WIKIPEDIA

Direct Client-to-Client

Direct Client-to-Client (**DCC**) (originally **Direct Client Connection**[1][2][3]) is an IRC-related sub-protocol enabling peers to interconnect using an IRC server for handshaking in order to exchange files or perform non-relayed chats. Once established, a typical DCC session runs independently from the IRC server. Originally designed to be used with ircII it is now supported by many IRC clients. Some peer-to-peer clients on napster-protocol servers also have DCC send/get capability, including TekNap, SunshineUN and Lopster. A variation of the DCC protocol called SDCC (Secure Direct Client-to-Client), also known as DCC SCHAT supports encrypted connections. An RFC specification on the use of DCC does not exist.

DCC connections can be initiated in two different ways:

- The most common way is to use CTCP to initiate a DCC session. The CTCP is sent from one user, over the IRC network, to another user.
- Another way to initiate a DCC session is for the client to connect directly to the DCC server. Using this method, no traffic will go across the IRC network (the parties involved do not need to be connected to an IRC network in order to initiate the DCC connection).

Contents

History

ircII was the first IRC client to implement the CTCP and DCC protocols.[4] The CTCP protocol was

implemented by Michael Sandrof in 1990 for ircII version 2.1.[5] The DCC protocol was implemented by Troy Rollo in 1991 for version 2.1.2,[6] but was never intended to be portable to other IRC clients.[7][8]

Common DCC applications

DCC CHAT

The CHAT service enables users to chat with each other over a DCC connection. The traffic will go directly between the users, and not over the IRC network. When compared to sending messages normally, this reduces IRC network load, allows sending of larger amounts of text at once, due to the lack of flood control, and makes the communication more secure by not exposing the message to the IRC servers (however, the message is still in plaintext).

DCC CHAT is normally initiated using a CTCP handshake. The user wishing to establish the connection sends the following CTCP to the target, `DCC CHAT` *`protocol ip port`*, where *ip* and *port* are the IP address and port number of the sender, and are expressed as integers. *protocol* is `chat` for standard DCC CHAT. The receiving party can then connect to the given port and IP address.

Once a connection is established, the protocol used for DCC CHAT is very simple: users exchange CRLF-terminated messages. Messages that begin with an ASCII 001 (control-A, represented below by `[^A]`) and the word `ACTION`, and are terminated by another ASCII `001`, are interpreted as emotes: `[^A]ACTION waves goodbye[^A]`.

DCC Whiteboard

This is an extension to DCC CHAT, allowing simple drawing commands to be sent as well as lines of text. DCC Whiteboard is initiated with a handshake similar to DCC CHAT, with the protocol `chat` replaced by `wboard`: `DCC CHAT wboard` *`ip port`*.

Once the connection is established, the two clients exchange CRLF-terminated messages. Messages that begin (and optionally end) with ASCII `001` are interpreted as special commands; the command `ACTION` represents an emote, while others cause lines to be drawn on the user's whiteboard surface, or allow the two clients to negotiate a set of features.

DCC SEND

The SEND service allows users to send files to one another. The original specification for the handshake did not allow the receiver to know the total file size nor to resume a transfer. This has made clients introduce their own extensions to the handshake, many of which have become widely supported.

The original handshake consisted of the sender sending the following CTCP to the receiver: `DCC SEND` *`filename ip port`*.

As with DCC CHAT, *ip* and *port* are the IP address and port where the sending machine will be

listening for an incoming connection. Some clients enclose filenames with spaces in double quotes. It is common practice to add the file size as a last argument: `DCC SEND` *`filename ip port file size`*.

At this point, the original specification had the receiver either connect to the given address and port and wait for data, or ignore the request, but for clients supporting the DCC RESUME extension, a third alternative is to ask the sender to skip part of the file by sending the CTCP reply: `DCC RESUME` *`filename port position`*.

If the sending client supports DCC RESUME, it will reply with, `DCC ACCEPT` *`filename port position`*, and the receiver can connect to the given address and port and listen for data to append to an already existing file.

Data is sent to the client in blocks, each of which the client must acknowledge by sending the total number of bytes received in the form of a 32-bit network byte order integer. This slows down connections and is redundant because of TCP. The send-ahead extension relieves this problem somewhat by not waiting for the acknowledgements, but since the receiver still has to send them for every block it receives, in case the sender expects them, it is not solved completely.

Another extension, TDCC, or turbo DCC, removes the acknowledgements, but requires a slightly modified handshake and is not widely supported. Older versions of TDCC replaced the word SEND in the handshake with TSEND; later versions use the word SEND but append a `T` after the handshake, making this version of TSEND compatible with other clients (as long as they can parse the modified handshake).

DCC SEND exploit

The DCC send exploit can refer to two bugs, a variant buffer overflow error in mIRC triggered by filenames longer than 14 characters[9] and an input validation error in some routers manufactured by Netgear, D-Link and Linksys, triggered by the use of port `0`.[10][11] The router exploit, in particular, may be triggered when the phrase '`DCC SEND`' followed by at least 6 characters without spaces or newlines appears anywhere in a TCP stream on port 6667, not just when an actual DCC SEND request has been made.

DCC XMIT

The XMIT service is a modified version of DCC SEND that allows for resuming files and cuts down on wasteful traffic from the ACK longs. XMIT is not widely supported.

The XMIT handshake differs somewhat from the SEND handshake. The sender sends a CTCP offering a file to the receiver: `DCC XMIT` *`protocol ip port`*`[` *`name`*`[` *`size`*`[` *`MIME-type`*`]]]`

Square brackets here enclose optional parts. *protocol* is the protocol to use for the transfer; only `clear` is defined presently. Unlike standard DCC SEND, *ip* can be in the additional forms of standard dotted notation for IPv4, or either hexadecimal or mixed notation for IPv6. To leave an early parameter empty, but still supply a later one, the earlier one can be specified as `-`. If the receiver does not implement the protocol used, it will send back a CTCP reply of the format: `ERRMSG DCC CHAT` *`protocol`* `unavailable`.

CHAT is used here to maintain compatibility with the error messages sent by the extended DCC CHAT. If the receiver declines the transfer, it sends the following CTCP reply: `ERRMSG DCC CHAT` *`protocol`* `declined`.

Other errors are reported in the same fashion. If the receiver is willing and capable of receiving the file, it will connect to the given address and port. What happens then depends on the protocol used.

In the case of the `clear` protocol, the XMIT server will, upon receiving a connection, send a 32-bit `time_t` in network byte order, representing the file's modification time. Presumably based on the modification time of the local file, the client will then send another network byte order `long`, an offset which the server should seek to when sending the file. This should be set to zero if the whole file is wanted, or the size of the local file if the client wishes to resume a previous download.

While faster than SEND, XMIT carries one of the same limitations in that it is impossible to tell how big the file is, unless its size is specified in the CTCP negotiation or known beforehand. Furthermore, it is not possible to resume a file past the two gigabyte mark due to the 32-bit offset.

Passive DCC

In a normal DCC connection the initiator acts as the server, and the target is the client. Because of widespread firewalling and reduction of end-to-end transparency because of NAT, the initiator might not be able to act as a server. Various ways of asking the target to act as the server have been devised:

DCC Server

This extension to normal DCC SEND and CHAT was introduced by the IRC client mIRC. DCC Server has moderate support, but is not standard on all clients (see Comparison of Internet Relay Chat clients).

It allows the initiation of a DCC connection by IP address, without the need of an IRC server. This is accomplished by the receiving client acting as a server (hence the name) listening (usually on port 59) for a handshake from the sender.

For a CHAT, the initiator sends `1000` *`initiator nick`*. The target then replies with, `1000` *`target nick`*, and the rest proceeds according to standard DCC CHAT protocol.

For a SEND, the initiator sends `1200` *`initiator nick filesize filename`*. The target replies with, `1210` *`target nick resume position`*, where *resume position* is the offset in the file from which to start. From here the transfer proceeds as a normal DCC SEND.

DCC Server also supports mIRC-style file servers and DCC GET.

RDCC

DCC Server provides no way specifying the port to use, so this has to be negotiated manually, which is not always possible, as one of the sides may not be a human. RDCC is a handshake mechanism for DCC Server, which in addition to the port also provides the IP address of the server,

which the client might not be able to find otherwise because of host masking. It is not widely supported.

The initiator requests the port the target is listening on by sending the CTCP query, `RDCC function comment`, where *function* is `c` for chat, `s` for send, or `f` for file server.

The target may then CTCP reply with, `RDCC 0 ip port`, where *ip* and *port* have the same meanings as for normal DCC SEND and CHAT. After this the initiator connects to the *ip* and *port*, and a DCC Server handshake follows.

DCC REVERSE

Unlike DCC Server, where the handshake is handled over a direct IP connection, DCC REVERSE has a normal CTCP handshake, similar to the one used by DCC SEND. This is not widely implemented. The sender offers a file to the receiver by sending the CTCP message: `DCC REVERSE filename filesize key`. *key* is a 1–50 characters-long string of ASCII characters in the range 33–126, and acts as an identifier for the transfer.

If the receiver accepts, it sends the CTCP reply, `DCC REVERSE key start ip port`

Here, *start* is the position in the file from which to start sending, *ip* is the IP address of the receiver in standard dotted notation for IPv4, or hexadecimal notation for IPv6. The sender then connects to the ip address and port indicated by the receiver, and a normal DCC SEND follows. Both the sender and receiver can cancel the handshake by sending the CTCP reply, `DCC REJECT REVERSE key`.

DCC RSEND

This is the KVIrc client's alternative to DCC REVERSE. The sender offers a file by sending the CTCP: `DCC RSEND filename filesize`. The receiver can then accept by CTCP replying with, `DCC RECV filename ip port start`, and the sender connects to the receiver and sends as during a normal DCC SEND.

Reverse / Firewall DCC

This passive DCC mechanism is supported by at least mIRC, Visual IRC, HexChat, KVIrc, DMDirc, Klient, Konversation, and PhibianIRC. The sender offers a file by sending the CTCP message, `DCC SEND filename ip 0 filesize token`. *ip* is the IP address of the sender in network byte order, expressed as a single integer (as in standard DCC). The number `0` is sent instead of a valid port, signaling that this is a Reverse DCC request. *token* is a unique integer; if TSEND is being used (by a client that supports it), the letter `T` is appended to the token, letting the receiver know it doesn't need to send acknowledgements.

The receiver can accept the file by opening a listening socket and responding with the CTCP message, `DCC SEND filename ip port filesize token`. This is identical to the original Reverse DCC message, except the *ip* and *port* identify the socket where the receiver is listening. *token* is the same as in the original request, letting the sender know which request is being accepted. (Because this message follows the same format as a regular DCC send request, some servers which filter DCC requests may require the sender to add the receiver to their "DCC allow"

list.)

The sender then connects to the receiver's socket, sends the content of the file, and waits for the receiver to close the socket when the file is finished.

When the RESUME extension to the SEND protocol is used, the sequence of commands becomes (with >> indicating an outgoing message on the initiating side, and << response by its peer):

```
>> DCC SEND filename ip 0 filesize token
<< DCC RESUME filename 0 position token
>> DCC ACCEPT filename 0 position token
<< DCC SEND filename peer-ip port filesize token
```

After which the protocol proceeds as normal (i.e. the sender connects to the receiver's socket).

File servers (FSERVs)

A DCC *fserve*, or file server, lets a user browse, read and download files located on a DCC server.

Typically, this is implemented with a DCC CHAT session (which presents the user with a command prompt) or special CTCP commands to request a file. The files are sent over DCC SEND or DCC XMIT. There are many implementations of DCC file servers, among them is the FSERV command in the popular mIRC client.

See also

- CTCP (Client-to-client protocol)
- XDCC (eXtended DCC)

References

1. https://www.troy.rollo.name/opensource.html
2. "DCC negotiation and connection" (http://www.kvirc.net/doc/doc_dcc_connection.html).
3. http://www.irchelp.org/protocol/ctcpspec.html
4. Piccard, Paul; Brian Baskin; George Spillman; Marcus Sachs (May 1, 2005). "IRC Networks and Security". *Securing IM and P2P Applications for the Enterprise* (1st ed.). Syngress. p. 386. ISBN 1-59749-017-2. "The authors of the ircII software package originally pioneered file transfers over IRC."
5. See the 'NOTES' and 'source/ctcp.c' files included with ircii-2.1.4e.tar.gz (http://download.ircii.org/historical/unsorted/clients/unix/ircii/ircii-2.1.4e.tar.gz)
6. See the 'UPDATES' and 'source/dcc.c' files included with ircii-2.1.4e.tar.gz (http://download.ircii.org/historical/unsorted/clients/unix/ircii/ircii-2.1.4e.tar.gz)
7. Troy Rollo (January 20, 1993). "/dcc" (https://groups.google.com/group/alt.irc/msg/fc5de46e023d5ffe). Newsgroup: alt.irc (news:alt.irc). Usenet: 1993Jan20.222051.1484@usage.csd.unsw.OZ.AU (news:1993Jan20.222051.1484@usage.csd.unsw.OZ.AU). Retrieved November 10, 2010.

8. Rollo, Troy. "A description of the DCC protocol" (http://www.irchelp.org/irchelp/rfc/dccspec.html) . irchelp.org. Retrieved November 10, 2010. "The first comment I should make is that the DCC protocol was never designed to be portable to clients other than IRCII. As such I take no responsibility for it being difficult to implement for other clients."
9. "SecurityFocus exploit information" (http://www.securityfocus.com/bid/8880).
10. " 'DCC Send' vulnerability on Netgear routers" (http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-1068).
11. " 'DCC Send' vulnerability on Linksys routers" (http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-1067).

External links

- A description of the DCC protocol (http://www.irchelp.org/irchelp/rfc/dccspec.html) *(note: Most IRC-clients and networks have implemented extensions to the DCC protocol. The DCC commonly used today has evolved quite a bit from what this document describes.)*
- DCC negotiation and connection (http://www.kvirc.de/docu/doc_dcc_connection.html)
- A description of the Turbo DCC protocol (http://www.visualirc.net/tech-tdcc.php)
- A description of the DCC Whiteboard protocol (http://www.visualirc.net/tech-wboard.php)