# MESSAGING LANGUAGE EXTENSIONS FOR UNICON

by

Steven Eric Lumos

Bachelor of Science
University of Nevada, Las Vegas
1997

A report submitted in partial fulfillment
of the requirements for the

**Master of Science Degree**
**Department of Computer Science**
**Howard R. Hughes College of Engineering**

**Graduate College**
**University of Nevada, Las Vegas**
**August 2000**

Copyright ©2000 by Steven Eric Lumos
All Rights Reserved

ABSTRACT

**Messaging Language Extensions for Unicon**

by

Steven Eric Lumos

Dr. Clinton Jeffery, Examination Committee Chair
Assistant Professor of Computer Science
University of Nevada, Las Vegas

A messaging language provides highly-integrated connectivity (networking) along with context sensitivity [29]. 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.

TABLE OF CONTENTS

ABSTRACT . . . . . iii

LIST OF FIGURES . . . . . v

LIST OF LISTINGS . . . . . vi

LIST OF TABLES . . . . . vii

ACKNOWLEDGMENTS . . . . . viii

CHAPTER 1 INTRODUCTION . . . . . 1
- REBOL . . . . . 1
- Icon and Unicon . . . . . 4

CHAPTER 2 UNICON WITH MESSAGING EXTENSIONS . . . . . 6
- The quote Program . . . . . 6
- Messaging Files . . . . . 7
- Supported Protocols . . . . . 7
- Finger . . . . . 7
- HTTP . . . . . 9
- POP . . . . . 10
- SMTP . . . . . 11
- Conclusions . . . . . 12

CHAPTER 3 THE LIBTP LIBRARY FOR TRANSFER PROTOCOLS . . . . . 13
- The Architecture . . . . . 13
- The Discipline . . . . . 13
- The Methods . . . . . 15
- Using `libtp` . . . . . 16
- Adding a New Method . . . . . 17
- Adding a New Discipline . . . . . 17
- Conclusions . . . . . 17

APPENDIX A PROGRAMS . . . . . 19
- StockTracker . . . . . 19
- tpget . . . . . 24
- wtrace . . . . . 26

APPENDIX B UNICON MESSAGING REFERENCE . . . . . 30
- Operators . . . . . 30
- Functions . . . . . 31

APPENDIX C ACRONYMS . . . . . 33

REFERENCES . . . . . 34

LIST OF FIGURES

1 An execution of the quote fetcher program . . . . . . . . . . . . . . . . . . . . . . . . 4
2 A screenshot of the StockTracker program . . . . . . . . . . . . . . . . . . . . . . . . 19

## LIST OF LISTINGS


1 A simple REBOL program . . . 1
2 Stock quote fetcher in REBOL . . . 3
3 Stock quote fetcher in Unicon . . . 6
4 General URL retrieving program . . . 9
5 A Unicon program to read email from a POP server . . . 10
6 A Unicon program to automatically email a web page . . . 11
7 The general `libtp` discipline definition . . . 14
8 The Unix discipline . . . 14
9 The general `libtp` method definition . . . 15
10 The parsed URL structure . . . 16
11 The Tprequest_t structure . . . 16
12 The Tpresponse_t structure . . . 16
13 StockTracker.icn: A longer messaging example . . . 19
14 tpget.c, using the `libtp` library . . . 24
15 wtrace.c, extending a discipline . . . 26

LIST OF TABLES


1 REBOL built-in types . . . . . . . . . . 2
2 Unicon messaging file operations . . . . . . . . . . 8
3 Short vs. Long form of HTTP access . . . . . . . . . . 9
4 The methods included with `libtp` . . . . . . . . . . 15
5 The `libtp` API . . . . . . . . . . 18

ACKNOWLEDGMENTS

Thanks to my advisor, Dr. Clinton Jeffery for giving me the opportunity to learn and hack on Unicon. The idea to implicitly detect the end of a request was suggested by John Kilburg. Thanks to Dr. Tom Nartker and the Information Science Research Institute, for providing one of the best jobs a graduate student ever had.

Special thanks to Karen, for a great many things.

# CHAPTER 1

# INTRODUCTION

Tightly integrated features for Internet programming are a useful and important addition to any programming language. The Internet is becoming ubiquitous. In the future, even home appliances such as refrigerators and coffee makers are expected to be connected to the Internet in one way or another. Even if that doesn't happen, the Internet is clearly becoming an increasingly important part of society, from news services to home shopping and even stock trading, and programs that communicate over the Internet will become more numerous than programs that do not. Programming languages should adapt to this situation by making Internet programming easier.

The creators of Unix recognized the value of the file abstraction and they designed Unix so that arbitrary devices such as tape drives and displays could be treated by programmers as if they were files. This meant that programmers only needed to learn a single interface and made Unix more portable [4]. Later, when networking support came to Unix, the file abstraction was not carried over. Instead, a new set of system calls was added for creating and managing network connections. Today's programming languages should return to the file abstraction for Internet programming. Major Internet protocols such as Hypertext Transfer Procotol (HTTP) should be encapsulated so that the file abstraction can be applied to Internet *resources* instead of just low-level network connections.

## REBOL

Messaging languages represent a powerful new paradigm in programming. Internet programming in REBOL (Relative Expression-based Object Language) is very tightly integrated. For example, a REBOL program to fetch a web page and email it to somebody is simply:

Listing 1: A simple REBOL program

```
send user@example.org read http://www.rebol.com/
```

Surely the most noticeable thing about Listing 1 is its conciseness. This is possible due to tightly integrated networking, because 'Uniform Resource Locator (URL)' and 'email address' are actually types in REBOL (along with a rich set of other domain-specific types, see Table 1), and because the behavior of **send** and **read** is dependent on the type of the object to which they are applied. Of course the program in Listing 1 does no error handling, any server or network errors become runtime errors and stop execution of the program.

REBOL also supports the concept of *dialecting*, which allows the language to be extended in order to adapt to the domain to which it is applied, in the same way that humans expand their language with jargon appropriate to the field they work in. A video studio might develop a REBOL dialect with commands like:

```
queue tape to 0:36:35
roll tape
wait until 0:37:07
wipe tape to image with effect 3
key title-text
```

Dialecting seems to be a useful language feature, but it is seperate from messaging and beyond the scope of this report.

Table 1: REBOL built-in types (from [27])

| **Type** | **Literal Representations** |
|---|---|
| Numbers | 1234 -432 3.1415 1.23E12 0,01 1,2E12 |
| Times | 12:34 20:05:32 0:25:34 |
| Dates | 20-Apr-1998 20/Apr/1998 20-4-1998 1998-4-20<br>1980-4-20/12:32 1998-3-20/8:32-8:00 |
| Money | $12.34 USD$12.34 CAD$123.45 DEM$1234,56 |
| Tuples | 3.1.5 (version number)<br>255.255.0 (RGB color)<br>131.216.22.6 (IP address) |
| Strings | "Here is a string"<br>{Here is another way to write<br>a string that spans many lines<br>and contains "quoted" strings.} |
| Tags | <title> </body> <font size="2" color="blue"> |
| Email | info@rebol.com pres-bill@oval.whitehouse.gov |
| URLs | http://www.rebol.com<br>ftp://ftp.rebol.com/sendmail.r<br>mailto:info@rebol.com |
| Files | %data.r %images/photo.jpg %../scripts/*.r |
| Issues | #707-467-8000 (phone number)<br>#000-1234-5678-9999 (credit card)<br>#MFG-932-741-A (model number) |
| Binary | #{0A6B4728C4DBEF5} (hex-encoded)<br>64#{45kLKJOIU8439LKJklj} (base64-encoded) |

Listing 2 contains a longer REBOL program. The program retrieves a stock quote from the Yahoo! Finance web site [39] and displays it in an easy-to-read format (an example execution of the program appears in Figure 1).

Listing 2: Stock quote fetcher in REBOL

```
1  REBOL [
2      Title:   "Fetch stock quotes from finance.yahoo.com"
3      Date:    3-May-2000
4      File:    %quote.r
5      Author:  ["Steve Lumos" slumos@cs.unlv.edu]
6      Version: 0.0
7  ]
8
9  ;; Format is:
10 ;;   Symbol,Last Trade,Last Update,Change,Open,Hi,Lo,Volume, e.g.
11 ;;
12 ;;   "LNUX",38.015625,"4/20/2000","3:59PM",+0.015625,40.6875,41,36,253500
13
14 fatal: func [ e [ error! ] ]
15 [
16         de: disarm e
17         if de/type = 'user [
18             print de/arg1
19             quit
20         ]
21         arg1: de/arg1
22         arg2: de/arg2
23         arg3: de/arg3
24         print get in get in system/error de/type de/id
25         quit
26 ]
27
28 if not string? system/script/args [
29     print reform [ "usage:" system/options/script "<ticker symbol>" ]
30     quit
31 ]
32
33 stock: first parse system/script/args " "
34
35 url: join http:// ["finance.yahoo.com/d/quotes.csv?s="
36                    stock "&f=sl1d1t1c1ohgv&e=.csv"]
37
38 if error? e: try [ quote: read url ] [ fatal e ]
39
40 foreach [symbol last date time change open high low vol] parse quote "," [
41     print rejoin [ "Quote for " symbol " at " time " on " date ]
42     print ""
43     print rejoin ["    Open:          " open]
44     print rejoin ["    Last Trade:    " last]
45     print rejoin ["    Change:        " change]
46     print rejoin ["    Today's Range: " low " - " high]
47     print rejoin ["    Volume:        " vol]
48     print ""
49 ]
50 quit
```

Figure 1: An execution of the quote fetcher program

```
% ./quote.r sunw
Quote for SUNW at 4:01PM on 5/3/2000

    Open:          87.5
    Last Trade:    87.375
    Change:        -0.875
    Today's Range: 83.8125 - 89.0625
    Volume:        18465400
```

Listing 2 demonstrates several features of the REBOL language. Lines 1–7 are the standard REBOL header, which is required to start any REBOL program (although the minimal '**REBOL** [ ]' is allowed). Lines 14–26 and 33–36 show how *words* are defined. Words can refer to any type, including functions and objects. In this case, the word `fatal` is being defined as a function which takes a value of type `error!`, while `stock` and `url` are being defined as a string and URL. In the case of `url`, it is necessary to keep the `http://` part *outside* the following list, or else `url` will be defined as a string, and the **read** operation will fail since it is not defined on strings. Most of the real work in the program is done in the '**read** url' part of line 38. A typical C program that solves this problem and does proper error handling might require a hundred lines for what can be expressed in two words of REBOL. On the other hand, it should be apparent that the conciseness of REBOL is quickly lost once any kind of robust error checking is needed.

The syntax of REBOL is designed to be easy for non-programmers. Although it is syntactically similar to Forth, it is often compared to REXX, another language intended for non-programmers. Whether or not REBOL is easier for non-programmers is beyond the scope of this report, but more pertinent is how programmers respond to it. Many programmers are unwilling to learn a language that is syntactically much different from languages they already know unless there is a large benefit in doing so. A discussion on the Internet community site Slashdot [33] prompted many responses such as:

- "...why should people use a 'new language' just to do simple little tasks like downloading a webpage or sending mail?",
- "I agree that the syntax in REBOL (given the tiny example that I looked at) is odd enough that I probably wouldn't want to spend the time learning it.", and
- "The very fact that it's different also means that it will never be very widely adopted."

If programmers accept the value of messaging languages but don't want to learn a new syntax then it becomes desirable to combine messaging language features with a traditional programming language. That is the subject of my work.

## Icon and Unicon

Icon [11] is a very high level, general purpose programming language with a focus on text processing. Although the syntax of Icon is similar to Algol-like languages such as C or Pascal, Icon has a very original semantic feature known as *goal-directed evaluation.* In most programming languages, certain expressions return boolean values which may be mapped onto integers and may or may not be available to the programmer as a type. In Icon, all expressions either *succeed* and produce a value, or they *fail.* Icon implements goal-directed evaluation by combining this idea with the concept of *generators*, expressions which can return a value but then be resumed to return additional values. When part of an expression fails in Icon, control can backtrack to a previous part of the expression to see if it can generate a new value. This continues until the entire expression succeeds or there are no more values. For example, in '**if** L[i := (1 **to** 10)] = 5 **then** **write**(i)', the expression '(1 **to** 10)' is a generator which takes on values between the left and right operands

(returning the next value in the sequence each time it is resumed). In this way, values between 1 and 10 will be assigned to the value i and used as an index into list L (since assignment evaluates to the value of the right operand) until an i is found which satisfies the *goal*, L[i] = 5.

Another nice feature of Icon is that communicating control flow in terms of success and failure instead of boolean values allows expressions to evaluate to useful values other than true or false. For example, the Icon relational operator < returns the value of its right operand, so constructions such as '**if** 2 < x < 10 **then** ...' actually make sense.

Unicon (the unified, extended dialect of Icon or University of Nevada Icon) is a descendent of Icon [15,16]. Unicon exists because the creators recognized the value of having Icon's expressiveness and goal-directed evaluation for a larger class of problems than those to which Icon is typically applied. In order to make Icon generally applicable to today's common programming problems, Unicon adds several new features to Icon, including:

**OOP** Object-oriented programming through classes with data and procedure members.

**POSIX** Much of the Portable Operating System Interface (POSIX) [38] is available as built-in functions in Unicon, including functions for (low level) TCP and UDP network communication.

**ODBC** New functions for accessing database systems though Microsoft's Open Database Connectivity (ODBC) [37] interface standard has been added.

The feature set of Unicon makes it an attractive target for messaging language extensions. The language is high level and easy to program while retaining a familiar syntax, and the text-processing features inherited from Icon are highly applicable to the processing of information gathered from Internet resources such as web pages.

CHAPTER 2

# UNICON WITH MESSAGING EXTENSIONS

The goal of the Unicon Messaging Extensions is to provide tightly integrated network services in a messaging language-like way while at the same time adding as little as possible to the *size* of the language (in particular, no new built-in functions) and leveraging programmer intuition wherever possible.

## The quote Program

As a quick introduction, Listing 3 shows a Unicon version of the stock quote program. The output of this program is identical to the output from the REBOL version in Listing 2. Listing 3 illustrates several features of the Unicon Messaging Extensions. Lines 13–15 show how a messaging file is opened, **open**() is called with the URL and a mode of "m", optionally followed by some header field definitions whose behavior is dependent on the URL. Notice how header fields are specified in a very natural way. The condition in line 17 shows how to access fields in the server response (the Icon syntax for table access has been extended for use on messaging files). Status codes are defined by the protocol being used and reasonable values are filled in if the protocol does not provide them. Finally, lines 18 and 19 show the standard file access functions **reads**() and **close**() being called on the messaging file.

Listing 3: Stock quote fetcher in Unicon

```
1  procedure usage()
2      return "usage: " || &progname || " <ticker symbol>"
3  end
4
5  procedure main(args)
6      if *args ~= 1 then stop(usage())
7
8      symbol := args[1]
9
10     yahoourl := "http://finance.yahoo.com/d/quotes.csv?s=" || symbol ||
11                 "&f=sl1d1t1c1ohgv&e=.csv"
12
13     yahoo := open(yahoourl, "m",
14                   "User-Agent: Unicon Quote Fetcher/0.0",
15                   "X-Unicon: http://icon.cs.unlv.edu/")
16
17     if yahoo["Status-Code"] < 300 then {
18         csv := reads(yahoo, -1)
19         close(yahoo)
20
21         quote := list (0, "")
22         csv ? {
23             while put(quote, tab(upto(','))) do move(1)
24             put(quote, tab(0))
25         }
26     }
27     else {
```

```
28          stop(yahoo["Status-Code"], " ", yahoo["Reason-Phrase"])
29          close(yahoo)
30      }
31
32      write("Quote for ", quote[1][2:-1], " at ", quote[4][2:-1],
33            " on ", quote[3][2:-1])
34      write()
35      write("    Open:         ", quote[6])
36      write("    Last Trade:   ", quote[2])
37      write("    Change:       ", quote[5])
38      write("    Today's Range: ", quote[8], " - ", quote[7])
39      write("    Volume:       ", quote[9])
40  end
```

Messaging Files

Unicon messaging uses the existing file type and extends built-in functions and operators to recognize and handle messaging files. Table 2 lists the basic operations in their most common form. A more complete reference appears in Appendix B.

Which operations are legal depends on the service being used. Some services are read-only or write-only such as POP and Simple Mail Transfer Protocol (SMTP), while others like HTTP are read-write. In the case of a read-write service, a special protocol is required for a complete transaction. Imagine the case where a program wants to send a large (i.e. larger than main memory) file. The standard way to handle this situation is to read part of the file into a memory buffer, write the contents of the buffer, read the next part of the file, write the next part of the file, etc. Since multiple writes are required, we must provide some way to signify that all of the file has been written, so the system can notify the remote server. One way to handle this would be to add a new built-in procedure (perhaps called "endwrite") that could be called after all of the file had been written. However, this would increase the size of the language by adding a new built-in function, and it is also counter to programmer-intuition since writing ordinary files does not require calling the endwrite procedure. Instead, the Unicon Messaging Extensions specify the end of write implicitly. As soon as any operation is done which would require a result from the server (for example F["Status-Code"]), the request is considered to be ended, the server is notified, and the response is read.

Supported Protocols

The current implementation supports four protocols, (Finger, HTTP, POP, and SMTP), although attention has been paid toward making the addition of new protocols easy (see Chapter 3). Where possible, the system has been designed so that different Internet services can be accessed in a protocol-independent fashion, but for maximum flexibility and robust error handling, there are also protocol specific features which are made available to Unicon programs. These are discussed for each protocol in the following sections.

Finger

The Finger protocol [40] is a simple protocol for obtaining information about users of an Internet host. This is the simplest protocol supported by the Unicon Messaging Extensions. The `grab.icn` example program in Listing 4 works just as well for Finger requests without modifications. The format used for a finger URL is: `finger://`*host*[`:`*port*][`/`[`/w` ]*username*], where /w is an optional component which specifies the "long" version of the response (what "long" means is implementation dependent and varies among servers). Examples:

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

Table 2: Unicon messaging file operations

| Syntax | Purpose |
|---|---|
| M := **open**(U, `"m"`[, header1,...]) | "Open a messaging file". Opens a connection to the server and starts the request. |
| **close**(M) | Closes the connection and frees operating system resources. *All programs should explicitly call* ***close*** *on messaging files.* |
| **delete**(M, N1[, N2 , ...]) | Deletes messages from a Post Office Protocol (POP) mailbox. |
| **pop**(M) | Returns the top message from a POP mailbox and deletes it. |
| S := **read**(M) | Reads and returns one line of the response body. |
| S := **reads**(M, N) | Reads and returns *N* characters of the response body. |
| **write**(M, S1[, S2 ,...]) | Sends the body part of a request (e.g. HTTP POST or email). |
| **writes**(M, S1[, S2 ,...]) | Sends the body part of a request, but without automatically appending a newline character. |
| **delete**(M, N1[, N2,...]) | For POP connections, deletes messages numbered N1, N2, . . . . |
| M[S] | Returns the value of the field named *S* from the response header. |
| M[N] | For messaging file opened on a POP URL, returns the Nth message. |
| !M | Generates lines from a response or messages for POP connections. |

Values have the following meanings: **U** – a URL, **M** – a messaging file, **S** – a string, **N** – an integer.

## HTTP

The Hypertext Transfer Procotol (HTTP) [1,6] is the protocol of the World Wide Web (WWW), and is expected to be the most used protocol supported by the Unicon Messaging Extensions. HTTP access may be read-only or a write followed by a read of the response. Additionally, a read may be considered to be "long" form (the default), or "short" form (selected by the `"ms"` flag instead of just `"m"`), which translates to an HTTP HEAD request which returns only the header from the response instead of the header and body (see Table 3).

Table 3: Short vs. Long form of HTTP access

| **Command** | **Headers** | **read**(F) |
|---|---|---|
| F := **open**("http://localhost/abc", "m") | F["Field-Name"] | *Returns one line of body* |
| F := **open**("http://localhost/abc", "ms") | F["Field-Name"] | *FAILS* |

A notable feature of HTTP is that it allows for the server to notify a client when a resource has been moved to a new location. When this is done, the server will send a response code of 301, 302, 303, or 307, and put the URL of the new location in a header field named "Location". The example program in Listing 4 handles this case but is also general enough to be used for other protocols, such as Finger.

Listing 4: General URL retrieving program

```
1  procedure main(args)
2    if *args < 1 then stop("usage: ", &progname, " url")
3
4    # Connect to the host specified in the URL, sending some custom
5    # header fields.
6    f := open(args[1], "m",
7             "User-Agent: Unicon Grab 0.0",
8             "X-Unicon: http://icon.cs.unlv.edu/") |
9       stop(args[1], ": can't open")
10
11   repeat {
12      if f["Status-Code"] < 300 then {
13         # If the server returns a successful status code, read in the
14         # result 64k at a time and write it out.
15         while writes(reads(f, 65535))
16         exit(0)
17      }
18      else if f["Status-Code"] < 400 & \f["Location"] then {
19         # If the server returns a 3xx error, check for a Location:
20         # header and follow if found.
21         newloc := f["Location"]
22         close(f)
23         f := open(newloc, "m",
24                   "User-Agent: Unicon Grab 0.0",
25                   "X-Unicon: http://icon.cs.unlv.edu/") |
26            stop(newloc, ": can't open")
27      }
28      else {
29         # Some other error, so tell the user what the server told us.
30         stop(f["Status-Code"], " ", \f["Reason-Phrase"] | "")
31      }
```

```
32   }
33 end
```

Both the POST and PUT methods are supported for HTTP writes, depending on how the program sets the Content-Type header field. If the Content-Type contains the string “form”, (e.g. “multipart/form-data”) then the POST method is used, otherwise the PUT method is used. Whether to use POST or PUT is dictated by the server. The POST request is usually used for the processing of fill-in forms expressed in Hypertext Markup Language (HTML), although sometimes used for file upload as well, while the PUT method is meant specifically for file uploading. In either case, the programmer is responsible for doing any necessary encoding before calling **write**(), due to the large number of possibilities.

## POP

The purpose of the Post Office Protocol (POP) is to allow the reading of email from leaf-node class hosts (i.e. office PCs), although POP servers are also used by organizations with a large number of Internet hosts for consolidating email by user instead of user and host. The format of a POP URL is: `pop://`*user*`:`*pass*`@`*host*[`:`*port*]. Note that only the service port is optional.

Instead of being viewed as a file, a POP connection is viewed as a list where each complete message is a list member. The standard list operations generation (!), subscripting ([ ]), and **pop**() work as expected. However, **read**() does *not* work because there is no definition that both makes sense and is consistent with the rest of Unicon.

It is especially important to properly close a POP connection when deleting messages. This is because the definition of POP [19] specifies that no deletions are actually done unless and until the client correctly ends the session.

Listing 5 is an example of using POP. The program prints each message stored in the given POP mailbox and deletes them.

Listing 5: A Unicon program to read email from a POP server

```
1  # DANGER: This program deletes messages from your server! The copy of
2  # the message that is displayed on your screen is the only copy you
3  # will EVER SEE. Do not run this program unless you understand the
4  # implications.
5  procedure main(args)
6     if *args ~= 1 then {
7        stop("usage: ", &progname, " <pop url>\n",
8             "   <pop url> is of the form pop://user:password@popserver:port")
9     }
10
11    url := args[1]
12
13    if url[1:5] ~== "pop:"then
14       stop(url, " not a POP url")
15
16    mailbox := open(url, "m") | stop("can't open ", url)
17    write(repl("=", 75))
18    while write(pop(mailbox)) do {
19       write(repl("=", 75))
20       CheckStatus(mailbox)
21    }
22    write(repl("=", 75))
23    close(mailbox)
24 end
25
26 procedure CheckStatus(m)
27    if m["Status-Code"] >= 300 then {
```

```
28          close(m)
29          stop("POP error: ", m["Status-Code"],
30               " ", \m["Reason-Phrase"] | "")
31      }
32      else {
33          writes("POP success: ", m["Status-Code"],
34                " ", \m["Reason-Phrase"] | "")
35      }
36  end
```

## SMTP

Almost all Internet email is transported using the Simple Mail Transfer Protocol (SMTP). SMTP support allows Unicon programs to send email to any Internet address. The destination email address is given in the form `mailto:`*user*`@`*domain*. Extensions to the mailto: URL to specify header fields and message body are not supported, as this is better done using the mechanism already in place.

Sending email requires two pieces of information that would be inconvenient to specify in the program, so they are instead specified by setting environment variables. `UNICON_SMTPSERVER` should be set to the hostname of the mail relay for the user running the program and `UNICON_USERADDRESS` should be set to the email address of the user, which is also used as a default for the 'From' header field. On Unix, if either of these values is not specified, then defaults are constructed from the username of the user running the program and the name of the host where the program is running. If either of these values cannot be determined, a runtime error results.

Listing 6 shows a Unicon program that takes a web page URL and a mailto URL and sends the web page as email.

Listing 6: A Unicon program to automatically email a web page

```
1   procedure main(args)
2       if *args ~= 2 then stop("usage: ", &progname, " url address")
3
4       web := open(args[1], "m",
5                   "User-Agent: Unicon Grab 0.0",
6                   "X-Unicon: http://icon.cs.unlv.edu/") |
7           stop(args [1] || ": can't open")
8
9       if web["Status-Code"] < 300 then {
10          mail := open(args[2], "m",
11                       "Subject: " || args [1],
12                       "X-Note: automatically sent by Unicon") |
13              stop(args [2], ": can't open")
14
15          every write(mail, !web)
16          close(web)
17          close(mail)
18      }
19      else {
20          write("ERROR: ", web["Status-Code"], " ", \web["Reason-Phrase"] | "")
21      }
22  end
```

The apparent complexity of Listing 6 relative to Listing 1 is due to error checking. When error checking and optional arguments are removed, Listing 6 can be written as

```
procedure main(args)
    every write(m:=open(args[2], "m"), !(open(args[1], "m")))
    close(m)
end
```

The main difference between this program and the REBOL version is that certain things must be explicit: declaring the main procedure and arguments, opening the Internet connections, and closing one of the connections. The latter is due to the need to signal that all of the data has been written as explained in the Messaging Files section. The other two are minor for any user of traditional programming languages. Note also that this program accepts an arbitrary website and email address where Listing 1 does not. Since Unicon (like most programming languages) implicitly parses command line arguments into a list, the parameterized version of the program in Unicon is actually shorter than if it was not. In REBOL the program arguments are given as a single string and must be parsed into words explicitly as shown in Listing 2.

## Conclusions

The Unicon Messaging Extensions bring Internet programming support to the Unicon language. Internet programming is easy in Unicon thanks to the extensions' use of already existing abstractions to leverage programmer intuition.

The Unicon Messaging Extensions are available from `<http://www.cs.unlv.edu/~slumos/municon.html>` as *μnicon*, a complete Unicon distribution with modifications for messaging. It is expected that the messaging extensions will be kept up-to-date with or become part of the main Unicon distribution in the future.

# CHAPTER 3

# THE LIBTP LIBRARY FOR TRANSFER PROTOCOLS

All of the networking operations in the Unicon Messaging Extensions are handled by the `libtp` library. The purpose of the `libtp` library is to abstract many different transfer protocols (e.g. HTTP, SMTP) into a clear and consistent API. Rather than try to support every aspect of a particular protocol, the library supports only those features necessary to accomplish most tasks. A major design goal is to make the library both easy to extend with new protocols and easy to port to new operating system interfaces, which it accomplishes by using the discipline and method architecture used by AT&T Labs and described in [35]. It is not a goal of the library to be usable without knowledge of the actual protocols, and anyone wishing to use the library should also consult the RFC documents relevant to the protocols being used.

## The Architecture

The key feature of the discipline and method architecture is that it makes explicit two interfaces in the library: *disciplines* which hold system resources and define routines to acquire and manipulate them, and *methods* which define the higher-level algorithms used to access those resources. For example, the Vmalloc memory allocation library by Kiem-Phong Vo [34] is divided into disciplines which abstract system interfaces to acquire memory, and methods which encapsulate algorithms for allocating that memory to the application. This separation not only makes the library more portable, but allows the algorithms to be applied to different kinds of memory (e.g. shared memory) without major rewriting.

The `libtp` library needs to be able to open a network connection and perform read and write operations on it, so the discipline contains these functions while the method defines standard routines for communicating with a server using a specific protocol. Each protocol interface is defined only in terms of the discipline, so making the code work on different operating systems or with different networking subsystems is just a matter of implementing the proper routines. Since applications can also define methods, the library can be extended with new protocol interfaces easily. Both methods and disciplines can be added and selected at runtime, and without requiring access to the library source code.

## The Discipline

The general `libtp` discipline appears in Listing 7. In the implementation, the discipline is a C structure whose members are pointers to functions. These functions define a complete API for acquiring and manipulating all of the system resources needed by all of the methods and (it is hoped) any conceivable method. By convention, every discipline function takes a pointer to the current discipline as its last argument and every method takes a pointer to a library handle which contains a pointer to the current discipline as its first argument, so the discipline functions are always available when needed.

It is also possible to extend an existing discipline. In general, defining a new discipline always involves extending the default discipline to hold some system dependent data (usually a connection handle such as a file descriptor on Unix or a `SOCKET*` on Windows) as well as defining all of the discipline functions. Listing 8 shows how the Unix discipline extends the general discipline to include a Unix file descriptor. New disciplines should always include a Tpdisc_t member first and system specific data last so that a pointer to any discipline can be typecast to Tpdisc_t and passed through the many discipline independent functions.

Listing 7: The general `libtp` discipline definition

```
typedef struct _tpdisc_s    Tpdisc_t;       /* discipline */

typedef int       (*Tpconnect_f)(char* host, u_short port, Tpdisc_t* disc);
typedef int       (*Tpclose_f)(Tpdisc_t* disc);
typedef ssize_t   (*Tpread_f)(void* buf, size_t n, Tpdisc_t* disc);
typedef ssize_t   (*Tpreadln_f)(void* buf, size_t n, Tpdisc_t* disc);
typedef ssize_t   (*Tpwrite_f)(void* buf, size_t n, Tpdisc_t* disc);
typedef void*     (*Tpmem_f)(size_t n, Tpdisc_t* disc);
typedef int       (*Tpfree_f)(void* obj, Tpdisc_t* disc);
typedef int       (*Tpexcept_f)(int type, void* obj, Tpdisc_t* disc);
typedef Tpdisc_t* (*Tpnewdisc_f)(Tpdisc_t* disc);

struct _tpdisc_s
{
  Tpconnect_f connectf;  /* establish a connection */
  Tpclose_f   closef;    /* close the connection */
  Tpread_f    readf;     /* read from the connection */
  Tpreadln_f  readlnf;   /* read a line from the connection */
  Tpwrite_f   writef;    /* write to the connection */
  Tpmem_f     memf;      /* allocate some memory */
  Tpfree_f    freef;     /* free memory */
  Tpexcept_f  exceptf;   /* handle exception */
  Tpnewdisc_f newdiscf;  /* deep copy a discipline */
  int type;              /* TP_FILE, TP_SOCKET (not used currently) */
};
```

Listing 8: The Unix discipline

```
struct _tpunixdisc_s
{
  Tpdisc_t tpdisc;
  int fd;         /* File descriptor for file or socket */
};
```

In addition to functions for managing network connections, the discipline also includes functions for memory allocation and exception handling. The discipline and method architecture defines a very useful convention for exception handling. Exceptions are passed as integers to the exceptf function along with some exception specific data. The function can do arbitrary processing and then return -1, 0, or 1, which instructs the library to try the operation again (1), return an error to the caller (-1), or execute some default action (0). As an example of what can be done, the Vmalloc library handles an out-of-memory exception by performing a garbage collection and returning 1 to make the caller retry the request. In the Unix discipline, the convention is useful for handling certain low-level errors such as `EINTR` (interrupted system call). Instead of explicitly checking for `EINTR` in every place it may occur, every error causes an exception (e.g. `TP_EREAD`), and the exception handler returns 1 if the system error is one of those that is considered to be transient.

Besides creating a new discipline, it is sometimes useful to redefine only some of the functions of a provided discipline. The *wtrace* program in Listing 15 of Appendix A does this by defining replacement functions which are actually wrappers around the original functions, then saving the original functions and installing the new ones with the lines:

```
disc = tp_newdisc(TpdUnix);
tpexcept = disc->exceptf;
disc->exceptf = exception;
tpread = disc->readf;
disc->readf = readf;
tpreadln = disc->readlnf;
```

```
disc->readlnf = readlnf;
tpwrite = disc->writef;
disc->writef = writef;
```

## The Methods

Each Internet protocol is encapsulated in a method, which is defined in terms of the general discipline. This allows the algorithms for communicating with servers to be applied on different operating systems and even different types of transmission channels simply by defining a discipline for them. Listing 9 contains the general method definition and Table 4 lists the included methods.

Listing 9: The general `libtp` method definition

```
typedef struct _tpmethod_s Tpmethod_t;  /* method */

typedef int             (*Tpmbegin_f)(Tp_t* tp, Tprequest_t* req);
typedef Tpresponse_t* (*Tpmend_f)(Tp_t* tp);
typedef int             (*Tpmclose_f)(Tp_t* tp);
typedef int             (*Tpmfreeresp_f)(Tp_t* tp, Tpresponse_t* resp);

struct _tpmethod_s
{
  Tpmbegin_f    beginf;     /* begin a request */
  Tpmend_f      endf;       /* end a request */
  Tpmclose_f    closef;     /* cleanup and close connection */
  Tpmfreeresp_f freerespf;  /* free the response record */
  Tpstate_t     state;
};

typedef int Tpstate_t;
enum _tpstate_e {
  CLOSED   = 0,
  CONNECTING = 1,
  CONNECTED = 2,
  WRITING = 4,
  READING = 8,
  CATCH    = 16,  /* not used currently */
  BAD      = 32
};
```

Table 4: The methods included with `libtp`

| Method | Protocol |
| --- | --- |
| TpmDaytime | Daytime [25] |
| TpmFinger | Finger [40] |
| TpmHTTP | Hypertext Transfer Procotol (HTTP) [1,6] |
| TpmPOP | Post Office Protocol (POP) [19,21] |
| TpmSMTP | Simple Mail Transfer Protocol (SMTP) [24] |

## Using `libtp`

To use `libtp` a program first needs a URL which is expressed in the form of the structure shown in Listing 10. The program can either fill in this structure manually or use the uri_parse() function which takes a character string containing a URL and parses it into the structure. Next, the program calls tp_new() with the parsed URL, the method (usually specified by the URL), and the discipline to use, getting a handle of type Tp_t in return. This handle is passed to all of the API functions and eventually released with tp_free(), which also closes the connection.

Listing 10: The parsed URL structure

```
typedef struct _uri {
    int    status;    /* Success or error code */
    char  *scheme;    /* Access scheme (http, mailto, etc) */
    char  *user;      /* Username for authentication */
    char  *pass;      /* Password for authentication */
    char  *host;      /* Server hostname */
    int    port;      /* Service port number */
    char  *path;      /* Pathname (file, email address, etc) */
} URI, *PURI;
```

Communication with the remote server follows a transaction model whereby the program tells the library to send a request using the structure in Listing 11 and receives a response in the structure shown in Listing 12.

Listing 11: The Tprequest_t structure

```
typedef int Tpreqtype_t;
enum _tpreqtype_e {
    NOOP=1,
    GET, HEAD, POST, PUT,              /* HTTP */
    DELE, LIST, QUIT, RETR, STAT,      /* POP3 */
    DATA, HELO, MAIL, RCPT             /* SMTP */
};

typedef struct _tprequest_s Tprequest_t;

struct _tprequest_s
{
    Tpreqtype_t type;      /* kind of request */
    char*       header;    /* ready-to-send header */
    char*       args;      /* request argument string */
};
```

Listing 12: The Tpresponse_t structure

```
typedef struct _tpresponse_s Tpresponse_t;

struct _tpresponse_s
{
    int    sc;        /* error/success code */
    char*  msg;       /* message/string part of response */
    char*  header;    /* header part of response */
};
```

Three methods for making requests are provided, depending on the functionality required. The most general is to call tp_begin() which will start the request, including sending all headers. The

program may then call tp_write() as many times as necessary to send the body of the request, and then tp_end() to finish the request and retrieve the status code, reason phrase (if any), and headers (if any) from the server. Depending on the protocol and request type, the program might call tp_read() in order to read the body part of the response. The tp_sendreq() function is easier to use when the request does not have a body part. It is equivalent to calling tp_begin() immediately followed by tp_end(). The program may still call tp_read() to receive the body of the response. The last request function, tp_quickreq(), is for very simple protocols where requests and responses do not even have headers. It reads the request from a buffer, returns the response in another buffer, and handles management of the structures automatically. See Table 5 for the definitions of these functions.

If the protocol is of the type where the connection remains open after a transaction, more requests may be made. When done using a connection, the program should call tp_free () to release the handle. The complete `libtp` API is given in Table 5.

## Adding a New Method

To add a new method to the library, start with the `tpm.template` file in the `libtp` distribution. Replace XX in identifiers with your protocol name (e.g. XXbegin() to httpbegin()), and fill in the function bodies. Functions which use the Tprequest_t structure usually include a **switch** statement which takes different actions based on the value of req−>type, so appropriate request types should be added to the Tpreqtype_t enum in `tp.h`. Use one of the existing methods as an example. When the method has been written, add **extern** Tpmethod_t∗ TpmXX; to the end of `tp.h` and `tpmXX.o/tpmXX.c` to the OBJS and SRCS variables in `Makefile.in`. Then run `make` to rebuild the library with the new method.

If the method is not going to be added to the library, follow the instructions above but don't modify `tp.h` or `Makefile.in`. As long as the method follows the correct form, you can use it simply by compiling it into your program and passing it to the tp_new() function when opening a connection.

## Adding a New Discipline

Adding a new discipline is simply a matter of extending the general discipline structure to include any system-dependent data and defining all of the functions. As with adding a method, the discipline can be added to `Makefile.in` and `tp.h` to make it a part of the library. You are strongly encouraged to read through the Unix discipline in `tpdunix.c` and `tpdunix.h` paying special attention to the definition of the unixexcept() function. Proper use of exception handling will make the job of discipline writing much easier.

## Conclusions

The `libtp` library provides access to Internet resources through a single standard API. Practice has shown that it is easy to extend the library with new protocol methods. Although only a Unix discipline has been implemented at this time, the discipline and method architecture has been strictly adhered to and the library is therefore believed to be highly portable.

The library is available as part of the *μnicon* distribution at `<http://www.cs.unlv.edu/~slumos/municon.html>`.

Table 5: The `libtp` API

| Function Definition | Notes |
|---|---|
| URI* uri_parse(**char** *uri); | Parses the URL in the string uri into the structure given in Listing 10 and returns a pointer to it. The status member of the URI structure will contain `URI_OK` on success, or an error number which can be used as an index into the `_uri_errlist` array to obtain a string representation of the error. |
| **void** uri_free (URI* puri); | Releases the URI structure returned by uri_parse (). |
| Tpdisc_t* tp_newdisc(Tpdisc_t* disc); | Allocates a new discipline structure and fills it with values from disc. |
| Tp_t* tp_new(URI* puri, Tpmethod_t* meth, Tpdisc_t* disc); | Opens a connection to puri using method meth and discipline disc |
| **void** tp_free(Tp_t* tp); | Closes connection (if necessary) and releases the handle. |
| **int** tp_begin(Tp_t* tp, Tprequest_t* req); | Begins a request. Use tp_write () to send the body of the request and tp_end() to finish the request. Used by HTTP POST and PUT methods, and SMTP. |
| Tpresponse_t* tp_end(Tp_t* tp); | Finishes the request and returns the server response code and headers. Use tp_read() or tp_readln () to retrieve the response body if necessary. |
| Tpresponse_t* tp_sendreq(Tp_t* tp, Tprequest_t* req); | For services that have short requests, but potentially long responses. Use tp_read() or tp_readln () to retrieve the response body if necessary. Used for HTTP GET and HEAD requests, Finger, etc. |
| **int** tp_quickreq(Tp_t* tp, **char*** req, **char*** resp, size_t n); | For very simple protocols with short (or no) requests and responses. Internal structures are managed automatically, and no other functions should be called. Used for Daytime requests. |
| ssize_t tp_read(Tp_t* tp, **void*** buf, size_t n); | Reads n bytes from the server into buf. |
| ssize_t tp_readln(Tp_t* tp, **void*** buf, size_t n); | Reads one line (up to maximum of n bytes) from the server and adds a null termination. The line is returned in buf. |
| ssize_t tp_write(Tp_t* tp, **void*** buf, size_t n); | Sends n bytes from buf to the server. |
| **char*** tp_headerfield(**char*** header, **char*** field); | Returns a pointer to the value part of a header field. |

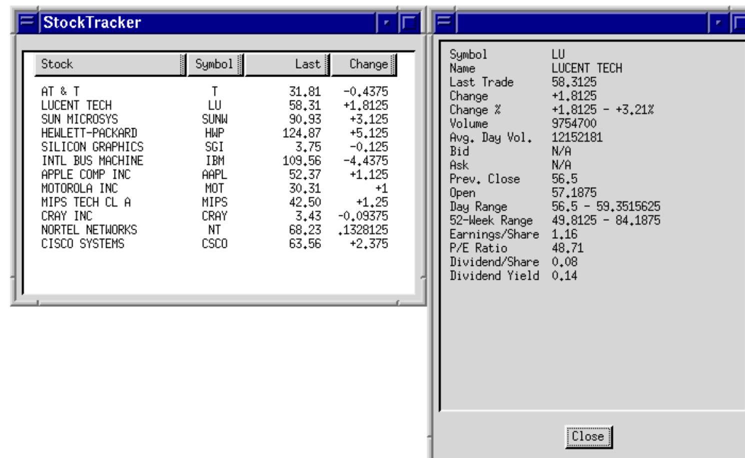
# APPENDIX A

# PROGRAMS

## StockTracker

StockTracker is a program that helps with tracking stocks thoughout the day. For stocks listed on the command line, it uses the object oriented Unicon GUI to display last trade and change data in a table format. The display is updated periodically from the Yahoo! Finanace web site. Clicking on a stock will open a second window displaying detailed information on the stock.

Figure 2: A screenshot of the StockTracker program




Listing 13: StockTracker.icn: A longer messaging example

```
1  link printf
2
3  global lastread, stocks
4
5  class dialog : _Dialog(Stocks, tcStock, tcSymbol, tcLast, tcChange)
6     method handle_Stocks(ev)
7        local selected , detailList
8        static dDetails
9        case ev.event of {
10          &lpress : {
11             selected := Stocks$get_contents()[Stocks$get_which_down()][2]
12             /dDetails := Details()
13             /dDetails.is_open & dDetails.show_modeless()
14             dDetails. tlDetails . set_contents(GetDetails(selected))
15             }
16          &lrelease : {
17             Stocks$clear_selections ()
```

```
18             }
19         default: write("Stocks: ", ev.event)
20         }
21     end
22
23     method handle_tcStock(ev)
24         Stocks$set_contents(sortf(Stocks$get_contents (), 1))
25     end
26
27     method handle_tcSymbol(ev)
28         Stocks$set_contents(sortf(Stocks$get_contents (), 2))
29     end
30
31     method handle_tcLast(ev)
32         local L, row
33         L := Stocks$get_contents()
34         every row := !L do row[3] := real(row[3])
35         L := sortf(L, 3)
36         every row := !L do row[3] := sprintf("%.2r", row[3])
37         Stocks$set_contents(L)
38     end
39
40     method handle_tcChange(ev)
41         local L, row
42         L := Stocks$get_contents()
43         every row := !L do row[4] := real(row[4])
44         L := sortf(L, 4)
45         every row := !L do row[4] := fmtChange(row[4])
46         Stocks$set_contents(L)
47     end
48
49     method handle_default(ev)
50         if ev.event == "q"then exit()
51     end
52
53     method tick()
54         local yahoo, yahoourl, row, tContents
55
56         lastread := GetSummary(stocks)
57         tContents := ParseCSV(lastread)
58         every row := !tContents do {
59             row[3] := sprintf ("%.2r", row[3])
60             row[4] := fmtChange(row[4])
61         }
62         Stocks.set_contents (tContents)
63     end
64
65     method dialog_event(ev)
66         case ev$get_component() of {
67             Stocks : handle_Stocks(ev)
68             tcStock : handle_tcStock(ev)
69             tcSymbol : handle_tcSymbol(ev)
70             tcLast : handle_tcLast(ev)
71             tcChange : handle_tcChange(ev)
72             default : handle_default(ev)
73         }
74     end
75
76     method init_dialog()
```

```
77    end
78
79    method end_dialog()
80    end
81
82    method setup()
83       local fwidth
84        self$set_attribs ("size=380,250", "bg=light grey", "label=StockTracker")
85        self$set_ticker (60000)
86       Stocks := Table()
87       Stocks$set_pos("5", "15")
88        Stocks$set_size ("370", "230")
89        Stocks$set_attribs ("bg=white")
90        Stocks$set_tooltip ("Select stock for more stats")
91       Stocks$set_contents ([])
92        Stocks$set_select_one ()
93       fwidth := 9
94       tcStock := StaticTableColumn()
95        tcStock$set_label ("Stock")
96        tcStock$set_internal_alignment ("l")
97       tcStock$set_column_width(16*fwidth)
98        tcStock$set_attribs ("bg=light grey")
99       Stocks$add(tcStock)
100      tcSymbol := StaticTableColumn()
101      tcSymbol$set_label("Symbol")
102      tcSymbol$set_internal_alignment("c")
103      tcSymbol$set_column_width(6*fwidth)
104      tcSymbol$set_attribs("bg=light grey")
105      Stocks$add(tcSymbol)
106      tcLast := StaticTableColumn()
107       tcLast$set_label ("Last")
108       tcLast$set_internal_alignment ("r")
109      tcLast$set_column_width(9*fwidth)
110       tcLast$set_attribs ("bg=light grey")
111      Stocks$add(tcLast)
112      tcChange := StaticTableColumn()
113      tcChange$set_label("Change")
114      tcChange$set_internal_alignment("r")
115      tcChange$set_column_width(7*fwidth)
116      tcChange$set_attribs("bg=light grey")
117      Stocks$add(tcChange)
118      self$add(Stocks)
119   end
120
121   method component_setup()
122       self$setup ()
123   end
124
125    initially
126       self$_Dialog . initially ()
127 end
128
129 class Details : _Dialog(bClose, tlDetails )
130   method dialog_event(ev)
131      case ev$get_component() of {
132         bClose : {
133              self .dispose()
134              }
135          }
```

```
136    end
137
138    method init_dialog()
139    end
140
141    method end_dialog()
142    end
143
144    method setup()
145        self$set_attribs ("size=300,400", "bg=light gray")
146        bClose := TextButton()
147        bClose$set_pos(146, 390)
148        bClose$set_align("c", "b")
149        self$set_focus (bClose)
150        bClose$set_label("Close")
151        bClose$set_internal_alignment("c")
152        self$add(bClose)
153        tlDetails := TextList()
154        tlDetails$set_pos ("5", "5")
155        tlDetails$set_size ("290", "350")
156        tlDetails$set_contents ([""])
157        self$add( tlDetails )
158    end
159
160    method component_setup()
161        self$setup ()
162    end
163
164    initially
165        self$_Dialog . initially ()
166 end
167
168 class StaticTableColumn : TableColumn()
169    method handle_event(e)
170        return self$TextButton.handle_event(e)
171    end
172 end
173
174 procedure main(args)
175    local d
176
177    if *args = 0 then
178        stop("usage: ", &progname, " <stocks to display>")
179
180    stocks := args
181
182    d := dialog()
183    d$show_modal()
184 end
185
186 procedure GetSummary(stocks)
187    local s, yahoo, yahoourl
188
189    yahoourl := "http://finance.yahoo.com/d/quotes.csv?e=.csv&f=nsl1c1"
190    every yahoourl ||:= ("&s=" || !stocks)
191
192    yahoo := open(yahoourl, "m",
193                  "User-Agent: Unicon",
194                  "X-Unicon: http://icon.cs.unlv.edu/")
```

```
195
196     if yahoo["Status-Code"] >= 300 then {
197        write(&errout, "Yahoo error: ", yahoo["Status-Code"], " ",
198              \yahoo["Reason-Phrase"] | "")
199        fail
200        }
201     s := reads(yahoo, −1)
202     close(yahoo)
203     return s
204 end
205
206 procedure GetDetails(stock)
207     local s, yahoo, yahoourl, dlist , i
208     static fields , fw
209
210     initial {
211         fields := ["Symbol", "Name", "Last Trade", "Change", "Change %",
212                    "Volume", "Avg. Day Vol.", "Bid", "Ask", "Prev. Close",
213                    "Open", "Day Range", "52-Week Range", "Earnings/Share",
214                    "P/E Ratio", "Dividend/Share", "Dividend Yield"]
215        fw := 0
216        every fw <:= ∗(!fields)
217        fw +:= 2
218     }
219
220     yahoourl := "http://finance.yahoo.com/d/quotes.cvs?e=.csv&"||
221                 "f=snl1c1cva2bapomwerdy&s=" || stock
222     yahoo := open(yahoourl, "m",
223                    "User-Agent: Unicon",
224                    "X-Unicon: http://icon.cs.unlv.edu/")
225     if yahoo["Status-Code"] >= 300 then {
226        write(&errout, "Yahoo error: ", yahoo["Status-Code"], " ",
227              \yahoo["Reason-Phrase"] | "")
228        fail
229        }
230     dlist := ParseCSV(reads(yahoo, −1))[1]
231     close(yahoo)
232     s := list ()
233     every i := (1 to ∗fields ) do
234        put(s, left( fields [i ], fw ) || dlist [i ])
235     return s
236 end
237
238 procedure ParseCSV(csv)
239      local row, col , cell , parsed, nlchar
240
241      nlchar := '\n\r'
242      parsed := list ()
243      csv ? while line := tab(upto(nlchar)) do {
244           row := list ()
245           line ? while col := (tab(upto(',')) | tab(0)) do {
246                col ? {
247                     if ="\"" then {
248                          cell := ""
249                          while cell ||:= tab(upto('\\')) do {
250                               move(1)
251                               cell ||:= move(1)
252                          }
253                          cell ||:= tab(upto('"'))
```

```
254                 }
255                 else {
256                     cell := tab(upto(',')) | tab(0)
257                 }
258             }
259             put(row, cell)
260             move(1) | break
261         }
262         put(parsed, row)
263         tab(many(nlchar))
264     }
265     return parsed
266 end
267
268 procedure fmtChange(c, prec)
269     local s
270     /prec := 2
271     if c > 0 then s := "+"else s := ""
272     s ||:= integer(c * 10.0^prec) * (10.0^-prec)
273     return s
274 end
```

tpget

This program is a simple example of using the `libtp` library. It retrieves a file via HTTP and sends its contents to standard output.

Listing 14: tpget.c, using the `libtp` library

```
1  #include "tp.h"
2
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <errno.h>
7  #include <unistd.h>
8
9  Tpexcept_f tpexcept;
10
11 char* url;
12
13 int exception(int e, void* obj, Tpdisc_t* disc)
14 {
15   int rc = tpexcept(e, obj, disc);
16   if (rc == TP_RETURNERROR) {
17     if (errno != 0) {
18       perror(url);
19     }
20     else {
21       switch (e)
22       {
23         case TP_EHOST:
24           fputs(url, stderr); fputs(": Unknown host\n", stderr);
25           break;
26
27         default:
28           fputs(url, stderr); fputs(": Error connecting\n", stderr);
29       }
```

```
30       }
31       exit (1);
32    }
33    else {
34       return rc;
35    }
36 }
37
38 int main(int argc, char **argv)
39 {
40    Tp_t*           tp;
41    Tpdisc_t*       disc;
42    Tprequest_t     req = { GET, "User-Agent: libtp/0.0\r\n"};
43    Tpresponse_t* resp;
44
45    PURI puri;
46    int i;
47    char buf[8192];
48    ssize_t nread;
49
50    if (argc < 2) {
51       fprintf (stderr, "usage: %s url\n", argv[0]);
52       exit (1);
53    }
54
55    url = argv[1];
56    puri = uri_parse(url);
57    if (puri->status != URI_OK) {
58       fprintf (stderr, "%s: %s\n", url, _uri_errlist [puri->status]);
59       exit (1);
60    }
61
62    if (strcmp(puri->scheme, "http") != 0) {
63       fprintf (stderr, "%s: not a http URL\n", url);
64       exit (1);
65    }
66
67    disc = tp_newdisc(TpdUnix);
68    tpexcept = disc->exceptf;
69    disc->exceptf = exception;
70
71    tp = tp_new(puri, TpmHTTP, disc);
72    resp = tp_sendreq(tp, &req);
73    while((nread = tp_read(tp, buf, sizeof(buf)))) {
74       fwrite(buf, 1, nread, stdout);
75    }
76    tp_free(tp);
77
78    return 0;
79 }
```

wtrace

The *wtrace* program is an example of extending a provided discipline with new functionality. In this case, the disciplines are replaced with wrapper functions which trace the action of the internal read and write routines for debugging purposes. It also demonstrates how the exception handling function may be extended to exit gracefully on certain errors. This obviates the usual practice of checking the return value of every function call for errors and makes the program easier to read.

Listing 15: wtrace.c, extending a discipline

```
1  /* wtrace.c: Dump a trace of communication with a web server. */
2
3  /* This program makes an HTTP request (GET by default) and shows the
4   * communication that goes on between the server and the program, by
5   * replacing the default discipline with one that wraps readf,
6   * readlnf, and writef. */
7
8  #ifdef HAVE_CONFIG_H
9  #include "config.h"
10 #endif
11
12 #include "tp.h"
13
14 #ifdef STDC_HEADERS
15 #include <stdio.h>
16 #include <stdlib.h>
17 #include <string.h>
18 #endif
19
20 #ifdef HAVE_ERRNO_H
21 #include <errno.h>
22 #endif
23
24 #ifdef HAVE_SYS_ERRNO_H
25 #include <sys/errno.h>
26 #endif
27
28 #ifdef HAVE_UNISTD_H
29 #include <unistd.h>
30 #endif
31
32 Tpexcept_f tpexcept;
33 Tpread_f   tpread;
34 Tpreadln_f tpreadln;
35 Tpwrite_f  tpwrite;
36
37 char* url;
38
39 int exception(int e, void* obj, Tpdisc_t* disc)
40 {
41   int rc = tpexcept(e, obj, disc);
42   if (rc == TP_RETURNERROR) {
43     if (errno != 0) {
44       perror(url);
45     }
46     else {
47       switch (e)
```

```
48          {
49            case TP_EHOST:
50              fputs(url , stderr ); fputs(": Unknown host\n", stderr);
51              break;
52
53            default:
54              fputs(url , stderr ); fputs(": Error connecting\n", stderr);
55          }
56        }
57        exit (1);
58      }
59      else {
60        return rc;
61      }
62  }
63
64  ssize_t  readf(void* buf, size_t  n, Tpdisc_t* disc)
65  {
66    ssize_t  nread;
67
68    if ((nread = tpread(buf, n, disc)) <= 0) {
69      return nread;
70    }
71
72    if (write (2, "\n<<", 3) < 0) {
73      perror("write");
74      exit (1);
75    }
76
77    if (write (2, buf, nread) < 0) {
78      perror("write");
79      exit (1);
80    }
81
82    if (write (2, ">>\n", 3) < 0) {
83      perror("write");
84      exit (1);
85    }
86
87    return nread;
88  }
89
90  ssize_t  readlnf(void* buf, size_t  n, Tpdisc_t* disc)
91  {
92    ssize_t  nread;
93
94    if ((nread = tpreadln(buf, n, disc)) <= 0) {
95      return nread;
96    }
97
98    if (write (2, "\n<<", 3) < 0) {
99      perror("write");
100     exit (1);
101   }
102
103   if (write (2, buf, nread) < 0) {
104     perror("write");
105     exit (1);
106   }
```

```
107
108     if (write(2, ">>\n", 3) < 0) {
109       perror("write");
110       exit(1);
111     }
112
113     return nread;
114   }
115
116   ssize_t writef(void* buf, size_t n, Tpdisc_t* disc)
117   {
118     ssize_t nwritten;
119
120     if (write(2, "\n[[", 3) < 0) {
121       perror("write");
122       exit(1);
123     }
124
125     if (write(2, buf, n) < 0) {
126       perror("write");
127       exit(1);
128     }
129
130     if (write(2, "]]\n", 3) < 0) {
131       perror("write");
132       exit(1);
133     }
134
135     nwritten = tpwrite(buf, n, disc);
136     return nwritten;
137   }
138
139   int main(int argc, char **argv)
140   {
141     Tp_t*           tp;
142     Tpdisc_t*       disc;
143     Tprequest_t     req = { HEAD, "User-Agent: libtp/0.0\r\n"};
144     Tpresponse_t* resp;
145
146     PURI puri;
147     int i;
148     char* type = "head";
149     char buf[8192];
150
151     if (argc < 2) {
152       fprintf(stderr, "usage: %s url\n", argv[0]);
153       exit(1);
154     }
155
156     for (i=1; i<argc; ) {
157       if (argv[i][0] != '-') {
158         url = argv[i];
159         break;
160       }
161
162       switch (argv[i][1])
163       {
164         case 't':
165           type = argv[i+1];
```

```
166             i += 2;
167             break;
168 
169           default:
170             fprintf (stderr , "Unrecognized option: %s", argv[i]);
171             exit (1);
172         }
173     }
174 
175     if (strcasecmp(type, "get") == 0) {
176       req.type = GET;
177     }
178 
179     puri = uri_parse(url);
180     if (puri->status != URI_OK) {
181       fprintf (stderr , "%s: %s\n", url, _uri_errlist [puri->status]);
182       exit (1);
183     }
184 
185     if (strcmp(puri->scheme, "http") != 0) {
186       fprintf (stderr , "%s: not a http URL\n", url);
187       exit (1);
188     }
189 
190     disc = tp_newdisc(TpdUnix);
191     tpexcept = disc->exceptf;
192     disc->exceptf = exception;
193     tpread = disc->readf;
194     disc->readf = readf;
195     tpreadln = disc->readlnf;
196     disc->readlnf = readlnf;
197     tpwrite = disc->writef;
198     disc->writef = writef;
199 
200     tp = tp_new(puri, TpmHTTP, disc);
201     resp = tp_sendreq(tp, &req);
202     while(tp_read(tp, buf, sizeof(buf))) {
203       ; /* Everything is done by the tracing read */
204     }
205     tp_free(tp);
206 
207     return 0;
208 }
```

APPENDIX B

# UNICON MESSAGING REFERENCE

This is a reference for those Unicon operators and functions which have been extended. For a reference to the complete Unicon language, see [16].

## Operators

---

| | |
|---|---|
| !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 server the messaging file M refers to. A POP connection is treated as a list of messages, so !M will generate messages from 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 |

The first form returns a string containing the value of the specified header field in the server response. The special values `Status-Code` and `Reason-Phrase` refer to the result of the request and its descriptive text (e.g. 404 and "Not Found" for HTTP). All of the values are protocol dependent. Only `Status-Code` is guaranteed to be non-null and is filled in by the system if the protocol does not provide one. Status codes follow the convention whereby codes less than 300 mean success, 300–399 means a error occurred which may be correctable, and 400 or above means that a fatal error occurred.

The second form is only valid for POP connections. The expression M[n] returns the n-th message in the specified mailbox.

Examples:

```
M := open("http://icon.cs.unlv.edu/", "m")
if (M["Status-Code"] >= 300) then
  stop(M["Status-Code"], " ", (\M["Reason-Phrase"] | ""))
```

*(Note that you must handle the case where Reason-Phrase is null.)*

```
M := open("pop://user:password@pop.myisp.net", "m")
write(M[3]) # Writes the 3rd message in the mailbox
```

Functions

---

close(file) : file — close a messaging file

**close**(M) completes any pending request, closes the connection to the server and returns any 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 , ..., Nn) deletes all messages numbered Nx from a POP server and returns M. It always succeeds. As a feature of POP, messages are not actually 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.

---

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

**open**(U, "m", H1 , ..., Hn) connects to the Internet server specified by the URL U and sends H1 through Hn as headers for the request part of the transaction, or fails if the connection can not be made.

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 given 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.

Examples:

```
M := open("http://icon.cs.unlv.edu/", "m", "User-Agent: Unicon") # Web site

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 test",
          "X-Unicon: Sent with Unicon!") # Email message
```

---

pop(file) : string? — 'pop' message

**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 line

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

---

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

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

---

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

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

---

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

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

# APPENDIX C

## ACRONYMS

**FTP** File Transfer Protocol. An Internet protocol for interactively transfering files from one host to another [26].

**HTML** Hypertext Markup Language [32].

**HTTP** Hypertext Transfer Procotol. HTTP is the protocol used by the World Wide Web. It is described in RFC1945 [1] and RFC2616 [6].

**IP** Internet Protocol. A low-level networking protocol. Most other protocols are implemented on top of a networking layer that is in turn implemented on top of IP.

**ODBC** Open Database Connectivity. A standard, open program interface for accessing online relational databases [37].

**POP** Post Office Protocol. A standard protocol for downloading email over a network, usually to a personal computer or workstation [19]. Somtimes written as POP3 when referring to version 3 of the protocol.

**POSIX** Portable Operating System Interface. A standard for operating system interfaces, including programmer and end-user levels [38].

**REBOL** Relative Expression-based Object Language. REBOL is a messaging language. It is described in Chapter 1. See also [28].

**RFC** Request for Comments. Documents which define Internet standards.

**SMTP** Simple Mail Transfer Protocol. Used for transmission of email messages from one point to another, possibly relaying through several servers along the way [24].

**TCP** Transport Control Protocol. A low level networking protocol. Almost all of the other protocols cited usually operate on top of TCP, which in turn operates on top of IP (TCP/IP).

**UDP** User Datagram Protocol. A low level networking protocol. Unlike TCP, UDP does not have built-in features for flow-control or reliable transmission.

**URL** Uniform Resource Locator. A method for specifying an Internet resource, including protocol, hostname, and path [2, 5, 9, 12, 22].

**W3C** World Wide Web Consortium. An industry consortium/standards body which oversees the protocols and other standards used by servers and clients on the WWW.

**WWW** World Wide Web. The World Wide Web is a collection of standards for serving and accessing information over the Internet. The World Wide Web Consortium (W3C) [32] oversees most of those standards.

REFERENCES


[1] T. Berners-Lee, R. Fielding, and H. Frystyk. RFC 1945: Hypertext Transfer Protocol — HTTP/1.0, May 1996. Status: INFORMATIONAL.

[2] T. Berners-Lee, L. Masinter, and M. McCahill. RFC 1738: Uniform resource locators (URL), December 1994. Status: PROPOSED STANDARD.

[3] Thomas W. Christopher. *Icon Programming Language Handbook, Beta Version.* Tools of Computing LLC, P.O. Box 6335, Evanston IL, 60204-6335, 1996. Available online as <http://www.toolsofcomputing.com/iconprog.pdf>.

[4] Daniel Cooke, Joseph Urban, and Scott Hamilton. Unix and beyond: An interview with Ken Thompson. *Computer*, 32(5):58–64, May 1999.

[5] R. Fielding. RFC 1808: Relative uniform resource locators, June 1995.

[6] R. Fielding, J. Gettys, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol — HTTP/1.1, June 1999. Status: DRAFT STANDARD.

[7] N. Freed and N. Borenstein. RFC 2045: Multipurpose Internet Mail Extensions (MIME) part one: Format of Internet message bodies, November 1996. Status: DRAFT STANDARD.

[8] N. Freed and N. Borenstein. RFC 2046: Multipurpose Internet Mail Extensions (MIME) part two: Media types, November 1996. Status: DRAFT STANDARD.

[9] R. Gellens. RFC 2384: POP URL scheme, August 1998. Status: PROPOSED STANDARD.

[10] Ralph E. Griswold and Madge T. Griswold. *The Implementation of the Icon Programming Language*. Princeton University Press, Princeton, NJ, USA, 1986.

[11] Ralph E. Griswold and Madge T. Griswold. *The Icon programming language*. Peer-to-Peer Communications, San Jose, CA, USA, third edition, 1997.

[12] P. Hoffman, L. Masinter, and J. Zawinski. RFC 2368: The mailto URL scheme, July 1998.

[13] Peter E. Hoffman. finger url specification. <http://www.ics.uci.edu/pub/ietf/uri/draft-ietf-uri-url-finger-03.txt>. Note: This is an *expired* IETF draft, but there does not appear to have ever been an update.

[14] E. Huizer. RFC 1844: Multimedia E-mail (MIME) user agent checklist, August 1995. Status: INFORMATIONAL.

[15] Clinton Jeffery. The Unicon homepage. <http://icon.cs.unlv.edu/>.

[16] Clinton Jeffery, Shamim Mohamed, Robert Parlett, and Ray Pereda. *Programming with Unicon*. Draft manuscript. Available online as <http://icon.cs.unlv.edu/ib/ib.pdf>.

[17] L. Masinter. RFC 2388: Returning values from forms: multipart/form-data, August 1998. Status: PROPOSED STANDARD.

[18] J. Myers. RFC 1734: POP3 AUTHentication command, December 1994. Status: PROPOSED STANDARD.

[19] J. Myers and M. Rose. RFC 1939: Post Office Protocol — version 3, May 1996. Status: STANDARD.

[20] E. Nebel and L. Masinter. RFC 1867: Form-based file upload in HTML, November 1995. Status: EXPERIMENTAL.

[21] R. Nelson. RFC 1957: Some observations on implementations of the Post Office Protocol (POP3), June 1996. Status: INFORMATIONAL.

[22] C. Newman. RFC 2192: IMAP URL scheme, September 1997. Status: PROPOSED STANDARD.

[23] PHP: Hypertext preprocessor. <http://www.php.net/>.

[24] J. Postel. RFC 821: Simple mail transfer protocol, August 1982. Status: STANDARD.

[25] J. Postel. RFC 867: Daytime protocol, May 1983.

[26] J. Postel and J. K. Reynolds. RFC 959: File transfer protocol, October 1985. Status: STANDARD.

[27] REBOL in ten steps. <http://www.rebol.com/rebolsteps.html>. Accessed 2 May 2000.

[28] REBOL Technologies. <http://www.rebol.com/>. For general information on the Rebol language.

[29] What is a messaging language? <http://www.rebol.com/>. Accessed 27 October 1999, no longer available online.

[30] New messaging language REBOL transforms information exchange over networks. <http://www.rebol.com/news8a01.html>, October 1998. Accessed 2 May 2000.

[31] D. Smallberg. RFC 876: Survey of SMTP implementations, September 1983. Status: UNKNOWN.

[32] (homepage) the World Wide Web Consortium. <http://www.w3c.org/>.

[33] Various. REBOL the “messaging language”. <http://slashdot.org/articles/99/05/14/1921231.shtml>, May 1999. Accessed 2 May 2000.

[34] Kiem-Phong Vo. Vmalloc: A general and efficient memory allocator. *Software—Practice and Experience*, 26(3):357–374, March 1996. Also available online as <http://www.research.att.com/sw/tools/vmalloc/vmalloc.ps>.

[35] Kiem-Phong Vo. An architecture for reusable libraries. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 184–195. IEEE Computer Society Press, 1998. Also available online as <http://www.research.att.com/sw/tools/sfio/dm.ps>.

[36] Kenneth Walker. A run-time implementation language for Icon. Technical Report IPD261, The University of Arizona Icon Project, June 28 1994.

[37] What is... ODBC. <http://www.whatis.com/odbc.htm>. Accessed 2 May 2000.

[38] What is... POSIX. <http://www.whatis.com/posix.htm>. Accessed 2 May 2000.

[39] Yahoo! finance. <http://finance.yahoo.com/>. Accessed 2 May 2000.

[40] D. Zimmerman. RFC 1288: The finger user information protocol, December 1991. Status: DRAFT STANDARD.