A Model Based Framework for Effective Web of Things Development

Roberto Manione
Media on Line, http://www.taskscript.com
Torino, Italy
roberto.manione@gmail.com

Abstract— This paper discusses the adoption of a Model Based approach in the Web of Things development as a way to simplify the task. A set of primitives is introduced to handle the http protocol at the model level directly so that application designers need not know programming and networking details.

A Model Based Integrated Development Environment, TaskScript, has been extended with such primitives: first results are reported, demonstrating the feasibility and effectiveness of the approach, particularly with OS-less systems using low cost and low energy consumption 8-bit microcontrollers: real case applications have been effectively developed in matter of hours.

Keywords—Internet of Things; Web of Things; Reactive Systems; Model Based Design; Automatic Code Generation; IDE

I. INTRODUCTION

The idea discussed in this paper is the development of WoT devices via a Model Based approach. This allows designers to develop their WoT devices thinking at graphic level only, leaving the software to take care of the implementation details, such as code generation, Interrupt handling, scheduling of parallel tasks, protocol handling, and so on. The http protocol, like others, has been abstracted and exposed at the Model level. Thanks to this abstraction, application programmers gain enough visibility and control on what happens at the network interface avoiding to learn the majority of the technical details.

II. MODEL BASED DESIGN OF REACTIVE SYSTEMS

Model Based design is an approach that allows designers to develop their system through the instantiation and connection of primitives chosen from a predefined set. Such a set, i.e. the modeling Language, is defined according to analogies with the real domain at hand, in such a way that designers can feel comfortable in using the primitives, thinking they are using the real components.

WoT devices can be modeled as Reactive Systems [1] having one more class of channels for interaction with the periphery, that is, the http interface. Such a choice leads to a broad class of powerful devices: their local processing capability, intrinsically parallel, combined with the http communication capability, allows implementing systems with two kinds of control loops acting at the same time:

- **local control loop(s)**, with low and predictable times, to observe and control local signals for regulation purposes;
- **remote control loop(s)**, to update a cloud-based logic, about the controlled process(es) and, possibly, to receive updated control constants.

In the reminder of this section a Model Based IDE targeting the Reactive Systems domain, TaskScript [3], is introduced, to discuss the http abstraction. The TaskScript **Model Based language** is a combination of two *graphical declarative languages* allowing to easily and exhaustively express:

- global behavior of the system, expressed as **control flow**: it allows to easily define sophisticated flows of control with arbitrary threads of parallelism, through a graphic formalism close to the GRAFCET language [2], in turn derived from Petri Nets, and first standardized as EN 60848. The control flow is a digraph where nodes are generalized states and arcs are guarded transitions; one transition fires iff the state upstream of the transition is enabled and its guard is true. After firing, the upstream state is deactivated, while its downstream one is activated; “forks” and “joins” are available, to allow one transition to activate and deactivate more states at the same time.
  To enhance the language expressivity, states have 4 phases: notActive, Active, onEntry, OnLeave.
- local behavior of each state in the system, expressed as transformational logic (i.e. the **data flow)** upon signals (i.e. variables) in effect for each and every state when in Active, onEntry, onLeave phases respectively (up to 3 different data flows can be provided, one for each state phase). The graphic formalism is straightforward as it uses logical operators (i.e. &, |, ^) for Boolean variables and arithmetic operators (i.e. +, -, *, /, %) for arithmetic ones.

To enhance the language power, several data flow primitives have been defined with own internal state; they are counters, delays, edges, communicators (both point to point and network ones). Each primitive reads one Boolean input, *enable*, and produces one output, *done*, meaning: the primitive keeps in its reset state as soon and as long as its *enable* input is false; when this input goes to true it starts working; as soon and as long as it reaches its final state the *done* output is set to true; should *enable* revert to false before successful termination, it aborts its job, going back to its reset state.

Systems described in the TaskScript language are implemented by the built in compiler and code generator according to a cooperative multitasking model; this produces a

really efficient code executable by a simple multitasking microkernel (i.e. no need for context switching & preemption).

One key property of the above described modeling Language is that it is **self contained**: in other words a TaskScript model is an exhaustive representation of a Reactive System; there is no need to review the generated code.

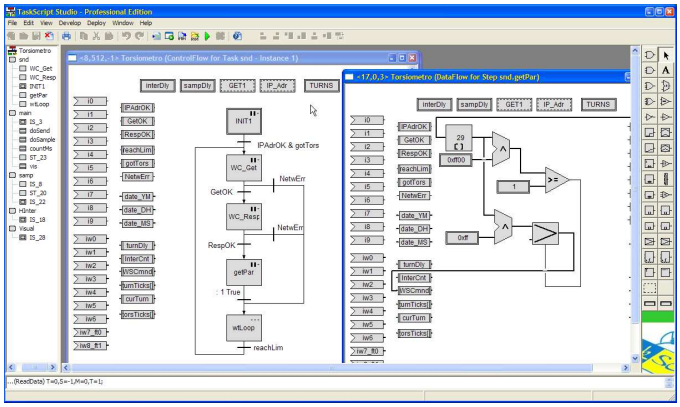

Fig. 1. Screenshot of the TaskScript Studio IDE (with http client example)

III. Modeling the HTTP interaction

Http interactions are abstracted through a set of *network communicator* primitives, with internal state: WsGet, WsResp, WcGet, WcResp which handle transmission and reception of data in the local buffer to/from the remote recipient. They work in pairs, each one dealing with one direction of the transmission, on the same channel (i.e. the socket); in detail:

- **Server role**: WsGet waits for a client to open a socket and send its http query; if the path in the query string matches the expected one the arguments (if any) are returned in the buffer; the primitive then sets its *done* to true to notify its successful termination; however the socket is kept open to send the reply; WsResp formats the data in the local buffer and sends back the response on the same socket; then it closes the socket and sets its *done* to true.
  Client role: WcGet opens a socket to the requested URL and sends its http query, appending the data passed in the local buffer (if any) to the query string; the primitive sets its *done* to true, true to notify its successful termination (the socket is kept open to wait for the reply); WcResp waits on the same socket for the response and parses the received data into a local buffer, then closes the socket and sets is "done" to true.

For both roles the http request uses the get verb: variables passed in the query string can be formatted according to several options, such as Base64, integer, or hex; http responses are formatted as XML, JSON or JSONP fragments.

The picture below shows an example of http client: WsGet (→S) and WsResp (S←) are used, exploiting their respective *enable* and *done* control variables; the resulting behavior is that WsGet is active as soon as the netwEna variable becomes true and keeps active until it receives a valid query, in such a case it terminates activating WsResp through its done variable, which sends a response on the same channel; upon completion, the netwOk variable is set.

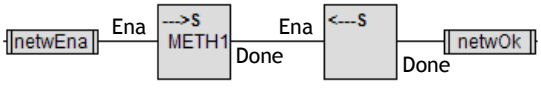

Fig. 2. http client role implementation as catenation of WsGet and WsResp

A dedicated Boolean variable is set in case of http failure (i.e. timeout) allowing the application to handle the problem.

IV. Discussion of results

Several http-Server and http-Client applications have been implemented with the described IDE: typically they observe and control signals, send actual readings to the remote end and update the regulation set points criteria from the remote end.

In some cases they are directly queried by e web browser implementing an HMI (in HTML/JQuery) allowing visualizing the control process state and changing the regulation set points; the number of states in the model is in the range 20 to 50.

The applications have been implemented through the TaskScript Studio IDE v1.7 and uploaded to a proprietary OS-less IoT board equipped with the Microchip® PIC18F87J608 bit microcontroller at 6.25 MIPS, pre-flashed with the TaskScript run-time (integrating the Microchip tcp/ip protocol stack), with Ethernet interface. The prototype sizes 3.2" by 2".

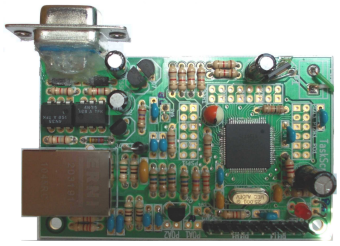

Fig. 3. TaskScript board prototype with PIC18F87J60 and Ethernet Interface

The development time for each application did not exceed one day, including debugging and simulation, carried out before deploying the generated code on the target board.

The IDE generated machine executable code for maximum efficiency; for the above applications the model originated code sized from 2 to 4 kBytes (without considering the TaskScript run-time and the tcp/ip stack).

Once deployed on the target board (via TFTP, integrated within the IDE) the models have been executed, interacting with both the physical devices and the http far end; the http interactions were scheduled periodically, in a 5 to 30 s range; sample and regulation loops iterated within 25 ms and 200 ms respectively. For the above board the kernel cycle time, i.e. the average time spent in the evaluation of all the non NotActive states once + kernel house keeping laid within 0.3 to 0.6 ms.

The average latency over http in a Server application answering to queries issued by a HMI running in a web browser was <10 mS, according to the Firebug statistics.

V. Conclusions

The abstraction of the http protocol within a Model Based framework has been discussed. The IDE environment implementing them has been used to design a number of test applications, each of which was completed in a few hours.

Performance evaluation on the implemented applications confirms the soundness of the approach and shows high efficiency, even using low cost boards equipped with an 8 bit Microcontroller in an OS-less environment.

The proposed approach can be a way to empower a wider population of users to implement and "publish" their WoT.

References

[1] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", Sci. Comput. Programming 8 (1987), 231-274

[2] René David et Hassane Alla, Du Grafcet aux réseaux de Petri, Paris, Hermès, coll. « Traité des nouvelles technologies / Automatique », 1992, 2e éd. (1re éd. 1989), 500 p. (ISBN 2-86601-325-5).

[3] The TaskScript web site http://www.taskscript.com.