FREDDY: A Web Browser-friendly Lightweight Data-Interchange Method Suitable for Composing Continuous Data Streams

Shohei Yokoyama[1], Isao Kojima[2], and Hiroshi Ishikawa[1]

[1] Shizuoka University, Japan
[2] National Institute of Advanced Industrial Science and Technology, Japan

Abstract. As a remarkable lightweight data-interchange format for use with web browsers, JSON is well known. Recently, web browsers have come to support rich applications called Software as a Service (SaaS) and Cloud Computing. Consequently, data interchange between web servers and web browsers is an important issue. A singleton, an array, or a nested object (tree) can be represented by JSON, which is based on a subset of the JavaScript Programming Language. It is valuable for SaaS applications because JavaScript programs can parse JSON data without the need for special programs. However, web browsers and JSON are poorly designed to address large amounts of data and continuous data streams, e.g. sensing data and real time data. We propose here a novel data format and a data-interchange mechanism named "FREDDY" to address this deficiency. It is not merely a subset of the JavaScript; it can represent semi-structured data, just as JSON does. Moreover, FREDDY is suitable for composing a continuous data stream on web browsers. Using the small JavaScript library of FREDDY, web applications can access streaming data via a SAX-style API; it works on all major browsers. Herein, we explain FREDDY and evaluate the throughput of our implementation. We loaded 400 MByte streaming data using our 5 kByte library of FREDDY.

1 Introduction

Once it was believed that web browsers were useful merely to display static pages that web servers would provide one after another. However web pages are no longer static, as exemplified by Google Maps, which uses dynamic HTML and Asynchronous JavaScript + XML (Ajax)[15]. Using JavaScript, web pages can access Web servers, download data, and update pages themselves. This technique has laid the foundations for next-generation web applications such as SaaS[16], Web2.0[13], and cloud computing. In this case, the main logic of applications is partially located on a client side and partially located on a server side. We will address implementations of web applications with the question of how data that applications need can be accommodated. JavaScript applications on web browsers always run inside a security sandbox. Consequently, all data must be either on primary storage (main memory) of a local computer or on secondary storage (hard disks) of web servers, which is not of the local computer. That is to say, data interchange between a web server and a web browser is an extremely important issue for web applications.

Ajax and JavaScript Object Notation (JSON: RFC4627)[5] are attractive solutions for interchange of data between web servers and web browsers. However, the combination of Ajax and JSON can download only a chunk of data simultaneously. They cannot handle continuous data streams, e.g. sensing data and real time data. Numerous sensors are installed in devices from toilets to satellites, but no lightweight integration method exists for handling sensing data on the Web.

The purpose of this paper is to propose a lightweight data stream interchange mechanism named FREDDY. Our implementation provides a *Simple API for XML (SAX)*[3] style programming. Therefore, FREDDY can accommodate not only a continuous data stream, but also semi-structured data, equivalently to XML. Using FREDDY, users can accommodate continuous data streams via a SAX event handler, which is written in JavaScript.

In this study, we also evaluate the throughput of our implementation of FREDDY when the application loads a large SAX event stream. The results of experiments show good throughput, about 1 MByte/s, of data interchange between a web server and a web browser. However, we do not emphasize the velocity of data interchange; also, FREDDY is a lightweight method in terms of the security sandbox of web browsers. The remainder of this paper is organized as follows. Section 2 expresses an overview of FREDDY. The data format and our implementations of FREDDY are described in Section 3 and Section 4. In Section 5, we present our experiments and evaluations. In Section 6, we describe related works. Finally, Section 7 concludes the paper.

2 Overview of FREDDY

The main contributions (Fig. 1) of this paper are: (a) a lightweight data format that is suitable for streaming data exchange using only JavaScript, (b) a streaming delivery mechanism on the Web, (c) a lightweight library, whose size is about 5 kByte, to realize SAX-style programming for handling both semi-structured data and continuous data streams.

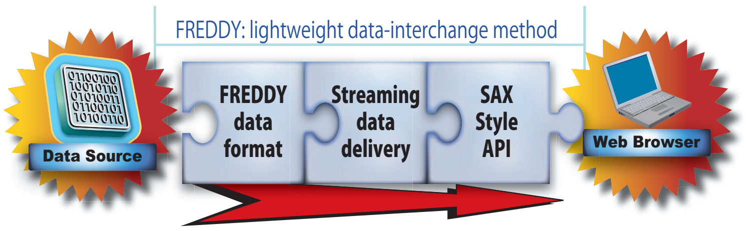


Fig. 1. Software components and data flow of FREDDY and JSON.

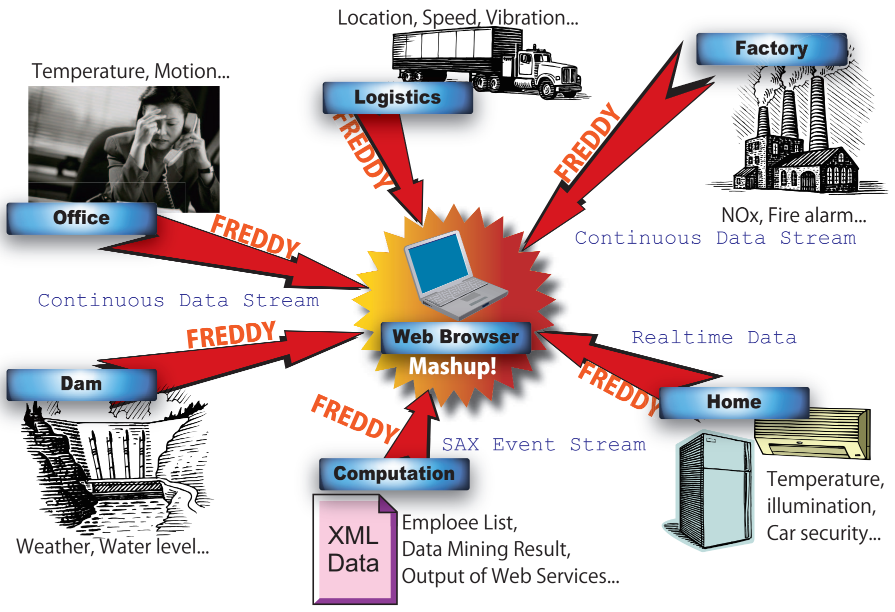


Fig. 2. Our Goal, Stream Data Mashups using FREDDY.

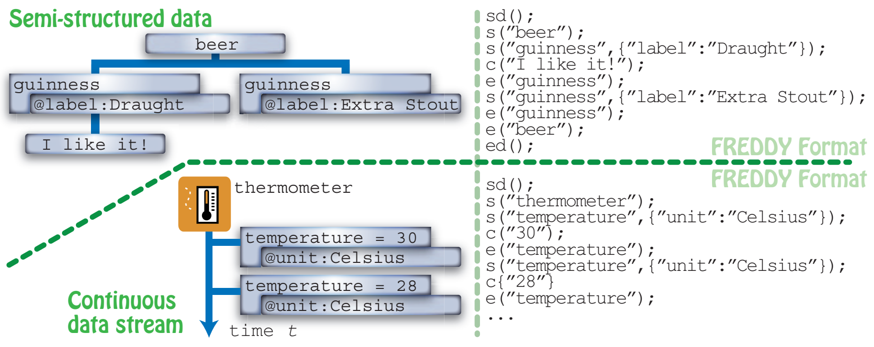
Figure 2 presents the goal of our research. FREDDY provides a lightweight JavaScript API that is equivalent to the SAX API of XML document processing. In fact, SAX API is the de-facto standard API. For that reason, we expect that many users have sufficient knowledge of SAX. Using the proposed FREDDY, users can easily develop Web mashups that compose web services to output continuous data streams. Because space is limited, we have concentrated on data interchange and have devoted scant attention to how to translate the output signal of a sensor into our proposed data format.

3 Data model

3.1 Outline of Data Format

The data format for FREDDY, FREDDY Format, can represent both a flat data stream and a semi-structured data stream. The FREDDY format is simple. Figure 3 portrays instances of the FREDDY format as an XML tree and a data stream representation. What the right of that figure readily clarifies is that the FREDDY format uses function calls written in JavaScript. Each line of FREDDY data expresses a type of SAX event and its property. We named each function call *event container* a generic name.

The main characteristic of the FREDDY Format, contrasted against that of JSON, is that it is *splittable* because it is a simple repetition of event containers. Actually, FREDDY realizes streaming between web servers and web browsers



```
sd();
s("beer");
s("guinness",{"label":"Draught"});
c("I like it!");
e("guinness");
s("guinness",{"label":"Extra Stout"});
e("guinness");
e("beer");
ed();
```

```
sd();
s("thermometer");
s("temperature",{"unit":"Celsius"});
c("30");
e("temperature");
s("temperature",{"unit":"Celsius"});
c{"28"}
e("temperature");
...
```

Fig. 3. Examples of semi-structured data representation: the four expressions are mutually equivalent.

Table 1. Event containers

SAX event	Usage	Event container
XML Document start	Data stream start	`ds();`
XML Document end	Data stream end	`ed();`
XML Text Node	Property	`c(`*value*`);`
XML Element start	Tuple start	`s(`*element-name* , *attr*†`);`
XML Element start‡	Tuple start‡	`S(`*simplified-name*, *attr*†`);`
XML Element end	Tuple end	`e(`*element-name*†`);`

†: optional argument
‡: with simplified element name

by sending and receiving small fragments of all data one by one. Later in this paper, a more precise account of the split FREDDY format is provided. We now address the event container in detail.

3.2 Event container

Table 1 portrays a list of all event containers. The events of the XML Document start and XML Document end are represented as `ds()` and `de()`. The data stream must start with a `ds()` event container and end with a `de()` event container. The two containers must appear only once.

The second argument of `c(...)`, which represents an XML Text node, is optional. It is always omitted from the representation of XML Tree because the XML Text Node has only a value.

The events of an XML Element start and end are represented as `s(...)`, `S(...)`, and `e(...)`. The argument of `e(...)` is optional. The element name is associated with an XML Element start event, which corresponds to that if no argument is given. The reason is data size reduction. For the same reason, the second argument of `s(...)` and `S(...)`, which represents XML Element Attributes, is optional.

3.3 Compression of verbose elements' name

Actually, XML is known to be verbose by design[12], particularly in terms of elements that appear many times. The SAX event stream has the same problem. For example, bibliographic information of Digital Bibliography and Library Project (DBLP)[1] has about 2,300,000 <author> tags, about 600,000 <inproceedings> tags, and about 350,000 <journal> tags.

Efficient SAX event stream interchange must tackle that data verbosity. In this context, XML compression is an important issue. In addition, XML compression is our concern: we proposed XML compression according to the simplified element name[7, 17]. The FREDDY Format tackles XML verbosity using the method of simplified element names. The reason that two event containers exist for the XML Element start event is data size reduction.

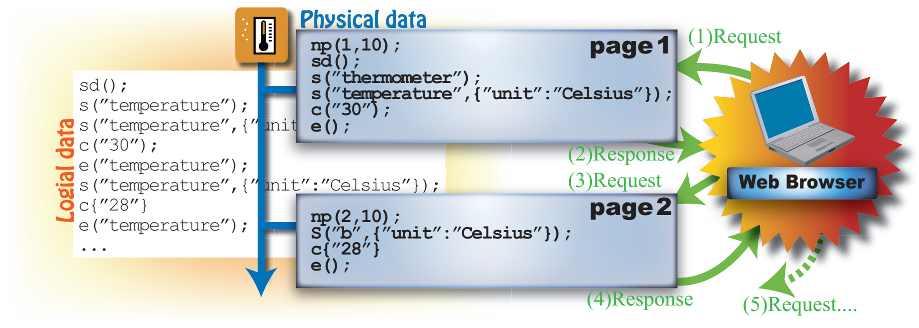
See Fig. 3. Two <Guinness> tags and two <temperature> tags exist. If the element name is <a> instead of <Guinness> and <temperature> then the amounts of data become small. This is the conceptual foundation of the compression method.

Algorithm 1 shows an algorithm for creation of an XML Element start event container from the element name. Whenever the parser captures the element start event, this procedure is called, where input T is a stack of visited element names and argument *name* is an element name. The procedure then returns a simplified element name derived from the original element name.

Algorithm 1: SimplifiedFREDDY(T, *name*)

```
procedure GetSimpleName($idx$)
  $X \leftarrow [a, b \ldots y, z, A, B \ldots Y, Z, 0, 1 \ldots 8, 9]$
  $len \leftarrow$ LengthOf($X$)
  if $len \leq idx$
    then
      $y \leftarrow X[idx\%len]$
      $z \leftarrow idx/len$
      return (GetSimpleName($z$) + $y$)
      comment: + means connection
    else return ($X[idx]$)

main
  if $T[name]$.IsExist()
    then
      $evtContainer \leftarrow$ "S('" + $sName$ + "')"
      return ($T, evtContainer$)
    else
      comment: First appearance of the element
      $idx \leftarrow$ LengthOf($T$)
      $sName \leftarrow$ GetSimpleName($idx$)
      $T[name] \leftarrow sName$
      $evtContainer \leftarrow$ "s('" + $name$ + "')"
      return ($T, evtContainer$)
```



```
sd();
s("temperature");
s("temperature",{"unit
c("30");
e("temperature");
s("temperature",{"unit":"Celsius"});
c{"28"}
e("temperature");
...
```

```
np(1,10);
sd();
s("thermometer");
s("temperature",{"unit":"Celsius"});
c("30");
e();
```

```
np(2,10);
S("b",{"unit":"Celsius"});
c{"28"}
e();
```

Fig. 4. Split of FREDDY data.

`GetSimpleName(...)` function generates the simple name for each new incoming element name. The original element name will be replaced with the simple name whenever this element re-appears later in this XML document. The original element name itself is kept within the XML document by not replacing the first entry of this element name.

3.4 Splitting data streams into small fragments

A salient difference between FREDDY and JSON is that FREDDY data can be split by line into valid fragments, which maintains a subset of the JavaScript programming language. All lines of FREDDY data are actually a subset of JavaScript.

Switching our attention to the continuous data stream, the data sources (e.g. thermometer and NOx sensor, etc.) always output data continuously, but HTTP, which enables any Web browser to communicate with any Web server, cannot handle continuous data streams. Therefore, FREDDY can split continuous data streams into small fragments, named Pages, as physical data.

Figure 4 presents an example of split data. The function-call `np(`*`pointer, sleep-time`*`);` in the first line of each Page is a pointer to the next Page. FREDDY can start downloading the subsequent page when the function `np` is called. The Pages are valid JavaScript code. For that reason, the web browsers can download them dynamically using HTTP.

Actually, FREDDY uses the dynamic <script> tag technique for downloading Pages. The dynamic <script> tag technique is a kind of Ajax based on Dynamic HTML. JavaScript applications can append HTML elements into the DOM tree of the HTML page. For example, if <img> tag with the src attribute is appended directly to the inside of the <body> tag of the DOM tree, then the image that the src attribute refers to is readily apparent. Similarly, using the dynamic <script> tag, one can access all HTTP-enabled resources.

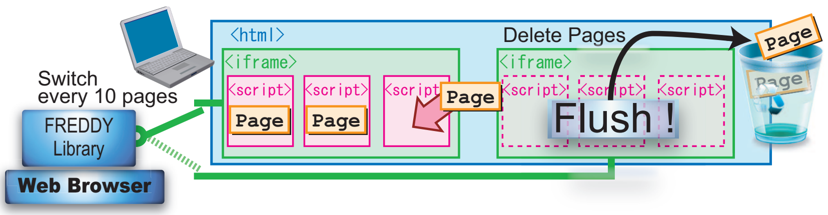


Fig. 5. Our client implementation.

4 Implementation

4.1 Dynamic <script> tag

In the following section, we describe our implementation for data interchange using FREDDY. The SAX-style programming makes only one pass through the document from top to bottom. For that reason, FREDDY deletes Pages after execution because they become unnecessary. An important problem was that web browsers did not release the <script> element from memory even when FREDDY deleted the element from the DOM tree. For that reason, our implementation adopts a hybrid of <iframe> and <script> because the web browser released the tags from the memory of the client PC when FREDDY flushed the <iframe> element.

Because of the hybrid implementation, FREDDY achieves both lower processing costs and higher throughput. Our client implementation is portrayed in Fig. 5. Actually, FREDDY uses two <iframe> elements alternately. FREDDY appends <script>, which includes a Page, as a child node of one <iframe> element at the beginning. It flushes the <iframe> and switches the other <iframe> area if the number of Pages containing the <iframe> area reaches 10. Two <iframe> are used because of the synchronism of the dynamic <script> tag technique. That is to say, when Pages are appended into a <iframe>, the other <iframe> element is flushed; then Pages are released from memory simultaneously.

4.2 Streaming delivery system

Figure 6 shows how web applications load a data stream over HTTP protocol. The data interchange between data source and web application consists of the following four steps:

1. Raw data are translated into Pages of FREDDY format by the gateway specializing in each data source.
2. FREDDY requests to the data source and receives Pages one by one using the dynamic <script> tag technique.
3. The downloaded page is executed using a JavaScript engine of a web browser.

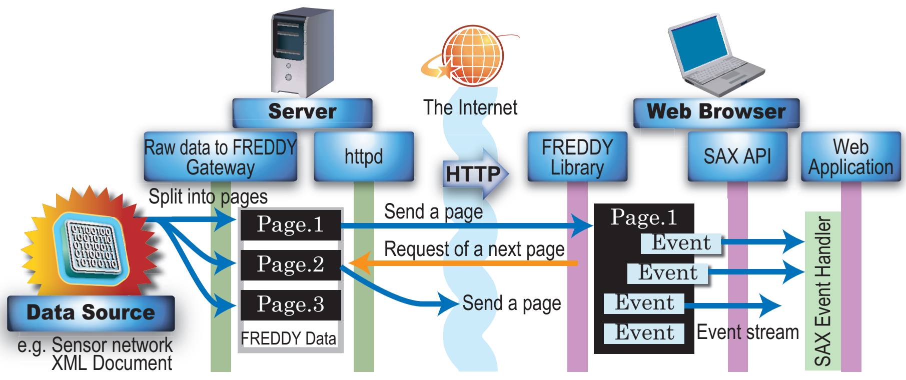


Fig. 6. Continuous Data Stream over HTTP.

4. The SAX event is noticed to the user defined event handler. If the function `np(...)` is called, then FREDDY requests download of the following Page.

It seems complex, but, in fact, it is very simple for users because the steps above are hidden by our implementation. Therefore, the web application can receive a continuous data stream easily via a SAX event handler. Furthermore, the size of the FREDDY library, which is written in JavaScript, is only 5 kByte because it is sufficiently lightweight to include into web applications.

4.3 JavaScript SAX API of FREDDY

Next, describing the usage of FREDDY, the FREDDY streaming delivery system is executed behind the SAX API, so that the SAX API is the only interface for FREDDY.

The usage of JavaScript SAX API has three steps. The first is creating methods of the SAX event handler. The name of the method is the same as that of the methods of Java DefaultHandler class. The next is creating instances of both the SAX Parser and event handler and setting the handler to the parser. Finally, execute and start parsing document.

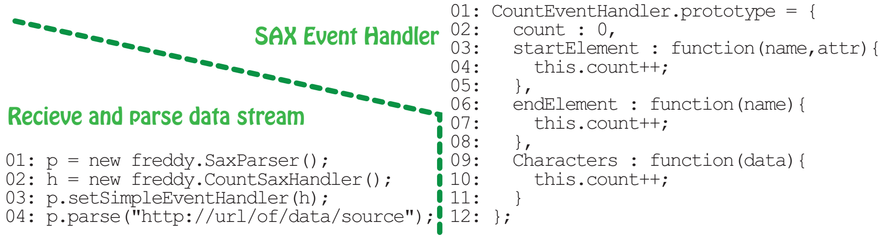
For example, if the events of a documents are counted, then an event handler can be created, as portrayed in Fig. 7 left. The right part of Fig. 7 shows the code to count up the events of the data.

This is a general procedure related to all SAX Parsers, so that FREDDY is not only accessible by web programmers; it is also easily applicable by XML programmers.

5 Experiments and results

5.1 Dataset and environment

For experiments, we used large-scale data of four XML documents up to 400 MByte. The three small XML documents *dataS.xml*, *dataM.xml*, and *dataL.xml*



```
p = new freddy.SaxParser();
h = new freddy.CountSaxHandler();
p.setSimpleEventHandler(h);
p.parse("http://url/of/data/source");
```

```
CountEventHandler.prototype = {
  count : 0,
  startElement : function(name,attr){
    this.count++;
  },
  endElement : function(name){
    this.count++;
  },
  Characters : function(data){
    this.count++;
  }
};
```

Fig. 7. JavaScript code for using FREDDY.

Table 2. Machine environment

	Web Server	Client A	Client B
Hardware		IBM ThinkPad X41 Tablet	DELL Precision 390
CPU	Intel Xeon 2.33 GHz	Pentium M 1.6 GHz	Core2 Duo 2.4 GHz
Memory	4GB	1GB	2GB
OS	Linux (Fedora Core 7)	Windows XP	Windows Vista
HTTPD	Apache 2.2.4		

are 1 MByte, 10 MByte, and 100 MByte files created using the `xmlgen` from the XMark benchmark project[14]; the biggest XML document is a 400 MByte DBLP bibliographic information document.

The computers used for the experiments are described in Table 2. We used two different computers to estimate performance: a desktop computer (*Client A*) and a laptop computer (*Client B*). The client machines and the server machine are connected via a Giga-bit Ethernet network.

Regarding the case in which FREDDY is used as an intermediate form for XML handling with a web browser, the system has the following four tiers: (1) a client machine on which the web browser is running, (2) a web server which hosts web applications, (3) an SAX event stream-to-FREDDY gateway server, and (4) a web server which holds XML documents. However, we specifically address the interchange of FREDDY data between a server and a client. Therefore, the three servers described above are located on the same server.

5.2 FREDDY vs. JSON

Actually, JSON is well known as a browser-friendly, lightweight data-interchange format. As described earlier, JSON has a simple structure and great power of expression, but it has limited scalability. For large amounts of data, it is impractical to store all data into main memory using JSON. For this reason, we propose a novel data-interchange method, FREDDY, which is suitable for use with large amounts of data. We have performed measurements of FREDDY for comparison with JSON.

In this experiment, we measured the execution time for loading the whole XML document of each *dataS.xml*, *dataM.xml*, *dataL.xml*, and *dblp.xml* using

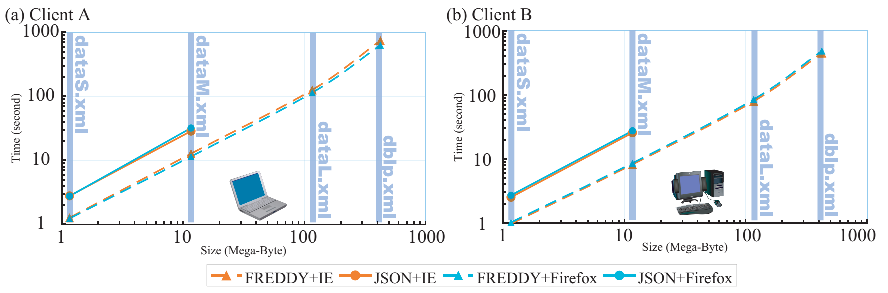


Fig. 8. Execution time of FREDDY and JSON.

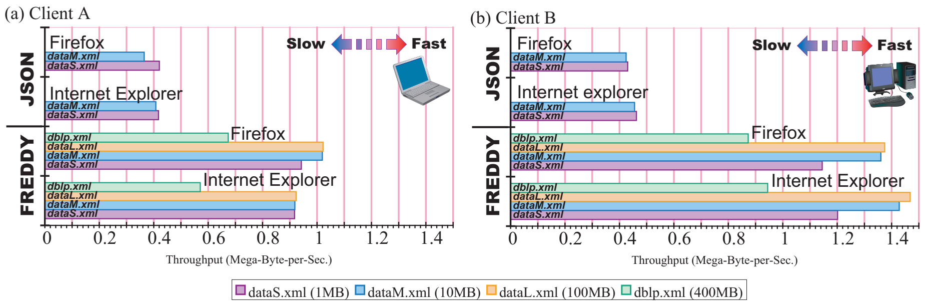


Fig. 9. Throughput (MByte/s) of FREDDY and JSON.

FREDDY and JSON on both Internet Explorer (Microsoft Corp.) and Mozilla Firefox of the client machines: *Client A* and the *Client B*. We used an XML-to-JSON gateway, which is implemented using IBM[11]. We measured all cases, which are combinations of 2 clients, 2 browsers, 2 systems, and 4 datasets, 10 times each. In this case, JSON was unable to load *dataL.xml* and *dblp.xml* because the server had exhausted the allowed memory size.

Results of this experiment are presented in Fig. 8 and Fig. 9. Figure 8 shows the average execution time of FREDDY and JSON. The throughput, the amount of loaded data per second, is presented in Fig. 9. The results of the experiment are that, irrespective of data size, the throughput of FREDDY is greater than that of JSON. We shall next examine the results more carefully.

- **Internet Explorer versus Firefox.**
  We can find no significant difference of loading times between Internet Explorer (Microsoft Corp.) and Mozilla Firefox, but Internet Explorer (Microsoft Corp.) sometimes failed to load Pages. Therefore, we developed a retransmission mechanism for FREDDY.
- **Client A versus Client B.**
  The result of FREDDY shows that *Client B* is faster than *Client A*. However, regarding the result of JSON, we can find no obvious difference between *Client A* and *Client B*. The result suggests that FREDDY is bottlenecked

by CPU power. On the other hand, we infer that the result of JSON is not influenced by client-machine specifications.

– **FREDDY versus JSON.**
  Because JSON must store all data into the main memory, it is difficult for JSON to handle a large amount of data. The result also shows that JSON was unable to load datasets of 100 MByte and 400 MByte.
  Figure 9 presents that FREDDY is faster than JSON.
  We find that FREDDY has a feature resembling that of SAX in the context of XML processing.

6 Related Works

The proposed FREDDY splits large data into small fragments for data interchange. Indeed, splitting large data into small fragments is a common solution to the problem of data interchange. Nevertheless, some problems persist in implementation of data interchange on the web browser because a JavaScript web application program must always run inside of a security sandbox. Applying common problems of information technology to the web application domain is a recent trend of research.

Klein and Spector proposed distributed computation of genetic programming via Ajax[9]. In the context of data interchange, Huynh et al. proposed sophisticated user interfaces for publishing structured data on the Web[6], but that method merely addresses the contents' presentation. It includes no solution for large documents.

The target of comparison in this study, JSON, has spread quickly on the Internet. The W3C proposed an application for semantic web for representation of SPARQL query results[4]. The JSON-RPC[2] is a lightweight remote procedure call protocol, which resembles XML-RPC. Results of several studies suggest it as a future direction of FREDDY development.

Natarajan describes an innovative transport layer protocol for data interchange on the Web[10]. However, to the best of our knowledge, no method exists for the application layer. FREDDY is an application layer method. Consequently, users use FREDDY in an existing HTTP and TCP/IP environment.

An extremely important issue related to JavaScript applications is security management. Jackson and Wang tackle the security of cross-domain scripting[8]. We also devote attention to the security of web applications, but a more comprehensive study of security is beyond the scope of this paper.

7 Conclusions

As described herein, we have presented FREDDY, a browser-friendly lightweight data-interchange method that layers efficient data stream interchange between a web server and a web browser. We also proposed an SAX style API to load a continuous data stream. Results of our experiments show that FREDDY can be a good solution for data interchange on the Web.

The future direction of this research will be one that encompasses data sources. We seek to focus attention how to translate raw data into FREDDY format. We plan to extend the design to an infrastructure of a web mashup that can communicate between the Web and a sensor network. We believe that JSON and FREDDY can serve as a basis for data interchange for web applications.

References

1. Digital bibliography and library project (dblp). `http://dblp.uni-trier.de/`.
2. Json-rpc. `http://json-rpc.org/`.
3. Sax. `http://sax.sourceforge.net/`.
4. K. G. Clark, L. Feigenbaum, and E. Torres. Serializing sparql query results in json. `http://web5.w3.org/TR/2007/NOTE-rdf-sparql-json-res-20070618/`.
5. D. Crockford. Introducing json. `http://json.org/`.
6. D. F. Huynh, D. R. Karger, and R. C. Miller. Exhibit: lightweight structured data publishing. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 737–746, New York, NY, USA, 2007. ACM Press.
7. H. Ishikawa, S. Yokoyama, S. Isshiki, and M. Ohta. Project xanadu: Xml- and active-database-unified approach to distributed e-commerce. In *DEXA '01: Proceedings of the 12th International Workshop on Database and Expert Systems Applications*, pages 833–837, Washington, DC, USA, 2001. IEEE Computer Society.
8. C. Jackson and H. J. Wang. Subspace: secure cross-domain communication for web mashups. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 611–620, New York, NY, USA, 2007. ACM Press.
9. J. Klein and L. Spector. Unwitting distributed genetic programming via asynchronous javascript and xml. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1628–1635, New York, NY, USA, 2007. ACM Press.
10. P. Natarajan, J. R. Iyengar, P. D. Amer, and R. Stewart. Sctp: an innovative transport layer protocol for the web. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 615–624, New York, NY, USA, 2006. ACM Press.
11. S. Nathan, E. J. Pring, and J. Morar. Convert xml to json in php. `http://www.ibm.com/developerworks/xml/library/x-xml2jsonphp/`.
12. W. Ng, W.-Y. Lam, and J. Cheng. Comparative analysis of xml compression technologies. *World Wide Web*, 9(1):5–33, 2006.
13. T. O'Reilly. What is web 2.0. `http://www.oreilly.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20%.html`.
14. A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. Xmark: a benchmark for xml data management. In *VLDB'01: Proceedings of the International Conference on Very Large Data Bases*, pages 974–985, Hong Kong, China, 2001.
15. Wikipedia, the free encyclopedia. Ajax. `http://en.wikipedia.org/wiki/Ajax_%28programming%29`.
16. Wikipedia, the free encyclopedia. Software as a service. `http://en.wikipedia.org/wiki/Software_as_a_service`.
17. S. Yokoyama, M. Ohta, and H. Ishikawa. An xml compressor by simplified element name (in japanese). In *Proceedings of DBWeb2000*, 2000.