ComFlux: Dynamic creation of pervasive applications from plug-and-play modules

Raluca Diaconu (presenter), Jean Bacon, Jie Deng, Jatinder Singh
University of Cambridge, UK
Email: {first.last}@cl.cam.ac.uk

Abstract

ComFlux is a middleware that enables an external management regime, whereby component interactions, communication methods and security constraints can be dynamically defined, extended and updated at runtime –without requiring redeployment or changes to the application logic of the components themselves, and without imposing constraints on system design. ComFlux offers means to 'instruct' components how, when and with whom they should interact, taking into account usability, modularity and access control.

The scope of this tutorial is to show the need for external command and control over IoT components, in order to enable their usage in ways and by means not envisaged by their creators. We demonstrate how to build applications dynamically using an open implementation of ComFlux. This two hour tutorial will be divided in two parts. The first part will argue the necessity of an infrastructure for external management and the model that enables it. The second part will demonstrate how to build applications dynamically using an open implementation of ComFlux to control IoT devices.

Index Terms

IoT Enabling Technologies, Broadly Applicable IoT Techniques and Methods, Security and Privacy for IoT.

Introduction

The Internet of Things (IoT) has been built upon a large range of sensors, actuators, mobile devices and wearables, integrated and driven by a range of software services. The real value of pervasive computing lies beyond its myriad technologies; it emerges from the interactions between components. Thus, the power of a component may be revealed when reused and adapted to fit new contexts. Nowadays, driven by human dynamics, heterogeneous devices and their mobility create increasingly complex environments in which dynamism and creativity can bring great value.

To date, functional requirements have typically been met by composing software and hardware components into "silos of things", managed in isolation. For instance, consumer solutions for "smart home" attempt to integrate a large range of IoT devices into a unified platform. Many development platforms focus on exploiting event-driven architectures and involve the network stack, cloud services, and a suite of APIs for web or smartphone. They require an account registration and sometimes a periodic fee.

We argue that components' functionalities could potentially be used for many diverse purposes, outside their initial scope. To achieve this, we designed and implemented ComFlux [1] a middleware to 'instruct' components how, when and with whom they should interact, taking into account usability, modularity and access control. In ComFlux, reconfiguration of capabilities and connection management happens dynamically at runtime, without requiring application-level intervention. This paves the way for new functionality, where existing components can be leveraged across application boundaries, in ways and for reasons not envisaged by their designers, and increases the longevity of components through updates and new uses.

External control and communication management raise important security concerns, in that only authorised components should be able to perform reconfiguration actions. ComFlux integrates mechanisms to respond to security concerns: the possibility to use encrypted channels, authentication and access control over data sources.

ComFlux has a modular design and provides mechanisms to extend and adapt the capabilities of an application at runtime including functional evolution of applications and software lifecycle updates. Different modules can be plugged in and out dynamically to allow flexible functionality.

The intended scope of our middleware is to enable policy enforcement in highly distributed systems, as we find in IoT technologies [2]. We therefore implemented ComFlux as a fully-fledged middleware incorporating a rich range of functionality. ComFlux was first deployed to support interactive applications for museums, in a dynamic, ad-hoc, people-centric environment. The focus was to support the development and deployment of a range of functionalities for interactive applications, as infrastructure for enabling interactive public spaces. We demonstrated this, and more, for “Science Museum Lates”[1].

The key benefit of this tutorial is to provide a practical demonstration of the potential for dynamic, external command and control capabilities, focusing on flexibility, reliability, and usability of IoT components. ComFlux offers not only an API to the developer but also opens up its reconfiguration layer externally.

a) Intended Audience: The intended audience is broad, relevant to those interested in the infrastructure necessary for supporting the IoT and pervasive computing in general. The practical demonstrators will be of particular relevance for those involved in practical IoT research (i.e., code) as well as those designing, developing and deploying IoT solutions. Understanding of security, authentication, access control, network layer communication or APIs is helpful but not necessary.

We will discuss the necessity of enabling secure external command and control for IoT devices and we will go hands-on in building components that allow external command and control. We’ll discuss the modular approach, the runtime reconfiguration, and will bring some connected IoT devices to use as a show case.

b) Tutorial: The two-hour long tutorial will be divided into two parts. The first will introduce the main concepts of external reconfiguration as well as our approach using ComFlux, see Section -0b. The second part will be a demonstration of the capabilities enabled with ComFlux. The participants will get acquainted with the use of the API for building components. Then we will show the power of our model by controlling the components using tools for dynamic reconfiguration and external communications management. The tutorial will conclude by proposing ways in which users can contribute or reuse the existing code base, see Section -B4 for more detail.

Tutorial Part I: External control and command with ComFlux Middleware (1 hour)

A. Model

In the first part we will introduce the ComFlux model, see [1] for more detail. It allows components to be instructed externally at runtime with whom they should interact, what are the means to communicate with new components, and when, under which circumstances they should interact. Components can reconfigure themselves, via their API, and certain components can control others. Our model is based on the following key external runtime controls:

a) External communication control: an authorised component can instruct another component to connect or disconnect to/from some specified component(s) and transmit or receive data.

b) External component (re)configuration: an authorised component can reconfigure other components’ functionality. This can involve changing communication protocols in a plug-and-play manner and adding/removing access privileges over other components; e.g., a command component can instruct a component to load a new communication interface for UDP, instead of TCP connections, if its battery runs low, or instruct a sensor component to relax its data access permissions in an emergency.

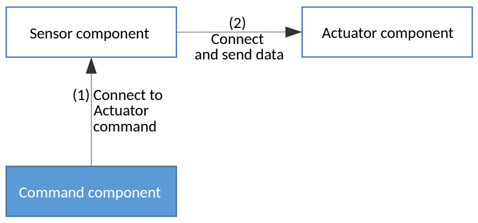

Fig. 1: Example of external reconfiguration

B. Implementation

We built ComFlux as a messaging middleware, in which external control and communication mediation occurs with respect to data transfer. It was designed to demonstrate the power of reconfiguration and first applied to “interactive experiences” in public spaces; though its functionality is more widely applicable. The current implementation, examples and documentation are open source and available online at www.comflux.net.

1) A messaging architecture: ComFlux is built as a framework for *typed* message exchange between *endpoints*. Applications define endpoints by specifying a *communication type* and a *message structure*. Both the sender and the receiver check the message against a *message structure* (or *schema*) that allows the specification of simple and complex types. Communication occurs between endpoints that are *mapped* (*connected*) and can have one of the following types:

- *source–sink*: unidirectional communication between a source(producer) and any number of sinks (consumers);

[1]Science Museum Lates – http://www.sciencemuseum.org.uk/visitmuseum/plan_your_visit/lates

supports event driven (publish-subscribe) model

- *request–response*: a client issues a request to a server which replies with one response; for mapping to be possible both request and response definitions must match
- *request–response+*: as above, but the server can issue one or more responses
- *streaming source–sink*: unidirectional communication source–sink of unstructured/streaming data.

Endpoints' definitions contain one of these types along with a message structure for sources and sinks, and two structures for request and response endpoints. Streaming endpoints are unstructured for end-to-end communication.

To keep the discovery and endpoint messages generic, components maintain a (reconfigurable) *manifest* that describes and identifies the component. The manifest enables schema negotiation on mapping requests, and message type matching. The manifest is the information shared externally to facilitate discovery and communication. It contains an updatable collection of information about itself, e.g., supported interfaces, addresses, access methods, public keys or user-declared information, and its endpoints, e.g., their type, message structure and permissions.

a) The core: is the backbone of the middleware and includes command and control functionality. It carries out data flow management and has the necessary mechanisms for external reconfiguration. It decouples the management plane from the application logic by maintaining endpoints and orchestrating functionality.

b) Communication and access control modules: are functional units that plug into the core dynamically. They are reusable building blocks which are loaded and unloaded at runtime. Modules are built as dynamic libraries that implement an interface (a set of predefined functions) recognised by the core. ComFlux offers communication modules that carry the communication over a specific medium, and access control modules for authentication and endpoint access.

c) The ComFlux API: is available to the application programmer as a set of functions that provide internal endpoint and middleware configuration functionality. To use the middleware, the application imports the ComFlux library which automatically spawns the core process. The API provides the interfaces for sending/receiving data and configuring the core.

2) Modules: The power of ComFlux is in providing a mechanism to extend and adapt the functionality of an application at runtime based on pluggable modules. The core specifies interfaces for communication and access control. Corresponding modules are built as dynamic libraries which are then loaded (linked) at runtime by means of internal (API) and external (endpoint) commands.

a) Communication modules: Components' external communication occurs via endpoints through the core, abstracting the underlying communication substrates. They may be either *Bridge modules* that expect a middleware core/component on the other side of the communication channel (currently handling TCP, UDP and SSL); or *Interfacing modules* that interface with other services (currently MQTT brokers and REST components).

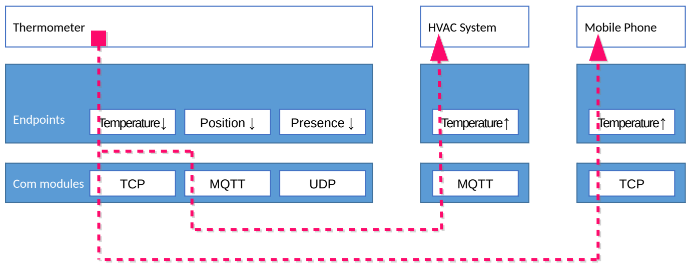

Fig. 2: Data flow involving a thermometer component with a temperature data source (↓) and two consumers (↑), a HVAC system and a mobile phone. The HVAC system supports only MQTT while the mobile phone application uses TCP sockets.

b) Access control modules: Component interaction is subject to authentication and access control. Access control modules implement an *authentication* mechanism and maintain *access control lists (ACL)* for each endpoint, describing the components that may interact with it. They are fine-grained –access control for each endpoint– and flexible –multiple identities and permissions that can be configured at runtime. Currently supported authentication mechanisms are username/password, SSL certificate and Kerberos.

3) Resource Discovery Components (RDCs): RDCs offer the means for dynamic selection and communication reconfiguration at runtime, thus enabling communication components that were unknown *a priori*. A component may *register* and periodically update its manifest with one or more RDCs. The RDC provides a *lookup* service, returning the addresses of components whose manifest and endpoint descriptions match the criteria in the query, e.g., a facilities management component may want to communicate with all light components in a building. If all the lights are registered with an RDC, the switch only needs to query that RDC for the lights of interest, allowing it to map (probe) the relevant devices in the building. Resource discovery is particularly important in interacting with mobile and dynamic components, which may 'come and go'. In ComFlux, the `map_lookup` command instructs a component to query all known RDCs and retrieve the lists of components and their manifests that match the query. The core then attempts a direct connection to communication modules supported by both. For example, a switch component may want to communicate with all light components in the environment. If all components are registered with an RDC, the switch may only need to query that RDC for the list of all light bulbs.

4) Supporting external reconfiguration: is achieved by exposing *control endpoints* that access the functionality in the control layer. We make use of these capabilities with *Swiss Knife*, our external command line tool. Control endpoints behave and function like regular endpoints. Their messages are directly handled by the core functions that resolve API calls. In this way both configuration and data functionality can be discoverable and negotiated at runtime. Loading/unloading modules, mediating connections, etc., can be executed by third parties at runtime.

Tutorial Part II: Practical demonstration (1 hour)

This part of the tutorial will contain an explanation of the main concepts through using a range of practical examples. This includes the use of connected devices such as smart lights, and possibly people's phones, to visually and tangibly display the capabilities of the infrastructure, and to illustrate the technical ease by which such complex functionalities can be realised. The practical tutorial is structured as follows:

a) Building a ComFlux component (15 min): This involves becoming acquainted with the ComFlux API by presenting some practical examples. The ComFlux implementation is open source and available at www.comflux.net. The repository contains the source code for the middleware, the API, the modules, the RDC, the Swiss Knife tools along with use cases and documentation. After that, we will demonstrate basic component building. These examples will make clear the use of endpoint instantiation, mapping, and message passing. Participants, if interested, may also try hands-on the middleware demonstration.

b) Using Resource Discovery (15 min): The next step is to interact with components that are not known a priori. Using the RDC we will show how components may register and publish information about themselves to other devices in the environment. Consequently, components may also select devices in the environment and connect to those that are relevant (e.g., select all the lights in the environment and map to them).

- Register a component with an existing RDC
- Lookup and map to components in the environment dynamically, based on various parameters
- Seamlessly select and map to new components in the environment.

c) Using external reconfiguration commands (20 min): The key contributions of ComFlux are the external command and control capabilities. The main part of this tutorial is dedicated to explaining the use of Swiss Knife, the command line reconfiguration tool included with the code. We will demonstrate the following commands:

- List everything from the RDC
- Map to an endpoint and communicate via messages
- Register a component with an RDC by means of an external command
- Load communication and access control modules.

d) Conclusion (10 min): We complete the tutorial by describing our future plans and proposing possible collaborations and contributions to the open source code.

References

[1] R. Diaconu, J. Bacon, J. Deng, and J. Singh, "Comflux: External composition and adaptation of pervasive applications," in *International Conference on Pervasive Computing and Communications, PerCom 2018 (Submitted).* IEEE, 2018.

[2] J. Singh, T. F. J. Pasquier, J. Bacon, J. E. Powles, R. Diaconu, and D. M. Eyers, "Big ideas paper: Policy-driven middleware for a legally-compliant internet of things," in *International Middleware Conference, Trento, Italy*, 2016.

Raluca Diaconu is a Research Associate at the Computer Laboratory, University of Cambridge where she is working on infrastructure for pervasive computing. She has completed a PhD in scalability for virtual and mixed reality worlds at Université Pierre et Marie Curie, Paris. Her current interests are interactive experiences using emerging technologies, IoT, Mixed Reality.