Postgres on the wire

A look at the PostgreSQL wire protocol

Jan Urbański
j.urbanski@wulczer.org

Ducksboard

PGCon 2014, Ottawa, May 23

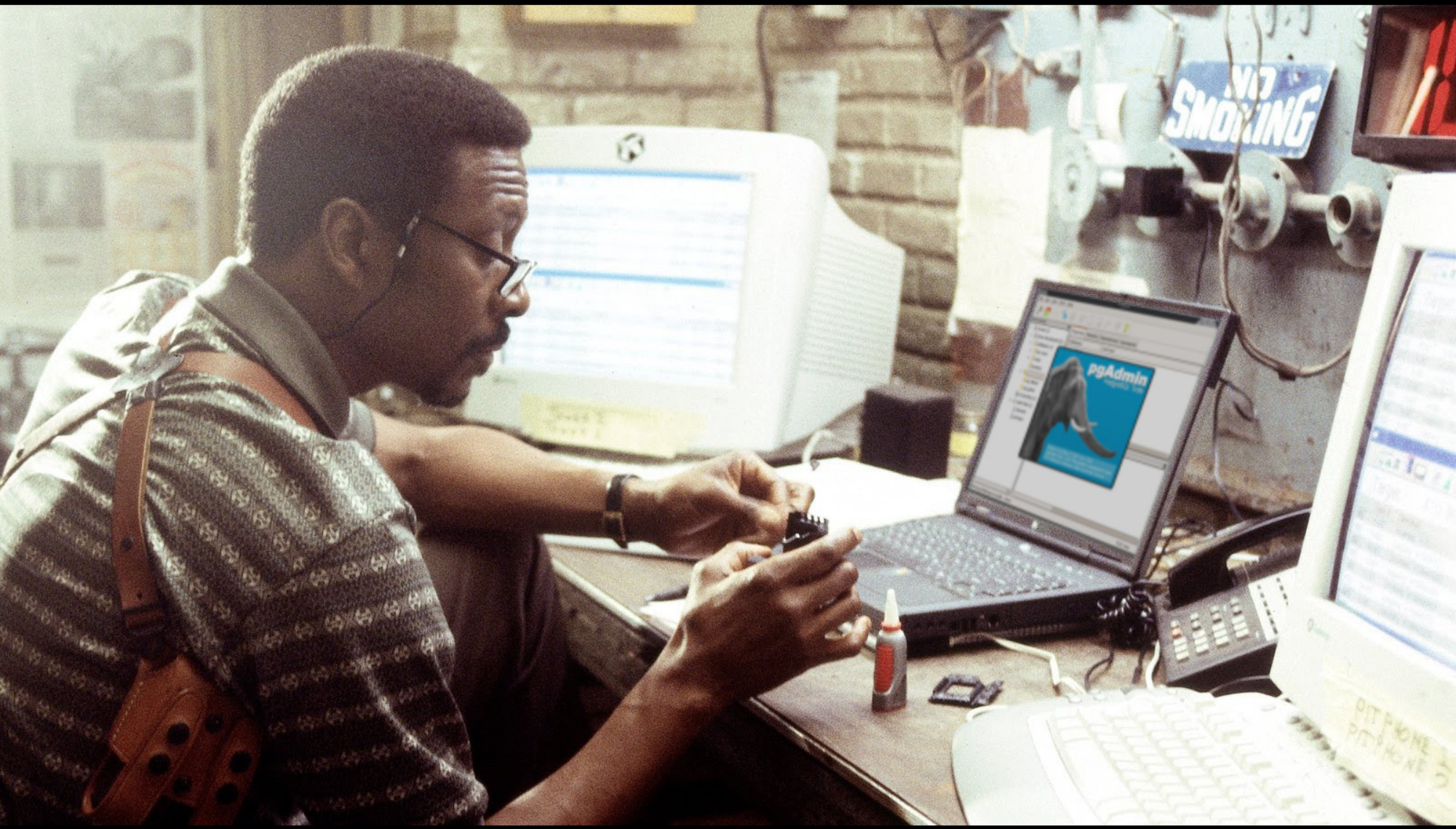
For those following at home

Getting the slides

```
$ wget http://wulczer.org/postgres-on-the-wire.pdf
```

Getting the source

```
$ https://github.com/wulczer/postgres-on-the-wire
```


NO
SMOKING
pgAdmin

1. Protocol basics
   - Frame format
   - Message flow
2. Sending queries
   - Simple protocol
   - Extended protocol
3. Other features
   - The COPY subprotocol
   - Less known FEBE features
   - Future development

Outline

Protocol versions

- the 2.0 protocol got introduced in 6.4, around 1999
  - protocol versioning got added in the previous release
- the 3.0 got introduced in 7.4, in 2003
- the server still supports protocol 1.0!
- 3.0 has some new features
  - extended query protocol
  - COPY improvements
  - overall better frame structure

Handling incoming connections

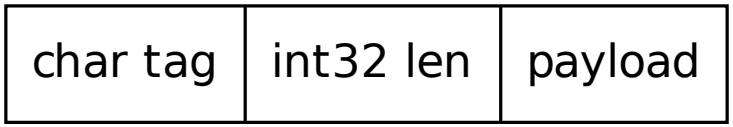
Connections are received by the postmaster process, which immediately forks a new process to deal with them.

- any parsing issues won't affect the postmaster
- authentication is done after a process is forked
- closing the connection results in terminating the backend
  - but the backend needs to notice that first
  - killing the client might not terminate the running query

FEBE frame format

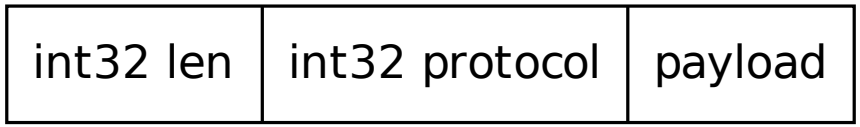
Virtually all messages start with an ASCII identifier, followed by length and payload.

Regular packet



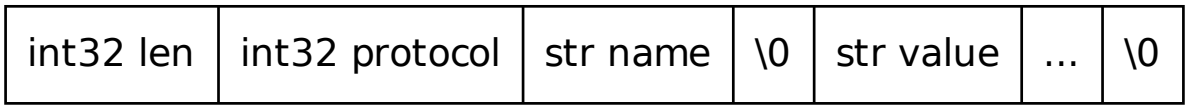
The exception is the startup packet, which starts with the length followed by the protocol version.

Startup packet



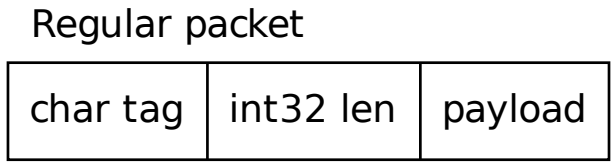
Startup packet

Startup packet



- the very first bit of data received by the backend is parsed as the startup packet
- starts with a 32 bit protocol version field
- in protocol 2.0 it had a fixed length, in 3.0 it's variable length
- what follows is a list of key/value pairs denoting options
  - some keys, like `user`, `database` or `options` are special
  - the rest are generic GUC options

Regular data packet



- starts with an ASCII identifier
- a 32 bit message length follows
  - this means you can't send a query that's larger than 1 GB
- interpretation of the payload depends on the identifier

Outline

Authentication

AuthenticationRequest

'R'	int32 len	int32 method	optional other

- if a connection requires authentication, the backend will send a AuthenticationRequest
- there are several authentication types that can be demanded
  - plain-text or MD5 password
  - it's up to the server to require plain text or encrypted
  - GSSAPI, SSPI
- if no auth is necessary, the server sends AuthenticationOK

Encrypted password exchange

AuthenticationRequestMD5

'R'	int32 len	int32 method	char[4] salt

The MD5 AuthenticationRequest message includes a 4 byte salt.

$$pwdhash = md5(password + username).hexdigest()$$
$$hash = 'md5' + md5(pwdhash + salt).hexdigest()$$

- using a salt prevents replay attacks
- double-hashing allows the server to only store hashes

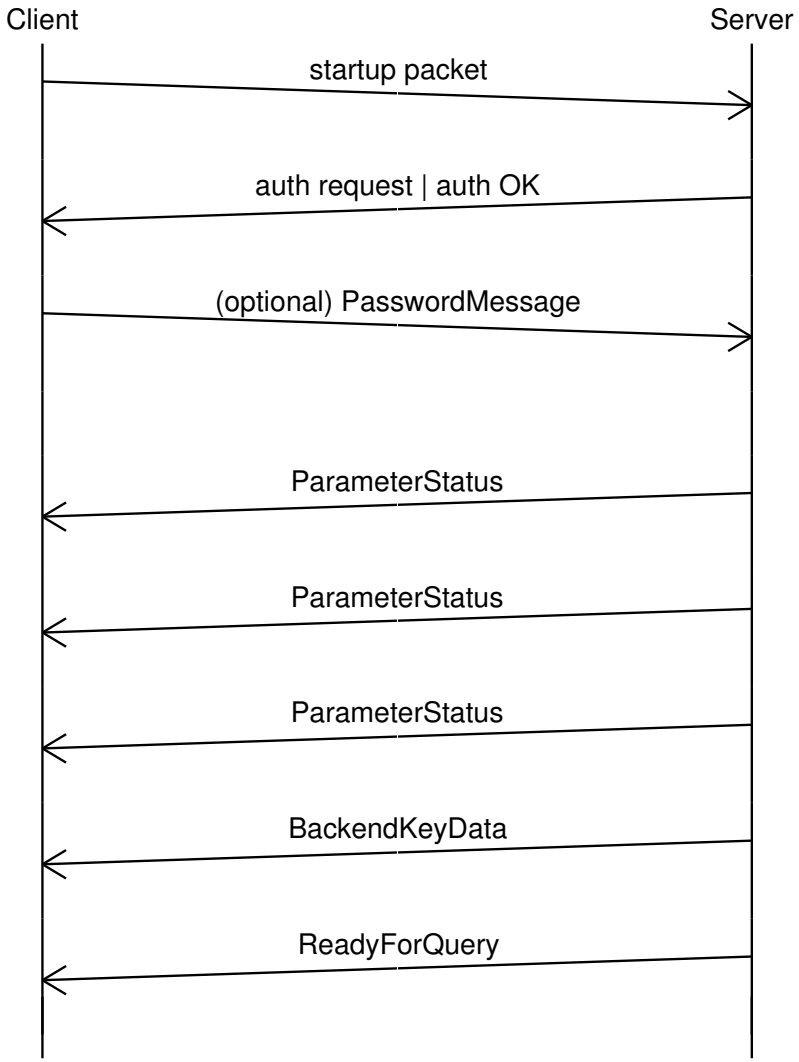
Parameter status

ParameterStatus

'S'	int32 len	str name	str value

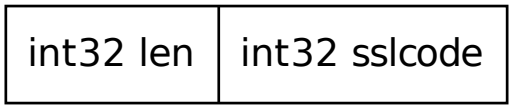
- the server notifies clients about important parameters
- first batch of ParameterStatus messages is sent on startup
  - some of them are informative, like `server_version`
  - others are critical for security, like `client_encoding`
  - others yet are important for the client, like `DateStyle`
- when any of those parameters gets set, the server notifies the client on the next occasion

Basic message flow



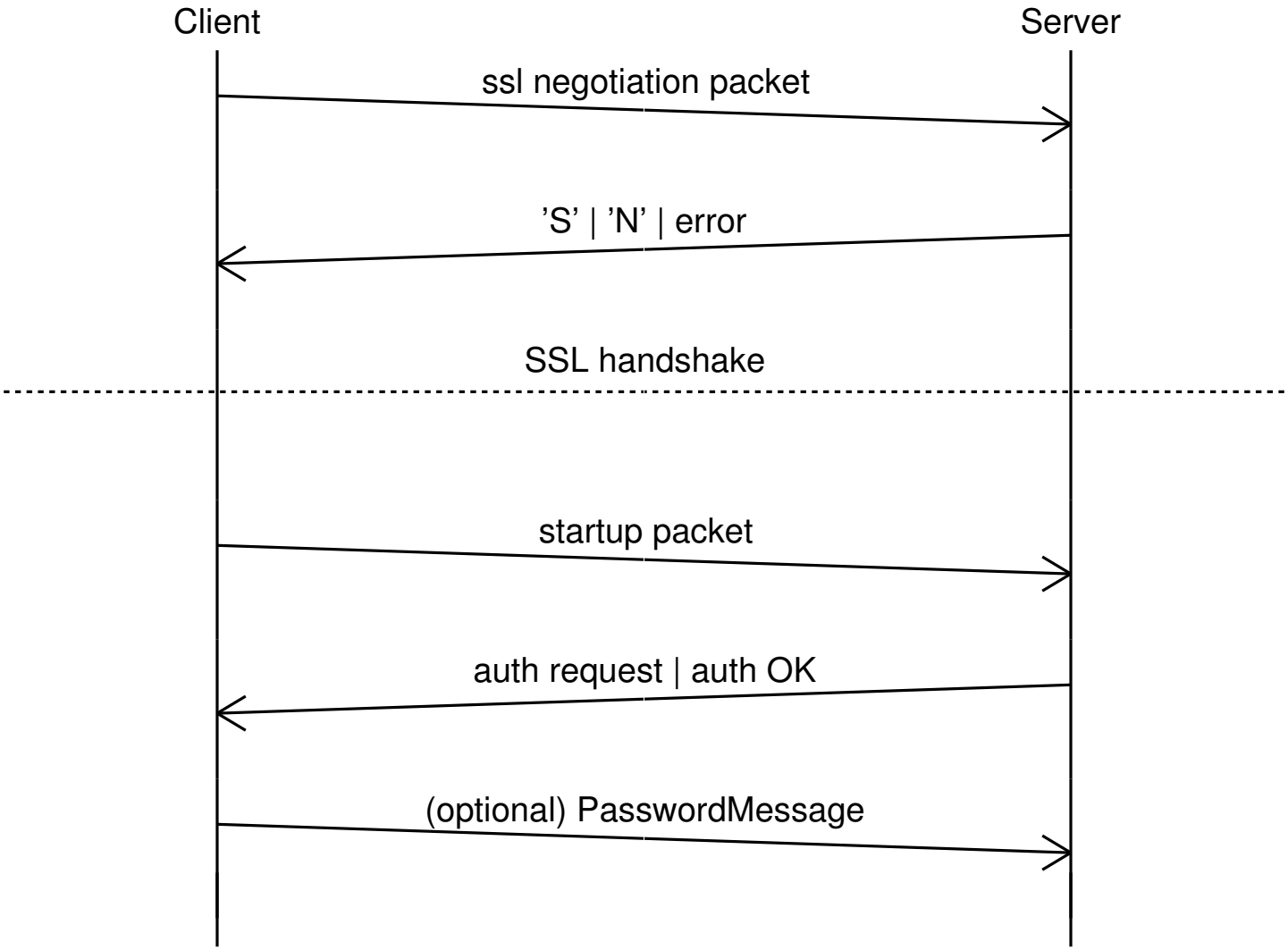
Encryption

SSL negotiation



- the startup packet can use a dummy protocol version to ask for SSL support
- the server responds with with status byte or an error message
- the client can reconnect or abort if the response is negative

SSL message flow



Cancellation

Cancel request

int32 len	int32 cancelcode	int32 pid	int32 secret

- the cancel key is transmitted by the server upon connection
- cancelling queries requires opening separate connection
- another dummy protocol version is sent to ask for cancellation
- the cancellation message includes the process ID and a 32 bit key
  - theoretically open to replay attacks, but can be sent over SSL
  - libpq does not, so most applications will transmit it in the open

Handling errors

ErrorResponse

'E'	int32 len	char code	str value	\0	char code	str value	\0	...	\0

- the ErrorResponse message is sent for all kinds of errors
  - both for authentication errors and client errors
- it is a list of key-value fields
  - in 2.0 it was just a string, in 3.0 it has structure
  - example error fields are: message, detail, hint and error position
  - detailed down to the source file and line, great for fingerprinting

Tools

- standard tools like tcpdump or tshark work
- Wireshark has built-in support for deparsing the protocol
  - but only for protocol 3.0
- pgShark is a very nice tool that works with the Postgres protocol

pgShark examples

generate a report from a pcap file

```
$ pgs-badger < dump.pcap
```

display live protocol info

```
$ pgs-debug --interface eth0
```

dump SQL from a 2.0 protocol connection on a nonstandard port

```
$ pgs-sql -2 --port 5433
```

Outline

Binary vs text data

- every type has a text and binary representation
- depending on compile-time options, timestamps are either 64 bit integers or floating point values
  - this is why `integer_datetimes` is sent in ParameterStatus
- the client can choose if they want text or binary data
- the exact format for each type doesn't seem to be documented anywhere
  - but that's what C code is for :)

Simple query protocol

- client sends an SQL command
- server replies with RowDescription detailing the structure
  - each column has a name
  - the type OID, length and modifier (like `char(16)`)
  - each column is marked as containing binary or text output
- after that a DataRow message is sent for every row
- finally, the server sends CommandComplete and ReadyForQuery

Simple query frames

Query

'Q'	int32 len	str query

RowDescription

'T'	int32 len	int16 numfields

+

str col	int32 tableoid	int16 colno	int32 typeoid	int16 typelen	int32 typmod	int16 format

...

DataRow

'D'	int32 len	int16 numfields	int32 fieldlen	char[fieldlen] data	...

Simple query frames cont.

CommandComplete

'C'	int32 len	str tag

ReadyForQuery

'Z'	int32 len	'I' or 'T' or 'E'

Detecting transaction status

- the ReadyForQuery message includes transaction status
- this is useful for things like psql's prompt or, more importantly, pgbouncer
- the transaction status only got included in protocol 3.0
  - for 2.0 libpq does string comparison to try and track the status

Detecting transaction status

- the ReadyForQuery message includes transaction status
- this is useful for things like psql's prompt or, more importantly, pgbouncer
- the transaction status only got included in protocol 3.0
  - for 2.0 libpq does string comparison to try and track the status

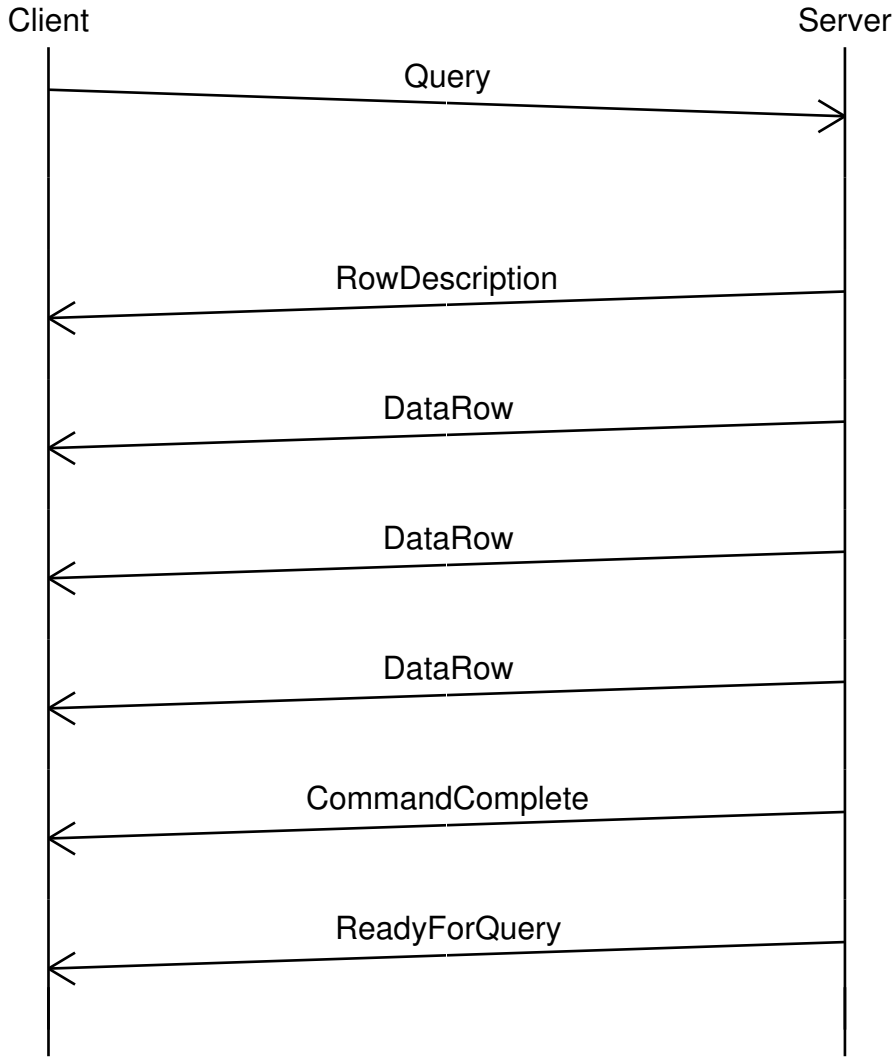
fe-protocol2.c

```
By watching for messages (...), we can do a passable
job of tracking the xact status.  BUT: this does not
work at all on 7.3 servers with AUTOCOMMIT OFF.
(Man, was that feature ever a mistake.)  Caveat user.
```

Simple query protocol cont.

- several commands can be sent in one query string
  - the server sends one CommandComplete per query
  - in case of errors it's up to the client to figure out which one failed
- sending an empty string yields a special EmptyQueryResponse instead of CommandComplete
- the simple protocol always returns text data, except for binary cursors

Simple query protocol flow



Outline

Extended query protocol

- query execution is split into separate steps
- each step is confirmed by a separately server message, but they can be sent consecutively without waiting
- allows separating parameters from the query body

```
SELECT admin FROM users WHERE login = '$var'
```

- disallows sending several commands in one query

```
SELECT * FROM posts WHERE id = $var
```

Extended query protocol

- query execution is split into separate steps
- each step is confirmed by a separately server message, but they can be sent consecutively without waiting
- allows separating parameters from the query body

```
SELECT admin FROM users WHERE login = 'x' or 1=1; --'
```

- disallows sending several commands in one query

```
SELECT * FROM posts WHERE id = $var
```

Extended query protocol

- query execution is split into separate steps
- each step is confirmed by a separately server message, but they can be sent consecutively without waiting
- allows separating parameters from the query body

```
SELECT admin FROM users WHERE login = 'x' or 1=1; --'
```

- disallows sending several commands in one query

```
SELECT * FROM posts WHERE id = 1; delete from posts;
```

Extended query protocol

- query execution is split into separate steps
- each step is confirmed by a separately server message, but they can be sent consecutively without waiting
- allows separating parameters from the query body

```
SELECT admin FROM users WHERE login = $1
```

- disallows sending several commands in one query

```
SELECT * FROM posts WHERE id = $1
```

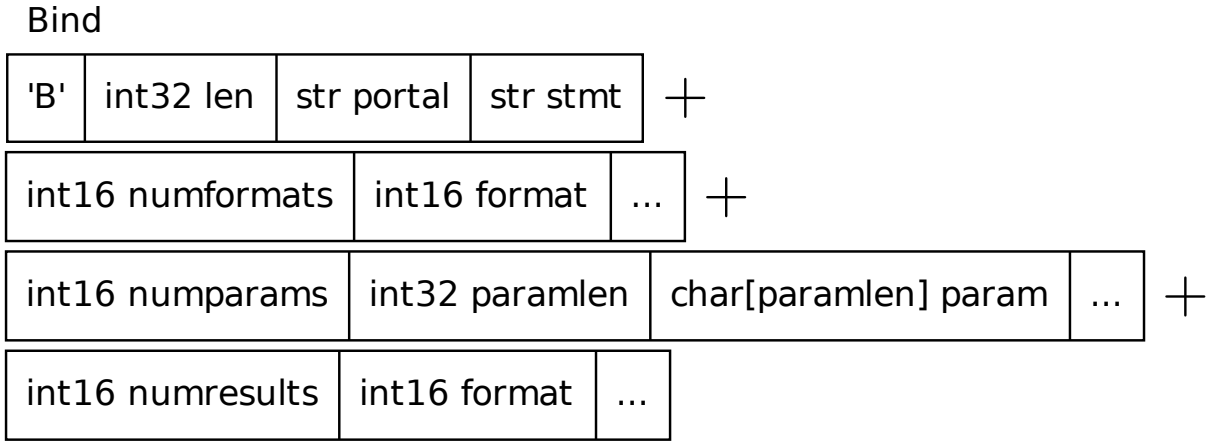
Parse messages

Parse

'P'	int32 len	str stmt	str query	int16 numparams	int32 paramoid	...

- first, the client sends a Parse message with the query string
- it can contain placeholders (`$1, $2, ...`) for parameters
- for each parameter you can specify its type
  - disambiguate between `select foo(1)` and `select foo('x')`
- the statement can be optionally given a name
  - unnamed statements live until the next unnamed statement is parsed
  - named statements need to be explicitly deallocated

Bind messages



- after the query is parsed, the clients binds its parameters
- an output portal is created for a previously parsed statement
  - an empty string can be used for the portal name
- for each parameter, its format (binary or text) and value are specified
- finally, for each output column, the requested output format is sent

Interlude - Describe messages

Describe

'D'	int32 len	'S' or 'P'	str name

ParameterDescription

't'	int32 len	int16 numparams	int32 paramoid	...

- clients can ask for a description of a statement or a portal
- statement descriptions are returned as two separate messages: ParameterDescription and RowDescription
- portal descriptions are just RowDescriptions
- clients can use Describe to make sure they know how to handle data being returned

Execute messages

Execute

'E'	int32 len	str portal	int32 rowlimit

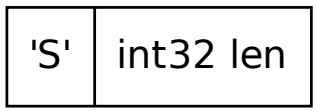
- once the output portal is created, it can be executed
- the output portal is referred to by name
- can specify the number of rows to return, or 0 for all rows
- a series of DataRow messages follow
- no RowDescription is sent

Execute messages cont.

- after the portal has been run to completion, CommandComplete is sent
- if the requested number of rows is less than what the portal would return a PortalSuspended message is sent
- AFAIK, only JDBC actually exposes limits for Execute
- libpq doesn't even have code to handle PortalSuspended...

Sync messages

Sync

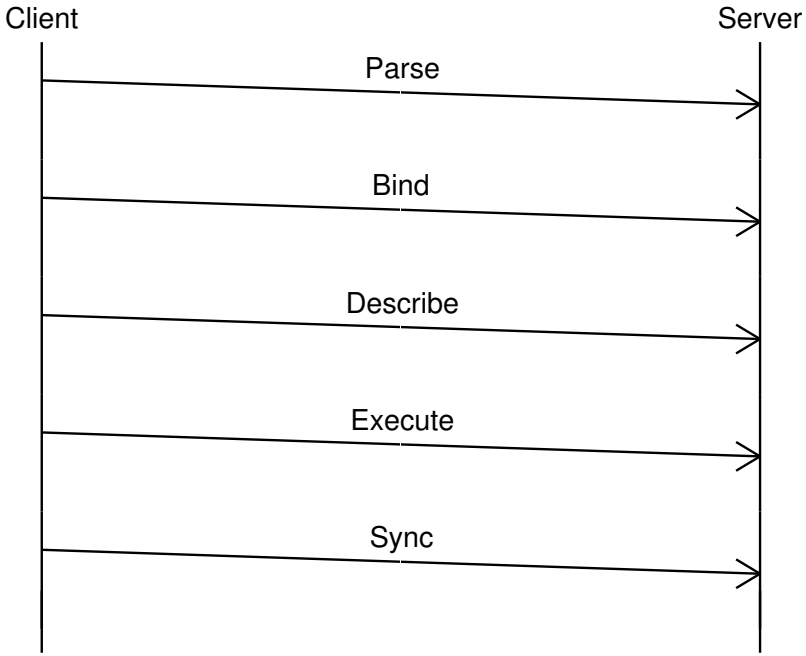


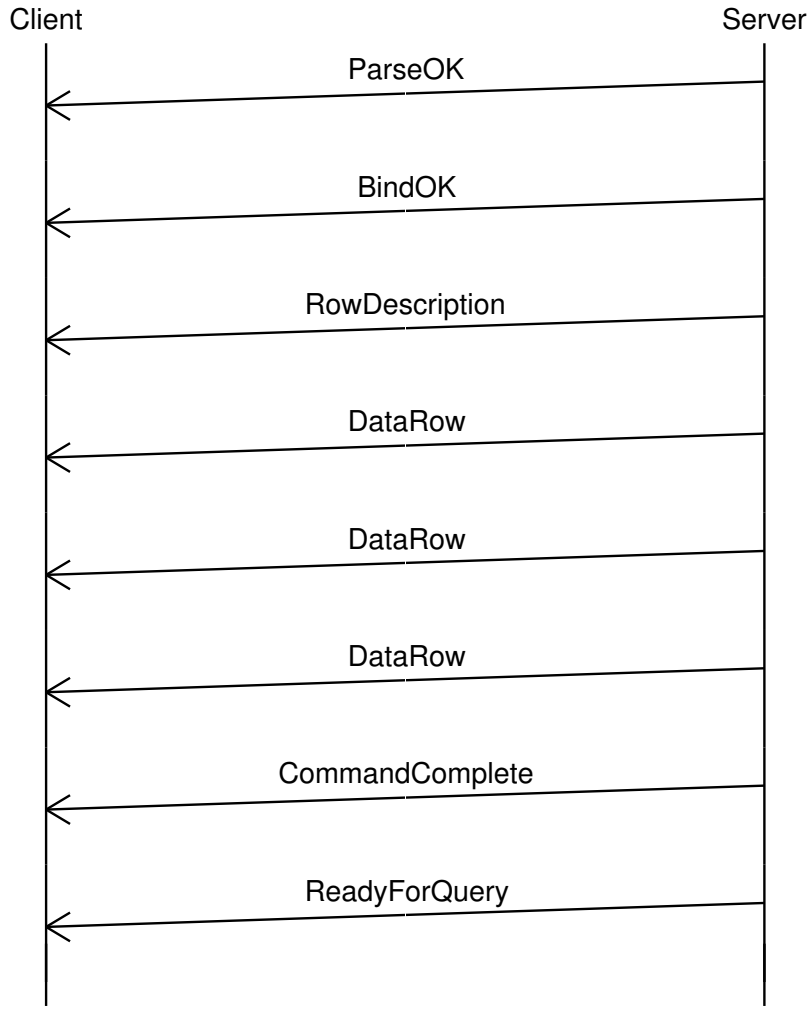
- an extended protocol query should end with a Sync
- upon receiving Sync the server closes the transaction if it was implicit and responds with a ReadyForQuery message
- in case of earlier errors, the server sends an ErrorResponse and then skips until it sees a Sync

Extended query protocol summary

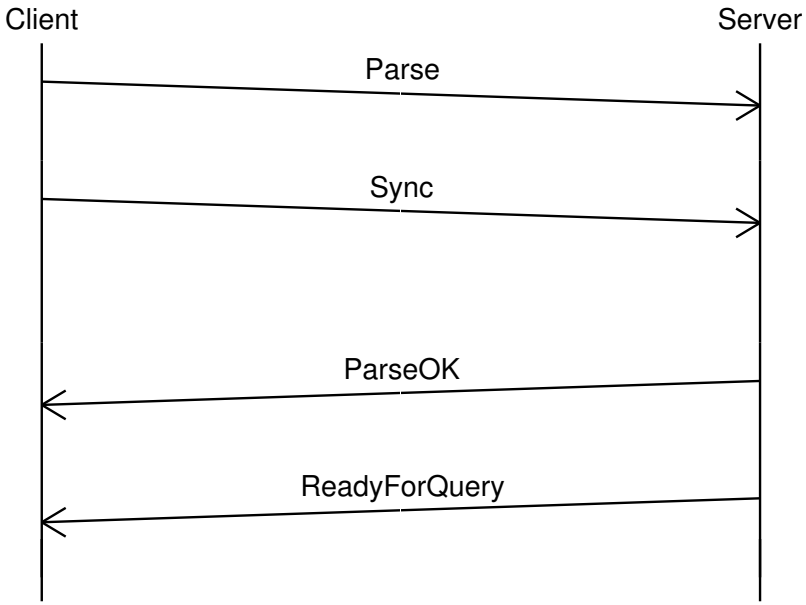
- queries are parsed at Parse stage
- queries are planned at Bind stage
- queries are executed at Execute stage
- with statement logging, these three steps will be timed and logged separately

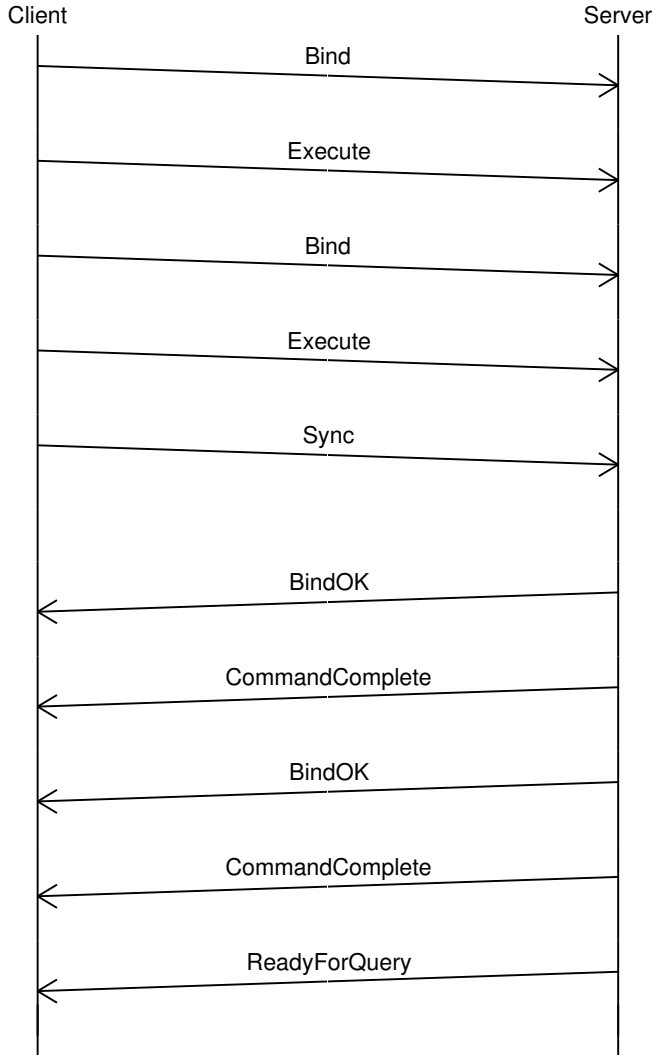
Extended query protocol flow





Advanced extended protocol usage





Outline

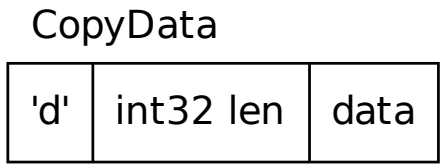
Entering COPY mode

CopyInResponse

'G'	int32 len	int8 format	int16 numfields	int16 format	...

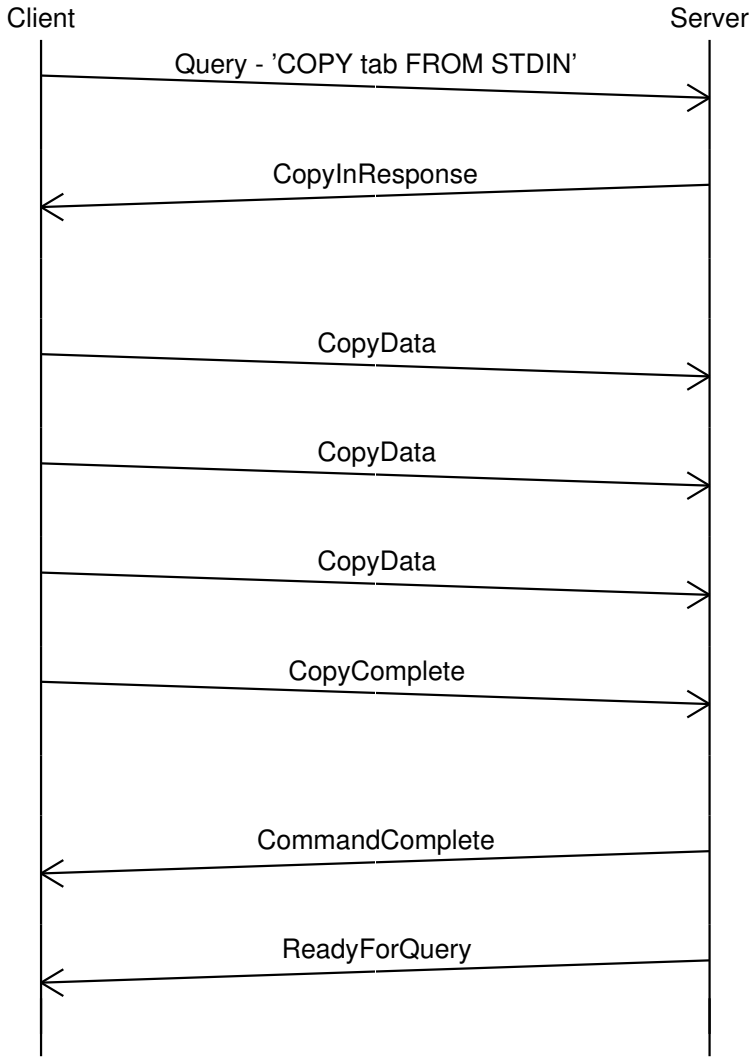
- sending `COPY FROM STDIN` or `COPY TO STDIN` puts the connection in COPY mode
- this can happen both during simple and extended query processing
- CopyInResponse and CopyOutResponse indicate that the backend has switched to COPY mode
- they specify the overall format (text or binary) and the format for each column
  - currently if the overall format is binary, all columns are binary

Sending COPY data



- CopyData messages are simply binary blobs
- to stop `COPY FROM`, the client can send a CopyFail message
- when transfer is complete, the client sends CopyDone
- in case of backend errors, an ErrorResponse is sent
- there is no way for the frontend to stop a `COPY TO` operation, short of cancelling or disconnecting

COPY subprotocol flow



Outline

Asynchronous operation

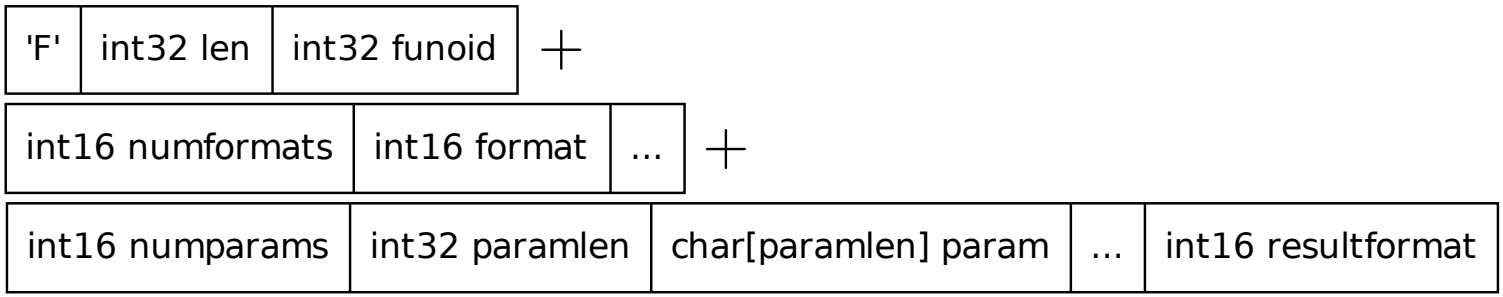
NotificationResponse

'A'	int32 len	int32 pid	str channel	str payload

- some messages can appear at any moment during the connection
  - ParameterStatus
  - NoticeResponse
  - NotificationResponse
- NOTIFY messages are only sent when a transaction is committed, but you should expect them at any time
- notices can be sent at any moment

Fast-path interface

FunctionCall



- a specialised interface for calling functions
- separate protocol message, FunctionCall, similar to Query
  - the function is identified by its OID
  - arguments format and values are specified similar to Bind
- libpq documentation calls it "somewhat obsolete" :)
- can be substituted by a named Parse followed by Bind/Execute
- still used by libpq for large object functions

Replication subprotocol

- entered using a special `replication` parameter in the startup packet
- switches the server to a mode where only the simple query protocol can be used
- instead of SQL, the server accepts replication commands
  - for example, `START_REPLICATION` or `BASE_BACKUP`
- responses are a mix of RowDescription/DataRow and COPY subprotocol data

Outline

Protocol version 4.0

There are surprisingly few gripes about protocol 3.0, but some proposals have been floated on the development list.

- protocol compression
- adding nullable indicator to RowDescription
- multi-stage authentication, allowing falling back to a different authentication method
- negotiating the protocol version
- in-band query cancellation
- sending per-statement GUC

Questions?