TITLE
Asst3: The Decidedly Uncomplicated Message Broker

DUE
Wed 11 Dec 2019 23:55

NUMBER OF RESUBMISSIONS ALLOWED
Unlimited

ACCEPT RESUBMISSION UNTIL
Wed 11 Dec 2019 23:55

Asst3: The Decidedly Uncomplicated Message Broker

CONTENTS

A. Introduction
B. Client Operation
C. Server Operation
D. Packaging
E. Commands
  - E.0 `HELLO:` initiate a user session
  - E.1 `GDBYE:` stop a user session
  - E.2 `CREAT:` create a new msg box on server
  - E.3 `OPNBX:` open an existing msg box
  - E.4 `NXTMG:` get next msg from current msg box
  - E.5 `PUTMG:` put msg in currently open msg box
  - E.6 `DELBX:` delete msg box from server
  - E.7 `CLSBX:` close current msg box
F. Notes

A. Introduction

The Decidedly Uncomplicated Message Broker (DUMB) is a protocol for implementing a remote mail box that is engineered to be quite terse and simple.

It dispenses with the extra services and robust error reporting of SMTP and IMAP (//RFC 5321 and // RFC 3501) in favor of a more **direct protocol format**.

In particular, every **command** is **six characters** long (excluding **data** and **delimiters**) and every **server response code** consists of a **three-character-long success/error response**, with a further **five characters** for an **error code** (excluding **data**).

This simplifies the protocol and eases implementation.

B. Client Operation

On start, a **DUMB client** should be invoked with the **IP address** or **hostname** and **port number** of a **DUMB server** as **command-line arguments**.

If the **DUMB client** should try **three times** to connect to a **DUMB server** at the **address/hostname** and **port** given. If it **can not connect** after **three tries**, or **something other than the correct reply** is sent in response to a `HELLO` command (see E.0 below), the client should **report an error** to the user and **return/shut down**.

If it does **connect** and gets the **correct reply** to a `HELLO` command (see E.0 below), the **DUMB client** should **continue accepting commands** from the user until it **disconnects** from the **DUMB server** (see E.1 below).

This may span the user **creating, deleting, opening, putting, getting and closing multiple messages** from a **series of message boxes**.

In order to simplify the messaging, **no identity tags, descriptors or keys are handed out to users**, as the client is intended to only work with **a single message box at a time**.

All **message box commands are context-sensitive** in order to enable this simplification, applying to **whichever message box the user has currently opened**, as **issuing a message box command without one open is an error.**

The client should, **on connection**, report **success to the user** and **wait for user commands**.

It should not expect the user to enter commands directly in DUMB protocol format, but instead should recognize first an English description of the command.

The English descriptions are:

`quit`	(which causes: E.1 `GDBYE`)
`create`	(which causes: E.2 `CREAT`)
`delete`	(which causes: E.6 `DELBX`)
`open`	(which causes: E.3 `OPNBX`)
`close`	(which causes: E.7 `CLSBX`)
`next`	(which causes: E.4 `NXTMG`)
`put`	(which causes: E.5 `PUTMG`)

On '`help`' the client should list the commands above.

If a command requires an argument,
the **client** should **prompt for it after the command**
with the **command name** as part of the **prompt**.

If the command does not require an argument,
it should be **sent by the client immediately**.

The **client** should also **interpret server replies**
and **print out non-protocol text for the user**, explaining the **response**.

e.g.

```
> open
Okay, open which message box?
open:> mybox
Success! Message box 'mybox' is now open.
>
```

-or-

```
> open
Okay, open which message box?
open:> m
Error. Command was unsuccessful, please try again.
> mybox
That is not a command, for a command list enter 'help'.
> open
Okay, open which message box?
open:> mmybox
Error. Message box 'mmybox' does not exist.
> open
Okay, open which message box?
open:> mybox
Success! Message box 'mybox' is now open.
>
```

C. Server Operation

On start, a DUMB server should be invoked with the port number it should listen on.
The port number should be strictly greater than 4K (not base-10).

On receiving a connection request,
the DUMB server should create a new thread to handle the client commands.

The **server** should **report all successful events** to `stdout` and all errors to `stderr` by listing the **timestamp, client identity, the successful command or the error incurred. Data should not be output**.

e.g. Event output

```
1737 11 Dec 128.5.2.1 connected
1737 11 Dec 128.5.2.1 HELLO
1738 11 Dec 192.168.10.1 GDBYE
1738 11 Dec 192.168.10.1 disconnected
1739 11 Dec 128.5.2.1 OPNBX
```

e.g. Error output

```
1922 9 Mar 17.14.12.4 ER:EXIST
1927 9 Mar 199.123.4.1 ER:WHAT?
1928 9 Mar 199.123.4.1 ER:WHAT?
1928 9 Mar 199.123.4.1 ER:OPEND
```

The server should maintain a single message box structure that is shared between all connections.

Since it is **shared data**, the **message boxes** should be **controlled** for **simultaneous/parallel access** between **clients running in multiple threads**.

Each **message box** should have its own **synchronization mechanism** that should **control any change of state (open/closed).**

A **client** can only **modify the data** contained in a **message box** (put/next message) i**f it currently has the box open**.

A **message box can be opened by only one client at a time**, as **opening a message box gives that client exclusive access to the message box until the same client closes it**.

The **entire message box store should have its own general synchronization mechanism** which should **control modification of the message box store** (create/delete a box).

Care should be taken to **handle synchronization carefully** so that the **server does not cause client threads to deadlock**.

Single message box commands should not block, as there are **errors for a user trying to open a box already open, or modifying a box it does not have open**.

This does not mean that there can be inconsistent access, **but if a message box is busy or locked**, the user should **get an error instead of blocking on a single-box command**, so make sure to use **non-blocking synchronization for their implementation**.

Commands that affect the entire message box store, like creating or deleting a box, are expected to be fairly rare, and are allowed to block, so use blocking synchronization for their implementation.

The DUMB server holds all data in main memory.
It has no explicit shutdown command or actuation,
and is **quit** with a **Ctrl+C** from the **foreground** on the **server side**.

D. Packaging

Deliver your implementation in a `DUMB.tgz`.

`DUMB.tgz` should be a gzipped-tar file consisting of:

`testplan.txt`	(testing done to determine the DUMB client/server work)
`DUMBserver.c`	(server source code)
`DUMBclient.c`	(client source code)
`Makefile`	(build scripting)
`...`	(any necessary additional libraries or headers)

A DUMB deployment should be built with:

`make all`	(compile server and client to executables named: "`DUMBserve`" and "`DUMBclient`", respectively)
`make client`	(compile client executable named: "`DUMBclient`", only)
`make serve`	(compile client executable named: "`DUMBserve`", only)

DUMBclient must require an **IP address** or **hostname**
and a **port number** of a **DUMB server** to connect to:

e.g.
```
./DUMBclient 128.74.1.2 14091
```
-or-
```
./DUMBclient www.whitehouse.gov 8912
```

DUMBserve must require a **port number** to start up the **DUMB server** on:
e.g.
```
./DUMBserve 19294
```

E. DUMB Commands

The DUMB server is entirely reactive, only responding to client commands,
so the commands below are detailed from the point of view of the client side.

E.0 HELLO

Initiate a session with a DUMB server.
The client side sends a basic message to start interaction with the server.

The client expects a response from the server in order to verify it is there and listening.

If an incorrect response comes back, the client should disconnect,
report an error to the user and shut down, else if correct report success.

CORRECT RESPONSES

`HELLO DUMBv0 ready!`
If the server responds with the above,
you are now connected to the server and can report success to the user.

e.g.

```
HELLO
HELLO DUMBv0 ready!
```

INCORRECT RESPONSES

<anything other than correct response>
If something other than the expected response comes back,
report error to the user and close the connection.

E.1 GDBYE

Stop a session with a DUMB server.
The client side sends a message to stop interaction with the server.

The client expects no response text from the server, since the server should close the connection.

If any response comes back, the client should report an error to the user,
else if no bytes can be read, it should report success.

The server should close the client's open message box,
if the user had one open and did not close it before disconnecting.

CORRECT RESPONSES

<socket can not be read from and was closed on server side>
If the socket can not be read from, and was closed,
report success to the user and shut down the program.

INCORRECT RESPONSES

<anything other than correct response>
If something other than the expected response comes back,
report error to the user and do not close the connection.

E.2 CREAT arg0

Create a new user/message box name on the DUMB server.
The client sends the command and a name for the new message box the user wants to create.

The client expects a response to determine if the message box was created or not,
which it reports to the user in either case.

`arg0`: The name for a new message box

The name must be 5 to 25 characters long and start with an alphabetic character.

CORRECT RESPONSES

`OK!`

If the server responds with the above, you have just created the message box.
Creating the message box does not give you access to it.

To gain access, you should `OPNBX` (see E.3 below).

e.g.

```
CREAT blurp
OK!
```

INCORRECT REPSONSES

`ER:EXIST`

If the server responds with the above,
a box with that name already exists and it can not be created again.

e.g.

```
CREAT urple
OK!
CREAT urple
ER:EXIST
```

`ER:WHAT?`

Your message is in some way broken or malformed.

e.g.

```
CREATurp
ER:WHAT?
```

-or-

```
CREA urp
ER:WHAT?
```

-or-

```
CREAT 1
ER:WHAT?
```

E.3 `OPNBX arg0`

Open an existing user/message box name on the DUMB server.
The client sends the command and a name for the existing message box the user wants to open.

The client expects a response to determine if the message box was opened or not, which it reports to the user in either case.

`arg0`: The name for a new message box

The name must be an existing name 5 to 25 characters long, starting with an alphabetic character.

CORRECT RESPONSES

`OK!`

If the server responds with the above,
you have just opened the message box and have exclusive access to it until you close it.

e.g.
```
OPNBX blurp
OK!
```

INCORRECT RESPONSES

`ER:NEXST`

If the server responds with the above, a box with that name does not exist, so it can not be opened.

e.g.
```
OPNBX blurp
ER:NEXST
```

`ER:OPEND`

If the server responds with the above, a box with that name is currently opened by another user, so you can not open it.

e.g.
```
OPNBX blurp
ER:OPEND
```

`ER:WHAT?`

Your message is in some way broken or malformed.

e.g.
```
OPNBXurp
ER:WHAT?
```
-or-
```
PNBX urp
ER:WHAT?
```
-or-
```
OPNBX 1
ER:WHAT?
```

E.4 NXTMG

Get the next message from the currently open message box.
The client sends the command to get the next message in the box it currently has open.

Messages are stored in a queue, so messages are handed back in the same order they were PUTMG.

Getting the message deletes it from the server side.

The '!' is a required separator between the command and byte length, and then the byte length and message. Any other '!' after the first two should be in the message and is part of the data.

The client expects a response including the message text or an error, which it reports to the user in either case.

Message text is written to standard out as ASCII chars.

CORRECT RESPONSES

`OK!arg0!msg`
If the server responds with the above, you have just fetched the next message, deleting it from the server side.

`arg0`: the character representation of the length of the message, not including the '!' required by protocol
`msg`: the message's bytes, presumed to be ASCII characters

e.g.
```
NXTMG
OK!5!hello
```
-or-
```
NXTMG
OK!12!Oh hai, Mark!
```
-or-
```
NXTMG
OK!1!.
```

INCORRECT RESPONSES

`ER:EMPTY`
If the server responds with the above, there are no messages left in this message box.

e.g.
```
NXTMG
ER:EMPTY
```

`ER:NOOPN`
If you currently have no message box open, the server responds with the above.
(the message box not existing should result in the same error, since if it does not exist, you can not open it)

e.g.
```
NXTMG
ER:NOOPN
```

`ER:WHAT?`
Your command is in some way broken or malformed.

e.g.
```
NEXTMESSAGE
ER:WHAT
```

E.5 PUTMG!arg0!msg

Put a message in to the currently open message box.
The client sends the command to put a message in to the box it currently has open.

Putting the message stores it on the server side.

Messages are stored in a queue, so messages are handed back/removed from the message box with NXTMG (see above) in the same order they were PUTMG.

The '!' is a required separator between the command and byte length, and then the byte length and message. Any other '!' after the first two should be in the message and is part of the data.

The client expects a response indicated the message is stored, or an error, which it reports to the user in either case.

arg0: the number of bytes of the message, following the '!' required by the protocol
msg: the bytes that make up the message, presumed to be ASCII characters

CORRECT RESPONSES

OK!arg0
If the server responds with the above,
you have just put the message in the box you have open on the server side.

arg0: the character representation of the length of the message

e.g.
```
PUTMG!5!hello
OK!5
```
-or-
```
PUTMG!12!Oh hai, Mark!
OK!12
```
-or-
```
PUTMG!1!.
OK!1
```

INCORRECT RESPONSES

ER:NOOPN
If you currently have no message box open, the server responds with the above.

e.g.
```
PUTMG!12!Oh hai, Mark!
ER:NOOPN
```

ER:WHAT?
Your command is in some way broken or malformed.

e.g.
```
PUTMG12!Oh hai, Mark!
ER:WHAT?
```
-or-
```
PUTMG!5!Oh hai, Mark!
ER:WHAT?
```
-or-
```
PUTMG!12Oh hai, Mark!
ER:WHAT?
```

E.6 DELBX arg0

Delete a user/message box name on the DUMB server.
The client sends the command and a name of a message box to delete.

The client expects a response to determine if the message box was deleted or not, which it reports to the user in either case.

Note that you can not have the message box you want to delete open, so you are not guaranteed exclusive access to it when you run this command and, based on the order your command is received, you might be pre-empted by another user.

`arg0`: The name of the message box to delete

The name must be 5 to 25 characters long and start with an alphabetic character.

CORRECT RESPONSES

`OK!`
If the server responds with the above, you have just deleted the message box.

e.g.
```
DELBX blurp
OK!
```

INCORRECT RESPONSES

`ER:NEXST`
If the server responds with the above, a box with that name does not exist, so it can not be deleted.

e.g.
```
DELBX blurp
ER:NEXST
```

`ER:OPEND`
If the server responds with the above, a box with that name is currently open, so you can not delete it. (any user opening the box will lock it open, including yourself)

e.g.
```
DELBX blurp
ER:OPEND
```

`ER:NOTMT`
If the server responds with the above, the box you are attempting to delete is not empty and still has messages.

e.g.
```
DELBX blurp
ER:NOTMT
```

`ER:WHAT?`
Your command is in some way broken or malformed.

e.g.
```
DELBXurp
ER:WHAT?
```
-or-
```
ELBX urp
ER:WHAT?
```
-or-
```
DELBX 1
ER:WHAT?
```

E.7 `CLSBX arg0`

Close the user/message box name on the DUMB server that you currently have open.
The client sends the command and a name of a message box to close.

The client expects a response to determine if the message box was closed or not, which it reports to the user in either case.

`arg0`: the name of the message box to close

The name must be 5 to 25 characters long and start with an alphabetic character.

CORRECT RESPONSES

`OK!`

If the server responds with the above, you have just closed the message box.

e.g.

```
CLSBX blurp
OK!
```

INCORRECT RESPONSES

`ER:NOOPN`

If the server responds with the above, you do not currently have that box open, so you can't close it.

(the message box not existing should result in the same error,
since if it does not exist, you could not have opened it)

e.g.

```
CLSBX blurp
ER:NOOPN
```

`ER:WHAT?`

Your command is in some way broken or malformed.

e.g.

```
CLSBXurp
ER:WHAT?
```

-or-

```
LSBX urp
ER:WHAT?
```

-or-

```
CLSBX 1
ER:WHAT?
```

F. Notes

It is rumored that an intern working in unit testing at Snark Co. insisted there was an error in the DUMB protocol above while it was briefly in the RFC stage.

Since this was holding up the testing and approval process on an important product as well affecting morale and being poor teamwork in general, the intern was fired.

The Senior Protocol Engineer retired the same week,
posting, `"have phun, gerbers!"`
on the DUMB project team `listserv` and has been dark since.

Code the DUMB protocol as detailed above,
if however an "undocumented feature" is found while implementing it,
please make a note of it and add an error message to handle it.

A protocol enhancement of this type will result in additional remuneration.

P.S.
There is a pretty nasty flaw in the protocol.
If you find it and patch it by adding a single error message to one command,
you are eligible for extra credit.

If you do patch it in, note it in your documentation and be sure to maintain the protocol's error format.