Optimization of Processing the Large Data Stream in Web-interface

Nataliya V. Papulovskaya[1], Artem A. Rapoport[2]

[1] Ural Federal University named after the first President of Russia B.N.Yeltsin, Yekaterinburg, Russia; pani28@yandex.ru
[2] Uberall GmbH, Berlin, Germany

Abstract. The paper presents description of the problems related to a large amount of data frequently received from a Web-server by the Web-interface causing the insufficient performance of the latest one. A comparative analysis of methods for updating the data in the Web-interface was made, and the optimal method for updating the data in the realtime Web-application was chosen. The paper also provides an example of optimization of data processing using the data buffering and implementation of this example in the modern JavaScript.

Keywords: data processing, data buffering, big data, Web-interface, Webprogramming, WebSocket

1 Introduction

Multiple issues about insufficient performance exist in the modern Web-applications development. These issues include: slow first page load because of large amounts of data; low responsiveness or even freezing of an interface when updating a lot of data, and others.

Web-applications can store more than 900 GB of data on the server side (or in a database), for example, the data of the students from all around the world [1]. Therefore, one of the most significant problems in the development of the Web-frontend part in the Web-applications is the low speed of handling the big data flow sent from the server to the Web-client (the latter most often being a Web-browser). Implementation of updating the visual data in a Web-client can be done in different ways: update (and re-render) the data only after a specific user action; use polling (in this case the Web-client sends requests to the server with a time interval), or using the network messaging protocol, other than HTTP (HyperText Transfer Protocol), which would allow the Web-server and Web-browser to communicate in real-time (*e.g.* WebSocket) and others [2].

This research shows the advantages of data updating implementation by using the Websocket comparing to other implementations of real-time data Web-applications and the efficiency of using an additional data handling optimization by application of buffering.

CEUR-WS.org/Vol-2076/paper-13.pdf

2 Comparison of data updating implementations

Implementation of updating the visual data in a Web-client can be done by well-known HTTP-requests, which are fired by some user actions. For instance, this exact method is used in the search engine of the Google, USA: by pressing the search button, the user sends HTTP-request to the server, and after some short period of time, the Web-client receives the response and updates the information on the page. Listing 1 contains the example of a simple HTTP-request written in the JavaScript.

Listing 2-1. HTTP-request example, written in JavaScript

```
import axios from 'axios'; //HTTP-requests library
import ClientStore from './ClientStore'; //store of Web-client
/** Function that requests the data from server and receives
 *   an answer
 *   @function getInfoFromServer
 *   @return {void}   */
function getInfoFromServer() {
   axios.get('/api/info')
   //Got 'positive' response with the data from the server
   .then((response) => {
   //Rewrite the data in the Web-client store
      ClientStore.rewriteContent(response.data);
   })
   //Got 'negative' response from the server (an error).
   .catch((response) => {
      //Inform the user about the error
      console.error('Error while receiving response = ',
                             response.data);
   });
}
```

Implementation of updating the visual data in a Web-client can be done by the help of HTTP-polling. In this case, the same simple HTTP data request is used, but it is repeated with a time interval, so that the Web-client can show the real-time information (Listing 2 has the example written in the JavaScript). This method is easy to implement and can be used when developing a simple Web-interface of a network device to observe relatively small amounts of data, for example, monitor one parameters table of a network device in real time.

Listing 2-2. HTTP-polling example, written in the JavaScript

```
import axios from 'axios'; //HTTP-requests library
import ClientStore from './ClientStore'; //store of Web-client
let refreshId = null; //id of the polling interval
/* @function setInfoPolling
*  @return {void}   */
function setInfoPolling() {
   refreshId = setInterval(
```

```
    () => {
        axios.get('/api/info')
        //Got 'positive' response
        // with the data from the server
            .then((response) => {
                //Rewrite the data in the Web-client store
                ClientStore.rewriteContent(response.data);
                })
            //Got 'negative' response from the server
            .catch((response) => {
                //Inform the user about the error
                console.error('Error while receiving response = ',
                    response.data);
            });
        }, 1000 );
}
/* @function stopInfoPolling   @return {void}  */
function stopInfoPolling() {
    if (refreshId) {
        clearInterval(refreshId);
    }
}
```

Finally, the WebSocket can be an implementation of updating the visual data. In this case, the Web-client establishes a connection with the server and subscribes to some necessary topics (themes) of data, and the server sends the data when and only when it is obligatory, for example, when the data has been changed in the database. (Listing 3 contains WebSocket example implementation written in the Javascript). Any Web-application with real-time data can be set as a good WebSocket implementation (as in the short example), because the WebSocket is the most performance-wise and server-load-wise efficient transport [3],[4]. Yandex.Mail uses the WebSocket to load messages (e-mails) from the server in real time (Fig.1).

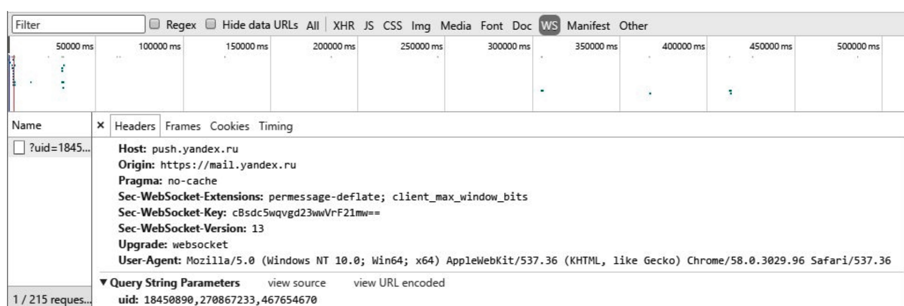

Fig. 1. WebSocket-connection on the Yandex.Mail page

Listing 2-3. Example of the WebSocket session handling, written in JavaScript

```
const DATA_URL = 'data';
const GET_SNAPSHOT_RPC = 'getSnapshot';

/* Handler that abstracts the AutobahnJS (WAMP) methods
*  @class SessionHandler */
class SessionHandler {
   /* Private field that stores session object
    * @private  */
   _session = null;
   /* @constructor
    * @param {?Session} [session = null]
    * Client-server session. Null by default. */
   constructor(session = null) {
      this.setSession(session);
   }
   /* Method that stores the session in the
    *    SessionHandlers instance
    * @method setSession
    * @param {?Session} [session = null]
    *    Client-server session. Null by default.
    * @return {void} */
   setSession = (session = null) => {
      this._session = session;
   };
   /* Callback-function that handles subscription messages
    * @see http://autobahn.ws/js/reference.html#subscribe
    *    Autobahn documentation
    * @callback subscriptionCallbackFn
    * @param {Array} args   array with event payload
    * @param {Object} kwargs   object with event payload
    * @param {Object} details   object with event metadata
    * @return {void} */

   /* Method that subscribes to data updates messages
    * @method subscribeToData
    * @param {subscriptionCallbackFn} callbackFunction
    * function that should be executed when received a message
    * @return {Promise}  */
   subscribeToData = (callbackFunction) => {
     if (!this._session) {
        console.error('Tried to subscribe to device data but
no session was specified in SessionHandler');
        return null;
     }
     return this._session.subscribe(DATA_URL, callbackFunction);
   };

   /* Method that requests snapshot (for the cold start)
```

```
     * @method getSnapshot
     * @return {Promise}  */
    getSnapshot = () => {
    if (!this._session) {
        console.error('Tried get snapshot but no session
was specified in SessionHandler');
        return null;
    }
    return this._session.call(GET_SNAPSHOT_RPC);
    };
}
```

The deduction (that most efficient option to handle big data is WebSocket) appears when comparing the options described above using several criteria (Table 1) because the size of the messages flow between the Web-client and the server is the smallest and most justified at this point. The method of updating data only by user action is not suitable for the application with real-time data, and polling is not suitable for it when the data becomes bloated. Moreover, with using polling, the Web-client always sends requests, even when the data in the database was not changed. Multiple WebSocket protocol "wrappers" without the standardization, on the other hand, are the apparent draw back for a developer. In this case, the developer should choose a protocol supported both on the Web-client and on the server side. Despite of the tiny amount of box solutions of this protocol data transfer implementations, the WAMP (the Web Application Messaging Protocol) [6] might be a suitable open-standard protocol that allows one to implement messaging though WebSocket is relatively simply [7].

3 Solution overview

To solve the issue of excessive messages flow from the server to the Web-client, for example, in the case of monitoring a huge network, handling of each received message should contain minimum amount of calculations. So that the overall message handling time is decreased, which gives the Web-client more time to execute other tasks (for example such heavy tasks as DOM (Document Object Model) rendering) in the one-thread Web-client. This can be efficiently done with using data buffering. This kind of buffering works alike the buffering in a CPU[8]. More specifically, the Web-client does not handle the messages immediately after it receives them from the server, but stores them in a buffer, not into an array but into one message object. So applying the object to the store data is simple and should only be done once per time interval.

Let us assume that Web-application (that is needed to be developed) can have either huge or small amount of data in the database. For example, in the case of network monitoring application with real-time data. If there is a low amount of network nodes (assuming their quantity is from 1 up to 50), the buffering will not change much in terms of performance because the modern computers have enough calculation resources to handle telemetries of such nodes amount in a

short time (if every message does not lead to recalculation and re-render of the network map graph, of course).

Table 1. Comparison of the different ways of data updating in the Web-client

Comparison criteria	Updates only by user actions	Polling	WebSocket
Possible to create an application with real-time data updates	no	yes	yes
Big number of boilerplates (ready-made implementations) and, therefore, simplicity of development	yes	yes	no
Web-client to server requests amount	Depends on the frequency of the user actions	Requests are sent by an interval, *e.g.* every 2 seconds	There is only one request: to establish the connection to the server
Necessity of implementation of the big data flow optimization methods in the Web-client	Data flow can be large only when there is a high amount of direct user requests. It should be handled on the server, so that other users do not experience lags during working with the interface	The polling is used in the real-time data applications; therefore, the data should be updated quite often. On the other hand, if the server contains too much data, then the polling, even with the data clustering implementation, might become inappropriate decision.	The server sends the data to the Web-client by himself; therefore, if the server observes frequent changes of the data in the database, all the changes will be sent to the Web-client as a large messages flow.

It follows that the buffering influence on visual Web-interface user experience is somehow needed to be minimized when there are not so many network nodes, or, more specifically, messages from the server (when the messages come from the server less frequently than the buffering interval). On the other hand, the buffering impact on the frequency of the data updates is also needed to be increased in order to avoid stack overflow of the calculation operations and, thus, avoid the interface freezing when the messages start to come from the

server more frequently. To implement this, the timeout of updating the data should be refreshed when no message was received in the buffering interval at the shortest point and when the sum of the time intervals (including the time refresh-iterations) becomes somewhat critical to do a force update at the longest point. For instance, assuming minimum update interval is 200ms, if the Web-client receives only one message during these 200 ms, the data from this message will be applied to the data in the Web-client's store immediately after this time interval. Otherwise, if the Web-client gets more than 1 message per these 200 ms, the timeout will be refreshed and will wait for the next message again, then the timeout will be refreshed again *etc*. This will go on until the total sum of the time intervals becomes critical, *e.g.* equal to 1 second. In this case, all the data, which were merged from the messages (that were received during this 1 second) will be applied to the Web-client's data store. Thus, when the messages flow is large, the update occurs only 1 time per second and when the messages flow is smaller, the update can occur from as frequent as 1 time per 200 ms to as frequent as 1 time per 1 second.

Listing 4 contains the implementation of the algorithm above written in JavaScript, EcmaScript 2015 standard [9]. There, the UpdateHandler class does the accumulating and applying the accumulated data to the Web-client's data store. In this implementation, the class handles the messages about the events that occur in a massive network, therefore, the buffering needed to be introduced so that the operations delay becomes minimal. The external WebSocket message handlers pass the messages with data updates of events, occurred in the network, to the "bufferedUpdate" method. Assuming that the constant named Constants.DEFAULT_UPDATE_TIMEOUT is equal to 200 ms, the implementation of the algorithm described above is presented in the listing.

Listing 3-4. The example of updates handling with buffering usage, written in the JavaScript

```
import * as Constants from '../Constants';
import * as eventsActions from 'actions/eventsActions';

class UpdateHandler {
   /* id of the timeout with minimum interval */
   _interval = null;

   /* id of the timeout with maximum interval
    * (when the force update occurs)      */
   _maxInterval = null;

   /* @type {Boolean}
    * Boolean flag of the force update    */
   _forceApply = false;

   /* Main data store instance            */
   _store = null;
```

```
/* @type {String}
 * Updating mode (can be either set
 * to updating or accumulating)        */
_mode = null;

/* @type {Object}
 * Object with accumulated updates of type
 * {events: Map}                        */
_accumulatedUpdates;

constructor(store) {
   this._accumulatedUpdates = {events: new Map()};
   /* initializing the value for the accumulated updates */
   this._store = store;
   //saving the instance of the main store in the field
   clearInterval(this._interval); //stopping updates
   this._startBatchedWaitingTime();
   /* initiating  external  timeout with the interval
      of the force update */
}

/* @private
 * @method _applyAccumulatedUpdatesIfNeeded
 * Method that applies the data, that was accumulated
 * during the interval of updating
 * If there was no message from the server,
 * data is not needed to be applied     */
_applyAccumulatedUpdatesIfNeeded = () => {
   if (this._accumulatedUpdates.events.size > 0) {
      clearTimeout(this._maxInterval);
      /* stopping 'external' timeout with the
         interval of the force update */
      clearInterval(this._interval);
      //stopping 'internal' timeout
      this._startBatchedWaitingTime();
      /* initiating 'external' timeout with
        the interval of the force update */
      this.applyAccumulatedUpdates();
      //applying accumulated data updates to the store
   }
};

_startBatchedWaitingTime = () => {
   // setting force update flag to false
   this._forceApply = false;
   // Setting the force update flag to true after 1 second
   this._maxInterval = setTimeout(
            this._clearBatchedWaitingTime,
            Constants.DEFAULT_UPDATE_TIMEOUT * 5);
```

```
  };

  _clearBatchedWaitingTime = () => {
    this._forceApply = true;
  };

  bufferedUpdate(updates) {
    //stopping 'internal' timeout
    clearInterval(this._interval);
    //adding updates (accumulating)
    this._accumulateUpdates(updates);
    if (this._forceApply === true) {
      /* if the force update flag is set to true
        updating the data */
      this._applyAccumulatedUpdatesIfNeeded();
      //exiting the current method
      return;
    }
    this._interval = setInterval(() => {
      /* creating 'internal' timeout,
        after which the data update is called */
      this._applyAccumulatedUpdatesIfNeeded();
    }, Constants.DEFAULT_UPDATE_TIMEOUT);
  }

  _accumulateUpdates(updates) {
    if (!updates) { return; }
    updates.forEach(update => {
      /* else, for each update add the data
        to accumulated updates object */
      this._accumulatedUpdates.events =
        UpdateHandler.appendEventMessage(
          this._accumulatedUpdates.events,
          update);
    });
  }
  static appendEventMessage(initialValue, newMessage) {
    /* Here, depending on the message structure,
      new data about the network events is added */
    return initialValue.set(newMessage.id, newMessage.val);
  }
  applyAccumulatedUpdates() {
    const {events} = this._accumulatedUpdates;
    //applying data to the store
    this._store.dispatch(eventsActions.updateEvents(events));
    /* initializing the value for the accumulated updates */
    this._accumulatedUpdates.events = new Map();
  }
}
```

This algorithm was successfully implemented and used in the task of creating the Web-interfaces of network and geographical map that might contain up to 10 thousand of network devices and around 50 thousand wireless broadband links between them. The Web-server can send messages quicker than 1 message per 1 ms in the configuration with such a big network. Therefore the one-thread Web-client could not be able to handle and apply the data without a significant visual delay (when not using buffering).

Applying adaptive methods of the data handling, on the other hand, allows one to increase performance while decreasing the calculations amounts for any width of the data flow.

4 Conclusion

Modern Web-programming technologies allow the developers to implement the Web-applications that do not have the excessive calculations and optimization operations. Especially, it is important when having a low amount of data but also do not have the significant calculations delay when having a big data flow. Different development tasks require using different techniques, methods, and approaches to handle information and optimize the data flow. Nevertheless, using the WebSocket and buffering in the Web-client is an efficient way to organize and handle the data updating in the Web-application and to optimize the data flow.

References

1. AWS Case Study: Kaplan. https://aws.amazon.com/solutions/case-studies/kaplan/
2. Liping, G., Dongfang, G., Naixue, X., Changhoon, L.: CoWebDraw: a real-time collaborative graphical editing system supporting multi-clients based on HTML5. Multimedia Tools and Applications. Vol. 77, 4, 5067–5082 (2018)
3. Chto takoe Long-Polling, WebSockets, SSE i Comet. https://myrusakov.ru/long-polling-websockets-sse-and-comet.html
4. Postojannoe soedinenie mezhdu brauzerom i serverom. https://www.insight-it.ru/interactive/2012/postoyannoe-soedinenie-mezhdubrauzerom-i-serverom/
5. Kotov, A., Krasil'nikov, N.: Klasterizacija dannyh. http://yury.name/internet/02ia-seminar-note.pdf
6. WAMP - The Web Application Messaging Protocol. http://wampproto.org/
7. GitHub - WAMP in JavaScript for Browsers and NodeJS. https://github.com/crossbario/autobahn-js
8. Muller, H., Flynn, M. J.: Processor Architecture and Data Buffering. IEEE Transactions on computers. Vol. 41, 10, 1211-1222 (1992)
9. ECMAScript 2015 Language Specification – ECMA-262 6th Edition. http://www.ecma-international.org/ecma-262/6.0/