Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 52 (2015) 210 – 217

Procedia Computer Science

6th International Conference on Ambient Systems, Networks and Technologies (ANT 2015)

Calvin – Merging Cloud and IoT

Per Persson*, Ola Angelsmark

Ericsson Research, Mobilvägen 1, 223 62 Lund, Sweden

Abstract

Developing applications for IoT and Cloud is difficult for a number of reasons; even without considering the inherent complexity of distributed computing, there are several competing platforms, programming languages and communication protocols. It can be argued that this is holding back the industry as a whole: Applications are difficult to write, deploy and manage. In this position paper we present *Calvin*, a hybrid framework combining ideas from the Actor model and Flow Based Computing. We show that by dividing applications into four well-defined aspects – *describe, connect, deploy*, and *manage* – we get an intuitive method for application development, and a flexible, distributed platform for deploying and managing applications. Additionally, we keep Calvin language and platform agnostic by only prescribing a lightweight runtime API, with a limited number of specified communication protocols – with no requirements on the carrier – for communicating with runtimes, between runtimes, and passing data between components.

Peer-review under responsibility of the Conference Program Chairs

Keywords: cloud computing ; internet of things ; IoT ; distributed computing ; application development ; position paper

1. Introduction

The current state of application development for the Internet of Things (IoT) is is best summarized as fragmented. Any developer starting out in the area needs to either learn a range of different communication protocols, platforms, and programming environments, or decide on which vendor specific solution is the least constricting. Should the developer also be interested in connecting the application to external devices or services, such as a cloud based database, there are yet more choices between more or less standardized solutions. Add to this the complexity of managing a distributed application, at initial deployment and during its life time, a distinctly non-trivial task, and it is clear that this situation is not sustainable. The effort required to build even modest sized applications can be daunting, and porting it to a new platform, or extending it with new functionality, is oftentimes not even feasible.

The goal of this position paper is to show that it does not have to be this way. We present Calvin, a novel way of building and managing distributed applications for the Networked Society[1] which, for an application developer, is more akin to building with LEGO® bricks than traditional programming. If the pieces fit together, then they will

* Corresponding author.
E-mail address: per.persson@ericsson.com

Peer-review under responsibility of the Conference Program Chairs
doi:10.1016/j.procs.2015.05.059

work together, and the developers need not worry about communication protocols, distribution, or other details. As is the case with LEGO®, where new bricks can be bought separately, or borrowed from other models, it is hoped that as Calvin matures, the full unconstrained creativity of developers can be reached with less effort, allowing them to innovate on top of next generation devices, networks and data centers.

2. Background

With cloud services becoming mainstream, and IoT on the verge of a major breakthrough, it has become increasingly clear that cloud and IoT applications are complex in nature and non-trivial to develop and manage, not to mention applications combining the two. Current solutions are therefore often built as "silos," single use applications developed in an ad-hoc, client-server like fashion, a business model that scales poorly.

We intend to show that restating the problems of distributed applications in a hybrid framework combining ideas from the Actors[2] and Flow Based Programming[3] paradigms is a valid way forward.

Our work on Calvin is based on our previous experience working with dataflow and actor-based paradigms for multicore systems which gave efficient solutions in terms of performance and resource demands, as well as leading to reduced development time compared to other approaches.[4,5] The basic idea behind Calvin is that a distributed system is very similar to a multicore system, albeit with (wildly) heterogeneous cores, and large differences in throughput and latency between them.

Our observations on the complications introduced by distributed systems are not new, and on a high level the proposed solution is not unlike Mobile Agents (MA) that were popular in academia during the 1990's, but did not make a lasting impact. In a paper on the state-of-the-art of MA from 2000, Gray et al. described MA succinctly[6]:

> A mobile agent is an executing program that can migrate, at times of its own choosing, from machine to machine in a heterogeneous network. On each machine, the agent interacts with stationary service agents and other resources to accomplish its task.

Lange and Oshima[7] wrote a concise piece on the benefits of MA in 1999 where they list the following pros of Mobile Agents; they 1) reduce the network load, 2) overcome network latency, 3) encapsulate protocols, 4) execute asynchronously and autonomously, 5) adapt dynamically, 6) are naturally heterogeneous, and 7) are robust and fault-tolerant. Although Calvin is not a Mobile Agent system, the goals we set out for the design were precisely the above, but with an important addition which is *improved developer efficiency*.

An important observation stated by both Lange and Oshima, and Gray et al. is that no applications *require* mobile agents[6,7], and considering the recent surge of connected devices, we agree that no applications *require* Calvin. It is, however, our sincere belief that neither IoT, nor the Cloud will reach its full potential of establishing the Networked Society without a platform addressing all of the above criteria, as well as letting application developers focus on their unique ideas rather than implementation details of networks, protocols, operating systems, and host hardware.

3. Calvin

Much of the power and flexibility of Calvin stems from dividing applications into four separate, well defined, aspects:

Describe The functional parts of applications, made into reusable components
Connect The interactions between components, described as graphs
Deploy The instantiation of the application according to the graphs
Manage The autonomous, dynamic, mapping of components to hardware over the lifetime of the application

To be able to span components ranging from small sensors with very little computing capacity to complex calculations performed in the cloud, as well as keeping Calvin language- and platform-agnostic, we only prescribe a lightweight runtime API, and few simple protocols for communicating with runtimes, communication between runtimes, and between components. At the time of this writing, we have a full-fledged prototype runtime written in python. However, there are a number of programming languages with support for distributed applications, which of course is most helpful when implementing a Calvin runtime. Candidates for future implementations include Erlang, for obvious reasons, and Scala, in order to gain access to the ubiquitous Java VM.

3.1. Describe

A central concept in Calvin is the *actor*. An actor is a reusable software component representing e.g. a device, or some aspect of it, a computation, or a service. Actors can only communicate with each other by passing *tokens* over *ports*. Data on input ports are processed by actors in atomic *actions*, representing consumption, processing, and production of tokens, and any tokens produced are passed to the output ports.

To write an actor, a developer describes the actions, their input/output relations, the conditions for a particular action to be triggered, and state the priority order between actions.

The requirement that actors only communicate via ports means that this is the only way the application can influence the state of an actor, which makes it possible to move, or migrate, an actor from one runtime to another. The speed of the migration then depends mostly on the size of the internal state, the computational power of the devices involved, and the speed of the transport. It is important to separate the actor from the *resource* it represents, and when we talk about actor migration, we mean the actor and not the underlying resource.

Calvin comes with a set of standard actors encapsulating fundamental operations such as I/O and file handling, some text processing and, of course, network communication. Typically, device manufacturers add value to their connected devices by supplying an actor with support for the device, enabling it to participate in a Calvin application. Similarly, a service provider can provide actors representing services of varying size and complexity.

A useful feature of Calvin is that any 3rd party can provide an actor for a service by wrapping e.g. an existing REST-service, or providing an actor for a "dumb" device by writing a proxy actor for it. This lowers the barrier for Calvin uptake in the market, since there is no need to have a Calvin runtime on every constituent part in an application.

3.2. Connect

To create an application for the Calvin platform, we form a directed graph by connecting the ports of a number of actors. In the description of an actor, no assumption is made on how, or to what, it is connected; it simply processes data on its ports, unconcerned with anything beyond them. In the connect stage, information about how actors are connected is supplied in a straightforward fashion, best illustrated in a simple example. Here we use *CalvinScript*, a small, intuitive, declarative language to describe an application. A script, shown in Fig. 2 (left) consists of two parts: The first lists which actors to use, including any parameters they need, and the second how they are connected. The first three lines instantiate actors `src`, `sum`, and `snk`, and the last two lines connect the `tick` output port of `src` (which reacts to some kind of event occurrence, be it radioactive decay or visitors to an exhibit) to the input port `in` of the `sum` actor (which will produce a cumulative sum of the occurrences), and the output port `out` of `sum` is connected to the `snk` actor, which will display the output in the given format, here `plaintext`.

3.3. Deploy

While we now have a description of an application, we have neither specified anything regarding where the various actors should execute, nor how data should be transported between them. This is, in fact, handled dynamically during deployment of the application *and* continuously over its lifetime. Key to this functionality is the lightweight distributed runtime present on a number of nodes accessible to the user/operator deploying (and later maintaining) an application. These runtimes form a mesh network, wherein actors in a running application can migrate from one runtime to another, and how data is transported depends on the locations of actors.

Since actor migration in most cases is cheap and almost instantaneous, a simple way of deploying an application is to pass the application script to the closest accessible runtime. Once the runtime has instantiated and connected the actors locally, the distributed execution environment can move actors to any accessible runtime based on e.g. resource, locality, connectivity, or performance requirements.

The trivial deployment algorithm – specifying for each actor where it should execute – is impractical, as a range of factors influence the performance of an application. Local factors, such as the load on a runtime, either caused by more actors being deployed on the same runtime, or a change in the workload of existing actors, could trigger a migration of actors, as could external factors, such as network congestion or hardware changes.

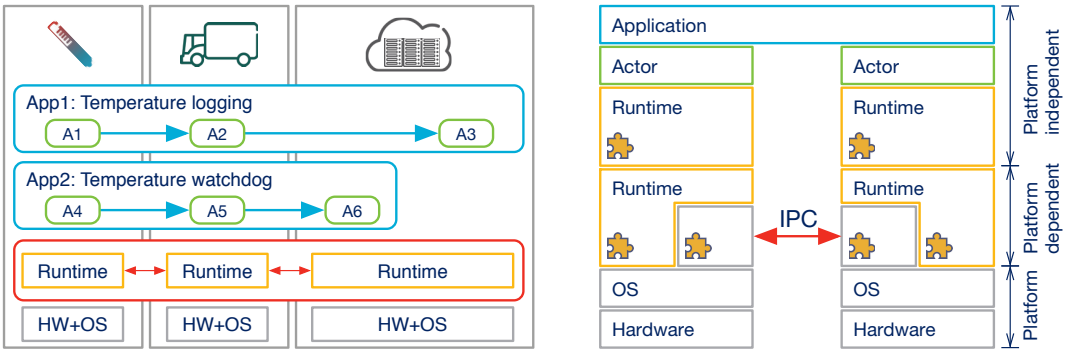

Fig. 1. **Left**: The distributed execution environment (red) formed by a mesh of runtimes (orange) is seen by the applications (blue) as a single platform. Applications, themselves transparently distributed across the network, consist of actors (green) representing resources such as devices, compute nodes, and storage nodes. **Right**: The Calvin software stack consists of a platform dependent layer handling data transport and presenting the platform specific features like sensors, actuators, and storage in a uniform manner to the platform independent layer that is responsible for runtime coordination, migration decisions, and the API exposed to actors. Many aspects of the runtime are extensible through a plug-in mechanism.

3.4. Manage

Once an application is running, it enters the managed phase. The distributed execution environment monitors the applications, handling e.g. migration of running actors, scaling, updates, and error recovery. Management also includes keeping track of resource usage, and allows for fine grained accounting, making it possible to, for example, charge customers for system usage with high resolution based on which actors are used, how frequently their actions fire, or how much data flows through the ports. Deployment and management are key areas for future research.

3.5. Application examples

In the example in Fig. 1 (left), the temperature of heat sensitive products, say frozen fish, is monitored during transport by two Calvin applications: A temperature logger and a temperature watchdog. In the first application, a thermometer, with limited computational and storage capacity, comprises an actor that continuously reports the current temperature to an actor on a nearby runtime with more resources, e.g. a laptop next to the driver. This actor sends a log of reported temperatures, either at set intervals or when sufficient bandwidth becomes available, to an actor on a company wide server that keeps track of the temperature of a whole fleet of vehicles.

In the second application, a different actor residing on the thermometer runtime, one which only sends an output when the temperature rises above (or falls below) a set value, is acting as a watchdog. This actor is connected to one on the laptop by the driver, which issues an alert, for example in the form of a beep or a flashing icon on the screen, and forwards the alarm to an actor on a server which can log the issue, and schedule maintenance of the vehicle.

4. A deep dive into Calvin

4.1. Runtime architecture

The high level architecture of Calvin is shown in Fig. 1 (right). Starting from the bottom, we have the hardware, the physical device or data center, and the operating system (OS) it exposes. Together, the hardware and OS constitute a platform for the runtime, and on top of this resides the platform *dependent* part of Calvin. Here we handle communication between runtimes, and through a plug-in mechanism we support any kind of transport layer (WiFi, BT, i2c) and protocol, including efficient implementation of data transport between actors sharing the same runtime. This is also where inter-runtime communication and coordination takes place. This layer also provides an abstraction of platform functionality, such as I/O and sensing capabilities, to the higher levels of the runtime in a uniform manner. Again, the actual algorithms controlling the behavior use a plug-in mechanism for flexibility and extensibility.

Above the platform dependent level of the runtime we have the platform *independent* runtime layer, providing the interface towards the actors. This is where e.g. the scheduler resides, a key component of the Calvin runtime. Since it has impact on things like latency and throughput, it too is extensible. On top of this, in the uppermost layers, we find the actors and the applications built from actors.

Not shown in this picture are the connections between actors as prescribed by the application. This is because the connections are *logical*, and *where* the actors reside is decided by the runtimes, and data transport is provided by the runtimes. It is important for the migration functionality that actors are unaware of their actual location. They only know that their location requirements, if any, are satisfied.

A Calvin runtime is inherently multi-tenant. Once an application is deployed and actors are migrated according to requirements and constraints, they may (and will) share runtimes with actors from other applications. The runtime can be configured to grant different access to resources depending on which application an actor is part of (or rather the access level of the user running the application.) There are a number of possible ways of implementing this, but at its simplest, a runtime may be divided into a "private" part with full access, and a "public" part with limited access.

There can also be restrictions on which actors are allowed on a runtime; simple ones, such as counters or timers, may be allowed, but more computationally expensive ones, e.g. image processing, may be limited or banned.

4.2. Flavors of Calvin

Calvin is meant to be portable across languages and platforms, and it is our goal to support as many different platforms as possible, ranging from small sensor devices to data centers. To enable this, we emphasize the isolation between actors. Only the format of data passed between ports is standardized, and how data is processed *inside* an actor is irrelevant. Similarly, how a runtime is implemented does not matter, as long as it adheres to the inter-runtime communication protocol and accepts the common set of control commands available to users and operators.

To provide a basis for device manufacturers and 3rd party developers to port the Calvin runtime to various platforms, we will release a reference implementation of Calvin, written in python, under an Open Source license during the first half of 2015. The python implementation will run on a wide range of platforms, but to cover the tiniest devices a porting effort will be needed. Our intention is to form a community around Calvin in order to grow the number of supported platforms and build a library of reusable actors.

4.3. Anatomy of an actor

In its basic form, an actor consists of ports, actions, and preconditions under which actions can fire. All communication with other actors is handled by unidirectional ports — in effect first in-first out queues. Apart from this there is also an initialization function and a priority list for the actions; should multiple actions be applicable at a given time, the priority list determines which one is actually applied. An action always has a *condition*, stating when the action can be applied in terms of the ports involved. A common situation is that there should be one or more tokens available on the in-ports, and space available for the result on the out-ports. Additionally, it is possible associate a *guard* with the action. A guard can be more complex than a condition; for efficiency, a condition is limited to superficial inspection of tokens – for example determining the number of tokens available – whereas a guard can inspect the data in the tokens, and prevent the action from firing depending on this data.

4.4. Further application examples

In this section we outline the control software of a simple vending machine in order to demonstrate how easy it is to construct and reuse components. In this context, a component is a hierarchical construct of actors and their connections that can be used as if it was an actor, thereby hiding complexity.

The controller, shown in Fig. 2 (left), consists of actors for handling a keypad, an item database, a dispenser, and money. The functionality of each of these components can, of course, be arbitrarily complex; it is non-trivial to build the electronics and software for determining which coins and bills to accept and which to reject, but this problem has already been solved for us, and we can write an actor, as a component, to handle this, and then use it in a component for handling money.

```
1. moneyhandler :  MoneyHandler(currency="euro")
2. dispenser :  Dispenser()
3. keypad :  KeyPad()
4. itemdb :  Database()

5. keypad.number > itemdb.choice
6. itemdb.value > moneyhandler.request
7. itemdb.choice > dispenser.choice
8. moneyhandler.ok > dispenser.ok
```

```
1. component MoneyHandler(currency) request -> ok {
2.   coindetector :  CoinDetector(currency)
3.   billdetector :  BillDetector(currency)
4.   cardreader :  CardReader()
5.   acceptor :  CashAcceptor()

6.   request > acceptor.request
7.   coindetector.coins > acceptor.coins
8.   billdetector.bills > acceptor.bills
9.   cardreader.card > acceptor.card
10.  acceptor.ok > ok
11. }
```

Fig. 2. **Left:** A vending machine controller described in CalvinScript. **Right:** A CalvinScript component for handling money.

Similarly, a card reader, either off-the-shelf with a "homemade" actor running on a separate runtime as interface, or one outfitted with a Calvin runtime and a choice of actors, can be included in the component.

Figure 2 (left) depicts a sample money handling component. The component consists of detectors for bills and coins, a card reader for handling credit card purchases, and an acceptor, a simple logical unit for determining if the amount supplied is enough to cover the requested purchase. There is one parameter, `currency`, which is passed on to the detector actors, determining which currency the component should be configured to handle.

There are two external ports exposed; an in-port, `request`, connected to the request port of the acceptor, and an out-port `ok`, connected to the same port of the acceptor.

The other actors of the vending machine script can be developed in the same way, following a top-down approach, and it is straightforward to extend the software with telemetry and remote logging to keep track of inventory, temperature, or tampering alarms off-site, by adding corresponding actors to the scripts.

As before, the application has no knowledge of, or need for knowing, where any of these actors are executing. If the keypad is replaced with a keyboard, or a touchscreen, and the moneyhandler is changed into an acceptor for credit card numbers, we have a "virtual" vending machine, or a web shop.

5. Benefits of Calvin

The Calvin runtime is very lightweight and can be ported to basically any microcontroller with communication capabilities (a prerequisite for IoT), or run hosted in the cloud, providing a truly heterogeneous application environment. Actors execute asynchronously and autonomously per definition. They can also encapsulate protocols, such as REST or SQL queries, as well as device specific I/O functionality. Since actors can freely migrate between runtimes with very little overhead, the actors of any application can (by decisions made autonomously by the runtimes) adapt dynamically to changes in load and/or network conditions. The same mechanism also allows runtimes to group actors for performance reasons, say low latency requirements.

While we have not yet submitted Calvin to rigorous failure and robustness testing, the mechanisms used for migration of actors is providing the core of failure recovery and we expect to achieve a high level of robustness.

In terms of productivity, we have found that given a set of actors it is very easy to create applications using CalvinScript. Furthermore, for a device manufacturer wanting to support Calvin, the main obstacle is porting the runtime to a device that cannot reuse any existing port, whereas writing actors representing device functionality once the runtime is ported is a straightforward process. Finally, for a service provider, or 3rd party developer, wrapping e.g. a database service in an actor is as easy (or hard) as using the service in the first place.

We therefore believe that Calvin meets all of the goals we set out to achieve with the platform, as stated in Section 2.

6. Alternative approaches to reducing fragmentation in the IoT and cloud area

Many players in the IoT and cloud area have noted and tried to address the problem of increasing fragmentation. We will now look at some of the proposals and discuss how they relate to Calvin. First up are three frameworks which are quite similar to Calvin, followed by a more traditional, device centric, approach.

6.1. IFTTT

A service framework that has gained attention recently is IFTTT[8] which lets users connect applications and/or services such that a certain task is performed when a particular event occurs. This relation is described with a simple statement called a *recipe* of the form `if <this> then <that>`, hence the name of the service. The `<this>` part is called a *trigger*, and the `<that>` part is called an *action*.

IFTTT provides a number of channels, each with its own set of triggers and actions. Currently there are over 160 channels available, including e.g. Facebook, Evernote, email, DropBox, Philips Hue, and Nest. Recipes can combine actions from one channel with triggers from another channel and vice versa. The result is a surprisingly simple way to create user defined services, such as the example on the IFTTT webpage, `if "Any new photo by you: ltibbets" then "Add file from URL to Linden Tibbets' Dropbox"`, that copies every photo tagged with a particular username to that person's DropBox storage.

While IFTTT is truly simple to use, it lacks the expressiveness of a programming (scripting) language traditionally used to implement this kind of service chaining. In comparison, CalvinScript achieves the same level of simplicity while retaining the full expressiveness of traditional programming languages with conditionals, loops, hierarchy, etc.

6.2. NoFlo

Another recent development is NoFlo[9], a Flow-Based Programming environment implemented in JavaScript. Similarly to Calvin, NoFlo applications are described as graphs where the nodes are NoFlo components, and the edges represent how data flows between components. The components are JavaScript objects defining the input and output ports and the operations that should be performed on incoming data. Applications are described by a JSON data structure that can be generated from the FBP domain specific language, which is quite similar to CalvinScript.

NoFlo programs can be deployed wherever JavaScript can be deployed and is thus highly portable, and pre-built components can be installed with ease using a package management system. NoFlo is free software hosted on GitHub.

Compared to Calvin, NoFlo is more mature, and clearly a powerful environment as demonstrated by the author of NoFlo who wrote a port of the Jekyll web publishing software using NoFlo.[10] A major difference between the two, is that Calvin relies on the graph property of the application to fully exploit the distributed nature of the application, allowing actors to migrate, manually or autonomously, from one device to another while the application is running, whereas NoFlo deployment is static in nature. Although we have yet to test it, Calvin and NoFlo are similar enough that they should be able to interoperate using a protocol conversion actor/NoFlo-component.

6.3. Orleans

Microsoft have recently announced their plans to open source Orleans[11], an Actor-based platform providing a programming model, a runtime, and access to "virtual" actors.[12] Here virtual is used in the sense of virtual memory, meaning that the system creates an illusion to the application (and programmer) of actors as always available, never failing, and of (almost) infinite processing power (by instantiating more copies). This provides a powerful environment for the programmer, by hiding much of what makes developing distributed systems hard.

In comparison, the focus of Orleans is distributed, high-performance cloud applications, whereas Calvin is focused on covering everything from tiny IoT devices to cloud services in a unified paradigm and platform.

6.4. ARM mbed

In a move to make development for embedded, primarily IoT-devices easier, ARM has launched the *mbed* initiative[13] together with a number of partners, including Ericsson. At the core is the mbed OS, an operating system for ARM cores that provides support for sensors and actuators, as well as a number of communication technologies, such as WiFi and Bluetooth LE. Optionally integrated with the mbed OS is a licensable mbed device server that provides an HTTP/CoAP stack, thus providing a means to access and manage devices through a secure service. Perhaps the most exciting part about mbed is the web based development environment where a community of users contribute to a growing repository of components, documentation, and applications.

While mbed offers a solution for the fragmentation issue at the lower levels of IoT programming, exposing functionality through the device server, in effect a device proxy, it does not address the overall complexity of creating and managing distributed applications, which is what we want to reduce by the introduction of Calvin. However, here we have an excellent use-case for the Calvin runtime. By running the Calvin platform on top of the mbed OS, we get an almost symbiotic relationship where both benefits. The Calvin ecosystem can grow by allowing others to write, and use, actors encapsulating components and functionality by reusing code from the component database, and the mbed community benefits from simplified application development and maintenance offered by Calvin.

7. Conclusion

We set out to create a framework that would merge IoT and cloud in a unified programming model. Our goals were to come up with a solution that would empower developers by hiding protocol and data transport details, to embrace system heterogeneity rather than avoid it, simplify management by allowing decisions to be made autonomously by the runtimes, and improving communication efficiency by avoiding a direct device-to-cloud client/server approach.

From a developer perspective, we think that using CalvinScript to write applications combining IoT and cloud services is a good choice. In particular, it is a way of consolidating the micro-service trend with single-purpose small devices waiting to be explored for a larger task.

For device manufacturers and service providers, Calvin provides a means to increase the attractiveness of their products by providing a new way to expose their offerings, in addition to traditional means, with a fairly low effort.

Finally, for operators and system maintainers the level of autonomy achievable in Calvin will hopefully lessen their burden of managing the expected 50 billion devices. The latter is also important from the point of communication networks where an, apparently exponentially, increasing amount of data have to be coped with. Moving computations towards the edge of the networks by increasing the programmability of IoT devices should reduce the data that traverses the core networks on its way to the, relatively few, data centers for processing.

Calvin is still in its early stages of development, and currently not all the desired functionality is fully implemented. By making Calvin available under an Open Source license and making the architecture extensible we hope that a community can form around Calvin and explore its possibilities to the fullest.

Our current work is focused on making sure the foundation is sound. We are working on the security and routing mechanisms required to make autonomous migration of the distributed application smooth, but there are many aspects left to explore. Important parts such as scheduling, deployment and management, are topics for future research, and while reference implementations are in place, it is likely much more efficient implementations exist.

More hands-on examples of future work include providing a domain specific language that can be used to generate target and language specific actors, adding more transport mechanisms, e.g. AllJoyn[14], providing ports of the runtime to e.g. mbed OS and Linux, and providing the runtime as a Platform-as-a-Service for use in data centers.

References

1. Ericsson, ; 2013;Now I get the Networked Society. URL: `http://ow.ly/yP10Y`.
2. Hewitt, C.. Actor model of computation: Scalable robust information systems 2010;URL: `http://arxiv.org/abs/1008.1459`.
3. Morrison, J.P.. *Flow-Based Programming, 2nd Edition: A New Approach to Application Development*. Paramount, CA: CreateSpace; 2010.
4. Carlsson, A., Eker, J., Olsson, T., von Platen, C.. Scalable parallelism using dataflow programming in multimedia and radio applications. *Ericsson Review* 2010;.
5. Eker, J., Janneck, J.. Dataflow programming in CAL — balancing expressiveness, analyzability, and implementability. In: *Signals, Systems and Computers (ASILOMAR), 2012 Conference Record of the Forty Sixth Asilomar Conference on*. 2012, p. 1120–1124.
6. Gray, R.S., Kotz, D., Cybenko, G., Rus, D.. Mobile agents: Motivations and state-of-the-art systems 2000;.
7. Lange, D.B., Oshima, M.. Seven good reasons for mobile agents. *Commun ACM* 1999;**42**(3):88–89.
8. IFTTT Inc., ; 2015;IFTTT – Put the internet to work for you. URL: `https://ifttt.com`.
9. Bergius, H.; 2015;NoFlo – flow-based programming for javascript. URL: `http://noflojs.org/`.
10. Bergius, H.; 2013;Why did I reimplement Jekyll using NoFlo. URL: `http://bergie.iki.fi/blog/noflo-jekyll/`.
11. Microsoft Research, ; 2015;Project "Orleans". URL: `http://research.microsoft.com/en-us/projects/orleans/`.
12. Bykov, S., Geller, A., Kliot, G., Larus, J., Pandya, R., Thelin, J.. Orleans: Cloud computing for everyone. In: *ACM Symposium on Cloud Computing (SOCC 2011)*. ACM; 2011, .
13. ARM Ltd., ; 2105;ARM mbed IoT Device Platform. URL: `https://mbed.org`.
14. AllJoyn framework. 2014;URL: `https://allseenalliance.org`.