

JAMScript: A Coordination Language for Clouds and Internet of Things

Muthucumaru Maheswaran
School of Computer Science
McGill University

April 17, 2015

1 Overview

The purpose of JAMScript is to create a programming language system for Internet of Things (IoT). IoT can have highly heterogeneous configurations ranging from highly resource constrained devices to fully fledged computing nodes. Therefore, in most deployments, the ‘things’ that constitute the tiny devices cannot perform the required computations and storage all by themselves. This means programming IoT often involves programming a distributed collection of computing elements including the things and cloud-based virtual computers.

The JAMScript presents a *single* node programming model to the programmer that will allow programming IoT and Cloud without explicitly dealing with program distribution and data transfers among the different computing elements. According to [1], a programming system is made up of a coordination language and a computation language. The coordination language is responsible for controlling the computation activities while the computation language is responsible for creating the code for performing the computations.

In JAMScript, C and JavaScript are used as the computational languages. The C language is called upon to perform the computations very efficiently (i.e., with minimal overhead). So, C is suitable for programming the low-power devices or for running computationally intensive tasks in cloud supplied resources. JavaScript is suitable for tasks that could be hooking up to other services or are not computationally intensive.

New constructs in JAMScript are introduced to coordinate the execution of the C and JavaScript components. In [1], Gelernter and Carriero argue that a computational language needs coordination to perform its activities. Because a computational language like C does not have facilities such as process creation, it relies on the ad hoc mechanisms provided by the operation systems (e.g., `fork()` system calls).

2 JAMScript Design

According to Papadopoulos and Arbab [2] coordination languages can be either data-driven or control-driven. The data-driven approaches for coordination create a shared dataspace through which the computation processes can communicate with each other. With data-driven coordination, the coordination language is explicitly involved in managing the data interaction between the computational processes. In the control-driven coordination approach, the coordination language provides constructs for managing the control flow among the processes without any concern for the data exchanges or the structure of the data being exchanged. The JAMScript is a hybrid coordination language that provides both data-driven and control-driven constructs.

To perform the control-driven coordinations, JAMScript introduces a notion called *activity*. An activity is a piece of computation that can either run exclusively on the cloud, thing, or straddle both. By default an activity runs asynchronously from the main thread of the JAMScript application. However, JAMScript provides constructs that can be used by the programmer to specify different execution patterns for the activities. The activity is spawned by the main thread of the JAMScript program, which itself could be running on the thing or the cloud. So an activity could be running on the cloud or thing.

The constructs provided by JAMScript can be categorized into two groups: (i) activity definition and (ii) activity control. Listing 1 shows the activity definition construct. An activity is made up of C and JavaScript program segments that are glued together by the JAMScript constructs. The JAMScript constructs specify the control flow among the different segments and also the format of the messages that are being passed between them. To maintain familiarity, we reuse C function declaration statements in the JAMScript constructs along with new additions introduced by the JAMScript language.

An activity can have many different types of code segments. As shown in Listing 1, the primary segment of the activity is specified using the `jamdef` keyword followed by a C function declaration. An optional *in* clause is used to specify the name space in which the activity should be posted. The primary segment of an activity could be defined in two different ways: using JavaScript statements for its entirety or C statements for its entirety. When the primary segment is defined using JavaScript, it compiles to a JavaScript function and a C stub function is inserted into the C program to invoke the newly created JavaScript function from the C side. On the JavaScript side, this function is posted in a prespecified namespace location (as specified using the optional *in* clause) and can be accessed by other JavaScript functions. When the primary segment is defined using C, a wrapper function is generated in JavaScript that can be used to invoke this function. Again the wrapper is made available in the namespace used to post other JAMScript generated JavaScript functions.

An activity can have several different types of blocks associated with it besides the primary block. A primary block specification can list a set of arguments that are passed-by-value to the block at its invocation. The pass-by-value mechanism in JAMScript is different from the one in C. In C, with pointer arguments a function has the ability to manipulate values in the calling scope. In JAMScript, however, even pointers cannot get back to the values in the outer scope because the JAMScript runtime makes a copy of the invocation parameters such that the same values are available in case the activity needs to be restarted. Although the parameters are detached from the ones in the outer scope of the function, they are not immutable. The changes made to the formal parameters during an activity remain available for other blocks until the activity completes.

The *oncomplete* block is the one that is executed by the JAMScript runtime on the successful completion of the primary block of the activity. Like other blocks of the activity, the *oncomplete* block also has access to the formal parameters that were passed into the primary block. In addition, the *oncomplete* block has access to the return parameters of the primary block. The arguments specified as the formal argument of the *oncomplete* block should match the return parameter of the primary block. One unique aspect of the *oncomplete* block is its capability to affect the variables in the outer scope of the activity.

An activity has several blocks and only *oncomplete* is guaranteed to run on the same node as the calling function (could be the main thread of the JAMScript program or an activity itself). So activity starts with the main thread of JAMScript and ends at the main thread. We reduce the “coupling” between the activity and main thread by using a form of *snapshot isolation* to copy the formal parameters into the activity memory. Once the activity is in the *oncomplete* block we allow the activity to *synchronize* its copies of the memory variables with the corresponding copies in the main thread using the `jamsync()` function.

The arguments specified in the declaration of the *onreturn* block should match the return value of the primary block because the *onreturn* gets its parameters values from the return values of the primary block.

```
1 jamdef rettype c_function_decl(args) [ in name_space ] {
```

```

2
3 // (a) JavaScript code here or (b) C code here
4 // this is the primary implementation for the c_function declared
5 // in the above prototype
6 // this runs on the primary device (the device that runs the primary impl.)
7
8 // return appropriate value that matches "rettype"
9
10 } [ oncomplete [ optional_function_name ] (ret-type-args) {
11
12 // (a) C code here or (b) JavaScript code here
13 // this part is optional and runs in the complementary device
14 // to the device that runs the primary implementation
15
16
17 } ] [ oncatch (jamexception e of typeX) [ optional_name_for_exc_handler ] (
    args_for_primary, error_info) {
18
19 // Retry the computation under the condition that exception of typeX occurred
20 // when the primary implementation ran on the primary device.
21 // There could be many different retry blocks corresponding to different
    exceptions.
22
23 } ] [ oncancel [ optional_name_for_cancel_c_function ] (args) {
24
25 // Perform the operations to cancel the operations performed by the primary
26 // and/or the retry blocks.
27
28 // We need provide functions to detect which branches were already taken: primary
    alone
29 // or primary + retry
30
31 } ]

```

Listing 1: Syntax of the activity definition construct

We can compose using jamcall statements following at least two different patterns: iterative and recursive. In the iterative pattern, we make a jamcall blocking. Once a jamcall statement has completed, we can launch the second jamcall by putting them into a looping structure provided by the computational language. In the recursive pattern, the jamcall statements are chained such that the onreturn block of a jamcall statement can invoke the same jamcall statement or other jamcall statements. The recursive calls will end depending on the terminating condition.

```

1 jamcall void c_function_decl(args) [ at ( node_spec ) ] [ scope ( any|all ) ]
2 [ async ( true ) ] {
3
4 // Activity calls are here.
5 // An activity is called by invoking the corresponding c_function with parameters
6 // activities get launched in parallel at the target. If multiple nodes are
7 // specified, all activities listed here get launched on each node.
8
9 // The activity definition specifies the return parameter of each activity
10 // on successful completion we get a value of the specified type or we get
11 // NULL in the case of an error.
12
13 // The return "value" of the jamcall is an array of values (lets say we have m
    statements).
14 // Each value corresponds to an activity.
15

```

```

16 } [ onreturn [ opt_func_name ] (arg_1, arg_2, arg_3, ... arg_m, char *status) {
17
18     // Once the activities we have launched by the call-block have successfully
19     // completed
20     // or generated exception, we are here. If ANY is specified for the scope, then we
21     // can
22     // be here if at least one node has completed the activity or all nodes have
23     // generated
24     // the exceptions. If ALL is specified for the scope, we are here only when all
25     // nodes
26     // have reported success or generated exceptions.
27     // In the case of ANY, some pending activities will be cancelled
28 } ]

```

Listing 2: Syntax of the activity control construct

```

1 jamdef double trigfunc(int a, double x) {
2
3     // JavaScript code inside here..
4     require ('jamlib');
5     try {
6         res = a / sin(x);
7
8     } catch (e) {
9         if (e instanceof RangeError) {
10             // 105 here is just arbitrary code.
11             // We could even extract a code from e
12             jamexception(105, "RangeError");
13         }
14
15     }
16     return res;
17
18 } onreturn printresult(double rval) {
19
20     // This is actually C code!
21     printf("Returned value %f", rval);
22
23 } oncatch (jamexception e) double error_redo(int a, double x, int code, char *msg) {
24
25     // In this particular example, redoing does not make any sense.
26     // See NOTE below
27
28     console.log("Error! Evaluating the trig function");
29     return 0.0;
30 }

```

Listing 3: An activity definition example

3 Composing Activities

4 Building Fault Tolerance

References

- [1] D. Gelernter and N. Carriero, “Coordination Languages and Their Significance.” *Communications of the ACM*, vol. 35, no. 2, pp. 96–107, 1992.
- [2] G. A. Papadopoulos and F. Arbab, “Coordination Models and Languages.” Elsevier, 1998, pp. 329–400.