Viceroy - on the implementation of a Peer to Peer network

Oren Dobzinski and Anat Talmy
The Hebrew University of Jerusalem, Israel
{orend,anatt}@cs.huji.ac.il

August 8, 2003

Abstract

We present the implementation of the Viceroy network, a peer-to-peer lookup network. We give a complete description of the requirements and the goals of this project, along with a full discussion of implementation details and the main algorithms in the project. In addition, we present several new routing algorithms, a brief description of their implementation, and a comparison to existing routing algorithms.

1 Introduction

This is a description of an implementation of The Viceroy network, proposed by Dahlia Malkhi, Moni Naor and David Ratajczak in [Viceroy].

The Viceroy network is a constant degree, peer-to-peer lookup network, which behaves as a DHT (Distributed Hash Table), i.e. distributes resources stably to nodes in a dynamic, distributed network. Its main purpose is to efficiently look up resources, where no central authority exists and where the network is dynamically changing. The Viceroy network is a composition of an approximate butterfly network and a connected ring of predecessor and successor links. Therefore, each server has three outgoing links to chosen long-range contacts - **up**, **left** and **right** links, and two ring connections, one to its **successor** and one to its **predecessor**. Up, left and right connections are considered the 'butterfly' links and the predecessor and successor - 'ring' links. [1]

Due to the independent structure of the network adding or removing any node can be achieved without any global coordination , and only local updates are required.

Locating resources is done in an efficient $O(log(n))$ hops, where n is the total number of nodes. Similar results have so far only been achieved in non-constant degree networks such as [Chord] and [Tapestry].

[1] There exist two additional level links in the more complex version of the network, which is not yet implemented, pointing to the next and previous peer in a given peer's level.

We supply a full implementation of the Viceroy network, and a demonstration is available at `http://www.cs.huji.ac.il/~anatt/viceroy/viceroy.html`. The details of the implementation are relevant, as many peer-to-peer projects, such as the implementation of the [DH] network are currently based on our implementation. In addition, we suggest several routing algorithms in the context of the Viceroy network and we analyze their performance with respect to the path length.

Our implementation is highly modular and easily extensible. It separates logic and implementation so that any underlying communication layer can be easily plugged into the code, and any algorithm can be replaced without changing a single line of code. We achieved this modularity by using the Remote Proxy pattern [Gamma95], and by defining common interfaces instead of using specific algorithms, using them as in the Strategy pattern [Gamma95]. We hope that this modularity will facilitate future extensions and usages of our implementation as it has already helped existing ones.

2 The Viceroy Project

The viceroy project is divided into 4 different parts, which were implemented by different students. In this paper we deal with the first two parts only.

Design and inter-level interfaces design: Implemented by Oren Dobzinski and Anat Talmy.

The main lookup algorithms: Also known as the logic part, was implemented by Oren Dobzinski and Anat Talmy.

The network level: Implemented by Danny Bickson.

The data transfer client: Implemented by Roy Werber.

3 Design Goals

When we thought of the overall design of the system, and implementation of the logic level, we kept the following goals in mind:

Correctness: No real analysis can be done with an incorrect or an inaccurate system. Therefore, we included inherent correctness checks, which verify all our assumptions during coding. Those internal checks prevent the system from entering incorrect states, that otherwise may not have been detected. In addition, we included a set of unit tests to detect failures that were not detected earlier, and to check the main modules of the system.

Modularity: As several developers worked on this project together, it had to be modular, without dependencies that are not well defined. Therefore, we have a large set of interfaces that define the interaction between levels.

Moreover, the Event-Driven model decouples implementation of the levels from their interfaces and defines all the issues regarding the interaction between them.

Flexibility: We wanted to be able to easily change the implementation of any algorithm. This goal was achieved by using the Strategy pattern for the lookup algorithms, and by enhancing the modularity of the system by using interfaces instead of concrete classes in all references to them, such as parameters, data members and return values.

Extensibility: We wanted to enable advanced research on the Viceroy network. Therefore, we provided a way to add complexity to future analysis, such as statistical analysis. We also made a separation between objects and interfaces that seemed redundant at first glance, such as the separation between *LookupPeer*, *ViceroyPeer* and *AuthorizedLookupPeer*. This was done in order to keep everything as general as possible and so achieve better extensibility.

4 Logic Implementation

The construction of the Viceroy network was implemented exactly as described in the paper [Viceroy]. This holds for all algorithms described below:

Lookup algorithm: We implemented the simple version and added four different lookup algorithms of our own (for a detailed description of all five algorithms, see section 8)

Level selection: The goal is to have a balanced network such that every level contains more or less the same number of nodes. Therefore, the independent level selection procedure of the nodes must be consistent with this property. When a peer joins the system, it performs an estimation of the number of peers currently active in the system, n. Since no global authority exists, the peer must use local data only, so it bases its estimation of n on the distance to its successor. Then, the level this peer picks is selected uniformly among $[1 \dots \log(n)]$.

ID selection: Every peer that enters the system selects an ID independently and uniformly from [0;1). The default ID length is 128-bits. If an ID clash is detected at any stage, a procedure of changing the ID is performed: the peer that detected the clash adds several precision bits to its ID until the clash is resolved.

Join and Leave algorithms: Whenever a peer joins or leaves the network, some local changes in the network occur. When a node joins the network, it is given responsibility over a stretch of possible Resources whose mappings are in the range $(predecessor.id \dots this.id]$. The joining peer must request this data from its successor, which drops those values as soon as

the data transfer is over. The reverse procedure happens when a node leaves, or when some node discovers that its successor left unexpectedly. These actions usually cause a structural change in the network, as some peers might need to find new butterfly connections, or re-select their level if they have a new successor.

5 The Packages in the Project

Our project consists of 4 packages:

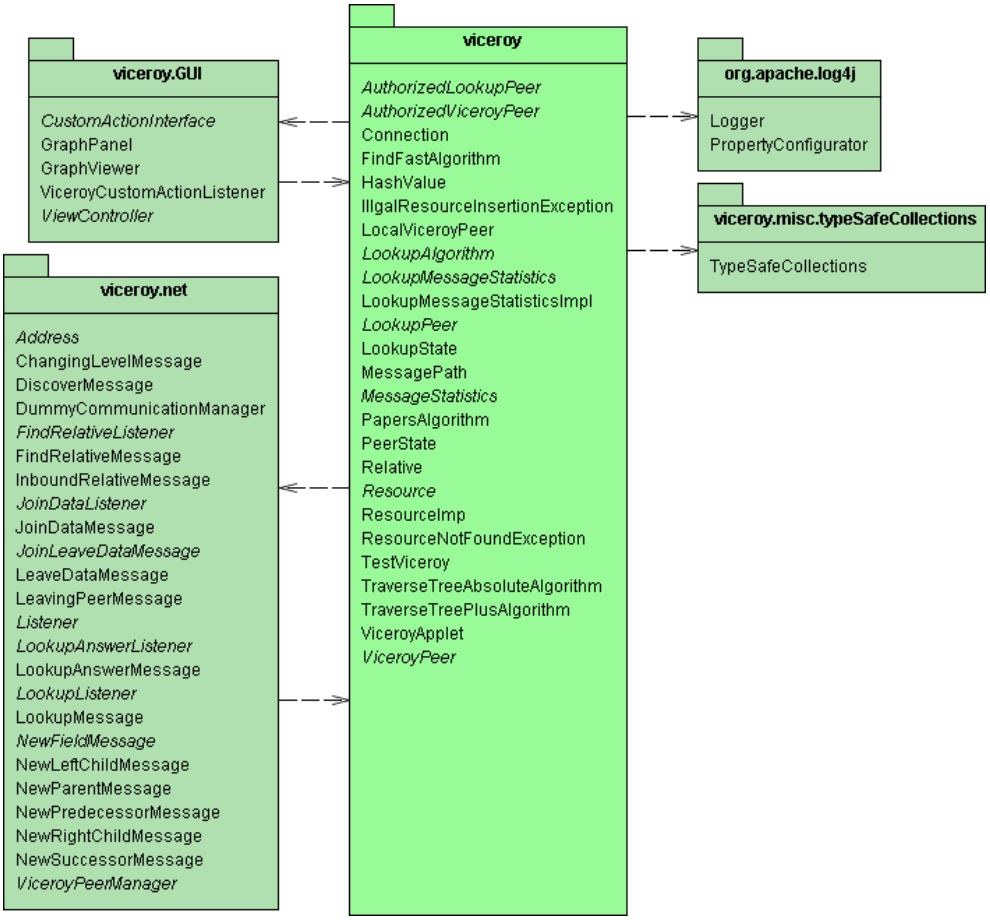

Figure 1: The Viceroy Packages

- The *viceroy* package contains the algorithms and the main entities in the Viceroy network.

- The *viceroy.net* package contains the communication infrastructure for the Viceroy network.
- The *viceroy.GUI* package contains classes and interfaces for the graphical user interface of the local viceroy network.
- The *viceroy.misc.typeSafeCollections* contains a set of utility classes, which implement the notion of type-safe Collections, which serve as the best substitute of C++ templates in Java.

5.1 The Viceroy Package

All lookup networks consist of objects who conform to the *LookupPeer* interface. We supplied this interface in order to support other network topologies, besides the Viceroy network topology. We also supplied an *AuthorizedLookupPeer* interface, which represents a *LookupPeer* that can initiate lookup requests, joins and leaves, whereas the *LookupPeer* only responds to requests. The other interfaces are Viceroy-specific, and refer to the topology of the Viceroy network in the most general way. The two topology-specific interfaces are *ViceroyPeer* and *AuthorizedViceroyPeer*. In most of the API we use *ViceroyPeer*. The Viceroy network consists of *LocalViceroyPeers*, which is an implementation of *AuthorizedViceroyPeer*. Each such peer has several connections to the Viceroy network; these links are represented as *ViceroyPeer* fields. In a distributed system, these fields will actually be *ReomteViceroyPeers*, representing a remote peer in a different computer (a stub). In a single computer system, these fields will be *LocalViceroyPeers*, however, it does not matter, as those links are viewed as *ViceroyPeers*. Therefore, running and testing our Viceroy network on any computer system would not require any change in our source code. This is actually a unique implementation of the Remote Proxy pattern [Gamma95].

The *LocalViceroyPeer* uses the *ViceroyPeerManager* as its communication server. This object is responsible for all communication matters. As we used an Interface to describe it, any specific implementation can be used, e.g., Jxta implementation, TCP/IP, RMI, or even a simple single-computer implementation.

After registering itself with the communication manager (and after receiving a real-world address), the peer does not use the *ViceroyPeerManager* directly. Instead, it calls the appropriate methods in its connection fields (successor, predecessor and such - all are *ViceroyPeers*). For example, when receiving a lookup message, the peer will do the following:

```
LocalViceroyPeer.lookup(Resource valueToLook, ViceroyPeer returnAddress):
    if (this.level ≠ 1)
        this.successor.lookup(valueToLook, returnAddress);
    ...
```

In a distributed system, the successor is actually a *RemoteViceroyPeer*, so its lookup message will simply create a new *LookupMessage*, and send it (using the *ViceroyPeerManager*) to the actual remote peer. The *LocalViceroyPeer*

registers a set of Listeners (specific to each message), which define the behavior of dealing with incoming messages. Each incoming message is treated as an event, which activates the corresponding *XXXReceived()* methods in the *LocalViceroyPeer*.

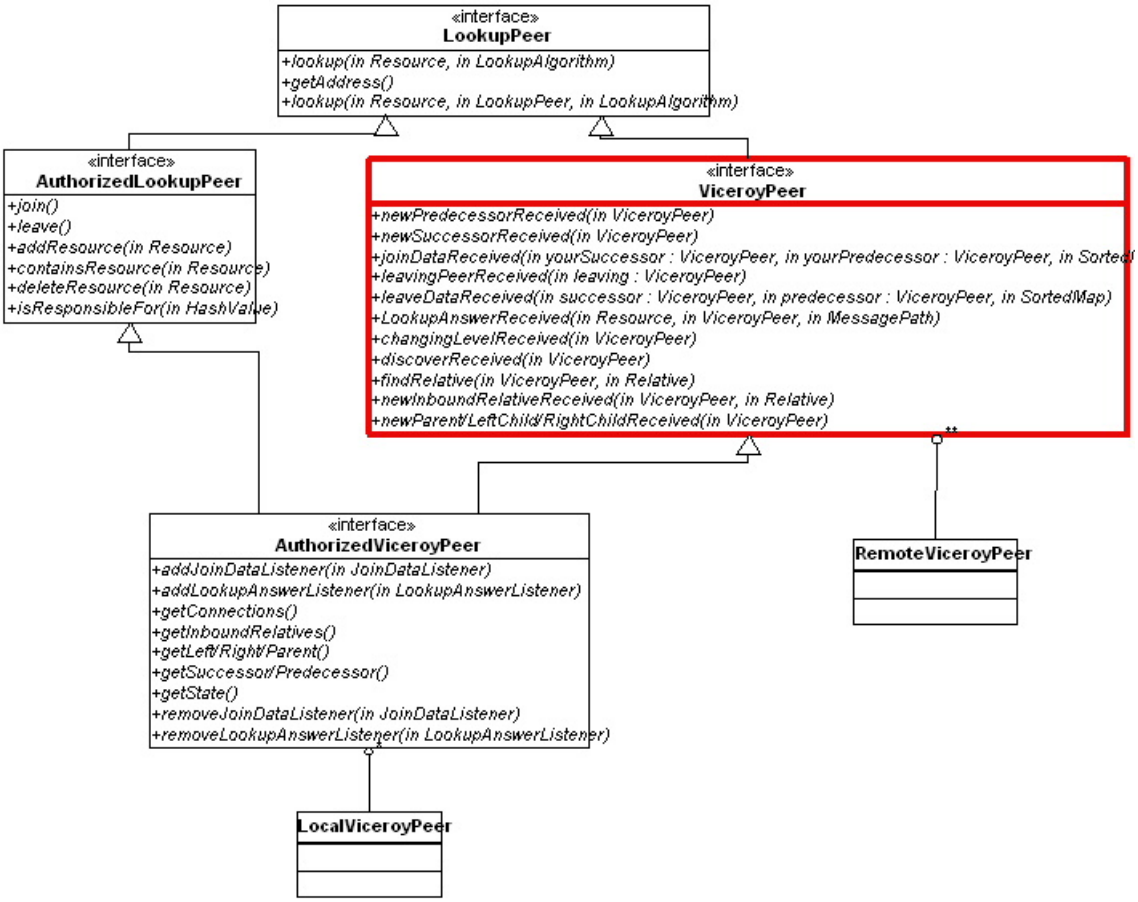

Figure 2: Main Classes in viceroy package

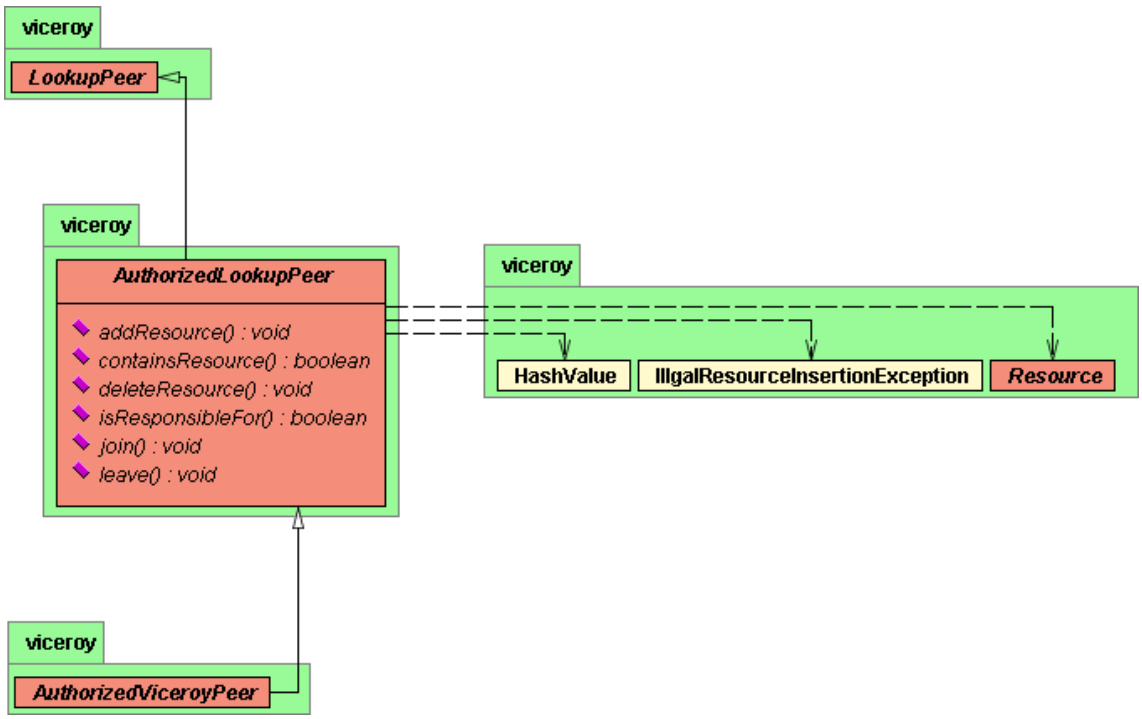

Figure 3: *AuthorizedLookupPeer* UML Diagram

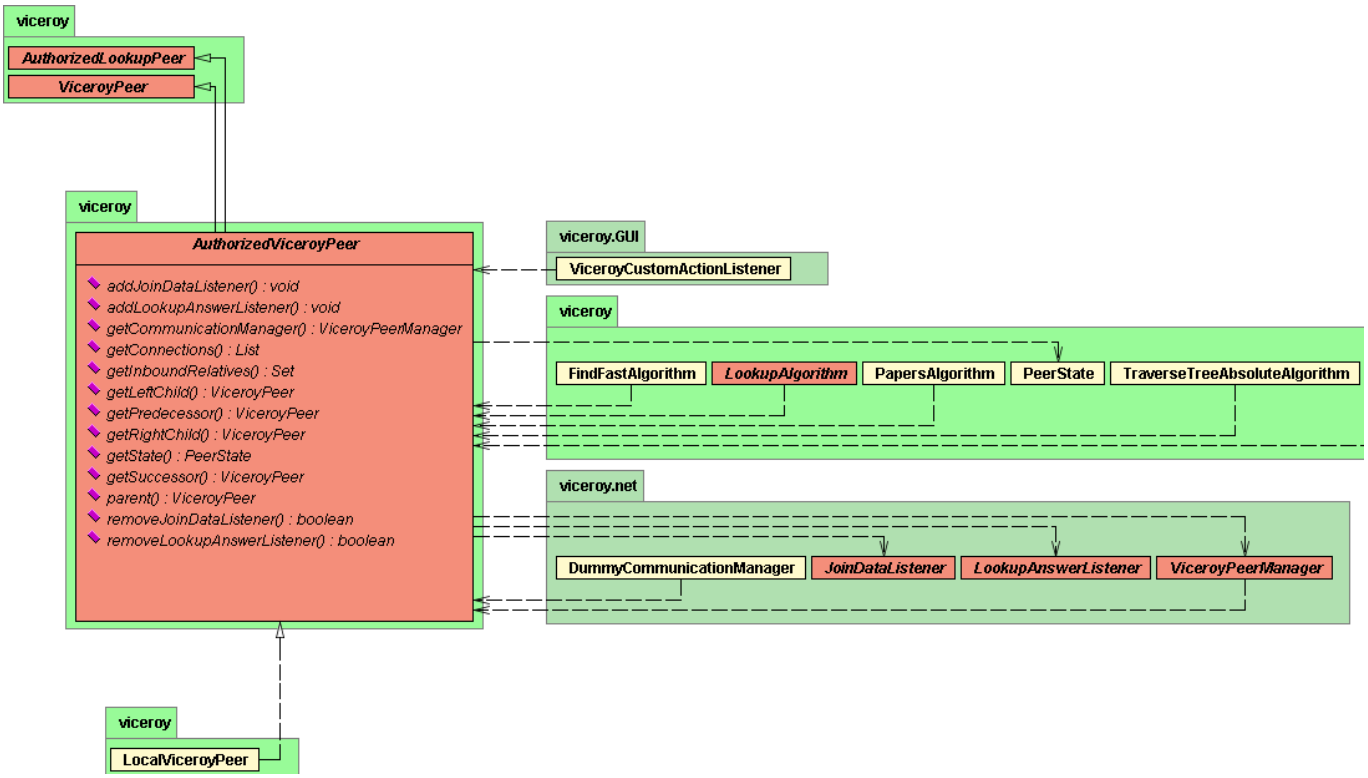

Figure 4: *AuthorizedViceroyPeer* UML Diagram

5.2 The Viceroy.net Package

We defined a *Listener* interface, and a set of message-specific interface listeners. The implementing classes are actually private inner classes inside *LocalViceroyPeer*. We also defined a *Message* interface, and implemented all relevant messages classes. We haven't implemented two interfaces: *ViceroyPeerManager* and *Address*. The implementer of those interfaces can do it using Jxta, TCP/IP, RMI etc.

5.3 The viceroy.misc.typeSafeCollections package

We extended the implementation of Piet Jonas in [Secure type-safe collections], who supplied a better type-safety than Java's standard Collection Framework. Our classes implement the standard interfaces supplied with the Java SDK (*List*, *Set*, *Map* etc.) and partly compensate the lack of a template mechanism in Java 1.4.

5.4 The viceroy.GUI package

This package consists of an applet that gives the user an opportunity to get a graphical view of the Viceroy network; It is a fully-functional system that can perform all available operations on the peers, including join, leave and lookup. We provide several views of the system, each demonstrating a different aspect of it.

To keep the GUI as general as possible, we included a *ViewController* interface, which represents a viewable entity in the GUI. This enables anyone who implements this interface to use the GUI. For example, the network level implementation can be tested and previewed using this GUI, provided that some classes implement the *ViewController* interface.

6 Interaction with higher and lower levels

The logical level has interaction with the network level as well as the client level. We designed the system in a way that completely decouples the implementation of other levels from the logical level: Actual messages can be sent using any protocol. Moreover, our implementation of *AuthorizedViceroyPeer* (named *LocalViceroyPeer*) does not even have the notion of sending a message.

The interaction with the network level is done using a set of listeners that wait for *Message* objects to arrive. Such an object is not necessarily the real message that is being passed. It is only a wrapper for the relevant data in a specific action, such as lookup. A class named *CommunicationManager*, implemented by the lower level, manages the communications between the levels.

The interaction with the client level is also done using a set of listeners. The client is able to register listeners for various events, such as *LookupAnswerReceived*. The asynchronous nature of the system enforces an Event-Driven model.

7 Unit Tests and Internal Correctness Checks

We added Unit Testing classes for all the main classes. Those tests are an inherent part of every version-release. Those test classes were intentionally not removed from the final version in order to ease future extensions and additions to the project. We also used the new **assert** keyword in the Java language for internal correctness checks. Those checks can be turned off at run time by not using the **-ea** option when launching the program. These correctness checks provide a reliable way to find many of the implementation bugs as soon as something goes wrong, and they are extremely important in a distributed network like Viceroy.

8 Lookup Algorithms

We have investigated several routing algorithms in the context of the Viceroy network. We have implemented a statistics program that compares the effects of different lookup algorithms on a given Viceroy topology, and conducted a small research of this analysis (for details see section 11).

We used the Strategy pattern [Gamma95] for the various lookup algorithms: A *LookupAlgorithm* object is passed in the lookup process, so that each peer asks for its next move on the lookup using the *nextOnLookup(...)* method. This way, it would be easy to add new lookup algorithms, by extending the current algorithms.

Generally speaking, all lookup algorithms are supposed to find a *LookupPeer*, which is responsible for a given *Resource*. When looking up a Resource, we actually apply a given hash function on that *Resource*, and look up the result, which is in the range $[0 \dots 1)$. Responsibility for a given *Resource* means that this calculated value is in the stretch of that *LookupPeer* (between the peers ID and its predecessors ID). Note that the *LookupPeer* is not guaranteed to hold this *Resource*, but this is the only possible place to look for this *Resource*.

In the following subsection we will briefly overview the currently implemented algorithms.

8.1 Paper Algorithm

The algorithm presented in [Viceroy] is a three-phased routing algorithm: in the first stage, the lookup query is routed to the root by using the **up** links (known as Proceed To Root phase). In the second phase (known as Traverse Tree phase), the query traverses the tree, by choosing either the **left** or the **right** links. This phase continues until the required son does not exist, or until it overshoots [2] the target. In the final phase (Traverse Ring), the lookup query traverses the ring using the **successor** and **predecessor** links until the node responsible for the value is found.

PROCEED TO ROOT PHASE (AuthorizedViceroyPeer peer, Resource valueToLook, boolean viaParent)

```
if (peer.getLevel() ≠ 1)
    if ( viaParent && peer.parent() ≠ null)
        goto Parent in Proceed to Root phase
    else
        goto Successor in Proceed to Root phase
else
    goto TraverseTree phase
```

[2] overshoots means: true if $\frac{1}{2^{(child.level-1)}} <$ ClockwiseDist(child,val)

Traverse Tree phase(AuthorizedViceroyPeer peer, Resource valueToLook)
    if(clockwiseDist(peer, valueToLook) $< \frac{1}{2^{level}}$ &&
      leftChild $\neq$ null &&
      $\neg$overshoot(leftChild, valueToLook))
        goto leftChild in Traverse Tree phase
    else if (clockwiseDist(peer, valueToLook) $\geq \frac{1}{2^{level}}$ &&
      rightChild $\neq$ null &&
      $\neg$overshoot(rightChild, valueToLook))
        goto rightChild in Traverse Tree phase
    else goto Traverse Ring phase

Traverse Ring phase (AuthorizedViceroyPeer peer, Resource valueToLook)
    if peer.isResponsibleFor(valueToLook)
        return peer
    else
      if(valueToLook.isInStretch(peer, successor)
        goto Successor in Traverse Ring phase
      else
        goto Predecessor in Traverse Ring phase

8.2 PapersPlus Algorithm

Similar to the Paper Algorithm, plus a simple optimization check. The extra check occurs in the Traverse Tree phase, where before doing anything, it checks if the left child or the right child is responsible for the lookup value. If so, then without any additional check, it goes to the ring phase with the child that is responsible as its parameter.

The altered Traverse Tree phase (AuthorizedViceroyPeer peer, Resource valueToLook)
    if(leftChild $\neq$ null && leftChild.isResponsible(valueToLook))
        goto leftChild
    if(rightChild $\neq$ null && rightChild.isResponsible(valueToLook))
        goto rightChild
    if(clockwiseDist(peer, valueToLook) $< \frac{1}{2^{level}}$ &&
      leftChild $\neq$ null &&
      $\neg$overshoot(leftChild,valueToLook))
        goto leftChild
    else if (clockwiseDist(peer, valueToLook) $\geq \frac{1}{2^{level}}$ &&
      rightChild $\neq$ null &&
      $\neg$overshoot(rightChild, valueToLook))
        goto rightChild
    else goto Traverse Ring phase

8.3 TraverseTreePlus Algorithm

A change in the Traverse Tree phase of the lookup algorithm: If one of the children exists, go to one of them in any case (even if they overshot the target). Else go to Traverse Ring phase. Currently, there is no threshold and the algorithm proceeds down the tree as long as there are children.

The altered Traverse Tree phase (AuthorizedViceroyPeer peer, Resource valueToLook)
if(clockwiseDist(peer, valueToLook) $< \frac{1}{2^{level}}$ &&
leftChild $\neq$ null &&
$\neg$overshoot(leftChild, valueToLook))
goto leftChild
else if (clockwiseDist(peer, valueToLook)$\geq \frac{1}{2^{level}}$ &&
rightChild $\neq$ null &&
$\neg$overshoot(rightChild, valueToLook))
goto rightChild
else if (leftChild $\neq$ null && rightChild == null)
goto leftChild
else if (rightChild $\neq$ null && leftChild == null)
goto rightChild
else if (leftChild $\neq$ null && rightChild)
goto the child that least overshoots
else goto Traverse Ring phase

8.4 TraverseTreeAbsolute Algorithm

In the Traverse Tree phase the algorithm chooses the closest node (absolute) to the target out of the current node and its children. If the left/right child is closer to the target, then it goes to this child and continue in this phase. Else (the current node is closer to the target), it goes to Traverse Ring phase.

8.5 FindFast Algorithm

A greedy algorithm. Performs a greedy search on the absolute distances to the target chosen from all possible inbound and outbound connections and goes to the closest connection. This algorithm does not use the structure of the Viceroy network at all, but rather it treats all the links equally and only tries to find the closest link to the target.

Find Fast Algorithm (AuthorizedViceroyPeer peer, Resource valueToLook)
if (peer.isResponsibleFor(valueToLook)
return peer
else
list all Inbound and outbound Connections of this peer

```
  remove all already visited peers in this lookup request
//it is possible that a certain peer will be visited twice in a lookup,
//because of the locality of the greedy algorithm and it might cause
//endless loops.

    return findAbsoluteClosestTo(connections, valueToLook)
```

9 Extending The System

Future extensions of the systems can be done virtually anywhere, as we used interfaces all around, and kept them as general as possible. Nice ideas for extensions can be:

- Implementing *LookupAlgorithm* in other ways and comparing their results.
- Extending *LocalViceroyPeer* with a class that implements Join, Leave and level selection differently.
- Giving a more interesting implementation of *LookupMessageStatistics* and analyzing the results in upper levels.
- Introducing a caching mechanism using a combination of the given implementation of *AuthorizedViceroyPeer* and a new implementation of it.
- Giving a different implementation of the communication level, using various protocols.

10 Enabling Analysis of the Network

We have constructed the infrastructure for a thorough statistical analysis of the system. It consists of *MessagePath*, *MessageStatistics* and *LookupMessageStatistics* interfaces, along with basic implementation of it. Currently, we only record the time it takes the lookup message to reach any peer in its path, the path itself and the type of the connection (**left child**, **successor** etc).

Obviously, far more interesting data could be collected along the path, but this was left for future extensions of the system. The current analysis is based only on the number of peers along the path; Further and more thorough research might be done here, for example: calculating averages of sending times, standard deviations of times of lookup results, finding bottlenecks along the path and estimation of the load balance in different areas.

11 Consequences From Basic Analysis

We have implemented a small statistics program that compares the results of different lookup algorithms on a given Viceroy topology, where peers join in at

random, exactly as they will do in a real Viceroy network. Here are some of the results we got, based on a Viceroy network: For a net containing 1000 peers, 200 lookups were performed with the Papers algorithm and the average result was: 60.95 lookup steps; while for the same net size and the same number of lookups, which were performed with the FindFast algorithm the result average was: 11.24 lookup steps. The median values were 25 for Papers and 10 for FindFast.

See Figure 5 for the full results:

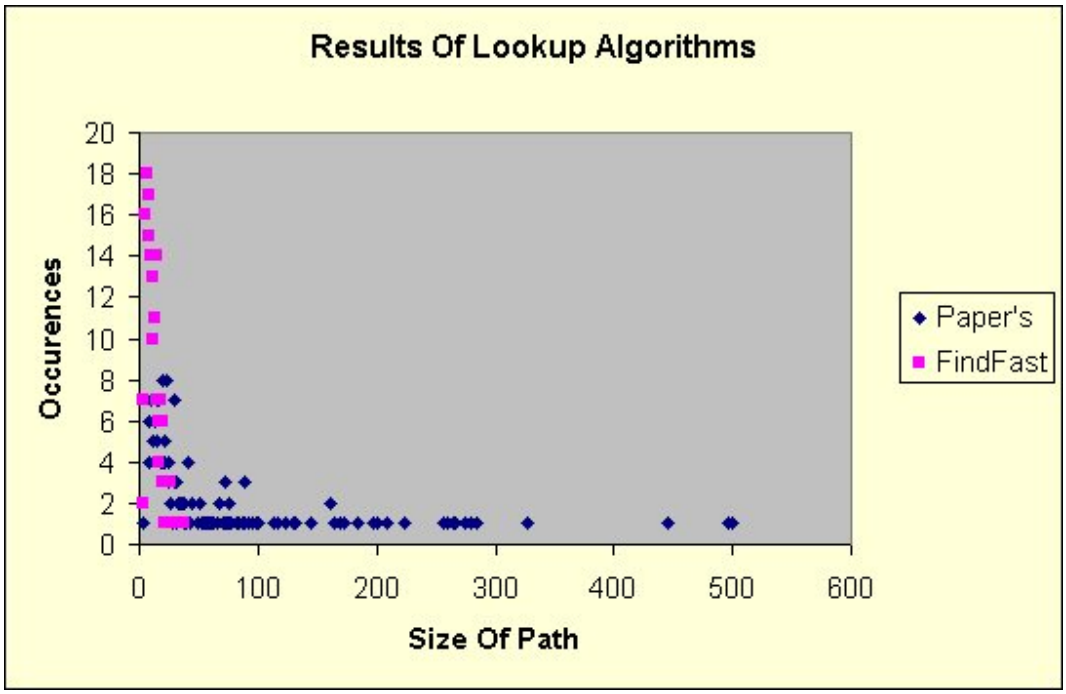

Figure 5: lookup results

12 Acknoledgments

We would like to thank our advisor Dahlia Malkhi for the helpful guidance in this project.

References

[Gamma95] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software, Adisson-Wesly, Riding, MA, 1995. ISBN: 0201633612

[Pastry] P. Drushel and A. Rowstron. "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems". *Proceeding of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware),* November 2001.

[D2B] P. Fraigniaud and P. Gauron. "The Content-Addressable Network D2B". Technical Report 1349, LRI, Univ. Paris-Sud, France, January 2003.

[Viceroy] D. Malkhi, M. Naor and D. Ratajczak. "Viceroy: A scalable and dynamic emulation of the Butterfly". In *Proceeding of the 21 st ACM Symposium on Principles of Distributed Computing (PODC'02)*, July 2002.

[DH] M. Naor and U. Wieder. "Novel Architectures for P2P Applications: the Continuous-Discrete Approach". In *The Fifteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '03)*, 2003.

[Chord] I. Stoica, R. Morrise, D. Karger, M. F. Kaashoek and H. Balakrishnan. "Chord: A scalable peer-to-peer lookup service for Internet applications". In *Proceedings of the SIGCOMM 2001*, August 2001.

[Tapestry] B.Y. Zhao, J. D. Kubiatowicz and A.D. Joseph. "Tapestry: An infrastructure for fault-tolerant wide-area location and routing". U.C. Berkeley Technical Report UCB/CDS-01-1141, April, 2001.

[Secure type-safe collections] Piet Jonas "Secure type-safe collections". http://www.javaworld.com/javaworld/jw-04-2001/jw-0427-collections.html