Communication Procotol between the Frontend and POSTGRES Backend
(Version of: $Date: 90/07/30 13:50:21 $)

1. Communication

1.1. Communication Ports

A frontend communicates to the POSTGRES backend using IPC ports. There are two ways to specify the communication port: (1) the environment variable PGPORT; (2) the optional argument *-p port* to the terminal monitor. If PGPORT and *-p port* specify different ports, the one specified by *-p port* is used. If no port is specified, a default port (currently 4321) is used. Ports other than the default port is used mostly for debugging.

1.2. Initialization of the Communication

When the communication starts, the frontend passes a string (referred to as *init string* thereafter) to the postmaster. The postmaster processes the init string, starts up a backend, and initialize the backend with the init string. The init string is in the format:

init string:	user-name,database,tty,[pgoption] <newline>
user-name:	the user's login
database:	the database to access
tty:	the tty in which the backend debugging message is displayed, default is /dev/null, used for debugging.
pgoption:	optional arguments

A typical init string is:

"ywang,ywang,/dev/ttyp4,<newline>"

A user could be registered with a password. However, password is not required for every user. When users are registered with the POSTGRES manager, they can decide whether they want to use passwords.

Having received the init string, the backend checks: (1) if the user is not a valid POSTGRES user, send a *deny* message back; (2) if the user is registered with a password, ask the frontend for the password; if the password supplied is not correct, send a *deny* message to the frontend; (3) if the user does not have access to the database specified, send back a *deny* message; (4) if all three tests are successful, send back an *authorization* message.

The message from backend to frontend:

ask for password:	W
authorize:	Y <backend_pid> <verification#> <backend_pid> is the pid of the backend process <verification#> is a code used when abortion is requested
deny:	N <errorcode> <errormsg> <errorcode> and <errormsg> are defined in Section 4.

The message from frontend to backend:

send password:	W <password>

we will use a protected port for communication, so the <password> will be transferred as a string. The backend will then encrypt it and compare with the encryption stored in the USER relation.

If the backend sends a Y to the frontend, the communication has been established, the frontend may start sending queries for execution. If the backend sends an N to the frontend, it will terminate the communication at the same time, the frontend can either quit or repeat the above process with a different init string.

1.3. Optional Arguments

These optional arguments can also be executed as a function.

command = yes
    Return the query command
append_oid = yes
    return the oid of the appended tuple (only when a single tuple is appended)
number_tuples = yes
    return the number of tuples updated (replaced, deleted, etc.)
max_length = nbytes
    truncate large data objects to nbytes when retrieving or fetching.
blank_fetch = yes
    fetch all the qualified tuples when retrieve into blank portal, if
    this is not set, no tuple is returned until the first fetch.
blank_ascii = yes
    transfer tuples from a blank portal in binary format.
data_ascii = yes
    transfer tuples from data portal in ASCII format.
network = yes
    convert binary values to network byte-ordering during transfer.

1.4. Status

The init string (except optional arguments) has been implemented, but nothing else. An error with the init string is not reported until the first query is sent to the backend.

2. Executing Queries

After the frontend has been authorized to access the database, it can send queries to the backend for execution. POSTGRES supports two kinds of **portals**: blank portals and data portals. Usually blank portals are used by the terminal monitor, data portals used by more sophisticated frontend applications.

2.1. Messages from Frontend to Backend

to send a query: Q <xactid> <query>
to execute a function: F <function_name> <arg_list>

2.2. Messages from Backend to Frontend

to signal an error: E <xactid> <errorcode> <line#> <column#> <errormsg>
    line# and column# is where the error occurred within the query

successful query: {portal} command
portal: P/A <xactid> <pname> {tuple_group} Z
tuple_group: T <nfields> {tupe_info} {tuple}
tupe_info: <fname> <adtid> <adtsize>
tuple: D <bitmap> <tuple_length> {value}
value: [<value_length>] <value_block>
    value_length is the number of bytes in the value-block.
    a value_length only precedes a variable size data object.
command: C <xactid> <command> return-values Z
return-values: [O <oid>] [N <no_tuples>]
note that {R <remark>} can appear before any identifiers.

A successful function execution may return a primitive value, a group of tuples, or even a portal. The protocol for portal and tuples should be similar to what described above. The primitive value returned is:

function_value: V <value_length> <value_block>

The frontend should know how to interpret these values. A set of functions need to be added to LIBPQ for handling of function execution.

We may want to group the return values from backend into large blocks to save the network transmission cost. But this will not happened soon.

2.3. Status

Query execution and tuple returning (except <tuple_length>) have been implemented. Binary data transfer has not been implemented yet. Nothing is there to support execution of functions. Returning of oid and no_tuples have not been implemented yet. The identifier Z is not there. We still need to figure out the syntax for arg_list.

3. Messages from Frontend to Postmaster

The frontend may need a second socket connected to the same backend (we may live without a second socket). The frontend definitely need someway to abort a transaction. Postmaster has two ports, one is for the start up of communication, the other one is a datagram port receiving and processing packages sent to it. The datagram port can be used to implement the abortion facility. However, it needs a considerable amount of work to get abortion to work.

to ask for a socket:	S
to abort a xact:	B <backend_pid> <verification#>

In both cases, the backend should send back some message.

Nothing discussed in this section has been implemented.

4. Definitions of Terms

string:	strlen characters
strlen:	int[4] is the number of characters in the string, not including the 4 bytes in strlen.
string[n]:	string with a max length of n
backend_pid:	int[4]
verification#:	int[4]
errorcode:	int[2] is the code of the error. Each code represents one kind of error. The errormsg will only pass the arguments of the error.
errormsg:	string[80] error messages sent by the backend
password:	string
xactid:	int[5] is the transaction id of the current transaction. we do not know at this point how xactid will be used.
query:	string is the query sent to the backend
function_name:	string is the name of the function to be invoked
arg_list:	some binary format
line#:	int[4]
column#:	int[4]
remark:	string[80] are the remarks sent by the backend, the frontend just drops it on the floor right now, could be used later on.
pname:	string is the portal name
fname:	string is the field (attribute) name

nfields:	int[2] is the number of fields (attributes) in the tuples.
adtid:	int[4] is the adtid of a data type
adtsize:	int[2] is the number of bytes of a data type. -1 if the type is of variable size.
bitmap:	int[n] where n = ceiling(nfields/8) is the bitmap of a tuple. Bitmap is sent 4 bytes at a time.
tuple_length:	int[4]
value-length:	int[4] value_length is the number of bytes in the value-block, not including the 4 bytes of itself.
value_block:	a stream of bytes
command:	string[20] is the query command passed from the backend, such as retrieve, fetch, etc.
oid:	int[4]
no_tuples:	int[4]
identifier:	a single char an identifier always precedes the information being passed.

5. Identifiers

W:	ask for password
Y:	frontend authorized
N:	frontend denied
Q:	query (from frontend to backend)
F:	to execute a function
E:	error message
P:	synchronized portal a synchronized portal is a normal portal specified in the user's query command.
A:	asynchronized portal an asynchronized portal is a portal defined by rules, which are triggered when a query is executed.
T:	type information the type block for a tuple group
D:	a tuple
C:	query command executed (from backend to frontend)
Z:	end of an information block
R:	remarks
S:	asking for a socket
B:	abort a xact

6. Implementation Plan

6.1. Version 1

To implement data portal, we need two functions to convert byte ordering. All other code is in ˜postgres/src/util/{printtup.c, dumptup.c, pcomm.c}. To add oid to the backend would cost two weeks of Serge's time.

6.2. Version 2

What will go into Version 2 has not been decided yet.

7. Notes

1. POSTGRES is currently a SINGLE user system. Therefore oid's allocated by different backends may not be unique.
2. all integers transferred between frontend and the backend should first be converted to network byte ordering, and then changed back when received.
3. what should be done with very large data objects? anything the frontend thinks appropriate.
4. how to append or replace tuples with binary attributes? use fast path.
5. how to execute a function from TM? use \f.
6. <function_name> is currently char16.

8. Future Extensions

1. help for database schema similar to the "help" in INGRES.
2. friendly error message.
3. a frontend "help" about backslash commands.
4. after an update query, print out the #tuples updated.

9. Issues

1. number of portals which can be open at the same time (currently 3 or 4).