

Priority-enabled Load Balancing for Dispersed Computing

Aaron Paulos*, Soura Dasgupta[†], Jacob Beal*, Yuanqiu Mo[†], Jon Schewe*, Alexander Wald*, Partha Pal*, Richard Schantz*, and J. Bryan Lyles[‡]

*Raytheon BBN Technologies, Cambridge, MA 02138

Email: {aaron.paulos, jake.beal, jon.schewe, alexander.wald, partha.pal, richard.schantz}@raytheon.com

[†]Department of Electrical and Computer Engineering, University of Iowa, Iowa City, IA 52242

Email: {soura-dasgupta, yuanqiu-mo}@uiowa.edu

[‡]University of Tennessee Knoxville, Knoxville, Tennessee 37996

Email: jblyles@acm.org

Abstract—Opportunistic managed access to local in-network compute resources can improve the performance of distributed applications and reduce the dependence on shared network resources. Instead of backhauling application data to a centralized cloud data center for processing, networked services may be adaptively and continuously dispersed into shared compute resources that are closer to the source of need. While this approach has several benefits, support for mission-aware access to computation is often an afterthought, and is implemented as a brittle extension over traditional load-balancer solutions.

In this work, we investigate the design of two priority-aware resource allocation strategies and two load-balancing dispatching strategies as first class citizens in an open-source dispersed computing middleware. We present a control theoretic analysis of these load-balancing primitives to identify weaknesses and strengths in our design, and recommend future directions. In parallel, we prototype two priority-aware allocation algorithms to validate our priority predictions. In initial experiments our prototype shows substantial gains in processing prioritized load. Finally, we make our source-code and experimental configurations open source.

Keywords—dispersed computing, middleware, load-balancing

I. INTRODUCTION

Cloud and edge computing models support dynamic computational needs by coupling elastic pools of low-cost centrally located compute resources alongside pre-positioned and provisioned edge resources. Applications that require low-latency processing or publish large amounts of sensed data can be handled locally in edge clusters, where excess traffic may be overflowed into cloud data centers when edge resources are constrained. As adoption of this model evolves and powerful computation and memory resources emerge within the network itself, the computational boundary is further blurred enabling widely-distributed opportunistic dispersal of computation into the network instead of just clustered resources.

Supported by Defense Advanced Research Projects Agency (DARPA) contract HR001117C0049. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. This document does not contain technology or technical data controlled under either U.S. International Traffic in Arms Regulation or U.S. Export Administration Regulations. DARPA DISTAR 33423, 09/02/20, Distribution "A" (Approved for Public Release, Distribution Unlimited).

Edge and fog computing [1] describe dispersed computational networks that smear the boundaries of edge-clusters and centralized clouds. Examples include emerging urban wireless networks [2] interspersing compute resources and access points, edge cloudlets [3], [4], and mixed tiered compute-as-a-service hierarchies proposed by telecoms and ISPs [5]. All share the notion of a distributed, reservable, and programmable network compute point (NCP), which can benefit individual application quality-of-service (QoS) and collective mission operation in face of highly dynamic deployments.

Building upon NCPs, benefits are envisioned for a broad set of locavore applications, ranging from waste and water management to smarter retail store automation [6]. Parallel use cases exist in the defense domain, where scheduling and mission aware access to resources is paramount. For example, NCPs can provide bandwidth efficient low-latency compute bridges for tactical and dynamic multi-actor digital world building for situational awareness, or for federated search missions (e.g., threat identification or rescue) in remote and austere environments. One can envision scenarios where success is predicated on resource access. For example, a critical search and characterization mission relying on sensed aggregation and filtering to identify and classify emergent threats would be favored over a lower-priority world building operation, e.g., a drone-based orthomosaic image stitching map augmentation service.

To enable mission effective utilization of NCPs, operators require middleware infrastructure to natively support inter-related functions. First, it must provide optimized, adaptive placement of data and services to minimize backhauled traffic and application latency. Second, resource monitoring and input curation, decision making algorithms, and actuation of service placements (plan execution), must be accurate, robust, and resilient in face of emergent failures, overload, and other network events. Finally, it must support prioritized resource allocation and dispatching operations as first order citizens, where mission operators can define aspects such as service importance to a mission collective and the middleware design is then fully capable of enforcing prioritized resource access.

In this work, we design, analyze, and prototype two priority-

aware resource allocation and dispatching strategies in the Mission-oriented Adaptive Placement (MAP) middleware [7]. MAP is a distributed, multi-layer decision making and resource management middleware. It has shown promise in distributing computational services in a scalable and resilient way to minimize backhaul traffic and improve client performance, even in face of changes in demand and availability of network and computing resources [8]. Unlike state-of-the-art resource allocation middleware [9], [10] that rely upon proven and external hardware and software load balancers [11], [12], [13], MAP manages both resource allocation and client assignment (dispatching) in a unified scheduler and management domain. This makes it easy to design, analyze, implement prioritized load-balancing over both allocation and dispatching operators.

We make the following four contributions. First, we design and prototype greedy and reservation-based priority-aware resource allocation strategies in an open-source agent-based middleware. Second, in a theoretic analysis, we characterize, identify, and enumerate the strengths and weaknesses of two priority-based allocation and two dispatching strategies. We show how demand prediction can improve outcomes in future work. Third, we present empirical results on a networked testbed that verifies the observations in our analysis, and further shows variable improvements up to a 4.1x improvement in processing of prioritized load over a baseline case. Fourth, we open-source our code and experiment configurations.

This paper is structured as follows. Section two provides background on MAP and describes our priority-aware allocation designs. Section three characterizes the strengths and weaknesses of allocation and dispatching operators. Section four presents experimental results supporting our analytic observations. Section five discusses our contributions in the context of related work. In the final section we summarize our findings and discuss next steps.

II. BACKGROUND AND PRIORITY-AWARE ALLOCATION

To support varied dispersed computing needs, we implement a greedy and reservation-based priority-aware allocation algorithm in the regional load-balancing gateway (RLG) component of MAP [7], [8]. In MAP (Figure 1), a RLG manages a pool of tightly connected NCPs, referred to as a region. A region can be thought of as a set of NCPs, such as clusters on rooftops in a dense urban area [14] or compute racks mounted on defense vehicles (e.g., L-ATVs [15]). Within a region, the RLG assumes that NCPs are client- or edge-facing and that the connectivity (i.e., bandwidth, loss-rates) between clients and NCPs is of similar quality. While outside of the scope of this paper, to manage very large scale global compute networks, MAP uses a divide-and-conquer approach employing DCOP algorithms [16], [17] to manage many regions.

A RLG is a sensor-sink and actuator that monitors NCP states and applies best-fit bin-packing and DNS-based load-balancing to manage client load. It manages *service allocation* or how a limited number of NCPs are allocated to a particular service type (e.g., a web app) and *dispatching* or how clients should be load-balanced into a number of NCPs running a

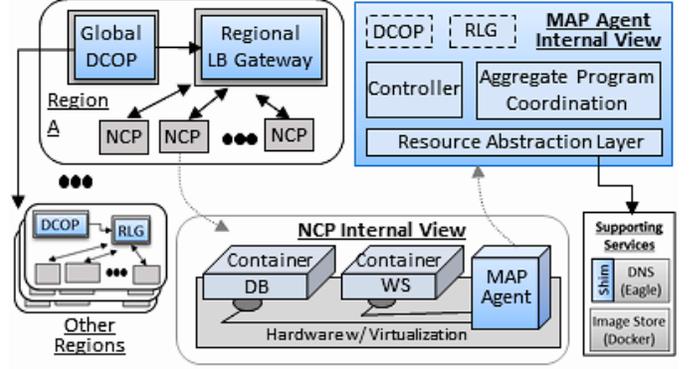


Fig. 1. MAP Middleware. Network organization, agent internals, and supporting services. Middleware as blue boxes.

requested service. We implement *service priority* support as a relative importance of a given service type with respect to other services that might be executing at the same time. We use a simple numeric scale where higher numbers indicate more importance, where the complexity of evaluating utility to a mission-collective is placed in the allocation algorithm.

We design and implement two priority-aware resource allocation algorithms. In the spirit of fair queuing approaches, a *Priority Weighted Reservation* algorithm grants active services a weighted fair share of an upper-bound of available NCP resources to prevent starvation of lower-priority services. In contrast, a *Greedy Group* algorithm provides best possible resource access to support demand for high-priority services over lower priority services. We list pseudo-code for these designs here, and make our source code for this work available at <https://github.com/map-dcomp>.

Algorithm 1 Priority Weighted Reservation

```

RC ← NumRegionContainers
SS ← ServiceSpecificationList < service >
PS ← PriorityOrderActiveServiceList < service >
AM ← AllocationMap < service, containers >
for all S in SS do
  AM.addContainers(S, 1)
  RC ← (RC - 1)
PW ← calcPriProportionalWeightMAP(PS, RC)
for all S in PS do
  WCF ← Floor(RC * PW(S))
  AM.addContainers(S, WCF)
  RC ← (RC - WCF)
for all S in PS such that RC > 0 do
  WC ← RC * PW(S)
  WCF ← Floor(WC)
  if Modulo(WCF, WC) > 0 then
    AM.addContainers(S, 1)
    RC ← (RC - 1)
{For brevity, we omit fraction tie-breaking procedures.}

```

Listing 1 summarizes the priority-weighted reservation al-

gorithm. As inputs, the algorithm takes a number of regional containers (i.e., a compartmentalized resource slice of an NCP) available for hosting services, the list of all services that are specified as *potentially active* (e.g., required for a mission), and a list of active services, or those that are currently under client load. As an output, the algorithm will populate an allocation map that defines the number of container resources available for each service. For listing brevity, we assume the active service list is pre-sorted in descending priority order.

As a minimal requirement, each specified service at configuration time is allocated a single container at run time. The `calcPriProportionalWeightMAP` method weights each *active services* target priority as a fraction of the total priority of the active set of services within the region, and then calculates a share of the remaining containers for allocation. This is stored in a map $PW \langle \text{service}, \text{float} \rangle$, where the float represents the number of containers to allocate for each service. A second-pass allocation step assigns the `FLOOR` of the float value to each priority service and updates the remaining regional containers counter. A final series of steps allocate fractional weightings to services in descending order of priority while regional resources remain. As a result, this algorithm will create a fair-share of an upper-allocation bound across the pool of active services.

Algorithm 2 Greedy Group

```

RC ← NumRegionContainers
SS ← ServiceSpecificationList < service >
AM ← AllocationMap < service, containers >
PM ← PriorityGroupMapDescendingOrder <
priority, List < service >>
for all S in SS do
  AM.addContainers(S, 1)
  RC ← (RC - 1)
for all G in PM such that RC > 0 do
  GD ← sumServiceDemandInGroup(G)
  AC ← Min(RC, Floor(GD))
  for all S in PM(G).getServiceList() do
    NT = AC/S.listSize()
    AM.addContainers(S, NT)
  RC ← (RC - AC)
{For brevity, we omit handling for multiple service types at
the same priority and for fractional allocation targets.}

```

Listing 2 shows the greedy group algorithm. As inputs it takes the number of regional containers and the service specification list. Greedy also produces an allocation map as an output. Since greedy works over priority groups, this algorithm is provided a priority group map as an input that can be iterated over in descending order of priority. The priority group map further indexes service lists by priority group. A first step of the algorithm preforms a base allocation of one container for each service in the specification list.

This implementation will traverse the priority group map in descending order while containers can still be allocated.

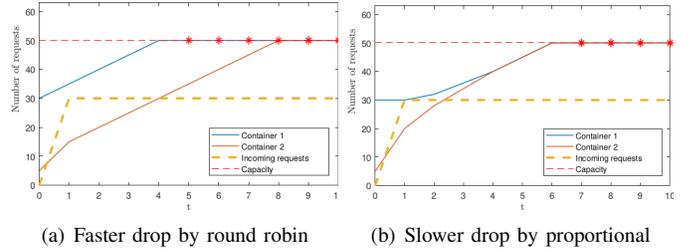


Fig. 2. Round robin vs. proportional distribution without container allocation and with slow processing (no request completed). Capacity is 50 req/container. In this case, $L_1(0) = 30$ and $L_2(0) = 5$. In left figure, container 1 drops at time $t = 5$. In right figure, $L_1(t) \geq L_2(t)$ for $t < 5$, and proportional distribution will drop at time $t = 7$. Stars indicate dropping.

At each iteration, it will look at the sum of service demand across all service instances within the priority group, and then take a minimum of the remaining regional containers and the sum total to determine an allocation target for the group. This minimum result is then used as an allocation target for the service set. The algorithm will then split that group allocation target equally across different service types within the priority group. For brevity, we omit logic that addresses handling of factions and more than one service type in the same priority group. As an outcome, this algorithm will greedily allocate resources to higher-priority services first, potentially starving lower-priority services.

III. ANALYSIS OF ALLOCATION AND DISPATCHING

Here our analysis considers two RLG roles: *allocation* - to spin or decommission containers when the load exceeds or falls below designated proportions of capacity, and *dispatching* - to publish client-associating DNS plans. We first examine possible RLG heuristics for publishing plans for DNS to distribute incoming demand - without heed to service priorities.

The two heuristics are: *round robin*, where incoming load is assigned to containers in succession, without considering available capacity. The second is *proportional distribution* where loads are assigned to containers in proportion to their available capacity. With round robin the container with the highest initial load will retain the largest load, *proportional tends to equalize loads across the containers*. As a result, in extreme cases the first container will fill up quickly leading to service drops (e.g., decreased QoS). Figure 2, with $L_i(t)$ as the load in container i at time t , depicts a scenario where no containers are spun and shows proportional dropping later. Figure 3 depicts container spinning: round robin drops before total capacity is exceeded but proportional does not.

We now consider services with different priorities comparing (A) *greedy allocations*, where the needs of a higher priority service are met first; and (B) *proportional reservation* where services are given a capacity slice based upon priority. Allocation is in proportion to the slice with no service allocated more than it needs. Figure 4 assumes that 100 containers are available for allocation between three services with priorities 5, 3, and 2. As shown in the second column, at time t they need 30%, 45%, and 60% of the 100 containers,

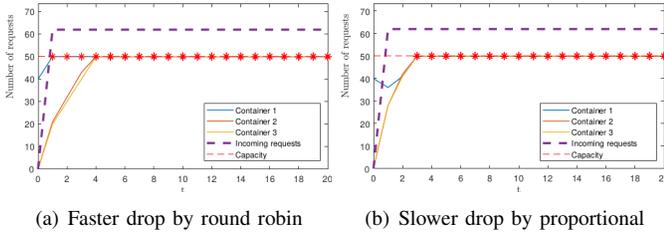


Fig. 3. Round robin vs. proportional with container allocation. Capacity is 50 req/container. Container 1 gets close to capacity at time $t = 0$, and two more containers are spun up. Under proportional distribution, in 3b, jobs drop only after all containers are exhausted. Under round robin, 3a, the first container will drop jobs before others are full. Stars indicate dropping.

Scenario: 3 Priorities Competing (slice of 100 containers need at timestep)				MAP RLG at time t (Containers Allocated)		Prediction Recommendation (Allocation at t and $t+1$)	
	Known Load at $t-1$	Known Load at t	Estimated Load at $t+1$	Proportional Reservation (some inversion)	Greedy (no inversion)	Greedy Prediction (Assigned)	Greedy Prediction (Reserved)
Service 1 (high)	20	30	40	30	30	30	10
Service 2 (med)	35	45	55	42	45	45	10
Service 3 (low)	30	60	90	28	25	5	0

Fig. 4. Comparisons of priority resource allocation strategies: outcomes and prediction recommendation. First three columns illustrate scenario, middle two show MAP allocation, and final two illustrate a predictive recommendation.

respectively. Accordingly, as shown in the fifth column, service 1 is assigned its required 30 containers as it has a max reserve of 50 (proportional to service 2 and 3). However, services 2 and 3 are granted 3/5 and 2/5 of the remaining 70 containers, and as a result service 2 is given only 42 of the 45 it requires, while service 3 gets 28. The resulting priority inversion is a disadvantage of (B). In contrast, under greedy allocations, in the sixth column, services 1 and 2 are assigned all they need, with the remaining going to low. Herein lies the key difference between (A) and (B). While (B) leads to priority inversion, (A) does not. Yet in the absence of demand prediction, greedy allocations may end up assigning more to a higher priority service than it needs, starving others of needed resources. This can lead to suboptimal performance when edge resources are consumed, and outflow is required to the datacenter.

As an example of the suboptimal performance, Figure 5 depicts two regions (edge and cloud) with 12 containers each and services A, B, and C in priority order. Suppose a proportional reservation of five, three and one containers, according to service priorities are made in the edge region for A, B and C, respectively, followed by two successive units of C arrive in successive times. After the second arrival, having exhausted its reserved containers in the edge, C is suboptimally assigned to the cloud, even though, neither higher priority service has utilized the empty containers assigned to them.

Much of this conservatism can be reduced through predicting not just arrival rates but also processing rates, or equivalently the net rate of change. Indeed the absence of such prediction may even cause instabilities through unnecessary service drops. Thus consider a scenario where all containers are assigned to a high priority service. Then as these are

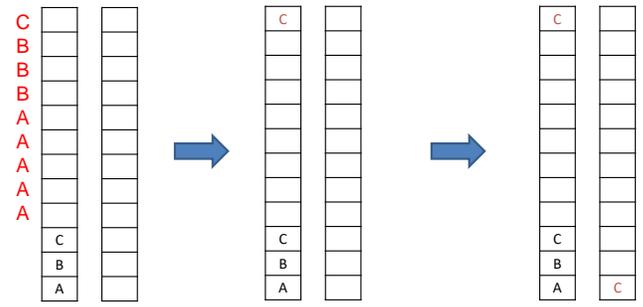


Fig. 5. Conservatism of proportional reservation with only load prediction. Edge and cloud regions each with one NCP and 12 containers. Service A has high priority, followed by B, and C. Initially 5, 3, and 1 containers in the edge region are reserved for A, B, and C. Two successive units of C arrive in successive times. After the second arrival, having exhausted its reserved containers in the first region, C is suboptimally assigned to the cloud.

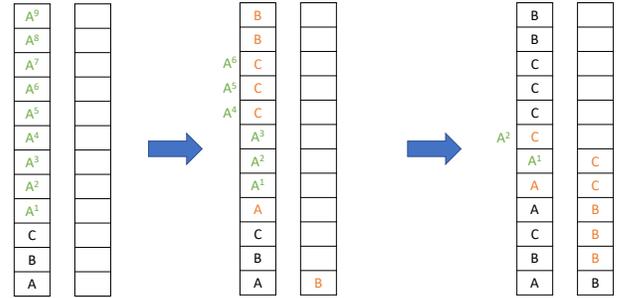


Fig. 6. Reduced conservatism due to prediction of arrivals and processing (identical setup to Fig. 5). A is predicted to grow at one unit per round, and B and C at three units per round. Further, C takes one round to complete, while B takes 7 rounds to complete. Thus C is accommodated in round 2. Numbers in green are the containers that will be eventually given to A.

occupied at the start of a round, new low priority services will be denied containers in the next round even if in the intervening period all load is processed.

The lack of conservatism due to such prediction is depicted in Figure 6. In this case A is predicted to grow at one unit per round, and B and C at three units per round. Further, C takes one round to complete, while B takes 7 rounds to complete. Then even though nine containers have been reserved for A (depicted in green), the arrival of three units of C and two of B in the next round does not preclude their being assigned containers from the edge, reducing the amount of suboptimal allocations. The effect of prediction is further illustrated in Figure 4, where column four linearly depicts the estimated load for each service (40, 55, and 90 for A, B, and C) in the next round. Accordingly, greedy allocation at time t and reservations for $t+1$ depicted in the last two columns, respectively, show the avoidance of priority inversion.

A. The Role of Prediction

Reasonable prediction of both the arrival rate and the processing rate can further supplement proportional assignment which postpones and even prevents job drops. *Both predictions are equally important.* For example suppose a job is initially assigned two containers A and B, and incoming demand fills

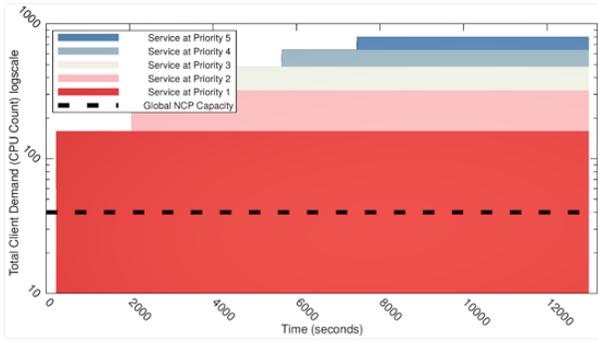


Fig. 7. Stress test demand for increasingly important services.

them up. Containers C and D will spin up, and absent estimates of processing rates, will assign no jobs to A and B in the next cycle assuming they are full. This despite the fact that A and B may be emptied by the time of advent of the next cycle. This could lead to C and D dropping jobs even though the NCP has space left, and in some scenarios under constant demand, to oscillatory behavior with periodic and frequent job drops.

Predictions regarding processing rates can in fact be as simple as monitoring the incoming load and the change in the load from each cycle to the next. Or one can use a longer historical record to make more sophisticated predictions. For example one can choose the net predicted change in available capacity as a weighted average of previous changes. If needed, the weights can be chosen adaptively, e.g., by minimizing the square of the prediction error.

IV. EXPERIMENTAL VALIDATION

Results from experiments with the greedy and priority weighted implementations confirm our priority analysis. We also compare the performance of our work against a no-MAP baseline. We use PC3000s on Emulab [18] as a testbed, with a single container/vCPU/node. We use 40 NCPs and 40 clients sparsely connected as a topology. For stimulus, edge client hardware executes 20 clients in parallel for each priority group under test. In our preliminary tests we use 5 priority groups, which means 100 client threads per node, for a total of 4,000 clients. Client threads for each priority group are uniformly distributed across the topology. Individual client requests are specified to minimally impact the network, as such, they do not suffer impacts common in network-bound scenarios.

In Figure 7 we graph the input demand scenario for this test (log scale y-axis). The 40 container processing potential is shown as the dashed black line. Any shaded area above the black line is excess demand. This scenario is designed as a stress test where the effects of prioritization will be readily apparent. Each client request is an attempt to apply a 20% CPU load for 60 seconds. For a prioritized service, we implement a thread-per connection server backed by the FakeLoad library [19]. This experimental design allows a service "hard-fit" client requests (e.g., 5 20% requests will succeed in a window, a 6th request will fail). This ensures that both the experiment setup and analysis tasks are easy to understand.

Figure 8 shows two complementary trace views from three experimental configurations. The top row shows how containers are allocated for one of five priority services over time. The bottom row graphs load across the collective CPUs were for each service type over time. From right to left, the columns show a trace of a greedy experiment, a reservation experiment, and a no map experiment. In all graphs, the black dotted line at 40, shows the ceiling for processing or allocation. The color shaded areas in the graphs correspond to measurements collected for one of five service priority groups. These plots illustrate some fine points of these experiments. First, the no map baseline statically allocates resources, much like how an operator might provision an edge cluster. Second, unlike the no-map configurations, the with-map configurations are dynamic and control both the (de)allocation and dispatching, thus the scale sloped up in the with map graphs for both allocation and load processed. Third, by visually comparing the greedy and reservation runs, it is easy to see (i) how priority inversion manifests in the reservation configuration for priority waves P2-P5 given the demand input shown in Figure 7 and (ii) how the greedy configuration will starve any lower priority service in the demand mix. Fourth, as designed, the priority prototypes faithful reflect the behaviors we expected in both intent and our analysis in Section III.

In Figure 9 we compare application load processed under the two priority algorithms against a static pre-allocation of 8 containers for each of the 5 priority waves. To make this comparison, we calculate a load ratio between MAP and no-MAP experiments. We curate this data by summing the load processed from the start of a service priority grouping until the planned preemption of that grouping. As an example, we look at load across the 40 NCP topology for P1 from 5 minutes, until P2 starts at 35 minutes. Considering our test scenario, this means P1 - P4 are active for 30 minutes under this data analysis approach, while P5 is active 90 minutes.

With this metric we see processed load gains vary from 156% to 408%. Ratio gains are a function of topology and configuration, and how rapidly MAP adapts to the flow of prioritized client demands. On the one hand, the baseline does not amortize the allocation cost of scaling up and down container resources, this means it is ready to process client requests the instant they arrive. In contrast, the datasets supporting MAP roll-up all of the allocation and deallocation costs associated with adapting to each new wave. Similarly, if we restrict the baseline to a single NCP per-priority application in this topology (i.e., fixed edge cluster versus a truly dispersed compute environment), we would observe an 8-fold improvement in gains with MAP between 1,250% and 3,250%.

While Figure 9 offers insight against the baseline, additional analysis is needed to assess the two designs. Within the P1 wave both designs will be on equal footing because they start from the same blank slate. We expect the two algorithms to produce similar results in this wave, and they are within 1% difference. The small variance shown here is the result of averaging load data across three iterations. We use three runs here due to a number of practical considerations, e.g., experi-

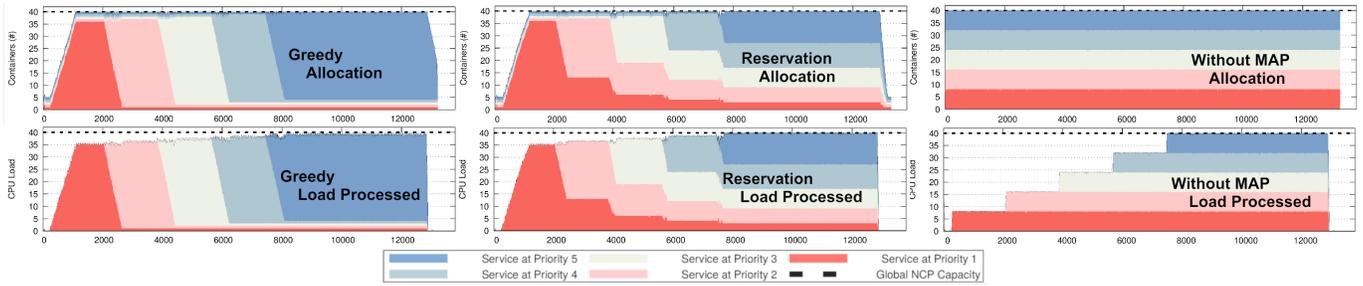


Fig. 8. Experimental traces for greedy, reservation, and no map experiments. Two row shows allocated containers for each service priority type. Bottom row shows load generated on containers within network. Peak resource line is shown in black at 40 NCPs.

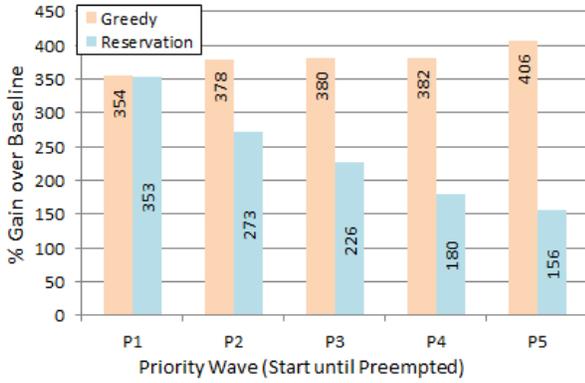


Fig. 9. Load processed in each of five priority waves - comparison of prioritized work for greedy and reservation runs against no-map.

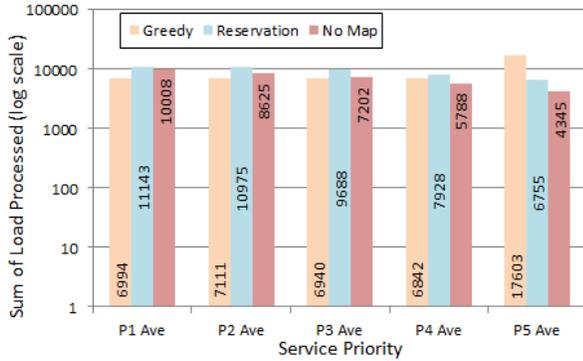


Fig. 10. Load processed in each of five priority waves - summary of total work for no-map and priority algorithms (log-scale y-axis).

mentation on real hardware like Emulab is time consuming and subject to resource contention with other researchers, but more importantly because the observed sum load variance was both minimal and stable across runs. Further examining the greedy waves P2-P4, we see approximately similar gains, as expected. This implies the priority implementations are both effectively and efficiently and predictable in allocation and deallocation behaviors. As mentioned before, P5 runs longer by design, so its gains are larger than the first four waves.

Figure 9 appears to show greedy outperforming reservation for the P2-P5 waves. Based upon the metric definition, this is

expected. By design, greedy will always reallocate resources to the highest priority group, in contrast, the reservation approach is offering a fairer share to competing priority groups. That means as more and more stressful load accumulates on the system and the reservation approach divides resources, and as a result there is less compute power to service the next wave. The descending pattern in this figure illustrates this point.

In Figure 10 we examine the sum of raw load processed. On the y-axis, in log-scale, we average the sum of load processed for each priority service across the three iterations of each experimental configuration. The x-axis shows priority service grouping, comparing the greedy, reservation, and no-map datasets. Examining this data we see a few trends. First, in the no-map case, the load processed by each successive priority group declines. This can be explained when considering that resources are held constant for each priority group and the experimental time for each priority group is less than the next. Second, in the greedy data set, for groups P1-P4, the amount of work done for each group has little variance. This is expected as the greedy algorithm artificially limits both the time and resources available to process any planned demand. Furthermore, it illustrates a small variance across groups, this further supports the credibility of the smaller sample size. Third, in reservation, with the exception of the P5 group all services process more load than the greedy and no-map configurations. Similar to the time and allocation observations above, this is because the reservation approach allows lower priority services to continue working, both through allocation and dispatching, they enjoy a better degree of service.

A. Discussion

These experiments support our analytic analysis and show *substantial* improvements over the configured baseline. However, experimental application to targeted scenarios will help to better understand the value from these enhancements. For example, given these outcomes, is there a community or operational preference for one of the two algorithms, and more generally, is the base-design of compute priority as an importance a satisfactory complement to prioritization schemes in other infrastructure (e.g., the networks). Furthermore, while our previous work has evaluated MAP's performance a number of generic deployment graphs, would operators requiring prioritization desire support for more spe-

cialized or challenging computational graphs (e.g., extremely limited connectivity, or with mobile aspects). Another area of experimental investigation should look at direct comparisons to the state-of-practice ecosystem for resource management of truly dispersed compute resources. As described in Section V, to our knowledge MAP’s unified scheduling and management domain is sufficiently unique. This makes direct comparisons of the presented priority schemes to both the tightly controlled cluster management software and their disjoint dispatching routines - *that would enable comparable dispersed computing capabilities* - a serious undertaking.

V. RELATED WORK

Recent resource management challenges in dispersed computing is extensively covered in survey literature [20], [21], [1]. These works recognize many ongoing and open challenges in managing dispersed heterogeneous devices and scheduling resource access in face of shifting compute networks. They further suggest a wide variety of applications and use-cases for networked compute environments. *Perhaps most striking is that collectively these works survey over 400 publications, but only list a single reference of priority-support in the middleware [22] and load-balancing area.* While less surprising, there were no mentions of mission or mission-awareness, which is more often used in defense settings.

While not explicitly called out in surveys, load-balancing with priority support is an active area of interest [23], [24], [25], [26], [27], [28]. Unlike work presented here, these focus on the centralized cloud environment, which is complementary to our work. Further afield, in application-specific domains and IoT settings, there is solution-focused middleware with priority support [29], [30]. Our study of tightly coupled and general strategies for performing prioritized-dispatching and allocation complements these works.

In state-of-practice, cloud and cluster management platforms such as OpenStack with Nova and Neutron [10] and Kubernetes [9] are commonly used tools for orchestrating shared resource access and scaling applications within tightly clustered compute environments. They are also make up part of the currently recommended solutions to address emerging 5G compute concerns [14]. Since the MAP prototype is most related to container virtualization and wide-area load-balancing, which is an active area of interest in defense areas [31], [32], we narrow our related works discussion to describe research and solutions related to the latter technology.

Scalable containerized deployment within Kubernetes is efficient and effective and well adopted. Unlike the widely dispersed computing environments that MAP’s algorithms are attempting to target, Kubernetes is mostly focused on cluster management. While Pods’ do support priority, like much of this infrastructure’s design, Pod’s are always co-located and co-scheduled. As such it relies on a centralized scheduler called kube-scheduler, that performs a two step assignment process of filtering available resources by application requirements followed by scoring to rank suitable assignments to control allocation. Over the years, this technology has

considered approaches to federation support [33] and the research community has also look at the proposed designs in some detail. In [34], the authors identify limitations to load-balancing across V1 federation. They further presented a compelling architecture and implementation of a portable load balancer for supporting load migration across distinct clusters. At the time of this writing, Kubernetes has since abandoned the design cited in Takahashi’s work and still maintains limited load-balancing support naively.

By design, Kubernetes delegates load-balancing of clients to hardware or software load-balancers. In the former case, this is often hardware provided by the cluster provisioner/leaser. In the latter case, the community tends to leverage proven load-balancing techniques, such as software proxies [11], [12], [13]. Behind these cluster-facing load-balancers, many solutions further employ DNS-based load-balancing, which is has been extensively covered in literature [35], [36], [37], [38]. As a state-of-practice solution, this implies that users requiring end-to-end support for prioritized resource allocation and dispatching will need to thread such a construct through multiple component systems. This can lead to brittle outcomes.

VI. CONCLUSION AND DISCUSSION

Combining scalable orchestration middleware with software load-balancers results in efficient and effective response to client load in concentrated computational pools. However, as decoupled components, such solutions often support prioritized service placement and dispatching only in extensions for truly dispersed environments. Furthermore, while mission-critical scenarios (e.g., disaster recovery) make heavy use of off-the-shelf orchestration frameworks, they also require prioritize access to limited non-centralized resources in such highly dynamic settings. This often leads to inefficient outcomes such as over-or-under provisioned edge resources or requiring manual orchestration of prioritized trade-offs in brittle ways and at slow timescales. As a result, such scenarios will likely make little to no use of additional dynamic resources available within the network itself.

We design, analyze, and prototype priority-aware resource allocation and dispatching as first-order citizens in an agent-based middleware for adaptively managing in-network compute resources. We analytically describe the strengths and weaknesses of these designs to show a clear benefit from proportional load dispatching over round-robin solutions. We suggest incorporating prediction to enhance the proposed solutions, and validate analytic findings on a network testbed. Experimental scenarios show marked improvement up to 4.1x over a baseline centralized case. Finally, we open source our source code and experimental configurations, in part, to support independent corroboration of our results.

This work suggests immediate next steps. First, as our analysis shows, to more accurately manage priority workflows reactive load-balancing solutions will benefit from a prediction module capable of quantifying the amount of work in future steps. One approach we have considered is to use a Martingale model to characterize workloads that users might care about.

Second, in the spirit of large-scale dispersal with multi-layer systems such as MAP, algorithmic decision making will need to be carefully designed and analyzed to allow prioritization decisions to flow down across layer, especially in face of novel compute environments and application mixtures. We plan to leverage defense/commercial dual-use scenarios and use-cases, e.g., user-plane virtual network function placement in 5G environments, to motivate such large scale investigations.

REFERENCES

- [1] Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P. Jue, "All one needs to know about fog computing and related edge computing paradigms: A complete survey," *Journal of Systems Architecture*, 2019.
- [2] Powder, "Powder: Platform for open wireless data-driven experimental research," web <https://powderwireless.net>, 2019.
- [3] Akamai, "Cloudlet applications — akamai," web url=<https://www.akamai.com/us/en/products/performance/cloudlets/>, 2019.
- [4] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, Oct 2009.
- [5] CenturyLink, "Cdn edge computing — centurylink," web url=<https://www.centurylink.com/business/networking/cdn-edge-compute.html>, 2019.
- [6] Charith Perera, Yongrui Qin, Julio C. Estrella, Stephan Reiff-Marganiec, and Athanasios V. Vasilakos, "Fog computing for sustainable smart cities: A survey," *ACM Comput. Surv.*, vol. 50, no. 3, pp. 32:1–32:43, June 2017.
- [7] J. Schewe, Et. Al., "Github - map-dcomp/map-code," web <https://github.com/map-dcomp/map-code>, 2019.
- [8] Aaron Paulos, Soura Dasgupta, Jacob Beal, Yuanqiu Mo, Khoi Hoang, J. Bryan Lyles, Partha Pal, Richard Schantz, Jon Schewe, Ramesh Sitaraman, Alexander Wald, Christabel Wayllace, and William Yeoh, "A framework for self-adaptive dispersal of computing services," in *IEEE SASO Workshops*.
- [9] Kubernetes, "Production-grade container orchestration," web <https://kubernetes.io/>, 2019.
- [10] OpenStack, "Openstack docs: Stein," web <https://docs.openstack.org/stein/>, 2019.
- [11] HAProxy, "Haproxy - the reliable, high-performance tcp/http load balancer," web www.haproxy.org, 2019.
- [12] Apache, "mod_proxy - apache http server version 2.4," web url=https://httpd.apache.org/docs/2.4/mod/mod_proxy.html, 2019.
- [13] NGINX, "Nginx docs — http load balancing," web url=<https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/>, 2019.
- [14] Open Network Automation Platform, "Onap architecture - onap master documentation," web url=<https://docs.onap.org/en/latest/guides/onap-developer/architecture/index.html>, 2020.
- [15] OSHKOSH Defense, "Integrated product support," 2020.
- [16] Khoi D. Hoang, Christabel Wayllace, William Yeoh, Jacob Beal, Soura Dasgupta, Yuanqiu Mo, Aaron Paulos, and Jon Schewe, "New distributed constraint satisfaction algorithms for load balancing in edge computing: A feasibility study," in *10th International Workshop on Optimization in Multiagent Systems (OptMAS)*, May 2019.
- [17] Khoi D Hoang, Christabel Wayllace, William Yeoh, Jacob Beal, Soura Dasgupta, Yuanqiu Mo, Aaron Paulos, and Jon Schewe, "New distributed constraint reasoning algorithms for load balancing in edge computing," in *International Conference on Principles and Practice of Multi-Agent Systems (PRIMA)*, October 2019.
- [18] Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb, and Jay Lepreau, "Large-scale virtualization in the emulab network testbed," in *USENIX 2008 Annual Technical Conference*, Berkeley, CA, USA, 2008, ATC'08, pp. 113–128, USENIX Association.
- [19] Marten Sigwart, "Github - msigwart/fakeload," web <https://github.com/msigwart/fakeload/>, 2020.
- [20] Z. Hao, E. Novak, S. Yi, and Q. Li, "Challenges and software architecture for fog computing," *IEEE Internet Computing*, vol. 21, no. 2, pp. 44–53, Mar 2017.
- [21] Carlo Puliafito, Enzo Mingozzi, Francesco Longo, Antonio Puliafito, and Omer Rana, "Fog computing for the internet of things: A survey," *ACM Trans. Internet Technol.*, vol. 19, no. 2, pp. 18:1–18:41, Apr. 2019.
- [22] M. Mohamed E. Yigitoglu, L. Liu, and H. Ludwig, "Foggy: A framework for continuous automated iot application deployment in fog computing," in *In Proceedings of the 6th IEEE International Conference on AI and Mobile Services*, 2017, AIMS 17, p. 3845.
- [23] G. Raj, D. Singh, and A. Bansal, "Load balancing for resource provisioning using batch mode heuristic priority in round robin (pbr) scheduling," in *Confluence 2013: The Next Generation Information Technology Summit (4th International Conference)*, 2013, pp. 308–314.
- [24] S. Ghosh and C. Banerjee, "Dynamic time quantum priority based round robin for load balancing in cloud environment," in *2018 Fourth International Conference on Research in Computational Intelligence and Communication Networks (ICRCICN)*, 2018, pp. 33–37.
- [25] K. K. Patel, M. R. Desai, and D. R. Soni, "Dynamic priority based load balancing technique for vm placement in cloud computing," in *2017 International Conference on Computing Methodologies and Communication (ICCMC)*, 2017, pp. 78–83.
- [26] F. Sadia, N. Jahan, L. Rawshan, M. T. Jeba, and T. Bhuiyan, "A priority based dynamic resource mapping algorithm for load balancing in cloud," in *2017 4th International Conference on Advances in Electrical Engineering (ICAEE)*, 2017, pp. 176–180.
- [27] O. Lemesshko, M. Yevdokymenko, and O. Yeremenko, "Load-balancing fast rerouting model with providing fair priority-based traffic policing," in *2019 IEEE International Scientific-Practical Conference Problems of Infocommunications, Science and Technology (PIC S T)*, 2019, pp. 538–542.
- [28] S. Patil and S. R. Deshmukh, "Development of control strategy to demonstrate load priority system for demand response program," in *2019 IEEE International WIE Conference on Electrical and Computer Engineering (WIECON-ECE)*, 2019, pp. 1–6.
- [29] Kyle E. Benson, Georgios Bouloukakis, Casey Grant, Valérie Issarny, Sharad Mehrotra, Ioannis Moscholios, and Nalini Venkatasubramanian, "FireDEX: A prioritized iot data exchange middleware for emergency response," in *Proceedings of the 19th International Middleware Conference*, New York, NY, USA, 2018, Middleware '18, p. 279292, Association for Computing Machinery.
- [30] M. Song, G. Luo, and E. Haihong, "A service discovery system based on zookeeper with priority load balance strategy," in *2016 IEEE International Conference on Network Infrastructure and Digital Content (IC-NIDC)*, 2016, pp. 117–119.
- [31] G. Lewis, S. Echeverra, S. Simanta, B. Bradshaw, and J. Root, "Tactical cloudlets: Moving cloud computing to the edge," in *2014 IEEE Military Communications Conference*, 2014, pp. 1440–1446.
- [32] W. Smith, G. Kuperman, M. Chan, E. Morgan, H. Nguyen, N. Schear, B. Vu, A. Weinert, M. Weyant, and D. Whisman, "Cloud computing in tactical environments," in *MILCOM 2017 - 2017 IEEE Military Communications Conference (MILCOM)*, 2017, pp. 882–887.
- [33] Kubernetes, "Federation - kubernetes," web <https://kubernetes.io/docs/tasks/federation/>, 2019.
- [34] Kimitoshi Takahashi, Kento Aida, Tomoya Tanjo, and Jingtao Sun, "A portable load balancer for kubernetes cluster," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, New York, NY, USA, 2018, HPC Asia 2018, pp. 222–231, ACM.
- [35] Y. S. Hong, J. H. No, and S. Y. Kim, "Dns-based load balancing in distributed web-server systems," in *The Fourth IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, and the Second International Workshop on Collaborative Computing, Integration, and Assurance (SEUS-WCCIA'06)*, 2006, pp. 4 pp.–.
- [36] Zhong Xu, Rong Huang, and L. N. Bhuyan, "Load balancing of dns-based distributed web server systems with page caching," in *Proceedings. Tenth International Conference on Parallel and Distributed Systems, 2004. ICPADS 2004.*, 2004, pp. 587–594.
- [37] M. Colajanni, P. S. Yu, and V. Cardellini, "Dynamic load balancing in geographically distributed heterogeneous web servers," in *Proceedings. 18th International Conference on Distributed Computing Systems (Cat. No.98CB36183)*, 1998, pp. 295–302.
- [38] M. et al Colajanni, "adaptive ttl schemes for load balancing of distributed web servers," *ACM SIGMETRICS Performance Evaluation Review*, 1997.