Concurrent programming on the web with Webstream

Theodore W. Hong and Keith L. Clark
Department of Computing
Imperial College of Science, Technology, and Medicine
180 Queen's Gate, London SW7 2BZ, United Kingdom
Tel +44 20 7594-8233 / Fax +44 20 7581-8024
{twh1,klc}@doc.ic.ac.uk

December 4, 2001

Abstract

We describe Webstream, a language to simplify the development of client-side web applications, particularly web-aware information agents. Webstream encapsulates web documents as streams of messages passing between concurrent lightweight threads, permitting operations to be carried out lazy-evaluation style while documents are in the process of being retrieved. Streams can be pipelined through filters which perform transformations such as parsing raw text into HTML tags, extracting subset streams based on arbitrary computable patterns within tags or regular expression-like sequences of tags, and combining or splitting streams in various ways. These facilities allow the easy construction of a wide variety of web applications such as crawlers, meta-search engines, and information extraction agents.

Keywords: web programming, agents, information extraction
Approximate word count: 3000 words

1 Introduction

The world-wide web is usually regarded as a large collection of multimedia information resources. However, it is also increasingly becoming a rich computation environment in its own right, supporting a diverse software ecology of programs whose native domain of operation is the reading, processing, and writing of web information resources. Such programs include web crawlers, meta-search engines, comparison shopping agents, personal newspaper agents, and more. Web resources themselves have grown in scope from simple static collections of files to dynamic database queries, gateways to other applications like email or multimedia streams, and interfaces to server-side computations.

Computation on the web is similar in many ways to computation on files or objects: web resources can be read or queried and can point to one another. However, web resources also have their own special characteristics; notably that they are accessed over a wide-area network, and that they typically follow particular structural conventions. Wide-area network access entails issues of latency and reliability. It often takes a long time to send data over the network, and error conditions are common, making retrieving web resources more complex. On the other hand, many web resources are structured HTML documents, making them easier to process than free text.

Webstream is a web programming language designed to address these issues and simplify the development of web applications, particularly web-aware information agents. It is implemented as an extension layer on the April agent programming language and can be used either on its own or to add web capabilities to April agents. Programming in Webstream uses a paradigm of streams and pipelines which permits a high degree of concurrency to maximize throughput in the face of network latency. Webstream also provides a rich set of components to parse and process HTML documents, including sequence and pattern matching facilities. Furthermore, data from multiple sources can be easily merged, split, and rearranged in a variety of ways. A wide variety of web applications can be quickly written in Webstream using only a small amount of code.

In section 2, we give a general overview of the Webstream architecture, including discussion of its pipeline paradigm and some relevant features of the underlying April language. Section 3 lists the various pipeline components and composition operators in detail, while section 4 presents an annotated sample application, a multithreaded web crawler. Section 5 reviews related work, and section 6 concludes.

2 Webstream architecture

2.1 April

Webstream is built on top of the April agent programming language. April[4] is a distributed language based on a paradigm of cheap forking of a large number of mobile interacting processes that communicate via asynchronous message passing. A collection of April processes has a natural correspondence to a system of autonomous agents which can maintain private state, deliberate, move from place to place, and communicate with other agents. April processes are identified by handles which can be used to deliver messages to agents anywhere on the network. These messages can contain data ranging from atomic symbols to tuples and lists to function and procedure closures. Complex symbolic structures can be represented using arbitrarily nested tuples and lists and processed using pattern matching. Closures permit easy code mobility at any scale, from dynamic exchange of methods between objects to migration of a full agent and state between machines.

Each April process has an incoming message queue. It reads messages from the queue with a choice statement, consisting of a set of guarded commands of the form:

```
receive {
    message_pattern_1 :: test ->> action
|
    message_pattern_2...
}
```

During execution, the process tests each queued message in turn against the patterns in the choice statement. If a message matches and the optional test succeeds, the process will execute the corresponding action and consume the message. The test is a general expression which can test values in the incoming message, examine state variables, send and receive messages, or execute arbitrary code. If none of the pending messages match, they stay in the queue and the process suspends until a timeout occurs or more messages arrive, at which time the queue is examined again from the beginning.

2.2 Streams and pipelines

Webstream represents web documents as streams of messages passing between April processes. A stream is created when a data source process initiates a download of a web page. As the document's data is downloaded, the process sends out messages carrying incremental blocks of retrieved data. These message streams are operated on by filter processes that receive messages containing data, perform some transformation, and send on messages containing the results. Filters provide operations such as parsing raw text into HTML tags, extracting substreams based on arbitrary computable patterns within tags, extracting regular expression-like sequences of tags, and combining or splitting streams in various ways. Typically, several such filters or other components will be chained in series in a pipeline.

Here is an example of a Webstream pipeline:

```
get_url("http://www.yahoo.com/") |> tags() |> elems("a")
    |> head(5) |>>;
```

An application executing this pipeline will fork its components as procedure calls executing in concurrent April processes. Components are joined by the `|>` operator, which directs messages from the process on its left to the process on its right.

The pipeline begins with a `get_url` data source which retrieves data from a webserver using HTTP. The raw text stream is then parsed into a stream of HTML tags by the `tags` filter. Next, the data passes through two selection filters: `elems`, which selects tags of a given type, and `head`, which outputs the initial portion of its input. Finally, the pipeline terminates with `|>>`, which sends its output back to the parent application. (See Figure 1 for an illustration.) The overall effect of this sequence is to extract the first five anchor tags from Yahoo.

The usefulness of the pipeline paradigm is its ability to easily express arbitrary combinations of components from a standard toolbox. Further, since messaging is asynchronous, all of the components in a pipeline can execute in parallel with one another and with the page download. This permits answers to be produced at the tail of the pipeline while data is still being fed in at the head, in a manner similar to lazy evaluation. In this example, the first five anchors might be identified and sent back before the web page is completely downloaded. By contrast, most other web programming languages require documents to be downloaded in full before processing begins. Multiple pipelines downloading different pages can also execute concurrently with each other. Since web downloads often have long latencies, significant performance benefits can result.

3 Components

Webstream provides a toolbox of basic components for creating and parsing data streams, extracting tags and sequences of tags, and splitting and combining streams in various ways.

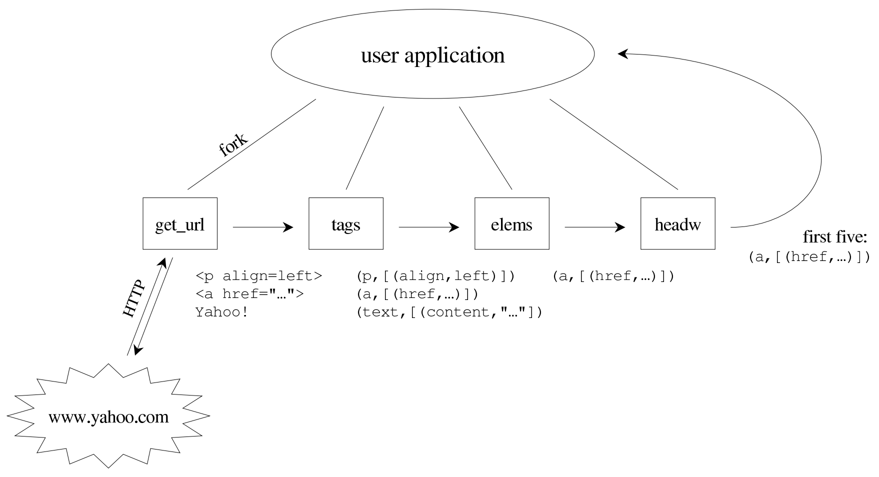

Figure 1: A sample pipeline. As the messages in the stream pass through the pipeline, they are transformed from lines of raw text to parsed tag structures consisting of pairs of tag types and attribute-value lists and then filtered twice.

3.1 Data sources

`get_url` and `post_url` create new streams from URLs by initiating an HTTP connection and requesting the document using the GET or POST methods, respectively. They take as arguments a URL to retrieve and (for `post_url`) a set of attribute-value pairs.

```
-- send GET request for New York Times front page
get_url("http://www.nytimes.com/");

-- send POST request to Yahoo stock quote service
-- set parameter "s" (stock symbol) to IBM
-- set parameter "d" (quote type) to 5-day chart
post_url("http://quote.yahoo.com/q", [("s","IBM"), ("d","5d")]);
```

Protocol-dependent processing is performed on the HTTP response to the Webstream request. For example, this might involve re-requesting data which has moved to a different URL. The retrieved text is split into chunks as it arrives and sent out in a stream of messages. Streams can also be created from local files or resumed partway through a previously interrupted download.

3.2 Parsing

`tags` parses a stream of unparsed HTML text into a stream of structures representing tags. Each tag is represented as a pair of the tag's type and a list of its attributes and their values. Free text appearing between tags is converted to a special tag type, `text`, having a single `content` attribute whose value is the corresponding literal text. A typical usage might be:

```
-- parse New York Times home page
get_url("http://www.nytimes.com/") |> tags() |>>;
```

3.3 Selection

`elems` filters a stream of tags, selecting tags matching a given type. `elemset` does the same for tags belonging to a set of types. (Note that the examples shown here and in the following subsections are fragments; for clarity, the full pipelines have been omitted.)

```
-- select anchor tags
...|> elems("a") |>...

-- select applets and images
...|> elemset(["applet", "img"]) |>...
```

`pats` selects tags matching a given pattern specified by a boolean function closure. This function is applied to each incoming tag, and those yielding `true` are selected. Any April code fragment returning a boolean value can be used, which might perform arbitrary computations, send and receive messages, or even spawn subsidiary pipelines. For example, `pats` might used in conjunction with `elems` and an appropriate function closure to select the broken links in a document:

```
-- select broken links
...|> elems("a") |> pats(broken) |>...
```

where `broken` is a function defined separately that extracts URLs from anchor tags and attempts to retrieve them using a second pipeline.

head, drop, and tail select the first N, all but the first N, or the last N tags, respectively:

```
-- skip the first 10 tags
...|> drop(10) |>...
```

3.4 Sequence extraction

grep searches a stream for contiguous non-overlapping sets of tags matching a given regular expression on the alphabet of tag types. This filter is particularly useful for writing wrappers to extract information from web pages. For example, the following pipeline:

```
-- select table columns containing text with bold tags
...|> grep("td text (b text /b text)* /td") |>...
```

would extract sequences such as:

```
<td>DarWilliams.com - the green version</td>
<td>A Little <b>Dar</b> <b>Williams</b> Page</td>
<td>Since 1995, <b>Dar</b> Web has been the flagship...</td>
```

Such sequences might appear in search results when search terms (here "Dar Williams") are highlighted. Performing this type of sequence matching is a key part of information extraction applications like comparison-shopping agents or meta-search engines.

3.5 Composition

Several operators are provided to compose pipelines together. The or operator combines two pipelines using OR-parallel semantics. In OR-parallelism, the composite pipeline returns the data from whichever subpipeline succeeds first. This behavior is useful when the sources queried provide redundant mirrors of the data wanted and either will do.

```
-- choose fastest mirror site (US or Ireland)
get_url("http://www.cpan.org/") or
    get_url("http://cpan.indigo.ie/") |>>;
```

The resulting pipeline waits for one of its input streams to start sending data and then forwards that stream to its output, terminating the other.

AND-parallel semantics can be obtained by using sequential pipeline statements. In AND-parallelism, the composite pipeline merges the data from all of its subpipelines. This behavior is useful when the sources queried provide different subsets of the data and their combination is wanted. Since pipeline statements return immediately after forking the appropriate processes, a sequence of such statements will execute in parallel and all outputs will be merged in the incoming message queue of the main process.

```
    -- conduct parallel meta-search queries
    get_url("http://www.google.com/search?q=april") |>>;
    get_url("http://www.altavista.com/cgi-bin/query?q=april") |>>;
```

`union`, `intersect`, and `diff` merge pipelines according to the usual set semantics. That is, `union` merges two pipelines with AND-parallelism and removes duplicates. `intersect` selects messages which are received from both pipelines, while `diff` operator selects those sent by its left-hand argument but not its right-hand argument.

```
    -- merge two bookmark files
    (get_file("bkmark1.htm") |> tags() |> elems("a"))
        union (get_file("bkmark2.htm") |> tags() |> elems("a")) |>>;
```

Streams can also be split by giving a list of pipelines after a `|>` operator. This causes each incoming message from the left-hand side to be multicast to all of the pipelines on the right-hand side.

```
    -- select first and last 10 tags
    ...|> [head(10) |>>, tail(10) |>>];
```

4 Sample application: multithreaded web crawler

To give a flavor of what Webstream programs look like, we give an annotated partial listing of a sample application, a multithreaded web crawler. (The listing has been somewhat simplified to clarify the presentation.)

The program takes three parameters: an initial URL, a maximum number of pages to visit, and a maximum number of pipelines allowed to be active at any one time (to limit resource consumption). It begins by initializing its counters and starting a pipeline to extract the anchor tags from the initial web page.

```
program {
    main(initurl, maxpages, maxpipes) {
        pages: 1;
        pipelines: 1;

        get_url(initurl) |> tags() |> elems("a") |>>;
```

While there are pipelines still executing, the program loops over a message receive statement.

```
        while pipelines > 0 do {
            receive {
```

An incoming tag is matched by the pattern `wsio_tag(T)`, where `T` is a variable that becomes bound to the tag. If the maximum number of pages has not been reached and the number of active pipelines is below the limit, the crawler prints the link, starts a new pipeline to follow it, and increments the page and pipeline counters.

```
                wsio_tag(T) :: pages < maxpages
                    && pipelines < maxpipes ->> {

                    new_url = extract_href(T);
                    new_url ++ "\n" >> stdout;
                    get_url(new_url) |> tags() |> elems("a") |>>;

                    pages := pages + 1;
                    pipelines := pipelines + 1;
                }
```

The pipeline counter is decremented when a pipeline finishes, sending a `wsio_end` message.

```
              | wsio_end ->> {
                    pipelines := pipelines - 1;
                }
```

If a tag arrives but the page limit has been reached, the program just prints the link without starting a new pipeline.

```
              | wsio_tag(T) :: pages >= maxpages ->> {
                    new_url = extract_href(T);
                    new_url ++ "\n" >> stdout;
                }
```

Finally, if the page limit has not been reached but there are too many pipelines executing already, the message will not satisfy any of the message receive tests and will be left in the queue. The main process then suspends until the next message arrives, at which time it will recheck its message queue from the beginning. When a pipeline slot becomes available, the deferred link will be picked up and processed.

```
            }
        }
    }
} execute main;
```

The crawler terminates when the specified maximum number of pages have been retrieved or no more links can be found and all pipelines are finished.

During execution, many pipelines will be concurrently downloading different pages and extracting their links. The outputs of these pipelines are all merged in AND-parallel in the crawler's message queue, as described in section 3.5. This means that the crawler does not have to wait for any single request to complete, but can eagerly follow links from any of the active requests as they progress, reducing the time spent waiting for dead links. The resulting crawl can be regarded as as a breadth-first traversal where links from pages closer to the user in terms of latency are followed first.

5 Related work

One of the first descriptions of the web as a computation environment was given by Cardelli and Davies[3], who described some of the features of the web computing model and the characteristics that a web programming language might need. They introduced the notion of *service combinators*, operators that can be used to build complex control structures from primitive page retrieval actions. Combinators include sequential or concurrent execution, timeout, and repetition, and can be used to model various web browsing activities and strategies. Webstream's pipeline composition operators perform an analogous function.

The service combinator concept was implemented by WebL[6], a Java-based scripting language which has similar capabilities to Webstream. WebL also provides a *markup algebra* for processing retrieved web pages. Operators in the markup algebra create and manipulate *pieces*, contiguous regions of text within a page. For example, all the tables or all the occurrences of some fixed sequence could be turned into piece-sets. Other operators create derived sets, such as the set of pieces in one set which completely contain pieces in another set. Webstream's filter operators have greater expressive power and can specify a wider range of sets.

Another language for processing web pages is Editor[1], a text manipulation language which uses word processor-like "cut and paste" operations to rewrite pages. Editor programs can act as wrappers for web pages, providing extraction methods that return various values sifted from within HTML source code. This is used as the basis for applying a relational database abstraction to the web in Araneus[2]. The language is quite low-level, oriented towards searching and replacing individual tokens, and programs in it are rather cumbersome. Neither Editor nor WebL provide concurrency between downloading and processing, requiring instead that documents be completely retrieved before processing begins.

LogicWeb[5, 7] is an interesting effort to apply a declarative, rather than procedural, abstraction to the web. Web pages are retrieved using the `download/4` logical predicate, which incrementally binds one of its arguments to a term representing the data as it arrives. By combining `download/4` with concurrent logic programming constructs, activities such as switching a request to a mirror site or repeating a request until it succeeds can be expressed. Once downloaded, a page is converted into a small logic program that is a collection of clauses representing facts about it, such as its title or a list of its links, plus any clauses explicitly added by the page's author using special LogicWeb markup tags. These facts can then be used as a knowledge base context for satisfying goals, for example to search for a page on a given subject. However, the conversion predicate is limited to enumerating tags of different types (links, images, etc.) and does not provide support for general parsing.

Web libraries have also been developed for existing general-purpose languages. Compared to these libraries, Webstream generally provides a more concise and higher-level abstraction over web programming operations. For example, a C program using the `libwww` library takes dozens of lines just to initialize the interface and download one web page, where Webstream needs only a single pipeline statement. The web crawler program described in section 4 takes about 40 lines of code, excluding comments and blank lines, while a comparable Java program that performs the same task using the `java.net` and `javax.swing.text.html` libraries takes around 100 lines and requires careful synchronization coding. More importantly, these libraries do not have the capacity for agent programming integration that Webstream does, or the support for sequence and pattern matching on tags needed by information extraction applications.

6 Conclusion

Webstream is a new programming language intended to simplify the development of client-side web applications. It is designed to address the special characteristics of computation on the web such as network access and structural conventions. Webstream's concurrency and lazy pipeline evaluation greatly increase the efficiency of web applications such as crawlers and meta-search engines. It also provides a rich set of operators for parsing, transforming, and extracting data from web documents that are useful for information extraction applications. Using the pipeline concept, these operators can be arbitrarily composed in simple yet powerful ways. Combined with its integration with the April agent programming language, Webstream provides a useful platform for easily constructing a wide variety of web applications.

References

[1] P. Atzeni and G. Mecca. Cut and paste. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 144–153. ACM Press, 1997.

[2] P. Atzeni, G. Mecca, and P. Merialdo. Semistructured and structured data in the web; going back and forth. *SIGMOD Record*, 26(4):16+, Dec. 1997.

[3] L. Cardelli and R. Davies. Service combinators for web computing. Research Report 148, DEC SRC, June 1997.

[4] K. L. Clark and F. G. McCabe. April—agent process interaction language. In M. Wooldridge and N. Jennings, editors, *Intelligent Agents*, pages 324–340, Berlin, 1995. LNAI 890, Springer-Verlag.

[5] A. Davison and S. W. Loke. A concurrent logic programming model of the web. Technical Report 98/28, Dept. of Computer Engineering, Prince of Songkla University, 1998.

[6] T. Kistler and H. Marais. WebL – a programming language for the web. *Computer Networks and ISDN Systems*, 30:259–270, 1998.

[7] S. W. Loke and A. Davison. LogicWeb: Enhancing the web with logic programming. *The Journal of Logic Programming*, 36:195–240, 1998.