Introduction

The Postgres system normally operates by means of exchanging messages over a TCP stream between a client program (typically naive about the internal format of the database files) and a backend, which performs the actual database operations.

Many messages are passed in both directions using a common format: a capital letter, followed by 4 bytes of binary data (usually a transaction ID), followed by an ascii (human-readable) string terminated with a newline character. We'll call this the simple message format, the capital letter the type-id, the 4 bytes of data the tx-id, and the part up to the newline, the message body.

Simple Exchanges

The simplest exchange happens when an append, create, replace, delete, destroy, or retrieve into an alternate class is performed, where no instance data needs to be transferred from the backend to the client. The frontend sends a request using the simple message format with type-id ‘‘Q’’, and the query itself as the message body (much as a user would have typed it to the ‘‘monitor’’ program). (The difference is that newlines typed to the ‘‘monitor’’ program are actually translated to spaces by the monitor before the query is executed, whereas the query string after the ‘‘Q’’ needs neither a newline or a space.)

The backend replies with a simple message with type-id ‘‘C’’, with the name of the command as the message body.

As the backend operates, it may send advisory messages to the frontend, which may be sent to debugging files or ignored. These are transmitted as simple messages with type-id ‘‘N’’. Advisory messages do not terminate the operation of the command, and the frontend must loop through these messages. Another type of advisory message, ‘‘R’’ for a remark, used to be generated by the backend, but is no longer used.

The normal mode of operation is for a client to initiate the query and suspend further operation pending completion of the command, by calling the routine PQexec(). PQexec() does a blocking read from the data stream, parses the messages, and processes notices as they arrive.

(If an alternative asynchronous interface were constructed so that the thread of control in the client were to continue past the point of the call to the backend, and there were a great many notices generated, then the backend could become blocked waiting for the frontend to clear out the communications channel, potentially resulting in deadlock.)

The backend may also encounter some problem in the execution of the command severe enough to abort further processing; in this case it transmits a simple message with type-id ‘‘E’’, followed by the error message to be printed as the message body.

While processing one query, a backend may receive advice of an asynchronous event associated with the ‘‘Notify’’ command. This is a simple message with type-id ‘‘A’’, and the message body giving a portal name, from which the frontend can retrieve additional data. ‘‘Notify’’ and ‘‘Listen’’ are not yet implemented, so currently the frontend will generate an error message if it receives an ‘‘A’’ from the backend.

Queries returning data and portals.

If the client does a retrieve returning tuple data into a portal, then a fetch command will transfer data from the backend into the client using the protocol described below. The same protocol is used for retrieves returning tuple data with no portal specified, by use of the default portal ‘‘blank’’.

The backend first transmits a simple message with type-id ‘‘P’’ naming the portal to be used, as the message body. The next message may begin with a ‘‘T’’, ‘‘B’’, or ‘‘D’’, in addition to the other simple message types described above (errors, warnings, or async portal notification, completion messages). Messages of type ‘‘T’’, ‘‘B’’, and ‘‘D’’ do not use the simple message format.

A type ‘‘T’’ message denotes the beginning of a tuple type description. It is followed by a 2 byte integer giving the number of fields in the tuple. Each field has the attribute name (terminated with a newline), followed by 4 bytes of its abstract data-type id, followed by two bytes describing the size of the object.

A type “B” message indicates the transmission of an actual tuple as binary data. This would be generated by a sequence

RETRIEVE IPORTAL foo (...)
FETCH 5 foo

The first thing after the “B” is a bit mask indicating which fields are present (non-null). For each field which is present, there are 4 bytes to give the length of the data, followed by the data itself.

A type “D” message denotes the beginning of an actual tuple, described as ASCII strings instead of binary data, such as the result of either a RETRIEVE or a RETRIEVE PORTAL. Here too, the “D” is followed by a bitmap to describe which fields are present in this tuple. For each field, there are four bytes describing the length of the data for this field, this time including the four bytes of length (as opposed to type “B”, above). The data fields may contain embedded newlines or non-ASCII data without confusing protocol operation since each field is described by a byte-count.

The Copy Command.

Additional message types and complexity may be associated with the copy command, in the case of copying to or from stdout or stdin. The implementation of the command violates the normal model of a request being transmitted and the front end blocking until completion.

The cases of copy-to and copy-from behave differently. In copy-to, the frontend transmits the copy command as a “Q” type simple message (as it was sent with PQexec()). If the backend replies with a simple message of type “D”, the routine PQexec() knows that it may return to the application for it to transmit the data to the backend. The data stream is self describing in either binary or normal mode, and the application then calls PQendcopy(), which will block waiting for an acknowledgement. The acknowledgement is in the form of a single byte “Z” from the backend.

In copy-from, the backend transmits a simple message of type “B”, so that PQexec() may return control to the application code invoking it, which will then retrieve the data from the common TCP connection directly. As in the copy-to case, the data is self describing, and the application should consume all of it. The data will then be followed by a single byte “Z”, which is checked and consumed by calling PQendcopy() as above.

Remote Large−Object Access and the Fastpath Protocol.

The 4.1 version of the Postgres Reference Manual does not guarantee support of complete Fastpath functionality; however, the same mechanism is used for invoking access to the Large−Object system, which is supported by 4.1.

The message used for invoking a function is an extension of the simple message of type “F”, with a function ID in addition to the transaction ID, followed by four bytes of expected return value length, followed by four bytes specifying the number of arguments to the function. For each argument to the function, the size of the argument is transmitted as a 4 byte integer, followed by the data itself. Integers are expanded to 4 byte quantities. Ascii strings have a special length value, and are terminated by a newline (and must not have an embedded newline).

The backend always replies with a message of type “V” acknowledging the function call. In the case where the function returns no values, the message continues with an ASCII “0”, terminating the message (without the concluding newline that most simple messages have). In the case where a value is returned, the message continues instead with a function id, which is transmitted instead of a transaction ID. Then a “G” is followed by four bytes containing the length of the returned result and the result itself. The frontend library enforces that the length of the returned result may not be longer than the length the user indicated could potentially be returned. If the user has declared that the function returns an int, 4 bytes are retrieved from the connection. If the returned length is the magic cookie value (VAR_LENGTH_RESULT), a newline terminated string is drained from the connection (with the newline replaced by a null character). Otherwise, the indicated number of bytes are transferred from the datastream to the user's supplied buffer.

Backend and Frontend Files.

The backend files which use this protocol are:

tcop/dest.c:

BeginCommand()
EndCommand()
SendCopyBegin()
ReceiveCopyBegin()
NullCommand()

tcop/fastpath.c:

SendFunctionResult()
HandleFunctionResult()

access/common/printtup.c:

printtup()
printtup_internal()

utils/error/elog.c:

elog(va_list)

The frontend files which use this protocol are:

libpq/fe-pqexec.c:

process_portal()
read_remark()
PQfn()
PQfsread()
PQfswrite()
PQexec()
PQendcopy()

libpq/fe-dumpdata.c:

dump_type()
dump_tuple()
dump_tuple_internal()
finish_dump()
dump_data()

User-Level Response

Some libpq routines return information codes to the application program. These codes are similar to those which are used for communication between the frontend and backend, but some of the codes have different meanings. The following describes those frontend routines called by the user which return codes.

PQfn(): This routine returns ''G'' if there is a return value in the result_buf passed in, and ''V'' if there is no return value. NULL is returned on error.

PQexec(): Returns ''E'' on a fatal error (the backend died) and ''R'' on a non-fatal error (such as that generated by a backend elog(WARN)). If the user's query generated no return value, PQexec returns a ''C<command>'', where the command is the one the user sent. ''BCOPY'' means that the copy command began successfully, and the DBMS is sending data to the user's application. ''DCOPY'' means a copy into the database began successfully, and the DBMS is waiting to receive data from the user's application. ''I'' means that the user sent an empty query. The backend transmits any notifies (part of the asynchronous protocol) and then sends ‘‘I’’ to indicate it is done.