USENIX Association

# Proceedings of the First Symposium on Networked Systems Design and Implementation

San Francisco, CA, USA
March 29–31, 2004

# MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks*

Adolfo Rodriguez, Charles Killian, Sooraj Bhat †, Dejan Kostić
Duke University
{razor,ckillian,dkostic}@cs.duke.edu

Amin Vahdat ‡
UC San Diego
vahdat@cs.ucsd.edu

**Abstract**

Currently, researchers designing and implementing large-scale overlay services employ disparate techniques at each stage in the production cycle: design, implementation, experimentation, and evaluation. As a result, complex and tedious tasks are often duplicated leading to ineffective resource use and difficulty in fairly comparing competing algorithms. In this paper, we present MACEDON, an infrastructure that provides facilities to: i) specify distributed algorithms in a concise domain-specific language; ii) generate code that executes in popular evaluation infrastructures and in live networks; iii) leverage an overlay-generic API to simplify the interoperability of algorithm implementations and applications; and iv) enable consistent experimental evaluation. We have used MACEDON to implement and evaluate a number of algorithms, including AMMO, Bullet, Chord, NICE, Overcast, Pastry, Scribe, and SplitStream, typically with only a few hundred lines of MACEDON code. Using our infrastructure, we are able to accurately reproduce or exceed published results and behavior demonstrated by current publicly available implementations.

## 1 Introduction

Designing and implementing robust, high-performance networked systems is difficult. Overcoming this difficulty is increasingly important as an ever larger fraction of the world's infrastructure comes to rely upon networked systems. Challenges include network and host failures, highly variable communication patterns, race conditions, reproducing bugs, and security. While the advent of higher-level programming languages such as Java has raised the level of abstraction and somewhat eased this burden, most programmers are still faced with the daunting task of re-inventing appropriate techniques for dealing with asynchronous, failure-prone network environments known by a handful of elite programmers.

We seek to explore the appropriate programming models and development environments with the twin goals of: i) making it easier to advance the state of the art in building robust networked systems, and ii) bringing this state of the art to programmers at large. While this is an ambitious effort, this paper attempts to uncover some of the relevant issues by focusing on programming language and runtime support for designing, implementing, and evaluating an emerging class of distributed services, overlay networks. We initially focus on a few types of overlays (distributed hash tables, DHTs, [22, 25, 30] and application-layer multicast [6, 9, 12, 16, 23, 24, 31]), though we believe our framework is applicable to other classes of overlays such as indirect routing (e.g. RON [2]), 6Bone, and BGP.

We view current overlay research as following a cycle consisting of four phases, each of which suffers from a number of challenges. First, an overlay researcher designs an algorithm that optimizes for network metrics such as latency and provides for application behavior such as $O(\lg n)$ routing hops in DHTs. In the second phase, one or more implementations are created to evaluate algorithm performance. For example, many researchers create hand-crafted simulators for evaluating performance under scale and live implementations for evaluation in real settings. Creating such implementations is often tedious and difficult, both due to the size of software components needed and the complexity of such functionality.

Using an algorithm's implementations, researchers use experimentation to gather run-time performance data in the third phase. Usually, this includes both simulation (such as with the network simulator, ns [29]) and small-scale live Internet runs (e.g. PlanetLab [19]). Unfortunately, custom simulation does not capture the full intricacies of network behavior such as congestion

*This research is supported in part by the National Science Foundation (EIA-9972879, ITR-0082912), Hewlett Packard, IBM, Intel, and Microsoft. Additional information is available on the MACEDON website: http://www.cs.duke.edu/∼razor/MACEDON.

†Now a student at Georgia Institute of Technology, sooraj@cc.gatech.edu

‡Supported by NSF CAREER award (CCR-9984328).

and queuing. While ns might address this shortcoming, it faces scalability limitations beyond a few hundred nodes, making overlay evaluation problematic. Live deployment certainly provides an existence proof, but does not enable evaluation under scale or highly dynamic conditions. The final phase, evaluation, involves processing the information generated through experimentation using hand-crafted tools. Researchers subsequently modify their implementations in light of code bugs or suboptimal performance. They employ disparate implementation techniques, causing the evaluation of competing overlays to reflect differences in implementation methodologies rather than in algorithmic principles and design.

To address these limitations, we present MACEDON, an infrastructure to simplify the design, development, evaluation, and comparison of large-scale overlays. In MACEDON, researchers specify algorithm behavior in terms of event-driven finite state machines (FSMs) consisting of system *states*, *events* (e.g. message reception, remote node failure, etc.), and *transitions* indicating the actions to take in response to events. From this high-level specification, MACEDON generates code for a variety of experimentation infrastructures leveraging shared (but extensible) libraries. The libraries implement much of the base overlay maintenance functionality, such as thread and timer management, network communication, debugging, and state serialization. As such, improvements in system support can be equally applied to all protocols. Ultimately, these system mechanisms enable fair comparisons of the merits of individual *algorithms* rather than artifacts of particular *implementations*.

MACEDON currently generates native C++ that runs unmodified in live Internet settings, including PlanetLab, and the ModelNet large-scale network emulator [27]. ModelNet enables us to subject overlays of thousands of nodes to the characteristics of large network topologies, capturing both scale and realism. MACEDON eliminates the need to maintain multiple versions of the same algorithm for different evaluation infrastructures. We provide built-in support for tracking popular overlay evaluation metrics, such as average delay penalty, communication overhead, and communication stretch. Our evaluation tools enable researchers to gain deeper understanding into the complex behavior of their algorithms, thus closing the streamlined development cycle.

To validate the utility of our approach, we have implemented of a number of overlays in the MACEDON framework. We have leveraged MACEDON to guide our design of AMMO [21] and Bullet [16]. Our MACEDON implementations also include Chord [25], NICE [4], Overcast [13], Pastry [22], Scribe [24] and SplitStream [6]. Here, we compare our generated code with published results and publicly available implementations. Our comparison indicates that MACEDON is able to reproduce or exceed performance of these systems, with concise system descriptions consisting of a few hundred lines. Using a standard API, applications and protocols coded to services of one overlay may easily switch to another providing similar functionality. For instance, the Scribe application-layer multicast protocol can be switched from using Pastry to Chord by changing a single line in its MACEDON specification. In isolation, our protocol implementations constitute an important contribution: the validation of results published separately by other authors. Taken together, they demonstrate the generality and utility of the MACEDON framework for developing and comparing overlays. Over time, we hope that code for a wide variety of overlays will become publicly available, further lowering the barrier to experiment with new ideas in this space.

This paper is organized as follows. We define overlays in terms of an abstraction in Section 2. Section 3 gives details of overlay implementation using the MACEDON language. We present validation of our methodology in Section 4. Section 5 compares MACEDON with other implementation infrastructures and describes other related work. We conclude with future work in Section 6.

## 2 Overlay Abstraction

We seek to provide a representation of distributed algorithms that is expressive enough to characterize the intricacies of different protocols, yet simple enough to facilitate implementation. To this end, we identify common characteristics of overlays and describe our FSM-based approach of describing them. We show how we use this unifying abstraction to enable concise descriptions of a wide array of network protocols. While it is not feasible to prove that all overlays share these characteristics, we have yet to encounter one that does not.

At a high level, an overlay network is a distributed algorithm where nodes establish logical *peer* or *neighbor* relationships with some subset of global participants forming a logical network *overlayed* atop the IP substrate. Examples include advanced communication semantics provided by multicast overlays and network maintenance as performed by BGP routers. A subset of these overlays export APIs that allow applications to transmit data through them. Our initial MACEDON implementation focuses on these algorithms, though we note that MACEDON is a generic framework for developing a wide variety of distributed systems. In particular, our initial work targets distributed hash tables (DHTs) and application-level multicast, described further in Section 5.

## 2.1 FSM representation

Overlay nodes maintain local state regarding their current activities and communicate with neighbors through control messages. They use periodic timers to schedule future processing and may receive application commands instructing them to perform an operation. The fundamental premise of our approach is that these characteristics can be succinctly described by event-driven finite state machines (FSMs). In this model, *events* such as message reception, scheduled completion of timers, and application commands, trigger the overlay protocol to perform protocol *actions*. Actions include setting local node state, transmitting new messages, scheduling timers, and delivering application data, though this is not an exhaustive list. Events may occur nearly simultaneously, perhaps requiring the serialization of local state. In addition, events may cause the protocol to move from one *system state*, or phase of execution, to another. Behavior toward an event while in a certain system state may be different when in another state. In summary, we believe that we can sufficiently capture an overlay's intricate behavior by describing its system states, local node state, events, and the response to these events in this FSM framework. The following subsections describe the components of the MACEDON FSM abstraction.

### 2.1.1 Node state

Each overlay node maintains local state describing its current position and activities. Local state determines each node's relationship to current neighbors. For example, a tree-based overlay (e.g. Overcast) will have parent and children neighbors. The behavior of the node toward a peer may be different depending on its peer type. It may also maintain a list of *potential* peers to establish future peer relationships. This functionality is not required in certain overlays, when nodes either establish a peer relationship with or delete any knowledge of a potential peer. In other overlays, such as RON [2], such state may include a list of *all* nodes present in the overlay. Node state may also include specialized information that identifies characteristics about this node's position in the overlay. Examples include bandwidth estimates to neighbor nodes as in Overcast [13] and routing tables in DHTs. We term this type of node state *state variables*.

In addition, algorithms have system states that represent high-level phases of processing. For example, upon initialization, a node in the overlay may enter a"joining" phase where a join request message is transmitted to a node already in the overlay. A "probing" state could be where nodes probe a certain population of overlay participants, for instance, to reduce latency in the overlay.

### 2.1.2 Events

In our target systems, asynchronous events move a node from one system state to another, performing subsequent actions such as sending a message. Events include timer expirations, message reception, and API function calls. In message reception, a node processes the message, performing appropriate actions in response. For instance, a node receiving a join request may attempt to add the joining node as its neighbor.

When a scheduled timer expires, the node performs functions appropriate to this timer. For example, a NICE [4] node schedules timers to check protocol invariants. If a node cluster is unsuitably large or small, the node initiates a cluster split or merge. The node would change its system state and perform a number of coordinating actions. In another event, an application issues commands to the overlay. Upon receiving the call, the node may change its state and perform appropriate actions. Application commands fall into two categories, control commands for administrative operations and data commands for transmitting data through the overlay.

The distinction between control and data operations is central to MACEDON's handling of asynchronous events. Control operations modify node state and are exclusively serialized within a protocol instance. Data operations simply read node state, enabling shared protocol access. In MACEDON, events may occur simultaneously. For example, an application may spawn multiple threads, each of which can make an API call (control or data) into the overlay instance, thereby leading to potential race conditions. Likewise, multiple timer and transport threads may execute simultaneously. By allowing multiple data operations to proceed simultaneously, MACEDON exploits the advantages of multithreaded programming to achieve superior performance in delivering data through the overlay.

Overlay developers classify transitions as *control* (requiring write access to node state) or *data* (only read access). Using this classification, we determine the proper level of protocol instance locking on a per-transition basis. Each instance is secured with a read/write lock. Control operations secure the lock exclusively for writing, while data operations use read locking to allow multiple threads to execute in parallel, increasing performance when working threads block or a multi-processor is available.

### 2.1.3 Actions

A *transition*, representing a series of related actions, is uniquely identified by (event, FSM state). That is, the current system state determines an algorithm's response to specific events. For example, once a node has joined

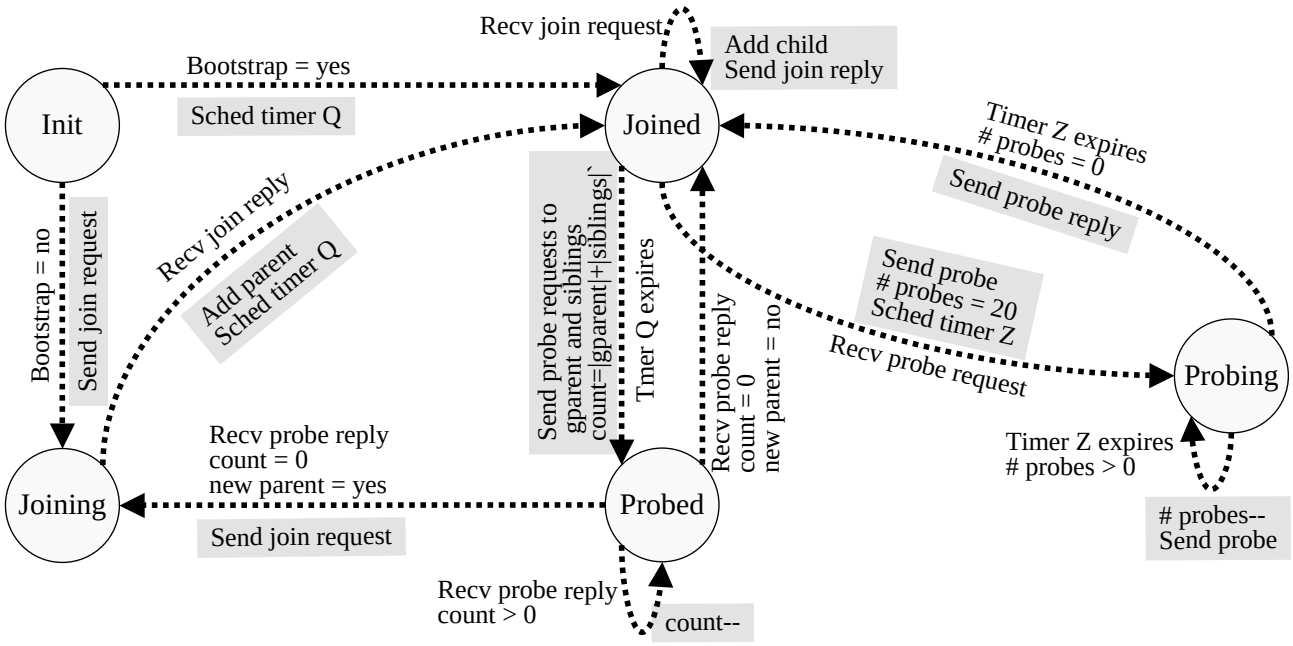


Figure 1: Portion of the Overcast algorithm representation. Circles represent FSM states. Directed edges identify transitions, with events as unshaded text and actions in shaded text boxes.

an overlay, it may transition from the joining phase and transmit a confirmation. Actions include scheduling a timer, transmitting a message, and changing node state. Overlays also employ periodic timers to execute overlay maintenance, check invariants, or some other periodic action. For example, Chord periodically checks and repairs its routing (finger) table entries.

Messages provide the fundamental mechanism for coordinating distributed actions and transmitting data. In Chord, a node transmits route repair requests to its neighbors. Chord nodes also transmit data messages through the overlay enabling application-to-application communication. Finally, an overlay protocol specifies which state variables change upon an event. In Chord, a node gathers route replies to determine if its route entries are stale, and if so, updates them accordingly.

#### 2.1.4 An example: Overcast

An overlay's FSM behavior is specified in a *mac file*. To aid specification, it is helpful to first describe the high-level behavior of the algorithm graphically, as illustrated in Figure 1. The figure shows Overcast's [13] five system states and associated transitions. The protocol begins in the "init" state from which it transitions to the "joined" state if this node is the bootstrap node (i.e. the designated root of the overlay). Otherwise, it transmits a join request to the bootstrap. A joined node (including the bootstrap) receiving a join request will add the incoming node to its child neighbor list and transmit a join reply to confirm the process. Upon receiving the join reply, the joining node stores its new parent in its parent neighbor list and transitions to the "joined" state.

The Q timer allows joined nodes to periodically evaluate their position. When the timer expires, a node initiates probes from its grandparent and siblings (we omit the details of how a node acquires this information) and enters the "probed" state. It uses a state field to count the number of nodes probing it. Upon receiving a probe request, nodes send equally-spaced probes at some defined rate using the Z timer. Once all probes are transmitted, the probing node transmits the probe reply and returns to the "joined" state. After the probed node gathers the necessary replies from all nodes (count=0), it decides whether it should move to a new parent. If it moves, it again enters the "joining" state and sends a join request to the new parent. Otherwise, it simply returns to the "joined" state. Section 3 describes how this high-level representation is captured in the Overcast mac file.

### 2.2 The MACEDON API

Overlay algorithms typically target specific types of applications. An important characteristic of their implementation is the API they export. For example, a multicast overlay must export a send function to disseminate

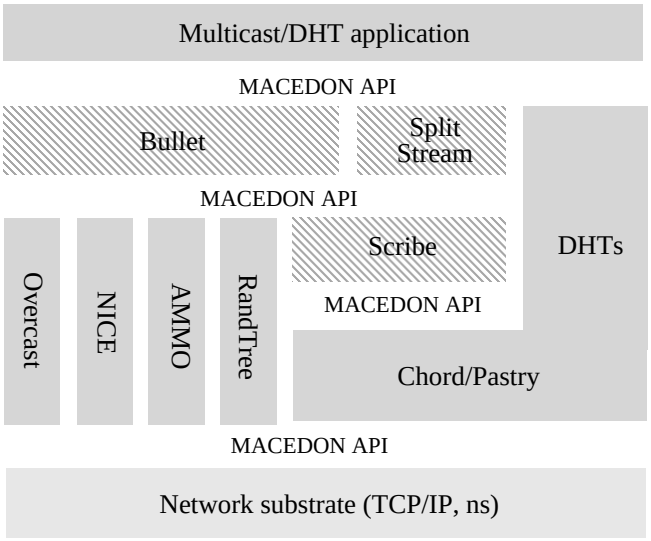


Figure 2: The MACEDON protocol stack.

data through the overlay. While sometimes obfuscated in design, we believe it is imperative for overlay implementations to provide appropriate APIs to application developers. A number of recent efforts [10, 22] have made initial steps at creating a single, standard API. We adopt an API similar to [10] and further enable API extensibility for protocol-specific functionality.

A standard API enables MACEDON applications to select underlying overlays without modification. In general, overlays support multicast or route primitives that route data from a source to destination(s) *through* the overlay. Typically, overlays provide upcalls at each routing hop so that intermediate nodes can perform application-specific functionality. For example, an intermediate Scribe node receiving a join request for a group will add the group to its list of multicast sessions and propagate the request toward the destination, thus building a reverse-path distribution tree.

Protocol layering (Figure 2) is central to implementing algorithms in MACEDON. The MACEDON protocol stack is divided into three components: application, multiple protocol layers, and network substrate (ns or TCP/IP). Much like the TCP/IP stack, higher layers in MACEDON use the services of lower layers. Bullet, for example, uses a simple randomly constructed tree, RandTree, for baseline data distribution.

Figure 3 illustrates a simplified version of the API that MACEDON overlays export. We provide an extensible upcall and downcall mechanism to perform protocol-specific collaboration across layers in the stack. As instances of this mechanism, we describe `forward()`, `deliver()`, and `notify()` (extensible upcalls are handled using the generic handler). A node calls `forward()`

```
typedef int (*macedon_forward_handler)
   (char *msg, int size, int type,
    int nextHop, macedon_key nextHopKey);
typedef void (*macedon_deliver_handler)
   (char *msg, int size, int type);
typedef void (*macedon_notify_handler)
   (int type, int size, int *neighbors);
typedef int (*macedon_upcall_handler)
   (int operation, void *arg);
macedon_init(macedon_key bootstrap, int prot);
void macedon_register_handlers(
  macedon_forward_handler,macedon_deliver_handler,
  macedon_notify_handler,macedon_upcall_handler);
int macedon_create_group(macedon_key groupID);
void macedon_join(macedon_key groupID);
void macedon_leave(macedon_key groupID);
int macedon_route(macedon_key dest, char *msg,
     int size, int priority);
int macedon_multicast(macedon_key groupID,
     char *msg, int size, int priority);
int macedon_anycast(macedon_key groupID,
     char *msg, int size, int priority);
int macedon_routeIP(int dest, char *msg,
     int size, int priority);
```

Figure 3: Simplified MACEDON API.

once it makes a message routing decision. Intermediate nodes can change the message or its destination or quash the message altogether. The `notify()` upcall allows lower-layer protocols to inform higher layers of changes in neighbor lists (a higher layer may require this direct knowledge). An application optionally registers its upcall handlers with the `macedon_register_handlers()` function. At least one handler is necessary if the application is to receive any data through the overlay (having null handlers would be used when evaluating just the construction process of different overlays).

Figure 3 also shows `macedon_init()` that initializes an overlay identified by the application-specified well-known protocol value (akin to protocol values in IP). Once an application initializes and registers its handlers, it can send and receive data. For unicast data, the overlay must implement *routing* functionality that determines which neighbor receives data packets next. The `macedon_route()` function accepts a message and destination in the form of a macedon_key, meaning it is not necessarily an IP address (it could be a hash of an IP address or name). A similar primitive is `macedon_routeIP()` that enables native IP-based communication with an IP host.

Multicast primitives include `macedon_create_group()` to create sessions. Its sole input is the value, or handle, associated with the session (group). Receivers join and leave a session with `macedon_join()` and

```
<PROTOCOL SPECIFICATION>: <HDRS>
  <STATE AND DATA><TRANSITIONS><ROUTINES>(0,1)
<HDRS>: "protocol" <name> ["uses" <base>](0,1)
  "addressing " ["hash"|"ip"]
  "trace_" ["off"|"low"|"med"|"high"]
<STATE AND DATA>: <CONSTANTS> <STATES>
  <NEIGHBOR TYPES> <TRANSPORTS>(0,1)
  <MESSAGES> <STATE VARS>
<STATES>: "states { " [<name> ";"]* "}"
<TRANSPORTS>: "transports {" <TRANSPORT>+ "}"
<TRANSPORT>: ["TCP"|"UDP"|"SWP"] <name> ";"
<MESSAGES>: "messages {" <MESSAGE>* "}"
<MESSAGE>: <transport name>(0,1) <name>
  "{" <MESSAGE FIELDS>* "}"
<STATE VARS>: "auxiliary data {"
  [<LOCAL VAR>|<NEIGHBOR VAR|<TIMER VAR>]*"}"
<NEIGHBOR VAR>: "fail_detect"(0,1) <name>
  <size>(0,1) <size>(0,1) ";"
<TIMER VAR>: "timer" <name><period>(0,1) ";"
<TRANSITIONS>: "transitions {"[<STATE EXPR>
  [<API TRANS>|<TIMER TRANS>|<MESSAGE TRANS>]
  <TRANS OPTIONS> "{" <code> "}"]* "}"
<API TRANS>: "API " <API TYPE>
<API TYPE>:  "init"|"route"|"routeIP"|
  "multicast"| ... |"join"|"upcall_ext"
<TIMER TRANS>:  "timer" <name>
<MESSAGE TRANS>: ["forward"|"recv"] <message>
```

Figure 4: MACEDON grammar highlights.

`macedon_leave()`, specifying the group value. Similar to `macedon_route()`, `macedon_multicast()` requires a session's ID instead of a node's destination address. `macedon_collect()` introduces a new primitive to traditional overlay APIs. It essentially performs the opposite of multicast, where data originates at non-root nodes and is collected via the distribution tree toward the root. Intermediate nodes can summarize data in an application-specific manner, ultimately delivering a global summary to the tree's root. We believe that a number of applications could benefit from this communication paradigm.

## 3 MACEDON Framework

This section describes how a developer can specify overlay behavior in MACEDON. We give an overview of the language and discuss its expressiveness. We also describe how MACEDON captures subtle implementation details that greatly influence overlay performance.

### 3.1 Grammar Overview

Figure 4 highlights the MACEDON language grammar. It allows a developer to define a `PROTOCOL SPECIFICATION` that MACEDON translates into working code. There are three main headers in mac file. The protocol header specifies the name of the protocol and optionally a base protocol for layering. For example, one could specify "protocol scribe uses pastry" to run Scribe over Pastry, or "protocol scribe uses chord" to change the underlying DHT. In this manner, one could perform a direct comparison between the two DHTs in support of application-layer multicast. The addressing header specifies whether the protocol uses IP- or hash-based addressing. One could add other types of addressing, for example, to test new hashing algorithms or node identifier schemes. The tracing header can be set to any of four increasing levels of automatic tracing.

The `STATE AND DATA` section includes definitions of states, neighbor types, transports, messages, and state variables. The `STATES` portion defines the allowed set of protocol FSM states. The "init" state is automatically generated as the starting state for all protocols. For Overcast, state definitions are: (refer to Figure 1):

```
states { joining; probing; probed; joined; }
```

The `NEIGHBOR TYPES` section specifies the sets of neighbors the protocol tracks (and their maximum number). Neighbor types may specify optional fields, such as delay, to track on a per-neighbor basis. Note that a field might itself be a set of neighbors. Returning to our example, Overcast nodes have parent and children neighbors:

```
neighbor_types {
  oparent 1 { ... // fields omitted }
  ochildren MAX_CHILDREN { ... // fields omitted }
}
```

The protocol also specifies persistent state variables. In addition to standard language types, neighbor sets, and multidimensional arrays of such types, state variables can specify timers with a specified expiration period. A neighbor list may be labeled "fail_detect", instructing MACEDON to monitor these neighbors for failure. Upon detecting failure, MACEDON will invoke an overlay's error API transition. Our Overcast specification includes the following state variables:

```
state_variables {
  oparent papa;  // parent neighbor
  ochildren kids;  // children neighbors
  oparent grandpa;  // grandparent neighbor
  ochildren brothers;  // sibling neighbors
  int probed_node;  // node we are probing
  int probes_to_send;  // count of probes left
  timer keep_probing;  // timer Z
  timer probe_requester;  // timer Q
  ...  // fields omitted for brevity
}
```

In MACEDON, the lowest-layer protocol specifies the transports it uses and associates transport instances with each message via `TRANSPORTS` and `MESSAGES` definitions. Messages may contain many fields, including standard

language types and neighbor sets. Communication in MACEDON can be reliable, congestion-friendly (using TCP), unreliable, congestion-unfriendly (using UDP), or reliable, congestion-unfriendly (using a simple sliding window protocol, SWP). It is sometimes advantageous to use *multiple* blocking transports (e.g. TCP) of the same type. This is particularly evident when one message has higher priority than another. If the transport is blocked sending low priority messages, it is unable to send any available high priority messages until the connection is unblocked. By defining multiple transport based on priority, this problem is easily overcome. For example, Overcast includes:

```
transports {
  SWP HIGHEST;
  TCP HIGH;
  TCP MED;
  TCP LOW;
  UDP BEST_EFFORT;
}
messages {
  BEST_EFFORT join { }
  HIGHEST join_reply { int response; }
  HIGHEST probe_request { ... // fields omitted }
}
```

Overcast includes three TCP transports, as well as one SWP and one UDP, and associates each message to the appropriate transport. In higher layers, a specification associates messages with a default *service class* or priority. A higher layer invokes the layer below to transmit the message, passing the desired priority along with it. The lower layer determines how to process the message at the given priority.

The `TRANSITIONS` section describes the bulk of an overlay's behavior. The developer uses a set of MACEDON primitives to describe the actions that result from triggered events. All transitions are scoped by a FSM state expression, thereby allowing a protocol to specify different behavior based on its current system state. A developer may specify transition-specific options, such as write versus read serialization (write semantics are assumed by default). There are three types of transitions: API, timer, and message. While our Overcast specification is too large to include here, we summarize a few transitions (with actions removed for brevity):

```
transitions {
  any API route [locking read;] { ... }
  probing timer keep_probing [locking read;] { ... }
  !(joining|init) recv join { ... }
}
```

An API transition enables layers to communicate with layers directly above and below in the MACEDON stack. The "init" API is called by a higher layer to initialize protocol state and schedule necessary timers. "route", "routeIP", "multicast", "anycast", and "collect" represent requests to transmit data. Our example shows the declaration of Overcast's route API with read locking semantics. "create_group", "join", and "leave" are control calls for managing multicast session state. The remaining API calls represent atypical or extensible calls into the code, including notifying upper layers of a changed neighbor set, generic "upcall_ext" and "downcall_ext" to provide extensible specification of layer-to-layer collaboration, and failure detection ("error"). Our current implementation assumes the failure of a peer node if no message has been received from it in $f$ seconds, a configurable parameter. If communication has ceased for $g < f$ seconds (another parameter), MACEDON triggers a heartbeat request/response sequence to solicit communication. Appropriate failure detection is an ongoing area of research. We consider MACEDON to be an appropriate framework for such research.

A timer transition occurs upon a timer expiration. In Overcast, the "keep_probing" timer fires when a node is transmitting probes. In this case, the node is in the "probing" state and follows read locking semantics since no node state is modified within the transition. Finally, a message transition is called in response to message reception. In addition to state scoping, these transitions are scoped by message type, enabling different transitions for different messages. In MACEDON, messages are *delivered* (this is the final destination) or *forwarded* (this node should forward the message). In our example, we have specified a join message reception when the state matches the expression "`!(joining|init)`", i.e. the Overcast node is in "joined", "probing", or "probed" states. This transition modifies state variables and makes use of the default write locking semantics.

### 3.2 Code Generation

MACEDON generates API-consistent code, termed the MACEDON *agent*, from an algorithm's specification. MACEDON parses a specification and translates it into executable C++ code that uses library functions and the MACEDON code engine including timer and transport subsystems (we also have partial support for generating ns code for better reproducibility of results). The engine and code libraries are common to all overlay implementations, increasing evaluation consistency and code reuse. While our current infrastructure does not yet support other programming languages such as Java, it is the subject of ongoing work.

The translation phase involves the declaration of protocol messages, states, neighbor types, state variables, and transitions. We create a demultiplexing function to receive data packets from a MACEDON interoperability layer that in turn interfaces with ns or native TCP/IP sockets. Upon receiving a message, the demul-

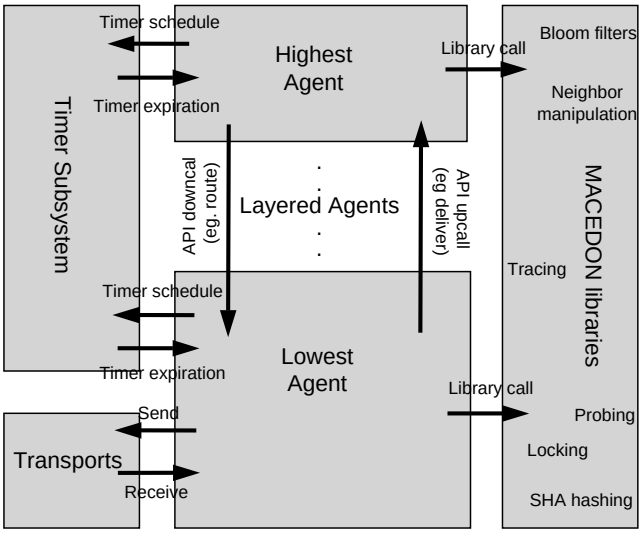


Figure 5: MACEDON agents.

tiplexing function calls the appropriate transition function based on the node's current state and the message type. MACEDON translates API specifications and timer transitions in much the same way.

Figure 5 outlines the resulting structure of MACEDON agents. MACEDON subsystems are implemented with thread pools that process timer and transport events. Along with application threads, they invoke transitions in agents. Timers can be employed by any layer in the MACEDON stack. However, only the lowest-layer agent may interact directly with the transport subsystem. Likewise, only the highest-layer agent interacts directly with the application. Though this example only shows two layered agents, MACEDON supports layering an arbitrary number of agents.

## 3.3 Specifying Actions

This section describes how an overlay developer invokes transition actions in MACEDON. While this could be done solely in the target programming language (i.e. C++), MACEDON provides libraries for invoking commonly-used actions, including the necessary functions to interface with our timer and transport subsystems as well as invoking cross-layer upcalls and downcalls. The MACEDON library collection is extensible, allowing users to add their own library routines. For example, we have created a library that manipulates bloom filters. The remainder of this section describes support for other commonly-used actions based on a sample transition of the Overcast specification, given in Figure 6. Line 2 of our sample transition shows how to access the "response" field of the incoming "join_reply" message.

```
1 joining recv join_reply {
2   if (field(response) == 1) {
3     if (neighbor_size(papa)) {
4       neighbor_oparent *pops =
5                 neighbor_random(papa);
6       route_remove(pops->ipaddr, 0, 0, -1);
7       neighbor_clear(papa);
8     }
9     neighbor_add(papa, from);
10    state_change(joined);
11    timer_resched(probe_requester, PINT);
12    neighbor_oparent *pops =
13              neighbor_random(papa);
14    if (neighbor_query(brothers, from)) {
15      neighbor_ochildren *newp =
16        neighbor_entry(brothers, from);
17      pops->delay = newp->delay;
18    }
19   upcall_notify(papa, NBR_TYPE_PARENT);
20  }
21  else {
22    if (neighbor_size(papa)) {
23      state_change(joined);
24    }
25    else { ... // omitted }
26  }
27 }
```

Figure 6: A sample Overcast transition.

Perhaps the most basic action specified within transitions is in changing system state, as specified in lines 10 and 23. Line 11 shows how we invoke the MACEDON timer subsystem to schedule a timer event. Finally, we demonstrate an upcall invocation in line 19.

### 3.3.1 Transmitting Messages

Overlay protocols transmit messages via lower layers (or underlying network substrate). MACEDON has built-in transmission primitives of the form:
⟨*API*⟩_⟨*msg*⟩(⟨*dest*⟩, ⟨*fields*⟩, ⟨*buffaddr*⟩, ⟨*buffsize*⟩, ⟨*pri*⟩)
Line 6 of our sample Overcast transition illustrates how we transmit the remove message to our "old" parent once we have determined that a move will occur. By specifying a buffer address and size of zero, this message will not be appending application data. Finally, the -1 priority requests use of the message's default transport.

### 3.3.2 Neighbor Management

MACEDON provides primitives to simplify neighbor list management. Our sample transition makes heavy use of these facilities. Lines 3 and 22 illustrate the `neighbor_size` function that returns the size of a neigh-

bor list. Line 9 adds a neighbor while line 7 shows how to clear a neighbor list. Neighbor lists can be queried as in line 14 ("from" is the source address of the inbound "join_reply" message) and accessed directly as in line 16. Finally, lines 5 and 13 illustrate selecting a random entry from a neighbor list.

Typically, overlays compare potential edges along some performance metric, such as round-trip time (e.g., NICE). Overcast estimates bandwidth by measuring the delay associated with receiving some number of probes at a sustained bandwidth. Line 17 shows how neighbor entries store this information. Additional neighbor entry fields could be maintained in such a manner.

#### 3.3.3 Explicit Thread Serialization

While locking behavior is specified on transition declarations, an overlay developer may required explicit access to an agent's (protocol instance) lock. That is, conditions under which locking is required may depend on intricate behavior within the transition itself. In this case, the transition could employ the `Lock_Write()`, `Lock_Read()`, and `Unlock()` primitives. In our experience, however, transition-based locking has been adequate for all the overlays we have considered.

## 4 Evaluation

In this section, we evaluate MACEDON's ability to: i) facilitate overlay design, implementation, and evaluation, ii) implement a broad range of algorithms with good performance and scalability characteristics, and iii) enable comparisons of competing overlay technologies. While it is not practical to prove that MACEDON will be able to meet the demands of all distributed algorithms, we use our success with of a broad variety of modern overlays to support our goal of qualitatively improving the way overlay research is conducted.

### 4.1 Expressiveness

One key contribution of this work is the implementation and validation of a broad range of network overlays in the MACEDON environment, including: AMMO [21], Bullet [16], Chord [25], NICE [4], Overcast [13], Pastry [22], Scribe [24], and SplitStream [6]. Figure 7 summarizes the lines of code (LOC) counts for each of these MACEDON specifications. NICE, being a more complex protocol than all others required approximately four weeks of skilled programmer's time to implement and debug. Its MACEDON specification is approximately 500 LOC

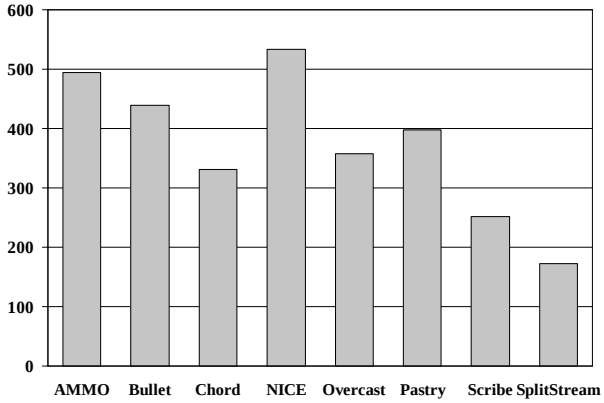


Figure 7: Lines of code used in various algorithm specifications.

while its generated C++ code is over 2500. The MACEDON operating environment is around 3500 LOC, yielding an estimated total of 6000 C++ LOC to completely implement NICE from scratch.

On the other end of the spectrum, SplitStream's MACEDON specification is under 200 lines of code, primarily because SplitStream, being layered on top of Scribe and Pastry, exploits functionality provided by those systems. Implementing SplitStream also required small changes to our Scribe implementation, primarily since the description of SplitStream [6] requires changes to Scribe's "pushdown" function. Though SplitStream and Scribe are originally designed to run over Pastry, we note that MACEDON's layering feature in conjunction with its standard API allows us to switch underlying DHT layers easily. For instance, while our experiments show results for SplitStream running over Pastry, we are currently experimenting with using Chord as the underlying DHT.

### 4.2 Validation

This section provides validation of a subset (abbreviated for space reasons) of our MACEDON-generated implementations as compared to published results or freely available code distributions (MIT's lsd Chord [17] and FreePastry [20]). We further note that results included in [16] and [21] were achieved through MACEDON using the mac specifications described in this paper.

We believe that our results confirm the generality, accuracy, and performance of our infrastructure. We use the ModelNet [27] infrastructure to emulate large-scale Internet topologies, capturing hop-by-hop congestion and queuing behavior. For our NICE validation, we used extracted information from [4] to re-create the authors' Internet-like topology. Our evaluation infrastructure

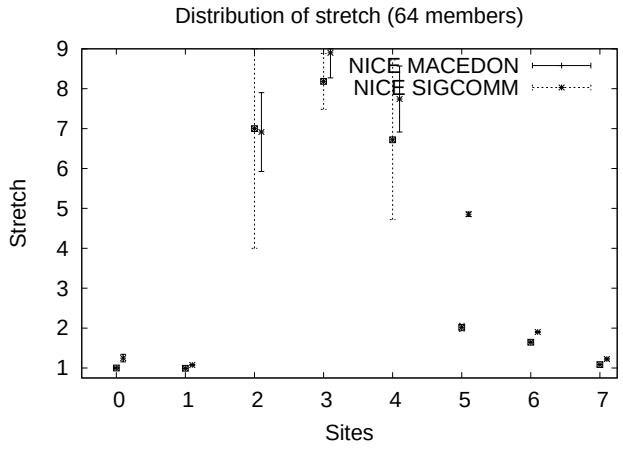


Figure 8: Observed stretch for published and MACEDON NICE implementations.

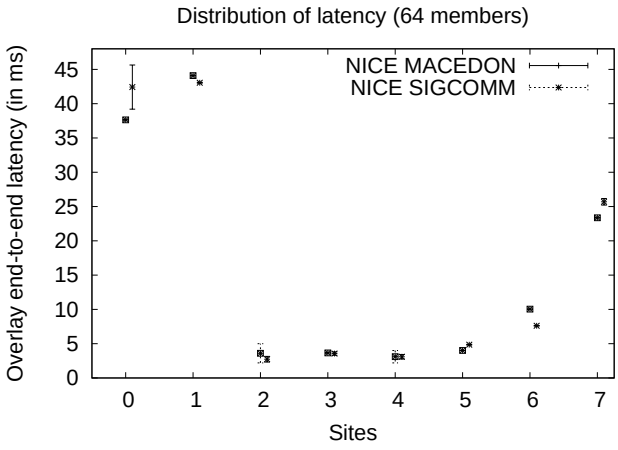


Figure 9: Observed latencies for published and MACEDON NICE implementations.

allows us to extract features of the resulting overlay by using ModelNet routing and topology information. For all other experiments, we use 20,000-node INET [8] topologies with varying numbers of clients (200–1000). In all cases, we run our experiments on (2–50) 1.4Ghz Pentium-III machines running Linux 2.4.23. We multiplex multiple node instances on these machines. All traffic passes through our 1Ghz Pentium-III ModelNet cores running FreeBSD-4.9. While all results in this paper use ModelNet, we note that we have successfully run smaller experiments (50+ nodes) over PlanetLab [19] (refer to [16] for sample results).

#### 4.2.1 NICE

To validate our implementation of NICE, we run the same experiments described in [4] for small-scale Internet scenarios (64 nodes) and compare our results with the published values. Figures 8 and 9 show the average observed stretch and latency for NICE nodes in each of eight different Internet sites as reported in Figures 15 and 16 in [4] versus our MACEDON implementation. We slightly offset the MACEDON values to the right for clarity. Our results closely match the published results, with only a minor discrepancy in one of the sites. We believe this is due to our implementation lacking the probe time binning strategy presented in [4], though adding this to our implementation is straightforward.

#### 4.2.2 Chord

We validated our MACEDON Chord implementation by comparing it to the MIT distribution, lsd. We used a 20,000-node INET topology with 1000 Chord participants for this experiment. We made two modifications to the MIT code to first dump all routing tables every two seconds (something already available in the MACEDON implementation via debugging features) and to use a smaller hash function, since our implementation of Chord only uses a 32-bit hash address space (nodes hash to the same hash address in MACEDON and lsd). We calculated correct routing tables for each node given global knowledge of all nodes joining the system.

Figure 10 shows the convergence of routing tables toward the correct values over time (averaged per-node) for MACEDON and lsd. The graphs shows two MACEDON curves, corresponding to two different settings of the "fix fingers" timer. This timer triggers Chord to route a repair request message to a random finger (routing) table entry. The ultimate destination of this message responds, allowing the requesting node to verify the correctness of that route entry. While the lsd code dynamically adjusts the period of the fix fingers timer, our current MACEDON implementation only supports static periods (1 and 20 seconds in this experiment).

The optimal strategy for dynamically adjusting protocol parameters such as timer periods is unclear. For example, our static 1-second strategy outperforms lsd's dynamic strategy. The converse is true with a 20-second timer setting, as convergence is much slower in this case. In both MACEDON cases, routing tables converge steadily as nodes join the Chord ring, eventually leveling off once all nodes have joined. In lsd, convergence is not as steady as fix fingers timers are dynamically adjusted. The goal of this experiment is to demonstrate MACEDON's ability to match or exceed that of lsd. Further, we note that MACEDON enables researchers to more effectively explore different dynamic timer strategies.

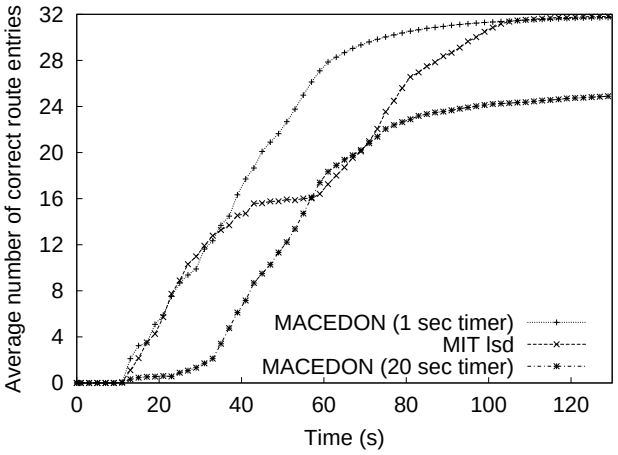


Figure 10: Convergence toward correct routing tables for MIT and MACEDON Chord implementations.

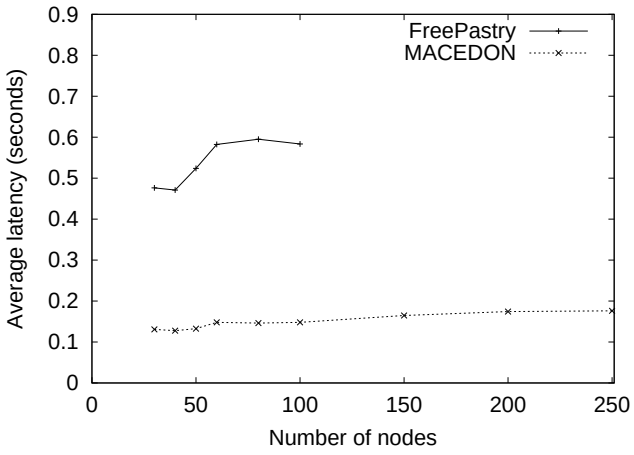


Figure 11: Average latency of received Pastry packets.

#### 4.2.3 Pastry

One goal of the MACEDON framework is to enable rapid prototyping of distributed systems while maintaining the performance and low-level optimizations available from hand-crafted C/C++ implementations. As one initial validation of our success against this metric, we compare the performance of the Pastry algorithm [22] implemented in MACEDON and within FreePastry [20]. MACEDON provides a high-level specification language with many of the same benefits of Java, along with libraries and routines specifically tailored to DHTs and overlays. However, it produces C++ code that does not suffer from some of the memory and performance overheads associated with Java and RMI. Our MACEDON Pastry implementation consists of 400 semicolons versus approximately 1,500 semicolons in the Java FreePastry implementation[1]. To quantify these benefits, we developed a simple test application to validate our Pastry implementation. Each application instance streams at some target data rate (10Kbps in this example) by sending 1000-byte packets at the given interval. On each data send, the application chooses a destination ID uniformly at random from the hash address space.

We estimate end-to-end delays for MACEDON Pastry and FreePastry [20] (using the RMI protocol). We varied the number of randomly selected nodes from our 20,000-node topology. For both systems, we allowed routing tables to converge for 300 seconds before streaming data. Due to our low streaming rate, intended targets received essentially all packets transmitted to them. For both systems, each node received approximately the same number of packets corresponding to the size of hash address space portion it owns. Figure 11 illustrates the average per-packet delays. We were unable to run FreePastry beyond 100 participants (two instances per physical machine) due to insufficient memory on our hardware. We have successfully run 20 MACEDON instances on these same machines. The graph shows that average latency in MACEDON is approximately 80% lower than in FreePastry, largely attributable to Java's RMI overhead. While FreePastry's "wire" protocol has yielded more favorable results (comparable to MACEDON in some cases), it is unstable in the current FreePastry release. Overall, our results show promise for MACEDON's ability to enable rapid prototyping while maintaining system performance.

[1]The FreePastry distribution consists of almost 15,000 semicolons, with significant functionality beyond Pastry. Our estimate is a conservative count based on manual inspection of the source tree.

### 4.3 Comparing Overlays

One important contribution of MACEDON is the creation of a fair and consistent overlay evaluation framework. To this end, MACEDON generates native TCP/IP code, allowing it to leverage ModelNet emulation and live deployment across the Internet, including the PlanetLab testbed (to support simulation environments, we also provide limited ns compatibility). MACEDON can automatically extract vital topology information from ns and ModelNet, allowing it to evaluate overlays against a wide array of metrics. Without such global information, it is impossible to accurately gauge an overlay's performance under certain metrics. For example, MACEDON can extract routing tables from ns and ModelNet to report the expected performance along metrics such as link stress, latency stretch, and relative delay penalty (RDP).

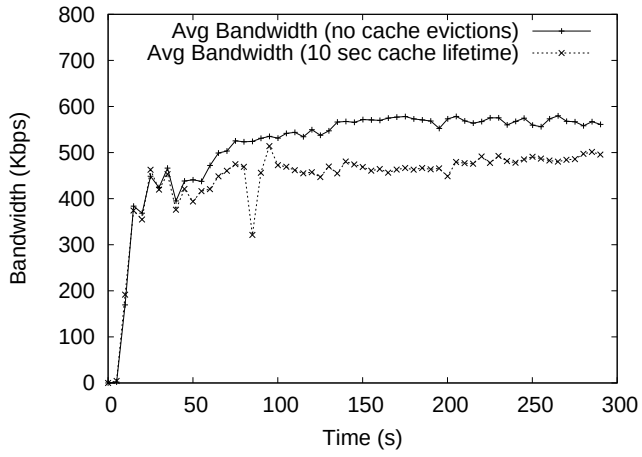


Figure 12: SplitStream bandwidth for two cache policies.

Disparate evaluation techniques have led to the use of many different performance evaluation metrics in overlay comparison. While an evaluation may be concerned with low link stress, it is unclear whether it is relevant to all applications. A high-bandwidth link could have link stress of hundreds for a chat-like application and perform well, while a link stress of two over a modem link for a video distribution would likely be unacceptable. As a result, it is challenging to choose the appropriate evaluation metric. MACEDON attempts to bridge this gap by providing a framework that can report a variety of popular evaluation metrics. We believe that MACEDON will encourage evaluation across more performance metrics, allowing the metrics themselves to be evaluated.

Our SplitStream experiments are designed to demonstrate MACEDON's ability to experiment with a variety of protocol features. For these tests, we created 300-node SplitStream forests. We developed a multicast application that streams 1000-byte packets at a predetermined rate (600Kbps for our experiments). Only one node is designated as the stream source while all other nodes join the multicast session as receivers. We first allow Pastry routing tables to converge by idling the system over the 300 seconds. Figure 12 shows the resulting per-node average bandwidth over time after the convergence period for two SplitStream flavors. SplitStream and Scribe use `macedon_routeIP()`, requesting that data be delivered directly over IP. Pastry does this by maintaining a *location cache* that maps hash addresses to IPs. Cache entries have an associated lifetime, thereby avoiding stale mappings that could lead to inefficient routing (a node could receive packets for a hash address it no longer owns). With cache eviction disabled, SplitStream delivers an average of 580 Kbps to each node (since no additional nodes enter the overlay, cache entries remain correct). With a one-second cache lifetime, bandwidth drops to 500 Kbps as additional bandwidth is consumed to re-establish stale cache entries. In summary, we believe that MACEDON is appropriate for carrying out such detailed and uniform protocol comparisons.

## 5 Related Work

MACEDON currently supports two types of overlays, distributed hash tables (DHTs) [11, 22, 25, 30] and application level multicast [4, 6, 12, 13, 15, 16, 22]. DHTs and their applications [9, 24, 23, 31] use hashing to map data objects and nodes to a logical address space for request routing. The hash value of a node determines which portion of the hash address space it *owns* and therefore, which data objects it will serve. By ensuring sublinear node (routing table) state and overlay width and depending on uniform server distribution (using consistent hashing), these overlay algorithms exhibit high performance and scalability.

Built on top of Pastry (or any other DHT), Scribe [24] creates multicast distribution trees rooted at the DHT node owning the multicast session ID. Receivers enter the session by routing join requests toward the root. Intermediate nodes along the path subsequently create a reverse path forwarding tree. Building on Scribe's success, SplitStream [6] uses multiple Scribe trees for data striping, thereby achieving higher bandwidth.

Other popular multicast overlays do not make use of DHTs. Most, including Overcast [13], NICE [4], and AMMO [21] create distribution trees optimized toward application-specific performance. In contrast, Bullet [16] creates a mesh where nodes exchange *summary tickets* that are used to select data peers. Nodes with disjoint data peer with one another. Since data is received from a number of carefully selected peers, Bullet nodes receive much higher bandwidth relative to tree-based overlays.

### 5.1 Evaluation Methodologies

The ns [29] network simulator provides a standard framework for accurate simulation of network protocols. Unfortunately, packet-level, congestion-aware simulation is costly, leading to inadequate scaling properties when evaluating overlays over a few hundred in size. For smaller-scale scenarios, ns provides an efficient and inexpensive mechanism for system evaluation. In the end, many researchers have created their own simulators, sacrificing accuracy for scale. These simulators tend to provide packet-level simulation but fail to account for congestion, packet loss, and queuing delays.

Such scale limitations are overcome by network emulation such as with ModelNet [27]. It enables the emulation of native IP applications by subjecting packets to link restrictions as specified by a network topology. It emulates routers' queuing delay and congestion. In our experiences, thousands of overlay nodes can run on 20-50 commodity PCs in the ModelNet environment. The same code runs unmodified in production Internet environments and testbeds, including PlanetLab [19]. As a result, ModelNet's accuracy and scalability makes it an appropriate choice for large-scale evaluation.

Neko [26] is another environment for developing and evaluating distributed algorithms. Similar to MACEDON, it allows the same algorithm specification to be used in a simulator and in a live system. MACEDON, however, provides a DSL, a set of libraries that address common issues in distributed algorithm development, and a generic API that facilitates interoperability between overlay algorithms and applications.

### 5.2 Related Languages

MACEDON is broadly related to domain-specific languages (DSLs) that typically generate functional code from domain-specific representations. Teapot [7] is a DSL for writing cache-coherence protocols. Like MACEDON, Teapot describes protocol behavior with the use of event-driven finite state machines. Teapot can generate "continuations" that allow nodes to suspend processing while waiting for a particular event. Unlike MACEDON, code generated by Teapot is not self-contained since the user must hand-code message handling functions. Additionally, Teapot's target domain (cache coherence protocols) is somewhat smaller than MACEDON's domain. Another domain-specific language is the Devil Interface Description Language (IDL) [18] designed for a substantially different domain than MACEDON, but is related in design. IDL can be used as documentation for hardware interfaces and can help driver development by reducing the burden of low-level programming. Devil also includes semantics for verifying specifications.

There has been substantial research in network protocol specification and implementation. RTAG [3] uses a context-free attribute grammar for protocol specification, emphasizing simplicity and portability. The grammar is used to capture event sequences allowed by the protocol. Morpheus [1] is an object-oriented language tailored for high-performance protocol implementations. It constrains a protocol designer to a set of design disciplines derived from experience, advocates the use of simple protocols that are selected and combined at runtime, and capitalizes on the knowledge of common patterns in protocol processing to optimize generated object code. Prolac [14], a lightweight object-oriented language, focuses on readability, modularity and extensibility. Its authors offer positive experiences with a TCP implementation. Prolac's *actions* allow arbitrary C code to be included; it is inserted into the C code produced by the Prolac compiler. Relative to these efforts, MACEDON is specifically geared toward overlay networks, focusing on a standard API, explicit support for protocol layering, and language support for common overlay functionality.

Beyond system specification, a number of languages target high-level design and protocol verification. These range from the highly mathematical, such as IOA [28] to more programmatic languages, such as TLA [5]. In contrast to MACEDON, neither generates functional code. IOA is an Input/Output Automaton specification language, allowing designers to specify one or more automatons to describe their system. IOA tools perform simulated execution that suggest likely invariants and automatically prove seemingly tedious portions of system specification. for formal verification. TLA is a high-level specification in a highly mathematical language. It is intended to be a design aid, and, combined with its model checker, can be used to find and remove flaws from system designs before system implementation.

## 6 Conclusions and Future Work

We have presented MACEDON to facilitate the design and implementation of overlay algorithms. Our system provides a domain-specific language for specifying the high level behavior of overlays such as DHTs and application-level multicast. MACEDON provides a common infrastructure that enables fair and consistent overlay evaluation. We make use of an overlay-generic API that enables protocol layering and facilitates porting applications from one overlay to another. Our results show that MACEDON can greatly decrease development and evaluation effort while yielding overlay implementations that closely resemble or outperform published results, including those for AMMO, Bullet, Overcast, NICE, Chord, Pastry, Scribe, and SplitStream. We believe that MACEDON can be used as an educational tool to understand the intricacies of overlay algorithms. Finally, we believe that the MACEDON vision extends beyond overlay algorithms to include a wider class of distributed algorithms, though this is the subject of future work.

## Acknowledgments

We would like to thank David Becker and Ken Yocum for their help with ModelNet. In addition, we thank our shepherd, Timothy Roscoe, Rebecca Braynard, and anonymous reviewers who provided excellent feedback.

## References


[1] Mark B. Abbott and Larry L. Peterson. A Language-Based Approach to Protocol Implementation. *IEEE/ACM Transactions on Networking*, 1(1):4–19, 1993.

[2] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient Overlay Networks. In *Proceedings of SOSP 2001*, October 2001.

[3] David P. Anderson and Lawrence H. Landweber. A grammar-based methodology for protocol specification and implementation. In *Proceedings of the ninth symposium on Data communications*, pages 63–70. ACM Press, 1985.

[4] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *Proceedings of ACM SIGCOMM 2002*, pages 165–175, 2002.

[5] Brannon Batson and Leslie Lamport. High-level specifications: Lessons from industry. In *Proceedings of the First International Symposium on Formal Methods for Comp onents and Objects*, Leiden, The Netherlands, March 2003.

[6] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. SplitStream: High-Bandwidth Multicast in Cooperative Environments. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, October 2003.

[7] Satish Chandra, Bradley Richards, and James R. Larus. Teapot: a domain-specific language for writing cache coherence protocols. *IEEE Transactions on Software Engineering*, 25(3):317–333, May/June 1999.

[8] Hyunseok Chang, Ramesh Govindan, Sugih Jamin, Scott Shenker, and Walter Willinger. Towards Capturing Representative AS-Level Internet Topologies. In *Proceedings of ACM SIGMETRICS*, June 2002.

[9] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, October 2001.

[10] Frank Dabek, Ben Zhao, Peter Drushcel, John Kubiatowicz, and Ion Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. In *2nd International Workshop on Peer-to-peer Systems (IPTPS'03)*, February 2003.

[11] Indranil Gupta, Ken Birman, Prakash Linga, Al Demers, and Robbert van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[12] Yang hua Chu, Sanjay G. Rao, Srinivasan Seshan, and Hui Zhang. Enabling Conferencing Applications on the Internet using an Overlay Multicast Architecture. In *Proceedings of ACM SIGCOMM*, August 2001.

[13] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and Jr. James W. O'Toole. Overcast: Reliable Multicasting with an Overlay Network. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, October 2000.

[14] Eddie Kohler, M. Frans Kaashoek, and David R. Montgomery. A Readable TCP in the Prolac Protocol Language. In *SIGCOMM*, pages 3–13, 1999.

[15] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, Abhijeet Bhirud, and Amin Vahdat. Using Random Subsets to Build Scalable Network Services. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, March 2003.

[16] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, October 2003.

[17] Massachusetts Institute of Technology. *lsd*, 2004. http://-www.pdos.lcs.mit.edu/chord/.

[18] Fabric Merillon, Laurent Reveillere, Charles Consel, Renaud Marlet, and Gilles Muller. Devil: An IDL for hardware programming. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Imp lementation (OSDI'2000)*, San Diego, California, October 2000.

[19] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of ACM HotNets-I*, October 2002.

[20] Rice University. *FreePastry*, 2004. http://www.cs.rice.edu/-CS/Systems/Pastry/FreePastry/.

[21] Adolfo Rodriguez, Dejan Kostić, and Amin Vahdat. Scalability in Adaptive Multi-Metric Overlays. In *The 24th International Conference on Distributed Computing Systems (ICDCS)*, March 2004.

[22] Antony Rowstron and Peter Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. In *Middleware'2001*, November 2001.

[23] Antony Rowstron and Peter Druschel. Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, October 2001.

[24] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The Design of a Large-scale Event Notification Infrastructure. In *Third International Workshop on Networked Group Communication*, November 2001.

[25] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer to Peer Lookup Service for Internet Applications. In *Proceedings of the 2001 SIGCOMM*, August 2001.

[26] Peter Urban, Xavier Defago, and Andre Schiper. Neko: A Single Environment to Simulate and Prototype Distributed Algorithms. In *ICOIN*, pages 503–511, 2001.

[27] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[28] Toh Ne Win, Michael D. Ernst, Stephen J. Garland, Dilsu n Kirli, and Nancy Lynch. Using simulated execution in verifying distributed algorithms. In *Fourth International Conference on Verification, Model Checking and Abstrac t Interpretation*, pages 283–297, New York, January 2003.

[29] Ellen W. Zegura, Kenneth Calvert, and M. Jeff Donahoo. A Quantitative Comparison of Graph-Based Models for Internet Topology. *IEEE/ACM Transactions on Networking*, 5(6), December 1997.

[30] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

[31] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-Area Data Dissemination. In *Proceedings of the Eleventh International Workshop on Network and Operating System Support for Digital Audio and Video*, 2001.