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

Adolfo Rodriguez, Charles Killian, Sooraj Bhat, Dejan Kostić, and Amin Vahdat *†
Department of Computer Science
Duke University
{razor,dkostic,vahdat}@cs.duke.edu

ABSTRACT

Overlay networks have become a popular mechanism for meeting the increasing demands of today's Internet services. Examples include distributed hash tables (DHTs) for file and object location and application-layer multicast for the delivery of streamed multimedia content. Currently, researchers and developers designing and implementing overlay algorithms employ disparate techniques at each stage in the life-cycle of overlays including design, code development, debugging, evaluation, and deployment. As a result, algorithmic principles, code infrastructure, development effort, and evaluation mechanics are duplicated and re-invented leading to both ineffective use of resources and difficulty in performing fair comparisons among competing algorithms and architectures.

We present MACEDON, an infrastructure that provides facilities to: i) specify overlay algorithms in a concise domain-specific language; ii) automatically generate algorithm code capable of executing in popular simulation and emulation environments as well as live across cluster and wide-area settings; iii) leverage a generic overlay network API geared at simplifying the interoperability of overlay implementations and applications; and iv) utilize a common framework for the consistent evaluation of overlays. Ultimately, the MACEDON framework eases algorithm design, accelerates code development, leverages code re-usability, facilitates consistent algorithm evaluation, and aids in understanding the complex behavior of overlays. We have used MACEDON to implement and evaluate a number of overlay network algorithms, including Bullet, Chord, NICE, Overcast, Pastry, Scribe, and SplitStream, typically with only a few hundred lines of MACEDON code. Using our development and evaluation 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. Addressing this difficulty is becoming increasingly important as an ever larger fraction of the world's infrastructure comes to rely upon networked systems. Challenges include failures, lack of fate-sharing among system components, highly variable communication patterns, race conditions, reproducing bugs, security concerns, etc. While the advent of higher-level programming languages such as Java has raised the level of abstraction and eased this burden, most programmers are still faced with the task of reinventing the appropriate techniques for dealing with asynchronous, failure-prone network environments known by a handful of elite programmers.

The larger goal of our work is to explore the appropriate programming and development environment with the twin goals of: i) making it easier to advance the state of the art in building robust and high performance networked systems, and ii) bringing this state of the art to programmers at large. While this is clearly an ambitious effort, this paper attempts to uncover some of the relevant issues by focusing on a programming language and runtime environment for the design, implementation, and evaluation of an emerging class of distributed services, overlay networks including distributed hash tables (DHTs) [15, 19, 21, 24] and application-layer multicast [3, 5, 7, 8, 11, 16, 18, 20, 25].

We aim to ease the implementation, deployment, and evaluation of overlays by allowing system developers to specify their protocol behavior at a high level, using a replicated state machine approach. Developers describe the set of system states their algorithm may be in, the set of messages that each node may transmit, and their algorithm's response to stimuli, such as message reception, remote node failure, timer expiration, etc. Runtime support and a set of libraries ease protocol development, providing support for common functionality, such as thread management, timers, debugging, network communication, and locking. We show how these system mechanisms enable fair comparisons of the merits of individual *algorithms* and *protocols* rather than artifacts of a particular *implementation*. In our common development framework, any improvements in system support for networked systems can be equally applied to all protocols.

Today, the state of the art in protocol development typically consists of simulating overlay behavior using a custom-built simulator or running small-scale (consisting of tens of nodes) experimental evaluations of working code across the Internet. These approaches suffer from a number of drawbacks. First, custom simulation infrastructures typically do not capture the full intricacies of network conditions such as congestion and queuing. The ns simulation infrastructure might address this shortcoming, but unfortunately it faces scalability limitations beyond a few hundred participants and was not designed to simulate distributed systems, making overlay evaluation problematic. Live deployment certainly provides an existence proof, but it does not allow experimenters to understand behavior under real scale or under highly dynamic conditions. Perhaps the most important limitation of current development and evaluation techniques however lies in the fact that individual researchers are evaluating their individual (often very insightful) ideas in isolation. It is difficult to compare one's own implemen-

*Technical Report CS-2003-09, September 2003.
†This research is supported in part by the National Science Foundation (EIA-9972879, ITR-0082912), Hewlett Packard, IBM, Intel, and Microsoft. Vahdat is also supported by an NSF CAREER award (CCR-9984328).

tation in a custom simulation environment to someone else's results. While code for some overlays is becoming publicly available in some cases, they are in various stages of maturity and robustness, resulting in comparison that reveal artifacts from implementation differences rather than from the relative merits of individual algorithms.

To address some of these limitations, we present the design, implementation, and evaluation of MACEDON (Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks), an infrastructure to simplify the development, debugging, experimentation, evaluation, and comparison of large-scale overlays in support of our more general goal of supporting robust, high-performance networked services. MACEDON's approach of specifying algorithm behavior in terms of finite state machines allows developers to think of their algorithm in the context of an event-driven architecture consisting of states, stimulants, and transitions to other states. Next, our domain-specific language allows the developer to focus, at a high level, on the specifics of the algorithm, simplifying the development process. 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. MACEDON makes much of the base functionality required to construct and maintain overlays available through shared (but extensible) libraries. Thus, in the common case, code for managing timers, probing for network conditions, etc. may be shared between different overlay algorithms. Further, any performance and functionality in these base libraries automatically become available to other algorithms built using MACEDON. We also support debugging and performance evaluations as first-class system features. There is built-in support for distributed event logging and for tracking popular overlay evaluation metrics, such as average delay penalty, communication overhead, and communication stretch.

One important feature of MACEDON is that, starting from a single high-level description, we are able to generate code for a variety of evaluation infrastructures. MACEDON currently generates ns code, enabling controlled and repeatable evaluation of system behavior under moderate scale. At the same time, we generate C++ code that runs unmodified in both live Internet settings, including the PlanetLab testbed [13], and the ModelNet large-scale network emulation environment [22]. ModelNet enables us to subject overlays consisting of thousands of nodes to the characteristics, including per-hop queuing and congestion, of target large-scale network topologies (e.g., we typically run with 20,000-node INET [4] generated topologies). In this way, MACEDON eliminates the need to maintain multiple versions of the same algorithm, one suitable for simulation and a second for live evaluation.

To validate the utility of our approach, we have completed a full implementation of a number of overlays in the MACEDON framework. These include Bullet [11], Chord [21], NICE [2], Overcast [9], Pastry [15], Scribe [18] and SplitStream [3]. Using descriptions of experiments run with these systems in published work, we compare experimental results for our generated code with the published results or those obtained via direct comparison with available implementations. Our comparison indicates that MACEDON is able to reproduce or exceed the performance of these systems, with concise system descriptions typically consisting of a few hundred lines of MACEDON code. Further, protocols may be generically layered on top of one another, by taking advantage of standard APIs between layers. For instance, the Scribe application-layer multicast protocol can be switched from running on top of a Pastry DHT to running on top of a Chord DHT by changing a single line in the protocol specification.

In isolation, our implementation of these protocols constitute an important contribution: the validation of results published separately by other authors. Taken together, our implementations and validations demonstrate the generality and utility of the MACEDON framework for developing and comparing network overlays.

The rest of this paper is organized as follows. Section 2 describes related overlay algorithms and the state of the art evaluation methodologies. We detail our system model in section 3 and describe the MACEDON API in 4. Section 5 gives details of how overlays are implemented using the MACEDON language, while section 6 identifies the pertinent characteristics of MACEDON's operating environment. We present validation of our methodology in section 7 and conclude with future work in section 8.

2. RELATED WORK

The MACEDON framework primarily aims at facilitating the implementation and evaluation of overlay networks. We roughly classify these into related camps, distributed hash table (DHT) overlays [15, 19, 21, 24] and application level multicast [2, 3, 7, 8, 10, 11, 9, 19]. DHT overlays and the applications [5, 16, 18, 20, 25] built on top of them use a hash-based approach 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 using sublinear node state, ensuring sublinear overlay width, and depending on uniform server distribution (by use of a consistent hash function or uniform random number generator) these overlay algorithms hope to limit the distance (in terms of whatever applicable performance metric) that a data or route request must travel, yet still scale to millions of nodes.

Because these overlays route requests and packets, each node in the overlay must maintain a *routing table* that indicates the direction that packets must follow to arrive at their destination. The Chord [21] system uses a circular routing paradigm that enables nodes to route toward a data object in $O(\lg n)$ steps. Nodes in the Chord ring join at uniformly distributed positions around the ring. By using a consistent hashing function, data requests likewise map to uniform positions around the Chord ring. The routing table at each node is $O(\lg n)$ in size, corresponding to segments in the circle of exponentially increasing size. In this way, Chord nodes can make great progress in the hash address space when routing towards points that are far away. Once the request is close to its eventual destination, jumps become increasingly smaller as the request zeroes in on its target.

Similar to Tapestry [24] and Plaxton [14], another interesting DHT technology is Pastry [19] which routes requests through the overlay based on address prefixes. Nodes joining the overlay are pseudo-randomly assigned a node ID that will determine its location in the overlay. In Pastry, a node peers with nodes based on the number of high-order bits in common. For every digit in its node ID, a node has a routing entry for each possible value of that digit. In this way, a node forwards data requests based on the entries corresponding to the highest order bit that differs between its address and the address of the request. At every hop during the routing process, a data request gets one digit closer to its destination, enabling Pastry to guarantee that the overlay's width is $O(\lg n)$.

Built on top of Pastry (though it can be easily implemented on top of many DHT systems), Scribe [18] enables application level multicast functionality. It does so by creating multicast distribution trees rooted at the node in the DHT overlay responsible for the multicast session ID. Source nodes create a multicast group by routing a request toward the multicast root that essentially acts as a rendezvous point. Receivers join the multicast by routing join requests, through Pastry, toward the root. Intermediate nodes along the path maintain state about multicast sessions, specifically regarding which nodes have registered with it to receive data, their children in the tree. They then forward the join requests if they have not joined the distribution tree themselves, thus creating a reverse path forwarding distribution tree. Building on Scribe's success, SplitStream [3] promotes the use of multiple Scribe trees accross which to stripe data, thereby achieving higher bandwidths.

Other popular application level multicast overlays do not make use of DHT functionality. The Overcast [9] system also provides a mechanism for the dissemination of high-bandwidth multicast data over a network overlay. In Overcast, nodes in the overlay cache a large amount of the multicast data so that they may provide the data stream to peer nodes in the overlay at the appropriate (measured) bandwidth. Nodes measure achievable bandwidth to their parent and compare it with the bandwidth attainable from its siblings. If the bandwidth is adequate, a node may move under a sibling, thereby reducing the amount of work its previous parent was responsible for.

NICE [2] maintains a hierarchy of clusters for the dissemination of latency sensitive data. Nodes form clusters in a *layer* with nearby nodes and elect the center of the cluster to represent the cluster in the next higher layer. This node recursively forms part of a cluster with other nodes in the higher layer that in turn elect a leader to send in the next higher layer. The leader of the highest layer becomes the rendezvous point of the distribution tree. Data to be multicast is sent to lower layers if it is received from a higher layer and vice versa. Ultimately, the NICE distribution is shown to perform well in terms of latency, while maintaining scalability.

SARO [10] makes use of a random subset utility called RanSub to deliver uniformly random subsets of global participants to each node. Nodes form an overlay multicast distribution tree and probe members of these subsets to search for better connectivity. While ensuring scalability, this method of making overlay transformations converges quickly to a good overlay. Also built on RanSub, Bullet [11] promotes the creation of a mesh on top of a control tree overlay. In Bullet, nodes append small *summary tickets* to RanSub messages allowing nodes to estimate the useful of peers received in random subsets. Mesh edges are established with nodes housing disjoint data, thereby increasing the bandwidth received by a nodes.

The ns [23] network simulator provides a simulation system to evaluate network protocols. Specifically, most overlay networks have used ns as the de-facto methodology for accurate simulation. Unfortunately, packet-level, congestion-aware simulation is costly, leading to inadequate scaling properties when evaluating overlays over a few hundred in size. However, for small-scale scenarios, ns provides an efficient and inexpensive mechanism for system evaluation. In the end, many overlay network evaluations have created their own simulators that sacrifice accuracy for the sake of running for larger scale scenarios. 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 that employed by ModelNet [22]. ModelNet enables the large-scale emulation of native IP applications by subjecting IP packets to bandwidth and latency restrictions as specified by a network topology. It emulates routers' queuing delay and congestion. In our experiences, thousands of overlay participants can run on 20-50 commodity PCs in the ModelNet environment. The same code runs unmodified in production Internet environments and testbed, including PlanetLab [13]. As a result, ModelNet's accuracy, scalability, and ability to emulate traffic for live IP applications makes it an appropriate choice for large-scale evaluation.

3. SYSTEM MODEL

The main challenge of MACEDON is to provide an overlay algorithm representation powerful enough to characterize the intricacies of different protocols, yet simple enough to facilitate implementation. To this end, we attempt to identify the common characteristics of overlay algorithms. In this section, we show how we use these unifying abstractions to design the MACEDON framework capable of concisely representing a wide array of network protocols.

3.1 Algorithm Characteristics

The first step in designing the MACEDON framework is abstracting common characteristics of overlay algorithms. This enables us to better understand the differences between algorithms and to provide infrastructural support to implement these differences. Further, the identification of common overlay functionality leads to the MACEDON libraries that implement them. These libraries promote code reuse and encourage fair and consistent overlay evaluation. In this manner, improvements within libraries of basic overlay functions benefit all protocols.

3.1.1 *Application Programming Interfaces*

Since overlay algorithms are typically created for the use of specific types of applications, an important characteristic of overlay implementations is the APIs they export. For example, the implementation of an algorithm designed for application level multicast must export a sending function to disseminate the data through the overlay. While sometimes obfuscated in the design of overlay algorithms, we believe it imperative for algorithm implementations to provide appropriate APIs to application developers. A number of recent efforts [6, 19] have made initial steps at creating a single all-encompassing API. In this area, MACEDON promotes a similar API to that described in [6] and enables the extensibility of the API for protocol-specific functionality. Section 4 provides additional detail about the MACEDON API.

3.1.2 *Node state*

Because our target overlays are distributed algorithms, each node maintains a certain amount of state that identifies the node's position in the overlay and the node's current activities. This determines each node's relationship to its current peers, or *neighbors* in the overlay. For example, a tree-based overlay will have parent and children neighbors. The behavior of the node toward a peer may be different depending on which type of node the peer is. It may also maintain a list of *potential* neighbors to which it may choose to establish peering relationships in the future. This functionality is not required in certain overlays, for instance when nodes acquire knowledge of potential peers and either establish the relationship or delete any knowledge of the peer. In other overlays, such as RON [1], 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. An example would be the bandwidth estimates to neighbor nodes used by Overcast [9].

3.1.3 Overlay transformations

Additional node state may be gathered as needed to perform some overlay functions. Under certain circumstances, overlay nodes evaluate their position in the overlay and make decisions based on what they perceive would form a better overlay. During such a process, it may obtain some short-lived state to make its decision. For example, Chord nodes periodically route requests through the overlay to destination addresses of their routing (finger) entries. Replies are gathered to help the nodes determine whether their routing entries are out of date, and if so, they update them accordingly. By doing so, the node is changing the list of nodes it peers with, performing a *transformation* on the overlay. Nodes that initiate transformations are the *coordinators* of the transformation. They may also need to perform some localized locking of affected peers for the transformation. These nodes are subsequently unlocked at the conclusion of the transformation.

3.1.4 Communication semantics and timers

Another common characteristic shared by overlay algorithms is that of message-based communication. Nodes communicate with one another via messages indicating, for example, that a node needs to be locked or that a node's state should change. As it pertains to TCP/IP, overlay implementations could be thought of as an additional layer between the transport and application layers, the overlay layer. Hence, the overlay protocol utilizes lower layer TCP/IP functionality to deliver messages that coordinate and synchronize the efforts of nodes in the overlay. Depending on the semantics of the overlay function, communication could make use of native IP multicast or IP unicast communication. The communication could be reliable, congestion-friendly (using TCP), unreliable, congestion-friendly (using TFRC [12]), unreliable, congestion-unfriendly (using UDP), or reliable, congestion-unfriendly (using a simple sliding window protocol, SWP).

Overlay algorithms also employ periodic timers to aid in coordination of overlay functions and transformation. Timers are usually used by overlays to periodically execute overlay maintenance or check overlay invariants. For example, NICE [2] nodes periodically check their cluster groups to determine if size bounds are violated. If a cluster is too large, the node will initiate a transformation to split the cluster. Likewise, if a cluster is too small, the node will initiate a transformation to merge the cluster with another cluster.

3.1.5 Edge comparisons

While an overlay node may use protocol-specific information to determine whether a transformation in an overlay should be made (as is the case in Chord routing table maintenance), other overlays compare potential overlay edges along some specific performance metric. This could be as simple as checking the amount of time it takes to receive a message of a certain size or attempting to estimate the round trip time associated with the edge. An estimate of an edge's performance is used to determine the most attractive edge. This is typically done in overlays with the use of *probes*. Probes can be used to estimate end-to-end transmission latency, as in NICE [2], or available bandwidth, as in Overcast [9]. Our overlay abstraction allows for the specification of these types of probes. However, we note that efficient probing is the subject of orthogonal work. Future work will attempt to incorporate a standard probing mechanism across all overlays, perhaps by employing some standard tool such as *sprobe* to facilitate the process. Determining *which* nodes to probe, however, is left to the designer of the overlay algorithm and is algorithm specific. Some overlays [1, 7, 8] maintain global membership knowledge, while others infer the appropriate nodes to probe from messages received from their peers [10].

3.1.6 Joining an Overlay

One interesting problem prevalent in current overlay networks is the mechanisms by which nodes *join* the system. A node wishing to participate in the overlay will contact some node already in the system for aid in entering. The problem lies in finding the appropriate bootstrap node. Overlays generally employ out-of-band functionality to execute a join, such as the use of DNS name resolution. Usually, the goal is to direct the entering node to a close-by participant of the overlay. One interesting technique, expanding ring join, leverages native IP multicast with increasing scope. A joining node, J, sends a scoped multicast join request that is seen only by close nodes such as A and B. If A and B are unable or unwilling to act as J's agent in entering the overlay, J retransmits its join request with increased scope so that M and N also see it. Thus J can continuously increase the scope of its request until a node responds. By identifying such a common aspect of overlay algorithms, new technology such as *anycast* can be seamlessly integrated into the joining procedures of overlay algorithms implemented with MACEDON.

In a simpler scenario, MACEDON overlays may depend on an out-of-band technique employed by the application to find the appropriate bootstrap node. In the case of application level multicast, a rendezvous point (such as the source of the data) may serve as the bootstrap. Since this issue is still an open research problem, it is, like probing, a problem orthogonal to MACEDON. In fact, we view MACEDON as the appropriate infrastructure for carrying out research in techniques for efficient overlay join operations in addition to efficient probing techniques.

3.1.7 Protocol layering

One central idea to implementing algorithms in MACE-

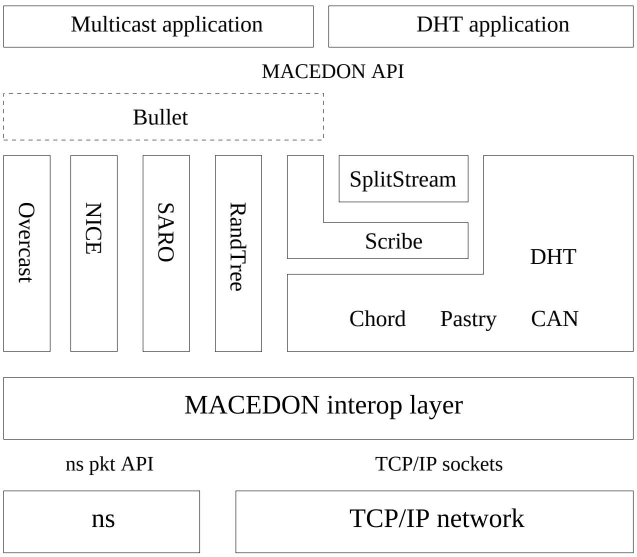

Figure 1: The MACEDON protocol stack

DON is layering, where overlay algorithms are layered on top of one another as illustrated in figure 1. This view of the entire MACEDON protocol stack is divided into three components: the application, the multiple layers of MACEDON protocols, and the network substrate (ns or TCP/IP). Much like the TCP/IP protocol stack, lower layers in the MACEDON stack provide services to higher layers. Bullet, for example, can be made to function over a simple randomly constructed tree (termed RandTree).

3.2 Representing Overlays in MACEDON

Having identified common characteristics of a broad class of overlays, we make the case that overlay network algorithms react to input as in a traditional finite state machine (FSM) where nodes move from one state to another based on the input. These *transition stimulants* include the reception of messages and the completion of timers. As a result, if we can describe this input and the resulting operations that occur as a result of the input, we can, provide a complete abstraction of overlay network algorithms. These operations include the setting of local node state and the transmission of messages. In essence, MACEDON provides a mechanism to create an event-driven overlay in which the inputs that drive the events as well as the actions occurring as a result of them are succinctly specified in an overlay-specific language. The following subsections describe the components of this specification.

3.2.1 Node state

States in an algorithm's FSM represent high-level phases of processing. For example, upon initialization, a node in the overlay may probably enter a *joining* phase where a join request is transmitted to a node already in the overlay. Many algorithms contain a probing state where nodes probe a certain population of overlay participants, for instance to reduce latency. Auxiliary state describes additional fields that nodes maintain. Auxiliary state would include, for example, the list of neighbors that a node is peering with or fields such as grid coordinates in CAN [15], a routing table in Pastry [19], or root delay in SARO [10].

3.2.2 Transition Stimulants

Transition stimulants provide the stimulus to move from one phase of operation (FSM state) to another. They allow the execution of the algorithm to progress. Transition stimulants include:

- *Reception of messages:* A node receives a message and processes it. It determines what if anything should be done in response to such a message. For example, a node receiving a join request would attempt to add the joining node as its neighbor in the overlay.

- *Timer expirations:* When a scheduled timer expires, the node may perform any functions corresponding to this timer. For example, a NICE [2] node schedules timers to check invariants of the protocol. If the cluster it belongs is unsuitably large or small, it will initiate a split or merge transformation. This would cause the node to change state and to perform a number of actions as specified in the action list.

- *API function calls:* An application issues a data path call. The overlay, upon receiving the call, may change its state, perform some actions, or make transformations in the overlay. While data path calls typically do not lead to state changes, the MACEDON framework does not prohibit this.

Because transition stimulants may occur simultaneously, it is important that MACEDON allow multithreaded execution, particularly as it corresponds to the data path. That is, while control threads may and sometimes, should, be serialized (e.g. for correctness), data threads should execute in parallel to increase performance when working threads block or if a multi-processor is available. Hence, our guiding principle is that stimulants be identified as *control* or *data* path related. MACEDON can hence use read/write locks to serialize access to the algorithm's state.

3.2.3 Actions

An action describes the processing that an agent performs as a result of a transition stimulant. This includes the following:

- Changing node state. Using MACEDON state management, an overlay algorithm can specify which fields in its state change upon message reception. For instance, this may be used by a particular algorithm to update its neighbor list or to change its state.

- Transmitting a message. For example, a node coordinating a local change with its neighbors would send messages to all of these neighbors that they must now change their state. MACEDON locks can be used by the coordinator to ensure that the operation is atomic, though many overlay algorithms do not make use of local locks.

- Scheduling a timer. As an example, a Chord node schedules a timer upon joining the Chord ring to periodically check its finger table, correcting any incorrect entries.

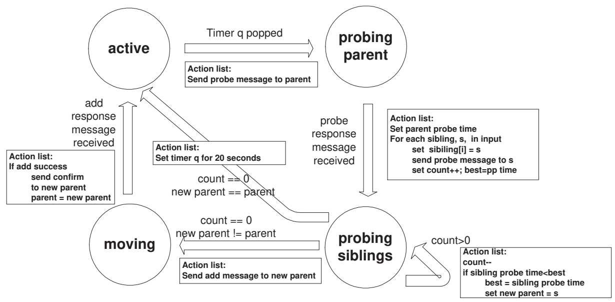

Figure 2: Portion of the Overcast algorithm representation

Figure 2 illustrates a simplified portion of the algorithm representation for Overcast [9]. It shows the FSM states corresponding to an Overcast's node probing of its parent and siblings. Initially in active state, the Overcast node transitions into the probing parent state upon the expiration of the invariant timer. The node first probes its parent to determine current networking conditions to it and probes its siblings, whose addresses were returned by the parent in the probe response. Upon receiving this probe response from its parent, the node stores the sibling addresses and a count of them in its local state, sends probes to all of its siblings, and initializes the local field indicating the best probe time seen thus far. It receives probe replies from children, tracking the best one seen thus far. Once all sibling probe responses have been received, the Overcast algorithm determines whether an overlay transformation is required. In this case, an add request is sent to the potential parent. This request could be accepted or rejected, and as such, the node's auxiliary state (parent neighbor list) is updated. In either case, the node returns once again to its active state, scheduling a timer to repeat this procedure some time in the future. A similar representation can be created for all overlay functionality, thus leading to the MACEDON language specification as described in Section 5.

4. THE MACEDON API

Promoting a standard API enables MACEDON applications to select the algorithms to use without the need to modification. In general terms, overlay networks enable the communication of one or more nodes with one another. They support either a multicast primitive or a route primitive, which routes data from a destination to the source *through* the overlay. Typically, overlays provide upcalls at each *routing hop* that enable intermediate nodes to perform application-specific functions. For example, an intermediate Scribe node receiving a join request for a particular group will add that group to its list of multicast sessions and then propagate the request toward the destination, thus building a multicast distribution tree.

Figure 3 illustrates a simplified version of the API that MACEDON-generated overlays export. MACEDON provides an extensible upcall and downcall mechanism which overlay developers can use to effect protocol-specific between layers in the MACEDON stack. As instances of this mechanism, we illustrate the prototype of the `forward`, `deliver`, and `notify` upcalls (extensible upcalls are handled using the generic upcall handler). `forward` is called by the overlay

```
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(int bootstrap_node, int protocol);
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 transport);
int macedon_multicast(macedon_key groupID,
                      char *msg, int size,
                      int transport);
int macedon_anycast(macedon_key groupID,
                    char *msg, int size,
                    int transport);
int macedon_routeIP(int dest, char *msg,
                    int size, int transport);
```

Figure 3: Simplified MACEDON API

once a routing decision has been made. This allows intermediate nodes to potentially change the message, change its destination, or quash the message altogether. The `notify` upcall allows lower-layered protocols to inform higher layers of changes in neighbor sets. An application registers its upcall handlers with the `macedon_register_handlers` function. Note that registering handlers is optional. However, at least one handler is necessary if the application is to receive any data through the overlay (having null handlers would be used, for example, when evaluating just the construction process of different overlays). Figure 3 also shows the prototype for the `macedon_init` function that is called by the application when it wishes to use an overlay algorithm. It specifies a well-known protocol value for the overlay (this is akin to protocol values in IP). Once an application has initialized and registered its handlers, it is able to send and receive multicast or unicast data. To enable unicast data, an overlay must implement *routing* functionality as is common in DHT overlays such as Chord, Tapestry, Pastry, and CAN. The `macedon_route` function accepts a destination and a message. In this case, the destination is a macedon_key, meaning it is not necessarily an IP address. Typically, this key will be a hash of a network or IP address and is of configurable length (32-, 128-, and 160-bit values are common). Another unicast primitive is the `macedon_routeIP` function that enables native IP-based communication with an IP host. These data packets will not flow through the overlay.

Multicast data primitives include a function to create a multicast session, `macedon_create_group`. Its sole input is the value (name) to be associated with the session (group). This provides the application with a handle for the specific session. Receivers of this session join and leave it with the `macedon_join` and `macedon_leave` functions, specifying the corresponding group's ID value. Finally, nodes originate multicast data with `macedon_multicast`. Similar to the `macedon_route` function, the multicast function only differs in that it requires a session's ID instead of a node's destination address. A related function, `macedon_collect` introduces a new primitive to traditional overlay APIs. It essentially performs the opposite function of the multicast method, where data originates at non-root nodes and is collected via the distribution tree up toward the root. This enables intermediate nodes to summarize data in an application-specific manner at each step in the tree, ultimately delivering a global summary to the tree's root. We believe that a number of applications could benefit from this communication paradigm and we are investigating this model as part of future work.

5. LANGUAGE SUPPORT

This section provides both the grammar for the MACEDON language and a detailed description of the MACEDON implementation of Scribe, an application-layer multicast protocol.

5.1 MACEDON Grammar

The grammar for implementing a protocol using MACEDON is described in Figure 4. The MACEDON grammar allows one to define a *PROTOCOL SPECIFICATION*, which the MACEDON engine translates into working code as described in Section 6. A *PROTOCOL SPECIFICATION* includes four main sections, described in detail below:

5.1.1 HEADERS

There are 3 main headers. The protocol header specifies the name of the protocol and optionally a base protocol for layering. For example, while DHTs are commonly built directly on IP, overlay multicast trees may be built on top of a lower layer. Thus, one could specify "protocol scribe uses pastry" to run SCRIBE on top of Pastry, or alternately "protocol scribe uses chord" to change the underlying DHT. Assuming both DHTs export the same API, one could also specify "protocol scribe uses chord" to 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 the four increasing levels of tracing messages output.

5.1.2 STATE AND DATA

The *STATE AND DATA* section defines the state and data used by the protocol. This includes defining constants, node types, states, neighbor types, optionally transports, messages, and other auxiliary data. *NODE TYPES* specifies the different types of nodes, for example differentiating between sender and receiver, or for nodes at different levels of hierarchy. *STATES* defines the allowed set of protocol states, for instance "init", "joining", or "joined." These states will be used later on in the *TRANSITIONS* section. *NEIGHBOR TYPES* specifies the sets of neighbors the protocol tracks (and their maximum size), such as a set of nodes with proximate node IDs, a set of nodes close in latency, or a routing table. These *NEIGHBOR TYPES* may further specify optional *NEIGHBOR FIELDS* to track on a per-neighbor basis. Note that a *NEIGHBOR FIELD* might itself be a set of neighbors. Optionally, the protocol may next specify that different types of messages it sends should be transmitted using different transport protocols. Specifically, *TRANSPORTS* and *MESSAGE* definitions inform

```
<PROTOCOL SPECIFICATION>:
  <HEADERS> <STATE AND DATA>
  <TRANSITIONS> <ROUTINES>(0,1)

<HEADERS>:
  "protocol" <protocol name>
    ["uses" <base protocol>](0,1)
  "addressing " ["hash"|"ip"]
  "trace_" ["off"|"low"|"med"|"high"]

<STATE AND DATA>: <CONSTANTS> <NODE TYPES>
  <STATES> <NEIGHBOR TYPES> <TRANSPORTS>(0,1)
  <MESSAGES> <AUXILIARY STATE>

<CONSTANTS>: "constants {"
  [<code type> <const name> "=" <value> ";"]* "}"
<NODE TYPES>:
  "node_types {" [<node type name> ";"]* "}"
<STATES>: "states { init;" [<state name> ";"]* "}"

<NEIGHBOR TYPES>:
  "neighbor_types {" <NBR TYPE>* "}"
<NBR TYPE>: <neighbor type name> <size> [";" |
  "{" <NBR FIELDS>+ "}"]
<NBR FIELDS>: <TYPE> <neighbor field name> ";"

<TRANSPORTS>: "transports {" <TRANSPORT>+ "}"
<TRANSPORT>:
  ["TCP"|"UDP"|"TFRC"|"SWP"] <transport name> ";"

<MESSAGES>: "messages {" <MESSAGE>* "}"
<MESSAGE>: <transport name>(0,1) <message name>
  "{" <MESSAGE FIELDS>* "}"
<MESSAGE FIELDS>:
  <TYPE> <message field name> <size>(0,1) ";"

<AUXILIARY STATE>: "auxiliary data {"
  [<LOCAL VARIABLE>|<TIMER VARIABLE>]* "}"
<LOCAL VARIABLE>: <code type> <variable name>
  <size>(0,1) <size>(0,1) ";"
<NEIGHBOR VARIABLE>:
  "fail_detect"(0,1) <neighbor type name>
  <variable name> <size>(0,1) <size>(0,1) ";"
<TIMER VARIABLE>:
  "timer" <timer name> <period>(0,1) ";"

<TRANSITIONS>:
  "transitions {" [<INIT>|[<API TRANSITION>|
    <TIMER TRANSITION>|<MESSAGE TRANSITION>]* "}"
<INIT>: "init join {" <code> "}"
<API TRANSITION>: "API " <API TYPE> "{" <code> "}"
<TIMER TRANSITION>: <STATE EXPRESSION>
  "timer" <timer name> "{" <code> "}"
<MESSAGE TRANSITION>: <STATE EXPRESSION>
  ["forward"|"recv"] <message> "{" <code> "}"
<STATE EXPRESSION>: "any" | !(0,1)<state name>

<ROUTINES>: "routines {" <coded routines> "}"

<API TYPE>: "route" | "routeIP" | "multicast" |
  "anycast" | "collect" | "error" |
  "create_group" | "join" | "leave" | "notify" |
  "downcall_ext" | "upcall_ext"
<TYPE>: <neighbor type name> | <code type>
```

Figure 4: MACEDON Grammar

MACEDON that the particular messages should be sent using a particular transport. MACEDON currently supports TCP, UDP, TFRC, or SWP (a simple reliable sliding window protocol). Messages may contain any number of fields, which may include standard language types, neighbor sets, or arrays of neighbor sets. Finally, the protocol specifies *AUXILIARY STATE* for any auxiliary state or variables it should maintain. This auxiliary state is the only application-specific state maintained from transition to transition, and so should be used for any extra state the protocol needs to remember. In addition to standard language types, neighbor sets, and multidimensional arrays of such types, auxiliary state can specify timers, which may have a period in seconds they will fire once scheduled. The auxiliary state section may also specify that a neighbor data element is "fail_detect", which tells MACEDON to generate code to monitor that neighbor for its failure, and to perform a callback to the protocol upon failure.

5.1.3 TRANSITIONS

TRANSITIONS are the entry points into the protocol and are responsible for the bulk of the work in the protocol. For each transition type, the programmer will use a familiar language (C++ in our current prototype), along with a set of MACEDON primitives to interface with agents, timers, and libraries. There are four types of transitions: *INIT*, *API TRANSITION*, the *TIMER TRANSITION*, and a *MESSAGE TRANSITION*. At the beginning of execution, the protocol is initialized by calling the special *INIT* transition, to initialize state and to schedule any necessary future events. *API Transition* enables 12 types of calls. These represent entry points into the protocol when either the upper or lower layer is asking the protocol to take some action. "route", "routeIP", "multicast", "anycast", and "collect" all represent requests to send data. "create_group", "join", and "leave" are common abstractions for managing node or session state. The remaining API calls represent atypical or extensible calls into the code, including notifying upper layers of a changed neighbor set, the detection of a *fail_detect* node's failure ("error"), or more generic "upcall_ext" and "downcall_ext" to provide extensibility for those protocols which need a richer API than the base.

A *TIMER TRANSITION* represents an entry into the protocol where the protocol itself requests to be woken up at after a certain delay. This is commonly used to perform periodic actions such as stabilization, or non-periodic actions such as recovery from a timed-out join. The *TIMER TRANSITION* is further enriched by specifying the states where timer expiration should be handled, or by providing different transitions for timers depending on the state of the protocol. For example, if a node has a state called "joined," a timer called "recovery," and should periodically call a recovery routine whenever it is not in the "joined" state, it can schedule a periodic timer and provide a transition such as "!joined timer recovery { do_recovery(); }". At the same time, perhaps the protocol designer wishes to know when the timer fires and the node is joined, as a sanity check a transition "joined timer recovery { dump_state(); }" could be added. Finally, the *MESSAGE TRANSITION* variants take place based on three factors: i) the message type, ii) the current protocol state, and iii) whether the node is being asked to forward the message or receive it (in some cases both will happen).

5.1.4 ROUTINES

ROUTINES are similar to transitions, except that while transitions are entry points into the code from outside the protocol, routines are called from the transitions. As such, the protocol designer may optionally create routines for convenience and modularity.

In addition to the grammar, MACEDON provides a number of useful primitives to simplify interaction with agents, timers, and libraries. For example, to send data in the code portions, the primitive is: *⟨API type⟩_⟨message name⟩(⟨destination⟩, ⟨message fields ...⟩)*. Sending additional application data can be sent using *⟨API type⟩withdata_⟨message name⟩(⟨destination⟩, msg, size, ⟨message fields ...⟩)*. Timers can be accessed by using primitives like *timer_resched(⟨timer name⟩, ⟨wait time⟩)* and *timer_cancel(⟨timer name⟩)*. Other API calls may be made (API notify, join, leave, create_group, upcall_ext, and downcall_ext) by a call such as ⟨API type⟩(⟨parameters ...⟩). Library calls are available to access things such as bloom filters and neighbor sets. Neighbor sets can be iterated over using *neighbor_foreach*, added to using *neighbor_add*, removed from using *neighbor_remove*, etc. When a message transition is started, the fields of the message may be accessed simply as *field(⟨message field name⟩)*.

5.2 An Example Specification: Scribe

Figure 5 presents a subset of our Scribe implementation, an application-layer multicast protocol, as a concrete example of a MACEDON-based implementation. While many overlay algorithms are standalone, Scribe is layered on top of a DHT overlay, such as Pastry or Chord. This example shows Scribe over Pastry, as specified by line 1, part of the *HEADERS*. It would however, be trivial to layer Scribe on top of Chord instead, by commenting out line 1, and uncommenting line 2. Similarly other DHTs implementing the same API could be swapped out trivially.

In this example, Scribe nodes use hash addressing (line 2), and in lines 4-5, defines the *STATES* "init" and "joined.". "init" is a required state which all protocols start in, and in this case "joined" is the state indicating the local Scribe node is ready to participate in a multicast tree. Other overlay algorithms have more possible states that typically include a joining phase or a transformation phase. Overlays also specify neighbor types that represent peer relationships. For example, an overlay tree would have parent and children neighbor types. Accordingly, Scribe defines its neighbor types (lines 6-14), namely children (line 6, each set containing up to MAX_CHILDREN), session (lines 7-13, containing up to MAX_SESSIONS), and visited (line 14, containing up to MAX_NODES). MAX_CHILDREN, MAX_SESSIONS, and MAX_NODES are defined in the *CONSTANTS* block, not shown. The "visited" neighbor type is used in conjunction with anycast messages to prevent visiting a node twice, and is passed around in the anycast message (see lines 19-23). While neither the "visited" nor "children" neighbor types contain additional fields, note that "session" does. The "session" neighbor type, representing state for a Scribe tree, contains a neighbor set of children (line 8), and three integer variables (lines 9-11) used in conjunction with the heartbeat timer to maintain active relationships in the Scribe tree. Finally, an integer (line 12) records whether the local node is subscribed to the Scribe tree, and a boolean records whether the local node is the

```
1: protocol scribe uses pastry
2: //protocol scribe uses chord
3: addressing hash
4: states {  init;
5:          joined; }
6: neighbor_types {   children MAX_CHILDREN;
7:           session MAX_SESSIONS {
8:                            children kids;
9:                247 int ticks_since_heartbeat;
10:              int tick_count_without_data;
11:                 int tick_count_for_renew;
12:                           int subscribed;
13:                              bool isRoot; }
14:           visited MAX_NODES;                 }
15: messages {  REL_NORMAL create_group {
16:                               int group_id; }
17:             UNREL_NORMAL data {
18:                               int group_id; }
19:             REL_NORMAL anycast {
20:                               int group_id;
21:                              int from_addr;
22:                       visited nodesVisited;
23:                          children siblings; }
24:             ..... }
25: auxiliary_state {               int myhash;
26:                            session my_sess;
27:                    timer hbTimer TIMER_INT; }
28: transitions {
29:  init join {           myhash = hashof(me);
30:           timer_resched(hbTimer,TIMER_INT);
31:                       state_change(joined); }
32:  joined recv create_group {
33:    int gr = field(group_id);
34:    if(!neighbor_query(my_sess, gr)) {
35:     neighbor_add( my_sess, gr );
36:     neighbor_notify(my_sess, NBR_TYPE_TREE);
37:     neighbor_info(my_sess, gr, subscribed)=0;
38:    }
39:    neighbor_session *temp =
40:                 neighbor_entry(my_sess, gr);
41:    temp->isRoot=true;
42:    temp->ticks_since_heartbeat = 0;
43:    temp->tick_count_for_renew = 0;
44:    temp->tick_count_without_data = 0;
45:  }
46:  joined recv data {
47:    int gr = field(group_id);
48:    neighbor_session *s;
49:    if (neighbor_query(my_sess, gr)) {
50:      if(upcall_forward(gr, msg, size,
51:              COMM_TYPE_MULTICAST) == 0) {
52:        s = neighbor_entry(my_sess, gr);
53:        s->tick_count_without_data = 0;
54:        s->ticks_since_heartbeat = 0;
55:        if(s->subscribed)
56:          upcall_deliver(msg, size,
57:                        COMM_TYPE_MULTICAST);
58:        foreach_neighbor (neighbor_children*,
59:                         a, s->kids)
60:        routeIPwithdata_data( a->ipaddr, msg,
61:                             size, gr );
62:  } } }
63:  API create_group {
64:    route_create_group( groupID, groupID );
65:  }
66:  joined timer hbTimer { ..... }
67:  .....
68: }
```

Figure 5: Portions of the Scribe specification

root of the tree (line 13).

The *MESSAGES* block is defined next, in lines 15-24. Messages are described in the MACEDON language and include a list of fields specified within each message type. Message declarations allow MACEDON to marshall fields automatically on behalf of the overlay. Scribe defines three messages: “create_group”, “data”, and “anycast”. The first two only contain the group ID as a field, while “anycast” (lines 19-23) additionally contains, the “from_addr” which represents the original sender, a list of visited nodes, and a list of siblings to the recipient. Another artifact of these messages is that they are specified to receive different classes of service. In this case they are using either the “REL_NORMAL” transport (lines 15 and 19) or the “UNREL_NORMAL” transport (line 17). This allows Scribe to inform the lower layer (Pastry) of the type of transport to use when sending the data (reliable vs. unreliable, normal vs. expedited). This transport specification is optional – without it the default transport for the lower layer will be used. It is not necessary to explicitly declare data fields since MACEDON allows application data to be appended to any overlay message, and specifically to Scribe’s “data” and “anycast” messages.

This session list is declared in the auxiliary state portion of the specification (lines 25-27). Auxiliary state describes fields needed by nodes to function properly, beyond the simple FSM state. This includes neighbor lists, timer declarations, and any other required node state. In our example, Scribe only needs to maintain its session list (line 26), the hash ID of the node (line 25), and a heartbeat timer (line 27) to recover from tree parents which appear not to be forwarding this node data. The heartbeat timer is declared with a firing period of TIMER_INT, a constant (not shown), but it will not start to fire until after it has been scheduled for the first time. This is done next in the “init join” transition (line 30). If TIMER_INT were not specified, when scheduled the timer would fire only once each time it is scheduled.

The most important portion of a MACEDON algorithm specification is the transitions that describe the actions that must be performed in moving from one state to another (or perhaps the same) state. Actions include setting the FSM state or some auxiliary state, sending a message, or scheduling a timer, and more than one of these can be performed during a transition. Auxiliary state includes neighbor lists and any other fields needed to coordinate overlay transformations. The transitions identify the stimulus of each transition and the state where such a transition occurs. In our Scribe example, we show the transitions occurring when a node joins (lines 29-31) and receives a message of type create_group (lines 32-45) or of type data (lines 46-62). We also show a transition that occurs when the application issues a command to create a multicast group (lines 63-65), and the declaration of the heartbeat transition (line 66).

The special case transition, “init join”, is always executed to initialize the node. In many overlays, a node will contact a bootstrapping node to join the overlay, typically the source of multicast data or a rendezvous point found by some out of band means such as a DNS lookup. In Scribe, a node simply schedules the heartbeat timer (line 30), changes state to joined (line 31) and exits.

Data receipt messages are identified by the keywords *recv* and *forward*. In this example, when a “create_group” message is received in the “joined” state, the specified transition will execute. Then, if the check in line 34 returns true, then the group does not already exist as a session neighbor to this node. Thus, it is added (line 35), the upper layer is notified (line 36), and the subscribed field is initialized to 0, since the local node has not called "join" on that group ID. Then a bit more state is initialized in lines 41-44, whether or not it was a new group, recording that this node is the root of the tree and reseting heartbeat counters to 0. The "data" message execution has a similar declaration in line 46. It executes by first ensuring that this node has state for this session (via the neighbor_query in line 49, and exits if false). Then the node invokes the upper layer's forward handler (lines 50-51), giving the upper layer a chance to terminate the message before delivering it locally (line 56), and routing it directly to each of Scribe's children for this session. The foreach_neighbor in line 58 iterates over the list of children, and the routeIPwithdata_data in line 60 invokes the lower layer's routeIP API transition, passing it both the application data and a data message with group_id set to gr. This transition also resets some counters associated with the heartbeat timer.

Finally, our brief Scribe excerpt defines the create_group API call in lines 63-66 and declares the "hbTimer" transition (line 66). The "API create_group" in turn causes the lower layer to route a create_group message to the node managing that groupID. The second groupID in line 64 is the data to be placed in the groupID field of the message. The declaration of "hbTimer" indicates that if the timer "hbTimer" fires while the node is in the "joined" state, this code should be executed. It is not shown due to its length, but manages the counters shown above, and takes corrective action when necessary.

6. OPERATING ENVIRONMENT

MACEDON's ability to create high-performance implementations of specified algorithms depends on the operating infrastructure that implements common functionality. This section describes how we generate code that links with a number of libraries to produce implementations capable of achieving high data rates. Further, we discuss MACEDON's support for debugging and evaluation.

6.1 Code Generation

Once an algorithm is specified, MACEDON can generate code that implements the algorithm and codes to the MACEDON API. We term this generated code the MACEDON *agent*. While the details of code generation are straightforward, we present a high level overview. MACEDON parses the algorithm specification and translates it into executable C++ code that uses MACEDON library functions, timer and transport subsystems.

The translation phase involves the declaration of protocol messages, states, neighbor types, and auxiliary state, 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 demultiplexing function calls the appropriate transition function based on the node's current state and the message type. This code is linked with a MACEDON overlay engine that provides support for scheduling threads, timers, and transports. This code, as well as the MACEDON libraries, are common to all overlay implementations, increasing consistency in evaluation and promoting

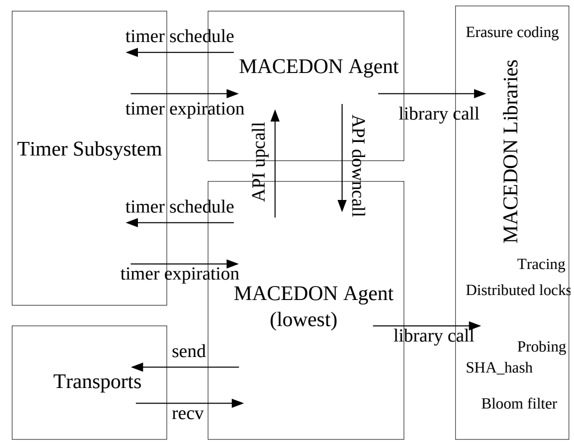

Figure 6: MACEDON Agents

code reuse. MACEDON translates API specifications and timer processing transitions in the same way as message processing transitions. Figure 6 outlines the implementation structure of the MACEDON agent. The lowest-layer agent is the only layer that interacts directly with the transport subsystem. Though this example only shows two layered agents, MACEDON supports arbitrary layering.

6.2 Thread Management

To alleviate concurrency issues when multiples messages are received or timers are triggered, MACEDON, by default, provides serialization of control paths. MACEDON protocol instances are serialized using a lock that supports read and write semantics. In this manner, operations that change the state of the overlay must lock the instance for writing, while operations that simply read it (such as when sending data) lock the object for reading. By default, a MACEDON overlay is locked for writing when a control message is received, a non-data path API call is made, and a timer expires. Functions on the data path (received data messages and data API calls) simply acquire the read lock allowing the protocols state to be examined. The net effect is increased throughput as data events can be processed in parallel. The conditions under which locks are acquired may depend on the algorithm, in which case these defaults are overridden. In our experience, this default locking behavior is adequate for most of today's overlays.

6.3 Algorithm Evaluation

One of the important contributions of MACEDON is the creation of a fair and consistent evaluation evaluation framework. To this end, MACEDON enables interoperability with ns for small-scale algorithm evaluation. In addition, because MACEDON-generated code also functions unmodified in a native TCP/IP environment, it can leverage the large-scale network emulation offered by ModelNet [22] or live deployment across the Internet, including PlanetLab. This enables the large-scale evaluation of real application code under Internet-like topologies. In addition, MACEDON has the capability of automatically extracting vital topology information from both ns and ModelNet, allowing it to evaluate overlays against a wide array of metrics. Without such global information (as is the case with live Internet experiments), it is impossible to accurately gauge an overlay's performance under certain metrics. For example, MACEDON can extract routing tables from nodes in ns and ModelNet and use that in conjunction with link information (latency, bandwidth, etc) to report the expected performance of an overlay along metrics such as link stress, node stress, latency stretch, and relative delay penalty (RDP).

Currently, the disparate use of evaluation techniques has lead to the use of many different performance evaluation metrics in overlay algorithm comparison. While an overlay evaluation may be concerned with showing low link stress, it is unclear whether that metric is important to application requirements. A high-bandwidth backbone link could have a link stress on the order of a hundred for a chat-like application and perform very well for that application, while link stress of two over a modem link for a video distribution application would be unacceptable. As a result, choosing the right evaluation metric is a challenge in overlay evaluation today. MACEDON attempts to bridge this gap by providing an evaluation framework that can report a broad variety of popular evaluation metrics. In doing so, we believe that MACEDON will encourage evaluation across more performance metrics, in effect allowing the metrics themselves to be evaluated.

6.4 Debugging Support

In addition to this evaluation infrastructure, MACEDON has a built-in debugging and tracing facility. By default, an overlay's execution will automatically generate trace entries when an event occurs: a node's state changes, a message is transmitted or received, a timer expires, or an API function call is executed. An overlay developer may choose the types of messages of interest and, for example, have the contents of those message automatically traced as well. These traces prove invaluable when debugging the flow of transformation and other coordinated overlay actions. While not currently implemented, we believe that the MACEDON framework can be extended to aid even further in algorithm design and implementation, particularly in the area of race conditions.

7. EVALUATION

In this section, we evaluate MACEDON's ability to: i) facilitate overlay implementation and evaluation and ii) implement a broad range of algorithms with good performance and scalability characteristics. While it is not practical to prove that MACEDON will be able to meet the demands of arbitrary overlays, we use our successful implementation of a broad variety of modern overlays originally developed by other researchers to suggest that such a goal may be attained, particularly given MACEDON's extensibility.

One key contribution of this work is the implementation and validation of a broad range of network overlays in the MACEDON environment, including: NICE [2], SplitStream [3], Overcast [9], Bullet [11], Pastry [19], Scribe [18], and Chord [21]. Figure 7 presents the lines of code 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 550 lines of code (LOC) while its generated C++ code is over 2300. The MACEDON operating environment is around 3700 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 around 200 lines of code, primarily because SplitStream, being layered on top of Scribe and Pastry, exploits functionality already provided by those systems. It is also noteworthy that implementing SplitStream also required changes to our Scribe implementation, primarily since the description of SplitStream [3] 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 the standardized API allows us to switch underlying DHT layers easily. Our experiments show results for SplitStream running over (Scribe running over) both Pastry and Chord.

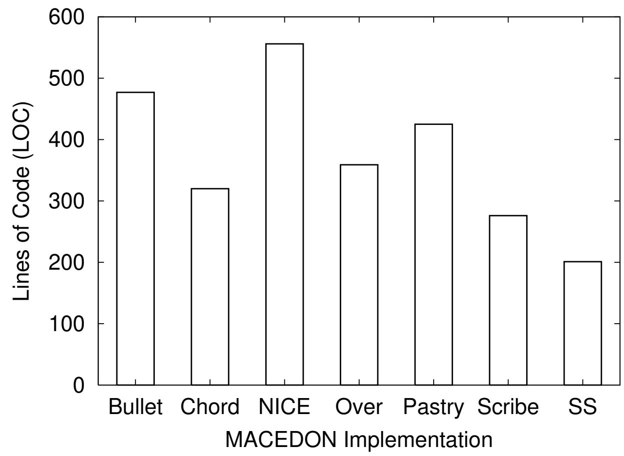

Figure 7: Lines of code used in various MACEDON protocol implementations.

The remainder of 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 implementation and FreePastry [17]). We believe that these results confirm the generality, accuracy, performance, and utility of our infrastructure. We use the ModelNet [22] infrastructure to emulate large-scale Internet topologies that faithfully capture hop-by-hop congestion and queuing behavior. For our NICE validation, we used the topology information provided in [2] to create an Internet-like topology that mimics that used in the author's original wide-area evaluation. Our evaluation infrastructure allows us to quickly extract features of the resulting overlay by extracting routing tables and topology information from ModelNet. For all other experiments, we use 20,000-node INET [4] topologies with varying numbers of clients. In all case, we run the overlay code on between 2–50 1.4Ghz Pentium-III machines running Linux 2.4.18. We multiplex multiple instances of our overlays on these machines. All traffic passes through our 1Ghz Pentium-III ModelNet core running FreeBSD-4.5. While all the results we show use ModelNet, we note that we have successfully run large-scale MACEDON-generated code over PlanetLab [13]. We omit these results for brevity.

7.1 NICE

To validate our implementation of NICE, we run the same experiments described in published work [2] for small-scale Internet scenarios and compare our results with the pub-

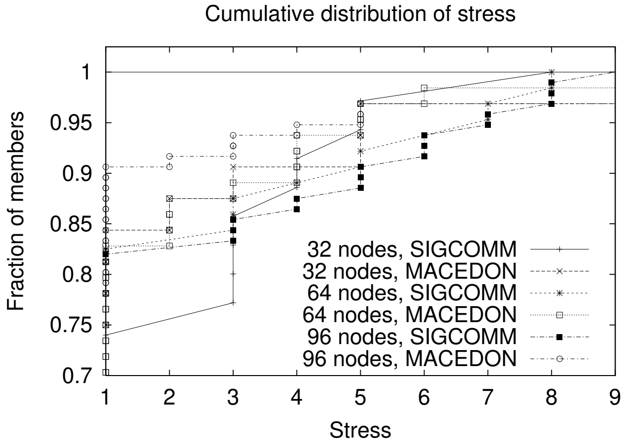

Figure 8: CDF of node stress for published and MACEDON NICE implementations.

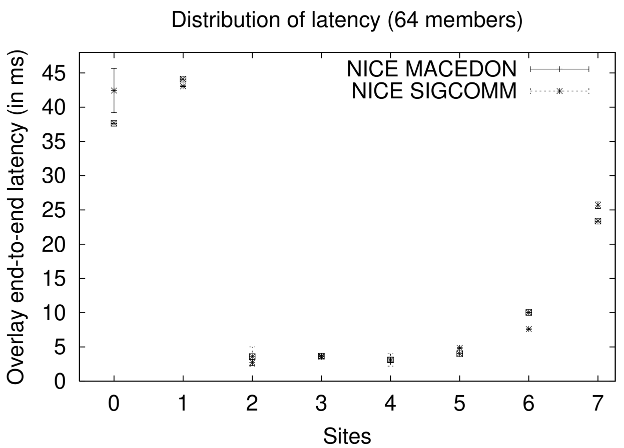

Figure 9: Observed latencies for published and MACEDON NICE implementations.

lished values. Compare figure 8 with figure 14 from [2]. Both plot a CDF of node stress (node fan-out) for 32, 64, and 96 nodes running across eight Internet sites. Our results closely resemble the published results with one minor exception. The root of our NICE tree has node stress of 16-18 in our runs. This is an artifact of a minor, yet correct difference (verified with the authors) in our implementation: we allow the rendezvous to be present in the highest layer and all lower layers. For simplicity, Banerjee et al. restricted the rendezvous point to only be present in the highest layer of the NICE overlay. Figures 9 and 10 show the average observed latency and stretch metrics for nodes in the different sites as reported in [2] versus our MACEDON implementation. 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 [2], though adding this to our implementation is straightforward.

7.2 Chord

We validated our MACEDON Chord implementation by

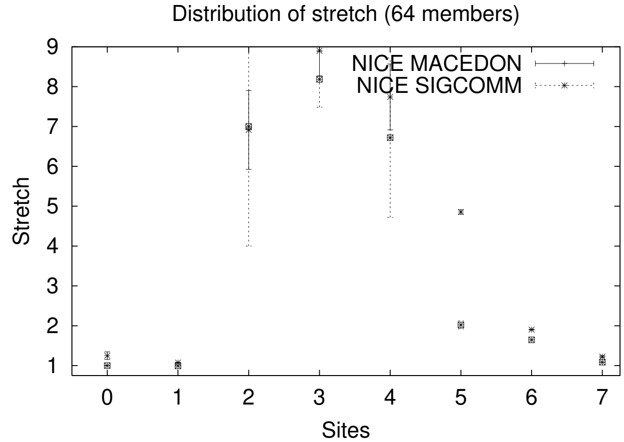

Figure 10: Observed stretch for published and MACEDON NICE implementations.

comparing results obtained with it to those of the MIT Chord distribution (lsd). We used an 20,000 node INET topology with 30 Chord participants for this experiment. We made two modifications to the MIT code to first dump the routing table of every node every two seconds (something which was 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. We calculated off-line the correct routing table for each node given all hash addresses of nodes joining the system. Figure 11 shows the convergence of these

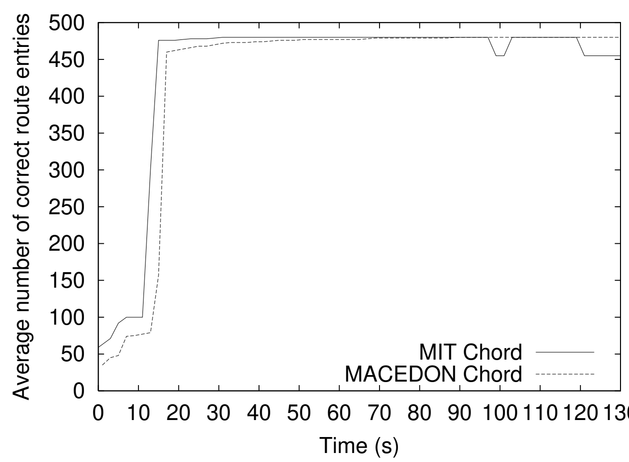

Figure 11: Convergence toward correct routing tables for MIT and MACEDON Chord implementations.

routing tables toward the correct values over time. For these results, we simply add the number of correct routing entries for each node at each two-second interval and average it for 15 randomly selected nodes of the 30 Chord participants. Because we are using a 32-bit hash address space, the total number of route entries is $15 * 32 = 480$. While having slightly slower convergence than the MIT version, our Chord implementation matches its behavior closely. One interesting anomaly that we observed with the MIT implementation is that some routes were periodically invalidated as can be seen at times 100 and 120. The MIT code makes use of RPCs to build and maintain routing tables. During these periods, we noticed that some RPCs had timed out, leading to the invalidation of these routes.

While we feel we have accurately implemented Chord and our implementation behaves much like the MIT version, we were unable to perform such a comparison for larger numbers of nodes. While the MACEDON implementation can scale to 1000 nodes, we were unable to run the MIT code beyond 50 nodes.

7.3 Pastry

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 used our application (C++ and Java versions) to estimate source to receiver delays for each packet transmitted both for the MACEDON Pastry implementation and FreePastry [17]. We varied the number of nodes participating in the overlay and selected them at random from our 20,000-node topology. For both systems, we allowed nodes to join and routing tables to converge by waiting 300 seconds before attempting to stream. Because our streaming rate was (purposely) low, intended targets received essentially all packets transmitted to them. For both systems, each node received approximately the same number of packets (within a small error) corresponding to the size of the portion of hash address space it owns. Figure 12 shows the average per-packet delays for the FreePastry and MACEDON implementations. Though MACEDON is able to scale to larger numbers of nodes, the FreePastry code was unable to run beyond 90 participants. Further, while MACEDON exhibits lower, slowly-growing delay in all cases, FreePastry appears to have performance problems when running with less than 40 nodes.

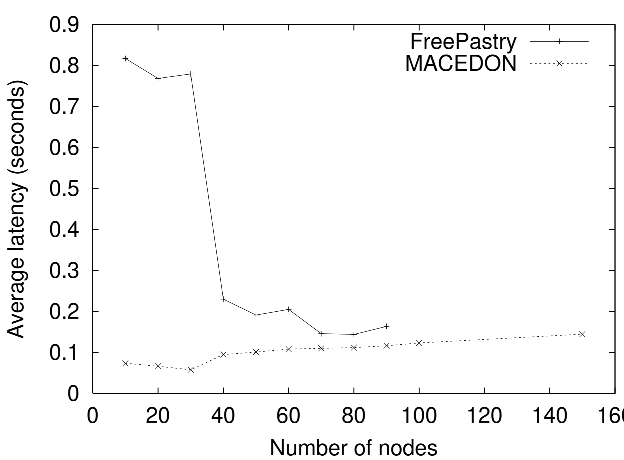

Figure 12: Average latency of received packets.

7.4 SplitStream

Because SplitStream is not supported in the current available version of FreePastry, we were unable to validate our implementation against any publicly available distribution. Instead, our SplitStream experiments are designed to test MACEDON's layering capability. For these tests, we created 200-node SplitStream forests using both Pastry and Chord as the DHT underlying Scribe. We developed a multicast application that streams 1000-byte packets at a predetermined rate (600Kbps for our experiments). In this case, only one node is designated as the source of the stream while all other nodes join the multicast session to receive data. We first allow routing tables to converge by idling the system over the first 300 seconds of each experiment

Figure 13 shows the resulting per-node average bandwidth over time for both SplitStream versions. While SplitStream

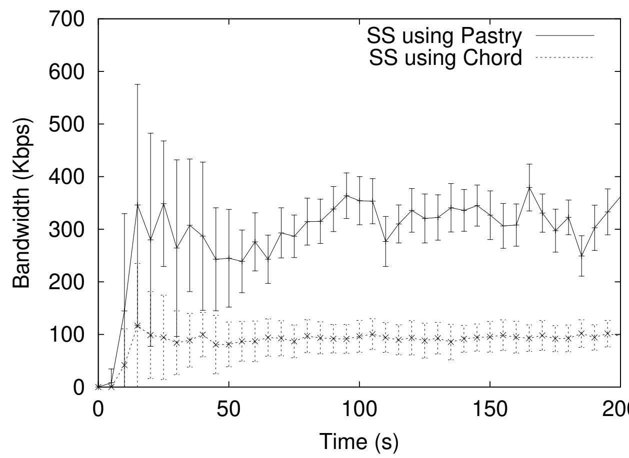

Figure 13: Achieved bandwidth over SplitStream.

running over Pastry delivers an average of 300 Kbps to each node, the Chord version achieves only 100 Kbps. We believe this difference in performance to be largely due to two main factors. First, SplitStream's is able to construct superior interior-node-disjoint trees with Pastry. In Chord, Scribe trees share more interior nodes, leading to more nodes being pushed down. Second, Pastry's locality properties allow it to choose nodes that have more attractive connectivity. Such options are not available with Chord. Clearly, a goal of this work is not to carry out a detailed comparison of Pastry versus Chord as a substrate for application-layer multicast. However, our experiments do demonstrate the value of MACEDON as an enabler for carrying out detailed and uniform protocol comparisons.

8. CONCLUSIONS AND FUTURE WORK

We have presented the MACEDON framework aimed at facilitating the design of overlay algorithms and their implementation. Our system provides a domain-specific language for specifying the high level behavior of overlay algorithms, including content distribution networks, distributed hash tables, and application-layer multicast. MACEDON provides a common evaluation infrastructure that enables fair and consistent overlay evaluation under a wide variety of network conditions. 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 Bullet, Overcast, NICE, Chord, Pastry, Scribe, and SplitStream. We believe that MACEDON can be used as an educational tool helping researchers understand the intricacies of overlay algorithms. Finally, we believe that the MACEDON vision extends beyond overlay algorithms to include a wider class of decentralized algorithms, though this is the subject of future work.

9. REFERENCES

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

[2] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In *Proceedings of ACM SIGCOMM 2002*, pages 165–175, 2002.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] Yang hua Chu, Sanjay Rao, and Hui Zhang. A Case For End System Multicast. In *Proceedings of the ACM Sigmetrics 2000 International Conference on Measurement and Modeling of Computer Systems*, June 2000.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] Jitedra Padhye, Victor Firoiu, Don Towsley, and Jim Krusoe. Modeling TCP Throughput: A Simple Model and its Empirical Validation. In *ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 303–314, Vancouver, CA, 1998.

[13] 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.

[14] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing Nearby Copies of Replicated Objects in a Distributed Environment. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1997.

[15] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Content Addressable Network. In *Proceedings of SIGCOMM 2001*, August 2001.

[16] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Application-level Multicast using Content-Addressable Networks. In *Third International Workshop on Networked Group Communication*, November 2001.

[17] Rice University. *FreePastry*, 2003. http://www.cs.-rice.edu/CS/Systems/Pastry/FreePastry/.

[18] A. Rowstron, A-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The Design of a Large-scale Event Notification Infrastructure. In *Third International Workshop on Networked Group Communication*, November 2001.

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

[20] 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.

[21] 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.

[22] 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.

[23] Ellen W. Zegura, Kenneth L. Calvert, and Samrat Bhattacharjee. How to model an internetwork. In *IEEE Infocom*, volume 2, pages 594–602, San Francisco, CA, March 1996. IEEE.

[24] 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.

[25] 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.