Harnessing the Power of the Web
(Web Automation and Libwww-perl)

Nadav Har'El*

February 26, 2004†

Abstract

Wielding a Web browser (such as Mozilla or Internet Explorer), the world is at your fingertips. Stock quotes, newspapers, your bank statement — all are just a few clicks away. You can also send SMSs for free, renew your library books, and check your university grades. Web automation allows you to harness all this power and more, by scripting what normally are interactive surfing sessions. We talk about Web automation, its past, present and future, give some useful real-world examples of what one can do with it, and introduce Libwww-perl, a Perl module for Web automation.

Contents

*Email: nyh@math.technion.ac.il; Works a researcher in IBM Haifa Research Lab

†Appeared in proceedings of YAPC Israel 2004

1 Introduction

1.1 Historic survey

ARPANET, the Internet's predecessor, went online in 1969. But in the seventies and eighties, and as late as the first half of the nineties, the network was a bubble, almost separate from the real world around it. Most of the people you knew in real life (outside your university or research company) were not reachable by email, and conversely, many of the people you did communicate with over the network you didn't know in real life. Most services you got from the Internet — such as Usenet, email, FTP file repositories — were not front-ends of any "real-life" institutions, and the institutions you had daily real-world contact with, such as your bank or phone company, had no online presence.

This meant that outside the bubble of computer file sharing and cooperation with research colleagues, being online gave you little *power*. For all daily "real-world" tasks — such as paying your bills, finding your bank balance, sending a letter to your senator, or ordering a product from a catalog — you still had to use so-called "real" communication methods, such as the telephone or the post (or even physically traveling to some office and making the request).

But the Internet did not remain a bubble forever. In 1989, Tim Burners-Lee, a CERN researcher, invented a new model, the *World Wide Web*[1], which was supposed to facilitate the use of the Internet by ordinary people, rather than just highly-trained researchers. People would be able to intuitively jump from content to content using *hyperlinks*. Content would no longer be pure text, but also include graphics, sounds, and more. Content could be interactive, based on the user's requests. In 1993, Mosaic, the first graphical *browser* for the World Wide Web was written by Marc Andreessen of NCSA, an event which sparked the phenomenal growth of the World Wide Web.

The growth of the World Wide Web was an upward spiral of supply and demand, or content and readers. As more interesting content was added to the Web, more curious users began to use Mosaic. At the same time ordinary people, not only researchers and students, started getting Internet connections from commercial Internet Service Providers. The more Mosaic was used, the more real-life entities, such as people and companies, wanted to have a Web page of their own. Initially, just a few companies with a small advertisement pages, but as the number of readers grew, more and more companies created pages you could actually interact with, and even buy products and services from.

By the dawn of the new century, the Internet has become commonplace in most industrialized countries. A large percentage of the population has an Internet connection, and they expect companies and government institutions to supply services and information through the Internet, mostly using the Web. Using the Internet now gives you real-world power. Wielding a Web browser (such as Mozilla or Internet Explorer), the world is at your fingertips. Stock quotes, newspapers, your bank statement — are all just a few clicks away. You can order a book from a retailer in another continent, or order food from your local grocery store.

Moreover, usage of the Web (and the Internet as a whole) continues to grow. More people are getting an Internet connection, companies and organizations are increasing their online presence and their array of online services, and more and more information that was previously obtainable only in the real world (e.g., by going to a library and opening a book) is now available on the Web. Inevitably, some online tools are being invented that had no real-world equivalent (e.g., ICQ or search engines). But unlike a decade earlier, the Internet is now so intertwined with your real world that your use of these tools grants you powers in the real world.

1.2 Today — the power of automation

But with the advent of Web information sources and services, comes a unique opportunity - that of automation.

For example, imagine that you want to keep track of your checking-account balance, so that you know when you're running low on funds and need to withdraw some money from your savings account. Calling

[1]the two key features of the World Wide Web model were the URL (Uniform Resource Locator) and HTML (the Hyptertext Markup Language). The underlying concept of *hypertext* is based on existing ideas, like the "memex" machine that Vannevar Bush envisioned in 1945 (see [2]), or Ted Nelson's Project Xanadu. But in 1989, the Internet was finally ripe for such ideas.

your bank clerk every day asking for your balance, or stopping by an ATM every day is quite annoying, if at all feasible.

Thanks to the fact that most banks now have Web interfaces, you can replace these time-consuming tasks by a quick "surfing" to the bank's Web site, where you can check your balance with a few mouse clicks. But remembering to browse your bank's Web site every single day, just in case some rare event (you running out of funds) happens, is still tedious. More tedious than it needs to be. This entire task can be automated: a computer program that you run on your own Internet-connected machine can tirelessly simulate your surfing of the bank's site every day, and notify you when the event of your interest has taken place. By doing this, you are finally harnessing the full extent of the power that the Web gives you.

It is worthy noting that while competent programmers can create such Web-automation programs[2], not only programmers can benefit from the idea of web automation. In fact, more and more public and commercial services and sites appear that do nothing more than automate the use of other Web sites. For example, a service exists (in Israeli universities) to automatically renew library books. Several commercial services exist for "bid sniping" on Ebay. Several free programs can send SMSs by using Web gateways. (All these services will be explained in detail below.) Surely, these services *could have been* provided by the original service provider (the library, Ebay, etc.) but they were not — and the beauty of the concept outlined here is that it doesn't matter, another party can easily provide these missing services simply by automating what are normally interactive Web surfing sessions.

1.3 The future

The automation methods we described above are based on the idea that a piece of software can simulate an interactive Web-surfing session, as if a real human is using a Web browser to access the Web site. Recently, it has been argued that it is wasteful for one computer program (the automation program) to communicate with another computer program (the Web server) using a human-oriented language (web pages with text and graphics, textual forms, etc.). Not only is it wasteful, it is prone to errors: the automation program cannot really understand what the server tells it, often cannot tolerate changes in the page format, despite these changes not being meant for it, but rather for the human audience.

Instead, the concept of *Web Services* has been proposed (see [13]). Web Services are meant to be used by machines (i.e., the automation programs we discuss), not directly by humans. The requests and answers are sent as XML files with strict formats, which only convey machine-readable meaning, without any of the human-oriented visual clutter that comprises most ordinary Web-pages.

Where such a Web Service is available, using this interface instead of the human-oriented one for communicating with a Web site is indeed better. For example, consider an Amazon.com associate creating a site that sells a subset of Amazon's books (this example is described in greater detail later). Originally, this associate would have needed to parse Amazon's human-readable Web pages to extract the information he or she wanted to display (e.g., the prices of the books). As the look of these human-readable pages changed often, associates found themselves needing to modify these extraction scripts several times a year. But in 2002, Amazon adopted a Web Services interface parallel to its normal human interface. Book details are now retrievable as XML data, and as this data is not intended for human eyes its format is stable and does not undergo periodic cosmetic "improvements".

The problem with the idea of Web Services, however, is that people rarely (if ever) view it as the primary function of a Web site. Amazon.com is first and foremost intended for human viewing, and the Web Services was an afterthought, an added value. This means that for a Web site with a limited budget, its Web Services interface is likely to be very lacking, if it exists at all, and not cover all the services offered to the human audience of the site. Two years after Amazon's adoption of the Web Services standard, the vast majority of Web sites have no support for it at all. On the other hand, because all Web sites still cater to humans and have normal Web interfaces, the method described in this article of automating the use of ordinary Web sites is applicable in virtually all Web sites in existence, and requires no cooperation from the Web site designer.

For Web Services to completely usurp the Web automation described here, the roles of the machine- and human-readable Web would have to be reversed: Site designers would need to start with creating a machine-usable Web Service, and only then build a human-usable interface (i.e, an ordinary Web site) on top of it.

[2] We'll explain below how, and recommend using Perl and the Libwww-perl library

This is similar to the original Unix program design philosophy, where command-line and pipe-based tools are first written, and only later a graphical user interface is built around them. Another option is to develop Web-site-building frameworks that generate human and Web-Services interfaces concurrently, with neither of these interfaces considered more basic than the other. Whether these philosophies will be adopted in the Web still remains to be seen. Until that happens, automating the use of ordinary Web sites, as described in this article, will remain a powerful and useful technique.

1.4 This article

The rest of this article is structured as follows: We begin by giving more examples of how automation of Web surfing can be useful. We continue to survey a few approaches on how to actually do that automation, followed by a short tutorial on our method of choice — Libwww-perl (a Perl module).

2 Some Examples of Web Automation

This section gives some examples of how useful Web automation can be. Most of these are actual tasks that I or my friends were interested in automating in the last few years, and were implemented using Libwww-perl or similar alternatives (see the next section on a few implementation options).

2.1 Renewing library books

Universities have traditionally been in the forefront of networking. In the beginning of the 90s, when hardly anything or anybody was connected to the Internet, all libraries of Israeli universities were already fully computerized and networked. Instead of coming to the library to query the availability of some book, renew a book you had loaned, or order a book, you could do those things over the network. Because the Web did not exist yet, this system (called *Aleph*) had an ad-hoc textual interface over a Telnet connection.

When given the possibility to renew their loaned library books, students wondered why not get a computer to do it every day, automatically, and send them a warning by email when this is no longer possible (i.e., somebody had ordered one of the books). This was usually done using the TCL/Expect language (the most common way to automate an interactive Telnet connection, at the time), and later this was even offered as a centralized service — one person collected a list of many reader IDs and email addresses, and automatically renewed the books for all of them.

In recent years, the Telnet-based library services have been replaced by Web sites, which are more convenient for the users and could still be automated easily, just as we explain. In fact, renewing your library books now only requires fetching one page, easily done with one "curl" command on machines which have this useful free-software utility.

2.2 Sending SMSs

Consider a Web automation script that does something useful, say fetching your bank balance. Instead of running this script yourself every day, you may want to instruct the computer to run it every day automatically, and only inform you if something interesting happens. But how will you be informed of this event? Until recently, being informed by email was the norm: If something of interest happened, the script would send you an email.

But email had a serious problem: to read your email you needed to manually log in and check your email. Light users did this rarely, while heavy users found themselves logging-in several times a day, just to check if they have any important new mail. Moreover, how was Web automation to have an impact on ordinary people when most "ordinary" people did not have email accounts?

It would be far more convenient if you could be notified on a small mobile device that you were carrying anyway, and was always on. Pagers fit this description, but for various reasons they never caught on. A similar service on mobile phones, called Short Message Service (SMS) was introduced in the late nineties[3] and

[3]The first experimental SMS is believed to have been sent in late 1992, but the service only became available in Israel in 1999.

got a very warm reception. Users began wishing that they could get notifications, and even announcement of new email, directly to their cellular phone. And thanks to the developed state of the Web in the late 90s and to Web automation, they could:

In the early 90s, it was already possible to send pager messages from a computer. But this usually required the computer to make a modem phone call to the service company, a solution which was costly and required a special hardware setup. When SMS was introduced, modems were no longer in vogue, and the Web was the latest hype. So starting in 1999, all cellular providers in Israel have a form on their Website with which you can send SMSs to phones of this provider[4].

These SMS-sending forms can, and naturally were, automated. The author of this article released in 1999 the free script SendSMS [10] (written in Perl and Libwww-perl) which let its users send SMSs from any Internet-connected machine to any Israeli mobile phone. This script was later used by many to send SMSs to their phone when new email arrive, or when other events happened. Some of these events were themselves the result of Web automation: notification of library events, new grades, stock prices and bank balances were all sent by various users to mobile phones rather than to email accounts.

Scripts similar to SendSMS were also developed for other providers all over the world. Some providers charge money for these SMSs, but for many people, the convenience is worth the price.

2.3 Checking your bank balance

Many years ago, banks kept their records on paper. This has long given way to computers, but until the mid 90s you still had to physically go to an ATM, or call your banker, to do something in your account or even just to query your balance. In the mid 90s, banks started offering computer access from home, using a modem connection and proprietary software. In the end of the 90s, this scheme was replaced by easier, more standard and more flexible Web interfaces.

These Web interfaces let a person check the balance of his account, follow the activity in his account (like checks being cashed), check the performance of his savings accounts and stock portfolio, and so on. But better yet, all of this could now be automated. You could get your computer to notify you when your balance is running low, send you your balance by SMS every day, send you an SMS whenever a check was cashed, and so on. In Israel, several people released free software (written in Perl and Libwww-perl) which automate the connection to the Websites of most Israeli banks, and extract the requested information on your account.

2.4 Following stocks, funds and price indices

In Israel (at least), newspapers have a few pages dedicated to listing the latest prices of stocks, bonds, mutual funds, foreign currency, stock market indices, etc. When relevant, these papers also publish other economic indices, like the price index announced monthly. People who invest in the stock market, used to browse these lists often (sometimes daily), checking how their investments were doing and contemplating on changing them.

But daily searching a big table printed in a tiny font for information on your stocks was tedious. When Web sites started carrying this information (e.g., Yahoo Finance for the American stock market, and the Globes online newspaper for the Israeli market), it became much easier to follow your favorite stocks. You can automatically extract from the Web sites the information about the stocks that interest you, and then have this information sent to you daily, or just when certain events of interest happen (say, a stock has changed by more than 10%). And doing this is not only easy, it doesn't cost a penny.

2.5 Following bills

> *A duck walks into a drugstore and asks for a condom. The druggists says: "Here you are. Shall I put it on your bill?". The duck says "Hey! What kind of duck do you think I am?"*

[4] These services started out free and unlimited for everyone. Providers changed their terms several times, but sending a limited number of SMSs per day to your own phone, for free, is still a reality in Israel

In the modern world, many of the services you pay for give you credit. When you make a phone call you don't pay immediately (like you do when you use a payphone) — instead, the phone company adds a charge to your bill. You may see this bill, with a month-full of calls, only after a few weeks. Similarly, your credit-card company collects charges throughout the month and adds them to a bill which is sent to you monthly.

Because nowadays all of this billing process is done by computers, these companies already have the technical ability to produce partial, true-to- the-minute bills at any time. These started as special services on special media (e.g., see your cable TV bill on your TV, or listen to your cellphone bill on your cellphone), but many companies now have a Web site where you can see your up-to-date, true-to-the-minute, bill.

There is plenty you can do by automating the use of these bill sites. For example, you can send yourself daily summaries of credit card charges that were made the day before. You can automatically monitor your child's cellphone bill and alert you when it is growing too fast. People have even used these sites for checking for suspicious (and probably fraudulent) activity on their expense, such as somebody using their phone during the night.

2.6 Directories and Schedules

In recent years, most directories and schedules are available on the Web. Phone directories, zipcode directories, bus and train schedules, TV schedules, movie screening times, and so on, are all a few clicks away, and the automatic extraction of this information is therefore easy.

Many ways in which these can be useful can be imagined. You can write a script that each week mails you the airing times of your favorite TV show in the upcoming week, and even SMSs you a few minutes before the show starts. You can write a script which takes a list of your friends' names and finds you the phone number for all of them. You can alert yourself when a certain movie comes to a cinema near you. You can fetch the schedule of your favorite bus without going through the bus company's graphical navigation system.

2.7 Electronic ballot stuffing

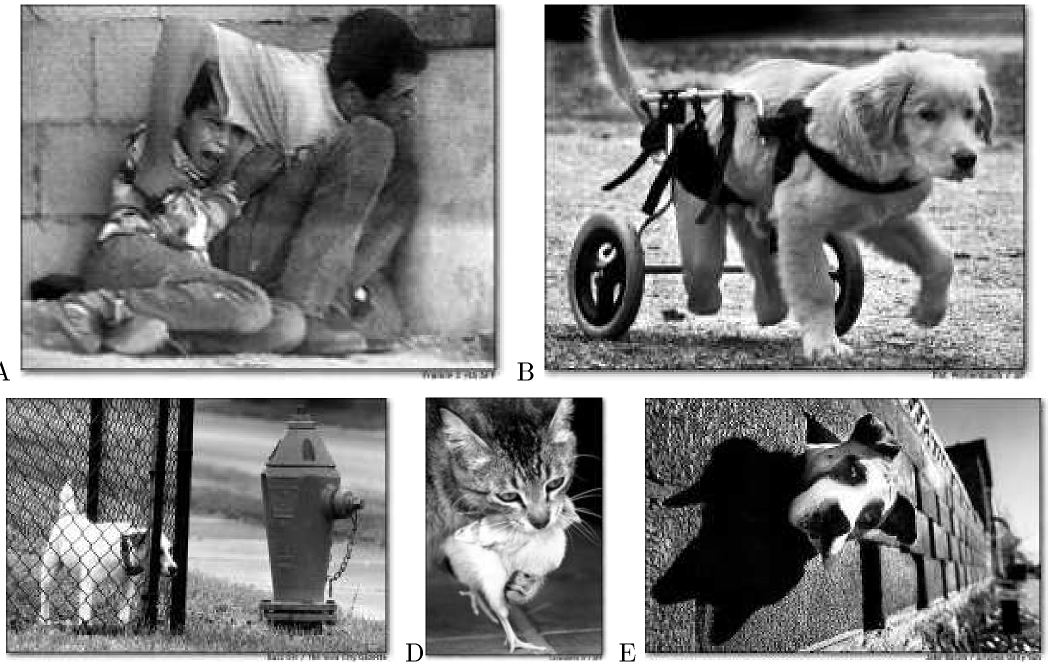
In December 2000 and early 2001, MSNBC ran on its Web site the "The Year in Pictures 2000" survey (see [8]). Each week during that year, the site's readers were asked to pick the picture of the week, and at the end of the year the readers could choose among these pictures-of-the-week, and decide which will be crowned "*picture of the year*".

This could have been an uneventful Web poll and generated results that nobody cared about, like thousands of other polls that took place that year. But this poll was special — it was to become the site of a politically-motivated cyber-battle. One of the candidate pictures, titled "*A death in Gaza*" (figure 1A), showed Jamal Al-Durrah and his 12-year-old son Mohammed hiding behind a barrel during Israeli-Palestinian clashes in the Gaza Strip on Sep. 30, 2000. Seconds after this picture, a bullet from one of the fighting sides struck the boy and killed him. Having received a lot of media attention, this was truly one of the most memorable pictures of the year.

But "A death in Gaza" was also a controversial image, because of what it meant to each side. Each side claimed that the other side's bullet killed little Mohammed, and that this was an image of the other side's brutality. The international media chose a third meaning: instead of worrying who fired the shot, it made this picture a symbol of the Palestinian misery. This made this picture into an anti-Israeli symbol.

Some Israelis became alarmed when "A death in Gaza" reached the leading position in MSNBC's survey. In the beginning of January, someone passed a chain-letter among Israeli users. The letter said that Palestinians had organized a massive voting campaign which got this picture into the lead, and that Israelis should prevent this Palestinian plan from succeeding by voting for any other picture except "A death in Gaza". The letter continued that the polling site had special security measures in place to prevent the same person from voting several times, so Israelis should pass that mail to all their friends and ask them to vote too.

But if you read the earlier sections of this article, you know that getting thousands of humans to make votes isn't really necessary. Whatever happens during a real person's vote — some pages getting viewed in sequence, and most likely some cookie being sent to prevent another vote — can be automated. A computer



A B C D E

Figure 1: Some of the MSNBC's "The Year in Pictures 2000" candidates. From top left:
A. "*A death in Gaza*", B. "*Tough Pup*", C. "*So Near, So Far*", D. "*Mother Instinct*", E. "*Peeka-bow-wow*"

can tirelessly make hundreds or thousands of votes per hour, and go on for days. Several computers working in parallel could cast votes even faster.

And indeed, this is what one Israeli did, as the media later reported (see [11, 9, 8, 5]). In about a week, several million automated votes moved "A death in Gaza" down to sixth place. Cute pictures of dogs and cats were chosen to fill the top five places, among them the pictures in figure 1B–E. Figure 1B ("*Tough Pup*") , which took the lead, shows Jake, a 4-month-old golden retriever who doesn't have hind legs, getting around his Scarborough, Maine, neighborhood in a special wheelchair. The puppy was injured when his mother rejected him as an infant and attacked him.

Computer-savvy Saudis later "fought back" and tried to move "A death in Gaza" back to first place. But by then, it was obvious to everyone, including MSNBC, that this had long ceased being a real public-opinion poll. A few weeks later, the competition was withdrawn with no winner being declared.

Since this ill-fated poll, Web site owners have come up with a number of ways to prevent certain features of their site from being used repeatedly or in an automated manner. IP-address rate-limiting prevents the same computer from casting too many votes, but also causes serious problems for networks where a large number of people sit behind one NAT or Web proxy. A better technique asks the user to perform some task that only humans can supposedly do, such as to read skewed text shown in an image. However, this technique is not perfect either — some humans cannot use it (e.g., dyslexics, the blind) and inevitably, computer software could be written to OCR that skewed text.

2.8 Bid sniping

As commerce on the Web bloomed, online stores had to find ways to differentiate themselves from other stores, and started experimenting with various business models and pricing models. One of the pricing models that was copied from the "real world" was that of an auction. The best known, and most successful auction site is Ebay [3], which operates as an online auction house, an intermediate between independent buyers and sellers which use its services as a bidding platform.

To a real auction house, you come yourself or send a proxy (a representative, or an agent) that carries out your preferred bidding strategy. Online equivalent of such human proxies have been envisioned, and often called *mobile agents*. Mobile agents are tiny programs you write that implement your bidding strategy, and are sent to the online market place to compete against and interact with other people's mobile agents, just like humans would in a real-world marketplace.

However, Ebay has no support for generic mobile agents. It basically gives you only one type of agent — one that is given a maximum bid price, and then proceeds to raise the bid every time that you are outbid, unless the current bid has already passed your maximum bid. You are allowed to change the instructions you gave your agent at any time during the bidding process (which typically lasts a few days).

Many people noticed, however, that bidding early, while your opponents still have time to react, can reveal your interest to them and allows them time to increase their maximum bid and win the item. If you only bid in the last minute of the auction, you are likely to leave your opponent with no time to react, and unable to increase their maximum bid. Because Ebay does not extend an auction's duration even if a bid is placed at the last minute, this practice, known as *bid sniping*, actually works. However, bidding manually at the last moment is hard, because that last moment often comes at a very inconvenient time (imagine having to be by the computer at 3:59 AM for the last minute of some auction, just to save a few dollars).

Ebay doesn't have mobile agents, so you can't send them one that makes its first bid only late in the game. But there is a way around this, and it is (of course) web automation. You write what is basically an non-mobile agent: a program that waits until the last minute of the auction, and only then gives Ebay the new instructions, as if a human typed them in. Your non-mobile agent could potentially be much more sophisticated: for example, it could monitor the progression of highest bids and the number of bids (Ebay doesn't show you who made which bid), as well as perhaps historic data of other auctions of similar items (which is available on Ebay), and use that information to make smarter decisions on how to bid next.

Several companies already exist which provide you with Ebay bid sniping services for a small commission. Presumably, none of them are Ebay insiders, and they all use the web-automation method, just as described here.

2.9 Virtual store building

Many e-commerce sites, such as Amazon.com, have an associate (or affiliate) program. If you are an Amazon associate, you are supposed to link to individual books (say) on Amazon's site, and if someone buys a book by following your link, you get a commission (up to 15% of the book's price).

Many associates have a Web site about a particular subject or author, and want to have on it a "virtual store" listing all of Amazon's books with that subject or author. Doing these listings by hand is tedious and soon becomes out-of-date (as Amazon modifies their inventory or prices). Instead, virtual store owners often write a program to automatically fetch from Amazon's Website the list of books with the subject or author of interest, extract the actual information from this list and generate a new page with the desired format (and all links to Amazon books changed to contain the associate's code).

As we already mentioned above, this web-automation technique suffered from a major drawback: As the look of Amazon's Web pages changed often, associates found themselves needing to modify the extraction scripts several times a year. In 2002, Amazon adopted a "Web Services" interface parallel to its normal human interface. Book details are now retrievable as XML data, and as this data is not intended for human eyes its format is stable and does not undergo periodic cosmetic "improvements". This makes this Web-Services interface much more automation-friendly than the normal (human-oriented) Web interface.

2.10 Fetching web-mail

As the Web grew in popularity, some Internet applications that used to have their own designated protocols and software started appearing as services on a Web site accessed by an ordinary browser. One good example is email — instead of using special email-access protocols like POP3 and dedicated email software, many people started using Web-mail services like Hotmail or Yahoo! Mail.

Ironically, once Web-mail services became common-place, some people wished they were more like the traditional mail protocols, and wanted to download all their mail messages, and view them locally with some email software. This is because email software often had more features and a better user interface than the Web interface of Hotmail or Yahoo! Mail.

This was done, of course, by simulating a user session that reads all the available mail using the Web interface. FetchYahoo [4] is such a program, written in Libwww-perl, for Yahoo! Mail.

3 Implementing web automation

We've seen many examples of how useful it is to write programs that pretend they are a user browsing a Web site. But how does one write such programs?

One of the most common approaches is to write such web-automation programs in Perl using the Libwww-perl module. Libwww-perl (which will be explained in detail in the next section) provides an API for fetching pages, sending forms, dealing with cookies, parsing HTML, and everything else that a Web automation program might need. Libwww-perl is very powerful, but suffers from one serious pitfall: it is relatively hard to use. The API is low-level, and actions which are easy to describe ("just submit the second form with Username filled as ...") are hard to code, and things that are obvious in a browser (like keeping cookies across page fetches) need to be coded explicitly. Many times the programmer needs to resort to reading HTML and running sniffers to understand why his automated session is not doing what a real user session did.

There are other solutions similar to Libwww-perl in spirit. Libcurl [7] is another very powerful Web client API, originally written in C, but now has bindings in over a dozen languages, including Perl. Certain things can even be done with a shell script, using the `curl` command-line utility. For some simple tasks, that is the simplest solution of all (on systems that have curl already installed).

A different approach is to run a real browser, and somehow automate it in a very high-level manner. Lynx, a textual browser, is sometimes used and automated with the Expect language. This sort of high-level automation is easier (cookies are passed automatically, redirections are followed, and so on, just like in a normal user session) but is harder to control exactly what gets done, and achieve optimal performance (e.g., not requesting unnecessary pages).

Recently, several systems have been proposed that should make it easier to automate web sessions. Some like [6] have a meta-language which describes common interaction types like logins in a simple manner. Some work by "recording" a user's real session, and replaying it later with modified parameters. But such solutions are not yet as popular, or as proven, as Libwww-perl.

4 A Libwww-perl Primer

Libwww-perl is a powerful Perl library that lets you write web clients. It provides classes and functions that can fetch pages (using protocols like HTTP and HTTPS), send forms, deal with cookies, passwords, and everything else that a Web automation program might need. Libwww-perl together with Perl's renowned flexibility are a very powerful combination, allowing one to automate the use of nearly any site.

This section will be a short introduction to Libwww-perl. It assumes prior knowledge of Perl, as well as WWW concepts like HTTP, URL, forms and cookies. It also assumes the reader already has the necessary modules installed[5].

We do not presume to explain each of libwww-perl's numerous features. Instead, we will introduce some of the most important ones, while building two simple tools as examples. Libwww-perl's comprehensive manual pages are recommended supplementary reading. Start with `LWP(3)` and continue with manual pages named after each of the classes listed in that page.

4.1 Stock prices

We will start out with a very simple task: write a program "quote" that will find the latest known price of an American stock given its ticker symbol, by using `finance.yahoo.com`. The program should give an error

[5]On modern Linux distributions, Libwww-perl comes preinstalled. If it is not installed on your system, follow the instructions on [1]; Basically this requires you to get the `libwww-perl`, `libnet`, `URL`, `MIME-Base64`, `HTML-Parser` and `Digest-MD5` packages from CPAN, and install them.

if the given symbol does not exist, or if it cannot do what it was asked (because Yahoo is not responding, or another reason). For example,

$ `quote GM`
49.21
$ `quote '^DJI'` *(Yahoo's shorthand for the Dow-Jones Industrial index)*
10,598.00
$ `quote XYZ`
quote: XYZ is not a valid ticker symbol.

Libwww-perl automation usually starts by manually browsing the site in question, assessing what login forms need to be filled, whether cookies are in used, and so on. In this case we notice that we're in luck: in order to receive the information page about General Motors Corp., whose ticker symbol is GM, all we need to do is to fetch the page "`http://finance.yahoo.com/q?s=GM`" — no cookies or POST forms were necessary. This example makes use of very few libwww-perl features, so it is a good example to start with.

We start the script checking that we indeed were given exactly one argument.

```
if($#ARGV!=0){
  print STDERR "Usage: $0 <symbol>\n";
  exit(1);
}
```

Libwww-perl is an object-oriented library. It implements classes handling requests, responses, the cookie jar (an aptly-named class for holding cookies), and so on. The first thing our program should do is to create an object of the "user agent" class, `LWP::UserAgent`. This object will be our simulated browser — it can be configured[6], and is used later for making HTTP requests.

```
use LWP::UserAgent;
my $ua = new LWP::UserAgent;
```

As we found above, we now want to fetch the page `http://finance.yahoo.com/q?s=`*symbol*. This is done by creating a request object, of type `HTTP::Request`:

```
my $request =
      HTTP::Request->new('GET', "http://finance.yahoo.com/q?s=$ARGV[0]");
my $response = $ua->request($request);
```

We created a request object (of type `HTTP::Request`) given the GET request method[7], and the URL. Then, the user-agent object was told to carry out that request. The result of this is a new object of type `HTTP::Response` that we put in the `$response` variable.

This response objects contains the HTTP response, which includes an error or success code, some headers, and content. Before trying to extract the desired information from the response's content, we should check that the request was successful (and not resulted in error or redirection). Otherwise, we just print an error message and exit:

```
if(!$response->is_success){
  print STDERR "Sorry, couldn't get $ARGV[0] page from Yahoo. The error:\n".
     $response->status_line."\n";
  exit(2);
}
```

By now, we have successfully fetched the content of the Yahoo page on the given stock, and all that remains to do is some string processing: we need to extract the stock price from the HTML content we got. At this point of the development, it is typical to temporarily add a command like

[6] for example, in order to use a proxy, or to masquerade as other browsers by choosing the User-Agent header
[7] other common request types are POST (often used for forms) and HEAD.

```
print $response->content;    ## just for debugging purposes!
```

and view this content for different input parameters. In this case we want to find a special string in the content that will tell us that an invalid ticker symbol was given, and the strings which surround the stock value when a valid ticker symbol is given. After a little experimentation, we end up with the following code:

```
if($response->content =~ /not a valid ticker symbol/){
  print STDERR "$0: $ARGV[0] is not a valid ticker symbol.\n";
  exit(3);
} elsif($response->content =~
          /(Last Trade|Index Value):(<[^>]*>)*([0-9][0-9.,]*)/){
  print "$3\n";
} else {
  print STDERR "$0: unexpected content in $ARGV[0] page from Yahoo.\n";
  print STDERR $response->content;
  exit(3);
}
```

The first case prints an error in when Yahoo told us (via an elaborate HTML page) that the ticker symbol we gave it does not exist. The second case finds in Yahoo's HTML result the number after the "Last Trade:" or "Index Value:" labels (possibly separated from that label by HTML tags), and prints that number. If the page contained neither of these cases, we complain about the unexpected result, and exit.

Note that we chose to extract the stock value from the content using somewhat coarse string-processing methods. These usually suffice, but Libwww-perl also includes full HTML parsing capabilities (which we will not demonstrate here), in the form of the `HTML::Parser` (or `HTML::TokeParser`, `HTML::PullParser`) class.

This concludes the first example of Libwww-perl. Remember that in addition to the `LWP` manual page, all the classes we just mentioned, `LWP::UserAgent`, `HTTP::Request` and `HTTP::Response` have good individual manual pages. These explain many possibilities and options that we didn't cover in this example.

4.2 Sending SMSs

Our second Libwww-perl example involves a slightly more complicated task. ICQ.com's Website allows anyone with a free ICQ account to send SMSs to mobile phone from a number of providers all over the world. Our task would be to create a script that is called as `sendsms phonenum message` that sends the given message to the given phone number. The code shown here is a simplified version of [10]. To simplify the exposition, we omitted code that repeats connections which had a temporary failure, and we assume that the phone number is in Israel (country code 972).

Automating the ICQ homepage requires a few HTTP features that we haven't seen in the previous example. We need to fetch multiple pages, fill forms, and keep cookies from page to page. We'll see how these can be done with Libwww-perl.

We begin the program by using three modules. We've already seen `LWP::UserAgent`. Additionally, `URI::Escape` is necessary for the `uri_escape` function (used for building form parameters), and `HTTP::Cookies` is necessary for handling cookies:

```
use LWP::UserAgent;
use URI::Escape;
use HTTP::Cookies;
```

We set `$phonenum` and `$message` according to the user's parameters, removing spaces and punctuation from the number, and also set the ICQ user and password:

```
die "Usage: $0 phonenum message\n" if ($#ARGV+1 != 2);
my $phonenum=$ARGV[0];
$phonenum =~ s/[ ()-]//go;
my $message=$ARGV[1];
```

```
my $user = '123456';       # These obviously need to be configured
my $password = 'paSwOrD';
```

As before, we need to create a User Agent object for making HTTP requests. In this example, we configure this user agent by asking it to pretend it is Netscape 4.73 on a Windows 95 platform (just for fun), and to use the `http_proxy` environment variable (if one exists) to choose a proxy.

```
my $ua = new LWP::UserAgent;
$ua->agent("Mozilla/4.73 [en] (Win95; I)");
$ua->env_proxy();
```

When we go to ICQ's SMS sending page (http://web.icq.com/sms/), we notice that we are asked to login by filling a form. When a user submits a form, what the browser actually does is to make an HTTP request of type POST to the URL specified by the form, with the form variables as the *content* of the request. Here is how this is done with Libwww-perl:

```
$req = new HTTP::Request POST => "http://web.icq.com/newlogin/1,,,00.html";
$req->content_type('application/x-www-form-urlencoded');
$req->content(
      "karma_user_login=".uri_escape($user, '^A-Za-z0-9')."&".
      "karma_user_passwd=".uri_escape($password, '^A-Za-z0-9')."&".
  "lang=eng&karma_product_id=21&karma_success_url=http%3A%2F%2Fweb.icq.com".
  "%2Fsms%2Finbox%2F%3Fdsfp%3D0&karma_fail_url=%2Flogin%2Flogin_page%3Fkar".
  "ma_product_css%3Dicq2go%26karma_success_url%3Dhttp%253A%252F%252Fweb%25".
  "2Eicq%252Ecom%252Fsms%252Finbox%252F%253Fdsfp%253D0%26karma_forget%3D%2".
  "6karma_service%3D&karma_service=");
$res = $ua->request($req);
```

Note that the so-called *url-encoded* content of the form looks like pairs of assignments *var=value*, separated by & characters. The variable names or the values cannot contain the actual & character, and a few other characters, so in this example we used the `uri_escape` function to appropriately "escape" any non-alphanumeric character in the username or password.

The long and ugly string that follows the username and password (`"lang=eng&`...) sets a bunch of hidden fields that were found to be present in the form. We could have fetched the login page first, and from its HTML automatically figure out the form URL and all the hidden fields. By copying the URL and hidden fields manually, like we did here, we achieved less robust solution, but a quicker one (there is one less URL to load).

If the login was successful, we expect to see a redirection to a URL containing "sms/inbox" (this string was actually given as part of the ugly hidden parameters above). In other words, we expect to see in the result the status code 301 ("moved permanently") and an appropriate "location" header:

```
if($res->code!=301 || $res->header('location') !~ m@/sms/inbox/@){
    print STDERR "Failed login to ICQ\n";
    exit 1;
}
```

But the most important thing that we got from the login is a set of cookies. We need to remember those and pass them later when we go to send the message. Cookies are held in an object of type `HTTP::Cookies`:

```
my $cookie_jar = HTTP::Cookies->new;
$cookie_jar->extract_cookies($res);
```

We continue to follow the user's interactive session, to see what request we should do next. This "detective work" can be done by using a sniffer, the browser's "show source" option, or perhaps even a tool like Mozilla's "Live HTTP Headers" extension [12]. We notice that after the login, the user fills another form containing the country, provider, phone number and message. When sending this form, we also need to send the cookies we got earlier:

```
$req = new HTTP::Request POST => "http://web.icq.com/sms/send_msg_tx/1,,,00.html";
$req->content_type('application/x-www-form-urlencoded');
$req->content("country=972&prefix=%2B972&uSend=1&charcount=".
  (160-length($message))."&".
  "carrier=".substr($phonenum,1,2)."&".
  "tophone=".substr($phonenum,3)."&".
  "msg=".uri_escape($message, '^A-Za-z0-9'));
$cookie_jar->add_cookie_header($req);
$res = $ua->request($req);
```

Like before, when the form submission was successful (and we finally sent the message), we get a redirection:

```
if($res->code!=301 || $res->header('location') !~ m@^/sms/thanks/@){
    print STDERR "Failed to send message\n";
    exit 1;
}
print STDERR "Sent successfully.\n";
```

References

[1] Gisle Aas. libwww-perl. http://lwp.linpro.no/lwp/.

[2] Vannevar Bush. As we may think. *The Atlantic Monthly*, July 1945. http://www.theatlantic.com/unbound/flashbks/computer/bushf.htm.

[3] Ebay. http://www.ebay.com/.

[4] FetchYahoo. http://livehttpheaders.mozdev.org/.

[5] Amira Howeidy. They killed al-dorra again. *Al-Ahram Weekly On-line*, March 29 2001. http://weekly.ahram.org.eg/2001/527/re4.htm.

[6] http://kamajii.sourceforge.net/.

[7] Libcurl. http://curl.haxx.se/libcurl/.

[8] MSNBC's The Year in Pictures 2000. http://www.msnbc.com/modules/ps/yip_2000/launch.asp, December 2000.

[9] Dean E. Murphy. Mideast hatreds boil up in a photo contest. *The New York Times*, March 2 2001. http://www.nytimes.com/2001/03/02/technology/02CONT.html The page http://www.loper.org/~george/trends/2001/Mar/79.html also includes an addendum from March 22.

[10] Nadav Har'El. SendSMS. http://nadav.harel.org.il/software/sendsms.

[11] The Associated Press. "media image falls victim to intifada cyberwar". *The Jerusalem Post*, March 22 2001. www.jpost.com/Editions/2001/03/22/LatestNews/LatestNews.23397.html.

[12] Daniel Savard. Live http headers. http://livehttpheaders.mozdev.org/.

[13] W3C. W3C web services activity statement. http://www.w3.org/2002/ws/Activity, January 2002.