The following paper was originally published in the
Proceedings of the First USENIX Workshop on Electronic Commerce
New York, New York, July 1995.

Generic Extensions of WWW Browsers

Ralf Hauser and Michael Steiner
IBM Research Division
Research Laboratory, Rueschlikon, Switzerland

Generic Extensions of WWW Browsers

Ralf Hauser Michael Steiner
Information Technology Solutions Department, IBM Research Division
Zürich Research Laboratory, CH-8803 Rüschlikon, Switzerland
tel: +41.1.724-8426, fax: +41.1.710-3608, email: {rah, sti}@zurich.ibm.com

Extended Abstract

August 10, 1995

Abstract

Current WWW browsers provide two main services: communication and information rendering. While this is sufficient for some purposes, many future applications will need more sophisticated processing on the user side before server-data can be presented to the user or before the user input can be transferred to the servers. For example, electronic payments ought to be seamlessly integrated into a customer's browser to enable him to shop via the Web. This paper proposes two pragmatic approaches to fulfill these requirements without altering current browser technology. We conclude with the proposal of a generalized extension framework for WWW browsers.

1 Introduction

The World Wide Web (WWW) has grown extremely fast in the past months, not only quantitatively but also qualitatively. New services are being added on a day-by-day basis and range from telephone directories to movie databases and "shopping malls". In particular, the "POST"-Method of HTTP [1] and the "FORMS"-feature of HTML[2] have made the WWW technology suitable to most Internet applications.

With new commercial applications emerging daily, it is impossible to anticipate all future needs and build a single browser that fulfills all requirements. One example for this are security requirements: the simple authentication mechanisms of early HTTP versions are no longer sufficient. Advanced services such as secure payments and intellectual property rights protection require new mechanisms based on multi-party security.

Therefore, browsers should provide interfaces to allow the easy addition of arbitrary extensions. Most current browsers support the concept of *external viewers* based on Internet Media (MIME) Types [3]. The users of a WWW browser can freely configure their preferred viewers, enabling them to receive an open-ended set of document types with this feature. However, permitting only external viewers as extensions is not sufficient. Services such as confidentiality, integrity protection and packetizing data are only intermediate processing steps. They are *filters* and need to return output back to the browser controlling the overall application run.

WWW-browsers are also proposed as exclusive user interfaces[1] or as HTTP engines. The browser is then the slave of a different application. This cannot be handled by the MIME extension feature, and the different browser implementations address this issue of "remote control" in different ways.

In the next section we shows how rudimentary filter extensions can be implemented with existing technology and virtually any browser without modifications of either source or object code. We use the example of a secure purchase and payment of a digital movie to illustrate the approaches. In section 3 we outline a generic framework that accommodates both filters as well as remote control.

[1] Shah [4] envisions using the WWW system as operating environment and the browsers as generic user interfaces also to other applications.

2 Short-Term Approaches

For short- and mid-term solutions, an extension mechanism must not require changes of the browsers but should depend only on the features implemented by the majority of available browsers.

We propose two pragmatic approaches with minimal dependencies on implementation specifics to "snap-in" a client-side extension into today's browsers . The first approach relies only on external viewers that use a special MIME type, and the second approach is based on a small daemon resembling the HTTP server (`httpd`) running on the user's local machine.

As an example to demonstrate our approaches we take the scenario that a User likes to purchase a movie from a server using a secure payment scheme (e.g *iKP*[5]).

2.1 External Viewer Approach

Figure 1 shows the purchase of a movie based on external viewers. This minimal approach consists of

1. a payment "extension" MIME viewer entailing
   - user interface routines to obtain the user's confirmation to join the purchase contract, and the
   - cryptographic mechanisms to build a secure digital credit cart slip.
2. a method to "re-integrate" the extension's output into the WWW browser.

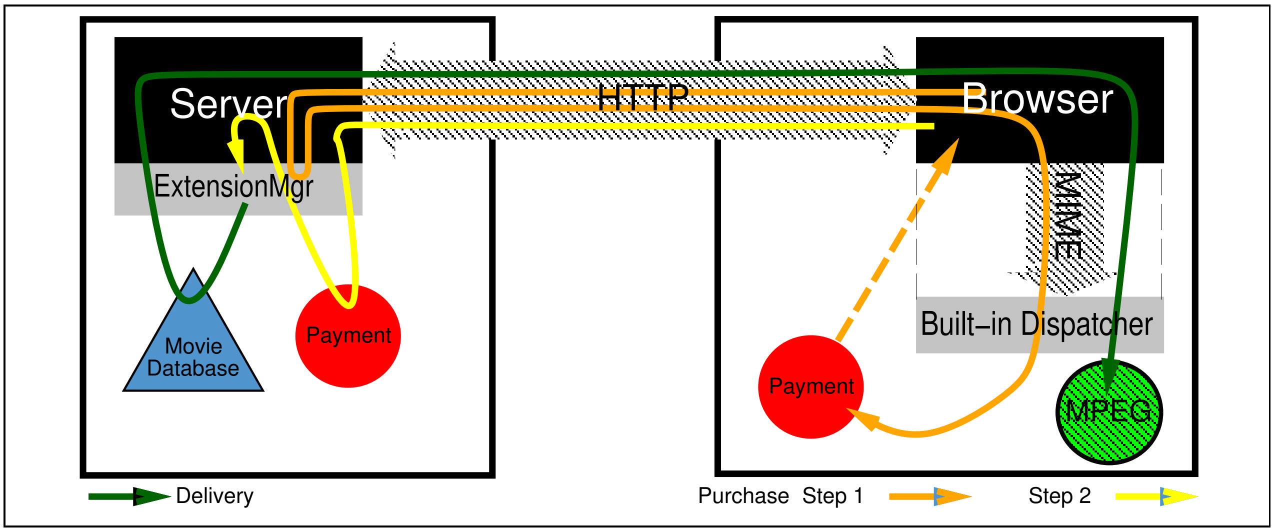

Figure 1: Payment of a Movie based on external viewers

Re 1: Whenever some input requires further processing on the user side, the server tags it with a specific MIME-type and submits it to the browser[2]. In our example of the purchase of a digital movie, the browser transparently passes this information to a "payment" extension in the form of a MIME viewer. The extension then displays contract parameters such as duration, number of frames per second, resolution, and price. Then it prompts the user for a confirmation. For small amounts, this confirmation can be provided automatically by the user's device. For larger amounts, the payment extension's policy will require the user to explicitly enter a PIN or a password to unlock the user's credit card for the generation of a digital credit-card slip.

Re 2: Such a re-integration of the information produced for a payment meta-protocol aims to feed data back to the browser or at employing the HTTP communication infrastructure that is already in place from

[2]For example, the user has filled out an order form and passed it to the shop's server. A first plausibility check, for example, whether the required amount is on stock or whether it requires extra freight negotiations, can already be performed by the server before turning the information of this flow into the first flow of the payment protocol under the pertinent special MIME type.

the browser to the server. In this way the payment extensions to not have to care about communication and can be unaware of potential gateways to cross on the way to the merchant. The first version of remote control features of both Mosaic[3] and Netscape[4] does not permit external applications to transmit more information transparently to a server via a browser than can be fitted into a URL. Most implementations expect this URL string to be only a few hundred characters long. For many situations this length is insufficient (e.g., with RSA a simple signed or encrypted block would already require 170 bytes).

One intermediate solution is to include the digital slip in a hidden input field of an HTML form that can be arbitrarily long. The entire html document containing this form is then deposited in a local file with a standard name. This file can be brought back into the browser in two ways:

- The last action of the payment extension is to cause the browser to load this local file by the aforementioned remote-control features. The user must then click *one* more time to send the form (containing the slip) to the server.
- If the browser entails no remote control feature, the server already provides a link to the standard local filename when supplying the user with the last information rendered by the HTML viewer. In this case, the last action of the browser is to inform the user that the local file is ready and the pertinent link can be clicked. This approach even requires *two extra user interactions* which will be unnecessary in implementations of long-term frameworks.

Evaluation

Such a short-term solution allows one to "**snap**" almost arbitrary extensions "**in**" to existing browsers. The requirements for the communication and operating environments are minimal. The solution, however, has two drawbacks. First, the filtering process sporadically needs user input to continue operation - input that is unnecessary from a logical point of view. Second, post-processing filters (i.e. filters that are employed after the last logically necessary user input) require an extra communication exchange with the server because the MIME dispatcher of the browser only operates on incoming data. This extra communication exchange is not possible with certain extensions[5].

2.2 Client-Side HTTP Daemon

The second short-term approach is to have an `httpd`-like daemon on the user's machine as shown in Figure 2. For security reasons, this extension `httpd` only accepts connections on a well-defined port from browsers on the same machine. When the payment for an electronic purchase is about to take place, the user clicks on a URL (provided by the shop's server) and causes a normal *POST* request to be sent to this local `httpd` (e.g.: `http://localhost:54321/buy`). The body of this request contains the information to assemble the digital credit card slip. To obtain further user input (such as credit card numbers or the sensitive PIN for transaction approval) this extension `httpd` either directly pops up with a special window for user keyboard input, or collects this information through the browser by sending back an appropriate HTML form.

There are two approaches for the extension `httpd` to transmit the final order with the digital slip to the server:

1. It puts the complete order information into a URL addressed to the shop's server. It then sends this URL with a *HTTP Redirect directive* to the browser. The browser therefore automatically issues a normal HTTP request with that URL to the server of the shop.
2. It acts as a *Proxy*; therefore, it contacts the shop's server directly and forwards the shop's response to the browser.

The main problem with this approach is how the shop's server can find out the port number of the extension `httpd` to construct the mentioned local URL. The answer is three-fold:

---

[3] http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/remote-control.html

[4] http://home.netscape.com/info/APIs/x-remote.html

[5] For example, communication-oriented post-processing such as packetizing bulk data, because the `GET` and `POST` method remain the only available communication primitives.

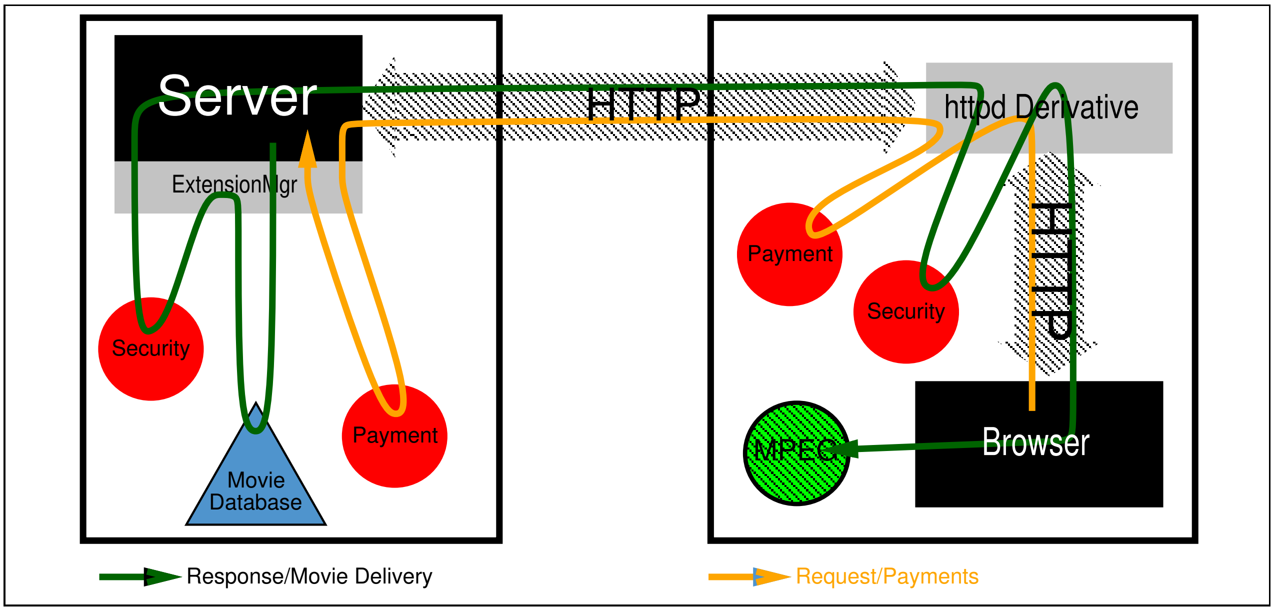

Figure 2: Payment with a Client-Side Extension HTTP-Daemon acting as a Proxy

- A well-known port will be established for the most common case.
- If the user cannot use this port for any reason, the port will be published as a special MIME-type of the form `extension-mgr/port-number`, where port-number stands for the port actually used. This will be sent with the HTTP *ACCEPT* header-field to the server and can be used to generate the URLs.
- If all this fails, the user will have to manually fill out a form that explicitly asks for the extension's port number, and then send it to the server.

Evaluation

The advantage is that this approach requires no extra user interaction to keep the filtering process alive. Additionally state can be kept easily between different communication exchanges.

Order transmission approach (1) has the advantage that the extension `httpd` only interacts with the user's browser and, therefore, can be totally shielded from outside communication. On the other hand we have to consider the limitation mentioned before of the amount of information that can be fitted into a URL.

Order transmission approach (2) does not suffer from such a size restriction. It can additionally react on faults as timeouts (e.g by sending retries) and pre-process the response of the shop's server. Figure 2 shows how the security extension could provide copyright protection by decrypting the delivered digital movie. The main disadvantage of this approach is that the daemon must be aware of proxies or gateways (e.g SOCKS) to always be able to reach the shop.

3 General Client Side Extension Framework

The problems of our approaches presented in the last section led us to look for a long term solution. This solution should also support certain extensions as load balancing by redirecting certain URLs to local files or geographically closer servers and outlining of document structures ([6]) cannot be handled easily by the approaches mentioned before. We can relax our requirement of not changing browsers. Therefore we propose a change in the architecture of browsers.

In the currently popular WWW browsers, the HTML viewer, the HTTP communications module, and a primitive "MIME document dispatcher" are combined in one monolithic package. Some implementations add some limited capabilities of handling remote control events as well.

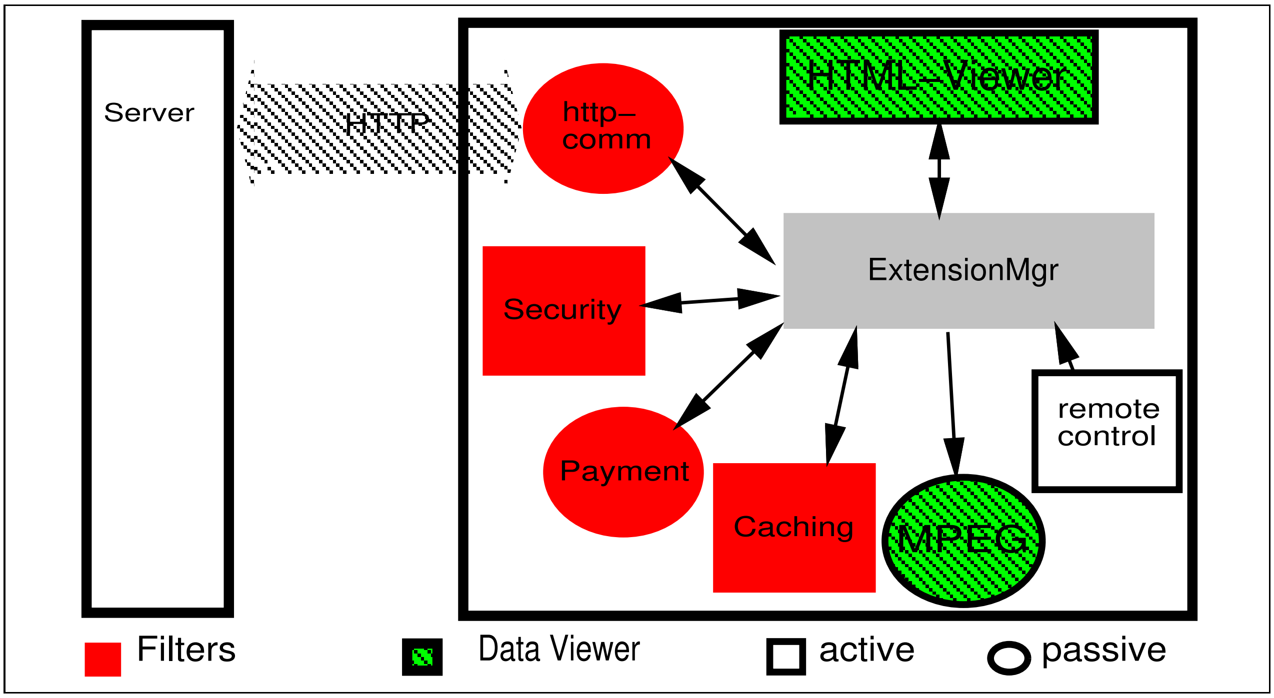

Figure 3: Examples of current Extensions

This architecture can be greatly simplified and its flexibility can be increased if we view the browser as a generic *extension manager* who acts merely as a dispatcher. Not only MIME extensions, but also the communication toolbox and the HTML viewer are subsumed as extensions.

This results in an architecture with only three logical components:

- the **extension manager**,
- **passive** extensions.
- **active** extensions.

Passive extensions are called by the extension manager. Their purpose is to either filter and transform data, to serve as gateways to different protocols or to render data.

Active extensions control the execution of the extension manager and initiate "web-actions". Note that the two types of extensions are not mutually exclusive. The HTML-viewer is clearly a data viewer but when it reacts on user interaction and issues a request it can also be considered as an active extension.

This leads to the following sketch of a minimal interface defining the interaction between the extension manager and extensions/external applications.

The extension manager exports two functions for active extensions. Parameters named `request` or `response` are required to be valid HTTP requests rsp. responses:

- **ProcessURL**(
  ```
  IN:  request
  IN:  returnMeResponse?
  OUT: [response if returnMeResponse?]
  ),
  ```
- **RenderMIME**(
  ```
  IN:  response
  IN:  associated URL
  ),
  ```

The extension has similarly to provide one or both of the following two functions:

- **HandleURL**(
  ```
  IN:  request
  OUT: requestOrResponse
  ),
  ```
- **HandleMIME**(
  ```
  IN:  response
  OUT: [requestOrResponse if not(IsRenderer?)]
  ),
  ```

These functions are registered either statically with a configuration file or dynamically by calling specific registration functions of the extension manager. Together with the callback function a regular expression is registered. The extension manager tries it to match with requested URLs rsp. MIME-types to find out which extension to take. If matching extensions are found, the extension manager will take the best match and call the associated callback-function.

Once the call to a filter returns, the extension manager inspects `requestOrResponse` and depending on the type of the return value looks for the next MIME or URL filter to call. Data viewer do not return any data and the extension manager stops processing.

If an external application wants the resulting data of a `ProcessURL` [6] call regardless of its MIME type, it will set `returnMeResponse?` in the `ProcessURL` call. Instead of handing the final response to a data viewer according to the MIME-type matching rules the extension manager will return it to the caller.

3.1 Example

Our example begins with a user who has already browsed the price list of a movie-on-demand WWW shop and has made a choice. Therefore, the HTML viewer knows the URL of the selected digital movie as well as the desired payment protocol. The user's click is processed in the following steps:

1. The HTML viewer will initiate the operation by calling the extension manager with:
   ```
   ProcessURL(
     ``POST 3KP://movie.com/ HTTP/1.0

      Merchandise=Entertainment/movie324.mpg,
      Value=$5,
      Shop-Name=MovieComInc,
      Shop-Certificate=...``,
      FALSE
      returnDataHandle
   ).
   ```
2. The extension manager looks for a registered extension and will match "$\hat{3}$kp:*". It will call the payment extension then with:
   ```
   HandleURL(
     ``POST 3KP://movie.com/ HTTP/1.0

      Merchandise=Entertainment/movie324.mpg,
      Value=$5,
      Shop-Name=MovieComInc,
      Shop-Certificate=...``,
      returnDataHandle
   ).
   ```
   The payment extension will generate a payment token and puts the following HTTP request in `returnDataHandle`:

[6] `ProcessURL` together with `RenderMIME` replace the current "remote-control" features of some browsers.

```
''POST http://movie.com/3kp-buy HTTP/1.0

 Merchandise=Entertainment/movie324.mpg,
 PayToken="XYZ" ''
```

Note that the payment extension may actively call the extension manager during its operation through the `ProcessURL` interface, for example, to retrieve some other party's public key.

3. Inspecting `returnDataHandle` the extension manager determines that it should call the HTTP communications module[7]. This module handles the HTTP request and returns the HTTP response with the requested document from the remote server in `requestOrResponse`:

```
''HTTP/1.0 200 OK
 Date:  Friday, 23-Jun-95 13:52:57 GMT
 MIME-version:  1.0
 Content-type:  application/copyright_protect

 ..  encrypted data ...''
```

4. Because the movie-on-demand shop has some copyright protection in place, the responses MIME type will match the copyright protection extension and call the `HandleMIME` callback corresponding to this extension. In a simple case, this callback would decrypt the data with a session key and return the movie with MIME type MPEG.

5. Finally, the extension manager will hand the movie324.mpg data to the movie viewer (e.g. an MPEG viewer).

3.2 Remarks

The HTTP communications module need not be the only one capable of changing the parameters from HTTP requests to HTTP responses. A caching filter returns the document if it is already in the cache, and an access filter for a browser at a public place will return a document containing an error message if it refuses some requests that may infer high costs or are against the charter for which the terminal was provided. Protocol gateways (e.g., ftp, gopher, mailto) are also examples of filters that semantically convert HTTP requests into HTTP responses.

Response-side filters may also issue new requests, for example, for pre-fetching of referenced documents.

Integrity protection, confidentiality, and packetizing data for long messages require symmetrical filtering also on the server side. The Common Gateway Interface (CGI) [7] allows filtering on the server side but unfortunately, this interface only foresees one filtering step. To obtain multiple filtering steps, an extension manager can be placed within this single filter and emulate the flexibility and configurability of the client-side extension manager.

It is hoped that such a framework with standardized extension manager semantics and interfaces will allow arbitrary developers to offer their own, compatible and powerful extensions.

4 Comparison with Existing Approaches

In the final paper this section will discusses the current proposals, namely:

- Kristol's RFC for a HTTP extension mechanism [8],
- NCSA's *Common Client Interface (CCI)* [9], already available as a prototype,
- Spyglass's *Software Development Interface* [10],

[7] We call HTTP communications module the piece of code that implements the http semantics on a specific network infrastructure. Most current WWW browsers assume TCP/IP to be the infrastructure - in the future, this may change.

- W&A's API for WWW Applets[11],
- others.

These proposals are compared with our framework, and their shortcomings are analyzed.

5 Conclusions

This paper has presented two short-term solutions of how to **snap** almost arbitrary extensions **in**to current WWW browsers without altering their source code or executables. Our example illustrates how these two solutions enable electronic payments with acceptable user interfaces and highest security already with existing browser technology. We have built two prototypes that provided proof of these concepts.

The requirements for extra user input, which is logically not justified, or for peculiar "internal" browser inter-component communication is a sign that the technology originally was not designed for the use described here.

Therefore we have stated our own view of a sound architecture for flexibly configurable and composable WWW browsers of the future.

References

[1] Tim Berners-Lee, R. T. Fielding, and H. Frystyk Nielsen. *Hypertext Transfer Protocol*, March 1995. Internet Draft, Expires September 8, 1995.

[2] HyperText Markup Language Specification - 2.0. Internet Draft, February 1995. Expires June 19, 1995.

[3] J. Postel. Media type registration procedure. Internet Request for Comment RFC 1590, March 1994.

[4] Rawn Shah. The World Wide Web System as an Operating Environment. http://www.rtd.com/people/rawn/os-paper.html, 1994.

[5] Mihir Bellare, Juan Garay, Ralf Hauser, Amir Herzberg, Hugo Krawczyk, Michael Steiner, Gene Tsudik, and Michael Waidner. iKP – a family of secure electronic payment protocols. In *First USENIX Workshop on Electronic Commerce*. USENIX, July 1995.

[6] Alan F. Slater. Extending W3 clients. In *Second International Conference on the World-Wide Web*, pages 899–908, Chicago, October 1994.

[7] Rob McCool. *The Common Gateway Interface*. NCSA, 1.1 edition, 1994.

[8] David M. Kristol. A proposed extension mechanism for HTTP. Internet Request for Comment RFC, January 1995.

[9] NCSA Mosaic Common Client Interface, September 1994. Version 1.0, http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/cci-spec.html.

[10] Paul Rohr. Software Development Interface, December 1994. http://www.spyglass.com/techreport/iapi.htm.

[11] Bert Bos. W&A an API for WWW applets, v1.1. http://www.let.rug.nl/ bert/W3A/W3A.html, February 1995.