Overlay Weaver: An Overlay Construction Toolkit

Kazuyuki Shudo, Yoshio Tanaka, and Satoshi Sekiguchi

Grid Technology Research Center, National Institute of Advanced Industrial Science and Technology, AIST Tsukuba Central 2, Tsukuba, Ibaraki, 305-8568, Japan

Abstract

A layered model of structured overlays has been proposed and it enabled development of a routing layer independently of higher-level services such as DHT and multicast. The routing layer has to include other part than a routing algorithm, which is essential for routing. It is routing process, which is common to various routing algorithms and can be decoupled from a routing algorithm.

We demonstrated the decomposition by implementing an overlay construction toolkit Overlay Weaver. It facilitates implementation of routing algorithms and we could multiple well-known algorithms just in hundreds of lines of code with the toolkit. The decomposition also enables multiple implementations of the common routing process. Two implementations the toolkit provides perform iterative and recursive routing respectively. Additionally, to our knowledge, the toolkit is the first feasibility proof of the layered model by supporting multiple algorithms and the higher-level services.

Such modular design contributes to our goal, which is facilitation of rapid development of realistic routing algorithms and their application. We demonstrates that Overlay Weaver supports the goal by conducting large-scale tests and comparisons of algorithms on a single computer. The resulting algorithm implementations work on a real TCP/IP network as it is.

Key words: Overlay Network, Structured Overlay, Distributed Hash Table, Emulation

1 Introduction

It is essential for large-scale Internet-wide applications to construct an application-level network in an autonomous and decentralized way. Such a network is

Computer Communications (Special Issue on Foundations of Peer-to-Peer Computing),
Elsevier Science, Volume 31, Issue 2, pp.402-412, February 5, 2008
(available online on August 14, 2007)

called an overlay network because it is based on the underlying physical network, but its structure is different from the underlying network. An overlay provides elemental functions to build distributed applications such as lookup and multicast, and maintains performance and fault tolerance with more than millions of computers.

It requires a fair amount of work to incorporate an overlay algorithm into application software. For example, distributed hash table (DHT), a representative application of structured overlays, itself has thousands of lines of code at least, because it needs various jobs like communication and storage management other than the overlay algorithm. Its application must be larger and more complicated than it.

Dabek et al. proposed layered abstractions of structured overlays [1]. They separated common services such as DHT, multicast and anycast from an underlying routing layer. The model suggests that it enables separated design and implementation of a routing layer named key-based routing (KBR) from the common higher-level services.

The routing layer is monolithic even with the layered abstractions. Routing algorithm is not the only part of the routing layer, which has to perform other jobs including communication. Furthermore, the routing layer has to include other part than a routing algorithm, which is the essential for routing. It is routing process, which is common to various known routing algorithms. it can be decoupled from a routing algorithm.

We demonstrated the decomposition by implementing an overlay construction toolkit *Overlay Weaver* [2]. We decoupled a routing algorithm from the common routing process by designing a programming interface between them. The interface has been able to support well-known algorithms, Chord, Pastry, Tapestry and Kademlia. Additionally, to our knowledge, the toolkit is the first feasibility proof of the layered abstractions proposed by Dabek et al. [1], because the toolkit provides the multiple routing algorithms and multiple higher-level services such as DHT and multicast.

The decomposition facilitates development of routing algorithms. Implementations of the above-mentioned well-known algorithms are just in hundreds of lines of code. The decomposition also allows multiple implementations of the common routing process to be combined with routing algorithms. The toolkit provides two implementations which perform iterative and recursive routing respectively.

Such modular design contributes to our goal. It is facilitation of rapid development of realistic routing algorithms and their application. Overlay Weaver provides an emulator of distributed environment to support the goal. It enables algorithm developers to test and improve their algorithms with large number of nodes on a single computer. The resulting algorithm implementations work in a real environment.

The following section discusses related work. We show a programming interface, which separates the routing process from the algorithms in Section 3. Section 4 describes the components and tools the toolkit provides and how they support overlay design, implementation, evaluation, and comparison. In Section 5, we demonstrate that we could implement various well-known algorithms with little code. We also present the results of tests and comparisons of the algorithms on a large scale.

2 Related Work

MACEDON [3,4] and Mace [5] are overlay construction software which support multiple routing algorithms as does the Overlay Weaver.

A user describes an algorithm in MACEDON language, which is like C/C++ but specific to the overlay description. MACEDON translates the description into executable C++ code. The generated code communicates using TCP or UDP, and MACEDON can generate code for a network simulator ns even though this is "partial support" [3]. MACEDON provides distributed hash table (DHT) implementations, i.e., Chord and Pastry.

The amount of code MACEDON requires to describe an algorithm is comparable to Overlay Weaver. The difference is only up to 50 percent of the amount of code for Overlay Weaver (Section 5.1). MACEDON reduced the amount by introducing a domain-specific language and Overlay Weaver achieved similar results by the separation of routing process and algorithms. Possible problems each approach introduces are as follows. MACEDON's approach involves a higher learning cost for dedicated language. On the other hand, routing driver (Section 3) of Overlay Weaver provides no specific support to implementation of unstructured overlays, which require a larger amount of code. However, it is possible to implement unstructured overlays directly using the toolkit's messaging services (Section 4).

Experiments with MACEDON have been performed on an Internet emulator ModelNet [6]. The number of underlying computers ranged from 2 to 50 in the emulation. The maximum number of emulated nodes was only 1000 and only experiments on overlay construction (routing table stabilization) was conducted on such a scale. Compared with this, we demonstrate the emulation of 4000 nodes, which is discussed in Section 5.2.

The original length of IDs in Chord is 160 bits and 128 bits in Pastry. but

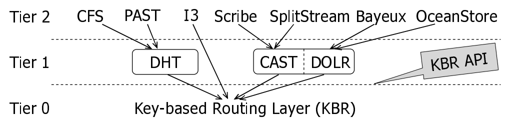


Fig. 1. Key-based routing (KBR) (Figure 1 from Dabek et al.[1])

both are 32 bits in MACEDON. The integer type `int` the dedicated language provides is 32 bits and the shortened ID length might be a natural consequence of this. We describe an algorithm in a general-purpose language (Java) for Overlay Weaver and the expressive power of the language is sufficient to represent any algorithm.

Mace [5] is a successive project following MACEDON. The ID length is 160 bits in Pastry, and this is not shortened.

An algorithm description for MACEDON or Mace is dedicated to one of routing styles, iterative and recursive. In opposition to them, a single description for Overlay Weaver can work in the both styles.

There are libraries that implement structured overlays, such as Bamboo [7,8], Chimera and Tapestry [9,10], Khashmir [11], FreePastry [12], SharkyPy [13], and OPeN [14] that are all available. However, all the libraries support a single algorithm and we do not have a choice.

P2psim [15] is a simulator for peer-to-peer protocols. It provides multiple algorithm implementations such as Chord, Accordion, Koorde, Kelips, Tapestry, and Kademlia. They are all DHT, which are structured overlays. In the Accordion proposal [16], Li et al. emulated 3000 nodes with p2psim to compare the proposed algorithm with Chord.

Simulation has the advantage being reproducible. Emulation of a concurrent system, which is what Overlay Weaver does, is non-deterministic and its results are not constant. Algorithm implementations for p2psim on the other hand do not work on a real network. They require a relatively large amount of code because they include low-level processes such as remote procedure calls (RPC). The amount of code is several times the length of code for Overlay Weaver and MACEDON (Section 5.1).

3 Decomposition of Routing Layer

Dabek et al. called routing process on structured overlays key-based routing (KBR) [1]. Following their model, higher-level services are constructed on the KBR layer, which is tier 0. Such services include DHT, multicast, anycast and message delivery.

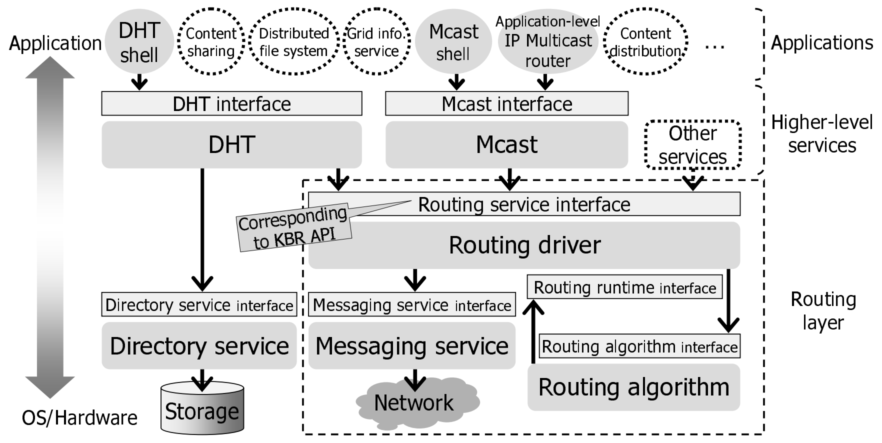


Fig. 2. Components of runtime in Overlay Weaver

Overlay Weaver also followed this model and the routing layer is separated from the DHT service and multicasting service. The model enables choices of multiple components for each layer and applications are combined with various routing algorithms without any modifications.

It is desirable for application developers to be able to choose an algorithm according to their purposes. On the other hand, algorithm developers want to implement their algorithms easily and a number of algorithms are provided and can be compared with their own algorithms. A toolkit can satisfy these requirements by facilitating the implementation of algorithms.

However, the routing layer is monolithic even with the layered abstractions. Its implementation requires a fair amount of work. Routing algorithm is not the only part of the routing layer, which has to perform other jobs including communication. Furthermore, we found that the routing layer includes other part than a routing algorithm. It is routing process, which is common to various known routing algorithms. it can be decoupled from a routing algorithm.

To facilitate the implementation of algorithms, we factored out the routing process from the routing layer by designing a programming interface between the common routing process and routing algorithms. Overlay Weaver provides implementations of the common routing process and an algorithm developer does not have to implement it. As a result, we could describe well-known algorithms just in hundreds of lines of code (Section 5.1).

Figure 2 illustrates the components for organizing runtime in the toolkit. In the figure, the routing driver and its subcontractors, the routing algorithm and messaging service, correspond to the routing layer in Dabek et al.'s proposal (Figure 1) [1]. The *routing driver* is a component that conducts the common routing process. The routing driver extracts routing information from the routing algorithm to perform routing.

The toolkit can provide multiple implementations of common routing processes due to decomposition. Two components are available and they are interchangeable without any modifications to the implementations of algorithms. They are different in their routing style. One performs iterative routing and the other does recursive routing (Section 3.4).

3.1 Interface of Routing Algorithm

This section describes the interface for the routing algorithm exposed to the routing driver. The following descriptions are in Java style because the toolkit has been implemented in Java.

```
RoutingContext initialRoutingContext(
ID target)
```

This method returns the initial value of routing context, which is transferred between nodes during routing. For example, in case of Koorde, the routing context consists of i (the ID of imaginary node) and $kshift$ in Figure 3 in Kaashoek et al. [17]. Here `target` is the target ID for routing.

```
IDAddressPair[] closestNodes(ID target,
int maxNumber, RoutingContext context)
```

This method returns a sorted array of nodes whose IDs are closest to the specified ID `target`. Note that the definition of closeness varies according to each algorithm. The maximum number of returned nodes is `maxNumber`. The routing driver treats the returned nodes as candidates for the next hop in order. One of the returned nodes may be a node executing this method itself.

An argument `context` is `null` for an algorithm which does not need the routing context. An implementation of an algorithm can return a context by attaching it to a candidate for the next hop.

Here `ID` is a type, which represents a key for routing and the ID of a node. An instance of `IDAddressPair` is a pair made up of an ID and a communication address. With UDP and TCP implementations of the messaging service, an address is a pair made up of an IP address and a port number.

```
IDAddressPair[] adjustRoot(ID target)
```

This method is invoked in the last stage of routing on a node whose ID is closest to the target ID for routing. An argument `target` is the target ID. This method returns candidates for the root node, which is the goal of routing. Where the node itself is the root, it returns `null` or an empty array.

`void join(IDAddressPair[] route)`

When a node tries to join an overlay, the routing driver conducts routing using the ID of the joining node as the target ID. This method is invoked on the joining node after routing. The argument `route` is the resulting route from the joining node to the root node (before routing).

```
void join(IDAddressPair joiningNode,
IDAddressPair lastHop, boolean isRootNode)
```

This method is invoked on every node on the resulting route while routing to a joining node. The argument `joiningNode` is the joining node and `lastHop` is the previous node on the route. True is given as the argument `isRootNode` if the root node executes this method, otherwise false is given.

`void touch(IDAddressPair from)`

This method is called when a node, on which the algorithm implementation is running, receives a message from `from`.

`void forget(IDAddressPair node)`

This method instructs a node to exclude the node `node` from its routing table. A node executes this method when it fails to communicate with `node` a certain number of times sequentially.

`BigInteger distance(ID to, ID from)`

It returns the distance between two specified IDs. Here `BigInteger` is a type that represents multiple-precision integers.

This method is not part of the routing algorithm interface. It is not called by the routing driver and is only used internally in algorithm implementations even though it has been commonly believed that this function affects the character of a routing algorithm. Practically, there are a certain number of algorithms which implement this method, but an implementation of a routing algorithm does not have to expose this method to the routing driver.

3.2 Interface Implementation by Each Algorithm

Overlay Weaver provides implementations of well-known routing algorithms, Chord, Pastry, Tapestry, and Kademlia. Table 1 lists which method for the routing algorithm interface is implemented by each algorithm. The two `join`s in the table correspond to the `join`s in Section 3.1 in order. The core of the

Table 1
Correspondence between methods specified by Routing Algorithm interface and routing algorithms

	Chord	Pastry	Tapestry	Kademlia
`closestNodes`	×	×	×	×
`adjustRoot`	× ‡1			
`join`(1)	×			
`join`(2)		×	×	
`touch`		×	×	× ‡3
`forget`	×	×	×	×
`distance`	×	×	‡2	×

interface is `closestNodes`. The routing driver requires all algorithm implementations to provide it. It tends to have more code than the other methods because of its importance. In case of Kademlia implementation, its logic (`Kademlia.java`) consists of 171 lines of code and `closestNodes` has 69 lines, which is over 1/3 of the logic.

Chord implements `adjustRoot` (‡1). With Pastry, Tapestry, and Kademlia, the routing driver can reach the root by following the closest nodes on nodes because the closest node is the root node. However, in Chord, the closest node's successor is the root node. Because of this, adjustment is required after the closest node is found. The `adjustRoot` does the adjustment.

Tapestry does not use the distance between IDs for routing. It determines the next hop according to its own procedure. Tapestry is not required to implement `distance` because of this (‡2). Pastry does not use the distance in a stage of routing involving the routing table either. However, Pastry uses `distance` to sort nodes in the leaf set which is used in the last stage of routing.

When `touch` is called, an implementation of the algorithm adds the specified sending node to its routing table. Although this behavior is peculiar to Kademlia in the four implemented algorithms (‡3), the Pastry and Tapestry implementations this toolkit provides follow this behavior.

3.3 Algorithm-dependent Process

A node on an overlay generally has to construct and maintain its own routing table, which is required to perform routing in a decentralized and autonomous way. Certain algorithms send and receive messages to construct or maintain their routing tables.

Table 2
Routing Driver's capabilities utilized by routing algorithms

	Chord	Pastry	Tapestry	Kademlia
(1) Confirmation of other nodes' aliveness		×	×	×
(2) Sending/receiving message	×	×	×	‡1
(3) Routing to root node	× ‡2			
(4) Routing to closest node	× ‡3			

The toolkit itself does not communicate this way because it is heavily dependent on each algorithm. On the algorithm-side, the routing algorithm has to perform this by itself. The routing driver provides part of functions that the routing driver and messaging service have for the routing algorithm. The functions that are provided are as follows. They have been utilized via the routing runtime interface in Figure 2.

(1) Confirmation of other nodes' aliveness by sending/receiving a PING and ACK
(2) Sending/receiving an arbitrary message
(3) Routing to the root node of the specified ID
(4) Routing to the node which is closest to the specified ID

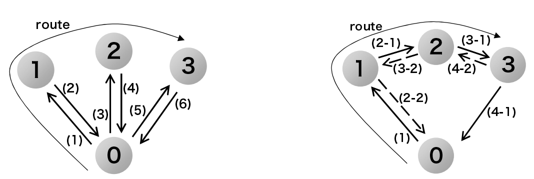
An implementation of the routing algorithm can confirm other nodes' aliveness just by invoking the function provided by the routing driver.

Table 2 lists which function is utilized by each algorithm.

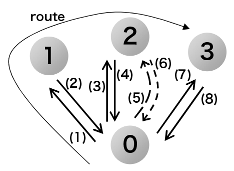
One of characteristics of Kademlia [18] is that it does not perform any communication to construct or maintain a routing table. A node maintains its routing table by observing the communication for routing. The implementation of Kademlia in the toolkit does not send or receive algorithm-dependent messages (‡1) and invocations of `touch` cause the routing table to be maintained.

Chord implementation needs routing to the root node to construct its finger table (‡2).

In Section 3.2, we mentioned that the closest node is not the root node in Chord. Overlay Weaver provides two implementations of Chord (Section 4.5), and one of these, which completes the routing table when joining an overlay, requires routing to the closest node (‡3) in addition to routing to the root node.



a. Iterative routing b. Recursive routing



c. Iterative routing (by Overlay Weaver)

Fig. 3. Routing styles

3.4 Routing Process

Overlay Weaver provides two implementations of the routing driver, which conducts common routing processes. The two perform iterative routing and recursive routing [7]. A user of the toolkit can choose either one at runtime.

Figure 3 illustrates communication for both styles of routing. The circle and arrow including the dotted arrow stand for a node and a message being passed. The numbers in parentheses indicate the order the message is passed in. As the figure shows, the node that initiated routing in the iterative style issues queries to each node along the route. A query is forwarded along the route in the recursive style.

The dotted lines in Figure 3 stand for messages being passed that are not necessarily performed in usual iterative/recursive routing. The (5) and (6) in Figure 3 c indicate a query and a response for `adjustRoot` (Section 3.1). In the algorithms the toolkit provides, only Chord requires `adjustRoot` to be implemented and the other algorithms do not cause (5) or (6).

The dotted lines in Figure 3 b are responses to routing queries and they are just to confirm queries have been delivered. The routing driver sends and receives them to carry out routing with unreliable transport like UDP. However, a node on the route gives priority to the forwarding of a routing query over confirmation. Because of this, recursive routing is expected to have a shorter routing time than iterative routing [19] where communication latency dominates.

The number of messages passed in single routing is as follows: $2n$ in iterative routing and $2n + 1$ in recursive routing except algorithms that require `adjustRoot` like Chord. Here n is the number of hops. This is $2(n + 1)$ in the case of iterative routing with algorithms that need `adjustRoot`. Note that the TCP implementation of messaging service (Section 4) sends a response via the same connection for a query corresponding to the response. It also does connection pooling. Due to this, establishment of connections, which takes a relatively long time, is carried out just $n + 1$ times at maximum.

Routing with Kademlia and Concurrent Queries In routing with Chord, Pastry, and Tapestry, a node determines the next hop to be a node whose ID is closest to the target ID to the best of its knowledge. Kademlia performs routing differently. A node in Kademlia maintains multiple k nodes whose IDs are closest to the target ID during routing and it sends queries to all nodes it keeps [18].

A routing driver implementation performing iterative routing can issue multiple queries concurrently. Queries before `adjustRoot` such as (1) and (3) in Figure 3 c can be issued concurrently. In Kademlia, 3 queries run concurrently [18]. We can specify this concurrency parameter at runtime.

The iterative routing implementation can perform Kademlia-style routing and concurrent queries. The reason it has not been implemented in the recursive routing implementation is the difficulty in avoiding wasteful duplicated queries being sent to a node. In iterative routing, it is normal for the initiating node to maintain a set of nodes that have been queried and it is easy to avoid duplicated queries. However, in recursive routing, concurrent queries are performed by multiple distributed nodes and it is not a trivial matter to share such sets between nodes. Nevertheless, concurrent queries are possible with circulation avoiding techniques common to flooding and gossip, in which a queried node replies that it has dealt with a similar query. These techniques permit many duplicated queries.

An existing problem is how to perform concurrent queries efficiently in recursive style.

4 Toolkit Structure

This section describes runtime and tools being organized into Overlay Weaver, an overlay construction toolkit. We explain how these support algorithm design, testing, and comparisons.

The toolkit consists of the components outlined in Figure 2 and the following tools.

- Distributed environment emulator
- Scenario generator
- Message counter
- Messaging visualizer

There are multiple components for parts of runtime other than the routing driver and routing algorithm and users can choose one of these. As an example, the toolkit provides the following four components.

- UDP implementation
- TCP implementation
- Emulating implementation, which performs inter-thread communication (for emulator)
- Emulating implementation, which performs inter-thread communication over a network (for distributed emulator)

The UDP component implements the UDP hole punching technique, which enables a node to join an overlay from the inside of a NA(P)T router if the router is compatible with the technique.

Three components are provided for the directory service. These offer the function of a local hash table, which the DHT service utilizes.

- An implementation based on the Berkeley DB Java Edition [20]
- On-memory implementation using a hash table in the standard class library of Java
- On-memory implementation, which saves its data in persistent storage periodically

4.1 Higher-Level Services and Sample Applications

Higher-level services are implemented on the routing layer described in Section 3. Applications are expected to use them rather than using the routing layer directly (Figure 2). The toolkit provides the Mcast in addition to DHT as higher-level services. It performs a multicast on an overlay. An application joins and leaves a multicast group, which is identified by an ID, and multicasts messages to all nodes in the group. An application can be directly aware of the distribution tree for a multicast. One sample application, the IPv4 multicast router, uses this feature.

The toolkit provides sample applications, i.e., the DHT shell and Mcast shell, which use DHT and Mcast services. We can invoke these services and control them via a character terminal and network. An algorithm developer can use these shells and the emulator (Section 4.2) together to test a new algo-

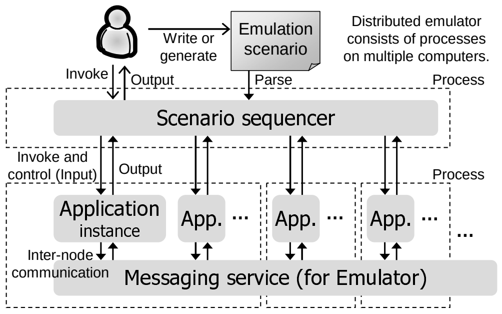


Fig. 4. Structure of emulator

```
# invoke the 1st instance
class ow.tool.dhtshell.Main
schedule 0 invoke
# invoke 999 instances with an argument "emu0" every 500 msec
arg emu0
schedule 500,500,999 invoke
# send a put request to a node invoked 124th 510 sec from the start
schedule 510000 control 123 put a_key a_value bar 300
# send a get request to a node invoked 235th 515 sec from the start
schedule 515000 control 234 get a_key
```

Fig. 5. An example of a scenario

rithm and compare it with existing ones. Section 5 demonstrates that such comparison is possible.

4.2 Emulator

The toolkit includes the Distributed environment emulator, which can host thousands of nodes on a single computer. Algorithm developers can improve new algorithms and their implementations rapidly by testing them iteratively on the emulator. The developers can also make large-scale and fair comparisons of new and existing algorithms by running the same emulation scenario for different algorithms. Additionally, implemented algorithms work on a real network in addition to the emulator. The toolkit gives the results of algorithm research to applications directly.

Figure 4 illustrates the structure of the emulator. There are two ways to run the emulator, running it on a single computer and combining multiple computers to run it. In both cases, the emulator reads and executes the same emulation scenario. The emulator assigns a virtual hostname to an invoked application instance. The messaging service deals with communication between virtual hosts. It forwards a message from a virtual host to another one with its best effort. In case that multiple computers host a single emulator, a message is encapsulated and forwarded via UDP or TCP between the computers. In

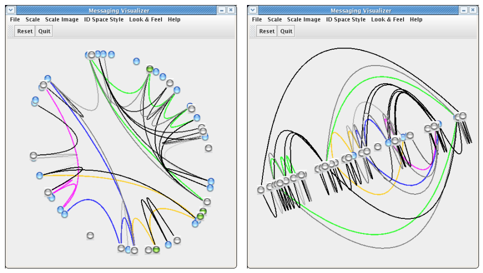

Messaging Visualizer
File Scale Scale Image ID Space Style Look & Feel Help
Reset Quit
Messaging Visualizer
File Scale Scale Image ID Space Style Look & Feel Help
Reset Quit

Fig. 6. Messaging visualizer

both cases, output from applications is collected and printed out to the user's console, making it easy to analyze output after emulation.

Figure 5 shows an example of a scenario. A scenario file first has instructions to invoke application instances, which include a class name and arguments. The scenario sequencer reads the instructions and invokes application instances as threads. The scenario can also have instructions executed by the invoked instances. An instruction consists of strings given to the instance, an identifier of the instance, and time for the instruction to be given. The instruction is given via standard input and it can take any form which the application accepts. For example, the DHT shell (Section 4.1) accepts instructions to print out a routing table, and suspend and resume a node, in addition to putting, getting and removing a key-value pair.

We can produce a scenario in various ways as though writing it by hand and generating it by computer. It is a promising way to convert a real-world trace into a scenario for the emulator. The toolkit includes the scenario generator and it is also possible to generate simple scenarios driving DHT shells by it.

4.3 Message Counter

Implementations of the messaging service can report all communications via the network. The message counter is a tool that receives reports and counts them for all message types.

Logging and analyzing the log is generally a powerful method of observing communication after an execution finished. The message counter enables such statistics to be collected in the execution time.

4.4 Messaging Visualizer

The messaging visualizer is a tool to visualize nodes and communication between them at the time this happens. Figure 6 shows screenshots of the tool.

It draws a node at a position according to its ID on a circle, a line, or various shapes. The arcs drawn between nodes stand for messages between them. The tool also draws distribution trees constructed for a multicast.

This tool supports an intuitive understanding of algorithm behavior. Not only does it facilitate tests of algorithm implementations but it is also useful for demonstrating algorithms.

The visualizer collects communication reports with the same facilities as the message counter. This enables the visualizer to work both on an emulator and a real network, but it involves the same number of messages sent to the visualizer, double the number on a network.

The visualizer imposes a burden on an emulator by visualization in addition to doubling the number of messages. This results in reducing the maximum number of emulated nodes. We confirmed that up to 300 nodes could be visualized on a 1.7 GHz Pentium M processor running on Linux 2.6.15 on VMware hosted by Windows XP.

4.5 Algorithm Implementations and Parameters

Overlay Weaver provides implementations of well-known structured overlay algorithms, i.e., Chord, Pastry, Tapestry, and Kademlia. This section describes the details and parameters used in Section 5.

Chord The toolkit implements not only a normal algorithm with the stabilizing process but also the one in Figure 6 in Stoica et al. [21]. In the latter algorithm, a node completes its routing table when joining an overlay. We call the latter “Chord-Fig6”.

The interval of calls for the procedure `stabilize` is first 10 seconds. This is being expanded up to 120 seconds where the routing table does not change. The interval of `fix_fingers` is a value equals to or between 5 seconds and 600 seconds according to how the finger table is accomplished.

Pastry The length of a digit in an ID is set to 4 bits ($b = 4$) even though it is configurable at runtime. The size of a leaf set is set to 8 ($|L| = 8$).

The implementation performs “periodic routing table maintenance” as described in Section 2.2 of Castro et al. [22] even though it is different from the original Pastry proposal [23]. The interval for maintenance is set to 60 seconds.

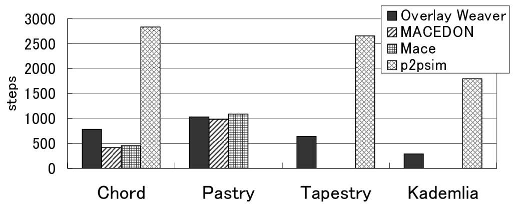


Fig. 7. Lines of code for each routing algorithm

Tapestry The implementation treats an ID as a sequence of hexadecimal digits ($\beta = 16$) following Zhao et al. [9]. This is the same parameter as in Pastry.

Current implementation has neither backup links ($c = 1$) nor backpointers.

A node reactively detects failure in other nodes by failure in communication for routing. Note that a node sends periodic queries to detect failure if it follows Zhao et al.'s algorithm [9] completely.

Kademlia The length of the k-bucket is set to 20 following the Kademlia paper [18] ($k = 20$). The number of nodes maintained during routing, whose IDs are closest to the target ID is also 20 according to the paper. However, the number of closest nodes returned by a queried node is set to 5, not 20.

The concurrency parameter for concurrent queries (Section 3.4) is set to 3 according to the paper ($\alpha = 3$).

5 Evaluation

One of our goals was to give results of algorithm research to applications directly. Overlay Weaver facilitates algorithm design and implementation to achieve that goal. In this section, we evaluate the toolkit in terms of the following four aspects that reflect this goal.

- Easy implementation of structured overlay algorithms
- Confirmation of behavior of algorithms by large-scale emulation
- Fair comparison of algorithms
- Work on real network

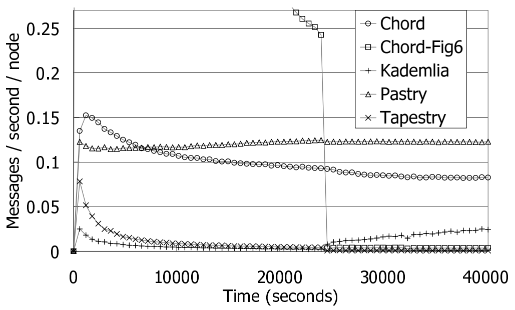


Fig. 8. Number of messages per second per node being passed between emulated 4000 nodes (iterative routing)

5.1 Amount of Code Implementing Algorithms

Figure 7 shows the lines of code required to implement routing algorithms compared with MACEDON 1.2.1 [3,4], Mace 0.9 [5] and p2psim 0.3 [15]. The version of Overlay Weaver is 0.6 here. Here the shown numbers are lines of code except blank lines and comments.

We could implement all algorithms just in hundreds of lines of code with the toolkit. Pastry needed most code but this was only about 1000 lines even though it contains 148 lines of code to output string and HTML representation of a routing table. This resulted from the decomposition of the routing layer (Section 3).

Compared with MACEDON, which provided a domain-specific language, Chord implementation with Overlay Weaver required less than 2 times as much code as MACEDON. Overlay Weaver needed almost same amount of code as MACEDON to implement Pastry. P2psim involved relatively more code because algorithm implementation itself involved low-level processes such as communication and RPC. Note that the Chord implementation for MACEDON is a subset of what described in Stoica et al. [21]. The size of an ID is 32 bit, not 160 bit, and the number of successors which a node can keep is just one.

5.2 Emulation of 4000 Virtual Nodes

This section presents the results of emulating 4000 nodes to demonstrate that large-scale emulation is possible. Note that the purpose of the following experiments was not to compare algorithms but rather demonstrate that the toolkit enables evaluation and a fair comparison of algorithms.

We ran the following experiments on a computer with a 3.4 GHz Pentium 4 processor (Prescott), 1 GB of memory, and Linux 2.6.15. The Java runtime

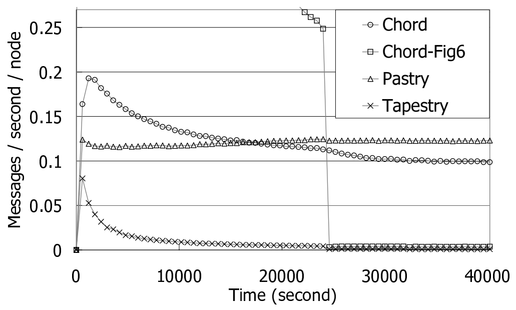


Fig. 9. Number of messages per second per node being passed between emulated 4000 nodes (recursive routing)

we used here was the Java 2 Standard Edition (J2SE) 5.0 Update 6.

Figures 8 and 9 are proofs that Overlay Weaver supports quantitative comparison of algorithms by large-scale emulation. They plot the average number of messages per second per node in emulations of 4000 nodes. The numbers are the average for 10 minutes. The emulation scenario was as follows. After 4000 nodes had joined an overlay every 6 seconds, all nodes waited 100 seconds. A node put a value on a DHT every 2 seconds 4000 times. All nodes waited 100 seconds. A node got a value from a DHT every 2 seconds 4000 times. We determined which node does the puts and gets randomly when generating a scenario with the scenario generator.

We can generally evaluate and compare algorithms by logging and analyzing the logged results. This method is sufficient for metrics that a node can calculate alone, such as the amount of data a DHT service is keeping and how a routing table converges. Logging with time stamps is effective to investigate metrics that involve multiple nodes. We can collect the log and extract intended results. Otherwise, we can utilize the message counter (Section 4.3) for this purpose. The numbers in Figure 8 and 9 were counted by the message counter.

Figure 8 and 9 plot behavior of the implemented algorithms, which was as follows.

- Chord and Pastry sent more messages than Kademlia and Tapestry because the two sent messages periodically to maintain overlays.
- Chord-Fig6 sent a large number of messages to complete routing tables when joining an overlay.
- Kademlia sent and received PING and ACK messages to determine whether to keep or to replace a node in a routing table. Kademlia sent more messages even when performing put/get because every transmission of messages could trigger off this confirmation.

We demonstrated that it was possible to emulate 4000 nodes with a common

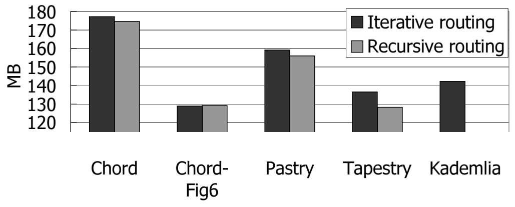


Fig. 10. Memory consumption by emulator while running 1000 nodes

PC to investigate the behavior of algorithms. This section also outlined methods of collecting data for a fair comparison of algorithms and presented results produced by the message counter.

Scalability of Emulator The following are factors that possibly restrict the number of nodes an emulator can host.

- Number of threads the OS kernel and command-line shell allow
- Size of virtual address space
- Amount of physical memory
- Processing throughput of computer

These parameters and which ones will become bottlenecks depend on specific computers.

Linux 2.6.15, we used here, could manage up to 32768 threads by default on a single computer when running on a 32 bit processor. The number of threads a user process can create is a half of 32768. Because a DHT shell requires up to 4 threads a node, this version of Linux could emulate about 4000 instances of it.

As the number of emulated nodes is increased, the virtual address space assigned to the stack for each thread can starve. For example, where 256 KB of memory is assigned to a thread, only 16384 threads can be created on a 32 bit processor because it provides up to 4 GB of virtual address space. In such cases, we can invoke more threads by adjusting the amount of memory for the stack.

Figure 10 plots the amount of memory an emulator consumes when it runs 1000 nodes of the DHT shell. The numbers are resident set size (RSS) for the emulator process. The RSS shows the amount of memory consumed by a process because the computer we used did not have paging space on the disk. The 1000 nodes consume only 177 MB of memory with Chord, which required the most memory in the algorithms we investigated. Because of it, we can estimate that a computer with 1 GB of free memory can emulate about 5000 nodes without other constraints such as number of threads.

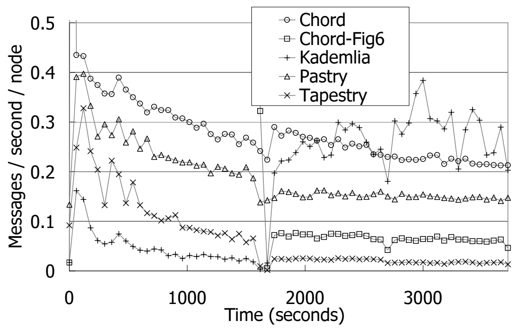


Fig. 11. Number of messages per second per node passed between real 197 computers (iterative routing)

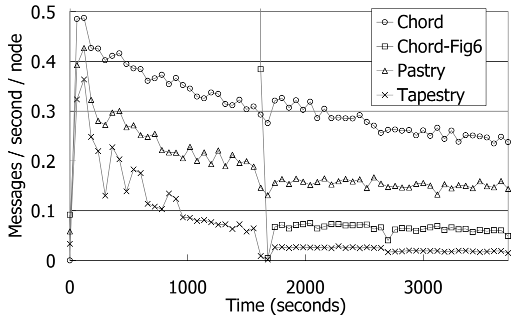


Fig. 12. Number of messages per second per node passed between real 197 computers (recursive routing)

5.3 Test on Real Network

In this section, we also demonstrate that Overlay Weaver can work on a real network. We conducted experiments with 197 computers, which consisted of 196 PCs with 3.06 GHz Xeon processors in addition to the PC described in Section 5.2. All PCs had a Gigabit Ethernet interface. The 196 PCs were part of the AIST Super Cluster (ASC) and connected to a single Ethernet switch. The bandwidth between the 196 PCs and the remaining PC was also 1 Gbps. Linux 2.6.24 ran on the 196 PCs and the Java runtime was J2SE 5.0 Update 6. We invoked a DHT shell on each computer and controlled them via a network.

Figures 11 and 12 plot the average number of messages per second per node. The numbers are the average for a minute. We counted the numbers using the message counter. The control scenario was as follows. After 197 nodes had joined an overlay every 8 seconds, all nodes waited 100 seconds. A node put a value on a DHT every 2 seconds 500 times. All nodes waited 30 seconds. A node got a value from a DHT every 2 seconds 500 times. We determined which node does the puts and gets randomly similar to Section 5.2.

This experiment demonstrated that the toolkit works on a real network with about 200 computers.

6 Conclusion and Future Work

This paper described the design of the toolkit called Overlay Weaver, which we have been developing as the groundwork for future research on overlay algorithms. We demonstrated that it is possible to implement various well-known routing algorithms just in hundreds of lines of code with the toolkit, and we investigated their behavior by emulating 4000 nodes on a single computer. It was also demonstrated that about 200 computers on a real network can construct an overlay.

We described a programing interface that decouples a routing algorithm from the common routing process. This decomposition facilitated algorithm implementation and enabled multiple implementations of the common routing process.

We are promoting use and application of the toolkit by third parties. The promotion is getting results including algorithm and application implementations. A party has implemented Symphony [24] and the other reported their implementation of EpiChord [25] with the toolkit. Applications reported include RDF document search and statistics of web accesses.

Our future work includes the following.

- Implement other algorithms to reinforce the adequacy of the design of the routing algorithm interface. We could implement Koorde [17] after implementing the algorithms demonstrated in the paper.
- Conduct larger scale experiments. For example, a 64 bit computer with a memory more than 4 GB and a distributed emulator with multiple computers will enable it.
- Investigate how the toolkit supports unstructured overlays.

References

[1] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz, Ion Stoica, *Towards a Common API for Structured Peer-to-Peer Overlays*, Proc. IPTPS'03 (2003).

[2] *Overlay Weaver: An Overlay Construction Toolkit*, http://overlayweaver.sourceforge.net/ .

[3] Adolfo Rodriguez, Charles Killian, Sooraj Bhat, Dejan Kostić, *MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks*, Proc. NSDI'04, pp.267–280 (2004).

[4] *The MACEDON project*, http://macedon.ucsd.edu/ .

[5] *The Mace project*, http://mace.ucsd.edu/ .

[6] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase and David Becker, *Scalability and Accuracy in a Large-Scale Network Emulator*, Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI '02) (2002).

[7] Sean Rhea and Dennis Geels and Timothy Roscoe and John Kubiatowicz, *Handling Churn in a DHT*, Proc. USENIX '04 (2004).

[8] *The Bamboo Distributed Hash Table*, http://www.bamboo-dht.org/ .

[9] Ben Y. Zhao, others, *Tapestry: A Resilient Global-Scale Overlay for Service Deployment*, Journal on selected area in communications, Vol.22, No.1, pp.41–53 (2004).

[10] *Chimera and Tapestry*, http://p2p.cs.ucsb.edu/chimera/ .

[11] *Khashmir*, http://khashmir.sourceforge.net/ .

[12] *FreePastry*, http://freepastry.org/FreePastry/ .

[13] *SharkyPy*, http://www.heim-d.uni-sb.de/ heikowu/SharkyPy/ .

[14] *OPeN library*, http://p2p.cs.mu.oz.au/software/OPeN .

[15] *p2psim: a simulator for peer-to-peer protocols*, http://pdos.csail.mit.edu/p2psim/ .

[16] Jinyang Li, Jeremy Stribling, Robert Morris, M. Frans Kaashoek, *Bandwidth-efficient management of DHT routing tables*, Proc. NSDI'05 (2005).

[17] M. Frans Kaashoek and David R. Karger, *Koorde: A simple degree-optimal distributed hash table*, Proc. IPTPS'03 (2003).

[18] Petar Maymounkov and David Mazières, *Kademlia: A Peer-to-peer Information System Based on the XOR Metric*, Proc. IPTPS'02 (2002).

[19] Frank Dabek and Jinyang Li and Emil Sit and James Robertson and M. Frans Kaashoek and Robert Morris, *Designing a DHT for low latency and high throughput*, Proc. NSDI'04, pp.85–98 (2004).

[20] *Berkeley DB Java Edition*, http://www.sleepycat.com/products/bdbje.html .

[21] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan, *Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications*, Proc. ACM SIGCOMM 2001, pp.149–160 (2001).

[22] Miguel Castro, Manuel Costa, Antony Rowstron, *Debunking some myths about structured and unstructured overlays*, Proc. NSDI'05 (2005).

[23] Antony Rowstron, Peter Druschel, *Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems*, Proc. Middleware 2001 (2001).

[24] Gurmeet Singh Manku, Mayank Bawa, Prabhakar Raghavan, *Symphony: Distributed Hashing in a Small World*, Proc. USITS '03 (2003).

[25] Ben Leong, Barbara Liskov, Erik Demaine, *EpiChord: Parallelizing the Chord Lookup Algorithm with Reactive Routing State Management*, Proc. ICON 2004, Vol.1, pp.270–276 (2004).