Dynamic Adaptation for Elastic System Services using Virtual Servers

Abhishek Kulkarni
adkulkar@indiana.edu
Indiana University

Hugh Greenberg
hng@lanl.gov
Los Alamos National Laboratory

Michael Lang
mlang@lanl.gov
Los Alamos National Laboratory

Andrew Lumsdaine
lums@indiana.edu
Indiana University

***Abstract*—A vast majority of legacy runtime systems and middleware prevalent in cluster and supercomputing environments are static in nature. Due to the rising scale and complexity of high-performance computing systems, the static nature of systems software would prospectively impede its scalability and resilience. Traditionally, the mobility of servers is further limited since services are statically bound to specific communication endpoints. To address these challenges imminent for exascale-class systems, distributed middleware needs to support dynamic reconfiguration, redundant and replicated state, and adaptation where the number of servers can vary according to the load in the system.**

We identify the key features necessary from the underlying network infrastructure to support dynamic adaptation and elasticity in distributed system software, and describe the implementation of a high-performance middleware library that implements the proposed interface. We discuss several novel approaches for dynamic resolution using range computations performed by hosts (in software) and by switches (in hardware), and compare the performance on contemporary Ethernet networks. Finally, we validate the benefits offered by our library with two different applications—a scalable DHCP server and an elastic key-value store.

I. Introduction

The rising scale and complexity of high-performance computing (HPC) systems is likely to necessitate a significant change in system design over the next decade. Current HPC system designs suffer from lack of scalable solutions for infrastructure and management. Owing to the greater potential for component failures in the future, system software and runtime services will need to be failure-resistant, adaptive and self-healing such that sustained operation of the system is achieved without manual intervention. A vast majority of HPC system services for monitoring, booting, job and resource management, I/O forwarding, fabric management, runtime systems supporting newer programming models, and communication libraries are still designed with centralized global state and hence are susceptible to single points of failure. They are inherently static in nature where certain configuration parameters relevant to the system cannot be changed at runtime, e.g. number of servers. Further, servers lack mobility as they are bound to fixed communication endpoints. Consequently, traditional system services are not adaptive, e.g. servers cannot dynamically replicate or merge based on the load in the system.

To address the above issues that limit scalability and resilience of distributed middleware, we discuss certain key tenets underlying the implementation of these systems, and the changes necessary to achieve server dynamism, mobility and adaptivity.

Elasticity: At larger scales, elastic system services are critical for efficient power management. In current systems, the servers are, by design, over-provisioned to handle the heaviest imaginable load, but sit idle consuming power when not needed. Instead of dedicating over-provisioned nodes to run all of the servers indefinitely, dynamic adaptation and elasticity allows recruiting more servers as the demand increases, and turning off servers (and the corresponding nodes) as the demand subsides. Furthermore, elasticity permits even redistribution of load in the system, and decreases the likelihood of server failures due to higher load. Elastic behavior not only depends on the distribution of load in the system, but also the ability of servers to freely replicate and merge without any client intervention.

Resiliency: System services need to be failure-resistant and self-healing to achieve sustained operation without frequent manual intervention. Localized failures can be handled through a combination of server mirroring and migration. Mirroring ensures that a server's state is replicated and that failure of a single server in the replica group does not result in global service unavailability. Replicas are synchronized with each other using standard consistency protocols. Server mobility is achieved by migrating the associated state, spawning a server at the new location, and notifying all active clients of the server's new location. In presence of dynamic server discovery, this typically involves updating a global location registry maintained by an external name-server.

The above aspects are closely related to one another and enable dynamic behavior in distributed system software. These properties are typically achieved entirely within the middleware through software indirection involving, for e.g., distributed hash tables (DHT) or directory services. In this paper we demonstrate that by leveraging the network hardware, the desired properties can be achieved transparent to the system software, while providing more optimization opportunities for an efficient implementation. Our key idea is to decouple resolution and lookup from the servers and delegate it to the network hardware. In doing so, the initial bootstrap phase for these systems is significantly simplified and allows servers to adaptively merge, split and migrate. The underlying network automatically *figures* out how to route requests to their destination.

Our main contributions in this paper are the following:

- We identify the necessary features for server dynamism, adaptivity and mobility, and discuss how these can be used as a building block to construct elastic system services (§II).
- We discuss the several trade-offs involved in the design and implementation of such a library including techniques based on packet sniffing, server spoofing and virtualized networking (§III - §IV).
- Finally, the design and implementation of a scalable DHCP service and a distributed elastic key-value store is presented, exemplary of dynamic, elastic system services (§V - §VI).

The remainder of this paper is structured as follows: Section III discusses the issues in achieving dynamic server behavior and proposes a design for a communication library for implementing distributed system software. Section IV presents the trade-offs involved in a high-performance implementation of the library on top of commodity networks. In Section V, we describe the implementation of a scalable DHCP service, `sdhcp`, in detail followed by preliminary results. Section VI presents the evaluation of a elastic persistent key-value store extended to use our approach. The related work is described in Section VIII. Finally in Section IX, we conclude and discuss future directions for this work.

II. Design Goals

A majority of contemporary distributed services are static in nature. Such services communicate over out-of-band channels often based on slower IP-based networks; whereas the high-speed communication network is reserved for data transfer between application processess. As such, servers communicate with clients (or peers) using physical endpoints. However, physical addresses cannot be oversubscribed trivially due to resource limits, such as the limit on number of ports or number of active connections per node. Further, these addresses are statically bound to physical hosts and thus failure-prone due to network or host unavailability. Due to the rising scale and complexity of HPC systems, we conjecture that the static nature of systems software would prospectively impede its scalability and resilience. In anticipation of increased failures, services need to be reconfigurable, dynamically load-balanced and fault-tolerant.

Elasticity: Dynamic load-balancing entails redistribution of servers for reducing latency and increasing throughput. In addition to this, dynamic server adaption permits recruitment or relinquishment of physical resources in response to the active system load. The dynamic response to demand makes services responsive as more servers are available when the system load increases. As the demand subsides and extra server processes are no longer needed, it is wasteful to keep a large number of idle services occupying cores and using power. The solution is to have servers absorb their current state into another server, relinquish their state and return to the pool ready to be used by another service process.

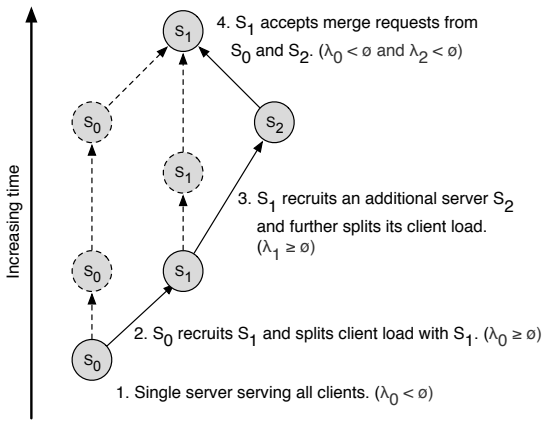


Fig. 1: Adaptive splitting and merging of server state in response to client load. The number of client requests served per second (λ) acts as a trigger for adpative load-balancing.

Determining when to recruit more servers or merge servers depends on a generic cost function based on metrics such as network bandwidth, CPU utilization or the rate of incoming client requests. In the example shown in Figure 1, as server S_0's request rate (λ_0) exceeds a threshold (ϕ), it recruits a new server S_1 and splits its client load. At a later time, S_1's client request rate exceeds the threshold and S_1 recruits S_2 to further share its client load. Finally, when the client request rates decrease below a threshold, a merge process begins. Both S_0 and S_2 transfer their clients to S_1 and exit back to the available pool. The merging and splitting of server state involves additional protocol complexity between servers to maintain consistent state between the replicas.

Resilience: For stateless system services, resilience can be equated with availability without any loss of generality. For stateful distributed services, however, resilience can be achieved by replicating the state with adequate redundancy. System complexity increases as the number of failure modes increase. The most simplistic model is that of a fail-over process that maintains a heartbeat to the primary server and when the heartbeat is lost, a replica takes over the unresponsive server's function. The replication factor is often determined based on the service's availability requirements. By strategic topology-aware placement of replicas at specific physical locations, one could limit failures to nodes, mid-planes, racks or other hierarchical failure domains.

We postulate that the above characteristics are strongly desirable for scalable, elastic system software where failures occur commonly, and particularly where the scale of the system naturally leads to distributed server state. In the next section, we describe the design and implementation approaches to achieve these properties in the network.

III. Design

To achieve dynamic behavior, we decouple server addresses from their implementation-specific transport layer addresses. The design is generic to accommodate any suitable network or transport-level instantiation so long as clients can communicate to a virtual address. We reiterate our design goals here:

- *Endpoint Virtualization:* The clients and servers should be able to communicate via virtual identifiers represented by a flat keyspace rather than via static endpoints (such as a fixed IP address).
- *Adaptive Routing:* The physical or overlay network should be able to adaptively route packets based on the current *home* of a virtual identifier.
- *Filtered Receives:* Finally, the server must allow accepting packets targeted to any of the virtual identifier that it currently owns.

The flat keyspace for addressing allows flexibility in naming and permits mapping of hierarchical namespaces over a flat keyspace. Filtered receives of packets with adaptive routing ensures that servers, irrespective of their location, can receive packets addressed to these virtual endpoints.

Virtual Server Identifier: Each server is identified by a *virtual identifier*, an m-bit identifier that represents the type of a service. Multiple servers can respond to a single virtual identifier forming multicast groups. A suitable instantiation of this identifier depends on the underlying network; for instance, it can be represented by the 48-bit physical address on Ethernet networks, or as a 128-bit address on IPv6 networks. One possible

encoding of the virtual identifier on Ethernet networks is shown in Figure 2. The hierarchical identifier space allows multiple servers to use a common library implementation by serving requests based on the first k-bits of the request. The $(m-k)$-bit request identifier allows dynamic load-balancing by evenly distributing requests from clients to different servers. The choice of a suitable hash function is left entirely to the application. To permit a software DHT-based routing scheme, we restrict the hash function such that the virtual identifier space $\mathcal{K}$ forms a metric space $(\mathcal{K},\ d)$ with a *distance* metric, $d : \mathcal{K} \times \mathcal{K} \rightarrow \mathbf{R}$.

By convention, we define a hierarchical namespace for available services. These services are divided into regions grouped according to the type of service offered. Each region can be inhabited by multiple service implementations. For instance, wildcards such as "`rsmgr.*`" can be used to refer to multiple resource manager implementations. Wildcards are prohibited on the first-level parent namespace for efficient range lookups. The suffix-preserving hash function shown below allows efficient range queries to lookup available services.

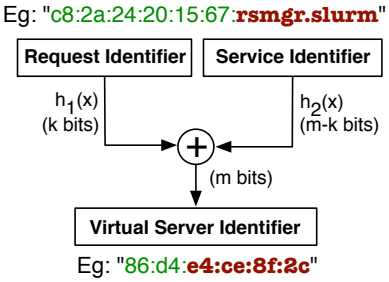


Fig. 2: A Virtual Server Identifier. These virtualize the addressing of hosts in the network and permit dynamic routing through a variety of implementations discussed in Section IV.

Server Resolution: Clients use virtual identifiers to locate a server. The available servers respond to ranges of these virtual identifiers. The ranges are divided between available servers as a means to distribute service load. For instance, if a single server is available due to light load or initial booting of the system, then that server responds to all of the virtual identifiers that a client could potentially hash to. As more servers are dynamically recruited, each server monitors a disjoint range of virtual identifiers and responds to the subset of clients that hash to identifiers in its range. This scheme is depicted below in Figure 3.

It is important to note that the above requirements do not necessarily preclude the implementation of an identifier lookup-table and resolution entirely in software. A distributed hash-table such as Chord where each server participates as a node in the DHT can be used to resolve a virtual identifier to the correct node. This, however, is potentially expensive as it often requires multiple additional round trips to locate the server before communicating with it.

We represent physical resources such as servers and switches using $\mathcal{P}$. The function $listen$ associates an identifier with a server $s \in \mathcal{P}$. The affinity of an identifier is given by the surjective, non-injective (many-to-one) *home* function, $H : \mathcal{K} \rightarrow \mathcal{P}$[1], and its inverse provides a way to query the

[1] H is equivalent to the bind operation of our programmatic API since each server is uniquely identified by a token.

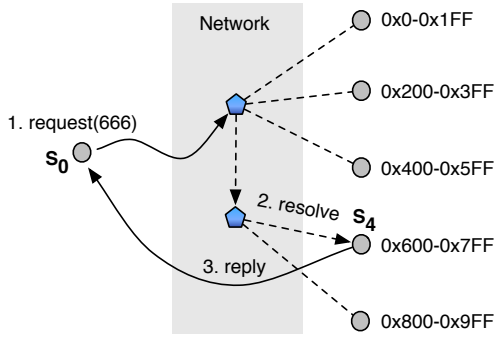


Fig. 3: Server resolution. In this example, each server serves a disjoint range of (512) virtual identifiers. By looking up the identifier mapping in the TCAM, the *switches* forward the request to its appropriate destination, S_4.

identifier range associated with a given server: $H^{-1} : \mathcal{P} \rightarrow \mathbf{K}$.

The adaptive routing in the network is performed through the repeated application of the iterated function $lookup_n : \mathcal{P} \times \mathcal{K} \rightarrow \mathcal{P}$, which accepts virtual identifiers representing the source and the target, and returns the virtual identifier representing the next hop. Each application of the function, hence, yields a physical resource p that is closer (in terms of distance d) to the target.

$$lookup_i(\mathrm{k_i},\ \mathrm{k_n}) = \begin{cases} \mathrm{k_{i+1}} & \text{if } 0 < i < n \\ \mathrm{k_n} & \text{if } i = n \end{cases}$$

$$d(\mathrm{k_i},\ \mathrm{k_n}) = \begin{cases} d(\mathrm{k_{i+1}},\ \mathrm{k_n}) - c & \text{if } 0 < i < n \\ 0 & \text{if } i = n \end{cases}$$

The $lookup_i$ operation is conceptually simple: it looks up the key associated with the next hop in the *forwarding table*. In absence of a proper match, a default forwarding policy, such as broadcasting, is used.

The core interface of our library is shown in Figure 4 and Figure 5. The interface is similar to the POSIX interface (socket API) for communication. The `send` and `recv` operations on *tokens* allow communication between the servers and clients. Tokens are handles to keys (virtual identifiers) registered in the library. Clients have to `resolve` a service in order to obtain a key for it. Servers can `listen` and consequently receive packets for a range of keys. When using the core communication interface, the library sets the forwarding table appropriately such that peers can communicate using arbitrary keys. Finally, the forwarding table can also be explicitly modified using the interface shown in Figure 6. The vectorized interface to add forwarding rules in bulk (ranges) is omitted for brevity.

Extended Interface – Managing Server State: The primitives that we have described so far allow adaptive communication between peers based on virtual identifiers. Essentially, the core interface exposed by our library is sufficient for the implementation of a distributed hash-table (DHT) where keys can relocate. This is enough for services that are stateless or the ones that can synthesize their state at runtime, such as a distributed

```
// resolve a service
key_t key = resolve(service)

// bind for communication
token_t t = bind(key)

// communication
send(t, data, length)
recv(t, data, length)
```

Fig. 4: Core Client interface

```
// announce a new service
key_t key = announce(service)
// listen for a range of keys
listen([key0, ..., keyN])

// accept a new token for communication
token_t t = accept(key)

// communication
send(t, data, length)
recv(t, data, length)
```

Fig. 5: Core Server interface

```
// add a forwarding entry
add_flow(table, src, dst, port)
// delete an existing table entry
delete_flow(table, src, dst, port)
// update dst/port of an existing entry
update_flow(table, src, dst, port)

// commit updates to the fwding table
sync_flow(table)
```

Fig. 6: Forwarding Table Manipulation interface

load monitoring service. Even the simplest of commnoly used system services though—for instance, the scalable DHCP service that we discuss in Section V—maintain state that has to be accounted for in order to make the service truly mobile.

Now we discuss state management primitives that are offered as an extended interface of our library. These primitives serve as a building block for an adaptive, distributed key-value store. Particularly, we associate a data object (state) with each virtual identifier. When servers relocate, the backing state can either be moved entirely, or copied to the new location with the old state designated to a mirror. The high-level state management primitives are shown in Table I. We ignore process management operations (e.g, spawning a remote server) and assume that they, if required, happen out-of-band. Peers have pre-assigned identifiers, and listen for state-management requests from other peers.

The function $get\_data : \mathcal{K} \rightarrow \mathcal{D}$ returns the state associated with a virtual identifier at the current server. The pseudocode

Operation	Description
`split(peer,range)`	Splits the virtual identifier range (and the associated state) between the current server and a remote peer.
`merge(peer,range)`	Merge a peer's identifier range and state with the calling server's state.
`mirror(peer)`	Replicate a server by copying the range and state to a remote peer.
`move(peer)`	Move a server's state to a remote peer.

TABLE I: Extended Server Operations.

for handling a split request is shown in Algorithm 1. First, the server checks for the validity of the requested range. A new split request is sent to the peer to notify it of a split operation. Any requests sent by the client during a split are buffered by the peer until the state has been committed at the peer. The entries for the keys that lie in the requested range are removed from the server's forwarding table ($selftable$) and added to the peer's forwarding table ($peertable$). Note that this operation need not be atomic. To ensure that requests do not end up in a limbo, they are buffered at the server and resent to the same destination address after $peertable$ has been updated. Finally, the state associated with the keys is moved to the peer. The mirror operation is similar to split with the notable difference that the state is copied, instead of moved, to the peer. Mirrors are replicas handled entirely by the servers. The keys associated with mirrors are not duplicated in the forwarding tables.

Our design, shown in Figure 7, is informed by the overheads associated with different implementation approaches that we discuss in Section IV. In particular, for servers to accept network traffic based on dynamic rules, we rely on a *virtual switch manager* which manages local flow rules. The *route controller* controls adaptive routing in the network. The *load balancer* initiates adaptive operations such as splitting or merging of server processes, while the *state manager* ensures that replicated state associated with the server remains consistent.

Furthermore, servers recruit peers to act as mirrors to replicate

Algorithm 1: Basic algorithm for server split.

Input: Peer $peer \in \mathcal{P}$, Range $r_{min},\ r_{max}$

```
// check the validity of range interval
self ← my_token();
{[c_min, c_max]} ← find_matching_range(self, r_min);
assert c_min ≤ r_min && c_max ≥ r_max;
// Set the forwarding entries, and send data objects
selftable ← get_forwarding_table(self);
peertable ← get_forwarding_table(peer);
split_req ← begin_split(peer, r_min, r_max);
foreach key k in [r_min ... r_max] do
    delete_flow(selftable, *, k, port);
    add_flow(peertable, *, k, port);
    data ← get_data(k);
    move(peer, data, sizeof(data));
end_split(split_req)
```

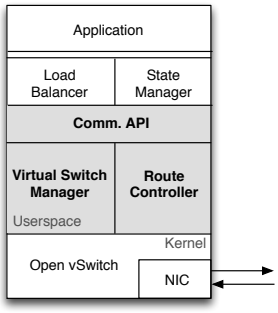


Fig. 7: High-level architecture of our library and its associated components. The shaded region represents the core library interface presented in Figures 4 to 6.

their data. Automatic recovery and mirroring mechanisms ensure that loss of service to a section happens only in case of numerous simultaneous failures across the entire cluster. In such an eventuality, the consistency protocol guarantees that the data associated with the remaining servers is in a consistent state. When there is a failure, one of the mirrors takes over as the server and adopts the clients of the failed process. To ensure consistency, we use a variation of the Fast Byzantine Multi-Paxos algorithm [1]. The main variation with our algorithm is how different acceptor responses are handled. We do not attempt to negotiate the correct response if there is a quorum when responses differ. Since each server could act as a mirror to several active servers, its responses could differ depending upon the state of each server process. Presently, mirrors acts as full replicas which results in larger replicated state; in the future, we intend to explore erasure coding schemes for redundant state.

IV. Implementation

The generality of our design allows a variety of implementations on different network hardware. We take a layered implementation approach where a core library responsible for server address virtualization, resolution and communication is provided as a building-block for elastic system services. Consistency, reliability and fault tolerance is implemented *à-la-carte* according to the service's requirements. The software implementation used in the two case studies in Section V and Section VI is different but uses the same core features. In this section, we explore some of the implementation techniques, discuss their benefits and drawbacks, and finally evaluate the overheads associated with each.

A. Ethernet Bridging

Our current Ethernet-based implementation operates entirely at OSI layer 2. The applications that we discuss use the endpoint identifier space to distribute load, and use UDP to avoid persistent connections to peers.

Endpoint Virtualization: Endpoints (virtual identifiers) are simply encoded as MAC (media access control) addresses. The entire 2^{48} address range is available; to avoid collisions with existing physical network interfaces, we choose a sub-range (xE-xx-xx-xx-xx-xx) assigned by the IANA exclusively to Locally Administered Address Ranges. Clients start with no prior knowledge of the server configuration. We use consistent hashing to locate available services and uniformly distribute client requests amongst available servers.

Adaptive Routing: A simple layer-2 learning switch forwards a packet on the right port based on the destination MAC adress entry in the switch's TCAM (ternary content addressable memory) table. The default policy is to flood the network (with a broadcast) if an unknown destination is encountered. When the server with the said MAC address responds, the switch *learns* the ingress port of the packet and associates it with the MAC address. We presently ignore aging of forwarding table entries and the corresponding replacement policies.

Switch broadcasts (Reactive): : A simple, but inefficient, implementation ignores adaptive routing in the network. For each *compulsory miss* (the first time an address is routed to), the switches broadcast the packet on every egress port. This can also be achieved by explicitly encoding the keys as multicast addresses. This scheme is potentially disruptive and can lead to unavailability due to a phenomenon known as *broadcast storm*.

Address spoofing (Proactive): : To limit the number of switch broadcasts, servers could a priori *teach* the learning switch of the virtual identifiers that it is serving. When the servers specify an intent to listen for a range of addresses by calling `listen(kmin,kmax)`, the implementation sends a spoofed packet to the switch[2] addressed from the MAC address $k \in [k_{min} \ldots k_{max}]$ to itself.

Software-defined networking: : Even though training the switches using spoofing allows indirect manipulation of the switch's forwarding table, it can significantly limit the line-rate at which switching occurs. Explicit manipulation of the forwarding table gives us complete control over how packets are routed through the network. We discuss this scheme in more detail in Section VII-A.

Filtered Receives: Thus far we have discussed the implementation techniques for clients to communicate to virtual identifiers (encoded as MAC addresses), and the network to route them to the right server. Physical network interfaces, however, drop any packets that are not addressed to device's MAC address (unless they are multicast addresses). The filtered receives make a server accept all of the packets for the address range that it is serving. This can be done in one of the following ways.

Promiscuous mode: : When a physical network device is put in the *promiscuous mode*, all of layer 2 traffic is received by the operating system. The servers listen for raw traffic on the network interface, and respond to packets meant for the address that they are serving. This scheme results in severe degradation of throughput due entirely to the overheads of packet sniffing.

OS Filtering: : Some of the overheads of sniffing can be mitigated by filtering the received packets in the kernel. The Berkeley Packet Filter (BPF) infrastructure, typically used by firewalls such as iptables, allows actions on packets based on certain policies.

Software switching: : A related approach is to perform switching in software, or use a virtual network device implemented in software. The emergence of software-defined networking has pushed hardware vendors to offer explicit software

[2]We ensured that these spoofed packets addressed from an arbitrary MAC address to itself go to the switch, instead of short-circuiting through the loopback device interface.

control over the control and data plane. Burgeoning frameworks such as Intel DPDK (Data Plane Development Kit) and Cisco usNIC enable packet processing in software. Virtualizated (or para-virtualized) networking interfaces such as OpenVZ, VirtIO and others allow low-latency, high-throughput communication between virtual machines. We use Open vSwitch [2] for filtering received packets by manipulating local flow rules.

B. Experimental Evaluation

Here we evaluate the overheads of the possible implementation strategies that were mentioned in the previous section. Instead of low-level network instrumentation, we rely on a *black-box* methodology where we measure the effect of adaptive routing and filtered recieves on end-to-end packet latencies. Since the network behavior is heavily influenced by the scale and layout of the network, we consider two experimental setups: a medium-sized setup with 16 nodes connected to a 10G switch, and a larger setup with 500 nodes connected to 3 GigE switches.

Experimental Setup: The smaller setup consists of 16 Dell PowerEdge R720 nodes, each equipped with 16-cores 2.6GHz Intel Xeon E5-2670 processors, and 32GB memory. The nodes are connected via a 10G NIC to a Dell PowerConnect 8024F switch. The operating system used was Ubuntu Linux 12.04.5 with kernel version 3.2.0. The larger testbed consists of 500 2.6GHz AMD Opteron 252 with 8GB of RAM. The nodes were connected to 3 Black Diamond BD10808 10G switches. The OS on the nodes was Ubuntu 12.10 with kernel version 3.13. We used Open vSwitch 2.3.1 for managing local flow rules.

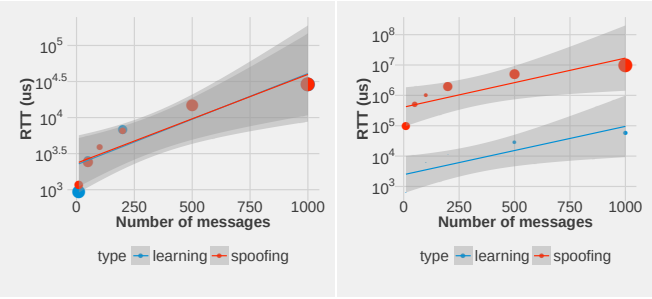


Fig. 8: Overhead of switch broadcasts for a 16-node setup (left) and a 500-node setup (right) in sending a packet to a range of MAC addresses sequentially. The *learning scheme* causes a switch broadcast on every packet send, whereas flows are pre-installed in the *spoofing* scheme. The size of the dots represents the variance of the collected sample.

The first evaluation compares the overhead of *adaptive routing* in the network. Our naive approach relies on a learning switch broadcasting an unknown packet to all of its outgoing ports. This *reactive* approach (**learning**) contrasts the *proactive* approach (**spoofing**), where all of the peers program the switch by sending spoofed packets before starting any communication. The microbenchmark is set up to send a 60-byte message from a host to a random destination MAC address and receive it on a remote host. The round trip time (RTT) of the communication between the peers is measured. As we wish to charaterize the switch's behavior, we run the experiment repeatedly by increasing the number of messages each time and performing a linear regression on the measured values. This ensures that any effects due to essential, background communication (such as ARP requests) can be factored out of our measurements. The absolute values, determined by the slope of the regression line, are given in Table II.

Machine size	Switches	Type	RTT per message (us)
16 nodes	1	Learning	28.64
		Spoofing	57.44
1024 nodes	3	Learning	28.84
		Spoofing	9988.92

TABLE II: Adaptive routing overhead for sending packets to random service-prefixed MAC addresses.

First, we choose a service-prefixed virtual identifier representation where the first k bits represent the service identifier. For this case, as we see from Table II, the number of nodes and switches do not influence the RTT for the learning scheme. Spoofing is about 2x slower at 16-nodes, but ~370x slower at the larger scale! This can be confirmed from Figure 8. We expect spoofing to be slower than learning because it involves a successful lookup in the TCAM table. Essentially, the way we generate random MAC addresses for the test has a direct influence over the time it takes to lookup the TCAM table. In the service-prefixed case, since the first few bits of the address remain fixed, a lookup in the forwarding table has to match the most specific entry by the virtue of longest-prefix match. This scheme incurs considerable overhead depending on the number of entries in the table that have the same leading k bits. This explains the non-intuitive result we see where the prepopulation of service-prefixed entries considerably slows down the lookup operation.

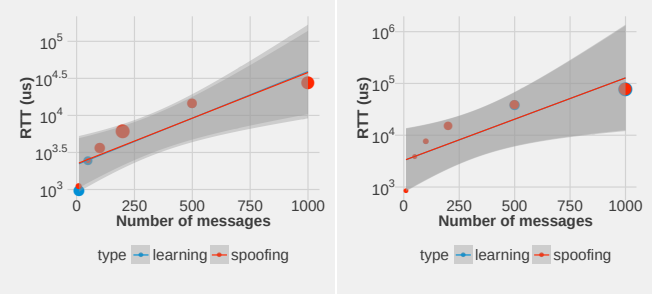


Fig. 9: Overhead of switch broadcasts for a 16-node setup (left) and a 500-node setup (right). The scheme differs from the one in Figure 8 as the range of destination MAC addresses that we send to follow a uniform random distribution.

Our hypothesis is confirmed by the results shown in Table III. If the virtual identifier is suffixed by the service identifier, the MAC addresses that packets are sent to have uniformly random prefixes, and the spoofing scheme is 3x slower than the learning scheme as shown in Figure 9. It is important to note that although learning does better than spoofing, it broadcasts a packet to every node connected to the switch, and subsequently incurs filtering overhead at every host.

For our previous tests, the network interfaces of the destination hosts were put in promiscuous mode so that they could receive any destination MAC address. In general, this is expensive in case of randomized simultaneous communication between peers. To demonstrate this, multiple peers concurrently communicated with multiple servers and the average total time

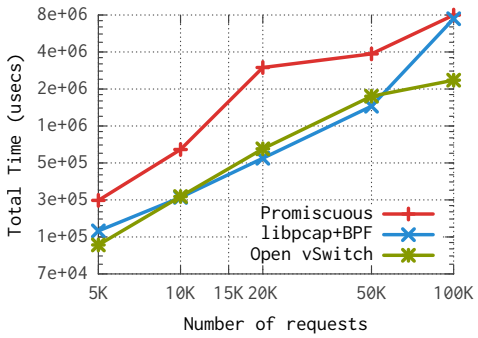


Fig. 10: Overhead of filtering receives using different strategies: Promiscuous indicates receiving all packets and filtering in user-space; libpcap+BPF involves filtering in the OS by installing a packet filter; Open vSwitch relies on a virtual switch implementation to filter a receive.

Machine size	Switches	Type	RTT per message (us)
16 nodes	1	Learning	27.85
		Spoofing	76.80
1024 nodes	3	Learning	27.48
		Spoofing	77.16

TABLE III: L2-switch learning overhead for sending to random service-suffixed MAC addresses.

for serving all requests was measured. In Figure 10, we observe that while filtering in user-space is expensive, Open vSwitch performs close to 3x better when there are many concurrent pending requests to receive.

V. A Scalable DHCP service: SDHCP

A common runtime service for clusters is that of preparing a node for operation. This involves assigning an initial network-layer address to the node, serving it an operating system image, and initializing the node before it is added to the resource pool. The anticipated load for a boot service varies between all nodes booting simultaneously to single nodes booting at random intervals. Traditional booting schemes rely on dynamic broadcast protocols that create single points of failure as the loss of a boot controller node may result in part of the machine being unavailable. While multicast protocols such as Trivial File Transfer Protocol (TFTP) serve well, it is our experience that implementations of multicast booting do not handle sporadic node requests well. Further, simultaneous multicast booting is very demanding on shared resources such as parallel file systems resulting in system administrators booting clusters in sections rather than all at once. We implemented a prototype elastic service similar to DHCP, called `sdhcp`, to hand out unique network identifiers to clients.

`sdhcp` starts as a single server running on a single node. The entire IP space is defined in a configuration file and is read only by the master server. Other auxiliary data (node state, IP addresses, hardware addresses) is stored in memory. As clients are served IP addresses, `sdhcp` turns clients into servers as needed. Some servers are used as "mirrors" for redundancy, whereas others serve a range of the IP space and handle client requests to redistribute load. These roles are not mutually exclusive and some servers may perform both.

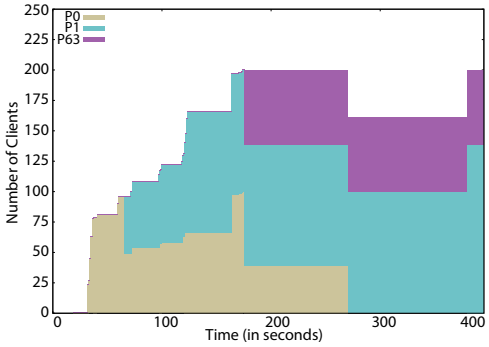
The master server immediately turns the first client it discovers into a mirror. As additional clients are served, client entries are first committed to the mirror before a response is sent to the client. As more clients are given an IP address, some are turned into additional mirrors for the first server. When there are enough mirrors (three, in our current implementation), the server delays handling client requests and gives a range of the IP space it owns to another server. We call this operation a *split*. When an `sdhcp` server detects that it is overloaded, it will attempt to identify a server for the *split*. If a server is not available, it promotes a client into a server. As client requests diminish, the number of servers required for distributing the server load can be reduced. Therefore, servers that are receiving few or no client requests attempt to perform a *merge* operation with other servers. If successful, a *merge* transfers the clients and the IP space served from one server to the other. Merging minimizes the number of unnecessary servers and can reduce them to simply mirrors. The current implementation of `sdhcp` uses the number of entries in the client table and the rate of requests from clients to determine when to split or merge.

As mentioned above, `sdhcp` uses mirrors to replicate data belonging to servers actively responding to requests sent to virtual hardware addresses. Each state change to an active server must first be committed to its mirrors before it takes effect. This includes: a new client entry, a split operation, a merge operation, a new mirror, a mirror deletion, and server recovery. Further, servers maintain active contact with their mirrors through periodic heartbeat once every minute. When an active server fails, the mirror with the smallest IP address attempts to take over the IP space, virtual hardware address space, and the previously served clients belonging to the failed server. If successful, the mirror in charge of the recovery will change the ownership of the relevant data to itself, and begin serving instead of the failed server. The fail-over process is demonstrated in Figure 11a. P_0 is the first server and splits to P_1 at approximately time $t = 60$. All clients involved in this run obtained an IP address at $t = 160$. Then at around $t = 180$, P_0 splits to P_{63}. At this point, 3 servers are available to handle client requests, even though all clients have completed. Each server is in charge of the clients that would hash to any of the virtual identifiers the server is responding to. At $t = 270$, P_0 dies. P_1 has the smallest IP address, therefore it initiates the recovery process and is successful at $t = 380$. P_1 absorbs the clients, the IP range, and the virtual identifier range that previously belonged to P_0.

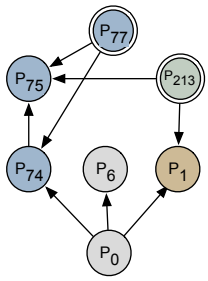
In certain cases, automated recovery is not possible. This would occur due to the following scenarios: the first server started fails before it mirrors to another server; there is no majority of mirrors responding in order to perform the recovery operation; and the entire system fails due to a power failure. In the second case, `sdhcp` will not attempt recovery in order to avoid multiple servers serving the same IP and virtual identifier range.

A. Experimental Evaluation

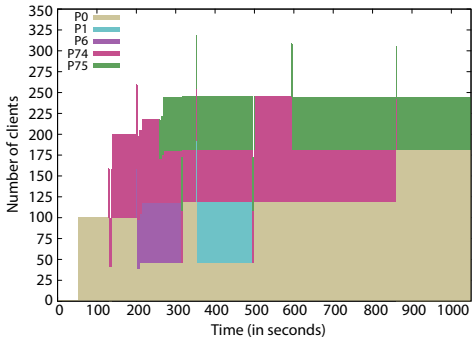
Experimental setup: All tests were completed on a 256-node 1024-core cluster based on dual-socket AMD Opteron dual-core processors, running Debian Linux with a 2.6.32 kernel, and using a 1GiG Ethernet interconnect. The nodes' clocks were synchronized with ntpd. We gathered individual log files from the servers and extracted time-stamped messages to compile these results.



(a) Migration of clients being served after a failure. At time 270, server P_0 dies. At time 380, P_1 recovers the IP space and clients served by P_0.



(b) Dynamic split-merge sequence of `sdhcp` servers serving DHCP client requests.



(c) Dynamic behavior of `sdhcp` servers. At time 300, there are a maximum of 4 servers simultaneously serving requests to achieve higher throughput.

Fig. 11: Elastic behavior of `sdhcp` that shows servers splitting and merging over time.

Dynamic behavior: To show dynamic behavior of `sdhcp`, we spawned a single initial server and started clients on 245 nodes. The number of servers dynamically spawned in these tests were dependent on the amount of memory each server was allowed to use to store its client data. A server was allowed approximately 50 clients, which forced the initial server process to recruit more servers when its table was close to full. This number was chosen based on the number of clients involved in the test. If a larger table size was chosen, less dynamic behavior would have been observed. Once client requests start being sent to the first server, the first server recruits mirrors and splits its virtual address range and IP range with the newly recruited server. Figure 11b shows which server processes split to load balance the client requests. Due to the 50 entry limit on the number of clients per server, five servers were needed to respond to the clients.

To show the dynamic behavior over time, Figure 11c traces `sdhcp` servers as they split and merge. The number of clients being served is shown on the y-axis, while time advances on the x-axis. At time $t = 0$, the initial `sdhcp` server responds to as many clients as possible before it reaches its predefined limit on table storage. It then converts one of its clients into a server and distributes a selection of its clients to the new server, in this case P_{74}, at approximately $t = 130$. At $t = 200$, P_0 again splits, this time to P_6. Next, at $t = 240$, P_{74} splits to P_{75} and at $t = 250$, all the clients have been served. When $t = 310$, P_6 merges to P_0. A server decides to initiate a merge if its saved count of client requests reaches zero. This number is decreased by the cost function over time. In this example, P_6 served clients at approximately $t = 210$. After those clients were served, P_6 does not handle any more client requests; giving the cost function time to reduce the saved value of client requests to zero. Each server has its own count of the client requests it received and thus makes decisions on this local value. This explains why at $t = 350$, P_0 splits to P_1 even though P_6 recently merged. The behavior described continues until there are only two servers remaining: P_{75} and P_0. The large spikes in the graph are due to slight differences in the times of the operations reported by the servers. The differences in time can be attributed to following: other requests in the queue, thread scheduling, and clock drift.

Resilience: Starting with a single server process, `sdhcp` responds to client requests and recruits servers to act as replicated data stores. For this test, we set a maximum of five mirrors for a server process. There is a performance trade-off with the amount of communication required to keep the servers handling client requests and the mirrors consistent; the higher the maximum mirror count the higher the cost to keep the mirrors consistent. The mirrors created by this experiment, P_{77} and P_{213}, are shown in Figure 11b as double-circled nodes. Recall that only P_0, P_1, P_6, P_{74}, and P_{75} are actually serving IP addresses to clients, whereas the other servers are only involved in the replication of client data. To test resilience, we showed how the distributed `sdhcp` servers would continue to provide services after killing and restarting nodes in the system. Recall from Figure 11a, we ran a test with 200 clients and 3 `sdhcp` servers, and proceeded to kill the initial server P_0. After the failure is detected, P_1 recovers the IP space, virtual address range, and clients served by P_0. Depending on the amount of data absorbed by P_1 and the amount of data P_1 was already serving, P_1 could decide to split to a new server from one of its mirrors and redistribute its load.

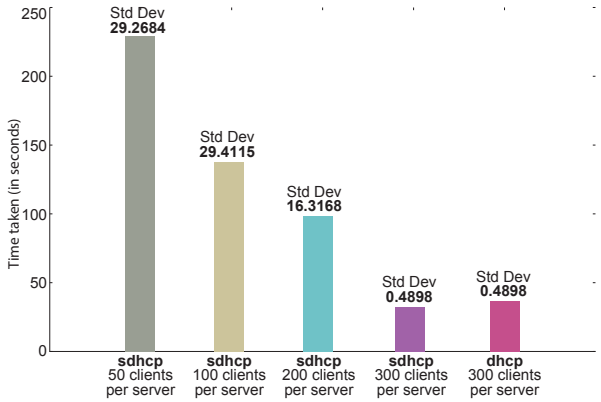


Fig. 12: `sdhcp` performance compared to standard DHCP. The y-axis denotes the time taken to serve 245 clients.

Performance: For the performance investigation, we compared the time for `sdhcp` and standard DHCP [3] to service all of the 245 client requests for the test system. We then varied the maximum number of clients that the `sdhcp` server could hold in its table space, which also changed the number of `sdhcp` servers dynamically created. To get a baseline of performance, we first configured `sdhcp` with a maximum client table size of

300, large enough to hold all clients in the test system. In this case, only one mirror was obtained before all the clients were served. In addition, no splitting occurred since the table was not filled enough for a split to be initiated. For this baseline case, `sdhcp` was slightly faster than DHCP; this is shown in the two right-most bars in Figure 12. The 3 leftmost bars in the figure show `sdhcp` configured with varying maximum client table sizes: 200, 100 and 50. The lower the client table size, the more servers are split to respond to client requests. This also takes more time due to the number of mirroring servers that are needed to keep the tables between servers consistent. Since the maximum number of mirrors a server can use is bounded, these costs are eventually amortized and the approach demonstrates good scalability.

VI. Elastic-KVS: A Persistent Key-Value Store

Most system services are stateful and require that the distributed state of the system be stored in a persistent medium. And to that end, a distributed key-value store is being considered as a building block for large-scale distributed system services [4].

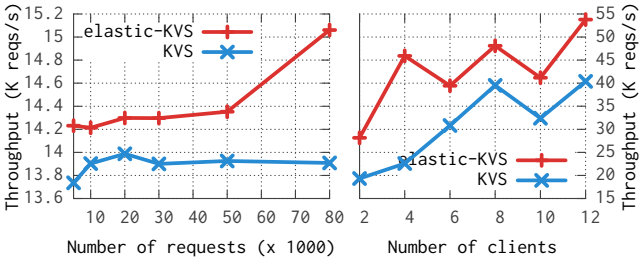


Fig. 13: Performance comparison of elastic-KVS with standard KVS. The jagged lines (on the right) are due to oversubscription of mulitple clients on the same physical nodes.

We implemented a persistent file-backed key-value store where the value associated with each key is stored in a separate file. The servers concurrently manage incoming client connections, handle concurrent client requests, and manage file I/O required for each request. The server loop for the standard KVS is managed using `select()`, and the file I/O is performed using POSIX `read()` and `write()` operations. *elastic-KVS* is the adaptive variant of standard KVS where servers can split, merge, move or mirror dynamically. It uses UDP on top of the raw L2 communication interface exposed by our library.

Operation	Local (s)	Remote (s)
Mirror	1.078	3.336
Split	1.289	3.782
Merge	1.304	2.544
Move	-	3.871

TABLE IV: Overheads of *elastic* operations for KVS servers. Mirror only measures the cost of spawning another instance of the server. Split & Merge have to manipulate switch flow rules. Additionally, Merge & Move have to shutdown a running instance of the server.

To test the performance of elastic-KVS, we ran a benchmark that performed a `put`, `get` or `delete` operation on one of the randomly chosen 256 keys. Multiple concurrent clients were used to synchronously send requests to the server. The workloads were pregenerated in-memory so that clients were not a bottleneck. From Figure 13, we notice that the throughput of standard-KVS saturates sooner due to the overhead of the `select()` loop and file-descriptor management when serving multiple concurrent requests. For elastic-KVS, as the requests are filtered in the virtual switch, we see a much higher receive throughput in both cases. The peak bandwidth of the standard KVS server saturates at 14K reqs/sec, whereas elastic-KVS achieves up to 10% more throughput with many outstanding concurrent requests. The average cost of elastic operations (split, merge, mirror, move) is shown in Table IV. The "Local" case measures the time it takes to perform the operation on virtual servers on the same physical node. This provides a lower bound on the overheads of the elastic operations while ignoring the process management costs (spawning or killing a remote process). We see that these operations take just a few seconds even when multiple nodes are involved and allow fast adaptation in response to the varying load.

VII. Discussion

We have demonstrated our approach towards building elastic system services on contemporary Ethernet networks. We now discuss how recent networking trends can be used to implement and improve our proposed approach.

A. Software-defined Networking

Software-defined Networking (SDN) is an emerging approach for decoupling the *control* plane of a network from its *data* plane. It allows explicit control over routing through direct, fine-grained programming of the network. OpenFlow [5] is the standard protocol that exposes control over forwarding in the network hardware. Using OpenFlow, servers can act as controllers to implement new policies and topologies while still forwarding at line rate. By design, we chose our table manipulation interface (shown in Figure 6) to match the OpenFlow client API. SDN enables direct control over adaptive routing in the network. Flow rules are inserted or updated each time an adaptive operation such as split or merge is to carried out. As Openflow-compliant switches become more pervasive, our proposed scheme would be able to target a variety of platforms.

B. High-speed Interconnects

Custom HPC systems like Cray or Blue Gene still use Ethernet based management networks; although Blue Gene/Q has an option for IB or 10G Ethernet on its I/O nodes. To take advantage of the high-speed interconnect and gain an advantage from topology-aware placement of services, it would be preferable to have services communicating over the high-speed interconnection network. We consider Infiniband (IB) as a likely target for implementation. With IB, each node has a GUID (global unique identifier) and a LID (local identifier); the GUID is equivalent to the MAC address in Ethernet space. The LID is assigned by the subnet manager and is only unique within a subnet. The centralized subnet manager is a weak point for large-scale IB deployments. Fail-over subnet managers can be configured, however, they only act on fail-over and do not help with the active work of mapping and configuring the fabric. Distributed computation of fabric routes is an open research question, but these routes could be precomputed and loaded in a distributed way with our subnet-manager based prototype.

RDMA over Converged Ethernet (RoCE) is an extension to the IB architecture [6] where the IB traffic is encapsulated in L2 headers. RoCE allows implementation of an Ethernet infrastructure over the InfiniBand transport. It further enables applications to leverage SDN to dynamically manage flows for IB traffic.

VIII. Related Work

Fault tolerance and load-balancing are fundamental notions in the design of distributed systems, and have been explored in numerous contexts in the past. A Distributed Hash Table (DHT) introduces abstraction of location in distributed systems facilitating both fault-tolerance and load-balancing. Chord [7] was the first to propose the notion of virtual servers as a means of balancing load. By allocating $log\ N$ virtual servers per real node, Chord ensures that with high probability the number of objects per node is within a constant factor of an optimal distribution. However, to achieve load balancing, their scheme assumes that nodes are homogeneous, objects have the same size, and that object IDs are uniformly distributed. Virtualization of location using consistent hashing [8] has been used to address load imbalance in structured P2P systems [9]. Location virtualization has also been achieved using network protocols such as JXTA [10] and mobile IPv6 [11]. The techniques are similar with the difference that we implement the resolution directly using the network hardware. We use process replicas to mirror service state for failure recovery, whereas distributed cloud services such as ZooKeeper [12] require state to be saved in a transaction store on a physical disk.

The implementation techniques described in Section IV have been explored previously in scalable Ethernet designs. Seattle [13] proposes flat-addressing and one-hop resolution of hosts by hashing to a MAC address. PortLand [14] encodes the location of hosts in a psuedo MAC address (PMAC) to allow distributed location discovery and fault-tolerant routing. Our focus was to extend these dynamic capabilities to user-space applications for building elastic system services.

Mobility of virtual machines employs techniques closely related to ours. Prior work on migration of virtual machines has focused on *how* [15] and *when* [16] to migrate virtual machines without interrupting them. Live router migration has been proposed as a network-management primitive previously [17]. Our goal was to expose programmable control over the network for system services. More recently, the trend towards Network Function Virtualization [18] addresses virtualization of network middleware in software.

Finally, elastic and malleable characteristics have been explored in the context of traditional HPC runtime systems such as MPI [19] and Charm++ [20]. In such systems, the load balancing capabilities directly complement elasticity in the middleware where residual processes can be cleaned up as the communicator or the job shrinks. For shrinking or expanding a partition, both of these approaches require checkpointing of state and re-bootstrapping the runtime. By using virtual servers, we can achieve online migration of servers resulting in much lower overheads for elasticity.

IX. Conclusions

In distributed systems, location virtualization has been a popular technique towards achieving fault-tolerance and load-balancing. In this paper, we have introduced a design for dynamic discovery and adaptation of processes as a building block for constructing large-scale, elastic system services. We discussed several implementation techniques to efficiently realize the design on contemporary Ethernet networks, and demonstrated its effectiveness for two distributed system services. We believe that such dynamic and adaptive middleware techniques may prove critical as we move towards systems with higher node counts.

Acknowledgements

This work was supported in part by the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC52-06NA25396 with Los Alamos National Security, LLC, and by the National Science Foundation under awards CNS-1042537 and CNS-1042543 (PRObE).

References

[1] L. Lamport, "Fast paxos," *Distributed Computing*, vol. 19, pp. 79–103, 2006.

[2] B. Pfaff, J. Pettit *et al.*, "Extending networking into the virtualization layer," in *Proc. of workshop on Hot Topics in Networks (HotNets-VIII)*, 2009.

[3] I. S. Consortium, "DHCP - dynamic host configuration protocol," 2011.

[4] K. Wang, A. Kulkarni *et al.*, "Using simulation to explore distributed key-value stores for extreme-scale system services," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 9:1–9:12.

[5] N. McKeown, T. Anderson *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[6] InfiniBand, TM, "Infiniband Architecture Specification, Release 1.2.1 Annex A16: RoCE," 2010.

[7] I. Stoica, R. Morris *et al.*, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, ser. SIGCOMM '01. New York, NY, USA: ACM, 2001, pp. 149–160.

[8] D. Karger, E. Lehman *et al.*, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, ser. STOC '97. New York, NY, USA: ACM, 1997, pp. 654–663.

[9] A. Rao, K. Lakshminarayanan *et al.*, "Load balancing in structured P2P systems," in *IPTPS*, ser. Lecture Notes in Computer Science, M. F. Kaashoek and I. Stoica, Eds., vol. 2735. Springer, 2003, pp. 68–79.

[10] L. Gong, "Jxta: A network programming environment," *IEEE Internet Computing*, vol. 5, no. 3, pp. 88–95, May 2001.

[11] C. E. Perkins and D. B. Johnson, "Mobility support in ipv6," in *Proceedings of the 2Nd Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '96. New York, NY, USA: ACM, 1996, pp. 27–37.

[12] P. Hunt, M. Konar *et al.*, "Zookeeper: wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, ser. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 11–11.

[13] C. Kim, M. Caesar, and J. Rexford, "Floodless in seattle: a scalable ethernet architecture for large enterprises," in *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, ser. SIGCOMM '08. New York, NY, USA: ACM, 2008, pp. 3–14.

[14] R. N. Mysore, A. Pamboris *et al.*, "Portland: a scalable fault-tolerant layer 2 data center network fabric." in *SIGCOMM*, vol. 9, 2009, pp. 39–50.

[15] C. Clark, K. Fraser *et al.*, "Live migration of virtual machines," in *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, ser. NSDI'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 273–286.

[16] T. Wood, P. J. Shenoy *et al.*, "Black-box and gray-box strategies for virtual machine migration." in *NSDI*, vol. 7, 2007, pp. 17–17.

[17] Y. Wang, E. Keller *et al.*, "Virtual routers on the move: live router migration as a network-management primitive," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4, pp. 231–242, 2008.

[18] J. Martins, M. Ahmed *et al.*, "Clickos and the art of network function virtualization," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 459–473.

[19] A. Raveendran, T. Bicer, and G. Agrawal, "A framework for elastic execution of existing mpi programs," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, May 2011, pp. 940–947.

[20] A. Gupta, B. Acun *et al.*, "Towards Realizing the Potential of Malleable Parallel Jobs," in *Proceedings of the IEEE International Conference on High Performance Computing*, ser. HiPC '14, Goa, India, December 2014.