TCP/IP Internetworking With `gawk`

Edition 1.1
April, 2002

Jürgen Kahrs
with Arnold D. Robbins

Published by:

Free Software Foundation
59 Temple Place — Suite 330
Boston, MA 02111-1307 USA
Phone: +1-617-542-5942
Fax: +1-617-542-2652
Email: gnu@gnu.org
URL: http://www.gnu.org/

ISBN 1-882114-93-0

This is Edition 1.1 of *TCP/IP Internetworking With* `gawk`, for the 3.1.1 (or later) version of the GNU implementation of AWK.

Table of Contents

Preface

In May of 1997, Jürgen Kahrs felt the need for network access from `awk`, and, with a little help from me, set about adding features to do this for `gawk`. At that time, he wrote the bulk of this book.

The code and documentation were added to the `gawk` 3.1 development tree, and languished somewhat until I could finally get down to some serious work on that version of `gawk`. This finally happened in the middle of 2000.

Meantime, Jürgen wrote an article about the Internet special files and '`|&`' operator for *Linux Journal*, and made a networking patch for the production versions of `gawk` available from his home page. In August of 2000 (for `gawk` 3.0.6), this patch also made it to the main GNU `ftp` distribution site.

For release with `gawk`, I edited Jürgen's prose for English grammar and style, as he is not a native English speaker. I also rearranged the material somewhat for what I felt was a better order of presentation, and (re)wrote some of the introductory material.

The majority of this document and the code are his work, and the high quality and interesting ideas speak for themselves. It is my hope that these features will be of significant value to the `awk` community.

Arnold Robbins
Nof Ayalon, ISRAEL
March, 2001

2 TCP/IP Internetworking With gawk

1 Networking Concepts

This chapter provides a (necessarily) brief intoduction to computer networking concepts. For many applications of `gawk` to TCP/IP networking, we hope that this is enough. For more advanced tasks, you will need deeper background, and it may be necessary to switch to lower-level programming in C or C++.

There are two real-life models for the way computers send messages to each other over a network. While the analogies are not perfect, they are close enough to convey the major concepts. These two models are the phone system (reliable byte-stream communications), and the postal system (best-effort datagrams).

1.1 Reliable Byte-streams (Phone Calls)

When you make a phone call, the following steps occur:

1. You dial a number.
2. The phone system connects to the called party, telling them there is an incoming call. (Their phone rings.)
3. The other party answers the call, or, in the case of a computer network, refuses to answer the call.
4. Assuming the other party answers, the connection between you is now a *duplex* (two-way), *reliable* (no data lost), sequenced (data comes out in the order sent) data stream.
5. You and your friend may now talk freely, with the phone system moving the data (your voices) from one end to the other. From your point of view, you have a direct end-to-end connection with the person on the other end.

The same steps occur in a duplex reliable computer networking connection. There is considerably more overhead in setting up the communications, but once it's done, data moves in both directions, reliably, in sequence.

1.2 Best-effort Datagrams (Mailed Letters)

Suppose you mail three different documents to your office on the other side of the country on two different days. Doing so entails the following.

1. Each document travels in its own envelope.
2. Each envelope contains both the sender and the recipient address.
3. Each envelope may travel a different route to its destination.
4. The envelopes may arrive in a different order from the one in which they were sent.

5. One or more may get lost in the mail. (Although, fortunately, this does not occur very often.)
6. In a computer network, one or more *packets* may also arrive multiple times. (This doesn't happen with the postal system!)

The important characteristics of datagram communications, like those of the postal system are thus:

- Delivery is "best effort;" the data may never get there.
- Each message is self-contained, including the source and destination addresses.
- Delivery is *not* sequenced; packets may arrive out of order, and/or multiple times.
- Unlike the phone system, overhead is considerably lower. It is not necessary to set up the call first.

The price the user pays for the lower overhead of datagram communications is exactly the lower reliability; it is often necessary for user-level protocols that use datagram communications to add their own reliability features on top of the basic communications.

1.3 The Internet Protocols

The Internet Protocol Suite (usually referred to as just TCP/IP)[1] consists of a number of different protocols at different levels or "layers." For our purposes, three protocols provide the fundamental communications mechanisms. All other defined protocols are referred to as user-level protocols (e.g., HTTP, used later in this book).

1.3.1 The Basic Internet Protocols

IP
: The Internet Protocol. This protocol is almost never used directly by applications. It provides the basic packet delivery and routing infrastructure of the Internet. Much like the phone company's switching centers or the Post Office's trucks, it is not of much day-to-day interest to the regular user (or programmer). It happens to be a best effort datagram protocol.

UDP
: The User Datagram Protocol. This is a best effort datagram protocol. It provides a small amount of extra reliability over IP, and adds the notion of *ports*, described in Section 1.3.2 [TCP and UDP Ports], page 5.

---

[1] It should be noted that although the Internet seems to have conquered the world, there are other networking protocol suites in existence and in use.

TCP The Transmission Control Protocol. This is a duplex, reliable, sequenced byte-stream protocol, again layered on top of IP, and also providing the notion of ports. This is the protocol that you will most likely use when using `gawk` for network programming.

All other user-level protocols use either TCP or UDP to do their basic communications. Examples are SMTP (Simple Mail Transfer Protocol), FTP (File Transfer Protocol), and HTTP (HyperText Transfer Protocol).

1.3.2 TCP and UDP Ports

In the postal system, the address on an envelope indicates a physical location, such as a residence or office building. But there may be more than one person at a location; thus you have to further quantify the recipient by putting a person or company name on the envelope.

In the phone system, one phone number may represent an entire company, in which case you need a person's extension number in order to reach that individual directly. Or, when you call a home, you have to say, "May I please speak to ..." before talking to the person directly.

IP networking provides the concept of addressing. An IP address represents a particular computer, but no more. In order to reach the mail service on a system, or the FTP or WWW service on a system, you must have some way to further specify which service you want. In the Internet Protocol suite, this is done with *port numbers*, which represent the services, much like an extension number used with a phone number.

Port numbers are 16-bit integers. Unix and Unix-like systems reserve ports below 1024 for "well known" services, such as SMTP, FTP, and HTTP. Numbers 1024 and above may be used by any application, although there is no promise made that a particular port number is always available.

1.4 Making TCP/IP Connections (And Some Terminology)

Two terms come up repeatedly when discussing networking: *client* and *server*. For now, we'll discuss these terms at the *connection level*, when first establishing connections between two processes on different systems over a network. (Once the connection is established, the higher level, or *application level* protocols, such as HTTP or FTP, determine who is the client and who is the server. Often, it turns out that the client and server are the same in both roles.)

The *server* is the system providing the service, such as the web server or email server. It is the *host* (system) which is *connected to* in a transaction. For this to work though, the server must be expecting connections. Much as there has to be someone at the office building to answer the phone[2], the server process (usually) has to be started first and be waiting for a connection.

The *client* is the system requesting the service. It is the system *initiating the connection* in a transaction. (Just as when you pick up the phone to call an office or store.)

In the TCP/IP framework, each end of a connection is represented by a pair of (*address*, *port*) pairs. For the duration of the connection, the ports in use at each end are unique, and cannot be used simultaneously by other processes on the same system. (Only after closing a connection can a new one be built up on the same port. This is contrary to the usual behavior of fully developed web servers which have to avoid situations in which they are not reachable. We have to pay this price in order to enjoy the benefits of a simple communication paradigm in `gawk`.)

Furthermore, once the connection is established, communications are *synchronous*.[3] I.e., each end waits on the other to finish transmitting, before replying. This is much like two people in a phone conversation. While both could talk simultaneously, doing so usually doesn't work too well.

In the case of TCP, the synchronicity is enforced by the protocol when sending data. Data writes *block* until the data have been received on the other end. For both TCP and UDP, data reads block until there is incoming data waiting to be read. This is summarized in the following table, where an "X" indicates that the given action blocks.

Protocol	Reads	Writes
TCP	X	X
UDP	X	
RAW	X	

[2] In the days before voice mail systems!

[3] For the technically savvy, data reads block—if there's no incoming data, the program is made to wait until there is, instead of receiving a "there's no data" error return.

2 Networking With gawk

The awk programming language was originally developed as a pattern-matching language for writing short programs to perform data manipulation tasks. awk's strength is the manipulation of textual data that is stored in files. It was never meant to be used for networking purposes. To exploit its features in a networking context, it's necessary to use an access mode for network connections that resembles the access of files as closely as possible.

awk is also meant to be a prototyping language. It is used to demonstrate feasibility and to play with features and user interfaces. This can be done with file-like handling of network connections. gawk trades the lack of many of the advanced features of the TCP/IP family of protocols for the convenience of simple connection handling. The advanced features are available when programming in C or Perl. In fact, the network programming in this chapter is very similar to what is described in books such as *Internet Programming with Python*, *Advanced Perl Programming*, or *Web Client Programming with Perl*.

However, you can do the programming here without first having to learn object-oriented ideology; underlying languages such as Tcl/Tk, Perl, Python; or all of the libraries necessary to extend these languages before they are ready for the Internet.

This chapter demonstrates how to use the TCP protocol. The other protocols are much less important for most users (UDP) or even untractable (RAW).

2.1 gawk's Networking Mechanisms

The '|&' operator introduced in gawk 3.1 for use in communicating with a *coprocess* is described in section "Two-way Communications With Another Process" in *GAWK: Effective AWK Programming*. It shows how to do two-way I/O to a separate process, sending it data with print or printf and reading data with getline. If you haven't read it already, you should detour there to do so.

gawk transparently extends the two-way I/O mechanism to simple networking through the use of special file names. When a "coprocess" that matches the special files we are about to describe is started, gawk creates the appropriate network connection, and then two-way I/O proceeds as usual.

At the C, C++, and Perl level, networking is accomplished via *sockets*, an Application Programming Interface (API) originally developed at the University of California at Berkeley that is now used almost universally for TCP/IP networking. Socket level programming, while fairly straightforward, requires paying attention to a number of details, as well as using binary data. It is not well-suited for use from a high-level language like awk. The special files provided in gawk hide the details from the programmer, making things much simpler and easier to use.

The special file name for network access is made up of several fields, all of which are mandatory:

`/inet/`*`protocol`*`/`*`localport`*`/`*`hostname`*`/`*`remoteport`*

The '`/inet/`' field is, of course, constant when accessing the network. The *localport* and *remoteport* fields do not have a meaning when used with '`/inet/raw`' because "ports" only apply to TCP and UDP. So, when using '`/inet/raw`', the port fields always have to be '`0`'.

2.1.1 The Fields of the Special File Name

This section explains the meaning of all the other fields, as well as the range of values and the defaults. All of the fields are mandatory. To let the system pick a value, or if the field doesn't apply to the protocol, specify it as '`0`':

protocol Determines which member of the TCP/IP family of protocols is selected to transport the data across the network. There are three possible values (always written in lowercase): '`tcp`', '`udp`', and '`raw`'. The exact meaning of each is explained later in this section.

localport Determines which port on the local machine is used to communicate across the network. It has no meaning with '`/inet/raw`' and must therefore be '`0`'. Application-level clients usually use '`0`' to indicate they do not care which local port is used—instead they specify a remote port to connect to. It is vital for application-level servers to use a number different from '`0`' here because their service has to be available at a specific publicly known port number. It is possible to use a name from '`/etc/services`' here.

hostname Determines which remote host is to be at the other end of the connection. Application-level servers must fill this field with a '`0`' to indicate their being open for all other hosts to connect to them and enforce connection level server behavior this way. It is not possible for an application-level server to restrict its availability to one remote host by entering a host name here. Application-level clients must enter a name different from '`0`'. The name can be either symbolic (e.g., '`jpl-devvax.jpl.nasa.gov`') or numeric (e.g., '`128.149.1.143`').

remoteport
Determines which port on the remote machine is used to communicate across the network. It has no meaning with '`/inet/raw`' and must therefore be 0. For '`/inet/tcp`' and '`/inet/udp`', application-level clients *must* use a number other than '`0`' to indicate to which port on the remote machine they want to connect. Application-level servers must not fill this field with a '`0`'.

Instead they specify a local port to which clients connect. It is possible to use a name from '/etc/services' here.

Experts in network programming will notice that the usual client/server asymmetry found at the level of the socket API is not visible here. This is for the sake of simplicity of the high-level concept. If this asymmetry is necessary for your application, use another language. For gawk, it is more important to enable users to write a client program with a minimum of code. What happens when first accessing a network connection is seen in the following pseudocode:

```
if ((name of remote host given) && (other side accepts connection)) {
  rendez-vous successful; transmit with getline or print
} else {
  if ((other side did not accept) && (localport == 0))
    exit unsuccessful
  if (TCP) {
    set up a server accepting connections
    this means waiting for the client on the other side to connect
  } else
    ready
}
```

The exact behavior of this algorithm depends on the values of the fields of the special file name. When in doubt, the following table gives you the combinations of values and their meaning. If this table is too complicated, focus on the three lines printed in **bold**. All the examples in Chapter 2 [Networking With gawk], page 7, use only the patterns printed in bold letters.

PROTOCOL	LOCAL PORT	HOST NAME	REMOTE PORT	RESULTING CONNECTION-LEVEL BEHAVIOR
tcp	**0**	**x**	**x**	**Dedicated client, fails if immediately connecting to a server on the other side fails**
udp	0	x	x	Dedicated client
raw	0	x	0	Dedicated client, works only as root
tcp, udp	**x**	**x**	**x**	**Client, switches to dedicated server if necessary**
tcp, udp	**x**	**0**	**0**	**Dedicated server**
raw	0	0	0	Dedicated server, works only as root
tcp, udp, raw	x	x	0	Invalid
tcp, udp, raw	0	0	x	Invalid
tcp, udp, raw	x	0	x	Invalid
tcp, udp	0	0	0	Invalid
tcp, udp	0	x	0	Invalid
raw	x	0	0	Invalid
raw	0	x	x	Invalid
raw	x	x	x	Invalid

In general, TCP is the preferred mechanism to use. It is the simplest protocol to understand and to use. Use the others only if circumstances demand low-overhead.

2.1.2 Comparing Protocols

This section develops a pair of programs (sender and receiver) that do nothing but send a timestamp from one machine to another. The sender and the receiver are implemented with each of the three protocols available and demonstrate the differences between them.

2.1.2.1 '/inet/tcp'

Once again, always use TCP. (Use UDP when low overhead is a necessity, and use RAW for network experimentation.) The first example is the sender program:

```
# Server
BEGIN {
  print strftime() |& "/inet/tcp/8888/0/0"
  close("/inet/tcp/8888/0/0")
}
```

The receiver is very simple:

```
# Client
BEGIN {
  "/inet/tcp/0/localhost/8888" |& getline
  print $0
  close("/inet/tcp/0/localhost/8888")
}
```

TCP guarantees that the bytes arrive at the receiving end in exactly the same order that they were sent. No byte is lost (except for broken connections), doubled, or out of order. Some overhead is necessary to accomplish this, but this is the price to pay for a reliable service. It does matter which side starts first. The sender/server has to be started first, and it waits for the receiver to read a line.

2.1.2.2 '/inet/udp'

The server and client programs that use UDP are almost identical to their TCP counterparts; only the *protocol* has changed. As before, it does matter which side starts first. The receiving side blocks and waits for the sender. In this case, the receiver/client has to be started first:

```
# Server
BEGIN {
  print strftime() |& "/inet/udp/8888/0/0"
  close("/inet/udp/8888/0/0")
}
```

The receiver is almost identical to the TCP receiver:

```
# Client
BEGIN {
  "/inet/udp/0/localhost/8888" |& getline
  print $0
  close("/inet/udp/0/localhost/8888")
}
```

UDP cannot guarantee that the datagrams at the receiving end will arrive in exactly the same order they were sent. Some datagrams could be lost, some doubled, and some out of order. But no overhead is necessary to accomplish this. This unreliable behavior is good enough for tasks such as data acquisition, logging, and even stateless services like NFS.

2.1.2.3 '`/inet/raw`'

This is an IP-level protocol. Only `root` is allowed to access this special file. It is meant to be the basis for implementing and experimenting with transport-level protocols.[1] In the most general case, the sender has to supply the encapsulating header bytes in front of the packet and the receiver has to strip the additional bytes from the message.

RAW receivers cannot receive packets sent with TCP or UDP because the operating system does not deliver the packets to a RAW receiver. The operating system knows about some of the protocols on top of IP and decides on its own which packet to deliver to which process. Therefore, the UDP receiver must be used for receiving UDP datagrams sent with the RAW sender. This is a dark corner, not only of `gawk`, but also of TCP/IP.

For extended experimentation with protocols, look into the approach implemented in a tool called SPAK. This tool reflects the hierarchical layering of protocols (encapsulation) in the way data streams are piped out of one program into the next one. It shows which protocol is based on which other (lower-level) protocol by looking at the command-line ordering of the program calls. Cleverly thought out, SPAK is much better than `gawk`'s '`/inet`' for learning the meaning of each and every bit in the protocol headers.

The next example uses the RAW protocol to emulate the behavior of UDP. The sender program is the same as above, but with some additional bytes that fill the places of the UDP fields:

---

[1] This special file is reserved, but not otherwise currently implemented.

```
BEGIN {
  Message = "Hello world\n"
  SourcePort = 0
  DestinationPort = 8888
  MessageLength = length(Message)+8
  RawService = "/inet/raw/0/localhost/0"
  printf("%c%c%c%c%c%c%c%c%s",
      SourcePort/256, SourcePort%256,
      DestinationPort/256, DestinationPort%256,
      MessageLength/256, MessageLength%256,
      0, 0, Message) |& RawService
  fflush(RawService)
  close(RawService)
}
```

Since this program tries to emulate the behavior of UDP, it checks if the RAW sender is understood by the UDP receiver but not if the RAW receiver can understand the UDP sender. In a real network, the RAW receiver is hardly of any use because it gets every IP packet that comes across the network. There are usually so many packets that `gawk` would be too slow for processing them. Only on a network with little traffic can the IP-level receiver program be tested. Programs for analyzing IP traffic on modem or ISDN channels should be possible.

Port numbers do not have a meaning when using '`/inet/raw`'. Their fields have to be '`0`'. Only TCP and UDP use ports. Receiving data from '`/inet/raw`' is difficult, not only because of processing speed but also because data is usually binary and not restricted to ASCII. This implies that line separation with `RS` does not work as usual.

2.2 Establishing a TCP Connection

Let's observe a network connection at work. Type in the following program and watch the output. Within a second, it connects via TCP ('`/inet/tcp`') to the machine it is running on ('`localhost`') and asks the service '`daytime`' on the machine what time it is:

```
BEGIN {
  "/inet/tcp/0/localhost/daytime" |& getline
  print $0
  close("/inet/tcp/0/localhost/daytime")
}
```

Even experienced `awk` users will find the second line strange in two respects:

- A special file is used as a shell command that pipes its output into `getline`. One would rather expect to see the special file being read like any other file ('`getline < "/inet/tcp/0/localhost/daytime")`').

- The operator '|&' has not been part of any awk implementation (until now). It is actually the only extension of the awk language needed (apart from the special files) to introduce network access.

The '|&' operator was introduced in gawk 3.1 in order to overcome the crucial restriction that access to files and pipes in awk is always unidirectional. It was formerly impossible to use both access modes on the same file or pipe. Instead of changing the whole concept of file access, the '|&' operator behaves exactly like the usual pipe operator except for two additions:

- Normal shell commands connected to their gawk program with a '|&' pipe can be accessed bidirectionally. The '|&' turns out to be a quite general, useful, and natural extension of awk.
- Pipes that consist of a special file name for network connections are not executed as shell commands. Instead, they can be read and written to, just like a full-duplex network connection.

In the earlier example, the '|&' operator tells getline to read a line from the special file '/inet/tcp/0/localhost/daytime'. We could also have printed a line into the special file. But instead we just read a line with the time, printed it, and closed the connection. (While we could just let gawk close the connection by finishing the program, in this book we are pedantic and always explicitly close the connections.)

2.3 Troubleshooting Connection Problems

It may well be that for some reason the program shown in the previous example does not run on your machine. When looking at possible reasons for this, you will learn much about typical problems that arise in network programming. First of all, your implementation of gawk may not support network access because it is a pre-3.1 version or you do not have a network interface in your machine. Perhaps your machine uses some other protocol, such as DECnet or Novell's IPX. For the rest of this chapter, we will assume you work on a Unix machine that supports TCP/IP. If the previous example program does not run on your machine, it may help to replace the name 'localhost' with the name of your machine or its IP address. If it does, you could replace 'localhost' with the name of another machine in your vicinity—this way, the program connects to another machine. Now you should see the date and time being printed by the program, otherwise your machine may not support the 'daytime' service. Try changing the service to 'chargen' or 'ftp'. This way, the program connects to other services that should give you some response. If you are curious, you should have a look at your '/etc/services' file. It could look like this:

```
# /etc/services:
#
# Network services, Internet style
#
```

```
# Name     Number/Protcol  Alternate name # Comments

echo        7/tcp
echo        7/udp
discard     9/tcp          sink null
discard     9/udp          sink null
daytime     13/tcp
daytime     13/udp
chargen     19/tcp         ttytst source
chargen     19/udp         ttytst source
ftp         21/tcp
telnet      23/tcp
smtp        25/tcp         mail
finger      79/tcp
www         80/tcp         http      # WorldWideWeb HTTP
www         80/udp         # HyperText Transfer Protocol
pop-2       109/tcp        postoffice    # POP version 2
pop-2       109/udp
pop-3       110/tcp        # POP version 3
pop-3       110/udp
nntp        119/tcp        readnews untp  # USENET News
irc         194/tcp        # Internet Relay Chat
irc         194/udp
...
```

Here, you find a list of services that traditional Unix machines usually support. If your GNU/Linux machine does not do so, it may be that these services are switched off in some startup script. Systems running some flavor of Microsoft Windows usually do *not* support these services. Nevertheless, it *is* possible to do networking with `gawk` on Microsoft Windows.[2] The first column of the file gives the name of the service, and the second column gives a unique number and the protocol that one can use to connect to this service. The rest of the line is treated as a comment. You see that some services ('`echo`') support TCP as well as UDP.

2.4 Interacting with a Network Service

The next program makes use of the possibility to really interact with a network service by printing something into the special file. It asks the so-called `finger` service if a user of the machine is logged in. When testing this program, try to change '`localhost`' to some other machine name in your local network:

[2] Microsoft prefered to ignore the TCP/IP family of protocols until 1995. Then came the rise of the Netscape browser as a landmark "killer application." Microsoft added TCP/IP support and their own browser to Microsoft Windows 95 at the last minute. They even back-ported their TCP/IP implementation to Microsoft Windows for Workgroups 3.11, but it was a rather rudimentary and half-hearted implementation. Nevertheless, the equivalent of '`/etc/services`' resides under '`C:\WINNT\system32\drivers\etc\services`' on Microsoft Windows 2000.

```
BEGIN {
  NetService = "/inet/tcp/0/localhost/finger"
  print "name" |& NetService
  while ((NetService |& getline) > 0)
    print $0
  close(NetService)
}
```

After telling the service on the machine which user to look for, the program repeatedly reads lines that come as a reply. When no more lines are coming (because the service has closed the connection), the program also closes the connection. Try replacing `"name"` with your login name (or the name of someone else logged in). For a list of all users currently logged in, replace *name* with an empty string (`""`).

The final `close` command could be safely deleted from the above script, because the operating system closes any open connection by default when a script reaches the end of execution. In order to avoid portability problems, it is best to always close connections explicitly. With the Linux kernel, for example, proper closing results in flushing of buffers. Letting the close happen by default may result in discarding buffers.

When looking at '`/etc/services`' you may have noticed that the '`daytime`' service is also available with '`udp`'. In the earlier example, change '`tcp`' to '`udp`', and change '`finger`' to '`daytime`'. After starting the modified program, you see the expected day and time message. The program then hangs, because it waits for more lines coming from the service. However, they never come. This behavior is a consequence of the differences between TCP and UDP. When using UDP, neither party is automatically informed about the other closing the connection. Continuing to experiment this way reveals many other subtle differences between TCP and UDP. To avoid such trouble, one should always remember the advice Douglas E. Comer and David Stevens give in Volume III of their series *Internetworking With TCP* (page 14):

> When designing client-server applications, beginners are strongly advised to use TCP because it provides reliable, connection-oriented communication. Programs only use UDP if the application protocol handles reliability, the application requires hardware broadcast or multicast, or the application cannot tolerate virtual circuit overhead.

2.5 Setting Up a Service

The preceding programs behaved as clients that connect to a server somewhere on the Internet and request a particular service. Now we set up such a service to mimic the behavior of the '`daytime`' service. Such a server does not know in advance who is going to connect to it over the network. Therefore, we cannot insert a name for the host to connect to in our special file name.

Start the following program in one window. Notice that the service does not have the name '`daytime`', but the number '`8888`'. From looking at '`/etc/services`', you know that names like '`daytime`' are just mnemonics for predetermined 16-bit integers. Only the system administrator (`root`) could enter our new service into '`/etc/services`' with an appropriate name. Also notice that the service name has to be entered into a different field of the special file name because we are setting up a server, not a client:

```
BEGIN {
  print strftime() |& "/inet/tcp/8888/0/0"
  close("/inet/tcp/8888/0/0")
}
```

Now open another window on the same machine. Copy the client program given as the first example (see Section 2.2 [Establishing a TCP Connection], page 12) to a new file and edit it, changing the name '`daytime`' to '`8888`'. Then start the modified client. You should get a reply like this:

```
Sat Sep 27 19:08:16 CEST 1997
```

Both programs explicitly close the connection.

Now we will intentionally make a mistake to see what happens when the name '`8888`' (the so-called port) is already used by another service. Start the server program in both windows. The first one works, but the second one complains that it could not open the connection. Each port on a single machine can only be used by one server program at a time. Now terminate the server program and change the name '`8888`' to '`echo`'. After restarting it, the server program does not run any more, and you know why: there is already an '`echo`' service running on your machine. But even if this isn't true, you would not get your own '`echo`' server running on a Unix machine, because the ports with numbers smaller than 1024 ('`echo`' is at port 7) are reserved for `root`. On machines running some flavor of Microsoft Windows, there is no restriction that reserves ports 1 to 1024 for a privileged user; hence, you can start an '`echo`' server there.

Turning this short server program into something really useful is simple. Imagine a server that first reads a file name from the client through the network connection, then does something with the file and sends a result back to the client. The server-side processing could be:

```
BEGIN {
  NetService = "/inet/tcp/8888/0/0"
  NetService |& getline
  CatPipe    = ("cat " $1)    # sets $0 and the fields
  while ((CatPipe | getline) > 0)
    print $0 |& NetService
  close(NetService)
}
```

and we would have a remote copying facility. Such a server reads the name of a file from any client that connects to it and transmits the contents of the named file across the net. The server-side processing could also be the execution of a command that is transmitted across the network. From this example, you can see how simple it is to open up a security hole on your machine. If you allow clients to connect to your machine and execute arbitrary commands, anyone would be free to do '`rm -rf *`'.

2.6 Reading Email

The distribution of email is usually done by dedicated email servers that communicate with your machine using special protocols. To receive email, we will use the Post Office Protocol (POP). Sending can be done with the much older Simple Mail Transfer Protocol (SMTP).

When you type in the following program, replace the *emailhost* by the name of your local email server. Ask your administrator if the server has a POP service, and then use its name or number in the program below. Now the program is ready to connect to your email server, but it will not succeed in retrieving your mail because it does not yet know your login name or password. Replace them in the program and it shows you the first email the server has in store:

```
BEGIN {
  POPService  = "/inet/tcp/0/emailhost/pop3"
  RS = ORS = "\r\n"
  print "user name"            |& POPService
  POPService                    |& getline
  print "pass password"         |& POPService
  POPService                    |& getline
  print "retr 1"                |& POPService
  POPService                    |& getline
  if ($1 != "+OK") exit
  print "quit"                  |& POPService
  RS = "\r\n\\.\r\n"
  POPService |& getline
  print $0
  close(POPService)
}
```

The record separators `RS` and `ORS` are redefined because the protocol (POP) requires CR-LF to separate lines. After identifying yourself to the email service, the command '`retr 1`' instructs the service to send the first of all your email messages in line. If the service replies with something other than '`+OK`', the program exits; maybe there is no email. Otherwise, the program first announces that it intends to finish reading email, and then redefines `RS` in order to read the entire email as multiline input in one record. From the POP RFC, we know that the body of the email always ends with a single line containing a single dot. The program looks for this using '`RS = "\r\n\\.\r\n"`'. When it finds this sequence in the mail message, it quits. You can invoke this program as often as you like; it does not delete the message it reads, but instead leaves it on the server.

2.7 Reading a Web Page

Retrieving a web page from a web server is as simple as retrieving email from an email server. We only have to use a similar, but not identical, protocol and a different port. The name of the protocol is HyperText Transfer Protocol (HTTP) and the port number is usually 80. As in the preceding section, ask your administrator about the name of your local web server or proxy web server and its port number for HTTP requests.

The following program employs a rather crude approach toward retrieving a web page. It uses the prehistoric syntax of HTTP 0.9, which almost all web servers still support. The most noticeable thing about it is that the program directs the request to the local proxy server whose name you insert in the special file name (which in turn calls '`www.yahoo.com`'):

```
BEGIN {
  RS = ORS = "\r\n"
  HttpService = "/inet/tcp/0/proxy/80"
  print "GET http://www.yahoo.com"     |& HttpService
  while ((HttpService |& getline) > 0)
     print $0
  close(HttpService)
}
```

Again, lines are separated by a redefined `RS` and `ORS`. The `GET` request that we send to the server is the only kind of HTTP request that existed when the web was created in the early 1990s. HTTP calls this `GET` request a "method," which tells the service to transmit a web page (here the home page of the Yahoo! search engine). Version 1.0 added the request methods `HEAD` and `POST`. The current version of HTTP is 1.1,[3] and knows the additional request methods `OPTIONS`, `PUT`, `DELETE`, and `TRACE`. You can fill in any valid web address, and the program prints the HTML code of that page to your screen.

Notice the similarity between the responses of the POP and HTTP services. First, you get a header that is terminated by an empty line, and then you get the body of the page in HTML. The lines of the headers also have the same form as in POP. There is the name of a parameter, then a colon, and finally the value of that parameter.

[3] Version 1.0 of HTTP was defined in RFC 1945. HTTP 1.1 was initially specified in RFC 2068. In June 1999, RFC 2068 was made obsolete by RFC 2616, an update without any substantial changes.

Images (‘.png’ or ‘.gif’ files) can also be retrieved this way, but then you get binary data that should be redirected into a file. Another application is calling a CGI (Common Gateway Interface) script on some server. CGI scripts are used when the contents of a web page are not constant, but generated instantly at the moment you send a request for the page. For example, to get a detailed report about the current quotes of Motorola stock shares, call a CGI script at Yahoo! with the following:

```
get = "GET http://quote.yahoo.com/q?s=MOT&d=t"
print get |& HttpService
```

You can also request weather reports this way.

2.8 A Primitive Web Service

Now we know enough about HTTP to set up a primitive web service that just says `"Hello, world"` when someone connects to it with a browser. Compared to the situation in the preceding section, our program changes the role. It tries to behave just like the server we have observed. Since we are setting up a server here, we have to insert the port number in the ‘localport’ field of the special file name. The other two fields (*hostname* and *remoteport*) have to contain a ‘0’ because we do not know in advance which host will connect to our service.

In the early 1990s, all a server had to do was send an HTML document and close the connection. Here, we adhere to the modern syntax of HTTP. The steps are as follows:

1. Send a status line telling the web browser that everything is okay.
2. Send a line to tell the browser how many bytes follow in the body of the message. This was not necessary earlier because both parties knew that the document ended when the connection closed. Nowadays it is possible to stay connected after the transmission of one web page. This is to avoid the network traffic necessary for repeatedly establishing TCP connections for requesting several images. Thus, there is the need to tell the receiving party how many bytes will be sent. The header is terminated as usual with an empty line.
3. Send the `"Hello, world"` body in HTML. The useless `while` loop swallows the request of the browser. We could actually omit the loop, and on most machines the program would still work. First, start the following program:

```
BEGIN {
  RS = ORS = "\r\n"
  HttpService = "/inet/tcp/8080/0/0"
  Hello = "<HTML><HEAD>" \
          "<TITLE>A Famous Greeting</TITLE></HEAD>" \
          "<BODY><H1>Hello, world</H1></BODY></HTML>"
  Len = length(Hello) + length(ORS)
```

```
    print "HTTP/1.0 200 OK"           |& HttpService
    print "Content-Length: " Len ORS |& HttpService
    print Hello                       |& HttpService
    while ((HttpService |& getline) > 0)
       continue;
    close(HttpService)
  }
```

Now, on the same machine, start your favorite browser and let it point to `http://localhost:8080` (the browser needs to know on which port our server is listening for requests). If this does not work, the browser probably tries to connect to a proxy server that does not know your machine. If so, change the browser's configuration so that the browser does not try to use a proxy to connect to your machine.

2.9 A Web Service with Interaction

Setting up a web service that allows user interaction is more difficult and shows us the limits of network access in `gawk`. In this section, we develop a main program (a `BEGIN` pattern and its action) that will become the core of event-driven execution controlled by a graphical user interface (GUI). Each HTTP event that the user triggers by some action within the browser is received in this central procedure. Parameters and menu choices are extracted from this request, and an appropriate measure is taken according to the user's choice. For example:

```
BEGIN {
  if (MyHost == "") {
     "uname -n" | getline MyHost
     close("uname -n")
  }
  if (MyPort ==  0) MyPort = 8080
  HttpService = "/inet/tcp/" MyPort "/0/0"
  MyPrefix    = "http://" MyHost ":" MyPort
  SetUpServer()
  while ("awk" != "complex") {
    # header lines are terminated this way
    RS = ORS = "\r\n"
    Status   = 200          # this means OK
    Reason   = "OK"
    Header   = TopHeader
    Document = TopDoc
    Footer   = TopFooter
    if        (GETARG["Method"] == "GET") {
        HandleGET()
    } else if (GETARG["Method"] == "HEAD") {
        # not yet implemented
```

```
    } else if (GETARG["Method"] != "") {
        print "bad method", GETARG["Method"]
    }
    Prompt = Header Document Footer
    print "HTTP/1.0", Status, Reason       |& HttpService
    print "Connection: Close"              |& HttpService
    print "Pragma: no-cache"               |& HttpService
    len = length(Prompt) + length(ORS)
    print "Content-length:", len           |& HttpService
    print ORS Prompt                       |& HttpService
    # ignore all the header lines
    while ((HttpService |& getline) > 0)
        ;
    # stop talking to this client
    close(HttpService)
    # wait for new client request
    HttpService |& getline
    # do some logging
    print systime(), strftime(), $0
    # read request parameters
    CGI_setup($1, $2, $3)
  }
}
```

This web server presents menu choices in the form of HTML links. Therefore, it has to tell the browser the name of the host it is residing on. When starting the server, the user may supply the name of the host from the command line with '`gawk -v MyHost="Rumpelstilzchen"`'. If the user does not do this, the server looks up the name of the host it is running on for later use as a web address in HTML documents. The same applies to the port number. These values are inserted later into the HTML content of the web pages to refer to the home system.

Each server that is built around this core has to initialize some application-dependent variables (such as the default home page) in a procedure `SetUpServer`, which is called immediately before entering the infinite loop of the server. For now, we will write an instance that initiates a trivial interaction. With this home page, the client user can click on two possible choices, and receive the current date either in human-readable format or in seconds since 1970:

```
function SetUpServer() {
  TopHeader = "<HTML><HEAD>"
  TopHeader = TopHeader \
     "<title>My name is GAWK, GNU AWK</title></HEAD>"
  TopDoc    = "<BODY><h2>\
    Do you prefer your date <A HREF=" MyPrefix \
    "/human>human</A> or \
```

```
    <A HREF=" MyPrefix "/POSIX>POSIXed</A>?</h2>" ORS ORS
  TopFooter = "</BODY></HTML>"
}
```

On the first run through the main loop, the default line terminators are set and the default home page is copied to the actual home page. Since this is the first run, `GETARG["Method"]` is not initialized yet, hence the case selection over the method does nothing. Now that the home page is initialized, the server can start communicating to a client browser.

It does so by printing the HTTP header into the network connection ('`print ... |& HttpService`'). This command blocks execution of the server script until a client connects. If this server script is compared with the primitive one we wrote before, you will notice two additional lines in the header. The first instructs the browser to close the connection after each request. The second tells the browser that it should never try to *remember* earlier requests that had identical web addresses (no caching). Otherwise, it could happen that the browser retrieves the time of day in the previous example just once, and later it takes the web page from the cache, always displaying the same time of day although time advances each second.

Having supplied the initial home page to the browser with a valid document stored in the parameter `Prompt`, it closes the connection and waits for the next request. When the request comes, a log line is printed that allows us to see which request the server receives. The final step in the loop is to call the function `CGI_setup`, which reads all the lines of the request (coming from the browser), processes them, and stores the transmitted parameters in the array `PARAM`. The complete text of these application-independent functions can be found in Section 2.9.1 [A Simple CGI Library], page 24. For now, we use a simplified version of `CGI_setup`:

```
function CGI_setup(   method, uri, version, i) {
  delete GETARG;         delete MENU;        delete PARAM
  GETARG["Method"] = $1
  GETARG["URI"] = $2
  GETARG["Version"] = $3
  i = index($2, "?")
  # is there a "?" indicating a CGI request?
  if (i > 0) {
    split(substr($2, 1, i-1), MENU, "[/:]")
    split(substr($2, i+1), PARAM, "&")
    for (i in PARAM) {
      j = index(PARAM[i], "=")
      GETARG[substr(PARAM[i], 1, j-1)] = \
                                   substr(PARAM[i], j+1)
    }
  } else {    # there is no "?", no need for splitting PARAMs
    split($2, MENU, "[/:]")
  }
```

```
}
```

At first, the function clears all variables used for global storage of request parameters. The rest of the function serves the purpose of filling the global parameters with the extracted new values. To accomplish this, the name of the requested resource is split into parts and stored for later evaluation. If the request contains a '`?`', then the request has CGI variables seamlessly appended to the web address. Everything in front of the '`?`' is split up into menu items, and everything behind the '`?`' is a list of '*`variable=value`*' pairs (separated by '`&`') that also need splitting. This way, CGI variables are isolated and stored. This procedure lacks recognition of special characters that are transmitted in coded form[4]. Here, any optional request header and body parts are ignored. We do not need header parameters and the request body. However, when refining our approach or working with the `POST` and `PUT` methods, reading the header and body becomes inevitable. Header parameters should then be stored in a global array as well as the body.

On each subsequent run through the main loop, one request from a browser is received, evaluated, and answered according to the user's choice. This can be done by letting the value of the HTTP method guide the main loop into execution of the procedure `HandleGET`, which evaluates the user's choice. In this case, we have only one hierarchical level of menus, but in the general case, menus are nested. The menu choices at each level are separated by '`/`', just as in file names. Notice how simple it is to construct menus of arbitrary depth:

```
function HandleGET() {
  if (       MENU[2] == "human") {
    Footer = strftime() TopFooter
  } else if (MENU[2] == "POSIX") {
    Footer = systime()  TopFooter
  }
}
```

The disadvantage of this approach is that our server is slow and can handle only one request at a time. Its main advantage, however, is that the server consists of just one `gawk` program. No need for installing an `httpd`, and no need for static separate HTML files, CGI scripts, or `root` privileges. This is rapid prototyping. This program can be started on the same host that runs your browser. Then let your browser point to `http://localhost:8080`.

It is also possible to include images into the HTML pages. Most browsers support the not very well-known '`.xbm`' format, which may contain only monochrome pictures but is an ASCII format. Binary images are possible but not so easy to handle. Another way of including images is to generate them with a tool such as GNUPlot, by calling the tool with the `system` function or through a pipe.

---

[4] As defined in RFC 2068.

2.9.1 A Simple CGI Library

HTTP is like being married: you have to be able to handle whatever you're given, while being very careful what you send back.
Phil Smith III,
`http://www.netfunny.com/rhf/jokes/99/Mar/http.html`

In Section 2.9 [A Web Service with Interaction], page 20, we saw the function `CGI_setup` as part of the web server "core logic" framework. The code presented there handles almost everything necessary for CGI requests. One thing it doesn't do is handle encoded characters in the requests. For example, an '`&`' is encoded as a percent sign followed by the hexadecimal value: '`%26`'. These encoded values should be decoded. Following is a simple library to perform these tasks. This code is used for all web server examples used throughout the rest of this book. If you want to use it for your own web server, store the source code into a file named '`inetlib.awk`'. Then you can include these functions into your code by placing the following statement into your program (on the first line of your script):

```
@include inetlib.awk
```

But beware, this mechanism is only possible if you invoke your web server script with `igawk` instead of the usual `awk` or `gawk`. Here is the code:

```
# CGI Library and core of a web server

# Global arrays
#   GETARG --- arguments to CGI GET command
#   MENU   --- menu items (path names)
#   PARAM  --- parameters of form x=y

# Optional variable MyHost contains host address
# Optional variable MyPort contains port number
# Needs TopHeader, TopDoc, TopFooter
# Sets MyPrefix, HttpService, Status, Reason

BEGIN {
  if (MyHost == "") {
     "uname -n" | getline MyHost
     close("uname -n")
  }
  if (MyPort ==  0) MyPort = 8080
  HttpService = "/inet/tcp/" MyPort "/0/0"
  MyPrefix    = "http://" MyHost ":" MyPort
  SetUpServer()
  while ("awk" != "complex") {
    # header lines are terminated this way
    RS = ORS    = "\r\n"
    Status      = 200             # this means OK
```

```
    Reason      = "OK"
    Header      = TopHeader
    Document    = TopDoc
    Footer      = TopFooter
    if        (GETARG["Method"] == "GET") {
        HandleGET()
    } else if (GETARG["Method"] == "HEAD") {
        # not yet implemented
    } else if (GETARG["Method"] != "") {
        print "bad method", GETARG["Method"]
    }
    Prompt = Header Document Footer
    print "HTTP/1.0", Status, Reason     |& HttpService
    print "Connection: Close"            |& HttpService
    print "Pragma: no-cache"             |& HttpService
    len = length(Prompt) + length(ORS)
    print "Content-length:", len         |& HttpService
    print ORS Prompt                     |& HttpService
    # ignore all the header lines
    while ((HttpService |& getline) > 0)
        continue
    # stop talking to this client
    close(HttpService)
    # wait for new client request
    HttpService |& getline
    # do some logging
    print systime(), strftime(), $0
    CGI_setup($1, $2, $3)
  }
}

function CGI_setup(   method, uri, version, i)
{
    delete GETARG
    delete MENU
    delete PARAM
    GETARG["Method"] = method
    GETARG["URI"] = uri
    GETARG["Version"] = version

    i = index(uri, "?")
    if (i > 0) {  # is there a "?" indicating a CGI request?
        split(substr(uri, 1, i-1), MENU, "[/:]")
        split(substr(uri, i+1), PARAM, "&")
        for (i in PARAM) {
            PARAM[i] = _CGI_decode(PARAM[i])
```

```
            j = index(PARAM[i], "=")
            GETARG[substr(PARAM[i], 1, j-1)] = \
                                 substr(PARAM[i], j+1)
        }
    } else { # there is no "?", no need for splitting PARAMs
        split(uri, MENU, "[/:]")
    }
    for (i in MENU)     # decode characters in path
        if (i > 4)      # but not those in host name
            MENU[i] = _CGI_decode(MENU[i])
}
```

This isolates details in a single function, `CGI_setup`. Decoding of encoded characters is pushed off to a helper function, `_CGI_decode`. The use of the leading underscore ('`_`') in the function name is intended to indicate that it is an "internal" function, although there is nothing to enforce this:

```
function _CGI_decode(str,   hexdigs, i, pre, code1, code2,
                            val, result)
{
   hexdigs = "123456789abcdef"

   i = index(str, "%")
   if (i == 0) # no work to do
      return str

   do {
      pre = substr(str, 1, i-1)   # part before %xx
      code1 = substr(str, i+1, 1) # first hex digit
      code2 = substr(str, i+2, 1) # second hex digit
      str = substr(str, i+3)      # rest of string

      code1 = tolower(code1)
      code2 = tolower(code2)
      val = index(hexdigs, code1) * 16 \
            + index(hexdigs, code2)

      result = result pre sprintf("%c", val)
      i = index(str, "%")
   } while (i != 0)
   if (length(str) > 0)
      result = result str
   return result
}
```

This works by splitting the string apart around an encoded character. The two digits are converted to lowercase characters and looked up in a string of hex digits. Note that `0` is not in the string on purpose; `index` returns zero when it's not found, automatically giving the correct value! Once the hexadecimal value is converted from characters in a string into a numerical value, `sprintf` converts the value back into a real character. The following is a simple test harness for the above functions:

```
BEGIN {
  CGI_setup("GET",
  "http://www.gnu.org/cgi-bin/foo?p1=stuff&p2=stuff%26junk" \
       "&percent=a %25 sign",
  "1.0")
  for (i in MENU)
      printf "MENU[\"%s\"] = %s\n", i, MENU[i]
  for (i in PARAM)
      printf "PARAM[\"%s\"] = %s\n", i, PARAM[i]
  for (i in GETARG)
      printf "GETARG[\"%s\"] = %s\n", i, GETARG[i]
}
```

And this is the result when we run it:

```
$ gawk -f testserv.awk
 ⊣ MENU["4"] = www.gnu.org
 ⊣ MENU["5"] = cgi-bin
 ⊣ MENU["6"] = foo
 ⊣ MENU["1"] = http
 ⊣ MENU["2"] =
 ⊣ MENU["3"] =
 ⊣ PARAM["1"] = p1=stuff
 ⊣ PARAM["2"] = p2=stuff&junk
 ⊣ PARAM["3"] = percent=a % sign
 ⊣ GETARG["p1"] = stuff
 ⊣ GETARG["percent"] = a % sign
 ⊣ GETARG["p2"] = stuff&junk
 ⊣ GETARG["Method"] = GET
 ⊣ GETARG["Version"] = 1.0
 ⊣ GETARG["URI"] = http://www.gnu.org/cgi-bin/foo?p1=stuff&
p2=stuff%26junk&percent=a %25 sign
```

2.10 A Simple Web Server

In the preceding section, we built the core logic for event-driven GUIs. In this section, we finally extend the core to a real application. No one would actually write a commercial web server in `gawk`, but it is instructive to see that it is feasible in principle.

The application is ELIZA, the famous program by Joseph Weizenbaum that mimics the behavior of a professional psychotherapist when talking to you. Weizenbaum would certainly object to this description, but this is part of the legend around ELIZA. Take the site-independent core logic and append the following code:

```
function SetUpServer() {
  SetUpEliza()
  TopHeader = \
    "<HTML><title>An HTTP-based System with GAWK</title>\
    <HEAD><META HTTP-EQUIV=\"Content-Type\"\
    CONTENT=\"text/html; charset=iso-8859-1\"></HEAD>\
    <BODY BGCOLOR=\"#ffffff\" TEXT=\"#000000\"\
    LINK=\"#0000ff\" VLINK=\"#0000ff\"\
    ALINK=\"#0000ff\"> <A NAME=\"top\">"
  TopDoc    = "\
   <h2>Please choose one of the following actions:</h2>\
   <UL>\
   <LI>\
   <A HREF=" MyPrefix "/AboutServer>About this server</A>\
   </LI><LI>\
   <A HREF=" MyPrefix "/AboutELIZA>About Eliza</A></LI>\
   <LI>\
   <A HREF=" MyPrefix \
      "/StartELIZA>Start talking to Eliza</A></LI></UL>"
  TopFooter = "</BODY></HTML>"
}
```

SetUpServer is similar to the previous example, except for calling another function, SetUpEliza. This approach can be used to implement other kinds of servers. The only changes needed to do so are hidden in the functions SetUpServer and HandleGET. Perhaps it might be necessary to implement other HTTP methods. The igawk program that comes with gawk may be useful for this process.

When extending this example to a complete application, the first thing to do is to implement the function SetUpServer to initialize the HTML pages and some variables. These initializations determine the way your HTML pages look (colors, titles, menu items, etc.).

The function HandleGET is a nested case selection that decides which page the user wants to see next. Each nesting level refers to a menu level of the GUI. Each case implements a certain action of the menu. On the deepest level of case selection, the handler essentially knows what the user wants and stores the answer into the variable that holds the HTML page contents:

```
function HandleGET() {
  # A real HTTP server would treat some parts of the URI as a file name.
  # We take parts of the URI as menu choices and go on accordingly.
  if(MENU[2] == "AboutServer") {
    Document    = "This is not a CGI script.\
      This is an httpd, an HTML file, and a CGI script all \
      in one GAWK script. It needs no separate www-server, \
```

```
      no installation, and no root privileges.\
      <p>To run it, do this:</p><ul>\
      <li> start this script with \"gawk -f httpserver.awk\",</li>\
      <li> and on the same host let your www browser open location\
           \"http://localhost:8080\"</li>\
      </ul>\<p>\ Details of HTTP come from:</p><ul>\
            <li>Hethmon:  Illustrated Guide to HTTP</p>\
            <li>RFC 2068</li></ul><p>JK 14.9.1997</p>"
  } else if (MENU[2] == "AboutELIZA") {
    Document    = "This is an implementation of the famous ELIZA\
        program by Joseph Weizenbaum. It is written in GAWK and\
/bin/sh: expad: command not found
  } else if (MENU[2] == "StartELIZA") {
    gsub(/\+/, " ", GETARG["YouSay"])
    # Here we also have to substitute coded special characters
    Document    = "<form method=GET>" \
      "<h3>" ElizaSays(GETARG["YouSay"]) "</h3>\
      <p><input type=text name=YouSay value=\"\" size=60>\
      <br><input type=submit value=\"Tell her about it\"></p></form>"
  }
}
```

Now we are down to the heart of ELIZA, so you can see how it works. Initially the user does not say anything; then ELIZA resets its money counter and asks the user to tell what comes to mind open heartedly. The subsequent answers are converted to uppercase characters and stored for later comparison. ELIZA presents the bill when being confronted with a sentence that contains the phrase "shut up." Otherwise, it looks for keywords in the sentence, conjugates the rest of the sentence, remembers the keyword for later use, and finally selects an answer from the set of possible answers:

```
function ElizaSays(YouSay) {
  if (YouSay == "") {
    cost = 0
    answer = "HI, IM ELIZA, TELL ME YOUR PROBLEM"
  } else {
    q = toupper(YouSay)
    gsub("'", "", q)
    if(q == qold) {
      answer = "PLEASE DONT REPEAT YOURSELF !"
    } else {
      if (index(q, "SHUT UP") > 0) {
        answer = "WELL, PLEASE PAY YOUR BILL. ITS EXACTLY ... $"\
                 int(100*rand()+30+cost/100)
      } else {
        qold = q
        w = "-"                 # no keyword recognized yet
        for (i in k) {          # search for keywords
          if (index(q, i) > 0) {
            w = i
            break
          }
        }
```

```
        if (w == "-") {         # no keyword, take old subject
          w    = wold
          subj = subjold
        } else {                # find subject
          subj = substr(q, index(q, w) + length(w)+1)
          wold = w
          subjold = subj        #  remember keyword and subject
        }
        for (i in conj)
           gsub(i, conj[i], q)   # conjugation
        # from all answers to this keyword, select one randomly
        answer = r[indices[int(split(k[w], indices) * rand()) + 1]]
        # insert subject into answer
        gsub("_", subj, answer)
      }
    }
  }
  cost += length(answer) # for later payment : 1 cent per character
  return answer
}
```

In the long but simple function SetUpEliza, you can see tables for conjugation, keywords, and answers.[5] The associative array k contains indices into the array of answers r. To choose an answer, ELIZA just picks an index randomly:

```
function SetUpEliza() {
  srand()
  wold = "-"
  subjold = " "

  # table for conjugation
  conj[" ARE "     ] = " AM "
  conj["WERE "     ] = "WAS "
  conj[" YOU "     ] = " I "
  conj["YOUR "     ] = "MY "
  conj[" IVE "     ] =\
  conj[" I HAVE "  ] = " YOU HAVE "
  conj[" YOUVE "   ] =\
  conj[" YOU HAVE "] = " I HAVE "
  conj[" IM "      ] =\
  conj[" I AM "    ] = " YOU ARE "
  conj[" YOURE "   ] =\
  conj[" YOU ARE " ] = " I AM "

  # table of all answers
  r[1]   = "DONT YOU BELIEVE THAT I CAN  _"
```

[5] The version shown here is abbreviated. The full version comes with the gawk distribution.

```
    r[2]    = "PERHAPS YOU WOULD LIKE TO BE ABLE TO _ ?"
    ...
    # table for looking up answers that
    # fit to a certain keyword
    k["CAN YOU"]      = "1 2 3"
    k["CAN I"]        = "4 5"
    k["YOU ARE"]      =\
    k["YOURE"]        = "6 7 8 9"
    ...
  }
```

Some interesting remarks and details (including the original source code of ELIZA) are found on Mark Humphrys' home page. Yahoo! also has a page with a collection of ELIZA-like programs. Many of them are written in Java, some of them disclosing the Java source code, and a few even explain how to modify the Java source code.

2.11 Network Programming Caveats

By now it should be clear that debugging a networked application is more complicated than debugging a single-process single-hosted application. The behavior of a networked application sometimes looks noncausal because it is not reproducible in a strong sense. Whether a network application works or not sometimes depends on the following:

- How crowded the underlying network is
- If the party at the other end is running or not
- The state of the party at the other end

The most difficult problems for a beginner arise from the hidden states of the underlying network. After closing a TCP connection, it's often necessary to wait a short while before reopening the connection. Even more difficult is the establishment of a connection that previously ended with a "broken pipe." Those connections have to "time out" for a minute or so before they can reopen. Check this with the command '`netstat -a`', which provides a list of still "active" connections.

2.12 Where To Go From Here

Now, you have learned enough to build your own application. You could, for example, take part in the Loebner Contest to win $100,000. The Loebner Prize is the first formal instantiation of a Turing Test. Hugh Loebner agreed with The Cambridge Center for Behavioral Studies to underwrite a contest designed to implement the Turing Test. Dr. Loebner pledged a Grand Prize of $100,000 for the first computer whose responses were indistinguishable from a human's. Each year an annual prize of $2000 and a bronze medal is awarded to the *most* human computer. The winner of the annual contest is the best entry relative to other entries that year, irrespective of how good it is in an absolute sense. Here is an example of a conversation with the winning program of 1997:

```
(c)1997 Cambridge Center For Behavioral Studies all rights reserved [CONVERSE]
[Intelligent Research Ltd.] Start at: [1997/04/29 08:59:40]
PROGRAM[10:39:20]Hi, my name is Catherine, what's yours?
JUDGE02[10:39:36]My name is Nathan.
PROGRAM[10:39:37]Well, it's nice to meet you, Nathan.
PROGRAM[10:39:45]Did you see that story on CNN last night about the lesbian
PROGRAM[10:40:03]couple who came out at a White House party on Sunday?
JUDGE02[10:40:45]No.  I'm sorry I missed that.  How did the Prez take it?
PROGRAM[10:40:47]Ellen Degeneres was one of them - she was kissing her lover
PROGRAM[10:41:08]in front of Bill Clinton. Do you think he was very
PROGRAM[10:41:21]embarrassed?
JUDGE02[10:41:37]You tell me.  I didn't see it.
PROGRAM[10:41:38]Are you still a Friend of Bill Clinton?
JUDGE02[10:42:22]Yes.  I would like to talk about something else.  Do you know anything about sculpture.
PROGRAM[10:42:24]Well, I suppose he's doing his best like most of us, but he
PROGRAM[10:42:35]does seem to attract an UNUSUAL amount of bad publicity,
PROGRAM[10:42:51]wouldnt you say--can it all be bad luck?
JUDGE02[10:43:20]I asked you if you knew anything about sculpture.
PROGRAM[10:43:21]OK, but what happens now--does it all drift on, with
PROGRAM[10:43:36]everyone waiting to see how much his friends have said about
PROGRAM[10:43:47]him under threat of lots and lots of jail-or will something
PROGRAM[10:44:01]dramatic actually happen?
JUDGE02[10:44:19]Do you know what sculpture is?
PROGRAM[10:44:20]Well, youre more a fan of his than I am I guess---though I
PROGRAM[10:44:34]cant SWEAR what Id have done in that locked hotel room, I
PROGRAM[10:44:51]must say.
JUDGE02[10:45:26]Have you ever been to the Salmagundi Club?
PROGRAM[10:45:29]Never. to go back to the original discussion Time to get
PROGRAM[10:45:43]off this one I think and onto something more down to earth!
JUDGE02[10:46:20]What is more down to earth than art?
PROGRAM[10:46:21]I don't know what -- to be honest. -
PROGRAM[10:46:30]I suppose it depends on where you're coming from, but as the
PROGRAM[10:46:46]song goes in My Fair Lady, "Why can't a woman be more like a
PROGRAM[10:47:03]man? "
JUDGE02[10:47:52]I liked that song but I prefer "Why can't children be more like we are"
JUDGE02[10:48:10]from another play the name of which I cannot remember.
PROGRAM[10:48:12]What, if anything, do you enjoy about being a man?
JUDGE02[10:49:01]I would enjoy all aspects of life no matter what my gender.
PROGRAM[10:49:02]Do you think of yourself as being attractive?
```

This program insists on always speaking about the same story around Bill Clinton. You see, even a program with a rather narrow mind can behave so much like a human being that it can win this prize. It is quite common to let these programs talk to each other via network connections. But during the competition itself, the program and its computer have to be present at the place the competition is held. We all would love to see a `gawk` program win in such an event. Maybe it is up to you to accomplish this?

Some other ideas for useful networked applications:

- Read the file '`doc/awkforai.txt`' in the `gawk` distribution. It was written by Ronald P. Loui (Associate Professor of Computer Science, at Washington University in St. Louis, `loui@ai.wustl.edu`) and summarizes why he teaches `gawk` to students of Artificial Intelligence. Here are some passages from the text:

  > The GAWK manual can be consumed in a single lab session and the language can be mastered by the next morning by the average student. GAWK's automatic initialization, implicit coercion, I/O support and lack of pointers forgive many of the mistakes that young programmers are likely to make. Those who have seen C but not mastered it are happy to see that GAWK retains some of the same sensibilities while adding what must be regarded as spoonsful of syntactic sugar.
  > ...
  > There are further simple answers. Probably the best is the fact that increasingly, undergraduate AI programming is involving the Web. Oren Etzioni (University of Washington, Seattle) has for a while been arguing that the "softbot" is replacing the mechanical engineers' robot as the most glamorous AI testbed. If the artifact whose behavior needs to be controlled in an intelligent way is the software agent, then a language that is well-suited to controlling the software environment is the appropriate language. That would imply a scripting language. If the robot is KAREL, then the right language is "turn left; turn right." If the robot is Netscape, then the right language is something that can generate '`netscape -remote 'openURL(http://cs.wustl.edu/~loui)'`' with elan.
  > ...
  > AI programming requires high-level thinking. There have always been a few gifted programmers who can write high-level programs in assembly language. Most however need the ambient abstraction to have a higher floor.
  > ...
  > Second, inference is merely the expansion of notation. No matter whether the logic that underlies an AI program is fuzzy, probabilistic, deontic, defeasible, or deductive, the logic merely defines how strings can be transformed into other strings. A language that provides the best support for string processing in the end provides the best support for logic, for the exploration of various logics, and for most forms of symbolic processing that AI might choose to call "reasoning" instead of "logic." The implication is that PROLOG, which saves the AI programmer from having to write a unifier, saves perhaps two

> dozen lines of GAWK code at the expense of strongly biasing the logic and representational expressiveness of any approach.

Now that `gawk` itself can connect to the Internet, it should be obvious that it is suitable for writing intelligent web agents.

- `awk` is strong at pattern recognition and string processing. So, it is well suited to the classic problem of language translation. A first try could be a program that knows the 100 most frequent English words and their counterparts in German or French. The service could be implemented by regularly reading email with the program above, replacing each word by its translation and sending the translation back via SMTP. Users would send English email to their translation service and get back a translated email message in return. As soon as this works, more effort can be spent on a real translation program.
- Another dialogue-oriented application (on the verge of ridicule) is the email "support service." Troubled customers write an email to an automatic `gawk` service that reads the email. It looks for keywords in the mail and assembles a reply email accordingly. By carefully investigating the email header, and repeating these keywords through the reply email, it is rather simple to give the customer a feeling that someone cares. Ideally, such a service would search a database of previous cases for solutions. If none exists, the database could, for example, consist of all the newsgroups, mailing lists and FAQs on the Internet.

3 Some Applications and Techniques

In this chapter, we look at a number of self-contained scripts, with an emphasis on concise networking. Along the way, we work towards creating building blocks that encapsulate often needed functions of the networking world, show new techniques that broaden the scope of problems that can be solved with `gawk`, and explore leading edge technology that may shape the future of networking.

We often refer to the site-independent core of the server that we built in Section 2.10 [A Simple Web Server], page 27. When building new and non-trivial servers, we always copy this building block and append new instances of the two functions `SetUpServer` and `HandleGET`.

This makes a lot of sense, since this scheme of event-driven execution provides `gawk` with an interface to the most widely accepted standard for GUIs: the web browser. Now, `gawk` can rival even Tcl/Tk.

Tcl and `gawk` have much in common. Both are simple scripting languages that allow us to quickly solve problems with short programs. But Tcl has Tk on top of it, and `gawk` had nothing comparable up to now. While Tcl needs a large and ever-changing library (Tk, which was bound to the X Window System until recently), `gawk` needs just the networking interface and some kind of browser on the client's side. Besides better portability, the most important advantage of this approach (embracing well-established standards such HTTP and HTML) is that *we do not need to change the language*. We let others do the work of fighting over protocols and standards. We can use HTML, JavaScript, VRML, or whatever else comes along to do our work.

3.1 PANIC: An Emergency Web Server

At first glance, the `"Hello, world"` example in Section 2.8 [A Primitive Web Service], page 19, seems useless. By adding just a few lines, we can turn it into something useful.

The PANIC program tells everyone who connects that the local site is not working. When a web server breaks down, it makes a difference if customers get a strange "network unreachable" message, or a short message telling them that the server has a problem. In such an emergency, the hard disk and everything on it (including the regular web service) may be unavailable. Rebooting the web server off a diskette makes sense in this setting.

To use the PANIC program as an emergency web server, all you need are the `gawk` executable and the program below on a diskette. By default, it connects to port 8080. A different value may be supplied on the command line:

```
BEGIN {
  RS = ORS = "\r\n"
  if (MyPort ==  0) MyPort = 8080
```

```
  HttpService = "/inet/tcp/" MyPort "/0/0"
  Hello = "<HTML><HEAD><TITLE>Out Of Service</TITLE>" \
     "</HEAD><BODY><H1>" \
     "This site is temporarily out of service." \
     "</H1></BODY></HTML>"
  Len = length(Hello) + length(ORS)
  while ("awk" != "complex") {
    print "HTTP/1.0 200 OK"          |& HttpService
    print "Content-Length: " Len ORS |& HttpService
    print Hello                      |& HttpService
    while ((HttpService |& getline) > 0)
       continue;
    close(HttpService)
  }
}
```

3.2 GETURL: Retrieving Web Pages

GETURL is a versatile building block for shell scripts that need to retrieve files from the Internet. It takes a web address as a command-line parameter and tries to retrieve the contents of this address. The contents are printed to standard output, while the header is printed to '`/dev/stderr`'. A surrounding shell script could analyze the contents and extract the text or the links. An ASCII browser could be written around GETURL. But more interestingly, web robots are straightforward to write on top of GETURL. On the Internet, you can find several programs of the same name that do the same job. They are usually much more complex internally and at least 10 times longer.

At first, GETURL checks if it was called with exactly one web address. Then, it checks if the user chose to use a special proxy server whose name is handed over in a variable. By default, it is assumed that the local machine serves as proxy. GETURL uses the `GET` method by default to access the web page. By handing over the name of a different method (such as `HEAD`), it is possible to choose a different behavior. With the `HEAD` method, the user does not receive the body of the page content, but does receive the header:

```
BEGIN {
  if (ARGC != 2) {
    print "GETURL - retrieve Web page via HTTP 1.0"
    print "IN:\n    the URL as a command-line parameter"
    print "PARAM(S):\n    -v Proxy=MyProxy"
    print "OUT:\n    the page content on stdout"
    print "    the page header on stderr"
    print "JK 16.05.1997"
    print "ADR 13.08.2000"
    exit
```

```
    }
    URL = ARGV[1]; ARGV[1] = ""
    if (Proxy     == "")  Proxy     = "127.0.0.1"
    if (ProxyPort ==  0)  ProxyPort = 80
    if (Method    == "")  Method    = "GET"
    HttpService = "/inet/tcp/0/" Proxy "/" ProxyPort
    ORS = RS = "\r\n\r\n"
    print Method " " URL " HTTP/1.0" |& HttpService
    HttpService                      |& getline Header
    print Header > "/dev/stderr"
    while ((HttpService |& getline) > 0)
      printf "%s", $0
    close(HttpService)
  }
```

This program can be changed as needed, but be careful with the last lines. Make sure transmission of binary data is not corrupted by additional line breaks. Even as it is now, the byte sequence `"\r\n\r\n"` would disappear if it were contained in binary data. Don't get caught in a trap when trying a quick fix on this one.

3.3 REMCONF: Remote Configuration of Embedded Systems

Today, you often find powerful processors in embedded systems. Dedicated network routers and controllers for all kinds of machinery are examples of embedded systems. Processors like the Intel 80x86 or the AMD Elan are able to run multitasking operating systems, such as XINU or GNU/Linux in embedded PCs. These systems are small and usually do not have a keyboard or a display. Therefore it is difficult to set up their configuration. There are several widespread ways to set them up:

- DIP switches
- Read Only Memories such as EPROMs
- Serial lines or some kind of keyboard
- Network connections via `telnet` or SNMP
- HTTP connections with HTML GUIs

In this section, we look at a solution that uses HTTP connections to control variables of an embedded system that are stored in a file. Since embedded systems have tight limits on resources like memory, it is difficult to employ advanced techniques such as SNMP and HTTP servers. `gawk` fits in quite nicely with its single executable which needs just a short script to start working. The following program stores the variables in a file, and a concurrent process in the embedded system may read the file. The program uses the site-independent part of the simple web server that we developed in Section 2.9 [A Web Service with Interaction], page 20. As mentioned there, all we have to do is to write two new procedures `SetUpServer` and `HandleGET`:

```
function SetUpServer() {
  TopHeader = "<HTML><title>Remote Configuration</title>"
  TopDoc = "<BODY>\
    <h2>Please choose one of the following actions:</h2>\
    <UL>\
      <LI><A HREF=" MyPrefix "/AboutServer>About this server</A></LI>\
      <LI><A HREF=" MyPrefix "/ReadConfig>Read Configuration</A></LI>\
      <LI><A HREF=" MyPrefix "/CheckConfig>Check Configuration</A></LI>\
      <LI><A HREF=" MyPrefix "/ChangeConfig>Change Configuration</A></LI>\
      <LI><A HREF=" MyPrefix "/SaveConfig>Save Configuration</A></LI>\
    </UL>"
  TopFooter  = "</BODY></HTML>"
  if (ConfigFile == "") ConfigFile = "config.asc"
}
```

The function `SetUpServer` initializes the top level HTML texts as usual. It also initializes the name of the file that contains the configuration parameters and their values. In case the user supplies a name from the command line, that name is used. The file is expected to contain one parameter per line, with the name of the parameter in column one and the value in column two.

The function `HandleGET` reflects the structure of the menu tree as usual. The first menu choice tells the user what this is all about. The second choice reads the configuration file line by line and stores the parameters and their values. Notice that the record separator for this file is `"\n"`, in contrast to the record separator for HTTP. The third menu choice builds an HTML table to show the contents of the configuration file just read. The fourth choice does the real work of changing parameters, and the last one just saves the configuration into a file:

```
function HandleGET() {
  if(MENU[2] == "AboutServer") {
    Document  = "This is a GUI for remote configuration of an\
      embedded system. It is is implemented as one GAWK script."
  } else if (MENU[2] == "ReadConfig") {
    RS = "\n"
    while ((getline < ConfigFile) > 0)
       config[$1] = $2;
    close(ConfigFile)
    RS = "\r\n"
    Document = "Configuration has been read."
  } else if (MENU[2] == "CheckConfig") {
    Document = "<TABLE BORDER=1 CELLPADDING=5>"
    for (i in config)
      Document = Document "<TR><TD>" i "</TD>" \
        "<TD>" config[i] "</TD></TR>"
    Document = Document "</TABLE>"
  } else if (MENU[2] == "ChangeConfig") {
```

```
    if ("Param" in GETARG) {            # any parameter to set?
      if (GETARG["Param"] in config) {  # is  parameter valid?
        config[GETARG["Param"]] = GETARG["Value"]
        Document = (GETARG["Param"] " = " GETARG["Value"] ".")
      } else {
        Document = "Parameter <b>" GETARG["Param"] "</b> is invalid."
      }
    } else {
      Document = "<FORM method=GET><h4>Change one parameter</h4>\
        <TABLE BORDER CELLPADDING=5>\
        <TR><TD>Parameter</TD><TD>Value</TD></TR>\
        <TR><TD><input type=text name=Param value=\"\" size=20></TD>\
            <TD><input type=text name=Value value=\"\" size=40></TD>\
        </TR></TABLE><input type=submit value=\"Set\"></FORM>"
    }
  } else if (MENU[2] == "SaveConfig") {
    for (i in config)
      printf("%s %s\n", i, config[i]) > ConfigFile
    close(ConfigFile)
    Document = "Configuration has been saved."
  }
}
```

We could also view the configuration file as a database. From this point of view, the previous program acts like a primitive database server. Real SQL database systems also make a service available by providing a TCP port that clients can connect to. But the application level protocols they use are usually proprietary and also change from time to time. This is also true for the protocol that MiniSQL uses.

3.4 URLCHK: Look for Changed Web Pages

Most people who make heavy use of Internet resources have a large bookmark file with pointers to interesting web sites. It is impossible to regularly check by hand if any of these sites have changed. A program is needed to automatically look at the headers of web pages and tell which ones have changed. URLCHK does the comparison after using GETURL with the `HEAD` method to retrieve the header.

Like GETURL, this program first checks that it is called with exactly one command-line parameter. URLCHK also takes the same command-line variables `Proxy` and `ProxyPort` as GETURL, because these variables are handed over to GETURL for each URL that gets checked. The one and only parameter is the name of a file that contains one line for each URL. In the first column, we find the URL, and the second and third columns hold the length of the URL's body when checked for the two last times. Now, we follow this plan:

1. Read the URLs from the file and remember their most recent lengths
2. Delete the contents of the file

3. For each URL, check its new length and write it into the file
4. If the most recent and the new length differ, tell the user

It may seem a bit peculiar to read the URLs from a file together with their two most recent lengths, but this approach has several advantages. You can call the program again and again with the same file. After running the program, you can regenerate the changed URLs by extracting those lines that differ in their second and third columns:

```
BEGIN {
  if (ARGC != 2) {
    print "URLCHK - check if URLs have changed"
    print "IN:\n    the file with URLs as a command-line parameter"
    print "    file contains URL, old length, new length"
    print "PARAMS:\n    -v Proxy=MyProxy -v ProxyPort=8080"
    print "OUT:\n    same as file with URLs"
    print "JK 02.03.1998"
    exit
  }
  URLfile = ARGV[1]; ARGV[1] = ""
  if (Proxy     != "") Proxy     = " -v Proxy="     Proxy
  if (ProxyPort != "") ProxyPort = " -v ProxyPort=" ProxyPort
  while ((getline < URLfile) > 0)
     Length[$1] = $3 + 0
  close(URLfile)      # now, URLfile is read in and can be updated
  GetHeader = "gawk " Proxy ProxyPort " -v Method=\"HEAD\" -f geturl.awk "
  for (i in Length) {
    GetThisHeader = GetHeader i " 2>&1"
    while ((GetThisHeader | getline) > 0)
      if (toupper($0) ~ /CONTENT-LENGTH/) NewLength = $2 + 0
    close(GetThisHeader)
    print i, Length[i], NewLength > URLfile
    if (Length[i] != NewLength)  # report only changed URLs
      print i, Length[i], NewLength
  }
  close(URLfile)
}
```

Another thing that may look strange is the way GETURL is called. Before calling GETURL, we have to check if the proxy variables need to be passed on. If so, we prepare strings that will become part of the command line later. In `GetHeader`, we store these strings together with the longest part of the command line. Later, in the loop over the URLs, `GetHeader` is appended with the URL and a redirection operator to form the command that reads the URL's header over the Internet. GETURL always produces the headers over '`/dev/stderr`'. That is the reason why we need the redirection operator to have the header piped in.

This program is not perfect because it assumes that changing URLs results in changed lengths, which is not necessarily true. A more advanced approach is to look at some other header line that holds time information.

But, as always when things get a bit more complicated, this is left as an exercise to the reader.

3.5 WEBGRAB: Extract Links from a Page

Sometimes it is necessary to extract links from web pages. Browsers do it, web robots do it, and sometimes even humans do it. Since we have a tool like GETURL at hand, we can solve this problem with some help from the Bourne shell:

```
BEGIN { RS = "http://[#%&\\+\\-\\./0-9\\:;\\?A-Z_a-z\\~]*" }
RT != "" {
   command = ("gawk -v Proxy=MyProxy -f geturl.awk " RT \
              " > doc" NR ".html")
   print command
}
```

Notice that the regular expression for URLs is rather crude. A precise regular expression is much more complex. But this one works rather well. One problem is that it is unable to find internal links of an HTML document. Another problem is that '`ftp`', '`telnet`', '`news`', '`mailto`', and other kinds of links are missing in the regular expression. However, it is straightforward to add them, if doing so is necessary for other tasks.

This program reads an HTML file and prints all the HTTP links that it finds. It relies on `gawk`'s ability to use regular expressions as record separators. With `RS` set to a regular expression that matches links, the second action is executed each time a non-empty link is found. We can find the matching link itself in `RT`.

The action could use the `system` function to let another GETURL retrieve the page, but here we use a different approach. This simple program prints shell commands that can be piped into `sh` for execution. This way it is possible to first extract the links, wrap shell commands around them, and pipe all the shell commands into a file. After editing the file, execution of the file retrieves exactly those files that we really need. In case we do not want to edit, we can retrieve all the pages like this:

```
gawk -f geturl.awk http://www.suse.de | gawk -f webgrab.awk | sh
```

After this, you will find the contents of all referenced documents in files named '`doc*.html`' even if they do not contain HTML code. The most annoying thing is that we always have to pass the proxy to GETURL. If you do not like to see the headers of the web pages appear on the screen, you can redirect them to '`/dev/null`'. Watching the headers appear can be quite interesting, because it reveals interesting details such as which web server the companies use. Now, it is clear how the clever marketing people use web robots to determine the market shares of Microsoft and Netscape in the web server market.

Port 80 of any web server is like a small hole in a repellent firewall. After attaching a browser to port 80, we usually catch a glimpse of the bright side of the server (its home page). With a tool like GETURL at hand, we are able to discover some of the more concealed or even "indecent" services (i.e., lacking conformity to standards of quality). It can be exciting to see the fancy CGI scripts that lie there, revealing the inner workings of the server, ready to be called:

- With a command such as:

  ```
  gawk -f geturl.awk http://any.host.on.the.net/cgi-bin/
  ```

  some servers give you a directory listing of the CGI files. Knowing the names, you can try to call some of them and watch for useful results. Sometimes there are executables in such directories (such as Perl interpreters) that you may call remotely. If there are subdirectories with configuration data of the web server, this can also be quite interesting to read.

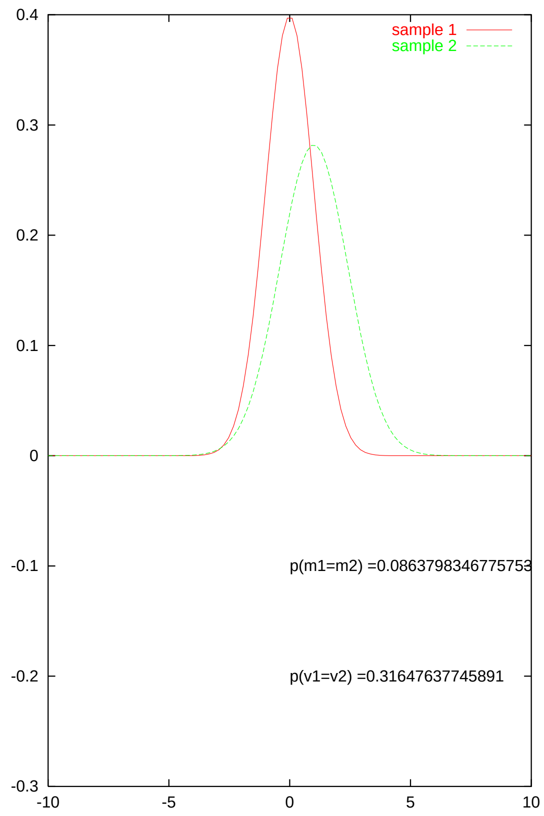
- The well-known Apache web server usually has its CGI files in the directory '`/cgi-bin`'. There you can often find the scripts '`test-cgi`' and '`printenv`'. Both tell you some things about the current connection and the installation of the web server. Just call:

  ```
  gawk -f geturl.awk http://any.host.on.the.net/cgi-bin/test-cgi
  gawk -f geturl.awk http://any.host.on.the.net/cgi-bin/printenv
  ```

- Sometimes it is even possible to retrieve system files like the web server's log file—possibly containing customer data—or even the file '`/etc/passwd`'. (We don't recommend this!)

Caution: Although this may sound funny or simply irrelevant, we are talking about severe security holes. Try to explore your own system this way and make sure that none of the above reveals too much information about your system.

3.6 STATIST: Graphing a Statistical Distribution



In the HTTP server examples we've shown thus far, we never present an image to the browser and its user. Presenting images is one task. Generating images that reflect some user input and presenting these dynamically generated images is another. In this section, we use GNUPlot for generating '`.png`', '`.ps`', or '`.gif`' files.[1]

The program we develop takes the statistical parameters of two samples and computes the t-test statistics. As a result, we get the probabilities that the means and the variances of both samples are the same. In order to let the user check plausibility, the program presents an image of the

[1] Due to licensing problems, the default installation of GNUPlot disables the generation of '`.gif`' files. If your installed version does not accept '`set term gif`', just download and install the most recent version of GNUPlot and the GD library (`http://www.boutell.com/gd/`) by Thomas Boutell. Otherwise you still have the chance to generate some ASCII-art style images with GNUPlot by using '`set term dumb`'. (We tried it and it worked.)

distributions. The statistical computation follows *Numerical Recipes in C: The Art of Scientific Computing* by William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. Since `gawk` does not have a built-in function for the computation of the beta function, we use the `ibeta` function of GNUPlot. As a side effect, we learn how to use GNUPlot as a sophisticated calculator. The comparison of means is done as in `tutest`, paragraph 14.2, page 613, and the comparison of variances is done as in `ftest`, page 611 in *Numerical Recipes*.

As usual, we take the site-independent code for servers and append our own functions `SetUpServer` and `HandleGET`:

```
function SetUpServer() {
  TopHeader = "<HTML><title>Statistics with GAWK</title>"
  TopDoc = "<BODY>\
   <h2>Please choose one of the following actions:</h2>\
   <UL>\
    <LI><A HREF=" MyPrefix "/AboutServer>About this server</A></LI>\
    <LI><A HREF=" MyPrefix "/EnterParameters>Enter Parameters</A></LI>\
   </UL>"
  TopFooter  = "</BODY></HTML>"
  GnuPlot    = "gnuplot 2>&1"
  m1=m2=0;    v1=v2=1;    n1=n2=10
}
```

Here, you see the menu structure that the user sees. Later, we will see how the program structure of the `HandleGET` function reflects the menu structure. What is missing here is the link for the image we generate. In an event-driven environment, request, generation, and delivery of images are separated.

Notice the way we initialize the `GnuPlot` command string for the pipe. By default, GNUPlot outputs the generated image via standard output, as well as the results of `print`(ed) calculations via standard error. The redirection causes standard error to be mixed into standard output, enabling us to read results of calculations with `getline`. By initializing the statistical parameters with some meaningful defaults, we make sure the user gets an image the first time he uses the program.

Following is the rather long function `HandleGET`, which implements the contents of this service by reacting to the different kinds of requests from the browser. Before you start playing with this script, make sure that your browser supports JavaScript and that it also has this option switched on. The script uses a short snippet of JavaScript code for delayed opening of a window with an image. A more detailed explanation follows:

```
function HandleGET() {
  if(MENU[2] == "AboutServer") {
    Document  = "This is a GUI for a statistical computation.\
      It compares means and variances of two distributions.\
      It is implemented as one GAWK script and uses GNUPLOT."
  } else if (MENU[2] == "EnterParameters") {
    Document = ""
```

```
if ("m1" in GETARG) {     # are there parameters to compare?
  Document = Document "<SCRIPT LANGUAGE=\"JavaScript\">\
    setTimeout(\"window.open(\\\"" MyPrefix "/Image" systime()\
     "\\\",\\\"dist\\\", \\\"status=no\\\");\", 1000); </SCRIPT>"
  m1 = GETARG["m1"]; v1 = GETARG["v1"]; n1 = GETARG["n1"]
  m2 = GETARG["m2"]; v2 = GETARG["v2"]; n2 = GETARG["n2"]
  t = (m1-m2)/sqrt(v1/n1+v2/n2)
  df = (v1/n1+v2/n2)*(v1/n1+v2/n2)/((v1/n1)*(v1/n1)/(n1-1) \
       + (v2/n2)*(v2/n2) /(n2-1))
  if (v1>v2) {
      f = v1/v2
      df1 = n1 - 1
      df2 = n2 - 1
  } else {
      f = v2/v1
      df1 = n2 - 1
      df2 = n1 - 1
  }
  print "pt=ibeta(" df/2 ",0.5," df/(df+t*t) ")"  |& GnuPlot
  print "pF=2.0*ibeta(" df2/2 "," df1/2 "," \
        df2/(df2+df1*f) ")"                      |& GnuPlot
  print "print pt, pF"                           |& GnuPlot
  RS="\n"; GnuPlot |& getline; RS="\r\n"    # $1 is pt, $2 is pF
  print "invsqrt2pi=1.0/sqrt(2.0*pi)"            |& GnuPlot
  print "nd(x)=invsqrt2pi/sd*exp(-0.5*((x-mu)/sd)**2)" |& GnuPlot
  print "set term png small color"               |& GnuPlot
  #print "set term postscript color"             |& GnuPlot
  #print "set term gif medium size 320,240"      |& GnuPlot
  print "set yrange[-0.3:]"                      |& GnuPlot
  print "set label 'p(m1=m2) =" $1 "' at 0,-0.1 left"  |& GnuPlot
  print "set label 'p(v1=v2) =" $2 "' at 0,-0.2 left"  |& GnuPlot
  print "plot mu=" m1 ",sd=" sqrt(v1) ", nd(x) title 'sample 1',\
    mu=" m2 ",sd=" sqrt(v2) ", nd(x) title 'sample 2'" |& GnuPlot
  print "quit"                                         |& GnuPlot
  GnuPlot |& getline Image
  while ((GnuPlot |& getline) > 0)
      Image = Image RS $0
  close(GnuPlot)
}
Document = Document "\
<h3>Do these samples have the same Gaussian distribution?</h3>\
<FORM METHOD=GET> <TABLE BORDER CELLPADDING=5>\
<TR>\
<TD>1. Mean    </TD>
<TD><input type=text name=m1 value=" m1 " size=8></TD>\
<TD>1. Variance</TD>
<TD><input type=text name=v1 value=" v1 " size=8></TD>\
<TD>1. Count   </TD>
<TD><input type=text name=n1 value=" n1 " size=8></TD>\
</TR><TR>\
<TD>2. Mean    </TD>
<TD><input type=text name=m2 value=" m2 " size=8></TD>\
```

```
      <TD>2. Variance</TD>
      <TD><input type=text name=v2 value=" v2 " size=8></TD>\
      <TD>2. Count   </TD>
      <TD><input type=text name=n2 value=" n2 " size=8></TD>\
      </TR>                    <input type=submit value=\"Compute\">\
      </TABLE></FORM><BR>"
    } else if (MENU[2] ~ "Image") {
      Reason = "OK" ORS "Content-type: image/png"
      #Reason = "OK" ORS "Content-type: application/x-postscript"
      #Reason = "OK" ORS "Content-type: image/gif"
      Header = Footer = ""
      Document = Image
    }
  }
```

As usual, we give a short description of the service in the first menu choice. The third menu choice shows us that generation and presentation of an image are two separate actions. While the latter takes place quite instantly in the third menu choice, the former takes place in the much longer second choice. Image data passes from the generating action to the presenting action via the variable `Image` that contains a complete '`.png`' image, which is otherwise stored in a file. If you prefer '`.ps`' or '`.gif`' images over the default '`.png`' images, you may select these options by uncommenting the appropriate lines. But remember to do so in two places: when telling GNUPlot which kind of images to generate, and when transmitting the image at the end of the program.

Looking at the end of the program, the way we pass the '`Content-type`' to the browser is a bit unusual. It is appended to the '`OK`' of the first header line to make sure the type information becomes part of the header. The other variables that get transmitted across the network are made empty, because in this case we do not have an HTML document to transmit, but rather raw image data to contain in the body.

Most of the work is done in the second menu choice. It starts with a strange JavaScript code snippet. When first implementing this server, we used a short `"<IMG SRC=" MyPrefix "/Image>"` here. But then browsers got smarter and tried to improve on speed by requesting the image and the HTML code at the same time. When doing this, the browser tries to build up a connection for the image request while the request for the HTML text is not yet completed. The browser tries to connect to the `gawk` server on port 8080 while port 8080 is still in use for transmission of the HTML text. The connection for the image cannot be built up, so the image appears as "broken" in the browser window. We solved this problem by telling the browser to open a separate window for the image, but only after a delay of 1000 milliseconds. By this time, the server should be ready for serving the next request.

But there is one more subtlety in the JavaScript code. Each time the JavaScript code opens a window for the image, the name of the image is appended with a timestamp (`systime`). Why this constant change of name for the image? Initially, we always named the image `Image`, but then the Netscape browser noticed the name had *not* changed since the previous request and displayed the previous image (caching behavior). The server core is implemented so that browsers are told *not* to cache anything. Obviously HTTP requests do not always work as expected. One way to circumvent the cache of such overly smart browsers is to change the name of the image with each request. These three lines of JavaScript caused us a lot of trouble.

The rest can be broken down into two phases. At first, we check if there are statistical parameters. When the program is first started, there usually are no parameters because it enters the page coming from the top menu. Then, we only have to present the user a form that he can use to change statistical parameters and submit them. Subsequently, the submission of the form causes the execution of the first phase because *now* there *are* parameters to handle.

Now that we have parameters, we know there will be an image available. Therefore we insert the JavaScript code here to initiate the opening of the image in a separate window. Then, we prepare some variables that will be passed to GNUPlot for calculation of the probabilities. Prior to reading the results, we must temporarily change `RS` because GNUPlot separates lines with newlines. After instructing GNUPlot to generate a '`.png`' (or '`.ps`' or '`.gif`') image, we initiate the insertion of some text, explaining the resulting probabilities. The final '`plot`' command actually generates the image data. This raw binary has to be read in carefully without adding, changing, or deleting a single byte. Hence the unusual initialization of `Image` and completion with a `while` loop.

When using this server, it soon becomes clear that it is far from being perfect. It mixes source code of six scripting languages or protocols:

- GNU `awk` implements a server for the protocol:
- HTTP which transmits:
- HTML text which contains a short piece of:
- JavaScript code opening a separate window.
- A Bourne shell script is used for piping commands into:
- GNUPlot to generate the image to be opened.

After all this work, the GNUPlot image opens in the JavaScript window where it can be viewed by the user.

It is probably better not to mix up so many different languages. The result is not very readable. Furthermore, the statistical part of the server does not take care of invalid input. Among others, using negative variances will cause invalid results.

3.7 MAZE: Walking Through a Maze In Virtual Reality

In the long run, every program becomes rococo, and then rubble.
Alan Perlis

By now, we know how to present arbitrary '`Content-type`'s to a browser. In this section, our server will present a 3D world to our browser. The 3D world is described in a scene description language (VRML, Virtual Reality Modeling Language) that allows us to travel through a perspective view of a 2D maze with our browser. Browsers with a VRML plugin enable exploration of this technology. We could do one of those boring '`Hello world`' examples here, that are usually presented when introducing novices to VRML. If you have never written any VRML code, have a look at the VRML FAQ. Presenting a static VRML scene is a bit trivial; in order to expose `gawk`'s new capabilities, we will present a dynamically generated VRML scene. The function `SetUpServer` is very simple because it only sets the default HTML page and initializes the random number generator. As usual, the surrounding server lets you browse the maze.

```
function SetUpServer() {
  TopHeader = "<HTML><title>Walk through a maze</title>"
  TopDoc = "\
    <h2>Please choose one of the following actions:</h2>\
    <UL>\
      <LI><A HREF=" MyPrefix "/AboutServer>About this server</A>\
      <LI><A HREF=" MyPrefix "/VRMLtest>Watch a simple VRML scene</A>\
    </UL>"
  TopFooter  = "</HTML>"
  srand()
}
```

The function `HandleGET` is a bit longer because it first computes the maze and afterwards generates the VRML code that is sent across the network. As shown in the STATIST example (see Section 3.6 [STATIST], page 43), we set the type of the content to VRML and then store the VRML representation of the maze as the page content. We assume that the maze is stored in a 2D array. Initially, the maze consists of walls only. Then, we add an entry and an exit to the maze and let the rest of the work be done by the function `MakeMaze`. Now, only the wall fields are left in the maze. By iterating over the these fields, we generate one line of VRML code for each wall field.

```
function HandleGET() {
  if (MENU[2] == "AboutServer") {
    Document  = "If your browser has a VRML 2 plugin,\
      this server shows you a simple VRML scene."
  } else if (MENU[2] == "VRMLtest") {
    XSIZE = YSIZE = 11              # initially, everything is wall
    for (y = 0; y < YSIZE; y++)
       for (x = 0; x < XSIZE; x++)
          Maze[x, y] = "#"
    delete Maze[0, 1]              # entry is not wall
```

```
      delete Maze[XSIZE-1, YSIZE-2]  # exit  is not wall
      MakeMaze(1, 1)
      Document = "\
#VRML V2.0 utf8\n\
Group {\n\
  children [\n\
    PointLight {\n\
      ambientIntensity 0.2\n\
      color 0.7 0.7 0.7\n\
      location 0.0 8.0 10.0\n\
    }\n\
    DEF B1 Background {\n\
      skyColor [0 0 0, 1.0 1.0 1.0 ]\n\
      skyAngle 1.6\n\
      groundColor [1 1 1, 0.8 0.8 0.8, 0.2 0.2 0.2 ]\n\
      groundAngle [ 1.2 1.57 ]\n\
    }\n\
    DEF Wall Shape {\n\
      geometry Box {size 1 1 1}\n\
      appearance Appearance { material Material { diffuseColor 0 0 1 } }\n\
    }\n\
    DEF Entry Viewpoint {\n\
      position 0.5 1.0 5.0\n\
      orientation 0.0 0.0 -1.0 0.52\n\
    }\n"
    for (i in Maze) {
      split(i, t, SUBSEP)
      Document = Document "    Transform { translation "
      Document = Document t[1] " 0 -" t[2] " children USE Wall }\n"
    }
    Document = Document "  ] # end of group for world\n}"
    Reason = "OK" ORS "Content-type: model/vrml"
    Header = Footer = ""
  }
}
```

Finally, we have a look at `MakeMaze`, the function that generates the `Maze` array. When entered, this function assumes that the array has been initialized so that each element represents a wall element and the maze is initially full of wall elements. Only the entrance and the exit of the maze should have been left free. The parameters of the function tell us which element must be marked as not being a wall. After this, we take a look at the four neighbouring elements and remember which we have already treated. Of all the neighbouring elements, we take one at random and walk in that direction. Therefore, the wall element in that direction has to be removed and then, we call the function recursively for that element. The maze is only completed if we iterate the above procedure for *all* neighbouring elements (in random order) and for our present element by recursively calling the function for the present element. This last iteration could have been done in a loop, but it is done much simpler recursively.

Notice that elements with coordinates that are both odd are assumed to be on our way through the maze and the generating process cannot terminate as long as there is such an element not being `deleted`. All other elements are potentially part of the wall.

```
function MakeMaze(x, y) {
  delete Maze[x, y]     # here we are, we have no wall here
  p = 0                 # count unvisited fields in all directions
  if (x-2 SUBSEP y   in Maze) d[p++] = "-x"
  if (x   SUBSEP y-2 in Maze) d[p++] = "-y"
  if (x+2 SUBSEP y   in Maze) d[p++] = "+x"
  if (x   SUBSEP y+2 in Maze) d[p++] = "+y"
  if (p>0) {            # if there are univisited fields, go there
    p = int(p*rand())   # choose one unvisited field at random
    if        (d[p] == "-x") { delete Maze[x - 1, y]; MakeMaze(x - 2, y)
    } else if (d[p] == "-y") { delete Maze[x, y - 1]; MakeMaze(x, y - 2)
    } else if (d[p] == "+x") { delete Maze[x + 1, y]; MakeMaze(x + 2, y)
    } else if (d[p] == "+y") { delete Maze[x, y + 1]; MakeMaze(x, y + 2)
    }                   # we are back from recursion
    MakeMaze(x, y);     # try again while there are unvisited fields
  }
}
```

3.8 MOBAGWHO: a Simple Mobile Agent

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.
C. A. R. Hoare

A *mobile agent* is a program that can be dispatched from a computer and transported to a remote server for execution. This is called *migration*, which means that a process on another system is started that is independent from its originator. Ideally, it wanders through a network while working for its creator or owner. In places like the UMBC Agent Web, people are quite confident that (mobile) agents are a software engineering paradigm that enables us to significantly increase the efficiency of our work. Mobile agents could become the mediators between users and the networking world. For an unbiased view at this technology, see the remarkable paper *Mobile Agents: Are they a good idea?*.[2]

When trying to migrate a process from one system to another, a server process is needed on the receiving side. Depending on the kind of server process, several ways of implementation come to mind. How the process is implemented depends upon the kind of server process:

- HTTP can be used as the protocol for delivery of the migrating process. In this case, we use a common web server as the receiving server pro-

[2] `http://www.research.ibm.com/massive/mobag.ps`

cess. A universal CGI script mediates between migrating process and web server. Each server willing to accept migrating agents makes this universal service available. HTTP supplies the `POST` method to transfer some data to a file on the web server. When a CGI script is called remotely with the `POST` method instead of the usual `GET` method, data is transmitted from the client process to the standard input of the server's CGI script. So, to implement a mobile agent, we must not only write the agent program to start on the client side, but also the CGI script to receive the agent on the server side.

- The `PUT` method can also be used for migration. HTTP does not require a CGI script for migration via `PUT`. However, with common web servers there is no advantage to this solution, because web servers such as Apache require explicit activation of a special `PUT` script.
- *Agent Tcl* pursues a different course; it relies on a dedicated server process with a dedicated protocol specialized for receiving mobile agents.

Our agent example abuses a common web server as a migration tool. So, it needs a universal CGI script on the receiving side (the web server). The receiving script is activated with a `POST` request when placed into a location like '`/httpd/cgi-bin/PostAgent.sh`'. Make sure that the server system uses a version of `gawk` that supports network access (Version 3.1 or later; verify with '`gawk --version`').

```
#!/bin/sh
MobAg=/tmp/MobileAgent.$$
# direct script to mobile agent file
cat > $MobAg
# execute agent concurrently
gawk -f $MobAg $MobAg > /dev/null &
# HTTP header, terminator and body
gawk 'BEGIN { print "\r\nAgent started" }'
rm $MobAg      # delete script file of agent
```

By making its process id (`$$`) part of the unique file name, the script avoids conflicts between concurrent instances of the script. First, all lines from standard input (the mobile agent's source code) are copied into this unique file. Then, the agent is started as a concurrent process and a short message reporting this fact is sent to the submitting client. Finally, the script file of the mobile agent is removed because it is no longer needed. Although it is a short script, there are several noteworthy points:

Security
: *There is none.* In fact, the CGI script should never be made available on a server that is part of the Internet because everyone would be allowed to execute arbitrary commands with it. This behavior is acceptable only when performing rapid prototyping.

Self-Reference
: Each migrating instance of an agent is started in a way that enables it to read its own source code from standard input and use

the code for subsequent migrations. This is necessary because it needs to treat the agent's code as data to transmit. `gawk` is not the ideal language for such a job. Lisp and Tcl are more suitable because they do not make a distinction between program code and data.

Independence
: After migration, the agent is not linked to its former home in any way. By reporting '`Agent started`', it waves "Goodbye" to its origin. The originator may choose to terminate or not.

The originating agent itself is started just like any other command-line script, and reports the results on standard output. By letting the name of the original host migrate with the agent, the agent that migrates to a host far away from its origin can report the result back home. Having arrived at the end of the journey, the agent establishes a connection and reports the results. This is the reason for determining the name of the host with '`uname -n`' and storing it in `MyOrigin` for later use. We may also set variables with the '`-v`' option from the command line. This interactivity is only of importance in the context of starting a mobile agent; therefore this `BEGIN` pattern and its action do not take part in migration:

```
BEGIN {
  if (ARGC != 2) {
    print "MOBAG - a simple mobile agent"
    print "CALL:\n    gawk -f mobag.awk mobag.awk"
    print "IN:\n    the name of this script as a command-line parameter"
    print "PARAM:\n    -v MyOrigin=myhost.com"
    print "OUT:\n    the result on stdout"
    print "JK 29.03.1998 01.04.1998"
    exit
  }
  if (MyOrigin == "") {
     "uname -n" | getline MyOrigin
     close("uname -n")
  }
}
```

Since `gawk` cannot manipulate and transmit parts of the program directly, the source code is read and stored in strings. Therefore, the program scans itself for the beginning and the ending of functions. Each line in between is appended to the code string until the end of the function has been reached. A special case is this part of the program itself. It is not a function. Placing a similar framework around it causes it to be treated like a function. Notice that this mechanism works for all the functions of the source code, but it cannot guarantee that the order of the functions is preserved during migration:

```
#ReadMySelf
/^function /                    { FUNC = $2 }
/^END/ || /^#ReadMySelf/        { FUNC = $1 }
FUNC != ""                      { MOBFUN[FUNC] = MOBFUN[FUNC] RS $0 }
```

```
(FUNC != "") && (/^}/ || /^#EndOfMySelf/) \
                                { FUNC = "" }
#EndOfMySelf
```

The web server code in Section 2.9 [A Web Service with Interaction], page 20, was first developed as a site-independent core. Likewise, the `gawk`-based mobile agent starts with an agent-independent core, to which can be appended application-dependent functions. What follows is the only application-independent function needed for the mobile agent:

```
function migrate(Destination, MobCode, Label) {
  MOBVAR["Label"] = Label
  MOBVAR["Destination"] = Destination
  RS = ORS = "\r\n"
  HttpService = "/inet/tcp/0/" Destination
  for (i in MOBFUN)
     MobCode = (MobCode "\n" MOBFUN[i])
  MobCode = MobCode  "\n\nBEGIN {"
  for (i in MOBVAR)
     MobCode = (MobCode "\n  MOBVAR[\"" i "\"] = \"" MOBVAR[i] "\"")
  MobCode = MobCode "\n}\n"
  print "POST /cgi-bin/PostAgent.sh HTTP/1.0"  |& HttpService
  print "Content-length:", length(MobCode) ORS |& HttpService
  printf "%s", MobCode                         |& HttpService
  while ((HttpService |& getline) > 0)
     print $0
  close(HttpService)
}
```

The `migrate` function prepares the aforementioned strings containing the program code and transmits them to a server. A consequence of this modular approach is that the `migrate` function takes some parameters that aren't needed in this application, but that will be in future ones. Its mandatory parameter `Destination` holds the name (or IP address) of the server that the agent wants as a host for its code. The optional parameter `MobCode` may contain some `gawk` code that is inserted during migration in front of all other code. The optional parameter `Label` may contain a string that tells the agent what to do in program execution after arrival at its new home site. One of the serious obstacles in implementing a framework for mobile agents is that it does not suffice to migrate the code. It is also necessary to migrate the state of execution of the agent. In contrast to *Agent Tcl*, this program does not try to migrate the complete set of variables. The following conventions are used:

- Each variable in an agent program is local to the current host and does *not* migrate.
- The array `MOBFUN` shown above is an exception. It is handled by the function `migrate` and does migrate with the application.
- The other exception is the array `MOBVAR`. Each variable that takes part in migration has to be an element of this array. `migrate` also takes care of this.

Now it's clear what happens to the `Label` parameter of the function `migrate`. It is copied into `MOBVAR["Label"]` and travels alongside the other data. Since travelling takes place via HTTP, records must be separated with `"\r\n"` in `RS` and `ORS` as usual. The code assembly for migration takes place in three steps:

- Iterate over `MOBFUN` to collect all functions verbatim.
- Prepare a `BEGIN` pattern and put assignments to mobile variables into the action part.
- Transmission itself resembles GETURL: the header with the request and the `Content-length` is followed by the body. In case there is any reply over the network, it is read completely and echoed to standard output to avoid irritating the server.

The application-independent framework is now almost complete. What follows is the `END` pattern that is executed when the mobile agent has finished reading its own code. First, it checks whether it is already running on a remote host or not. In case initialization has not yet taken place, it starts `MyInit`. Otherwise (later, on a remote host), it starts `MyJob`:

```
END {
  if (ARGC != 2) exit     # stop when called with wrong parameters
  if (MyOrigin != "")     # is this the originating host?
    MyInit()              # if so, initialize the application
  else                    # we are on a host with migrated data
    MyJob()               # so we do our job
}
```

All that's left to extend the framework into a complete application is to write two application-specific functions: `MyInit` and `MyJob`. Keep in mind that the former is executed once on the originating host, while the latter is executed after each migration:

```
function MyInit() {
  MOBVAR["MyOrigin"] = MyOrigin
  MOBVAR["Machines"] = "localhost/80 max/80 moritz/80 castor/80"
  split(MOBVAR["Machines"], Machines)           # which host is the first?
  migrate(Machines[1], "", "")                  # go to the first host
  while (("/inet/tcp/8080/0/0" |& getline) > 0) # wait for result
    print $0                                    # print result
  close("/inet/tcp/8080/0/0")
}
```

As mentioned earlier, this agent takes the name of its origin (`MyOrigin`) with it. Then, it takes the name of its first destination and goes there for further work. Notice that this name has the port number of the web server appended to the name of the server, because the function `migrate` needs it this way to create the `HttpService` variable. Finally, it waits for the result to arrive. The `MyJob` function runs on the remote host:

```
function MyJob() {
  # forget this host
  sub(MOBVAR["Destination"], "", MOBVAR["Machines"])
```

```
    MOBVAR["Result"]=MOBVAR["Result"] SUBSEP SUBSEP MOBVAR["Destination"] ":"
    while (("who" | getline) > 0)             # who is logged in?
      MOBVAR["Result"] = MOBVAR["Result"] SUBSEP $0
    close("who")
    if (index(MOBVAR["Machines"], "/") > 0) {   # any more machines to visit?
      split(MOBVAR["Machines"], Machines)       # which host is next?
      migrate(Machines[1], "", "")              # go there
    } else {                                    # no more machines
      gsub(SUBSEP, "\n", MOBVAR["Result"])      # send result to origin
      print MOBVAR["Result"] |& "/inet/tcp/0/" MOBVAR["MyOrigin"] "/8080"
      close("/inet/tcp/0/" MOBVAR["MyOrigin"] "/8080")
    }
  }
```

After migrating, the first thing to do in `MyJob` is to delete the name of the current host from the list of hosts to visit. Now, it is time to start the real work by appending the host's name to the result string, and reading line by line who is logged in on this host. A very annoying circumstance is the fact that the elements of `MOBVAR` cannot hold the newline character (`"\n"`). If they did, migration of this string did not work because the string didn't obey the syntax rule for a string in `gawk`. `SUBSEP` is used as a temporary replacement. If the list of hosts to visit holds at least one more entry, the agent migrates to that place to go on working there. Otherwise, we replace the `SUBSEP`s with a newline character in the resulting string, and report it to the originating host, whose name is stored in `MOBVAR["MyOrigin"]`.

3.9 STOXPRED: Stock Market Prediction As A Service

Far out in the uncharted backwaters of the unfashionable end of the Western Spiral arm of the Galaxy lies a small unregarded yellow sun.

Orbiting this at a distance of roughly ninety-two million miles is an utterly insignificant little blue-green planet whose ape-descendent life forms are so amazingly primitive that they still think digital watches are a pretty neat idea.

This planet has — or rather had — a problem, which was this: most of the people living on it were unhappy for pretty much of the time. Many solutions were suggested for this problem, but most of these were largely concerned with the movements of small green pieces of paper, which is odd because it wasn't the small green pieces of paper that were unhappy.

Douglas Adams, *The Hitch Hiker's Guide to the Galaxy*

Valuable services on the Internet are usually *not* implemented as mobile agents. There are much simpler ways of implementing services. All Unix systems provide, for example, the `cron` service. Unix system users can write a list of tasks to be done each day, each week, twice a day, or just once.

The list is entered into a file named '`crontab`'. For example, to distribute a newsletter on a daily basis this way, use `cron` for calling a script each day early in the morning.

```
# run at 8 am on weekdays, distribute the newsletter
0 8 * * 1-5   $HOME/bin/daily.job >> $HOME/log/newsletter 2>&1
```

The script first looks for interesting information on the Internet, assembles it in a nice form and sends the results via email to the customers.

The following is an example of a primitive newsletter on stock market prediction. It is a report which first tries to predict the change of each share in the Dow Jones Industrial Index for the particular day. Then it mentions some especially promising shares as well as some shares which look remarkably bad on that day. The report ends with the usual disclaimer which tells every child *not* to try this at home and hurt anybody.

```
Good morning Uncle Scrooge,

This is your daily stock market report for Monday, October 16, 2000.
Here are the predictions for today:

        AA      neutral
        GE      up
        JNJ     down
        MSFT    neutral
        ...
        UTX     up
        DD      down
        IBM     up
        MO      down
        WMT     up
        DIS     up
        INTC    up
        MRK     down
        XOM     down
        EK      down
        IP      down

The most promising shares for today are these:

        INTC            http://biz.yahoo.com/n/i/intc.html

The stock shares to avoid today are these:

        EK              http://biz.yahoo.com/n/e/ek.html
        IP              http://biz.yahoo.com/n/i/ip.html
        DD              http://biz.yahoo.com/n/d/dd.html
        ...
```

The script as a whole is rather long. In order to ease the pain of studying other people's source code, we have broken the script up into meaningful parts which are invoked one after the other. The basic structure of the script is as follows:

```
BEGIN {
  Init()
  ReadQuotes()
  CleanUp()
  Prediction()
  Report()
  SendMail()
}
```

The earlier parts store data into variables and arrays which are subsequently used by later parts of the script. The `Init` function first checks if the script is invoked correctly (without any parameters). If not, it informs the user of the correct usage. What follows are preparations for the retrieval of the historical quote data. The names of the 30 stock shares are stored in an array `name` along with the current date in `day`, `month`, and `year`.

All users who are separated from the Internet by a firewall and have to direct their Internet accesses to a proxy must supply the name of the proxy to this script with the '`-v Proxy=name`' option. For most users, the default proxy and port number should suffice.

```
function Init() {
  if (ARGC != 1) {
    print "STOXPRED - daily stock share prediction"
    print "IN:\n    no parameters, nothing on stdin"
    print "PARAM:\n    -v Proxy=MyProxy -v ProxyPort=80"
    print "OUT:\n    commented predictions as email"
    print "JK 09.10.2000"
    exit
  }
  # Remember ticker symbols from Dow Jones Industrial Index
  StockCount = split("AA GE JNJ MSFT AXP GM JPM PG BA HD KO \
    SBC C HON MCD T CAT HWP MMM UTX DD IBM MO WMT DIS INTC \
    MRK XOM EK IP", name);
  # Remember the current date as the end of the time series
  day   = strftime("%d")
  month = strftime("%m")
  year  = strftime("%Y")
  if (Proxy     == "")  Proxy     = "chart.yahoo.com"
  if (ProxyPort ==  0)  ProxyPort = 80
  YahooData = "/inet/tcp/0/" Proxy "/" ProxyPort
}
```

There are two really interesting parts in the script. One is the function which reads the historical stock quotes from an Internet server. The other is the one that does the actual prediction. In the following function we see how the quotes are read from the Yahoo server. The data which comes from the server is in CSV format (comma-separated values):

```
Date,Open,High,Low,Close,Volume
```

```
9-Oct-00,22.75,22.75,21.375,22.375,7888500
6-Oct-00,23.8125,24.9375,21.5625,22,10701100
5-Oct-00,24.4375,24.625,23.125,23.50,5810300
```

Lines contain values of the same time instant, whereas columns are separated by commas and contain the kind of data that is described in the header (first) line. At first, `gawk` is instructed to separate columns by commas ('`FS = ","`'). In the loop that follows, a connection to the Yahoo server is first opened, then a download takes place, and finally the connection is closed. All this happens once for each ticker symbol. In the body of this loop, an Internet address is built up as a string according to the rules of the Yahoo server. The starting and ending date are chosen to be exactly the same, but one year apart in the past. All the action is initiated within the `printf` command which transmits the request for data to the Yahoo server.

In the inner loop, the server's data is first read and then scanned line by line. Only lines which have six columns and the name of a month in the first column contain relevant data. This data is stored in the two-dimensional array `quote`; one dimension being time, the other being the ticker symbol. During retrieval of the first stock's data, the calendar names of the time instances are stored in the array `day` because we need them later.

```
function ReadQuotes() {
  # Retrieve historical data for each ticker symbol
  FS = ","
  for (stock = 1; stock <= StockCount; stock++) {
    URL = "http://chart.yahoo.com/table.csv?s=" name[stock] \
          "&a=" month "&b=" day   "&c=" year-1 \
          "&d=" month "&e=" day   "&f=" year \
          "g=d&q=q&y=0&z=" name[stock] "&x=.csv"
    printf("GET " URL " HTTP/1.0\r\n\r\n") |& YahooData
    while ((YahooData |& getline) > 0) {
      if (NF == 6 && $1 ~ /Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec/) {
        if (stock == 1)
          days[++daycount] = $1;
        quote[$1, stock] = $5
      }
    }
    close(YahooData)
  }
  FS = " "
}
```

Now that we *have* the data, it can be checked once again to make sure that no individual stock is missing or invalid, and that all the stock quotes are aligned correctly. Furthermore, we renumber the time instances. The most recent day gets day number 1 and all other days get consecutive numbers. All quotes are rounded toward the nearest whole number in US Dollars.

```
function CleanUp() {
  # clean up time series; eliminate incomplete data sets
  for (d = 1; d <= daycount; d++) {
    for (stock = 1; stock <= StockCount; stock++)
```

```
      if (! ((days[d], stock) in quote))
          stock = StockCount + 10
    if (stock > StockCount + 1)
        continue
    datacount++
    for (stock = 1; stock <= StockCount; stock++)
      data[datacount, stock] = int(0.5 + quote[days[d], stock])
  }
  delete quote
  delete days
}
```

Now we have arrived at the second really interesting part of the whole affair. What we present here is a very primitive prediction algorithm: *If a stock fell yesterday, assume it will also fall today; if it rose yesterday, assume it will rise today.* (Feel free to replace this algorithm with a smarter one.) If a stock changed in the same direction on two consecutive days, this is an indication which should be highlighted. Two-day advances are stored in `hot` and two-day declines in `avoid`.

The rest of the function is a sanity check. It counts the number of correct predictions in relation to the total number of predictions one could have made in the year before.

```
function Prediction() {
  # Predict each ticker symbol by prolonging yesterday's trend
  for (stock = 1; stock <= StockCount; stock++) {
    if         (data[1, stock] > data[2, stock]) {
      predict[stock] = "up"
    } else if  (data[1, stock] < data[2, stock]) {
      predict[stock] = "down"
    } else {
      predict[stock] = "neutral"
    }
    if ((data[1, stock] > data[2, stock]) && (data[2, stock] > data[3, stock]))
      hot[stock] = 1
    if ((data[1, stock] < data[2, stock]) && (data[2, stock] < data[3, stock]))
      avoid[stock] = 1
  }
  # Do a plausibility check: how many predictions proved correct?
  for (s = 1; s <= StockCount; s++) {
    for (d = 1; d <= datacount-2; d++) {
      if         (data[d+1, s] > data[d+2, s]) {
        UpCount++
      } else if  (data[d+1, s] < data[d+2, s]) {
        DownCount++
      } else {
        NeutralCount++
      }
      if (((data[d, s]  > data[d+1, s]) && (data[d+1, s]  > data[d+2, s])) ||
          ((data[d, s]  < data[d+1, s]) && (data[d+1, s]  < data[d+2, s])) ||
          ((data[d, s] == data[d+1, s]) && (data[d+1, s] == data[d+2, s])))
        CorrectCount++
```

```
    }
  }
}
```

At this point the hard work has been done: the array `predict` contains the predictions for all the ticker symbols. It is up to the function `Report` to find some nice words to introduce the desired information.

```
function Report() {
  # Generate report
  report =        "\nThis is your daily "
  report = report "stock market report for "strftime("%A, %B %d, %Y")".\n"
  report = report "Here are the predictions for today:\n\n"
  for (stock = 1; stock <= StockCount; stock++)
    report = report "\t" name[stock] "\t" predict[stock] "\n"
  for (stock in hot) {
    if (HotCount++ == 0)
      report = report "\nThe most promising shares for today are these:\n\n"
    report = report "\t" name[stock] "\t\thttp://biz.yahoo.com/n/" \
      tolower(substr(name[stock], 1, 1)) "/" tolower(name[stock]) ".html\n"
  }
  for (stock in avoid) {
    if (AvoidCount++ == 0)
      report = report "\nThe stock shares to avoid today are these:\n\n"
    report = report "\t" name[stock] "\t\thttp://biz.yahoo.com/n/" \
      tolower(substr(name[stock], 1, 1)) "/" tolower(name[stock]) ".html\n"
  }
  report = report "\nThis sums up to " HotCount+0 " winners and " AvoidCount+0
  report = report " losers. When using this kind\nof prediction scheme for"
  report = report " the 12 months which lie behind us,\nwe get " UpCount
  report = report " 'ups' and " DownCount " 'downs' and " NeutralCount
  report = report " 'neutrals'. Of all\nthese " UpCount+DownCount+NeutralCount
  report = report " predictions " CorrectCount " proved correct next day.\n"
  report = report "A success rate of "\
           int(100*CorrectCount/(UpCount+DownCount+NeutralCount)) "%.\n"
  report = report "Random choice would have produced a 33% success rate.\n"
  report = report "Disclaimer: Like every other prediction of the stock\n"
  report = report "market, this report is, of course, complete nonsense.\n"
  report = report "If you are stupid enough to believe these predictions\n"
  report = report "you should visit a doctor who can treat your ailment."
}
```

The function `SendMail` goes through the list of customers and opens a pipe to the `mail` command for each of them. Each one receives an email message with a proper subject heading and is addressed with his full name.

```
function SendMail() {
  # send report to customers
  customer["uncle.scrooge@ducktown.gov"] = "Uncle Scrooge"
  customer["more@utopia.org"           ] = "Sir Thomas More"
  customer["spinoza@denhaag.nl"        ] = "Baruch de Spinoza"
  customer["marx@highgate.uk"          ] = "Karl Marx"
  customer["keynes@the.long.run"       ] = "John Maynard Keynes"
  customer["bierce@devil.hell.org"     ] = "Ambrose Bierce"
  customer["laplace@paris.fr"          ] = "Pierre Simon de Laplace"
```

```
    for (c in customer) {
      MailPipe = "mail -s 'Daily Stock Prediction Newsletter'" c
      print "Good morning " customer[c] "," | MailPipe
      print report "\n.\n" | MailPipe
      close(MailPipe)
    }
  }
```

Be patient when running the script by hand. Retrieving the data for all the ticker symbols and sending the emails may take several minutes to complete, depending upon network traffic and the speed of the available Internet link. The quality of the prediction algorithm is likely to be disappointing. Try to find a better one. Should you find one with a success rate of more than 50%, please tell us about it! It is only for the sake of curiosity, of course. `:-)`

3.10 PROTBASE: Searching Through A Protein Database

Hoare's Law of Large Problems: Inside every large problem is a small problem struggling to get out.

Yahoo's database of stock market data is just one among the many large databases on the Internet. Another one is located at NCBI (National Center for Biotechnology Information). Established in 1988 as a national resource for molecular biology information, NCBI creates public databases, conducts research in computational biology, develops software tools for analyzing genome data, and disseminates biomedical information. In this section, we look at one of NCBI's public services, which is called BLAST (Basic Local Alignment Search Tool).

You probably know that the information necessary for reproducing living cells is encoded in the genetic material of the cells. The genetic material is a very long chain of four base nucleotides. It is the order of appearance (the sequence) of nucleotides which contains the information about the substance to be produced. Scientists in biotechnology often find a specific fragment, determine the nucleotide sequence, and need to know where the sequence at hand comes from. This is where the large databases enter the game. At NCBI, databases store the knowledge about which sequences have ever been found and where they have been found. When the scientist sends his sequence to the BLAST service, the server looks for regions of genetic material in its database which look the most similar to the delivered nucleotide sequence. After a search time of some seconds or minutes the server sends an answer to the scientist. In order to make access simple, NCBI chose to offer their database service through popular Internet protocols. There are four basic ways to use the so-called BLAST services:

- The easiest way to use BLAST is through the web. Users may simply point their browsers at the NCBI home page and link to the BLAST pages. NCBI provides a stable URL that may be used to perform BLAST searches without interactive use of a web browser. This is what we will do later in this section. A demonstration client and a '`README`' file demonstrate how to access this URL.

- Currently, `blastcl3` is the standard network BLAST client. You can download `blastcl3` from the anonymous FTP location.
- BLAST 2.0 can be run locally as a full executable and can be used to run BLAST searches against private local databases, or downloaded copies of the NCBI databases. BLAST 2.0 executables may be found on the NCBI anonymous FTP server.
- The NCBI BLAST Email server is the best option for people without convenient access to the web. A similarity search can be performed by sending a properly formatted mail message containing the nucleotide or protein query sequence to `blast@ncbi.nlm.nih.gov`. The query sequence is compared against the specified database using the BLAST algorithm and the results are returned in an email message. For more information on formulating email BLAST searches, you can send a message consisting of the word "HELP" to the same address, `blast@ncbi.nlm.nih.gov`.

Our starting point is the demonstration client mentioned in the first option. The '`README`' file that comes along with the client explains the whole process in a nutshell. In the rest of this section, we first show what such requests look like. Then we show how to use `gawk` to implement a client in about 10 lines of code. Finally, we show how to interpret the result returned from the service.

Sequences are expected to be represented in the standard IUB/IUPAC amino acid and nucleic acid codes, with these exceptions: lower-case letters are accepted and are mapped into upper-case; a single hyphen or dash can be used to represent a gap of indeterminate length; and in amino acid sequences, '`U`' and '`*`' are acceptable letters (see below). Before submitting a request, any numerical digits in the query sequence should either be removed or replaced by appropriate letter codes (e.g., '`N`' for unknown nucleic acid residue or '`X`' for unknown amino acid residue). The nucleic acid codes supported are:

```
A --> adenosine              M --> A C (amino)
C --> cytidine               S --> G C (strong)
G --> guanine                W --> A T (weak)
T --> thymidine              B --> G T C
U --> uridine                D --> G A T
R --> G A (purine)           H --> A C T
Y --> T C (pyrimidine)       V --> G C A
K --> G T (keto)             N --> A G C T (any)
                             -  gap of indeterminate length
```

Now you know the alphabet of nucleotide sequences. The last two lines of the following example query show you such a sequence, which is obviously made up only of elements of the alphabet just described. Store this example query into a file named 'protbase.request'. You are now ready to send it to the server with the demonstration client.

```
PROGRAM blastn
DATALIB month
EXPECT 0.75
BEGIN
>GAWK310 the gawking gene GNU AWK
tgcttggctgaggagccataggacgagagcttcctggtgaagtgtgtttcttgaaatcat
caccaccatggacagcaaa
```

The actual search request begins with the mandatory parameter 'PROGRAM' in the first column followed by the value 'blastn' (the name of the program) for searching nucleic acids. The next line contains the mandatory search parameter 'DATALIB' with the value 'month' for the newest nucleic acid sequences. The third line contains an optional 'EXPECT' parameter and the value desired for it. The fourth line contains the mandatory 'BEGIN' directive, followed by the query sequence in FASTA/Pearson format. Each line of information must be less than 80 characters in length.

The "month" database contains all new or revised sequences released in the last 30 days and is useful for searching against new sequences. There are five different blast programs, blastn being the one that compares a nucleotide query sequence against a nucleotide sequence database.

The last server directive that must appear in every request is the 'BEGIN' directive. The query sequence should immediately follow the 'BEGIN' directive and must appear in FASTA/Pearson format. A sequence in FASTA/Pearson format begins with a single-line description. The description line, which is required, is distinguished from the lines of sequence data that follow it by having a greater-than ('>') symbol in the first column. For the purposes of the BLAST server, the text of the description is arbitrary.

If you prefer to use a client written in gawk, just store the following 10 lines of code into a file named 'protbase.awk' and use this client instead. Invoke it with 'gawk -f protbase.awk protbase.request'. Then wait a minute and watch the result coming in. In order to replicate the demonstration client's behaviour as closely as possible, this client does not use a proxy server. We could also have extended the client program in Section 3.2 [Retrieving Web Pages], page 36, to implement the client request from 'protbase.awk' as a special case.

```
{ request = request "\n" $0 }

END {
  BLASTService     = "/inet/tcp/0/www.ncbi.nlm.nih.gov/80"
  printf "POST /cgi-bin/BLAST/nph-blast_report HTTP/1.0\n" |& BLASTService
```

```
    printf "Content-Length: " length(request) "\n\n"        |& BLASTService
    printf request                                          |& BLASTService
    while ((BLASTService |& getline) > 0)
        print $0
    close(BLASTService)
  }
```

The demonstration client from NCBI is 214 lines long (written in C) and it is not immediately obvious what it does. Our client is so short that it *is* obvious what it does. First it loops over all lines of the query and stores the whole query into a variable. Then the script establishes an Internet connection to the NCBI server and transmits the query by framing it with a proper HTTP request. Finally it receives and prints the complete result coming from the server.

Now, let us look at the result. It begins with an HTTP header, which you can ignore. Then there are some comments about the query having been filtered to avoid spuriously high scores. After this, there is a reference to the paper that describes the software being used for searching the data base. After a repitition of the original query's description we find the list of significant alignments:

```
Sequences producing significant alignments:                      (bits)  Value

gb|AC021182.14|AC021182 Homo sapiens chromosome 7 clone RP11-733...    38  0.20
gb|AC021056.12|AC021056 Homo sapiens chromosome 3 clone RP11-115...    38  0.20
emb|AL160278.10|AL160278 Homo sapiens chromosome 9 clone RP11-57...    38  0.20
emb|AL391139.11|AL391139 Homo sapiens chromosome X clone RP11-35...    38  0.20
emb|AL365192.6|AL365192 Homo sapiens chromosome 6 clone RP3-421H...    38  0.20
emb|AL138812.9|AL138812 Homo sapiens chromosome 11 clone RP1-276...    38  0.20
gb|AC073881.3|AC073881 Homo sapiens chromosome 15 clone CTD-2169...    38  0.20
```

This means that the query sequence was found in seven human chromosomes. But the value 0.20 (20%) means that the probability of an accidental match is rather high (20%) in all cases and should be taken into account. You may wonder what the first column means. It is a key to the specific database in which this occurence was found. The unique sequence identifiers reported in the search results can be used as sequence retrieval keys via the NCBI server. The syntax of sequence header lines used by the NCBI BLAST server depends on the database from which each sequence was obtained. The table below lists the identifiers for the databases from which the sequences were derived.

GenBank	`gb|accession|locus`
EMBL Data Library	`emb|accession|locus`
DDBJ, DNA Database of Japan	`dbj|accession|locus`
NBRF PIR	`pir||entry`
Protein Research Foundation	`prf||name`
SWISS-PROT	`sp|accession|entry name`
Brookhaven Protein Data Bank	`pdb|entry|chain`
Kabat's Sequences of Immuno...	`gnl|kabat|identifier`
Patents	`pat|country|number`

GenInfo Backbone Id	`bbs|number`

For example, an identifier might be '`gb|AC021182.14|AC021182`', where the '`gb`' tag indicates that the identifier refers to a GenBank sequence, '`AC021182.14`' is its GenBank ACCESSION, and '`AC021182`' is the GenBank LOCUS. The identifier contains no spaces, so that a space indicates the end of the identifier.

Let us continue in the result listing. Each of the seven alignments mentioned above is subsequently described in detail. We will have a closer look at the first of them.

```
>gb|AC021182.14|AC021182 Homo sapiens chromosome 7 clone RP11-733N23, WORKING DRAFT SEQUENCE, 4
            unordered pieces
         Length = 176383

 Score = 38.2 bits (19), Expect = 0.20
 Identities = 19/19 (100%)
 Strand = Plus / Plus

Query: 35    tggtgaagtgtgtttcttg 53
             |||||||||||||||||||
Sbjct: 69786 tggtgaagtgtgtttcttg 69804
```

This alignment was located on the human chromosome 7. The fragment on which part of the query was found had a total length of 176383. Only 19 of the nucleotides matched and the matching sequence ran from character 35 to 53 in the query sequence and from 69786 to 69804 in the fragment on chromosome 7. If you are still reading at this point, you are probably interested in finding out more about Computational Biology and you might appreciate the following hints.

1. There is a book called *Introduction to Computational Biology* by Michael S. Waterman, which is worth reading if you are seriously interested. You can find a good book review on the Internet.
2. While Waterman's book can explain to you the algorithms employed internally in the database search engines, most practicioners prefer to approach the subject differently. The applied side of Computational Biology is called Bioinformatics, and emphasizes the tools available for day-to-day work as well as how to actually *use* them. One of the very few affordable books on Bioinformatics is *Developing Bioinformatics Computer Skills*.
3. The sequences *gawk* and *gnuawk* are in widespread use in the genetic material of virtually every earthly living being. Let us take this as a clear indication that the divine creator has intended `gawk` to prevail over other scripting languages such as `perl`, `tcl`, or `python` which are not even proper sequences. (:-)

4 Related Links

This section lists the URLs for various items discussed in this chapter. They are presented in the order in which they appear.

Internet Programming with Python
`http://www.fsbassociates.com/books/python.htm`

Advanced Perl Programming
`http://www.oreilly.com/catalog/advperl`

Web Client Programming with Perl
`http://www.oreilly.com/catalog/webclient`

Richard Stevens's home page and book
`http://www.kohala.com/~rstevens`

The SPAK home page
`http://www.userfriendly.net/linux/RPM/contrib/libc6/i386/spak-0.6b-1.i386.html`

Volume III of *Internetworking with TCP/IP*, by Comer and Stevens
`http://www.cs.purdue.edu/homes/dec/tcpip3s.cont.html`

XBM Graphics File Format
`http://www.wotsit.org/download.asp?f=xbm`

GNUPlot `http://www.cs.dartmouth.edu/gnuplot_info.html`

Mark Humphrys' Eliza page
`http://www.compapp.dcu.ie/~humphrys/eliza.html`

Yahoo! Eliza Information
`http://dir.yahoo.com/Recreation/Games/Computer_Games/Internet_Games/Web_Games/Artificial_Intelligence`

Java versions of Eliza
`http://www.tjhsst.edu/Psych/ch1/eliza.html`

Java versions of Eliza with source code
`http://home.adelphia.net/~lifeisgood/eliza/eliza.htm`

Eliza Programs with Explanations
`http://chayden.net/chayden/eliza/Eliza.shtml`

Loebner Contest
`http://acm.org/~loebner/loebner-prize.htmlx`

Tck/Tk Information
`http://www.scriptics.com/`

Intel 80x86 Processors
`http://developer.intel.com/design/platform/embedpc/what_is.htm`

AMD Elan Processors
`http://www.amd.com/products/epd/processors/4.32bitcont/32bitcont/index.html`

XINU `http://willow.canberra.edu.au/~chrisc/xinu.html`

GNU/Linux
`http://uclinux.lineo.com/`

Embedded PCs
`http://dir.yahoo.com/Business_and_Economy/Business_to_Business/Computers/Hardware/Embedded_Control/`

MiniSQL `http://www.hughes.com.au/library/`

Market Share Surveys
`http://www.netcraft.com/survey`

Numerical Recipes in C: The Art of Scientific Computing
`http://www.nr.com`

VRML `http://www.vrml.org`

The VRML FAQ
`http://www.vrml.org/technicalinfo/specifications/specifications.htm#FAQ`

The UMBC Agent Web
`http://www.cs.umbc.edu/agents`

Apache Web Server
`http://www.apache.org`

National Center for Biotechnology Information (NCBI)
`http://www.ncbi.nlm.nih.gov`

Basic Local Alignment Search Tool (BLAST)
`http://www.ncbi.nlm.nih.gov/BLAST/blast_overview.html`

NCBI Home Page
`http://www.ncbi.nlm.nih.gov`

BLAST Pages
`http://www.ncbi.nlm.nih.gov/BLAST`

BLAST Demonstration Client
`ftp://ncbi.nlm.nih.gov/blast/blasturl/`

BLAST anonymous FTP location
`ftp://ncbi.nlm.nih.gov/blast/network/netblast/`

BLAST 2.0 Executables
`ftp://ncbi.nlm.nih.gov/blast/executables/`

IUB/IUPAC Amino Acid and Nucleic Acid Codes
`http://www.uthscsa.edu/geninfo/blastmail.html#item6`

FASTA/Pearson Format
`http://www.ncbi.nlm.nih.gov/BLAST/fasta.html`

Fasta/Pearson Sequence in Java
`http://www.kazusa.or.jp/java/codon_table_java/`

Book Review of *Introduction to Computational Biology*
`http://www.acm.org/crossroads/xrds5-1/introcb.html`

Developing Bioinformatics Computer Skills
`http://www.oreilly.com/catalog/bioskills/`

GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

   B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

   C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

   D. Preserve all the copyright notices of the Document.

   E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

   F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

   G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

   H. Include an unaltered copy of this License.

   I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warrany Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

/

|

A

B

C

D

E

F

G

H

T

U

V

W

X

Y