NEON

The wire protocol

Heikki Linnakangas

NEON

History

- Current protocol is called “protocol version 3”
- Protocol version 3 was introduced in server version 7.4
- Version 2 support was dropped in server version 14

Message structure

- All communication is through a stream of *messages*.
- The first byte of a message identifies the message type, and the next four bytes specify the length of the rest of the message (this length count includes itself, but not the message-type byte).
- The remaining contents of the message are determined by the message type.

```
Q 12 SELECT 1
```

Message type (1 byte)

Message length (4 bytes)

Message content (depends on the message type)

Opening a connection

NEON

Handshake

Client sends the first message after opening the TCP connection

Can be one of:

- The “startup packet”
- SSLRequest
- GSS encryption request
- Query cancel packet
- TLS hello (since version 17)

NEON

The TLS handshake

- Client sends magic 4 bytes forming an SSLRequest
- Server responds with ‘Y’ or ‘N’
- If ‘Y’, the client then proceeds with TLS handshake
- If ‘N’, the client can continue without encryption, or hang up

- Starting with v17, the client can also start the TLS handshake immediately, without the SSLRequest. If the server does not support TLS, it will close the connection.

The startup packet

- After establishing TLS (if wanted), the client sends so-called *startup packet*
- The startup packet contains:
  - Protocol version
  - List of supported protocol extensions
  - Database name
  - User name
  - Optional settings that will be set to GUCs after startup
  - Typically, application_name

Authentication

The authentication consists of a number of challenge-response messages

The server sends one of the Authentication request messages, and the client responds:

- `AuthenticationCleartextPassword`
- `AuthenticationMD5Password`
- `AuthenticationSASL, AuthenticationSASLContinue, AuthenticationSASLFinal (for SCRAM)`
- `AuthenticationKerberosV5`
- `AuthenticationGSS, AuthenticationGSSContinue`
- `AuthenticationSSPI`

Once authentication is accepted, the server sends an `AuthenticationOK` message

Simplest possible handshake

Client

-> Startup packet

Server

<- AuthenticationOK

NEON

Protocol negotiation

- If the server does not support the minor protocol version requested by the client, it sends a `NegotiateProtocol` message with the highest version that it does support.
- The client can choose to continue with that version, or hang up
- The same applies to any protocol extensions that the server does not support

More complicated case

Client	Server
-> GSSRequst	
	<- ‘N’
-> SSLRequest	
	<- ‘Y’
-> TLS ClientHello	
	<- TLS ServerHello
-> TLS ChangeCipherSet	
	<- TLS ChangeCipherSet
-> Startup packet	
	<- NegotiateProtocol <- AuthenticationSASL
-> SASLInitialResponse	
	<- AuthenticationSASLContinue
-> SASLResponse	
	<- AuthenticationSASLComplete <- AuthenticationOK

Almost there..

After authentication, the server will send:

- `BackendKeyData`: this contains the “query cancellation key” that will be needed to perform query cancellation later. The client saves it somewhere
- `ParameterStatus`: Report current values of certain GUCs
- `ReadyForQuery`

Connection has now been established

- The server enters the normal query handling loop
- The client can now start sending queries

Running queries

NEON

Running queries

Two ways:

A) Simple query protocol
   - Supports “multi-statements”, i.e “SELECT ‘foo’; SELECT ‘bar’”

B) Extended query protocol
   - Query parameters, prepared statements, cursors

Simple query protocol

Client

-> Q SELECT * FROM table

Server

<- RowDescription
<- DataRow
<- DataRow
<- CommandCompletion: SELECT 2
<- ReadyForQuery

Extended query protocol

Client

-> Parse SELECT * FROM tbl WHERE id = $1

-> Bind 1234

-> Describe

-> Execute

Server

<- RowDescription

<- DataRow

<- DataRow

<- CommandCompletion: SELECT 2

<- ReadyForQuery

NEON

Parse

- The Parse message comes in two variants:
  - unnamed variant
  - named variant
- The unnamed variant is used for executing one-off queries
- The named variant creates a prepared statement that can be reused
  - These can also be created with at SQL level with the PREPARE statement

Bind

- The Bind message includes:
  - the prepared statement name (or empty string for the unnamed prepared statement)
  - Destination portal name (or empty string for the unnamed *portal*)
  - values for any query parameters
  - Whether to use binary or text format for each result column

- Creates a named *portal*, aka. *Cursor*
  - This shares the namespace with cursors declared at SQL level with DECLARE CURSOR and with cursors created e.g. in pl/pgsql functions
  - unnamed portal should used for one-off executions

Describe

Comes in two variants:

- Portal Describe
- Statement Describe

- The server will respond with a RowDescription message, with information about the columns and datatypes in the result set
- In statement variant, the server also sends a ParameterDescription message, with information about the query parameters

Execute

- Runs the portal, returns rows
- Can include a max. row count
  - You can fetch more by sending another Execute message

Synchronization

Client

-> Parse

-> Bind

-> Execute

-> **Sync**

Server

<- DataRows

<- CommandCompletion: SELECT 2

<- **ReadyForQuery**

NEON

Synchronization

- After each logical statement, the client sends a ***Sync*** message
- The server will buffer responses and doesn't send anything back until it sees a Sync or the buffer fills up
- The server can send an ErrorMessage at any time, and the connection enters "error mode" where any subsequent queries to fail too
- Sync closes the implicit transaction

NEON

Running multiple queries in one “logical statement”

Client

-> Parse SELECT ‘foo’ +Bind+Execute

-> Parse SELECT ‘bar’ +Bind+Execute

-> **Sync**

Server

<- DataRows

<- CommandCompletion: SELECT 1

<- DataRows

<- CommandCompletion: SELECT 1

<- **ReadyForQuery**

Running multiple queries in one “logical statement”

Client	Server
-> Parse SELECT ‘foo’ +Bind+Execute	
-> **Flush**	
	<- DataRows
	<- CommandCompletion: SELECT 1
-> Parse SELECT ‘bar’ +Bind+Execute	
-> **Sync**	
	<- DataRows
	<- CommandCompletion: SELECT 1
	<- **ReadyForQuery**

COPY protocol

NEON

COPY in and COPY out modes

COPY mode is started by sending a COPY command:

COPY in, client -> server:

```
COPY tbl FROM STDIN
```

COPY out, server -> client:

```
COPY tbl TO STDOUT
```

COPY in, client -> server

Client	Server
-> Q COPY mytbl FROM STDIN	
	<- CopyInResponse
-> CopyData	
-> CopyData	
-> CopyDone	
	<- CommandCompletion: COPY 123
	<- **ReadyForQuery**

NEON

COPY mode

- The server responds with CopyInResponse or CopyOutResponse
- At the end, the sender sends a CopyDone or CopyFail message to exit the copy mode
- The server can send an ErrorMessage at any time. The COPY mode still needs to be terminated with a CopyFail message
- After CopyDone, the server responds with CommandCompletion message, like with a normal query

NEON

CopyData messages

- CopyData messages are just a stream of data
- The content depends on the COPY format options
  - Text. CSV, BINARY, DELIMITER, ESCAPE and so forth
- In COPY out mode (server -> client), each CopyData message contains one row
- In COPY in mode (client -> server), the client is free to chunk the data as it wishes

Replication protocol

NEON

Replication protocol

- If you use the `replication` option in the startup packet, you open a replication connection instead of a regular one
- The wire protocol is the same
- Instead of SQL queries, there is a set of “replication commands” that you can run:
  - IDENTIFY_SYSTEM
  - TIMELINE_HISTORY
  - CREATE_REPLICATION_SLOT
  - START_REPLICATION

NEON

Physical Replication protocol

- The START_REPLICATION command enters COPY **both** mode
- Like COPY in/out modes, but both sides can send CopyData messages
- A nested protocol inside the wire protocol, each message is send as a CopyData message, with a nested message type header and payload

Messages include:

- XLogData
- Keepalive messages
- Hot standby feedback messages

NEON

Logical Replication protocol

- Started with the START_REPLICATION command
- Also a nested protocol using COPY mode

Messages include e.g.:

- Begin
- Insert/Update/Delete
- Commit

Cancellation

NEON

Query cancellation

If you hit CTRL-C in psql, for example, the client initiates “query cancellation”

- Query cancellation is performed by opening *another TCP connection*
- Instead of sending a startup packet, the client sends a CancelRequest message, with the secret token that it got from the server when the connection was established
- Can be TLS encrypted

So you want to build a connection pooler?

NEON

NEON

Connection state

- Prepared statements
- Cursors
- Current transaction
- SET variables
- Caches
- Cancellation

NEON

Protocol extendability

NEON

Two mechanisms

Minor version negotiation

- Current version is 3.0. If we introduce version 3.1, client and server can fall back to the lowest common supported version

Protocol extensions

- The startup packet can contain list of supported extensions, and can fall back to set of extensions supported by client and server

NEON

Protocol negotiation is currently untested

- There is only one minor version, 3.0
- There are no protocol extensions

If you're writing a pooler or server that uses the Postgres wire protocol, please implement the protocol negotiation to be future proof!

NEON

Two patches in progress that will extend the protocol

- Add new protocol message to change GUCs to be able to change protocol extension parameters by Jelte Fennema-Nio
- Make query cancel keys longer by Heikki Linnakangas

Thank you!

Questions?

Tip: Wireshark has built-in support for parsing the Postgres protocol