Hyper-G Client/Server Protocol (HG-CSP)

Version 7.05

Written by: Frank Kappe (fkappe@iicm.tu-graz.ac.at)
Gerald Pani (gpani@iicm.tu-graz.ac.at)
December 28, 1994

Contents

1 Introduction

This specification describes the protocol between a Hyper-G client and the Hyper-G server (more specifically, the *link server* of Hyper-G). The protocol is described on a low level (i.e. the socket level), as this documentation is supposed to be used to create new Hyper-G clients (including gateways to other protocols) on arbitrary platforms and with arbitrary programming languages. It should be explicitly noted that the standard Hyper-G components (which are written in C++) usually do not bother about individual bytes sent over the network, but use a C++ class *HgStub* that is able to pass objects between clients and server. This explains why the protocol is not designed to be human-readable. It is, however, compatible with the *dispatch* library of the InterViews toolkit [5, 6].

In addition to the HyperG Client/Server protocol there is also a Server/Server protocol (used to maintain information base consistency across server boundaries), a Client/Client protocol (allows Hyper-G clients to pass messages and events to each other using the server), as well as protocols used for communication between the individual parts of the Hyper-G server (*hgserver, dbserver, ftserver, dcserver*; for an overview the reader is referred to the Hyper-G Architectural Design Document). These protocols are beyond the scope of this document.

1.1 Status of this document

The document you are currently looking at describes the Hyper-G Client/Server protocol version 7.05. An up-to-date version is available in hypertext form on the Hyper-G server of Graz University of Technology. If you prefer reading linear text on paper, a PostScript version is available by anonymous ftp from *ftp.iicm.tu-graz.ac.at* in file */pub/Hyper-G/papers/Protocol.ps.*

1.2 Glossary

This section is to define the meaning of a number of words that are used throughout this documentation.

- **anchor:** An anchor denotes a section of a document (perhaps all of it). For example, it may be a word or phrase in a text document, or a small area in a large picture. It is either the source or the destination of a link.
- **attribute:** Attributes are pieces of information attached to objects (e.g., as author, creation time, title, keywords, expiration time, type of object etc.). One can then use some of these to search the database for documents with certain attributes.
- **author:** In the Hyper-G system, the word 'author' refers to the user who entered a piece of information into the database–not necessarily the person who originally created the information. For example, in the case of a picture of the 'Mona Lisa', the author would be the person that inserted that picture into the system, rather than Leonardo da Vinci.
- **client:** The part of Hyper-G that displays documents and realizes the user interface.

- **cluster:** A specialized collection object (see below). Documents that are members of clusters should be visualized together (this is how multimedia objects are realized in Hyper-G).
- **collection:** A group of items, which may be either documents or collections. Can be compared to "directory" in a file system. This definition yields a "collection hierarchy", which looks like a tree, but with any number of "roots" and with the property that a collection may be a subcollection of several collections (- acyclic graph). For example, a biochemistry collection might be a subcollection of both a biology collection and a chemistry collection.
- **document:** A type of object that is intended to be displayed by the client. A document must be a member of at least one "parent collection", and may have a number of source and/or destination anchors attached. Hyper-G supports a number of document types, of which text documents are treated a bit special by the server (their contents are searchable), which is why in the protocol there are specialized functions that deal with text documents.
- **object:** Generic term for an entity stored in the Hyper-G database. Documents, links, anchors, and collections are examples of objects. An object has a set of attributes, each of which has one or more values. In particular, every object is assigned a unique object identifier.
- **link:** The essance of hypertext. A user may click on one endpoint of a link (the "source anchor") and jump to the other endpoint (the "destination anchor").
- **search:** All objects inserted into the Hyper-G database can automatic be searched for. The protocol supports basically two kinds of searches: Boolean searches based on attributes of objects, and full text searches that are defined for text documents only.
- **server:** The part of Hyper-G that stores the documents, links, etc.

2 References

[1] KAPPE F.: "Hyper-G: A Distributed Hypermedia System". In LEINER B. (editor), *Proc. INET '93, San Francisco, California*, pages DCC–1–DCC–9, Internet Society, August 1993.

[2] KAPPE F.: *Hyper-G Text Format (HTF)*. Technical Report, December 1993. Available by anonymous ftp from iicm.tu-graz.ac.at in file pub/Hyper-G/doc/HTF.ps.

[3] KAPPE F. and MAURER H.: "Hyper-G: A Large Universal Hypermedia System and Some Spin-Offs". *ACM Computer Graphics, experimental special online issue*, May 1993. Available by anonymous ftp from siggraph.org in directory publications/May_93_online/Kappe.Maurer.

[4] KAPPE F., PANI G., and SCHNABEL F.: "The Architecture of a Massively Distributed Hypermedia System". *Internet Research: Electronic Networking Applications and Policy*, 3(1):10–24, Spring 1993.

[5] LINTON M. A., CALDER P. R., and VLISSIDES J. M.: "The Design and Implementation of InterViews". In *Proc. USENIX C++ Workshop, Santa Fe, New Mexico*, November 1987.

[6] LINTON M. A., VLISSIDES J. M., and CALDER P. R.: "Composing User Interfaces with InterViews". *IEEE Computer*, 22(2):8–22, February 1989.

3 An Overview of the Protocol

The protocol employed is based on TCP/IP, connection-oriented, asynchronous, and (almost) stateless. The client sends messages to the server that initiate server transactions, and the server returns messages with the results of transactions and/or error codes.

Connection-oriented means that the TCP connection is not closed and re-opened between transactions, but stays open for the duration of a session.

Asynchronous means that, although usually the client will send a message to the server and wait for the return message that has the results of the requested transaction, this need not be so. A client may choose to issue a number of commands in succession, and look at the results later. Likewise, some commands do not require waiting for a response at all. Also, the server may choose to send an unsolicited message to the client at any time.

The protocol is almost stateless in the following sense: After some initial negotiation during the initialization of the connection, the available transaction commands may be issued in any sequence. The server does not maintain any state information (other than the Hyper-G database itself) or re-use (intermediate) results of previous transactions.

3.1 Opening and Closing the Connection

Opening a connection to the Hyper-G server is started by the client (of course) by opening a TCP connection to the server's well-known port number '418'.

The server then initiates a negotiation on the data format that is to be used for this connection. The Hyper-G servers are able to understand both formatted (i.e. clear text) and binary transfer modes. By default, the server will offer to do a binary connection by sending a 'T' (binary=true) character in response to the connection request. The client may also send a 'T' for binary or refuse to do so by sending an 'F' (for false). The client does not need to wait until it receives something from the server before sending this character.

When both have sent and received a 'T', binary mode is enabled. Client and server now need to agree on a certain byte order. Both send a 'B' if they would prefer big-engian, and 'l' if they would prefer little-endian byte order. If client and server receive the opposite of what they have sent, they will both perform byte swapping on their outgoing messages (incoming messages are always left unchanged).

After this negotiation is completed (and the message format to be used is thus defined) the server will send an initial `READY` message containing the server's identification parameters (see section 4.17), or an error message if the connection is refused for some reason. The client should also send a `READY` message in order to state it is willing to accept messages, and to agree on the protocol version, compression and distribution modes that are to be used (see section on message format below). Following this, messages may be sent in arbitrary order.

No special action is necessary to shut down the connection. The client may just close the TCP connection. The Hyper-G server may also drop the connection, typically after 12 hours of inactivity (i.e. no messages have been reveived from the client for 12 hours). On some servers (e.g. those that offer licensed material) this period may be significantly shorter (a few minutes).

3.2 Message Format

Once the initialization procedure is done, client and server communicate with each other by exchanging messages. The semantics of these messages are similar to remote procedure calls: The sends a message containing a server command; and, except for a few functions that do not require a response, the server then sends a message containing the response. Commands and responses are identified by a *message identifier*.

There's actually one exception to this: The server may send a `READY` response even when no `READY` command has been sent (see section 4.17).

A message is composed of the following elements:

1. **LENGTH**: An integer containing the length in characters of the message (**including** this integer). If the format has been set to BINARY (see section 3.1), this is a 32-bit unsigned integer. If the format is FORMATTED, it is sent as 10 characters, containing a right-justified string representation of a decimal number, i.e., spaces followed by a series of digits. This lets the receiver of the message know how much information to expect, and allocate appropriate buffers, etc.

2. **VERSION/MESSAGE-ID**: An integer normally used for an unique `MESSAGE-ID` except in the case of a `READY` message, were it is used for `VERSION` negotiating.

   If the format has been set to BINARY, the `VERSION/MESSAGE-ID` identifier is transmitted as a 32-bit integer. Otherwise (i.e. the format is FORMATTED) it is separated from the preceding `LENGTH` with a blank, and sent as a variable-length string representation of the decimal value (terminated by a blank).

   - **VERSION**: The `version` identifier is a 32-bit integer composed of three sub-fields:

     ```
      31 . . . 30 29 . . . 14 13 . . . 00
     +-----------+----------+-----------+
     |   flags   | reserved |  version  |
     +-----------+----------+-----------+
     ```

     The `version` field specifies the actual protocol version. The current version is 7.05. It will be incremented on major protocol releases (only those that require changes in the existing command structure; this is not necessary on minor bug fixes or new, additional commands), so that messages from old clients (servers) will recognized and can be dealt with appropriately by the server (client). The protocol version of the connection is defined as the minimum of the client and server version numbers, and is usually determined during the initial exchange of `READY` messages (see section 3.1).

     The `flags` field contains individual bits that switch on/off certain functionality. Currently, only the two most significant bits are defined (the `DISTRIBUTED` bit (31) and the `COMPRESSED` bit (30). If distributed bit is set, this means that the server should provide access to remote data (i.e. data that resides different Hyper-G servers) as well. Otherwise, only data on this server can be accessed. Setting this bit has an important effect on the ObjectIDs that are passed to the client: If `DISTRIBUTED` is not set, these are the physical Hyper-G ObjectIDs on this server, and they remain constant over sessions. If `DISTRIBUTED` is set, however, these are virtual ObjectIDs generated on the fly by the server, that cannot be re-used in future sessions. In any case, ObjectIDs are 32-bit unsigned integers. If compressed bit is set, this means that the server should send text and objects in compressed form.

The other flag bits and the *reserved* field are reserved for future use.

- **MESSAGE-ID**: The client should use this to determine which response belongs to which command. The response of a command has always the same `MESSAGE-ID` as the corresponding command. So the client could use increasing numbers to do this job.

3. **MESSAGE-TYPE**: Each message (i.e. command and response) is assigned a type. This lets the receiver of the message know which kind of message is being received, and therefore what kind of information to expect. The number for a given command is the same as the number for its response. These numbers are given in the next section.

4. **MESSAGE-DATA**: The parameters of the command (if this message is sent by the client) or the return values of the command (if sent by the server) follow. The data is composed as a sequence of strings and numbers, depending on the type of the message (command or response). Currently, only three different data types are used:

   - **Integer**: Sent as 32-bit integer, if in BINARY transmission mode. In FORMATTED mode it is sent as an optional `-` sign, followed by a series of decimal digits terminated by a space. ObjectIDs, enumerates (e.g. error codes) and boolean values are also sent as integers.
   - **String**: In both transmission modes, a string is sent as a sequence of characters, terminated by a zero byte. Consequently, a zero byte cannot be passed directly in a string.
   - **IntArray**: An array of integers may be sent by sending first an integer indicating the length of the array, and then the elements of the array, in order.

4 Description of Messages

This section contains a short description of each kind of message that may be sent, in the order of their message identifiers. The order of message identifiers assigned to messages is based on historic evolution of the protocol and of no further relevance. Some message identifiers are not assigned. These typically have been used by some previous version of the protocol and are either obsolete commands or have been replaced by a new command with another message identifier.

For each message, we provide the message identifier together with a symbolic name, the parameters sent and return values received (from the client viewpoint), a short description of what the procedure does, and possible error codes returned. Section 5 contains an overview of all error codes.

The notation *int* indicates that an integer is sent, *string* indicates that a string is sent, *int foo[]* indicates that an array of integers is sent; with the number of integers sent equal to the integer sent immediately before, except when noted otherwise. The notation *optional* indicates that the argument may not be sent (for example, if it is null). The notation *remainder* indicates that the argument fills up the rest of the message. The other notations should be obvious.

4.1 Command 2: GETDOCBYANCHOR

Send: int anchID

Recv: int error, int docID

Decription: This command returns the document ID (*docID*) of the document to which the given anchor (*anchID*) is attached to. If *anchID* is not an anchor but a document, *docID* is set to *anchID*, otherwise it is set to 0..

Errors: If *error* is not zero, an error has occured, e.g.:

RSERV_NRESP Remote server not responding.

See section 5 for a list of all possible error codes.

4.2 Command 3: GETCHILDCOLL

Send: int parentID

Recv: int error, int count, int childIDs[]

Decription: Retrieves the children (sub-collections only) of the given parent collection (*count* items). Use the GETCHILDDOCCOLL command to get the documents of the parent collection (for performance reasons, the server uses two different relations for sub-collections and sub-documents). Use the CHILDREN command to get both collections and documents.

Errors: If the parentID is not the object ID of a collection, or it contains no sub-collections, a count of 0 is returned. If *error* is not zero, an error has occured, e.g.:

RSERV_NRESP Remote server not responding.

See section 5 for a list of all possible error codes.

4.3 Command 4: GETPARENT

Send: int childID

Recv: int error, int count, int parentIDs[]

Decription: Retrieves the parents of the given childID.

Errors: If the child has no parents (i.e. it is a root collection), a count of 0 is returned. If *error* is not zero, an error has occured, e.g.:

RSERV_NRESP Remote server not responding.

See section 5 for a list of all possible error codes.

4.4 Command 5: GETCHILDDOCCOLL

Send: int parentID

Recv: int error, int count, int childIDs[]

Decription: Retrieves the children (document only) of the given parent collection (*count* items). Use the GETCHILDCOLL command to get the subcollections of the parent collection. Use the CHILDREN command to get both collections and documents.

Errors: If the parentID is not the object ID of a collection, or it contains no sub-documents, a count of 0 is returned. If *error* is not zero, an error has occured, e.g.:

RSERV_NRESP Remote server not responding.

See section 5 for a list of all possible error codes.

4.5 Command 7: GETOBJECT

Send: int objectID

Recv: int error, optional remainder objRecord

Decription: Use this command to retrieve the full object record of a given *objectID*. The object record (*objRecord*) is a string of lines that contain the object's attributes. It is sent only if *error* is 0 (OK), and fills the remainder of the message (the length can be determined by examination of the `LENGTH` element (see section 3.2).

Errors: If *error* is not zero, an error has occured, e.g.:

RSERV_NRESP Remote server not responding.

NOTFOUND The object could not be found (e.g., because the objectID was not a valid ID).

NOACCESS No read permission for the object, or it has expired, etc.

See also the security note at the OBJECTBYIDQUERY command.

4.6 Command 8: GETANCHORS

Send: int docID

Recv: int error, int count, int anchIDs[]

Decription: Returns a list of anchors that are attached to the given document (docID). There is no distinction between source and destination anchors. You need to fetch the anchor object to determine whether a given anchor is a source or destination anchor ('TAnchor=Src' or 'TAnchor=Dest').

Errors: If the given docID is not the object ID of a document, or no anchors are attached, a count of 0 is returned. If *error* is not zero, an error has occured, e.g.:

RSERV_NRESP Remote server not responding.

See section 5 for a list of all possible error codes.

4.7 Command 9: GETOBJBYQUERY

Send: string indexQuery

Recv: int error, int count, int matchIDs[]

Decription: This command is used to perform searches on the set of attributes of Hyper-G objects. It can only be applied to attributes that are indexed. Use the OBJECTBYIDQUERY command to match a set of objects returned by this command against other, non-indexed, attributes. The search is performed over the whole database of the server the client is connected to. Use the GETOBJBY-QUERYCOLL to restrict the search to parts of the collection hierarchy, and to perform distributed searches on other servers. The command returns the IDs of the matching objects, with one exception: If the object is a document that is a member of a cluster, the cluster's ID is returned instead.

Currently, the indexed attributes of Hyper-G objects are:

Attribute	valid for objects of type
Author	documents, anchors, collections
Group	user profiles
Hint	source anchors
Host	user profiles
Keywords	documents, collections
Name	collections
TimeCreated	documents, anchors, collections
TimeModified	documents, anchors, collections
Title	documents, destination anchors, collections
UName	user profiles
UGroup	user profiles
UServer	server objects

The syntax of *indexQuery* is as follows:

```
<expr> ::= "(" <expr> ")" |
           <expr> "||" <expr> |  /* OR */
           <expr> "&&" <expr> |  /* AND */
           <expr> "&!" <expr> |  /* AND NOT */
           <indexField> "=" <values> |
           <indexField> ">" <simplevalue> "<" <simplevalue>  /* BETWEEN */

<indexField> ::= "Title" | "Keyword" | "TimeCreated" |
           "TimeModified" | "Author" | "Name" | "Hint" |
           "UName" | "UGroup" | "Host" | "Host" | "Group" |
           "UServer"

<values> ::= <value> |
             <value> <values>  /* same as AND operator */

<value> ::= <simplevalue>      /* ordinary word */
            <simplevalue>"*"   /* all words with prefix simplevalue */
            <simplevalue>"^"   /* all words greater or equal to simplevalue */
```

Status: The list of indexed attributes can be extended in future versions.

Errors: If nothing is found no object IDs are returned (*count* is 0). If *error* is not zero, an error has occured, e.g.:

CMD_SYNTAX query is invalid

Q_OVERFLOW too many intermedate results

Note: When too many objects are found (i.e. more than a certain threshold value set in the server), not all possible matches will be returned (Q_OVERFLOW). This also applies to intermediate results, e.g. the query "`Title=atom && TimeCreated=93/01/01^`" will probably not be able to retrieve all documents created after Jan. 1, 1993 that contain the word "atom" in their title, even if there are only 5, when the rest of the database contains 100,000 documents created after Jan. 1, 1993. While this limitation will hopefully disappear in the future when the link server is supplied with a query optimizer, you may meanwhile circumvene this problem by first issuing this command with only the "Title=atom" query and then reducing the matches using the OBJECTBYIDQUERY command (which is a bit slower, however).

4.8 Command 10: GETOBJBYQUERYCOLL

Send: int collCount, int colls[], string indexQuery

Recv: int error, int count, int matchIDs[]

Decription: This command works like the GETOBJBYQUERY command. However, the search is not performed on the whole database of the server the client is connected to. Rather, it is limited to the set of collections supplied (*collCount, colls*) and their subcollections (recursively).

If the connection is a distributed one (i.e. the DISTRIBUTED flag has been set, see section 3.2), the command will also search remote collection (i.e. collection that physically reside on some other Hyper-G server) if they are explicitly supplied in the *colls* array. This feature can be used to perform the same query over a number of Hyper-G servers in parallel. Remote collections are not searched automatically when they only appear as some subcollection of one of the local collections specified.

Errors: If nothing is found, no object IDs are returned (*count* is 0). If *error* is not zero, an error has occured, e.g.:

RSERV_NRESP remote server not responding

CMD_SYNTAX query is invalid

Q_OVERFLOW too many intermedate results

Note 1: The limitations mentioned at the GETOBJBYQUERY command apply, with the additional problem that (partial) queries that match a large number of objects cannot be performed, even if applied only on a small subset of the collection hierarchy, until the query optimizer appears. Of course, in such a case one may first retrieve the IDs of this small subset, and then apply the OBJECTBYIDQUERY command.

4.9 Command 11: OBJECTBYIDQUERY

Send: int sendParents, int objCount, ind objIDs[], optional string query

Recv: int error, int matchCount, int matchIDs[], int offsets[], remainder records

Decription: Like GETOBJECT, this command is used to retrieve the object records (with the object's attributes) from object IDs. This command is more powerful in that it lets you retrieve a number of object records with one command. In addition, a query may be performed on the object records to narrow down the set of objects retrieved. The query is not limited to the indexed attributes, and is more powerful than the index query of the GETOBJBYQUERY[COLL] commands, but slower as it is performed directly on the attribute values.

When *sendParents* is non-zero, the command attaches a "Parents" attribute to every object record returned that specifies (one of) the parent collection(s) of that object (if applicable). *objCount* and *objIDs[]* specify the object IDs for which the object records should be returned. The syntax of *query* is as follows:

```
<expr> ::= "(" <expr> ")" |
           "!" <expr> |          /* NOT */
```

```
          <expr> "||" <expr> |  /* OR */
          <expr> "&&" <expr> |  /* AND */
          <attribute> <operator> <value>

<attribute> ::= /* any attribute name (Title, Author, DocumentType ...) */

<operator> ::= "=" |    /* equal */
               "<" |    /* less than (string compare) */
               ">" |    /* greater than (string compare) */
               "~"      /* regular expression matching */
```

The *matchCount* and *matchIDs[]* parameters return the IDs of the objects that are actually returned (see security note below), *records* contains a large string holding all object records, and *offsets[]* is an array of *matchCount* elements that holds the byte offsets of the individual object records in *records*.

Security note: It is here (and at the simple GETOBJECT command) where Hyper-G's read access rights are checked. This is because access rights to objects are contained within the *Rights* attribute, so that the object record has to be fetched from the low-level database in order to determine whether a user is allowd to access it. This is why it is normal that this call may return fewer object records than have been requested. The commands that return object IDs (e.g. GETCHILDCOLLS) always return all object IDs, even if the user is not supposed to access them. The client should be aware of this and not assume that because GETCHILDCOLLS returned a number of object IDs, they can be really retrieved with this command.

Errors: It is normal that this call may return zero object records (see note above).

4.10 Command 12: GETTEXT

Send: string objRecord

Recv: int error, optional remainder text

Decription: Use this command to retrieve the actual text document associated with the object record *objRecord*, which has been retrived before using the *GETOBJECT* or *OBJECTBYIDQUERY* command. The text will be in HTF format [2], but will contain no anchors. Alternatively, you may also use the PIPEDOCUMENT command to receive a (text) document asynchronoulsy on a second connection.

Errors: If the text could not be retrieved, *error* is set, in which case the *text* is not transmitted.

4.11 Command 14: INSDOC

Send: int parentCollID, string objRecord, optional remainder doc

Recv: int error, optional int docID

Decription: This command allows to insert a new document into the Hyper-G database. The *parentCollID* specifies the parent collection the document is to be inserted in; *objRecord* is a sequence of attribute-name, attribute-value pairs, like:

```
Type=Document
DocumentType=Image
Title="en:This is an English title"
Author=fkappe
TimeCreated=93/11/26 12:35:17
Rights=W:g tu-graz
```

This command can be used to insert new text documents (use the EDITTEXT command to change existing texts), in which case *doc* contains the text. The text must be in HTF format [2], without anchors.

On successful return (i.e. *error* is 0), *docID* is the new object ID that has been assigned for this object. For documents other than text (i.e., the *doc* is empty, this command only creates a Hyper-G object for the document. The document itself has to be inserted subsequently using the PUTDOCUMENT command.

Errors: If *error* is not zero, an error has occured, e.g.:

FATHERDEL Father collection specified does not exist (any more)

FATHNOCOLL Father collection specified is no collection.

NODOC Object Record passed is not a document.

NOLANGUAGE No or unknown language speified on *Title* attribute.

See section 5 for a list of all possible error codes.

4.12 Command 17: INSCOLL

Send: int parentCollID, remainder objRecord

Recv: int error, optional int collID

Decription: This command allows to insert a new collection into the Hyper-G database. The *parentCollID* specifies the parent collection the collection is to be inserted in; *objRecord* is a sequence of attribute-name, attribute-value pairs, like:

```
Type=Document
DocumentType=collection
Title="en:This is an English title"
Title="ge:Das ist ein deutscher Titel"
Author=fkappe
TimeCreated=93/11/26 12:35:17
```

On successful return (i.e. *error* is 0), *collID* is the new object ID that has been assigned for this collection.

Status: As the same effect can be accomplished using the INSERTOBJECT command, this command exists only for reasons of backward compatibility and is likely to disappear in future protocol versions. New implementations should use the INSERTOBJECT command.

Errors: If *error* is not zero, an error has occured, e.g.:

FATHERDEL Father collection specified does not exist (any more).

FATHNOCOLL Father collection specified is no collection.

NOLANGUAGE No or unknown language speified on *Title* attribute.

See section 5 for a list of all possible error codes.

4.13 Command 19: GETSRCSBYDEST

Send: int dest

Recv: int error, int count, int srcs[]

Decription: This command can be used to find the source anchor ids (*srcs*) of a given destination anchor id, or destination document id, when the destination is a whole document (*dest*).

Errors: If *error* is not zero, an error has occured, e.g.:

RSERV_NRESP remote server not responding

USR_BREAK break by user

See section 5 for a list of all possible error codes.

4.14 Command 22: MVCPDOCSCOLL

Send: int move, int fromID, int toID, int count, int docs[]

Recv: int error

Decription: This command allows to move or to copy a number of documents from one collection to another, or to unlink the documents from a collection.

In "move" mode, *move* is set to true, *fromID* specifies the collection the documents should be moved away from, and *toID* specifies their new parent collection. The document IDs are supplied in *docs[]* (*count* of them).

In "copy" mode, *move* is set to false, and *toID* specifies the collection the documents should be "copied" to, i.e. they are linked under this new collection as well as the old one. *fromID* is irrelevant in copy mode and should be set to 0. This command allows to attach a number of parent collections to a document.

If *toID* is set to 0 in move mode, the collections are "unlinked" from the collection given in *fromID*, but not moved anywhere. If *fromID* was the only parent collection of a given document, it is deleted from the database.

Errors: If *error* is not zero, an error has occured, e.g.:

FATHNOCOLL Source collection specified is no collection.

DESTNOCOLL Destination is not a collection.

SRCEQDEST Source and destination collection are identical.

NODOCS No documents supplied.

See section 5 for a list of all possible error codes.

4.15 Command 23: MVCPCOLLSCOLL

Send: int move, int fromID, int toID, int count, int colls[]

Recv: int error

Decription: Behaves like MVCPDOCSCOLL, but moves/copies/unlinks collections instead of documents. However, if in "unlink" mode *fromID* was the only parent collection of a given collection, it is deleted only if it is not empty.

Errors: See MVCPDOCSCOLL. In addition, the error code NOTEMPTY may occur.

4.16 Command 24: IDENTIFY

Send: int encrypt, string name, string passwd, optional string clientInfo

Recv: string userData

Decription: This command is used to identify the user to the Hyper-G database. This command must be used before any of the commands that modify the database can be used (otherwise, the NOACCESS error will occur when such a command is used). By default, the user is identified as "anonymous".

The *name* specifies the Hyper-G user name, while *passwd* is the password. If *encrypt* is set to true, the client supplies *passwd* in encrypted form (according to modified DES, same as UNIX passwords), otherwise it is passed in clear text. The *clientInfo* is used to transmit the type of client (for statistics). The server will return the object id of the users UserObject and the name of the user itself (e.g. "0x0000000 anonymous", if the identification failed).

Errors: none.

4.17 Command 25: READY

Send: (nothing)

Recv: optional remainder message

Decription: The READY command is rather special. It is send by the client after the initialization sequence (see section 3.1) to tell the server that the client is ready to receive messages, to agree on the protocol version to be used and to pass additional information concerning the protocol (see section 3.2). The client should NOT wait for a response.

An important difference to the other commands is that the server may send a READY response at any time (i.e. without the client sending a READY command first). This feature is used to distribute asynchronous events to the client.

Immediately after the initialization process (section 3.1) the server sends a message containing the string "`$ServerString$`serverstring" where *serverstring* is replaced by a string identifying the server (this string can be set by the server provider to identify the server, and is intended to be human-readable, e.g. "University of Some City CWIS"). This is currently the only message format defined. See also the COMMAND command.

Status: It is very likely that the set of defined message formats will be extended in the future. In particular, there will be messages that inform the client that certain objects in the database have been updated, and messages that can be used to pass clear-text information to users (e.g. "Server shutdown in 5 minutes").

Errors: none.

4.18 Command 26: COMMAND

Send: int respond, string command

Recv: optional remainder response

Decription: This (meta-)command is used to send commands to the Hyper-G server that are not directly related to manipulating objects. Rather, the (sub-)commands modify the server's state or retrieve administrative information from the server. Most of the commands do not require waiting for a response. In this case, *respond* should be set to false if asynchronous behaviour is wanted. The following commands are defined (in alphabetical order):

- **contwrite:** Enable write operations on the Hyper-G database (after a "stopwrite" command). Can only be issued by a user with system privileges. No response is returned.
- **kill [id]:** Kill (disconnect) connection with ID *id* (a number obtained using the "who" command). Can only be issued by a user with system privileges. No response is returned.
- **message [id] [string]:** Send an arbitrary message (contained in *string*) to connection *id*, which is either a user number obtained using the "who" command, or the string "ALL" in which case the message is broadcast to all users connected. Can only be issued by an identified user. No response is returned.
- **stat:** Return server statistics. Currently, the local server time, server start time, uptime in seconds, and the number of object records retrieved in the last minute, 15 minutes, 60 minutes, and since server start are returned.
- **ftstat:** Return fulltext server statistics. Currently, the server start time, uptime in seconds, and the number of queries in the last minute, 15 minutes, 60 minutes, and since server start are returned.
- **dcstat:** Return document server statistics. Currently, the server start time, uptime in seconds, and the number of text, image and other documents retrieved in the last minute, 15 minutes, 60 minutes, and since server start are returned.
- **stopserver:** Shut down the server. Can only be issued by a user with system privileges. No response is returned.
- **stopwrite:** Disable write operations on the database, e.g. during a backup. Users performing such operations will get a WRITESTOPPED error (see section 5). Use "contwrite" to enable write access afterwards. Can only be issued by a user with system privileges. No response is returned.
- **who:** Return a list of users currently connected to the server. For every user, a line with user number, user name, client host, login time, and time since the last transaction is returned. The line associated with this connection has an asterisk in the first column.

Errors: none.

4.19 Command 27: CHANGEOBJECT

Send: int objectID, remainder commands

Recv: int error

Decription: This command allows to remove, add, or modify individual attributes of an object record. The object is specified by *objectID*; commands adhere to the following syntax:

```
<command> ::= <remcmd> |
              <addcmd> |
              <remcmd> "\" <addcmd>

<remcmd>  ::= "rem " <attribute> "=" <value>

<addcmd>  ::= "add " <attribute> "=" <value>
```

Note that in order to delete or remove an attribute its old value has to be supplied (some attributes are allowed more than once). The form `attr=value\add attr=value` allows to modify attributes in one operation. The object should have been locked before using the GETANDLOCK command, and unlocked afterwards using UNLOCK.

Errors: If *error* is not zero, an error has occured, e.g.:

NOACCESS No permission to modify object.

CMDSYNTAX Error in command syntax.

CHANGEBASEFLD Change of base attribute (i.e. attributes that cannot be changed, like *ObjectID* or *Type*).

NOTREMOVED Attribute not removed (e.g. because the value was wrong).

FLDEXISTS Attribute already exists and may exist only once.

NOLANGUAGE No or unknown language supplied (*Title* attribute).

See section 5 for a list of all possible error codes.

4.20 Command 28: EDITTEXT

Send: int docID, string oldPath, string author, remainder text

Recv: int error

Decription: Use this command to replace an old HTF text with a new one. The *text* must be in HTF format [2], without anchors. The *docID* parameter specifies the ID of the document, which must already exist. The *oldPath* parameter specifies the old path variable of the document (this is verified against what is specified within the object record to ensure that nobody else has updated the document in the meantime). The *author* parameter is obsolete.

Status: The *author* parameter may be removed in future protocol versions.

Errors: If *error* is not zero, an error has occured, e.g.:

NOACCESS No permission to modify object.

NOTFOUND The *docID* was not found.

EXIST Somebody else has already modified the document in the meantime (*path* wrong).

LOCKED The document is locked by somebody else.

FTINDEX The full text indexer failed in indexing the document.

4.21 Command 29: GETANDLOCK

Send: int objID

Recv: int error, optional remainder objRecord

Decription: Similar to GETOBJECT: returns object record from object ID (provided that access permissions are OK). However, the object is also marked as "locked" in the database. That means that another user trying to use GETANDLOCK on the same object would fail (see "Errors"), until the object has been unlocked again (with the UNLOCK command), or the connection of the user that locked the object is closed.

Clients must use this mechanism to implement write-locks, to ensure that no two users can modify the same object at the same time. Specifically, the CHANGEOBJECT and NEWTEXT commands have to be surrounded by GETANDLOCK and UNLOCK. Unlike GETOBJECT, GETANDLOCK guarantees that the attributes returned are the current state of the object and cannot be modified by somebody else between operations.

Errors: If *error* is not zero, an error has occured, e.g.:

NOACCESS No permission to access object.

LOCKED Object is locked by another user; try again later.

See section 5 for a list of all possible error codes.

4.22 Command 30: UNLOCK

Send: int objID

Recv: (nothing)

Decription: Removes the lock flag from object *objID* after a previous GETANDLOCK operation on that object. The lock flag is also removed when the connection that locked the object is closed. Note that this command does not return a response.

Errors: none.

4.23 Command 31: INCOLLECTIONS

Send: int retColls, int oidCount, int oids[], int collCount, int colls[]

Recv: int error, int count, int returnIDs[]

Decription: A client may issue this command to check whether a set of objects (documents or collections) specified by *oids[]* is part of the collection hierarchy defined by *colls[]*. When *retColls* is false, the subset of *oids[]* that is part of the collection hierarchy (i.e. the documents or collections that are children of one ore more collections of *colls[]* or their subcollections, recursively) is returned in *returnIDs[]*. When *retColls* is true, however, the set of collections that have one or more objects of this subset as children are returned in *returnIDs*. This option allows a client to e.g. highlight the part of the collection hierarchy that contains the matches of a previous query, in a graphical overview.

Errors: It is normal that *returnIDs[]* may be empty.

4.24 Command 32: INSERTOBJECT

Send: string objRecord, string params

Recv: int error, optional int objID

Decription: Allows to insert an arbitrary object into the Hyper-G database. *objRecord* is the object record to insert, while *params* allows to specify parameters that are not part of the object record, e.g. the parent collection, if applicable.

The format of *params* depends on the type of object that is to be inserted. For anchors, a "Doc=0x12345678" value (with 0x12345678 replaced by the hexadecimal representation of the document ID of the document the anchor should be attached to) is required. For documents or collections, "Parent=0x12345678" denotes the object's parent collection. For link objects, "Src=0x12345678" specifies the source anchor, while an optional "Dest=0x12345678" denotes the destination (if not present the link is left open).

On successful return (i.e. *error* is 0), *objID* is the new object ID that has been assigned for this object.

Status: The params for links are likely to be rendered obsolete in new protocol versions, as the information currently stored in the link object will be stored directly in the source anchor in the future.

Errors: If *error* is not zero, an error has occured, e.g.:

FATHERDEL Father collection specified does not exist (any more)

FATHNOCOLL Father collection specified is no collection.

NOLANGUAGE No or unknown language speified on *Title* attribute.

See section 5 for a list of all possible error codes.

4.25 Command 33: INCOLLSCLUSTER

Send: int oidCount, int oids[], int collCount, int colls[]

Recv: int error, int count, int returnIDs[]

Decription: This command works similar to INCOLLECTIONS with the *retColls* parameter set to false (see section 4.23), with one exception: If the an object of *oids[]* is a document that is a member of a cluster, the cluster's ID is returned instead.

Errors: It is normal that *returnIDs[]* may be empty.

4.26 Command 34: GETOBJBYFTQUERY

Send: string query

Recv: int error, int count, int oids[], float weights[]

Decription: This command is used to perform searches on the contents of text documents stored in the Hyper-G server. Text documents that are inserted into the Hyper-G database are automatically indexed and ready for searching.

The search is performed over the whole database of the server the client is connected to. Use the GETOBJBYFTQUERYCOLL to restrict the search to parts of the collection hierarchy, and to perform distributed searches on other servers. The command returns the IDs of the matching text document, as well as a set of weights that can be used for ranking.

The syntax of *query* is as follows (expressions in ’[]’ are optional, ’’ means 0 or more occurences): means ”or”:

```
<expr> ::= <term> { <orop> <term> }

<term> ::= <factor> { <andop> <factor> }

<factor> ::= <node> [ "{" <float> "}" ]

<node> ::= <word> | "(" <expr> ")"

<orop> ::= "||" | "|"

<andop> ::= "&&" [ <optionlist> ]  |  "&" [ <optionlist> ]

<optionlist> ::= "[" <option> { "," <option> } "]"

<option> ::= "F" | "f"
```

The optionlist is extensible, to foresee proximity searches in the future.

Errors: When no objects are found, count is set to 0 and no objects and weights are returned.

4.27 Command 35: GETOBJBYFTQUERYCOLL

Send: int collCount, int colls[], string query

Recv: int error, int count, int oids[], float weights[]

Decription: This command works like the GETOBJBYFTQUERY command. However, the search is not performed on the whole database of the server the client is connected to. Rather, it is limited to the set of collections supplied (*collCount, colls*) and their subcollections (recursively).

If the connection is a distributed one (i.e. the DISTRIBUTED flag has been set, see section 3.2), the command will also search remote collection (i.e. collection that physically reside on some other Hyper-G server) if they are explicitly supplied in the *colls* array. This feature can be used to perform the same query over a number of Hyper-G servers in parallel. Remote collections are not searched automatically when they only appear as some subcollection of one of the local collections specified.

Errors: If nothing is found, no object IDs are returned (*count* is 0).

4.28 Command 36: PIPEDOCUMENT

Send: int docID, int port, string host, optional string special

Recv: int error

Decription: This command is used to actually retrieve a document from the Hyper-G server. In order to not block the command connection (i.e. the connection these commands are sent over) by sending large chunks of data (e.g. images) and to allow parallel receiving of a number of documents (e.g. movie and sound), the document is sent over a second connection.

The client needs to have opened the connection before and to listen on it. The internet address of the client host and the port number of the connection are passed in *host* and *port*. The Hyper-G server (more specifically, the document cache server) will contact the client on the specified port and send the document over. Note that the server returns the error code as soon as possible. It does not wait until the document is transmitted. The *special* is used to send some special information to the document server. Currently only "Refno=0x12345678" is used, which tells the document server to send a header in front of the document.

While text documents may be retrieved using this command, it is also possible to retrieve them over the command connection using the GETTEXT command.

Errors: If *error* is not zero, an error has occured, Errors may either come from the database (greater than 0 and less than 512) or from the document server (greater than 1024)

Possible database errors are:

NOTFOUND the *docID* does not exist

WRGTYPE either the document is not a local Hyper-G document (e.g. a remote gopher document), or its path variable is empty.

Possible document cache server errors are:

NOCONNECTION the connection to the document server is broken

CONNECTCLIENT the document server could not connect to the client

SERVERTIMEOUT the document server did not respond for a certain amount of time

OPENSTORE the given document was not found in the internal store

OPENFILE the given document was supposed to come from an external file which could not be opened

4.29 Command 37: DELETEOBJECT

Send: int objID

Recv: int error

Decription: Deletes the object associated with *objID*. Depending on the type of the object, other objects may be deleted as well (e.g. anchors attached to a document). If the object is a document, the preferred way is to use the MVCPDOCCOLLS command to remove it from a specific collection, because DELETEOBJECT removes it from all collections (even those that you do not have permission to see, e.g. another user's bookmark collection), provided that you have write permission on the object.

Errors: If *error* is not zero, an error has occured, e.g.:

NOACCESS No permission to delete object.

NOTEMPTY Attempt to delete non-empty collection.

See section 5 for a list of all possible error codes.

4.30 Command 38: PUTDOCUMENT

Send: int docID, int port, string host, string path, optional string special

Recv: int error

Decription: This command is the opposite of PIPEDOCUMENT. It allows to transmit a document to the server, using a secondary data connection. The client needs to open an internet port, listen on it, and pass the communication parameters (*host, port*) as parameters of this command. The server will then contact the client and retrieve the document. The document (i.e. the object record describing it) must have been created before. The *path* parameter specifies the path (i.e. the relative file name) under which the document should be stored on the server side (see status). The *special* is used to transmit some special information to the document server (currently unused).

Note: Text documents should NOT be created with this command. Rather, they should be created with the NEWTEXT command, which ensures that they get treated properly (e.g. passed to the full text server for indexing).

Status: The *path* parameter will disappear in future protocol versions.

Errors: If *error* is not zero, an error has occured.Errors may either come from the database (greater than 0 and less than 512) or from the document server (greater than 1024)

Possible database errors are:

NOTFOUND the *docID* does not exist

WRGTYPE either the document is not a local Hyper-G document (e.g. a remote gopher document), or its path variable is empty.

Possible document cache server errors are:

NOCONNECTION the connection to the document server is broken

CONNECTCLIENT the document server could not connect to the client

SERVERTIMEOUT the document server did not respond for a certain amount of time

OPENSTORE_READ the given document was not found in the internal store

OPENFILE the given document was supposed to come from an external file which could not be opened

4.31 Command 39: GETREMOTE

Send: string object

Recv: int error, remainder text

Decription: This command is used to retrieve documents originating from a non-native (i.e. not Hyper-G) server. The document is identified by the object record in the Hyper-G object syntax. Currently supported protocols are http and gopher.

A sample remote object:

```
ObjectID=0x00000000
Type=Document
Title=??:Libraries of the University of Minnesota Integrated Network Access
Sequence=20
DocumentType=Remote
Protocol=gopher
Host=pubinfo.ais.umn.edu
Port=0
GopherType=T
```

where only the fields 'Type=', 'DocumentType=', 'Protocol=', 'Host=', 'Port=' are required.

Errors: If *error* is not zero, an error has occured. Errors can only come from the document cache server inside the Hyper-G server complex, and have numbers larger than 1024.

NOCONNECTION the connection to the document server is broken

SERVERTIMEOUT the document server did not respond for a certain amount of time

INVALIDOBJECT error in the object

CONNECTREMOTE the document server could not connect to the non-native server

4.32 Command 40: GETREMOTECHILDREN

Send: string object, optional string query

Recv: int error, int count, int offsets[], remainder records

Decription: This command is used to retrieve "children" of documents originating from a non-native (i.e. not Hyper-G) server. It works similar to the GETREMOTE command and accepts an additional query string. The resulting documents (e.g., a gopher directory listing) are transformed into Hyper-G objects and returned.

Errors: If *error* is not zero, an error has occured. Errors can only come from the document cache server inside the Hyper-G server complex, and have numbers larger than 1024.

NOCONNECTION the connection to the document server is broken

SERVERTIMEOUT the document server did not respond for a certain amount of time

INVALIDOBJECT error in the object

CONNECTREMOTE the document server could not connect to the non-native server

4.33 Command 41: PIPEREMOTE

Send: string host, int port, string object, int refno

Recv: int error

Decription: This command is used to retrieve documents originating from a non-native (i.e. not Hyper-G) server. The document is identified by the object record in the Hyper-G object syntax. Currently supported protocols are http and gopher.

The command works similar to the PIPEDOCUMENT command in that the document is transmitted over a connection different from the connection the commands are sent over.

The client issuing the command must have opened an internet port prior to that. The address of the client's host and the port nummber are passed as *host* and *port*. The document server then will connect there and send the document. The *object* contains an object as described in GETREMOTE. Since multiple documents may be retrieved in parallel, the client may want to know how to associate the documents (i.e. the connections pending on its port) and the requests made. To help from this the client may specify the optional *refno*, which is then sent along with the document inside a header (amongst some other information about the document, such as the size).

Errors: If *error* is not zero, an error has occured. Errors can only come from the document cache server inside the Hyper-G server complex, and have numbers larger than 1024.

NOCONNECTION the connection to the document server is broken

CONNECTCLIENT the document server could not connect to the client

SERVERTIMEOUT the document server did not respond for a certain amount of time

INVALIDOBJECT error in the object

CONNECTREMOTE the document server could not connect to the non-native server

4.34 Command 42: HG_BREAK

Send: (nothing)

Recv: (nothing)

Description: This command is used to break the currently active command.

4.35 Command 43: HG_MAPID

Send: int servId, int oId

Recv: int error, int virtOId

Description: This command is used to create a virtual object id from a global object id (given by servId and oid).

4.36 Command 44: CHILDREN

Send: int parentId

Recv: int error, int count, int children[]

Decription: Retrieves the children (collections and documents) of the given parent collection (*count* items).

Errors: If the parentID is not the object ID of a collection, or it contains no children, a count of 0 is returned. If *error* is not zero, an error has occured, e.g.:

RSERV_NRESP Remote server not responding.

See section 5 for a list of all possible error codes.

5 Error Codes

The error codes returned by the Hyper-G Server are divided into three groups, corresponding to the server module that generated the error:

1. Error codes between 1 and 511 are link server errors.
2. Error codes between 512 and 1023 are full text server errors.
3. Error codes between 1024 and 2047 are document cache server errors.

An error code of 0 means that no error occurred.

The following table shows the link server error codes with their associated mnemonic names and a short (human-readable) description that could be output on a terminal, for example.

Code	Mnemonic	Description
1	NOACCESS	Access denied
2	NODOCS	No documents?
3	NONAME	No collection name
4	NODOC	Object is not a document
5	NOOBJ	No object received
6	NOCOLLS	No collections received
7	DBSTUBNG	Connection to low-level database failed
8	NOTFOUND	Object not found
9	EXIST	Collection already exists
10	FATHERDEL	Father collection disappeared
11	FATHNOCOLL	Father collection not a collection
12	NOTEMPTY	Collection not empty
13	DESTNOCOLL	Destination not a collection
14	SRCEQDEST	Source equals destination
15	REQPEND	Request pending
16	TIMEOUT	Timeout
17	NAMENOTUNIQUE	Name not unique
18	WRITESTOPPED	Database now read-only; try again later
19	LOCKED	Object locked; try again later
20	CHANGEBASEFLD	Change of base-attribute
21	NOTREMOVED	Attribute not removed
22	FLDEXISTS	Attribute exists
23	CMDSYNTAX	Syntax error in command
24	NOLANGUAGE	No or unknown language specified
25	WRGTYPE	Wrong type in object
26	WRGVERSION	Client version too old
27	CONNECTION	No connection to other server
28	SYNC	Synchronization error
29	NOPATH	No path entry
30	WRGPATH	Wrong path entry
31	PASSWD	Wrong password (server-to-server server authentication)
32	LC_NO_MORE_USERS	No more users for license"
33	LC_NO_MORE_DOCS	No more documents for this session and license
34	RSERV_NRESP	Remote server not responding
35	Q_OVERFLOW	Query overflow
36	USR_BREAK	Break by user
37	N_IMPL	Not implemented

The following table shows the full text server error codes with their associated mnemonic names and a short (human-readable) description that could be output on a terminal, for example.

Code	Mnemonic	Description
513	FTCONNECT	No connection to fulltext server
514	FTTIMEOUT	Connection timed out
515	FTINDEX	Something wrong with fulltext index
516	FTSYNTAX	Query syntax error

The following table shows the document cache server error codes with their associated mnemonic names and a short (human-readable) description that could be output on a terminal, for example.

Code	Mnemonic	Description
1024	OK	no error
1025	REQUESTPENDING	request pending
1026	NOCONNECTION	no connection to document server
1027	WRONGVERSION	wrong protocol version
1028	NOTINITIALIZED	not initialized
1029	BADREQUEST	bad request
1030	BADLRN	bad document number
1031	OPENSTORE_WRITE	cannot write to local store
1032	OPENSTORE_READ	cannot read from local store
1033	READSTORE	store read error
1034	WRITESTORE	write error
1035	CLOSESTORE	close error
1036	BADPATH	bad path
1037	NOPATH	no path
1038	OPENFILE	cannot open file
1039	READFILE	cannot read from file
1040	WRITEFILE	cannot write to file
1041	CONNECTCLIENT	could not connect to client
1042	ACCEPT	could not accept connection
1043	READSOCKET	could not read from socket
1044	WRITESOCKET	could not write to socket
1045	(unused)	
1046	TOOMUCHDATA	received too much data
1047	TOOFEWDATA	received too few data
1048	(unused)	
1049	NOTIMPLEMENTED	not implemented
1050	USERBREAK	user break
1051	INTERNAL	internal error
1052	INVALIDOBJECT	invalid object
1053	JOBTIMEOUT	job timed out
1054	OPENPORT	cannot open port
1055	NODATA	received no data
1056	NOPORT	no port to handle this request
1057	NOTCACHED	document not cached
1058	BADCACHETYPE	bad cache type
1059	OPENCACHE_WRITE	cannot write to cache
1060	OPENCACHE_READ	cannot read from cache
1061	NOSOURCE	do not know what to read
1062	CLOSECACHE	could not insert into cache
1063	CONNECTREMOTE	could not connect to remote server
1064	LOCKREFUSED	lock refused

A An example C client

This appendix contains a simple C program that opens a connection to a Hyper-G server, performs a "command who" command, and prints the result (i.e. the list of users currently connected to the server). It uses a library of low-level C functions that encapsulate most of the client/server communication. The header file (hg_comm.h) and source code (hg_comm.c) is listed after hg_who.c.

```
/*******************************************************************
 *                                                                 *
 * hg_who.c - print who is logged in to the Hyper-G server right now. *
 *                                                                 *
 * Usage: hg_who [server] [port]                                   *
 *                                                                 *
 *******************************************************************/

#include <stdio.h>
#include <sys/param.h>

#include "hg_comm.h"

#define SERVER_STRING    "$ServerString$"

extern int version;
char server_string[256];

int port = 0;

extern void print_msg(hg_msg *msg, char *str, int txt);

main(int argc, char *argv[])
{
     int  sockfd;
     char server_host[MAXHOSTNAMELEN];
     hg_msg *msg;
     int len;

     if ( argc >= 2 )
          strcpy(server_host, argv[1]);
     else
          gethostname(server_host, sizeof(server_host));

     if ( argc == 3 )
          port = atoi(argv[2]);

     if ( (sockfd = open_hg_connection(server_host, port)) < 0 )  {
          fprintf(stderr, "open_hg_connection returned -1\n");
          return(-1);
     }

     version = VERSION;
     /* version = VERSION | F_DISTRIBUTED | F_COMPRESSED; */

     if ( (msg = initialize_hg_connection(sockfd)) == NULL )  {
```

```
          fprintf(stderr, "initialize_hg_connection returned NULL\n");
          close(sockfd);
          return(-1);
     }
     if (msg->version_msgid & F_VERSION < VERSION) {
          fprintf(stderr, "Warning: used version %d less than %d\n",
                  msg->version_msgid & F_VERSION, VERSION);
     }
     version = msg->version_msgid;
     len = strlen(SERVER_STRING);
     strncpy(server_string, (msg->buf)+len, 255);
     server_string[255] = '\0';

     free(msg->buf);
     free(msg);

     fprintf(stdout, "\n%s\n", server_string);

     if ( send_command(sockfd, WHO_COMMAND) == -1 )
          fprintf(stderr, "send_command (who) returned -1\n");
     else
          msg = recv_command(sockfd);
     if ( msg != NULL )  {
          print_msg(msg, "WHO_COMMAND", 1);
          free(msg->buf);
          free(msg);
     }
     else
          fprintf(stderr, "recv_command (who) returned NULL\n");

     close_hg_connection(sockfd);

     exit(0);
}

void print_msg(hg_msg *msg, char *str, int txt)
{
     char *ptr;
     int  i;

     fprintf(stdout, "\nprint_msg: >>%s<<\n", str);
     fprintf(stdout, "print_msg: length  = %d\n", msg->length);
     fprintf(stdout, "print_msg: msgid = %d\n", msg->version_msgid);
     fprintf(stdout, "print_msg: msg_type  = %d\n", msg->msg_type);
     if ( msg->length > HEADER_LENGTH )  {
          ptr = msg->buf;
          for ( i = 0; i < msg->length-HEADER_LENGTH; i++ )  {
               if ( *ptr == '\n' )
                    fprintf(stdout, "%c", *ptr++);
               else if ( iscntrl(*ptr) )
                    fprintf(stdout, ".", *ptr++);
               else
                    fprintf(stdout, "%c", *ptr++);
          }
```

```
    }
    fprintf(stdout, "\n\n");
}
```

```
/*********************************************************************
 *                                                                   *
 * hg_comm.h - a simple C interface to Hyper-G that lets you send and *
 *            receive commands.                                      *
 *                                                                   *
 *********************************************************************/

#ifndef hg_comm_h
#define hg_comm_h

#define HG_SERVER_PORT   418

#define READY_MESSAGE    25
#define COMMAND_MESSAGE  26

#define F_DISTRIBUTED 0x80000000
#define F_COMPRESSED  0x40000000
#define F_VERSION     0x00003fff
#define VERSION       705L     /* 7.05 */

#define HEADER_LENGTH 12

#define STAT_COMMAND_STR  "stat"
#define WHO_COMMAND_STR   "who"
#define STAT_COMMAND      1
#define WHO_COMMAND       2

typedef struct  {
     int  length;
     int  version_msgid;
     int  msg_type;
     char *buf;
} hg_msg;


extern int  open_hg_connection(char *server_name, int port);
extern void close_hg_connection(int sockfd);
extern hg_msg *initialize_hg_connection(int sockfd);

extern int send_ready(int sockfd);
extern int send_command(int sockfd, int command);

extern hg_msg *recv_hg_msg(int sockfd);
extern hg_msg *recv_ready(int sockfd);
extern hg_msg *recv_command(int sockfd);

#endif
```

```
/*******************************************************************
 *                                                                 *
 * hg_comm.c - a simple C interface to Hyper-G that lets you send and *
 *            receive commands.                                    *
 *                                                                 *
 *******************************************************************/

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/param.h>
#include <errno.h>
#include <malloc.h>
#include <stdlib.h>

#include "hg_comm.h"

static int set_nonblocking(int fd);
static int set_blocking(int fd);

static int hg_read_exact(int sockfd, char *buf, int size);
static int hg_read(int sockfd, char *buf, int size);
static int hg_write(int sockfd, char *buf, int size);

static int   send_hg_msg(int sockfd, hg_msg *msg, int length);
static void  build_msg_header(hg_msg *msg, int length, int version_msgid, int msg_type);
static char *build_msg_int(char *buf, int val);
static char *build_msg_str(char *buf, char *str);
static int   swap(int val);


int version      = VERSION;
/* int version   = VERSION | F_DISTRIBUTED | F_COMPRESSED; */
static int msgid =  0;
static int sock_flags   = -1;
static int non_blocking =  0;
static int do_swap      =  0;


int open_hg_connection(char *server_name, int port)
{
     int sockfd;
     struct sockaddr_in server_addr;
     struct hostent *hp;
     char   server_host[MAXHOSTNAMELEN];

     if ( (hp = gethostbyname(server_name)) == NULL )  {
          perror("gethostbyname");
          return(-1);
```

```
    }

    bzero((char *)&server_addr, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    if ( port != 0 )
         server_addr.sin_port = htons(port);
    else
         server_addr.sin_port = htons(HG_SERVER_PORT);
    bcopy(hp->h_addr, (char *) &server_addr.sin_addr, hp->h_length);

    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 )  {
         perror("socket");
         return(-1);
    }

    if (connect(sockfd, (struct sockaddr *) &server_addr, sizeof(server_addr)) < 0) {
         perror("connect");
         close(sockfd);
         return(-1);
    }

    if ( (sock_flags = fcntl(sockfd, F_GETFL, 0)) == -1 )
         perror("open_hg_connection: F_GETFL");

    if ( set_nonblocking(sockfd) == -1 )  {
         fprintf(stderr, "open_hg_connection: set_nonblocking");
         return(-1);
    }

    return(sockfd);
}


hg_msg *initialize_hg_connection(int sockfd)
{
    char buf, c;
    hg_msg *ready_msg;
    int i = 0x01;
    int len;

    buf = 'T';
    if ( hg_write(sockfd, &buf, 1) == -1 )  {
         fprintf(stderr, "initialize_hg_connection: hg_write (1) returned -1\n");
         return(NULL);
    }

    if ( hg_read_exact(sockfd, &buf, 1) == -1 )  {
         fprintf(stderr, "initialize_hg_connection: hg_read (1) returned -1\n");
         return(NULL);
    }
    if ( buf == 'F' )  {
         fprintf(stderr, "initialize_hg_connection: got >>F<< from server\n");
         return(NULL);
    }
```

```
    if ( buf != 'T' )  {
        fprintf(stderr, "initialize_hg_connection: neither >>F<< nor >>T<< from server\n");
        return(NULL);
    }

    buf = c = ( *(char *)&i )  ? 'l' : 'B';
    if ( hg_write(sockfd, &buf, 1) == -1 )  {
        fprintf(stderr, "initialize_hg_connection: hg_write (2) returned -1\n");
        return(NULL);
    }
    if ( hg_read_exact(sockfd, &buf, 1) == -1 )  {
        fprintf(stderr, "initialize_hg_connection: hg_read (2) returned -1\n");
        return(NULL);
    }
    if ( c != buf )  {
        fprintf(stderr, "initialize_hg_connection: swapping necessary\n");
        do_swap = 1;
    }

    if ( (ready_msg = recv_ready(sockfd)) == NULL )  {
        fprintf(stderr, "initialize_hg_connection: recv_ready returned NULL\n");
        return(NULL);
    }

    return(ready_msg);
}


static int set_nonblocking(int fd)
{
    if ( sock_flags == -1 )
        sock_flags = fcntl(fd, F_GETFL, 0);
    if ( fcntl(fd, F_SETFL, O_NONBLOCK) == -1 )
        return(-1);
    non_blocking = 1;
    return(0);
}


static int set_blocking(int fd)
{
    if ( fcntl(fd, F_SETFL, sock_flags) == -1 )
        return(-1);
    return(0);
}


static int hg_read_exact(int sockfd, char *buf, int size)
{
    int try = 0;
    int len = 0;

    if ( !non_blocking )
        set_nonblocking(sockfd);
```

```
     while ( size > 0 )  {
          len = recv(sockfd, (void *) buf, size, 0);
          if ( len == -1 )  {
               if ( (errno == EAGAIN) || (errno == EWOULDBLOCK) )  {
                    if ( ++try > 5 )
                         return(-1);
                    sleep(3);
               }
               else return(-1);
          }
          else  {
               size -= len;
               buf  += len;
               try   = 0;
          }
     }

     return(0);
}


static int hg_read(int sockfd, char *buf, int size)
{
     int try = 0;
     int len = 0;

     if ( !non_blocking )
          set_nonblocking(sockfd);
     while ( len == 0 )  {
          len = recv(sockfd, (void *) buf, size, 0);
          if ( len == -1 )  {
               if ( (errno == EAGAIN) || (errno == EWOULDBLOCK) )  {
                    if ( ++try > 5 )
                         return(-1);
                    sleep(3);
               }
               else return(-1);
          }
     }
     return(len);
}


static int hg_write(int sockfd, char *buf, int size)
{
     int try = 0;
     int len = 0;

     if ( !non_blocking )
          set_nonblocking(sockfd);
     while ( size > 0 )  {
          len = send(sockfd, (void *) buf, size, 0);
          if ( len == -1 )  {
               if ( (errno == EAGAIN) || (errno == EWOULDBLOCK) )  {
```

```
                if ( ++try > 5 )
                     return(-1);
                sleep(3);
           }
           else return(-1);
      }
      else  {
           size -= len;
           buf  += len;
           try   = 0;
      }
   }
   return(0);
}


hg_msg *recv_hg_msg(int sockfd)
{
     hg_msg *msg;

     if ( (msg = (hg_msg *)malloc(sizeof(hg_msg))) == NULL )  {
          perror("recv_hg_msg");
          return(NULL);
     }

     if ( hg_read_exact(sockfd, (char *)&(msg->length), 4) == -1 )  {
          fprintf(stderr, "recv_hg_msg: hg_read (1) returned -1\n");
          free(msg);
          return(NULL);
     }

     if ( hg_read_exact(sockfd, (char *)&(msg->version_msgid), 4) == -1 )  {
          fprintf(stderr, "recv_hg_msg: hg_read (2) returned -1\n");
          free(msg);
          return(NULL);
     }

     if ( hg_read_exact(sockfd, (char *)&(msg->msg_type), 4) == -1 )  {
          fprintf(stderr, "recv_hg_msg: hg_read (3) returned -1\n");
          free(msg);
          return(NULL);
     }

     if ( msg->length > HEADER_LENGTH )  {
          if ( (msg->buf = (char *) malloc(msg->length-HEADER_LENGTH)) == NULL )  {
               perror("recv_hg_msg");
               free(msg);
               return(NULL);
          }
          if ( hg_read_exact(sockfd, msg->buf, msg->length-HEADER_LENGTH) == -1 )  {
               fprintf(stderr, "recv_hg_msg: hg_read (4) returned -1\n");
               free(msg->buf);
               free(msg);
               return(NULL);
```

```
          }
     }
     else
          msg->buf = NULL;

     return(msg);
}


hg_msg *recv_ready(int sockfd)
{
     hg_msg *ready_msg;

     if ( (ready_msg = recv_hg_msg(sockfd)) == NULL )  {
          fprintf(stderr, "recv_ready: recv_hg_msg returned NULL\n");
          return(NULL);
     }
     if ( ready_msg->msg_type != READY_MESSAGE )  {
          fprintf(stderr, "recv_ready: recv_hg_msg returned wrong message\n");
          return(NULL);
     }

     return(ready_msg);
}


hg_msg *recv_command(int sockfd)
{
     hg_msg *comm_msg;

     if ( (comm_msg = recv_hg_msg(sockfd)) == NULL )  {
          fprintf(stderr, "recv_command: recv_hg_msg returned NULL\n");
          return(NULL);
     }
     if ( comm_msg->msg_type != COMMAND_MESSAGE )  {
          fprintf(stderr, "recv_command: recv_hg_msg returned wrong message\n");
          return(NULL);
     }

     return(comm_msg);
}


static int send_hg_msg(int sockfd, hg_msg *msg, int length)
{
     char *buf, *tmp;

     if ( length < HEADER_LENGTH )  {
          fprintf(stderr, "send_hg_msg: bad msg\n");
          return(-1);
     }

     if ( (tmp = buf = (char *)malloc(length)) == NULL )  {
          perror("send_hg_msg");
```

```
          return(-1);
     }

     memcpy(tmp, (char *) &(msg->length), 4);
     tmp += 4;
     memcpy(tmp, (char *) &(msg->version_msgid), 4);
     tmp += 4;
     memcpy(tmp, (char *) &(msg->msg_type), 4);
     if ( msg->length > HEADER_LENGTH )  {
          tmp += 4;
          memcpy(tmp, msg->buf, length-HEADER_LENGTH);
     }

     if ( hg_write(sockfd, buf, length) == -1 )  {
          fprintf(stderr, "send_hg_msg: hg_write returned -1\n");
          free(buf);
          return(-1);
     }

     free(buf);
     return(0);
}


int send_ready(int sockfd)
{
     hg_msg ready_msg;

     build_msg_header(&ready_msg, HEADER_LENGTH, version, READY_MESSAGE);
     ready_msg.buf = NULL;

     if ( send_hg_msg(sockfd, &ready_msg, HEADER_LENGTH) == -1 )  {
          fprintf(stderr, "send_ready: send_hg_msg returned -1\n");
          return(-1);
     }

     return(0);
}


int send_command(int sockfd, int command)
{
     hg_msg comm_msg;
     char *comm_str, *tmp;
     int  respond = 1;
     int  length;

     if ( command == STAT_COMMAND )
          comm_str = STAT_COMMAND_STR;
     else
          comm_str = WHO_COMMAND_STR;
     length = HEADER_LENGTH + sizeof(respond) + strlen(comm_str) + 1;

     build_msg_header(&comm_msg, length, msgid, COMMAND_MESSAGE);
```

```
    if ( (comm_msg.buf = (char *)malloc(length-HEADER_LENGTH)) == NULL )  {
        perror("send_command");
        return(-1);
    }


    tmp = build_msg_int(comm_msg.buf, respond);
    tmp = build_msg_str(tmp, comm_str);


    if ( send_hg_msg(sockfd, &comm_msg, length) == -1 )  {
        fprintf(stderr, "send_ready: send_hg_msg returned -1\n");
        free(comm_msg.buf);
        return(-1);
    }

    free(comm_msg.buf);
    return(0);
}


static void build_msg_header(hg_msg *msg, int length, int version_msgid, int msg_type)
{
    if ( do_swap )  {
        msg->length  = swap(length);
        msg->version_msgid = swap(version_msgid);
        msg->msg_type  = swap(msg_type);
    }
    else  {
        msg->length  = length;
        msg->version_msgid = version_msgid;
        msg->msg_type  = msg_type;
    }
}


static char *build_msg_int(char *buf, int val)
{
    int tmp;

    tmp = do_swap ? swap(val) : val;
    memcpy(buf, (char *)&tmp, 4);

    return(buf+4);
}


static char *build_msg_str(char *buf, char *str)
{
    int len = strlen(str)+1;

    memcpy(buf, str, len);
```

```
    return(buf+len);
}


static int swap(int val)
{
    int tmp;

    ((char*)&tmp)[0] = ((char*)&val)[3];
    ((char*)&tmp)[1] = ((char*)&val)[2];
    ((char*)&tmp)[2] = ((char*)&val)[1];
    ((char*)&tmp)[3] = ((char*)&val)[0];

    return(tmp);
}


void close_hg_connection(int sockfd)
{
    shutdown(sockfd, 2);
    close(sockfd);
}
```