BASS: A Resource Orchestrator to Account for Vagaries in Network Conditions in Community Wi-Fi Mesh

Manasvini Sethuraman
msethuraman3@gatech.edu
Georgia Tech
Atlanta, Georgia, USA

Anirudh Sarma
anirudhs@gatech.edu
Georgia Tech
Atlanta, Georgia, USA

Netra Ghaisas
nghaisas6@gatech.edu
Georgia Tech
Atlanta, Georgia, USA

Adwait Bauskar
adwait.bauskar@gatech.edu
Georgia Tech
Atlanta, Georgia, USA

Ashutosh Dhekne
dhekne@gatech.edu
Georgia Tech
Atlanta, Georgia, USA

Anand Sivasubramaniam
anand@cse.psu.edu
Pennsylvania State University
University Park, Pennsylvania, USA

Umakishore Ramachandran
rama@gatech.edu
Georgia Tech
Atlanta, Georgia, USA

Abstract

We investigate the issue of deploying applications on a set of loosely coupled compute devices, connected through a wireless mesh, typical in community networks. Wireless mesh networks experience significant temporal and spatial variations in link bandwidth. When application components, modeled as a directed acyclic graph, need to be scheduled on such a mesh with bandwidth constraints (and variations), the problem of mapping components to specific compute nodes becomes an instance of bin packing with constraints of CPU, memory, and bandwidth limits within the mesh. To make the scheduling tractable, we propose BASS (Bandwidth Aware Scheduling System), and develop heuristics for scheduling, based on the directed graph topology of the application components. We evaluate BASS on an emulated mesh using bandwidth traces collected from an actual wireless testbed - CityLab. Detailed evaluations show that contemporary orchestration frameworks can plug in BASS to provide better end-to-end performance for the applications deployed on the mesh while reducing resource utilization.

CCS Concepts

• **Computer systems organization** → **Distributed architectures**; **Distributed architectures**; • **Networks** → **Wireless access networks**.

MIDDLEWARE '24, December 2–6, 2024, Hong Kong, Hong Kong

ACM ISBN 979-8-4007-0623-3/24/12.
https://doi.org/10.1145/3652892.3700754

ACM Reference Format:

Manasvini Sethuraman, Anirudh Sarma, Netra Ghaisas, Adwait Bauskar, Ashutosh Dhekne, Anand Sivasubramaniam, and Umakishore Ramachandran. 2024. BASS: A Resource Orchestrator to Account for Vagaries in Network Conditions in Community Wi-Fi Mesh. In *25th International Middleware Conference (MIDDLEWARE '24), December 2–6, 2024, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3652892.3700754

1 Introduction

Today's internet landscape is diverse, both in terms of the underlying infrastructure and the applications that run on it. At one end, we have individual users at the edge, running applications on their personal devices, and at the other end, we have cloud operators who provision compute and memory, running services in data centers. The two are connected through a combination of a volatile edge network, and a relatively stable backbone. The edge network which provides the "last mile" connectivity to end users is often the most variable, in capacity, reliability, and availability. This last mile comprises traditional broadband, cellular networks, wireless ISPs, satellite, and mesh networks. While service deployment and operation for data centers with reliable network connectivity has been well studied, the performance of these services in settings exposed to the vagaries of weather, obstructions, and other unpredictable conditions, as experienced by wireless mesh networks, is not well understood. Fig. 1 depicts a community mesh network, with applications deployed on it. Running applications like messaging, video sharing, and other services is standard in community mesh networks [24]. These mesh networks often provide essential services to support severe weather situations, search and rescue operations after disasters, etc. In fact, a community-run wireless mesh network in New York City's Red Hook neighborhood was the only operational network during Hurricane Sandy [4]. Furthermore, past research has shown that wireless infrastructure is

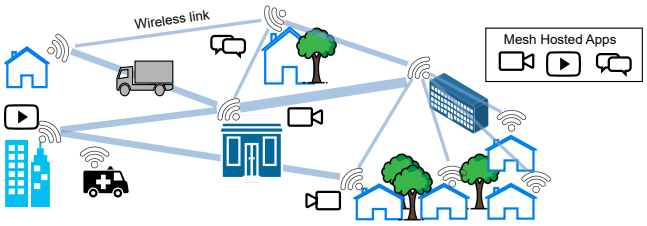


Figure 1: Applications hosted on a wireless mesh infrastructure. Wireless links have different bandwidths and also experience temporal variations due to environmental factors, such as reflections from a truck or attenuation from foliage.

usually faster to recover from weather-related outages [44], and sometimes, wireless infrastructure is the only available option for connectivity to its users [54], and without services running on this infrastructure, simply having localized connectivity is of limited value.

An attribute that differentiates mesh networks from datacenter networks is the bandwidth variability on links between nodes. In this work, we show that bandwidth variations are commonplace in wireless links, and impact the performance of applications running at the edge. While many prior works perform traffic-aware placement of applications, they do so in resource rich environments like data centers, with reliable connectivity, for which there are theoretical formulations, solvers, and approximation algorithms [6, 39, 46, 49]. However, our focus is on simple heuristics that allow for agile application component placement mechanisms, which are practical and adaptive to fluctuations in bandwidth capacity on constrained facilities at the edge. While orchestrators like k3s [48] have been deployed in edge environments, they operate on edge clusters with wired links [7], or in the edge-cloud continuum with fixed bandwidth [26], but not in wireless mesh environments with variable bandwidth.

Wireless links vary in bandwidth over time due to interference and fading effects, and services may require migration when their resource requirements are no longer met. Resource management is further complicated by the fact that on ad-hoc wireless mesh networks, routing is decentralized, and therefore the developer has limited control over how packets are routed. Changing the routing behavior in response to bandwidth variations could be a potential solution, and there are proposals to make a hierarchical SDN-like control plane for wireless mesh networks [25]. In this work, we instead build bandwidth awareness into the resource orchestration system that runs on top of the wireless mesh-connected compute nodes, capable of working with any routing mechanism, as long as there is no partitioning of the network and/or node failures.

In this work, we develop heuristics for making scheduling decisions tractable. We show that our scheduler, which is developed as an extension for k3s is able to make feasible scheduling decisions and leads to lower tail latencies and higher bitrates for users than the current state-of-the-art. Assuming that an application is made up of multiple interconnected components modeled as a graph, and bandwidth requirements between components modeled as edge weights, we build heuristics based on the topology of this component graph. We develop two heuristics for component placement, using a modified breadth first traversal of the component graph, and by identifying the most bandwidth intensive paths in the component graph. Since bandwidth fluctuations are commonplace on wireless networks, we also investigate strategies for migrating application components, when the bandwidth requirements of the component are no longer met, in the current deployment configuration. To do so, we rely on a lightweight probing mechanism aimed at maintaining some small, spare capacity on each link in the network, paired with passive measurement of the application's use of the allocated capacity (termed goodput). Our evaluations show that being bandwidth aware on wireless networks can improve application performance.

The contributions of this work are fourfold:

- Network bandwidth variability is widely prevalent in wireless mesh networks. We show that bandwidth becomes an important consideration in service placement at the edge. Instead of using existing tools for measuring point-to-point and link bandwidth, we design and develop a lightweight mechanism to measure bandwidth variations.
- Present-day orchestrators, like k3s, which are intended for edge environments, but are based on the architecture of schedulers meant for cloud/datacenter settings often do not address link capacity variations. We show through evaluations that such link capacity changes affect the performance of a variety of applications. Beyond the obvious bandwidth-intensive applications like video conferencing or camera stream processing, applications that care about end-to-end latency and at the outset do not seem bandwidth-intensive, may still be impacted by changes in link capacity due to the placement of components that may, on aggregate, exchange significant amounts of data. To mitigate the impact of bandwidth variation, we introduce two metrics that use an application component's link usage pattern, and the link's available excess capacity, to determine if component migration may be necessary, in order to meet the application's bandwidth requirement.
- Based on limitations in the state-of-the-art, we develop greedy heuristics to map the components of the application's dataflow graph onto the nodes, prioritizing the network bandwidth needs/availability while maintaining other intracomponent resource capacities as hard constraints. We implement these new proposals in BASS with modules for monitoring network-wide bandwidth, triggers for invoking re-orchestration, and subsequent migration of application components. We also explore tradeoffs for fine-tuning system parameters to produce the best outcomes for an application.
- Using three different example applications that are representative of workloads that will likely be deployed in these environments, we show that it is possible to improve video download quality from 240 Kbps to 480 Kbps for a subset of affected participants in a video conference, improve the end-to-end latency of camera processing by 100 ms just by being cognizant of link capacities, and reduce tail latency of a social network application from 66 seconds to 28 seconds, in the presence of bandwidth variations.

The rest of the paper is organized as follows: §2 motivates the problem of scheduling applications in wireless environments. The

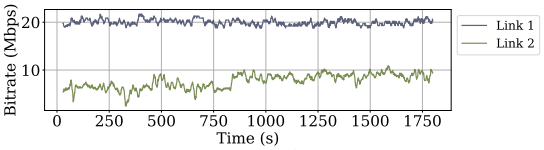


Figure 2: Bandwidth variation (10-second rolling mean) on two different links in the CityLab Testbed. Bandwidth in the first link is relatively stable (mean=19.9 Mbps, std=10% from mean) compared to the second link (mean=7.62 Mbps, std=27% from mean).

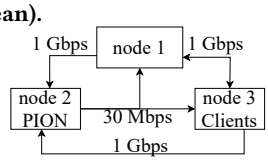


Figure 3: Experiment setup showing change to the outgoing bandwidth at node 2. The interface originally has 1 Gbps capacity. During the experiment we limit outgoing traffic at node 2 to 30 Mbps.

scheduling heuristics and system design are covered in §3 and §4, followed by a brief description of the implementation in §5, and detailed evaluations in §6. §7 briefly summarizes the related work in the space of scheduling in various environments. Finally, we discuss the limitations and future work in §8, and conclude the paper in §9.

2 Motivation

Our work is based on the following observations: (1) wireless links can vary widely in bandwidth over time, and (2) state-of-the-art schedulers do not account for this in deploying and/or orchestrating edge applications.

2.1 Variation in Bandwidth of Wireless Links

In order to understand how link capacity fluctuates in the wild, we collected bandwidth traces from CityLab [56], an outdoor 802.11n deployment in an urban area in Antwerp. We found time-varying behavior of bandwidth availability even during hours where user traffic was minimal, as shown in Fig. 2. This time varying throughput on wireless links has also been reported in a study of wireless 802.11g and 802.11a wireless backbone [28], and on 802.11an links [8].

2.2 Existing Schedulers on Wireless Networks

We first demonstrate that existing schedulers like k3s [48], a Kubernetes [29] based orchestrator suited to resource constrained environments, are agnostic to variations in link bandwidth. We deploy Pion [52], an open source video conferencing platform (an important application at the edge) as a container in a 3-node cluster managed by k3s. The cluster consists of Ubuntu 18.04 VMs on a bridged LAN. Using `tc` [33], we restrict the bandwidth between the clients (on node 3) of the Pion server (on node 2) and the node on which Pion is deployed, as shown in Fig. 3. Ideally, Pion should be moved to the other unaffected node (node 1), but today's schedulers do not perform bandwidth-aware migrations because they are bandwidth oblivious. The end users' quality of experience degrades, even though a different deployment of application components could have maintained the experience. Fig. 4 shows that

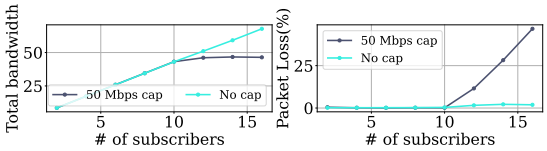


Figure 4: Variation in per client bandwidth and packet loss with link capacity for Pion.

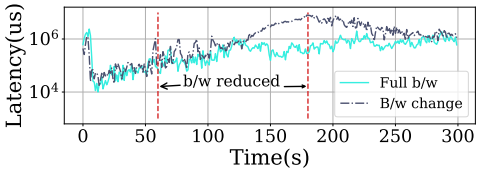


Figure 5: Variation in average end-to-end latency in social network application from DeathStarBench when bandwidth is sufficient (cyan line) and when network conditions change and bandwidth becomes insufficient at 25 Mbps (navy line).

bitrate experienced by users worsens, and packet loss significantly increases when we increase the number of participants beyond 10 on a bottleneck link.

For latency sensitive applications with some bandwidth requirements between the components, the reduction in bandwidth translates to an increase in end-to-end latency, as shown in Fig. 5. We deploy one of the applications from DeathStarBench [20] (social network, another important application at the edge, especially during disaster recovery) on a 3-node cluster. We traffic shape one of the links, reducing the bandwidth to 25 Mbps for 2 minutes during the experiment (same topology as Fig. 3). We then run DeathStarBench's benchmarking tool at 400 requests per second (exponential arrival), and observe the average end-to-end latency at every second. Latency increases by an order of magnitude during the bandwidth restricted period.

These scenarios can be mitigated if we can (i) detect variations in the bandwidth of different links, (ii) evaluate whether these link bandwidths can still meet the communication demand between application components mapped to the nodes at either side of these links, and (iii) migrate (to nodes with links that have the necessary bandwidth) and/or consolidate (co-location of communicating components on a node avoids network altogether) offending application components, so long as the resource constraints on the node are satisfied. Motivated by these observations, we build BASS, to perform bandwidth-aware application deployment and migration.

3 BASS Scheduling Heuristics

In this section, we state the assumptions we made about the environment in which BASS operates and then describe the heuristics used for scheduling and migration.

3.1 Assumptions

We consider a community mesh scenario: wireless network infrastructure connecting compute nodes, and compute nodes that comprise a combination of heterogeneous devices such as Raspberry Pi's, desktops, and server-grade machines. On these networks, the monitoring, maintenance, and expansion of the mesh is typically done by community members/volunteers. For example, NYCMesh has a dedicated set of volunteers who run the effort and maintain

a set of repositories and scripts on GitHub for ease of management [43].

An application deployed on this networked set of nodes may consist of multiple components that can be expressed as a directed acyclic graph (DAG). We discuss how the dependencies can be specified for a component in §5. We assume that a centralized orchestrator on a robust node manages the applications deployed on the mesh. Since wireless mesh networks are expected to be small, with tens of nodes, a centralized orchestrator is sufficient. Further, we assume that the orchestrator either collects metrics or provides hooks to collect resource usage metrics.

3.2 Heuristics for Component Deployment

The BASS scheduler is responsible for (i) mapping application components on the available nodes under constraints of bandwidth, latency, compute, and memory, when an application is first deployed, and (ii) for redeploying components owing to changes in the link capacities that may affect application SLOs. We next discuss these two scheduling aspects.

3.2.1 (Static) Initial Placement

Consider an application whose components can be spread across multiple nodes in the physical infrastructure. Each node may be able to accommodate one or more components. Our heuristics are based on the intuition that components with the highest data rates between them will benefit from being co-located, as long as the cumulative CPU and memory requirements of the components can be accommodated at that node. The idea of co-locating communicating components is common in serverless platforms, where network transfers are avoided since they contribute to a significant increase in end-to-end latency for function chains [2, 37].

To schedule a component, we first rank nodes based on their CPU, memory, and combined capacity across all of the node's links. We pack the node with application components as long as its capacity permits. We use two heuristics to decide which components should be placed on a node. The ordering is determined by traversing the application's component graph, which we assume to be a DAG. A component ordering *a,b,c*, implies that either *a* and *b*, or, *b* and *c*, or all three, should be co-located. An application may consist of components that do some parallel processing, or components that interact in stages. The former class would have high outdegrees at some vertices, and the latter would contain long paths in the application DAG. To handle these two classes of applications, we propose two heuristics: (1) modified breadth-first (BFS) traversal of the DAG and (2) longest-path in the application component DAG. The developer is expected to pick the heuristic that is best suited to the application's data flow.

Breadth-First Heuristic We topologically sort the DAG of application components to identify the root of the BFS tree. We then greedily explore the edges in the order of decreasing bandwidth requirements (i.e., edge weights) from the root of the BFS tree. While inserting components into the queue for traversal, we sort the yet unexplored components by the edge bandwidth to the currently explored component. The pseudocode is listed in Algorithm 1. The input G denotes the graph of application components, with edge weights representing the bandwidth requirement between two components. The *sourceComponent* is determined through a topological sort of the vertices (O(|V| + |E|). The algorithm sorts the queue of yet unexplored vertices (|V|log|V|), for every vertex, so its complexity is O((|V|+|E|) + ($|V|^2$log|V|)) at worst, where |V| denotes the number of vertices and |E| denotes the number of edges in the application graph.

Algorithm 1: Breadth First Heuristic

```
1  Function BreadthFirst(G, sourceComponent):
2      visited := {}, parents := {}, paths := {}, componentOrder
        :=[]
3      queue := [sourceComponent]
4      while !queue.Empty() do
5          currentComp := queue.pop(0)
6          visited[currentComp] = True
7          componentOrder.append(currentComp)
8          queue.sort(by=dep.weight)
9          for dep : currentComp.dependencies do
10             if !visited[dep] then
11                 parents[dep] = currentComp, visited[dep] =
                    True
12                 paths[dep] = paths[currentComp] +
                    G.weight[currentComp][dep]
13                 dep.weight = paths[dep]
14                 queue.append(dep)
15     return componentOrder
```

Longest Path Heuristic In prior work, the longest unweighted chains in terms of number of edges in a topologically sorted DAG of functions were identified as candidates for placement on the same node [2]. We use a weighted version of the same methodology, using bandwidth requirements as edge weights. We identify the most bandwidth intensive paths within an application component graph and try to co-locate the components along those paths on the same node. Once all the vertices reachable from the starting node v, in a topologically sorted graph have been visited, we backtrack from the leaf vertices to find the longest path (i.e., the paths with the largest sum of edge weights) leading up to v. We colocate as many components on the path on the same node as possible. We repeat this process until all the vertices have been traversed. The pseudocode for this heuristic is listed in Algorithm 2. Since we remove the longest chains each time from the graph, at worst, we will repeat the BFS |V| times, once from each vertex in the DAG, therefore the worst case complexity is O(|V|(|V| + |E|)).

Fig. 6 illustrates the difference in component ordering achieved by the two heuristics. The breadth-first heuristic, suitable for applications with large fanouts, will choose components for placement in the order 1, 3, 2, 4, 5, 7, 6 and try to pack the available nodes. The longest-path heuristic, suitable for linear pipelines, will select 1, 2, 4, 5, 7, 3, 6 and then pack the available nodes. The components will then be placed on the nodes already ranked by their resource availability. Intuitively, the longest path heuristic reduces latency by minimizing data transfer between nodes by co-locating components with the most data intensive flow. For applications with

Algorithm 2: Longest Path Heuristic

```
1  Function LongestPath(G):
2      components := topoSort(G)
3      startVertex := components[0], visited := { },
         componentOrder := [ ]
4      while componentOrder.length < G.vertices do
           // Regular breadth first search
5          parents, pathLength := BFS(G, startVertex, visited)
6          maxLen := 0
7          lastVertex := startVertex
8          for dep,length : pathLength do
9              if pathLength[dep]> maxLen then
10                 maxLen = pathLength[dep]
11                 lastVertex = dep
12         visited[startVertex] = true
13         while lastVertex != startVertex do
14             nextVertex = parents[lastVertex]
15             visited[nextVertex] = true
16             componentOrder.Append(nextVertex)
17             lastVertex = nextVertex
18         startVertex := findUnvisitedVertex(G, visited)
19     return componentOrder
```

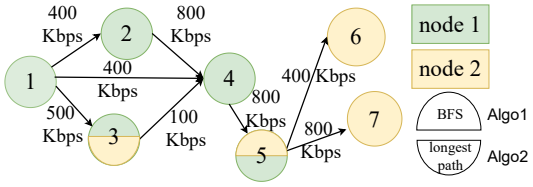


Figure 6: Application component DAG example: the color at the top and bottom half of the component indicates the node chosen for the component by the BFS and longest-path heuristics respectively, assuming each node has 4 cores and each component requires 1 core.

high fan-outs in their DAG, the BFS heuristic tries to co-locate consumers with the producers of that data.

3.2.2 (Dynamic) Component Migration

When the bandwidth on a link fluctuates, it may affect a particular component's SLO. We try to migrate the affected component to a different node in order to mitigate the effects of bandwidth fluctuation. There are two situations in which migration may take place: (1) When a component generates traffic such that the link's capacity is almost used up, and (2) when the link's capacity degrades so much that the component's goodput is affected. In the first scenario, migration is triggered by the component's bandwidth usage, which can be detected via application level monitoring, whereas the second scenario can be handled using a probing mechanism proposed in the next section. affected

We identify component pairs whose bandwidth requirements are not being met, or likely to be not met, based on the bandwidth requirement that the components specify, and the available capacity at the node. We compute the fraction of the allocated bandwidth

Algorithm 3: Component Migration Heuristic

```
1  Function
     FindComponentsToMigrate(G, threshold, headroom):
2      migrationCandidates := [ ]
3      availableBw := computeAvailableBw()
4      for component: components do
5          for dep: component.dependencies do
6              link := availableBw[component.node][dep.node]
7              goodput := dep.bandwidth / dep.required
8              if (goodput > threshold) && (link.bandwidth <
                 dep.bandwidth + headroom) then
9                  migrationCandidates.append(component)
10     sort(migrationCandidates, by=bandwidth)
11     finalCandidates := migrationCandidates
12     for candidate: migrationCandidates do
13         for dep: candidate.dependencies do
14             if migrationCandidates.contains(dep) then
15                 finalCandidates.remove(dep)
16     return migrationCandidates
```

quota the component has used (i.e., the goodput), as an indicator for component migration. We migrate a component when its goodput falls below a system defined threshold in response to the changes in link capacity. Starting with the component with the largest bandwidth requirement, we iterate over all of its dependencies, and delete each dependency from the migration list, if present. We repeat the process until all the components in the migration list have been processed. This way, by migrating only one component of the dependency pair, we avoid cascading effects. The pseudocode for selecting migration candidates is listed in Algorithm 3. We traverse the application DAG to find bandwidth requirement violations, therefore, at worst, we have O(|V| + |E|) operations. While rescheduling a component in need of migration, we first identify candidate nodes, where the component already has dependencies deployed. We re-deploy the component on the node which ranks highest in terms of the number of existing deployed dependencies, and with sufficient CPU, memory, and bandwidth to accommodate the component, minimizing inter-node data transfer when possible.

4 BASS Design

In this section, we present the architecture of our system (Fig. 7), describing each part in depth.

Our system consists of three main parts: ① Scheduler for deploying application components, ② Network monitor for gathering the information about wireless links, and ③ Controller for making decisions about component migration.

4.1 BASS Scheduler

The scheduler maps components to nodes for deployment by applying the heuristics described in §3. It is also responsible for rescheduling components being migrated (using strategies described in §3), by querying the net-monitor (described next) to gauge link capacities and the and link usage on the mesh.

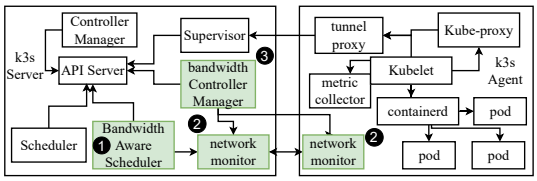


Figure 7: System Architecture. Green boxes are our additions to the k3s ecosystem. The k3s server is a centralized entity that communicates with worker nodes. The network monitor component runs on every node.

4.2 BASS Net-Monitor

The net-monitor is responsible for gathering metrics about bandwidth between nodes, and bandwidth usage between different application components. While Kubernetes monitors network traffic (more for the injection/delivery rates over time at the end-points rather than the time-varying bandwidth of the links), it does not treat network bandwidth as a resource. We add the capability to use network bandwidth as an additional constraint to modify scheduling behavior.

Detecting Bandwidth Changes. Our measurements from the wireless links on the CityLab testbed indicate that link bandwidth can significantly fluctuate over time. To avoid repeatedly probing the link, while still maintaining reliable estimates of available bandwidth, we devise a scheme to minimize the number of times we probe the link, as well as the amount of data consumed during the probe. We perform two types of probing: (1) max capacity probing, where we flood the link to know the full link capacity; and (2) headroom probing, where we only probe the link to understand whether a certain amount of spare capacity is available.

Max. Capacity Probing. When the system starts up, we perform a round of probing in the net-monitor to understand the capacity of each link by flooding each link with packets. We then cache these values in the net-monitor to respond to network capacity queries from the scheduler or the bandwidth controller, until a new capacity probe request is made by the bandwidth controller.

Headroom Probing. Headroom is the spare capacity the system is expected to maintain on every link, to allow for sudden bursts and variance in the link capacity to make applications more resilient to them. We specify headroom as a percentage of the link capacity. During monitoring, we only check that there is enough headroom on the links that the component is expected to use. This way, we avoid flooding the link in order to determine its capacity. During normal probing cycles, we only perform headroom probing. When a change is detected in the available headroom, the controller checks the system for violation of bandwidth requirements and determines if components need to be rescheduled.

An example of component migration is shown in Fig. 8. We consider a component pair that requires at least 8 Mbps bandwidth, initially deployed on nodes 3 and 4 respectively (see Fig. 15(a) for topology). The bandwidth of node3-node4 link is set to 25 Mbps. We set the threshold for migration at 50% goodput, and headroom to 4 Mbps (~20% of link capacity) on each link. We set a conservative headroom probing frequency at 30 seconds (0.6% network overhead), however, this frequency can be adjusted to suit application requirements. The component pair's goodput changes (teal hatched line) in sync with node3-node4 link capacity. At around t=540, we reduce the link capacity (dotted blue line). The controller, noticing a drop in the headroom capacity (vertical dashed red line), triggers a full probe at t=634. The probe eventually completes at t=750. The controller now is aware that the bandwidth has dropped on the link (solid blue line with dots). At t=840, the headroom (4 Mbps) requirement on node3-node4 link is no longer met, and goodput falls below the 50% threshold, and the total capacity on the link is 7 Mbps, triggering a migration from node4 to node1 at t=870. After the migration, we see that the goodput immediately increases. We reduce the node1-node3 capacity at t=1119 (dotted orange line), and increase node3-node4 capacity back to 25 Mbps (solid orange line with dots). At t=1240, the headroom for node1-node3 falls below the 4 Mbps threshold, and the goodput falls below 50%, triggering a migration from node1 back to node4.

Network Resource Monitoring. We estimate path bandwidth availability by first measuring the link bandwidth between 1-hop neighbors and then applying `traceroute` (to get the path between the pair of nodes). Then we determine the capacity of the node pair to be the bottleneck link along the path. We deploy a daemon on each node in our cluster to periodically measure link bandwidth. We gather two sets of metrics: TX/RX bytes between (1) each pair of nodes; and (2) between application components. The first metric is necessary to estimate if the current deployment will satisfy application SLOs. When an application is scheduled for the first time, we only look at the bandwidth requests by the application while making scheduling decisions. When the bandwidth availability in the system changes, we use both metrics to make reconfiguration decisions.

4.3 BASS Controller

The controller is responsible for deciding if applications have the desired amounts of resources they requested. When there are bandwidth fluctuations, an application component may have to be relocated to a different node to avoid bandwidth requirements not being satisfied. The controller fetches each component's resource requirements from the component configurations, listens to the net-monitor for network capacity/usage updates, and in the event of bandwidth change leading to a component not getting sufficient bandwidth, instructs the scheduler to reschedule the component. The scheduler also has access to net-monitor, and makes the final decision about where the offending component should be placed. A migration is not free, however. The application's performance may degrade due to temporary service unavailability, while the component is being moved. Depending on how long the migration takes, the disruption to availability may not be amortized. To avoid reacting to transient changes in bandwidth availability, we ensure that there is a "cooldown" period between the detection of low bandwidth and the migration trigger. The controller uses two parameters to make migration decisions: (1) available headroom capacity on the link, which is determined using headroom probing, described in §4.2, and (2) goodput threshold (described in §3.2.2). If the goodput falls below the set threshold on a constrained link, or if the component uses the link to the extent that the headroom on the link shrinks even without capacity change on the link, migration may be triggered by the controller.

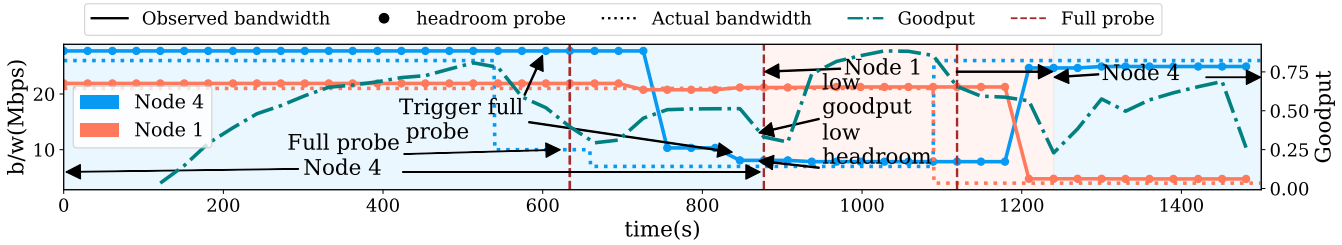


Figure 8: Illustration of migration on bandwidth change. Orange and blue lines indicate the actual/observed bandwidth on nodes 1 and 4, respectively.

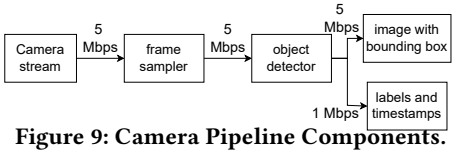


Figure 9: Camera Pipeline Components.

5 Implementation

BASS is built a set of extensions to k3s, a lightweight orchestrator that conforms to the Kubernetes API[1].

Specifying Bandwidth Requirements. BASS takes an application's component DAG, with edge weights specifying the maximum bandwidth requirements (gathered through independent offline profiling) as input. Bandwidth requirements between components do not change for the duration of the application's deployment and are added to the metadata section of the application's deployment file.

Monitoring Network Usage. We use iPerf3 to gather bandwidths between nodes and *traceroute* to identify paths between nodes, and a Berkeley Packet Filter [16] to measure data exchanged between nodes. The network statistics are exposed through a gRPC endpoint on every node in the cluster. To measure bandwidth usage between pods, we enable Istio sidecar injection [27, 34] for the application. The metrics are logged into Prometheus [55], from which they are queried via the HTTP API. The monitoring services consist of about 500 lines of code (LoC), in a combination of C, Go, and Python.

Scheduling all components at once. Kubernetes deploys one pod at a time. Since we want to consider bandwidth requirements *between* pods, our scheduler extension cannot use this strategy. We instead wait for all of the pods in the application to be received by the custom scheduler and build the dependency graph before applying scheduling heuristics. The scheduler is implemented in Go in about 2k LoC.

Bandwidth Controller. The controller is written as a separate component in Go (1k LoC), that listens to the monitoring services and gets inter-pod communication metrics from Prometheus. It then compares the traffic against the available bandwidth and makes rescheduling decisions.

6 Evaluations

In this section, we first describe our microbenchmark experiments, aimed at understanding: (1) the effectiveness of our scheduling algorithm in a scenario with no bandwidth variations, and (2) the usefulness of application component migration after initial scheduling, under controlled bandwidth variation. We perform microbenchmark experiments on a 3 to 4 node cluster on CloudLab [15]. For the evaluations on real world bandwidth variations, we emulate the layout of a subset of the nodes, and the link capacities in CityLab wireless testbed on CloudLab.

6.1 Applications

We first briefly describe the open source applications used in the evaluations. Application components are containerized and the images are pre-downloaded on nodes as-is. The applications have different characteristics with regard to being network or CPU bound.

Video Conferencing. This application has a single component server, which all participants (clients) connect to. The server collects video feeds from participants and forwards those feeds to other participants in the conference, thereby requiring significant outgoing bandwidth at the node where the component is placed. The application is network bound, and we use the average bitrate per client as the metric for evaluating application performance. We use Pion [52], an open source SFU (selective forwarding unit or the conference server) for video conferencing based on WebRTC.

Camera Processing. The camera processing application emulates a camera pipeline (Fig. 9) for a given recorded mp4 video, from which an object detector is used to annotate frames in real time. We use `ffmpeg` [18] to publish the video to an RTP stream server. The streamed video is read by a sampling application that picks frames that are dissimilar and feeds them into a YOLO object detector [19]. The object detector publishes two streams: one consisting of annotated images and the other with just the text labels. In addition to being bandwidth intensive, the application is CPU bound in the object detector stage of the pipeline, and network bound at the output of the camera stream and frame sampler, and input to the image listener.

Social Network. This Social Network application is a collection of 27 microservices, consisting of front end servers, backend services, caches, and databases published as a part of the DeathStarBench microservice benchmark [20]. The application requires low latency communication between microservices, and predominantly performs RPC calls, alongside some caching and database accesses. While the application may not necessarily seem bandwidth intensive at the level of a user request, there are complex patterns of interaction between the component microservices which can induce bandwidth dependence, as we will show. As a result, the co-location (or lack thereof) of certain pairs of services, and sudden drops in the bandwidth of these non-colocated services, can influence the end-to-end latency of the application.

[1]Source code: https://github.com/Manasvini/mesh-bw-scheduler

6.2 Microbenchmarks

We evaluate our scheduler against the default k3s scheduler for scheduling under static bandwidth conditions and then compare our heuristic scheduler with and without migration support under time-varying bandwidth conditions.

6.2.1 Setup

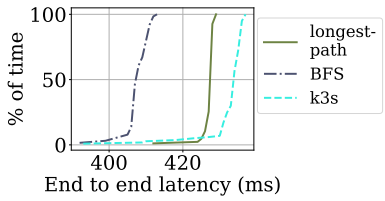
We use a cluster of CloudLab machines (c6525-25g or d710 depending on availability), connected over a LAN. One node runs the k3s control plane and BASS's extensions to the k3s's ecosystem (left-half of the of the system architecture Fig. 7). Since for the first experiment, we are not evaluating migrations, we disabled BASS's bandwidth controller.

6.2.2 (Static) Initial Component Placement

In this set of experiments, we examine the importance of being cognizant of bandwidth requirements between different components of an application in deciding the initial placement of components across nodes. We study the performance of the camera processing application and social network, as a function of their initial deployments.

Camera Processing. We run the camera processing pipeline for 30 minutes on three c6525-25g machines (AMD EPYC 7302P processor, 16 cores, 128 GB RAM) and plot the end to end latency of the pipeline. We use a 12-second video clip of a traffic intersection and play it on loop for 5 minutes for evaluating the end to end latency of the pipeline. We observe that the BFS scheduler (mean latency 410 ms) performs better than the longest-path scheduler (428 ms) and default k3s (433 ms) schedulers (Fig. 10(a)). The greedy breadth first/maximum bandwidth heuristic works well in this scenario since much of the data transfer happens in the first two stages of the pipeline (vis a vis the longest-path heuristic heuristic which prioritizes longer chains). As shown in Fig. 10(b), the BFS scheduler places the camera stream and frame sampler on the same node, and the remaining components on the other node. On the other hand, the longest-path scheduler places the object listener alone on one different node and the remaining components on the other node. Note that the longest chain would normally include the object detector, but because of the detector's high CPU demand, it doesn't fit on the same node as the sampler and the camera stream.

Social Network. We deploy the social network application on four d710 (Intel Xeon E5530 with 4 cores, 2 threads, 12 GB RAM) machines and study the impact of component placement on the end-to-end latency. We consider a fixed request distribution of 100-300 requests/second (RPS) and report the mean and standard deviation of p99 latency across five trials. When there are no bandwidth restrictions (Fig. 11), the tail latency is comparable between the longest-path and k3s default scheduling policy. On the other hand, when we restrict the bandwidth to 25 Mbps on one node, we observe that the tail latency difference between the default k3s scheduling and the bandwidth aware longest-path heuristic is about two orders of magnitude higher at higher request rates (200 and 300). Components with higher data rate requirements were often co-located on the same node when the longest-path scheduler was used, in comparison to the k3s scheduler.



(a)

Component	BFS	longest-path	k3s
camera stream	node 1	node 1	node 1
frame sampler	node 1	node 1	node 2
object detector	node 2	node 2	node 1
image listener	node 2	node 1	node 2
object listenet	node 2	node 1	node 2

(b)

Figure 10: (a) End to end latency for camera processing application under different scheduling policies on 3-node cluster with no bandwidth limits. (b) Camera pipeline component placement by each scheduler.

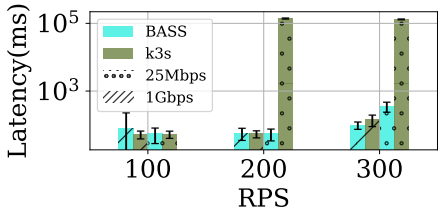


Figure 11: Comparison of p99 latency of heuristic and default schedulers under no bandwidth constraint in the network and bandwidth on one node restricted to 25Mbps at different request rates.

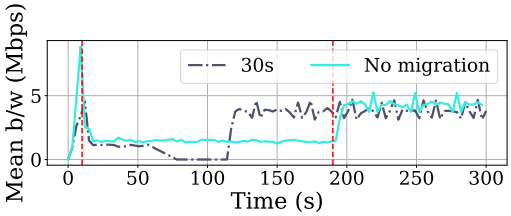


Figure 12: Effect of bandwidth variation in video conferencing application under different bandwidth querying intervals. Red vertical lines indicate when bandwidth restrictions are imposed and lifted in the experiment.

From these experiments, we see that bandwidth aware placement can help improve the end-to-end latency of two types of applications. Even without link constraints, end-to-end latency decreases when we use bandwidth aware placement due to the significant communication bandwidth required between the components. In the social network application, the necessity for network bandwidth aware placement becomes apparent in the presence of constrained links. The application exchanges messages between its multiple components frequently. The communication overhead between nodes grows when there is a link capacity constraint, thereby necessitating bandwidth aware component placement.

6.2.3 (Dynamic) Component Migration

We next show the need for dynamic migration with bandwidth variations, and the effectiveness of our mechanisms. In this experiment, we initially do not set any bandwidth restriction. We deploy the application using our longest-path scheduler. Ten seconds into the experiment, we introduce bandwidth limits on two of the nodes in the cluster. The restriction persists for 3 minutes and then is lifted. On every node, BASS's network monitor makes headroom

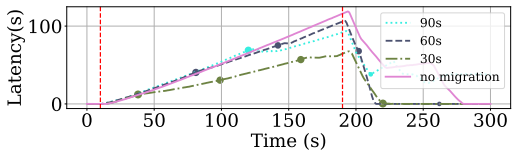


Figure 13: Effect of bandwidth variation on end-to-end-latency, under 25 Mbps throttling. The red vertical lines denote the start and end of the bandwidth restriction. The dots on each line indicate component migration events. The violet line shows the default behavior without any migrations.

probes to its neighbors every 30 seconds. We use the same setup as in Fig. 3, with three c6525-25g machines.

Video Conferencing. The Pion server is initially deployed on node 2. We set the number of participants in the conference to 9. One participant shares their video, and others only receive one stream. We then measure the bandwidth experienced by each participant receiving the video. When the application's bandwidth requirements are evaluated every 30 seconds (cyan line in Fig. 12), the bandwidth violation is soon discovered and the application is migrated from node2 to node1, where the link bandwidth satisfies the application's requirements, whereas when there is no migration strategy in place, the Pion server remains deployed at node2 (initial location). With BASS, for the duration of the migration, however, lasting 30 seconds, the participants experience temporary disruption to the video stream. This 30 second period is the time taken to reschedule the application and reestablish the RTC connections. Once the connection is reestablished, the clients experience the same bitrate as before the bandwidth limitation. When no migration is performed, the clients experience poor bitrate for the entire 3-minute duration of the bandwidth disruption.

Iteration	Components exceeding link utilization quota	Components migrated
1	6	2
2	1	1
3	1	1

Table 1: Social Network component migration across successive iterations of the scheduler for a 30 second bandwidth querying interval when bandwidth is reduced to 25 Mbps at node 3.

Social Network. We consider a fixed request distribution of 400 RPS on 3 625-25g nodes. Here, to study the effect of component migration to nodes with better link capacity, we enable component scheduling on all 3 nodes. We throttle the bandwidth on the outgoing interface of nodes 2 and 3 during the experiment. We then report the average latency at every second for the duration of the experiment which runs for 5 minutes. We find that not migrating components results in up to 50% higher latency than when components are migrated (violet line in Fig. 13). We consider various monitoring intervals for migration (30, 60, and 90 seconds), and for this bandwidth variation scenario, evaluating the placement of components every 30 seconds has the best impact on tail latency reduction. We inspect the set of components that were relocated to node 1, and we find that, though several components were experiencing insufficient availability of bandwidth, only a few were migrated (Table 1). For example, in the first iteration, two of the components that were identified for migration were communicating with each other, (and have some of the highest data rates of all components in the application), but only one member of the component pair was actually migrated. As described in §3, we do not migrate both a component and its dependency in one shot.

6.3 Evaluations on Emulated Mesh Network

In this section, we describe the performance of BASS on an emulated mesh testbed, with bandwidth variations collected from a citywide deployment of wireless routers previously described in §2.1. We set up a 5-node subset of the CityLab topology as shown in Fig. 15(a) on CloudLab VMs. We run all control plane functions on one of the VMs, while the remaining 4 run applications. Since we expect resources on a mesh to be heterogeneous, the VMs are configured with 8GB RAM and either 12 (3 VMs) or 8 (2 VMs) cores. All the VMs are located on a C8220 machine (Intel E5-2660v2 20 cores, 256 GB RAM, 2 threads per core, 2 sockets) and run Ubuntu 22.04. Since we work with one bandwidth trace and a fixed input (request arrival is constant for social network, and the same video is used as input for the camera pipeline and video conferencing), we consider a single trial lasting around 20 minutes for each experiment. We evaluate BASS's efficacy towards answering the following questions:

(1) How effective is BASS's component placement strategy?
(2) Can BASS leverage migrations to improve application performance?
(3) How to select the right thresholds for migration?

6.3.1 (Static) Initial Placement

We investigate the end to end latency of the camera processing application under time-varying bandwidth conditions with different schedulers. We deploy the camera processing pipeline on a set of 4 worker nodes. We use 4 cores for the sampler, and 8 for the detector components.

The results in Table 2 show the utility of BASS compared to native k3s. We compare the end-to-end latency achieved by the schedulers in two scenarios: no bandwidth variation, where the bandwidth on the links is set to the maximum value observed in the CityLab trace (i.e., the baseline), and the trace with time-varying bandwidth on each link. There is only a small difference (around 2 ms for BFS) in the end-to-end latency for both the BASS strategies for application component placement (BFS and longest-path), regardless of bandwidth variations. On the other hand, it can be seen that the latency inflation in the presence of bandwidth variation is close to 20% when k3s default scheduler is used. We did not any observe component migrations for this workload because the headroom thresholds were not violated even when there were bandwidth variations. Similar to the microbenchmark evaluations, the BFS heuristic does better with this workload than the longest-path heuristic.

Scenario	BFS (ms)	longest-path (ms)	k3s (ms)
No bandwidth variation	540	551	577
With bandwidth variation	538	552	692

Table 2: Median end-to-end latency on CityLab emulated mesh, observed with, and without bandwidth variation using different schedulers

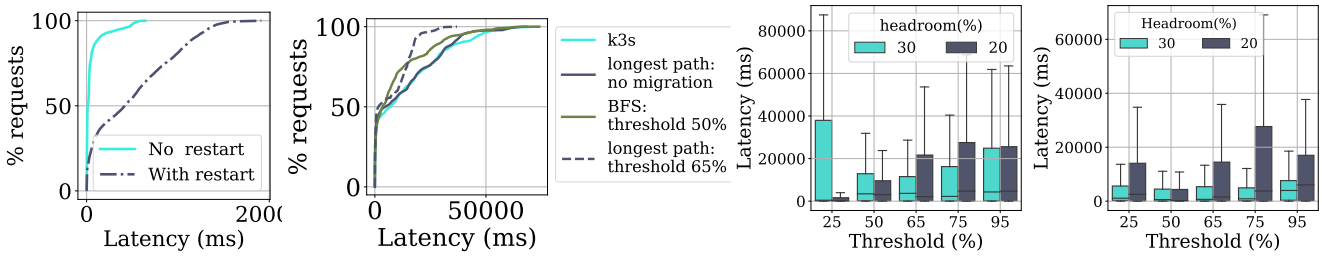


Figure 14: (a): CDF of end-to-end latency when restarting a component (b): CDF showing the latency comparison of different schedulers for the social network application. We picked the threshold and headroom combination for each algorithm that yielded the lowest upper quartile latency from the (c) and (d) plots. (c) and (d): End-to-end latency under different capacity headroom and bandwidth use thresholds for BFS and longest-path, respectively.

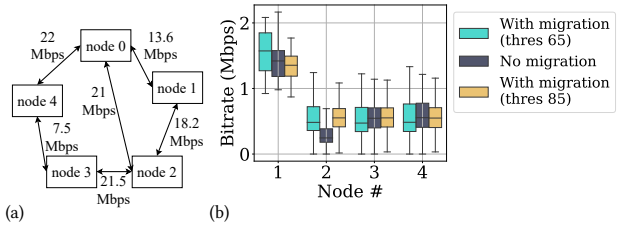


Figure 15: (a) 5-Node subset of the Citylab topology. The links on the testbed are wireless. The measured bandwidth values are averaged over half an hour. Links are bidirectional, and have similar bandwidth in both directions. (b) Effect of bandwidth variation on average bitrate for participants at each node in the cluster.

Take-away. The camera processing application, though bandwidth intensive, does not experience significant inflation in end to end latency under the bandwidth variations in the trace because the initial placement itself suffices in meeting the bandwidth needs of the applications, and the variations are not severe enough to require migration.

6.3.2 (Dynamic) Component Migration

In this section, we explore how BASS is able to leverage migrations to improve performance of the video conferencing and social network applications.

Video Conferencing. For this experiment, we deploy the Pion video conference server on one of the 4 worker nodes. We investigate download bitrates experienced by clients in a video conference under variable bandwidth conditions, and if they can be improved by migrating the Pion server. To do so, we first quantify the overhead associated with migration: it takes around 20 seconds for the server to be restarted, and the WebRTC connections to be reestablished. At each worker node (nodes 1 through 4 in Fig. 15(a)), we create 3 clients that are in a video conference, which share their feed and subscribe to all the other participants' video feed. We replay the bandwidth trace collected from the CityLab testbed, and start a video conference that lasts 10 minutes, with 3 participants at each node. We expect the bandwidth variations will cause the application's bandwidth requirements to be violated, and necessitate migration. In Fig. 15(b), we plot the bitrate experienced by the clients on average, when (1) no migration is performed (2) migration is performed at two thresholds of link utilization (65% and 85%). We don't consider lower thresholds for migration because of the significant overhead in reestablishing RTC connections. Both strategies result in some improvement in the median bitrate of the participants at node 1 from 1.4 (without migration) to 1.6 Mbps (with migration, 65% threshold), and for the participants at node 2 from 240 Kbps (without migration) to 480 Kbps (with migration, 65% threshold); there is no improvement for the participants in the other two nodes.

Take-away: Migrating the conference server could help in mitigating periods of low download bitrate for participants in the video conference. The cost of reestablishing connections (20 seconds) amortizes over time as long as the conference lasts at least tens of minutes.

Social Network. For this application, we investigate the impact of migrating components on the end-to-end latency. We compare the performance of BASS schedulers against k3s under a time varying bandwidth trace. We show that despite migration costs, there may be overall improvements in latency. For all the experiments, we run the workloads at 50 RPS, comprising both reads and writes.

Migration entails that a component be restarted, and hence we quantify the overhead of restarting an application component while running the workload at a fixed request rate (50/second). In Fig. 14(a), the end-to-end latency increases on average, from 552 ms to 4.9 seconds. Hence, we need to consider the overhead of temporary increase in end-to-end latency as a consequence of migration.

We next investigate the performance of BASS scheduling heuristics under a variable bandwidth trace, where migration of components may become necessary. In Fig. 14(b), we plot the end-to-end latency of the workload under the longest-path and BFS heuristics, along with the k3s scheduler, and the longest-path heuristic without migrations. We observe that without migration enabled, the longest-path heuristic does slightly better than k3s, but the real gains in latency reduction in the face of bandwidth variation come from right-timed migrations. We find that the longest-path scheduling performs slightly better than BFS scheduling for this workload. The criteria for collocating components in the two algorithms is slightly different. The longest-path scheduling favors placing long chains of dependent components on the same compute node, which works well for the common frontend-service-cache-database traffic pattern that this application uses multiple times. In contrast, BFS greedily places components with the highest bandwidth requirements in a breadth-first manner. The p99 latency longest-path

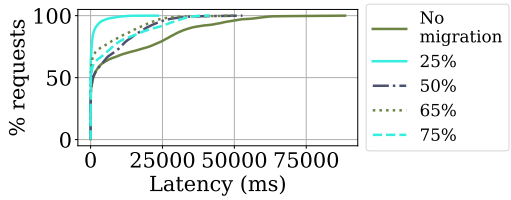


Figure 16: Performance of longest-path scheduler on a workload with exponential request arrival.

heuristic with migration support is 28 seconds, in comparison to the 66 seconds observed in the default k3s scheduler.

Take-away: Despite the inflation in end to end latency that restarting a component (as a consequence of a migration decision) can create, migration of a component that is experiencing a shortfall in available bandwidth can produce an overall reduction in end-to-end latency for the application.

6.3.3 Finding the right migration threshold

Two parameters govern the behavior of the bandwidth controller component in BASS—link utilization threshold for migration (i.e., migration threshold), and the headroom capacity that the system needs to maintain at all times. The first parameter is an indicator of the application's link usage, and the second is an indicator of spare capacity available on the link. We also investigate how these parameters are dependent on the application's traffic patterns. This knowledge may be useful for the developer in tuning the system to suit their application's needs and traffic patterns.

We investigate the relationship between these parameters and end-to-end latency for the social network application. For a request rate of 50 RPS, we evaluate different thresholds for migration, based on the component's link utilization: 25, 50, 65, 75 and 95%, as illustrated in Fig. 14(c) and (d) for the BFS and longest path scheduling respectively. When we set the migration threshold too low (e.g., 25%), we end up performing migrations when they could have been easily avoided, due to the tradeoff between migration cost of component (eviction and subsequent scheduling and re-deployment) and the transience of the link capacity change. At the other extreme, if the threshold is too high (e.g., 75%), we may wait too long before performing a migration, leading to prolonged increases in end-to-end latency. The 50% or 65% threshold is able to balance between the two extremes.

Next, we look at the effect of request distribution on different migration thresholds by using an exponential distribution for request arrival (commonly used to model arrival rates). We set the mean arrival rate to 50 RPS and set the headroom capacity to 20%. We used the longest-path scheduler with different link utilization thresholds for component migration. We find that lower migration thresholds in general perform better for this scenario (Fig. 16). Unlike the fixed request distribution, where the traffic rate on the link can be expected to not change by orders of magnitude, the requests arrive in bursts in an exponential distribution. Since many components have low request rates most of the time, early migration does not inflate latency to the same degree in the exponential workload as it did in the fixed workload.

Take-away: Choosing the right setting for headroom and link utilization thresholds play a role in reducing end-to-end latency, and in turn are influenced by the traffic patterns exhibited by the application.

6.3.4 System Overheads

In this section we discuss the overheads that are incurred by BASS, specifically, we look at per-component scheduling latency, overheads from performing graph traversals, and overheads of network monitoring.

We compare the per-component scheduling latency of k3s versus BASS (longest-path scheduler) on one C8220 machine on CloudLab running 5 VMs (same as §6.3). We find that the per-component scheduling latency on average across the two systems is comparable (see Table 3). However, BASS incurs a one-time overhead of processing the application component DAG before scheduling. We note that while this overhead is in the order of tens of milliseconds, the scheduler runs only when applications are being deployed, or components are being migrated. Bandwidth fluctuations needing a component migration happen in the order of minutes (as observed from the CityLab traces [56]). Thus the scheduling overhead accounts for less than 0.01% of the application run time.

In comparison to k3s, we incur the network cost of periodically probing wireless links. Our headroom probing methodology ensures that the overhead is bounded. We probe the links every 30 seconds for 1 second, for 10% of the link's capacity, accounting for 0.3% of the network traffic. We noticed that on the CityLab bandwidth trace, a full probe was required only three times in the 20 minute duration of the trace when running the socialnetwork workload. Full probes also last only 1 second, and therefore the overhead from network probing is negligible (0.25% of the link traffic).

Application	Avg. Latency (ms)		Std. Dev (ms)	
	k3s	BASS	k3s	BASS
Social Network	1.27	1.5	0.08	0.1
Video Conference	1.28	1.28	0.08	0.07
Camera	1.27	1.4	0.08	0.56

Table 3: Per component scheduling latency comparison

Application	comps.	Avg. processing time(ms)	Std. dev(ms)
Social Network	27	63.86	10.77
Video Conf.	1	26.31	0.99
Camera	5	30.59	3.37

Table 4: DAG processing times for all applications

7 Related Work

In this section, we briefly review existing literature on applications of wireless mesh networks, and resource scheduling on clusters of various kinds.

Services on Wireless Mesh Networks. Existing wireless mesh networks like Guifi [24] support various services like video conferencing, messaging and web servers. Prior work also explored architectures for bandwidth intensive operations like live streaming on wireless mesh networks [36]. Roofnet demonstrates that even though ad-hoc networks are unplanned in their positioning of new nodes, performance, in terms of throughput is still acceptable [5]. They also investigate various protocols for efficient decentralized routing on these ad-hoc networks. Wireless mesh networks have also been explored as a feasible option for providing communication facilities during disaster response scenarios in difficult terrain [3] and for emergency management [10].

Dynamism in Wireless Networks. Last mile connectivity is known to be vary in both space and time for wireless links, including cellular [47, 53] as well as unlicenced links [8, 28]. In the Ninux community mesh networks, a subset of nodes is shown to be important in routing, and their failures could significantly impact the network [35].

General purpose Orchestrators. Kubernetes [29, 48] and its lightweight variants are general purpose orchestrators that primarily consider CPU and memory as the main resources for scheduling. In addition, they support extensions that make use of tools like `tc` to limit the bandwidth usage by a component [12, 58] to reduce bandwidth resource contention. However, such general purpose schedulers lack the capability to monitor bandwidth variability in wireless mesh networks and mitigate application SLO violations that stem from temporal bandwidth loss.

Orchestrators at the Cloud. Systems which improve orchestration do not treat bandwidth as a first-class citizen when scheduling [14, 21, 23]. Graphene [23] considers critical paths for task pipelines but does not consider the communication requirements between tasks. Similalry, Medea [21] models placement of long running jobs as an ILP but does not account for the inter-node bandwidth as a constraint in the formulation. Prior proposals [6, 9, 22, 38, 46] consider network bandwidth as a resource during orchestration only to mitigate SLO violations due to resource contention. Scheduling decisions in these past works target datacenter environments that assume stable link capacities and the ability to run solvers [6, 9, 39] which are infeasible for resource constrained wireless mesh environments. For context, A Philadelphia-based mesh network [45] has about 30 nodes, requiring 900 constraints for path bandwidth alone. Finally, scheduling of function pipelines modeled as DAGs have been considered in several Function-As-A-Service (FaaS) systems [2, 32]. These systems do consider the data transfer between components, but they are built to run on stable networks and don't have to consider migrating components when link capacities change. In contrast, BASS incorporates bandwidth requirements for initial application deployment and continuously monitors for bandwidth availability to proactively migrate application components to account for variability in bandwidth prevalent in wireless mesh networks.

Orchestrators at the Edge. There are research efforts aimed at edge environments, taking various resource constraints into account (in addition to CPU and memory), such as latency [40], bandwidth [31, 57, 59], and energy [42, 60]. Prior efforts have also looked at orchestrating geodistributed compute resources for latency sensitive applications across the edge-cloud continuum [11, 50] do not model the cost of migrating services when underlying network capacity changes, and assume reliable connectivity to the wide area network. Preliminary efforts aimed at measuring the performance of orchestration platforms for low power devices [17] and optimizing power consumption for specific applications, such as camera networks [1] have been proposed which are complementary to BASS. Prior work has incorporated bandwidth awareness [41] for specific domains (e.g. DNNs), but require modification to the application code, whereas BASS is general and works across applications and does not require changes to application code.

8 Discussion and Future Work

In this work, we considered the impact of bandwidth variations on application performance and proposed a few ideas to mitigate the impact. We considered a set of compute nodes connected wirelessly and created a system to monitor changes in bandwidth in order to inform scheduling. In the current implementation of the system, bandwidth requirements are independently determined through offline profiling. Determining the bandwidth requirements of every component pair in an application is cumbersome work for the developer. As a part of future work, we plan to introduce automated online profiling for gathering bandwidth requirements for scheduling components once an application has been deployed. In our evaluations, under a time varying bandwidth trace, we investigated combinations of headroom capacity and link utilization thresholds that yield the best performance for two of our workloads. The two parameters are configurable and the developer can specify the threshold based on what works best for their use case. We leave the exploration of automated migration parameter tuning to future work.

In the current iteration of the work, we assume that components are either stateless or that state can be discarded without impacting correctness (e.g., caches), and that container images are readily available on every node (this is at most a one time cost). In order to perform stateful migration, we will have to save the state of the container, or some part of it, and possibly transport it across the network to a different node and may incur additional data transfer cost. Medes [51] performs memory deduplication of container states using CRIU [13], which is a tool for saving container states in user space. Similar techniques are being proposed in Kubernetes for checkpointing [30], though this is mainly intended for forensic analysis, and not for live migration.

We proposed two different heuristics for component placement on a wireless mesh. The two heuristics are essentially based on the fan out or depth of an application's data flow graph. The applications we investigated favored one or the other, but it is possible that a subgraph of the application may have high fanout, and another part could be a deeper pipeline. A potential avenue of future research is combining the two heuristics depending on the application specifics.

9 Conclusion

In this work, we explored the effects of treating network bandwidth as an important property in the decision for application component placement. We developed heuristics for this placement and investigate the impact of component migration on application SLOs. We expect our modifications to k3s to benefit wireless mesh networks, making them resilient to bandwidth fluctuations.

Acknowledgments

We would like to thank our anonymous reviewers from both submission cycles and members of the Embedded Pervasive Lab at Georgia Tech for their thoughtful feedback that helped to improve this paper. This work was funded in part by NSF grants (CCF-1909004, SCC-2125354, CNS-2008368, CCF-2211018), and a gift from Microsoft Corp.

References

[1] Kevin Abas, Katia Obraczka, and Leland Miller. Solar-powered, wireless smart camera network: An iot solution for outdoor video monitoring. Computer Communications, 118:217–233, 2018.

[2] Mania Abdi, Samuel Ginzburg, Xiayue Charles Lin, Jose Faleiro, Gohar Irfan Chaudhry, Inigo Goiri, Ricardo Bianchini, Daniel S Berger, and Rodrigo Fonseca. Palette load balancing: Locality hints for serverless functions. In Proceedings of the Eighteenth European Conference on Computer Systems, pages 365–380, 2023.

[3] Usman Ashraf, Amir Khwaja, Junaid Qadir, Stefano Avallone, and Chau Yuen. Wimesh: leveraging mesh networking for disaster communication in resource-constrained settings. Wireless Networks, 27:2785–2812, 2021.

[4] Carly Berwick. Yesterday's Internet Isn't Good Enough for Tomorrow's Cities. https://nextcity.org/features/internet-connection-mesh-networks-resilience, 2016. [Online; accessed 18-Feb-2023].

[5] John Bicket, Daniel Aguayo, Sanjit Biswas, and Robert Morris. Architecture and evaluation of an unplanned 802.11b mesh network. In Proceedings of the 11th Annual International Conference on Mobile Computing and Networking, MobiCom '05, page 31–42, New York, NY, USA, 2005. Association for Computing Machinery.

[6] Ofer Biran, Antonio Corradi, Mario Fanelli, Luca Foschini, Alexander Nus, Danny Raz, and Ezra Silvera. A stable network-aware vm placement for cloud systems. In 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012), pages 498–506, 2012.

[7] Difei Cao, Jinsun Yoo, Zhuangdi Xu, Enrique Saurez, Harshit Gupta, Tushar Krishna, and Umakishore Ramachandran. Microedge: A multi-tenant edge cluster system architecture for scalable camera processing. In Proceedings of the 23rd ACM/IFIP International Middleware Conference, Middleware '22, page 322–334, New York, NY, USA, 2022. Association for Computing Machinery.

[8] Llorenç Cerdà-Alabern, Axel Neumann, and Pau Escrich. Experimental evaluation of a wireless community mesh network. In Proceedings of the 16th ACM International Conference on Modeling, Analysis amp; Simulation of Wireless and Mobile Systems, MSWiM '13, page 23–30, New York, NY, USA, 2013. Association for Computing Machinery.

[9] Jianhai Chen, Kevin Chiew, Deshi Ye, Liangwei Zhu, and Wenzhi Chen. Aaga: Affinity-aware grouping for allocation of virtual machines. In 2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA), pages 235–242, 2013.

[10] Francesco Chiti, Romano Fantacci, Leonardo Maccari, Dania Marabissi, and Daniele Tarchi. A broadband wireless communications system for emergency management. IEEE Wireless Communications, 15(3):8–14, 2008.

[11] Ka-Ho Chow, Umesh Deshpande, Veera Deenadhayalan, Sangeetha Seshadri, and Ling Liu. Atlas: Hybrid cloud migration advisor for interactive microservices, 2023.

[12] CNI. bandwidth-plugin, 2021. https://www.cni.dev/plugins/current/meta/bandwidth/.

[13] CRIU. CRIU, 2023. https://criu.org/Main_Page.

[14] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Kairos: Preemptive data center scheduling without runtime estimates. In Proceedings of the ACM Symposium on Cloud Computing, SoCC '18, page 135–148, New York, NY, USA, 2018. Association for Computing Machinery.

[15] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In Proceedings of the USENIX Annual Technical Conference (ATC), pages 1–14, July 2019.

[16] eBPF. bpf-helpers, 2022. https://man7.org/linux/man-pages/man7/bpf-helpers.7.html.

[17] Rafael Fayos-Jordan, Santiago Felici-Castell, Jaume Segura-Garcia, Jesus Lopez-Ballester, and Maximo Cobos. Performance comparison of container orchestration platforms with low cost devices in the fog, assisting internet of things applications. Journal of Network and Computer Applications, 169:102788, 2020.

[18] FFmpeg. A complete, cross-platform solution to record, convert and stream audio and video., 2022. https://ffmpeg.org/.

[19] Frank Schmitz. Object detection using deep learning with Yolo, OpenCV and Python via Real Time Streaming Protoco, 2022. https://github.com/foschmitz/yolo-python-rtsp.

[20] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 3–18, 2019.

[21] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. Medea: Scheduling of long running applications in shared production clusters. In Proceedings of the Thirteenth EuroSys Conference, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[22] IC Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. Usenix, 2016.

[23] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. GRAPHENE: Packing and Dependency-Aware scheduling for Data-Parallel clusters. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pages 81–97, Savannah, GA, November 2016. USENIX Association.

[24] guifi.net. Commons Telecommunications Network Open, Free and Neutral . https://guifi.net, 2004. [Online; accessed 18-Feb-2023].

[25] Akram Hakiri, Aniruddha Gokhale, and Pascal Berthou. Software-defined wireless mesh networking for reliable and real-time smart city cyber physical applications. In Proceedings of the 27th International Conference on Real-Time Networks and Systems, RTNS '19, page 165–175, New York, NY, USA, 2019. Association for Computing Machinery.

[26] Yiwen Han, Shihao Shen, Xiaofei Wang, Shiqiang Wang, and Victor C.M. Leung. Tailored learning-based scheduling for kubernetes-oriented edge-cloud system. In IEEE INFOCOM 2021 - IEEE Conference on Computer Communications, pages 1–10, 2021.

[27] Istio. Istio, 2022. https://istio.io/.

[28] Roger P. Karrer, Ístvan Matyasovszki, Alessio Botta, and Antonio Pescapé. Experimental evaluation and characterization of the magnets wireless backbone. In Proceedings of the 1st International Workshop on Wireless Network Testbeds, Experimental Evaluation & Characterization, WiNTECH '06, page 26–33, New York, NY, USA, 2006. Association for Computing Machinery.

[29] Kubernetes. Scheduling, Preemption and Eviction. https://kubernetes.io/docs/concepts/scheduling-eviction/, 2022.

[30] Kubernetes. Forensic container checkpointing in Kubernetes, 2023. lhttps://kubernetes.io/blog/2022/12/05/forensic-container-checkpointing-alpha/ .

[31] Adisorn Lertsinsrubtavee, Mennan Selimi, Arjuna Sathiaseelan, Llorenç Cerdà-Alabern, Leandro Navarro, and Jon Crowcroft. Information-centric multi-access edge computing platform for community mesh networks. In Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies, pages 1–12, 2018.

[32] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. Faasflow: Enable efficient workflow execution for function-as-a-service. ASPLOS '22, page 782–796, New York, NY, USA, 2022. Association for Computing Machinery.

[33] Linux. tc. https://man7.org/linux/man-pages/man8/tc.8.html, 2001. [Online; accessed 18-Feb-2023].

[34] Lyft. Envoy, 2022. https://www.envoyproxy.io/.

[35] Leonardo Maccari. Detecting and mitigating points of failure in community networks: A graph-based approach. IEEE Transactions on Computational Social Systems, 6(1):103–116, 2019.

[36] Leonardo Maccari, Luca Baldesi, Renato Antonio Lo Cigno, Jacopo Forconi, and Alessio Caiazza. Live video streaming for community networks, experimenting with peerstreamer on the ninux community. In Proceedings of the 2015 Workshop on Do-It-Yourself Networking: An Interdisciplinary Approach, DIYNetworking '15, page 1–6, New York, NY, USA, 2015. Association for Computing Machinery.

[37] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. Wisefuse: Workload characterization and dag transformation for serverless workflows. Proc. ACM Meas. Anal. Comput. Syst., 6(2), jun 2022.

[38] Ying Mao, Jenna Oak, Anthony Pompili, Daniel Beer, Tao Han, and Peizhao Hu. Draps: Dynamic and resource-aware placement scheme for docker containers in a heterogeneous cluster. In 2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC), pages 1–8. IEEE, 2017.

[39] Xiaoqiao Meng, Vasileios Pappas, and Li Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In 2010 Proceedings IEEE INFOCOM, pages 1–9, 2010.

[40] Matteo Nardelli, Valeria Cardellini, and Emiliano Casalicchio. Multi-level elastic deployment of containerized applications in geo-distributed environments. In 2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud), pages 1–8, 2018.

[41] Vinod Nigade, Pablo Bauszat, Henri Bal, and Lin Wang. Jellyfish: Timely inference serving for dynamic edge networks. In 2022 IEEE Real-Time Systems Symposium (RTSS), pages 277–290, 2022.

[42] Zhaolong Ning, Jun Huang, Xiaojie Wang, Joel J. P. C. Rodrigues, and Lei Guo. Mobile edge computing-enabled internet of vehicles: Toward energy-efficient scheduling. IEEE Network, 33(5):198–205, 2019.

[43] NYC Mesh. A complete, cross-platform solution to record, convert and stream audio and video., 2024. v.

[44] Ramakrishna Padmanabhan, Aaron Schulman, Dave Levin, and Neil Spring. Residential links under the weather. In Proceedings of the ACM Special Interest Group on Data Communication, pages 145–158. 2019.

[45] Philadelphia Community Wireless. Network Coverage Map, 2021. https://phillycommunitywireless.org/networkmap/.
[46] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pages 805–825. USENIX Association, November 2020.
[47] Darijo Raca, Jason J. Quinlan, Ahmed H. Zahran, and Cormac J. Sreenan. Beyond throughput: A 4g lte dataset with channel and context metrics. In Proceedings of the 9th ACM Multimedia Systems Conference, MMSys '18, page 460–465, New York, NY, USA, 2018. Association for Computing Machinery.
[48] Rancher. Lightweight Kubernetes. https://k3s.io/, 2019.
[49] Huzur Saran and Vijay V Vazirani. Finding k cuts within twice the optimal. SIAM Journal on Computing, 24(1):101–108, 1995.
[50] Enrique Saurez, Harshit Gupta, Alexandros Daglis, and Umakishore Ramachandran. OneEdge: An Efficient Control Plane for Geo-Distributed Infrastructures. In SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021, pages 182–196. ACM, 2021.
[51] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. Memory deduplication for serverless computing with medes. In Proceedings of the Seventeenth European Conference on Computer Systems, EuroSys '22, page 714–729, New York, NY, USA, 2022. Association for Computing Machinery.
[52] Sean DuBois. The Open Source, Cross Platform Stack for Real-time Media and Data Communication, 2020. https://github.com/pion/.
[53] Manasvini Sethuraman, Anirudh Sarma, Ashutosh Dhekne, and Umakishore Ramachandran. Foresight: planning for spatial and temporal variations in bandwidth for streaming services on mobile devices. In Proceedings of the 12th ACM Multimedia Systems Conference, pages 227–240, 2021.
[54] Esther Showalter, Nicole Moghaddas, Morgan Vigil-Hayes, Ellen Zegura, and Elizabeth Belding. Indigenous internet: Nuances of native american internet use. ICTD '19, New York, NY, USA, 2019. Association for Computing Machinery.
[55] SoundCloud. From metrics to Insight, 2022. https://prometheus.io/.
[56] Jakob Struye, Bart Braem, Steven Latré, and Johann Marquez-Barja. The citylab testbed—large-scale multi-technology wireless experimentation in a city environment: Neural network-based interference prediction in a smart city. In IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), pages 529–534. IEEE, 2018.
[57] Lin Wang, Lei Jiao, Ting He, Jun Li, and Henri Bal. Service placement for collaborative edge applications. IEEE/ACM Transactions on Networking, 29(1):34–47, 2021.
[58] Cong Xu, Karthick Rajamani, and Wesley Felter. Nbwguard: Realizing network qos for kubernetes. In Proceedings of the 19th International Middleware Conference Industry, Middleware '18, page 32–38, New York, NY, USA, 2018. Association for Computing Machinery.
[59] Mingjin Zhang, Jiannong Cao, Lei Yang, Liang Zhang, Yuvraj Sahni, and Shan Jiang. Ents: An edge-native task scheduling system for collaborative edge computing. In 2022 IEEE/ACM 7th Symposium on Edge Computing (SEC), pages 149–161. IEEE, 2022.
[60] Tongxin Zhu, Tuo Shi, Jianzhong Li, Zhipeng Cai, and Xun Zhou. Task scheduling in deadline-aware mobile edge computing systems. IEEE Internet of Things Journal, 6(3):4854–4866, 2019.