Creating C-based TCP Data Feed Connections to Kdb+

AquaQ Analytics Limited

Authors

This document was prepared by:

Kent Lee (primary author) Paul McCabe Chris Patton Ronan Pairceir Dermot French Ben Sharpe John Ludlow		

Revision History

Version Number	Revision Date dd/mm/yyyy	Summary of Changes	Document Author
1.0	01/02/2013	Initial Release	See list above

Table of Contents

1 Company Background

AquaQ Analytics Limited (www.aquaq.co.uk) is a provider of specialist data management, data analytics and data mining services to clients operating within the capital markets and financial services sectors. Based in Belfast, the company was set up in April 2011. Our domain knowledge, combined with advanced analytical techniques and expertise in best-of-breed technologies, helps our clients get most out of their data.

The company is currently focussed on three key areas:

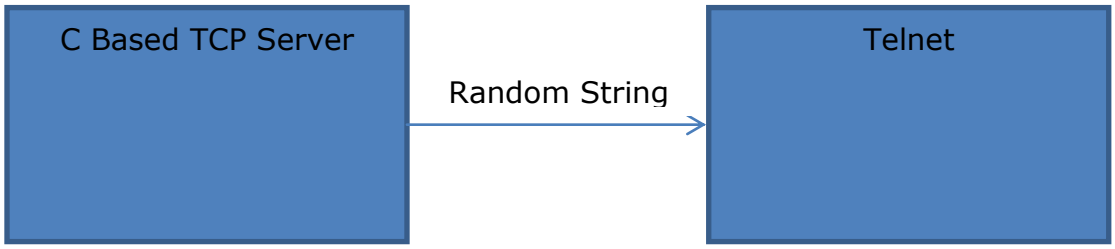
- Kdb+ Consulting Services – Development, Training and Support (both onsite and offsite).
- Real Time GUI Development Services (both onsite and offsite).
- SAS Analytics Services (both onsite and offsite).

For more information on the company, please email us on info@aquaq.co.uk.

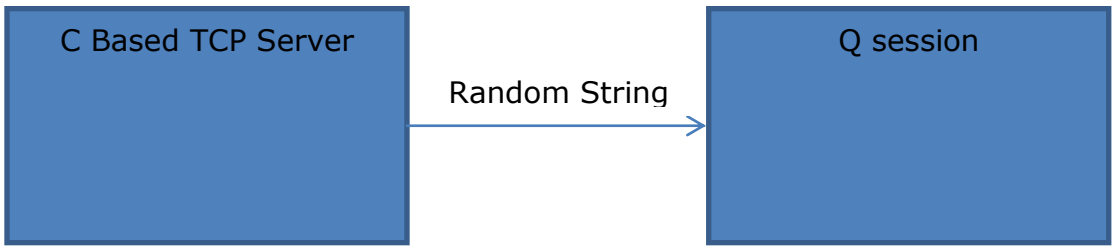
2 Introduction

This document provides an overview and accompanying sample code that the reader can use to construct a simple TCP socket connection to kdb+. Many market data feeds (such as Reuters SSL, EBS and many others) provide simple text based updates available over a TCP socket. This data can then be delivered to kdb+ and processed in a user defined manner.

This documentation only looks at the very basic messages. A simple feed architecture is shown below. Here a client can connect to a server, at which point the server will publish messages to the client, as long as the connection remains valid. After the client disconnects, the server will exit and die. A basic example, not involving kdb+ can be demonstrated using telnet.



In the more practical example a socket connection will be created from kdb+ and any messages published by the server will be dispatched to kdb+ via a predefined callback.



In this example, the message will be assumed to have a specific form, containing only ASCI characters and terminating with a new line (\n) character.

Note that this document was created by a junior developer in both C and Q, which indicates that the bridge is relatively straightforward to create.

3 Requirements

1) Demo version of kdb+ 2.8 (32-bit)

2) Linux/Unix (32-bit)

3) GCC compiler

4) echoclnt.zip

5) echoserv.zip

4 C Based TCP data feed

The TCP server code can be obtained from http://www.paulgriffiths.net/program/c/echoserv.php.
To start a TCP server, open a UNIX session.
Type "make" in the command prompt to compile the codes into executable.
Type "./echoserv {port number}" to start the TCP server.

This TCP server will first create a listening socket and bind it with the current session's socket address. Then it will accept the connection from a client, generate a random message and send it back to the connected client on an interval of 1 second. Lastly when the client is disconnected, the server will close the connection and die.

4.1 Connect to TCP server using telnet

```
wooikent@wooikent-VirtualBox:~/C_Prac/echoserv2$ ./echoserv 5625
Tue Feb  5 10:43:57 2013
: Connection opened
Tue Feb  5 10:43:57 2013
: sent 5a7nCr
 Tue Feb  5 10:43:58 2013
: sent 7szy4BDS
 Tue Feb  5 10:43:59 2013
: sent 3JzM
 Tue Feb  5 10:44:00 2013
: sent 3RV1
 Tue Feb  5 10:44:01 2013
: sent 5zlvWO
 Tue Feb  5 10:44:02 2013
: sent 1t
 Tue Feb  5 10:44:03 2013
: sent 7L01LZnX
 Tue Feb  5 10:44:04 2013
: sent 5IMS8P
wooikent@wooikent-VirtualBox:~/C_Prac/echoclnt2$ telnet localhost 5625
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
5a7nCr
7szy4BDS
3JzM
3RV1
5zlvWO
1t
7L01LZnX
5IMS8P
```

4.2 Connect to TCP server using q

Compilation

A simple makefile is created to make compiling easier. This can be done by creating a file named "Makefile" as such:

```
echoclntq.so:echoclntq.c k.h
	gcc -D KXVER=2 -shared echoclntq.c -o echoclntq.so
```

Note that "k.h" header file and "c.o" linker are needed for kdb+ to interact with C. A tutorial on creating a simple makefile can be found in http://mrbook.org/tutorials/make/.

To compile the C code, just type "make" in a UNIX session.

```
wooikent@wooikent-VirtualBox:~/C_Prac/echoclnt2$ make
gcc -D KXVER=2 -shared echoclntq.c -o echoclntq.so
```

Q script to connect to the server

```
//define a function to load the C based connect function into q
conn:`:./echoclntq 2:(`conn;3)

//define a function to load the C based disconnect function into q
disconn:`:./echoclntq 2:(`disconn;1)

//define a function to show the message sent from the C based server at a particular time
upd:{[x] 0N!string[.z.t],": ",x}

//define a function to show the error message sent from the C base server at a particular
time
oneerror:{[x] 0N!string[.z.t],": ",x }

-1"e.g. conn[127.0.0.1;5625;`upd] to start connection\ne.g disconn[] to disconnect\nupd
and oneeror function are defined"
```

"2:" function is used to dynamically load a C function into kdb+.

The left argument is a symbol representing the name of the dynamic library from which to load the function. The right argument is a list of a symbol which is the function name and an integer integrating the number of arguments for the function.

The "./" here is pointing towards the current directory. The .so file is currently stored in the current directory. It can be stored in LD_LIBRARY_PATH and "./" can be ignored when loading the .so file.

Start and end connection

To start a connection to the TCP server, specify the arguments for the "conn" function and run it. The arguments are the ipaddress of the server, the port number of the server and the name of callback function.

To end the connection, use the "disconn" function.

```
wooikent@wooikent-VirtualBox:~/C_Prac/echoserv2$ ./echoserv 5625
Tue Feb  5 13:40:10 2013
: Connection opened
Tue Feb  5 13:40:10 2013
: sent 5a7nCr
 Tue Feb  5 13:40:11 2013
: sent 7szy4BDS
 Tue Feb  5 13:40:12 2013
: sent 3JzM
 Tue Feb  5 13:40:13 2013
: sent 3RV1
 Tue Feb  5 13:40:14 2013
: sent 5zlvWO
wooikent@wooikent-VirtualBox:~/C_Prac/echoclnt2$ q echoclntq.q
KDB+ 2.8 2013.01.04 Copyright (C) 1993-2013 Kx Systems
l32/ 1()core 495MB wooikent wooikent-virtualbox 127.0.1.1 PLAY 2013.04.04

e.g. conn[`127.0.0.1;5625;`upd] to start connection
e.g. disconn[] to disconnect
upd and oneerror function are defined
-1
q)conn[`127.0.0.1;5625;`upd]
q)"13:40:10.466: a7nCr"
"13:40:11.468: szy4BDS"
"13:40:12.469: JzM"
"13:40:13.470: RV1"
"13:40:14.471: zlvWO"
disconn[]
```

5 C code explanation

5.1 TCP Server

Random String Generator

To generate random string containing alphanumeric characters and numbers, a reference can be obtained from http://stackoverflow.com/questions/440133/how-do-i-create-a-random-alpha-numeric-string-in-c. This function creates a character list containing alphanumeric characters and numbers and put the random character into the specified character's array (depending on how many characters you want to put in).

The code works fine, but it would be ideal to generate random string with random length! So the "len" input is excluded and random integer between 1 and 10 is generated as the length of the string.

```
void gen_random(char *s){

	int i;

	static const char alphanum[] =
	"0123456789"
	"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
	"abcdefghijklmnopqrstuvwxyz";

	/* Create random string between length 1 to 10 (rand()%10+1)*/
	for ( i = 0 ; i < rand()%10+1 ; ++i) {
		s[i] = alphanum[rand() % (sizeof(alphanum) - 1)];
	}

	s[i] = 0;
}
```

Create a socket

First the TCP server will create a listening socket. Then it will setup its socket address structure (i.e. the ipaddress and port number) and bind it to the listening socket. Lastly, it will call listen() function to activate the listening socket.

```
    /*  Create the listening socket  */

    if ( (list_s = socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
       fprintf(stderr, "ECHOSERV: Error creating listening socket.\n");
       exit(EXIT_FAILURE);
    }


    /*  Set all bytes in socket address structure to
        zero, and fill in the relevant data members   */

    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family      = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port        = htons(port);


    /*  Bind our socket addresss to the
       listening socket, and call listen()  */

    if ( bind(list_s, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0 ) {
       fprintf(stderr, "ECHOSERV: Error calling bind()\n");
       exit(EXIT_FAILURE);
    }

    if ( listen(list_s, LISTENQ) < 0 ) {
       fprintf(stderr, "ECHOSERV: Error calling listen()\n");
       exit(EXIT_FAILURE);
    }
```

Accepting connections

The TCP server use accept() function to accept any connection from the client.

```
/*  Wait for a connection, then accept() it  */

if ( (conn_s = accept(list_s, NULL, NULL) ) < 0 ) {
     fprintf(stderr, "ECHOSERV: Error calling accept()\n");
     exit(EXIT_FAILURE);
}
```

Message Structure

Structure of the message is important so that the client knows how to read the message sent by the server. The message structures sent by this TCP sever is as such (i.e. "5hello\n"):

Size Indicator	Message	End of Message
1 byte	*n* bytes	'\n'

```
/*  Put the random string into the message structure  */
gen_random(buffer);
sprintf(bufferR,"%i%s\n",strlen(buffer),buffer);
```

5.2 TCP Client

Parameters type

To make sure the parameters are in the correct format, the inputs can be converted into type format and check them accordingly.

```
/*  Inspect each K object for type */

if ( -6 != portq->t ){krr("porttype");return (K)0;}
PORT = portq->i;
if ( -11 != ipaddressq->t ){krr("iptype");return (K)0;}
IPADD = ipaddressq->s;
if ( -11 != callback->t ){krr("cbtype");return (K)0;}
CALLBACK = callback->s;
```

Create a socket and connect to the TCP server

First the TCP client will create a connection socket. Then it will setup its socket address structure (i.e. the ipaddress and port number). Lastly, it will call connect() function to connect to the TCP server.

```
/*  Create the listening socket  */

if ( (conn_s = socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
       krr("nosocket");
       return (K)0;
}

/*  Set all bytes in socket address structure to
    zero, and fill in the relevant data members   */

memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family      = AF_INET;
servaddr.sin_port        = htons(PORT);

/*  Set the remote IP address  */

if ( inet_aton(IPADD, &servaddr.sin_addr) <= 0 ) {
       krr("ipaddr");
       return (K)0;
}

/*  connect() to the remote echo server  */

if ( connect(conn_s, (struct sockaddr *) &servaddr, sizeof(servaddr) ) < 0 ) {
       krr("noconnect");
       return (K)0;
}
```

Creating Listening Callback

Remember that the TCP server sends a message every one second. So the TCP client needs to go into a loop so that whenever TCP server sends a message, the callback function will be triggered so that the response will be passed into q. sd1(handle,function) command is useful in this case.

```
/*  Retrieve response everytime the server send something  */

sd1(conn_s,rtnmsg);
```

Receiving messages

Now the object to be returned to q must be in K type, a K function (rtnmsg) is created that receive the response from TCP server, put it in K format and send it to the q session.

Remember the message structure from TCP server, the function reads the first byte of the response as the size of the message then read the rest of the message according to the size.

The function also checks where the message is complete or not.

After reading the message, the function sends it to q using "k(handle,function,params,(K)0)" command. Note that "function" here is in q format and the "params" has to be in K format. A C coded string can be converted into K format using "kpn" command.

```
/* A function to retrive response from server */
K rtnmsg(){

       /* Put the first byte into c to let us know how what length the message is */
       char c[1];
       if ( read(conn_s, c, 1) <= 0 ){
              onerror("nobyte");
              sd0(conn_s);
              close(conn_s);
              return (K)0;
       }
        int len = atoi(c);

       /* Read the message into bufferR */
       if( read(conn_s, &bufferR, len) <= 0 ){
              onerror("nomsg");
              sd0(conn_s);
              close(conn_s);
              return (K)0;
       }

       /* Read the end of message return an error if message invalid */
       char suc[1];
       read(conn_s, suc, 1);
       if ( *suc != '\n' ){
              onerror("msgntcmplt");
              sd0(conn_s);
              close(conn_s);
              return (K)0;
       }

       /* Send the response to Q */
        k(0,CALLBACK,kpn(bufferR,len),(K)0);

    return (K)0;

}
```

Disconnect from TCP server

Using sd0 function, TCP client will stop receiving messages from TCP server.

```
/*  Close the connection to TCP server  */
K disconn(K x){

    sd0(conn_s);
    close(conn_s);
    return (K)0;

}
```

Error Alerts

To make sure the q session receives an alert when there is an error occurred during the message transmission, onerror function is created to send an error string to q session for further use.

```
/*  Send an error message to q session  */
void onerror( char *message)
{

       k(0,"oneerror",kpn(message,strlen(message)),(K)0);

       return;

}
```

For example, when the server is killed and there is no message sent to the client, then "onerror" function will be triggered.

```
wooikent@wooikent-VirtualBox:~/C_Prac/echoserv2$ ./echoserv 5625
Tue Feb  5 14:04:21 2013
: Connection opened
Tue Feb  5 14:04:21 2013
: sent 5a7nCr
 Tue Feb  5 14:04:22 2013
: sent 7szy4BDS
 Tue Feb  5 14:04:23 2013
: sent 3JzM
 Tue Feb  5 14:04:24 2013
: sent 3RV1
 Tue Feb  5 14:04:25 2013
: sent 5zlvWO
Killed
wooikent@wooikent-VirtualBox:~/C_Prac/echoclnt2$ q echoclntq.q
KDB+ 2.8 2013.01.04 Copyright (C) 1993-2013 Kx Systems
l32/ 1()core 495MB wooikent wooikent-virtualbox 127.0.1.1 PLAY 2013.04.04

e.g. conn[`127.0.0.1;5625;`upd] to start connection
e.g. disconn[] to disconnect
upd and oneerror function are defined
-1
q)conn[`127.0.0.1;5625;`upd]
q)"14:04:21.551: a7nCr"
"14:04:22.556: szy4BDS"
"14:04:23.558: JzM"
"14:04:24.559: RV1"
"14:04:25.560: zlvWO"
"14:04:26.085: nobyte"
```

Another error alert is when a parameter is specified incorrectly or there is something wrong with the connection to TCP server, krr() function will be triggered.

```
wooikent@wooikent-VirtualBox:~/C_Prac/echoclnt2$ q echoclntq.q
KDB+ 2.8 2013.01.04 Copyright (C) 1993-2013 Kx Systems
l32/ 1()core 495MB wooikent wooikent-virtualbox 127.0.1.1 PLAY 2013.04.04

e.g. conn[`127.0.0.1;5625;`upd] to start connection
e.g. disconn[] to disconnect
upd and oneerror function are defined
-1
q)conn["127.0.0.1";5625;`upd]
'iptype
```

```
wooikent@wooikent-VirtualBox:~/C_Prac/echoclnt2$ q echoclntq.q
KDB+ 2.8 2013.01.04 Copyright (C) 1993-2013 Kx Systems
l32/ 1()core 495MB wooikent wooikent-virtualbox 127.0.1.1 PLAY 2013.04.04

e.g. conn[`127.0.0.1;5625;`upd] to start connection
e.g. disconn[] to disconnect
upd and oneerror function are defined
-1
q)conn[`127.0.0.1;1234;`upd]
'noconnect
```

6 Possible Extensions

The example could be easily extended to support the sending of message to the server. Eg, the initial message could contain logon information encoded as a string. This could be achieved using the same methodology in conjunction with the "send" function.

http://www.paulgriffiths.net/program/c/echoclnt.php is a good reference where a connected client can send message to a server.

7 References

For more information about interfacing and extending with C, please refer to the following links:

http://code.kx.com/wiki/Cookbook/InterfacingWithC

http://code.kx.com/wiki/Cookbook/ExtendingWithC