Subject: [patch 3/5][RFC - ipv4/udp checkpoint/restart] : c/r the socket information and options
Posted by Daniel Lezcano on Wed, 06 Jun 2007 12:18:09 GMT
View Forum Message <> Reply to Message

From: Daniel Lezcano <dlezcano@fr.ibm.com>

This patch defines a set of netlink attributes to store/retrieve socket
options.

 * At dump time, a netlink message specify the inode of the socket to
   be checkpointed. The socket is retrieved with the inode number. A
   new netlink message is built in order to store the socket information.
   The type, state and socket options are stored into it and the netlink
   message is transmitted to the requestor.

 * At restore time, the netlink message contains the type of the socket.
   A new socket is created, using this type and the attributes are browsed
   in order to use the values to restore the differents options.

The choice of the C/R is to stick as much as possible to the user/kernel
frontier. For this reason, the kernel_{set,get}sockopt are used. That allows
to reduce code and delegate the different checks to the corresponding function.
Unfortunatly, some get/set are not symetric, so some options can be retrieved
but not set and vice-versa. For this reason, there are a few helpers, and the
option definitions contains a GET|SET|BOTH flag.

Signed-off-by: Daniel Lezcano <dlezcano@fr.ibm.com>
---
 include/linux/af_inet_cr.h | 61 ++++
 net/ipv4/af_inet_cr.c | 640 ++-
 2 files changed, 680 insertions(+), 21 deletions(-)

Index: 2.6.20-cr/net/ipv4/af_inet_cr.c
===
--- 2.6.20-cr.orig/net/ipv4/af_inet_cr.c
+++ 2.6.20-cr/net/ipv4/af_inet_cr.c
@@ -12,36 +12,644 @@
 #include <net/genetlink.h>
 #include <net/sock.h>
 #include <linux/fs.h>
+#include <linux/syscalls.h>
 #include <linux/af_inet_cr.h>

 /*
- * af_inet_cr_nldump : this function is called when a netlink message is received
- * with AF_INET_CR_CMD_DUMP command.

```
+ * Netlink message policy definition
+ */
+static struct nla_policy af_inet_cr_policy[AF_INET_CR_ATTR_MAX] = {
+ [AF_INET_CR_ATTR_INODE]              = { .type = NLA_U32 },
+
+ [AF_INET_CR_ATTR_SOCK_STATE]          = { .type = NLA_U32 },
+ [AF_INET_CR_ATTR_SOCK_TYPE]           = { .type = NLA_U32 },
+
+ [AF_INET_CR_ATTR_SOCKOPT_BROADCAST]    = { .type = NLA_FLAG },
+ [AF_INET_CR_ATTR_SOCKOPT_DEBUG]        = { .type = NLA_FLAG },
+ [AF_INET_CR_ATTR_SOCKOPT_DONTROUTE]    = { .type = NLA_FLAG },
+ [AF_INET_CR_ATTR_SOCKOPT_KEEPALIVE]    = { .type = NLA_FLAG },
+ [AF_INET_CR_ATTR_SOCKOPT_OOBINLINE]    = { .type = NLA_FLAG },
+ [AF_INET_CR_ATTR_SOCKOPT_PASSCRED]     = { .type = NLA_FLAG },
+ [AF_INET_CR_ATTR_SOCKOPT_REUSEADDR]    = { .type = NLA_FLAG },
+ [AF_INET_CR_ATTR_SOCKOPT_TIMESTAMP]    = { .type = NLA_FLAG },
+ [AF_INET_CR_ATTR_SOCKOPT_SNDBUF_ULOCK] = { .type = NLA_FLAG },
+ [AF_INET_CR_ATTR_SOCKOPT_RCVBUF_ULOCK] = { .type = NLA_FLAG },
+
+ [AF_INET_CR_ATTR_SOCKOPT_RCVBUF]       = { .type = NLA_U32  },
+ [AF_INET_CR_ATTR_SOCKOPT_SNDBUF]       = { .type = NLA_U32  },
+ [AF_INET_CR_ATTR_SOCKOPT_PRIORITY]     = { .type = NLA_U32  },
+ [AF_INET_CR_ATTR_SOCKOPT_RCVLOWAT]     = { .type = NLA_U32  },
+
+ [AF_INET_CR_ATTR_SOCKOPT_RCVTIMEO] = { .len = sizeof(struct timeval) },
+ [AF_INET_CR_ATTR_SOCKOPT_SNDTIMEO] = { .len = sizeof(struct timeval) },
+ [AF_INET_CR_ATTR_SOCKOPT_LINGER]   = { .len = sizeof(struct linger)  },
+ [AF_INET_CR_ATTR_SOCKOPT_BINDTODEVICE]  = { .len  = IFNAMSIZ },
+};
+
+/*
+ * Generic netlink family definition
+ */
+static struct genl_family af_inet_cr_family = {
+ .id             = GENL_ID_GENERATE,
+ .name           = "af_inet_cr",
+ .version        = 0x1,
+ .maxattr        = AF_INET_CR_ATTR_MAX - 1,
+};
+
+/*
+ * socket options association with netlink attribute
+ */
+struct af_inet_cr_optattr socket_options[] = {
+ { SO_BROADCAST,    AF_INET_CR_ATTR_SOCKOPT_BROADCAST,    0, BOTH },
+ { SO_DEBUG,        AF_INET_CR_ATTR_SOCKOPT_DEBUG,        0, BOTH },
+ { SO_DONTROUTE,    AF_INET_CR_ATTR_SOCKOPT_DONTROUTE,    0, BOTH },
+ { SO_KEEPALIVE,    AF_INET_CR_ATTR_SOCKOPT_KEEPALIVE,    0, BOTH },
```

```
+ { SO_OOBINLINE,    AF_INET_CR_ATTR_SOCKOPT_OOBINLINE,    0, BOTH },
+ { SO_PRIORITY,     AF_INET_CR_ATTR_SOCKOPT_PRIORITY,     0, BOTH },
+ { SO_RCVLOWAT,     AF_INET_CR_ATTR_SOCKOPT_RCVLOWAT,     0, BOTH },
+ { SO_RCVBUF,       AF_INET_CR_ATTR_SOCKOPT_RCVBUF,       0, GET  },
+ { SO_SNDBUF,       AF_INET_CR_ATTR_SOCKOPT_SNDBUF,       0, GET  },
+ { SO_REUSEADDR,    AF_INET_CR_ATTR_SOCKOPT_REUSEADDR,    0, BOTH },
+ { SO_TIMESTAMP,    AF_INET_CR_ATTR_SOCKOPT_TIMESTAMP,    0, BOTH },
+ { SO_LINGER,       AF_INET_CR_ATTR_SOCKOPT_LINGER,       0, BOTH },
+ { SO_RCVTIMEO,     AF_INET_CR_ATTR_SOCKOPT_RCVTIMEO,     0, BOTH },
+ { SO_SNDTIMEO,     AF_INET_CR_ATTR_SOCKOPT_SNDTIMEO,     0, BOTH },
+ { SO_BINDTODEVICE, AF_INET_CR_ATTR_SOCKOPT_BINDTODEVICE, 0, SET  },
+};
+
+
+/*
+ * socket_lookup : search for socket using the inode number
+ *
+ * @sb : superblock associated to the sockfs
+ * @ino : the inode number associated with the socket
+ * @sock : the socket resulting from the lookup
+ *
+ * Returns 0 on succes or if the call fails:
+ *  -ENOENT : inode is not found
+ *  -ENOTSOCK: the inode found is not associated with a socket
+ *  -EINVAL: unexpected error
+ */
+static inline int socket_lookup(struct super_block *sb, unsigned long ino,
+     struct socket **sock)
+{
+ int ret;
+  struct inode *inode;
+
+  inode = ilookup_unhashed(sb, ino);
+  if (!inode)
+  return -ENOENT;
+
+ ret = -ENOTSOCK;
+  if (!S_ISSOCK(inode->i_mode))
+  goto out;
+
+ ret = -EINVAL;
+  *sock = SOCKET_I(inode);
+  if (!*sock)
+  goto out;
+
+ ret = 0;
+out:
+ iput(inode);
```

```
+  return ret;
+}
+
+/*
+ *  af_inet_cr_opt2attr: convert a socket option to a netlink attribute and push it
+ *  to the skbuff
+ *
+ * @skb : the skbuff to be filled
+ * @sock : the socket to retrieve options
+ * @optlevel : the level of the option
+ * @optattr : the correpondance between the option and the attribute
+ *
+ * Return 0 on sucess, < 0 otherwise
+ */
+int af_inet_cr_opt2attr(struct sk_buff *skb, const struct socket *sock, int optlevel,
+   const struct af_inet_cr_optattr *optattr)
+{
+ int attr = optattr->attr;
+ int optname = optattr->optname;
+ int type = af_inet_cr_policy[attr].type;
+ int optlen;
+ char *optbuf = NULL;
+ int optval;
+ int ret;
+
+ if (!(optattr->get_and_set & GET))
+                return 0;
+
+ switch (type) {
+ case NLA_UNSPEC:
+  optlen = af_inet_cr_policy[attr].len;
+  optbuf = kmalloc(optlen, GFP_KERNEL);
+  if (!optbuf)
+   return -ENOMEM;
+  ret = kernel_getsockopt((struct socket *)sock, optlevel,
+      optname, optbuf, &optlen);
+  if (ret)
+   goto out;
+  ret = nla_put(skb, attr, af_inet_cr_policy[attr].len, optbuf);
+  goto out;
+ case NLA_U32:
+  optlen = sizeof(optval);
+  ret = kernel_getsockopt((struct socket *)sock, optlevel,
+      optname, (void *)&optval, &optlen);
+  if (ret)
+   goto out;
+  ret = nla_put_u32(skb, attr, optval);
+  goto out;
```

```
+ case NLA_U8:
+  optlen = sizeof(optval);
+  ret = kernel_getsockopt((struct socket *)sock, optlevel,
+     optname, (void *)&optval, &optlen);
+  if (ret)
+   goto out;
+  ret = nla_put_u8(skb, attr, optval);
+  goto out;
+ case NLA_FLAG:
+  optlen = sizeof(optval);
+  ret = kernel_getsockopt((struct socket *)sock, optlevel,
+     optname, (void *)&optval, &optlen);
+  if (ret)
+   goto out;
+  if (optval)
+   ret = nla_put_flag(skb, attr);
+  goto out;
+ default:
+  ret = -EINVAL;
+               goto out;
+ };
+ out:
+ if (optbuf)
+  kfree(optbuf);
+        return ret;
+}
+
+/*
+ * af_inet_cr_opt2attr : convert a netlink attribute to a socket option
+ *  and set the option to the socket
+ *
+ * @info : the generic netlink message
+ * @sock : the socket to set options
+ * @optlevel : the level of the option
+ * @optattr : the correpondance between the option and the attribute
+ *
+ * Return 0 on sucess, < 0 otherwise
+ */
+int af_inet_cr_attr2opt(const struct genl_info *info, struct socket *sock,
+   int optlevel, const struct af_inet_cr_optattr *optattr)
+{
+        int optname = optattr->optname;
+        int attr = optattr->attr;
+ int ret, type = af_inet_cr_policy[attr].type;
+ struct nlattr *nla = info->attrs[attr];
+ char *optbuf;
+ int optlen;
+ int optval;
```

```
+
+ if (!(optattr->get_and_set & SET))
+  return 0;
+ if (!nla)
+                return 0;
+
+ switch (type) {
+        case NLA_FLAG:
+  optval = nla_get_flag(nla);
+  break;
+ case NLA_U8:
+  optval = nla_get_u8(nla);
+  break;
+        case NLA_U32:
+  optval = nla_get_u32(nla);
+  break;
+ case NLA_UNSPEC:
+  optlen = af_inet_cr_policy[attr].len;
+  optbuf = kmalloc(optlen, GFP_KERNEL);
+  if (!optbuf)
+   return -ENOMEM;
+  nla_memcpy(optbuf, nla, optlen);
+  ret = kernel_setsockopt(sock, optlevel, optname,
+     optbuf, optlen);
+  kfree(optbuf);
+  goto out;
+ default:
+  ret = -EINVAL;
+  goto out;
+        }
+
+ optlen = sizeof(optval);
+ ret = kernel_setsockopt(sock, optlevel, optname,
+    (void *)&optval, optlen);
+out:
+ return ret;
+}
+
+/*
+ * dump_sockopt_sndbuf : retrieve the userlock flag on the sndbuf option and
+ *  add it to the netlink message
+ *
+ * @sock : the socket to retrieve the userlock
+ * @skb : the skbuff containing the netlink message
+ *
+ * Returns 0 on success, < 0 otherwise
+ */
+static inline int dump_sockopt_sndbuf(struct socket *sock,
```

```
+          struct sk_buff *skb)
+{
+ int ret;
+ struct sock *sk = sock->sk;
+
+ if (sk->sk_userlocks & SOCK_SNDBUF_LOCK) {
+  ret = nla_put_flag(skb, AF_INET_CR_ATTR_SOCKOPT_SNDBUF_ULOCK);
+  if (ret)
+   return ret;
+ }
+
+ return 0;
+}
+
+/*
+ * dump_sockopt_rcvbuf : retrieve the userlock flag on the rcv option and
+ *  add it to the netlink message
+ *
+ * @sock : the socket to retrieve the userlock
+ * @skb : the skbuff containing the netlink message
+ *
+ * Returns 0 on success, < 0 otherwise
+ */
+static inline int dump_sockopt_rcvbuf(struct socket *sock,
+          struct sk_buff *skb)
+{
+ int ret;
+ struct sock *sk = sock->sk;
+
+ if (sk->sk_userlocks & SOCK_RCVBUF_LOCK) {
+  ret = nla_put_flag(skb, AF_INET_CR_ATTR_SOCKOPT_RCVBUF_ULOCK);
+  if (ret)
+   return ret;
+ }
+
+ return 0;
+}
+
+/*
+ * dump_sockopt_bindtodevice : retrieve the bind device option and add it
+ *  to the netlink message. Note that is the name of the device which is
+ *  dumped, not the index
+ *
+ * @sock : the socket to retrieve the bindtodevice option
+ * @skb : the skbuff containing the netlink message
+ *
+ * Returns 0 on success, < 0 otherwise
+ */
```

```
+static inline int dump_sockopt_bindtodevice(struct socket *sock,
+         struct sk_buff *skb)
+{
+ int ret;
+ struct sock *sk = sock->sk;
+ struct net_device *dev = dev_get_by_index(sk->sk_bound_dev_if);
+
+ if (!sk->sk_bound_dev_if)
+  return 0;
+
+ dev = dev_get_by_index(sk->sk_bound_dev_if);
+ if (!dev)
+  return -EINVAL;
+
+ ret = nla_put(skb, AF_INET_CR_ATTR_SOCKOPT_BINDTODEVICE,
+        IFNAMSIZ, dev->name);
+ dev_put(dev);
+
+ return ret;
+}
+
+/*
+ * dump_sockopt_sockopt : retrieve the socket options and store them
+ *  to the netlink message
+ *
+ * @sock : the socket to retrieve the bindtodevice option
+ * @skb : the skbuff containing the netlink message
+ *
+ * Returns 0 on success, < 0 otherwise
+ */
+static int dump_sockopt(struct socket *sock, struct sk_buff *skb)
+{
+ size_t i, len;
+ int ret;
+
+ len = sizeof(socket_options)/sizeof(struct af_inet_cr_optattr);
+
+ for (i = 0; i < len; i++) {
+  ret = af_inet_cr_opt2attr(skb, sock, SOL_SOCKET, &socket_options[i]);
+  if (ret)
+   return ret;
+ }
+
+ ret = dump_sockopt_bindtodevice(sock, skb);
+ if (ret)
+  return ret;
+
+ ret = dump_sockopt_sndbuf(sock, skb);
```

```
+ if (ret)
+  return ret;
+
+ ret = dump_sockopt_rcvbuf(sock, skb);
+ if (ret)
+  return ret;
+
+ return 0;
+}
+
+/*
+ * dump_socket : dump the socket state, type and options into a netlink message
+ *  and transmit the message to the sender of the dump command.
+ *
+ * @sock : socket to be dumped
+ * @pid : pid of the sender
+ *
+ * Returns 0 on success, < 0 otherwise
+ */
+static int dump_socket(struct socket *sock, pid_t pid)
+{
+ struct sock *sk = sock->sk;
+ unsigned short family = sk->sk_family;
+ struct sk_buff *skb;
+ void *msg_head;
+ int ret;
+
+ if (family != AF_INET)
+  return -EINVAL;
+
+ skb = genlmsg_new(NLMSG_GOODSIZE, GFP_KERNEL);
+ if (!skb)
+  return -ENOMEM;
+
+ msg_head = genlmsg_put(skb, pid, 0, &af_inet_cr_family, 0,
+         AF_INET_CR_CMD_DUMP);
+ ret = -ENOMEM;
+ if (!msg_head)
+  goto out;
+
+ ret = nla_put_u32(skb, AF_INET_CR_ATTR_SOCK_STATE, sock->state);
+ if (ret)
+  goto out;
+
+ ret = nla_put_u32(skb, AF_INET_CR_ATTR_SOCK_TYPE, sock->type);
+ if (ret)
+  goto out;
+
```

```
+ ret = dump_sockopt(sock, skb);
+ if (ret)
+  goto out;
+
+ ret = genlmsg_end(skb, msg_head);
+ if (ret < 0)
+  goto out;
+
+ ret = genlmsg_unicast(skb, pid);
+out:
+ if (ret)
+  nlmsg_free(skb);
+ return ret;
+}
+
+/*
+ * restore_sockopt_rcvbuf : extract rcvbuf value from the netlink
+ *  message and restore the socket rcvbuf option. Only of the value
+ *  is specified because the kernel multiplicate value by two, the
+ *  userlock flag is overwritten
+ *
+ * @sock : the socket to be restored
+ * @info : the netlink message
+ *
+ * Returns 0 on success, < 0 otherwise
+ */
+static inline int restore_sockopt_rcvbuf(struct socket *sock,
+      struct genl_info *info)
+{
+ struct nlattr *nla;
+ int val, ret;
+
+ nla = info->attrs[AF_INET_CR_ATTR_SOCKOPT_RCVBUF];
+ if (!nla)
+  return -EINVAL;
+
+ val = nla_get_u32(nla) / 2;
+
+ ret = kernel_setsockopt(sock, SOL_SOCKET, SO_RCVBUFFORCE,
+    (char *)&val, sizeof(val));
+ if (ret)
+  return ret;
+
+ if (!info->attrs[AF_INET_CR_ATTR_SOCKOPT_RCVBUF_ULOCK])
+  sock->sk->sk_userlocks &= ~SOCK_RCVBUF_LOCK;
+
+ return 0;
+}
```

```
+
+/*
+ * restore_sockopt_sndbuf : extract sndbuf value from the netlink
+ *  message and restore the socket sndbuf option. Only of the value
+ *  is specified because the kernel multiplicate value by two, the
+ *  userlock flag is overwritten
+ *
+ * @sock : the socket to be restored
+ * @info : the netlink message
+ *
+ * Returns 0 on success, < 0 otherwise
+ */
+static inline int restore_sockopt_sndbuf(struct socket *sock,
+      struct genl_info *info)
+{
+ struct nlattr *nla;
+ int val, ret;
+
+ nla = info->attrs[AF_INET_CR_ATTR_SOCKOPT_SNDBUF];
+ if (!nla)
+  return -EINVAL;
+
+ val = nla_get_u32(nla) / 2;
+
+ ret = kernel_setsockopt(sock, SOL_SOCKET, SO_SNDBUFFORCE,
+     (char *)&val, sizeof(val));
+ if (ret)
+  return ret;
+
+ if (!info->attrs[AF_INET_CR_ATTR_SOCKOPT_SNDBUF_ULOCK])
+  sock->sk->sk_userlocks &= ~SOCK_SNDBUF_LOCK;
+
+ return ret;
+}
+
+/*
+ * restore_sockopt_sndbuf : restore the socket option using the correponding
+ *  netlink attribute <-> socket option array
+ *
+ * @sock : the socket to be restored
+ * @info : the netlink message
+ *
+ * Returns 0 on success, < 0 otherwise
+ */
+static inline int restore_sockopt(struct socket *sock, struct genl_info *info)
+{
+ size_t i, len;
+ int ret;
```

```
+
+ len = sizeof(socket_options)/sizeof(struct af_inet_cr_optattr);
+
+ for (i = 0; i < len; i++) {
+  ret = af_inet_cr_attr2opt(info, sock, SOL_SOCKET, &socket_options[i]);
+  if (ret)
+   return ret;
+ }
+
+ ret = restore_sockopt_sndbuf(sock, info);
+ if (ret)
+  return ret;
+
+ ret = restore_sockopt_rcvbuf(sock, info);
+ if (ret)
+  return ret;
+
+ return 0;
+}
+
+/*
+ * restore_socket : restore the socket from the netlink message content
+ *
+ * @sock : the socket to be restored
+ * @info : the netlink message
+ *
+ * Returns 0 on success, < 0 otherwise
+ */
+static int restore_socket(struct socket *sock, struct genl_info *info)
+{
+ int ret;
+
+ struct nlattr *nla;
+
+ ret = -EINVAL;
+
+ nla = info->attrs[AF_INET_CR_ATTR_SOCK_STATE];
+ if (!nla)
+  goto out;
+ sock->state = nla_get_u32(nla);
+
+ nla = info->attrs[AF_INET_CR_ATTR_SOCK_TYPE];
+ if (!nla)
+  goto out;
+ sock->type = nla_get_u32(nla);
+
+ ret = restore_sockopt(sock, info);
+ if (ret)
```

```
+  goto out;
+out:
+ return ret;
+}
+
+/*
+ * af_inet_cr_nldump : this function is called when a netlink message is
+ *  received with AF_INET_CR_CMD_DUMP command.
+ *
  * @skb  : the netlink packet giving the restore command
  * @info : the generic netlink message
+ *
+ * Returns 0 on success, < 0 otherwise
  */
 static int af_inet_cr_nldump(struct sk_buff *skb, struct genl_info *info)
 {
- return 0;
+ struct sock *sk = skb->sk;
+ struct socket *sock = sk->sk_socket;
+ struct inode *inode = SOCK_INODE(sock);
+ struct super_block *sb = inode->i_sb;
+ int ret;
+ unsigned long ino;
+ struct nlattr *nla = info->attrs[AF_INET_CR_ATTR_INODE];
+
+ if (!nla)
+  return -EINVAL;
+
+ ino = nla_get_u32(nla);
+
+ ret = socket_lookup(sb, ino, &sock);
+ if (ret)
+  return ret;
+
+ return dump_socket(sock, info->snd_pid);
 }

 /*
  * af_inet_cr_nldump : this function is called when a netlink message is received
- * with AF_INET_CR_CMD_RESTORE command.
+ *  with AF_INET_CR_CMD_RESTORE command.
  * @skb  : the netlink packet giving the restore command
  * @info : the generic netlink message
+ *
+ * Returns 0 on success, < 0 otherwise
  */
 static int af_inet_cr_nlrestore(struct sk_buff *skb, struct genl_info *info)
 {
```

```
+
+ int ret;
+ int type;
+ void *msg_head;
+ struct socket *sock;
+ struct nlattr *nla;
+ struct sk_buff *answer;
+
+ answer = genlmsg_new(NLMSG_GOODSIZE, GFP_KERNEL);
+ if (!answer)
+  return -ENOMEM;
+
+ ret = -ENOMEM;
+ msg_head = genlmsg_put(answer, info->snd_pid, 0, &af_inet_cr_family, 0,
+         AF_INET_CR_CMD_RESTORE);
+ if (!msg_head)
+  goto out;
+
+ ret = -EINVAL;
+ nla = info->attrs[AF_INET_CR_ATTR_SOCK_TYPE];
+ if (!nla)
+  goto out;
+
+ type = nla_get_u32(nla);
+
+ ret = sock_create_kern(AF_INET, type, 0, &sock);
+ if (ret)
+  goto out;
+
+ ret = restore_socket(sock, info);
+ if (ret)
+  goto out_release;
+
+ ret = sock_map_fd(sock);
+ if (ret < 0)
+  goto out_release;
+
+ ret = nla_put_u32(answer, AF_INET_CR_ATTR_SOCK_FD, ret);
+ if (ret)
+  goto out_close;
+
+ ret = genlmsg_end(answer, msg_head);
+ if (ret < 0)
+  goto out_close;
+
+ ret = genlmsg_unicast(answer, info->snd_pid);
+ if (ret)
+  goto out_close;
```

```
+
  return 0;
-}

-/*
- * Netlink message policy definition
- */
-static struct nla_policy af_inet_cr_policy[AF_INET_CR_ATTR_MAX] = {
- [AF_INET_CR_ATTR_INODE] = { .type = NLA_U32 },
-};
+out_close:
+ sys_close(ret);
+out_release:
+ sock_release(sock);
+out:
+ nlmsg_free(answer);
+
+ return ret;
+}

 /*
  * Netlink dumping command configuration
@@ -62,16 +670,6 @@
 };

 /*
- * Generic netlink family definition
- */
-static struct genl_family af_inet_cr_family = {
- .id             = GENL_ID_GENERATE,
- .name           = "af_inet_cr",
- .version        = 0x1,
- .maxattr        = AF_INET_CR_ATTR_MAX - 1,
-};
-
-/*
  * af_inet_cr_init : this function is called at initialization
  * time. It register the generic netlink family associated with
  * this module and hang different ops with it.
Index: 2.6.20-cr/include/linux/af_inet_cr.h
===================================================================
--- 2.6.20-cr.orig/include/linux/af_inet_cr.h
+++ 2.6.20-cr/include/linux/af_inet_cr.h
@@ -1,5 +1,39 @@
 #ifndef _AF_INET_CR_H
 #define _AF_INET_CR_H
+
+#ifdef __KERNEL__
```

```
+
+#include <linux/net.h>
+#include <linux/skbuff.h>
+
+/*
+ * kernel_{get,set}sockopt
+ */
+#define NONE 0
+#define SET  1
+#define GET  2
+#define BOTH (SET|GET)
+
+/*
+ * Structure to associate a socket option with the netlink attribute
+ */
+struct af_inet_cr_optattr {
+ int optname;  /* option name */
+ int attr;     /* netlink attribute */
+ int unconnected; /* option related to unconnected sockets only */
+ int get_and_set; /* has kernel_{set,get}sockopt */
+};
+
+extern int af_inet_cr_opt2attr(struct sk_buff *skb,
+          const struct socket *sock,
+          int optlevel,
+          const struct af_inet_cr_optattr *optattr);
+
+int af_inet_cr_attr2opt(const struct genl_info *info,
+   struct socket *sock,
+   int optlevel,
+   const struct af_inet_cr_optattr *optattr);
+#endif
 enum {
  AF_INET_CR_CMD_UNSPEC,
  AF_INET_CR_CMD_DUMP,
@@ -10,6 +44,33 @@
 enum {
  AF_INET_CR_ATTR_UNSPEC,
  AF_INET_CR_ATTR_INODE,
+
+       /* socket state */
+ AF_INET_CR_ATTR_SOCK_STATE,
+ AF_INET_CR_ATTR_SOCK_TYPE,
+ AF_INET_CR_ATTR_SOCK_FD,
+
+ /* socket options */
+ AF_INET_CR_ATTR_SOCKOPT_BINDTODEVICE,
+
```

```
+ AF_INET_CR_ATTR_SOCKOPT_BROADCAST,
+ AF_INET_CR_ATTR_SOCKOPT_DEBUG,
+ AF_INET_CR_ATTR_SOCKOPT_DONTROUTE,
+ AF_INET_CR_ATTR_SOCKOPT_KEEPALIVE,
+ AF_INET_CR_ATTR_SOCKOPT_LINGER,
+ AF_INET_CR_ATTR_SOCKOPT_OOBINLINE,
+ AF_INET_CR_ATTR_SOCKOPT_PASSCRED,
+ AF_INET_CR_ATTR_SOCKOPT_PRIORITY,
+ AF_INET_CR_ATTR_SOCKOPT_RCVLOWAT,
+ AF_INET_CR_ATTR_SOCKOPT_RCVTIMEO,
+ AF_INET_CR_ATTR_SOCKOPT_SNDTIMEO,
+ AF_INET_CR_ATTR_SOCKOPT_RCVBUF,
+ AF_INET_CR_ATTR_SOCKOPT_REUSEADDR,
+ AF_INET_CR_ATTR_SOCKOPT_SNDBUF,
+ AF_INET_CR_ATTR_SOCKOPT_TIMESTAMP,
+ AF_INET_CR_ATTR_SOCKOPT_SNDBUF_ULOCK,
+ AF_INET_CR_ATTR_SOCKOPT_RCVBUF_ULOCK,
+
  AF_INET_CR_ATTR_MAX
 };
 #endif

--
```