MOK — System and Protocol Description

Mikael Cardell, mc@hack.org

Last changed 1994–12–14

About This Document

Note in 2003 to late readers: This is a document about a system I was designing in 1993 and 1994. It was the latest in a long line of clones of the KOM system I had written, but the first to use the client/server paradigm and with a novel idea about using a possibly programmable server using the Forth programming language. A working server was finised, with some test clients working, but nothing was used in production.

Please note that this is a draft specification. There is no guarantee that the finished system will look like the one described here. Everything in this document is subject to change. Refer to the author for the latest version.

Contents

Introduction

This document describes the the protocol used both in server to server and server to client communications in the MOK conference system. To set the MOK system in proper context, the document also includes a description of the philosophy and history of the Swedish originated KOM conference systems.

More to come...

A Short History of KOM

The KOM systems are a whole family of different conferencing systems with one thing in common; the user interface. They all derive from one program, called KOM, that was written in MACRO-10 by Torgny Tholerus for the PDP-10 way back in the 70s. The orginal idea of the KOM system came to Torgny as he was editing his paper zine *Anarkisten* (Swedish. English: *The Anarchist*) which was an open bulletin style zine — anyone could send in texts and everything was guaranteed to be published. Since Torgny was the sole editor, there was a lot of work and he could not keep up with the texts that kept coming in.

When he eventually started working as a programmer at QZ, the Stockholm Computer Center, he wrote the original KOM system. The system allowed a user to subscribe to any number of conferences, which could hold a large number of text entries ordered in subject threads where single entries within a subject had reply-links to each other. The system kept track of the number of entries in these conferences the user had read and presented the entries to the user in threaded order instead of chronologically.

The user was first presented to an original entry in a subject, the one starting the subject thread, then the first reply to this text and the reply to this reply. Then the system went back and checked if there are more replies to the original text and showed them in the same way. As you can see, it was some kind of recursive thread reading.

This was all new back then, but can be found in a lot of news readers and BBS programs nowadays. However, the KOM user interface was also something new. Or rather, it was the user interface of the TOPS-20 operating system transfered into a conferencing system. In this interface the user could execute any command at all times; there were no different levels within the system and all commands were instantly available. There were completion on all commands and parameters, so a user just had to type as much characters necessary to make the command unique in the current context to have it executed.

Instead of showing menus the KOM system suggested a command for the user. For instance, if the user just had read an entry with replies, the system would suggest that the user read the next reply, like this:

```
(Read the) next reply -
```

The '-' meaning that it expects you to either press return, to accept the suggested command, or to give another command, any command. The command executed if you do press return is called '`next reply`', but the words within paranthesis are added for clarity in the suggestion. If you instead decide to give the command '`skip`' at this moment, all the replies to the entry you just read will be marked as read and you will be presented with a new text and a new prompt.

This is a small made-up session log describing how a typical KOM session might have looked like:

```
   What is your name?
= mi c

   Mikael Cardell Lysator
```

```
   Password:
   You have 1 unread letter.
You have 2 unread entries in forum.
You have 6 marked entries.

   There are 22 other people present.
   Mikael Cardell Lysator - 1 unread.
   (Read the) next letter - <CR>
   Text 4711, 1993-01-17 23:41, from Helena Cardell
Recipient:  Mikael Cardell Lysator
Subject:  Babies

   You are now the father of a sweet little boy!
   (4711)
   (Go to the) next conference - <CR>
   Forum - 2 unread.
   (Read the) next entry - <CR>
   Text 114711, 1993-01-16 14:52, from Lars Aronsson
A reply to text 17 by Lars Willfor
Recipient:  Forum
Recipient:  The nonsense conference
Subject:  Nothing in particular

   Just writing along...
   (114711)
Reply by Lars Willfor in text 26262

   (Read the) next reply - <CR>
```

The original KOM was immensely popular among computer scientists and military researchers at the Stockholm University and the Swedish Military Research Agency (FOA). Later, when the system was opened to the public a lot of people from the early personal and home computer clubs learned to love KOM. The system was then released to the academic community free of charge and was spread to a number of large universities in Sweden.

So, it came to be that a lot of people liked the original KOM and tried to "port" it to micro computers so they could start a KOM of their own. Of course, this were no real ports, but re-writes in a high level language. One of the first "ports" were, naturally, called MicroKOM and ran under CP/M. Later, KOM systems appeared for MS-DOS, and even later some appeared for Amigas.

KOM systems are quite popular in the Swedish BBS community. In fact, a lot of BBS sysops thinks it is impossible to get some discussion going in a menu-based system. And it *is* a lot harder to feel inclined to post something in a menu system where you have to wade through 17 menus before you can get to the message menu.

When the last PDP-10s (DECSYSTEM 20s) in Swedish computer science departments was crapped during the late 1980s, early 1990s, the Swedish hacker community, desperately started looking for new KOM systems to use. Of course, the small one-user systems on personal computers simply would not do. At that moment some of the members of the academic computer club

Lysator at Linköping University started hacking on a Unix based KOM system.

The Lysator KOM system, LysKOM, was working in 1991 and it was based on a server/client based structure. The first working client was written in EMACS Lisp and has grown very big (and slow) over the years. There *are* other clients available too; a TTY client written in C and another (nilkom) written in C++ and TCL, two different X Window clients, tkom (written in C++ and TCL/Tk) and LyXKOM. There is also a semi-working perl client.

A commercial venture, KOMmunity Software, also started working on a Unix based KOM system, but the result was *a)* extremely expensive and *b)* done the Wrong Way — it did not resemble the original KOM very much in spirit.

However, both these systems had a major drawback — they could not support ordinary Internet mail or USENET News. So, rather recently some hackers from Stockholm hacked together something they called SklaffKOM. It is a real hack, but it can be said to be a news reader and mail program with a KOM interface.

This is the background to my own project, MOK, and the reason it looks the way it does.

General System Description

MOK is a generic publishing system, merging traits from systems such as MUD, Internet Relay Chat and the original KOM conference system. The system consists of a server program with a database and several client programs.

The server database holds generic information objects, or nodes, that can have several pieces of data "glued" to them. The nodes are combined to each other in a network of links, where the links determine the nature of their relationsship. Links between nodes in the server database can be of several different types; from the most simple bytecount into a local data file to Universal Resource Locators pointing to remote data.

The data that is glued to the nodes can be said to be the equivalent of text entries in the original KOM. The nodes themselves, however, is something entirely new. A node can hold several original KOM entries and they can be of any type.

Every user has a node of her own. This node is the connection between the physical user and the node network of the server database — it holds all the links to nodes of any importance to the user, i.e. subscribed nodes. The user can subscribe to and unsubscribe from *any* node in the database.[1]

Users are visible to each others within the system and can directly communicate with each other when they are present in the same "place".[2] Realtime discussions can take place in named channels, as in the Internet Relay Chat or in MUDs, and directly between users.

Data can be relayed from external systems, such as IRC, with the use of special relay clients. Such clients are also used to import and export ordinary Internet mail and USENET News as well as relaying data from systems such as FTP, Gopher and the World Wide Web.

[1] Some nodes, higher up in the node network can be implicitly subscribed when the user subscribes to a lower node, somewhat depending on the links between them. The user can then unsubscribe from these nodes, although never having explicitly subscribed to them.

[2] See below for more information about this and the definition of "place".

General Server and Database Description

A single node in the database contains two parts — a header and a body. The header contains information such as time of creation and time for last change for node data. The body is filled with data links.

A link in the node body also contains two parts — link type and link data. The type tells the system how to treat the link data. It can be just about anything; a simple bytecount into a local datafile, a node ID for another node in the database or a Universal Resource Locator, pointing to remote data.

The nodes in the server database are linked to each other in named trees. A single node in such a tree has links only to its branches. There can, however, be any number of branches.

The user subscription, which incidentally is just another type of link in the user node, stores, among other things, the highest node ID of the subscribed tree when the whole tree last had been requested. Since a user can request any node in the subscription at any time, there is also an array of node IDs to keep track of what has been read. When the entire tree has been requested the highest node ID is updated.

The subscriptions of a user, together with information about what the user currently reads, tells the server where the user is present, if currently logged into the system. Present, in this context, means visible to other users.

If, for instance, the user is subscribed to a node within a node tree, but is not subscribed to the root node, that user will only be present in the branch above the subscribed node. The user will not be visible to another user currently present in the root node. Seen the other way around the user will not notice other users as present as long as they are not reading that particular thread, even though they are present in the root node of the tree.

Server and Client Connection

The client and the server can speak to each other through any two-way eight bit channel. The client may be connected through a reliable TCP stream to the server program but may also be connected via a dialup connection without any underlying protocol what so ever. All data on the channel is sent as ASCII text even though all eight bits might be carrying data in `HH` strings.[3]

When the client request local data to be transferred, it is sent as packets that can have any size. The size depends on the connection and if the client wants to continue requesting and sending other data. It is up to the client to decide the size of the packet in the request, but this can be changed once the transfer has begun. Each packet contains information about sequential order and how many packets there are left to be sent. The client will know what packets belongs to which requests by the word reference number.

The transfer of remote data is entirely up to the client, which has to connect to a remote server and request the data from there, using the Universal Resource Locator provided by the local server to determine the way to do it. If the client cannot understand the way to get the remote data it is suggested that the client present the URL to the user for decision of further actions.

[3] See below for more on this.

Protocol Specification

The protocol spoken between client and server in the system is based on Forth syntax. In fact, in a future version there will probably be a possibility to send entire Forth programs to the server for evaluation.

The description of the protocol follows certain rules. All Forth words are written in the following style:

```
COST ( inventory --- price )
TIME ( timezone --- hour )
NAME ( --- nameaddr count )
```

This means that a word, say `COST`, expects an integer value on the stack and returns another integer value. Another word, `NAME` expects nothing on the stack, but returns two values, an address containing data and a number. I will use the suffix *-addr* whenever the value involved is a memory address. All other values are considered integers.

The protocol is made up of Forth words. Everything else sent are numbers or characters that stand for themselves. However, for readability there are a number of conventions used throughout this document.

An `ALTERNATE` set is a set of values that are given names in the protocol for ease of reading. The set provides a list of values where exactly one can be used at a time. The alternated sets are listed like this:

```
ALTERNATE FOO 0=HELLO ( addr count )
    1=HOWDY
```

This does not mean that a string of characters is sent; it is just a name for the value 0. Please observe that the arguments, `addr` and `count`, are sent *before* `HELLO`.

There are also ordinary sets of data, used as a convention to give a common name to several numbers. These are identified with the `SET` keyword.

Constructed Datatypes

Constructed data types are provided for clarity to the reader of this specification and does not effect what is actually sent.

`time` is given as a string according to ISO standard with the offset from Universal Time given. August 11, 1972 9.50pm Swedish time would for instance be expressed as:

```
19 HH 19720811215000+0200
```

`node-no` is a non-negative integer that provides a unique identifier for each node in the system.

```
ALTERNATE person 0=LOCAL ( node-no )
    1=REMOTE ( nameaddr count )
```

The `nameaddr` is an address where the string that contains the persons e-mail address is stored. The value of `count` is the length of the string.

```
SET subscription nodeaddr count ( Array of nodes allready
   requested )
    highest ( The highest requested node, except for the
   above )
    priority ( Subscription priority )
    subscription-for ( Node this is a subscription for
   )
    visited ( Time last visited )

ALTERNATE linktype 1=RCPT
    2=CC-RCPT
    3=PARTICIPANT
    4=REPLY-TO
    5=FOOTNOTE-TO
    6=ENTRY
    7=REPLY
    8=FOOTNOTE
    9=SUBSCRIPTION
    10=DATA
    11=URL

ALTERNATE datatype 1=ASCII
    2=ISO8859-1
    3=GIF

ALTERNATE link ( Links directed to this node )
    1=RCPT ( person )
    2=CC-RCPT ( person )
    3=PARTICIPANT ( person )
    4=REPLY-TO ( node-no )
    5=FOOTNOTE-TO ( node-no )
    ( Links directed from this node )
```

```
    6=ENTRY ( node-no )
    7=REPLY ( node-no )
    8=FOOTNOTE ( node-no )
    9=SUBSCRIPTION ( subscription )
    ( Local data )
    10=DATA ( position length datatype )
    ( Remote data )
    11=URL ( stringaddr count )
SET node changed-time ( The time the node was changed )
    created-time ( The time the node was created )
    nameaddr count ( Address and length of the node name
      )
    creator ( Link to the creator of the node )
    node-no ( ID of the node )
SET packet dataaddr count ( Adress and length of the packet
      data )
    left ( Number of packets left to be sent )
    number ( Serial number of this packet )
SET serverinfo started ( The time the server was started
      )
    protocol ( The protocol being used )
    version ( The server version )
```

Dialogue

A session starts, logically enough, with the client requesting to connect to the server. This is done by identifying the protocol being used and telling the server that someone would like to connect. It is the client, or a server in client mode that sends this request. It is done like this:

```
KOMCONN ( protocol nameaddr count --- )
```

What is sent is the protocol version being used and then the address and the length of a string containing a user name. This means that what actually is sent could be:

```
0 2 HH mc KOMCONN
```

Note that the user name is the name of the user at the client machine. This does not necessarily have anything to do with the name of the user mailbox in the conference system. The current protocol version, this protocol, is zero (0).

The request should be answered with a simple connection confirmation, like this:

```
OK
```

When the client receives this, the connection is established. Everything that is being sent on the channel will be Forth words and numbers. The client is allowed to send any number of requests before receiving an answer.

To keep things in order requests are counted by the server as they come in. The order the Forth words reach the server may be considered the same order they were sent out from the client, so the number of words sent from the start of a session, starting at one, identifies an answer with the corresponding request.

There are words to tell if a call went wrong or if it succeded. These are:

```
OK ( n --- )
ERROR ( error n --- )
```

They are used in the following way (client calls first):

```
1 FOO 17 BAR
1 OK 2 OK
```

The server works by threading certain requests off to a process of their own. Such requests are known as forking words and because of this it is very important that a client keeps track of what is sent out and in what order. The `OK` of the second word above can be received *before* the response for the first word.

Server Words

`.  ( n --- )` Dot sends back the value on top of the stack.

`.DATA ( size position --- )` Fetches data with a specified packet size from a the position requested. Sends back datapackets identified with the reference number of the request.

`.ID ( nameaddr count --- )` Print ID takes an address to a string and its length as arguments and sends back all the nodenames that matches that string.

`.LINKS ( link node-no --- )` Print links takes as an argument the first link of those that we request and the number of the node the links can be found in. If the link is a nil value, except for the linktype, all the links of that type will be printed.

`.NODE ( node-no --- )` Print node takes a node ID as an argument and sends back the node.

`.ON ( --- )` Print (who is) on sends the user IDs of the people logged on to the system.

`.PRE ( node-no --- )` Print present sends back the user IDs of the people currently present in `node-no`.

`.SINFO ( --- )` Print server information sends back a `serverinfo` with some information about the current server version.

`.TIME ( --- )` Print time tells the current time, according to the server.

`.WHO ( --- )` Print who (I am) sends back the user ID of the current user.

`AY ( n --- dataaddr count )` `AY`, for "array", is a word used to signify that directly following `AY` there will be `n` number of numbers in the input buffer. `AY` allots `n` cells to store these and reads them as they come and returns the address and the actual number of numbers stored.

`DATA! ( dataaddr count --- position )` Stores data found at address with the length `count` and pushes the position (bytecount) in a local datafile.

`HH ( count --- stringaddr realcount )` `HH` is a word that expects the length of a string on the stack. Directly following `HH` the string of the specified length should be sent. `HH` allots `count` bytes for the string and returns both the address and the number of alloted bytes.

`HIGH! ( highest node-no --- )` Stores `highest` as a new value of the highest read in the subscription regarding node `node-no`. This marks everything under node *highest* as read.

`LINK+ ( link node-no --- )` Adds a link to a node.

`LINK- ( link node --- )` Subtracts a certain link from a specified node.

`LOGIN ( passaddr count node-no --- )` Login expects three arguments, first a password string and its length, and then a node number to associate the password with.

`LOGOUT ( --- )` Logout disconnects a user and cannot fail.

`MSG ( dataaddr count node-no --- )` Message sends a message to another node. It expects three arguments, an address to a string, its length and the node.

`NAME+ ( nameaddr count --- )` Add name adds a new name to the name database.

`NAME- ( nameaddr count --- )` Subtract name subtracts a name from the name database.

`NODE! ( node --- )` Stores an allready created node.

`NODE+ ( creator nameaddr count extranameaddr count --- node-no)` Add node expects only the `node-no` of the owner or creator of the new node and the name and extra name of the node to be created.

`NODE- ( node-no --- )` Delete node markes a node as removed.

`PAS! ( newpassaddr count oldpassaddr count node-no --- )` Store password wants five arguments, first an address to a new password and its length, then the same for the old password and finally the node ID.

`PRE+ ( node-no --- )` Makes the user present within a specified node.

`PRE- ( node-no --- )` Subtracts the user's presence from the specified node.

`SHUTDOWN ( --- )`

`STOP ( ref-no --- )` Stops the transmission associated with the reference number `ref-no` at once.

`TPAK! ( packsize ref-no --- )` Stores a new value for the transmission packet size associated with `ref-no`.

`WID! ( stringaddr count --- )` Stores a new string as "What I Do".

Client Words

At any time the *client* must be prepared to deal with incoming requests that are not asked for. These requests are made up of Forth words, just like the requests the client sends to the server.

`AY ( n --- arrayaddr count )` `AY`, for "array", is a word used to signify that directly following `AY` there will be `n` number of numbers in the input buffer. `AY` allots `n` cells to store these and reads them as they come and returns the address and the actual number of numbers stored.

`CROWD ( --- )` This means that the maximum number of logins at the same time has been reached.

`FORCE ( node-no user )` You were forced to leave a node by another user. The arguments are the node and the user ID in that order.

`HH ( count --- stringaddr realcount )` `HH` is a word that expects the length of a string on the stack. Directly following `HH` the string of the specified length should be sent. `HH` allots `count` bytes for the string and returns both the address and the number of alloted bytes.

`UIN ( user )` A user, with ID `user` just logged in.

`UOUT ( user )` A user, with ID `user` just logged out.

`MIN ( msgaddr count sender recipient )` This is an incoming message. The arguments are a string and its length, the ID of the sender and the ID of the recipient, which not necessarily is a user node ID.

`NNEW ( node-no )` A new node has been created.

`NDEL ( node-no )` A node has been deleted.

`NNAME ( nameaddr count node-no)` The node with ID `node-no` has changed name to the string found in `nameaddr` with the length `count`.

`WID ( stringaddr count present-in user )` A user has changed her What-I-Do-string. The new string is found at `stringaddr`. The user is present in the node identified by `present-in`.

Protocol Examples

This is an example on how a dialoge between a client and a server might look like. Directly after the examples I have provided some explanations.

```
0 2 HH mc KOMCONN
OK
```

Request to login. We're using protocol 0 and I'm called "mc" where we are at the moment.

```
14 HH Mikael Cardell .ID
1 AY 17 2 OK
```

Lookup the name "Mikael Cardell" and see if you can find it in your database. The server replies that there is only one node with that name and that the ID of that node is 17.

```
3 HH foo 17 LOGIN
17 UIN
4 OK
```

Login as ID 17 with the password "foo". The server tells us that ID 17 just logged in, which it tells every user currently logged in, and then that our login request was succesfull.

```
17 23 HH Foo experience exchange 0 HH NODE+
18 NEWNODE
18 7 OK
```

Request to create a new node called "Foo experience exchange" with no external name. The server tells us that a new node has been created and then tells us that our request has gone through and that the ID of the new node is 18.

```
0 AY 0 0 18 9 17 LINK+
9 OK
```

Add a subscription to ID 18 for ID 17, the ID I logged in as. I am now subscribed to the newly created node.

```
15 HH Welcome to Foo!  DATA!
4711 11 OK
```

Store away some data in the local datafile. The response tells me where to find the data (at position 4711).

```
4711 10 18 LINK+
12 OK
```

Create a new textlink from node 18 to the data I just created. The introduction, the first text pointed to from node 18, is now available for others to read.