

The Unicon Messaging Facilities

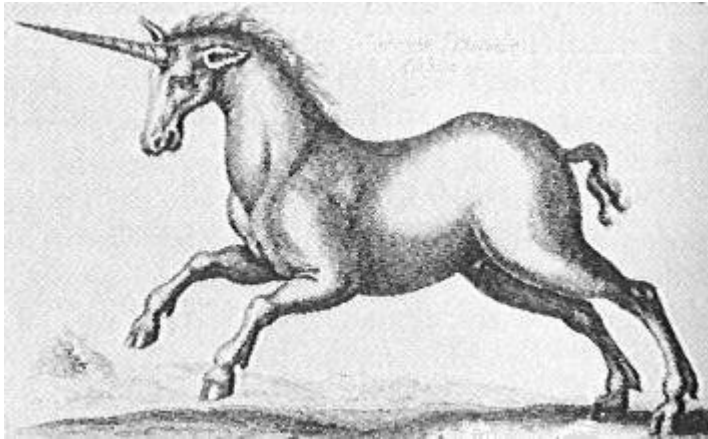
Steven E. Lumos

November 1, 2008; last edited 3/12/2010

Unicon Technical Report UTR #13

Abstract

A messaging language provides highly-integrated connectivity (networking) along with context sensitivity. This work explores the implementation of messaging in a more traditional language by adding messaging language features to the Unicon programming language. The resulting system makes it easy to write Unicon programs that take advantage of Internet resources by leveraging programmer intuition.



<http://unicon.org/utr/utr13.pdf>

Information Science
Research Institute
University of Nevada, Las Vegas

1. Introduction

The Unicon messaging facilities are part of the system interface that support high-level Internet programming by hiding Internet protocols and servers behind an expanded version of the standard file abstraction. The implementation of the messaging facilities does not add any functions or operators to Unicon. Instead, a new form of the file type is introduced, and existing functions and operators for dealing with files are extended to operate on the new subtype. Additionally, certain non-file operators have been extended to operate on messaging files in cases where it is appropriate or natural to do so.

This report was prepared by the author as user documentation. It was then edited by Clint Jeffery. The editor apologizes to the author for lost formatting and any other errors contained within. The interested reader is referred to the author's M.S. Thesis and the book Programming with Unicon for more extended discussions and examples of the use of these facilities.

2. A Short Example

A major goal of the messaging facility is to leverage the intuition of Unicon programmers. A Messaging program can be as simple as:

```
procedure main(args)
  f := open(args [1], "m")
  every write (! f )
  close ( f )
end
```

This program should be completely familiar to a Unicon programmer with the exception of "m" being passed as the mode argument to open(). This causes open() to interpret the filename argument as a URL instead of a filename. Differentiating between URLs and filenames in this way is necessary for correctness, since it is possible for a URL to also be a legal filename and there is no other behavior that will avoid unwelcome surprises in certain (admittedly pathological) cases.

The Messaging Facility also attempts to remain general whenever possible. The above code will work equally well for HTTP and Finger, and any future protocol which operates in a similar open-and-read mode. Unfortunately, we cannot say that this code works for all protocols, since there are protocols where choosing semantics for the above code is either impossible or inconvenient (i.e. a clear and intuitive interpretation could not be found).

3. The Importance of Explicit Close

Usually, a very high level language like Unicon is expected to automatically manage resources such as file handles. In particular, many programmers are accustomed to opening files and relying on the

runtime system (which may actually be relying on the operating system) to properly flush and close the files when variables go out of scope or at program termination. Unicon currently does the latter, implicitly closing files only at program termination, which can be a problem for certain protocols. A POP connection for example caches delete requests and only performs the actual deletions following a clean disconnect. In the current SMTP implementation, messages are only sent when a close request is made (we use close to signal the end of the message). Another problem that arises is certain protocols which are designed to handle only a single client at a time. When a POP connection is made, the mailbox being requested is locked, and other requests for the same mailbox are refused until the current connection is closed. If the program waits until exit to close a POP connection, the user will not be able to connect with any other program (or even the same program if it causes a POP connection to be made more than once during a single execution).

For these reasons it is recommended that all Messaging files be explicitly closed when they are no longer needed. Even for protocols such as HTTP where the server automatically closes its side of a connection after sending the response, the TCP layer underlying the protocol requires action from the client before the connection is truly closed.

4. Messaging Direction

One of the attributes held by a messaging file is direction, which is primarily dependent on the type of resource (protocol) being used. The possible directions are read-only, write-only, read-write, write-read, and none. The first three directions have the obvious meaning, the last is used for cases where all message data is accessed without using any of Unicons read or write operations. Since Unicon presents a POP mailbox as a list of messages accessed via the list operations, it has a direction of none. Write-read specifies that the the program is expected to do a number of writes followed by a number of reads, but never interleaving writes and reads. Certain cases of HTTP have write-read direction.

In the write-read case, the runtime has to be told when all writing has been completed so that it can notify the server. Instead of adding an explicit operation for this, any operation which would require a response from the server (e.g. read(F) or even F[" Status -Code"]) implicitly causes the runtime to finish the request and transition into the reading stage.

Direction specifies the operations allowed on a given messaging file and attempting to apply any other operation results in a runtime error.

5. Protocols

The current implementation supports the three most popular Internet protocols: HTTP, POP, and SMTP, and also Finger, which was implemented as a simple test case for the protocol library architecture. Although the following sections include some information about these protocols, the relevant RFC (Request for Comments) documents referenced should be consulted for complete and authoritative descriptions. It is also a good idea to at least scan the RFCs for a protocol you intend to use with the Messaging extensions, to get an idea of what to expect.

5.1. Finger

```
open("finger ://< host>[:<port >][[/ w ]<username>]", "m")
```

Finger is a simple user information protocol. A typical Finger server will report the real name of a user, whether they are currently logged in, and optionally append the contents of a file that the user creates (known as a “dot-plan” file). Since there is no standard URL for Finger requests, the Messaging Facility uses the form described in a draft standard[?], which is unfortunately expired but currently the closest thing to an authoritative description available. The optional /w component of the URL requests a “higher level of verbosity” from the server, but the exact meaning is not specified by the standard and varies among implementations. Finger queries are read-only operations. Examples:

```
open("finger :// nevada.edu/slumos", "m")
open("finger :// nevada.edu // w slumos", "m")
open("finger :// nevada.edu:79")
```

5.2. HTTP

```
open("http ://< host>[:<port>][/<path>]", "m[s]")
```

The Hypertext Transfer Protocol (HTTP) is the protocol of the World Wide Web. Using HTTP, a Unicon program can access a wide range of resources: static data files, online databases, and applications. HTTP messaging files can have any of the possible directions, although there is no specific support for write-only in the current implementation. The most common case is a read-only request initiated by a call to open() such as M := open("http :// icon . cs . unlv . edu/index .html", "m"). The contents of the index.html can be read using any of the normal read operations, and the header of the response may be retrieved by using the Unicon table operations on the messaging file (e.g. M["Date"]).

A Unicon program may simulate a user filling out an HTML form in two ways. The first is to encode the values of the fields in the URL (see [?]). The second is to send the fields to the web server as a request body, and is an example of write-read direction. To select write-read mode, the program must include a “Content-Type” header when the URL is opened. If the value of Content-Type contains the string “form” (e.g. “multipart/form-data”) then the HTTP POST method is used, otherwise PUT is used. The POST method is usually used for processing HTML forms, although it is sometimes used for file upload as well, while PUT is intended specifically for file uploading. In either case, the programmer is responsible for any necessary encoding (see mutils.icn).

If an opening mode of “s” is given in addition to “m”, a “short” request is performed. In this case the HTTP HEAD method is used, which causes the server to return only the header part of its response. In this case the connection has direction none, since the header fields are queried as table values.

One interesting feature of HTTP is that a server can notify the client when a resource has been moved to some other location. When this is done, the server will set the Status-Code to 301, 302, 303, or 307, and put the URL of the new location into the “Location” header field. The following program shows how to use this information to automatically fetch the URL at it’s new location:

```
procedure main(args)
```

```

if *args < 1 then stop("usage: ", &progrname, " url ")
# Connect to the host specified in the URL, sending some custom
# header fields .
f := open(args [1], "m",
    "User-Agent: Unicon Grab 0.0",
    "X-Unicon: http://icon.cs.unlv.edu/") |
stop(args [1], ": can't open")
repeat {
if f ["Status -Code"] < 300 then {
    # If the server returns a successful status code, read in the
    # result 64k at a time and write it out .
    while writes (reads(f , 65535))
    exit (0)
}
else if f ["Status -Code"] < 400 & \f["Location"] then {
    # If the server returns a 3xx error , check for a Location:
    # header and follow if found.
    newloc := f ["Location"]
    close ( f )
    f := open(newloc, "m",
        "User-Agent: Unicon Grab 0.0",
        "X-Unicon: http://icon.cs.unlv.edu/") |
    stop(newloc, ": can't open")
}
else {
    # Some other error , so tell the user what the server told us .
    stop( f ["Status -Code"], " ", \f["Reason-Phrase"] | "")
}
}
end

```

5.3. SMTP

```
open("mailto:<user>@<domain>", "m")
```

SMTP support allows Unicon programs to send messages to Internet email addresses. The destination email address is given as e.g. `mailto:unicon-group@cs.unlv.edu`. Extensions to the `mailto:` URL to specify header fields such as subject are not supported, this is better done using the header mechanism already in place.

In order to construct and send an email message, the runtime must know the address of the user sending the message, and the name of a SMTP server to connect to. Since both of these would be inconvenient to specify in the program, they are taken from two environment values, both of which have default values. `UNICON SMTPSERVER` should be set to the user's SMTP relay (SMTP without a relay is not supported) and defaults to the local host where where program is running. `UNICON USERADDRESS` should be set to a valid email address for the user sending the message, and defaults to `user@host` where `user` is the name of the user running the program and `host` is the name of the host

the program is running on. **Do not allow the default values to be used unless they are actually valid.**

The following program shows how to send a webpage to someone via email. An example use could be to have it run periodically to send current stock quotes to yourself throughout the day.

```
procedure main(args)
  if *args~ = 2 then stop("usage: ", &programe, " url mailto ")
  web := open(args [1], "m") | stop(args[1] || ": can' t open")
  if web["Status-Code"] < 300 then {
    mail := open(args [2], "m",
      "Subject : " || args [1],
      "X-Note: automatically send by Unicon") |
    stop(args[2] || ": can' t open")
    every write(mail , !web)
    close (web)
    close (mail)
  }
  else {
    write("ERROR: ", web["Status-Code"], " ", \web["Reason-Phrase"] | "")
  }
end
```

A. Function and Operator Reference

This section documents the new behavior of functions and operators which act on messaging files.

!M : string*	generate messages from POP
!M : string*	generate lines from a messaging file

The action of the generate operator depends on the type of the server the messaging file M refers to. A POP connection is treated as a list of messages, so !M will generate messages from the specified mailbox producing each message as a single string. For any other type of connection, !M generates lines of output from the server as strings. Examples:

```
M := open("http://icon.cs.unlv.edu/", "m")
every write (! M) # Writes the contents of a web page by lines
```

```
M := open("pop:// user :password@pop.myisp.net", "m")
every write (! M) # Writes messages one message at a time
```

M[string] : string	results header reference
M[number] : string	POP message reference

For protocols such as HTTP where responses consist of a header and body, M["S"] will evaluate to the value of the field named 'S'. In addition, two special fields are allowed:

- M["Status-Code"] evaluates to the integer code for the status of the request, and
- M["Reason-Phrase"] evaluates to the string containing a human-readable description of the status.

The familiar HTTP "404 Not found" result is an example of a Status-Code and Reason-Phrase. Most of the standard Internet protocols use this form of status reporting. For those protocols that don't (currently just Finger), a success or error code is faked by the protocol handling code. The Reason-Phrase however is not, and so Unicon programs should be written to handle the case when M["Reason-Phrase"] == &null.

Status codes follow a convention where a value less than 300 means success, 300-399 means an error occurred which may be correctable, and 400+ means that a fatal error occurred.

The second form (M[n]) is only valid for POP connections. The expression returns the n-th message in a POP mailbox. Examples:

```
M := open("http://icon.cs.unlv.edu/", "m")
if (M["Status-Code"] >= 300) then
  # Note: Must handle null Reason-Phrase
  stop(M["Status-Code"], " ", (M["Reason-Phrase"] | ""))
M := open("pop:// user :password@pop.myisp.net", "m")
write(M[3]) # Writes the 3rd message in the mailbox
```

close(file) : file close a messaging file

close (M) complete any pending request, closes any open connections to the server and returns resources associated with the file to the operating system. It returns the closed file.

delete(file, integer [, integer . . .]) : file delete a message

delete (M, N₁, ..., N_n) deletes all messages numbered Nx from a Post-office Protocol (POP) server and returns M. It always succeeds. As a feature of POP, messages are not irreversibly deleted until a successful close (M) is done. If the connection is lost (e.g. because the program exited) without an explicit close, no messages are actually deleted. Examples:

```
M := open("pop:// user :password@pop.myisp.net", "m")
delete (M, 1, 3, 5)
```

open(string, "m", . . .) : file? open messaging file

open(U, "m", H₁, ..., H_n) connects to the Internet server specified by the URL U and sends H₁ through H_n as headers for the request part of the translation. If the connection cannot be made, a runtime error results.

Some protocols specify default headers if they are not supplied by the program. For HTTP, the User-Agent field is automatically given the value "Unicon Messaging/10.0", and the Host field is give the host and port parts of the URL. The Host field is required by the standard and most programs should use the default value. For SMTP, the From field is automatically copied from the UNICON USERADDRESS environment variable if it is defined, or built from the username of the user running the program and the hostname of the host it is running on. Examples:

```

M := open("http://icon.cs.unlv.edu/", "m", "User-Agent: Unicon")
M := open("mailto:unicon-group@cs.unlv.edu", "m",
  "From: Steve Lumos <slumos@cs.unlv.edu>",
  "To: Unicon Group <unicon-group@cs.unlv.edu>",
  "Subject: Unicon Messaging Works!",
  "X-Unicon: Sent with Unicon!")

```

pop(file) : string? ‘pop’ message

A POP connection is seen in Unicon as a list of messages, so pop(M) will remove the first message in the POP mailbox specified by M and return it as a single string. No messages are actually deleted from the server until a successful close(M) is performed. Examples:

```

M := open("pop://user:password@pop.myisp.net", "m")
while write(pop(M))

```

read(file) : string? read a line

read(M) completes any pending request and reads a line from the server. The end of line marker is discarded. Examples:

```

M := open("finger://nevada.edu", "m")
while (s := read(M)) do {
  write(s)
}

```

reads(file, integer:1) : string? read characters

reads(M, n) completes any pending request and reads n characters from the server. If n = -1, the maximum number of characters possible are returned, which usually means the entire file. Examples:

```

M := open("http://icon.cs.unlv.edu/data", "m")
data := reads(M, -1)
M := open("http://www.files.com/bigfile.dat", "m")
while ( writes (reads(M, 4096)) # 4k buffer

```

write(x, . . .) : x write line

write (...) writes out its arguments, each followed by a newline. If any argument is a messaging file, subsequent arguments are written as the body part of a request to the server.

writes(x, . . .) : x write strings

writes (...) writes out its arguments. If any argument is a messaging file, subsequent arguments are written as the body part of a request to the server.