

```
1  #include <sys/types.h>
2  #include <sys/un.h>
3  #include <inttypes.h>
4  #include <unistd.h>
5  #include <stdlib.h>
6  #include <stdio.h>
7  #include <fcntl.h>
8  #include <netinet/in.h>
9  #include <string.h>
10 #include <errno.h>
11 #include <limits.h>
12 #include "cnxman-socket.h"
13 #include "libcman.h"
14
15 /* List of saved messages */
16 struct saved_message
17 {
18     struct sock_header *msg;
19     struct saved_message *next;
20 };
21
22 struct cman_handle
23 {
24     int magic;
25     int fd;
26     int zero_fd;
27     void *privdata;
28     int want_reply;
29     cman_callback_t event_callback;
30     cman_datacallback_t data_callback;
31     cman_confchgcallback_t confchg_callback;
32
33     void *reply_buffer;
34     int reply_buflen;
35     int reply_status;
36
37     struct saved_message *saved_data_msg;
38     struct saved_message *saved_event_msg;
39     struct saved_message *saved_reply_msg;
40 };
41
42 #define VALIDATE_HANDLE(h) do {if (!(h) || (h)->magic != CMAN_MAGIC) {errno = EINVAL; return -1;}} while (0)
43
44 /*
45  * Wait for an command/request reply.
46  * Data/event messages will be queued.
47  *
48  */
49 static int wait_for_reply(struct cman_handle *h, void *msg, int max_len)
50 {
51     int ret;
52
53     h->want_reply = 1;
54     h->reply_buffer = msg;
55     h->reply_buflen = max_len;
56
57     do
58     {
59         ret = cman_dispatch(h, CMAN_DISPATCH_BLOCKING | CMAN_DISPATCH_IGNORE_EVENT | CMAN_DISPATCH_IGNORE_DATA);
60     } while (h->want_reply == 1 && ret >= 0);
61
62     h->reply_buffer = NULL;
63     h->reply_buflen = 0;
64
65     /* Error in local comms */
66     if (ret < 0) {
67         return -1;
68     }
69     /* cnxman daemon returns -ve errno values on error */
70     if (h->reply_status < 0) {
71         errno = -h->reply_status;
72         return -1;
73     }
74     else {
75         return h->reply_status;
76     }
77 }
78
79
80
81 static void copy_node(cman_node_t *unode, struct cl_cluster_node *knode)
82 {
83     unode->cn_nodeid = knode->node_id;
84     unode->cn_member = knode->state == NODESTATE_MEMBER?1:0;
85     strncpy(unode->cn_name, knode->name, sizeof(unode->cn_name) - 1);
86     unode->cn_incarnation = knode->incarnation;
87     unode->cn_jointime = knode->jointime;
88
89     memset(&unode->cn_address, 0, sizeof(unode->cn_address));
90     memcpy(&unode->cn_address.cna_address, knode->addr, knode->addrlen);
91     unode->cn_address.cna_addrlen = knode->addrlen;
92 }
93
94 /* Add to a list. saved_message *m is the head of the list in the cman_handle */
95 static void add_to_waitlist(struct saved_message **m, struct sock_header *msg)
96 {
97     struct saved_message *next = *m;
```

```
98     struct saved_message *last = *m;
99     struct saved_message *this;
100
101     this = malloc(sizeof(struct saved_message));
102     if (!this)
103         return;
104
105     this->msg = malloc(msg->length);
106     if (!this->msg)
107     {
108         free(this);
109         return;
110     }
111
112     memcpy(this->msg, msg, msg->length);
113     this->next = NULL;
114
115     if (!next)
116     {
117         *m = this;
118         return;
119     }
120
121     for (; next; next = next->next)
122     {
123         last = next;
124     }
125     last->next = this;
126 }
127
128 static int process_cman_message(struct cman_handle *h, int flags, struct sock_header *msg)
129 {
130     /* Data for us */
131     if ((msg->command & CMAN_CMDMASK_CMD) == CMAN_CMD_DATA)
132     {
133         struct sock_data_header *dmsg = (struct sock_data_header *)msg;
134         char *buf = (char *)msg;
135
136         if (flags & CMAN_DISPATCH_IGNORE_DATA)
137         {
138             add_to_waitlist(&h->saved_data_msg, msg);
139         }
140         else
141         {
142             if (h->data_callback)
143                 h->data_callback(h, h->privdata,
144                                 buf+sizeof(*dmsg), msg->length-sizeof(*dmsg),
145                                 dmsg->port, dmsg->nodeid);
146         }
147         return 0;
148     }
149
150     /* Got a reply to a previous information request */
151     if ((msg->command & CMAN_CMDFLAG_REPLY) && h->want_reply)
152     {
153         char *replybuf = (char *)msg;
154         int replylen = msg->length - sizeof(struct sock_reply_header);
155         struct sock_reply_header *reply = (struct sock_reply_header *)msg;
156
157         if (flags & CMAN_DISPATCH_IGNORE_REPLY)
158         {
159             add_to_waitlist(&h->saved_reply_msg, msg);
160             return 0;
161         }
162
163         replybuf += sizeof(struct sock_reply_header);
164         if (replylen <= h->reply_bufflen)
165         {
166             memcpy(h->reply_buffer, replybuf, replylen);
167         }
168         h->want_reply = 0;
169         h->reply_status = reply->status;
170
171         return 1;
172     }
173
174     /* OOB event */
175     if (msg->command == CMAN_CMD_EVENT || msg->command == CMAN_CMD_CONFCHG)
176     {
177         if (flags & CMAN_DISPATCH_IGNORE_EVENT)
178         {
179             add_to_waitlist(&h->saved_event_msg, msg);
180         }
181         else
182         {
183             if (msg->command == CMAN_CMD_EVENT && h->event_callback) {
184                 struct sock_event_message *emsg = (struct sock_event_message *)msg;
185                 h->event_callback(h, h->privdata, emsg->reason, emsg->arg);
186             }
187
188             if (msg->command == CMAN_CMD_CONFCHG && h->confchg_callback)
189             {
190                 struct sock_confchg_message *cmsg = (struct sock_confchg_message *)msg;
191
192                 h->confchg_callback(h, h->privdata,
193                                     cmsg->entries, cmsg->member_entries,
194                                     &cmsg->entries[cmsg->member_entries], cmsg->left_entries,
195                                     &cmsg->entries[cmsg->member_entries+cmsg->left_entries], cmsg->joined_
```

```
196         }
197     }
198 }
199
200     return 0;
201 }
202
203 static int loopy_writev(int fd, struct iovec *iovptr, size_t iovlen)
204 {
205     size_t byte_cnt=0;
206     int len;
207
208     while (iovlen > 0)
209     {
210         len = writev(fd, iovptr, iovlen);
211         if (len <= 0)
212             return len;
213
214         byte_cnt += len;
215         while (len >= iovptr->iov_len)
216         {
217             len -= iovptr->iov_len;
218             iovptr++;
219             iovlen--;
220         }
221
222         if ((ssize_t)iovlen <= 0 )
223             break;
224
225         iovptr->iov_base = (char *)iovptr->iov_base + len;
226         iovptr->iov_len -= len;
227     }
228     return byte_cnt;
229 }
230
231
232 static int send_message(struct cman_handle *h, int msgtype, const void *inbuf, int inlen)
233 {
234     struct sock_header header;
235     int len;
236     struct iovec iov[2];
237     size_t iovlen = 1;
238
239     header.magic = CMAN_MAGIC;
240     header.version = CMAN_VERSION;
241     header.command = msgtype;
242     header.flags = 0;
243     header.length = sizeof(header) + inlen;
244
245     iov[0].iov_len = sizeof(header);
246     iov[0].iov_base = &header;
247     if (inbuf)
248     {
249         iov[1].iov_len = inlen;
250         iov[1].iov_base = (void *) inbuf;
251         iovlen++;
252     }
253
254     len = loopy_writev(h->fd, iov, iovlen);
255     if (len < 0)
256         return len;
257     return 0;
258 }
259
260 /* Does something similar to the ioctl calls */
261 static int info_call(struct cman_handle *h, int msgtype, const void *inbuf, int inlen, void *outbuf, int outlen)
262 {
263     if (send_message(h, msgtype, inbuf, inlen))
264         return -1;
265
266     return wait_for_reply(h, outbuf, outlen);
267 }
268
269 static cman_handle_t open_socket(const char *name, int namelen, void *privdata)
270 {
271     struct cman_handle *h;
272     struct sockaddr_un sockaddr;
273
274     h = malloc(sizeof(struct cman_handle));
275     if (!h)
276         return NULL;
277
278     h->magic = CMAN_MAGIC;
279     h->privdata = privdata;
280     h->event_callback = NULL;
281     h->data_callback = NULL;
282     h->confchg_callback = NULL;
283     h->want_reply = 0;
284     h->saved_data_msg = NULL;
285     h->saved_event_msg = NULL;
286     h->saved_reply_msg = NULL;
287
288     h->fd = socket(PF_UNIX, SOCK_STREAM, 0);
289     if (h->fd == -1)
290     {
291         int saved_errno = errno;
292         free(h);
293         errno = saved_errno;
```

```
294         return NULL;
295     }
296
297     fcntl(h->fd, F_SETFD, 1); /* Set close-on-exec */
298     memset(&sockaddr, 0, sizeof(sockaddr));
299     memcpy(sockaddr.sun_path, name, namelen);
300     sockaddr.sun_family = AF_UNIX;
301
302     if (connect(h->fd, (struct sockaddr *) &sockaddr, sizeof(sockaddr)) < 0)
303     {
304         int saved_errno = errno;
305         close(h->fd);
306         free(h);
307         errno = saved_errno;
308         return NULL;
309     }
310
311     /* Get a handle on /dev/zero too. This is always active so we
312     can return it from cman_get_fd() if we have cached messages */
313     h->zero_fd = open("/dev/zero", O_RDONLY);
314     if (h->zero_fd < 0)
315     {
316         int saved_errno = errno;
317         close(h->fd);
318         free(h);
319         h = NULL;
320         errno = saved_errno;
321     } else
322     {
323         fcntl(h->zero_fd, F_SETFD, 1); /* Set close-on-exec */
324     }
325     return (cman_handle_t)h;
326
327 cman_handle_t cman_admin_init(void *privdata)
328 {
329     return open_socket(ADMIN_SOCKNAME, sizeof(ADMIN_SOCKNAME), privdata);
330 }
331
332 cman_handle_t cman_init(void *privdata)
333 {
334     return open_socket(CLIENT_SOCKNAME, sizeof(CLIENT_SOCKNAME), privdata);
335 }
336
337 int cman_finish(cman_handle_t handle)
338 {
339     struct cman_handle *h = (struct cman_handle *)handle;
340     VALIDATE_HANDLE(h);
341
342     h->magic = 0;
343     close(h->fd);
344     close(h->zero_fd);
345     free(h);
346
347     return 0;
348 }
349
350 int cman_setprivdata(cman_handle_t handle, void *privdata)
351 {
352     struct cman_handle *h = (struct cman_handle *)handle;
353     VALIDATE_HANDLE(h);
354
355     h->privdata = privdata;
356     return 0;
357 }
358
359 int cman_getprivdata(cman_handle_t handle, void **privdata)
360 {
361     struct cman_handle *h = (struct cman_handle *)handle;
362     VALIDATE_HANDLE(h);
363
364     *privdata = h->privdata;
365
366     return 0;
367 }
368
369
370 int cman_start_notification(cman_handle_t handle, cman_callback_t callback)
371 {
372     struct cman_handle *h = (struct cman_handle *)handle;
373     VALIDATE_HANDLE(h);
374
375     if (!callback)
376     {
377         errno = EINVAL;
378         return -1;
379     }
380     if (info_call(h, CMAN_CMD_NOTIFY, NULL, 0, NULL, 0))
381         return -1;
382     h->event_callback = callback;
383
384     return 0;
385 }
386
387 int cman_stop_notification(cman_handle_t handle)
388 {
389     struct cman_handle *h = (struct cman_handle *)handle;
390     VALIDATE_HANDLE(h);
391 }
```

```
392         if (info_call(h, CMAN_CMD_REMOVENOTIFY, NULL, 0, NULL, 0))
393             return -1;
394         h->event_callback = NULL;
395
396         return 0;
397     }
398
399 int cman_start_confchg(cman_handle_t handle, cman_confchgcallback_t callback)
400 {
401     struct cman_handle *h = (struct cman_handle *)handle;
402     VALIDATE_HANDLE(h);
403
404     if (!callback)
405     {
406         errno = EINVAL;
407         return -1;
408     }
409     if (info_call(h, CMAN_CMD_START_CONFCHG, NULL, 0, NULL, 0))
410         return -1;
411     h->confchg_callback = callback;
412
413     return 0;
414 }
415
416 int cman_stop_confchg(cman_handle_t handle)
417 {
418     struct cman_handle *h = (struct cman_handle *)handle;
419     VALIDATE_HANDLE(h);
420
421     if (info_call(h, CMAN_CMD_STOP_CONFCHG, NULL, 0, NULL, 0))
422         return -1;
423     h->confchg_callback = NULL;
424
425     return 0;
426 }
427
428
429 int cman_get_fd(cman_handle_t handle)
430 {
431     struct cman_handle *h = (struct cman_handle *)handle;
432     VALIDATE_HANDLE(h);
433
434     /* If we have saved messages then return an FD to /dev/zero which
435        will always be readable */
436     if (h->saved_data_msg || h->saved_event_msg || h->saved_reply_msg)
437         return h->zero_fd;
438     else
439         return h->fd;
440 }
441
442 int cman_dispatch(cman_handle_t handle, int flags)
443 {
444     struct cman_handle *h = (struct cman_handle *)handle;
445     int len;
446     int offset;
447     int rcv_flags = 0;
448     char buf[PIPE_BUF];
449     VALIDATE_HANDLE(h);
450
451     if (!(flags & CMAN_DISPATCH_BLOCKING))
452         rcv_flags |= MSG_DONTWAIT;
453
454     do
455     {
456         int res;
457         char *bufptr = buf;
458         struct sock_header *header = (struct sock_header *)buf;
459
460         /* First, drain any waiting queues */
461         if (h->saved_reply_msg && !(flags & CMAN_DISPATCH_IGNORE_REPLY))
462         {
463             struct saved_message *smsg = h->saved_reply_msg;
464
465             res = process_cman_message(h, flags, smsg->msg);
466             h->saved_reply_msg = smsg->next;
467             len = smsg->msg->length;
468             free(smsg->msg);
469             free(smsg);
470             if (res || (flags & CMAN_DISPATCH_TYPE_MASK) == CMAN_DISPATCH_ONE)
471                 break;
472             else
473                 continue;
474         }
475         if (h->saved_data_msg && !(flags & CMAN_DISPATCH_IGNORE_DATA))
476         {
477             struct saved_message *smsg = h->saved_data_msg;
478
479             res = process_cman_message(h, flags, smsg->msg);
480             h->saved_data_msg = smsg->next;
481             len = smsg->msg->length;
482             free(smsg->msg);
483             free(smsg);
484             if (res || (flags & CMAN_DISPATCH_TYPE_MASK) == CMAN_DISPATCH_ONE)
485                 break;
486             else
487                 continue;
488         }
489         if (h->saved_event_msg && !(flags & CMAN_DISPATCH_IGNORE_EVENT))
```

```

490         {
491             struct saved_message *smsg = h->saved_event_msg;
492
493             res = process_cman_message(h, flags, smsg->msg);
494             h->saved_event_msg = smsg->next;
495             len = smsg->msg->length;
496             free(smsg->msg);
497             free(smsg);
498             if (res || (flags & CMAN_DISPATCH_TYPE_MASK) == CMAN_DISPATCH_ONE)
499                 break;
500             else
501                 continue;
502         }
503
504         /* Now look for new messages */

```

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

Also see events: [\[tainted_data_transitive\]](#)[\[tainted_data_transitive\]](#)[\[var_assign_var\]](#)[\[tainted_data\]](#)

```

505         len = recv(h->fd, buf, sizeof(struct sock_header), recv_flags);
506

```

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

```

507         if (len == 0) {
508             errno = EHOSTDOWN;
509             return -1;
510         }
511

```

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

```

512         if (len < 0 &&
513             (errno == EINTR || errno == EAGAIN))
514             return 0;
515

```

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

```

516         if (len < 0)
517             return -1;
518
519         offset = len;
520
521         /* It's too big! */

```

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

```

522         if (header->length > sizeof(buf))
523         {
524             bufptr = malloc(header->length);

```

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

```

525             if (!bufptr)
526                 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: [\[tainted_data_argument\]](#)[\[var_assign_var\]](#)[\[tainted_data\]](#)

```

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

```

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

Also see events: [\[tainted_data_argument\]](#)[\[tainted_data_transitive\]](#)[\[tainted_data_transitive\]](#)[\[tainted_data\]](#)

```

528             header = (struct sock_header *)bufptr;
529         }
530
531         /* Read the rest */

```

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

Also see events: [\[tainted_data_argument\]](#)[\[tainted_data_transitive\]](#)[\[tainted_data_transitive\]](#)[\[var_assign_var\]](#)

```

532         while (offset < header->length)
533         {
534             len = read(h->fd, bufptr+offset, header->length-offset);
535             if (len == 0) {
536                 if (bufptr != buf)
537                     free(bufptr);
538                 errno = EHOSTDOWN;
539                 return -1;
540             }
541
542             if (len < 0 &&
543                 (errno == EINTR || errno == EAGAIN)) {
544                 if (bufptr != buf)
545                     free(bufptr);
546                 return 0;
547             }
548

```

```
549         if (len < 0) {
550             if (bufptr != buf)
551                 free(bufptr);
552             return -1;
553         }
554         offset += len;
555     }
556
557     res = process_cman_message(h, flags, header);
558     if (bufptr != buf)
559         free(bufptr);
560
561     if (res)
562         break;
563
564     } while ( flags & CMAN_DISPATCH_ALL &&
565             !(len < 0 && errno == EAGAIN) );
566
567     return len;
568 }
569
570 /* GET_ALLMEMBERS returns the number of nodes as status */
571 int cman_get_node_count(cman_handle_t handle)
572 {
573     struct cman_handle *h = (struct cman_handle *)handle;
574     VALIDATE_HANDLE(h);
575
576     return info_call(h, CMAN_CMD_GETALLMEMBERS, NULL, 0, NULL, 0);
577 }
578
579 int cman_get_nodes(cman_handle_t handle, int maxnodes, int *retnodes, cman_node_t *nodes)
580 {
581     struct cman_handle *h = (struct cman_handle *)handle;
582     struct cl_cluster_node *cman_nodes;
583     int status;
584     int buflen;
585     int count = 0;
586     VALIDATE_HANDLE(h);
587
588     if (!retnodes || !nodes || maxnodes < 1)
589     {
590         errno = EINVAL;
591         return -1;
592     }
593
594     buflen = sizeof(struct cl_cluster_node) * maxnodes;
595     cman_nodes = malloc(buflen);
596     if (!cman_nodes)
597         return -1;
598
599     status = info_call(h, CMAN_CMD_GETALLMEMBERS, NULL, 0, cman_nodes, buflen);
600     if (status < 0)
601     {
602         int saved_errno = errno;
603         free(cman_nodes);
604         errno = saved_errno;
605         return -1;
606     }
607
608     if (cman_nodes[0].size != sizeof(struct cl_cluster_node))
609     {
610         free(cman_nodes);
611         errno = EINVAL;
612         return -1;
613     }
614
615     if (status > maxnodes)
616         status = maxnodes;
617
618     for (count = 0; count < status; count++)
619     {
620         copy_node(&nodes[count], &cman_nodes[count]);
621     }
622     free(cman_nodes);
623     *retnodes = status;
624     return 0;
625 }
626
627 int cman_get_disallowed_nodes(cman_handle_t handle, int maxnodes, int *retnodes, cman_node_t *nodes)
628 {
629     struct cman_handle *h = (struct cman_handle *)handle;
630     struct cl_cluster_node *cman_nodes;
631     int status;
632     int buflen;
633     int count = 0;
634     int out_count = 0;
635     VALIDATE_HANDLE(h);
636
637     if (!retnodes || !nodes || maxnodes < 1)
638     {
639         errno = EINVAL;
640         return -1;
641     }
642
643     buflen = sizeof(struct cl_cluster_node) * maxnodes;
644     cman_nodes = malloc(buflen);
645     if (!cman_nodes)
646         return -1;
```

```
647
648     status = info_call(h, CMAN_CMD_GETALLMEMBERS, NULL, 0, cman_nodes, buflen);
649     if (status < 0)
650     {
651         int saved_errno = errno;
652         free(cman_nodes);
653         errno = saved_errno;
654         return -1;
655     }
656
657     if (cman_nodes[0].size != sizeof(struct cl_cluster_node))
658     {
659         free(cman_nodes);
660         errno = EINVAL;
661         return -1;
662     }
663
664     for (count = 0; count < status; count++)
665     {
666         if (cman_nodes[count].state == NODESTATE_AISONLY && out_count < maxnodes)
667             copy_node(&nodes[out_count++], &cman_nodes[count]);
668     }
669     free(cman_nodes);
670     *retnodes = out_count;
671     return 0;
672 }
673
674 int cman_get_node(cman_handle_t handle, int nodeid, cman_node_t *node)
675 {
676     struct cman_handle *h = (struct cman_handle *)handle;
677     struct cl_cluster_node cman_node;
678     int status;
679     VALIDATE_HANDLE(h);
680
681     if (!node || strlen(node->cn_name) >= sizeof(cman_node.name))
682     {
683         errno = EINVAL;
684         return -1;
685     }
686
687     cman_node.node_id = nodeid;
688     strncpy(cman_node.name, node->cn_name, sizeof(cman_node.name) - 1);
689     status = info_call(h, CMAN_CMD_GETNODE, &cman_node, sizeof(struct cl_cluster_node),
690                     &cman_node, sizeof(struct cl_cluster_node));
691     if (status < 0)
692         return -1;
693
694     copy_node(node, &cman_node);
695
696     return 0;
697 }
698
699 int cman_get_node_extra(cman_handle_t handle, int nodeid, cman_node_extra_t *node)
700 {
701     struct cman_handle *h = (struct cman_handle *)handle;
702     int status;
703     VALIDATE_HANDLE(h);
704
705     status = info_call(h, CMAN_CMD_GETNODE_EXTRA, &nodeid, sizeof(int),
706                     node, sizeof(cman_node_extra_t));
707     if (status < 0)
708         return -1;
709
710     return 0;
711 }
712
713 int cman_get_subsys_count(cman_handle_t handle)
714 {
715     struct cman_handle *h = (struct cman_handle *)handle;
716     VALIDATE_HANDLE(h);
717
718     return info_call(h, CMAN_CMD_GET_JOINCOUNT, NULL, 0, NULL, 0);
719 }
720
721 int cman_is_active(cman_handle_t handle)
722 {
723     struct cman_handle *h = (struct cman_handle *)handle;
724     VALIDATE_HANDLE(h);
725
726     return info_call(h, CMAN_CMD_ISACTIVE, NULL, 0, NULL, 0);
727 }
728
729 int cman_is_listening(cman_handle_t handle, int nodeid, uint8_t port)
730 {
731     struct cman_handle *h = (struct cman_handle *)handle;
732     struct cl_listen_request req;
733     VALIDATE_HANDLE(h);
734
735     req.port = port;
736     req.nodeid = nodeid;
737     return info_call(h, CMAN_CMD_ISLISTENING, &req, sizeof(struct cl_listen_request), NULL, 0);
738 }
739
740 int cman_is_quorate(cman_handle_t handle)
741 {
742     struct cman_handle *h = (struct cman_handle *)handle;
743     VALIDATE_HANDLE(h);
744 }
```



```
745         return info_call(h, CMAN_CMD_ISQUORATE, NULL, 0, NULL, 0);
746     }
747
748
749 int cman_get_version(cman_handle_t handle, cman_version_t *version)
750 {
751     struct cman_handle *h = (struct cman_handle *)handle;
752     VALIDATE_HANDLE(h);
753
754     if (!version)
755     {
756         errno = EINVAL;
757         return -1;
758     }
759     return info_call(h, CMAN_CMD_GET_VERSION, NULL, 0, version, sizeof(cman_version_t));
760 }
761
762 int cman_set_version(cman_handle_t handle, const cman_version_t *version)
763 {
764     struct cman_handle *h = (struct cman_handle *)handle;
765     VALIDATE_HANDLE(h);
766
767     if (!version)
768     {
769         errno = EINVAL;
770         return -1;
771     }
772     return info_call(h, CMAN_CMD_SET_VERSION, version, sizeof(cman_version_t), NULL, 0);
773 }
774
775 int cman_kill_node(cman_handle_t handle, int nodeid)
776 {
777     struct cman_handle *h = (struct cman_handle *)handle;
778     VALIDATE_HANDLE(h);
779
780     if (!nodeid)
781     {
782         errno = EINVAL;
783         return -1;
784     }
785     return info_call(h, CMAN_CMD_KILLNODE, &nodeid, sizeof(nodeid), NULL, 0);
786 }
787
788 int cman_set_votes(cman_handle_t handle, int votes, int nodeid)
789 {
790     struct cman_handle *h = (struct cman_handle *)handle;
791     struct cl_set_votes newv;
792     VALIDATE_HANDLE(h);
793
794     if (!votes)
795     {
796         errno = EINVAL;
797         return -1;
798     }
799     newv.nodeid = nodeid;
800     newv.newvotes = votes;
801     return info_call(h, CMAN_CMD_SET_VOTES, &newv, sizeof(newv), NULL, 0);
802 }
803
804 int cman_set_expected_votes(cman_handle_t handle, int evotes)
805 {
806     struct cman_handle *h = (struct cman_handle *)handle;
807     VALIDATE_HANDLE(h);
808
809     if (!evotes)
810     {
811         errno = EINVAL;
812         return -1;
813     }
814     return info_call(h, CMAN_CMD_SETEXPECTED_VOTES, &evotes, sizeof(evotes), NULL, 0);
815 }
816
817 int cman_leave_cluster(cman_handle_t handle, int reason)
818 {
819     struct cman_handle *h = (struct cman_handle *)handle;
820     VALIDATE_HANDLE(h);
821
822     return info_call(h, CMAN_CMD_LEAVE_CLUSTER, &reason, sizeof(reason), NULL, 0);
823 }
824
825 int cman_get_cluster(cman_handle_t handle, cman_cluster_t *clinfo)
826 {
827     struct cman_handle *h = (struct cman_handle *)handle;
828     VALIDATE_HANDLE(h);
829
830     if (!clinfo)
831     {
832         errno = EINVAL;
833         return -1;
834     }
835     return info_call(h, CMAN_CMD_GETCLUSTER, NULL, 0, clinfo, sizeof(cman_cluster_t));
836 }
837
838 int cman_get_extra_info(cman_handle_t handle, cman_extra_info_t *info, int maxlen)
839 {
840     struct cman_handle *h = (struct cman_handle *)handle;
841     VALIDATE_HANDLE(h);
842 }
```

```
843         if (!info || maxlen < sizeof(cman_extra_info_t))
844         {
845             errno = EINVAL;
846             return -1;
847         }
848         return info_call(h, CMAN_CMD_GETEXTRAINFO, NULL, 0, info, maxlen);
849     }
850
851 int cman_send_data(cman_handle_t handle, const void *buf, int len, int flags, uint8_t port, int nodeid)
852 {
853     struct cman_handle *h = (struct cman_handle *)handle;
854     struct iovec iov[2];
855     struct sock_data_header header;
856     VALIDATE_HANDLE(h);
857
858     header.header.magic = CMAN_MAGIC;
859     header.header.version = CMAN_VERSION;
860     header.header.command = CMAN_CMD_DATA;
861     header.header.flags = flags;
862     header.header.length = len + sizeof(header);
863     header.nodeid = nodeid;
864     header.port = port;
865
866     iov[0].iov_len = sizeof(header);
867     iov[0].iov_base = &header;
868     iov[1].iov_len = len;
869     iov[1].iov_base = (void *) buf;
870
871     return loopy_writev(h->fd, iov, 2);
872 }
873
874
875 int cman_start_recv_data(cman_handle_t handle, cman_datacallback_t callback, uint8_t port)
876 {
877     struct cman_handle *h = (struct cman_handle *)handle;
878     int portparam;
879     int status;
880     VALIDATE_HANDLE(h);
881
882     /* Do a "bind" */
883     portparam = port;
884     status = info_call(h, CMAN_CMD_BIND, &portparam, sizeof(portparam), NULL, 0);
885
886     if (status == 0)
887         h->data_callback = callback;
888
889     return status;
890 }
891
892 int cman_end_recv_data(cman_handle_t handle)
893 {
894     struct cman_handle *h = (struct cman_handle *)handle;
895     VALIDATE_HANDLE(h);
896
897     h->data_callback = NULL;
898     return 0;
899 }
900
901
902 int cman_barrier_register(cman_handle_t handle, const char *name, int flags, int nodes)
903 {
904     struct cman_handle *h = (struct cman_handle *)handle;
905     struct cl_barrier_info binfo;
906     VALIDATE_HANDLE(h);
907
908     if (strlen(name) >= MAX_BARRIER_NAME_LEN)
909     {
910         errno = EINVAL;
911         return -1;
912     }
913
914     binfo.cmd = BARRIER_CMD_REGISTER;
915     strncpy(binfo.name, name, sizeof(binfo.name) - 1);
916     binfo.arg = nodes;
917     binfo.flags = flags;
918
919     return info_call(h, CMAN_CMD_BARRIER, &binfo, sizeof(binfo), NULL, 0);
920 }
921
922
923 int cman_barrier_change(cman_handle_t handle, const char *name, int flags, int arg)
924 {
925     struct cman_handle *h = (struct cman_handle *)handle;
926     struct cl_barrier_info binfo;
927     VALIDATE_HANDLE(h);
928
929     if (strlen(name) >= MAX_BARRIER_NAME_LEN)
930     {
931         errno = EINVAL;
932         return -1;
933     }
934
935     binfo.cmd = BARRIER_CMD_CHANGE;
936     strncpy(binfo.name, name, sizeof(binfo.name) - 1);
937     binfo.arg = arg;
938     binfo.flags = flags;
939
940     return info_call(h, CMAN_CMD_BARRIER, &binfo, sizeof(binfo), NULL, 0);
```

```
941     }
942 }
943
944 int cman_barrier_wait(cman_handle_t handle, const char *name)
945 {
946     struct cman_handle *h = (struct cman_handle *)handle;
947     struct cl_barrier_info binfo;
948     VALIDATE_HANDLE(h);
949
950     if (strlen(name) >= MAX_BARRIER_NAME_LEN)
951     {
952         errno = EINVAL;
953         return -1;
954     }
955
956     binfo.cmd = BARRIER_CMD_WAIT;
957     strncpy(bininfo.name, name, sizeof(bininfo.name) - 1);
958
959     return info_call(h, CMAN_CMD_BARRIER, &binfo, sizeof(bininfo), NULL, 0);
960 }
961
962 int cman_barrier_delete(cman_handle_t handle, const char *name)
963 {
964     struct cman_handle *h = (struct cman_handle *)handle;
965     struct cl_barrier_info binfo;
966     VALIDATE_HANDLE(h);
967
968     if (strlen(name) >= MAX_BARRIER_NAME_LEN)
969     {
970         errno = EINVAL;
971         return -1;
972     }
973
974     binfo.cmd = BARRIER_CMD_DELETE;
975     strncpy(bininfo.name, name, sizeof(bininfo.name) - 1);
976
977     return info_call(h, CMAN_CMD_BARRIER, &binfo, sizeof(bininfo), NULL, 0);
978 }
979
980 int cman_shutdown(cman_handle_t handle, int flags)
981 {
982     struct cman_handle *h = (struct cman_handle *)handle;
983     VALIDATE_HANDLE(h);
984
985     return info_call(h, CMAN_CMD_TRY_SHUTDOWN, &flags, sizeof(int), NULL, 0);
986 }
987
988 int cman_set_dirty(cman_handle_t handle)
989 {
990     struct cman_handle *h = (struct cman_handle *)handle;
991     VALIDATE_HANDLE(h);
992
993     return info_call(h, CMAN_CMD_SET_DIRTY, NULL, 0, NULL, 0);
994 }
995
996 int cman_set_debuglog(cman_handle_t handle, int subsystems)
997 {
998     struct cman_handle *h = (struct cman_handle *)handle;
999     VALIDATE_HANDLE(h);
1000
1001     return info_call(h, CMAN_CMD_SET_DEBUGLOG, &subsystems, sizeof(int), NULL, 0);
1002 }
1003
1004 int cman_replyto_shutdown(cman_handle_t handle, int yesno)
1005 {
1006     struct cman_handle *h = (struct cman_handle *)handle;
1007     VALIDATE_HANDLE(h);
1008
1009     send_message(h, CMAN_CMD_SHUTDOWN_REPLY, &yesno, sizeof(int));
1010     return 0;
1011 }
1012
1013 static int cman_set_quorum_device(cman_handle_t handle,
1014                                 int ops,
1015                                 char *name, int votes)
1016 {
1017     struct cman_handle *h = (struct cman_handle *)handle;
1018     char buf[strlen(name)+1 + sizeof(int)];
1019     VALIDATE_HANDLE(h);
1020
1021     memcpy(buf, &votes, sizeof(int));
1022     strncpy(buf+sizeof(int), name, strlen(name)+1 + sizeof(int) - 1);
1023     return info_call(h, ops, buf, strlen(name)+1+sizeof(int), NULL, 0);
1024 }
1025
1026 int cman_register_quorum_device(cman_handle_t handle, char *name, int votes)
1027 {
1028     if ((!name) || (strlen(name) > MAX_CLUSTER_MEMBER_NAME_LEN) || (votes < 0))
1029     {
1030         errno = EINVAL;
1031         return -1;
1032     }
1033     return cman_set_quorum_device(handle, CMAN_CMD_REG_QUORUMDEV, name, votes);
1034 }
1035
1036 int cman_unregister_quorum_device(cman_handle_t handle)
1037 {
1038     struct cman_handle *h = (struct cman_handle *)handle;
```

```
1039     VALIDATE_HANDLE(h);
1040
1041     return info_call(h, CMAN_CMD_UNREG_QUORUMDEV, NULL, 0, NULL, 0);
1042 }
1043
1044 int cman_poll_quorum_device(cman_handle_t handle, int isavailable)
1045 {
1046     struct cman_handle *h = (struct cman_handle *)handle;
1047     VALIDATE_HANDLE(h);
1048
1049     return info_call(h, CMAN_CMD_POLL_QUORUMDEV, &isavailable, sizeof(int), NULL, 0);
1050 }
1051
1052 int cman_get_quorum_device(cman_handle_t handle, struct cman_qdev_info *info)
1053 {
1054     struct cman_handle *h = (struct cman_handle *)handle;
1055     int ret;
1056     struct cl_cluster_node cman_node;
1057     VALIDATE_HANDLE(h);
1058
1059     cman_node.node_id = CLUSTER_GETNODE_QUORUMDEV;
1060     ret = info_call(h, CMAN_CMD_GETNODE, &cman_node, sizeof(cman_node), &cman_node, sizeof(cman_node));
1061     if (!ret) {
1062         strncpy(info->qname, cman_node.name, sizeof(info->qname) - 1);
1063         info->qstate = cman_node.state;
1064         info->qvotes = cman_node.votes;
1065     }
1066     return ret;
1067 }
1068
1069 int cman_update_quorum_device(cman_handle_t handle, char *name, int votes)
1070 {
1071     if ((!name) || (strlen(name) > MAX_CLUSTER_MEMBER_NAME_LEN) || (votes < 0))
1072     {
1073         errno = EINVAL;
1074         return -1;
1075     }
1076     return cman_set_quorum_device(handle, CMAN_CMD_UPDATE_QUORUMDEV, name, votes);
1077 }
1078
1079 int cman_get_fenceinfo(cman_handle_t handle, int nodeid, uint64_t *time, int *fenced, char *agent)
1080 {
1081     struct cman_handle *h = (struct cman_handle *)handle;
1082     int ret;
1083     struct cl_fence_info f;
1084     VALIDATE_HANDLE(h);
1085
1086     ret = info_call(h, CMAN_CMD_GET_FENCE_INFO, &nodeid, sizeof(int), &f, sizeof(f));
1087     if (!ret) {
1088         *time = f.fence_time;
1089         if (agent)
1090             strncpy(agent, f.fence_agent, sizeof(f.fence_agent) - 1);
1091         *fenced = ((f.flags & FENCE_FLAGS_FENCED) != 0);
1092     }
1093     return ret;
1094 }
1095
1096 int cman_get_node_addrs(cman_handle_t handle, int nodeid, int max_addrs, int *num_addrs, struct cman_node_address *addrs)
1097 {
1098     struct cman_handle *h = (struct cman_handle *)handle;
1099     int ret;
1100     char buf[sizeof(struct cl_get_node_addrs) + sizeof(struct cl_node_addrs)*max_addrs];
1101     struct cl_get_node_addrs *outbuf = (struct cl_get_node_addrs *)buf;
1102     VALIDATE_HANDLE(h);
1103
1104     ret = info_call(h, CMAN_CMD_GET_NODEADDRESSES, &nodeid, sizeof(int), buf, sizeof(buf));
1105     if (!ret) {
1106         int i;
1107
1108         *num_addrs = outbuf->numaddrs;
1109
1110         if (outbuf->numaddrs > max_addrs)
1111             outbuf->numaddrs = max_addrs;
1112
1113         for (i=0; i < outbuf->numaddrs; i++) {
1114             memcpy(&addrs[i].cna_address, &outbuf->addrs[i].addr, outbuf->addrs[i].addrlen);
1115             addrs[i].cna_addrlen = outbuf->addrs[i].addrlen;
1116         }
1117     }
1118     return ret;
1119 }
1120
1121 int cman_node_fenced(cman_handle_t handle, int nodeid, uint64_t time, char *agent)
1122 {
1123     struct cman_handle *h = (struct cman_handle *)handle;
1124     struct cl_fence_info f;
1125     VALIDATE_HANDLE(h);
1126
1127     if (strlen(agent) >= MAX_FENCE_AGENT_NAME_LEN) {
1128         errno = EINVAL;
1129         return -1;
1130     }
1131
1132     f.nodeid = nodeid;
1133     f.fence_time = time;
1134     strncpy(f.fence_agent, agent, sizeof(f.fence_agent) - 1);
1135     return info_call(h, CMAN_CMD_UPDATE_FENCE_INFO, &f, sizeof(f), NULL, 0);
1136 }
```

