

Department of Computer & Information Science

Technical Reports (CIS)

University of Pennsylvania

Year 2007

MOSAIC: Multiple Overlay Selection
and Intelligent Composition

Yun Mao*
Zachary G. Ives‡

Boon Thau Loo†
Jonathan M. Smith**

*University of Pennsylvania

†University of Pennsylvania, boonloo@cis.upenn.edu

‡University of Pennsylvania, zives@cis.upenn.edu

**University of Pennsylvania, jms@cis.upenn.edu

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-07-22.

This paper is posted at ScholarlyCommons.
http://repository.upenn.edu/cis_reports/654

MOSAIC: Multiple Overlay Selection and Intelligent Composition

Yun Mao Boon Thau Loo Zachary Ives Jonathan M. Smith

CIS Department, University of Pennsylvania

Abstract

Today, the most effective mechanism for remedying shortcomings of the Internet, or augmenting it with new networking capabilities, is to develop and deploy a new overlay network. This leads to the problem of multiple networking infrastructures, each with independent advantages, and each developed in isolation. A greatly preferable solution is to have a single infrastructure under which new overlays can be developed, deployed, selected, and combined according to application and administrator needs.

MOSAIC is an extensible infrastructure that enables not only the specification of new overlay networks, but also dynamic selection and composition of such overlays. MOSAIC provides *declarative networking*: it uses a unified declarative language (*Mozlog*) and runtime system to enable specification of new overlay networks, as well as their composition in both the control and data planes. Importantly, it permits dynamic compositions with both existing overlay networks and legacy applications. This paper demonstrates the dynamic selection and composition capabilities of MOSAIC with a variety of declarative overlays: an indirection overlay that supports mobility (*i3*), a resilient overlay (RON), and a transport-layer proxy. Using a remarkably concise specification, MOSAIC provides the benefits of runtime composition to simultaneously deliver application-aware mobility, NAT traversal and reliability with low performance overhead, demonstrated with deployment and measurement on both a local cluster and the PlanetLab testbed.

1 Introduction

The Internet faces many challenges today, ranging from the lack of protection against unwanted or harmful traffic to the increasing complexity and fragility of inter-domain routing. At the same time, the proliferation of new applications has led to growing demands for new capabilities such as mobility, content-based routing, and quality-of-service (QoS) routing.

Overlay networks [19] that use the existing Internet to provide connectivity for new services are both deployable [20] and enable innovation. However, despite deployment at global scale and emerging support for legacy applications [6], overlay networks now face several hurdles. First, they are often optimized for a specific appli-

cation and may not be useful in all contexts. Second, overlay networks are generally targeted at and limited to niche *vertical* domains (*e.g.*, mobility [33, 16], security [8], reliability [1]). Third, the networks do not normally interoperate or share their functionality. For example, resiliency [1] and mobility [27] 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 returns from her vacation in the Amazon rain forest and wants to share digital videos of her trip with interested friends. Not satisfied with the privacy protection and the quality of streams served by public video sharing sites, she decides to host a Web server for the videos on her laptop. While in principle the laptop is capable of this, in practice she faces multiple difficulties. First, because Alice brings her laptop along on her daily commute between school and home, its IP address changes. Second, at home she shares a cable modem with her roommate, so her laptop does not have a public IP address. Third, her ISP is experiencing intermittent routing failures to some peering ISPs due to a BGP misconfiguration.*

Each of the capabilities Alice requires can individually be supported by an overlay network — the challenge is in getting the overlays to work together. Emerging systems such as OCALA [6] have attempted to develop bridging capabilities among overlays — but we argue that a more general architecture, which supports *layering* as well as bridging, plus *development* and *extension* of overlay protocols — is necessary. MOSAIC (Multiple Overlay Selection and Intelligent Composition) is a unifying platform for developing, deploying, combining, and composing overlay networks — one capable of bridging between overlays, stacking them in layers, and allowing for rapid extensibility with new functionalities. It enables (1) rapid authoring and deployment of new overlay networks, (2) application-aware adaptivity to *select* and *compose* overlay networks to meet application needs, and (3) seamless support for legacy applications within the infrastructure.

MOSAIC is based on the model of *declarative networks* [13, 12], a declarative, database-inspired extensible infrastructure using *query languages* to specify behavior. Declarative networks utilize the Network Dat-

alog (*NDlog*) language as a natural and compact way to implement a variety of routing protocols and overlay networks. The ensuing multiple-orders-of-magnitude reduction in code size (*e.g.*, Chord [28] in 43 lines) significantly increases network extensibility and makes it easy for applications to adapt the overlays to suit their requirements. Support for declarative abstractions makes it easy to compose protocols, either vertically (layering) or horizontally (bridging).

MOSAIC builds upon the declarative networking model by introducing several new constructs. *Dynamic location specifiers*, combined with runtime types, enable flexible naming and addressing; *composable virtual views* support modularity and composability; data and control plane extensibility supports composition; declarative tunneling and proxying enable support for legacy applications. Several *Mozlog* features both meet the composition goals of MOSAIC, and are also useful language abstractions for declarative networking in general. For example, the use of composable views results in fewer lines of code and provides better error handling.

Organization: Section 2 describes the options for overlay composition. Section 3 presents an architectural overview of the MOSAIC infrastructure. Section 4 summarizes the main aspects of the original *NDlog* language, and describes our *Mozlog* extensions to enable various aspects of MOSAIC. In Section 5, we demonstrate that *Mozlog* specifications are amenable to execution within a distributed query processor via modifications to the P2 declarative networking system. Section 6 experimentally evaluates the compositions of various declarative overlays that support indirection (*i3* [27]), mobility, and resilience (RON [1]). We show that MOSAIC’s ability to support flexible compositions can enable application-aware mobility, flexibility, and resilience at low overheads. In Section 7, measurement results are presented for networks created on a local cluster and the PlanetLab testbed.

2 Overlay Composition

Network composition is the act of combining distinct parts or elements of existing networks to create a new network with new functionalities. Overlay composition is network composition of overlay networks, and so results in a new overlay network. We consider composition of overlays along two planes.

Data plane composition. The data planes of two overlay networks can be composed horizontally by *bridging* between the networks, or they can be composed vertically by *layering* one overlay over the other.

In *bridging* (see Figure 1), each overlay network runs

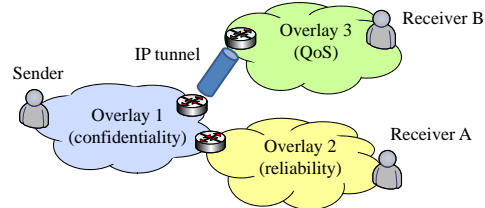


Figure 1: Overlay composition by bridging.

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 but not time-sensitive data to receiver A, while using a QoS overlay to deliver multimedia traffic to receiver B.

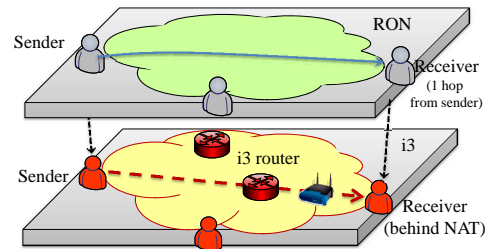


Figure 2: Overlay composition by layering.

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 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 *i3*, 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. We also reiterate that prior attempts to combine overlay networks [6] only support bridging but not layering.

Control plane composition. One overlay network’s control plane may be layered over either the data plane or the control plane of another overlay. For example, it is possible to build the control message channels of DHT

protocols 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 failure happens intermittently, churn rate is increased and may create unnecessary state inconsistency. Using a resilient overlay like RON can overcome some of the network failures to reduce churn.

Some overlay network protocols have complex, layered control planes. For example, both *i3* and DOA [2] use DHTs for either forwarding or lookup. RON and OverQoS 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.* [18], observed that on PlanetLab, each node had 1GB 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 [23] for DHT lookup and iPlane [14] 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

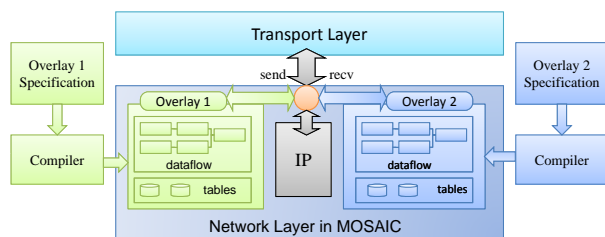


Figure 3: An overview of the MOSAIC architecture for network layer overlays.

MOSAIC is an architecture to design, implement and deploy *composable* overlay networks based on a data-centric declarative networking approach [13, 12], and directly addresses the architectural challenges imposed by overlay composition.

Figure 3 illustrates the MOSAIC architecture. MOSAIC allows users to define a network protocol using

declarative specifications rather than high level code: the actual implementation of the protocol is automatically generated by a declarative compiler. As a result, one can pay less attention to implementation details such as packet format, data (de)serialization, packet fragmentation, etc. Instead, the focus is on the protocol behavior.

The declarative approach provides a further benefit that we exploit in MOSAIC: since network definitions in MOSAIC separate specification from implementation, the system can (assuming the right constraints are met) freely replace the IP network 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, so long as they are reachable from one another.

MOSAIC is positioned at the network layer in the network stack to replace 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 raw byte data as the payload. 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.

In addition, we have added initial support for transport layer extensibility in MOSAIC by exposing TCP-style primitives to the overlays. These extensions and an implemented proof-of-concept are discussed in Section 4.6.

3.1 Specifying Overlay Networks

In MOSAIC, overlay specifications are written in *Mozlog*, which is a database-like declarative query language based on *NDlog* [12]. (We present the details of *Mozlog* and *NDlog* in Section 4). MOSAIC takes a *Mozlog* program, compiles it into distributed P2 dataflows [12], and deploys it to all nodes that participate 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 [9], 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.

3.2 Specifying Overlay Compositions

The *Mozlog* language upon which MOSAIC is based is inherently compositional: the results of one declarative query can be named as a *view*, which can be used as an input to another query (or view). However, layering and bridging among protocols require a certain amount of semantic understanding encoded in the rules.

Layering. Layering of a control or data plane over another overlay’s data plane is achieved by ensuring that every protocol uses opaque abstract addresses — rather than being bound to physical addresses. MOSAIC binds the abstract addresses of the upper layer to the logical addresses in the underlying overlay; for example, a RON node built on top of *i3* has `RON::(i3::key)` as opposed to `RON::IP`. In addition to the address composition, MOSAIC encapsulates the upper overlay’s data or control payload within the underlying overlay’s packet. The result is a seamless layering of one overlay network on top of another.

Additionally, 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 relevant database logical views (query results presented as a named table) to expose its composable components. These views typically encapsulate useful abstractions of state: *e.g.*, a distributed hash table’s contents 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 no interoperability issues arise.

Bridging. Bridging can be done in an explicit or implicit way in MOSAIC. In the implicit way, the desti-

nation has an address of an overlay in which the sender does not participate. MOSAIC tries to send to one of the overlay nodes via anycast [7, 4], which continues to forward the packet. In an explicit way, the end points find the gateway node that sits on both networks. Either source routing is used to specify an address such as `sr::[gateway, dest]`, which explicitly describes the data path, or address translation state that uniquely identifies the flow between the sender and the receivers is stored at the gateway so that it can perform indirection. We will describe several examples of such compositions in Section 6 using the *Mozlog* language.

3.3 Composition in MOSAIC

The end goal of MOSAIC is to match a composition of overlay networks to an application’s needs. This may be subject to constraints: certain nodes, either at the client-side or the server side, may have policies restricting which overlay networks are allowed to run within the local subnets. Moreover, there may be multiple alternative means to achieve the same goals, *e.g.*, using different overlays that provide the same capabilities, such as SOS [8] and OverDoSe [26] for DoS attack resistance.

The problem of automatic service composition is an entire field of research [17], and we leave a comprehensive treatment of the problem as future work. However, we assume the existence of the following metadata that assists in finding the appropriate compositions:

- A specification of *end-to-end application connectivity requirements*, included as a file associated with each client application.
- A set of *local overlay constraints* within (1) the client’s local subnet and (2) the server’s local subnet — each of which restricts what overlay networks may run locally.
- A global *system catalog*, in the form of tables, specifying (1) allowed compositions of overlays (both bridging and layering); (2) mappings from potential application requirements to overlays or overlay compositions that meet the requirements.
- A *composition rule list* specifying *Mozlog* rules for composing overlays, which is disseminated to the nodes willing to execute the composition.

The system catalog includes a predefined set of overlay properties relating to mobility, QoS, security, multicast, transcoding, etc. When an overlay is created, its provider publishes a *service description* to the system catalog, specifying which properties it satisfies.

During application setup, MOSAIC takes the list of application requirements and queries the system catalog

tables for satisfying overlay combinations that meet the local constraints at the client and server ends, use only legal overlay compositions, and satisfy the application requirements. Each composition represents a database join, and in essence the query finds a “path” of compositions through the list of compositions. The local system administrator generally makes the final decision to authorize a composition of overlays, and this information is stored for retrieval at runtime. We describe the runtime system and how it executes compositions in Section 5.

4 Mozlog Language

Before presenting *Mozlog*, we first provide a brief overview of the *NDlog* language upon which it is defined.

4.1 Core Language: *NDlog*

NDlog is based on Datalog [22]: a Datalog program consists of a set of declarative *rules*. Each rule has the form $p :- q_1, q_2, \dots, q_n$, which can be read informally as “ q_1 and q_2 and \dots and q_n implies p ”. Here, p is the *head* of the rule, and q_1, q_2, \dots, q_n 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 by the query), 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.

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

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

The rules r_1 and r_2 specify a distributed transitive closure computation, where rule r_1 computes all pairs of nodes reachable within a single hop from all input links, and rule r_2 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.

NDlog 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, representing reachable pairs of nodes from S to D .

4.2 Mozlog Extensions

In this section, we describe *Mozlog*’s language extensions to *NDlog* that are required for overlay composition. We categorize these extensions as follows:

Flexible naming and addressing: *Mozlog* supports *dynamic location specifiers* whose types (e.g., IP address or logical overlay identifier) are determined at runtime. In addition, location specifiers are decoupled from data attributes and made optional for local data. These two language extensions not only enable interoperability among multiple overlays, but provide multi-homing and mobility features (Section 4.3).

Data and control plane integration: *Mozlog* provides language support for forwarding on the data plane. This provides extensibility at both the control and data plane, and hence provides flexible composition of different overlay features on either plane (Section 4.4).

Modularity and reusability: *Mozlog* allows multiple declarative rules to be composed and modularized as *Composable Virtual Views (CViews)*. This enables features of different overlays to be modularized, hence facilitating composition of different features, and improved resource sharing. As an additional benefit, *Mozlog* provides more concise specifications and better abstractions for timeouts and exception handling (Section 4.5).

Special predicates: *Mozlog* provides several predicates for accessing the tun device and using TCP. This enables legacy application support at both the network and transport layers and creates the opportunity to build transport layer overlays (Section 4.6).

This section focuses purely on the *Mozlog* language extensions; we discuss implementation details in Section 5 and provide detailed use cases and experimental analysis in Section 6.

4.3 Flexible Naming and Addressing

Location specifiers in *NDlog* currently have two limitations. First, they are assumed to be IP addresses, hence limiting its usage to IP-based networks. As a result, it is not possible to express data placement in terms of overlay identifiers or differentiate data that belongs to dif-

ferent overlays. Second, location specifiers tightly couple data’s attributes to its location, hence limiting each host to store only data with a unique network address. This prevents multi-homing, an important requirement when each physical host may be simultaneously associated with several logical overlay networks. Third, mobility is not supported since any change in IP address will invalidate all local tables.

To address these limitations, we make two modifications. First, we *decouple* each data from its location specifier, and make them optional. Second, we associate all location specifiers with a runtime type.

4.3.1 Decoupling Location from Data

To enable the first modification, *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. For example, in the following rule,

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

`periodic` is a built-in local event that will be triggered every 10 seconds with a unique identifier `E`. 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 hostname `N` will be sent to the monitoring server `R`.

Decoupling data from its location enhances interoperability and reusability. Now 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.*, *i3*, RON, or IP). Finally, since it does not bind addresses to data, the language is friendly to mobility, where host movement (and hence resulting change in its IP address) does not invalidate its local tables.

4.3.2 Runtime Types for Location Specifiers

Our second modification 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 unique string identifier for an overlay network, and `nID` is a mandatory overlay node identifier. For example, `i3::0x123456789I` denotes an *i3* node with identifier `0x123456789I`, and `ron::"158.130.7.3:10000"`

denotes a RON node with IP address `158.130.7.3:10000`. 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 4.3. Rule `a1` can be rewritten as `a2`, in which the monitoring server `R` refers to an *i3* key generated as a hash of its name `N` instead of an IP address:

```
a2 alarm@R(L, N) :- periodic(E, 10), cpuLoad(L),
  nodeName(N), serverName(N), L>20,
  Key := f_shal(N), R:=i3::Key.
```

Dynamic location specifiers enable bridging of different overlays easily. For example, a gateway node `G` can physically host two overlay network nodes (one for *i3* and another for RON), and is addressable via either network. A source routing specifier is used to perform forwarding via the gateway node. For instance, node `Dest` in RON with address `sr::[i3::Gateway_key, ron::Dest]` is reachable from all hosts in the *i3* network. As an additional benefit, dynamic location specifiers enable addresses to be updated at runtime to switch between IP networks and various overlays. We provide a detailed example in Section 6.

4.4 Data and Control Plane Integration

Declarative networking previously focused on the control plane of networks. 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. Each overlay network has `send` and `recv` predicates that are used to specify data forwarding within an overlay. We provide an example based on the data plane of RON:

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

Rule `snd` expresses that for all non-local `Dest` addresses, the data packet (`Packet`) is sent along the next hop (`Next`) which is determined via a join with RON’s routing table (`ron.RT`) using `Dest` as the join key. These packets are then received via the rule `rcv` at node (`Dest`), which generates a `ron.recv(Packet)` 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 a detailed example in Section 6.

4.5 Modularity and Composability

In order to support overlay composition, *Mozlog* supports *Composable Virtual Views* (CViews). These define rule groups that, when executed together, perform a specific functionality.

4.5.1 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 K_1, K_2, \dots, K_n which are already bound to input values read from another predicate (intuitively, these are like input parameters to a function call). The remaining attributes, $\&R_1, \&R_2, \dots, \&R_m$, represent the *return values* from invoking the predicate given the input values. This is akin to the use of input bindings [21] in data integration, which were used to pass data into queryable Web forms to retrieve relation results.

We illustrate using a view definition for the following CView predicate `ping`(SrcAddr, DestAddr, &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. `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.

4.5.2 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 *i3* by composing Chord and RON. The Chord lookup in CView can be written as:

```
chord.lookup@Ldmk (Key, &DestID, &DestAddr)
```

Given a query on `Key`, it returns the lookup result: the Chord ID of the destination and the network address of the destination. A query with an unbound `Key` will be rejected by the compiler.

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 *i3* client tries to locate a private trigger that relays its traffic, it can leverage the RON measurement results and find the best private trigger.

```
/*schema: (Address, Key, RTT) */
materialize(bestPT, SAMPLE_LIFETIME, 1,
  keys(1), evict max(3)).
s1 bestPT(KeyAddr, K, RTT) :-
  periodic(E, SAMPLE_INTERVAL),
  localAddr(LocalAddr), K :=f_randID(),
  chord.lookup@LANDMARK(K, &_, &KeyAddr),
  ron.latency(LocalAddr, KeyAddr, &RTT).
s2 trigger@KeyAddr(NodeID, LocalAddr) :-
  periodic(E, TRIGGER_REFRESH_INTERVAL),
  node(NodeID), localAddr(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. 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 measurements 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). Not only is code reused, but network resources are saved. Second, CViews abstract details of asynchronous event-driven programming. In the `ping` example, nodes no longer are 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 original *NDlog* specifications. This enhances readability and makes the code even more concise: the use of CViews reduced the number of lines in Chord by 8 rules (from 43 to 35).

4.6 Special Predicates

To interact with legacy applications and provide more transport layer functionalities, *Mozlog* supports several built-in predicates for tun device access and TCP. The `tun` predicate has the following schema: `tun(IPPacket[, SrcIP, DestIP, Protocol, TTL])`. When MOSAIC receives an IP packet from `/dev/net/tun`, a `tun` tuple is injected into the dataflow. `IPPacket` is the whole IP packet including the header. `SrcIP`, `DestIP`, `Protocol` and `TTL` are corresponding attributes extracted from the IP header. When `tun` is an action generated by the rules, `IPPacket` will be sent to `/dev/net/tun`. Optionally, the IP header is updated based on the rest of the attributes if given.

We use the following two rules to demonstrate how to use the `tun` predicate:

```
p2p_tun tun@Peer(Pkt) :- tun(Pkt),
    Peer:="158.130.7.3:1086".
i3_tun tun@Peer(Pkt) :- tun(Pkt, Src, Dest),
    Key:=f_sha1(Dest), Peer:=i3::Key.
```

Rule `p2p_tun` sets up a point-to-point UDP tunnel between the local node and the remote MOSAIC node listening at the specific address and port. The peer IP is a constant UDP address. Similarly, rule `i3_tun` sets up a tunnel via `i3`. It uses the SHA-1 hash of the destination tunneling address as the `i3` key.

A second set of new predicates is TCP-related: Each predicate corresponds to a system call for TCP sockets. They are `tcp.listen`, `tcp.connect`, `tcp.accept`, `tcp.read`, `tcp.write` and `tcp.close`, provided in the CView syntax. This support provides a foundation for transport layer or session layer overlay [16, 11, 32] support inside MOSAIC.

An example use case would be to use `tcp.read` and `tcp.write` to forward packets from `Skt1` to `Skt2`.

```
fwdEvent(Skt1, Skt2) :- fwdEvent(Skt1, Skt2),
    tcp.read(Skt1, 0, &Packet),
    tcp.write(Skt2, Packet, &Size).
```

`tcp.read` has the schema of `tcp.read(Skt, Len, &Packet)`. That is, the query takes a socket descriptor and an integer `Len` as inputs and returns the actual packet when it is received from the socket. The socket descriptor is obtained from either `tcp.accept` or `tcp.connect`. Similarly, `tcp.write` sends `Packet` to `Skt2`. As a recursive rule, it keeps forwarding data packets until one of the sockets is closed.

5 Implementation

The MOSAIC platform uses the P2 [12] declarative networking system at its core, but adds significant new functionality. A translator generates *NDlog* rules from *Mozlog* rules to leverage the existing planner.

In addition, we modify the P2 planner and dataflow engine to generate execution plans that can accommodate new language features of *Mozlog*, specifically those related to runtime support for dynamic location specifier, data plane forwarding, and various transport and tunneling predicates.

5.1 Dataflow

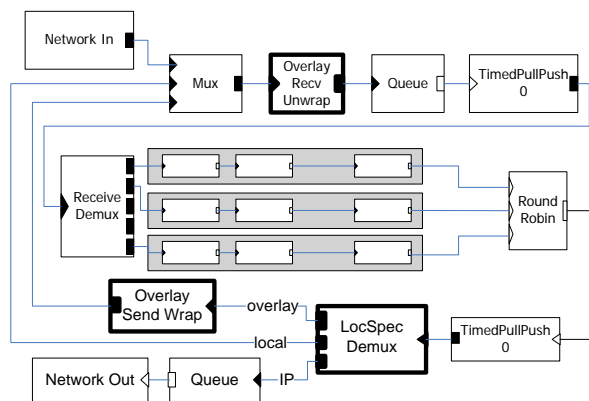


Figure 4: System dataflow with dynamic location specifiers.

Figure 4 shows a typical execution plan generated by compiling *Mozlog* rules. Similar to P2 dataflows, there are several network processing elements (denoted by `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 [12] 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 will extract 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 “unwrap” until the payload is retrieved. The `Packet` payload is then used as input to the dataflow via the `ReceiveDemux` element, and 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@ovname::ovaddr(F1, F2, ..., Fn)` are designated for overlay network `ovname` with address `ovaddr`. A new event tuple `ovname.send(ovaddr, tuplename(F1, F2, ..., Fn))` which denotes the `send` primitive of the overlay network `ovname` is generated (see Section 4.4). This new tuple is reinserted back to the same dataflow to be forwarded based on the overlay specification.

5.2 Compilation of CViews

The *Mozlog-to-NDlog* translator requires rewriting and expanding 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 4.5) 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. The 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 identifier `QID` 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 have explored several optimizations of CView compilation that are beyond the scope of this paper. These include *tail recursive view optimization* to reduce

communication overhead, *inline view expansion* by duplicating runtime CView elements to reduce demultiplexing overhead, and *local event shortcut* to shorten the dataflow paths and reduce scheduling overhead.

5.3 Special Predicates

The `tun-`, and `tcp-`related special predicates are treated differently from ordinary tuples in the dataflow by the planner. Each special predicate has a rule strand in the dataflow, between the `ReceiveDemux` element and the `RoundRobin` element (see Figure 4). For `tun`, 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 to an IP packet and writes to the `tun` device.

Each TCP predicate has a corresponding input and output event handler. To use `tcp.read` as an example, first the CView compiler translates each rule that contains `tcp.read` to the *NDlog* format, which generates a `tcp.read.init` event and waits for the `tcp.read.return` event. The `tcp.read.init` event is connected to the `tcpRead` element. It adds the socket descriptor to the `select()` pool of the P2 event loop. Once data is available, the P2 event loop calls back to the element, which then removes the socket descriptor from the `select()` pool, reads the packet, and sends a `tcp.read.return` event tuple containing the packet.

5.4 Legacy Support

MOSAIC adopts two mechanisms to support legacy applications at different layers. At the network layer, we use the `tun` device to provide overlay tunnels between legacy applications. For each end host, it takes a private IP address from `1.0.0.0/8` to avoid conflict from other public IP networks. After a legacy application sends a packet to an address in the `tun` network, the kernel redirects it to MOSAIC, which generates a `tun` tuple. Currently there is an address translation rule to use a special mapping table to translate the private IP address to the overlay address. This can be extended to use any name resolution service in the future by combining DNS request hijacking [6]. After address translation, the packet tunneling rules we described in Section 4.6 deliver the IP packet to the destination via the corresponding overlays.

To use IP layer overlays, such as `i3` or `RON`, IP tunneling is mandatory. For transport layer overlays that intend to replace or augment TCP, we provide an alternative way to leverage the dynamic library call interceptions. The environment variable `LD_PRELOAD` is set to our customized library to intercept the socket system calls at

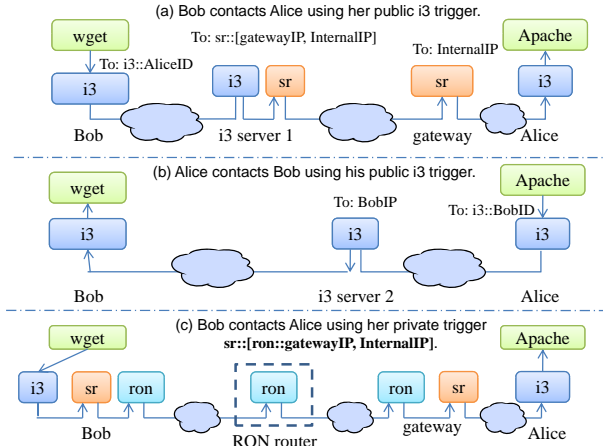


Figure 5: Alice hosts a Web server using MOSAIC.

the user space. Compared to the tun approach at the IP layer, library interceptions avoid the overhead of an extra memory copy between the kernel and user space, and exposes the connection oriented primitives from the application to the transport overlays. Based on the TCP predicates, we have implemented a SOCKS proxy [10] in our prototype, which can be viewed as a two-hop transport layer overlay that does source routing to traverse firewall.

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 `legacy.send` tuple from the dataflow to the overlay, and inject `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.

6 Composition Examples

We now demonstrate MOSAIC's ability to support flexible overlay compositions including bridging, layering and hybrid compositions.

Alice's challenges: Consider Alice's challenges of Section 1. An overlay composition can meet her needs. Suppose there is a publicly available *i3* overlay network, and Alice uses her gateway node at home to form a private RON network with her friends. For this scenario, we show the corresponding MOSAIC data structures: the global system catalog appears in Table 1, and the local overlay constraints are shown in Table 2.

Suppose Bob is a downloader whose overlay constraints are known to Alice. At school, Alice only needs mobility, but neither NAT traversal nor resilience. A

overlay	mobility	reliability	NAT
<i>i3</i>	Y	N	Y
sr	N	N	Y
RON	N	Y	N

Table 1: system catalog

	<i>i3</i>	sr	RON
Alice-Home	Y	Y	N
Bob	Y	Y	Y
Gateway (G)	N	Y	Y
Alice-School	Y	Y	N
<i>i3</i> -Server	Y	Y	N

Table 2: local overlay constraints

query on Table 1 returns *i3* as a mobility overlay supported by both Alice and Bob. Therefore, Alice and Bob connect via the *i3* overlay on IP. Alice inserts the public IP of her laptop into the public *i3* trigger. When Bob sends a packet to Alice's unique *i3* id `i3::AliceID`, the `i3.send` event in MOSAIC is invoked by MOSAIC runtime system. It first queries `chord.lookup` on the ID, then locates the *i3* server that holds the public trigger for the ID, and finally forwards the packet to the server. The server retrieves Alice's address from the trigger, and forwards the packet to Alice. Sending from Alice to Bob uses a similar scheme. As an optimization, Bob and Alice may exchange their actual IP addresses as the private triggers (known as *i3*-shortcut) to reduce indirection overhead, once they have located each other using the public triggers. In *Mozlog*, the private trigger (PT) address is retrieved using the query `bestPT@i3::K(K, PTAddr)`, where `K` is Alice's or Bob's ID.

When Alice comes home, she now needs mobility, NAT traversal via the gateway G, and a reliable connection between Bob and G. Mobility can still be provided by *i3*. NAT traversal requires overlay support from Alice, Bob, and the gateway, which leaves source routing as the only option. RON is also the only option for reliability. Since G does not run *i3*, Alice needs to provide a NAT-friendly address to *i3*. Therefore, *i3* should be layered over source routing. There are two data paths in *i3*: one via the public trigger, and one via the private trigger. The private trigger path is layered over the bridging of two paths as `sr::[ron::GatewayIP, InternalIP]`, providing both NAT traversal and data reliability. The path via public trigger is layered on source routing with `sr::[GatewayIP, InternalIP]`, because the *i3* server is not reachable from the gateway via the private RON network.

Alice is always reachable via the public trigger. Therefore, Alice's address change (*i.e.*, host mobility) is not a concern for Bob. Figure 5(a) and (b) illustrates the

dataflows of how they use the public triggers to communicate. The source routing address makes sure that Alice is reachable behind the NAT.

After Alice and Bob exchange private triggers, the packets from Bob are sent to the gateway node via their private RON network, then forwarded by the gateway to Alice as in Figure 5(c). Note that the RON router in the middle is only used when network failure happens between Bob and the gateway. Alice’s laptop is not on RON directly. Instead she forwards her packets to the nearest RON node (her gateway node), which forwards the packets to Bob via RON.

In this example, *i3* is the end-to-end overlay, which is layered on the overlay that bridges RON and Alice’s internal network. In our approach, composition decisions are made by the end users based on application scenarios. Furthermore, the composition is flexible: with a reliable network at school, Alice does not need to join RON at both locations.

If some policy does not allow RON on the gateway, Alice can deploy RON on her laptop. She can use `sr::[GatewayIP, InternalIP]` as her publicly reachable address to join RON, and use `ron::sr::[GatewayIP, InternalIP]` as her private trigger to exchange with Bob.

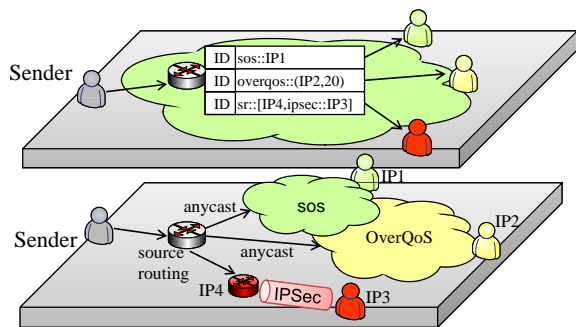


Figure 6: A publish/subscribe composition example using multicast.

Multicast for Pub/Sub Services: Consider a stock broker that publishes a stream of stock prices, and sends to the subscribers via multicast. Some of the subscribers are active traders, who demand the data in a timely fashion. Some of the users are under DoS attack. Some of them are connected via unsecure wireless links. In MOSAIC, we can compose the existing *i3*, SOS, OverQoS and IPsec overlays, as in Figure 6.

At the top layer, the publisher and the subscribers join an *i3* overlay. The published stock price has a unique *i3* ID, which is known to the subscribers. The subscribers form a multicast tree in *i3* and insert overlay-specific addresses into their leaf triggers. The traders

test	latency(ms)	throughput (KByte/s)
DirectIP	0.134	97994
OpenVPN	0.365	7999
MozTun	0.612	7419
RON	1.152	3358
<i>i3</i>	2.08	2023

Table 3: Overhead comparison in LAN

use `overqos::(IP2, 20ms)` to attain quality of service guarantees (latency within 20ms); the users under DoS attack use `sos::IP1` to allow packets with stock prices pass through, while blocking other unwanted IP traffic. Users with unsecure connections may redirect the traffic through an IPsec tunnel by inserting `sr::[IP4, ipsec::IP3]` into the trigger. In this scenario, the data plane of *i3* is layered over different overlays as well as IP; however, *i3* is unaware of this.

7 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.

In all our experiments, we make use of a declarative Chord implementation which consists of 35 rules. Our *i3* implementation uses Chord and adds 16 further rules. We also implement the RON overlay in 11 rules. Both *i3* and RON can be used by legacy applications via the `tun` device, as described in Section 4.6.

7.1 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 high-speed 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`.

7.1.1 Network Layer Overlay Overhead

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.

OpenVPN: OpenVPN [31] 2.0.9 is tunneling software that uses the tun device. 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 more than $0.2ms$, and the TCP throughput drops by a factor of more than 10. 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 7% throughput overhead of MozTun is solely attributed to the rule processing overhead in MOSAIC. Also, the latency increase of $0.25ms$ is due to the extra overhead incurred by the P2 dataflow engine, 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 with 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 *i3* servers, using Chord to provide lookup functionality. The remaining two nodes were selected as *i3* clients. A packet sent by the source *i3* client to the destination *i3* client went through the public trigger of the destination, which was hosted on the *i3* server of another cluster node. Since it introduced a level of indirection plus extra rule processing overhead, *i3* 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 (level of indirection of *i3* with tunneling), the extra latency ($2ms$) incurred is negligible for an application running on wide-area networks. Later, in Section 7.2, we will validate the performance of a composed overlay on the Planetlab testbed.

7.1.2 Transport Layer Overlay Overhead

Our proof-of-concept implementation of a transport layer overlay is a SOCKS proxy using 18 *Mozlog* rules. The

	TCP	SOCKS	SOCKS optimized
tput. (KB/s)	97994	8132	97186

Table 4: Overhead comparison in LAN between native TCP, SOCKS proxy in MOSAIC, and SOCKS proxy in MOSAIC with optimized dataflow

SOCKS protocol [10] is a transport layer protocol, which can be viewed as a transport-layer overlay network with one level of indirection for firewall traversal. We used the library interception technique mentioned in Section 5.4 to support legacy TCP applications.

We deployed our SOCKS proxy on the client and used `iperf` to measure TCP throughput between the client and the server. From the measurement results in Table 4, we observe that by using a different virtualization technique, the SOCKS proxy achieves better throughput than OpenVPN in Section 7.1.1. In addition, by applying *inline view expansion* and *local event shortcut* optimizations as described in Section 5.2, the throughput increases dramatically and approaches that of native TCP. The performance improvement obtained by our SOCKS proxy suggests that MOSAIC is able to translate and optimize high-level declarative specifications to efficient implementations for the data plane. A detailed performance study on the optimization is outside the scope of this paper and is a subject of future work.

7.2 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 *i3*, 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 to a resolution.

Our experimental setup is as follows. As our end-host, we used a Linux PC in Edison, NJ with a high speed cable modem connection 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 Salt Lake City, UT with a modified version of `wget` that records the download throughput.

These two nodes in NJ and UT, plus three additional nodes in Philadelphia, Berkeley, and Ithaca, were used to form a private RON network. We further selected 44 nodes from PlanetLab, mostly in the US, to run *i3*. During the experiment, in order to validate the functionality of resilient routing provided by RON, we manually injected network failures by 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.

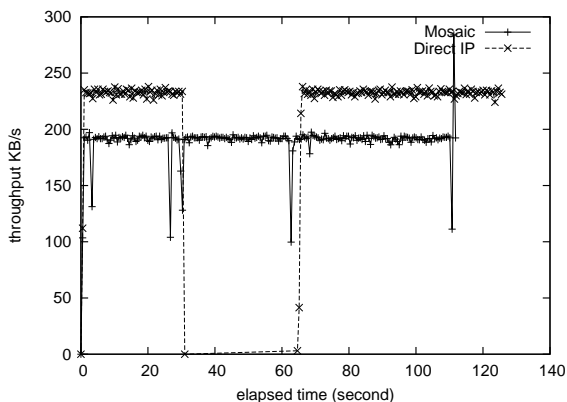


Figure 7: Throughput comparison between overlay composition in Mosaic vs direct IP connection during network failure. Network failures were injected 30 seconds after experiment start, and removed after 30 additional seconds.

Figure 7 shows the throughput of the download over time for MOSAIC and DirectIP. 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: Our 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 was able to complete the download in a shorter time despite lower throughput, due to the re-

siliency of RON.

8 Related Work

Overlays such as *i3* [27] and *TRIAD* [5] provide a deployable solution for new network capabilities. Examples of new capabilities include DDoS resistance [8, 26], performance and reliability [25, 1], and QoS [29]. Overlay-based approaches to network infrastructure have been used to enhance routing and to experiment with new ways to specify and deploy networks [13, 12, 24]. *MOSAIC* augments such systems by combining their best features to construct a new composite overlay.

Composing a plurality of heterogeneous networks was proposed in *Metanet* [30], and also examined in *Plutarch* [3]. *Oasis* [15] and *OCALA* [6] 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. In contrast, *MOSAIC*’s primary focus is on overlay 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.

9 Conclusions and Future Work

In this paper, we presented *MOSAIC*, an extensible infrastructure that enables not only the specification of new overlay networks, but also dynamic selection and composition of such overlays. *MOSAIC* provides *declarative networking*: it uses a unified declarative language (*Mozlog*) and runtime system to enable specification of new overlay networks, as well as their composition in both the control and data planes. We 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.

Our research is proceeding in several directions. First, we are exploring techniques for automatic overlay composition, given application requirements, overlay properties and constraints. Second, building upon our initial

language support for the transport layer, we are exploring adding mechanisms for extensible transport and session layer overlays [16, 11, 32]. Such extensibility will be useful in the context of mobile computing, and in environments where there is a high degree of network and device heterogeneity during an application session. Finally, we are also exploring better ways to exploit CViews for composition and sharing at finer granularity, by combining individual feature sets from multiple overlays to meet application needs.

References

- [1] D. Andersen, H. Balakrishnan, M. Kaashoek, and R. Morris. Resilient overlay networks. In *Proc. SOSP*, 2001.
- [2] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish. A Layered Naming Architecture for the Internet. In *Proc. SIGCOMM*, 2004.
- [3] J. Crowcroft, S. Hand, R. Mortier, T. Roscoe, and A. Warfield. Plutarch: An Argument for Network Pluralism. In *Proc. FDNA*, 2003.
- [4] M. Freedman, K. Lakshminarayanan, and D. Mazieres. OASIS: Anycast for any service. In *Proc of NSDI*, 2006.
- [5] M. Gritter and D. Cheriton. An Architecture for Content Routing Support in the Internet. In *USITS*, 2001.
- [6] D. Joseph, J. Kannan, A. Kubota, K. Lakshminarayanan, I. Stoica, and K. Wehrle. OCALA: An architecture for supporting legacy applications over overlays. In *Proc. NSDI*, 2006.
- [7] D. Katabi and J. Wroclawski. A framework for scalable global IP-anycast (GIA). In *SIGCOMM*, 2000.
- [8] A. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure overlay services. In *Proc. SIGCOMM*, 2002.
- [9] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.
- [10] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. SOCKS Protocol Version 5. RFC1928, 1996.
- [11] Y. Li, Y. Zhang, L. Qiu, and S. S. Lam. SmartTunnel: Achieving reliability in the internet. In *INFOCOM*, 2007.
- [12] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *Proc. SOSP*, 2005.
- [13] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *Proc. SIGCOMM*, 2005.
- [14] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iplane: An information plane for distributed services. In *Proc. OSDI*, Nov 2006.
- [15] H. V. Madhyastha, A. Venkataramani, A. Krishnamurthy, and T. Anderson. Oasis: An Overlay-Aware Network Stack. In *Operating Systems Review*, pages 41–48, 2006.
- [16] Y. Mao, B. Knutsson, H. Lu, and J. M. Smith. DHARMA: Distributed Home Agent for Robust Mobile Access. In *IEEE INFOCOM*, 2005.
- [17] N. Milanovic and M. Malek. Current solutions for web service composition. *IEEE Internet Computing*, 8(6):51–59, Nov 2004.
- [18] A. Nakao, L. Peterson, and A. Bavier. A Routing Underlay for Overlay Networks. In *Proc. SIGCOMM*, 2003.
- [19] L. Peterson, S. Shenker, and J. Turner. Overcoming the Internet Impasse Through Virtualization. In *HotNets-III*, 2004.
- [20] PlanetLab. Global testbed. <http://www.planet-lab.org/>.
- [21] A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering queries using templates with binding patterns. In *ACM Symposium on Principles of Database Systems*, 1995.
- [22] R. Ramakrishnan and J. D. Ullman. A Survey of Research on Deductive Database Systems. *Journal of Logic Programming*, 23(2):125–149, 1993.
- [23] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. Opendht: a public dht service and its uses. In *Proc. SIGCOMM*, 2005.
- [24] A. Rodriguez, C. Killian, S. Bhat, D. Kostic, and A. Vahdat. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks”. In *Proc. of NSDI*, March 2004.
- [25] S. Savage, T. Anderson, A. Aggarwal, D. Becker, N. Cardwell, A. Collins, E. Hoffman, J. Snell, A. Vahdat, G. Voelker, and J. Zahorjan. Detour: A Case for Informed Internet Routing and Transport. In *IEEE Micro*, Jan 1999.
- [26] E. Shi, D. Andersen, A. Perrig, and I. Stoica. OverDoSe: A Generic DDoS solution using an Overlay Network. Technical Report CMU-CS-06-114, Carnegie Mellon University, 2006.
- [27] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. In *Proc. SIGCOMM*, 2002.
- [28] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of SIGCOMM*, Aug 2001.
- [29] L. Subramanian, I. Stoica, H. Balakrishnan, and R. Katz. OverQoS: An Overlay Based Architecture for Enhancing Internet QoS. In *Proceedings of NSDI*, 2004.
- [30] J. T. Wroclawski. The Metanet. In *Proc. Workshop on Research Directions for the Next Generation Internet*, 1997.
- [31] J. Yonan. OpenVPN: Building and Integrating Virtual Private Networks. <http://www.openvpn.net>.
- [32] M. Zhang, J. Lai, A. Krishnamurthy, L. Peterson, and R. Wang. A transport layer approach for improving end-to-end performance and robustness using redundant paths. In *Proc of USENIX ATC*, 2004.
- [33] S. Q. Zhuang, K. Lai, I. Stoica, R. H. Katz, and S. Shenker. Host Mobility using an Internet Indirection Infrastructure. In *ACM/Usenix Mobisys*, 2003.