NMDC Protocol

Fredrik Ullner
<ullner@gmail.com>
version 1.3, March 2013

Table of Contents

1. Abstract

Neo-Modus Direct Connect (NMDC) is a text protocol for a client-server network. The same protocol structure is used both for client-hub and client-client communication. This document is split into two parts; the first shows the structure of the protocol, while the second implements a specific system using this structure.

NMDC was written by Jon Hess, in the implementation of the (proprietary) NMDC client and hub software, but was never documented. The protocol was reverse-engineered later on by others, and soon followed open implementations. Extensions have followed over the years and may be incompatible with the original NMDC software.

2. Authors

The protocol's basic outline was written by Jon Hess in 1998 for the NMDC client. Major influencers in the development that followed of NMDC include (in no particular order): Jacek Sieka, Sid, aDe, David Marwood, Suxxx, Stefan Gorling, Sphinx, HaArd and others.

This document uses information from many sources, including but not limited to;

- Ptokax Wiki http://wiki.ptokax.ch and http://wiki.ptokax.org
- TeamFair wiki http://www.teamfair.info/wiki/
- The Nighthawk forum
- Mutor protocol doc http://mutor.no-ip.com:413/protocol.htm

This document was compiled by Fredrik Ullner.

3. Version history

The latest draft of the next version of this document as well as intermediate and older versions can be downloaded from $URL: https://nmdc.svn.sourceforge.net/svnroot/nmdc/trunk/NMDC.txt $. This version corresponds to $Revision: 25 $.

3.1. Version 1.3

Fredrik Ullner <ullner@gmail>, 2013-03-09

- Changed note about escape sequences
- Added IPv4 and IPv6 extensions
- More clarification on $MyInfo's flag
- Moved some commands to/from extensions
- Added note about XmlBzList and DC++'s dropped support
- Added TLS and DHT extensions
- Clarified (corrected) $UserIP and UserIP2
- Added $Cancel and noted that $Canceled was only used briefly
- Added QP extension
- Added $Failover extension
- Added security considerations to the protocol
- Added URI scheme description

3.2. Version 1.2

Fredrik Ullner <ullner@gmail.com>, 2013-01-16

- Added $ZOn and ZPipe0 feature.
- Fixed use of ASCIIDoc regarding a "link" in $Key.
- Added some minor descriptions to the extensions Feed and OpPlus.
- Added note about *Ref* in a $Lock.
- Added SaltPass feature
- Introduced a new category, "Extensions (commands)" to signify that a command require an a $Support and/or is not in the original implementations.
- Added note about source of information within this document.
- Changed $MyINFO according to how the flag is used.

3.3. Version 1.1

Fredrik Ullner <ullner@gmail.com>, 2013-01-02

- Added additional commands and extensions to document

3.4. Version 1.0

Fredrik Ullner <ullner@gmail.com>

- Initial release

4. Protocol

4.1. General

- Most messages begin with a *$* (dollar sign).
- Most messages end with a *|* (pipe).
- Command names and parameters use single space (' ') and additional dollar signs as separators.
- There is no standardization on code page to use. The local charset of the computer system is commonly used. The original implementation used win.1252.
- Hub port defaults to 411. The client should then try 412, 413 etc.
- Client - Client port defaults to 412.
- Some command names are incorrectly spelled, per the original protocol. This document notes when a spelling error is *intentional*, per the original protocol.
- The hub should validate that the client sending the message is actually the correct user.
- This document does not (mainly) use *<* and *>* brackets for illustration purposes; any occurrence should be taken literally, unless otherwise noted.
- This document use *[* and *]* brackets for optional data; any literal occurrence will be noted as such.

There are different types of routed messages:

Client to Hub
Client to Hub to Client
Hub to Client
Hub to Hub
Hub to Hublist
Pinger to Hub

4.1.1. Status messages

Most clients send a chat message to all users to indicate that a user has been kicked. That is, if a client A kicks client B, client A will send out a message to client C and D to indicate that the client B is kicked. To avoid being potentially flooded by this message, clients have implemented a de facto message to indicate kicks. <john> is kicking peter because: I am not a dog person| The key aspect here is the two phrases "is kicking" and "because:". Including these two parts, regardless of the other content, will cause most clients to show the message as a status bar message.

Note that the hub will not actually treat this as a kick.

4.1.2. Reference to hub in $Lock

To identify the source of the hub in a client - client connection, some clients add a reference to the hub in the $Lock that is sent, where the information is placed in the Pk parameter.

This information can also be provided in a client - hub connection.

Example:

```
$Lock EXTENDEDPROTOCOLABCABCABCABCABC Pk=DCPLUSPLUS0.666Ref=example.com:411
```

4.1.3. Escape sequences

There are two ways protocol delimiters are displayable (dollar sign, pipe etc);

Method 1

By using /%DCNXXX%/ where XXX is the decimal number for an ASCII character.

This escape sequence method is used by all implementations for the $Key/$Lock sequence (see below). This is also used by multiple implementations for normal viewing purposes (e.g. in a chat message).

This escape sequence predates the second method listed below.

The following escape sequences exist for this method;

Escape sequence	Description
/%DCN000%/	0x0a, null character. Not allowed elsewhere in the protocol.
/%DCN005%/	Enquiry character. Used by some commands.
/%DCN036%/	$, dollar sign. Used by (almost) all commands.
/%DCN096%/	`, grave accent.
/%DCN124%/	|, pipe. Used at end of commands.
/%DCN126%/	~, tilde.

Example:

```
<John> I am going to display a dollar sign /%DCN036%/ and then a pipe /%DCN124%/.|
```

Method 2

By using the HTML equivalent of the character.

This escape sequence was created in and is used by DC++.

Note that DC++ uses the first method for the $Key/$Lock sequence. DC++ uses this method for displayment purposes (e.g. chat messages).

The following escape sequences exist for this method;

Escape sequence	Description
&#36;	$, dollar sign. Used by (almost) all commands.
&#124;	|, pipe. Used at end of commands.
&amp;	&, ampersand.

Example:

```
<John> I am going to display a dollar sign &#36; and then a pipe &#124;.|
```

4.1.4. $Key/$Lock sequence

After the first NMDC client and hub was introduced, others began to create their own implementation of the NMDC protocol. The original NMDC client and hub was freeware but not open source. In a way to combat other implementations, the author of the protocol decided upon a way to restrict the other implementations by a pseudo-cryptographic key exchange.

This exchange is typically referenced as a pseudo-Diffie-Hellman key exchange for its similarity, although they are not equivalent.

The $Key and $Lock commands are not listed as extensions as they exist in at least NMDC hub 1.0.25 and NMDC client 1.0 Preview Build 9.1.

4.2. Security Considerations

4.2.1. Distributed Denial of Service Potential

The ongoing operation of NMDC's peer to peer connections provides opportunities for both malicious clients on certain hubs and malicious hubs to coordinate clients of those hubs in connecting en masse to an arbitrarily specified IP address and port. Hubs SHOULD mitigate this risk by only relaying such connection requests when it can verify that the IP addresses and ports contained therein belong to clients initiating such peer to peer connection requests. Clients

SHOULD effect similar amelioration through both attempting to reconnect only to hubs to which they have initially connected once and by providing upon connection referrer information regarding which hub has relayed to them the IP and port to which they are connecting.

See also "Reference to hub in $Lock".

4.2.2. Case-sensitivity Mismatches and Duplicated Shares Entries

Both accidentally due to differences in filesystem case-sensitivity assumptions and intentionally by malicious clients can arise filelists containing either entires either entirely or up to case identical. Client software SHOULD detect these cases and avoid wasteful downloading.

4.2.3. Filelist Processing Memory Usage

The bzip2-compressed filelists that NMDC URIs ending in a separator are required to process potentially expand due to malicious construction to cause a processing client to exhaust its memory or address-space capacity. Likewise, even non-malicious filelists can grow arbitrarily large such that in combination with other memory requirements of client software, similar exhaustion occurs. Client software therefore SHOULD bound bzip2 decompression memory usage and use streaming XML APIs to process filelists.

See [1] for more information.

4.2.4. Excessive Local Storage Consumption

The zlib compression supported by NMDC for client-client transfers permits construction of transferred data which consume disk space substantially more rapidly for the downloading client than network bandwidth would indicate. Client software SHOULD monitor downloads for impending storage capacity limits with a view to prevent local denials of service resulting from exceeding them.

4.3. URI scheme

See the URI scheme document following this document.

4.4. Commands

4.4.1. Chat message

```
<nick> message|
```

Contexts: Client → Hub → Client, Hub → Client

Will send a chat message by *nick*. The hub should broadcast the message to all clients if a client send this.

Note that the brackets (< and >) are required. Some implementations allow spaces in the nickname; be aware that other implementations may simply search for the first occurrence of a space. If the chat message contain a pipe, it should be replaced by its HTML equivalent.

Example:

```
<John> cats are cute|
```

4.4.2. $To

```
$To: othernick From: nick $<nick> message|
```

Contexts: Client → Hub → Client, Hub → Client

Will send a private message from one user to another user. The same rules as a basic chat message apply.

Example:

```
$To: john From: peter $<peter> dogs are more cute|
```

4.4.3. $ConnectToMe

```
$ConnectToMe RemoteNick SenderIp:SenderPort
```

Contexts: Client → Hub → Client

Request that remotenick connect to the sending user for an active TCP connection for file transfers. Clients sending this must allow incoming TCP connections. There is no default port.

Example:

```
$ConnectToMe john 192.168.1.2:412|
```

Note that the above is for older clients.

Newer versions of NMDC client include the sender's nick as well (but may not be compatible with all implementations);

```
$ConnectToMe SenderNick RemoteNick SenderIp:SenderPort
```

Example:

```
$ConnectToMe peter john 192.168.1.2:412|
```

4.4.4. $RevConnectToMe

```
$RevConnectToMe SenderNick RemoteNick|
```

Contexts: Client → Hub → Client

Request that the remotenick send a ConnectToMe back to SenderNick. Clients sending do not allow incoming TCP connections whereas the remote user does (or should).

Example:

```
$RevConnectToMe peter john|
```

4.4.5. $Ping

```
$Ping sender_ip:sender_port|
```

sender_ip is the IP address of the remote client.

sender_port is the port of the remote client that is being listened to.

4.4.6. $GetPass

```
$GetPass|
```

Contexts: Hub → Client

Request that the client send a password that correspond to a user account (matched by the user's nick name).

4.4.7. $MyPass

```
$MyPass password|
```

Contexts: Client → Hub

Providing the password (in plain text) to the hub after a request. Implementations should be aware of potential pipe and dollar signs in the password.

Example:

```
$MyPass qwerty|
```

4.4.8. $LogedIn

```
$LogedIn nick|
```

Contexts: Hub → Client

Sent to users who successfully log in. Note that the message should only be sent to operators. Note that some implementations do not use this command. The spelling of this command is not a mistake.

Example:

```
$LogedIn john|
```

4.4.9. $Get

```
$Get file$offset|
```

Contexts: Client → Client

file is the remote location (including path) of the file.

offset is the byte offset to start at for resuming files.

Example:

```
$Get C:/Uploads/myfile.txt$15|
```

4.4.10. $Send

```
$Send|
```

Contexts: Client → Client

This is used as a way to specify that the file should be sent. The uploader should proceed to stream the amount of bytes requested previously.

4.4.11. Provide file size

```
$FileLength file_size|
```

Contexts: Client → Client

This command is used as a way to provide the size of the file requested.

4.4.12. $GetListLen

```
$GetListLen|
```

Contexts: Client → Client

Get file list size.

4.4.13. $ListLen

```
$ListLen file_size|
```

Contexts: Client → Client

This command is used as a way to provide the size of the file list of the client.

4.4.14. $Direction

```
$Direction direction number|
```

Contexts: Client → Client

This command is used as a way to decide which party should be allowed to download.

The direction is either *Upload* or *Download*, depending on whether the connection request is a ConnectToMe or ReverseConnectToMe.

The number is a random number (above 0), typically between 1 and 32767. If both clients want to download, the client with the highest number gets to start downloading first. If the numbers are equal, the connection is closed.

This command should be sent;

- After a $Key command or
- After a $Lock command with EXTENDED information

I.e., if the $Lock is EXTENDED, $Direction MUST be sent after the $Lock command and not the $Key command.

Implementation note: The highest number DC++ sends is 32767 (2^15-1, 0x7FFF) as the Neo-Modus Direct Connect client will disconnect if it receives a higher number. Clients can send a higher value, in risk of losing compatibility.

Implementation note: A potential bug, that may be fixed, in DC++ revolved around the following circumstance;

- Client A has 0 free slots
- Client B has free slots (doesn’t matter how many)
- Client A and client B want a file from one another
- Client A loses the random number "battle"

Client B will not get a slot because of client A does not have any slots available. DC++'s status bar

of client B will say "Connecting..." instead of saying "No slots available" that is the de facto message. A suggested patch involve "cheating"; client B will use a high random number after it has lost and will return to normal when it has won.

4.4.15. $Cancel

```
$Cancel
```

Contexts: Client → Client

This command is used as way to indicate that the transfer of a file should be cancelled prematurely.

This command is sent by the downloader. When the uploader receives the message, it should stop sending data followed by a $Canceled. It is possible that some implementations continue to send some data after the $Cancel and $Canceled; the downloader should discard this data.

This command was only implemented in NMDC v.1 and not in subsequent version or in other clients.

Note that there is no pipe (|) at the end of this command!

4.4.16. $Canceled

```
$Canceled
```

Contexts: Client → Client

This command is used as way to indicate that the transfer of a file was cancelled prematurely.

This command is sent by the uploader.

This command was only implemented in NMDC v.1 and not in subsequent version or in other clients.

The spelling of this command is not a mistake.

Note that there is no pipe (|) at the end of this command!

4.4.17. $BadPass

```
$BadPass|
```

Contexts: Hub → Client

Indicates that the supplied password is invalid. The client shall be immediately disconnected after this message is sent.

4.4.18. $HubIsFull

```
$HubIsFull|
```

Contexts: Hub → Client

Indicates that the hub has reached its maximum amount of users and will not accept additional users.

4.4.19. $ValidateDenide

```
$ValidateDenide nick|
```

Contexts: Hub → Client

The requested nick name is already taken by or is reserved for another user. The spelling of this command is not a mistake.

Example:

```
$ValidateDenide john|
```

4.4.20. $MaxedOut

```
$MaxedOut|
```

Contexts: Client → Client

Sent by a client to another when there are no more slots available upon request of files.

See the QP extension for an extended $MaxedOut.

4.4.21. $Failed

```
$Failed message|
```

Contexts: Client → Hub, Client → Hub → Client, Client → Client, Hub → Client

General purpose fail command. Implementations may use this command to signify files that are not available. $Failed is usually sent in response to a GetZBlock, UGetBlock and UGetZBlock.

4.4.22. $Error

```
$Error message|
```

Contexts: Client → Hub, Client → Hub → Client, Client → Client, Hub → Client

General purpose fail command. Implementations may use this command to signify files that are not available. $Failed is usually sent in response to a GetZBlock, UGetBlock and UGetZBlock.

4.4.23. $Search

```
$Search ip:port search_string|
$Search Hub:nick search_string|
```

Contexts: Client → Hub → Client

The former is sent by active clients and the latter for passive clients. The IP is the client's own IP address and the port is an open UDP port that accept incoming UDP traffic. "Hub" should be taken literally. The nick is the nickname of the searching user.

The search string

The string describing the file or directory the client is searching for. It is made up of a question mark (*?*) delimited string as follows:

```
size_restricted?is_max_size?size?data_type?search_pattern
```

size_restricted is *T* if the search should be restricted to files of a minimum or maximum size, otherwise *F*.

is_max_size is *F* if size_restricted is *F* or if the size restriction places a lower limit on file size, otherwise *T*.

size is the minimum or maximum size of the file to search for (according to is_max_size) if size_restricted is *T*, otherwise 0.

data_type restricts the search to files of a particular type. It is an integer selected from:

- 1 - For any file type
- 2 - For audio files
- 3 - For compressed files
- 4 - For document files
- 5 - For executable files
- 6 - For picture files
- 7 - For video files
- 8 - For folders
- 9 - For TTH searching

search_pattern is used by other users to determine if any files match. If it is a TTH search, search_pattern should be "TTH:hash".

Spaces in the search pattern are replaced by a dollar sign *$*. [CHECK: "As with all NMDC messages, *$* and *|* are escaped with "$" and "|", with *&* being further replaced with "&"."]

Examples:

> Active search:
>
> ```
> $Search 192.168.1.5:412 T?T?500000?1?Gentoo$2005
> $Search Hub:SomeNick T?T?500000?1?Gentoo$2005
> ```

> Passive search:
>
> ```
> $Search 192.168.1.5:412 F?T?0?9?TTH:TO32WPD6AQE7VA7654HEAM5GKFQGIL7F2BEKFNA
> $Search Hub:SomeNick F?T?0?9?TTH:TO32WPD6AQE7VA7654HEAM5GKFQGIL7F2BEKFNA
> ```

4.4.24. $SR

```
$SR source_nick result free_slots/total_slots<0x05>hub_name (hubip[:port])[<0x05>target_nick]|
```

Contexts: Client → Hub → Client, Client → Client

Sent by a client when a match to a search is found.

If the $Search was a passive one, the $SR is returned via the hub connection (TCP). In this case, <0x05>target_nick must be included on the end of the $SR. The hub must strip the deliminator and <target_nick> before sending the $SR to target_nick. If the search was active, it is sent to the IP address and port specified in the $Search via UDP.

Result is one of the following:

- filename<0x05>filesize
- directoryname

File size shall be in bytes.

"<0x05>" used above for deliminators are the 5th character in the ASCII character set. The brackets should consequentially not be taken literally.

The port for the hub only needs to specified if its listening port is not the default (411).

On UNIX, the path delimiter / must be converted to \ for compatibility.

Implementations should send a maximum of 5 search results to passive users and 10 search results to active users.

Free_slots are the amount of available slots left. Total_slots are the total amount of slots.

For files containing TTH, the hub_name parameter is replaced with TTH:hash

Examples:

> Active result:
>
> ```
> $SR User1 ponies.txt<0x05>437 3/4<0x05>Testhub (192.168.1.1:411)|
> $SR User5 images 0/4<0x05>Testhub (192.168.1.1:411)|
> $SR User6 pictures 0/4<0x05>Testhub (192.168.1.1)|
> ```

> Passive result:
>
> ```
> $SR User1 ponies.txt<0x05>437 3/4<0x05>Testhub (192.168.1.1:411)<0x05>User2|
> $SR User5 images 0/4<0x05>Testhub (192.168.1.1:411)<0x05>User2|
> $SR User6 pictures 0/4<0x05>Testhub (192.168.1.1)<0x05>User2|
> ```

4.4.25. $MyINFO

```
$MyINFO $ALL nick description$ $<connection><flag>$mail$share_size$|
```

Contexts: Client → Hub (→ Client)

This command is part of the Client-Hub Handshake and during login, and is sent after the client receive $Hello with their own nick. Client resend this on any change. It's broadcasted to all clients.

The < and > use in here should not be taken literally; they are there to visually distinguish connection from flag. Consequently, the flag shall immediately follow the connection.

Connection is a string for the connection:

- Default NMDC1 connections types 28.8Kbps, 33.6Kbps, 56Kbps, Satellite, ISDN, DSL, Cable, LAN(T1), LAN(T3)
- Default NMDC2 connections types Modem, DSL, Cable, Satellite, LAN(T1), LAN(T3)

Later implementations also send other speeds. Implementations should simply display the value to the user and not create any inhibitions based on the reported speed.

Flag

Flag is a ASCII character (8 bit byte).

There are multiple implementations of this byte.

The values listed here are the bits within the flag byte. The following is the original:

Bit	Description
1	Normal or no status.
2	Away.
3	Server status. Used when the client has uptime > 2 hours, > 2 GB shared, upload > 200 MB.

Bit	Description
4	Fireball status. Used when the client has had an upload > 100 kB/s.

The values listed here are the bits within the flag byte. Newer implementations change the numbers to be extensions of the original and are the following:

Bit	Description
5	Support for TLS. For all connections when TLS extension is supported. For download only when TLS1 extension is supported.
6	Support for TLS upload when TLS1 extension is supported.
7	Support for IPv4 when IP64 extension is supported.
8	Support for IPv6 when IP64 extension is supported.

The values listed here are decimal values. Other implementations interpreted this to be the following:

Decimal	Description
1	Normal
2 and 3	Away
4 and 5	Server. Used when the client has uptime > 2 hours, > 2 GB shared, upload > 200 MB.
6 and 7	Server away
8 and 9	Fireball. Used when the client has had an upload > 100 kB/s.
10 and 11	Fireball away

Example verification with the first and second approach:

Flag calculation	Result
if Flag & 0x1	Normal status
if Flag & 0x2	Away status
if Flag & 0x4	Server status
if Flag & 0x8	Fireball status
if Flag & 0x10	TLS support for downloads
if Flag & 0x20	TLS support for uploads (with TLS1)
if Flag & 0x40	IPv4 support
if Flag & 0x80	IPv6 support

Example verification with the third approach:

Flag calculation	Result
if Flag == 1)	Normal status
if Flag == 2 OR Flag == 3	Away status
if Flag == 4 OR Flag == 5	Server status
if Flag == 6 OR Flag == 7	Server and away status
if Flag == 8 OR Flag == 9	Fireball status

Flag calculation	Result
if Flag == 10 OR Flag == 11	Fireball and away status

Tag

This addition to the description field was introduced to ease concerns by hub operators that DC++ users were joining too many hubs and not allowing enough uploads. (The original DC client only allowed users to join a single hub.) The tag is now the de facto standard.

Example:

> <++ V:0.673,M:P,H:0/1/0,S:2>

- "++": indicates the client (in this case, DC++)
- "V": tells you the version number
- "M": tells if the user is in active (A), passive (P), or SOCKS5 (5) mode
- "H": tells how many hubs the user is on and what is his status on the hubs. The first number means a normal user, second means VIP/registered hubs and the last one operator hubs (separated by the forward slash [/]).
- "S": tells the number of slots user has opened
- "O": shows the value of the "Automatically open slot if speed is below xx KiB/s" setting, if non-zero
- The brackets < and > shall be taken literally.

Note that the IPv4/IPv6 extensions add additional data to the M. See the IPv4/IPv6 extensions for further information.

Example: Note that the example uses *0x31* for signaling the flag as *Normal*, for ease of display.

> $MyINFO $ALL johndoe <++ V:0.673,M:P,H:0/1/0,S:2>$
> $LAN(T3)0x31$example@example.com1234\|

4.4.26. $GetINFO

```
$GetINFO <other_nick> <nick>|
```

Contexts: Client → Hub → Client

Request (general) client information.

<nick> is this sending client's nick.

<other_nick> is the nick of the user that *<nick>* wants to know about. The server must respond with exactly the $MyINFO command sent by *<other_nick>* to the hub.

Example:

> $GetINFO peter john|

4.4.27. $Hello

```
$Hello user|
```

Contexts: Hub → Client

When a new user logs in, the hub will send this command to the new user to inform them that they have been accepted for hub entry.

4.4.28. $Version

```
$Version version|
```

Contexts: Client → Hub

Sent by clients to the hub after $Hello is received to denote the version used for the client. This command is nowadays not used to denote the client's version but was used frequently by the original Neo-Modus Direct Connect (NMDC) client. The last version of the Neo-Modus client was 1.0091 and is what is commonly used by current clients. The default system locale is used, which means the period may be a comma etc.

Version is 1.0091 by default.

4.4.29. $HubName

```
$HubName name|
```

Contexts: Hub → Client

The name of the hub that should be displayed by clients to users. The name is sometimes interpreted as the "topic" (current discussion topic or general theme of the hub), in cases where $HubTopic does not exist. The hub could send different names to different users, as well as the ability for multiple hubs (that are inherently separated) to have the same name, so the client should not use the hub name as a unique identifier.

4.4.30. $GetNickList

```
$GetNickList|
```

Contexts: Client → Hub

Request that the hub send the nick names of all users that are connected.

4.4.31. $NickList

```
$NickList nick$$nick2$$nick3[...]|
```

Contexts: Hub → Client

Providing the full listing of users, including bots and operators. List is separated by "$$".

Example:

```
$NickList john$$peter$$richard$$marie$$sarah|
```

4.4.32. $OpList

```
$OpList nick$$nick2$$nick3[...]|
```

Contexts: Hub → Client

Providing the full listing of operators. This is a subset of the users provided in $NickList. List is separated by "$$".

The following should be sent if there are no operators online:

```
$OpList|
```

Example:

```
$OpList john$$peter|
```

4.4.33. $Kick

```
$Kick victim|
```

Contexts: Client → Hub → Client

Requests that the hub kicks a user (terminates the connection to the user). The hub will validate that the issuing user actually have permission to kick the other user. The message does not specify a reason to the kick; hubs may decide instead to send a default "you have been kicked" message. Time frame (of a potential ban) does not exist in the protocol, as it is decided by the hub's configuration. It is up to the hub to decide the course of action if the user is not allowed to issue the kick, but the majority of implementations will simply ignore the message or send a message back ("you are not allowed to issue the command"). Many clients that issue the Kick command will precede the message with a message directed (either normal main chat or with $To) to the offended user with the reason for the kick.

4.4.34. $Close

```
$Close victim|
```

Contexts: Client → Hub → Client

Requests that the hub kicks a user (terminates the connection to the user), but no message will be sent to the victim client. All other information is similar to $Kick.

4.4.35. $OpForceMove

```
$OpForceMove $Who:victim$Where:address$Msg:reason|
```

A request made by privileged users to redirect a user from the hub. The message should be sent to the offending user in a main chat message or in a $To message.

victim is nick name of the victim.

address is the address to redirect to, abiding to the specified rules for URIs.

reason is why the redirect was issued.

Example:

> $OpForceMove $Who:richard$Where:example.com:411$Msg:I think you'll like this hub better|

4.4.36. $ForceMove

```
$ForceMove address|
```

Informs the user that it should close its connection and instead connect to the address specified. Many implementations will send $ForceMove followed by a main chat message or a $To message.

Example:

> $ForceMove example.com:411| $To:richard From: OpNick $<peter> You are being re-directed to example.com:411 because: I think you'll like thus hub better.|

4.4.37. $Quit

```
$Quit nick|
```

Contexts: Hub → Client

Inform users in a hub that *nick* has disconnected from the hub. If a user desire to disconnect from a hub, the client should simply terminate the connection and not send $Quit.

Example:

> $Quit peter|

4.4.38. $Lock

```
$Lock lock Pk=pk|
```

Contexts: Hub → Client, Client → Client

This is a mechanism to request 'certification' for verification. The lock is a command that added by NMDC to keep other clients (such as DC++) from being able to use NMDC hubs. The lock require a response (given in a $Key) that needs to be calculated, as per the lock's data. Hubs are not required to check the validity of the response, but must still send it.

Note this command, and its use, are used for all connections.

This command should be the first to be sent in a client to hub connection. This command should be sent after both have used $MyNick in a client to client connection. Both clients (in a client to client connection) should send $Lock.

The lock data is a sequence of random characters, excluding space (' '), dollar sign ('$') and a pipe ('|').

Pk was originally intended for use with SSL/TLS keys for secure connections, but is not used by any implementation today. Some implementations send their name and version in this field.

Implementation note: In NMDC hub;

- Lock's length varied from 46 to 115 characters
- Pk's length was static with 16 characters.
- Lock's characters varied from ASCII code point 37 to ASCII code point 122.

Example: See $Supports for the use of EXTENDED.

> $Lock EXTENDEDPROTOCOLABCABCABCABCABCABC Pk=DCPLUSPLUS0.706ABCABC|
> $Lock EXTENDEDPROTOCOL_verlihub Pk=version0.9.8e-r2|

4.4.39. $Key

```
$Key key|
```

Contexts: Client → Hub, Client → Client

The key is given as a response for a $Lock command, as calculated by the following algorithm.

The key has exactly as many characters as the lock. Except for the first, each key character is computed from the corresponding lock character and the one before it. If the first character has index 0 and the lock has a length of len then:

for (i = 1; i < len; i++) { key[i] = lock[i] xor lock[i-1]; }

The first key character is calculated from the first lock character and the last two lock characters:

key[0] = lock[0] xor lock[len-1] xor lock[len-2] xor 5

Next, every character in the key must be nibble-swapped:

for (i = 0; i < len; i++) { key[i] = ((key[i]<<4) & 240) | ((key[i]>>4) & 15); }

Finally, the characters with the decimal ASCII values of 0, 5, 36, 96, 124, and 126 cannot be sent to the server. Each character with this value must be substituted with the string /%DCN000%/, /%DCN005%/, /%DCN036%/, /%DCN096%/, /%DCN124%/, or /%DCN126%/, respectively. The resulting string is the key to be sent to the server.

If your programming language of choice doesn't have xor or shift operations on characters, convert them to integers. If it doesn't have a bit shift at all, then x « y = x*(2y) and x » y = x/(2y) for all integers x and y (** is the exponent operation or "to the power of"). Be sure to use unsigned values everywhere and do not do sign extension. Shift operations always lose the high or low bit (they are not roll operations!). The & (and) and | (or) operations are all logical, not boolean (eg. 6 & 13 = 4, not 1).

When the hub connects to the hublist, it must undergo a similar Lock/Key negotiation. The <key> calculation is the same, but the special number 5 in the second step is replaced with the following value is not computed from the hub's listening port, but rather the random outgoing port that the Winsock selects. localport: 1) & 255

Note that the data sent should be in its raw form.

Example: Note that this (and every other) example uses '011010110110010101111001' as key, but it should be interpreted as a binary value.

> $Key 011010110110010101111001|

4.4.40. $MultiConnectToMe

```
$MultiConnectToMe remote_nick sender_ip:sender_port
```

This command is a client side command to take advantage of hub linking feature in the NeoModus Direct Connect Hub software. When a local client sends $MultiConnectToMe to the hub, it is forwarded on to every hub which is linked to by UDP, as in the $Search hub<→hub protocol command.

This command is used in conjunction with $MultiSearch.

This support was problematic, as the NeoModus Direct Connect Hub did not provide a way to indicate that it was part of a link, and further did not provide the ability to broadcast joins and parts of remote users. Some implementations would send $MultiConnectToMe commands even when the remote user was offline.

Example:

> $MultiConnectToMe peter 192.168.0.138:19346

4.4.41. $MultiSearch

```
$MultiSearch ip:port search_string|
```

This command is a client side command to take advantage of hub linking feature in the NeoModus Direct Connect Hub software. When a local client sends $MultiSearch to the hub, it is forwarded on to every hub which is linked to by UDP, as in the $Search hub<→hub protocol command.

This command is used in conjunction with $MultiConnectToMe.

For syntax on the parameters, see $Search.

4.5. Extensions (commands)

4.5.1. $BotList

```
$BotList nick$$nick2$$nick3[...]|
```

Contexts: Hub → Client

Providing the full listing of bots. This is a subset of the users provided in $NickList. List is separated by "$$".

Add BotList to the $Supports to indicate support for this.

Example:

```
$BotList marie|
```

4.5.2. $ADCGET

```
$ADCGET type identifier start_pos bytes|
```

Contexts: Client → Client

This is a port of the ADC approach of signifying the request. <start_pos> counts 0 as the first byte. <bytes> may be set to -1 to indicate that the sending client should fill it in with the number of bytes needed to complete the file from <start_pos>. <type> is a [a-zA-Z0-9]+ string that specifies the namespace for identifier and clients should recognize the type "file".

"file" transfers transfer the file data in binary, starting at <start_pos> and sending <bytes> bytes. Identifying must come from the namespace of the current hash.

Add ADCGet to the $Supports to indicate support for this. Support for ADCGet imply support for both $ADCGET and $ADCSND.

4.5.3. $ADCSND

```
$ADCSND type identifier start_pos bytes|
```

Contexts: Client → Client

This is used as a way to specify that the file should be sent. The uploader should proceed to stream the amount of bytes requested previously.

This is a port of the ADC approach of signifying sending. The parameters correspond to the ADCGET parameters except that if <bytes> equals -1 it must be replaced by the number of bytes needed to complete the file starting at <start_pos>.

Add ADCGet to the $Supports to indicate support for this. Support for ADCGet imply support for both $ADCGET and $ADCSND.

4.5.4. $UserIP

```
$UserIP nick|
$UserIP nick ip|
```

Contexts: Client → Hub, Hub → Client

A client may request IP address information about clients. The hub will respond with a command with the same name, but with different parameter data.

Example:

Client	Server
$UserIP johndoe|	
	$UserIP johndoe 192.168.1.2|

4.5.5. $UserIP extension

```
$UserIP nick1 IP1$$nick2 IP2[...]|
```

Contexts: Hub → Client

This is similar to the other $UserIP command, except that there is no request. The hub will, if the client signals support for UserIP2, send all users upon login.

The list consist of each user and their IP, including the connecting client. The hub should send each new client's IP when they login.

Note that $$ is used as delimiter.

Add UserIP2 to the $Supports to indicate support for this.

Example:

Client	Server
	$UserIP johndoe 192.168.1.2$$janedoe 192.168.1.3|

4.5.6. $BotINFO

```
$BotINFO description|
```

Contexts: Client → Hub

Bot description can be any string, usually information regarding, and address of, the hublist.

Add BotINFO to the $Supports to indicate support for this.

4.5.7. $HubINFO

```
$HubINFO name$address:port$description$max_users$min_share$min_slots$max_hubs$hub_type$hubowner_login|
```

Contexts: Hub → Client

Hub name, address and description are the items which will be shown in the hublist (some hubs have multiple addresses and this helps to get primary address). Description changes often on some hubs so this helps with updating it.

Max users, min share, min slots and maximum number of open hubs for the client helps users to find suitable hubs.

Hub type gives information about the hub software and script which gave the information.

Hubowner login is meant to help hubowners to edit information about their hub directly from the hublist portal It is usually an email address where the account/password information should be sent.

If the hub address is 127.0.0.1, the Hublist.org pinger will remove the hub from its database. (Or is supposed to.)

Add HubINFO to the $Supports to indicate support for this.

4.5.8. $HubTopic

```
$HubTopic topic|
```

Contexts: Hub → Client

The hub topic, be it current discussion topic or general theme of the hub, which allow users to quickly see what the discussion and file sharing themes are. Hub pingers frequently use this message for hub related information.

Add HubTopic to the $Supports to indicate support for this.

4.5.9. $Supports

```
$Supports extension1 extension2 [...]|
$Supports extension1 extension2 [...] |
```

Contexts: Client → Hub, Hub → Client, Client → Client

This command is used to negotiate and notify about protocol extensions.

If a client or hub implements an extension, the $Lock command MUST start with 'EXTENDEDPROTOCOL'.

This command MUST be sent before $Key.

Implementations should only send extension specific messages if the other party has signaled support for it.

There must be at least 1 (one) supported extension.

For client extensions, the extension name should be the same as the command name.

Note that DC++ 0.XXX added a space after the last extension. I.e., the first form is the original implementation of $Supports.

Example: The following example signals support for 7 (seven) different extensions. See corresponding command/extension for description.

> $Supports UserCommand NoGetINFO NoHello UserIP2 TTHSearch ZPipe0 GetZBlock|

4.5.10. Capabilities

```
$Capabilities [unknown]|
```

Contexts: Hub → Client

This command is used to negotiate and notify about protocol extensions, similar to the more popular $Supports. Its content is unknown as of date.

4.5.11. IN

```
$IN nick$data[$data]|
```

Contexts: Client → Hub, Hub → Client

This command is designed to replace the static $MyINFO. This command consist of separate smaller parts which all have a 1-byte identifier and are separated by a dollar sign ($).

If a client wishes to remove a part of its $IN (say for example its description), it must send this parts identifier immediatly followed by the dollar sign ($), indicating it is empty.

The order in which parts are sent is free to choose, but tag parts should be grouped together.

Add IN to the $Supports to indicate support for this.

This command deprecates $OpList and $BotList.

A client tag has only one identiifer indicating it is concerning the tag, but all parts of this tag also have their own 1-byte identifier. In the tag, the parts are delimited with a space (' ').

The data consist of two elements: identifier and its data.

The following table lists the identifiers:

Identifier (character code)	Description
D (68)	Description
T (84)	Tag
C (67)	Connection
F (70)	Status flag
E (69)	E-mail address
S (83)	Share size in bytes

Examples:

> $IN john$Dmy_new_description| $IN john$S1024| $IN john$S1024$$Dmy_new_description|

The tag has the following identifiers in the data:

Identifier (character code)	Description
c (99)	Client
v (118)	Version
m (109)	Mode
h (104)	Hub count, x/y/z (where x is the number of hubs as a normal user, y as the number of hubs as registered and z as the number of hubs as operator)
s (115)	Slots
f (102)	Free slots
l (108)	Bandwidth limit (L: and B: in $MyINFO)
o (111)	O: in $MyINFO
r (114)	R: in $MyINFO

Tag identifiers are separated by a space (' ').

Examples:

```
$IN john$Ts3| $IN john$Tc++ v0.666|
```

The status is a 32-bit integers where the bits are according to the following table:

Bit	Description
1	Default value
2	User is away
3	USer is server
4	User is fireball
5	User is OP
6	Client is a bot
7	Client is in DND mode
8	Client support encryption
9	Client supports partial search

Bit 5 and Bit 6 of the 32-bit status flag are used to indicate wether a user is an OP or wether the $IN string send by the hubsoft belongs to a bot. These values are read-only and should never be changed by a client. The hubsoft must disconnect for this. If a client connects and has successfully logged in with the correct password, the hubsoft will set bit 5 of the status flag and sends this part to all connected users, including the newly connected OP. The connecting client must save the status flag it received from the hub and use it when updating status such as away and fireball. However, as said, it is not allowed to set or reset bit 5 and bit 6 of the status flag.

Bit 7 of the 32-bit status flag indicates if the client is in DND-mode (Do-Not-Disturb). Unlike Away mode, this mode will prevent the client to receive any pm's. Clients are responsible themselves for setting/resetting this bit. Once this bit is set, and the hub receives a $To: string for a client in DND-mode, the hub must ignore the $To and reply to the sender with a message that the receiver is in DND-Mode. A client having this bit set, should automatically reset this bit when sending a pm itself. Ideally, newer clients supporting IN, may prevent themselves from sending $To strings to other clients having this bit set. The hub will always have the last the say in forwarding a PM or not. $To strings generated by the hubsoft to the client are not included in this and will be send regardlessly of status (i.e. messages from bots).

4.5.12. MCTo

```
$MCTo: target $sender message|
```

Contexts: Client → Hub, Hub → Client

This is a private-message to a single user that should be displayed as an ordinary chat-message.

Add MCTo to the $Supports to indicate support for this.

target is the client nick that should receive the message.

sender is the client nick that is the sender

message is the actual message.

Examples:

```
$MCTo john $peter Cats are cute|
$MCTo peter $john I like dogs|
```

4.5.13. $NickChange

```
$NickChange old_nick new_nick|
```

Contexts: Client → Hub

This allow a client to change the nick without logging out and logging back in.

old_nick is the old user nick.

new_nick is the new user nick.

Add NickChange to the $Supports to indicate support for this.

The client will have to update its own $MyINFO string and user list.

The hub will will send $ClientNick to the client to validate that the change is done. To all other users, a $Quit old_nick will be sent and a $MyINFO $ALL of the user with the new nick.

If the user is an operator, $OpList will be updated appropriately as well.

4.5.14. $ClientNick

```
$ClientNick new_nick|
```

This validates that the hub has acknowledged the change of nick during runtime that was initiated by $NickChange.

new_nick is the new user nick.

Add ClientNick to the $Supports to indicate support for this.

4.5.15. FeaturedNetworks

$FeaturedNetworks is a protocol extension of the APN MultiHubChatsystem to identify the different hubs and other entry points (IRC, Telnet) etc. It was primarily created to aid APN developers and to allow a better integration of multi hub chat systems.

Messages coming through the chat network are prefixed with an unique network node (entry point) identifier, normally three characters long. The central point of the chat network is a single hub that collects chat messages from one entry point and sends them back all other entry points.

Add FeaturedNetworks to the $Supports to indicate support for this.

Syntax

An APN MultiHubChat message normally has this format:

<(YYY)XXXXX> MMMMMMM|Where YYY is the entry point prefix, XXXXX is the username and MMMMMMM is the message.

To allow the different entry point handlers to identify which messages are coming from other multi hub chat entry points, there is a command send by the hub after login called FeaturedNetworks:

$FeaturedNetworks YYYYYY$YYY|YYY stands for one entry point to the chat network. In the biggest currently running instance of the APN MultiHubChat system, five or more different prefixes are used.

Implementation

If implemented, this command could be sent from one of the entry point hubs aswell to aid the client in distinguishing user set prefixes ([BBB], [psv], (FUG) etc..) from network prefixes, and can be used by bots to detect prefix abuse.

The prefixes are to be defined by the network administrator. In the biggest running instance of the network however, the following prefixes are used by several MHC plugins for the different entry points:

- APx - Hub x (DC, where x is Â¹,Â³ or Â²).
- APt - Telnet Access (Telnet Chat Plugin)
- IRC - IRC Access (IRC plugin)
- APs - Shoutcast (Shoutcast Announcer Plugin)

$FeaturedNetworks can be implemented by virtually any hub by either using the MHC bot or by injecting it into the server ? client stream using a textfile (like a MOTD).

4.5.16. $Z

```
$Z blob|
```

Contexts: Hub → Client

This command's intention is to compress (with ZLib) data to decrease bandwidth use. The blob uncompressed is one or more commands, e.g. a $Search followed by a $MyINFO.

Add ZLine to the $Supports to indicate support for this.

The command adds an escaping sequence:

Character	Escape
\	\\

Character	Escape
|	\P

4.5.17. $ZOn

```
$ZOn blob|
```

This command's intention is to compress (with ZLib) data to decrease bandwidth use. The blob uncompressed is one or more commands, e.g. a $Search followed by a $MyINFO.

Compression algorithm shall be LZ77.

The end of the block is an EOF.

Add ZPipe0 to the $Supports to indicate support for this.

The command adds no escaping.

4.5.18. $GetZBlock

```
$UGetBlock start bytes filename|
```

start is the 0-based (yes, 0-based, not like get that's 1-based) starting index of the file used

bytes is the number of bytes to send

filename is the filename.

The other client then responds "$Sending <bytes>|<compressed data>", if the sending is ok or "$Failed <errordescription>|" if it isn't.

If everything's ok, the data is sent until the whole uncompressed length has been sent. *bytes* specifies how many uncompressed bytes will be sent, not compressed, as the sending client doesn't know how well the file will compress. $Sending is needed to be able to distinguish the failure command from file data. Only one roundtrip is done for each block though, minimizing the need for maintaining states.

Compression: Compression is done using ZLib (v 1.1.4 in DC++ 0.21's case), using dynamic compression level. The compression level can of course be changed by the implementator to reduce CPU usage, or even just store compression in the case of non-compressible files, which then works as adler32 check of the transferred data.

Support of $GetZBlock also implies support for $UGetZBlock.

4.5.19. $UGetBlock

```
$UGetBlock start bytes filename|
```

This is the same command as $GetZBlock except this command is uncompressed and the filename is UTF-8 encoded.

The filename is encoded as UTF-8, which allows filenames to use characters that are not in the system's encoding. $UGetBlock must be implemented if XmlBZList is advertised.

4.5.20. $UGetZBlock

This is the same as $UGetBlock except that the stream is compressed.

4.5.21. $GetTestZBlock

This command is deprecated and was a test command during the development of $GetZBlock. Implementations should not use this command.

4.5.22. $Sending

```
$Sending diff|
```

Diff is the difference between the end byte and start byte. If the requested bytes was -1 then diff should be omitted.

This is sent as a response to $GetZBlock, $UGetBlock and $UGetZBlock.

The start and end byte are 0-based; the first byte of a file is assumed to be the byte number 0.

4.5.23. $ClientID

Add ClientID to the $Supports to indicate support for this.

4.5.24. $GetCID

Add ClientID to the $Supports to indicate support for this.

4.5.25. $UserCommand

```
$UserCommand type context details|
```

Add UserCommand to the $Supports to indicate support for this.

type is a positive integer describing the kind of command:

Value	Description
0	Separator
1	Raw
2	Raw nick limited (same as raw with the exception that it should only be used once per %[nick])
255	Erase all previously sent commands

context is a integer that controls which contexts the menus will be shown. The value is derived by logical OR of the following values;

Value	Description
1	Hub context
2	User context
4	Search context
8	File list

details differ depending on the type.

Type	Detail information
0 and 255	Leave this field empty
1 and 2	Detail should be *title$command*.

Escaping

Escaping of dollar and pipe is necessary for the <command> in *'raw*' mode. The DC++ escape sequence will be used. i.e. | for pipe, $ for dollar and & for the ampersand. Escaping is used for all fields before they are sent to the hub / shown to the user. As with all NMDC commands, they must be terminated by the pipe character

Details: Separator

```
$UserCommand 0 <context>|
```

Will add a menu separator (vertical bar) to the specified contexts (<context>). It is legal to add text (after the space) before the pipe, but it won''t be used (yet).

====== Details: Raw $UserCommand 1 <context> <title>$<raw>|

Will add a raw menu item with title <title> with command <raw>. This command must end with a |, if not it should be discarded. The raw command may be used to specify multiple commands to be sent to the hub.

Details: Raw nick limited

```
$UserCommand 2 <context> <title>$<raw>|
```

Is exactly the same as Raw, except that the command should only be run once per %[nick]. This is to prevent the client from sending out more than one message that disconnects someone. Generally, this is only useful in the User-File context (e.g. viewing Search Results) where it is possible to select one user multiple times.

Details: Erase

```
$UserCommand 255 <context>|
```

Will erase all commands that the hub has sent previously. This is for hubs/scripts that allow for updates while running. The erase all is intentional, keeping it simple. Note that contexts must still be used, and that erasing will remove all commands that match any of those ORed contexts (i.e. 7 will remove commands previously sent with any context of 1 through 7) but only from that context

Examples:

> $UserCommand 2 6 Kick$$To: %[nick] From: %[mynick] $<%[mynick]> You are being kicked====|$Kick %[nick]|| $UserCommand 255 1|

4.6. Extensions (features)

4.6.1. NoHello

Contexts: Client → Hub

This indicates that the client doesn't need either $Hello or $NickList to be sent to it when connecting to a hub. To populate its user list, a $MyINFO for each user is enough. $Hello is still accepted, for adding bots to the user list. DC++ still sends a $GetNickList to indicate that it is interested in the user list. During login, hubs must still send $Hello after $ValidateNick to indicate that the nick was accepted.

Add NoHello to the $Supports to indicate support for this.

4.6.2. ChatOnly

Contexts: Client → Hub

This indicates that the client only support chat capabilities to allow the client to bypass hub rules (that relate to file sharing). The client should be disconnected if it sends a $Search, $ConnectoMe or $RevConnect.

Add ChatOnly to the $Supports to indicate support for this.

4.6.3. QuickList

The terms NDC and NHUB are used to denote a client or hub not featuring QuickList and QDC and QHUB for those that do. The term DC is used when the type is not yet established or of no importance. EACH and ALL signals that a message is sent N times, one message for each connected user. Also note that the terms IF, MAY, SHOULD and MUST, have the same meaning as in the internet RFC specs. Walkthrough Connecting

A DC connects to QHUB and does nothing
QHUB sends $Lock which starts with EXTENDEDPROTOCOL
QHUB also sends $HubName
A DC must send $Key
A DC may also send $Supports QuickList| to signal in is in fact a QDC
QHUB responds with $Supports QuickList|

Identification

A QDC may but should not send $Version
A QDC must send $MyINFO

Authentication

QHUB may send $GetPass
QDC responds with $MyPass

QHUB may send $BadPass and disconnect
QHUB may send $ValidateDenide and disconnect

Acceptance

QHUB may send $LogedIn to signal that H: should not be incremented
QHUB sends all clients MyINFO to the QDC
QHUB sends $OpList to the QDC
QHUB may send $Hello to all clients but should send it to non QDC only
QHUB sends $MyINFO to all QDC clients

Explanation Connecting This part has been through some changes. There was an argument of having the client start with $Supports and then having the hub respond to that. Having the client start with $Supports as a response to EXTENDEDPROTOCOL is in a sense much cleaner.

• It has been found that $HubName can be sent pretty much at any time. Typically done in conjunction with $Lock.

• $Supports are in the same format as in the client protocol; *'$Supports <feat1> <feat2> <feat3>|'*

Identification

• sending $MyPass early has been removed from the spec.

Authentication

• As previously mentioned $GetPass is only sent if the user has an account and the $MyPass was not sent in the identification process. $BadPass or $ValidateDenide is sent when proper as usual.

Acceptance

From testing it has been found that $Hello and $MyINFO can be sent together without having to wait for a $GetINFO.

• Let the hub decide if the account should be treated as a VIP, i.e. not increment H: in the tag, hence it is not mandatory for accounts. A QDC should only respond to this message for account logins only.

• $OpList was missing.

Commands Sent

Connecting • $Lock • $HubName • $Supports

Identification

Authentication • $GetPass • $BadPass • $ValidateDenide

Acceptance • $LogedIn • $MyINFO • $MyINFO stream

Connected • $To • $Search • $SR • $ConnectToMe • $RevConnectToMe • $ForceMove

Commands Accepted

Messages that a QHUB listens to in each state, QHUB ignores otherwise;

Connecting • $Key • $Supports

Identification • $Version • $MyINFO

Authentication • $MyPass

Acceptance

Connected • $GetNickList • $MyINFO • $To • $Search • $SR • $ConnectToMe • $RevConnectToMe • $Kick • $OpForceMove

Notes • $ValidateNick deprecated and ignored • $GetNickList only valid when connected, a QDC does not receive a $NickList, but a series of $MyINFOs directly. • $Hello deprecated and ignored • $GetINFO deprecated and ignored • $MyINFO is always accepted as valid and denotes a new or updated client.

4.6.4. TTHSearch

This indicates that the client support searching for queued files by TTH. See $Search for details.

Add TTHSearch to the $Supports to indicate support for this.

4.6.5. XmlBZList

Usage: Supporing this means supporting utf-8 XML file lists with the following general structure:

<FileList Version="1" Generator="dc client name and version"> <Directory Name="xxx"> <Directory Name="yyy"> <File Name="zzz" Size="1"/> </Directory> </Directory> </FileList>

In each directory, including the root, the name of the entity must be case-insensitive unique in that level of the hierarchy.

Other fields may be added as necessary. DC++ for instance adds the TTH attribute to each file it knows the TTH root of in base32 encoding.

The file list is available as "files.xml.bz2" (vs MyList.DcLst), and is compressed using bzip2.

To retrieve unicode files from the file list, the client may also support the above GetZBlock and its utf-8 derivatives. Support for XmlBZList implies support for $UGetBlock, so files are guaranteed to be retrievable.

$UGetBlock follows $UGetZBlock semantics, but without compressing the data. The <bytes> parameter of $Sending specifies how many bytes will be sent.

Don't touch Version. Add your own, with a different name, if you feel compelled.

Don't trust Generator to determine features of the file list. It's there mainly for debugging and informative purposes.

Add XmlBZList to the $Supports to indicate support for this.

DC `supported the feature XMLBZList and the commands $GetBlock, $UGetBlock and $UGetZBlock in versions 0.307 to 0.695. DC` dropped support for the commands in version 0.696, whilst not removing the feature announcement. I.e., DC++ signals in the $Supports XMLBzList while it does not support the actual commands.

4.6.6. Minislots

This allows the other client to use a free slot for small files / file list.

$Supports is needed because the Neo-Modus DC client closes the file list browser when the connection is broken, which it becomes when the client on the other side tries to download its first file and fails because it has no real slot.

Add Minislots to the $Supports to indicate support for this.

4.6.7. TTHL

Supporting this means supporting the upload of tth leaf data. Instead of transfering the file itself, the TTH data of all leaves is transferred in binary. The size transferred back is the number of bytes of leaf data, from this and the file size the receiving client can calculate which level (tree depth) the sending client is offering. The receiver should obviosly check that the received leaf data is correct by rebuilding the tree and checking that it's recorded root matches.

Add TTHL to the $Supports to indicate support for this.

4.6.8. TTHF

Supporting this means supporting file identification by TTH root. This means supporting downloads by TTH root instead of share directory and name. The advantage is that moved files can still be found by the downloader without requeuing the file.

The extension adds a namespace "TTH" before the ADC file root. A TTHF filename has the following syntax:

TTH/<TTH root in base32, 192 binary bits>

i e the TTH namespace consists of TTH root values directly under the "TTH/" root.

The naming scheme is valid in all types (i e also for getting TTH leaves)

Add TTHF to the $Supports to indicate support for this.

4.6.9. ZLIG

Supporting this means that zlib compressed $ADCGET transfers are supported.

Add ZLIG to the $Supports to indicate support for this.

4.6.10. ACTM

Advanced Connect To Me (ACTM) is a replacement for the $ConnectToMe and $RevConnectToMe commands.

Add ACTM to the $Supports to indicate support for this.

Implementations supporting ACTM must reply to incoming $ConnectToMe and $RevConnectToMe requests, but will themselves always send $CTM or $RCTM.

During requests, the clients send a 4-digit hexadecimmal ID. This ID is an incremental number that

is given out for each $CTM that it sent. When a connection between two clients is established, the other party must echo back this 4-digit ID. This is done after $Supports but before $Direction. If the 4-digit ID is not any of the unhandled IDs given out by the requesting client, it must signal "$Error Invalid ID" and disconnect.

CTM

```
Client 1 to hub: $CTM client2_nick$client1_port$id|
Hub forwarding to client 2: $CTM client1_ip$client1_port$id|
```

RCTM

```
Client 1 to hub: $RCTM client2_nick|
Hub forwarding to client 2: $RCTM client1_nick|
Client 2 to hub: $CTM client1_nick$client2_port$id|
Hub forwarding to client 1: $CTM client2_ip$client2_port$id|
```

Example handshake:

Client party	Server party
	$MyNick peter|
	$Lock EXTENDEDPROTOCOLABCABCABCABCABCABC Pk=DCPLUSPLUS0.668ABCABC|
$MyNick john|	
$Lock EXTENDEDPROTOCOLABCABCABCABCABCABC Pk DCPLUSPLUS0.668ABCABC|	
	$Supports MiniSlots XmlBZList ADCGet TTHL TTHF ACTM GetZBlock ZLIG|
	$Direction Download 17762|
	$Key ...|
$Supports MiniSlots XmlBZList ADCGet TTHL TTHF ACTM GetZBlock ZLIG|	
$CTM A91E|	
$Direction Upload 6494|	
$Key ...|	
	$Get files.xml.bz2$1|

4.6.11. NoGetINFO

This indicates that the hub doesn't need to receive a $GetINFO froma client to send out $MyINFO. This is a variation of the QuickList proposal that is easy to implement and does half of QuickList's job.

Add NoGetINFO to the $Supports to indicate support for this.

4.6.12. BZList

Signals support for BZIP2 compressed file list instead of the Huffman encoded list that NMDC pioneered. The compressed file list is available for download under the name MyList.bz2 instead of MyList.DcLst and files.xml.bz2 instead of files.xml.

Add BZList to the $Supports to indicate support for this.

4.6.13. CHUNK

This is a protocol extension by Valknut that allows retrieval of sections of a file through a modified $Get syntax. The syntax is: $Get <filename>$<start-position>$<chunk-size>|

Add CHUNK to the $Supports to indicate support for this.

4.6.14. OpPlus

Characteristics of a hub. Indicates that the hub uses additional commands for operators. For example: $Ban, $TempBan, $UnBan, $GetBanList, $WhoIP, $Banned, $GetTopic, $SetTopic and more.

4.6.15. Feed

This feature offers additional protocol commands notice. Feature allows you to track all the actions of the individual or all users benefit from logging these actions, and notify operators of the hub acts committed by a newly created chat room.

4.6.16. SaltPass

This feature offers passwords to be salted and hashed, which means that passwords are no longer sent in plaintext. This adds "random data" to the $GetPass command.

The random data should be Base32 encoded.

The data that is sent back in the $MyPass shall be the password followed by the random data, passed through the Tiger algorithm and then encoded with Base32. I.e., base32(tiger_hash(password + data)).

Add SaltPass to the $Supports to indicate support for this.

Note that the example below for $MyPass is not literal.

The algorithm here is a port from ADC's GPA/PAS.

Example

```
$GetPass data|
$MyPass base32( tiger_hash( password + data ) )|
```

4.6.17. IPv4

This feature is used to indicate IPv4 support when a client is connecting from a IPv6 address.

The hub should send a $ConnectToMe to the client, to indicate a download from the hub. The client will connect to the hub address and port as in a normal client to client connection, this time using the client's IPv4 address. The hub should disconnect after it has received the $MyNick command.

The hub should later verify in connection attempts with other IPv4 clients that the IPv4 address matches.

Once the hub has received an IPv4 connection and gathered the IPv4 address, it should set the IPv4 bit in the $MyINFO command.

Add IPv4 to the $Supports to indicate support for this.

4.6.18. IPv6

This feature is used to indicate IPv6 support.

IPv6 addresses are specified in RFC 4291 form.

If the client has an IPv4 connection, it should also signal the use of IPv4. Clients should use the same TCP and UDP ports for both IPv4 and IPv6 if both are supported. The connection mode (passive, active, SOCKS5) can be different.

The hub is required to set the IPv6 bit in the $MyINFO command.

The connection mode in the $MyINFO tag changes from "M:X" where X is the connection mode (A = Active, B = Passive, 5 = SOCKS5) to "M:XY" where X is the connection mode of IPv4 and Y is the connection mode of IPv6. If a client does not support IPv4 or the IPv4 check failed, the first character will be *N* for not supported.

Add IP64 to the $Supports to indicate support for this.

A client that support IPv4 and IPv6 will only use one form when sending messages to a hub. The hub is responsible for translating the command into the correct IPv4/IPv6 address. E.g., if a $Search is sent with a IPv6 address, the hub will send the client's IPv4 address to those who only support IPv4. This minimizes the amount of traffic toward the hub. If a client sent a passive search request, then it is only sent to active users supporting the same TCP/IP protocol. This is regardless if the client is active in the other protocol. I.e., if a passive search request is sent with a IPv4 address, that search request will only be forwarded to IPv4 users and not *converted* to an IPv6 request, regardless if the client is active in IPv6.

4.6.19. TLS

This feature is used to indicate support for TLS encrypted client-client connections.

Implementations shall add an *S* to the (TLS) port in a $ConnectToMe.

Add TLS to the $Supports to indicate support for this.

Example:

```
$ConnectToMe john 192.168.0.1:412S|
```

4.6.20. DHT

This feature is used to indicate support for Distributed Hash Tables (DHT) for client-client connections.

This feature uses ADC commands for the DHT swarm [2].

Add DHT0 to the $Supports to indicate support for this.

4.6.21. Queue position

```
$MaxedOut queue_position|
```

This feature is a support for a queue numbering system for client-client connections. See [3] for further information on the ADC equivalent.

This feature extends $MaxedOut by adding a number after it.

Note that there is no feature to announce in $Supports.

Example:

```
$MaxedOut 2|
```

4.6.22. FailOver

```
$FailOver [host[,host]]|
```

This feature is used to indicate support for providing alternative hub addresses in the event that the hub is unavailable. See [4] for further information on the ADC equivalent.

Used a comma (,) to separate hub addresses.

The command can be sent at any time by the hub, overwriting or replacing the addresses of the previous command. An empty command (no hosts) clears the list of alternative addresses.

Add FailOver to the $Supports to indicate support for this.

Examples:

```
$FailOver|
$FailOver example.com:412|
$FailOver example.com,example.org:5555,adc://example.net:6666|
```

5. Examples

5.1. Client – Hub connection

See respective command for description of command and parameters.

Client	Hub
	$Lock EXTENDEDPROTOCOL_verlihub Pk=version0.9.8e-r2|
$Supports UserCommand UserIP2 TTHSearch ZPipe0 GetZBlock|	
$Key 01101011011001010111001|	
$ValidateNick johndoe|	
	<VerliHub> This hub is running version 0.9.8e-r2 (Monday Jul 20 2009) of VerliHub (RunTime: 2weeks 1hours / Current user count: 4)|
	<VerliHub> This hub is enhanced by plugman for Verlihub.|

Client	Hub
	$Supports OpPlus NoGetINFO NoHello UserIP2 HubINFO|
	$HubName Verlihub|
	$GetPass|
$MyPass mysecurepass55555|	
	$Hello johndoe|
$Version 1,0091|	
$GetNickList|	
$MyINFO $ALL johndoe <++ V:0.673,M:P,H:0/1/0,S:2>$ $LAN(T3)0x31$example@example.com1234|	
	$LogedIn johndoe|
	$NickList BotTestNickaaaVerliHubbbbOpChat|
	<MOTD> This is the MOTD message.|
	$HubTopic The topic of the hub is not set.|
	$OpList TestNickaaabbbVerliHubOpChat$$|
	$BotList BotVerliHubOpChat$$|

5.2. Client – Client connection

Note the connecting client is the client, but that the server speaks first.

See respective command for description of command and parameters.

Client	Server
	$MyNick bbb|
$MyNick aaa|	
$Lock EXTENDEDPROTOCOLABCABCABCABCABCABC Pk=DCPLUSPLUS0.706ABCABC|	
	$Lock EXTENDEDPROTOCOLABCABCABCABCABCABC Pk=DCPLUSPLUS0.777Ref=dchub://example.org:411|
$Supports MiniSlots XmlBZList ADCGet TTHL TTHF ZLIG|	
$Direction Download 31604|	
$Key 01101011011001010111100 1|	
	$Supports MiniSlots XmlBZList ADCGet TTHL TTHF ZLIG|
	$Direction Upload 82|
	$Key 01101011011001010111100 1|
$ADCGET file files.xml.bz2 0 -1 ZL1|	
	$ADCSND file files.xml.bz2 0 29725 ZL1|

6. License

1. https://dcpp.wordpress.com/2010/03/12/dc-0-75-and-older-vulnerable-to-bzip2-filelist-bomb/
2. http://strongdc.sourceforge.net/download/StrongDC_DHT.pdf
3. http://adc.sourceforge.net/ADC-EXT.html#_qp_upload_queue_notification
4. http://adc.sourceforge.net/ADC-EXT.html#_fo_failover_hub_addresses

Version 1.3
Last updated 2013-03-09 18:28:53 W. Europe Daylight Time