2024 IEEE/ACM Ninth International Conference on Internet-of-Things Design and Implementation (IoTDI) | 979-8-3503-7025-6/24/$31.00 ©2024 IEEE | DOI: 10.1109/IoTDI61053.2024.00006

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*—Distributed Internet-of-Things (IoT) applications operate in a dynamic environment, and therefore need to adapt in response to unexpected failures and changes in the operating conditions. Making IoT applications adaptive is challenging due to two reasons. First, an IoT application comprises multiple service components, each with a different performance and dependability requirement. Second, an application can be deployed in vastly different runtime infrastructures, each varying in the availability of resources, and sources of faults. Hence, an adaptivity solution must be both application-aware and infrastructure-agnostic.**

In this paper, we present a middleware system called *ImmunoPlane* that transparently provides adaptivity to IoT applications. *ImmunoPlane* provides a domain-specific language for users to declaratively state application-specific requirements, and it produces an adaptive deployment plan based on the given infrastructure and the user-provided application requirements. We show that *ImmunoPlane* can satisfy application requirements such as availability, throughput, and latency, under both failures and network congestion, in three different infrastructures.

***Index Terms*—Internet of Things, Adaptive systems, Middleware, Failure Tolerance, Domain-specific Language**

I. Introduction

Distributed Internet-of-Things (IoT) applications operate in a *dynamic environment* where unpredictable events such as device failures, network outages, and abrupt spikes in resource consumption can degrade their performance and correctness [1]. Making IoT applications *adaptive* to such dynamic operating conditions while satisfying user requirements, is critical to ensuring their utility. In this work, we focus on two important adaptivity goals for distributed IoT applications – *performance management* (coping with fluctuations in resources), and *fault tolerance* (coping with failures).

Ensuring adaptivity in IoT applications is challenging due to two reasons. First, IoT applications can have different performance and dependability requirements. Second, an IoT application can be deployed in different runtime infrastructures [2]. For example, consider a "*Home Security*" application used for surveillance of residences or storefronts (§II). This is a video analytics application that includes a *real-time processing pipeline* for live object detection and end-user notification, and a *batch processing pipeline* for video encoding and storage. This application has different *local requirements* (i.e., performance and dependability requirements) for each processing pipeline. For the live object detection task, it must deliver low end-to-end latency and high availability. For the video storage task, it must preserve the integrity of the stored footage.

To satisfy the above requirements, we must determine the appropriate *deployment plan*, which involves the *placement of service components*, and the *feedback control loop* mechanisms (i.e., monitoring and invoking adaptive actions) for coping with failures and resource fluctuations. For each given infrastructure, we need to adjust the deployment plan based on the available resources and the location of faults that can occur in that infrastructure. This is the focus of our work.

Existing systems that potentially provide adaptivity in the IoT domain have two limitations. They either assume a global requirement across all service components (thus lacking in application-awareness), or require the user to manually set up the monitoring and adaptation mechanisms (thus lacking in infrastructure-independence). For example, stream processing frameworks (SPFs) such as Apache Flink [3] and Spark [4] provide failure recovery and dynamic scaling. Both frameworks, however, target a specific class of applications, and do not support fine-tuning for local requirements, which is needed for IoT applications such as our Home Security example. On the other hand, coordination systems such as CHARIOT [5] and PCL (Program Control Language) [6] allow users to customize the deployment plan through *imperative APIs*, which explicitly tells the system to monitor a specific resource in a given infrastructure, and update/replicate a specific component. However, due to the imperative design, the user must write a different deployment plan for each new infrastructure.

Our goal is to design an adaptivity solution that works across different IoT infrastructures with minimal user effort, while also providing a user the ability to customize the solution for each target application. To this end, we present a middleware system called *ImmunoPlane* that provides adaptivity to IoT applications in an *infrastructure-agnostic* manner.

ImmunoPlane has two innovations to achieve its goal. First, *ImmunoPlane* provides a domain-specific language (DSL) for a user to declare *component-specific requirements*, allowing the middleware to be application-aware. Second, *ImmunoPlane* implements an algorithm that produces a concrete *adaptive deployment plan* based on these requirements, while taking into account the resource characteristics of the target infrastructure. *To the best of our knowledge, ImmunoPlane is the first IoT middleware to provide adaptivity while achieving both application-awareness and infrastructure-independence, and requiring minimal user effort.*

979-8-3503-7025-6/24/$31.00 ©2024 IEEE
DOI 10.1109/IoTDI61053.2024.00006

Contributions. Our work makes the following contributions.

- We propose a DSL that enables developers to express application-level performance and dependability requirements in a declarative fashion (§IV-B). This DSL allows users to declare component-specific local requirements without specifying the adaptation conditions and actions.
- We develop an algorithm to compute a concrete *adaptive deployment plan*, consisting of the placement of service components and a set of specialized *adaptation components* (§IV-C). Based on user-provided application requirements (expressed in the DSL), our algorithm constructs an application-specific search heuristic, which is then used to efficiently guide the search for a placement that satisfies the custom user requirements.
- We implement *ImmunoPlane*, a distributed middleware that delivers the deployment plan produced by the above procedure. *ImmunoPlane* is built on our prior work, OneOS [7], and has been publicly released [1].
- We evaluated *ImmunoPlane* across 14 different benchmark applications, taken from four papers (Yahoo Streaming Benchmark [8], RIoTBench [9], DSPBench [10], and ThingsJS [11]). We find that *ImmunoPlane* delivers three-nines (99.9%) availability (conservatively assuming a device failure every 28 minutes), maintains similar 99^{th} percentile latency as Flink at $15,000$ events/sec under congestion in a single route, and recovers from device failures within 2 seconds, on par with existing SPFs (§V). *ImmunoPlane* transparently provides adaptivity in three different infrastructures, with minimal user effort.

II. Motivating Example

We highlight the design complexities encountered by a user while developing an *adaptive* IoT application with an example of a *Home Security* application from prior work [7]. We use this application as a running example in this paper to describe how our proposed solution, *ImmunoPlane*, helps a user address these design challenges. The application setup that we describe is a simplified representation of popular video analytics applications such as those built on Microsoft Rocket [12]. The application has three high-level objectives: ① analyze a live video stream from a camera to detect events of interest – e.g., "a person moving" – and notify the end-user, ② store the video stream on persistent storage, and ③ provide a live feed of the video stream for viewing by the end-user. We assume that the application is written in JavaScript (Node.js).

The Home Security application comprises a set of communicating services that can be distributed over the *edge* and the *cloud*. Each of these services, or *application components*, consumes data from an upstream component, processes the data and then passes it on to a downstream component. The application can be expressed as a directed acyclic graph (DAG), as shown in Fig. 1. The nodes in the DAG represent the components, while the directed edges represent the flow of data from one component to another.

[1]Available at: https://github.com/DependableSystemsLab/OneOS/tree/ImmunoPlane

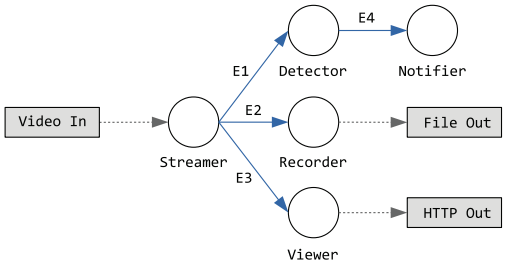


Fig. 1: Dataflow diagram representing the "Home Security" application example. Each node represents an individual service component, and edges represent the data flow from one service to another.

We briefly describe the individual components below:

- `Streamer` reads the input from a physical video device (e.g., a camera) and publishes the video frames in real time to three other components via three edges.
- `Detector` reads the frames from `Streamer`, performs image recognition tasks, and publishes messages about events of interest (e.g., "a person moving").
- `Notifier` is a messaging service that receives event data from the `Detector`, and sends a message to the end-user if any changes are detected.
- `Recorder` reads the frames from `Streamer`, and stores the frames as one-minute long video files on a disk.
- `Viewer` is a web service that reads the video frames from `Streamer`, and provides a live feed through the web.

Component-specific Local Requirements. In the *Home Security* application, the processing pipelines along different edges have different *performance and dependability requirements*, depending on the purpose and the processing pattern of each component. `Streamer` continuously sends video frames in real time to `Detector` (along edge E1), as long as the video device is on. `Detector` and `Notifier` (along edges E1 and E4) are continuous stream-processing services, and are expected to be *highly available* and have *low latency* as they serve a critical functionality in real-time (i.e., ① "secure the premises"). On the other hand, the `Recorder` is a batch-processing service that saves a video file after every 1-minute worth of video frames are buffered. For the `Recorder`, *integrity* is more important than availability, i.e., it is acceptable for the `Recorder` to not operate continuously, but it is not acceptable to lose the video frames it processes (as they may be security critical). Finally, the `Viewer` is a web server, and has neither availability nor integrity requirements.

As shown in our example, an IoT application contains multiple processing pipelines and each has its own *local requirement*. This is in contrast to stream-processing or parallel computing applications, which have end-to-end uniform *global requirements* over the entire processing pipeline. An IoT solution must, therefore, take into account the various component-specific performance and dependability requirements when it provides adaptivity to a given application.

Support for Diverse Runtime Infrastructures. Typically,

an application like the Home Security application would be deployed in different deployment environments such as residences, storefronts, or offices. For each environment, we need a different *adaptive deployment plan*, because the runtime infrastructure (i.e., the availability and connectivity of compute resources) is much more diverse in IoT environments than in cloud or enterprise environments. By *adaptive deployment plan*, we mean the appropriate *placement* of components and the dynamic application of *adaptive actions*. For each given infrastructure, the application components must be distributed strategically to deliver the performance required, and different adaptive actions might be required depending on the sources of fault present. For example, component migration might be used to deal with battery draining to low levels, and component parallelization might be used to deal with network congestion.

For example, consider two different infrastructures (Fig. 2). Infrastructure A has three edge devices and a cloud server. Edge Device 1 and 2 are configured to host three components, `Streamer`, `Detector`, and `Recorder`. Edge Device 3 is configured to provide redundancy for `Detector` and `Recorder`, so that in case Edge Device 2 fails, it transparently takes over to ensure that `Detector` is available. `Streamer` sends the video frames to both replicas of `Recorder` running on Edge Devices 2 and 3, to ensure the integrity of the recordings through redundant copies. `Notifier` and `Viewer` are hosted on Cloud Device 1 without any redundancies, relying instead on the reliability guarantees provided by the cloud.

However, this particular deployment plan does not transfer directly to Infrastructure B, which has a single edge device and three cloud devices. In Infrastructure B, since there is no redundant edge device, Edge Device 1 hosts both `Streamer` and `Detector` to avoid streaming video to the cloud. Therefore, to satisfy the availability and integrity requirements of the `Notifier` and `Recorder` components, the cloud devices are inevitably used. To meet latency requirements while sending data over the edge-cloud link, a cloud-based message broker is used to avoid creating multiple edge-cloud device connections.

The differences between runtime infrastructures require significant changes to the deployment plan, which can be quite cumbersome. Therefore, an adaptivity solution for IoT applications must allow users to be infrastructure-agnostic, and not ask them to develop a new plan for each new infrastructure.

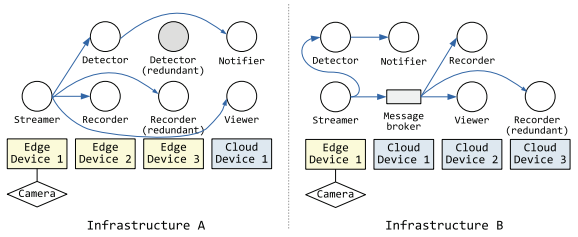


Fig. 2: Different deployment plans employed in two different infrastructures A and B for the same "Home Security" application. Infrastructure A makes use of redundant edge devices, whereas Infrastructure B relies on the higher reliability of cloud devices.

III. Challenges

What makes it difficult to build an adaptivity solution for IoT applications that is both infrastructure-agnostic and application-aware? There are two broad challenges: ① the diversity of application architectures, and ② the heterogeneity of the IoT environment. We detail them below.

A. IoT Application Diversity

IoT applications are very weakly classified, likely because the scope of the IoT is quite broad and general. This is in contrast to well-defined paradigms such as stream processing" [13] or "serverless computing" [14]. There is an extremely diverse set of IoT applications, ranging from smart farms, content-delivery systems, to federated learning systems [15]. The range of applications precludes well-defined systems architecture, and hence we can make few assumptions.

This diversity in application architecture makes it difficult to develop a general adaptation solution, because (1) an adaptation technique that works for one application may not work well for another, and (2) different applications have different performance and reliability requirements. We outline below the specific challenges we face in this regard.

- **(C1) User Requirements.** Different IoT applications have different requirements. Real-time video analytics applications might prioritize availability at the cost of detection accuracy [16], whereas ETL (extract-transform-load) applications might prioritize throughput and integrity over latency and availability. Thus, not only do applications have different requirements, the metrics that applications prioritize can be different from each another.
- **(C2) Dataflow Topologies.** There is no standard way to organize the computation pipeline of an IoT application. Our "Home Security" example contains three concurrent and independent pipelines, map-reduce applications contain parallel and synchronized pipelines, and event-processing applications contain concurrent and asynchronous pipelines. Without knowing what parts can be parallelized or what parts need to be synchronized, it is difficult to adapt for performance.
- **(C3) Data Types** IoT applications handle various types of data, ranging from tuples, JSON dictionaries, video segments, to raw byte arrays. Knowing *a priori* what type of data an application processes is necessary for applying adaptation techniques over the data stream. For example, we need to know a priori how to index a message to dynamically partition the stream for parallel processing.
- **(C4) Statefulness.** IoT applications can be stateful or stateless. In the case of stateful applications, the state can be centralized or distributed. The adaptation technique we can employ depends on the above. For example, we can easily scale a stateless application component horizontally by running multiple instances. However, we cannot do the same for a stateful application component, unless we synchronize the state across the replica components.

B. IoT Environment Heterogeneity

The IoT ecosystem encompasses various types of devices, each serving a different purpose, connected at different parts of the global network. Thus, the availability of physical resources such as processor, memory, and disk, as well as the network connectivity greatly varies across the devices [17]. As multiple resource and connectivity constraints need to be taken into account for different components, it becomes difficult to determine the optimal placement of computations (**C5**). For example, adding redundant components in a homogeneous network is straightforward, as they would run similarly across all the devices. In a heterogeneous network, however, we need to determine whether the redundant components would deliver the same level of performance, given that the host device has different hardware and connectivity. We have to tackle the additional complexity that arises from heterogeneity.

IV. Approach

We design and implement *ImmunoPlane*, a distributed middleware system that provides adaptivity to various classes of IoT applications, without requiring the user to make infrastructure-specific choices such as component configuration and placement. *ImmunoPlane* is a middleware between the application and the runtime infrastructure, and effectively provides an abstract *adaptivity layer* in which the configuration and placement decisions are made transparently.

A. System Overview

We first provide an overview of *ImmunoPlane* by describing the workflow from the perspective of a user. Let us assume that the user wants to deploy an IoT application such as the Home Security application. To gain adaptivity via *ImmunoPlane*, the user first describes the logical dataflow topology of the application as a DAG using the *ImmunoPlane* DSL. A `node` is used to define a service component, and an `edge` is used to express the communication link between two components. Then, for each of the `node`s and `edge`s in the DAG, the user declares requirements such as *high service availability* or *minimum link throughput* using policy directives.

Figure 3 shows the description of the Home Security application written in the *ImmunoPlane* DSL. Through various constructs (discussed in §IV-B), the *ImmunoPlane* DSL allows users to characterize the features of an application, so that it can provide an adaptation mechanism tailored for that application. Thus, we address the challenges C1 - C4 (§III-A).

To deploy the application, the user submits the DAG and the set of requirements to the *ImmunoPlane* Scheduler. The Scheduler is a separate component that has full visibility of the compute and network resources available on the *ImmunoPlane* Workers running on each of the devices in the network. We propose a novel placement algorithm in the Scheduler that takes into account the user requirements and the characteristics of the runtime infrastructure, thus addressing the challenge C5 in §III-B. Based on the user-provided requirements and the currently available resources, the Scheduler computes a concrete *adaptive deployment plan*, consisting of the placement

```
1  graph HomeSecurity (camera, outfile, clientEmail) {
2    node streamer: process("node", "streamer.js " + camera);
3    node detector: process("node", "detector.js", { outFormat: "
         json" });
4    node recorder: process("node", "recorder.js " + outfile);
5    node viewer: process("node", "viewer.js");
6    node notifier: process("node", "notifier.js " + clientEmail,
         { inFormat: "json" });
7
8    edge e1: streamer -> detector;
9    edge e2: streamer -> recorder;
10   edge e3: streamer -> viewer;
11   edge e4: detector -> notifier;
12 }
13
14 policy BasicPolicy () for HomeSecurity {
15     place (#camera || #webcam) streamer;
16     always () detector;
17     save (30) recorder;
18     min_rate (2.5) e1, e2, e3;
19 }
```

Fig. 3: The Home Security example application and its application-level requirements written as a `graph` and `policy` respectively, using the *ImmunoPlane* DSL

of `node`s and the necessary feedback control mechanisms for adapting to changing conditions. Each Worker runs the `node`s assigned to it and the corresponding adaptation mechanisms.

B. Domain-specific Language

As mentioned above, the two main constructs in our DSL are: ① a dataflow `graph` for describing the logical dataflow topology of the application, and ② a `policy` containing the performance and dependability requirements for each `node` and `edge` in the `graph`. We focus our discussion mainly on ② in this section, as our `graph` construct is based on a generic DAG used in many other flow-based programming systems such as Node-RED [18], Apache NiFi [19], and OneOS [7].

Dataflow Graph. We have added two features in our DAG construct. The first is *typed edges* for indicating the type of message exchanged between two `node`s. The message type information enables *ImmunoPlane* to correctly slice the message segments at the binary level in the network buffer. This further allows *ImmunoPlane* to dynamically route messages over parallel streams, and to sequence the messages coming from concurrent streams. The second feature is the support for *parallel (split/merge) edges* for indicating whether a dataflow can be safely parallelized without synchronizing the states. The Scheduler leverages this information to further tailor the adaptivity mechanism for the given application.

Policy Directives. A user provides a `policy` that accompanies a `graph`. In the `policy`, a user can declare various *requirements* for each `node` and `edge` in the `graph` using a set of *policy directives*. *ImmunoPlane* currently supports five types of directives, which we describe below. We found that these directives were sufficient for satisfying the requirements of the benchmark applications we used (§V); however, additional directives can easily be supported, if needed.

- `always x, y, ... z` – indicates that components `x`, `y`, and `z` must be highly available. While achieving

100% availability is extremely difficult, *ImmunoPlane* maximizes availability on a best-effort basis.

- `min_rate|max_rate (c) p, q, ... r` – indicates the minimum/maximum data transfer rate required for edges `p`, `q`, and `r` is `c` MB/s (alternatively, `c` can be provided in terms of messages/s using `mps(c)`)
- `order_by (c) p, q, ... r` – used to indicate the message order for merge edges (`-*>` in our DSL), if the downstream components expect to receive messages in a particular order. `c` can be an anonymous function invoked for each message to return a custom index value.
- `save (c) x, y, ... z` – indicates that the state of the components `x`, `y`, and `z` must be persistent and recoverable at least up to `c` seconds prior to a failure.
- `place (c) x, y, ... z` – used when a `node` has a hard placement constraint, evaluated by the boolean function `c`, e.g., it needs to be placed on a specific device.

Note that all the directives are *declarative*, namely they are used to *state a requirement*, rather than *imperatively* describing how to achieve the requirement. Further, none of the directives require any infrastructure-specific knowledge. We discuss how each directive influences the Scheduler's placement and configuration decisions in the next section.

C. Middleware Architecture

The *ImmunoPlane* middleware system consists of a Scheduler and a set of Workers (§IV-A) organized in the conventional manager-worker architecture. The Scheduler, working as the manager, is responsible for producing the adaptive deployment plan, and each Worker is responsible for executing the part of the plan for which it is responsible. The Worker manages the local components it hosts, and is able to pause, resume, and checkpoint a running component. It periodically reports its local resource usage (CPU, memory, and disk) to the Scheduler. Since the Worker's role is straightforward, we focus on the Scheduler's operation.

The Scheduler's decision making involves two stages: ① *graph configuration stage* in which the given dataflow topology is modified and instantiated, and ② *component placement stage* in which the `node` instances are assigned to the Workers. At the end of the procedure (shown in Algorithm 1, the Scheduler produces an *adaptive deployment plan*, which is basically a concrete placement of `node` instances, plus a set of special *adaptation nodes* that implement the feedback control loop for adaptation. The user-provided policy directives are used in both the stages for different purposes.

Graph Configuration Stage. During this stage, the abstract dataflow `graph` is *instantiated* – similar to a `class` being instantiated in a program – by enumerating the actual processes to be spawned for each of the `node`s. For example, there might be a single `node` in the `graph`, which can be instantiated as multiple replica processes, if the Scheduler decides to parallelize the `node`. We shall refer to such processes as `node` *instances* to differentiate them from the abstract `node`s.

Based on the policy directives, the Scheduler modifies the `graph` by adding special *adaptation nodes* to implement the appropriate adaptation mechanism. We discuss three different modifications, shown in Figure 4:

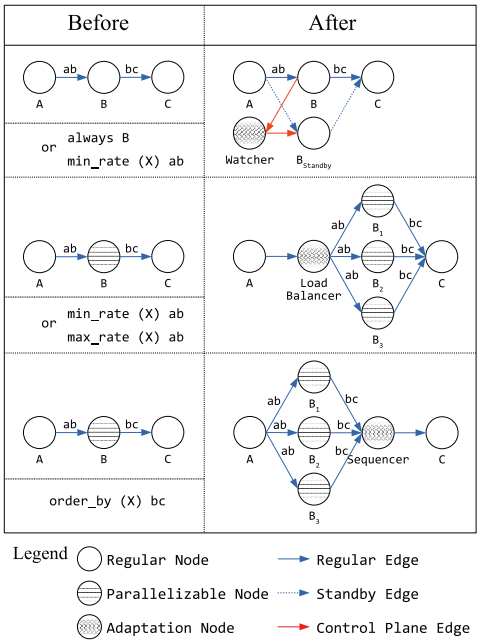


Fig. 4: Three types of adaptation nodes added to the dataflow graph based on the user-provided policy directives.

1) *Standby and Watch.* (Top row in Fig.4) For implementing a failover mechanism for a given primary `node` instance, a secondary *standby* `node` instance and a special *watcher* `node` (an adaptation node) is injected into the `graph`. The *watcher* monitors the liveness of the primary instance through a heartbeat mechanism, and activates the *standby* instance if the primary instance fails. This mechanism is employed when a `node` requires high availability (indicated by the `always` directive), or when a `node` requires a minimum input rate (indicated by the `min_rate` directive) but is not parallelizable.
2) *Load balancer.* (Middle row in Fig.4) For managing the data transfer rate along a parallel `edge`, a *load-balancer* `node` (an adaptation node) is injected between the source and the sink `node`s to maintain the required transfer rate (indicated by either the `min_rate` or `max_rate` directive). It dynamically routes the messages from the source to different sinks based on the backpressure observed at the outgoing links to the sinks.
3) *Sequencer.* (Bottom row in Fig.4) In some cases, a `node` that receives messages from multiple sources must receive them in a specific order (e.g., a component calculating running average over a time series data). The Scheduler injects a *sequencer* `node` (an adaptation node) right before the `node` that requires orderly data. While this node does not implement an adaptation mechanism on its own, it is a necessary component for ensuring the correctness of the application.

Aside from the modifications done at the graph level, the Scheduler also configures individual nodes and edges. For example, the Scheduler defines a checkpointing interval for a node if a save directive is used to indicate that its state should be persistent. Based on this node configuration, the Worker hosting this node periodically creates checkpoints.

Component Placement Stage. After a graph is instantiated and there are concrete node instances to be spawned, the Scheduler needs to determine the placement of the node instances on different Workers, such that all the user requirements can be satisfied. This placement problem is basically a type of *vector bin packing* problem [20], which is NP-hard. However, the Scheduler only aims to find one viable solution (not the optimal solution), and does not need to search the entire solution space. We use traditional search optimization techniques such as branch pruning and prioritization.

Given this problem formulation, we chose a baseline placement policy of *ranked round-robin placement*, in which we try to *minimize co-tenancy* of components rather than minimize the number of Workers used. By *ranked*, we mean that the round-robin placement prioritizes devices with more resources. The baseline policy is used if no policy directives are provided (i.e., the user does not specify any requirements). We chose a baseline policy of minimal co-tenancy as previous work [21], [22] has found that co-tenancy degrades overall performance.

The user-provided policy directives produce a custom heuristic used to guide the search process into different paths from where the baseline policy would have traversed, prioritizing paths that lead to solutions that satisfy all the given user requirements. For example, if there is a min_rate directive defined over the edge between nodes A and B, and assuming that A is assigned to Worker X, the Scheduler will override its aversion for co-tenancy and thus prioritize Worker X when placing node B. Such Worker selection priority is captured by a node-specific *score function* $score_{node}(worker)$ constructed from the policy directives. Unlike global search heuristics, our heuristics based on policy directives are localized to single search steps. Thus, the Scheduler is *application-aware*, as it derives a separate score function for each node, and applies a localized heuristic at each step of the search.

Algorithm 1 shows the pseudo-code for how the Scheduler constructs the score functions and searches for a placement. The main feature of this algorithm is in "flattening" the various node-specific requirements into a node-specific *score function*. As we iterate over each policy directive, we create a new score function for each node by combining the existing score function with the partial, directive-specific score function. The flattened score function simultaneously captures both *constraints* and *preferences* for a Worker, and is used both to eliminate requirement-violating Workers (branch pruning) and to favor requirement-optimizing Workers (branch prioritization) when selecting a Worker to place the node.

The algorithm works as follows. We search for a valid placement of node instances through an exhaustive depth-first search, using the score functions to guide the search. We start by looking for a Worker to place the most upstream node in the graph. We rank the Workers by using the score function, filtering out ineligible Workers whose score is zero (lines 27 - 28). We place the node instance on the highest ranking Worker, then move on to the next node instance (lines 30 - 31). We repeat the same process, ranking the eligible Workers and placing the node instance on the highest scoring Worker (lines 23 - 31, via recursion). If no eligible Worker is found (i.e., all the Workers have a score of zero), we backtrack to the previous node instance, and then place it on the next highest ranking Worker (lines 32 - 35). Though we perform an exhaustive search, the search is fast in practice due to our heuristics.

```
1  function PlaceApp(graph: Graph, policy: Policy) :
     Map<Node,Worker>
2      NodeList ← TopologicalSort(graph.nodes)
3      ScoreFunctions ← Map<Node, Func<Worker,float»
4      foreach declaration in policy.directives["place"] do
5          scoreFunc ←(worker: Worker) ⇒ (
6              declaration.predicate(worker) ? 1 : 0
7          )
8          foreach operand in declaration do
9              prevFunction ← ScoreFunctions[operand]
10             ScoreFunctions[operand] ←(worker: Worker) ⇒ (
11                 prevFunction(worker) × scoreFunc(worker)
12             )
13         end
14     end
15     /* repeat similar steps for all other directives */
16     Assignment ← Map<Node,Worker>
17     found ← FindAssignment(NodeList, Assignment)
18     if !found then
19         throw PlacementError
20     end
21     return Assignment
22 end
23 function FindAssignment(nodes: List<Node>, assign:
     Map<Node,Worker>) : bool
24     if nodes.length == 0 then
25         return true
26     end
27     scoreFunc ← ScoreFunctions[nodes[0]]
28     eligibleWorkers ← Workers.sort(scoreFunc)
29     foreach selected in eligibleWorkers do
30         assign[nodes[0]] ← selected
31         found ← FindAssignment(nodes[1:], assign)
32         if found then
33             return true
34         end
35         assign.remove(nodes[0])
36     end
37     return false
38 end
```

Algorithm 1: Placement Algorithm

We walk through the pseudocode using our Home Security application example (Fig. 3). At the beginning of the search, we create five score functions $score_{x,0}(worker)$, one for each node x. Initially, the score function returns a score based on the baseline policy. Then, for the place directive applied for the streamer node, a partial function $f_s(worker)$ is created for the streamer node, which returns either 1 or 0 based on whether the given $worker$ has a camera (lines 5 - 7). A new score function ($score_{s,1}(worker)$) is created for streamer, which is $score_{s,0}(worker) * f_s(worker)$ (lines 9 - 12). Next, for the min_rate directive, the score functions of both the source and sink nodes of the edges e1, e2, and

Requirement	Benchmarks	Adaptations Used	Evaluation Criteria
Availability	Home Security [11]	Standby and watch	Mean Time To Recover
Processing Latency	YSB [8] DSP-FraudDetection DSP-TrafficMonitoring DSP-AdsAnalytics DSP-ClickAnalytics DSP-SmartGrid DSP-SpikeDetection [10]	Load balancer, Sequencer	Percentile Latency
Throughput	DSP-LogProcessing DSP-WordCount [10] RIoT-ETL RIoT-STATS RIoT-TRAIN RIoT-PRED [9]	Load balancer, Standby and watch	Mean Throughput

TABLE I: Applications used for evaluation. The last column shows the evaluation criteria used for each application.

`e3` are updated in a similar manner. For example, for edge `e1` connecting `streamer` and `detector`, two partial functions $g_s(worker)$ and $g_d(worker)$ are created, for `streamer` and `detector` respectively. g_s returns a score based on the available bandwidth of all the outgoing links, such that a worker with the most egress bandwidth receives the highest score. $g_d(worker)$ also returns a score, but based on the available ingress bandwidth. The partial functions are then multiplied by the existing score functions for `streamer` and `detector`, producing $score_{s,2}$ and $score_{d,1}$ respectively. After processing all policy directives, each node has an associated score function $score_x(worker)$, which is used to evaluate the "goodness" of a Worker for placing the given node.

D. Implementation

We implemented the ImmunoPlane Scheduler and integrated it into OneOS [7], an IoT platform we developed in our prior work. The implementation uses C# and the .NET 2.0 standard.

V. Evaluation

We evaluate *ImmunoPlane* in two parts. *First*, we evaluate whether *ImmunoPlane* is able to provide adaptivity to different types of applications, and for different user requirements (i.e., *application-awareness*). *Second*, we evaluate whether *ImmunoPlane* can provide adaptivity under different deployment infrastructures, without requiring any programming or configuration effort (i.e., *infrastructure-independence*).

A. Part 1: Support for Different Requirements

Evaluation Strategy. We investigate *ImmunoPlane*'s ability to provide adaptivity to different applications with varying performance and dependability requirements. To this end, we selected a variety of benchmark applications with different dataflow topology and requirements. For each application, we expressed the dataflow `graph` and declared its requirements using the *ImmunoPlane* DSL (summarized in Table I). We used 14 benchmarks taken from 4 different papers, and broadly categorized them into three groups based on the application's requirements. We deploy each application on *ImmunoPlane*, and study the effect of both failures and network congestion.

Experimental Setup. Though *ImmunoPlane* was designed for IoT infrastructures (which might be more resource-constrained), we used a resource rich, cloud-like infrastructure for *this part* of the experiment, because we were interested in studying *ImmunoPlane*'s ability to adapt under *external events*, rather than internal events such as resource contention. We consider IoT-based infrastructures in the next part (§V-B).

We used six NUMA machines all located in the same rack. Each machine has two Xeon E5-2640 @ 2.5 GHz, with 6 cores each and with hyper-threading enabled (24 logical CPUs in total), 64 GB RAM, and 2.4 TB HDD. The machines were connected through a gigabit Ethernet switch, and have an average communication latency of 0.279±0.037 ms.

We hosted the Scheduler exclusively on a machine, and used five machines to run Workers. We configured *ImmunoPlane* to be sensitive to failures and network congestion, to promote adaptations for our experiment. The Workers were configured to send a heartbeat every second, and the Scheduler was configured to consider even a single missed heartbeat as a disconnection. To observe adaptation to network congestion, we configured the Workers to adapt after five continuous seconds of network requirement violations. This interval was long enough to tolerate natural jitter in throughput measurements, but short enough to detect true congestion. We consider three requirements, (1) availability, (2) latency and (3) throughput.

1) Availability Requirement: We evaluate the ability of *ImmunoPlane* to ensure high availability for the Home Security benchmark, even when the devices (Workers) hosting these components fail randomly. The Home Security benchmark (Fig. 1) contains three data processing pipelines. Among them, the pipeline including the `detector` and `notifier` components needs to be highly available (see Section II).

There are three steps in the experiment. First, using the DSL, we indicate that the `detector` and `notifier` components need to be highly available, using the `always` directive. We then submit the application description (written in the DSL as shown in Fig. 3) to the Scheduler, and wait until the application is deployed. At a random time during the application's execution, we crash the Worker hosting the `detector` component. Finally, we measure the mean time to recover (MTTR) of the `detector`, i.e., the time between the device failure and the restarting of `detector` component on another device.

We observed an average MTTR of 1364.6±275.1 ms. Based on the recovery time obtained, we compute the $availability = \frac{MTBF}{MTBF+MTTR}$. We plot the availability curve as a function of the MTBF (mean time between failures), which varies among devices. Fig. 5 shows the availability curve (blue line) in logarithmic scale. We evaluate whether we can achieve three-nines (99.9%) availability, which is the minimum considered as "high availability" in both academia [23], [24] and industry (e.g., most cloud service providers issue 25%-100% refunds if the availability drops below 99.0%, or two-nines [25], [26]). We find that *ImmunoPlane* can provide three-nines availability under random node failures, assuming the host machine fails every 23 minutes (1363.24 seconds). This is a conservative assumption, as many IoT devices have higher MTBFs, e.g., health-tracking IoT devices fail about four times a day [27].

2) Processing Latency Requirement: The Yahoo Streaming Benchmark (YSB) [8] and six of the DSPBench applications

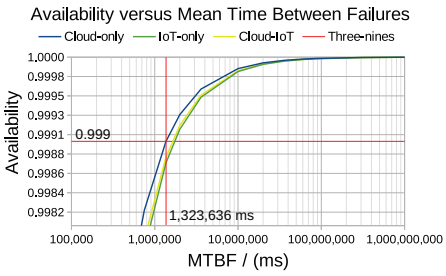


Fig. 5: Availability of application versus MTBF of the host machine, under a single node failure, in three different infrastructures. MTTR: in Cloud-only = 1365 ms, in IoT-only = 1714 ms, in Cloud-IoT = 1622 ms. Higher availability is better, lower MTTR is better.

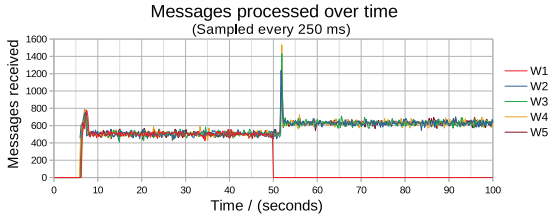


Fig. 6: Messages processed by each Worker (W1 to W5) every 250 ms, during the first 100 seconds of execution. Higher values are better.

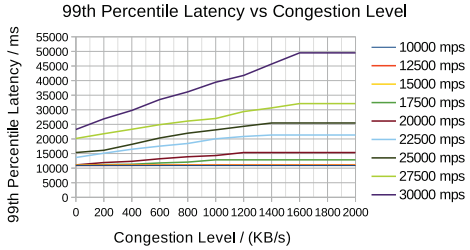


Fig. 7: 99th Percentile latency for the YSB application under varying levels of network congestion. The congestion level in the x-axis indicates the reduction in bandwidth, and the y-axis shows the 99th percentile latency (p99 latency). Lower values are better.

shown in Table I comprise parallel or concurrent processing pipelines, where the input workload is distributed among multiple components. For these applications, we evaluate *ImmunoPlane*'s ability to deliver low processing latency, even when under network congestion. We discuss the results for the YSB, but we obtained similar results for DSPBench programs.

The YSB consumes input messages via Kafka [28], filters and transforms messages in parallel, and commits processed messages into Redis [29]. Because the YSB was developed for SPFs such as Flink and Spark, we had to remove all Flink-specific deployment and configuration code in order to run them as *ImmunoPlane* components. We followed a similar process for the DSPBench applications, which had Storm-specific code. We verified that we preserved the application's semantics by comparing the outputs produced by our version against those produced by the original implementation.

We also used the unmodified, original test client written by the authors of YSB. We can set the input message rate (in *mps* – messages per second), and the corresponding benchmark performance is measured by reading the timestamps in the output messages and calculating the end-to-end processing latency. Prior to performing our experiments, we profile the baseline processing capacity of YSB (maximum message processing rate without violating end-to-end latency requirements, and under no faults). We measured the baseline capacity to be $\approx 20,000$ mps. We test *ImmunoPlane*'s ability to adapt for a given minimum required processing rate of N mps, which we indicate using the DSL directive `min_rate (N mps)`.

We consider two scenarios. In the first scenario, we randomly crash a Worker as we did for the Home Security benchmark. In the second scenario, we introduce congestion at a target link by using the Linux Traffic Control (`tc` [30]) to limit its bandwidth. At a random time during the application's execution, we limit the bandwidth between one of the parallel components and the sink component for a 60 second duration.

We observe that, upon failure, the volume of data handled by the failed Worker is evenly distributed among the rest of the Workers (Fig. 6), as *ImmunoPlane* employs round-robin message distribution unless a distribution policy was provided by the user. The end-to-end percentile latency is *unaffected by the failure*, as *ImmunoPlane* adapts quickly to the failure.

We examine how *ImmunoPlane* helps the application adapt as we vary the bandwidth at the target link (Fig. 7). The expected 99th percentile latency (p99) is around 10,000 ms [8]. We observe that, when the required message rate is under 15,000 mps, the p99 latency remains around 11,000 ms, unaffected by the congestion. For message rates between 17,500 and 20,000 mps, the application delivers expected performance without congestion. However, we observe a degradation in performance as congestion approaches 1 to 1.2 MB/s. This indicates a point at which the application is unable to deliver the performance required even with adaptation. As we increase the message rate further, the effect is more pronounced, with the p99 latency approaching 50 seconds. Thus, we infer that *ImmunoPlane* can adapt to deliver the required performance for YSB as long as the user requires processing rate of 15,000 mps or less, which is 75% of the maximum processing capacity. The user needs to profile the application a priori (as we did) to obtain the maximum processing capacity.

3) Throughput Requirement: We used four RIoTBench [9] applications and two of the DSPBench applications - these applications have similar dataflow topology as the benchmarks used in § V-A2. However, they prioritize throughput rather than latency, as they process large input files offline.

The RIoT-ETL and DSP-WordCount applications are similar to YSB; they have a parallel processing pipeline through which the input data is distributed uniformly. Therefore, the behavior we observe in RIoT-ETL and DSP-WordCount is the same as that of YSB. Upon a failure or congestion, the

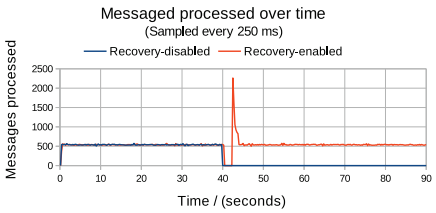


Fig. 8: Messages processed over time by the `Average` component in RIoT-PRED. Higher values are better.

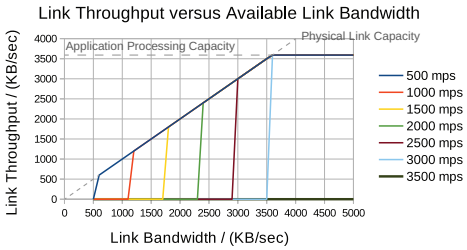


Fig. 9: Observed throughput versus available bandwidth for the `Average` component in RIoT-STATS. Higher values are better.

messages intended for the failed Worker are distributed to the other Workers. Therefore, we do not discuss these results.

The other three benchmarks in RIoTBench and the DSP-LogProcessing applications exhibit a different behavior when dealing with failure or congestion. Since these benchmarks contain pipelines that are not parallel, data cannot be routed to another component when a pipeline fails or is congested. Therefore, for the components that do not have parallelism, *ImmunoPlane* applies the *Standby and Watch* mechanism in order to make them highly available. Fig. 8 illustrates what happens to a non-parallel pipeline with and without the *Standby and Watch* mechanism. Without it, the application does not recover from the failure, and the processing stops immediately. With it, the stream is routed to the standby component upon failure. As a result, the end-to-end throughput requirement is satisfied.

To observe the effect of congestion, we perform similar experiments as those in § V-A2; we deploy the benchmark, then at a random point in time, we limit the bandwidth of the input stream of one of the components for 60 seconds using `tc`. We run this experiment for varying levels of required throughput, set using the `min_rate` directive.

Fig. 9 shows the observed link throughput for the target link (input to the `Average` component[2]) in the RIoT-STATS benchmark. The curve $y = x$ is the link capacity – throughput can never exceed the physical bandwidth. When we set the required throughput to 500 mps, we observe that the link throughput is at maximum capacity. The throughput plateaus when the bandwidth is greater than 3.5 MB/s, indicating that

[2]We refer the reader to the original RIoTBench [9] and DSPBench [10] papers for detailed topology and component description.

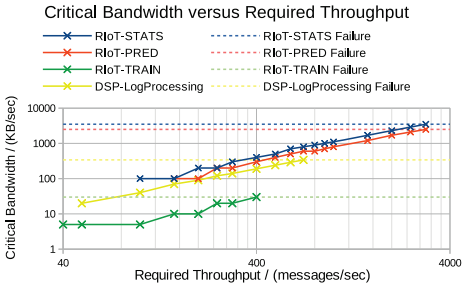


Fig. 10: Critical bandwidth at which adaptation is triggered, for different levels of user requirement.

the application cannot deliver a higher throughput. When the link bandwidth is set to 500 KB, the throughput drops to zero. This is the point (we call it the *critical bandwidth*) at which *ImmunoPlane* triggers adaptation, migrating the component to another Worker with a larger available bandwidth. As we increase the required throughput, we observe that the critical bandwidth also increases, indicating that *ImmunoPlane*'s adaptation strategy adjusts to the user requirements.

Fig. 10 shows the critical bandwidth for the benchmarks, plotted in log-log scale for different values of user-required throughput. We observe that the critical bandwidth varies among the benchmarks, showing that *ImmunoPlane* transparently adapts for different benchmarks. However, when the user requirement exceeds 3000 mps for RIoT-STATS and RIoT-PRED, *ImmunoPlane* cannot satisfy the requirement as it either exceeds the physical capacity or the application capacity. The throughputs for DSP-LogProcessing and RIoT-TRAIN are orders of magnitude smaller, but we observe the same trend.

Summary of Part 1. Using 14 benchmark applications, we evaluated *ImmunoPlane*'s ability to support different user requirements in a cloud-like infrastructure. For each benchmark, we have used our DSL to express the requirements in terms of policy directives, and then observed whether *ImmunoPlane* adapts to satisfy the requirements under random failures and different levels of congestion. We find that *ImmunoPlane* is able to deliver 99.9% availability even assuming a single device failure every 23 minutes. In terms of satisfying latency or throughput requirements, *ImmunoPlane* is able to satisfy them as long as the requirement is set to less than 75% of the baseline processing capacity. This is an approximate lower bound for the performance level under which *ImmunoPlane* can provide adaptivity, and is lower than those claimed for SPFs (90%) [31], [32]. Determining a precise performance requirement to set (such as *maximum sustainable throughput*) is out of the scope of this work – this is a subject of active research [31], [32]. We also observe that *ImmunoPlane* triggers adaptations at different congestion levels, depending on the requirements set by the user. Overall, *ImmunoPlane* can support different application-level requirements specified.

B. Part 2: Adaptivity in Different Infrastructures

In this part, we examine whether *ImmunoPlane* can transparently provide adaptivity in different deployment infrastructures. We take the same benchmark applications and requirements from the previous part (§ V-A), and deploy them in two additional infrastructures: *IoT-only*, and *Cloud-IoT* infrastructures. We introduce random device failures and congestions as we did for the *Cloud-only* infrastructure, and observe how *ImmunoPlane* adapts in different infrastructures.

Experimental Setup. We use two setups in addition to the cloud-only setup in §V-A. The first is the *IoT-only* testbed, which consists of four Raspberry Pi 3s and two Raspberry Pi 4s. The Raspberry Pi 3 is equipped with a quad-core 1.2 GHz ARM7, 1 GB RAM, and 32 GB Micro SD. The Raspberry Pi 4 has a quad-core 1.5 GHz ARM8, 4 GB RAM, and 64 GB Micro SD card. One of the Raspberry Pi 4s is used to host the Scheduler, and the five other Raspberry Pis run the Workers. They are connected wirelessly over a gigabit router. The second setup is the *Cloud-IoT* testbed, consisting of three NUMA machines – used in Part 1 – and three Raspberry Pi 3s. One of the NUMA machines is used for the Scheduler.

As before, we consider the same three requirements, namely availability, latency, and throughput.

1) Availability Requirement: We perform the same random failure experiment from Part 1 on the IoT-only and Cloud-IoT testbeds. For the Cloud-IoT testbed, we assign an IoT device (Raspberry Pi 3) to run the `streamer` component (the data source). From the user's perspective, no additional code or configuration is required to run the application. However, the Scheduler produces a different placement for each infrastructure to satisfy the availability requirement.

The MTTR is 1713.8 $\pm$ 311.5 ms in the IoT-only testbed and 1622.1 $\pm$ 207.9 ms in the Cloud-IoT testbed, taking marginally longer than in the Cloud-only (1364.6 ms) testbed. We calculate that *ImmunoPlane* can provide three-nines availability when the MTBF is 29 minutes (1712.09 seconds) or greater in the IoT-only testbed, and the MTBF is 27 minutes (1620.48 seconds) or greater in the Cloud-IoT testbed (Fig. 5). Due to slower recovery, a larger MTBF is required in the IoT-only and Cloud-IoT testbeds to achieve the same availability as in the Cloud-only testbed.

Note that in homogeneous infrastructures (Cloud-only and IoT-only), the Scheduler produces similar plans, distributing the application components uniformly among the Workers. However, in the Cloud-IoT testbed, the Scheduler predominantly favors cloud Workers, since there are more resources available on cloud machines. As a result, the two cloud machines host all the components except the `streamer`. Upon a failure, one of the IoT Workers restarts the failed components.

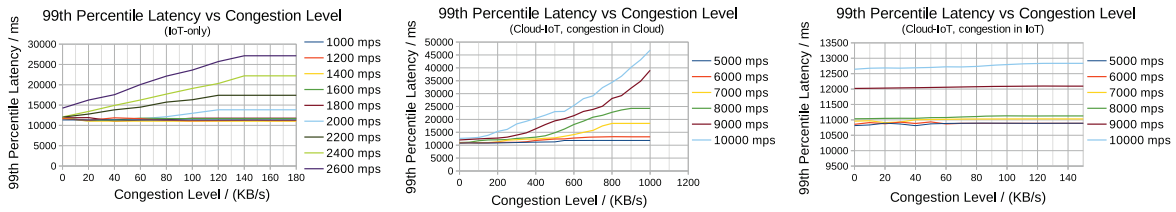
2) Processing Latency Experiment: We run the YSB in the two additional testbeds. Because each testbed has a different physical capacity, we set different input rates for them. We performed a profiling run to obtain the appropriate range of message rates (as we did for Part 1). For the IoT-only testbed, the input rate is set in the range of 1,000 to 2,600 mps, and for the Cloud-IoT testbed, the input rate is set in the range of 5,000 to 10,000 mps.

The general trend is that the p99 latency at different congestion levels (Fig. 11a) is similar to that of the Cloud-only testbed from Part 1 (Fig. 7). The only difference we observe is that the processing capacity of the IoT-only testbed is around 1,800 mps (12% of Cloud-only testbed).

In the Cloud-IoT testbed, we experimented with introducing congestion either in the cloud (Fig. 11b) or in the IoT links (Fig. 11c). We observe that the application performance is much more sensitive to congestion in the cloud. *ImmunoPlane* is able to adapt to congestion when the required throughput is under 5,000 mps or if the congestion is less than 200 KB/s. However, the performance degrades quickly as the congestion approaches 1 MB/s. When congestion was introduced in the IoT link, the application performance is not affected as much by varying congestion levels. Consequently, the application requirements are met as long as they are under 8,000 mps, but not when they are higher.

3) Throughput Requirement: Again, we run the same experiments from Part 1, in the IoT-only and Cloud-IoT testbeds. The overall behavior is similar to that for the Cloud-only testbed (§V-A3) – i.e., *ImmunoPlane* adapts by activating the standby component when the available link bandwidth becomes critically limited. In both testbeds, *ImmunoPlane* is able to adjust the critical bandwidth depending on the user requirement, just as it did in the Cloud-only testbed. Hence, we only highlight the notable differences. For the IoT-only testbed, we observe that the range of critical bandwidth is much lower (in the range of 50 to 200 KB/s), understandably because the throughput requirement is set to a lower value to reflect its physical capacity. For the Cloud-IoT testbed, we observe that the Scheduler always places the target component and its standby component on the cloud, so that upon a requirement violation, another cloud Worker takes over. Thus, the link throughput we observe for the Cloud-IoT testbed is in the same ballpark as in the Cloud-only testbed.

Summary of Part 2. We evaluated *ImmunoPlane*'s ability to adapt in different deployment infrastructures. For each infrastructure, we profiled the baseline processing capacity for each benchmark, so that we can set a reasonable performance requirement for our experiments. We find that *ImmunoPlane* is able to provide similar levels of availability across the Cloud-only, IoT-only, and Cloud-IoT testbeds. However, there are minor differences caused by IoT devices taking slightly longer than cloud devices to recover failed components. We observe that *ImmunoPlane* employs similar adaptation strategies in different types of homogeneous infrastructures (Cloud-only and IoT-only). In the Cloud-IoT testbed, *ImmunoPlane* favors running more work on the cloud, and thus is more sensitive to failures and congestion in the cloud. Overall, *ImmunoPlane* transparently produces deployment plans optimized for different infrastructures to meet the user requirements.



(a) IoT-only (b) Cloud-IoT, congestion in Cloud (c) Cloud-IoT, congestion in IoT

Fig. 11: 99th Percentile latency under varying levels of congestion, in different infrastructures (testbeds). Lower values are better.

VI. Related Work

Stream Processing Frameworks. As many IoT applications involve processing data streams, we studied stream processing frameworks (SPFs) to see whether they can provide adaptivity for IoT applications. SPFs such as Apache Kafka [28], Flink [3], Storm [33], and Spark [4] provide built-in adaptivity features such as failure recovery and load-balancing. A user can leverage these features through the framework's API, in an infrastructure-agnostic manner. However, SPFs are specialized for parallel processing of large datasets. Thus, they optimize either for end-to-end latency (e.g., Kafka, Flink, Storm) or for overall throughput (e.g., Spark). Although SPFs provide infrastructure-agnostic adaptation capabilities, they support only a fixed set of global requirements, and do not support component-specific local requirements.

IoT/Edge Computing Frameworks. IoT and edge computing frameworks generally do not provide built-in support for adaptivity. Instead, they provide specialized services (often cloud-based) that can serve as building blocks for adaptivity. For example, AWS IoT Greengrass [34] provides services such as IoT Device Shadow for monitoring devices, Amazon S3 for reliable data storage, and Amazon SQS for reliable messaging. By using the appropriate services, a user can customize their deployment plan. However, for each deployment environment, the user still needs to determine the appropriate placement and configuration, such that the application's requirements are satisfied. Although the cloud services reduce the development effort to some extent, the user still needs to design the concrete deployment plan for each application and environment.

Dataflow Optimization and Adaptivity Frameworks. There are frameworks that provide additional optimization and adaptivity features on top of existing runtime systems. These systems also aim to reduce the burden on the user.

Nemo [35] is an optimization framework that transparently produces an optimized deployment configuration for a target SPF. It compiles applications written in dataflow languages such as Beam [36] into an abstract *intermediate representation (IR)* DAG, from which concrete optimizations are generated for a target SPF such as Spark. A similar approach was taken in Musketeer [37] and Optimus [38], both focusing on systems such as Spark [4] and DryadLINQ [39]. The high-level idea of converting infrastructure-agnostic policies into concrete deployments is similar to our approach. However, they support only stream processing applications.

Steel [40] is an edge computing framework with motivations similar to ours. Steel automatically configures and deploys edge applications over the cloud and edge, and demonstrates the approach on top of the Azure ecosystem. Unlike *ImmunoPlane*, Steel requires the user to specify component placement, and does not allow users to specify different requirements.

Program Control Language (PCL) [6] is a coordination language used for enabling adaptivity in a distributed application. PCL provides APIs for specifying the metrics to observe and describing the adaptations to apply. While a user can customize the adaptation plan for each application, they must be aware of the performance characteristics of each target infrastructure.

CHARIOT [5] is a framework that facilitates autonomous management of IoT systems. It provides a DSL for describing the application components, their relationships, and the deployment configuration. For instance, it allows a user to describe which component to replicate and where, so that the component can tolerate failures. However, the user still needs to determine how to achieve the high-level user requirements using the provided functionalities in their DSL.

To summarize, systems such as Nemo and Steel transparently produce deployment plans, but they target a specific class of applications and do not support different requirements. In contrast, systems such as PCL and CHARIOT can be used to satisfy different requirements, but require the user to develop the concrete deployment plan, which requires user effort.

VII. Conclusion and Future Work

We presented *ImmunoPlane*, a middleware system that enables IoT applications to adapt to failures and network congestions in different runtime infrastructures, while satisfying their requirements, with minimal user effort. *ImmunoPlane* provides a DSL for users to express different requirements in a declarative manner, without having to describe the adaptation mechanism explicitly. By providing an adaptation layer, *ImmunoPlane* transparently converts the application requirements into an adaptive deployment plan for the underlying runtime infrastructure. We evaluated *ImmunoPlane* using 14 benchmarks, and showed that it supports various types of applications, and user requirements e.g., availability, latency, and throughput. Further, *ImmunoPlane* transparently supports the same requirements in different runtime infrastructures.

There are two directions for future work. First, *ImmunoPlane* supports common performance and dependability re-

quirements such as latency, throughput, and availability. In our future work, we want to consider requirements such as energy consumption when providing adaptivity. Second, we have conducted our experiments in a setting where there was little contention between components. We will examine the effect of resource contention in future work.

ACKNOWLEDGMENT

This work is supported by the National Defence Innovation for Defence Excellence and Security (IDEaS) program from the Department of National Defence of Canada. We thank the anonymous reviewers of IoTDI for their helpful comments.

REFERENCES

[1] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE IoT Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[2] S. A. Noghabi, L. Cox, S. Agarwal, and G. Ananthanarayanan, "The emerging landscape of edge computing," *GetMobile: Mobile Computing and Communications*, vol. 23, no. 4, pp. 11–20, 2020.

[3] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *The Bulletin of the Technical Committee on Data Engineering*, vol. 38, no. 4, 2015.

[4] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[5] S. Pradhan, A. Dubey, S. Khare, S. Nannapaneni, A. Gokhale, S. Mahadevan, D. C. Schmidt, and M. Lehofer, "Chariot: Goal-driven orchestration middleware for resilient iot systems," *ACM Trans. Cyber-Physical Systems*, vol. 2, no. 3, pp. 1–37, 2018.

[6] B. Ensink, J. Stanley, and V. Adve, "Program control language: a programming language for adaptive distributed applications," *J. Parallel and Distributed Computing*, vol. 63, no. 11, pp. 1082–1104, 2003.

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

[8] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng *et al.*, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *2016 IEEE Intl. Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2016, pp. 1789–1792.

[9] A. Shukla, S. Chaturvedi, and Y. Simmhan, "Riotbench: An iot benchmark for distributed stream processing systems," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 21, p. e4257, 2017.

[10] 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.

[11] K. Jung, J. Gascon-Samson, S. Goyal, A. Rezaiean-Asel, and K. Pattabiraman, "Thingsmigrate: Platform-independent migration of stateful javascript internet of things applications," *Software: Practice and Experience*, vol. 51, no. 1, pp. 117–155, 2021.

[12] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman, "Live video analytics at scale with approximation and delay-tolerance," in *14th USENIX Symp. Networked Systems Design and Implementation*, 2017.

[13] H. Isah, T. Abughofa, S. Mahfuz, D. Ajerla, F. Zulkernine, and S. Khan, "A survey of distributed data stream processing frameworks," *IEEE Access*, vol. 7, pp. 154 300–154 316, 2019.

[14] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski *et al.*, "Serverless computing: Current trends and open problems," *Research advances in cloud computing*, pp. 1–20, 2017.

[15] Y. Wu, "Cloud-edge orchestration for the internet of things: Architecture and ai-powered data processing," *IEEE IoT Journal*, vol. 8, no. 16, pp. 12 792–12 805, 2020.

[16] H. Lee, S. Noghabi, B. Noble, M. Furlong, and L. P. Cox, "Bumblebee: Application-aware adaptation for edge-cloud orchestration," in *2022 IEEE/ACM 7th Symp. Edge Computing (SEC)*. IEEE, 2022, pp. 122–135.

[17] L. Bittencourt, R. Immich, R. Sakellariou, N. Fonseca, E. Madeira, M. Curado, L. Villas, L. DaSilva, C. Lee, and O. Rana, "The internet of things, fog and cloud continuum: Integration and challenges," *Internet of Things*, vol. 3, pp. 134–155, 2018.

[18] (2023, May) Node-red. [Online]. Available: https://nodered.org/docs/

[19] (2023, Apr) Apache nifi overview. [Online]. Available: https://nifi.apache.org/docs.html

[20] R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder, "Heuristics for vector bin packing," *research. microsoft. com*, 2011.

[21] S. Blagodurov, A. Fedorova, E. Vinnik, T. Dwyer, and F. Hermenier, "Multi-objective job placement in clusters," in *Proc. Intl. Conf. High Performance Comput., Netw., Storage and Anal.*, 2015, pp. 1–12.

[22] X. Tang, H. Wang, X. Ma, N. El-Sayed, J. Zhai, W. Chen, and A. Aboulnaga, "Spread-n-share: improving application performance and cluster throughput with resource-aware job placement," in *Proc. Intl. Conf. High Performance Comput., Netw., Storage and Anal.*, 2019, pp. 1–15.

[23] A. Elmokashfi, D. Zhou, and D. Baltrünas, "Adding the next nine: An investigation of mobile broadband networks availability," in *Proc. 23rd Annu. Intl. Conf. Mobile Computing and Networking*, 2017, pp. 88–100.

[24] J. Mustafa, K. Sandström, N. Ericsson, and L. Rizvanovic, "Analyzing availability and qos of service-oriented cloud for industrial iot applications," in *2019 24th IEEE Intl. Conf. Emerging Technologies and Factory Automation*. IEEE, 2019, pp. 1403–1406.

[25] (2023, July) Aws service level agreements (slas). [Online]. Available: https://aws.amazon.com/legal/service-level-agreements/

[26] (2023, July) Iot core service level agreement (sla). [Online]. Available: https://cloud.google.com/iot/sla

[27] A. Mavrogiorgou, A. Kiourtis, C. Symvoulidis, and D. Kyriazis, "Capturing the reliability of unknown devices in the iot world," in *2018 Fifth Intl. Conf. Internet of Things: Systems, Management and Security*. IEEE, 2018, pp. 62–69.

[28] G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, J. Rao, J. Kreps, and J. Stein, "Building a replicated logging system with apache kafka," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1654–1655, 2015.

[29] (2023) Redis website. [Online]. Available: http://www.redis.io/

[30] M. Kerrisk. (2023, july) tc(8) - linux manual page. https://man7.org/linux/man-pages/man8/tc.8.html.

[31] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream data processing systems," in *2018 IEEE 34th Intl. Conf. Data Engineering (ICDE)*. IEEE, 2018, pp. 1507–1518.

[32] Z. Chu, J. Yu, and A. Hamdull, "Maximum sustainable throughput evaluation using an adaptive method for stream processing platforms," *IEEE Access*, vol. 8, pp. 40 977–40 988, 2020.

[33] J. Warren and N. Marz, *Big Data: Principles and best practices of scalable realtime data systems*. Simon and Schuster, 2015.

[34] (2023, May) Iot greengrass. [Online]. Available: https://docs.aws.amazon.com/greengrass/

[35] Y. Yang, J. Eo, G.-W. Kim, J. Kim, S. Lee, J. Seo, W. W. Song, and B.-G. Chun, "Apache nemo: A framework for building distributed dataflow optimization policies." in *USENIX Annu. Tech. Conf.*, 2019, pp. 177–190.

[36] H. Karau, "Unifying the open big data world: The possibilities of apache beam," in *2017 IEEE Intl. Conf. Big Data (Big Data)*. IEEE Computer Society, 2017, pp. 3981–3981.

[37] I. Gog, M. Schwarzkopf, N. Crooks, M. P. Grosvenor, A. Clement, and S. Hand, "Musketeer: all for one, one for all in data processing systems," in *Proc. 10th European Conf. Computer Systems*, 2015, pp. 1–16.

[38] Q. Ke, M. Isard, and Y. Yu, "Optimus: a dynamic rewriting framework for data-parallel execution plans," in *Proc. 8th ACM European Conf. Computer Systems*, 2013, pp. 15–28.

[39] Y. Y. M. I. D. Fetterly, M. Budiu, Ú. Erlingsson, and P. K. G. J. Currey, "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language," *Proc. LSDS-IR*, vol. 8, 2009.

[40] S. A. Noghabi, J. Kolb, P. Bodik, and E. Cuervo, "Steel: Simplified development and deployment of edge-cloud applications," in *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, 2018.