Particle: Ephemeral Endpoints for Serverless Networking

Shelby Thomas
UC San Diego
shelbyt@ucsd.edu

Lixiang Ao
UC San Diego
liao@eng.ucsd.edu

Geoffrey M. Voelker
UC San Diego
voelker@cs.ucsd.edu

George Porter
UC San Diego
gmporter@cs.ucsd.edu

Abstract

Burst-parallel serverless applications invoke thousands of short-lived distributed functions to complete complex jobs such as data analytics, video encoding, or compilation. While these tasks execute in seconds, starting and configuring the virtual network they rely on is a major bottleneck that can consume up to 84% of total startup time. In this paper we characterize the magnitude of this *network cold start* problem in three popular overlay networks, Docker Swarm, Weave, and Linux Overlay. We focus on *end-to-end startup* time that encompasses both the time to boot a group of containers as well as interconnecting them. Our primary observation is that existing overlay approaches for serverless networking scale poorly in short-lived serverless environments. Based on our findings we develop Particle, a network stack tailored for multi-node serverless overlay networks that optimizes network creation without sacrificing multi-tenancy, generality, or throughput. When integrated into a serverless burst-parallel video processing pipeline, Particle improves application runtime by 2.4–3× over existing overlays.

CCS Concepts

• **Computer systems organization → Cloud computing**.

Keywords

serverless, networking, burst parallel, lambda

SoCC '20, October 19–21, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8137-6/20/10.
https://doi.org/10.1145/3419111.3421275

ACM Reference Format:
Shelby Thomas, Lixiang Ao, Geoffrey M. Voelker, and George Porter. 2020. Particle: Ephemeral Endpoints for Serverless Networking. In *ACM Symposium on Cloud Computing (SoCC '20), October 19–21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3419111.3421275

1 Introduction

Serverless computing offers a high-level computing abstraction within the cloud computing landscape [18]. From a user perspective, it simplifies application deployment since the provider manages a much larger portion of resources including network, OSes, runtimes, and libraries which allows users to focus on their application code. Although providers initially designed serverless platforms to support web and API services, users can now launch thousands of parallel "functions" within a few seconds, dramatically increasing the elasticity of cloud computing resources.

A growing new use of these short-lived functions has been the emergence of "burst-parallel" jobs. Burst-parallel jobs are characterized as parallel tasks with very high fanout consisting of thousands of serverless functions, all deployed by a single user. Fouladi et al. [11] showed how to apply this approach to video encoding, reducing the encoding time for an industry-grade encoder from 149 minutes to 2.6 minutes. Ao et al. [4] applied a similar model to develop a cloud-based burst-parallel system for an end-to-end video processing pipeline that performed 4× better than Spark on video processing jobs. Compared to traditional serverless use cases, a single job consisting of thousands of concurrent functions has unique infrastructure requirements that current serverless platforms do not support efficiently.

In particular, the networking layer underpinning serverless platforms is particularly inefficient for burst-parallel applications. These applications use hundreds of concurrent serverless functions to complete complex tasks and, in lieu of native peer-to-peer networking capabilities on serverless platforms, must coordinate through intermediate storage [4, 10, 11, 20]. This workaround has been widely used thus far,

but is ad-hoc, application-specific, requires additional infrastructure services, and complicates user code. These drawbacks are an impediment to efficiently supporting a wider range of general burst-parallel data analytics, such as the "shuffle" phase in MapReduce-like applications [8, 28], message passing in typical scientific computing applications [12], and vertex traversals in communication graphs of dataflow systems [15, 24, 30].

While none of the major cloud providers today have burst-parallel optimized networking capabilities, enabling this kind of peer-to-peer networking has been of increasing interest in industry [31] and academia [18]. Two workarounds have been proposed, namely NAT hole punching and overlay networking, yet both have versatility and performance drawbacks, as we describe later in this paper.

To demonstrate this point, consider an application built with the Pywren [17, 29] runtime, shown in Figure 1. In this experiment, 100 containers start simultaneously, each performing a basic computation before sending an ack to a leader node when completed. For all three conventional approaches, starting 100 containers takes about 4.8 seconds. This network overhead is 4x longer than the application runtime, or 66–84% of the total serverless startup time; indeed, the network startup overhead can even exceed actual application execution. For burst-parallel applications, fast startup time is critical, much in the same way that conventional applications benefit from fast thread creation.

In this paper we characterize existing network approaches for serverless and propose an optimized network stack, Particle, to reduce network overhead when setting up networks for burst-parallel serverless jobs. Particle's key insight is that the network underpinning a burst-parallel job need not provide isolation between a user's containers, only between containers of different users. This trade-off is similar to why threads are more efficient to create as compared to processes, due to the difference in inter-thread isolation guarantees. We show that Particle can support a number of different serverless frameworks.

To summarize, the contributions of this paper are:

- An evaluation of the challenges of serverless burst-parallel networking with a focus on the network startup problem.
- An evaluation of three different designs to overcome network startup issues.
- A final design, Particle, that enables constant time network creation and startup. We evaluate Particle with microbenchmarks, serverless patterns, burst-parallel applications, multi-tenant settings, and verify no adverse effects on network throughput.

The source code for Particle is available at the following URL: https://github.com/shelbyt/socc20particle.

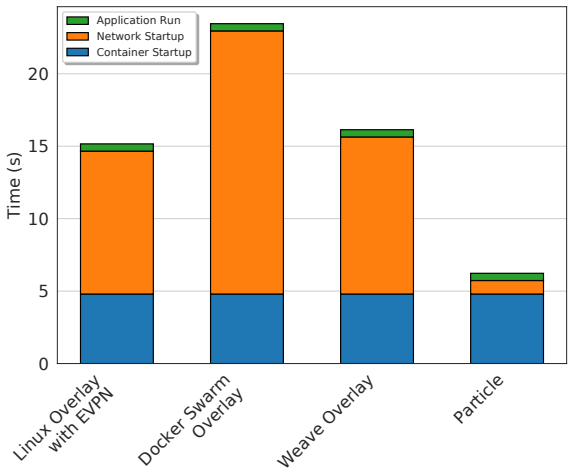

Figure 1: Time to connect 100 virtualized instances to an overlay network. Network start time takes up to 83% of the total startup time. When comparing to Docker Swarm Particle reduces network setup by 32×, reducing end-to-end start time by 3.5×.

2 Background and Motivation

Much effort on serverless systems has focused on container startup time separate from the role of the network for inter-function communication. As serverless evolves from independent single functions to coordinated burst-parallel applications, fast, versatile, and scalable network creation becomes increasingly critical to satisfy the bursty nature of this application class.

VXLAN-based overlay networks such as Weave, Linux Overlay, and Docker Swarm were designed to accommodate the versatility and scalability requirements of modern datacenter networks, but their implementation is tailored to support tens of strictly isolated long-running connections rather than thousands of short-running ones.

We describe the underlying mechanism for how overlay networks are architected today and benchmark each piece of overlay network creation at both the application level and kernel level. Our primary finding is that the overlay data plane interacts with containers in a way that introduces severe latency issues for many VXLAN-based overlays — an issue that is exacerbated when interconnecting hundreds of serverless functions. Fortunately, such a bottleneck provides the opportunity for addressing the problem in a portable and general manner.

Containerization is the most common isolation mechanism in serverless platforms and is used by Google Function, IBM OpenWhisk, and Azure Functions. Therefore we focus the rest of the paper on containers as an execution platform.

Total Connections / Nodes	Connection Time (s)
101 / 1	15.74
404 / 4	15.66
1616 / 16	15.99

Table 1: Scaling Up Nodes: End-to-end startup time remains relatively constant when increasing numbers of nodes while keeping the number of connections per node constant.

Total Connections	Namespace Setup Time (s)
100	10.02
400	38.90
1600	119.79

Table 2: Scaling Up Connections Per Node: In contrast to Table 1, increasing the number of connections/namespaces on a single node scales poorly.

2.1 Overlay Data and Control Planes

The underlying technology that enables overlays is the VXLAN protocol. VXLAN is an encapsulation protocol that wraps packets from a container group with unique identifiers (VNIs) that allow communication without compromising isolation. Devices connected in this way then form an overlay network. An overlay network consists of two distinct parts, the control plane and the data plane. The control plane is an in-network service that exists to manage overlay networks across multiple tenants. These connections are initiated by the data plane within each host. The data plane, unlike the control plane, exists only as long as the serverless application. It is responsible for forwarding data to the correct containers based on VNI, IP, and MAC address.

The VXLAN data plane requires each host to have a VXLAN Tunnel Endpoint (VTEP) that is responsible for VXLAN termination and encapsulation. When a packet is sent from one container to another using VXLAN, the VTEP on the host encapsulates the original Ethernet frame from the container with a VXLAN header. The encapsulated packet in turn is sent out of the host with an outer IP and MAC header. When another container receives the packet, the VTEP on the receiver side looks at the VNI and inner MAC addresses and delivers the payload to the appropriate container.

Overlay networks also require a control plane to manage VTEP routing information. The control plane keeps a mapping of host VTEPs, VNIs, and container MAC addresses. When a container on one host sends data to a container on a different host, the VTEP encapsulates the packet with the VNI and checks locally if the routing information exists. If it does not, the control plane is probed and then the packet is routed with the new route. Control plane implementations are diverse, with some using virtual routers [33], gossip protocols [9], BGP [6], and KVstores [7, 9].

The glue that holds both of these network planes together is the network namespace. The Linux network namespace mechanism creates new logical network stacks in the kernel that each have their own network devices, neighbor and routing tables, `/proc/net` directories, and other network stack state. A network namespace is created by calling `unshare` or `clone` with a `CLONE_NEWNET` flag depending on the implementation. In the context of the overlay, the Virtual Ethernet devices (VETH) are used as endpoints that then connect namespaces together and can be configured to have MAC and IP addresses.

Overheads with respect to the data plane are a function of setting up namespaces, while overheads in the control plane are a function of setting up routes. To understand which of these place a larger burden on the application we perform a scalability analysis for both.

2.2 Performance Bottlenecks

We aim to understand overlay performance bottlenecks without being tied to any specific overlay approach. To this end, we avoid using proprietary software for these microbenchmarks when possible and build our overlay network using native Linux commands to manage the data and control planes. To ground our understanding of system overheads, we carried out a small microbenchmark. This experiment ran on Amazon AWS, using c5.4xlarge machines with Ubuntu 18.04 on Linux kernel 5.0.0-1004. We created a fully functional overlay network using the BGP-based Quagga EVPN [5] as the virtual router and the Linux native `iproute2` v5.2.0 [14] to manage Docker containers and namespaces.

Scaling Node Counts: We first determine whether adding more nodes to an existing overlay network slows down the control plane. To answer this question, we first launched a 100-container cluster and created an overlay network interconnecting the containers on each node. We varied the number of nodes and recorded when new routes were added. Table 1 summarizes our observations, showing the time required to start a Docker container, initiate the data plane, connect to the control plane, and send data to a given receiver node. To increase the load on the BGP-based control plane, we scale up the number of nodes (and thus endpoints that connect to the control plane). The number of containers per host remains the same but the number that needs to be connected increases linearly until 1600 containers are networked. The extra connections are ones between VTEPs. **Takeaway**: On 16 nodes the performance impact from the control plane is negligible and within the margin of error at

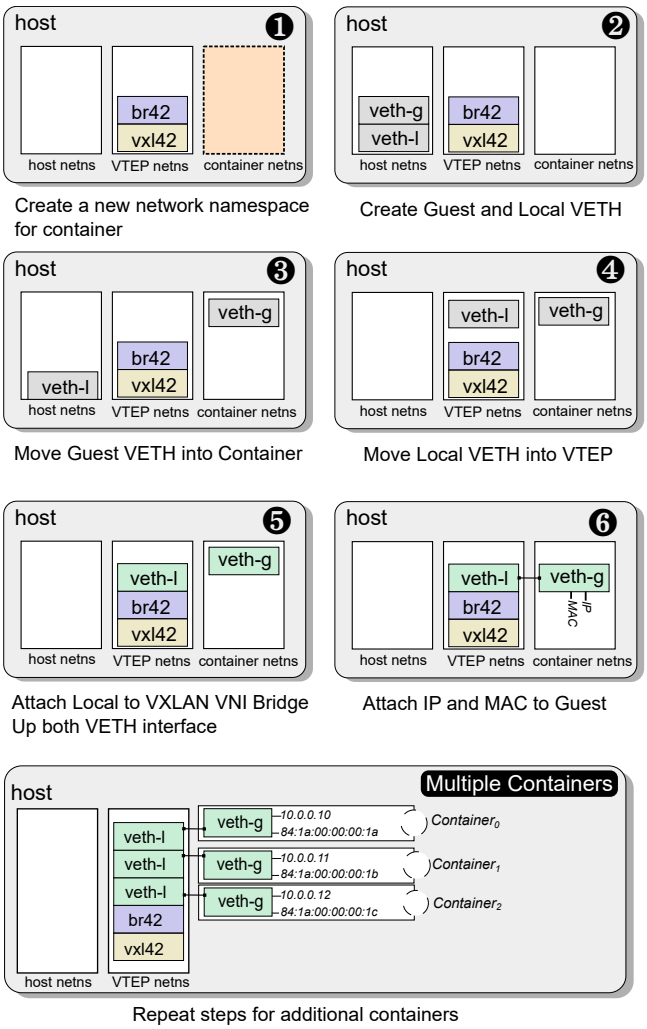

Figure 2: Creating a new network interface for the overlay dataplace involves a sequence of operations that are repeated for each new container. We refer to the network namespace as "netns". Initially only the VTEP, which communicates with the control plane, exists. BR is the bridge interface and VXL is the VXLAN interface attached to the bridge.

this observed scale. We will show that this outcome is not the case for the data plane.

Increasing Connections On A Single Node: Next we characterize the impact on data plane performance and scalability of adding containers to an overlay network. We create a single overlay network on a node and connect it to the control plane, varying the number of network namespaces added to this overlay. We focus our analysis on the overhead of the network namespaces themselves, rather than on container creation time.

Step	Time (s)	Percent of Total
S1	0.10	0.92%
S2	0.10	0.92%
S3	5.18	47.71%
S4	4.77	43.95%
S5	0.49	4.45%
S6	0.22	2.03%

Table 3: Breakdown of the steps in Figure 2 for the overlay data plane for 100 network namespaces. Most time is spent moving VETH devices between namespaces (steps S3 and S4).

Table 2 shows that the overall time increases linearly with the number of namespaces attached to the overlay (unlike what we observed with the control plane). Note that we show only the time to instantiate the network data plane. We measure scalability by varying the number of namespaces attached to this overlay network. We observe that the majority of the time is spent in the kernel.

Takeaway: Table 1 shows that end-to-end startup takes 15.74 seconds with 100 containers on one node. Table 2 shows that most of this time goes to networking the namespace together, more than 60% of the end-to-end startup time. As the number of network namespaces increases, so does the setup penalty. This lack of scalability is a major bottleneck for burst-parallel deployments on serverless.

2.3 The Role of Network Namespaces

Figure 2 illustrates the steps involved in adding a new network namespace to an overlay network. This process is similar for all overlay network software using a VXLAN-based overlay.

Initially the host instantiates a control plane namespace for the VTEP and the host's standard network namespace. First, we create a new guest network namespace. Next, we create a pair of VETH devices in the host namespace. From the host namespace, we place these VETH devices into the network namespace for the control plane and guest namespace. We then tether the local VETH with the VTEP's VXLAN and bridge interface, and turn up the local and guest interfaces. Finally, we establish a connection to the VTEP by setting an IP and MAC for the guest network namespace. At this point the guest is connected to the overlay and all data will transfer through the appropriate VNI. These steps are repeated for each new guest in the overlay network.

We further breakdown data plane creation by instrumenting the steps from Figure 2 with eBPF [22], and report relative and absolute times for connecting 100 network namespaces to the overlay. Table 3 breaks down execution time across

Serverless Networks	Isolated	Low Latency	IP Addressable	L3 Solution	Connection Type
NAT Hole Punching	✓	✗	✗	✓✗	Point-to-point
Kubernetes Pods	✓	-	✗	✗	Port multiplexing and volumes
Docker Host Networking	✗	✓	✓	✓	Direct IP
Overlay Network	✓	✗	✓	✓	Direct IP
Particle	✓	✓	✓	✓	Direct IP

Table 4: Comparison of capabilities and challenges for different serverless networking options: A serverless network solution must be suitable for a bursty low latency multi-tenant environment. Serverless systems must also be able to work with Layer-3 connectivity and provide a per function IP for direct interfunction communication [32]. Today's overlay networks have the appropriate control plane mechanisms for serverless environments but have high network startup latency.

the steps shown in Figure 2. Most of the time is spent in two steps, S3 and S4, and a negligible amount in others. While most of the other steps from Figure 2 either configure a VETH device or create a new one, S3 and S4 are the only ones that perform a *namespace crossing* and move a network interface, the VETH. The local VETH device is moved from the host network namespace into the control plane network namespace and the guest VETH is moved from the host network namespace into the guest network namespace.

Moving VETH devices is inherently expensive because the `dev_change_net_namespace` kernel routine performs a long-running task to ensure that the VETH device is safely moved while holding the `rtnetlink` semaphore. When a move is initiated, the kernel first informs all devices on the notifier chain that the VETH is being unregistered. Next, it removes the VETH device handle from the host namespace and flushes old configurations. Finally, it updates the VETH data structure to point to the new namespace, and informs the namespace and notifier chain that the device is live.

In terms of scalability, when more guests are added to the overlay each of these six steps are repeated, resulting in three different `unshare` calls and two namespace moves per container. This overhead accounts for the linear increase in time in Table 2. A design for a burst-parallel overlay network needs to address both the scalability and performance challenges.

2.4 Challenges of Existing Approaches

Table 4 compares the capabilities and challenges for different serverless networking options. Any serverless networking approach must be suitable for a bursty low latency multi-tenant environment and make minimum assumptions about the network and application layer. Additionally, based on previous work [13, 31, 32], serverless systems must also be able to work with Layer-3 connectivity, as nodes hosting lambda functions are not always Layer-2 adjacent.

Overlay networks are attractive because they fulfill most of the requirements, but current implementations have significant performance overheads. Container orchestrators such as Kubernetes use pods to consolidate containers under a single namespace with one routable IP per pod because of the "one-container-per-pod" commonly-used design pattern [25]. This approach potentially has higher startup latency as each pod starts a container, a network namespace, and a pause container. If we use hundreds of containers per pod to avoid this overhead, each container will need to communicate through application-managed port multiplexing or by creating a volume in the pod for containers to share. From a developer standpoint, changing applications to have port multiplexing logic and manage per-pod databases with related application logic incurs significant engineering costs.

Other alternatives also have significant limitations. Using the Docker host network fundamentally is not a multi-tenant solution, and NAT hole punching requires creating multiple point-to-point connection pairs, none of which are IP addressable. Based on our evaluation of existing approaches, we have designed Particle to satisfy existing serverless requirements, and focus on the ability to quickly generate and interconnect thousands of ephemeral network endpoints.

3 Particle Design

We present Particle, a networking architecture that optimizes network startup in burst-parallel serverless environments. Particle provides an ephemeral dynamically generated pool of IPs at an almost constant startup time. Rather than using memory-intensive caching techniques, Particle creates groups of network endpoints by first separating network creation from other user namespaces, and then optimizes the creation of network endpoints by eliminating serialisation points, batching calls, and consolidating VETH devices while maintaining per-function IPs. In this way, Particle can accelerate network namespace creation without any adverse effect on capability or generality for applications. Particle

addresses the challenges from §2.4 through three design principles:

Match Infrastructure to Application: Burst-parallel applications invoke hundreds to thousands of serverless instances to complete a single complex job. Today's underlying infrastructure is not optimized for the bursty nature of this application class since each serverless task is treated as a stateless independent function. Particle employs techniques to consolidate network infrastructure without compromising generality, programmability, or network versatility.

Generic Socket Interface: Containers that have their IPs allocated by Particle must be able to communicate with each other without the need for any specialized IPC protocol, system, or storage. Accessing third party or network-hosted services must also be possible. Containers must be able to use POSIX socket calls to communicate with each other (§3.1).

Portability: Particle makes minimum assumptions about the system where it is deployed. Porting Particle to additional overlay systems is straightforward as most overlays rely on the default Docker runtime for network provisioning. When integrated, Particle has no adverse effect on throughput.

3.1 Design Space Exploration

In this section we explore three different approaches for optimizing network startup: (1) namespace consolidation, (2) batching, and (3) virtual interface consolidation. We seek to understand the trade-offs in each optimization to inform our final Particle design. In Figure 3 we use microbenchmarks that focus on network creation time to compare the designs, and use the Linux Overlay data plane as the baseline (system configuration details in §5).

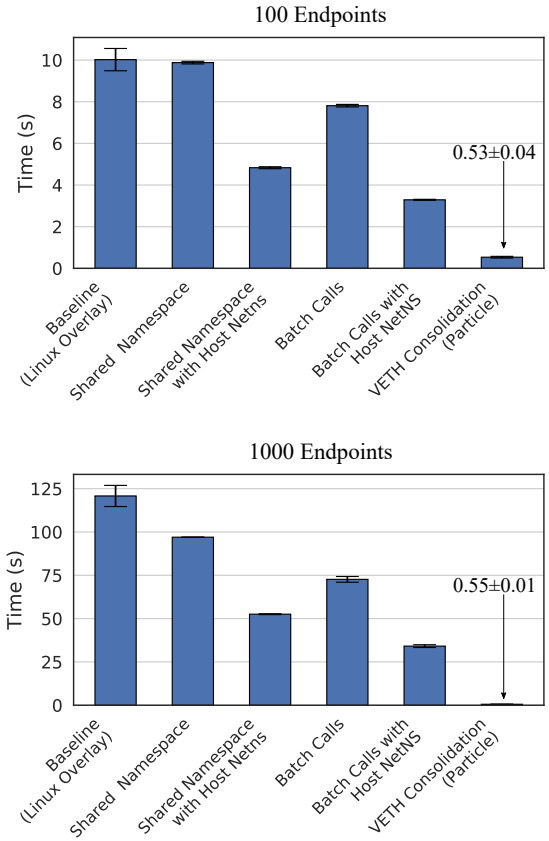

Figure 3: Time to connect 100 and 1000 concurrent network namespaces to an overlay. While the baseline and other optimizations increase linearly with more endpoints, VETH consolidation allows Particle's startup to remain close to constant when scaled up.

Design 1: Namespace Consolidation Based on our findings in §2.3, the network namespace itself is a contributor to high startup latency. One way to address this problem is to adopt what many container orchestrators such as Kubernetes [25] and Amazon Elastic Containers [3] do when co-locating related services under one network namespace and IP. These services perform namespace consolidation to simplify the management plane, but we can extend the traditional 'one-container-per-pod' model to 'many-containers-per-pod' as a performance optimization. This optimization is a natural fit for a burst-parallel environment where many serverless instances work together as part of a single task.

We explore this design by creating a new *root network namespace* for groups of containers, but each container maintains separate kernel namespaces (mnt, pid, ipc, user, cgroup) for other types of isolation. In this way namespaces can also be created for each tenant, while individual containers operate without needing to change assumptions about the environment. Each container is attached to the Particle root namespace and inherits all of its iptables and routing configuration without creating a network namespace itself. Since we want each container to have an addressable and routable IP address (§2.4), we create a VETH interface for each container inside the namespace with an IP and MAC address.

Microbenchmark results in Figure 3 show that this "shared namespace" design has a modest performance benefit when starting 1000 endpoints, but almost no performance impact for 100 endpoints: shared namespaces alone do not address the root issue shown in Table 3. When a new container is created the overlay controller must create a VETH pair for each new container and perform VETH moving. With a shared namespace, the only difference is that, rather than moving the VETH into a separate network namespace per container, the VETH moves into the shared network namespace.

This optimization can be taken a step further if the host namespace can be used rather than an additional shared namespace. Doing so reduces the number of namespaces and

it reduces one VETH move: the host namespace creates a VETH pair and only moves the local end into the VTEP. The trade-off is that this optimization does not match a multi-tenant setting as there is no isolation of the host interfaces.

Design 2: Batching and IP Pooling A major disadvantage with shared namespace consolidation is that, although it takes advantage of the fact that burst-parallel tasks can be consolidated within a single root network namespace, setting up the network namespace itself was still performed iteratively. The advantage to performing namespace consolidation is that it reduces the complexity of managing many namespaces in a burst-parallel environment. In the next design, we push the ideas further by performing a batching optimization to VETH creation inside the network namespace.

With batching, when a burst-parallel request is received, the system creates an IP pool based on a specified IP range and number of containers. Rather than wait until the container is created, all the necessary virtual interfaces for the data plane are created immediately and attached to the root network namespace. Once complete, the system then enters the control plane namespace of the overlay and sets up the corresponding VETH devices and attaches them in batch to the VXLAN port. One key benefit to batching is that it reduces context switching and the number of `unshare` calls. However, implementing batching alone still results in $O(N)$ namespace crossings and VETH movings.

For the same benchmark, Figure 3 shows that batching provides a 22% improvement over a standard Linux Overlay with 100 containers, and improves to 42% with 1000 containers. Although batching improves performance over Linux Overlay and a simple shared namespace, the system still performs many $O(N)$ operations within the namespace, albeit batched. For example, N different VETH devices, MAC addresses, and IP addresses are still being created.

Final Design: Virtual Interface Consolidation The first two designs develop a management plane, the root namespace, and the insight that creating the network, VETH and IPs in batches for a burst-parallel group improves performance. Unfortunately, neither of these designs significantly reduces the total number VETH devices that the system must create. Table 3 shows that regardless of batching and namespace consolidation, each VETH device created incurs an overhead. Additionally, for each container created there are still $O(N)$ VETH devices created and $O(N)$ VETH interfaces moved across namespaces. The first two designs improve performance, but do not address this last issue. As a final design element, we focus on making VETH device creation a constant time operation rather than a linear one. To do so, we create a single VETH device inside the root namespace and attach multiple IPs to this root VETH interface.

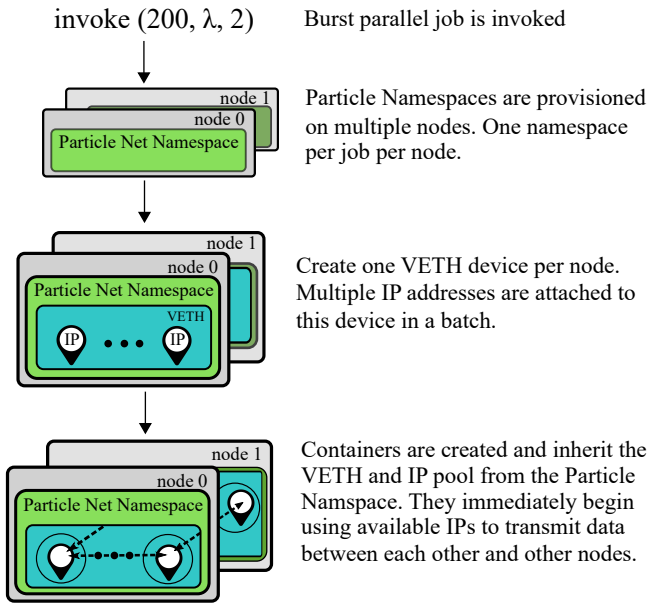

Figure 4: A Particle namespace with containers attached. Only the network namespace is shared among containers of a single application. Each Particle namespace has its own MAC address which is given to the VTEP for routing. Multiple tenants have different Particle namespaces.

In traditional overlay networks there is a one-to-one mapping between VETH device pairs and containers. This mapping is only necessary because each container resides in its own isolated environment. We dispense with per-container network isolation, attaching one VETH pair to the control and data planes. Multiple IPs and MACs are then attached to the root namespace's single root VETH interface. From the perspective of the control plane, all IP addresses attached to a VXLAN interface are routed.

Figure 3 shows that this new one-to-many mapping between VETH interfaces and IP addresses improves performance by an order of magnitude since only one namespace crossing is required for one burst-parallel job. VETH consolidation improves performance by a factor of 17× when creating 100 containers, and 213× with 1000 containers.

The absolute time to start 100 network namespaces is 534ms, and for 1000 network namespaces 553ms. The 534ms comes from two parts, creating a new root network namespace and attaching the overlay. The root namespace starts as a Docker container with only the loopback interface (`–net=none`) which takes on average 300ms. The remaining 234ms is the time to create the root VETH interface, attach it to the control plane network namespace, and add IP addresses.

At a high level Particle systematically replaces expensive *O(N)* "per-container" calls to be *O(1)* "per-job" calls. Based

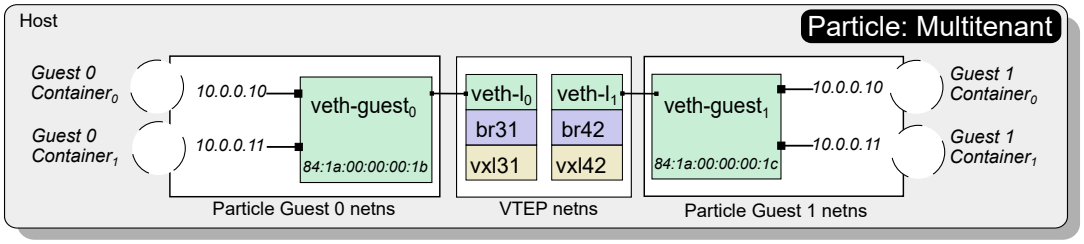

Figure 5: Multitenancy with Particle: Each application has its own Particle namespace to maintain network isolation from other tenants and the host. The control plane is responsible for provisioning extra VNIs in the form of additional VXLAN interfaces for additional tenants. Designing an overlay this way also allows each guest to use any IP address they want without restriction.

on our findings in §2.3 the creation of a container network involves several kernel locks that effectively serialize network creation. This overhead is exacerbated when trying to create hundreds of serverless instances and corresponding network endpoints to coordinate a single job.

Particle is designed to be integrated into existing overlay networks with minimum changes. As described in §2.2, overlay networks consist of a control plane and data plane. While the control plane varies among designs, all of them rely on a similar data plane implementation as shown in Figure 2.

Burst-parallel applications have the property that the logical compute unit is a batch of serverless instances working towards a single goal. As a result, while each container benefits from the standard isolation guarantees (process, file system, etc.), the network interface does not require strict isolation among instances. Particle does, however, still enforce strict network isolation from the host and other tenants.

Figure 4 illustrates Particle's architecture. A single namespace and VETH device are created per tenant per node. Multiple secondary IPs are then attached to the VETH device, creating an ephemeral per-job IP pool. The overlay enables these IPs to be routable through its own policies and mechanisms. Containers can then attach to available IPs and transmit data between containers both intra- and inter-node. When the job completes, Particle removes its namespace and IPs.

Figure 5 shows an example of VETH consolidation in a multi-tenant setting. Each guest has its own Particle namespace with a MAC that is shared with the VTEP. A single VETH interface can host thousands of secondary IP addresses for any container sharing the Particle (root) namespace. Applications have several different options for how to interface with this system.

3.2 Isolation and Application Interface

Consolidating VETH devices and namespaces of multiple containers into one virtual device in one namespace can have side effects for containers within an application. Separate namespaces isolate network resources and provide security isolation. If a container is compromised, other containers in different namespaces are unaffected. Because Particle consolidates network namespaces, it cannot provide the same granularity of security isolation. However, since Particle only consolidates namespaces of the same tenant/application, and different tenants are always isolated by separate namespaces, we consider this tradeoff acceptable for application patterns consisting of multiple serverless instances working together as part of the same application.

Conceptually, applications request different IP addresses for different containers, and Particle assigns those IPs to avoid conflicts. However, if the application in the containers do not respect the assignment, different containers for the same application can interfere with each other by trying to `bind` to the same IP address/port pair. One scenario where this can happen inadvertently is when an application runs multiple containers on the same host and shares a VETH device via Particle. If they try to bind to the same port with `INADDR_ANY`, a port conflict can occur. As a result, applications need to use the IP addresses assigned to them to avoid such conflicts. Rather than relying on the application to use the correct IP address, Particle can interpose by overriding libc's `bind` call (via an LD_PRELOAD mechanism) to ensure that the IP address arguments match the IP addresses assigned to the container. If the application does not use the dynamically-linked libc, or directly calls the `bind` syscall, Linux Seccomp [16] provides a mechanism to enforce the assignment of the IP addresses. Seccomp's filter mode allows specifying what arguments are acceptable for certain syscalls, in this case, assigned IPs as arguments for `bind`.

4 Implementation

Particle is implemented in C and is integrated into the `iproute2` tool included natively in Linux. Particle is not designed to be used directly, rather it is a core module that exists within an overlay system. As a result, Particle does not make any assumptions about what kind of control plane is being used.

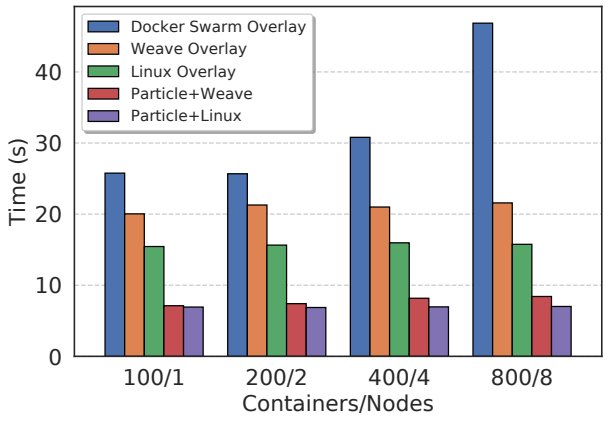

Figure 6: Multi-node Aggregation Application: Total time to start a group of containers, connect to an overlay, and coordinate an all-to-one aggregation job. Particle has no adverse effect on the overlay control plane as more nodes are added.

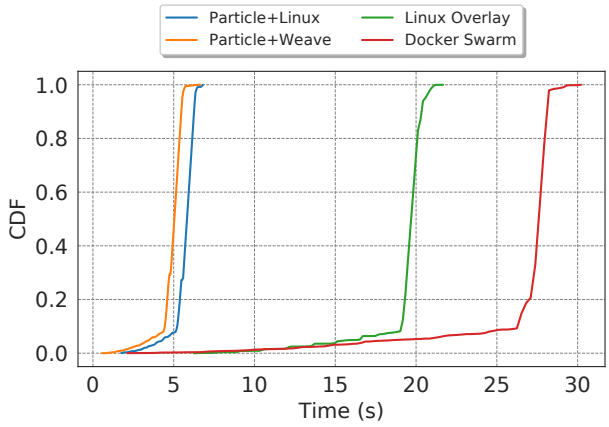

Figure 7: Multi-Node Shuffle Application: CDFs of messages sent in a shuffle application. 100 sender and receivers on two nodes exchange 10,000 messages. Overlays using Particle reduce startup time so application code starts almost immediately.

Porting Particle to an existing overlay network requires an adapter on the control and data plane side. For the data plane, now overlay systems do not need to create a network namespace when provisioning single containers. Additionally, they must create one additional container that is passed into Particle for the group namespace. For overlay systems that use a key-value store, a control plane adapter is required to pass the IPs into the database as a one-time operation.

In our evaluation we integrate Particle's module into the Linux Overlay and Weave Overlay systems. In both cases, these overlays pass in the namespace of the VTEP and must create a new container with no initial network. A pointer to this network namespace is also passed into Particle. At this point Particle has a handle to both a control plane network namespace (VTEP) and data plane network namespace. Based on how many containers are requested, Particle initializes the shared namespace with the same number of IPs. Context is switched back to the existing overlay system which advertises the route to the other members of the control plane based on the Particle MAC address.

5 Evaluation

We first evaluate Particle's startup performance for two communication patterns: aggregation and shuffle [18]. Next we evaluate Particle's performance on a real-world burst-parallel application, Sprocket. Finally, we look at Particle's performance in a multi-tenant setting.

In our experiments we use AWS EC2 `C5.4xlarge` instances, each with 24 vCPUs, 32 GB of memory, and 10 Gb/s network bandwidth. All instances are in the same virtual private cloud (VPC) and placement group for stable network performance. The instances run Ubuntu 18.04.2 LTS using a Linux 5.0.0 kernel with a default configuration. We use iproute2 version 5.2.0, Quagga router version 1.0.0, and Docker version 19.03.1-ce.

By default we launch containers concurrently with an optimal number of threads. We determine the optimal number by trying different values and choosing the one with maximum throughput. Due to the demanding nature of the burst-parallel benchmarks we let the applications themselves determine the number of threads and run the application on 96 core `C5.24xlarge` machines with 192 GB of memory and 25Gb/s bandwidth.

5.1 Serverless Communication Patterns

We evaluate Particle's performance at an application level by measuring its time to complete data aggregation and shuffle jobs. Because our focus is on container startup and network initialization time instead of data transfer rates, we send short synthetic messages in the data aggregation and shuffle jobs. As a result, it provides an upper bound on how fast data aggregation and shuffle jobs can be completed. By the time the tests complete, all containers are started, and all communication paths are established. This experiment is performed on multiple nodes and evaluates Particle against existing systems running the same job. We use Linux Overlay with EVPN, Weave, and Docker Swarm Overlay as comparison points. Each of these systems uses the default overlay configuration without additional parameters.

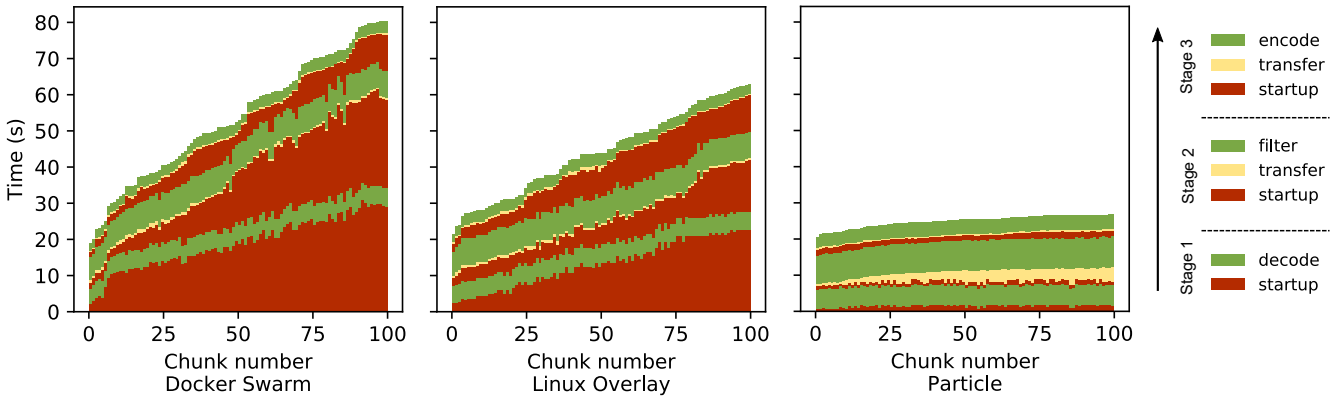

Figure 8: Timeline of Video Processing Pipeline: Comparing Docker Swarm, Linux Overlay, and Particle performance running a Sprocket video processing pipeline. Points on the x-axis represent the processing steps of a single video chunk (decode, filter, and encode), ordered by completion time. Particle eliminates the bottleneck in container startup, and runs 3× and 2.4× faster than Docker Swarm and Linux Overlay, respectively.

Aggregation. The aggregation benchmark tests an all-to-one communication pattern in which many containers send a short TCP message to one receiver container. Once created, sending containers try establishing a TCP connection to the receiving container until success. Figure 6 shows Particle's completion time compared to existing systems. We run 100 containers on each EC2 host, and we vary the number of hosts from 1 to 16 and, hence, the total number of containers from 100 to 1600.

Across multiple nodes Particle's performance is around 7 seconds. Out of this time, starting the 100 Docker containers on each node takes 5–6 seconds and starting Particle takes 500ms. The rest of the time is spent making the TCP connection to the receiver and waiting for the receiver to send back a timestamp message. Once the overhead of starting Docker containers is included, the 17× improvement from the microbenchmark is reduced to a 2–3× improvement.

While the performance of most systems remains constant when increasing the number of nodes, Docker Swarm Overlay increases super linearly when using more than four nodes. The Docker Overlay is a feature-rich control plane implementation that maintains global cluster state using RAFT [26]. In the context of burst-parallel applications many of these features such as load balancing and redundancy are less useful on a per-container context. Rather, they need to be implemented at a per-container-group granularity. Each container group implements its own redundancy protocols within the group [4] and each group needs to be load balanced.

Shuffle. Shuffle is a common communication pattern for data analytics workloads [27]. In this experiment we launch the same number of shuffle sender and receiver containers at the same time. Immediately after launching, senders keep trying to establish TCP connections to all receivers. When established, senders transmit short TCP messages to all the receivers. We use two nodes with 100 containers per node, totaling 10,000 messages sent.

	Startup	Data Transfer	Data Processing
Docker Swarm	69.86%	1.89%	28.25%
Linux Overlay	62.12%	2.50%	35.38%
Particle	17.77%	10.92%	71.31%

Table 5: Video Pipeline Breakdown: Percent of total run time spent in different operations in three networks. We take the average of three runs. Particle spends the most time in actual video data processing.

Figure 7 shows the CDFs of sent shuffle messages using different overlay systems. Our results show that the shuffle application on overlays with Particle (Particle+Linux Overlay and Particle+Weave) outperform existing systems. (Standard Weave is omitted as the bursty nature of the application causes multiple IP conflicts in its control plane, which increased shuffle time significantly.) Using Particle, nearly all of the senders and receivers are able to create TCP connections and exchange messages in 7–8 seconds, while on Docker Overlay and Linux Overlay the time is much larger at 27–30 seconds and 20–22 seconds. These results are also consistent with the results in Figure 6. In both applications, execution time is dominated by network setup.

5.2 Burst-Parallel Video Processing

We evaluate Particle's performance on a real-world burst-parallel application, the Sprocket [4] video processing pipeline.

Sprocket is a serverless system that takes a video as input and first decodes it into frames. These frames are then subject to various transforms such as object detection, facial recognition, or grayscale. After the transforms complete, the frames are finally re-encoded. We ported Sprocket's runtime to run locally and changed its communication module to work with Docker Swarm Overlay, Linux Overlay, and Particle+Linux Overlay.

Each stage (decode, transform, encode) is processed on a different nodes using 100 containers in each stage. The input video consists of 100 one-second video chunks, which are given as input to a first wave of 100 containers started at the same time. Once each video chunk has finished a stage, it signals the downstream service to start a new container and pull data via the overlay network. The process is repeated until the video chunk has passed through three stages.

Figure 8 shows the per-chunk processing timeline of a Sprocket pipeline. Before each pipeline stage, the containers must start and connect to the overlay network so data can be sent to the downstream machines. With Docker Swarm and Linux Overlay, the startup time dominates the overall processing time. Even though each chunk is started simultaneously during the first stage, the network causes a serialization effect that prevents the system from being truly burst-parallel. On subsequent stages, the containers are started on-demand,i.e., as soon as a chunk has finished processing it starts the next stage without a barrier. This freedom causes subsequent stages to take relatively less time as there is reduced contention on the machine.

Particle eliminates the bottleneck in startup so that all containers are started within 2 seconds. As a result, the Sprocket pipeline using Particle is 3× faster than using Docker Swarm and 2.4× faster than Linux Overlay.

The increased data transfer time between the Particle pipeline's decode stage and filter stage is because the elimination of serialization in container startup increases the number of concurrent data transfers. This change shifts the bottleneck to the network, temporarily congesting the network and slowing down the transfer step; in other words, Particle accelerates network startup to the point where network throughput becomes the bottleneck. This effect does not manifest between the filter and encode stages because processing of the decode-filter stage effectively spreads out the data transfer. When using Docker Swarm and Linux Overlay, container startup is much slower, spreading out data transfers between stages and preventing the system from fully utilizing the network.

Table 5 summarizes the percentage of time spent on each stage over three runs. Particle spends most of the time doing data processing, while other overlay networks spend substantial time in the startup stage. Particle's higher proportion of time in data transfer is a result of both reduced overall execution time and the network saturation discussed above.

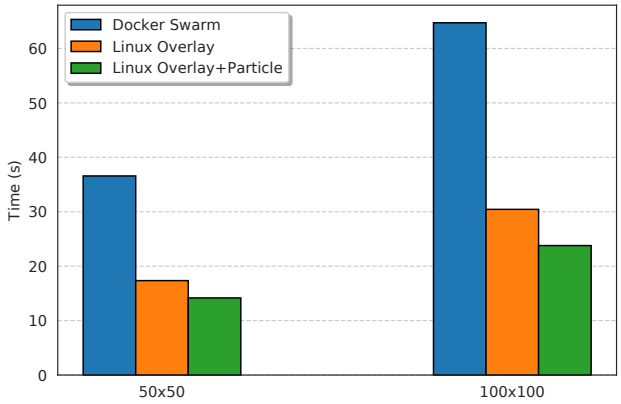

Figure 9: Burst-parallel Sort: In this distributed map-reduce sort, we measure the run time of Docker Swarm, Linux Overlay, and Particle. There are two configurations: 50×50 containers sort 3.2 GB of data, and 100×100 containers sort 6.4GB of data. Particle is 1.22–1.28× faster than Linux Overlay and 2.58–2.72× faster than Docker Swarm.

For a user paying for a serverless burst-parallel service, Particle enables the cost of a job to reflect meaningful work being done rather than infrastructure and setup time.

5.3 Burst-Parallel Sort

We evaluate Particle's performance on a map-reduce sort pipeline. Map-reduce sort has an all-to-all shuffle communication pattern, which is different from Sprocket. After the same number of mapper and reducer containers are started, mappers send different ranges of data values to different reducers, which wait until the completion of all mappers to start running quick-sort. All mappers are scheduled on one node and all reducers on another. Each mapper processes 64MB of data, and each reducer processes about the same amount. We compare the total run time of Docker Swarm, Linux Overlay, and Particle with varying numbers of mappers and reducers.

Figure 9 shows the performance of burst-parallel sort on Docker Swarm Overlay, Linux Overlay, and Particle+Linux using 50×50, and 100×100 containers over two nodes. Particle has the shortest run time in both cases. These results demonstrate Particle's shorter container network setup time has a direct improvement on application performance.

5.4 Multitenancy

Table 6 shows Particle's performance running multiple tenants on multiple nodes and compares it to a multi-node single-tenant setting. The total number of containers per machine is the same and we vary the number of tenants and

Tenants	Containers per Tenant	Runtime per Tenant	Variance per Tenant
1	100	6.93	-
2	50	6.92	0.07
5	20	7.06	0.06

Table 6: Multi-Tenancy Support: Time for a two-node cluster to connect 200 containers with multiple tenants and run the aggregation job. There is little noticeable effect in performance and variance for tenants.

threads per tenant proportionally. For example, when one tenant uses the machine it uses all 10 threads. When 2 tenants use the machine each uses 5 threads, and with 5 tenants each uses 2 threads. The small overhead when having more tenants is from lock contention — like other overlays, Particle holds a lock during network creation as it must `unshare` from the host net namespace to create a Particle namespace — as well as co-location bottlenecks in Docker and in the application.

5.5 Throughput

To verify that Particle does not have any effect on the network datapath, we ran a simple test that creates an overlay for 200 containers on two EC2 hosts. Each host runs 100 containers. We ran `iPerf3` to test network throughput between two randomly-chosen containers that are not on the same host. We found no negative effect on throughput when compared to Linux Overlay without Particle. This result confirms our expectation, as Particle makes no changes to the actual datapath that is responsible for packet transfer.

6 Discussion and Limitations

Multi-Node Scalability. Particle's common use case is to enable serverless networking for burst-parallel functions (containers) that are distributed across multiple hosts. Microbenchmarking showed that the overlay control plane connecting multiple hosts was not a bottleneck. This finding led us to focus on optimizing bottlenecks on each node, and evaluating the effect in a multi-node setting through multiple experiments (§2.2).

As the number of namespaces on a single node increases, namespace setup time increases proportionally. Particle solves both of these problems by reducing namespace setup time regardless of the number of namespaces. For jobs spanning multiple hosts, Particle reduces setup time on each host on which the job runs. Particle enables serverless providers to increase the number of containers per machine without compromising application latency. If we need to interconnect 100 containers for a burst parallel job, the spectrum is 100 containers on 1 machine or 1 container on 100 machines. The choice represents a trade-off between monetary cost and performance. Particle closes the gap between these options and enables a trade-off that improves performance without sacrificing cost.

Application to General Serverless Workloads. Containers are often started on different hosts to reduce load and improve the availability of the serverless functions. Particle is an optimization using overlay networks to address this multi-node case. Figure 2 shows the six steps necessary to setup an overlay network. A management container is not necessary to set up a network between containers on a single host, a bridge will suffice. The advantage of Particle is that it enables users to write programs as if they are still using a bridge, but the containers are available across multiple nodes. Particle is primarily optimized for this multi-node use case.

This paper shows that container overlays are one way that a serverless cloud provider can implement serverless networking. Unfortunately, overlay networks today are not optimized for this use case. With Particle, overlay networks can be created with a negligible amount of overhead on multiple nodes with thousands of serverless functions.

While Particle was motivated by burst-parallel applications, the lessons learned are not limited to it. The experiments show that the network namespace itself is a source of inefficiency in serverless, and a design like Particle can address this issue, achieving the greatest benefits if the VETH and/or namespace can be consolidated. If they cannot, the network namespace may be reused across multiple calls (also reducing cold start at the cost of higher memory usage).

Jobs and Network Namespace Sharing. In this paper we define a job as a single invocation of a computation run by one user. As a Particle namespace is cheap to create, the isolation level can be modified without loss of performance. On one extreme, a Particle namespace can be created for each tenant. In this case jobs that a tenant runs would not be isolated (Figure 5). At the other, every job can have its own Particle namespace that exists just for the job. The design enables providers to choose what isolation model in this spectrum is most appropriate for their use case.

Particle chooses to only relax the isolation of the network namespace to ensure that, if a single function fails, it does not cause a domino effect that corrupts other parts of the system (e.g., the file system) which in turn could cause further function failure.

7 Related Work

Container Orchestrators. Kubernetes and Amazon Elastic Containers use shared namespaces to simplify service management between shared jobs in a pod or task group. However, employing this method alone does not appreciably change startup latency, as discussed in §2.4 and §3.1.

Communication Alternatives. Pocket [20] is an intermediate storage layer for burst parallelism that employs multiple storage media (e.g., a Redis key-value store, AWS S3, etc.) to accommodate different workloads in a cost-efficient way. Locus [27] focuses on shuffle performance in burst-parallel applications. It uses a performance model to select the appropriate storage medium in the cloud. SAND [2] proposes a message queue approach for inter-container communication. While these systems improve on existing communication mechanisms, they incur extra infrastructure costs and lack the generality of a direct communication mechanism. Shredder [34] takes a completely different approach by performing compute directly inside storage nodes.

Alternative Virtualization Layers. Particle focuses on optimizing network startup for container-based serverless systems since containers are a dominant virtualization platform. However, Kata [19] and Firecracker [1] have proposed an alternate serverless virtualization architecture using microVMs. These microVMs employ TUN-TAP devices to build an overlay network rather than network namespaces, and therefore represent an entirely different approach to networking. As a result, evaluating and optimizing network startup and configuration in these architectures is an interesting open question.

Container Network Setup. Mohan et al. [23] identify that network creation and initialization account for the majority of latency in bursty container creation. They extend the idea of Pause containers [21] to pre-create network namespaces that can later be attached to new containers. This technique is effective but it introduces security issues in a multi-tenant setting as new containers are reusing cached network namespaces. Additionally, the caching overhead is linear with the number of namespaces, i.e., memory usage increases with more containers attached to the network. Particle only needs to create a single network namespace for a group of containers, making it faster and more memory efficient than the caching technique.

8 Conclusion

As serverless evolves to accommodate next generation applications such as burst-parallel, we need to reconsider long held notions about serverless design patterns. We take for granted that a long-running application will amortize costs for certain one-time operations, such as setting up infrastructure but in serverless burst-parallel, these one-time operations are repeated hundreds of times and the cost is paid on each invocation. In this paper we focused on a key bottleneck for burst-parallel applications, network startup time. We found that provisioning the network can be a significant portion of execution time. We closely examined the overheads in establishing connectivity among containers in overlay networks and designed a system, Particle, to address these issues. Particle maintains serverless application requirements of generality, versatility, and multitenancy while providing near constant network startup time on single and multi-node deployments. We show that in these scenarios Particle improves total application runtime by at least a factor of two over existing solutions. Particle shows that it is possible to enable serverless networking in multi-node deployments without compromising speed.

9 Acknowledgements

This work is supported by the National Science Foundation through grants CNS-1564185, CNS-1629973, CNS-1553490, and CNS-1763260 as well as a generous gift from Google, Inc. We would like to thank Tim Wagner for advice on this work as well as the anonymous SoCC reviewers for their useful feedback. We are also very grateful to Cindy Moore for managing software and systems used in this project.

References

[1] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA, February 2020. USENIX Association.

[2] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-performance Serverless Computing. In *Proceedings of the 2018 USENIX Annual Technical Conference*, USENIX ATC'18, pages 923–935, Berkeley, CA, USA, 2018. USENIX Association.

[3] Amazon Web Services. Amazon Elastic Container Service. https://aws.amazon.com/ecs/, 2020.

[4] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC'18, pages 263–274. ACM, 2018.

[5] ARM. Quagga Routing Suite. https://www.nongnu.org/quagga/, 2018.

[6] Calico. Calico. https://www.projectcalico.org/, 2019.

[7] CoreOS. Flannel. https://coreos.com/flannel, 2019.

[8] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[9] Docker. Docker. https://www.docker.com/, 2017.

[10] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *Proceedings of the 2019 USENIX Annual Technical Conference*, USENIX ATC'19, pages 475–488, 2019.

[11] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, Fast and Slow: Low-Latency Video Processing using Thousands of Tiny Threads. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'17, pages 363–376, 2017.

[12] Richard L Graham, Timothy S Woodall, and Jeffrey M Squyres. Open MPI: A Flexible High Performance MPI. In *Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics*, PPAM'05, pages 228–239. Springer, 2005.

[13] IBM. OpenWhisk Github. https://github.com/apache/openwhisk, 2020.
[14] iproute2. Iproute2 routing commands. https://git.kernel.org/pub/scm/network/iproute2/iproute2.git, 2019.
[15] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM European Conference on Computer Systems*, EuroSys 07, pages 59–72, Lisbon, Portugal, 2007. ACM.
[16] Jake Edge. A seccomp overview. https://lwn.net/Articles/656307/, 2015.
[17] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the Cloud: Distributed Computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC'17, pages 445–451, New York, NY, USA, 2017. ACM.
[18] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv preprint arXiv:1902.03383*, 2019.
[19] Kata Containers. Kata Containers. https://katacontainers.io/, 2020.
[20] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI 18, pages 427–444, 2018.
[21] Ian Lewis. The Almighty Pause Container. https://www.ianlewis.org/en/almighty-pause- container, October 2017.
[22] Matt Fleming. A thorough introduction to eBPF. https://lwn.net/Articles/740157/, 2017.
[23] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile Cold Starts for Scalable Serverless. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing*, HotCloud'19, 2019.
[24] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP'13, pages 439–455, New York, NY, USA, 2013. ACM.
[25] Official Kubernetes. Pod Overview. https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/, 2020.
[26] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.
[27] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI 19, pages 193–206, 2019.
[28] Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat. TritonSort: A Balanced Large-scale Sorting System. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 29–42, Berkeley, CA, USA, 2011. USENIX Association.
[29] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. Serverless Data Analytics in the IBM Cloud. In *Proceedings of the 19th International Middleware Conference Industry*, Middleware '18, pages 1–8, New York, NY, USA, 2018. ACM.
[30] Apache Spark. http://spark.apache.org/.
[31] Tim Wagner. Serverless Networking is the next step in the evolution of serverless. https://bit.ly/30kFoY9, 2019.
[32] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking Behind the Curtains of Serverless Platforms. In *Proceedings of the 2018 USENIX Annual Technical Conference*, USENIX ATC'18, pages 133–145, Boston, MA, July 2018. USENIX Association.
[33] Simple, resilient multi-host containers networking and more. https://github.com/weaveworks/weave.
[34] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 1–12, New York, NY, USA, 2019. Association for Computing Machinery.