JAMScript: A Coordination Language for Clouds and Internet of Things

Muthucumaru Maheswaran
School of Computer Science
McGill University

May 10, 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 asychronously 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 by 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 it entirity or C statements for its entirity. 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 can have several blocks and *oncomplete* is one of the blocks that 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.

An activity is meant to carry out a computational task as codified by its blocks on the things and cloud. In carrying out the specified computational task, the activity can encounter many erroneous conditions that can trigger exceptions. To handle such exceptions, the activity can have many *oncatch* blocks. Like

the *oncomplete* block, the *oncatch* blocks run at the computing node that launched the activity. However, the remedial action codified in the *oncatch* block could end up getting executed remotely and could trigger its own exceptions.

```
1  jamdef rettype c_function_decl(args) [cloneable] [ 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 occured
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

The last block of the activity is the *oncancel* block. An activity that has not yet started executing the *oncomplete* block can be cancelled by invoking the *jamcancel()* function on its handle. When a cancellation request is sent to the activity, the *oncancel* block of the activity will be invoked. For the cancellation to succeed, the *oncancel* block must return true. Activities that do not have the *oncancel* block defined by the programmer cannot be cancelled. If the *oncancel* block triggers an exception or returns false, the cancellation request will fail. The *oncancel* block can be used by the programmer to undo the side-effects of the operations carried out by the activity up until the point of cancellation.

Another important aspect of a JAMScript activity is whether it is cloneable or not as specified in Line 1 of Listing 1. If the `cloneable` option is specified for an activity, the activity can be cloned to increase fault tolerance. Activities that do not have external side-effects such as changing mutable state can be declared as cloneable. The ultimate responsibility of declaring an activity as cloneable or not lies with the programmer. The JAMScript compiler and runtime do not provide any guidance in this regard.

The simplest way to invoke an activity is to call it like any other C function as shown in Listing 2. The definition of the `trig_func()` invoked in Listing 2 is shown in Listing **??**. At completion, the activity `trig_func()` is supposed to return a double value. However, in Listing 2 the invocation of

`trig_func()` returns a handle to the JAMScript activity that was started by the invocation. Using the handle, we can cancel (Line 8), get the status (Line 10), or wait until the completion of the activity (Line 12). This simple form of JAMScript activity invocation already allows different ways of composing the activities. For example, we can chain the activities such that Activity B starts after Activity A completes. One way of codifying the dependency is to invoke Activity B in the *oncomplete* block of Activity A. Another way of codifying the pattern is to call `jamwait()` after invoking Activity A to wait for its completion and start Activity B afterwards.

```
1  #include <jamscript.h>
2  ...
3
4  jamactivity_t *act = trig_func(2.5, 10);
5  ...
6  // Illustration of the different actions on the handle
7  if (action == CANCEL)
8      cancelled = jamcancel(act);
9  else if (action == STATUS)
10     jamstatus_t stat = jamstatus(act);
11 else if (action == WAIT)
12     jamwait(act)
```

Listing 2: Simple invocation of a JAMScript activity

Once the handle is returned by the JAMScript activity invocation, we can use it to perform many control actions on the activity. The `jamcancel()` is a function provided by the JAMScript library to cancel an activity that is in progress. Calling `jamcancel()` on a handle that is already completed will return an error code (JAMCANCEL_FAILED). Similarly, we can determine the status of a JAMScript activity by calling `jamstatus()` on the handle of an activity. The status check returns a structure with two status indicators: first indicator is the state of the activity (RUNNING, WAITING, ERROR, COMPLETED) and the second indicator is the current location (PRIMARY_BLOCK, COMPLETE_BLOCK, EXCEPTION_BLOCK, CANCEL_BLOCK). Due to sensing delays, the status reported by `jamstatus()` may be erroneous.

The `jamwait()` function provides a way to wait for the completion of an activity. The `jamwait()` function is implemented using semaphores provided by the underlying operating system. If `jamwait()` is called on an activity that is already complete, the function would return immediately. The `jamwait()` call will fail if (a) the activity has already failed (i.e., generated an exception) or (b) the activity fails while `jamwait()` is waiting on it. In such a failure situation, the `jamwait()` call will return with the error code JAMWAIT_FAILED.

The simple coordination patterns discussed above are rather limiting in the ways in which they could control the activities. In particular, we need flexibility in controlling multiple activities. The `jamcall` construct given in Listing 3 is a more general way of coordinating the execution of activities. Before using the `jamcall` construct to coordinate the execution, we need to define the corresponding activites using the `jamdef` construct. Like the `jamdef`, the `jamcall` takes a primary block and an *oncomplete* block.

The `jamcall` specification has three important parameters: (a) node specification, (b) mapping method, and (c) blocking behavior. The node specification is the set of computing elements on which the activities should be executed. The addresses of the target computing elements are assumed to be obtained by some other means such as a call to the underlying runtime. The mapping method is one of the three: *on-all*, *on-one*, or *custom*. With *on-all*, each activity listed in the primary block of `jamcall` is executed on each node. So an activity is mapped to all the nodes (or computing elements) listed in the node specification. With *on-one*, each activity is mapped to precisely one node. If the number of activities exceed the number of nodes, then some nodes would get more activities and others would receive less. The exact number of activities mapped onto a computing element also depends on the processing speed of the computing element. Faster devices that complete the activities quicker can be mapped with more tasks. The last pa-

rameter specifies whether the construct is blocking or non-blocking. By default, if nothing is specified, the `jamcall` construct is blocking.

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.

```
jamcall void c_function_decl(args) [ at ( node_spec ) ] [ map ( on-all | on-one | custom ) ]
                                   [ async ( true ) ] {

    // Activity calls are here.
    // An activity is called by invoking the corresponding c_function with parameters
    // activities get launched in parallel at the target. If multiple nodes are
    // specified, all activities listed here get launched on each node.

    // The activity definition specifies the return parameter of each activity
    // on successful completion we get a value of the specified type or we get
    // NULL in the case of an error.

    // The return "value" of the jamcall is an array of values (lets say we have m statements).
    // Each value corresponds to an activity.

} [ oncomplete [(any|all)] [ opt_func_name ] (arg_1, arg_2, arg_3, ... arg_m, char * status) {

    // Once the actitives we have launched by the call-block have successfully completed
    // or generated exception, we are here. If ANY is specified for the scope, then we can
    // be here if at least one node has completed the activity or all nodes have generated
    // the exceptions. If ALL is specified for the scope, we are here only when all nodes
    // have reported success or generated exceptions.
    // In the case of ANY, some pending activities will be cancelled

} ]
```

Listing 3: Syntax of the activity control construct

```
jamdef double trigfunc(int a, double x) {

    // JavaScript code inside here..
    require ('jamlib');
    try {
        res = a / sin(x);

    } catch (e) {
        if (e instanceof RangeError) {
            // 105 here is just arbitary code.
            // We could even extract a code from e
            jamexception(105, "RangeError");
        }

```

```
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 4: An activity definition example

3 Composing Activities

4 Building Fault Tolerance

5 Questions/Concerns on the Document

1. There is confusion on what runs where in terms of the blocks of an activity. We need some clarity over there.
2. How recursive could the exceptions go? For instance, could we have a chain of exceptions?
3. Should we incorporate jamcancel, jamstatus, and jamwait into the language? At this point they are part of the library. Can we redesign things a bit?
4. Can we introduce an *onreport* construct into the activity? Using such a construct we can provide a constant stream of updates. This is very similar to Linda. We send a message and put a handler for the incoming message. The restriction is that all of the transactions have to be within an activity.

   We need to reconcile how the global (main thread) variables are going to be affected. We keep the activity variables isolated from the main thread until it reaches the *oncomplete* block.
5. We have a problem with the *oncomplete* block in `jamcall`. We can have many nodes running the primary block – i.e., the activities could be running at many different nodes. So each activity can have n results. The oncomplete block has a parameter for each activity. How do we handle the n number of copies? If it is just an array of values, then it might mess up the program. Can we somehow keep that hidden from the programmer? Can we examine one result at a time? What is the best solution?

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.