iOverlay: A Lightweight Middleware Infrastructure for Overlay Application Implementations

Baochun Li, Jiang Guo, and Mea Wang

Department of Electrical and Computer Engineering
University of Toronto
{bli,jguo,mea}@eecg.toronto.edu

Abstract. The very nature of implementing and evaluating fully distributed algorithms or protocols in application-layer overlay networks involves certain programming tasks that are at best mundane and tedious – and at worst challenging – even at the application level. In this paper, we present *iOverlay*, a lightweight and high-performance middleware infrastructure that addresses these problems in a novel way by providing clean, well-documented layers of middleware components. The internals of iOverlay are carefully designed and implemented to maximize its performance, without sacrificing the simplicity of application implementations using iOverlay. We illustrate the effectiveness of iOverlay by rapidly implementing a set of overlay applications, and report our findings and experiences by deploying them on PlanetLab, the wide-area overlay network testbed that iOverlay conveniently supports.

1 Introduction

Existing research in the area of application-layer overlay protocols has produced a sizable collection of real-world implementations of protocols and distributed applications in overlay networks. Examples include implementations of structured search protocols such as Pastry [1] and Chord [2], as well as overlay data dissemination such as Narada [3], NICE [4], SplitStream [5] and Bullet [6]. However, an interesting observation is that most of the existing work has resorted to simulations to evaluate the effectiveness of the proposed protocols.

The recent availability of global-scale implementation testbeds for application-layer overlay protocols, such as PlanetLab [7] and Netbed [8], makes it feasible to design, implement and deploy overlay protocols in a wide-area network, so that they may be evaluated in realistic environments rather than simulations. However, there still exist roadblocks that make it impractical to deliver a high-quality, high-performance and fully distributed real-world implementation of overlay applications entirely from scratch: such an implementation involves many software components that must work together, including certain programming tasks that are at best mundane and tedious – and at worst challenging – to code.

We observe that, among all the components of a distributed application or protocol implementation, only a few specific areas are interesting for research purposes, and are subject to changes and innovations. On the other hand, any realistic implementation of overlay applications must include a significant number of largely uninteresting elements, such as (1) bootstrapping wide-area nodes from a centralized authority; (2) implementing a multi-threaded message forwarding engine; (3) monitoring facilities to control, debug, and record the performance of distributed algorithms. The necessity of writing such supporting infrastructure slows down the pace of prototyping new applications and protocols.

In this paper, we present *iOverlay*, a lightweight and high-performance middleware infrastructure that is specifically designed from scratch to support rapid development of distributed applications and protocols over realistic testbeds. The design objectives of *iOverlay* are as follows. First, it seeks to provide a high-quality and high-performance implementation of a carefully selected number of features that are common or useful to most of the overlay application implementations. Second, it seeks to be as *generic* as possible, and minimizes the set of assumptions with respect to the objectives and nature of new applications. Third, it seeks to significantly simplify the implementation of distributed applications, to the extent that only the logics and semantics specific to the application itself need to be implemented by the application developer. In addition, it should not be necessary for the application developer to have any prior knowledge about the internal details of iOverlay, before starting a successful implementation. Finally, it seeks to design a well-documented, straightforward and clean interface between the application and iOverlay.

The remainder of this paper is organized as follows. In Sec. 2, we present the design and implementation of the iOverlay architecture. In Sec. 3, we present our own experiences with rapidly prototyping a set of overlay applications as case studies. Finally, we discuss iOverlay in light of related work (Sec. 4), and conclude the paper in Sec. 5.

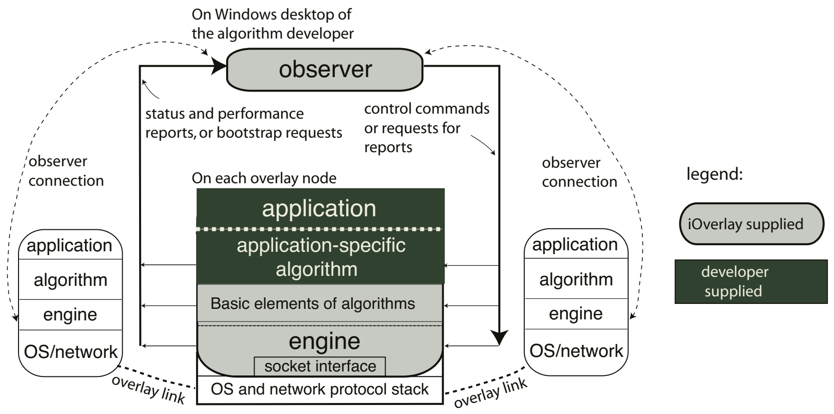


Fig. 1. The iOverlay architecture.

2 iOverlay: Design and Performance

iOverlay considers three layers in a distributed application: (1) the *message switching engine*, which performs indispensable tasks of switching application-layer messages. (2) the *algorithm*, which implements the application-specific distributed protocol beyond mundane tasks in the engine; and (3) the *application*, which produces and interprets the data portion of application-layer messages at both the sending and the receiving ends. This may include global storage systems that respond to queries, or publish-subscribe applications that produce events and interests. The ultimate objective is for the application developer to build new *algorithms* based on the engine, and to select an application to be deployed on top of the algorithm.

Architecturally, the iOverlay middleware infrastructure provides support to the application developer in all of these aspects. First, it implements a fully functional, virtualizable and high performance message switching engine, upon which the application-specific algorithm is built. Second, it implements common elements of selected categories of algorithms that are completely optional for the application developer to use. Third, it implements typical applications, which the algorithm developer may choose to deploy. Finally, it provides a centralized Windows-based graphical utility, referred to as the *observer*, for the purpose of monitoring, debugging, visualizing and logging various aspects of the distributed application. The iOverlay architecture, as discussed, is illustrated in Fig. 1.

2.1 Highlights

The iOverlay middleware design features the following highlights.

Simplified Interface. iOverlay is designed to have the simplest interface possible between the application-specific algorithm and the engine on each overlay node, in order to minimize the cost of entry to use iOverlay. The application developer only needs to be aware of *one* function of the engine: the `send` function, used for sending data or protocol messages to downstream or peer nodes. In addition to this function, the entire interface is designed to be completely *message driven*, in the sense that the algorithm only needs to *passively* process messages when they arrive or are produced by the engine. Since messages are distinguished by their *types*, a *message handler* that handles possible types is all that is required for the algorithm implementation. Further, the entire implementation of the application-specific algorithm is guaranteed to be executed in a single thread, and therefore does not need to use thread-safe data structures (those guarded with semaphores and locks).

Virtualized nodes. iOverlay features complete *virtualization* of overlay nodes in a distributed application. Each physical node in the wide-area network may easily accommodate from one to up to dozens of iOverlay nodes, depending on available physical resources such as CPU. Each iOverlay node has its own bandwidth specifications, such as the total bandwidth available to and from the node, separate upload and download available bandwidth, or per-link bandwidth limits. This adds to the flexibility of iOverlay deployment: if necessary, iOverlay may be entirely deployed in a local area network with a cluster of servers; or, for small-scale tests, on just a single server.

Maximized Performance and Flexibility. Finally, iOverlay is designed to maximize its performance. The engine is implemented from scratch with the C++ programming language and the native POSIX thread library in UNIX. It is portable across many UNIX variants. The observer is implemented in Windows using the C# programming language, guaranteeing rapid development of additional interface elements.

2.2 Internal Design

In iOverlay, we assume that all communication is in the form of *application-layer messages*, containing application data of a maximum length in bytes. Each message maintains a fixed 24-byte header, which includes the type and sender of the message, the application identifier that it belongs to, the sequence number and the size of the payload. To keep it simple, the content of a message is mostly immutable, and is initialized at the time of construction. In addition, the notion of a *node* in iOverlay is uniquely identified by its IP address and port number. The port number may be explicitly specified at start-up time; otherwise, the engine chooses one of the available ports.

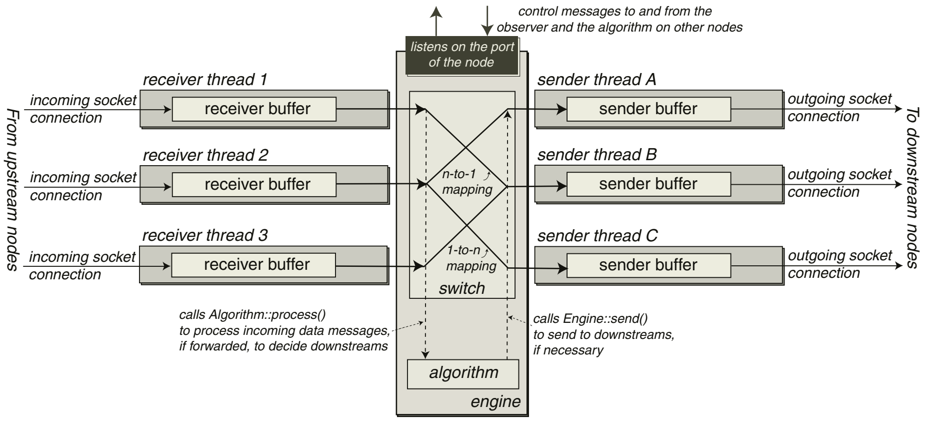

control messages to and from the observer and the algorithm on other nodes
listens on the port of the node
receiver thread 1
receiver thread 2
receiver thread 3
incoming socket connection
receiver buffer
From upstream nodes
sender thread A
sender thread B
sender thread C
sender buffer
outgoing socket connection
To downstream nodes
n-to-1 mapping
1-to-n mapping
switch
calls Algorithm::process() to process incoming data messages, if forwarded, to decide downstreams
calls Engine::send() to send to downstreams, if necessary
algorithm
engine

Fig. 2. The internal design of the engine.

The Message Switching Engine: A Close Examination

The engine of iOverlay is an application-layer message switch. We seek to design the engine such that it supports multiple competing traffic sessions, so that the application developer may easily test the performance of distributed algorithms under heavy cross traffic. It also has the capability to concurrently process both application data and protocol-specific messages.

We deploy a multi-threaded architecture to concurrently handle multiple incoming and outgoing connections, application-specific messages, as well as messages to and from the observer. Specifically, we use a thread-per-receiver and a thread-per-sender design, along with a separate *engine* thread for processing and switching messages using the application-specific algorithm. All receiver and sender threads use blocking receive and send operations, and the sender thread is suspended when the buffer is empty, to be signaled by the engine thread. We use a thread-safe circular queue to implement the shared buffers between the threads. Such a design is illustrated in Fig. 2.

We adopt such a design to avoid the complex wait/signal scenario where the receiver or sender buffer is shared by more than one reader or writer threads. Unlike the receiver and sender threads that "sleep" when the the buffer is full (receiver) or empty (sender), the engine thread constantly monitors the publicized port of the node (by using the non-blocking `select()` function) for incoming control messages from the observer, or from the algorithms of other nodes. If they exist, they are either processed within the engine, or sent to the algorithm to be processed, by calling the `Algorithm::process()` function. Next, it switches data messages from the receiver buffers to the sender buffers in a weighted round-robin fashion, with dynamically tunable weights (implemented in the `Engine::switch()` function). The skeleton of the engine thread is shown in Table 1.

Table 1. Design of the engine thread.

```
start the TCP server on the publicized port;
bootstrap from observer;
while not terminated
  if there are incoming messages on the port detected
  using non-blocking select()
    if the message is engine-related
      call Engine::process();
    else
      call Algorithm::process();
  call Engine::switch();
stop the TCP server.
```

Obviously, when the switch attempts to forward messages to downstreams, the choice of downstream nodes is at the sole discretion of the algorithm. Therefore, the engine consults with the algorithm by calling `Algorithm::process()`. There are two possibilities. First, the algorithm may locally process and consume the message. Second, the algorithm continues to forward the message to one or more downstream nodes, by calling the `Engine::send()` function. Only in the latter case does the engine forward the message to the sender buffers.

The tight coupling of the algorithm's and the engine's message processing components is intentional by design. First, they must reside in the same thread, since we prefer to avoid the cases where the developer needs to use thread-safe data structures when algorithms are developed with iOverlay. It is impossible to design a typical two-thread solution – where the engine processes control messages in one thread, and switches data messages in another – and still achieve such a favorable property of accommodating thread-unaware algorithms. Second, the seemingly complex "paradox" – at times the engine calls the algorithm, and at other times the algorithm calls the engine – is in fact straightforward, since the algorithm is always *reactive* and never *proactive*.

There are further complexities involved in the design of a switch. As a first example, there may be cases where messages are successfully forwarded to only a subset of the intended senders, but fail to be forwarded to the remaining ones, since their buffers are full. In this case, we label each message with its *set of remaining senders*, so that they may be tried in the next round. As a second example, in some scenarios a set of states needs to be shared and exchanged between active threads. For example, a receiver thread needs to notify the engine when a failed upstream node has been detected, such that the engine thread can clear up its data structures related to this node. To avoid complex thread synchronization between active threads, we extensively take advantage of the mechanism of passing application-layer messages across thread boundaries via the publicized port. Without a doubt, these complexities are completely transparent to the algorithm developer.

Finally, we may not only wish to forward verbatim messages in an application-layer switch, but also wish to merge or code multiple incoming messages into one outgoing message. In order to implement the most generic n-to-m mapping (such as coding messages from n multiple incoming connections to m downstreams), we allow `Algorithm::process()` to return a *hold* type, instructing the engine that the message is buffered in the algorithm, but its processing should be put on hold to wait for other messages from other incoming connections. It is up to the algorithm to implement the logic of merging or coding multiple messages after requesting a hold on them, and eventually producing a new message to be sent to downstreams. Using the *hold* mechanism, we have successfully implemented algorithms that perform overlay multicast with merging or network coding [9].

Salient Features

Handling of Failures. In iOverlay, we assume that the nodes themselves, the virtual link between nodes, as well as the application data sources may all fail prematurely. Transparent to the algorithm developer, iOverlay supports the automatic detection of failed nodes and links, and the automatic tear-down of relevant links after such failures. For example, if an upstream link in a multicast tree has failed, it causes a "Domino Effect" that fails all downstream links from this point. The engine is able to appropriately tear down these links without affecting any of the other active links, and to notify the algorithm of such failures. All terminations are graceful, and all affected links are smoothly dropped without side effects.

We have implemented a collection of exception handling mechanisms to detect and process such failures. Depending on the state of the sockets at the time of premature failures, we rely on a combination of mechanisms to detect that a node or a link may have failed: (1) exceptions thrown and timeouts at the socket level; (2) abnormal signals caught by the engine, such as the *Broken Pipe* signal; and (3) long consecutive periods of traffic inactivity, detected by throughput measurements. To avoid overhead, we do not use any forms of active probes or "heartbeat updates" for this purpose. Still, we are able to implement very responsive detections of link and node failures in most cases. In addition, the observer may choose to terminate a node at will, in which case all the data structures and threads in both the engine and the algorithm will be cleared up, and the program terminates gracefully.

Measurement of QoS Metrics. At the socket level, we have implemented mechanisms to measure the TCP throughput of a connection, as well as the round-trip latency and the number of bytes (or messages) lost due to failures. The results of these measurements are periodically reported to the algorithm and the observer. Upon requests from the algorithm, the available bandwidth and latency to any overlay nodes can be measured.

Emulation of Bandwidth Availability. In some cases, the algorithm developer prefers to test a preliminary algorithm under controlled environments, in which node characteristics are more predictable. iOverlay explicitly supports the emulation of bandwidth availability in three categories: (1) per-node total bandwidth: the total incoming and outgoing bandwidth available; (2) per-link bandwidth: the bandwidth available on a certain point-to-point virtual link; and (3) per-node incoming and outgoing bandwidth: iOverlay is able to emulate asymmetric nodes (such as nodes on DSL or cable modem connections) featuring disparate outgoing and incoming bandwidth availability. The emulated values may be specified at node start-up time, or within the observer at runtime. In the latter case, artificially emulated bottlenecks may be produced or relieved on the fly, in order to evaluate the adaptivity of the algorithm. To implement such emulations, we have wrapped the socket `send` and `recv` functions to include multiple timers in order to precisely control the bandwidth used per interval (the length of which may be specified by the algorithm).

Performance Considerations

The performance objective of the engine design is to "push" messages through the engine as quickly as possible, with the lowest possible overhead at the switch. Towards this objective, we have considered three directions of performance optimizations, and successfully implemented them in the current engine.

Persistent Connections. In order to avoid the unacceptable overhead of thread-level context switching at the operating system when a large number of threads are used, we implement both incoming and outgoing socket connections as *persistent connections*, in the sense that all the messages between two nodes are carried with the same connection, regardless of the applications they belong to. With persistent connections, we have avoided the creation of more threads when new distributed applications are deployed; instead, existing connections are reused.

Zero Copying of Messages. In order to avoid deep copying of entire messages when they pass through the engine, we have implemented a collection of mechanisms to ensure that only the references of messages are passed from the incoming socket all the way to the outgoing socket, and *no messages will be copied* in the engine at all. The algorithm may choose to copy messages, if necessary, supported by the copy constructor of the `Msg` class. In order to appropriately destruct messages whose references are shared by multiple threads, an elaborate thread-safe reference counting mechanism is in place in the core of the engine.

Footprint. The engine is meticulously designed and tested so that the memory footprint is minimized and stable (without leaks). For example, with a message size of 5 KB and a buffer capacity of 10 messages, the footprint of the engine is only 4 MB per active connection[1]. The optimized binary executable of the engine (with a simple testing algorithm) is only 100 KB. Such a footprint guarantees the scalability of iOverlay, especially when a large number of virtualized nodes are deployed on the same physical server.

The Observer

As a centralized monitoring facility, we have implemented the observer as a graphical tool in Windows. The observer implements the first level of bootstrap support, by responding to any bootstrap requests (messages of type *boot*) with a random subset of existing nodes that are alive. The number of initial nodes in such a subset is configurable. Once a node is bootstrapped, the observer periodically sends it a *request* message to request for status updates, which include lengths of all engine buffers, measurements of QoS metrics, and the list of upstream and downstream nodes. With these status updates, the observer may visually illustrate the current network topology of each of the applications with geographical locations of all nodes, on either the world or the North American map.

Further, the observer serves as a control panel and may take the following actions to control the status of the network: (1) controlling the emulated per-link and per-node bandwidth availabilities; (2) deploying an application; (3) asking a node to join or leave a particular application; and (4) terminating an application data source or a node. For the sake of flexibility, the observer is also able to send new types of algorithm-specific control messages to the nodes, with two optional integer parameters embedded in the header.

Basic Elements of Algorithms

Despite the tight coupling between the algorithm and the engine, the algorithm is placed in its own namespace with an object-oriented design. The basic and com-

[1] This is the case in Linux, which may be inferior with respect to footprint since `clone()` is usually used to support user-level POSIX threads.

monly used elements of an algorithm is defined and implemented in a generic base class referred to as *iAlgorithm*. We present two examples. First, it implements a default message handler, that handles known messages from the observer and the engine with a default behavior. For example, upon receiving the bootstrap message from the observer, it records the set of initial nodes in a local data structure referred to as `KnownHosts`. Second, *iAlgorithm* implements a `disseminate` function, which disseminates a message to a list of overlay nodes, with a specific probability p. This resembles the *gossiping* behavior in distributed systems. The default implementations of a library of functions in the *iAlgorithm* class serve as a set of basic utilities, and since application-specific algorithms are classes that inherit from *iAlgorithm*, the developer may choose to override any default behavior with application-specific implementations.

2.3 Interface Between iOverlay and Algorithms

Given the iOverlay design we have presented, how do we rapidly develop an application using iOverlay? Many design choices are made to reduce the complexity of developing new application-specific algorithms. First, the algorithm namespace extensively uses object orientation such that new algorithms may be built based on existing algorithm implementations. As we have discussed, a few basic elements of algorithms have already been provided by iOverlay. Second, the algorithm only needs to call *one* function of the engine: the *send* function. This greatly improves the learning curve of the interface. Finally, the algorithm is designed as a message handler, in the form of a *switch* statement on different types of messages. While processing each incoming message, internal states of the algorithm may be modified. The message handler should reside in the `process()` function. The skeleton of an algorithm is shown in Table 2.

In such a skeleton, it is not necessary for an algorithm to handle all the known message types from the engine or the observer. If a message type is not handled in the algorithm, the default `process()` function provided by the base *iAlgorithm* class takes this responsibility. In fact, the only message type that the algorithm must handle is the type `data`, indicating a data message. iAlgorithm provides default handlers for all other types of messages. It is also not necessary for an algorithm to handle abnormal return values when invoking the `send()` function. In fact, `send()` has a return type of `void`, and all abnormal results of sending a message are handled by the engine transparently. For example, if the destination node of the message fails, the algorithm is notified appropriately, again via messages produced by the engine.

Another important design decision is related to the destruction of messages. In order to completely eliminate memory leaks, we need to carefully assign the responsibilities of message destruction. Particularly, consider a message passed to the algorithm (by pointers) as a parameter in the *process* function. Should the engine or the algorithm be responsible for destructing the message after it has been processed? Further, when a message is constructed in the algorithm and passed to the *send* function of the engine, should the engine or the algorithm be responsible for destructing the message after it is sent? To simplify the tasks of

Table 2. Skeleton of the algorithm using iOverlay.

```
process(Msg * m)
  switch (m -> type())
  case sDeploy: (from observer)
    deploy an application source;
  case request: (from observer)
    send algorithm status updates to observer;
  case sTerminate: (from observer)
    terminate an application source;
  case BrokenSource: (from upstream)
    clear up internal states corresponding to the application
    source at upstream, since it has failed;
  case data: (from the engine)
    process, consume or forward the message using
    send(Msg * m, Node dest);
  case UpThroughput: (from the engine)
    record or process the throughput from an upstream;
  ... (process other engine or algorithm-specific types)
  default: (use the default behavior from iAlgorithm)
    iAlgorithm::process(m);
```

algorithm developers, we stipulate that *all message destructions are the responsibility of the engine*. The algorithm developer should never destruct messages, even if they have been constructed in the algorithm.

However, there exist a subtle problem with this solution even it works well at most times. When the algorithm receives a pointer to an engine-created message as a parameter of the *process* function, what if the algorithm passes the pointer back to the engine by using the *send* function? We distinguish treatments of this scenario depending on the type of the message. If the message is of type `data`, we have developed the engine carefully such that the algorithm can directly invoke *send* with the same message, guaranteeing zero copying of data messages. However, if the message is of any other type, we require the algorithm developer to clone the message before invoking *send* on the new copy. Performance-wise this is not a problem, since most protocol messages are very small in size.

2.4 Performance

With C++ on Linux, C# on Windows, and around 25,000 lines of code in total, we have completed a stable implementation of the entire iOverlay middleware infrastructure that we have presented. We now evaluate the results of such an implementation, focusing on the raw message switching performance of iOverlay nodes, especially when they are virtualized nodes on the same server. For this purpose, we execute iOverlay nodes on a single dual-CPU server with two Pentium III 1GHz processors, 1.5GB of memory, and Linux 2.4.25. The iOverlay engine is compiled with `gcc 3.3.3` with the most aggressive optimizations.

Since iOverlay nodes are multi-threaded user-level programs, the bottleneck of such switching performance under heavy load is the overhead of context switching among a large number of threads. We create such a load using a chain topology, and we test iOverlay with different number of nodes in the network. Before we deploy an application on the chain topology, we observe that the CPU load is `0.00`, which shows that iOverlay does not consume CPU resources without traffic. After we deploy an application that sends back-to-back traffic from one end of the chain to the other as fast as possible, we measure the end-to-end throughput, as well as the total bandwidth in the chain, calculated by the end-to-end throughput multiplied by the number of links. The total bandwidth represents the actual number of messages per second that have been switched or in transit in the network. Fig. 3 shows the iOverlay engine performance in this test, with a chain from two nodes to 32 nodes.

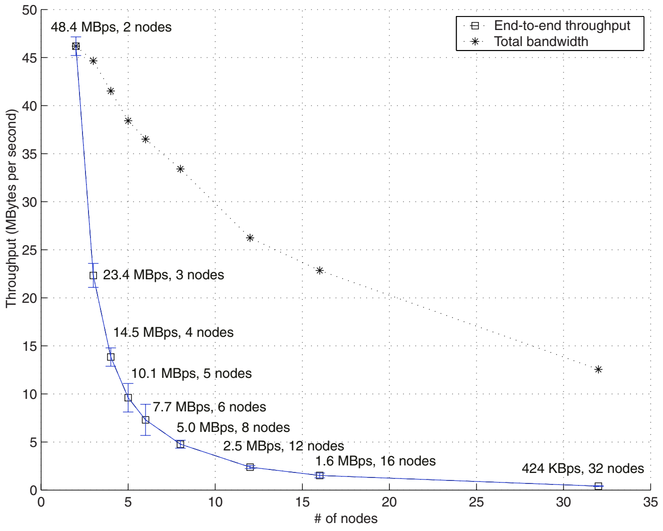


Fig. 3. The raw performance of the iOverlay engine.

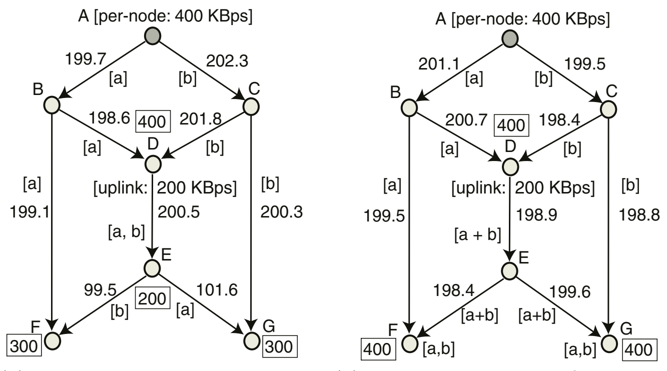
We have two noteworthy observations from this experiment. First, if we compare the two-node total bandwidth of 48.4 MBps and the three-node bandwidth of 46.8 MBps, the overhead of one user-level message switch is only 3.3%. Second, as the number of nodes increases, the overhead of context switching becomes more significant, due to the Linux implementation of POSIX threads using `clone()`. Still, even with a 32-node configuration, the sustained throughput is still 424 KBps, which is higher than the typical throughput of wide-area connections. This implies that we may potentially deploy dozens of nodes on a single physical node in a local-area or wide-area testbed, making it feasible to test the scalability of new applications in terms of the number of participants. Such performance is simply not achievable if, for example, Java is used rather than C++, or zero message copying is not enforced.

3 Case Studies

We believe that iOverlay is useful to support the rapid implementation of a wide range of applications and distributed algorithms in application overlay networks. In this paper, we undertake three case studies to highlight our own experiences of rapidly prototyping new algorithms and ideas using iOverlay as the middleware infrastructure.

3.1 Network Coding

The advantages of application-layer overlay networks arise from the fundamental property that overlay nodes, as opposed to lower-layer network elements such as routers and switches, are end systems and have capabilities far beyond basic operations of storing and forwarding. In the first case study, we implement a novel message processing algorithm that performs *network coding* on overlay nodes, using iOverlay. In such an algorithm, messages from multiple incoming streams are coded into one stream using linear codes in the Galois Field (and more specifically, with GF(2^8)). We are pleasantly surprised that, with one developer, such a non-trivial task is completed within a week. We have evaluated the network coding algorithm in the topology shown in Fig. 4, where we show the performance of the algorithm.



(a) Without network coding, A sends half of the messages to B, and the other half to C. The effective throughput to node D is 400 KBps, while nodes F and G receive 300 KBps.

(b) Network coding is performed on nodes D, F and G. The effective throughput to nodes D, F and G are 400 KBps, while B, C and E are "helper" nodes.

Fig. 4. Performance of network coding: an iOverlay case study.

Fig. 4(a) shows the results without using network coding. Node A is the data source with per-node bandwidth of 400 KBps, and node D has an uplink bandwidth of 200 KBps. Node A splits its data into two streams sent to B and C, respectively. In this case, B and C are not able to receive both streams, and are referred to as *helper* nodes. Based on iOverlay throughput measurements, the nodes D, E, F and G have received $400, 200, 300, 300$ KBps, respectively. In comparison, Fig. 4(b) shows the case where the coding algorithm $a+b$ in GF(2^8) is applied at node D on the two incoming streams. In this case, the nodes F and G are able to receive both streams a and b by decoding $a+b$ with a, achieving a throughput of 400 KBps. The trade-off, however, is that node E becomes a helper node, in addition to B and C. Our experiences with this case study have demonstrated both the advantages and the trade-offs of applying network coding on overlay nodes. We believe that such an experiment-based evaluation of network coding algorithms is not possible within such a short time frame, if iOverlay is not available as a substrate. For more details of our implementation on network coding, the interested reader is referred to our companion paper [10].

3.2 Construction of Data Dissemination Trees

In this case study, we are interested in the development and evaluation of new algorithms that construct data dissemination multicast trees in overlay networks, particularly in the scenario that the "last-mile" available bandwidth on overlay nodes is the bottleneck. With iOverlay, we have implemented a *node stress* aware algorithm to construct such multicast trees, where *node stress* is defined as the degree of a node in a data dissemination topology divided by the available "last-mile" bandwidth of the node.

The outline of this algorithm is as follows. Periodically, each node in the existing multicast session exchanges node stress information with its parent and child nodes. As a node A joins the multicast session, it first locates a node *that is currently in the tree* by using one of the utility functions supported in iOverlay, which disseminates a *sQuery* message. As the message is relayed to the first such node B in the tree, B compares its own node stress with its parent and child nodes. If B itself has the minimum node stress, it responds with an *sQueryAck* message, so that A becomes a new child of B in the tree. Otherwise, it recursively forwards the message to the node with the minimum node stress (parent or children), until the message reaches the minimum-stress node who sends the acknowledgment.

In order to evaluate such an algorithm in a comparative study, we have also implemented the *all-unicast* and *randomized* tree construction algorithms as control. In the all-unicast algorithm, node B – or any node who is aware of the source of the session (e.g., from the *sAnnounce* message in iOverlay) – simply forwards the *sQuery* to the data source of the session. In the randomized algorithm, node B directly sends the *sQueryAck* acknowledgment to A, and A will join the tree on receiving the first such acknowledgment.

We first experiment with a five-node data dissemination session, shown in Fig. 5, in which the data source is deployed on node S, and nodes A – D joins the session in the order of D, A, C, and B. The figure has been annotated with the per-node available bandwidth, as well as the throughput that we have obtained in our experiments. The node degree and stress are summarized in Table 3. It is very clear that, with respect to end-to-end throughput, our new

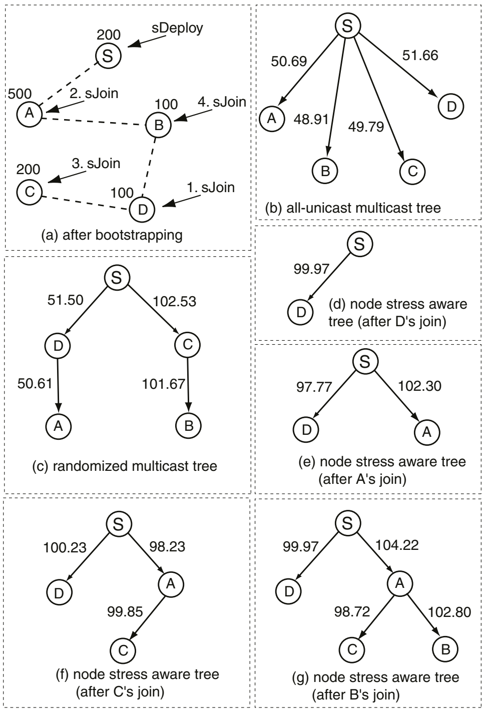

200
sDeploy
S
500
2. sJoin
A
100
4. sJoin
B
200
3. sJoin
C
100
1. sJoin
D
(a) after bootstrapping
50.69
51.66
48.91
49.79
(b) all-unicast multicast tree
51.50
102.53
50.61
101.67
(c) randomized multicast tree
99.97
(d) node stress aware tree (after D's join)
97.77
102.30
(e) node stress aware tree (after A's join)
100.23
98.23
99.85
(f) node stress aware tree (after C's join)
99.97
104.22
98.72
102.80
(g) node stress aware tree (after B's join)

Fig. 5. Tree construction algorithms: throughput (in KBytes per second).

algorithm has the upper hand. We have also observed that the topology of the node stress aware tree is not *optimal*, there may be better trees with respect to throughput. For example, in Fig. 5(g), if D is a child of A rather than S, throughput may be further improved, leaving possibilities for further research. Such experiment-based insights would not be possible without the substrate that iOverlay provides.

In the next experiment, we choose to evaluate the performance and stress tolerance of the node stress aware algorithm in large-scale overlay networks, by deploying it to a total of 81 wide-area nodes in PlanetLab. The per-node available bandwidth has been specified to a uniform distribution of 50 to 200 KBps for all the nodes, with the source node set at 100 KBps. By taking advantage of the deployment scripts in iOverlay, we are able to deploy, run, terminate and collect data from all 81 nodes, with one command for each operation. Fig. 6 shows the North American portion of the wide-area topology after 30 nodes have joined the data dissemination session.

The results we have obtained from these PlanetLab experiments are illustrated in Fig. 7. With respect to node stress, we may observe that the node stress aware algorithm has managed to approach the ideal case (*i.e.,* the vertical line at node stress 20) much better than the other cases. With respect to end-to-end throughput, we may observe that the throughput is much higher with the node stress aware algorithm.

Table 3. Tree construction algorithms: node degree and stress.

Node	node degree			node stress (1/100 KBps)		
	unicast	random	ns-aware	unicast	random	ns-aware
S	4	2	2	2.0	1.0	1.0
A	1	1	3	0.2	0.2	0.6
B	1	1	1	1.0	0.98	0.97
C	1	2	1	0.5	1.0	0.51
D	1	2	1	1.0	1.98	1.0

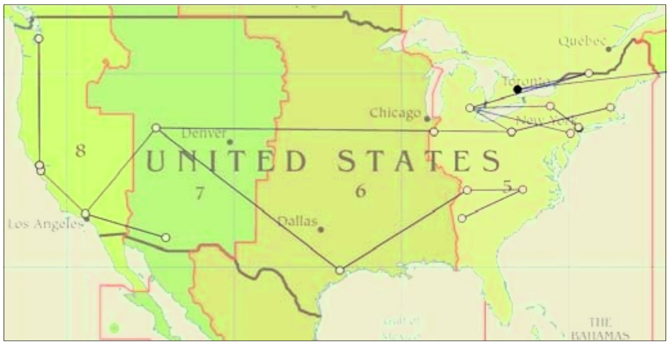
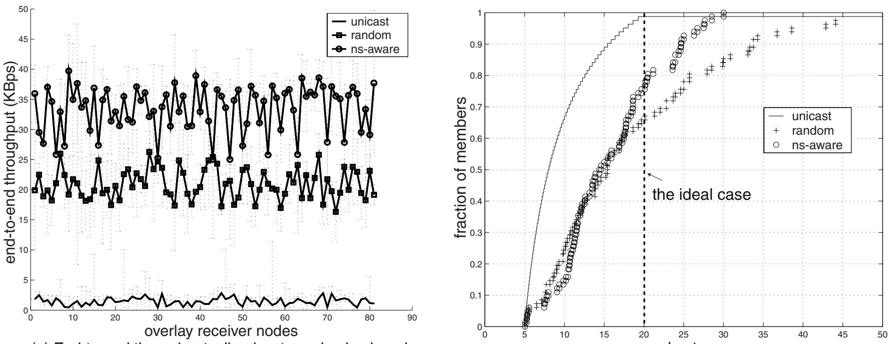


Fig. 6. The real-time wide-area topology produced by the node stress aware algorithm after 30 nodes have joined (only nodes that reside in North America have been shown, some nodes may reside in the same geographical location).



(a) End-to-end throughput: all-unicast, randomized, and ns-aware tree construction algorithms (dotted lines show the spread of measurements).

(b) Cumulative distribution of node stress: all-unicast, randomized and the node stress aware algorithms.

Fig. 7. Performance of the node stress aware algorithm using 81 wide-area nodes in PlanetLab. (a) end-to-end throughput; and (b) the cumulative distribution of node stress.

3.3 Service Federation in Service Overlay Networks

In some applications, data messages may need to be transformed (such as media or web data transcoding) by a series of third-party nodes (or *services*) before they reach their destinations. The process of provisioning a complex service by constructing a topology of a selected group of primitive services is known as *service federation* (or *composition*), within what is referred to as *service overlay networks* consisting of instances of primitive services. In order to start a service federation process, a specific *service requirement* needs to be specified, which includes the required primitive services in order to compose the federated service. As a case study, we have designed and implemented a new distributed algorithm, referred to as *sFlow*, to federate complex services that require service requirements in the generic form of directed acyclic graphs, with the aid of iOverlay and over a period of three weeks.

We outline the gist of the algorithm as follows. When a new service is established by the *sAssign* message from the observer, it locally maintains a *service graph* that represents the producer-consumer relationships among different types of services, and disseminates its existence to all its known hosts via the *sAware* message. The message is further relayed until an existing service node is reached, which forwards the message to the direct upstream and downstream nodes of the new service in its service graph. When a service federation session is started using the observer, the requirement for the complex service is specified in a *sFederate* message to the designated source service node. As this message is forwarded, each node applies a local algorithm to select the most bandwidth efficient downstream service node according to the requirement, until the sink service node is reached. The federation process is concluded with the deployment of actual data streams through the selected third-party services. In order to construct a high-quality service topology, the algorithm takes advantage of iOverlay's feature that measures point-to-point throughput to selected known hosts.

We start our experiments by implementing our new algorithm on 16 real-world nodes in PlanetLab, mostly in North America, to construct a service overlay network. The best-quality – *i.e.,* most bandwidth efficient – federated service according to a particular service requirement is presented in Fig. 8. Each node in Fig. 8 is labeled with a service identifier assigned to them by the observer. The edges indicate a live service federation session where live data streams are being transmitted. The end-to-end delay of this service session is 934.547 milliseconds, and the last hop average throughput is measured as 69374 bytes per second.

During the session, we record detailed statistics on bandwidth measurements and control message overhead on each of the 16 nodes, shown in Fig. 9. In this experiment, the sAware message overhead depends on the number of known hosts of each node, and the overhead of sFederate messages is sufficiently small, compared to that of sAware messages. The per-link and total per-node bandwidth are illustrated in Fig. 9(b) in descending order. Evidently, the overhead incurred by the algorithm is sufficiently small, and seven nodes are left untouched during the entire session of the protocol, since they do not host services or are not involved in the service federation process.

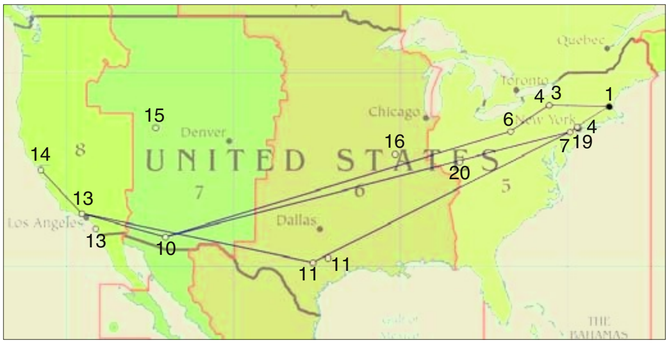


Fig. 8. The constructed complex service in a service overlay network.

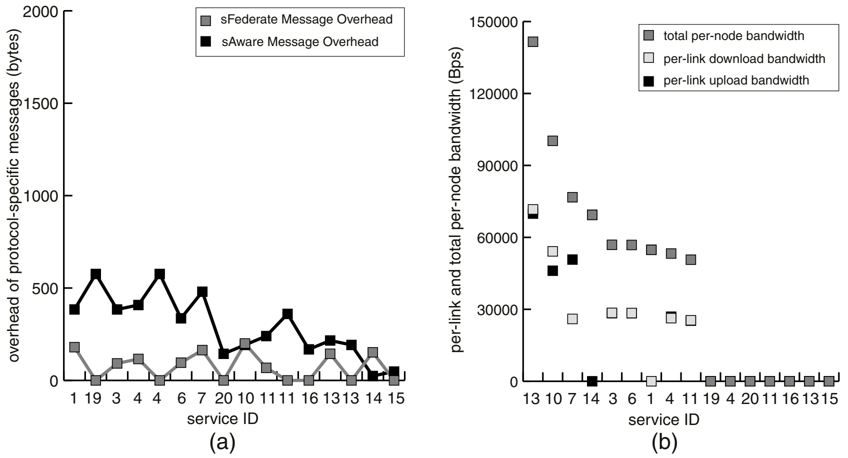


Fig. 9. Service federation: (a) control message overhead; (b) per-link and per-node bandwidth measurements on each of the overlay nodes. The overlay nodes are sorted by their per-node bandwidth availability.

4 Related Work

iOverlay was originally motivated by our own experiences of implementing distributed application implementations on overlays, when we have failed to locate a suitable middleware framework for such developments. The idea behind iOverlay originates from the *Flux OSKit* project [11] in operating system design, where a modular set of OS components are designed to be reusable, and to facilitate rapid development of experimental OS kernels. iOverlay provides a reusable set of components in the domain of overlay rather than OS implementations, and seeks to achieve similar design objectives that support rapid prototyping of new overlay-based distributed applications. Particularly, iOverlay is designed to minimize the bar of entry: in order for it to be useful, it is *not* required to have either knowledge about its internals, or extensive system-level programming skills. In addition, iOverlay is also designed to reside at a "higher level" than previous work on user-level network protocol stack implementations (e.g., Alpine [12]), and aims at the development of application-layer rather than network protocols, without the requirements of root privileges.

There exist previous work on using virtual machines (such as VMWare or User-Mode Linux) and support the deployment of full-fledged applications over a virtual network (e.g., [13]), as well as on emulation testbeds and environments to test network protocols in a virtualized and sandboxed environment (e.g., Netbed [8] and ModelNet [14]). In comparison, the objective of iOverlay is to facilitate the development of distributed applications and algorithms at the application layer, and iOverlay assumes the availability of a wide-area network testbed such as PlanetLab. Although iOverlay supports virtualizing multiple overlay nodes on a single physical node, all implementations are achieved at the user level beyond the abstraction of sockets. iOverlay is designed to be tightly coupled with applications and distributed algorithms, rather than a supporting infrastructure based on either virtual machines or emulation environments.

In particular, ModelNet [14] has introduced a set of ModelNet core nodes that serve as virtualized *kernel-level packet switches* with emulated bandwidth, latency and loss rates. Such kernel-level modifications may not be achievable in wide-area testbeds due to the lack of root privileges. The iOverlay engine, in contrast, implements *application-layer message switches*, that may be bundled with any new algorithms and deployed in the user space of any UNIX hosts. Thanks to the virtualization of iOverlay nodes, it is not required to have access to a large-scale network in order to experiment with large-scale application topologies.

To the best of our knowledge, there exist two previous papers that present similar objectives to iOverlay. First, the *PLUTO* project [15], an underlay topology service (or routing underlay) for overlay networks, based on PlanetLab. PLUTO is a layer between the overlay algorithms and the network, that exposes topological information to the algorithms. More specifically, it may expose information on connectivity, disjoint end-to-end paths between overlay nodes, as well as the distance between nodes in terms of a particular metric such as latency or router hops. We believe that iOverlay and PLUTO are completely *complementary* with each other, and that it is straightforward for the algorithm to simultaneously take advantage of both architectures. From the viewpoint of PLUTO, iOverlay is simply an overlay application. When it comes to measurement of metrics, iOverlay focuses on measuring the performance of active or potential overlay links, while PLUTO focuses on obtaining insights on the underlay physical topology. From this perspective, iOverlay operates at a higher level than PLUTO does, and PLUTO may be easily integrated into the overall iOverlay middleware architecture.

Second, the *Macedon* project [16] offers a common overlay network API by which any Macedon-created overlay implementation may be used. It features a new language to describe the behavior of an overlay algorithm, from which actual code can be generated using a code generator. As a result, Macedon allows algorithm designers to focus their attention on the algorithm itself, and less on tedious implementation details. Despite the similarities between the design objectives of Macedon and iOverlay, the design principles are drastically different. Macedon attempts to minimize the lines of code to be developed by the algorithm developer, by providing a new language to specify the characteristics of the algorithm. In contrast, iOverlay seeks to maximize the freedom and flexibility when designing new algorithms, by minimizing the API between the middleware and the application. While Macedon is able to support Distributed Hash Table based searching and overlay multicast algorithms, iOverlay is sufficiently generic to accommodate virtually any applications to be deployed on overlay networks, while still encapsulating tedious and common functional components such as message switching, throughput emulation, fault detection and recovery, as well as a centralized debugging facility. Our recent experiences of successfully and rapidly deploying a Windows-based MPEG-4 real-time streaming multicast application on iOverlay have verified our claims.

5 Concluding Remarks

We have been pleasantly surprised at how phenomenally rapidly one can develop fully distributed overlay applications using iOverlay. The evolution of features we have presented have been entirely demand-driven: rather than being designed *a priori*, with inevitably flawed vision of what new applications may need, iOverlay has been constantly refined and augmented, driven by the needs of new application implementations. From this experience, we conclude that research and implementation of overlay applications and algorithms are significantly aided by having reusable, extensible and customizable components that iOverlay provides. As a matter of fact, the burden on the application developer is completely shifted to the core portion of the application-specific algorithm, rather than subtle and mundane details that iOverlay has encapsulated. We are convinced that the full potential of iOverlay has yet to be realized. For example, the library of prefabricated algorithms may be significantly extended, in the form of new classes derived from the base *iAlgorithm* class. These new extensions may become foundations of similar categories of algorithms, which may further simplify the process of new application implementations. In addition, the PLUTO routing underlay may be integrated into the iOverlay framework as additional reusable components in the form of libraries, in order to support algorithms that need topological knowledge of the underlying IP topology.

References

1. Rowstron, A., Druschel, P.: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In: Proc. of IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001). (2001)

2. Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In: Proc. of ACM SIGCOMM. (2001)
3. Chu, Y., Rao, S.G., Seshan, S., Zhang, H.: A Case for End System Multicast. IEEE Journal on Selected Areas in Communications (2002) 1456–1471
4. Banerjee, S., Bhattacharjee, B., Kommareddy, C.: Scalable Application Layer Multicast. In: Proc. of ACM SIGCOMM. (2002)
5. Castro, M., Druschel, P., Kermarrec, A.M., Nandi, A., Rowstron, A., Singh, A.: SplitStream: High-Bandwidth Multicast in Cooperative Environments. In: Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003). (2003)
6. Kostic, D., Rodriguez, A., Albrecht, J., Vahdat, A.: Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh. In: Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003). (2003)
7. Peterson, L., Anderson, T., Culler, D., Roscoe, T.: A Blueprint for Introducing Disruptive Technology into the Internet. In: Proc. of the First Workshop on Hot Topics in Networks (HotNets-I). (2002)
8. White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An Integrated Experimental Environment for Distributed Systems and Networks. In: Proc. of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002), to appear. (2002)
9. Ahlswede, R., Cai, N., Li, S.Y.R., Yeung, R.W.: Network Information Flow. IEEE Trans. on Information Theory **IT-46** (2000) 1204–1216
10. Wang, M., Li, Z., Li, B.: A Case for Coded Overlay Flows. Technical report, Department of Electrical and Computer Engineering, University of Toronto, submitted for review to IEEE INFOCOM 2005 (2004) http://iqua.ece.toronto.edu/papers/case-coding.pdf.
11. Ford, B., Back, G., Benson, G., Lepreau, J., Lin, A., Shivers, O.: The Flux OSKit: A Substrate for Kernel and Language Research. In: Proc. of the 16th ACM Symposium on Operating Systems Principles (SOSP 1997). (1997)
12. Ely, D., Savage, S., Wetherall, D.: Alpine: A User-Level Infrastructure for Network Protocol Development. In: Proc. of the the 2001 USENIX Symposium on Internet Technologies and Systems (USITS 2001). (2001)
13. Jiang, X., Xu, D.: vBET: a VM-Based Emulation Testbed. In: Proc. of ACM Workshop on Models, Methods and Tools for Reproducible Network Research (MoMeTools 2003). (2003)
14. Vahdat, A., Yocum, K., Walsh, K., Mahadevan, P., Kostic, D., Chase, J., Becker, D.: Scalability and Accuracy in a Large-Scale Network Emulator. In: Proc. of 5th Symposium on Operating Systems Design and Implementation (OSDI 2002). (2002)
15. Nakao, A., Peterson, L., Bavier, A.: A Routing Underlay for Overlay Networks. In: Proc. of SIGCOMM 2003. (2003)
16. Rodriguez, A., Killian, C., Bhat, S., Kostic, D., Vahdat, A.: MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. In: Proc. of the USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004). (2004)