MOSAIC: Declarative Platform for Dynamic Overlay Composition

Yun Mao, AT&T Labs - Research, Boon Thau Loo, Zachary Ives and Jonathan M. Smith, University of Pennsylvania

Abstract

Overlay networks create new networking services using nodes that communicate using pre-existing networks. They are often optimized for specific applications and targeted at niche vertical domains, but lack interoperability with which their functionalities can be shared. MOSAIC is a declarative platform for constructing new overlay networks from multiple existing overlays, each possessing a subset of the desired new network's characteristics.

This paper focuses on the design and implementation of MOSAIC: composition and deployment of control and/or data plane functions of different overlay networks, dynamic compositions of overlay networks to meet changing application needs and network conditions, and seamless support for legacy applications. MOSAIC overlays are specified using *Mozlog*, a new declarative language for expressing overlay properties independently from their particular implementation or underlying network.

MOSAIC is validated experimentally using compositions specified in *Mozlog* in order to create new overlay networks with compositions of their functions: the *i*3 indirection overlay that supports mobility, the *resilient overlay network (RON)* overlay for robust routing, and the *Chord distributed hash table* for scalable lookups. MOSAIC uses runtime composition to simultaneously deliver application-aware mobility, NAT traversal and reliability. We further demonstrate MOSAIC's dynamic composition capabilities by Chord switching its underlay from IP to RON at runtime.

MOSAIC's benefits are obtained at a low performance cost, as demonstrated by measurements on both a local cluster environment and the PlanetLab global testbed.

1. Introduction

The Internet's architecture continues to evolve towards a ubiquitous communications medium interconnecting mobile personal devices, environmental sensors, and Web services. As new services (voice, video, emergency response, *etc.*) are deployed on the network, new extensions of the existing Internet architecture are required for new capabilities, such as efficient routing among mobile and wired nodes, location of proximity-based services, and wide-area service discovery and composition.

Overlay networks [31] are one way to deploy new services using the existing Internet for connectivity [32]. However, despite deployment at global scale and emerging support for legacy applications [15], overlay networks now face several hurdles. First, they are often optimized for a specific application and may not be useful in all contexts. Second, overlay networks are generally targeted at, and limited to, niche *vertical* domains (*e.g.*, mobility [42, 25], security [17], reliability [2]). Third, the networks do not normally interoperate or share their functionality. For example, resiliency [2] and mobility [38] provided by one overlay cannot easily be leveraged by other overlay networks. Recent proposals for "clean slate" redesign of the Internet itself will exacerbate this problem, as more and more overlays are proposed and implemented.

Example 1.1. *Alice and Bob use private networks behind separate NATs, and wish to communicate regularly via VoIP or video conferencing, occasionally sharing data from internal web servers with trusted friends. As Alice and Bob travel regularly, and their IP addresses change, continued contact and communications should be seamless.*

In principle, Alice and Bob can use a combination of *i*3 [38][1] for NAT traversal, ROAM [42] for mobility, RON [2] for reliability, and if DoS attack prevention is important, a secure overlay such as SOS [17] can be added. This type of custom overlay offers benefits over a monolithic approach, *e.g.*, Skype [36]: it can accommodate future application needs and changing network conditions. For example, RON may be excessive for a network with limited failures, and hence it may be desirable to remove it; whereas, in a partially-connected network, epidemic routing [40] would be desired. Alice and Bob may require *session-layer* mobility support, requiring DHARMA [25] instead of ROAM.

Combining overlays to achieve desired capabilities sounds straightforward, but it is challenging in practice. One must first identify combinations of overlays that can work together and provide the right set of capabilities. Then the mechanics of interconnecting the overlays must be tackled. Previous work [15] has shown that *bridging* between different overlays requires significant "glue code." *Layering* one overlay over another is generally not even feasible, as each layer assumes it is running directly over IP.

In this paper, we present a new point in the design space of network architectures, called MOSAIC [2], that aims to achieve extensibility based on the application of database techniques to the networking domain. MOSAIC is a system that provides a *declarative* framework for developing, deploying, combining, and composing overlay networks — one capable of bridging between overlays, stacking them in layers, dynamically changing the layers or bridges, and allowing for rapid extensibility with new functionalities. It enables (1) rapid creation and deployment of new overlay networks, (2) dynamic adaptivity to *compose* overlay networks to meet changing application needs and network conditions, and (3) seamless support for legacy overlay networks and applications within the infrastructure.

This approach enables modular reuse of resources and functions. It also facilitates rapid experimentation and the deployment of new network features. This is a major step

[1]Note that in this paragraph, *i*3 stands for Internet Indirection Infrastructure, ROAM stands for *Robust Overlay Architecture for Mobility*, DHARMA stands for *Distributed Home Agent For Robust Mobile Access*, RON stands for *Resilient Overlay Network*, and finally, SOS stands for *Secure Overlay Services*. Each system is described in detail in their respective citations.

[2]A mosaic is a larger pattern or picture constructed with small pieces of colored glass, stone, or other material. Likewise, MOSAIC builds useful overlay services from existing overlay components.

forward compared with existing hand-coded approaches [15] for manually bridging amongst different overlays.

MOSAIC is based on *declarative networking* [22, 21, 20, 19], a declarative, database-inspired extensible infrastructure using *query languages* to specify behavior. Declarative programming allows programmers to say "what" they want, without worrying about the details of "how" to achieve it. This programming paradigm makes it easy to compose protocols, either vertically (layering) or horizontally (bridging), since composition is largely confined to the "what", while composition of the "how" can be automated. It also provides better language and runtime support for dynamic adaption.

In MOSAIC, overlay compositions are specified in a high-level specification language, which is then further compiled into the *Mozlog* declarative networking language that defines the composed network protocols. Unlike previous declarative networking languages, *Mozlog* provides several novel language features essential for dynamic composition: (1) *dynamic location specifiers*, combined with runtime types, enable flexible naming and addressing; (2) *composable virtual views* support modularity and composability; (3) *data and control plane extensibility* supports composition; and (4) *declarative tunneling and proxying* enable support for legacy applications.

We note that although porting existing overlay code to the new platform is necessary for using many overlay services in MOSAIC, it is made simple because MOSAIC has a much higher-level set of abstractions for managing distributed state and its propagation. Hence the code is more concise and its properties are more straightforward to express. Moreover, because MOSAIC's abstractions are higher level, it is easier to specify protocols' sub-functions in a modular way. New protocols can be developed quickly in MOSAIC, and these can leverage sub-functions from existing protocols.

Our long-term vision is to automatically and dynamically compose overlays that are best suited for the application requirements. There are two parts to achieving this vision: (1) automatically reasoning about properties of composed overlays and their interactions with one another, and (2) the mechanisms of actually "gluing" together the overlays in a unified framework. The former problem is beyond the scope of this paper, and is considered a challenging problem to automate [7, 37]; today it can only be accomplished manually by human experts. The latter problem is the focus of our work, as we provide new capabilities that an expert can harness to get much better compositions than with the previous state of the art.

The rest of the paper is organized as follows. Section 2 describes the options for overlay composition. Section 3 presents an architectural overview of the MOSAIC infrastructure. Section 4 summarizes the aspects of *Mozlog* important to MOSAIC. Section 5 illustrates how the MOSAIC compiler automatically translates composition specifications into *Mozlog* rules. Section 6 shows example compositions developed using MOSAIC. In Section 7, *Mozlog* specifications are shown to be executable within a distributed query processor via modifications to the P2 declarative networking system. In Section 8, measurement results are presented for networks created on a local cluster and the PlanetLab testbed.

2. Background: Overlay Composition

Overlay network composition combines distinct parts or elements of existing overlay networks to create a new overlay network with new functionalities. Overlay composition can be achieved along both the data and control planes.

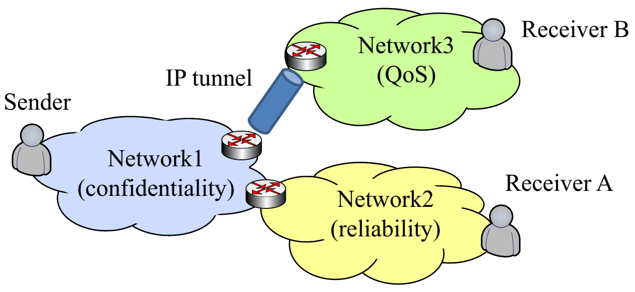

Figure 1: Overlay composition by bridging.

2.1. Data plane composition

The data planes of two overlay networks can be composed horizontally by *bridging* between the networks. They can be composed vertically by *layering* one overlay over the other.

In *bridging* (see Figure 1), each overlay network runs on top of the same substrate (*e.g.*, the IP network) directly. However, for a variety of reasons (*e.g.*, sending from a wireless to a wired network), it may be necessary to send a packet across multiple overlay networks to reach the receiver. This is usually done via a *gateway* node that belongs to both networks. If such gateways do not exist, two nodes from each network need to be connected via an IP tunnel to route packets. In Figure 1, a sending laptop using wireless may use an overlay that provides confidentiality to route traffic over the wireless links, then use an overlay with reliability guarantees to deliver important data that is not time-sensitive to receiver A, while using a QoS overlay to deliver multimedia traffic to receiver B.

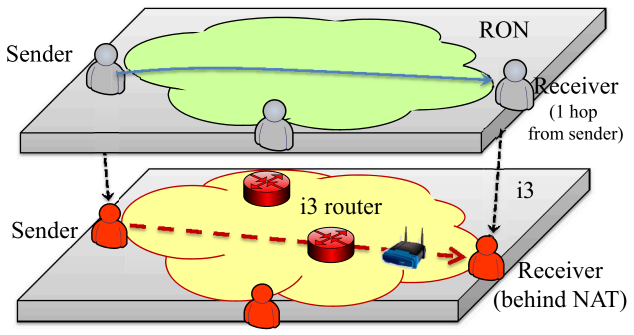

Figure 2: Overlay composition by layering.

In *layering*, logically a packet is routed within a single data plane of an existing overlay network. However, the data paths between the nodes inside the overlay may be constructed on top of other overlay networks, rather than using IP. For example, RON only works for nodes that have publicly routable IP addresses. As shown in Figure 2, by composing RON on top of another overlay protocol that enables NAT traversal, such as *i*3, nodes behind NAT should be able to join the RON network.

We note that the two data plane compositions listed above are not mutually exclusive; some data composition scenarios may combine both layering and bridging. Prior attempts [15] to compose overlay networks support bridging but not layering. Layering adds a powerful new composition primitive that enhances individual overlay network nodes with multiple new services.

2.2. Control plane composition

One overlay network's control plane may be layered over either the data plane or the control plane of another overlay network. For example, it is possible to build the control message channels of distributed hash table (DHT) protocols [4] such as Chord over the data plane of RON. Typically, the failure detection components of DHTs assume that hosts unreachable via IP are dead. In fact, some hosts may be alive and functioning, but temporary network routing failures may create the illusion of node failure to part of the overlay nodes. If the network failures occur intermittently, the churn rate is increased and may create unnecessary state inconsistency [11]. Using a resilient overlay such as RON can overcome some of the network failures and reduce churn. In a highly disconnected environment, one can use epidemic [40] forwarding of control plane messages.

Some overlay network protocols have complex, layered control planes. For example, both *i*3 and DOA [5] use DHTs for either forwarding or lookup. RON (an overlay that provides resilient routing) and OverQoS [39] (an overlay that provides quality-of-service routing) heavily depend on measurements of underlying network performance characteristics such as latency and bandwidth. When overlay networks are built from scratch over IP, it is conceivable that different logical overlays built on the same physical IP topology may duplicate the effort to maintain DHTs or perform network measurements. Nakao, *et al.* [29], observed that on PlanetLab, each node had 1GB of outgoing ping traffic daily: many overlay networks running on the same node were probing the same set of hosts without coordination. Such duplicated probing traffic can be wasteful, and interactions between probe traffic may introduce measurement error. A composition-driven approach is to build smaller elements that provide well defined interfaces (*e.g.*, OpenDHT [35] for DHT lookup and iPlane [23] for measurement) so that they can be easily composed with upper layer overlay network control planes to share rather than compete for resources.

3. MOSAIC Overview

In this section, we provide an overview of MOSAIC, and describe how it provides a framework for composing and re-composing overlay networks, by combining the use of overlay bridging and layering described in Section 2. Note that we do not currently tackle the issue of determining the appropriate compositions, but rather provide the overlay composition specification and implementation framework.

MOSAIC is designed to be deployed as a composition service on a shared overlay infrastructure where all nodes run the MOSAIC engine. Each node runs a MOSAIC engine that is responsible for running the overlay protocols. In addition, a *directory service* is deployed and shared by all infrastructure nodes, and maintains the meta information of each overlay to enable the composition process.

On this infrastructure, several overlay networks can co-exist, and are not necessarily deployed on all nodes. Individual overlay protocols are specified using the *Mozlog* declarative networking language, then compiled and executed in MOSAIC. Composed overlay networks are instantiated by leveraging existing deployed overlays, either by layering (above or below) or bridging with them. In addition, private networks outside of the infrastructure are bridged via public gateways with overlays deployed on this infrastructure. Since the composition glue code is written in *Mozlog*, it is most natural to implement each individual overlay as a declarative network in *Mozlog*. However, MOSAIC can also support legacy overlays with the use of an adapter (see Section 7.2).

In the rest of this section, we will describe the MOSAIC engine and its directory service in greater detail, followed by an overview of the composition process

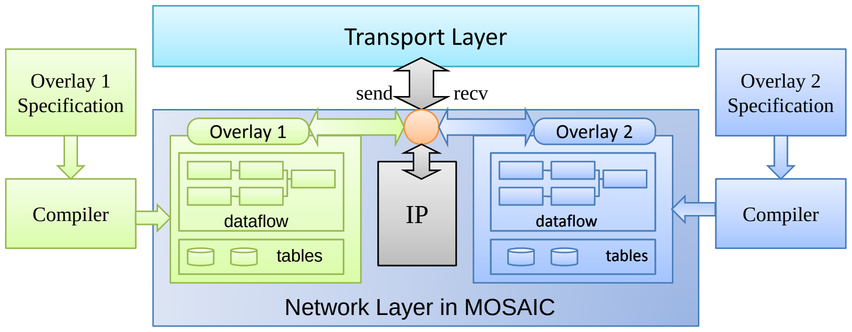

Figure 3: An overview of the MOSAIC engine for network layer overlays.

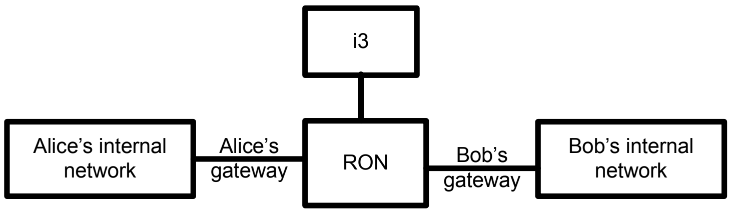

Figure 4: Graph of *i*3 layered over RON, and private networks of Alice and Bob bridged with RON.

MOSAIC engine. Figure 3 illustrates the MOSAIC engine from the perspective of a single node. MOSAIC is positioned at the network layer in the network stack, replacing IP. It exposes a simple interface to the transport layer by providing two primitives: `send(DestAddress, Packet)` and `recv(Packet)`. In IP, a packet consists of an IP header with fixed format and a data payload not interpreted by IP. In MOSAIC, `Packet` is represented abstractly as a structured data element, which might be a set of scalar values or even nested tuples. The encoding of this packet is up to the specific overlay protocol, and declarative mappings or transformations can convert between the packet formats of different overlays (see Section 4). `DestAddress` is a specially typed tuple, with the first attribute being the identifier of the overlay network to which the packet belongs. This identifier is used to demultiplex the `send` requests to different overlays or IP at the network layer. A `send` request will trigger a `recv` event at the node or nodes who own the `DestAddress` if the network successfully routes the packet.

Directory service. For each overlay running on the infrastructure, there is a directory service that maintains the following information: (1) A *unique identifier* for the overlay; (2) The list of *physical nodes* that are currently executing the overlay; (3) The list of users who can utilize the overlay, and their privileges (*e.g.*, whether they can bridge with this overlay; these privileges are set by an overlay's owner); and (4) Additional meta-data that describes the overlay, such as its attributes, node constraints, etc. As part of the process of creating a composed overlay, the user may issue queries to the directory, searching for existing overlays that meet their criteria for composition.

A back-of-the-envelope calculation suggests that the state of the directory is on the order of 100MB for a single overlay, given a reasonable MOSAIC deployment[3]. The directory server can be hosted by a centralized server or in a distributed fashion [9, 4] for scalability. The design choice of the directory service is orthogonal to the MOSAIC architecture. In this paper, we focus on the use of a centralized server. We note that a centralized service is sufficient for maintaining the metadata information for thousands of infrastructure nodes, as demonstrated by the PlanetLab Central [32] service.

Composition process. To create a composed overlay network, a MOSAIC user (*e.g.* a network administrator) first uses the directory service to locate overlay networks that meet their criteria for composition, and retrieves relative metadata information . Second, the administrator creates a *composition specification*, which is a high-level graph-based description of the desired component overlay networks and their interactions. Then, the specification is compiled into the *Mozlog* language used by MOSAIC's compiler, described later in Section 5. As part of this process, new code is created that "glues" the compositions together. Finally the generated *Mozlog* code is deployed to the physical nodes to start the new network, and the directory information is updated regarding the newly composed network.

MOSAIC's declarative approach provides major benefits. First, there are the benefits of declarative networks in terms of compactness and safety. Second, protocols are specified at a higher level, making them more modular. Finally, high-level composition

[3]Our calculations are based on the overlay network being deployed on 100,000 physical nodes with 1 million users on the MOSAIC infrastructure. The metadata size is 100 bytes for each physical node and user.

specifications have a significant potential for correctness checks and for making inferences about the compositions' attributes — and especially for reasoning about *property interactions* among different overlays. For example, an insecure overlay when bridged with a secure overlay will result in an end-to-end insecure overlay. A scalable lookup overlay will increase its robustness when executing over a resilient overlay, at the expense of its performance.

3.1. Composition Specifications

Figure 4 shows a graphical representation of a composition specification, based on the example scenario introduced in Section 1. Appendix A shows an example composition specification encoded in XML for this specific scenario. Each module (node) represents a component overlay network (*e.g.*, *i*3 and RON) deployed on the infrastructure, or a private network. The links represent *connectors*, where vertical and horizontal links denote layering and bridging, respectively. Here, the *i*3 overlay is layered over RON; Alice and Bob's private networks are bridged to RON. In addition to a unique overlay identifier, each module configuration consists of the following:

- **Physical node constraints**: When the overlay is first deployed, the user who created the overlay can constrain the set of nodes on which the overlay may be executed. This can be in the form of a prefix to indicate that nodes must be deployed on particular subnets, or enforce the inclusion of particular nodes (*e.g.* Alice's and Bob's gateways) must be on both the *i*3 and RON networks.

- **Attributes:** Each overlay network has properties that characterize its capabilities, including mobility, secure routing, NAT traversal, resilient routing, anonymity, private networks, etc. These properties can be queried by users to identify overlays that meet their requirements.

- **Code:** If a module is loaded for the first time, code can be included in the configuration. This can either be legacy code, or *Mozlog* specifications for declarative networks.

- **Default gateway:** Each module can specify a default gateway for bridging. In the absence of an explicitly specified gateway, the common physical node sitting on both networks is selected to serve as the gateway.

- **Access control:** MOSAIC supports restrictions on which users can utilize an overlay, and their privileges (*e.g.*, layering above or below, and bridging, etc.).

The connectors between modules have properties associated with them. Bridging (horizontal lines) must specify whether there are default gateways to be used, and whether tunneling is permitted. If two modules are specified to be bridged via a default gateway node, both overlays must run on the specified gateway. Layering (vertical lines) also has constraints on whether the overlay has to be layered on a subset or all of the nodes. In this example, to get the full benefits of RON, all *i*3 nodes should utilize RON as their underlay. However, this is not strictly required: *i*3 nodes that do not run RON will default to using IP. For both bridging and layering, one can further specify whether some connections replace existing ones.

3.2. Composition Compilation

Once the composition is specified, a *composition compiler* is used to generate the *Mozlog* code that "glues" together different overlay networks based on the specifications. The compiler is either a client-side software, or deployed as a service in conjunction with the directory service.

The compilation process can be performed in two different ways. First, a composition can *create* overlays, either from scratch where each module contains the code implementing each overlay, or incrementally where the new overlay is built on existing ones, *e.g.*, by adding new overlays over existing ones, or bridging overlays via identified gateways. Creating overlays incrementally requires the composition specifications to refer to existing overlays by their unique identifiers. Second, a composition can also *modify* overlays, which involves replacing existing modules with new ones, and this requires connectors to indicate that they are replacing existing composition links.

Given the above mechanisms, we outline how layering and bridging can be achieved by compiling modules and connectors, and provide a detailed process description and examples in Section 5. The first step is to perform basic checks to ensure all the links are legal, based on the attribute constraints and physical node constraints. *E.g.*, one cannot layer one overlay over another if they are configured for completely disjoint sets of nodes. Two overlays cannot be bridged if their bridge connector does not permit tunneling and the two overlays do not share any common node. Once validated, *Mozlog* rules for composition and all required overlay code are uploaded to relevant nodes for execution.

3.2.1. Layering

Layering of a control or data plane over another overlay's data plane is achieved by ensuring that every protocol uses *logical addresses* — rather than being bound to physical addresses. At runtime MOSAIC will bind (or rebind) the upper layer's logical address to the underlay address. These bindings are stored in a separate table that can be updated to facilitate dynamic changes to layering.

MOSAIC allows the control plane of one overlay network to layer over another overlay's control plane, accessing its internal state. Here, each overlay exports the state of its composable components, in the form of database logical views (query results presented as a named table). An example of such state is a distributed hash table's contents, which can be modeled as a relation with tuples associating keys and values. Importantly, accessing a neighboring protocol's state can be done within the overlays' specification language — there is no "impedance mismatch" between languages, and interoperability issues are minimal.

3.2.2. Bridging

Depending on requirements, bridging can be done either *pre-configured* or *on-demand* in MOSAIC.

- **Pre-configured method.** When the composition specification involves bridging multiple overlays, forwarding state is created on designated gateways based on the bridge connectors indicated in the composition specifications. When a sender sends a packet whose destination contains an address of an overlay in which the sender does not participate, MOSAIC routes the packet to the gateway, which then continues to forward the packet along the bridged overlay. In addition to a static gateway, the sender can also use a pre-configured anycast service [16, 10] to select and route packets to one of the overlay nodes, preferably close in terms of network distance to the sender.

- **On-demand method.** The sender utilizes source routing to explicitly describe the data path to the destination via designated gateways among different overlays found in the specification. Alternatively, the gateway holds address translation state that uniquely identifies the flow between the sender and the receivers, it performs indirection. The on-demand mechanism enables user-driven dynamic bridging. We will describe examples of such compositions in Section 6 using the *Mozlog* language.

3.3. Dynamic compositions

MOSAIC exploits *Mozlog*'s declarative model to facilitate *dynamic overlay composition*: since network definitions in MOSAIC separate specification from implementation, the system can (assuming the right constraints are met) freely replace either the IP or an existing overlay underneath one overlay network with a second overlay network — *i.e.*, it can *layer networks*. For example, the protocol used in RON is a modified link-state protocol, which is general enough to operate on any connected graph. The original RON implementation assumes IPv4 as a substrate, and hence it is hard-coded to use publicly routable IP addresses. In MOSAIC, protocols are written with a network-agnostic addressing scheme, so a RON overlay can instead use addresses from one or more lower-level overlay networks, provided they are reachable from one another. This allows MOSAIC to *dynamically switch* an existing overlay's underlay based on the network conditions, *e.g.*, an executing overlay that utilizes IP can dynamically layer itself over RON when routing losses are high, or further switch to an epidemic forwarding strategy when the network is disconnected.

Dynamic overlay switching in MOSAIC is achieved by changing the binding between an upper overlay's logical addresses and the underlying network and its (lower-level) addresses. This technique is overlay-agnostic. However, we must be careful to preserve application and overlay semantics. In particular, if dynamically switching maintains the *same endpoints* on route requests (as RON, above, does), then the switch is permissible. Likewise, if the lower overlay *state is not visible to the layers above*, and all endpoints provide the same functionality (*e.g.*, in a content distribution network), then the switch is also permissible. In other cases, we would need to re-architect the overlays and possibly the application to redistribute state over the new underlay, and to be tolerant of transient states.

4. The Mozlog Language

Having described MOSAIC's basic composition framework, we next present the *Mozlog* declarative networking language that is generated from the composition specifications. As with previous declarative networking languages [22, 21], *Mozlog* is based on the Datalog [34] query language, and extends Datalog in novel ways to support composition.

As background, each Datalog rule has the form `p :- q1, q2, ..., qn`., which can be read informally as "`q1` and `q2` and ... and `qn` imply `p`". Here, `p` is the *head* of the rule, and `q1`, `q2`,...,`qn` is a list of *literals* that constitutes the *body* of the rule. Literals are either *predicates* with *attributes* (which are bound to variables or constants), or boolean expressions that involve function symbols (including arithmetic) applied to attributes. Predicates in Datalog are typically relations, although in some cases they may represent functions.

Datalog rules can refer to one another in a cyclic fashion to express recursion. The order in which the rules are presented in a program is semantically immaterial; likewise, the order predicates appear in a rule is not semantically meaningful. Commas are interpreted as logical conjunctions (*AND*). The names of predicates, function symbols, and constants begin with a lowercase letter, while variable names begin with an uppercase letter.

Mozlog is a distributed variant of traditional Datalog, primarily designed for expressing distributed recursive computations common in network protocols. We illustrate *Mozlog* using a simple example of two rules that compute all pairs of reachable nodes:

```
r1 reachable(@S,D) :- link(@S,D).
r2 reachable(@S,D) :- link(@S,Z), reachable(@Z,D).
```

The rules `r1` and `r2` specify a distributed transitive closure computation, where rule `r1` computes all pairs of nodes reachable within a single hop from all input links, and rule `r2` expresses that "if there is a link from `S` to `Z`, and `Z` can reach `D`, then `S` can reach `D`." By modifying this simple example, we can construct more complex routing protocols, such as the distance vector and path vector routing protocols.

Mozlog supports a *location specifier* in each predicate, expressed with `@` symbol followed by an attribute. This attribute is used to denote the source location of each corresponding tuple. For example, all `reachable` and `link` tuples are stored based on the `@S` address field. The output of interest is the set of all `reachable(@S,D)` tuples stored at node `S`, representing reachable pairs of nodes from `S` to `D`.

In this Section, we highlight the *Mozlog* language itself. We provide details of the compilation process from composition specification to *Mozlog* and use cases in Section 5, and discuss implementation details in Section 7. We focus on key language features necessary to support overlay composition in Sections 4.1-4.4 and cover some related features in Section 4.5.

4.1. Addressing

Mozlog has two distinctive features for addressing nodes in the network. First, a location specifier is decoupled from the data tuple so that tuples can be accessed from multiple logical overlay networks that the host belongs to. Second, because multiple overlays are selected and composed dynamically, location specifiers are not bound to IP addresses anymore. Instead, each location specifier is associated with a runtime type which is bound to an overlay.

4.2. Decoupling Location from Data

Mozlog predicates have the following syntax:

```
predicate[@Spec](Attrib1,  Attrib2, ...)
```

In the absence of any location specifier, `predicate` is assumed to refer to local data. In this case, the rule body is executed as a Cartesian product across all input tables. For example, in the following rule,

```
a1 alarm@R(L,N) :- periodic(10), cpuLoad(L), nodeName(N),
                   monitorServer(R), L > 20.
```

`periodic` is a built-in local event that will be triggered every 10 seconds. The predicates `cpuLoad`, `nodeName`, and `monitorServer` are local tables. The rule specifies that for every 10 seconds, if the CPU load is above the threshold 20, an `alarm` event containing the current load `L` and host name `N` will be sent to the monitoring server `R`.

Decoupling data from its location enhances interoperability and reusability, as well as dynamic re-binding of addresses. Multiple overlays can interoperate (*i.e.*, exchange state) by sending network-independent data tuples in a common data representation. Moreover, since these rules are rewritten in a location-independent fashion, they can be reused on different network types (*e.g.*, *i*3, RON, or IP). Finally, since it does not bind addresses to data, the language is friendly to mobility, where host movement (and hence a resulting change in its IP address) does not invalidate its local tables.

4.2.1. Runtime Types for Location Specifiers

Another *Mozlog* feature involves adding support for runtime types to location specifiers. This feature is necessary for dynamically composing multiple overlays at runtime. Location specifiers are denoted by an `[oID::]nID` element, where `oID` is an optional overlay identifier, and `nID` is a mandatory overlay node identifier. For example, consider *i*3 and RON overlays with identifiers `i3_oid` and `ron_oid` respectively. `i3_oid::0x123456789I` denotes an *i*3 node with identifier `0x123456789I`, and `ron_oid ::12.34.56.78` denotes a RON node with IP address `12.34.56.78`. In the absence of any overlay identifier, IP is assumed.

At runtime, MOSAIC examines the location specifier of each tuple and routes it along the appropriate network. To illustrate the flexibility of our addressing scheme, consider the CPU load monitoring example from Section 4.1. Rule `a1` can be rewritten as `a2`, in which the monitoring server `R` refers to an *i*3 key generated as a hash of its name `N` instead of an IP address:

```
a2 alarm@R(L, N) :- periodic(10), cpuLoad(L), nodeName(N), serverName(SN),
                    L > 20, Key = f_sha1(SN), R = i3_oid::Key.
```

4.3. Data and Control Plane Integration

Overlay composition requires the integration of the data and control planes of multiple overlays. To achieve this, *Mozlog* enables declarative specification of the data plane behavior. Given an overlay `oid`, `oid.send` and `oid.recv` event predicates specify the data forwarding algorithm. We will describe how these `send` and `recv` events are generated within the dataflow execution framework later in Section 7. Focusing on the language feature now, we illustrate this feature via an example based on the data plane of an RON overlay `ron_oid`.

```
snd ron_oid.send@Next(Dest,Pkt) :- ron_oid.send(Dest, Pkt),
                                   ron_oid.RT(Dest, Next),
                                   localAddr(Local), Local != Dest.

rcv ron_oid.recv(Pkt) :- ron_oid.send(Dest, Pkt), localAddr(Local),
                         Local = Dest.
```

The table `ron_oid.RT` denotes the RON routing table. Rule `snd` expresses that for all non-local `Dest` addresses, the data packet (`Pkt`) is sent along the next hop (`Next`) which is determined via a join with RON's routing table (`ron_oid.RT`) using `Dest` as the join key. These packets are then received via the rule `rcv` at node (`Dest`), which generates an `oid.recv(Pkt)` event at `Dest`.

In *Mozlog*, the `send` and `recv` predicates are usually not directly used by other rules, but rather automatically invoked by the MOSAIC runtime engine when the location specifier type of a tuple matches the overlay. As a result, one can bridge the data planes of different overlays together, or layer the control plane of one overlay network over the data plane of another. We provide more details in Section 5.

4.4. Modularity and Composability

To support overlay composition, *Mozlog* supports *Composable Virtual Views* (CViews). The CView is a language construct for defining rule groups that, when executed together, perform a specific functionality.

4.4.1. CView Syntax and Usage

The syntax of CViews is as follows:

```
viewName[@locSpec](K1,K2,...,Kn, &R1,&R2,...,&Rm)
```

Each CView predicate has an initial set of attributes `K1,K2,...Kn` which are already bound to input values read from another predicate (intuitively, these are like input parameters to a function call). The remaining attributes, `&R1,&R2,...,&Rm`, represent the *return values* from invoking the predicate given the input values. This is akin to the use of input binding restrictions [33], a well-studied problem in the data integration literature, which were used to pass data into queriable Web forms to retrieve relation results.

We illustrate using a view definition for the following CView predicate `ping(Src, Dest, &RTT)`:

```
def ping(Src, Dest, &RTT) {
  p1 this.Req@Dest(Src,T) :- this.init(Src,Dest), T = f_now().
  p2 this.Resp@Src(T) :- this.Req(Src,T).
  p3 this.return(RTT) :- this.Resp(T), RTT = f_now()-T.
}
```

Any rule that must compute the RTT between two nodes can simply include the `ping` predicate in the rule body. Here, `this` is a keyword used to express the context of the CView. All predicates beginning with `this` are valid only locally within the `ping` CView. There are two new built-in events/actions: `this.init` and `this.return`. Rule `p1`, upon receiving event `this.init` along with the query keys `Src` and `Dest`, takes the current timestamp `T`, and passes the data to the host `Dest` as a ping request. After the destination node receives it in rule `p2`, a ping response event is immediately sent back to the source with the timestamp. In rule `p3`, the source node calculates the round trip time based on the timestamp and issues a `this.return` action that finishes the query processing.

4.4.2. Composition and Resource Sharing

CViews are a natural abstraction for composing control plane functionalities over different overlays. We provide an example to show how to construct trigger sampling in *i*3 by composing Chord and RON using their respective CView definitions. We consider the Chord lookup, whose CView is defined as follows: `chord.lookup@Ldmk(Key, &DestID, &DestAddr)`.

Given a query on `Key`, the CView returns the lookup result: the Chord ID (`DestID`) and the network address (`DestAddr`) of the destination. In addition, RON maintains several CViews to export the current pair-wise EWMA latency, bandwidth and loss rate measurement results. The latency CView is:

```
ron.latency(Src, Dest, &EWMA_RTT)
```

When an *i*3 client tries to locate a private trigger that relays its traffic, it can leverage the RON measurement results and find the best private trigger.

```
s1 bestPT(KeyAddr, K, RTT) :- periodic(SAMPLE_INTERVAL),
                              i3.underlay(LocalAddr), K = f_randID(),
                              chord.lookup@LANDMARK(K, &_, &KeyAddr),
                              ron.latency(LocalAddr, KeyAddr, &RTT).

s2 trigger@KeyAddr(NodeID, LocalAddr) :- periodic(TRIGGER_REFRESH_INTERVAL),
                                         node(NodeID),
                                         i3.underlay(LocalAddr),
                                         bestPT(KeyAddr, _, _).
```

The rules `s1-s2` are used by a local node `LocalAddr` to compute a private trigger with the lowest RTT from itself. Note that attributes denoted by "_" represent variables which are inconsequential to the output derived in the rule head, and are omitted for simplicity. Periodically, every `SAMPLE_INTERVAL` seconds, `LocalAddr` picks a random node and obtains a sample RTT. The sampling is performed by rule `s1` using the `chord.lookup` CView predicate to locate a node `KeyAddr` corresponding to a random identifier `K`. Then the `ron.latency` CView predicate obtains the RTT measurement between `LocalAddr` and `KeyAddr`. The use of CViews allows us to perform multiple distributed operations (Chord lookup, followed by RON measurement) all within a single rule. Based on the sampling result stored in `bestPT`, rule `s2` periodically refreshes the current best `trigger` at the node `KeyAddr`.

To summarize, the advantages of CViews are as follows. First, CViews promote code reuse and enable functionality composition between different overlays (as with the shared `ping` CView). Second, CViews abstract details of asynchronous event-driven programming. In the `ping` example, nodes are no longer required to maintain pending state for every ping message that was sent out: the compiler automatically takes care of that. This avoids the tedious churn and failure detection rules often required in other declarative languages. This enhances readability and makes the code even more concise: the use of CViews reduced the number of lines in the *Mozlog* version of Chord by 8 rules (from 43 to 35).

4.5. Legacy Application Support

Mozlog also supports a built-in `tun` predicate specifically reserved for representing tunneled traffic via the *tun* virtual network device. This allows legacy applications listening on the *tun* device to seamlessly tunnel traffic through MOSAIC overlay compositions. The `tun` predicate has the following schema: `tun(IPPkt [,SrcIP, DestIP, Protocol, TTL])`. `IPPkt` represents the IP packet that is being tunneled. In addition, the IP header fields `SrcIP`, `DestIP`, `Protocol` and `TTL` are optionally extracted and included as additional attributes when they are required in *Mozlog* rules. The following rules demonstrate the `tun` predicate for tunneling via a point-to-point and *i*3 overlay respectively:

```
p2p_tun tun@Peer(Pkt) :- tun(Pkt), Peer = "12.34.56.78:1086".
i3_tun tun@Peer(Pkt) :- tun(Pkt, Src, Dest), Key = f_sha1(Dest),
                         Peer = i3_oid::Key.
```

Rule `p2p_tun` sets up a point-to-point UDP tunnel between the local node and the remote node listening at the UDP address `12.32.56.78:1086`. This allows legacy applications at two end-points to communicate via a UDP tunnel implemented by MOSAIC. Similarly, in rule `i3_tun`, a tunnel is set up via the *i*3 overlay. All packets generated by the legacy application are sent via this rule to a remote legacy application running at the *i*3 node with logical address `Key` generated using the SHA-1 hash of the destination tunneling address. See Section 7.2 for implementation details.

5. Compiling Compositions

This section describes how the MOSAIC compiler automatically translates composition specifications into *Mozlog* rules. We first define the following reserved tables stored at each node, which are used in the composition process later:

- **`netAddress(OID,Addr)`** tracks all current addresses `Addr` of the overlays `OID` in which the local node participates. If a node `n` has a publicly reachable IP address, a default entry is added as `netAddress(0,current_ip)`, where 0 is a reserved ID for the Internet. `OID` can also refer to a bridged network, in which case `Addr` can refer to a source routing address (See Section 5.3). Other overlay specific addresses are maintained by the corresponding overlay modules.

- **`underlay(OID,Addr)`** is used in layering. It stores the mapping from an overlay's `OID` to its current underlay's runtime address `Addr` at the local node for each deployed overlay. By updating this table, one can switch the underlay being used.

- **`forward(OID,Addr)`** is used in bridging. It specifies that all packets designated for overlay `OID` are to be sent to the designated gateway with address `Addr`.

5.1. Compilation Steps

To create an overlay network composition from scratch, the MOSAIC compiler takes as input a composition specification as presented in Section 3.1, and then generates *Mozlog* rules that bridge and layer the appropriate overlay modules.

Algorithm 5.1 Pseudocode for composition process

```
1  input: spec as the composition specification
2
3  //Layering
4  for l in spec.composition.links:
5      match l with
6      Layering top over bottom:
7          if not top.nodelist ⊆ bottom.nodelist:
8              raise Exception("node sets are wrong for layering")
9          layering(top, bottom)
10
11 //bridging
12 for l in spec.composition.links:
13     match l with
14     Bridging first With second by type via gw as name:
15         //bridging two overlays
16         assert(gw ∈ first.nodelist and gw ∈ second.nodelist)
17         bridging(first,second,gw,type,name)
```

Algorithm 5.1 is pseudocode that illustrates this process. For ease of exposition, we base the translation on an abstract syntax of the composition specifications presented in Appendix B. This abstract syntax can be constructed by processing the earlier XML-based input in Appendix A. For instance, *spec.composition.module* in the abstract syntax refers to the corresponding child element *module* within the *composition* element of the XML tree.

In our pseudocode, we use a pattern matching syntax (keyword *match*) to examine and extract composition information. For instance, given a composition link *l*, a bridging-based link matches an initial keyword "Bridging", followed by the *first* overlay, the *second* overlay, and the gateway `gw`.

Lines 4-9 perform a check on the layering configuration, followed by the actual layering itself. For each layering link in the input specifications, the main constraint to be met is that the overlay nodes are also members of the underlay network. Here, we check that the nodes in the underlay network are a superset of nodes in the overlay. We use the pattern matching syntax to extract the overlay as *top* and the underlay as *bottom* for any links that involve layering. Line 9 performs the actual task of bridging after constraint checks are performed. This involves calling additional functions `layering`, shown as Algorithm 5.2 in Section 5.2.

Lines 11-16 check that all constraints are met for composition links that involve bridging multiple modules, followed by the actual bridging itself. If two modules are bridged via a gateway, this gateway node must belong to the node list of both modules. After the check is performed, the actual bridging is performed by invoking the `bridging` function, shown as Algorithm 5.3 in Section 5.3.

After the compilation, the rules are shipped to the corresponding physical nodes for deployment. To modify an existing network composition, most of the procedures remain the same except that the node membership sets of existing overlays are obtained by querying the directory service, and modified *Mozlog* rules are uploaded to the physical nodes to implement the new composition.

5.2. Layering

Algorithm 5.2 Pseudocode for layering-related rule generation

```
1  function layering(top, bottom):
2  input: overlay top, overlay bottom  // layer top over bottom
3  output: rules for layer bindings
4  for n in top.nodelist:
5      if top.oid ∉ n.deployed:  // network top is not deployed on node n
6          code = fetch(top.codeurl)
7          addRule(n, code)
8          addRule(n, "underlay(top.oid, Addr) :-
9                          netAddress(bottom.oid, Addr).")
10     else:
11         updateRule(n, "underlay(top.oid, Addr) :-
12                            netAddress(bottom.oid, Addr)")
```

Layering of a control or data plane over another overlay's data plane is achieved through the use of the `underlay` table describing bindings from each overlay node to its current runtime underlay address. Abstracting the bindings into a table provides a simple mechanism for switching overlays: MOSAIC can simply update the `underlay` table — changing both the underlay protocol and node address accordingly.

Given a composition specification with layering links, *Mozlog* rules are generated to implement the layering. Algorithm 5.2 shows the pseudocode for rule generation. The function `addRule(`n`,`r`)` is used to add a rule r to the node n. If executed at the composition service, this means that all generated rules have to be shipped from the service to be instantiated at each corresponding node.

If the overlay *top* is not deployed on the node n, we first add the overlay *Mozlog* implementation to the code to be deployed on node n (Lines 6-7), then bind the address of network *bottom* to the underlay table entry that belongs to network *top* (Line 8). Note that symbols in italic fonts, including n, *top.oid*, and *bottom.oid* are constants when added into the rule. If *top* is deployed, then there should be a rule that binds its `underlay` table already. We update that rule to the new binding (Lines 11-12).

We illustrate using an example where there are two RON overlays, layered over IP and *i*3. Based on the specifications, at every node, there are two instances of RON executing (`ron_oid1` and `ron_oid2`), and one instance of *i*3 (`i3_oid`). The following *Mozlog* rules `b1` and `b2` are generated to build the two networks:

```
b1 underlay(ron_oid1, U) :- netAddress(0, U).
b2 underlay(ron_oid2, U) :- netAddress(i3_oid, U).
```

Since `ron_oid1` utilizes IP for routing, rule `b1` takes as input `netAddress(@N,0,U)`, based on the executing node's default IP address. On the other hand, `ron_oid2` routes over *i*3, hence its underlay tuple stores the address of the underlying `i3_oid` node retrieved from the local `netAddress` table.

Note that the layering association is not static. A deployed, running overlay network can switch the underlying network from one to another by updating its underlay table entries at runtime. This enables dynamic overlay composition. We will discuss an example of dynamic switching in Section 6.

Next, the rule to update the `netAddress` table is generated for the newly created overlay. Because the rule is overlay specific, it is not automatically generated by the compiler, but provided by the overlay programmers. For example, consider the *i*3 and RON overlays with identifiers `i3_oid` and `ron_oid` respectively. In *i*3, its overlay address is the SHA-1 hash of the node's public key `K` (as shown in rule `d1`).

```
d1 netAddress(i3_oid, A) :- publicKey(K), A = i3_oid::f_sha1(K).
```

On the other hand, in RON, since its routable address is tightly coupled with its underlay, its address is its own underlay address (typically the IP address that RON uses) annotated with the overlay id as shown rule `d2`:

```
d2 netAddress(ron_oid, A) :- underlay(ron_oid,U), A = ron_oid::U.
```

Finally, data plane forwarding rules may also need to be slightly changed. We update the RON forwarding rules `snd` and `rcv` from Section 4.3 in the context of layering:

```
snd ron_oid.send@Next(Dest,Packet) :- ron_oid.send(Dest, Packet),
                                      ron_oid.RT(Dest, Next),
                                      underlay(ron_oid, Local),
                                      Local != Dest.
rcv ron_oid.recv(Packet) :- ron_oid.send(Dest, Packet),
                            underlay(ron_oid, Local),
                            Local = Dest.
```

The local address stored in `localAddr` is replaced by `underlay(ron_oid,Local)`, where `Local` is the current underlay address of the overlay `ron_oid`. Note that while the above rules achieve the same functionality as the previous two rules in Section 4.3, they are more flexible in allowing packets to route over underlays that can be switched at runtime.

Algorithm 5.3 Pseudocode for bridging-related rule generation

```
function bridging(first, second, bridgetype,gw, bn):
input: overlay first, overlay second, gateway gw, bridge name bn
output: bridging related rules

gwAddr = dns_lookup(gw)

for n in first.nodelist:
  if $n \neq gw$:
    if bridgetype="on-demand":
      addRule(n, "netAddress(bn, Addr) :-
                    netAddress@gwAddr(second.oid, SecondGWAddr),
                    netAddress(first.oid, FirstNodeAddr),
                    Addr = sr::[SecondGWAddr, FirstNodeAddr].")
    else: //pre-configured method
      addRule(n, "forward(second.oid, FirstGWAddr) :-
                     netAddress@gwAddr(first.oid, FirstGWAddr).")

for n in second.nodelist:
  if $n \neq gw$:
    if bridgetype="on-demand":
      addRule(n, "netAddress(bn, Addr) :-
                     netAddress@gwAddr(first.oid, FirstGWAddr),
                     netAddress(second.oid, SecondNodeAddr),
                     Addr=sr::[FirstGWAddr, SecondNodeAddr].")
    else: //pre-configured method
      addRule(n, "forward(first.oid, SecondGWAddr) :-
                     netAddress@gwAddr(second.oid, SecondGWAddr).")
```

5.3. Bridging

Algorithm 5.3 shows the pseudocode for bridging two overlays named *first* and *second* via a gateway node `gw`. In the pseudocode, we assume that the gateway `gw` is reachable by the Internet which serves as the management channel, hence its address `gwAddr` can be retrieved via DNS lookup on `gw`. Note that since `gw` participates in both networks, it has two overlay nodes with addresses `firstGWAddr` and `secondGWAddr`, one for each participating overlay that it bridges. As a result, at node `gw`, it maintains two `netAddress` entries for each one of its participating overlay node, i.e. entries `netAddress(first.oid, firstGWAddr)` and `netAddress(second.oid, secondGWAddr)`. In addition, each node maintains a `netAddress` entry for the overlay it participates in. For instance, a node `firstNodeAddr` in the *first* overlay will include an entry `netAddress( first.oid, firstNodeAddr)` in its local `netAddress` table.

We focus on explaining the pseudocode from the perspective of the *first* overlay (lines 7-16). The explanation for the *second* overlay is symmetrical. We first describe the *Mozlog* rule generated for the *on-demand* method (lines 9-13). Recall from Section 3.2.2 that in the on-demand method, source routing is used to create an explicit path from a node in one network to another node in a bridged network, via an intermediate gateway node. Our generated *Mozlog* rule (lines 10-13) distinguishes local predicates at each node `n` from remote predicates at the gateway (indicated by `@gwAddr`).

Unlike the *pre-configured case* discussed below, since source routing is used, no explicit forwarding state need be maintained. Instead, the main goal is to set up the appropriate `netAddress` entries for the new bridge such that source routes can be later created in order to route to nodes within the bridge networks. *Mozlog* supports a source routable address type of the form `sr::[gateway, dest]`, which explicitly describes the data path to `dest` via an intermediate gateway node. All nodes will automatically handle the forwarding of such messages to the next recipient in the path as specified in the source route. For instance, at overlay *first*, its `netAddress` table for the bridge `bn` maintains a source address `sr::[SecondGWAddr,FirstNodeAddr]`, which indicates that in order for a node in the *second* overlay to reach the node with address `FirstNodeAddr` in the *first* overlay, the packet has to first traverse to the gateway node `SecondGWAddr` in the *second* overlay, before the packet can be sent to the destination node `FirstNodeAddr`.

In the *pre-configured* method (lines 14-16), the address of the gateway `gw` for overlay `oid` is stored in a local `forward` table at each node. For instance, at the *first* overlay, its local `forward(second.oid, firstGWAddr)` table indicates that in order to route to the overlay *second*, the packet has to be first routed to `gw` node, specifically the `firstGWAddr` overlay node on the gateway that is also participating in the *first* overlay. When the packet reaches `firstGWAddr`, it is then forwarded to the appropriate node in overlay *second* based on the destination address stored in the packet. Unlike the earlier on-demand method, no source routing is required.

6. Composition Examples

MOSAIC can support flexible overlay compositions including bridging, layering and hybrid compositions. We present two examples, one that revisits the mobile VoIP example introduced in Section 1, and a second that illustrates dynamic composition. Appendix A gives an XML composition specification based on the first example.

overlay id	address
alice_net	alice_internal_ip
br1	sr::[alice_gateway_ip, alice_internal_ip]
br2	sr::[ron_oid::alice_gateway_ip, alice_internal_ip]
i3_oid	i3_oid::alice_id

Table 1: netAddress table at Alice

overlay id	address
0	alice_gateway_ip
alice_net	alice_gw_internal_ip
ron_oid	ron_oid::alice_gateway_ip

Table 2: netAddress table at Alice's gateway

6.1. VoIP between Alice and Bob

Consider the example of Section 1. An overlay composition can solve the problem. Suppose there is a publicly available *i*3 overlay network, and Alice uses her gateway node at home to form a private RON network with Bob and her other friends. Alice and Bob agree on the composition specification shown in Figure 4. Based on the overlay specification, MOSAIC generates the *Mozlog* rules to compose overlays together.

Because Alice's situation mirrors Bob's, we use Alice's rules and network state to explain the composition process. First, at Alice's gateway, we configure the RON overlay network over IP as:

```
c1 underlay(ron_oid,A) :- netAddress(0,A).
```

We then use bridging to create publicly reachable addresses `br1` and `br2` as shown in Table 1. `br1` bridges the internal network AliceNet with the public IP network, and `br2` bridges AliceNet with the RON network.

Finally, we layer *i*3 over the bridged networks we created. Because Alice wants to have reliability for VoIP, we choose the bridging overlay with `br2` as *i*3's underlay. The composition rule deployed at the Alice node is as follows:

```
c2 underlay(i3_oid,A) :- netAddress(br2,A).
```

When Bob initiates a VoIP call to Alice, he first uses Alice's *i*3 ID to look up her public trigger, and sends traffic to Alice via *i*3's indirection path. After they have located each other, they switch to the *i*3 shortcut data path as the underlay network specifies, which is layered on top of RON and can traverse internal networks (*e.g.*, those behind NATs) using source routing along the gateways.

6.2. Dynamic Composition of Chord over IP and RON

To illustrate dynamic composition, we use the Chord DHT to show the benefit of dynamically switching the underlying data path from IP to RON. In Chord, temporary network failures may create non-transitive connectivity between the nodes, possibly creating problems such as invisible nodes, routing loops and broken return paths [11]. Instead of altering the DHT protocol, an alternative is to layer Chord over a resilient routing protocol such as RON that eliminates non-transitivity. Layering Chord over RON can be viewed as trading scalability for performance.

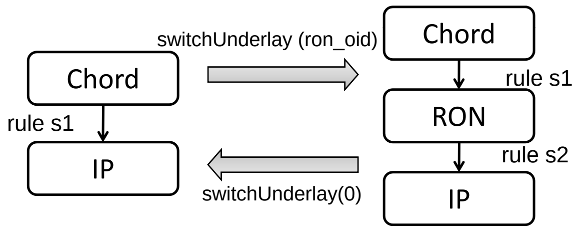

Figure 5: Dynamic composition of Chord over two different underlays (IP and RON).

The following rules define two types of layering: Chord over IP and Chord over RON (See Figure 5):

```
s1 underlay(chord_oid,A) :- netAddress(OID,A), switchUnderlay(OID).
s2 underlay(ron_oid,A) :- netAddress(0,A).
```

In `s1-s2`, we added a `switchUnderlay(OID)` predicate to switch Chord's underlay to that indicated by the `OID` variable. This `switchUnderlay` can itself be triggered by an event sent from the administrator based on changes to the overlay specifications. Rule `s1` indicates that Chord uses IP as the underlying address when `OID` is 0, and RON when `OID` is `ron_oid`. Rule `s2` defaults RON to use IP at all times. To switch between the two layering schemes, one only needs to generate `switchUnderlay` accordingly.

Dynamic switching is useful because the trade-off between scalability and performance is at the discretion of the Chord administrators, who can make decisions based on network conditions, requirements, etc. If a new overlay providing both resiliency and scalability (*e.g.* SOSR [14]) becomes available, one can switch Chord's underlay from RON to the new one to further improve scalability. Unlike restarting Chord from scratch, dynamic switching preserves existing state in the network such as key/value pairs without disrupting the DHT lookup service. Once the Chord underlay network address is changed on a node, the stabilization process will propagate it to the node's successors, predecessor and other nodes that have it in its finger table. We present our experimental evaluation of this example in Section 8.3.

7. Implementation

The MOSAIC platform builds on the P2 [21] declarative networking system, adding significant new functionalities. The P2 planner and dataflow engine have been revised to generate execution plans that accommodate new language features of *Mozlog*: specifically, those related to runtime support for dynamic location specifier, data plane forwarding, and interactions with legacy applications.

MOSAIC takes a *Mozlog* program, compiles it into distributed P2 dataflows [21], and deploys it to all nodes that participate in the overlay. A single node may host multiple overlay networks at the same time. P2 dataflows resemble the execution model of the Click modular router [18], which consists of elements that are connected together to implement a variety of network and flow control components. In addition, P2 elements include database operators such as joins, aggregation, selections, and projections that are directly generated from queries. Each local dataflow participates in a global, *distributed* dataflow across the network, with messages flowing among elements at different nodes, resulting in updates to local tables. The local tables store the state of intermediate and computed query results, including structures such as routing tables, the state of various network protocols, and data related to their resulting compositions. The distributed dataflows implement the operations of various network protocols. The flow of messages entering and leaving the dataflow constitute the network packets generated during query execution.

7.1. Dataflow Execution

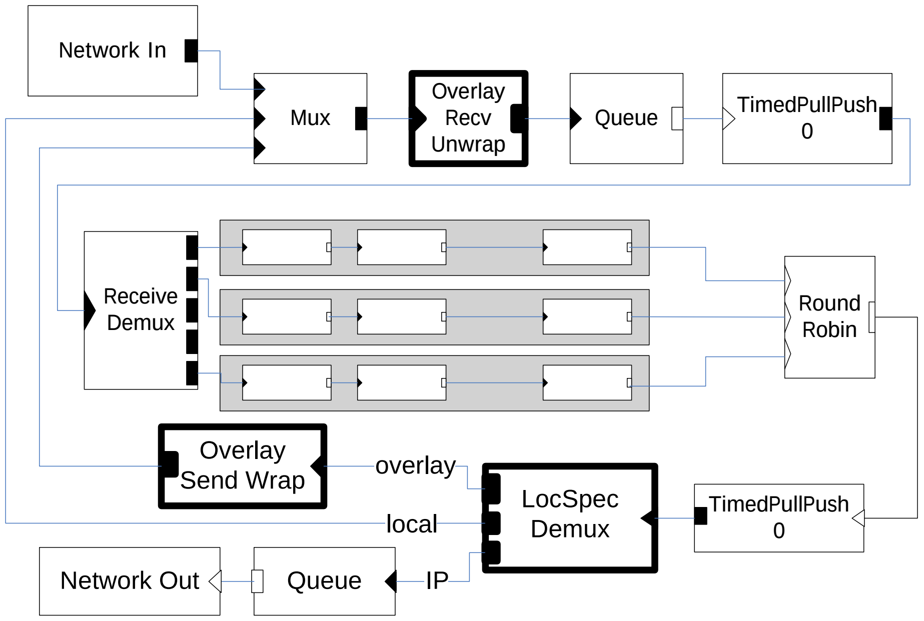

Figure 6: System dataflow & dynamic location specifiers.

Figure 6 shows a typical execution plan generated by compiling *Mozlog* rules. Similar to P2 dataflows, there are several network processing elements (`Network In` and `Network Out`) that connect to individual rule strands (inside the gray box) that correspond to compiled database operators. Here, we focus on our modifications, and the interested reader is referred to [21] for details on the dataflow framework.

To implement dynamic location specifiers and overlay forwarding on the data plane, we modify the planner to automatically generate three additional MOSAIC elements shown in **bold** in the dataflow: `OverlayRecvUnwrap`, `OverlaySendWrap`, and `LocSpecDemux`. The elements `OverlayRecvUnwrap` and `OverlaySendWrap` are used for de-encapsulation and encapsulation of tuples from overlay traffic.

At the top of the figure, the `Mux` multiplexes incoming tuples received locally or from the network. These tuples are processed by the `OverlayRecvUnwrap` element that extracts the overlay payload for all tuples of the form `overlay.recv(Packet)`, where `Packet` is the payload with type tuple. Since the payload may be encapsulated by multiple headers (for layered overlays), this element needs to be "unwrapped" until the payload is retrieved. The `Packet` payload is used as input to the dataflow via the `ReceiveDemux` element, which itself is then used as input to various rule strands for execution.

Executing the rule strands results in the generation of output tuples that are sent to a `LocSpecDemux` element. This element checks the runtime type of the location specifier, and then demultiplexes as follows:

- Tuples `tuplename(F1, F2, ..., Fn)` are local tuples and sent to the `Mux`.
- Tuples `tuplename@IPAddr(F1, F2, ..., Fn)` are treated as regular IP-based tuples and sent to the network directly.
- Tuples `tuplename@oid::ovaddr(F1, F2, ..., Fn)` are designated for overlay network `oid` with address `ovaddr`. A new event tuple `oid.send(ovaddr, tuplename(F1, F2, ..., Fn))` which denotes the `send` primitive of the overlay network `oid` is generated (see Section 4.3). This new tuple is reinserted back to the same dataflow to be forwarded based on the overlay specification.

7.2. Legacy Support

We use the *tun* device to provide overlay tunnels between legacy applications at the network layer. The `tun` predicate is reserved for legacy support, and tuples using it are treated differently from ordinary tuples in the dataflow. Each special predicate has a rule strand in the dataflow, between the `ReceiveDemux` element and the `RoundRobin` element (see Figure 6). Two elements `Tun::Tx` and `Tun::Rx` are inserted in the `tun` rule strand right after `ReceiveDemux`. `Tun::Rx` reads IP packets from the *tun* device, generates the `tun` tuple, and sends to the next element in the rule strand; `Tun::Tx` receives a `tun` tuple, formats it as an IP packet and writes to the *tun* device.

For each end host, it takes a private IP address from 1.0.0.0/8 to avoid conflicts with other public IP networks. After a legacy application sends a packet to an address in the *tun* network, the kernel redirects it to MOSAIC, where the `Tun::Rx` element generates a `tun` tuple. Currently there is an address translation rule that uses a special mapping table to translate the private IP address to the overlay address. This can be extended in the future to use any name resolution service by combining DNS request hijacking [15]. After address translation, the packet tunneling rules such as the rule `i3_tun` of Section 4.5 deliver the IP packet to the destination via the corresponding overlays. After the `tun` tuple is delivered to the remote node, it is redirected to the *tun* device by the element `Tun::Tx`, and finally the tunneled packet is received by the legacy application.

To support a legacy overlay that is not implemented in MOSAIC, we build an adapter for the overlay to interact with MOSAIC via the `send` and `recv` primitives. The adapter redirects the `legacy.send` tuple from the dataflow to the overlay, and injects the `legacy.recv` tuple upon overlay's packet reception. Because the legacy overlays are built on IP, they can only be bridged with other overlays or used as substrates underneath other networks, but cannot be layered on top of another overlay for either the control or the data plane.

7.3. Compilation of CViews

The *Mozlog*-to-*NDlog* translator rewrites and expands all CView rules into *NDlog* rules, which can then be compiled into dataflow strands using the P2 planner. The compilation process involves a query rewrite that takes as input all CView predicates, and expands them into multiple *NDlog* rules based on their view definitions.

Since this process resembles function call compilation, we reuse the terms *caller* and *callee*. A rule that takes an input CView predicate is the *caller*. The set of rules based on the view definition (*e.g.*, rules `p1-p3` in Section 4.4) comprises the operations of the *callee*.

In a typical C compiler, the caller maintains a stack, pushing local variables (execution context) and the return address before a call. Similarly, for each CView input predicate `viewName[@locSpec] (K1,...,Kn, &R1,...,&Rm)`, the execution context is all the bound variables `K1,...,Kn` and the variables that appear in the rule body before the CView term. The expanded rules are executed, and the local variables are stored in a designated internal context table. This local state is stored for the duration of view execution. Each expanded set of rules replaces the `this` prefix in the original view definition with a query context identifier `CID` that uniquely identifies the current invocation of the view, and a return address `RetAddr` of the caller. When the caller has finished executing all the rules for the view, the results are returned to the caller (`RetAddr`).

We use the following `ping` CView presented in Section 4.4 to demonstrate the CView compilation process. The following rules show the compilation result from the ping module. All predicates within the CView are appended with two fields, `CID` as the query context identifier and `LVReturnAddr` as the return address to the caller.

```
ping_p1 ping_pingReq(@RI,NI,T,CID,LVReturnAddr):-
          ping_init(@NI,RI,CID,LVReturnAddr), T = f_now().

ping_p2 ping_pingResp(@RI,T,CID,LVReturnAddr):-
          ping_pingReq(@NI,RI,T,CID,LVReturnAddr).

ping_p3 ping_return(@LVReturnAddr,Delay,CID):-
          ping_pingResp(@NI,T,CID,LVReturnAddr), Delay = f_now()- T.
```

Suppose the caller rule is

```
r1 pingResult(@NI,RI,Delay):- periodic(@NI,E,2),
    RI=DESTADDR, ping(@NI, RI, &Delay).
```

Rule `r1` periodically measures the RTT to the destination node `RI`. The translation result is showed as follows:

```
r1_cxt r1_ctxt(@NI,CID,E,RI) :- periodic(@NI,E,2),
                                RI=DESTADDR,
                                CID = f_rand().
r1_init ping_init( @NI,RI,CID,NI) :- r1_ctxt(@NI,CID,E,RI).
r1_return pingResult(@NI,RI,Delay):- r1_ctxt(@NI,CID,E,RI),
                                     ping_return(@NI,Delay,CID).
```

First, a table `r1_ctxt` is generated to store query identifiers and query context (bound variables before the CView term) locally. Second, rule `r1_cxt` generates a unique query identifier `CID` and saves the context variables (`NI`, `E`, `RI`). Then, rule `r1_init` invokes the ping CView, and finally rule `r1_return` takes the return tuple from the ping CView, which is joined with the saved context, and emits the result `pingResult`.

8. Evaluation

In this section, we present the evaluation of MOSAIC on a local cluster and on PlanetLab. First, we validate that *Mozlog* specifications for declarative networks, compositions, tunneling and packet forwarding are comparable in performance to native implementations. Second, we use our implementation to demonstrate feasibility and functionality, using actual legacy applications that run unmodified on various composed overlays using MOSAIC. Third, we evaluate the dynamic composition capabilities of MOSAIC.

In all our experiments, we make use of a declarative Chord implementation which consists of 35 rules. Our *i*3 implementation uses this Chord and adds 16 further rules. We also implement the RON overlay in 11 rules. Both *i*3 and RON can be used by legacy applications via the *tun* device, as described in Section 4.5.

8.1. LAN Experiments

To study the overhead of MOSAIC, we measured the latency and TCP throughput between two overlay clients within the same LAN. The experiment setup was on a local cluster with eight Pentium IV 2.8GHz PCs with 2GB RAM running Fedora Core 6 with kernel version 2.6.20, which are interconnected by gigabit Ethernet. While the local LAN setup and workload is not typical of MOSAIC's usage, it allows us to eliminate wide-area dynamic artifacts that may affect the measurements. We measured the latency using `ping` and TCP throughput using `iperf`.

In the experiments, we use the *tun* device to provide legacy application support for network layer overlays. MTU was reduced to 1250 bytes to avoid fragmentation when headers were added. The measurement results are shown in Table 3 for the following test configurations:

DirectIP: Two nodes communicate via direct IP, where `iperf` can fully utilize the bandwidth of the Gigabit network. This serves as an indication of the best latency and throughput achievable in our LAN.

test	latency(ms)	throughput (KByte/s)
DirectIP	0.10	97994
OpenVPN	0.30	13951
MozTun	0.50	8353
RON	0.71	5796
i3	1.31	3299

Table 3: Overhead comparison in LAN

OpenVPN: OpenVPN [1] is a widely used tunneling software system. Using OpenVPN version 2.0.9, we set up a point-to-point tunnel via UDP between two cluster nodes and disabled encryption and compression. The performance results provide a baseline for the overhead using the *tun* device virtualization. Compared to DirectIP, the latency increases by around 0.2*ms*, and the TCP throughput drops by a factor of more than 6. This overhead is inevitable for all overlay networks supporting legacy applications using the *tun* device, including those hosted on MOSAIC.
MozTun: We set up a static point-to-point tunnel in MOSAIC between two cluster nodes. MozTun and OpenVPN essentially have the same functionality except that MozTun is implemented in MOSAIC. The additional overheads in throughput and latency are solely from the MOSAIC dataflow processing overhead bounded by CPU capacity. In MozTun, the latency increased 0.20*ms* over OpenVPN, which is negligible when executed over wide-area networks.
RON: We ran the RON network using MOSAIC and utilize two nodes to run the measurements. Since RON does not provide any benefit in our LAN setting as there are no failures, the comparison to MozTun is used to show the extra overhead for rule processing in our implementation.
i3: Six nodes were set up as *i*3 servers, using Chord to provide lookup functionality. The remaining two nodes were selected as *i*3 clients. A packet sent by the source *i*3 client to the destination *i*3 client went through the public trigger of the destination, which was hosted on the *i*3 server of another cluster node. Since it introduced a level of indirection plus extra rule processing overhead, *i*3 added the most cost among the 5 configurations studied.

In summary, the overhead of MOSAIC is respectable: the throughput of MOSAIC's point-to-point tunneling (MozTun) is comparable to that obtained by using well-known tunneling software (OpenVPN). In the extreme case (the additional level of indirection of *i*3 with tunneling), the additional latency (1.2*ms*) incurred is negligible for an application running on wide-area networks. Later, in Section 8.2, we will validate the performance of a composed overlay on the Planetlab testbed.

8.2. Wide-area Composition Evaluation

We deployed MOSAIC on PlanetLab to understand the wide-area performance effects of using the system. We purposely chose a composed overlay including *i*3, RON, source routing, and tunneling for legacy applications (all implemented within MOSAIC in 69*Mozlog* rules) to bring the Alice example from the introduction and Section 6.1 to a resolution.

Our experimental setup is as follows. As our end-host, we used a Linux PC in New Jersey with a high speed cable modem connection (4Mbps downstream speed) as the gateway node, which performed NAT for a Thinkpad X31 laptop. The laptop functioned as our server, using Apache to serve a 21MB file. The file was downloaded from a machine in Utah with a modified version of `wget` that records the download throughput.

These two nodes in New Jersey and Utah, plus three additional nodes (two on the East Coast of the US, and one on the West Coast of the US), were used to form a private RON network. We further selected 44 PlanetLab nodes, mostly in the US, to run *i*3. During the experiment, we validated the functionality of resilient routing provided by RON by manually injecting network failures via changing the firewall rules on the gateway to block the downloader's traffic 30 seconds after `wget` was started; then we unblocked the traffic after another 30 seconds. For the purposes of comparison with the best case scenario, we repeated the same test using direct IP communication. Note that direct IP loses all the benefits of our composed overlay (no resilience, NAT, or mobility support), but achieves the best possible performance. Since our server was behind a NAT, in the direct IP experiment, we had to manually set up a TCP port forwarding rule on the gateway node to reach the Apache server. We repeated multiple runs of the experiments and observed no significant differences.

Figure 7 shows the throughput of the download over time for MOSAIC and DirectIP. Network failures were injected 30 seconds after experiment start, and removed after 30 additional seconds. We make the following observations. First, MOSAIC's performance over the wide area is respectable. Despite implementing the *entire* composed overlay (including legacy support for applications using MOSAIC) in *Mozlog*, we incurred only 20% additional overhead compared to using direct IP, while achieving the benefits of mobility, NAT support and resilient routing. The majority of the overhead comes from the extra packet headers for the composed overlay protocols—an overhead that is repaid with significant functionality. Second, with respect to the functionality of our composed overlay, we were able to achieve successful downloads from a server behind a NAT using MOSAIC. In addition, resilient routing was achieved: the RON network periodically monitored the link status and recovered from routing failures. Hence, during the period where we injected the routing failures, MOSAIC was able to make a quick recovery from failure, as is shown by the sustained throughput. On the other hand, DirectIP suffered a failure (and hence a drop of throughput to zero) during the 30-60 second period. Overall, MOSAIC finished the download in a shorter time despite lower throughput, due to the resiliency of RON.

8.3. Dynamic Overlay Composition

In our final experiment, we evaluate the dynamic composition capabilities of MOSAIC. Our setup consists of an 8-node cluster, where each node has a hardware configuration similar to the setup in Section 8.1.

As a baseline prior to the dynamic switching experiment, we made static comparisons between two composed networks: we executed *Chord-over-IP* and *Chord-over-RON* on our cluster, which consists of the Chord overlay on top of IP and RON respectively. Our network size is 16, where each machine executed two instances of the composed overlay nodes. In the steady state, each node periodically issues a lookup request. A lookup is *accurate* if the results of the lookup are correct, *i.e.*, the results point to the node whose key is the closest successor of the lookup key. Based on this definition, we compute the lookup accuracy rate, which is the fraction of accurate lookups over the duration of each experimental run at every 1 minute interval. Network link failures are emulated by changing the firewall settings in the cluster to drop packets between the selected nodes.

Figure 8 shows our evaluation results over a period of 20 minutes, with the first link failure at the 7th minute, then the second link failure at the 10th minute, and the failures recovered at the 16th minute. When the first link failure occurred, we observed that lookup accuracy of *Chord-over-IP* dropped to 93%. The accuracy further dropped to 86% when the second link failure occurred, only to recover when network connectivity was reestablished. On the other hand, *Chord-over-RON* continued to sustain high lookup accuracy ($> 99\%$) even in the face of network failures, due to its ability to find alternative routes quickly.

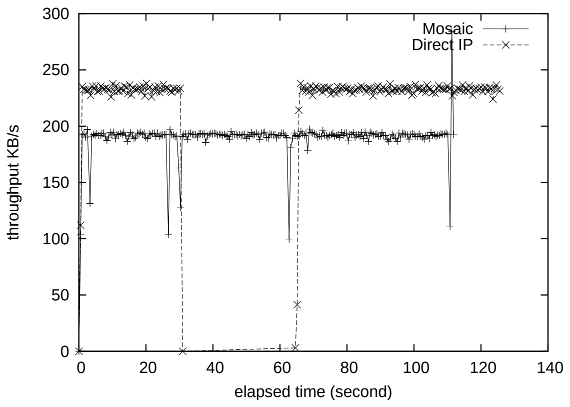

Figure 7: Throughput comparison between overlay composition in Mosaic vs direct IP connection during network failure.

Having compared the composed overlays separately, we next evaluate MOSAIC's dynamic switching capability, where we started with *Chord-over-IP*, and then switched our composition to *Chord-over-RON* after 7 minutes. This dynamic switching is achieved by merely changing the underlay address of Chord from IP to RON, as described in Section 6. Figure 9 shows the resulting lookup accuracy over a period of 15 minutes. We observe that during the process of switching its underlay from IP to RON, Chord continued to sustain high lookup accuracy, demonstrating that MOSAIC is able to performing dynamic switching seamlessly.

9. Related Work

Composing a plurality of heterogeneous networks was proposed in Metanet [41], and also examined in Plutarch [8]. One of the implementation examples to connect

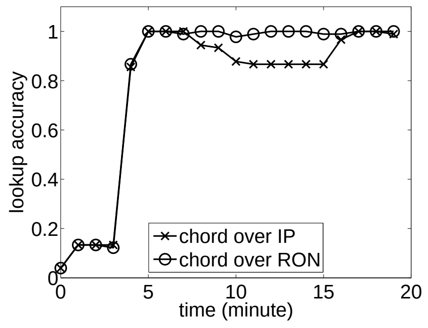

Figure 8: Lookup accuracy comparison between Chord over IP and Chord over RON.

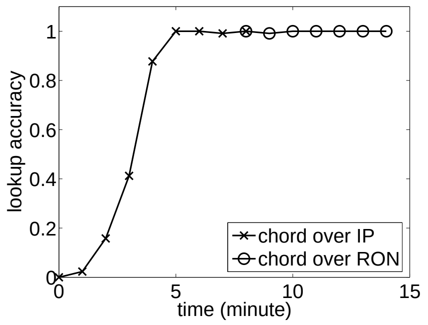

Figure 9: Chord lookup performance during dynamic underlay network switching from IP to RON.

multiple networks together is AVES [30]. Oasis [24] and OCALA [15] provide legacy support for multiple overlays. Oasis picks the best single overlay for performance. OCALA proposes a mechanism to *stitch* (similar to MOSAIC's bridge functionality) multiple overlay networks at designated gateway nodes to leverage functionalities from different overlays. Some projects focus on application level service composition, with different emphases, such as adaptive configuration based on network conditions [12], fault tolerance and personalization [27] and performance and QoS awareness in P2P environments [13].

In contrast, MOSAIC's primary focus is on overlay network specification and composition within a single framework. As a result, MOSAIC is complementary to OCALA and Oasis. MOSAIC's use of a declarative language results in more concise overlay network specification and composition, whose performance is quite comparable to native code. MOSAIC also provides support for layering in addition to bridging. Finally, MOSAIC is not limited to IP-based networks, supports dynamic composition, and routing primitives such as unicast and multicast. These benefits result in better extensibility and evolvability of MOSAIC over existing composition systems.

Another class of composition work is *Web service composition*, as surveyed by Milanovic and Malek [28]. Each Web service serves like a remote procedure call over HTTP or HTTPS, and provides a relatively basic functionality, which is described with additional pieces of information, either by a semantic annotation of what it does and/or by a functional annotation of how it behaves. The industry standard specification is Web Service Definition Language (WSDL) [6]. A complicated application logic is built by composing multiple Web services together, which is described separately in a flow specification, such as Business Process Execution Language for Web Services (BPEL4WS) [3]. The process of obtaining the composition flow is either manual or in some cases can be assisted by an AI planning software in the context of semantic Web.

Compared with network service composition, the major difference is that Web services are best described with request/response models, where usually only a single entity is involved in utilizing a service (the service may still be provided by multiple providers). On the other hand, network services are based on send/receive models, where two (a sender and a receiver) or more (in the case of multicast/broadcast) entities participate in the process. Goal-based AI planning work and automatic composition in Web services may provide a viable path towards automatic network composition.

In contrast to the declarative networking concept [22, 21], on which it is built, MOSAIC not only uses the declarative language to do quick prototyping of new overlay protocols, but also achieves interoperability among existing overlays by using simple yet flexible query-style interfaces between networks. This allows the system to provide automation in composing different networks together to implement a combination of multiple features. The *Mozlog* language provides new features to support composition, as well as legacy application support. Finally, the MOSAIC runtime system performance is optimized by both novel compiler-based techniques and careful engineering efforts so that query executions are up to 10 times faster than P2.

System	**Domain**	**Composition**	**Abstraction**	**Dynamic**
MOSAIC	Overlay and IP	Layer,Bridge	High (declarative)	Yes
Web services	WWW applications	Layer,Bridge	Varies	No
Oasis	Overlay	Bridge	Low	No
OCALA	Overlay	Bridge	Low	No

Table 4: Comparison of related work.

Table 4 compares MOSAIC against 3 representative systems along the dimensions of *domain* (layer in the software stack), *composition* (layering vs bridging or both), level of *abstraction* in the composition specification language, and whether the compositions can be dynamically adapted at runtime. With the exception of Web services composition, all of the systems that have surveyed target the overlay networks at the application layer. In a clean-slate Internet design, MOSAIC has the advantage that it can be deployed at the IP layer as well. Oasis and OCALA do not support layering, while MOSAIC and most Web service composition frameworks allow both layering and bridging of components. In terms of language abstractions, MOSAIC is the most declarative, via the use of *Mozlog*. WSDL is also a highly declarative language used in Web service compositions, although the range of abstractions can vary greatly in Web services. Finally, MOSAIC provides dynamic runtime adaptation of compositions based on changing network and application conditions. This feature is not present in the other systems.

10. Conclusions and Future Work

MOSAIC is an extensible infrastructure that enables the specification and dynamic composition of new overlay networks. MOSAIC provides *declarative networking*: it uses a unified declarative language (*Mozlog*) to specify new overlay networks, and a novel runtime to enable composition in both the control and data planes. We have demonstrated MOSAIC's composition capabilities via deployment and measurement on both a local cluster and the PlanetLab testbed, and showed that the performance overhead of MOSAIC is respectable compared to native implementations, while achieving the benefits of overlay composition.

This paper extends our previous work [26] as follows. First, in Section 4.4, we introduce the notion of Composable Virtual Views (CViews), which enables one to modularize Mozlog rules into component networks and features for reuse. Using examples such as the Chord DHT, we describe the syntax and usage of CViews. Second, in Section 7.3 , we describe the extension of MOSAIC to support CViews, via a rewrite process that translates CViews into regular *Mozlog* rules for execution. Third, in Section 5.1, we have included a detailed description of the compilation process, from composition specifications to declarative *Mozlog* programs for execution. Finally, in Appendices A and B, based on the composition scenario described in Section 6, we provide an example composition specification in XML, and its corresponding abstract syntax that is used as input to the compilation process.

This paper makes the following contributions:

- A novel network architecture (MOSAIC) that enables new overlay networks to be implemented, selected, and dynamically composed based on application requirements.
- The declarative programming language called *Mozlog* that concisely specify high-level network protocol specifications. The language is used for implementing component overlay networks, and also is a basis for implementing the composition "glue code" among different overlay networks.
- A MOSAIC prototype implementation that compiles composition specifications into *Mozlog*, which are further used to generate efficient distributed dataflow-based implementations. The source code of the *Mozlog* compiler and the dataflow engine is available under an open-source license at `http://trac.research.att.com/trac/mosaic`.
- Experimental results in a local cluster and the PlanetLab testbed that validate MOSAIC's capabilities in enable dynamic and flexible compositions.

One promising avenue for future work is in further exploring the ability to perform *automated network composition*, namely, given application requirements, network properties and constraints, can the composition service automate the process of forming a composition? MOSAIC provides a framework towards enabling such a capability, but substantial research is required to fully realize automated composition.

Acknowledgments

This material is based upon work supported in part by NSF CAREER CNS-0845552, NSF CNS-0721845, IIS-0513778, and IIS-0477972. Yun Mao was supported in part by the Olga and Alberico Pompa Professorship of Engineering and Applied Science.

Appendix A. Composition Specification Example

The following XML specification is based on the Alice-and-Bob composition presented in Section 6. The first part of the specification consists of *bindings*, that are used to initialize the IP addresses of various subnets (*AliceSubNet* and *BobSubNet*), and regular hosts that serve either as gateway nodes (*AliceBW* and *BobGW*) or regular host (*AlicePC*).

These bindings are then used as a basis for creating new or using existing *modules*. These modules essentially represent overlay networks components. For instance, the specifications below creates a new module for *AliceNet* and *BobNet*, and adds gateway nodes *AliceGW* and *BobGW* to an existing RON overlay identified by *ron_oid*. The RON overlay also includes a URL to a known location storing the implementation code for the overlay.

Finally, these modules are connected together via *link* specifications which compose modules either by layering or bridging. In this example, *AliceNet* and *RON* are bridged together via the *AliceGW* gateway, while *BobNet* and *RON* are bridged via the *BobGW* gateway. In addition, the *i3* overlay is layered over *RON*, which itself is layered over IP (default with 0 as identifier).

```
<mosaic>
 <bindings>
  <subnet>
   <name>AliceSubNet</name>
   <ip>10.1.1.0</ip>
   <mask>255.255.255.0</mask>
  </subnet>
  <subnet>
   <name>BobSubNet</name>
   <ip>10.2.1.0</ip>
   <mask>255.255.255.0</mask>
  </subnet>
  <node><name>AliceGW</name>
    <ip id="AliceSubNet">10.1.1.1</ip>
    <ip id="0">123.45.67.8</ip>
  </node>
  <node> <name>AlicePC</name>
    <ip id="AliceSubNet">10.1.1.12</ip>
  </node>
  <node><name>BobGW</name>
    <ip id="0">234.56.78.1</ip>
  </node>
 </bindings>
 <composition>
  <module oid="AliceNet" Name="IP" type="Existing" />
  <module oid="BobNet" Name="IP" type="Existing" />
  <module oid="ron_1" name="RON" type="Extend">
    <nodelist>
     <node>AliceGW</node>
     <node>BobGW</node>
    </nodelist>
    <codeurl>http://www.mosaic-system.net/ron/v1</codeurl>
  </module>
  <module oid="i3_1" name="i3" type="Extend">
    <nodelist>
     <node>AliceGW</node>
     <node>BobGW</node>
    </nodelist>
    <codeurl>http://www.mosaic-system.net/i3/v1</codeurl>
  </module>

  <link type="bridge" name="AliceBridge">
    <first>AliceNet</first>
    <second>ron_1</second>
    <gateway>AliceGW</gateway>
    <type>on-demand</type>
  </link>
  <link type="bridge" name="BobBridge">
    <first>BobNet</first>
    <second>ron_1</second>
    <gateway>BobGW</gateway>
    <type>on-demand</type>
  </link>
  <link type="layer">
    <top>i3_1</top>
    <bottom>ron_1</bottom>
  </link>
  <link type="layer">
    <top>ron_1</top>
    <bottom>0</bottom>
  </link>
 </composition>
</mosaic>
```

Appendix B. Abstract Syntax of Composition Specification

```
spec        ::=  Spec(bindings,composition)
bindings    ::=  binding | binding, bindings
binding     ::=  subnet | node
subnet      ::=  Subnet(NAME,ADDR,MASK)
node        ::=  Node(NAME, addrlist)
gateway     ::=  node
addrlist    ::=  ADDR | ADDR, addrlist
composition ::=  Composition( modules, links )
modules     ::=  module | module, modules
module      ::=  existing | new | extend
existing    ::=  Existing(NAME, OID)
codeurl     ::=  URL
new         ::=  New(NAME, OID, nodelist, [gateway], [codeurl])
extend      ::=  Extend(NAME, OID, nodelist)
nodelist    ::=  node | node, nodelist
links       ::=  link | link, links
link        ::=  layering | bridging
layering    ::=  Layering OID Over OID
bridging    ::=  Bridging OID With OID By BRIDGETYPE Via NAME As NAME
```

Figure B.10: Abstract syntax of composition specification

Figure B.10 shows the abstract syntax of the composition specification, that is used as a basis for representing the inputs to the composition algorithms in Section 5. We refer to terms in *Italic* as type identifiers, and terms in CAPS as primitive types. In particular, NAME (string), ADDR (network address), OID (overlay ID), MASK (network mask), BRIDGETYPE (either "on-demand" or "pre-defined" as two ways of bridging) and URL are primitive types.

The specifications are divided into *bindings* and *composition*. *Bindings* are used for initializing the name and address of each participating host in the composition. They can also refer to a group of nodes (subset). Each node is identified by its label (NAME) and IP address (ADDR), whereas each subnet additionally includes a network mask (MASK).

A composition is comprised of *modules* (*i.e.* component overlay) and *links* that compose overlays together. Each module specification can either refer to an existing overlay component (*Existing*), extend an existing component by adding more nodes (*Extend*), or be created entirely from scratch (*New*). When a new overlay component is created, it is initialized with the name of a component (NAME), a unique identifier (OID), a list of nodes participating in the component (nodelist), an optional default *gateway* (node), and an optional URL that references the code that implements the component. Each link specification refers to either *layering*, or *bridging*. Here, the layering specification layers one module over another, while the bridging specification bridges one module with another via a specified gateway node.

References

[1] OpenVPN: Building and Integrating Virtual Private Networks. http://www.openvpn.net.

[2] David Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient Overlay Networks. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[3] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business Process Execution Language for Web Services. 2003. http://www.ibm.com/developerworks/library/specification/ws-bpel/.

[4] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking Up Data in P2P Systems. *Communications of the ACM, Vol. 46, No. 2*, February 2003.

[5] Hari Balakrishnan, Karthik Lakshminarayanan, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Michael Walfish. A Layered Naming Architecture for the Internet. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2004.

[6] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. 2001. http://www.w3.org/TR/wsdl.

[7] D. Clark, C. Partridge, J. C. Ramming, and J. Wroclawski. A Knowledge Plane for the Internet. In *Proceedings ACM SIGCOMM Conference*, Karlsruhe, Germany, August 2003.

[8] J. Crowcroft, S. Hand, R. Mortier, T. Roscoe, and A. Warfield. Plutarch: An Argument for Network Pluralism. In *Proceedings of ACM SIGCOMM Workshop on Future Directions in Network Architecture (FDNA)*, 2003.

[9] John R. Douceur and Jon Howell. Distributed directory service in the farsite file system. In *Proceedings of the 7th symposium on Operating Systems Design and Implementation (OSDI'06)*, pages 321–334, Seattle, Washington, 2006.

[10] M. Freedman, K. Lakshminarayanan, and D. Mazieres. OASIS: Anycast for any service. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI'06)*, 2006.

[11] Michael J. Freedman, Karthik Lakshminarayanan, Sean Rhea, and Ion Stoica. Non-Transitive Connectivity and DHTs. In *Proceedings of the Second Workshop on Real, Large Distributed Systems (WORLD'05)*, 2005.

[12] Xiaodong Fu, Weisong Shi, Anatoly Akkerman, and Vijay Karamcheti. Cans: Composable, adaptive network services infrastructure. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS'01)*, 2001.

[13] Xiaohui Gu, Klara Nahrstedt, and Bin Yu. Spidernet: An integrated peer-to-peer service composition framework. In *Proceedings of the 13th International Symposium on High-Performance Distributed Computing (HPDC-13)*, pages 110–119, Honolulu, Hawaii, June 2004.

[14] Krishna P. Gummadi, Harsha Madhyastha, Steven D. Gribble, Henry M. Levy, , and David J. Wetherall. Improving the reliability of internet paths with one-hop source routing. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI'04)*, 2004.

[15] Dilip Joseph, Jayanthkumar Kannan, Ayumu Kubota, Karthik Lakshminarayanan, Ion Stoica, and Klaus Wehrle. OCALA: An architecture for supporting legacy applications over overlays. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI'06)*, 2006.

[16] Dina Katabi and John Wroclawski. A framework for scalable global IP-anycast (GIA). In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2000.

[17] A. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure overlay services. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2002.

[18] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.

[19] Boon Thau Loo. The Design and Implementation of Declarative Networks (Ph.D. Dissertation). Technical Report UCB/EECS-2006-177, University of California at Berkeley, 2006.

[20] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative Networking: Language, Execution and Optimization. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, June 2006.

[21] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing Declarative Overlays. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[22] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, Philadelphia, PA, 2005.

[23] Harsha V. Madhyastha, Tomas Isdal, Michael Piatek, Colin Dixon, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. iplane: An information plane for distributed services. In *Proceedings of the 7th symposium on Operating Systems Design and Implementation (OSDI'06)*, Nov 2006.

[24] Harsha V. Madhyastha, Arun Venkataramani, Arvind Krishnamurthy, and Thomas Anderson. Oasis: An Overlay-Aware Network Stack. In *Operating Systems Review*, pages 41–48, 2006.

[25] Yun Mao, Bjorn Knutsson, Honghui Lu, and Jonathan M. Smith. DHARMA: Distributed Home Agent for Robust Mobile Access. In *Proceedings of Annual Joint Conference of the IEEE Computer Communications Societies (INFOCOM)*, 2005.

[26] Yun Mao, Boon Thau Loo, Zachary G. Ives, and Jonathan M. Smith. MOSAIC: Unified Declarative Platform for Dynamic Overlay Composition. In *Proceedings of the 4th ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2008.

[27] Z. Morley Mao and Randy H. Katz. Achieving service portability using self-adaptive data paths. In *IEEE Communications Magazine special Issue on Service Portability and Virtual Home Environment*, Jan 2002.

[28] N. Milanovic and M. Malek. Current solutions for web service composition. *IEEE Internet Computing*, 8(6):51–59, Nov 2004.

[29] Akihiro Nakao, Larry Peterson, and Andy Bavier. A Routing Underlay for Overlay Networks. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2003.

[30] T. S. Eugene Ng, Ion Stoica, and Hui Zhang. A waypoint service approach to connect heterogeneous internet address spaces. In *Proceedings of the USENIX Annual Technical Conference (USENIX'01)*, Boston, MA, June 2001.

[31] Larry Peterson, Scott Shenker, and Jon Turner. Overcoming the Internet Impasse Through Virtualization. In *Proceedings of ACM HotNets-III*, 2004.

[32] PlanetLab. Global testbed. 2002. http://www.planet-lab.org/.

[33] Anand Rajaraman, Yehoshua Sagiv, and Jeffrey D. Ullman. Answering queries using templates with binding patterns. In *ACM Symposium on Principles of Database Systems (PODS)*, 1995.

[34] Raghu Ramakrishnan and Jeffrey D. Ullman. A Survey of Research on Deductive Database Systems. *Journal of Logic Programming*, 23(2):125–149, 1993.

[35] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. Opendht: a public dht service and its uses. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2005.

[36] Skype. Skype P2P Telephony. 2006. http://www.skype.com.

[37] Jonathan Smith. Application-private networks. In David Gries, Fred B. Schneider, Andrew Herbert, and Karen Sparck Jones, editors, *Computer Systems*, Monographs in Computer Science, pages 273–277. Springer New York, 2004.

[38] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet Indirection Infrastructure. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2002.

[39] Lakshminarayanan Subramanian, Ion Stoica, Hari Balakrishnan, and Randy Katz. OverQoS: An Overlay Based Architecture for Enhancing Internet QoS. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'04)*, 2004.

[40] Amin Vahdat and David Becker. Epidemic routing for partially-connected ad hoc networks. *Duke Technical Report CS-2000-06*, 2000.

[41] J. T. Wroclawski. The Metanet. In *Proceedings of Workshop on Research Directions for the Next Generation Internet*, 1997.

[42] Shelley Q. Zhuang, Kevin Lai, Ion Stoica, Randy H. Katz, and Scott Shenker. Host Mobility using an Internet Indirection Infrastructure. In *Proceedings of ACM/Usenix Mobisys*, 2003.