

eVIP Adjunct Helper

Evolved Virtual IP

INTERWORK DESCRIPTION

Copyright

© Ericsson AB 2015–2017. All rights reserved. No part of this document may be reproduced in any form without the written permission of the copyright owner.

Disclaimer

The contents of this document are subject to revision without notice due to continued progress in methodology, design and manufacturing. Ericsson shall have no liability for any error or damage of any kind resulting from the use of this document.

Trademark List

All trademarks mentioned herein are the property of their respective owners. These are shown in the document Trademark Information.



Contents

1	Introduction	1
1.1	Prerequisites	1
2	Adjunct Helper Function	3
3	Adjunct Helper Implementation	5
3.1	Socket Group	6
3.2	Interface Target Side Socket Group	7
3.3	Interface for Extending LBE with Adjunct Helper	8
3.4	Start and Stop of Adjunct Helper	9
3.5	Delete an Adjunct Helper	10
3.6	Upgrade an Adjunct Helper	11
3.7	Configure Distributor Flow Policy	11
3.8	Dynamic View	11
4	Code Example for Implementing an Adjunct Helper Distributor	13





1 Introduction

This document describes the interface between Evolved Virtual IP (eVIP) and an Adjunct Helper. It also describes the prerequisites for building Adjunct Helpers.

1.1 Prerequisites

The user must have general knowledge of how to implement Linux® kernel modules and in particular be familiar with netfilter.

It is recommended that the user has general system-level knowledge of eVIP.





2 Adjunct Helper Function

The Adjunct Helper function is used by eVIP to support specialized load distribution functions not available in the Linux Virtual Server (LVS). The use of an Adjunct Helper requires specific software design by the user and special measures must be taken during configuration.

No Adjunct Helper is required for the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP), as the generic load distribution functions in eVIP support these protocols. Load distribution with eVIP is per protocol case and done in either of the following ways:

- According to the built-in generic distribution policies of a Load Balancer Element (LBE)
- Through an Adjunct Helper according to the method implemented in a specific Adjunct Helper

An example of when the Adjunct Helper function has been used is an Adjunct Helper for the Stream Control Transmission Protocol (SCTP) developed for the Signalling System No.7 (SS7) Common Application Feature (CAF), a software common component product adapted to eVIP.

The method used is general in the eVIP framework and can be used for other protocols than SCTP. Architecturally, an eVIP system can accommodate multiple Adjunct Helpers, each serving a specific protocol.

For traffic to be distributed by an Adjunct Helper, the concept of socket group is used instead of target pools as grouping abstraction for targets. The target pool concept is only used together with the generic distribution policies in eVIP. This document only covers the case of Adjunct Helper assisted distribution.





3 Adjunct Helper Implementation

Adjunct Helper is a concept in the eVIP architecture to provide a way to extend the repertoire of an LBE with extra load distribution functions beyond what is already offered by the built-in generic distribution policies, see Figure 1. The use of an Adjunct Helper requires measures to be taken at design time.

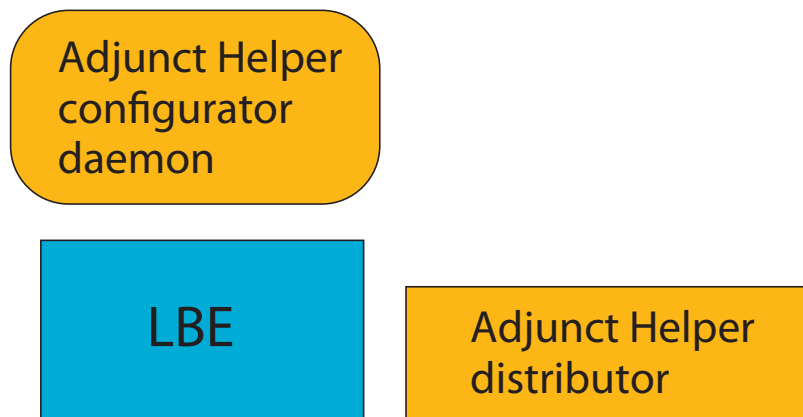


Figure 1 LBE Extended with Adjunct Helper

The main steps for implementing an Adjunct Helper are as follows:

- Software that ties a target to a socket group must be written on the target processor side.
- An Adjunct Helper distributor and an Adjunct Helper configurator daemon must be developed and deployed with each LBE to be functionally extended with an Adjunct Helper.
- A special flow policy must be configured for the Abstract Load Balancer (ALB) where the Adjunct Helper is used. This is done through the normal eVIP configuration setup.

An Adjunct Helper consists of a configurator daemon (user space) and a distributor (kernel space). An LBE with a helper configurator shares Linux network namespace.

An LBE can be extended with multiple Adjunct Helpers, each serving a specific protocol. When an LBE is extended with multiple helpers, the LBE and all its helper configurator daemons share network namespace.

The distributor is implemented as a Linux “xt_netfilter iptables kernel module”. The kernel module must be implemented so that it can handle multiple helper configurator daemons belonging to different network namespaces. The reason is to support typical eVIP configurations where two LBEs belonging to different ALBs are collocated on the same processing unit (processor blade).



Note: LBEs on the same node are running in different network namespaces, but in all other aspects they share the same environment such as Process Identity (PID) and mount namespace. The configurator demons must be designed carefully having this limitation in mind. For example, they cannot write logs using a common name in common directories like `/var/log` as they would end up overwriting each other's logs.

This document covers the eVIP-specific aspects of the distributor and provides sample code for a rudimentary TCP distributor that can be used for demonstration purposes.

3.1 Socket Group

Always when Adjunct Helpers are used, this requires the use of socket groups as distribution target grouping abstraction. The software to receive traffic on the target processors must in advance have been prepared for load distribution to a socket group.

With this method, on each target processing unit at the socket Application Programming Interface (API) level, a special socket option must be used to tie a particular target to a particular socket group, see Figure 2. Each socket group has an identity referred to in API calls.

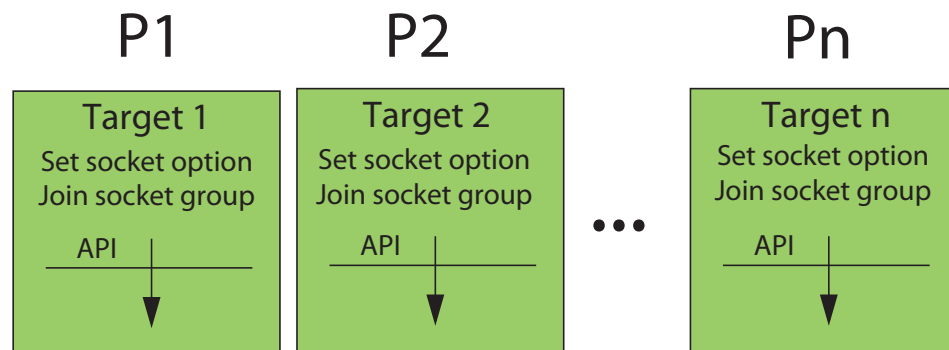


Figure 2 Formation of a Socket Group on a Collection of Blades

Protocol-specific distribution parameter data is, by setting socket data, passed from the software on the target to the Adjunct Helper configurator daemon. The distribution parameter data is passed in verbatim from the target to the Adjunct Helper configurator. The passed parameters in the data are specific to each implemented helper. The parameters are typically used to initialize the Adjunct Helper distributor, further causing the distributor to distribute packet traffic to a socket group on the target processing units. The distribution is done according to an algorithm implemented in the helper distributor.

For selecting the traffic to be handled by an Adjunct Helper distributor, a flow policy indicating the specific helper distributor and socket group by their identities must be configured in the eVIP ALB configuration. This configuration is done at installation time or can be added later as an in-service configuration change. The



flow policy directs the flows of the selected protocol to the helper distributor, which then handles distribution to target processors of the socket group.

3.2 Interface Target Side Socket Group

An interface for passing control data to a socket group in the Adjunct Helper can be accessed from any target blade by the use of some dedicated socket options.

Accessing the interface is done in three steps:

1. Selecting the ALB hosting the socket group
2. Joining a socket group
3. Forwarding control information

Selecting the ALB can be achieved by ALB Selection Policy (refer to Section Selection Policy Configuration in [eVIP Management Guide](#)) or by the following function call, which takes priority over selection policies:

```
#define SO_SELECT_ALB 0x1015
"setsockopt(socket, SOL_SOCKET, SO_SELECT_ALB, name, name_size) "
```

If no ALB with the specified name is known to eVIP, an ENODEV error is returned.

When ALB selection is done by the SO_SELECT_ALB function call, the socket gets “bound” to the selected ALB directly. Otherwise eVIP searches for a feasible ALB selection policy on the first socket group-related socket function call. Either way all subsequent function calls take effect in the selected ALB.

It is possible to change ALB for a socket by a SO_SELECT_ALB function call, but only if the joined socket group has been left first.

Calling SO_SELECT_ALB with an empty string on a socket that has left the joined socket group cancels any previous ALB selection irrespective of the selection method. Thus, the socket can be subject to any kind of ALB selection again.

The join socket group function is called to inform eVIP that a socket is joining a specific non-zero socket group (given in host byte order):

```
#define SO_JOIN_SO_GRP 0x1012
"setsockopt(socket, SOL_SOCKET, SO_JOIN_SO_GRP, &socket_grupp, 4) "
```

Only one socket group can be joined by a socket at a time. To join a different socket group, the socket must first leave the previously joined socket group. When a socket is closed, it automatically leaves the socket group. Leaving a previously joined socket group is also possible by using the following function call:

```
#define SO_LEAVE_SO_GRP 0x1014
"setsockopt(socket, SOL_SOCKET, SO_LEAVE_SO_GRP, &socket_grupp, 4) "
```



A socket that has joined a socket group can set control data for the socket (up to 128 kilobytes = 131,072 bytes) with the following function call:

```
#define SO_TAG_SO_DATA 0x1013  
"setsockopt(socket, SOL_SOCKET, SO_TAG_SO_DATA, data, data_size) "
```

This data can be updated directly by a new function call.

Note: An update here overwrites the complete set of old data. The data set here is passed to the configurator as a data blob. However, the internal blob structure is known to the configurator.

An example of data that can be passed to the configurator can be information to indicate open ports. In a second step, this information can be conveyed to the distributor. This is done in the Adjunct Helper developed for the SS7 CAF to distribute SCTP traffic.

The target side is required not to consider system failure cases programmatically, for example, the possibility of an LBE failure or a helper configurator daemon failure. The eVIP recovery machinery automatically takes care of the failure situations; hence no programmatic action is required on the target or socket side when implementing the socket groups.

3.3 Interface for Extending LBE with Adjunct Helper

The Adjunct Helper configurator configures the Adjunct Helper distributor and both must be implemented for each new Adjunct Helper. The Adjunct Helper configurator is implemented as a Linux daemon while the Adjunct Helper distributor is implemented as Linux “xt netfilter iptables kernel module”. The required specific load distribution logic is implemented in the iptables module.

The user-space daemon links in a software library and registers itself through a function call `LBE_register_protocol_control_handler`, which passes on the following:

- The name of the particular iptables module (given in `iptables_target_name`) to be used
- A set of function pointers to callback functions used to handle events from the target side related to the Adjunct Helper, for example, a socket in progress to join a socket group

The header file interface is as follows:



```
struct PCH_funcs {
    void (*realServerPortAdded)(unsigned int so_grp, unsigned int ⇒
        targetProcessorId, int socket_id, void *udata);
    void (*realServerPortRemoved)(unsigned int so_grp, unsigned int ⇒
        targetProcessorId, int socket_id, void *udata);
    void (*realServerPortData)(unsigned int so_grp, unsigned int ⇒
        targetProcessorId, int socket_id, void *sp_data, ⇒
        int sp_data_size, void *udata);};
#define LBE_REGISTRATION_OK 0
#define LBE_REGISTRATION_FAILED 1
int LBE_register_protocol_control_handler(unsigned int so_grp, char ⇒
    *iptables_target_name, struct PCH_funcs *notifications, void *udata);
```

The following apply:

- The `realServerPortAdded` function is started when a new socket or target joins a socket group.
- The `realServerPortRemoved` function is started when a socket or target is removed.
- The `realServerPortData` function is started when a socket in a socket group updates `SO_TAG_SO_DATA`.
- The `targetProcessorId` parameter corresponds to the `skb -> mark` set in the `iptables` module of the Adjunct Helper distributor. This way packets by policy-based routing in Linux are routed to the indicated target processor in `targetProcessorId`.
- The `socket_id` parameter is per target processor (blade) unique for each socket (on that blade).

When the Adjunct Helper registers itself, the configurator daemon, for each socket in a socket group, receives a callback `realServerPortAddedm(...)`. It also receives a callback `realServerPortData(...)` for each socket in the socket group that has set `SO_TAG_SO_DATA`.

3.4 Start and Stop of Adjunct Helper

The user-space daemon of an Adjunct Helper configurator is always collocated with an LBE and an Adjunct Helper distributor on the same processor blade. The LBE and Adjunct Helper configurator daemon run in the same network namespace.

A script for the Adjunct Helper can be deployed in the `/etc/lbe.d/` directory for the blade. Scripts here begin to execute, activated by the `start` parameter when the LBE starts. When the LBE terminates, the scripts are called with the `stop` parameter. A “modprobe” can be done to load the `iptables` module implementing the Adjunct Helper distributor. This brings in this module with other dependent modules into the kernel before the user-space daemon of the Adjunct Helper configurator is started.

**Note:**

- No explicit programmatic action is necessary to direct traffic to the `iptables` module. This is done automatically when the helper configurator daemon registers itself with the LBE. A configured flow policy in the ALB causes the traffic to be directed to the distributor.
- The responsible eVIP process waits until the successful start and stop of the Adjunct Helper, thus it is recommended to return from the script as fast as possible.

Once the Adjunct Helper is started, eVIP does not restart it in the event of a crash. To ensure that the user-space process is resilient against a crash, it must have some monitoring. eVIP recommends using the monitor service supplied by the Linux Distribution Extensions (LDE).

After the user-space daemon of the Adjunct Helper configurator has started, the daemon can begin to communicate with the (kernel-space) `iptables` module of the Adjunct Helper distributor, for example, by the Linux netlink protocol. Using the netlink protocol requires that the implementation is done so that configurators from multiple network namespaces can communicate with the distributor.

3.5 Delete an Adjunct Helper

As an operation and maintenance activity, an Adjunct Helper function deployed for a specific load balancing service in eVIP can be removed from a cluster harboring eVIP. The removal is done blade-wise, one blade at a time. The following procedure can be automated according to the methods and means provided by the specific cluster middleware harboring eVIP.

To remove an Adjunct Helper from the cluster:

1. Identify the blades where the type of Adjunct Helper (for example, Adjunct Helpers for the SCTP protocol) is deployed. All blades with an Adjunct Helper have corresponding start scripts deployed in directory `/etc/lbe.d/`.
2. For the type of Adjunct Helper selected for removal, remove the start script for the Adjunct Helper from directory `/etc/lbe.d/` on a blade where this Adjunct Helper is deployed.
3. Reboot the blade. This removes the “`xt_netfilter iptables` module”, which implements the Adjunct Helper distributor.
4. Repeat Step 2 and Step 3 for all blades from which the Adjunct Helper is to be removed.

The Adjunct Helpers are now removed from the blades.

5. Remove the configured distributor flow policy pertaining to the Adjunct Helper function removed in this procedure.



3.6 Upgrade an Adjunct Helper

The following procedure can be automated according to the methods and means provided by the specific cluster middleware harboring eVIP.

To upgrade an Adjunct Helper in a cluster:

1. Replace the old start script and software files for the Adjunct Helper configurator and Adjunct Helper distributor (“xt_netfilter iptables module”) with the new versions on all blades to be upgraded.
2. Reboot the blades. The new software version is activated during reboot.

3.7 Configure Distributor Flow Policy

A special flow policy must be configured for the ALB where the Adjunct Helper is used. This is done through the normal eVIP configuration practice, which can differ between a standalone configuration and a Component Based Architecture (CBA) configuration.

For example, in the `xml config` file typically used in the standalone version, the following flow policy directs incoming SCTP traffic from Front-End Elements (FEEs) in an ALB to an Adjunct Helper distributor. In the following example, the corresponding flow policy is named `mySpecialDistribution_1`:

```
<flow_policy name="mySpecialDistribution_1" protocol="sctp" =>  
so_grp="1234" address_family="ipv4" dest="10.0.0.1"/>
```

The destination address (Virtual IP (VIP) address) is here `10.0.0.1` and the targets belong to the socket group called `1234`. The protocol is shown here as `sctp`.

3.8 Dynamic View

3.8.1 Control Plane View

The distributor dynamically obtains its knowledge about targets from data sent from the helper configurator. In an earlier step the helper configurator has acquired this knowledge from the data passed from sockets in a socket group. For example, sockets joining or leaving a socket group or updates of socket data.

3.8.2 Data Plane View

In the forwarding data plane, packets for the protocol to be distributed by an Adjunct Helper must be directed to the helper distributor. This is done by a flow policy. The configured flow policy sends packets, for the selected protocol, to the `iptables` module implementing the Adjunct Helper distributor of an LBE.



The Linux `iptables` netfilter provides a general method for marking IP packets with a so called `skb -> mark`. A mark upon which policy-based routing can be applied with the mark here used as routing direction parameter.

The implemented distributor and `iptables` module can, for each arriving packet, inspect relevant fields and apply its own algorithmic procedures to obtain a target it distributes the packet to. Once this target is known, the distributor program marks the packet in progress by setting a mark, `skb -> mark`, thereafter the function call setting the mark must return `NF_ACCEPT`. The specific mark corresponds to a specific target, `targetProcessorId`, provided as information from the helper configurator daemon.

Further on, the eVIP framework takes over. By policy-based routing in Linux, the packet is routed, based on this set mark, to a path that funnels the packet to the selected target or socket.



4 Code Example for Implementing an Adjunct Helper Distributor

A demonstration code example illustrating how a rudimentary TCP Adjunct Helper distributor can be implemented, for example, for test purposes, is shown in Example 1.

```
/*xt_EVIPTCP.c*/

*   xt_EVIPTCP - Netfilter module to test raw socket distribution in eVIP
*
*   Description:
*   This module is used to test the distributor support in eVIP
*/

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/types.h>
#include <linux/skbuff.h>
#include <net/genetlink.h>
#include <net/sock.h>
#include <net/ip.h>

#include <linux/netfilter/x_tables.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Daniel Tenninge <daniel.andersson.tenninge@ericsson.com>");
MODULE_DESCRIPTION("Xtables: test distributor for eVIP.");
MODULE_ALIAS("ipt_EVIPTCP");

enum {
    TCP_ATTR_UNSPEC,
    TCP_ATTR_FW_MARK,
    TCP_ATTR_ADDED,
    __TCP_ATTR_MAX,
};
#define TCP_ATTR_MAX (__TCP_ATTR_MAX - 1)

static struct nla_policy tcp_genl_policy[TCP_ATTR_MAX + 1] = {
    [TCP_ATTR_FW_MARK] = { .type = NLA_U32 },
    [TCP_ATTR_ADDED] = { .type = NLA_U8 },
};

#define VERSION 1
static struct genl_family tcp_genl_family = {
    .id = GENL_ID_GENERATE,
    .hdrsize = 0,
    .name = "EVIPTCP",
    .version = VERSION,
    .maxattr = TCP_ATTR_MAX,
    .netnsok = 1,
};

enum {
    TCP_CMD_UNSPEC,
    TCP_CMD_FWMARK,
    __TCP_CMD_MAX,
};
#define TCP_CMD_MAX (__TCP_CMD_MAX - 1)

#define MAX_FWMARKS 128

struct ns_fwmarks {
    struct net *net;
    unsigned int fwmarks[MAX_FWMARKS];
    int n_fwmarks;
    int i_rr;
    struct ns_fwmarks *next;
};
```



```
struct ns_fwmarks *ns=0;

static int fwmark_msg(struct sk_buff *skb, struct genl_info *info)
{
    struct net *net=genl_info_net(info);
    struct nlattr *na;
    u32 fw_mark;
    u8 added;
    struct ns_fwmarks *marks;
    int i;

    if (info == NULL) {
        printk(KERN_ERR "info struct is empty\n");
        return 0;
    }

    na = info->attrs[TCP_ATTR_FW_MARK];
    if (na) {
        u32 *fw_data;
        fw_data = (u32 *)nla_data(na);
        if (fw_data) {
            fw_mark = *fw_data;
            printk(KERN_INFO "got fw mark: %u\n", fw_mark);
        } else {
            printk(KERN_ERR "error while receiving fw mark\n");
            return 0;
        }
    } else {
        printk(KERN_ERR "attribute TCP_ATTR_FW_MARK is missing\n");
        return 0;
    }
    na = info->attrs[TCP_ATTR_ADDED];
    if (na) {
        u8 *added_data;
        added_data = (u8 *)nla_data(na);
        if (added_data) {
            added = *added_data;
            printk(KERN_INFO "got added: %hu\n", (u16)added);
        } else {
            printk(KERN_ERR "error while receivinv added\n");
            return 0;
        }
    } else {
        printk(KERN_ERR "attribute TCP_ATTR_ADDED is missing\n");
        return 0;
    }

    for (marks=ns;marks && marks->net!=net;marks=marks->next);

    if (added) {
        /* A new fwmark has been added, add it to the list */
        if (marks->n_fwmarks==MAX_FWMARKS) {
            printk(KERN_INFO "Maximum fwmark reached\n");
            return 0;
        }
        for (i=0;i<marks->n_fwmarks;i++) {
            if (marks->fwmarks[i]==fw_mark) {
                printk(KERN_INFO "Mark already added\n");
                return 0;
            }
        }
        marks->fwmarks[marks->n_fwmarks]=fw_mark;
        marks->n_fwmarks++;
    } else {
        /* A fwmark has been removed, remove it from the list */
        for (i=0;i<marks->n_fwmarks;i++) {
            if (marks->fwmarks[i]==fw_mark) {
                marks->fwmarks[i]=marks->fwmarks[marks->n_fwmarks-1];
                marks->n_fwmarks--;
                return 0;
            }
        }
        printk(KERN_INFO "Mark not found\n");
    }
}
```



```

    return 0;
}

struct genl_ops tcp_genl_ops_fwmark = {
    .cmd      = TCP_CMD_FWMARK,
    .flags    = 0,
    .policy   = tcp_genl_policy,
    .doit     = fwmark_msg,
    .dumpit   = NULL,
};

static unsigned int tcp_tg(struct sk_buff *skb, const struct xt_target_param *par)
{
    struct net *net=skb->dev->nd_net;
    struct ns_fwmarks *marks;

    for (marks=ns;marks->net!=net;marks=marks->next);

    if (marks->n_fwmarks==0) {
        printk(KERN_INFO "sctp: got packet but no fwmark is added\n");
        return XT_CONTINUE;
    }

    marks->i_rr++;
    if (marks->i_rr>=marks->n_fwmarks) {
        marks->i_rr=0;
    }

    skb->mark = marks->fwmarks[marks->i_rr];
    return NF_ACCEPT;
}

static struct xt_target tcp_tg_reg __read_mostly = {
    .name      = "EVIPTCP",
    .revision  = 0,
    .family    = NFPROTO_IPV4,
    .target    = tcp_tg,
    .targetsize = 0,
    .me        = THIS_MODULE
};

static int init_ns(struct net *net)
{
    struct ns_fwmarks *marks=kmalloc(sizeof(struct ns_fwmarks),GFP_KERNEL);

    marks->n_fwmarks=0;
    marks->i_rr=0;
    marks->net=net;
    marks->next=ns;
    ns=marks;

    printk(KERN_INFO "Created NS\n");
    return 0;
}

static void exit_ns(struct net *net)
{
    struct ns_fwmarks *marks,*last;

    for (marks=ns,last=0;marks && marks->net!=net;last=marks,marks=marks->next);

    if (last) {
        last->next=marks->next;
    } else {
        ns=marks->next;
    }

    kfree(marks);

    printk(KERN_INFO "Removed NS\n");
}

static struct pernet_operations netns_ops = {
    .init=init_ns,

```



```
.exit=exit_ns
};

static int __init tcp_mt_init(void)
{
    int ret;

    /* Register with netlink */
    printk(KERN_INFO "tcp: registering generic netlink family\n");
    ret = genl_register_family(&tcp_genl_family);
    if (ret != 0) {
        printk(KERN_ERR "Could not register generic netlink family. (%d)\n", ret);
        return ret;
    }
    printk(KERN_INFO "tcp: registering ops\n");
    ret = genl_register_ops(&tcp_genl_family, &tcp_genl_opts_fwmark);
    if (ret != 0) {
        printk(KERN_ERR "Could not register ops. (%d)\n", ret);
        genl_unregister_family(&tcp_genl_family);
        return ret;
    }

    /* Register with netfilter */
    printk(KERN_INFO "tcp: registering xtables target\n");
    ret = xt_register_target(&tcp_tg_reg);
    if (ret < 0) {
        printk(KERN_ERR "tcp: could not register xtables. (%d)\n", ret);
        return ret;
    }

    /* Register network namespace hooks*/
    if ((ret=register_pernet_subsys(&netns_ops))) {
        printk(KERN_ERR "Could not register netns hooks\n");
        return ret;
    }

    return 0;
}

static void __exit tcp_mt_exit(void)
{
    genl_unregister_ops(&tcp_genl_family, &tcp_genl_opts_fwmark);
    genl_unregister_family(&tcp_genl_family);
    xt_unregister_target(&tcp_tg_reg);
    unregister_pernet_subsys(&netns_ops);
}

module_init(tcp_mt_init);
module_exit(tcp_mt_exit);
```

Example 1 Implementation of Rudimentary TCP Adjunct Helper Distributor