF_UNIX;

        if (connect(h->fd, (struct sockaddr *) &sockaddr, sizeof(sockaddr)) < 0)
        {
                int saved_errno = errno;
                close(h->fd);
                free(h);
                errno = saved_errno;
                return NULL;
        }

        /* Get a handle on /dev/zero too. This is always active so we
           can return it from cman_get_fd() if we have cached messages */
        h->zero_fd = open("/dev/zero", O_RDONLY);
        if (h->zero_fd < 0)
        {
                int saved_errno = errno;
                close(h->fd);
                free(h);
                h = NULL;
                errno = saved_errno;
        } else
                fcntl(h->zero_fd, F_SETFD, 1); /* Set close-on-exec */

        return (cman_handle_t)h;
}

cman_handle_t cman_admin_init(void *privdata)
{
        return open_socket(ADMIN_SOCKNAME, sizeof(ADMIN_SOCKNAME), privdata);
}

cman_handle_t cman_init(void *privdata)
{
        return open_socket(CLIENT_SOCKNAME, sizeof(CLIENT_SOCKNAME), privdata);
}

int cman_finish(cman_handle_t handle)
{
        struct cman_handle *h = (struct cman_handle *)handle;
        VALIDATE_HANDLE(h);

        h->magic = 0;
        close(h->fd);
        close(h->zero_fd);
        free(h);

        return 0;
}

int cman_setprivdata(cman_handle_t handle, void *privdata)
{
        struct cman_handle *h = (struct cman_handle *)handle;
        VALIDATE_HANDLE(h);

        h->privdata = privdata;
        return 0;
}

int cman_getprivdata(cman_handle_t handle, void **privdata)
{
        struct cman_handle *h = (struct cman_handle *)handle;
        VALIDATE_HANDLE(h);

        *privdata = h->privdata;

        return 0;
}


int cman_start_notification(cman_handle_t handle, cman_callback_t callback)
{
        struct cman_handle *h = (struct cman_handle *)handle;
        VALIDATE_HANDLE(h);

        if (!callback)
        {
                errno = EINVAL;
                return -1;
        }
        if (info_call(h, CMAN_CMD_NOTIFY, NULL, 0, NULL, 0))
                return -1;
        h->event_callback = callback;

        return 0;
}

int cman_stop_notification(cman_handle_t handle)
{
        struct cman_handle *h = (struct cman_handle *)handle;
        VALIDATE_HANDLE(h);

```

```
		if (info_call(h, CMAN_CMD_REMOVENOTIFY, NULL, 0, NULL, 0))
			return -1;
		h->event_callback = NULL;

		return 0;
	}

	int cman_start_confchg(cman_handle_t handle, cman_confchgcallback_t callback)
	{
		struct cman_handle *h = (struct cman_handle *)handle;
		VALIDATE_HANDLE(h);

		if (!callback)
		{
			errno = EINVAL;
			return -1;
		}
		if (info_call(h, CMAN_CMD_START_CONFCHG, NULL, 0, NULL, 0))
			return -1;
		h->confchg_callback = callback;

		return 0;
	}

	int cman_stop_confchg(cman_handle_t handle)
	{
		struct cman_handle *h = (struct cman_handle *)handle;
		VALIDATE_HANDLE(h);

		if (info_call(h, CMAN_CMD_STOP_CONFCHG, NULL, 0, NULL, 0))
			return -1;
		h->confchg_callback = NULL;

		return 0;
	}


	int cman_get_fd(cman_handle_t handle)
	{
		struct cman_handle *h = (struct cman_handle *)handle;
		VALIDATE_HANDLE(h);

		/* If we have saved messages then return an FD to /dev/zero which
		   will always be readable */
		if (h->saved_data_msg || h->saved_event_msg || h->saved_reply_msg)
			return h->zero_fd;
		else
			return h->fd;
	}

	int cman_dispatch(cman_handle_t handle, int flags)
	{
		struct cman_handle *h = (struct cman_handle *)handle;
		int len;
		int offset;
		int recv_flags = 0;
		char buf[PIPE_BUF];
		VALIDATE_HANDLE(h);

		if (!(flags & CMAN_DISPATCH_BLOCKING))
			recv_flags |= MSG_DONTWAIT;

		do
		{
			int res;
			char *bufptr = buf;
			struct sock_header *header = (struct sock_header *)buf;

			/* First, drain any waiting queues */
			if (h->saved_reply_msg && !(flags & CMAN_DISPATCH_IGNORE_REPLY))
			{
				struct saved_message *smsg = h->saved_reply_msg;

				res = process_cman_message(h, flags, smsg->msg);
				h->saved_reply_msg = smsg->next;
				len = smsg->msg->length;
				free(smsg->msg);
				free(smsg);
				if (res || (flags & CMAN_DISPATCH_TYPE_MASK) == CMAN_DISPATCH_ONE)
					break;
				else
					continue;
			}
			if (h->saved_data_msg && !(flags & CMAN_DISPATCH_IGNORE_DATA))
			{
				struct saved_message *smsg = h->saved_data_msg;

				res = process_cman_message(h, flags, smsg->msg);
				h->saved_data_msg = smsg->next;
				len = smsg->msg->length;
				free(smsg->msg);
				free(smsg);
				if (res || (flags & CMAN_DISPATCH_TYPE_MASK) == CMAN_DISPATCH_ONE)
					break;
				else
					continue;
			}
			if (h->saved_event_msg && !(flags & CMAN_DISPATCH_IGNORE_EVENT))
```

```
                {
                        struct saved_message *smsg = h->saved_event_msg;

                        res = process_cman_message(h, flags, smsg->msg);
                        h->saved_event_msg = smsg->next;
                        len = smsg->msg->length;
                        free(smsg->msg);
                        free(smsg);
                        if (res || (flags & CMAN_DISPATCH_TYPE_MASK) == CMAN_DISPATCH_ONE)
                                break;
                        else
                                continue;
                }

                /* Now look for new messages */
```

**Event tainted_data_argument**: Calling function "recv" taints argument "buf".

Also see events: [tainted_data_transitive][tainted_data_transitive][var_assign_var][tainted_data]

```
                len = recv(h->fd, buf, sizeof(struct sock_header), recv_flags);

```

At conditional (1): "len == 0": Taking false branch.

```
                if (len == 0) {
                        errno = EHOSTDOWN;
                        return -1;
                }

```

At conditional (2): "len < 0": Taking false branch.

```
                if (len < 0 &&
                    (errno == EINTR || errno == EAGAIN))
                        return 0;

```

At conditional (3): "len < 0": Taking false branch.

```
                if (len < 0)
                        return -1;

                offset = len;

                /* It's too big! */
```

At conditional (4): "header->length > sizeof (buf) /*4096*/": Taking true branch.

```
                if (header->length > sizeof(buf))
                {
                        bufptr = malloc(header->length);
```

At conditional (5): "!bufptr": Taking false branch.

```
                        if (!bufptr)
                                return -1;
```

**Event tainted_data_transitive**: Call to function "memcpy" with tainted argument "buf" transitively taints "bufptr".

**Event tainted_data_transitive**: Call to function "memcpy" with tainted argument "bufptr" returns tainted data.

Also see events: [tainted_data_argument][var_assign_var][tainted_data]

```
                        memcpy(bufptr, buf, sizeof(*header));
```

**Event var_assign_var**: Assigning: "(struct sock_header *)bufptr" = "header". Both are now tainted.

Also see events: [tainted_data_argument][tainted_data_transitive][tainted_data_transitive][tainted_data]

```
                        header = (struct sock_header *)bufptr;
                }

                /* Read the rest */
```

**Event tainted_data**: Using tainted variable "header->length" as a loop boundary.

Also see events: [tainted_data_argument][tainted_data_transitive][tainted_data_transitive][var_assign_var]

```
                while (offset < header->length)
                {
                        len = read(h->fd, bufptr+offset, header->length-offset);
                        if (len == 0) {
                                if (bufptr != buf)
                                        free(bufptr);
                                errno = EHOSTDOWN;
                                return -1;
                        }

                        if (len < 0 &&
                            (errno == EINTR || errno == EAGAIN)) {
                                if (bufptr != buf)
                                        free(bufptr);
                                return 0;
                        }

```

```
				if (len < 0) {
					if (bufptr != buf)
						free(bufptr);
					return -1;
				}
				offset += len;
			}

			res = process_cman_message(h, flags, header);
			if (bufptr != buf)
				free(bufptr);

			if (res)
				break;

		} while ( flags & CMAN_DISPATCH_ALL &&
			  !(len < 0 && errno == EAGAIN) );

		return len;
}

/* GET_ALLMEMBERS returns the number of nodes as status */
int cman_get_node_count(cman_handle_t handle)
{
		struct cman_handle *h = (struct cman_handle *)handle;
		VALIDATE_HANDLE(h);

		return info_call(h, CMAN_CMD_GETALLMEMBERS, NULL, 0, NULL, 0);
}

int cman_get_nodes(cman_handle_t handle, int maxnodes, int *retnodes, cman_node_t *nodes)
{
		struct cman_handle *h = (struct cman_handle *)handle;
		struct cl_cluster_node *cman_nodes;
		int status;
		int buflen;
		int count = 0;
		VALIDATE_HANDLE(h);

		if (!retnodes || !nodes || maxnodes < 1)
		{
			errno = EINVAL;
			return -1;
		}

		buflen = sizeof(struct cl_cluster_node) * maxnodes;
		cman_nodes = malloc(buflen);
		if (!cman_nodes)
			return -1;

		status = info_call(h, CMAN_CMD_GETALLMEMBERS, NULL, 0, cman_nodes, buflen);
		if (status < 0)
		{
			int saved_errno = errno;
			free(cman_nodes);
			errno = saved_errno;
			return -1;
		}

		if (cman_nodes[0].size != sizeof(struct cl_cluster_node))
		{
			free(cman_nodes);
			errno = EINVAL;
			return -1;
		}

		if (status > maxnodes)
			status = maxnodes;

		for (count = 0; count < status; count++)
		{
			copy_node(&nodes[count], &cman_nodes[count]);
		}
		free(cman_nodes);
		*retnodes = status;
		return 0;
}

int cman_get_disallowed_nodes(cman_handle_t handle, int maxnodes, int *retnodes, cman_node_t *nodes)
{
		struct cman_handle *h = (struct cman_handle *)handle;
		struct cl_cluster_node *cman_nodes;
		int status;
		int buflen;
		int count = 0;
		int out_count = 0;
		VALIDATE_HANDLE(h);

		if (!retnodes || !nodes || maxnodes < 1)
		{
			errno = EINVAL;
			return -1;
		}

		buflen = sizeof(struct cl_cluster_node) * maxnodes;
		cman_nodes = malloc(buflen);
		if (!cman_nodes)
			return -1;
```

```

	status = info_call(h, CMAN_CMD_GETALLMEMBERS, NULL, 0, cman_nodes, buflen);
	if (status < 0)
	{
		int saved_errno = errno;
		free(cman_nodes);
		errno = saved_errno;
		return -1;
	}

	if (cman_nodes[0].size != sizeof(struct cl_cluster_node))
	{
		free(cman_nodes);
		errno = EINVAL;
		return -1;
	}

	for (count = 0; count < status; count++)
	{
		if (cman_nodes[count].state == NODESTATE_AISONLY && out_count < maxnodes)
			copy_node(&nodes[out_count++], &cman_nodes[count]);
	}
	free(cman_nodes);
	*retnodes = out_count;
	return 0;
}

int cman_get_node(cman_handle_t handle, int nodeid, cman_node_t *node)
{
	struct cman_handle *h = (struct cman_handle *)handle;
	struct cl_cluster_node cman_node;
	int status;
	VALIDATE_HANDLE(h);

	if (!node || strlen(node->cn_name) >= sizeof(cman_node.name))
	{
		errno = EINVAL;
		return -1;
	}

	cman_node.node_id = nodeid;
	strncpy(cman_node.name, node->cn_name, sizeof(cman_node.name) - 1);
	status = info_call(h, CMAN_CMD_GETNODE, &cman_node, sizeof(struct cl_cluster_node),
			   &cman_node, sizeof(struct cl_cluster_node));
	if (status < 0)
		return -1;

	copy_node(node, &cman_node);

	return 0;
}

int cman_get_node_extra(cman_handle_t handle, int nodeid, cman_node_extra_t *node)
{
	struct cman_handle *h = (struct cman_handle *)handle;
	int status;
	VALIDATE_HANDLE(h);

	status = info_call(h, CMAN_CMD_GETNODE_EXTRA, &nodeid, sizeof(int),
			   node, sizeof(cman_node_extra_t));
	if (status < 0)
		return -1;

	return 0;
}

int cman_get_subsys_count(cman_handle_t handle)
{
	struct cman_handle *h = (struct cman_handle *)handle;
	VALIDATE_HANDLE(h);

	return info_call(h, CMAN_CMD_GET_JOINCOUNT, NULL,0, NULL, 0);
}

int cman_is_active(cman_handle_t handle)
{
	struct cman_handle *h = (struct cman_handle *)handle;
	VALIDATE_HANDLE(h);

	return info_call(h, CMAN_CMD_ISACTIVE, NULL, 0, NULL, 0);
}

int cman_is_listening(cman_handle_t handle, int nodeid, uint8_t port)
{
	struct cman_handle *h = (struct cman_handle *)handle;
	struct cl_listen_request req;
	VALIDATE_HANDLE(h);

	req.port = port;
	req.nodeid = nodeid;
	return info_call(h, CMAN_CMD_ISLISTENING, &req, sizeof(struct cl_listen_request), NULL, 0);
}

int cman_is_quorate(cman_handle_t handle)
{
	struct cman_handle *h = (struct cman_handle *)handle;
	VALIDATE_HANDLE(h);

```

```
        return info_call(h, CMAN_CMD_ISQUORATE, NULL, 0, NULL, 0);
}


int cman_get_version(cman_handle_t handle, cman_version_t *version)
{
        struct cman_handle *h = (struct cman_handle *)handle;
        VALIDATE_HANDLE(h);

        if (!version)
        {
                errno = EINVAL;
                return -1;
        }
        return info_call(h, CMAN_CMD_GET_VERSION, NULL, 0, version, sizeof(cman_version_t));
}

int cman_set_version(cman_handle_t handle, const cman_version_t *version)
{
        struct cman_handle *h = (struct cman_handle *)handle;
        VALIDATE_HANDLE(h);

        if (!version)
        {
                errno = EINVAL;
                return -1;
        }
        return info_call(h, CMAN_CMD_SET_VERSION, version, sizeof(cman_version_t), NULL, 0);
}

int cman_kill_node(cman_handle_t handle, int nodeid)
{
        struct cman_handle *h = (struct cman_handle *)handle;
        VALIDATE_HANDLE(h);

        if (!nodeid)
        {
                errno = EINVAL;
                return -1;
        }
        return info_call(h, CMAN_CMD_KILLNODE, &nodeid, sizeof(nodeid), NULL, 0);
}

int cman_set_votes(cman_handle_t handle, int votes, int nodeid)
{
        struct cman_handle *h = (struct cman_handle *)handle;
        struct cl_set_votes newv;
        VALIDATE_HANDLE(h);

        if (!votes)
        {
                errno = EINVAL;
                return -1;
        }
        newv.nodeid = nodeid;
        newv.newvotes  = votes;
        return info_call(h, CMAN_CMD_SET_VOTES, &newv, sizeof(newv), NULL, 0);
}

int cman_set_expected_votes(cman_handle_t handle, int evotes)
{
        struct cman_handle *h = (struct cman_handle *)handle;
        VALIDATE_HANDLE(h);

        if (!evotes)
        {
                errno = EINVAL;
                return -1;
        }
        return info_call(h, CMAN_CMD_SETEXPECTED_VOTES, &evotes, sizeof(evotes), NULL, 0);
}

int cman_leave_cluster(cman_handle_t handle, int reason)
{
        struct cman_handle *h = (struct cman_handle *)handle;
        VALIDATE_HANDLE(h);

        return info_call(h, CMAN_CMD_LEAVE_CLUSTER, &reason, sizeof(reason), NULL, 0);
}

int cman_get_cluster(cman_handle_t handle, cman_cluster_t *clinfo)
{
        struct cman_handle *h = (struct cman_handle *)handle;
        VALIDATE_HANDLE(h);

        if (!clinfo)
        {
                errno = EINVAL;
                return -1;
        }
        return info_call(h, CMAN_CMD_GETCLUSTER, NULL, 0, clinfo, sizeof(cman_cluster_t));
}

int cman_get_extra_info(cman_handle_t handle, cman_extra_info_t *info, int maxlen)
{
        struct cman_handle *h = (struct cman_handle *)handle;
        VALIDATE_HANDLE(h);

        if (!info || maxlen < sizeof(cman_extra_info_t))
        {
                errno = EINVAL;
                return -1;
        }
        return info_call(h, CMAN_CMD_GETEXTRAINFO, NULL, 0, info, maxlen);
}

int cman_send_data(cman_handle_t handle, const void *buf, int len, int flags, uint8_t port, int nodeid)
{
        struct cman_handle *h = (struct cman_handle *)handle;
        struct iovec iov[2];
        struct sock_data_header header;
        VALIDATE_HANDLE(h);

        header.header.magic = CMAN_MAGIC;
        header.header.version = CMAN_VERSION;
        header.header.command = CMAN_CMD_DATA;
        header.header.flags = flags;
        header.header.length = len + sizeof(header);
        header.nodeid = nodeid;
        header.port = port;

        iov[0].iov_len = sizeof(header);
        iov[0].iov_base = &header;
        iov[1].iov_len = len;
        iov[1].iov_base = (void *) buf;

        return loopy_writev(h->fd, iov, 2);
}


int cman_start_recv_data(cman_handle_t handle, cman_datacallback_t callback, uint8_t port)
{
        struct cman_handle *h = (struct cman_handle *)handle;
        int portparam;
        int status;
        VALIDATE_HANDLE(h);

/* Do a "bind" */
        portparam = port;
        status = info_call(h, CMAN_CMD_BIND, &portparam, sizeof(portparam), NULL, 0);

        if (status == 0)
                h->data_callback = callback;

        return status;
}

int cman_end_recv_data(cman_handle_t handle)
{
        struct cman_handle *h = (struct cman_handle *)handle;
        VALIDATE_HANDLE(h);

        h->data_callback = NULL;
        return 0;
}


int cman_barrier_register(cman_handle_t handle, const char *name, int flags, int nodes)
{
        struct cman_handle *h = (struct cman_handle *)handle;
        struct cl_barrier_info binfo;
        VALIDATE_HANDLE(h);

        if (strlen(name) >= MAX_BARRIER_NAME_LEN)
        {
                errno = EINVAL;
                return -1;
        }

        binfo.cmd = BARRIER_CMD_REGISTER;
        strncpy(binfo.name, name, sizeof(binfo.name) - 1);
        binfo.arg = nodes;
        binfo.flags = flags;

        return info_call(h, CMAN_CMD_BARRIER, &binfo, sizeof(binfo), NULL, 0);
}


int cman_barrier_change(cman_handle_t handle, const char *name, int flags, int arg)
{
        struct cman_handle *h = (struct cman_handle *)handle;
        struct cl_barrier_info binfo;
        VALIDATE_HANDLE(h);

        if (strlen(name) >= MAX_BARRIER_NAME_LEN)
        {
                errno = EINVAL;
                return -1;
        }

        binfo.cmd = BARRIER_CMD_CHANGE;
        strncpy(binfo.name, name, sizeof(binfo.name) - 1);
        binfo.arg = arg;
        binfo.flags = flags;

        return info_call(h, CMAN_CMD_BARRIER, &binfo, sizeof(binfo), NULL, 0);
```

```

}

int cman_barrier_wait(cman_handle_t handle, const char *name)
{
	struct cman_handle *h = (struct cman_handle *)handle;
	struct cl_barrier_info binfo;
	VALIDATE_HANDLE(h);

	if (strlen(name) >= MAX_BARRIER_NAME_LEN)
	{
		errno = EINVAL;
		return -1;
	}

	binfo.cmd = BARRIER_CMD_WAIT;
	strncpy(binfo.name, name, sizeof(binfo.name) - 1);

	return info_call(h, CMAN_CMD_BARRIER, &binfo, sizeof(binfo), NULL, 0);
}

int cman_barrier_delete(cman_handle_t handle, const char *name)
{
	struct cman_handle *h = (struct cman_handle *)handle;
	struct cl_barrier_info binfo;
	VALIDATE_HANDLE(h);

	if (strlen(name) >= MAX_BARRIER_NAME_LEN)
	{
		errno = EINVAL;
		return -1;
	}

	binfo.cmd = BARRIER_CMD_DELETE;
	strncpy(binfo.name, name, sizeof(binfo.name) - 1);

	return info_call(h, CMAN_CMD_BARRIER, &binfo, sizeof(binfo), NULL, 0);
}

int cman_shutdown(cman_handle_t handle, int flags)
{
	struct cman_handle *h = (struct cman_handle *)handle;
	VALIDATE_HANDLE(h);

	return info_call(h, CMAN_CMD_TRY_SHUTDOWN, &flags, sizeof(int), NULL, 0);
}

int cman_set_dirty(cman_handle_t handle)
{
	struct cman_handle *h = (struct cman_handle *)handle;
	VALIDATE_HANDLE(h);

	return info_call(h, CMAN_CMD_SET_DIRTY, NULL, 0, NULL, 0);
}

int cman_set_debuglog(cman_handle_t handle, int subsystems)
{
	struct cman_handle *h = (struct cman_handle *)handle;
	VALIDATE_HANDLE(h);

	return info_call(h, CMAN_CMD_SET_DEBUGLOG, &subsystems, sizeof(int), NULL, 0);
}

int cman_replyto_shutdown(cman_handle_t handle, int yesno)
{
	struct cman_handle *h = (struct cman_handle *)handle;
	VALIDATE_HANDLE(h);

	send_message(h, CMAN_CMD_SHUTDOWN_REPLY, &yesno, sizeof(int));
	return 0;
}

static int cman_set_quorum_device(cman_handle_t handle,
				    int ops,
				    char *name, int votes)
{
	struct cman_handle *h = (struct cman_handle *)handle;
	char buf[strlen(name)+1 + sizeof(int)];
	VALIDATE_HANDLE(h);

	memcpy(buf, &votes, sizeof(int));
	strncpy(buf+sizeof(int), name, strlen(name)+1 + sizeof(int) - 1);
	return info_call(h, ops, buf, strlen(name)+1+sizeof(int), NULL, 0);
}

int cman_register_quorum_device(cman_handle_t handle, char *name, int votes)
{
	if ((!name) || (strlen(name) > MAX_CLUSTER_MEMBER_NAME_LEN) || (votes < 0))
	{
		errno = EINVAL;
		return -1;
	}
	return cman_set_quorum_device(handle, CMAN_CMD_REG_QUORUMDEV, name, votes);
}

int cman_unregister_quorum_device(cman_handle_t handle)
{
	struct cman_handle *h = (struct cman_handle *)handle;
```

```
        VALIDATE_HANDLE(h);

        return info_call(h, CMAN_CMD_UNREG_QUORUMDEV, NULL, 0, NULL, 0);
}

int cman_poll_quorum_device(cman_handle_t handle, int isavailable)
{
        struct cman_handle *h = (struct cman_handle *)handle;
        VALIDATE_HANDLE(h);

        return info_call(h, CMAN_CMD_POLL_QUORUMDEV, &isavailable, sizeof(int), NULL, 0);
}

int cman_get_quorum_device(cman_handle_t handle, struct cman_qdev_info *info)
{
        struct cman_handle *h = (struct cman_handle *)handle;
        int ret;
        struct cl_cluster_node cman_node;
        VALIDATE_HANDLE(h);

        cman_node.node_id = CLUSTER_GETNODE_QUORUMDEV;
        ret = info_call(h, CMAN_CMD_GETNODE, &cman_node, sizeof(cman_node), &cman_node, sizeof(cman_node));
        if (!ret) {
                strncpy(info->qi_name, cman_node.name, sizeof(info->qi_name) - 1);
                info->qi_state = cman_node.state;
                info->qi_votes = cman_node.votes;
        }
        return ret;
}

int cman_update_quorum_device(cman_handle_t handle, char *name, int votes)
{
        if ((!name) || (strlen(name) > MAX_CLUSTER_MEMBER_NAME_LEN) || (votes < 0))
        {
                errno = EINVAL;
                return -1;
        }
        return cman_set_quorum_device(handle, CMAN_CMD_UPDATE_QUORUMDEV, name, votes);
}

int cman_get_fenceinfo(cman_handle_t handle, int nodeid, uint64_t *time, int *fenced, char *agent)
{
        struct cman_handle *h = (struct cman_handle *)handle;
        int ret;
        struct cl_fence_info f;
        VALIDATE_HANDLE(h);

        ret = info_call(h, CMAN_CMD_GET_FENCE_INFO, &nodeid, sizeof(int), &f, sizeof(f));
        if (!ret) {
                *time = f.fence_time;
                if (agent)
                        strncpy(agent, f.fence_agent, sizeof(f.fence_agent) - 1);
                *fenced = ((f.flags & FENCE_FLAGS_FENCED) != 0);
        }
        return ret;
}

int cman_get_node_addrs(cman_handle_t handle, int nodeid, int max_addrs, int *num_addrs, struct cman_node_address *addrs)
{
        struct cman_handle *h = (struct cman_handle *)handle;
        int ret;
        char buf[sizeof(struct cl_get_node_addrs) + sizeof(struct cl_node_addrs)*max_addrs];
        struct cl_get_node_addrs *outbuf = (struct cl_get_node_addrs *)buf;
        VALIDATE_HANDLE(h);

        ret = info_call(h, CMAN_CMD_GET_NODEADDRS, &nodeid, sizeof(int), buf, sizeof(buf));
        if (!ret) {
                int i;

                *num_addrs = outbuf->numaddrs;

                if (outbuf->numaddrs > max_addrs)
                        outbuf->numaddrs = max_addrs;

                for (i=0; i < outbuf->numaddrs; i++) {
                        memcpy(&addrs[i].cna_address, &outbuf->addrs[i].addr, outbuf->addrs[i].addrlen);
                        addrs[i].cna_addrlen = outbuf->addrs[i].addrlen;
                }
        }
        return ret;
}

int cman_node_fenced(cman_handle_t handle, int nodeid, uint64_t time, char *agent)
{
        struct cman_handle *h = (struct cman_handle *)handle;
        struct cl_fence_info f;
        VALIDATE_HANDLE(h);

        if (strlen(agent) >= MAX_FENCE_AGENT_NAME_LEN) {
                errno = EINVAL;
                return -1;
        }

        f.nodeid = nodeid;
        f.fence_time = time;
        strncpy(f.fence_agent, agent, sizeof(f.fence_agent) - 1);
        return info_call(h, CMAN_CMD_UPDATE_FENCE_INFO, &f, sizeof(f), NULL, 0);
}
```