Demo Abstract: ImmunoPlane - Middleware for Providing Adaptivity to Distributed Internet-of-Things Applications

Kumseok Jung*, Gargi Mitra†, Sathish Gopalakrishnan‡, and Karthik Pattabiraman§
Electrical and Computer Engineering
The University of British Columbia, Vancouver, Canada
Email: *kumseok@ece.ubc.ca, † gargi@ece.ubc.ca, ‡ sathish@ece.ubc.ca, § karthikp@ece.ubc.ca

***Abstract*—Making Internet-of-Things (IoT) applications adaptive under unexpected failures and resource fluctuations can be challenging. In this demo, we present *ImmunoPlane*, a middleware system that brings adaptivity to IoT applications. Using a distributed stream-processing application, we demonstrate how *ImmunoPlane* transparently configures and deploys the application such that it can adapt to network congestions and random device failures.**

***Index Terms*—Internet of Things, Adaptive systems, Middleware**

I. Motivation

Internet-of-Things (IoT) applications must provide a certain guaranteed level of performance and dependability, while anticipating failure events such as power outages, network outages, and abrupt changes in resource usage, all of which can randomly occur across the Internet. Therefore, *adaptivity* is a cross-cutting concern in IoT application deployment and must be readily provided to various applications (e.g., similar to how logging and caching systems are provided in a plug-and-play fashion in aspect-oriented programming). However, enabling adaptivity universally across IoT applications is difficult, as different IoT applications might prioritize different performance and dependability metrics. Further, the same IoT application can be deployed in vastly different infrastructures, each with different resource constraints and failure events.

Our goal is to design an adaptivity solution that is both *application-aware* and *infrastructure-agnostic*. The solution must enable applications with differing requirements to become adaptive, regardless of the infrastructure in which they are deployed. The key idea is to define a set of adaptation primitives, which can be selectively applied based on the user's requirements and the given infrastructure. The user would provide a high-level description of the application's performance and dependability requirements in a declarative manner, requiring as little effort as writing a configuration file.

II. Approach

We present a middleware system called *ImmunoPlane* [1], which provides adaptivity to IoT applications in a declarative manner, without requiring the user to make changes to the application code or writing infrastructure-specific customizations. At a high-level, the idea is to obtain application-specific information from the user, collect infrastructure-specific information through continuous monitoring, and then apply a combination of well-known adaptation techniques based on the deployment-specific optimization goals. To realize this idea, *ImmunoPlane* implements two key components as follows.

1. Domain-specific language (DSL). The *ImmunoPlane* DSL provides two main abstractions: the dataflow `graph` and a set of `policy` directives. The purpose of the `graph` is to capture the communication topology of the application components, which is necessary for making placement decisions such as component co-location and replication. The `policy` directives are used to express fine-grained application requirements, such as whether a component must be highly available, or whether a communication link must deliver a minimum throughput. The DSL allows a user to provide information about the application that are necessary for adaptation, without making any modifications to the application's own code.

2. Dataflow Reconfiguration and Placement Search. Based on the `graph` and `policy` expressed in the DSL, the *ImmunoPlane* Scheduler computes a placement of application components that satisfies the user's requirements, and employs a set of adaptation primitives to handle failure events. The Scheduler's algorithm can be described in two parts. First, the Scheduler reconfigures the `graph`, adding or replicating components. For example, if a component is required to be highly available, the Scheduler will add a "watcher" component to monitor the liveness of the target component, and add a standby component as a fallback. Second, the Scheduler applies a fine-grained, application-specific heuristic while searching for a requirement-satisfying placement.

III. Implementation

We implemented *ImmunoPlane* on top of OneOS [2], our previously-developed IoT platform written in C#[1]. In particular, we have added the *ImmunoPlane Scheduler*, which is responsible for interpreting the user requirements written in the *ImmunoPlane* DSL, and producing an adaptive deployment plan based on the given requirements. We have also modified the OneOS workers, such that they can be controlled by the *ImmunoPlane* Scheduler. Finally, we implemented a web-based

[1] https://github.com/DependableSystemsLab/OneOS

979-8-3503-7025-6/24/$31.00 ©2024 IEEE
DOI 10.1109/IoTDI61053.2024.00029

graphical user interface (GUI), which we call the *WebTerminal*, for monitoring and controlling the user applications (Fig. 1).

IV. Demo Script

Synopsis. We will use the WebTerminal for presenting this demo. In this demo, we deploy a long-running stream-processing application distributed across a cluster of *Immuno-Plane* Workers, and show how *ImmunoPlane* enables the application to adapt to random device failures and network congestions. During the course of this demo, we highlight three key aspects of *ImmunoPlane* as follows:

1) *Component placement* – Our cluster consists of devices with different compute resources. We show how *Im-munoPlane* determines where to place each application component based on the available resources.
2) *Adaptation under network congestion* – We introduce network congestion between a pair of *ImmunoPlane* Workers, and show how *ImmunoPlane* adjusts the mes-sage distribution among the Workers in response to the congestion.
3) *Adaptation under device failure* – We randomly crash one of the devices in the cluster, and show how *Immuno-Plane* recovers the application by migrating the failed components to another healthy Worker.

Demo Setup. We set up six remote machines, three of which are cloud servers with Xeon E5 processors and 64GB RAM, and three of which are Raspberry Pi 3s with ARM 7 processors and 1GB RAM. One of the cloud servers runs the *ImmunoPlane* Scheduler and the WebTerminal, while each of the other five devices runs a *ImmunoPlane* Worker. The WebTerminal server is bound to a public IP address and is accessible through a standard web browser.

Demo Script. On our local machine, we open a browser and access the WebTerminal by navigating to the host address of the WebTerminal server. We then log into the cluster through the WebTerminal. A screenshot of the WebTerminal user interface is shown in Fig. 1. We open the *Resource Monitor* page to view the list of Workers, their real-time resource usage, and the application components each Worker is hosting. We observe that the Workers are idle, as we have not run any user applications yet.

We then open the *Code Editor* page where we can view the source code of the application that we will be running, as well as the *dataflow graph* and *deployment policy* written in the *ImmunoPlane* DSL. The application we use in this demo is the DSP-TrafficMonitoring application [3], which has both a parallel component ("MapMatcher") as well as a centralized component ("SpeedCalculator"). We generate the input data continuously, instead of using a fixed-size input file. Using the *ImmunoPlane* DSL, we indicate that the "MapMatcher" component must deliver a minimum processing rate of 200 messages per second (*mps*). We also indicate that "SpeedCalculator" must be highly available. We then deploy the application using the WebTerminal.

We navigate back to the *Resource Monitor* page, where we are able to observe the decision that *ImmunoPlane* made in terms of replicating and placing each of the components. We observe that, *ImmunoPlane* has replicated the "MapMatcher" component across all the Workers. In addition, *ImmunoPlane* has created a stand-by "SpeedCalculator" component, which is configured to launch upon failures of the primary "Speed-Calculator" component. We observe that both the primary and stand-by "SpeedCalculator" components are placed on the cloud servers. In addition to the component placement, we observe the real-time message processing rate of each Worker.

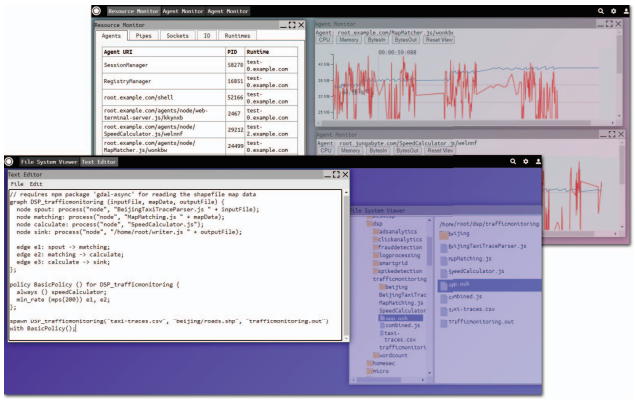

Fig. 1: Screenshot of the *ImmunoPlane* Web Terminal

Using a remote script, we emulate network congestion in one of the Raspberry Pi's by using the `tc` command. The command artificially limits the bandwidth of the ethernet interface of the target Worker. In the *Resource Monitor* page, we observe that the processing rate of the target Worker drops. After a small delay, we observe that the processing rate of other Workers increase proportionally.

Finally, we emulate a random device failure by terminating one of the *ImmunoPlane* Workers through remote access. We access the cloud machine hosting the primary "SpeedCalcu-lator" component, and shut down the *ImmunoPlane* Worker running on it. In the *Resource Monitor* page, we observe that the processing rates of all the Workers drop to zero momentarily. After about 1.5 seconds, we observe the status of the failed Worker reflected in the WebTerminal, as the Scheduler detects the Worker failure. *ImmunoPlane* initiates the recovery sequence by activating the stand-by "SpeedCalcu-lator" component and re-establishes the failed connections. We observe the processing rates of Workers return to 200 mps.

References

[1] K. Jung, G. Mitra, S. Gopalakrishnan, and K. Pattabiraman, "Immuno-plane: Middleware for providing adaptivity to distributed internet-of-things applications," in *Proceedings of the 9th ACM/IEEE Conference on Internet of Things Design and Implementation*, 2024.

[2] K. Jung, J. Gascon-Samson, and K. Pattabiraman, "Oneos: Middleware for running edge computing applications as distributed posix pipelines," in *2021 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2021, pp. 242–256.

[3] M. V. Bordin, D. Griebler, G. Mencagli, C. F. Geyer, and L. G. L. Fernandes, "Dspbench: A suite of benchmark applications for distributed data stream processing systems," *IEEE Access*, vol. 8, pp. 222 900–222 917, 2020.