Client Server Automation for JavaScript

Dr. David W. Ge June 16, 2016

Abstract

Maturing of server side JavaScript makes it possible to create new technologies to make web application development the same as local application development by automatically handling all client server related coding tasks. Comparing to local application development, one of major differences is that when asynchronous connection is used to execute server code, client code following the server code should be refactored into a callback function in order to keep the correct control flow. The question is that can all types of flow controls be automatically refactored using asynchronous server connections and callback functions? This paper analyzes following types of flow controls: sequential operations, code branching, looping, functional programming, client value enumeration and server value enumeration. It shows that with a stateless server, all these types of flow controls can be supported. Thus, a major obstacle is removed for fully automated client/server programming. A prototype of JavaScript preprocessor shows that client/server programming can be done in the same way as local programming.

Keywords

JavaScript, Web Application Programming, Cloud Computing, client Server Programming, Internet Computing, asynchronous programming

Contents

I. Introduction

New techniques of using JavaScript at server side, for example, V8Js for PHP [1][2][3], ClearScript for ASPX.NET [4], Node.js [5] for its own web servers, etc., make it possible to use one single language, JavaScript, to develop web applications for both client side coding and server side coding. This new technology advancement opens a new opportunity of creating new technologies to remove a boundary between client side programming and server side programming. That is, doing client/server programming in a same way of local programming; such efforts can be seen before appearing of server side JavaScript in Remote Procedure Calls (“to make the remote procedure call programming paradigm truly the same as that of the local procedure call programming paradigm” [6]), web services [7] and proprietary programming systems (“To bring Web application development to the same level as desktop application development” [8]). The difficulty of web programming is also a focus of researches (“Web application development still remains difficult today and lags behind conventional desktop application development” [10])

Besides other issues, a programming paradigm using Remote Procedure Calls and web services has one fundamental limitation comparing to desktop application development in that a developer cannot

simply put two server stub function calls together, one after another, to invoke two server functions . The developer should make new server side coding to group server calls together and create a new single server call to avoid unnecessary multiple server connections. Separation of client side coding and server side coding is thus inevitable.

One solution is to create new programming languages. For example, ZK Framework [8] creates languages named XUL and XUML; ZK engine translates programming into JavaScript code and AJAX calls. They claim that it is a system of letting developers use AJAX without JavaScript.

Because JavaScript is the most widely used language for client side web programming (JavaScript is also becoming a desktop language [9]), it is desirable to provide a solution for JavaScript programming, instead of creating a new language.

One solution can be a preprocessor to process JavaScript code made by programmers to take over all client/server related programming tasks. Let's call it a "Client Server Automation" preprocessor.

Such a preprocessor can be a standalone utility to be executed by programmers. A preprocessor can also be executed by JavaScript engine in web browsers or by web server on the fly; such a solution requires modification of JavaScript engine or web server.

One big problem in creating such a preprocessor is that programmer code needs to be refactored with callback functions (or continuations when available in JavaScript in the future) for asynchronous server operations.

It is not a simple task for programmers to manually design callback functions because it is counter-intuitive. It is a long-felt pain of "callback hell" ([11], [12], [13], [17]).

A "client server automation" preprocessor should allow a programmer to use intuitive control flows and automatically do code refactoring for the programmer.

To keep the original programming control flow, an easy solution is to block the running thread after starting an asynchronous server operation, as Rossi et al ([14]) do. It is not a good solution because blocking a thread eliminates benefits of using asynchronous operations.

Freeman et al [15] are "generating asynchronous application program output computer code from input computer code of a synchronous application." Freeman's solution has following drawbacks:

1. It uses 3 network connections when only one connection is needed. It uses 3 "client sends a request, server makes a response" to implement one "server sends a request, client makes a response" for implementing Freeman's solution.
2. It only applies to simple coding situation of sequential operations.

A new JavaScript features in ES6, Promise [18], can be used to deal with the so called problem of "callback hell". A JavaScript Promise is not to remove callbacks; it provides a coding pattern to capture issues in using callbacks: 1) at the time of completing an asynchronous function one callback function

should be called; 2) at a time of an exception raised by the asynchronous function another callback function should be called; 3) in a callback function create another Promise object to execute another asynchronous function; 4) execute a callback when several asynchronous functions all completed; 5) execute a callback when one of several asynchronous functions throw an exception.

Using Promise objects, to some developers JavaScript code is more readable. But it cannot replace "client server automation".

For example, suppose a programmer made following code: op1; op2; op3; op4.

If op2 and op3 are server operations then the programmer needs to refactor the above coding and use an AJAX call to execute op2 and op3, and use a callback to execute op4.

By using Promise, the programmer still needs to use an AJAX call to execute op2 and op3, the programmer needs to create a new Promise object to contain the AJAX call and execute a "then" method passing op4 into the "then" method as the callback. Such additional efforts make code more readable to some developers. But the programmers still have to deal with all the issues of client server programming: make asynchronous server calls, arrange callbacks.

Another new JavaScript feature, Generator [19], can be used to help making it easier to do client server programming.

By using Generator, the programmer still needs to use an AJAX call to execute op2 and op3, the programmer needs to create a Generator function to execute the AJAX call and use a "yield" keyword to make the execution wait for the AJAX call to finish. From outside of the Generator function, use a "next" call to invoke op4 which is waiting for the AJAX to finish. As with using Promise, such additional efforts of using Generator make code more readable to some developers. But the programmers still have to deal with all the issues of client server programming: make asynchronous server calls, arrange callbacks.

In the future, probably in ES7, C# features of "async" function and "await" operator could be available in JavaScript, such features will make it easier to implement "client server automation" but they cannot replace "client server automation". Code "op1; op2; op3; op4" can be refactored into such a pseudo code "op1; await a function to use an AJAX call to execute op2 and op3; op4;". This will be a huge change to the way of how JavaScript works: "op4;" will wait for an asynchronous operation to finish but the thread is not blocked during the waiting. But the client server issues remain: the programmer still has to create a function to use AJAX call for op2 and op3; op2 and op3 may involve client values to be arranged as upload values and make them available for op2 and op3 to use; if op4 needs values from server or generated by op2 or op3 then such values have to be downloaded; etc.

Another issue is that it needs to verify that all kinds of flow controls involving various mixtures of client server operations can be refactored with "async" functions and "await" operator. This work is not in the scope of this paper.

Without blocking a thread, and without assuming availability of continuations/coroutines and C#-like "async/await" feature, a "Client Server Automation" preprocessor has to refactor user programming in

such a way: from where an asynchronous server connection is arranged for the programmer, the rest of the code must be skipped; a callback function must be arranged to resume execution from the place where the code is skipped.

For very simple code cases, this can be done. The question is that is it possible to do it for all coding situations? This paper analyzes following flow controls to show that these flow controls can be refactored: sequential operations, code branching, code looping, functions and functional programming, client value enumeration and server value enumeration. The solution for server value enumeration is not ideal for a stateless web server, though.

Thus, a major obstacle of creating such a "Client Server Automation" preprocessor is removed.

A "Client Server Automation" preprocessor carries out following tasks.

- Identify and group client operations and server operations
- Identify client to server and server to client execution context switching points
- Generate AJAX connections for client to server switching points, or Node.js client to server messages for Node.js environment
- Generate callback functions for server to client switching points
- Identify and arrange download and upload values; imitate execution scopes so that downloaded server values are accessible by callback functions at client side, and uploaded client values are accessible by server operations.
- Identify state values and maintain state values at client side. A state value is a server value generated by a server operation in one server connection and used by server operations in subsequent server connections.
- Refactor flow controls with callback functions and generate asynchronous server operations at ends of code flow. An asynchronous server connection must be placed at an end of an execution path so that the JavaScript thread is not blocked. Proper callback must be arranged so that original flow control is maintained.

A prototype of such a "Client Server Automation" preprocessor for JavaScript is created; it supports V8Js, ClearScript and Node.js; it is tested with following web browsers: Chrome, Internet Explorer, FireFox, Opera, and Safari.

Server functionality, such as database accessing and updating, email sending and receiving, etc., is presented in JavaScript API in JavaScript files for developers to use. Developers use server functionality via such API in the same way as using client functionality. The "Client Server Automation" preprocessor processes code made by the developers and arranges a server side dispatcher to carry out desired server functionality.

Section II to section VII of this paper analyze JavaScript flow control structures and provide solutions for refactoring JavaScript flow controls to implement "Client Server Automation".

Section VIII presents a “Client Server Automation” runtime structure consists of a client side dispatcher in JavaScript and a server side dispatcher in server technology to be supported. It provides example server technologies including V8Js (for supporting PHP), ClearScript (for supporting ASPX) and Node.js (for supporting Node.js).

Section IX describes that in a “client server automation” execution environment, a programmer is shielded from server technologies by using “server technology independent API”.

Section X summarizes pros and cons of “client server automation”.

II. Sequential Operations

For code running at client side, for example, setting HTML element attributes, we say the code is running in “client execution context”. For code running at server side, for example, database accessing, we say the code is running in “server execution context”.

Fig. 1a shows a simple coding situation; coding parts 101, 103 and 104 form a sequence.

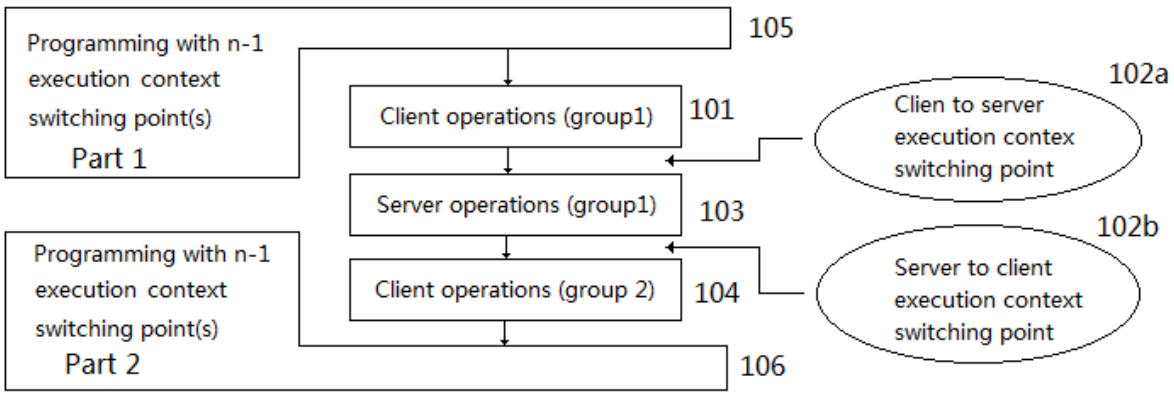


Fig. 1a

102a is a “client to server execution context switching point”; 102b is a “server to client execution context switching point”. A “client server automation” process can remove 102a and 102b by refactoring the code sequence with an AJAX call and a callback function, as shown in Fig. 1b.

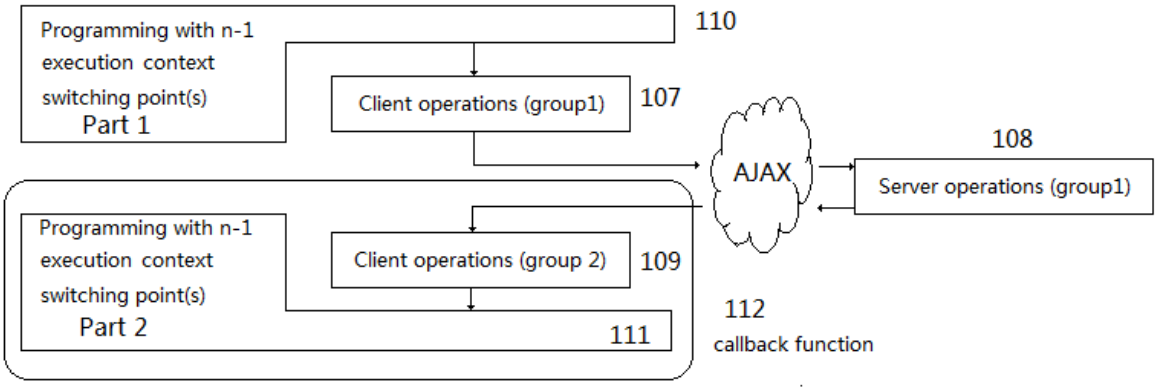


Fig. 1b

Referring to Fig. 1b, a "client server automation" preprocessor uses an AJAX call to execute "Server operations (group1)" 108, refactors "Client operations (group 2)" 109 and code part 111 into a callback function. Such a process removes one pair of "client to server context switching point" and "server to client context switching point" in the programming code.

III. Code Branching

Based on how client operations and server operations are mixed in a code branching, following 6 situations are handled by a "client server automation" preprocessor.

Situation 1

Definition: Situation 1 occurs when the first programming statement in each branch is a client operation and current execution context is server.

Processing by "Client Server Automation" Preprocessor:

The code branching is treated as a client operation.

A server to client execution context switching point is identified to be before the code branching.

The code branching is placed in a client callback function.

Each code branch is further processed by said preprocessor separately.

Situation 2

Definition: Situation 2 occurs when at least there is one branch having its first programming statement being a server operation and at least there is one client operation in one of code branches and current execution context is server.

Processing by "Client Server Automation" Preprocessor:

A server to client execution context switching point is identified to be at this code branching.

The code branching is duplicated at both server side and a client side callback function.

Server values used in the branching condition are preserved as download values separated from corresponding server variables.

Server operations at beginning of each branch are included in server side code branching to be generated.

Client side code branching is formed by client operations at beginning of each branch and client operations following server operations at beginning of each branch.

If a situation 2 code branching is nested at beginning of a code branch or following server operations at beginning of a code branch then the nested situation 2 code branching is also handled by duplicating code branching as defined above.

If a code branching other than situation 2 is nested at beginning of a code branch or following server operations at beginning of a code branch then the nested code branching is placed at a client side callback function.

Situation 3

Definition: Situation 3 occurs when current execution context is client and branching condition involves client values only.

Processing by "Client Server Automation" Preprocessor:

The code branching is treated as a client operation and there is not an execution context change.

Each branch is further processed by said preprocessor separately.

Situation 4

Definition: Situation 4 occurs when current execution context is client and branching condition involves server values.

Processing by "Client Server Automation" Preprocessor:

If the current execution context is within a callback function for a server connection then it is treated as situation 3 and the server values involved in the branching condition are download values treated as client values.

If the current execution context is not within a callback function for a server connection then a client to server execution context switching point is identified, client values used in the branching condition are upload values, server code is generated at server side to get server values used in the branching condition, the current execution context switches to be server execution context, processing of the code

branching is switched to situations of “current execution context is server” which includes situation 1 and situation 2.

Situation 5

Definition: Situation 5 occurs when all code branches are empty.

Processing by “Client Server Automation” Preprocessor:

This situation is not treated as a code branching but treated as an expression statement where the code branching condition is used as the expression.

Situation 6

Definition: Situation 6 occurs when current execution context is server and all branches do not involve client operations.

Processing by “Client Server Automation” Preprocessor:

This situation is treated as one single server operation.

IV. Looping

Flow control definition

A code looping flow control structure is formed by initialization operations, loop conditions at beginning or at ending of looping, repeated operations followed by increment operations, and operations after loop.

Each of above listed operations can be empty.

Among the loop conditions, repeated operations and increment operations there is at least one client operation and one server operation.

Referring to Fig. 2a, the code loop flow control structure executes in following way:

1. code execution starts with the initialization operations 201
2. and then repeated operations 204 and increment operations 227 are repeatedly executed
3. for each time of executing repeated operations 204 and increment operations 227, loop condition 202 at beginning of looping is evaluated before the execution and loop condition 207 at ending of looping is evaluated after the execution
4. if the loop condition evaluates to False then the operations after loop 208 are executed and the repeated operations 204 and the increment operations 227 are no longer executed, and the execution path breaks from the loop.

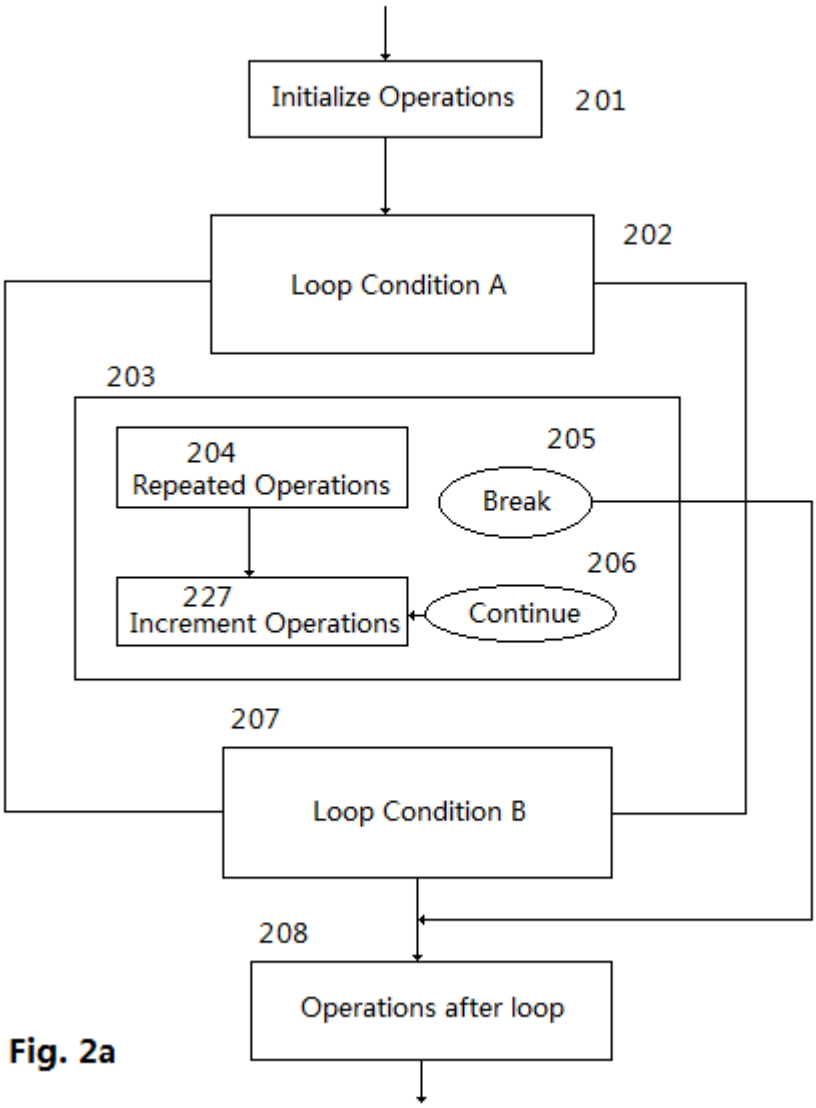


Fig. 2a

Processing by "Client Server Automation" Preprocessor

A recursive function is created to simulate code looping so that asynchronous server operations within a code looping can be used.

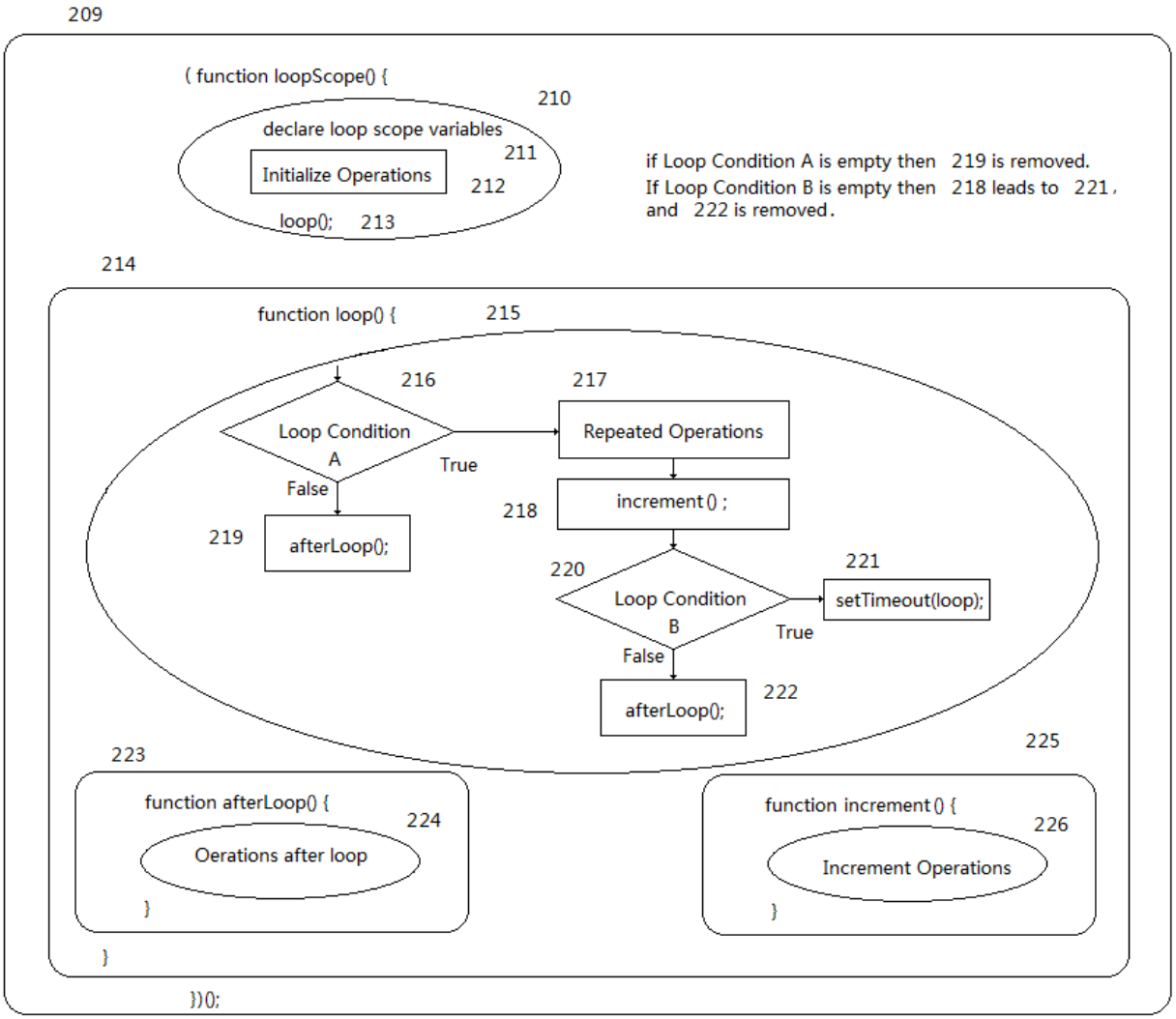


Fig. 2b

Referring to Fig. 2b, a “client server automation” preprocessor processes a code loop in following way.

1. a “loopScope function” 209 is generate to refactor a loop.
2. Following code is formed:
    a. variables declared inside the initialization operations are declared, as code 211
    b. followed by Initialization Operations 212
    c. followed by a call to a “loop function”, as defined below, as code 213.
3. Code 210 is generated by the “client server automation” preprocessor by processing the above formed code.
4. inside the “loopScope function” an “afterLoop function” 223 is generated; the preprocessor processes operations after loop and places generated client side code 224 inside “afterLoop function”

5. inside the “looScope function” an “Increment function” 225 is generated if the increment operations present; the preprocessor processes the increment operations and places generated code 226 inside the “increment function”
6. inside the “loopScope function” a recursive function is defined which is referred to as “loop function”
7. the preprocessor processes following programming and places generated client side code 215 inside “loop function”:
    a. if the loop condition 216 at beginning of looping presents and evaluates False then execute the “afterLoop function” 219 else execute repeated operations 217 followed by a call to the “increment function” 218
    b. if loop condition 220 at ending of looping presents and evaluates to False then execute the “afterLoop function” 222 else use a recursion 221 to call the “loop function”
    c. the process handles programming of a “continue” statement by generating a call to the “increment function” and a call to the “loop function” as one end of execution path in the “loop function”
    d. the process handles programming of a “break” statement by generating a call to the “afterLoop function” as one end of execution path in the “loop function”
    e. in a case of nested code looping, if an inner loop needs to be processed as defined above then all its outer loops are also processed using the process defined above with following exception: evaluation of loop conditions at ending of an outer loop, executing increment operations of the outer loop and a recursive call of the “loop function” for the outer loop are placed at the end of the “afterLoop function” for every inner loop immediately contained by the outer loop, where “immediately contained” means that an inner loop is not contained by another inner loop of the outer loop.

V. Client Value Enumeration

Flow control definition

Such a flow control structure can be shown by Fig. 3a.

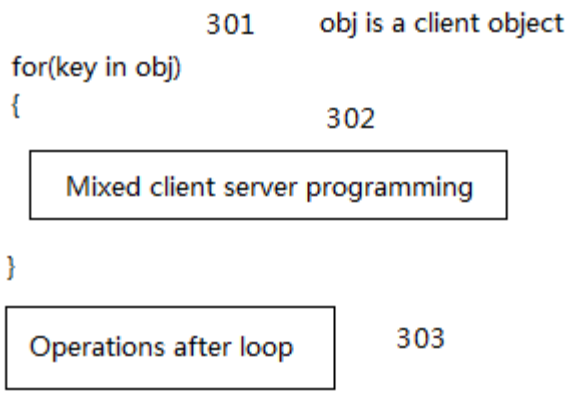


Fig. 3a

For an enumeration 301 of client values, code 302 is for handling each enumerated value. After enumerating all values, code 303 is executed.

If handling of each enumerated value includes asynchronous server operations then code refactoring is needed to maintain such a flow control structure.

Processing by "Client Server Automation" Preprocessor

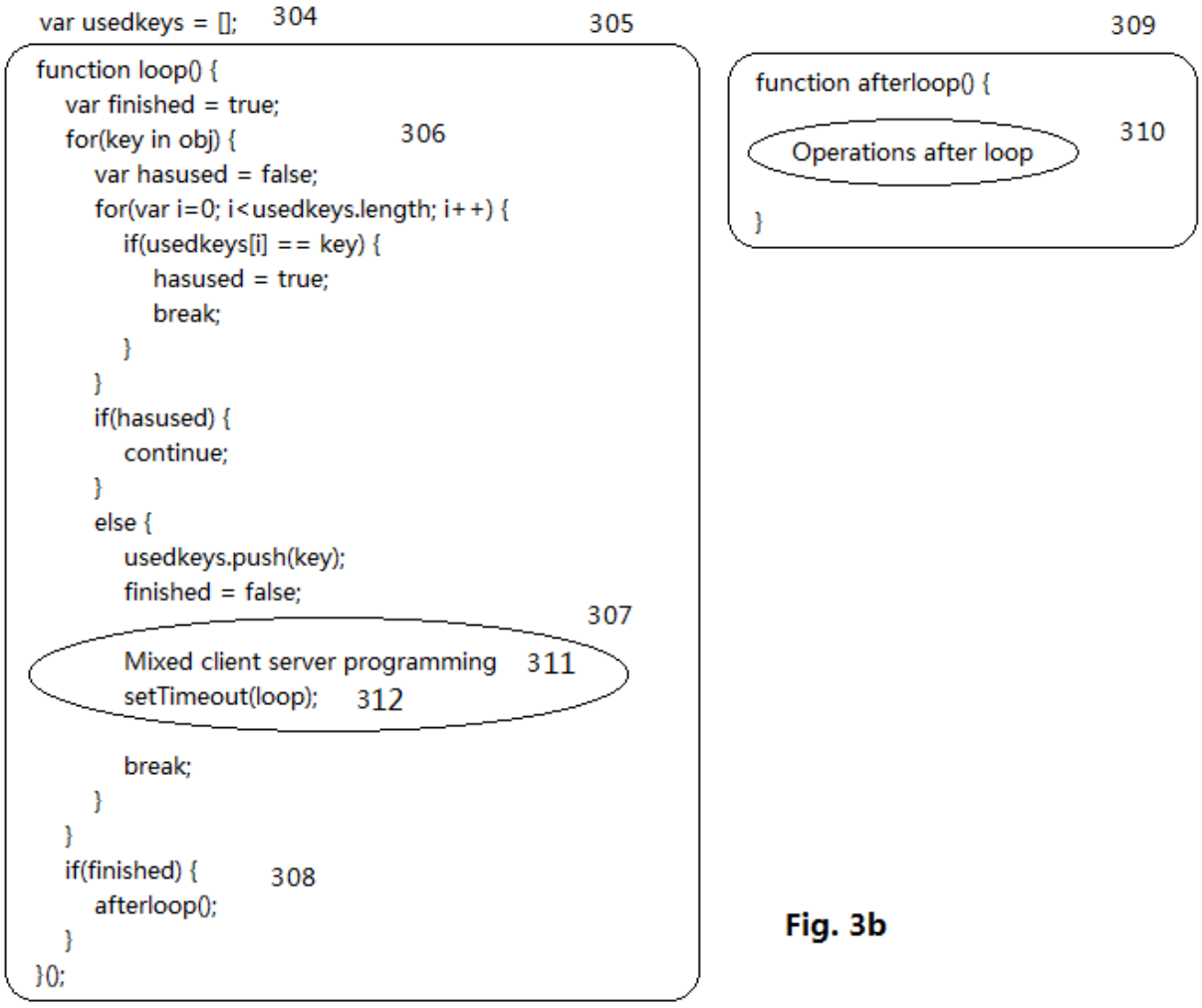


Fig. 3b

Referring to Fig. 3b a “client server automation” preprocessor uses a recursive function call to simulate an enumeration for refactoring the flow control structure so that asynchronous server operations can be used.

1. A function, “afterloop function” 309, is generated. The preprocessor processes code 303 in Fig. 3a and puts generated code 310 inside the “afterloop function”.
2. The preprocessor generates code to declare an array 304 and initialize it with an empty array, the array is for remembering used enumeration keys, it is referred to as “used key array”.
3. The preprocessor generates a function 305 following the code declaring and initializing the “used key array”, the function is referred to as “loop function”.
4. The preprocessor generates following code 306 inside “loop function”
    a. the enumeration and the “used key array” are used to get an unused key and corresponding value,
    b. if such an unused key is not found then “afterLoop function” is executed and the function finishes,

c. if such an unused key is found then the found unused key is pushed into the “used key array”, the enumeration breaks,
d. before breaking the enumeration is code 307 generated by the preprocessor processing the programming 311 for handling each enumerated value followed by a recursive call 312 to the “loop function”.

VI. Server Value Enumeration

Flow control definition

Such a flow control structure can be shown by Fig. 4a.

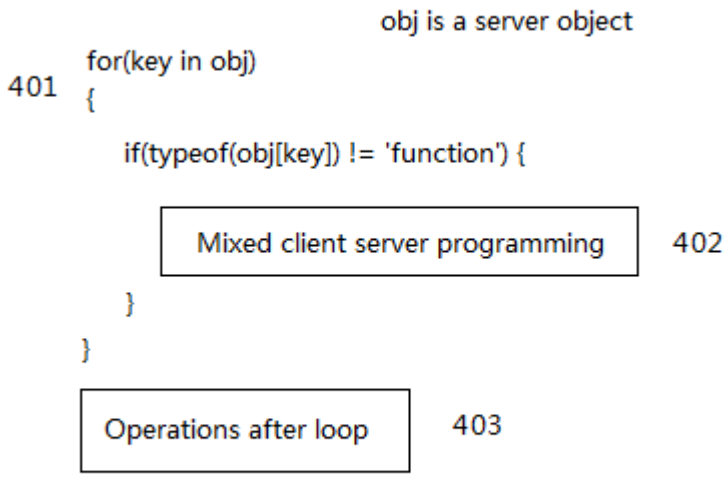


Fig. 4a

For an enumeration 401 of server values, programming 402 for handling each enumerated value includes client operations, programming 402 for handling each enumerated value is referred to as “handling code”.

Processing by “Client Server Automation” Preprocessor

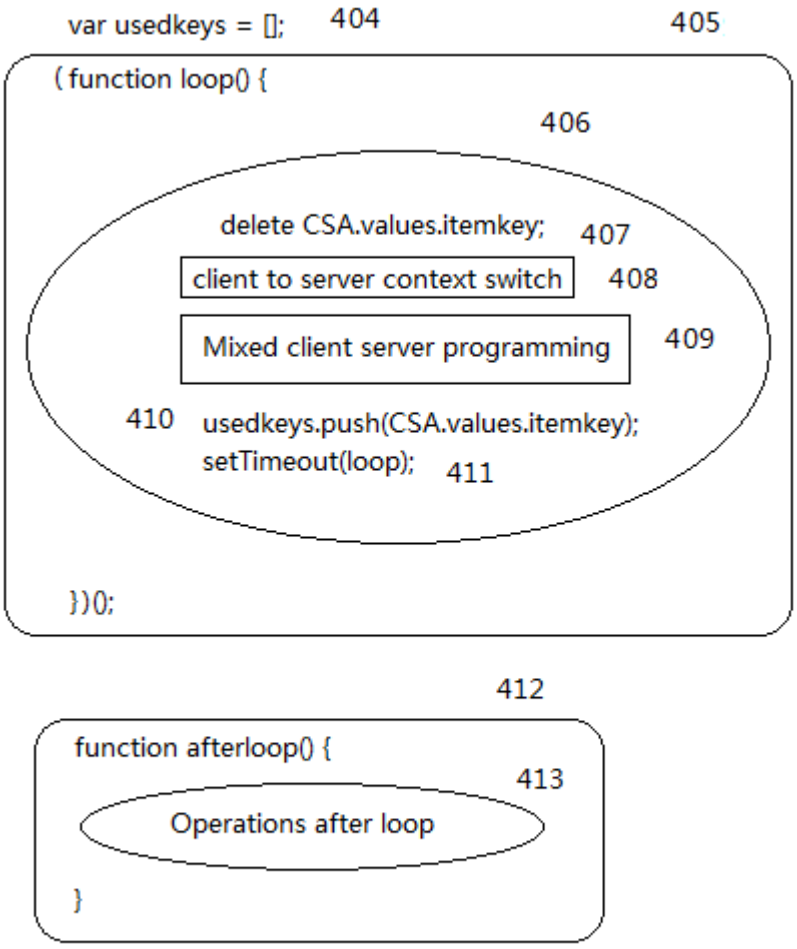


Fig. 4b

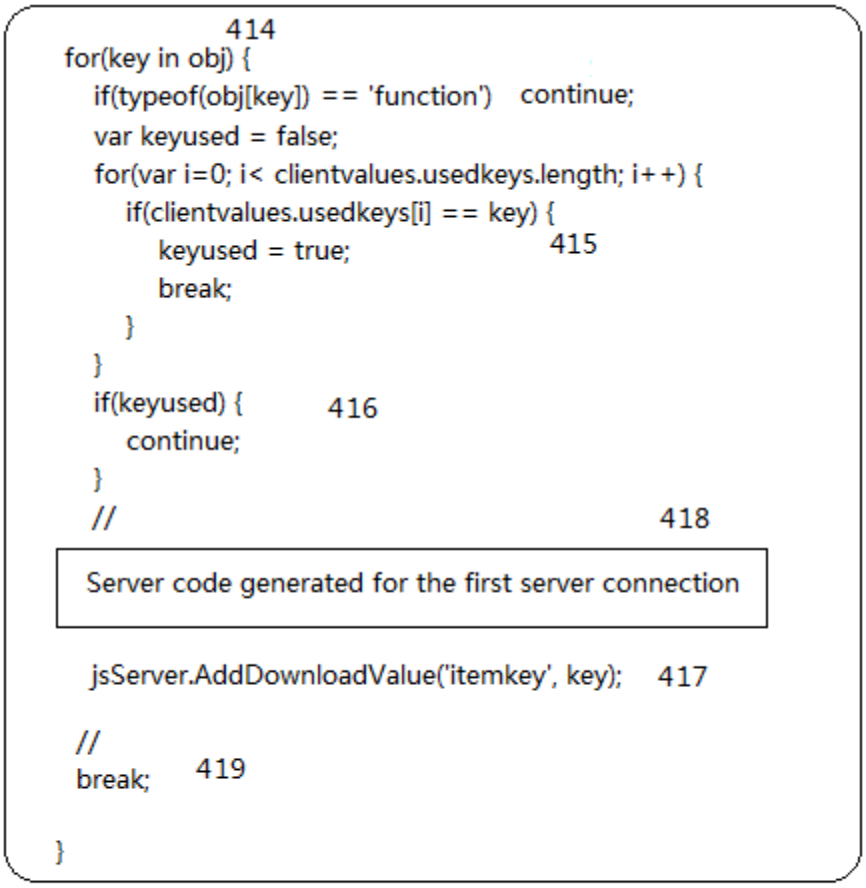


Fig. 4c Forming of server code for the first server connection

usedkeys must be included in upload value list.

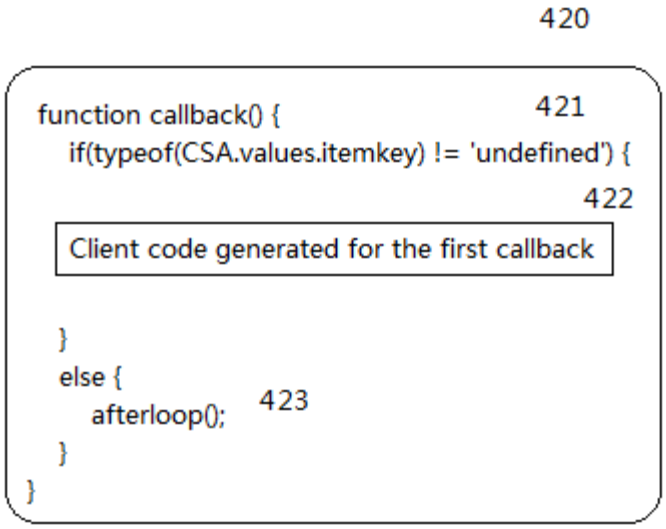


Fig. 4d Forming of client code for the first callback

Referring to Fig. 4b, a “client server automation” preprocessor refactors a server value enumeration as following.

1. The preprocessor creates a function 412, referred to as “afterLoop function”
2. The preprocessor processes programming following the enumeration and places generated client side code 413 inside “afterLoop function”
3. The preprocessor generates code to declare an array and initialize it with an empty array 404, the array is for remembering used enumeration keys, it is referred to as “used key array”
4. The preprocessor generates a function 405 following the code declaring and initializing the array, the function is referred to as “loop function”
5. The preprocessor processes following programming which is referred to as “loop coding” (407-411) and place generated client side code inside “loop function”:
    a. remove used key from download value list if it exists, see code 407
    b. switch execution context from client to server, see 408
    c. followed by the “handling code” 409
    d. followed by pushing used enumeration key in download value list to the “used key array”, see code 410
    e. and a recursive call to “loop function”, see code 411;
6. the code generated by the preprocessor processing “loop coding” (407-411) is referred to as “generated loop coding”; the preprocessor modifies “generated loop coding” to form code 406 in following way, referring to Fig. 4c,
    a. by including “used key array” in upload value list for the first server connection in “generated loop coding” and placing server code 418 for the first server connection in “generated loop coding” inside a server side enumeration 414,

b. the server side enumeration 414 uses the uploaded “used key array” to find an unused key and corresponding value,
c. if such an unused key is not found then the server code finishes and there is not an used key downloaded to client, see 415,
d. if such an unused key is found then the key is included in download value list, see 417, following server code generated by said preprocessor for the first server connection, see 418, and the server enumeration breaks, see 419, to finish the server connection;
e. referring to Fig. 4d, the preprocessor modifies “generated loop coding” by adding a condition evaluation code 421 to the first server connection’s callback function 420,
f. the condition 421 tests existence of an used key among download values,
g. if the used key exists in the download value list then code 422 for the first callback function in “generated loop coding” is executed,
h. if the used key does not exist in the download value list then “afterLoop function” 423 is executed.

The above solution needs to upload and download all enumerated keys because enumerated keys are saved at client side during enumeration, and because usually a server does not initiate a connection to a client. In a Node.js environment, a server can send messages to a client, a better solution similar to previous solution for “client value enumeration” can be used.

VII. Functions

Flow control definition

For a client side function, if some server operations are used within the function then not only refactoring of the code within the function is required, a refactoring of code calling the function is also required.

Processing by “Client Server Automation” Preprocessor

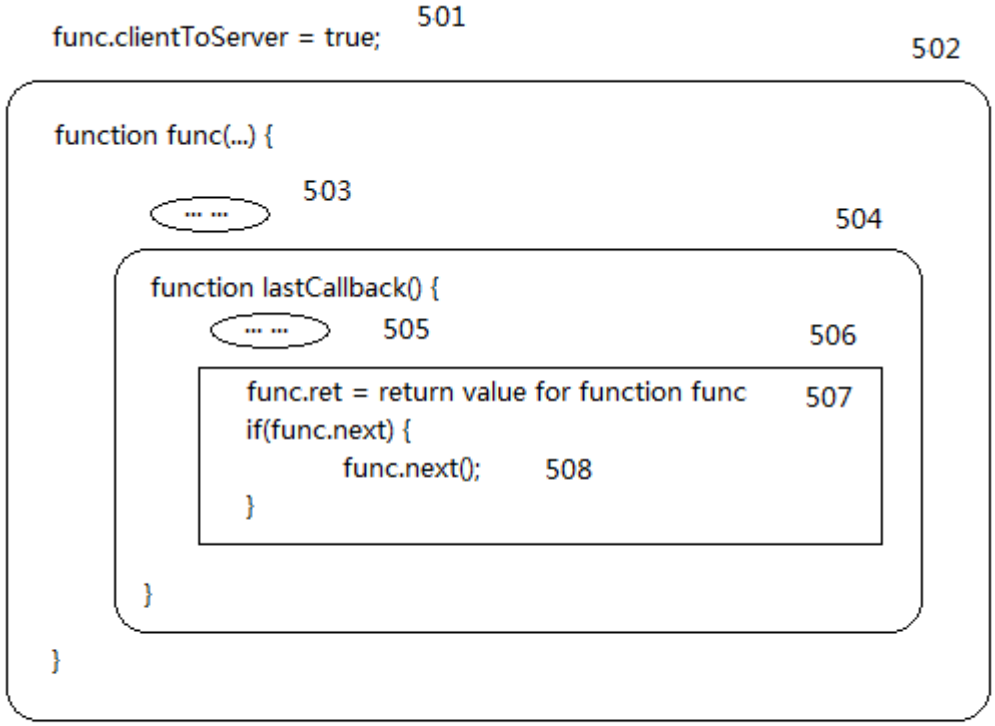
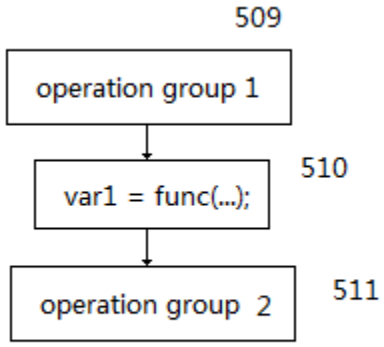


Fig. 5a



func is an asynchronous function marked with clientToServer attribute

Fig. 5b

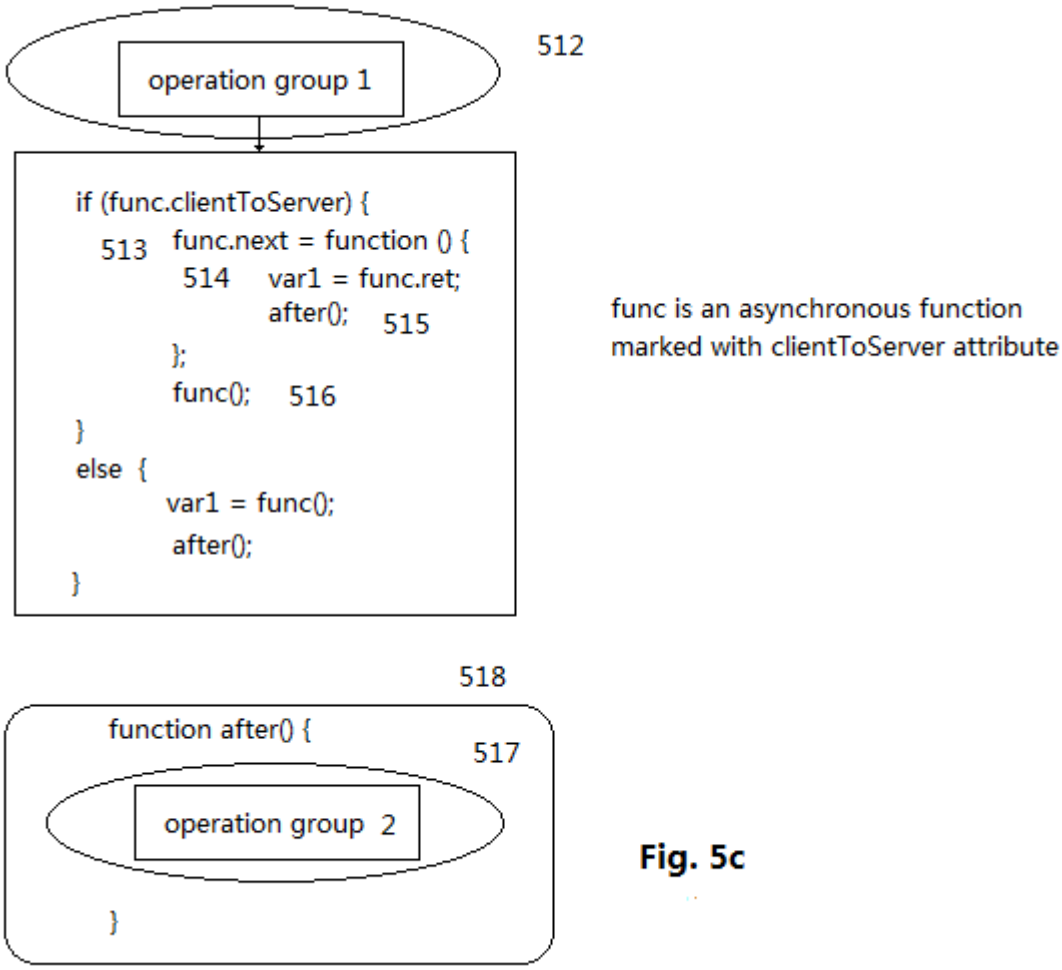


Fig. 5c

A “client server automation” preprocessor processes a client side function involving server operations in the following way.

1. Referring to Fig. 5a, a preprocessor processes the programming of said function and generates code 503 and 504 inside the function;
2. the preprocessor sets an attribute of the function to indicate that the function involves asynchronous server operations, said attribute is referred to as “run type attribute”, see code 501;
3. the preprocessor adds code 506 to the code 505 generated by the preprocessor for the function, the code 506 is to be appended to the end of the last server connection callback function 504,
4. the code to be appended includes assigning return value of said function to an attribute of the function, which is referred to as “return value attribute”, see code 507, and checking an attribute, which is referred to as “next function attribute”, of the function to see whether a function is assigned to said “next function attribute”, if a function is assigned to said “next function attribute” then the assigned function is executed, see code 508;
5. in a case of enabling multiple simultaneous calls of said function, said process assigns the “next function attribute” to a local variable at the beginning of the function and execute the local

variable at the end of last callback function instead of executing the “next function attribute” as mentioned above.

6. Referring to Fig. 5b, see code 510, to process a call to a function which has a “run type attribute” to indicate that the function involves asynchronous server operations, which is referred to as “asynch function”, see func in 510, if return value of calling the “asynch function” is assigned to a variable then the variable is referred to as “ret variable”, see var1 in 510; said process generates a function referred to as “after function”;
7. Referring to Fig.5c, the preprocessor processes programming following the call to the “asynch function” and places generated client side code 517 inside “after function” 518;
8. The preprocessor generates a function which is referred to as “next function”, see code 513,
9. code inside the “next function” includes assigning value of “return value attribute” of the “asynch function” to the “ret variable”, see code 514, and a call to “after function”, see code 515;
10. the preprocessor generates code to assign the “next function” to the “next function attribute” of the “asynch function”, see code 513, and followed by a call to the “asynch function”, see code 516;
11. if a calling of an “asynch function” is within an expression then a new variable is created to replace the calling of the “asynch function” in the expression, the new variable receives return value of the “asynch function” before evaluation of the expression

VIII. Client Server Automation Environments

Basic Structure

A “client server automation” enabled execution environment can be implemented by a “client side dispatcher” and one or more “server side dispatcher”, as shown by Fig. 6

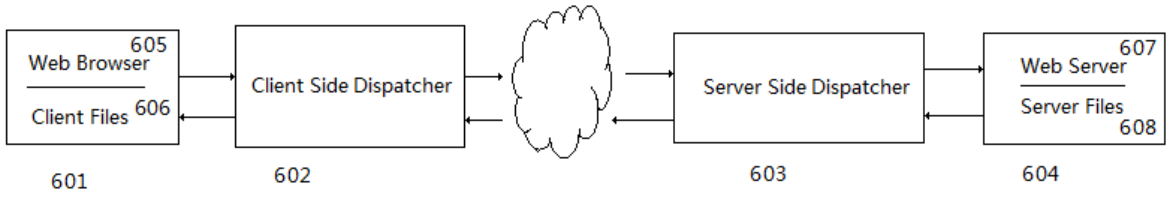


Fig. 6

Client Files 606 and Server Files 608 include files generated by a “client server automation” preprocessor by processing code developed by programmers.

A “client side dispatcher” 602 works at a client computer 601 and is used by client files 606 which is running in a web browser 605.

A “client side dispatcher” 602 is responsible for following tasks:

1. making server connections with a “server side dispatcher” 603,
2. uploading client data and server code to server,
3. processing data downloaded from server,
4. using downloaded values to imitate a code execution scope so that the downloaded values are accessible by callback functions,
5. and invoking callback functions

A “server side dispatcher” 603 works at a server computer 604, and it is connected by a “client side dispatcher” 602 through a web server 607, and invokes server files 608 to execute server functionality requested by client.

A “server side dispatcher” 603 is responsible for following tasks:

1. processing uploaded client values,
2. using uploaded values to imitate a code execution scope in the programming so that the uploaded values are accessible by server code to be executed,
3. executing server code specified by the server connection made by the client side dispatcher,
4. providing an application programming interface for server code to add download values to server response,
5. forming server response to client request

A “client side dispatcher” can be implemented in JavaScript. It is unrelated to how “server side dispatchers” are implemented.

A “server side dispatcher” can be implemented in various server languages.

Server Dispatcher by ASPX

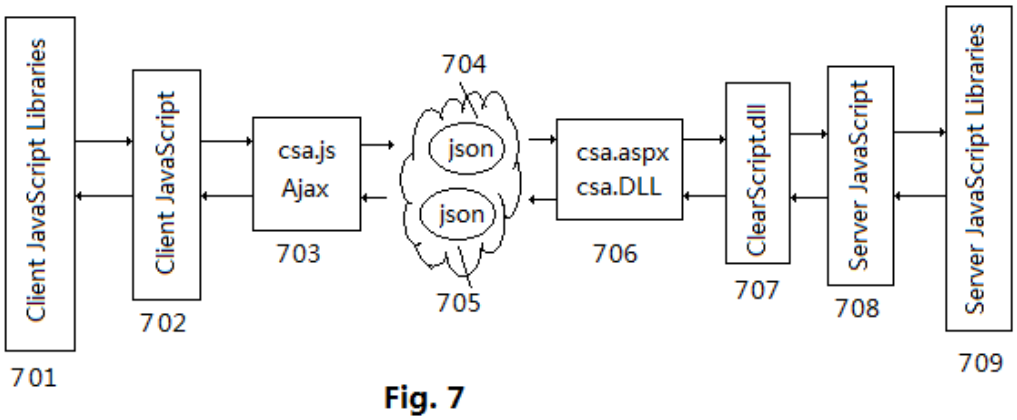


Fig. 7

In this arrangement, a server dispatcher 706 is implemented in Microsoft .Net Framework as a DLL and an ASPX web page file to be connected via Ajax by a client side dispatcher 703, which is implemented as a JavaScript file csa.js.

The client side dispatcher 703 sends data to the server side dispatcher 706 via Json 704.

The server side dispatcher 706 uses ClearScript.DLL ([4]) to execute server side JavaScript 708, which in turn calls server side JavaScript libraries 709 to executed required server functionality.

The server side dispatcher 706 sends data to the client side dispatcher 703 via Json 705.

Client JavaScript 702 and Server JavaScript 708 include files generated by a "client server automation" preprocessor by processing code made by programmers. Client JavaScript Libraries 701 and Server JavaScript Libraries 709 represent client side functionality and server side functionality available to the programmers.

Server Dispatcher by PHP

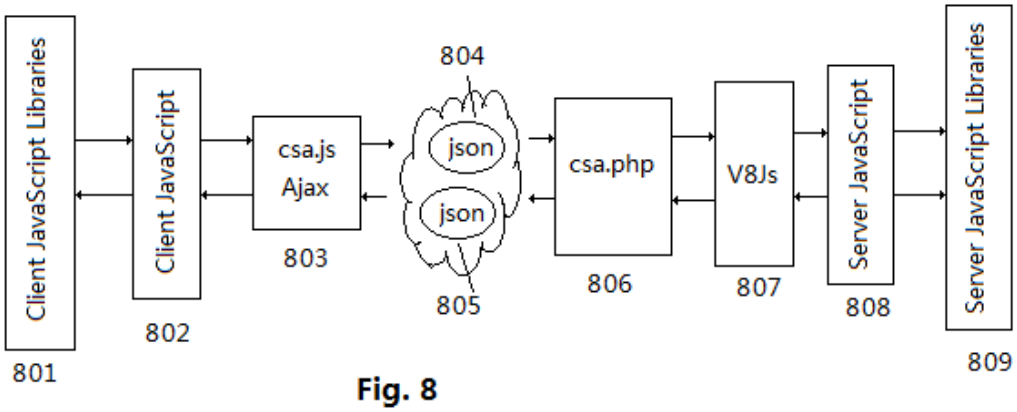


Fig. 8

In this arrangement, a server dispatcher 806 is implemented by a PHP file csa.php, which uses V8Js ([1], [2], [3]) to run Server JavaScript 808 which uses Server JavaScript Libraries for server functionality.

The client parts 803, 802 and 801 are the same as 703, 702 and 701 in Fig. 7.

Server Dispatcher by Node.js

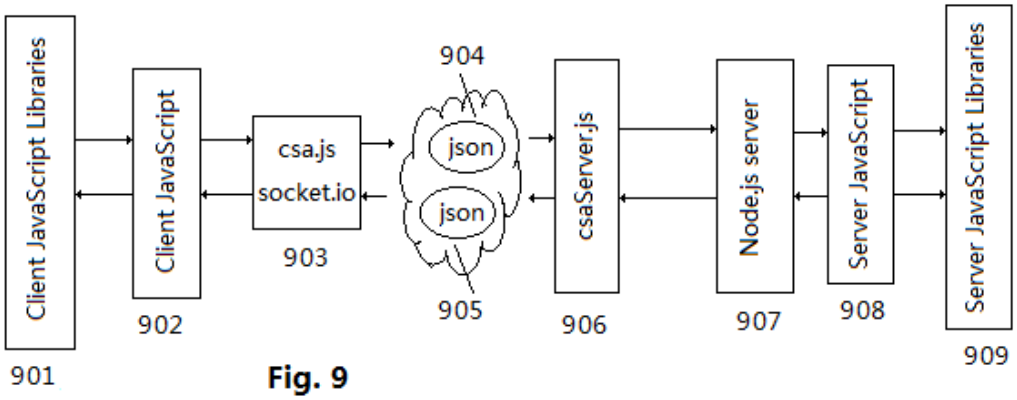


Fig. 9

In this arrangement, a server dispatcher 906 is implemented by a Node.js web server csaServer.js, which runs by a Node.js server 907.

The other parts are the same as in Fig. 7 and Fig. 8.

IX. Server-Technology-Independent API

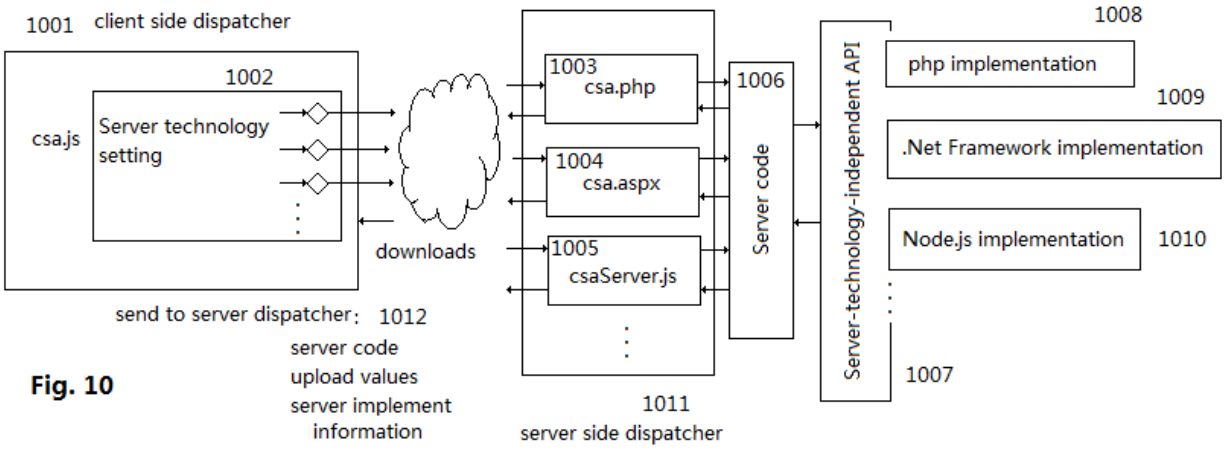


Fig. 10

To totally take over client server related programming issues, a programmer should be shielded from server implementations. A “Server-technology-independent API” can be used for this purpose. It also helps implementing a “client server automation” preprocessor; the preprocessor generates code without targeting specific server technology.

A “Server-technology-independent API” is represented by a set of JavaScript files defining JavaScript objects for server side functionalities.

To help “client server automation” preprocessor to processing code made by programmers, a JavaScript object represents a part of “Server-technology-independent API” may include attributes to indicate 1) it is a server object; 2) server implementation information for server functionality. For example, suppose a “Database” object represents database API; it may include following two attributes:

```
Database.prototype.RunAt = true;//run-at flag to indicate a server object

Database.prototype.ServerTypes = ['Database.DbExecuter'];//server implement information
```

- Note that the above two attributes in the API are for the preprocessor, not used by programmers.
- The “RunAt” attribute helps the preprocessor to identify server operations and client operations.
- Included among information 1012 uploaded to a server side dispatcher is the “ServerTypes” attribute so that a server side dispatcher can load needed server files to get required server functionality.
- Each server implementation 1008, 1009, 1010 and other implementations should implement server functionality in a way that at runtime correct implementation can be loaded based on information represented by “ServerTypes”.
- Each server side dispatcher interprets “ServerTypes” attribute to load correct server implementation. For example, for an ASPX implementation, ‘Database.DbExecuter’ can be a namespace for database related classes; for a PHP implementation, ‘Database.DbExecuter’ can be a PHP file name; for a Node.js implementation, ‘Database.DbExecuter’ can be a server side JavaScript file name; etc.

Referring to Fig. 10, a “client side dispatcher” 1001 uses a “Server technology setting” 1002 to indicate desired server technology. At a time of making AJAX call or a web socket call, the “Server technology setting” 1002 may direct the AJAX connection call or a web socket connection to connect to csa.php 1003, csa.aspx 1004, csaServer.js 1005 or another server dispatcher depending on the desired server technology.

The connected server dispatcher interprets uploaded data and code and invokes server code 1006. Server code 1006 is generated from programmer code by the preprocessor. Server code 1006 invokes server functionality implementation through “Server technology independent API” 1007, which invokes correct implementation 1008, 1009, 1010, or another implementation.

X. Conclusion

Below pros and cons of “client server automation” are summarized.

Pros:

- Client/server related programming tasks are all removed. A programmer does client/server programming in the same way as doing local programming. The client/server programming is thus made effortless.

- Asynchronous programming tasks are removed. A programmer focuses programming on fulfilling business logic instead of designing asynchronous executions and callback functions. Application programming becomes more intuitive, simple and code maintainability is much enhanced.
- Client/server data exchanges are arranged by the preprocessor, not by programmers. It can be made more efficient and less bug-prone. For example, the View State might slow down a web page for seconds and even minutes if it is not used properly, see [16].

Cons:

- For a programmer without any client/server considerations, un-optimized code may be programmed. For example, suppose a programmer mixes several client operations and server operations together in a coding, it causes the "client server automation" preprocessor to generate several network connections based on the coding. But an experienced programmer may put server operations together when possible and the "client server automation" preprocessor will generates less server connections. The "client server automation" preprocessor may do some optimizations for the programmer. More researches are needed in this area.
- Without modifying a JavaScript engine or a web server, a programmer needs an extra step of running a preprocessor utility before testing and distributing programming work. It is as if JavaScript was not an interpreted language. Preferably "client server automation" can be added to JavaScript engines.

This paper shows that "client server automation" can be added to JavaScript language. It can also be added to other languages. For example, it can be added to Object-C compiler so that mobile phone applications can be developed as local applications.

References

[1] The PHP Group, V8 JavaScript Engine Integration, 2015, http://php.net/manual/en/intro.v8js.php

[2] Black Duck Software Inc., V8Js Project Summary, 2015, https://www.openhub.net/p/v8js

[3] GitHub Inc., V8 JavaScript Engine for PHP, 2015, https://github.com/preillyme/v8js

[4] Microsoft Corporation, ClearScript, 2014, https://clearscript.codeplex.com/

[5] Joyent Inc., 2015, https://nodejs.org/

[6] L.E. Heindel, V.A. Kasten, Highly reliable synchronous and asynchronous remote procedure calls, Mar 1996, Conference Proceedings of the 1996 IEEE Fifteenth Annual International Phoenix Conference on Computers and Communications, 1996.

[7] Microsoft Corporation, How to: Create Asynchronous Web Service Methods, 2014, https://msdn.microsoft.com/en-us/library/vstudio/98t3s469(v=vs.100).aspx?cs-save-lang=1&cs-lang=csharp#code-snippet-1

[8] Henri Chen, Robbie Cheng, ZK : Ajax without JavaScript framework, 2007, Berkeley, CA : Apress ; New York : Distributed to the Book trade worldwide by Springer-Verlag New York, 2007.

[9] Apple Inc. (2015) Introduction to JavaScript for Automation Release Notes, https://developer.apple.com/library/mac/releasenotes/InterapplicationCommunication/RN-JavaScriptForAutomation/Articles/Introduction.html

[10] S Ducasse, A Lienhard, L Renggli, Seaside: A Flexible Environment for Building Dynamic Web Applications, IEEE Software, v24 n5 (Sept.-Oct. 2007): 56-63

[11] Stefan Fidanov, How to Elegantly Solve the Callback Hell of Node.JS and Express with Async.js (2015) https://www.terlici.com/2015/10/28/solving-node-callback-hell-asyncjs.html

[12] Brian Di Palma, Callback hell is a design choice, (2013) http://blog.caplin.com/2013/03/13/callback-hell-is-a-design-choice/

[13] Werner Schuster and Dio Synodinos, Virtual Panel: How to Survive Asynchronous Programming in JavaScript (2011), https://www.infoq.com/articles/surviving-asynchronous-programming-in-javascript

[14] Markku Rossi, J arvenpaa (Fl); J ukka Raanamo, Helsinki (Fl), METHOD FOR INVERTING PROGRAM CONTROL FLOW (2002), US Patent Pub. N0.: US 2002/0166000 A1, Pub. Date: NOV. 7, 2002

[15] Nathan T. Freeman, Colin MacDonald, Tim Tripcony, Automatic synchronous-to-asynchronous software application converter (2012) US Patent Publication number: US20120079463 A1, Publication date: Mar 29, 2012

[16] Scott Mitchell, Understanding ASP.NET View State (2004). http://msdn.microsoft.com/en-us/library/ms972976.aspx

[17] http://callbackhell.com/

[18] Mozilla Developer Network, Promise (2016) https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

[19] Mozilla Developer Network, function* (2016) https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function*