PhysarumSM: P2P Service Discovery and Allocation in Dynamic Edge Networks

Thomas Lin, Weiyu Zhao, Ivan Co, Andrew Chen, Henry Xu, and Alberto Leon-Garcia
Department of Electrical and Computer Engineering
University of Toronto, Toronto, ON, Canada
{t.lin, weiyu.zhao, ivan.co, andrewchiachun.chen, henryg.xu}@mail.utoronto.ca, alberto.leongarcia@utoronto.ca

***Abstract*—With the continuous growth in the number of mobile networked devices, and their rapidly improving compute capabilities, it has become possible to harness them as an extended cloud. This presents a clear opportunity to place latency-sensitive applications and services at the edge. As applications are increasingly based on the microservices and Network Function Virtualization (NFV) architectures, their overall performance will depend on the location of their constituent microservices relative to one-another. An extended cloud comprising mobile devices therefore results in a dynamic network, making it difficult for traditional orchestration systems in distant clouds to perform timely management and replacement of microservices to ensure the overall application or service is performant. We propose to address this challenge by decentralizing the service discovery and allocation logic, placing it in client microservices. This paper presents a P2P-based design and prototype system that empowers clients to discover desired services based on pre-defined QoS requirements. If none are found, clients identify compute nodes meeting the requirements to request a new service allocation.**

I. Introduction

In recent years, innovation in the networking and cloud computing space has been driven by *softwarization*, which realizes software implementations of logic that have traditionally been performed in hardware. Simultaneously, the proliferation of cloud computing spurred the adoption and growth of the microservices architecture, breaking traditional monolithic applications into separate, distributed, inter-communicating software processes known as *microservices*. The growth of softwarization and cloud computing inspired the Network Function Virtualization (NFV) architecture, which orchestrates blocks of simple virtualized network functions (VNFs) into *service chains* to create more complex network functions [1]. In the near future, an increasing number of devices will be embedded with networked computers (i.e. "smart" devices), thus expanding the IoT/edge computing tier [2]. As computing power grows in embedded smart devices, it becomes possible to virtualize them and place network functions and services closer to end-users, enabling ubiquitous computing and complementing Multi-access Edge Computing (MEC) [3].

The explosive growth of virtualizable devices at the edge and fog tiers presents a scalability challenge for orchestrating services and applications that rely on centralized management residing far away in traditional clouds. Complicating matters is the fact that these devices may be mobile (connecting, disconnecting, and changing locations), affecting the connectivity graph, bandwidth, and latency of links. This dynamic environment may negatively impact the performance of service chains and applications based on the microservices architecture. The problem becomes critical if said services and applications partake in demanding industrial settings, power grids, traffic management, etc. Thus, there exists a clear need for adaptive scheduling in a continuously changing network topology, replacing service instances as required to maintain the performance requirements of constituent microservices.

In our past work [4], we introduced an architecture and system that incorporated network metrics, e.g. latency and bandwidth, into the scaling and scheduling decisions of cloud management frameworks. Specifically, the system monitored the service latency and bandwidth as *perceived by clients*, and performed service scaling and placement to ensure the quality-of-service (QoS) per client. A Software-Defined Networking (SDN) solution was then invoked to re-direct the client's traffic to the new instance. The system thus provided service assurance in a logically centralized manner. This led us to ask: how can we continue to provide assurance in scenarios where the connectivity to centralized management services are disrupted (e.g. network congestion, natural disasters, etc.)?

In this work, we take a decentralized approach where the discovery and allocation of services are made by the clients. We argue that the clients are best positioned to detect when their received service quality has degraded (e.g. loss of available bandwidth, increased latency, service failure, etc.), and switch to another service instance, or trigger the allocation of a new instance on a nearby compute node (i.e. server that hosts virtualized services). Composite applications and services can thus react more quickly to network and service impairments, triggering restorative operations without the intervention of possibly distant centralized orchestrators that may not even be reachable. To this end, we leverage the cooperative, fault-tolerant, and self-organizing features of peer-to-peer (P2P) overlays to provide the necessary scalability, high-availability, and mobility support for services and applications in future edge/fog tiers and mobile ad-hoc networks [5].

This paper presents our proposed architecture and prototype implementation, PhysarumSM, of a decentralized P2P service mesh that empowers clients to discover and allocate service instances they need to communicate with. A P2P overlay presents an underlay-agnostic solution while enabling clients to perform automatic discovery of nearby service instances. A request sent to a service may automatically trigger a service instance allocation if no pre-existing nearby instance is discovered, or if the discovered instances do not meet pre-defined QoS requirements.

The rest of this paper is organized as follows. Section II provides a review of background and related works. In section III, we define and explain the general requirements for our envisioned system. Section IV then describes the high-level architecture of the system, followed by a presentation of our proof-of-concept implementation in section V. Evaluation of the prototype is then shared in section VI, and we conclude the paper in section VII.

II. Background and Related Work

This section presents a brief overview of background and related works to provide further context for the work introduced in this paper.

A. Service Meshes

Service meshes have become a popular for addressing the connectivity challenges introduced by the microservices architecture. They function as networking middleware, facilitating inter-service communications. Some common goals of service meshes are to provide: 1) service discovery; 2) load balancing; 3) fault tolerance; 4) traffic and health monitoring; 5) circuit breaking; and 6) access control [6].

Istio [7] is a currently popular open-source service mesh implementation built around the open-source Envoy [8] software proxy. The key capabilities of Istio include traffic management, secure communications, and monitoring the service mesh's performance and health. Its architecture consists of a centralized control plane, which is responsible for configuring the service mesh, and a data plane, which is composed of distributed Envoy proxies that forward and route the network traffic within the mesh. Istio injects a local proxy alongside each microservice in the mesh, intercepting all traffic to and from the service. This deployment pattern is often called the *sidecar pattern*. By itself, Istio does not perform service discovery, and instead relies on an external entity to update the Istio service registry with information regarding the set of services and their endpoints.

Consul [9] is another popular open-source service mesh solution. It features a built-in service discovery mechanism and utilizes a simple gossip protocol [10] to help disseminate membership information. It supports network segmentation and delegation of administrative privileges, enabling the service mesh itself to be virtualized. Consul maintains a logically centralized state in a distributed fashion by utilizing the Raft consensus protocol [11]. To provide connectivity services, it also uses the sidecar pattern to place proxies next to services.

B. Kubernetes Approach

Kubernetes [12], abbreviated as *K8s*, is a popular enabling technology for the deployment of applications built around the microservices architecture. Many of the open-source service mesh implementations that exist are built as K8s network plugins to enable quick adoption. Prior to service meshes, K8s still provided some basic service mesh functionality by orchestrating and manipulating Linux iptables, ebtables, tc, and routing tables. This complex combination built around legacy tools enabled basic connectivity, but created difficulties in debugging and tracing of errors when applications fail to function properly. Kubernetes has its own set of management services that comprise its control plane, typically deployed within a controller server, and utilizes a centralized datastore to maintain its state. Connectivity loss to either the control plane services or the datastore results in hung operations or a corrupted network state.

C. P2P in Edge Clouds

While P2P technologies have typically been used for content-based discovery, they have also long been proposed as a means for discovery within service-oriented environments [13], [14], [15]. As the concept of edge cloud computing becomes a reality, there has been renewed interest in using P2P. In [16] and [17], the authors proposed a P2P network comprised of IoT gateways that would enable applications to discover IoT resources in a federated domain. The authors of [17] demonstrated a working prototype system and evaluated it on an OpenStack-based cloud, demonstrating access and lookup latencies in the order of milliseconds.

In dynamic and volatile environments (e.g. disaster zones) where connectivity to traditional edge processing clouds may be disrupted, the authors of [18] used P2P as a fallback mechanism for discovering neighbouring resources to enable collaborative processing of data. In [19] and [20], the authors investigated the scheduling challenges in dynamic topologies with numerous mobile edge servers. They proposed algorithms for choosing candidate edge servers for offloading services in a decentralized fashion, the former taking task deadlines into account, and the latter taking node reliability into account. Our work serves to be an enabling framework for the scenarios and algorithms presented in these works.

In [21], we experimented with leveraging P2P in our own distributed edge clouds to enable a self-healing SDN data plane. When SDN controllers go down, the affected SDN switches that lose connectivity will automatically discover the nearest available controller, taking into account its version and compatibility, and connect to it.

III. Requirements

As we begin designing a system that enables clients to discover and allocate service instances, we first discuss and review the set of functional requirements and objectives.

Decentralized Discovery: Decentralizing service discovery enables clients to independently discover nearby service instances they wish to connect to. When link failures or routing table failures cut off clients from the core network, traditional centralized discovery mechanisms may fail, causing applications and services to malfunction. We note that a system can easily support centralized and decentralized methods, with one acting as the fallback for the other.

Client-Initiated Allocation: When a client's received quality from a given service is degraded (e.g. due to network latency, or increased demand and load from other clients, etc.), or predicted to degrade in the near future (i.e. due to trending or seasonality patterns), the service should be quickly scaled. Such scaling operations may be vertical or horizontal. In the horizontal case, the scheduling system should be client-aware, so that it takes into account the location and demand of existing clients and strategically places the new instance in a location that decreases the load on the existing instance(s) while potentially improving the QoS of one or more clients.

Monitoring of Infrastructure and Services: The system should be able to monitor the health of the underlying infrastructure, the services it hosts, as well as the QoS of the clients that use those services. Infrastructure information, such as compute and network metrics, enables the system to perform optimized placement of services based on the service's requirement constraints (e.g. RAM, CPU, latency, bandwidth, etc.). Additionally, collecting data and statistics on the applications and services will enable us to calculate their key performance indicators (KPIs), as well as see how changes to the underlying infrastructure can affect service and application performance. This information (raw or preprocessed) should be available to clients for use when they need to perform an allocation.

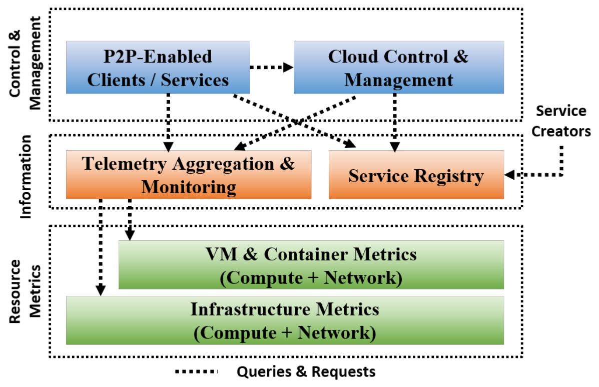
Security: As in any cloud environment, the system should provide security against compromised hosts and listeners on the wire. Common measures should be utilized, such as end-to-end traffic encryption to avoid traffic inspection, access control and authentication to ensure only authorized peers can communicate, and hashing of service images to ensure they have not been modified.

IV. Architectural Design

We now present a high-level architecture that aims to achieve a scalable, highly-available, and mobile-friendly service mesh. Our design leverages an underlying P2P substrate, comprised of a distributed hash table (DHT), that enables clients to discover nearby instances of a particular service based on hash identifiers. This approach implicitly supports mobility of compute nodes that host virtualized services.

A. System Overview

Figure 1 presents a generalized high-level architecture of our system. The architecture has three main tiers: a resource metrics tier at the bottom, an information tier in the middle, and a control & management tier at the top. The bottom tier consists of the physical infrastructure (e.g. compute, network, storage, wireless, etc.), and any virtualized versions of those resources, and is able to provide their utilization and performance metrics. The middle tier comprises telemetry and monitoring for aggregating and analyzing information from the bottom tier, as well as a service registry that contains the metadata of services that can be deployed on the infrastructure. Finally, the top tier contains any components that partake in



Control & Management
P2P-Enabled Clients / Services
Cloud Control & Management
Service Creators
Information
Telemetry Aggregation & Monitoring
Service Registry
Resource Metrics
VM & Container Metrics (Compute + Network)
Infrastructure Metrics (Compute + Network)
Queries & Requests

Fig. 1: Generalized high-level architecture.

the logical decision-making processes concerning the control and management of the infrastructure, as well as the services that run atop the infrastructure.

In accordance with our view that the P2P-enabled clients (which themselves may be microservices) have a say in the discovery and allocation process, we have placed the clients and services within the control & management tier. The clients and services have access to the monitoring information as well as the service registry, which allows them to make resource allocation and life cycle decisions, and to request the cloud control & management services to execute these decisions.

B. Telemetry, Monitoring, and Service Registry

The telemetry aggregation and monitoring component, within the information tier shown in Fig. 1, is responsible for ingesting, aggregating, processing, analyzing, and storing the raw data that is received from the underlying tier. The raw data received from the bottom tier contains metrics relating to the CPU, memory, disk I/O, and bandwidth utilization of wired and wireless networks. Processing and analysis of such data yields higher-level information such as KPIs of resources, services, and applications. A key expectation is for the monitoring system to be cloud-aware, meaning that it is able to discern whether a specific resource is physical or virtual, and map a given virtual resource to a location within the underlying physical topology. Users and clients querying such information should also be able to directly map metrics and information to a given resource (physical or virtual) or set of resources that comprise an application.

The service registry component, also within the information tier, contains metadata regarding the services that may be deployed. This component allows application and service creators to register new services or update existing ones. Each service entry should contain:

- A service name and associated service hash;
- A link to fetch the service image, possibly along with versioning information;
- Information regarding the network QoS parameters (e.g. minimum bandwidth requirement, maximum latency tolerance, jitter tolerance, etc.) required for clients that utilize the service to properly receive it. This allows clients

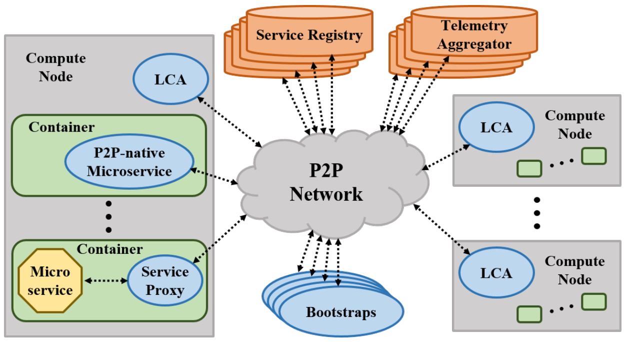


Fig. 2: System-level diagram of PhysarumSM.

communicating with the service instances to realize when the quality they are receiving or experiencing degrades;

- The compute requirements (e.g. RAM, CPU, disk space) of the service, so that the scheduling and allocation logic can identify a server with sufficient capacity to host the service; and finally
- A flag indicating whether the service is stateless or stateful. In the event that a client's received service quality degrades, it may not be able to switch to a different instance if the service is stateful. An example of a stateful service is streaming compression, whereas an example of a stateless service is a DNS query. Clients should be conscious of this distinction when deciding to switch services mid-stream or mid-job.

V. Proof-of-Concept Implementation

Figure 2 presents the system-level diagram of PhysarumSM[1], our proof-of-concept implementation, where each component connected to the P2P network is a node participating in the DHT. The colours of each component correspond to the tiers from Fig. 1. The P2P network enables discovery and communication between bootstrap nodes (initial peers for new P2P nodes joining the network), Life Cycle Agents (LCAs) (per-compute node daemon that controls the life cycle of virtualized resources), service registry and telemetry aggregator instances, and containerized microservices (either native P2P, or legacy TCP/IP packaged with a P2P proxy). We begin by describing the environment upon which our work was implemented and deployed on, followed by descriptions of each component in more detail.

A. Implementation Environment

Our prototype was developed, implemented, and deployed on the SAVI testbed [22], a Canada-based multi-region and multi-tenant academic cloud. SAVI features a fully SDN-enabled network using OpenFlow [23] switches (both physical and software), virtual machines (VMs) and containers based on the OpenStack ecosystem, and heterogeneous resources such as GPUs, FPGAs, and SDRs. SAVI's multi-tier topology includes VM-heavy core nodes, with multiple smaller edge nodes distributed across different university campuses.

[1]PhysarumSM: https://github.com/PhysarumSM

The core networking features of PhysarumSM is implemented using *libp2p*[2] v0.9.2, an open-source modular networking stack that allows users to compose highly configurable P2P services and applications. libp2p is a quickly evolving project, with a sizable community contributing to its development. More importantly, it places an emphasis on security, encrypting all P2P connections by default. Since libp2p is written in multiple languages, it enables P2P services implemented in different languages to communicate with each other. PhysarumSM is fully implemented using the Golang version of the library. As we deployed our prototype on multiple regions of the SAVI testbed, we are able to test and experiment with multi-tier applications.

B. Telemetry Aggregation & Monitoring

To aggregate raw telemetry collected from the infrastructure and services, we use the open-source Prometheus[3] database. Prometheus scrapes metrics collected from various data collection agents known as *exporters*. To collect hardware and OS statistics from our various compute nodes, we utilize Node Exporter, an open-source agent binary that is part of the Prometheus ecosystem. Using Prometheus' client libraries, we have also instrumented all the critical P2P services in the system (the ping monitor (bootstrap), the service registry, the life cycle agent (LCA), and the service proxy), so that they can export metrics to Prometheus. From these, we collect software-level metrics such as CPU utilization, memory utilization, number of threads, etc. Additionally, we can observe service-level metrics such as traffic bandwidth, the number of clients, and who those clients are. This provides comprehensive visibility into the overall state of the system.

In Fig. 2, we show the telemetry aggregator to be connected to the P2P network. This is because we can couple Prometheus with a P2P-enabled proxy that allows it to dynamically discover and scrape from the other P2P services. This simplifies the operation of Prometheus as its configuration does not have to be dynamically updated whenever new services join or existing services leave the network.

C. Bootstrap Node(s) / Ping Monitors

Initial well-known bootstrap addresses are required for many P2P systems. Bootstrap nodes serve as the initial peers of newly joined nodes, allowing them to discover other nodes participating in the network. Their role in the network naturally leads them to become nodes with a large number of directly connected peers, and thus most suitable to perform routing within the DHT. While libp2p supports a broadcast-based discovery system using multicast DNS (mDNS) [24] to discover initial peers, the SAVI testbed's current network policies forbid Ethernet broadcast and multicast frames.

The bootstraps serve a secondary purpose. Due to their highly-connected nature, they are ideally situated to serve as latency monitoring points. They periodically ping their directly connected peers to collect round-trip time (RTT)

[2]libp2p: https://libp2p.io/
[3]Prometheus: https://prometheus.io/

latency information. By strategically placing instances of these bootstrap monitoring nodes in key locations in the underlay, such as gateway routers where traffic must traverse through to reach peers in other networks, we can measure the latency of peers to well-known points and build a latency graph for the P2P network. We expose this information to Prometheus, allowing it to scrape the latency data.

D. *Service Registry*

As mentioned in subsection IV-B, the registry stores metadata regarding registered services. For each service, the current implementation stores a link to its container image, its compute requirements, and the network QoS requirements for its clients. This information plays a critical role when deciding the placement of services in relation to the clients. In [4], we previously introduced a client-aware QoS autoscaling and scheduling system that uses the QoS information in this registry to: 1) configure monitoring alarms that track the client's traffic latency and bandwidth; and 2) make scheduling decisions when a new instance is launched, placing it at a location that best meets the clients' needs.

In this work, we extend the role of the service registry to serve as a lookup service between human-readable service names and service hashes, thus playing a role similar to DNS within the traditional Internet architecture. A service hash is a unique identifier calculated based on a service's container image, thus preventing modified images from producing the same hash while simultaneously providing an implicit form of versioning. The mapping between human-readable names and service hashes allows developers of applications or services to simply specify the name of service they wish to connect to, which is then resolved to a service hash for querying the DHT. Applications and services can hard-code the hashes of services they communicate with if they wish to avoid potential incompatibilities in case the human-readable name-to-hash mapping is modified (e.g. a new version of the service is released), or if the registry becomes compromised.

The registry itself is implemented as a P2P-enabled service with a well-known service hash. This allows clients wishing to query the registry to discover the closest instance and direct its query to it. The service registry can thus be distributed across different regions and cloud tiers to offer redundancy and to reduce the access latency. It synchronizes its state with other registry instances by using etcd[4] under-the-hood, an open-source distributed key-value store.

E. *Life Cycle Agents*

The LCAs are per-compute node instances responsible for managing the life cycle of virtualized compute resources. Like the service registry, the LCA is implemented as a P2P-enabled service with a well-known service hash. The LCA leverages a virtualization driver to create, update, and delete virtualized compute resources. We currently use Docker[5] as our main virtualization engine, and so the LCA utilizes the Docker client as the virtualization driver. Each LCA monitors the service instances it creates by collecting data regarding their usage via a dedicated metrics channel offered by the service proxies. If the LCA determines that a proxy has not been used for a period of time, it retires the service instance by deleting it. In the current implementation, any P2P node can ask an LCA to spawn an instance of a given service; addressing access control policies is planned in future work.

The LCA proactively caches service images via a local cache, thus introducing the possibility of having *cold standbys* for important services. This avoids the need for an LCA to fetch and download the image when a service allocation is requested. In a similar vein, we can allocate service instances in a non-running state, creating *warm standbys*. We foresee these standby modes to be useful for enabling applications and services to continue operations in challenging environments with intermittent Internet connectivity.

F. *Service Proxies*

While P2P-enabled services can be natively built using our P2P library, in order to support compatibility with traditional TCP/IP services, we introduce a P2P-enabled proxy that can be used by TCP/IP clients and services alike to communicate with other services over the P2P network. Services wishing to advertise themselves can be represented by a P2P proxy in a manner following the sidecar deployment pattern. Clients wishing to access services over the P2P network can also connect through the proxy.

Our proxy allows clients to communicate with other services by tunneling their traffic over encrypted P2P connections. The proxy takes care of discovering the closest available instance of a target service that meets the service's QoS requirements (as defined in the service registry), and if none exists, the proxy will take care of requesting a new allocation of a service instance that meets the requirements. We currently support proxying HTTP, TCP, and UDP traffic. Additionally, clients can also request access through a P2P-based service chain. The proxy will take care of setting up the series of VNFs along the path, allocating new service instances if necessary.

The service(s) to connect to, and which type of traffic to proxy, are configurable at run-time via an HTTP-based management API. As shown in Fig. 3, the proxy opens local TCP/IP sockets that are mapped to services in the P2P network. Note that if a specific instance goes down, the socket does not close, as the proxy will automatically discover an alternative instance or allocate one if need be.

We now provide a brief description of the other proxy subcomponents shown in Fig. 3:

In-Memory Caching: The service proxies contain two in-memory caches: a peer cache, and a registry cache. The peer cache is used to store a set of peers representing instances of a given service. Each peer in the peer cache is periodically pinged to ensure liveliness and that they meet pre-defined QoS latency requirements, with non-conformant peers being removed. Additionally, the act of pinging also allows the proxy

[4]etcd: https://etcd.io/

[5]Docker: https://www.docker.com/

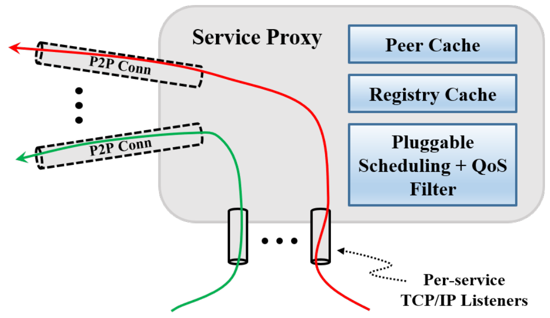

Service Proxy
Peer Cache
Registry Cache
Pluggable Scheduling + QoS Filter
P2P Conn
P2P Conn
Per-service TCP/IP Listeners

Fig. 3: Design of P2P service proxy to enable interoperability with traditional TCP/IP clients and services.

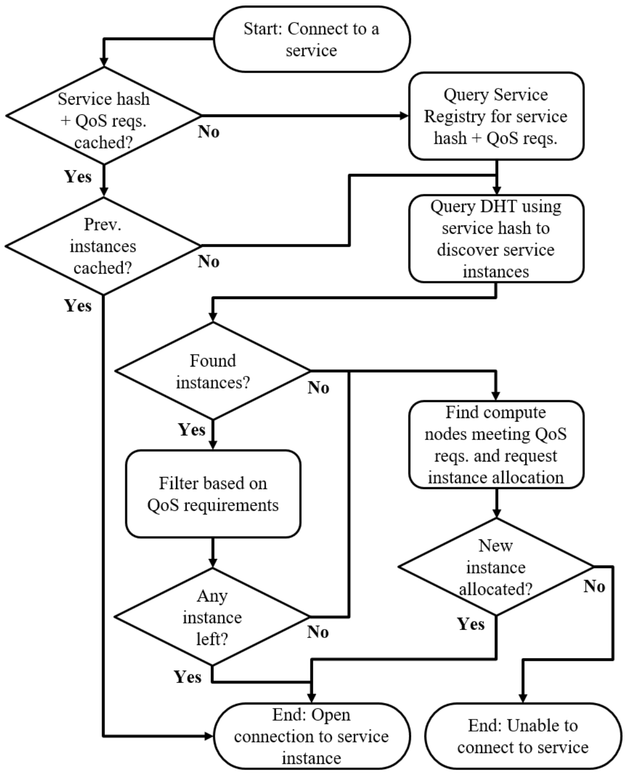

Start: Connect to a service
Service hash + QoS reqs. cached?
No
Yes
Query Service Registry for service hash + QoS reqs.
Prev. instances cached?
No
Yes
Query DHT using service hash to discover service instances
Found instances?
No
Yes
Filter based on QoS requirements
Find compute nodes meeting QoS reqs. and request instance allocation
New instance allocated?
No
Yes
Any instance left?
No
Yes
End: Open connection to service instance
End: Unable to connect to service

Fig. 4: Flow chart of process when establishing a connection with a service.

to get an estimate of the latency to each peer candidate. Simultaneously, the registry cache stores past results from querying the service registry. The use of in-memory caching avoids unnecessarily querying the registry and the DHT network, thus reducing the latency for establishing a connection to a service in subsequent connection attempts.

Scheduling: Each proxy contains a pluggable scheduling module that is used when deciding which compute node to request a new service allocation (and hence, which LCA to contact). The scheduling module can obtain metrics (e.g. available memory, CPU, etc.) regarding compute nodes capable of supporting the new service instance from Prometheus, and filter them based on the service's QoS requirements. Its pluggability enables the heuristics to evolve or be swapped, for example, to consider factors such as location in the network (i.e. core, edge, on-premise equipment, etc.), or scheduling policies (e.g. spread load, pack nodes to save energy, etc.).

G. Connection Management

Figure 4 shows the logical sequence of steps taken when a client node (which itself may be a microservice) in PhysarumSM wants to establish a connection with a service. This process covers four possible scenarios:

1) The client node has past registry results cached, and candidate peers for the target service cached. It can thus immediately establish connectivity to the service.
2) The client node is connecting to the service for the first time, and must query both the service registry and the DHT to discover possible service instances. Instances are found meeting the service's QoS requirements, and a connection can be made.
3) Similar to scenario 2, but no existing instances can be found (or if they were found, they failed to meet the service's QoS requirements). A new service instance must be allocated on a compute node able to meet the service's requirements. After successful allocation, a connection can be made.
4) Similar to scenario 3, but no compute node is found meeting the service's requirements (or they were found, but allocation failed). In this case, the connection attempt returns a failure.

Once a connection is established, clients continue to monitor the network QoS to the service instance. If the QoS experienced by a client is inconsistent and close to the threshold of the requirements, it may continuously disconnect and reconnect. To avoid this, we define two sets of QoS requirements in the service registry, *soft* and *hard*. If a client's perceived QoS violates the soft requirement, but meets the hard requirements, it waits and continuously monitors the quality to see if conditions will improve, eventually disconnecting if the quality fails to do so. If the quality violates the hard requirement, the client immediately disconnects. Once disconnected, a search for an alternative instance begins.

VI. Evaluation

We present a set of experiments and benchmarks to functionally validate our architecture and evaluate our prototype implementation. All experiments were conducted on the SAVI testbed using VMs, allocated with 2 CPU cores (Intel Xeon E5-2650 v4) and 2 GB of memory, as compute nodes. The VMs are virtualized on top of QEMU and KVM, and interconnected with Open vSwitches (OVS). The underlying physical servers that host the VMs are inter-connected via a mix of 1GE and 10GE switches. All VMs run Ubuntu 18.04 with a minimum kernel version of 4.15.0-96.

A. Discovery and Allocation

In the event that a service does not exist when a client makes a request, a new one must be dynamically allocated on-the-fly to service the client. We conduct an experiment to measure the agility of the system in this scenario, using a simple HTTP service that replies with a static payload message. For the experiment, we pre-cache the container image in the compute nodes to avoid having to fetch it from Docker hub, hence

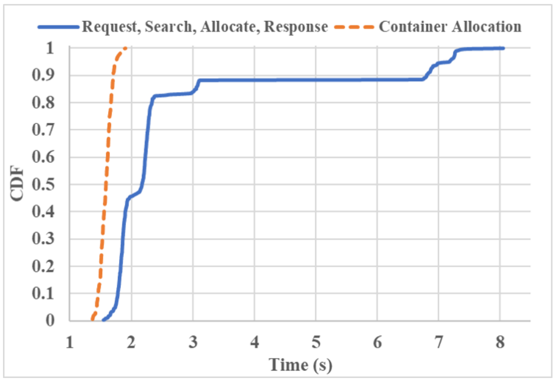


Fig. 5: Cumulative distribution function (CDF) of the HTTP request-response time when the requested service must be allocated. Median of total time = 2.17s. Allocation median = 1.58s.

representing a cold standby. Figure 5 shows, as the solid blue line, the cumulative distribution function of the total time from when an HTTP request is made, to the system searching the DHT and failing to discover a pre-existing instance, to the system dynamically allocating a new instance on a remote node, to when the HTTP response is received from the new instance. The median time to request, search, allocate, and respond is 2.17s. The dotted orange line shows the proportion of that total time taken by just the allocation stage itself (i.e. creating and launching a new container); with a median of 1.58s, the allocation stage comprises roughly 70% of the total time. The remaining time, comprising the HTTP request, DHT search, and HTTP response, has a median under 1s.

This experiment demonstrates how clients can rapidly search for and request deployment of missing services in a decentralized manner within seconds. Additionally, these results serve to showcase the mitigation time of a client when a service (or VNF) it was connected to suddenly becomes unavailable or unable to meet the service's QoS requirements. In [4], we presented a centralized solution that took up to 10s to mitigate a QoS violation between a client and service instance. Our P2P-based solution shows it can match, and on average exceed, the centralized mitigation time.

Currently, the total time shown in Fig. 5 has a long tail beyond the 90th percentile. We have root-caused this behaviour to libp2p's DHT implementation, based on S/Kademlia [25], which occasionally takes longer than expected to discover the new service after it is allocated and launched. There are ongoing efforts to improve the efficiency and performance of the DHT implementation [26], and we believe future versions will improve the performance of the DHT.

B. Throughput vs Service Chain Length

As explained in subsection V-F, our P2P service proxy allows configuration to enable service chaining through multiple P2P-enabled VNF services. When chaining through UDP VNFs, there is the possibility of packet loss, which increases as the chain grows. An experiment was conducted where we chained a series of simple UDP echo services (to minimize both latency and processing complexity that could lead to further loss) to quantify the severity of loss per P2P hop. In this experiment, we set the traffic source to transmit data at 800 Mbps, and additionally, we ensured that no two consecutive services are on the same server (i.e. traffic from one service to the next must traverse a switch and two hosts). Figure 6 reports the measured bandwidth at the receiving end of the chain as the length of the chain grows from 1 service (i.e. simple client-server model) to 10 services.

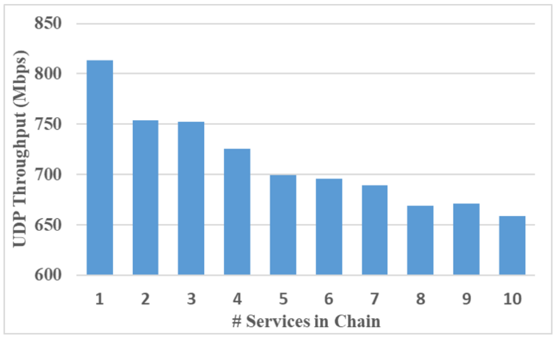
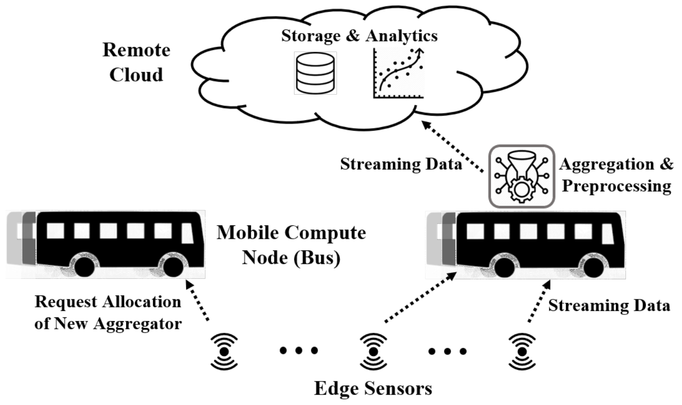


Fig. 6: Degradation of throughput vs length of UDP service chain. Throughput loss rate incurred per P2P hop is roughly 15 Mbps.



Remote Cloud
Storage & Analytics
Streaming Data
Aggregation & Preprocessing
Mobile Compute Node (Bus)
Request Allocation of New Aggregator
Streaming Data
Edge Sensors

Fig. 7: Three-tier data pipeline use-case scenario. Sensors stream to mobile aggregators (in buses), which streams to an end-service in a remote cloud.

It can be seen that on average, each UDP service added to the chain incurs a loss of approximately 15 Mbps in throughput. Code profiling indicates that the current bottleneck is due to the cryptographic operations involved to encrypt the P2P connections. This experiment shows the applicability of PhysarumSM for service chain applications needing throughput at rates up to hundreds of Mbps.

C. Data Pipeline Example

We present a motivating use-case scenario using a data processing pipeline application. Figure 7 shows the scenario, where sensors are deployed at the side of a road, and data is streamed to aggregators that are deployed in buses (i.e. mobile compute nodes) that periodically drive by. For each sensor stream, the aggregators pre-process the data to transform it into an intermediary format that is then streamed to a storage and analytics service deployed in a remote cloud. All three stages in the pipeline communicate over P2P.

Our scenario consists of 48 sensors, where each one is deployed in a container with 1 CPU core and 100 MB of

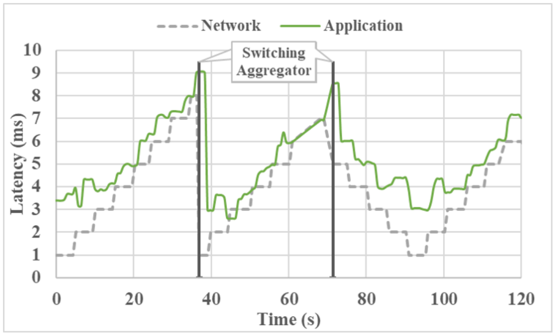


Fig. 8: Sensor-to-aggregator latency of a sensor. Aggregators are different instances on different compute nodes/buses.

memory to emulate a resource-constrained IoT device. The passing buses are emulated using remote VMs with 2 CPUs and 2 GB of memory, and their link latency to each sensor is dynamically varied from a maximum of 10ms down to a minimum of 1ms, and back up to 10ms. Additionally, the sensors' links are capped at 1 Mbps. To simulate an out-of-range bus, the connection is killed once the latency surpasses 10ms. Each bus drive-by takes roughly 100s, and buses are spaced at most 55s apart so that there will often be a bus within the optimal range.

The aggregator service is registered with a 5ms soft latency requirement and a 10ms hard limit. Figure 8 shows the sensor-to-aggregator latency of the underlying network versus the application latency, taken from the perspective of a specific sensor node. As the network latency steadily increases, representing a bus moving away, the application latency also increases. At roughly *t*=25s, the network latency surpasses 5ms, but the sensor continues monitoring the network latency to see if the condition persists. At roughly *t*=36s, the sensor requests a new aggregator instance be spawned on a nearby bus, and switches its connection. The pattern repeats as the second bus proceeds to move away, slowly increasing the latency and eventually triggering the sensor to request allocation and switch aggregators, at *t*=71s, to an approaching third bus.

D. System Overhead

We monitor the resource utilization to determine the overhead of the various components in PhysarumSM. The measurements, collected by Prometheus and summarized in Table I, are for the data pipeline application. The service proxy results are shown for each stage of the pipeline (averaged per instance). Logically, the service proxy's CPU and bandwidth utilization will be different for other applications. For the ping monitor, its bandwidth usage by roughly 13.31 kbps per peer, though this can be reduced by increasing the inter-ping interval. For the service registry, each lookup request transfers 7.91 kB of data (connection setup + request + response). The low overhead demonstrates the feasibility of deploying our system in dynamic edge nodes with limited resources.

Finally, we measure the network latency overhead due to the P2P service proxies. Using a simple HTTP service, we compared the RTTs between the P2P overlay versus a direct underlay connection, repeating for 10,000 iterations. This

TABLE I: Resource Utilization of Components (Data Pipeline App.) CPU model: Intel Xeon E5-2650 v4

Component	RAM (MB)	CPU (%)	Network
Ping Monitor (Bootstrap)	32.11	4.76	13.31 kbps per peer
Service Registry	48.53	0.89	7.91 kB per lookup
LCA	35.79	0.99	175.98 kbps
Service Proxy (Sensor)	40.47	0.88	65.45 kbps
(Aggregator)	36.88	0.49	107.74 kbps
(Analytics)	24.49	0.21	35.7 kbps

experiment was done within a single VM to reduce the effects from switches and links. For each P2P proxy in the path, the RTT increases by roughly 700μs (or 350μs unidirectional).

VII. Conclusion and Future Work

As the number of networked mobile devices grows and their compute capabilities increase, it becomes possible to harness them as an extended edge/fog cloud. However, their mobility results in a dynamic edge with an ever-changing topology and erratic network connectivity. This presents a problematic environment for composite applications and services built using the NFV and microservices architectures, whose performance is dependent on the placement of their constituent parts relative to one-another. Traditional orchestration systems located in far-away clouds are thus unable to contend with the mobility and expected scale of devices in a timely manner.

In this paper, we investigated a decentralized approach to give clients that need a particular service the ability to independently discover available nearby instances, or dynamically allocate a new one. We proposed a P2P-based architecture that meets these goals, implemented a prototype called PhysarumSM, and evaluated it on a multi-tier cloud environment. The evaluations demonstrated how clients are able to discover instances meeting a given QoS requirement in under a second without centralized management. If performant instances do not exist, they can be dynamically allocated on a nearby host. Thus, this work represents a critical step towards scalable, highly-available, and mobility-friendly composite applications and services in dynamic edge networks.

As future work, we plan to explore a variety of directions. To further enhance the system's ability to survive network partitions, we seek to enable node-to-node image transfers. Additionally, using trend analysis on historical data enables proactive service placement and image caching for warm and cold stand-bys. Adding access control capabilities will augment the system's security, guarding against unauthorized access and enabling multi-tenancy over the same DHT network. Algorithms for dynamically tuning the DHT parameters will provide adaptability to changing network conditions. Lastly, we plan to investigate opportunities to utilize our P2P system as a backup mechanism to provide reliability and auto-healing in geo-distributed OpenStack or K8s deployments that rely on centralized control plane services.

References

[1] ETSI, "Network Functions Virtualisation (NFV); Ecosystem; Report on SDN Usage in NFV Architectural Framework," in *ETSI GS NFV-EVE 005 V1.1.1*, 2015.

[2] Cisco, "Cisco Annual Internet Report (2018-2023)." [Online] Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html [Accessed: July 2020].

[3] ETSI, "Mobile-Edge Computing - Introductory Technical White Paper," in *ETSI White Paper*, 2014.

[4] T. Lin and A. Leon-Garcia, "Towards a Client-Centric QoS Auto-Scaling System," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, pp. 1–9, 2020.

[5] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, "A Survey and Comparison of Peer-to-Peer Overlay Network Schemes," *IEEE Communications Surveys Tutorials*, vol. 7, no. 2, pp. 72–93, 2005.

[6] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, "Service mesh: Challenges, state of the art, and future research opportunities," in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pp. 122–1225, 2019.

[7] S. Rajagopalan and L. Ryan, "Istio: A modern service mesh." GlueCon, 2017. [Online] Available: https://istio.io/latest/talks/istio_talk_gluecon_2017.pdf [Accessed: August 2020].

[8] M. Klein, "Lyft's Envoy: Experiences Operating a Large Service Mesh." SREcon Americas, 2017. [Online] Available: https://www.usenix.org/conference/srecon17americas/program/presentation/klein [Accessed: August 2020].

[9] P. Banks, "Consul: Service Mesh for Kubernetes and Beyond." HashiConf, Nov. 2018. [Online] Available: https://www.hashicorp.com/resources/consul-service-mesh-kubernetes-and-beyond [Accessed: August 2020].

[10] A. Das, I. Gupta, and A. Motivala, "SWIM: Scalable Weakly-consistent Infection-style Process Group Membership Protocol," in *Proceedings International Conference on Dependable Systems and Networks*, pp. 303–312, 2002.

[11] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, (USA), pp. 305–320, 2014.

[12] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes: Lessons learned from three container-management systems over a decade," *Queue*, vol. 14, pp. 70–93, Jan. 2016.

[13] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron, "One Ring to Rule Them All: Service Discovery and Binding in Structured Peer-to-Peer Overlay Networks," in *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, 2002.

[14] Q. A. Liang, J.-Y. Chung, and H. Lei, "Service Discovery in P2P Service-oriented Environments," in *The 8th IEEE International Conference on E-Commerce Technology and The 3rd IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services (CEC/EEE'06)*, 2006.

[15] Q. He, J. Yan, Y. Yang, R. Kowalczyk, and H. Jin, "A Decentralized Service Discovery Approach on Peer-to-Peer Networks," *IEEE Transactions on Services Computing*, vol. 6, no. 1, pp. 64–75, 2013.

[16] S. D. Bruda, F. Salehi, Y. Malik, and B. Abdulrazak, "A Peer-to-Peer Architecture for Remote Service Discovery," *Procedia Computer Science*, vol. 10, 2012.

[17] G. Tanganelli, C. Vallati, and E. Mingozzi, "Edge-Centric Distributed Discovery and Access in the Internet of Things," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 425–438, 2018.

[18] M. Le, Z. Song, Y. Kwon, and E. Tilevich, "Reliable and efficient mobile edge computing in highly dynamic and volatile environments," in *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*, pp. 113–120, 2017.

[19] W. Tang, X. Zhao, W. Rafique, L. Qi, W. Dou, and Q. Ni, "An offloading method using decentralized P2P-enabled mobile edge servers in edge computing," *Journal of Systems Architecture*, vol. 94, pp. 1 – 13, 2019.

[20] L. Tang, B. Tang, L. Tang, F. Guo, and J. Zhang, "Reliable Mobile Edge Service Offloading Based on P2P Distributed Networks," *Symmetry*, vol. 12, p. 821, May 2020.

[21] T. Lin, B. Park, H. Bannazadeh, and A. Leon-Garcia, "Deploying a Multi-Tier Heterogeneous Cloud: Experiences and Lessons from the SAVI Testbed," in *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, pp. 1–7, 2018.

[22] T. Lin, B. Park, H. Bannazadeh, and A. Leon-Garcia, "SAVI Testbed Architecture and Federation," in *Future Access Enablers for Ubiquitous and Intelligent Infrastructures*, pp. 3–10, 2015.

[23] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, Mar. 2008.

[24] S. Cheshire and M. Krochmal, "Multicast DNS (RFC 6762)," Feb. 2013. [Online] Available: https://tools.ietf.org/html/rfc6762 [Accessed: June 2020].

[25] I. Baumgart and S. Mies, "S/Kademlia: A Practicable Approach Towards Secure Key-Based Routing," in *2007 International Conference on Parallel and Distributed Systems*, 2007.

[26] "Critical path towards dht efficiency and performance." [Online] Available: https://github.com/libp2p/go-libp2p-kad-dht/issues/345 [Accessed: June 2020].