Proceedings of the
Linux Symposium

Volume One

July 20nd–23th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Gerrit Huizenga, *IBM*
Matthew Wilcox, *HP*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Matt Domsch, *Dell*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

DCCP on Linux

Arnaldo Carvalho de Melo
Conectiva S.A.
acme@{conectiva.com.br,mandriva.com,ghostprotocols.net}

Abstract

In this paper I will present the current state of DCCP for Linux, looking at several implementations done for Linux and for other kernels, how well they interoperate, how the implementation I'm working on took advantage of the work presented in my OLS 2004 talk ("TCP-fying the poor cousins") and ideas about plugabble congestion control algorithms in DCCP, taking advantage of recent work by Stephen Hemminger on having a IO scheduller like infrastructure for congestion control algorithms in TCP.

1 What is DCCP?

The Datagram Congestion Control Protocol is a new Internet transport protocol to provide unreliable, congestion controlled connections, providing a blend of characteristics not available in other existing protocols such as TCP, UDP, or SCTP.

There has been concern that the increasing use of UDP in application such as VoIP, streaming multimedia and massively online games can cause congestion collapse on the Internet, so DCCP is being designed to provide an alternative that frees the applications from the complexities of doing congestion avoidance, while providing a core protocol that can be extended with new congestion algorithms, called CCIDs, that can be negotiated at any given time in a connection lifetime, even with different algorithms being used for each direction of the connection, called Half Connections in DCCP drafts.

This extensibility is important as there are different sets of requirements on how the congestion avoidance should be done, while some applications may want to grab as much bandwidth as possible and accept sudden drops when congestion happens others may want not to be so greedy but have fewer oscillations in the average bandwidth used through the connection lifetime.

Currently there are two profiles defined for DCCP CCIDs: CCID2, TCP-Like Congestion Control[3], for those applications that want to use as much as possible bandwidth and are able to adapt to sudden changes in the available bandwidth like those that happens in TCP's Additive Increase Multiplicative Decrease (AIMD) congestion control; and CCID3, TCP Friendly Congestion Control (TFRC)[4], that implements a receiver-based congestion control algorithm where the sender is rate-limited by packets sent by the receiver with information such as receive rate, loss intervals, and the time packets were kept in queues before being acknowledged, indented for applications that want a smooth rate.

There are a number of RFC drafts covering aspects of DCCP to which interested people should refer for detailed information about the many aspects of this new protocol, such as:

- *Problem Statement for DCCP[1]*
- *Datagram Congestion Control Protocol (DCCP)[2]*
- *Profile for DCCP Congestion Control ID 2[3]*
- *Profile for DCCP Congestion Control ID 3[4]*
- *Datagram Congestion Control Protocol (DCCP) User Guide[5]*
- *DCCP CCID 3-Thin[6]*
- *Datagram Congestion Control Protocol Mobility and Multihoming[7]*
- *TCP Friendly Rate Control (TFRC) for Voice: VoIP Variant and Faster Restart[8]*

This paper will concentrate on the the current state of the author's implementation of DCCP and its CCIDs in the Linux Kernel, without going too much into the merits of DCCP as a protocol or its adequacy to any application scenario.

2 Implementations

DCCP has been a moving target, already in its 11th revision, with new drafts changing protocol aspects that have to be tracked by the implementators, so while there has been several implementations written for Linux and the BSDs, they are feature incomplete or not compliant with latest drafts.

Patrick McManus wrote an implementation for the Linux Kernel version 2.4.18, Implementing only CCID2, TCP-Like Congestion Control, but has not updated it to the latest specs and also has bitrotted, as the Linux kernel networking core has changed in many aspects in 2.6.

Another implementation was made for FreeBSD at the Luleå University of Technology, Sweden, that is more complete, implementing even the TFRC CCID. This implementation has since been merged in the KAME Project codebase, modified with lots of ifdefs to provide a single DCCP code base for FreeBSD, NetBSD, and OpenBSD.

The WAND research group at the University of Waikato, New Zealand also has been working on a DCCP implementation for the Linux kernel, based on the stack written by Patrick McManus, combining it with the the Luleå FreeBSD CCID3 implementation.

The DCCP home page at ICIR also mentions a user-level implementation written at the Berkeley University, but the author was unable to find further details about it.

The implementation the author is writing for the Linux Kernel is not based on any of the DCCP core stack implementations mentioned, for reasons outlined in the "DCCP on Linux" section later in this paper.

3 Writing a New Protocol for the Linux Kernel

Historically when new protocols are being written for the Linux kernel existing protocols are used as reference, with code being copied to accelerate the implementation process.

While this is a natural way of writing new code it introduces several problems when the reference protocols and the core networking infrastructure is changed, these problems were discussed in my “TCPfying the poor Cousins” [10] paper presented in 2004 at the Linux Symposium, Ottawa.

This paper will describe the design principles and the refactorings done to the Linux kernel networking infrastructure to reuse existing code in the author’s DCCP stack implementation to minimise these pitfalls.

4 DCCP on Linux

The next sections will talk about the design principles used in this DCCP implementation, using the main data structures and functions as a guide, with comments about its current state, how features were implemented, sometimes how missing features or potential DCCP APIs that are being discussed in the DCCP community could be implemented and future plans.

5 Design Principles

1. Make it look as much as possible as TCP, same function names, same flow.
2. Generalise as much as possible TCP stuff.
3. Follow as close as possible the pseudocode in the DCCP draft[2], as long as it doesn’t conflicts with principle 1.
4. Any refactoring to existing code (TCP, etc.) has to produce code that is as fast as the previous situation—if possible faster as was the case with TCP’s `open_request` generalization, becoming `struct request_sock`. Now TCP v4 syn minisocks use just 64 bytes, down from 96 in stock Linus tree; lmbench shows performance improvements.

Following these principles the author hopes that the Linux TCP hackers will find it easy to review this stack, and if somebody thinks that all these generalisations are dangerous for TCP, so be it, its just a matter of reverting the TCP patches and leaving the infrastructure to be used only by DCCP and in time go on slowly making TCP use it.

6 Linux Infrastructure for Internet Transport Protocols

It is important to understand how the Linux kernel internet networking infrastructure supports transport protocols to provide perspective on the refactorings done to better support a DCCP implementation.

A `AF_INET` transport protocol uses the `inet_add_protocol` function so that the IP layer can feed it packets with its protocol identifier as present in the IP header, this function receives the protocol identifier and a `struct net_protocol` where there has to be a pointer for a function to handle packets for this specific transport protocol.

The transport protocol also has to use the `inet_register_protosw` function to tell the inet layer how to create new sockets for this specific transport protocol, passing a `struct inet_protosw` pointer as the only argument, DCCP passes this:

```
struct inet_protosw
dccp_v4_protosw = {
.type    = SOCK_DCCP,
.protocol= IPPROTO_DCCP,
.prot    = &dccp_v4_prot,
.ops     = &inet_dccp_ops,
};
```

So when applications use `socket(AF_INET, SOCK_DCCP, IPPROTO_DCCP)` the inet infrastructure will find this struct and set socket->ops to inet_dccp_ops and sk->sk_prot to dccp_v4_prot.

The socket->ops pointer is used by the network infrastructure to go from a syscall to the right network family associated with a socket, DCCP sockets will be reached through this struct:

```
struct proto_ops inet_dccp_ops = {
.family     = PF_INET,
.owner      = THIS_MODULE,
.release    = inet_release,
.bind       = inet_bind,
.connect    = inet_stream_connect,
.socketpair= sock_no_socketpair,
.accept     = inet_accept,
.getname    = inet_getname,
.poll       = sock_no_poll,
.ioctl      = inet_ioctl,
.listen     = inet_dccp_listen,
.shutdown   = inet_shutdown,
.setsockopt=
sock_common_setsockopt,
.getsockopt=
sock_common_getsockopt,
.sendmsg    = inet_sendmsg,
.recvmsg    = sock_common_recvmsg,
.mmap       = sock_no_mmap,
.sendpage   = sock_no_sendpage,
};
```

Looking at this struct we can see that the DCCP code shares most of the operations with the other AF_INET transport protocols, only implementing the .listen method in a different fashion, and even this method is to be shared, as the only difference it has with inet_listen, the method used for TCP is that it checks if the socket type is SOCK_DGRAM, while inet_listen checks if its SOCK_STREAM.

Another point that shows that this stack is still in development is that at the moment it doesn't supports some ot the struct proto_ops methods, using stub routines that return appropriate error codes.

One of these methods, .mmap, is implemented in the Waikato University DCCP stack to provide transmission rate information when using the TFRC DCCP CCID, and can be used as well to implement an alternative sending API that uses packet rings in an mmaped buffer as described the paper "A Congestion-Controlled Unreliable Datagram API" by Junwen Lai and Eddie Kohler[12].

To go from the common struct proto_ops AF_INET methods to the DCCP stack the sk->sk_prot pointer is used, and in DCCP case it is set to this struct:

```
struct proto dccp_v4_prot = {
.name         = "DCCP",
.owner        = THIS_MODULE,
.close        = dccp_close,
.connect      = dccp_v4_connect,
.disconnect   = dccp_disconnect,
.ioctl        = dccp_ioctl,
.init         = dccp_v4_init_sock,
.setsockopt   = dccp_setsockopt,
.getsockopt   = dccp_getsockopt,
.sendmsg      = dccp_sendmsg,
.recvmsg      = dccp_recvmsg,
.backlog_rcv  = dccp_v4_do_rcv,
.hash         = dccp_v4_hash,
.unhash       = dccp_v4_unhash,
.accept       = inet_csk_accept,
.get_port     = dccp_v4_get_port,
.shutdown     = dccp_shutdown,
.destroy      =
dccp_v4_destroy_sock,
.max_header   = MAX_DCCP_HEADER,
.obj_size     = sizeof(struct
dccp_sock),
.rsk_prot     =
&dccp_request_sock_ops,
.orphan_count= &dccp_orphan_count,
};
```

Two of these methods bring us to a refactoring done to share code with TCP, denounced by the `.accept` method, `inet_csk_accept`, that previously was named `tcp_accept`, and as will be described in the next section could be made generic because most of the TCP infrastructure to handle SYN packets was generalised so as to be used by DCCP and other protocols.

7 Handling Connection Requests

DCCP connection requests are done sending a packet with a specific type, and this shows an important difference with TCP, namely that DCCP has an specific field in its packet header to indicate the type of the packet, whereas TCP has a flags field where one can use differenc combinations to indicate actions such as the beginning of the 3way handshake to create a connection, when a SYN packet is sent, while in DCCP a packet with type REQUEST is sent.

Aside from this difference the code to process a SYN packet in TCP fits most of the needs of DCCP to process a REQUEST packet: to create a mini socket, a structure to represent a socket in its embryonic form, avoiding using too much resources at this stage in the socket lifetime, and also to deal with timeouts waiting for TCP's SYN+ACK or DCCP's RESPONSE packet, synfloods (requestfloods in DCCP).

So the `struct open_request` TCP specific data structure was renamed to `struct request_sock`, with the members that are specfic to TCP and TCPv6 were removed, effectively creating a class hierarchy similar to the `struct sock` one, with each protocol using this structure creating a derived struct that has a `struct request_sock` as its first member, so that the functions that aren't protocol specific could be moved to the networking core, becoming a new core API usable by other protocols, not even necessarily an `AF_INET` protocol.

Relevant parts of `struct request_sock`:

```
struct request_sock {
struct request_sock *dl_next;
u8                  retrans;
u32                 rcv_wnd;
unsigned long       expires;
struct request_sock_ops *rsk_ops;
struct sock         *sk;
};
```

The `struct request_sock_ops` data structure is not really a new thing, it already exists in the stock kernel sources, within the TCP code, named as `struct or_calltable`, introduced when the support for IPv6 was merged. At that time the approach to make this code shared among TCPv6 and TCPv4 was to add an union to `struct open_request`, leaving this struct with this layout (some fields suppressed):

```
/* this structure is too big */
struct open_request {
struct open_request *dl_next;
u8                  retrans;
u32                 rcv_wnd;
unsigned long       expires;
struct or_calltable *class;
struct sock         *sk;
union {
  struct tcp_v4_open_req v4_req;
#if defined(CONFIG_IPV6) || defined
(CONFIG_IPV6_MODULE)
  struct tcp_v6_open_req v6_req;
#endif
  } af;
};
```

So there is no extra indirection added by this refactoring, and now the state that TCPv4 uses to represent syn sockets was reduced significantly as the TCPv6 state is not included, being moved to struct tcp6_request_sock, that is derived in an OOP fashion from struct tcp_request_sock, that has this layout:

```
struct tcp_request_sock {
struct inet_request_sock req;
u32                      rcv_isn;
u32                      snt_isn;
};
```

That is, derived from another new data structure, struct inet_request_sock, that has this layout:

```
struct inet_request_sock {
 struct request_sock req;
 u32                 loc_addr;
 u32                 rmt_addr;
 u16                 rmt_port;
 u16                 snd_wscale:4,
                     rcv_wscale:4,
                     tstamp_ok:1,
                     sack_ok:1,
                     wscale_ok:1,
                     ecn_ok:1;
struct ip_options    *opt;
};
```

Which bring us back to DCCP, where sockets in the first part of the 3way handshake, the ones created when a DCCP REQUEST packet is received, are represented by this structure:

```
struct dccp_request_sock {
struct inet_request_sock
dreq_inet_rsk;
u64                      dreq_iss;
u64                      dreq_isr;
};
```

This way TCP's struct open_request becomes a class hierarchy, with the common part (struct request_sock) becoming available for use by any connection oriented protocol, much in the same way struct sock is common to all Linux network protocols.

References

[1] Sally Floyd, Mark Handley, and Eddie Kohler, 2002 “Problem Statement for DCCP” draft-ietf-dccp-problem-00.txt

[2] Eddie Kohler, Mark Handley and Sally Floyd, 2005 “Datagram Congestion Control Protocol (DCCP)” draft-ietf-dccp-spec-11.txt

[3] Sally Floyd, Eddie Kohler, 2005 “Profile for DCCP Congestion Control ID 2: TCP-like Congestion Control” draft-ietf-dccp-ccid2-10.txt

[4] Sally Floyd, Eddie Kohler and Jitendra Padhye, 2005 “Profile for DCCP Congestion Control ID 3: TFRC Congestion Control” draft-ietf-dccp-ccid3-11.txt

[5] Tom Phelan, 2005 “Datagram Congestion Control Protocol (DCCP) User Guide” draft-ietf-dccp-user-guide-04.txt

[6] Eddie Kohler, 2004 “DCCP CCID 3-Thin” draft-ietf-dccp-ccid3-thin-01.txt

[7] Eddie Kohler, 2004 “Datagram Congestion Control Protocol Mobility and Multihoming” draft-kohler-dccp-mobility-00.txt

[8] Sally Floyd, Eddie Kohler, 2005 “TCP Friendly Rate Control (TFRC) for Voice: VoIP Variant and Faster Restart” draft-ietf-dccp-tfrc-voip-01.txt

[10] Arnaldo Carvalho de Melo, 2004 “TCPfying the Poor Cousins” Ottawa Linux Symposium, 2004

[11] Patrick McManus DCCP implementation for Linux 2.4.18

[12] Junwen Lai and Eddie Kohler, "A Congestion-Controlled Unreliable Datagram API" `http://www.icir.org/kohler/dcp/nsdiabstract.pdf`