Poster Abstract: A SimPy-based Simulation Testbed for Smart-city IoT Applications

Neha Karanjkar, Poorna Chandra Tejasvi and Bharadwaj Amrutur
Robert Bosch Centre for Cyber-Physical Systems, Indian Institute of Science, Bangalore

ABSTRACT

A real-time testbed that emulates a large number of IoT end-points generating traffic to the middleware/application layers can be used for debugging and performance evaluation of the smart-city software platforms prior to deployment. We propose an architecture for such a simulation testbed based on Python's SimPy library. The simulated IoT end-points communicate with the middleware in real-time and can also interact directly with each other and with a common shared environment. This makes the testbed particularly suited for modeling system-wide scenarios such as synchronized faults and power outages.

CCS CONCEPTS

• **Computing methodologies → Simulation tools**.

KEYWORDS

Internet of Things, Simulation Tools, Smart Cities

ACM Reference Format:

Neha Karanjkar, Poorna Chandra Tejasvi and Bharadwaj Amrutur. 2019. Poster Abstract: A SimPy-based Simulation Testbed for Smart-city IoT Applications. In *International Conference on Internet-of-Things Design and Implementation (IoTDI '19), April 15–18, 2019, Montreal, QC, Canada.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3302505.3312591

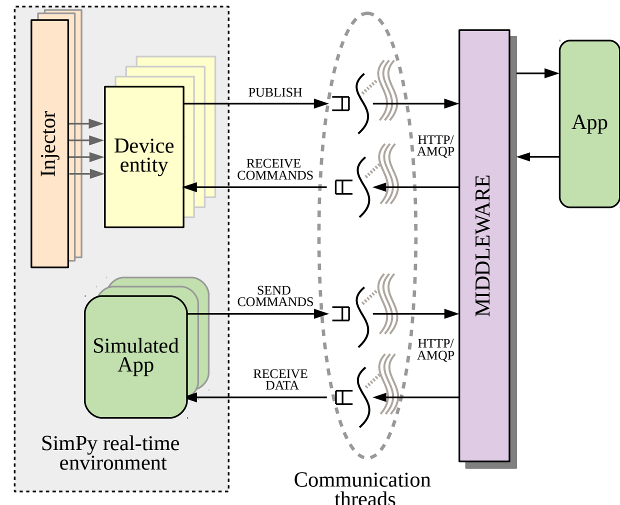

Figure 1: Architecture of the simulation testbed

1 INTRODUCTION

The ease of collection and sharing of data is a key requirement for future smart-city applications. A common middleware platform with standardized, open APIs that decouples the IoT end-points (sensors/devices) from applications is necessary for promoting interoperability in smart city ecosystems. Thus a typical smart-city IoT system can be thought of as consisting of four layers: a device/end-point layer, a network layer, a middleware layer and an application layer. The design and implementation of each of the layer requires a suitable test infrastructure. For development of the middleware and application layers, a testbed needs to emulate thousands of IoT end-points generating data and responding to control messages sent via the middleware in real-time. Such a testbed must be able to exercise all functional aspects of the middleware and should allow for flooding the middleware with simultaneous requests to evaluate its performance under realistic loading conditions. While real hardware end-points can be deployed within a testbed, their numbers are limited by cost. Instead, a hybrid testbed consisting of a large number of simulated entities and a smaller number of real hardware nodes is a practical alternative. Several commercial tools such as IoTIFY[3] and Amazon Web Services' IoT device simulator offer the means to simulate multiple end-points generating messages to a cloud application. While these tools provide an easy-to-use web interface, they are not as expressive as a discrete-event simulation framework. It is not possible, for instance, to model direct interactions between the end-points or to have heterogeneous systems with multiple types of devices. The use of a discrete-event simulation engine for modeling IoT systems has been described in works such as [1] and [2]. However, their focus is not on real-time testing of the middleware/application platforms.

In this paper we propose an architecture for a simulation testbed based on SimPy[5], a popular and expressive Python library for discrete-event simulations. The testbed allows **real-time, middleware-in-the-loop simulations** for functional and performance testing of the smart-city software platforms. The unique feature of this testbed is a systematic means of simulating distributed, large-scale phenomena (such as synchronized faults and power outages) and the ability to model direct interactions between end-points and a common shared environment. This is essential for functional testing in applications like smart grids and traffic management where the actuation of an IoT end-point can affect the state of the shared environment which can in-turn affect the sensor readings of the nearby end-points.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
IoTDI '19, April 15–18, 2019, Montreal, QC, Canada
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6283-2/19/04.
https://doi.org/10.1145/3302505.3312591

2 ARCHITECTURE

The testbed consists of multiple device entities which publish data and respond to control messages sent by applications over the middleware. Additionally, application entities (that consume data published by devices) can also be simulated within the same environment, enabling the testing of the middleware for multiple producer/consumer configurations. The testbed architecture is illustrated in Figure 1. The software for the testbed consists of the following main components:

(1) Device models: the software includes an extensible library of behavioral models for the device/application entities. These models are described as Python classes with attributes to store state information and a method to describe its main behavior. This method is invoked as a SimPy process at the start of simulation. Constructs such as time-delays and concurrent actions can be described within the behavior using constructs provided by SimPy. The user can populate the testbed with instances of these classes directly or can create derived classes to model end-points at the desired level of detail.

(2) Communication interfaces: Each device/app entity owns interfaces for communicating with the middleware asynchronously. The interface consists of a queue and a helper thread. For example, to publish a message, the device entity pushes the message into a queue. The message is then picked up by the helper thread and sent to the middleware using routines provided by the Middleware's SDK. Similarly, the helper thread responsible for receiving messages gets the messages from the middleware either by polling or by means of a call-back method and relays them to the entity via a queue. Such an architecture decouples the communication/network latencies from the simulation execution.

(3) Injector modules: An Injector module is the most important and unique feature of this architecture. Injectors are useful in modeling global phenomena and evolving environmental conditions (that can affect multiple end-points simultaneously). An Injector module is a Python object that resides within the same simulation environment and stores references to each of the device entities. It also has a `behavior()` method that is invoked as a SimPy process when simulation begins. The state-variables inside the device instances can be directly viewed/set from the Injector module. Thus, the Injector can be used to set sensor values, inject faults and activity or change the state of the devices at specific time instants. (No race conditions can arise in doing so because of the sequential execution at each time-step within the simulation engine.) As a simple example, Figure 2 shows the pseudo-code for an Injector module that injects faults into groups of devices at certain time instants, and another module that sets the devices' sensor values according to a linear pattern in space and time. Without the existence of such an Injector feature, the description of global phenomena would have to be broken down and incorporated into the local, device-level descriptions, making the testbed inflexible and the testing process tedious and prone to errors.

```
Behavior                                        FAULT INJECTOR
{
  wait until current_time == 100
  inject fault "power_failure" in devices 1 to 100
  wait for time_interval(randint(100,200))
  inject fault "sensor_failure" in devices 101 to 200
  ...
}
```

```
Behavior                                        STATE INJECTOR
{
  while(True)
  { wait for time_interval(1)
    t = current_time
    for i=1 to 100
      inject sensor value (100+t+10*i) in device i
  }
}
```

Figure 2: Examples of Injector modules

3 PRELIMINARY RESULTS

We have used the simulation testbed for functional and performance testing of the IUDX Middleware[4]. For a reasonably detailed model of smart-streetlight entities operating at 1 Hz, we observe that the testbed scales well up to 10^3 devices within a single process. While the number of devices that can be modeled within a single SimPy environment is much larger, the number of simulated entities is currently limited by the communication channels that can be opened within the process simultaneously. This number can be scaled further by having multiple SimPy simulations running as separate processes (on separate machines) and synchronizing loosely with respect to real-time via a service such as NTP. In this case, direct interactions between entities are restricted to those existing within the same simulation environment. Efforts in this direction are ongoing.

ACKNOWLEDGMENTS

The authors would like to thank Arun Babu, Vasanth Rajaraman and Abhay Sharma for discussions and feedback.

REFERENCES

[1] Giacomo Brambilla, Marco Picone, Simone Cirani, Michele Amoretti, and Francesco Zanichelli. 2014. A simulation platform for large-scale internet of things scenarios in urban environments. In *Proceedings of the First International Conference on IoT in Urban Space*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 50–55.
[2] Gabriele D'Angelo, Stefano Ferretti, and Vittorio Ghini. 2016. Simulation of the Internet of Things. In *High Performance Computing & Simulation (HPCS), 2016 International Conference on*. IEEE, 1–8.
[3] IoTIFY. 2018. IoTIFY: IoT device simulator. https://iotify.io/. [Online].
[4] IISc RBCCPS. 2018. Indian Urban Data Exchange (IUDX) for Smart Cities. http://www.rbccps.org/iudx/. [Online].
[5] SimPy Team. 2018. SimPy: Discrete-Event Simulation for Python (Version 3.0.10). https://simpy.readthedocs.io/en/latest/. [Online].