Technische Universität Braunschweig
Institut für Betriebssysteme und Rechnerverbund

Diplomarbeit

PLANETEN WACH HUND NETZ
Instrumenting Infrastructure for PlanetLab

cand. inform. Vitus Lorenz-Meyer

Aufgabenstellung: Prof. Dr. Lars Wolf
Betreuer: Dipl.-Inf. Jens Brandt

Braunschweig, den November 30, 2006

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angebenen Hilfsmittel verwendet habe.

Braunschweig, den November 30, 2006

TECHNISCHE UNIVERSITÄT BRAUNSCHWEIG
INSTITUT FÜR BETRIEBSSYSTEME UND RECHNERVERBUND
Prof. Dr. -Ing. M. Beigl . Prof. Dr.-Ing. L. Wolf

Braunschweig, den 01.05.06

Aufgabenstellung für die Diplomarbeit

PlanetenWachHundNetz: Instrumenting Infrastructure for Planetlab

vergeben an

Herrn cand. inform. Vitus Lorenz-Meyer
Matr.-Nr. 2691370, Email: derdoc@gmx.de

Aufgabenstellung

We are investigating distributed parallel prefix operations. Efficient implementations, like Google's MapReduce for example, utilize a reduction tree as well as involve locality-aware algorithms. In a static network reduction trees can be built once using a-priori known perfect proximity information and need never be touched again. In highly dynamic environrnents, such as peer-to-peer systems, this is substantially harder to achieve and maintain. From related work two types of reduction trees for dynamic systems that both build on structured overlays, also known as Distributed Hashtables (DHTs), seem to emerge. Both of these structures are not concemed with exploiting existing locality. We show how these data-structures can be augmented to be more efficient and take advantage of locality information that might exist in the underlying overlay. Distributed parallel prefix can be used to aggregate and thus reduce data from many sources. This is for example useful for statistics collection and applicationlevel monitoring. To validate our hypothesis we are building an application-level data collection system, called PlanetenWachHundNetz (PWHN-pronounced 'pawn'), which is German that loosely translates to 'PlanetaryWatchDogNet'. Our initial evaluation of it is going to be performed on the PlanetLab testbed.

Laufzeit: *6* Monate
Die Hinweise zur Durchführung von Studien- und Diplomarbeiten am TBR sind zu beachten .
Siehe : http://www.ibr.cs.tu-bs.de/kb/arbeiten/html/

Aufgabenstellung **und** Betreuung:

Prof. Dr. L. Wolf [illegible]

Vitus Lorenz-Meyer [illegible]

PLANETEN **W**ACH **H**UND **N**ETZ

INSTRUMENTING INFRASTRUCTURE FOR PlanetLab

VITUS LORENZ-MEYER

Department of Computer Science

to the

best FATHER in the world

and TANIA with love

PLANETEN **W**ACH **H**UND **N**ETZ

INSTRUMENTING INFRASTRUCTURE FOR PlanetLab

by

VITUS LORENZ-MEYER

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Computer Science

THE UNIVERSITY OF TEXAS AT EL PASO

December 2006

Acknowledgements

Firstly, I would like to thank Eric Freudenthal for believing in me. Without his effort I would not be here.

In addition, I would like to thank him for his endless comments on every paper I wrote, which have helped me perfect my writing style. Additionally, I would like to thank the whole *Robust Research Group* for their continuos support and helpful suggestions. Specifically, I thank Ryan Spring, Steve "Happystone" Gutstein and Michael "Sexy Mike" Havens.

Without the unconditional love of my girlfriend and now fiancé, Tania, this work would not have been possible.

I will be forever thankful.

Furthermore, I want to thank University of Texas at El Paso (UTEP) Computer Science (C.S.) Department professors and staff as well as the IT Department Staff for all their hard work and dedication. They provided me with the means to complete my degree and prepare for a career as a computer scientist. This includes (but certainly is not limited to) the following individuals:

Bea

Beatrix is the good fairy of the C.S. Building. Without her invaluable services I would not have been able to manage everything in Germany after my father's unexpected demise.

Leo and Josè

Together they keep the C.S. Department online. Thank you for everything.

Sergio and Tom

Sergio, the space manager at the Research and Academic Data Center (RADC) of UTEP was our primary contact for the main server room at UTEP. This is where our PlanetLab nodes were located. He was very kind and of great help for us.

Tom Mikelson, the IT guy that keeps the campus network running has helped us numerous times in setting up and wiring our PlanetLab nodes. Thank you so much for that.

This thesis was submitted to my supervising committee on October 13th, 2006.

Abstract

The distributed and highly dynamic nature of Peer-to-Peer (P2P) systems can make instrumentation and development difficult. Instrumenting a system aids development by producing debugging and status information. Moreover, collection of data from such systems has the potential to produce vast amounts of data that preclude the use of approaches that collect unfiltered logs from many participating nodes. My research investigates the dynamic generation of trees that permit the efficient aggregation of data extracted from nodes by enabling the use of parallel prefix computations on P2P systems. These automatically structured parallel prefix operations are used to permit the efficient collection of instrumentation data.

This paper describes PlanetenWachHundNetz (PWHN), an integrated instrumentation toolkit for distributed systems that uses the dynamically organized parallel prefix mechanisms I investigated. Like Google's MapReduce, PWHN utilizes user supplied programs to extract data from instrumented hosts and aggregate this data. In addition, convenient Graphical User Interfaces (GUIs) are provided to display collected data and generate plots.

In order to evaluate the effectiveness of the parallel prefix aggregation mechanisms I investigated, PWHN supports both a centralized and distributed aggregation mode. PWHN's first (centralized) implementation has been useful for instrumentation and tuning of the live Fern installation on hundreds of PlanetLab nodes, but is not scalable. In contrast, the second implementation employs more scalable algorithms that dynamically construct data aggregation trees using P2P techniques and thus is potentially scalable to much larger deployments.

Table of Contents

List of Tables

List of Figures

Listings

Chapter 1

Overview and Problem

1.1 Introduction

Distributed systems are generally designed to satisfy a target problem's requirements. These requirements frequently do not include instrumentation of the system itself. As a result, the designs typically do not include instrumentation infrastructure, which makes tuning and debugging difficult. Furthermore, focusing upon adding this infrastructure might distract an engineer from designing for performance and thus can introduce an unacceptably high overhead to the system even when it is not used - compared to a system designed without instrumentation in mind. The objective of PlanetenWachHundNetz (PWHN) is to develop infrastructure suitable to assist the instrumentation of self-organizing distributed applications that lack data collection and aggregation mechanisms.

The instrumentation and tuning of Peer-to-Peer (P2P) systems can require the analyses of data collected from a large number of nodes. If the data has not been selected and aggregated properly by the providing nodes, the resulting amount of data may be too large to use available communication and storage capabilities. Moreover, if it is sent to a single, central collection point for analysis, that node's resources might become saturated. Therefore, centralized solutions may be impractical. Thus, it may be desirable to aggregate summaries of data in a distributed manner, avoiding the saturation of a "master" host who would directly transmit instructions to and receive data from all of the other (slave) nodes.

Google's MapReduce was designed for the distributed collection and analysis of very large data-sets, in environments with groupings of participants located in dedicated computation centers that are known a-priori. A user of MapReduce specifies arbitrary programs for its map and reduce phases. The *Map* phase "selects" the data to be aggregated and outputs them in the form of intermediate key/value pairs; afterwards, the *Reduce* phase digests these tuples grouped by their key into the final output. These selection and aggregation programs can frequently be easily constructed using the many convenient scripting programs commonly available on unix systems and then linked against the MapReduce library (written in C++).

PWHN extends the MapReduce model to P2P. The primary difference between the environment MapReduce was designed for and P2P-networks is that the set of active participants and their logical groupings frequently changes in P2P systems. Conventional approaches of pre-structuring aggregation and distribution trees are inappropriate given P2P system's large *churn*[1] of membership.

A measuring infrastructure for distributed systems faces the challenges of selection and aggregation of relevant data under churn, while ensuring good usability.

PWHN utilizes P2P techniques including Key-Based Routing layers (KBRs) and a data-structure that extends existing algorithms for guiding the construction of an aggregation tree upon the internal structure of Distributed Hash Tables (DHTs).

1.1.1 Coral

Consider Coral [8], a load-balancing P2P-Content Distribution Network (CDN) implemented as a HTTP proxy for small-scale servers that cannot afford large-scale dedicated solutions like akamai. It is referenced by many private websites and provides high availability. Like many distributed applications Coral's construction does not include instrumenting and logging infrastructure, thus it is hard to hard to tune and determine its limits of scalability. After this was discovered, a logging and collection infrastructure was crafted onto Coral. This approach was centralized, a central server outside of Coral just requested all logs

[1]Churn means the characteristic of a normal P2P system that nodes frequently join and leave; thus, the membership "churns"

and inserted them into a database. This technique did not scale well and was therefore discontinued quickly.

We learned of Coral's misfortune and decided that it is a hard problem to collect and aggregate statistics about a P2P application. It was my motivation to build a system to make this process easier for application designers.

1.2 The Problem

The previous section introduced three distinct problems.

1. *Distributed Selection*: the problem of doing efficient filtering of data near to collection points. PWHN uses the approach of MapReduce that permits users to specify programs that are executed on the systems generating data that locally select (and possibly pre-process) the data to be collected.

2. *Distributed Aggregation*: efficient reduction of data from a large number of sources throughout a P2P system through the composition of appropriate associative and communitive operations to reduce the volume of data transmitted back to the "master". Solution hint: Logging and instrumentation systems are amenable to parallel prefix.

3. *Churn and how to build the tree*: refers to the problem of creating and maintaining an efficient dissemination and reduction tree in a highly dynamic environment. Churn of membership in P2P systems makes it necessary to *dynamically* generate aggregation trees among the set of nodes participating in the P2P system. Our solution leverages the structure of DHTs and makes use of hashes after the initial dissemination phase to avoid sending data twice.

1.3 Research Objective and Goal

The research objective of this thesis was to develop a system to efficiently collect, aggregate and summarize data collected from single nodes of a P2P application. This is done through extension of ideas from MapReduce, i.e. allowing arbitrary programs for all three phases; as well as by extending data-structures that build aggregation trees upon structured routing algorithms.

PWHN's principle objective is to produce a useful tool that instruments P2P systems, such as those investigated by the community of researchers who utilize PlanetLab to investigate self-structuring (P2P) applications and protocols. It was motivated in part by the failure of Coral's logging and measuring infrastructure. We wanted to design a program that could do better.

Google's MapReduce demonstrates the utility of a generic tool for aggregation for non-P2P systems because it effectively distributes the load of selection and aggregation, thus minimizing perturbation.

My work extends MapReduce to full-P2P environments through the utilization of structured P2P techniques and thus provides a testbed for evaluating the effectiveness of their approach.

1.4 Organization

This thesis is organized as follows. Chapter 2 describes previous work that is relevant to my research and introduces related data-structures. Chapter 3 introduces techniques utilized by PWHN to detect the set of active nodes and construct an aggregation tree. Chapter 4 provides greater detail of the design. Chapter 5 describes experiments and evaluates the results. Chapter 6 enumerates possible work that might be an interesting point for future research. Chapter chapter 7 concludes. A glossary lists all acronyms used in this thesis. Several appendices provide details of components utilized (Appendix A to Appendix E).

Chapter 2

Related work

2.1 Introduction

Several researchers have considered the challenge of collecting and aggregating data from a network of distributed information sources. Distributed-*Database Management Systems (DBMSs)* (see subsection 2.6.2) route results from systems that generate query responses towards systems that aggregate them, *Sensor Networks* (see subsection 2.6.4) gather data from widely distributed nodes with limited communication and computational resources, and lastly, monitoring of both distributed systems and applications collect meta-data (logs) and make use of *Aggregation Overlays* (see subsection 2.6.3) to send them to a central location and reduce them *en-route*.

A key problem of centralized aggregation approaches is that the amount of data sent towards one node for processing can saturate computational and network resources near to the "centralized" collection point. With the growth of large distributed systems, the importance of data aggregation is likely to increase. Thus, it is desirable to push as much computation of the collected data as possible out to a distributed aggregation infrastructure near to the sources. This has the added advantages of distributing load in the network more evenly as well as localizing network traffic.

Like D-DBMS in which computation is performed at aggregation nodes near to data sources in order minimize network congestion and distribute the computation required to select and join results, systems that generate logs, have the advantage that these logs mostly reflect the same statistics making them homogenous. Thus, if they can be aggregated using a commutative and associative operation they are amenable to reduction through the natural application of *parallel prefix* techniques (see section 2.2).

2.2 Parallel prefix and MapReduce

Addition is an associativ operator which, in this case, means the following: $x + (y + z) = (x + y) + z$. It is also commutative which can be written as: $x + y = y + x$. The same is true, for example, for min and max. The average function is only approximately an associativ function because the average of three inputs cannot be computed in any order. Nevertheless is it usable in parallel system by the addition of a count of the number of elements in addition to the sum of them.

As others have done ([22] et al.), it can be observed that this can be used to parallelize the algorithm for larger inputs.

Google's MapReduce provides a convenient interface to distributed parallel prefix operations. A user of MapReduce provides two scripts: one to map data to a "collection" space, and a "reduction" program which implements a commutative and associative "aggregation" operation. MapReduce solves the *aggregation* part of the problem introduction, as given in item 2.

Unlike MapReduce, which imposes (and benefits from) a tuple-structured system, PWHN sends opaque data, and thus is more flexible in what it can transmit. Moreover, the MapReduce library groups all intermediate tuples on the key and passes all values pertaining to the same key to the Reduce operator at once; whereas PWHN does not try to parse intermediate streams to determine their keys, but instead just passes two outputs to the Aggregator.

MapReduce does not support specification of an Evaluator, because it was not meant to produce a single ouput from the reduce phase, but rather one result per key. In contrast, PWHN allows the user to provide a program for the evaluation that is passed the last output from the Aggregator to be able to perform some post-processing.

2.3 Distributed Hash Tables (DHTs) and their Key-Based Routing layers (KBRs)

My work extends key-based routing techniques. This section provides an overview of DHT and key-based routing, which were developed as pragmatic solutions to challenges in P2P systems. In order to motivate and describe DHTs and KBR techniques, this section reviews the history and structure of these earlier systems. For pedagogic reasons it is a historical narrative.

2.3.1 A Short History of P2P

The *first Generation* of P2P started with `Napster` in 1999. Napster was a P2P file sharing system that permitted users to efficiently obtain files published by other users. Napster used a central server for indexing files and thus had a central point of failure. “Peers” participating in Napster both requested named files directly from and transmitted named files directly to other peers. However, Napster was not fully P2P because it utilized a central host responsible for maintaining an index of which “data” each peer contained. Thus, every peer transmitted a list of the files it shares to this master who would subsequently share the information with other peers. From this list each peer can determine from who to request specific files. A historical note: Napster was forced to discontinue service in 2001 as a result of litigation regarding its use to transmit copyrighted material.

Even before Napster’s shutdown, *Second Generation* P2P systems started to appear. Nullsoft’s Justin Frankel started to develop a software called `Gnutella` in early 2000, right after the company’s acquisition by AOL. They stopped the development immediately because of legal concerns, but it was too late since the software was already released and the protocol was reverse-engineered and subsequently taken over by the open-source community.

Gnutella was a fully P2P system because both searching and downloading of files was completely decentralized. The search relied on flooding: a client requesting a file transmitted a search message to all its known immediate peers specifying an initial Time-To-Live (TTL) of approximately 5 hops. Receiving peers searched through their own file list, returned a result when found and re-broadcasted the message with a decremented TTL. When the TTL reached zero, the message was discarded. This approach to searching a distributed index is neither efficient nor guaranteed to be complete, i.e. find scarce data in the network. Unlike Napster's central indexing server, Gnutella's indexing mechanism was fully distributed and thus proved very hard to shutdown. It is still in use, albeit with a updated protocol (version 2).

Third Generation is often used as a term to refer to everything that followed. Third generation P2P systems utilize both central and decentralized techniques. Instead of identifying files by name (which can be easily spoofed), these systems began employing hash digests[1] to refer to files and employing partial, often called *swarm*, downloading. This enables users to rename files, while still allowing the system to reference these files uniquely.

Swarm downloading splits large files into smaller parts and downloads different parts from different peers. This allows (mostly implicit) partial sharing before the file is downloaded completely, thus speeding up distribution.

The most notable third generation P2P network is the `edonkey2000` network, which is still widely in use. It also appeared around the time Napster was shut down, and in 2004 overtook *FastTrack* (owner of Morpheus, another SecondGen file-sharing network) to become the most popular file-sharing network. It uses central servers for file indexing and connects to peers directly for downloading, much like Napster did. Unlike Napster, eDonkey democratizes indexing by permitting that function to be implemented by multiple peers. As a result of this, eDonkey has also proven difficult to disable.

[1]A hash digest of a file is generated by hashing the file's contents using a well-known hash function, e.g. `SHA-1`.

2.3.2 The Advent of DHTs

Around 2001 researchers started to investigate ways to make searching in unorganized systems more efficient. This eventually lead to the conecpt of a Distributed Hash Table (DHT), of which the first four [37, 41, 45, 55] were all released in 2001.

DHTs are designed around these three central properties:

1. Decentralization,
2. scalability, and
3. fault tolerance.

A DHT is built upon the abstract notion of a *keyspace*. Ownership of this keyspace is split among all participating nodes, who are assigned random keys out of this space, according to a particular scheme that depends on the implementation. The DHT builds a **(structured) overlay network** over all nodes which allows it to find the owner of a given key in $O(log(n))$ time; for an n-bit id-space. Following [6], this (underlying) part of the DHT is commonly referred to as the Key-Based Routing layer (KBR).

The KBR guarantees to route a message to the owner of a given key in $O(log_k(n))$ hops for an n-bit wide ID-space. To do this, the KBR employs a k-ary search in the ID-space by recursively partitioning the search space in a divide-and-conquer fashion. For example the Chord DHT iteratively divides the search space in half; other radixes are possible. The*arity* depends on the base that the KBR defines its IDs to be. Without loss of generality I assume them to be base 2 in my examples, like most DHT implementations do today.

IDs are uniformly selected identifiers in a large discrete field. To change any number to any other, all that needs to be done is to "correct" digits in the source number starting from the beginning until the target number is reached. For this, at most "number-of-digits" steps are needed. Since the number of digits of any number depends on the base, this would take at most $O(log_k(n))$ for base k.

Thus, to send a message from any ID to any other we only have to "correct" one digit of the ID at each step. For IDs defined in base 2 this is basically employing a distributed binary-search by making a routing decision at each node. It checks in which range (subtree of the search-tree) the target ID falls and forwards the message to a node in that subtree.

DHTs export an interface that is close to that of a normal hash table, hence their name. Like a conventional (non distributed) hash table, a DHT stores $key : value$ pairs that can be efficiently indexed by "key."

Upon calling $put(key, val)$, the key is replaced by its hash using a well-known hash function by the DHT, which then sends message containing (key,val) to the owner of of the hash of the key $h(key)$. This message is forwarded to the node that owns this key, which in most implementations, is the node whose host-ID is closest to the given key. Later the value can be retrieved from this node using the hash of the key to forward a `get` message to it. A primary feature of consistent hashing is its "consistency," which means that all nodes use the same hash function and key space, and thus, should agree on the assignment of keys to nodes. Uniform distribution of the hash space is a secondary (but desirable) property.

Thus, the only difference between a traditional hash table and a DHT is that the hash bucket "space" is dynamically partitioned among participating nodes.

It is impossible to achieve fully reliable operation in a large distributed system. As a result, most P2P systems instead provide sufficient *best effort* reliability to be useful without attempting to be fully reliable. For example, a P2P system may attempt to "remember" data stored within it only for a short duration in time. However, the failure of a small set of nodes may cause the data to be lost, and a node requesting a forgotten datum may need to refer to a canonical origin server from where it can refresh the P2P system's memory. State stored within these systems is thus volatile; and in contrast to state stored within stable (hard) storage, these systems' states are referred to as *soft*. A participating node *forgets* stored tuples after a certain time, thus they need to be renewed if they are to be kept around. This ensures that tuples can move should the ownership change either by nodes leaving or joining or that they can disappear if not needed anymore without explicitly having to send a delete message. Apart from using soft-state, most implementations also use replication to prevent tuples from disappearing when nodes leave.

For a more complete description of how a DHT works, its KBR and their interfaces, refer to Appendix B. DHTs and their KBRs solve item 3 of the problem.

2.4 PlanetLab

PlanetLab [28, 30] is a consortium of research institutions distributed throughout our planet donating machines that contribute to a common goal. This goal consists of being able to test widely distributed services in their *natural habitat*, i.e. in a network that is comprised of nodes scattered around the globe, their connection to the Internet being the only thing they share. By joining PlanetLab, an institution agrees to donate at least two nodes to the cause. In exchange, the institution earns the permission to request slices of it. A slice can be seen as a "slice" of all the global resources of PlanetLab. The slice is a collection of Virtual Machines (VMs) running on a set of PlanetLab nodes. This slice can then be used to perform experiments that require a large number of distributed machines, for example experimental network services.

University of Texas at El Paso (UTEP) is a member of PlanetLab, which permitted me to use it to develop and evaluate PWHN.

For a description of the PlanetLab environment please refer to Appendix A.

2.5 Data structures

In this section, I describe two approaches for building aggregation trees upon DHTs that I investigated in my research. Several independent research groups developed their own structures for this. Since there exists no standard nomenclature for these algorithms (the authors of these papers do not name their algorithm), I introduce my own notation for these techniques to avoid confusion.

This description of the data structures used also appears in [24].

2.5.1 KBT

Almost all DHTs, sometimes referred to as *Prefix-based Overlays* or *Structured Overlays*, build on Plaxton et al.'s [31] groundbreaking paper (because of the first letters of the author's surnames - Plaxton, Rajaraman, and Richa - known as *PRR*) on routing in unstructured networks.

Key-Based Trees (KBTs) use the internal structure of the trees that Plaxton-style [31] systems automatically build for routing in the following way. These key-based routing protocols populate a flat identifier space by assigning long bit-strings to hosts and content which are simply 160-bit integer identifiers. Two keys that have n most significant bits of their IDs in common are described to have a "common prefix" of length n. Thus, "prefixes" of IDs function as "search guides" since they are prefixes of actual node keys. Now consider the following. Each node is the root of its own "virtual" tree. Both nodes at depth n have $(n-1)$-bit prefixes in common with their root, while the next (n^{th}) bit is 0 for the left and 1 for the right child. The result is a global, binary tree.

This assumes that the DHT fixes exactly one bit per hop. DHTs that fix more than one bit per hop will have a correspondingly higher branching factor. Since actual nodes will always have complete bit-strings, all internal nodes that are addressable by a prefix are "virtual," in the sense of the tree. The physical nodes are the leafs of the tree and can be reached from the root ("empty prefix") by a unique path. Since the relationship between tree nodes and their place in the ID-space is unambiguous, meaning that the tree is fully defined by a set of node keys, I have termed this data-structure Key-Based Tree (KBT).

Each DHT defines a distance metric that defines "nearby" nodes and guides searches.

Two examples of systems that use KBTs are Willow [48] and SOMO [54].

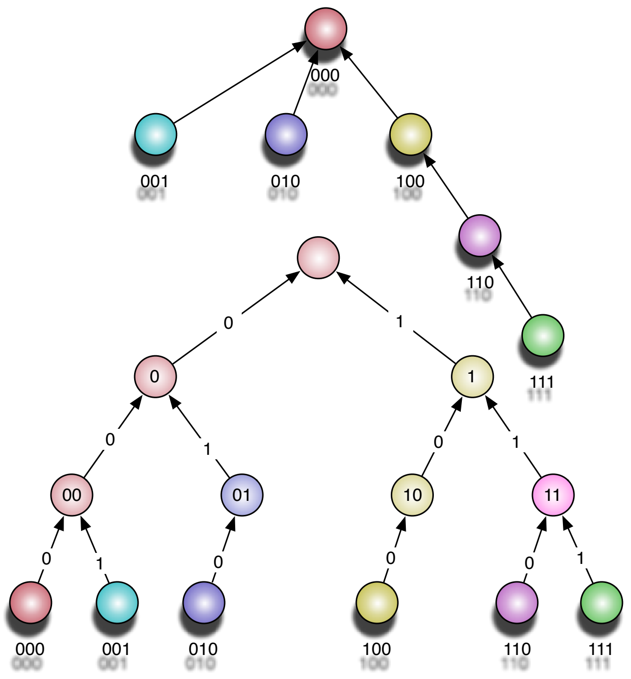


Figure 2.1: Sample KBT with six nodes using a 3-bit wide ID-space and a static approach in which the left child assumes the parent role ("Left-Tree"). The bottom part of the figure shows the "virtual" tree, whereas the top part shows the resulting real messaging routes.

Classification

The designers of an implementation of a KBT define its tie-breaking behavior, meaning which child will be responsible for its "virtual" parent, since both children match their parents' prefix. There are two basic approaches.

1. *Static* algorithms are deterministic for the same set of nodes and can only change if that set changes.

2. *Dynamic* algorithms determine the parent based on characteristics of the involved nodes, i.e. "in-vivo" (may even be evaluated in the form of queries). This has the effect that the choice is not deterministic, i.e. can be different for the same set of

nodes, and can change during its lifetime.

Choices of a dynamic approach are based on age, reliability, load, bandwidth or based on a changeable query. Static algorithms can only choose the node that is closest to a a deterministic point of the region (e.g. left, right, center). A static choice has the advantage of predictability, provided you have a rough estimate about the number of nodes. This predictability allows making *informed* guesses about the location of internal nodes in the ID space.

2.5.2 FTT

Routing towards a specific ID in key-based, structured overlays works by increasing the common prefix between the current hop and the target key until the node with the longest matching prefix is reached. The characteristic of DHTs, that a message from every node towards a specific ID takes a slightly different path, leads to the realization that the union of all these paths represents another tree which covers all live nodes - a different one for each ID. Thus, unlike KBTs, where the mapping nodes and the tree is fully defined by the set of node IDs, this tree is ambiguous - i.e. is dependent on finger table content.

These trees are like inverted KBTs; the exact node with the longest matching prefix is the root, whereas in a KBT every node could be the root depending on its tie-breaking algorithm, since every ID matches the empty prefix. Since this tree depends on the fingertables of all live nodes, I call it a FingerTable-based Tree (FTT).

Example systems that use FTTs are PIER [16, 17, 29] and SDIMS [53].

Classification

FTTs can be classified by their result-forwarding strategy: i.e. at what time aggregated results are sent to the parent. There are two possible options for this parameter:

1. *Directly after child-update*: The new result is sent to the parent directly after a child sends new data. This necessitates caching of incomplete results for running queries.

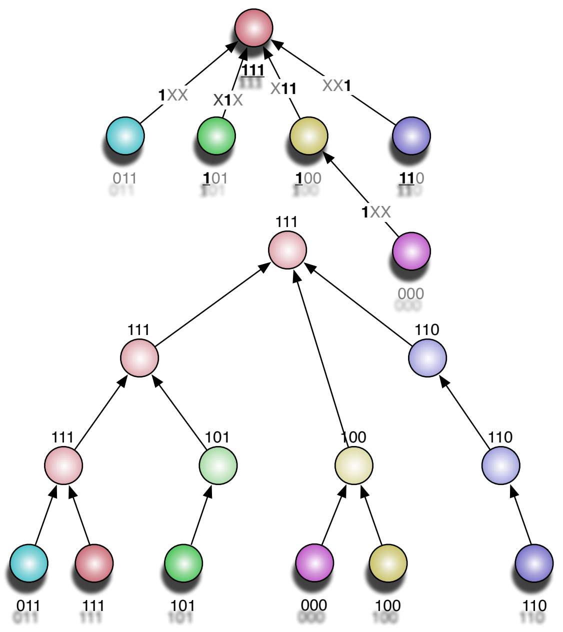


Figure 2.2: Top: Example graphical representation of all paths from 6 nodes towards Key(111) for a 3-bit wide ID-Space (the matching prefix is highlighted on each node and the corrected bit on each route), and the resulting FTT (bottom)

2. *After children complete*: New result is only sent to the parent after it is sufficiently complete (or stabilized). This necessitates a guess about the number of children which is another parameter when using this option.

2.5.3 Discussion

In this section I quickly compare the two introduced algorithms.

KBTs are binary if the underlying DHT fixes one bit per hop and every node is at the same depth ,i, i=*bits in ID*. Generally they are not balanced.

In contrast, FTTs are likely never balanced, and moreover not all nodes will be at the same level. In addition, they do not have to be binary. These properties make their usefulness as aggregation trees questionable, because of the possible lack of balance and completeness.

Some advantages of FTTs over a KBT are:

- There may be as many FTTs as needed by having the root depend on the hash of some characteristic of the query resulting in better load-balancing.
- Trees in FTTs can be rooted at any node.
- Trees in FTTs are kept in soft-state, thus there is no need to repair them, since they disappear after a short time.

The described data-structures both solve item 2 and, in addition, item 3 of the problem.

2.6 Survey of related P2P systems

This section is a review of P2P-systems which are classified using the nomenclature defined in the previous section.

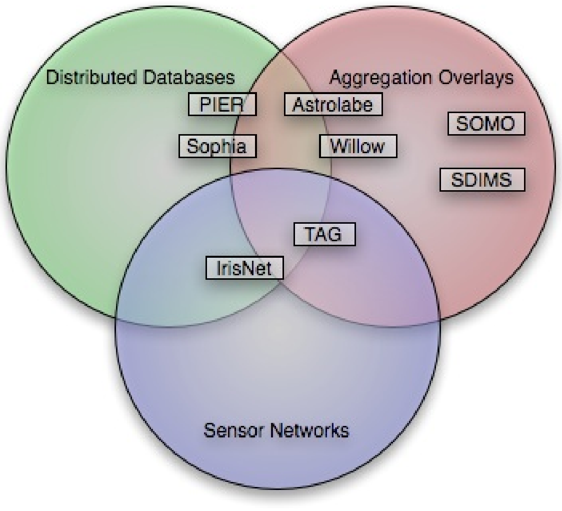


Figure 2.3: Rough classification of related systems into the three categories

2.6.1 Concepts

Routing Overlay (RO) All data collection and aggregation systems have to set up some kind of overlay network above the bare P2P transport network to provide routing. Consequently, I will call this abstraction the RO.

There are three common techniques.

1. flood,
2. gossip, and
3. using the KBR.

Flooding is the oldest and most inefficient approach of the three. It is used by some P2P file-sharing networks (see subsection 2.3.1).

Systems like Astrolabe (see 2.6.3) [47] employ a technique termed *gossiping*. The epidemic, gossiping protocol achieves eventual consistency by gossiping updates around, thus it has no need for routing since every node is assumed to know everything about the state of the system. The idea comes from the field of social studies from the realization that within a small group news propagate very fast by gossiping. It has an advantage in the case when inter-node status comparison does not dominate communication, i.e. when the group is small enough or the update-rate is low.

In most later systems routing services are provided by the DHT [20, 26, 37, 40, 41, 45, 55]. PRR's work [31] left room for aggregation in its trees, whereas most modern implementations disregard this feature.

Aggregation Overlay (AO) Query dissemination by branching and data concentration back up the tree is done through the AO. This overlay tends to resemble its most closely related natural object in the form of a tree. Though every system employs some kind of tree, even if it is not explicit as in a gossiping system, the building algorithms as well as the actual form of these trees are vastly different. Astrolabe, does not need to explicitly build a tree for its groups on run-time because it relies on the user specifying a name hierarchy on setup-time. Each name prefix is called a *zone*, and all those nodes whose Domain Name System (DNS) name starts with the same string are members of that specific zone. It builds a tree, whose nodes are the zones, out of the hierarchy, albeit one that might have a very high branching factor (number of sub-zones to each zone and number of nodes in each leaf-zone).

Summary In summary, designers of monitoring systems have the choice of maintaining only one global KBT, building a FTT for each query or query-type, or building their own algorithm that is not relying on a DHT, e.g. flooding or gossiping. Furthermore, they have to decide how to handle aggregation using these trees once they are built. Data has to be passed up somehow for aggregation. The obvious choice of re-evaluating aggregates on every update of any underlying value might not be the best choice, however. A lot of unused traffic results from rarely-read values that are updated frequently. Thus, a solution could be to only re-evaluate on a read. On the other hand, if frequently requested variables are seldomly written, this strategy leads to a lot of overhead again. Therefore, some systems, such as Scalable Distributed Information Management System (SDIMS) (cf. 2.6.3), adopt the concept of letting the aggregate requester choose these values. Known as *up-k* and *down-j* parameters, these values specify how far up the tree an update should trigger a re-evaluation, and after this, how far down the tree a result will be passed. Most systems implicitly use `all` for these values, this is why SDIMS' k and j parameters default to "all."

2.6.2 Distributed Databases

Distributed-*DBMSs* route results from systems that generate query responses towards systems that aggregate them. They are designed to closely approximate the semantics of traditional DBMS. The two most closely related systems are summarized in this section.

PIER

The ***P****2P* ***I****nformation* ***E****xchange &* ***R****etrieval* System [16, 17, 29], which is being developed by the DBMS team at UC Berkeley, is a project meant to be a fully-fledged, distributed query execution engine. It is a DHT-based, flat, relational DB. It has mostly been discontinued in favor of PHI [4].

Like commercial DBMS, it utilizes traditional *boxes-and-arrows* (often called `opgraph`) execution graphs. As its RO Pier uses a DHT. It has successfully been implemented on CAN [37], Bamboo [40] and Chord [45]. P2P Information Exchange & Retrieval (PIER) uses a single-threaded architecture much like SEDA [51], is programmed in Java and is currently deployed on PlanetLab [28, 30] (now called PHI). The basic underlying transport protocol PIER utilizes is Universal Datagram Protocol (UDP), although enriched by the UdpCC Library [40], which provides acknowledgments and Transmission Control Protocol (TCP)-style congestion handling.

A query explicitly specifies an opgraph using their language called UFL. Structured Query Language (SQL) statements that are entered get rendered into an opgraph by the system. This graph is then disseminated to all participating nodes using an index (see below) and is executed by them. For execution, PIER uses an $l$scan operator, which is able to scan the whole local portion of the DHT for values that match the query. To find nodes that might have data for a query, PIER supports three indices that can be used just like indices of any DBMS: true-predicate, equality-predicate and range-predicate. These predicates represent the AO.

Upon joining the network every PIER node sends a packet containing its own ID towards a well-known root that is a-priori hardcoded into PIER. This serve the purpose of a rendezvous point for building the tree. The next hop that sees this packet drops it and notes this link as a child path. These paths form a parent-child relationship among all the nodes, a tree. This is the true-predicate; it reaches every node.

The DHT key used to represent data stored within PIER is the hash of a composite of the table name and some of the relational attributes and carries a random suffix to separate itself from other tuples in the same table. Thus, every tuple can be found by hashing these distinguishing attributes. This is the equality-predicate.

The range-predicate is set by overlaying a Prefix Hash Tree (PHT [38]) on the DHT. It is still not fully supported by PIER.

Once result tuples for a query are produced, they are sent towards the requester by means of the DHT. The query is executed until a specifiable timeout in the query fires.

For aggregation queries, PIER uses one FTT per query. Every intermediate hop receives an upcall upon routing and gets a chance of changing the complete message; including its source, sink, and next hop.

PIER uses the Java type system for representing data; its tuples are serialized Java objects. They can be nested, composite or more complex objects. PIER does not care about how they are actually represented, other than through their accessor methods.

PIER is unique in its aggressive uses of the underlying DHT, such as for hashing tuples by their keys, finding joining nodes (PHT and indices), hierarchical aggregation and execution of traditional hashing joins. Although [15] suggests that PIER's performance is not as expected, it remains an interesting system for querying arbitrary data that is spread-out over numerous nodes.

Sophia

Sophia [50] is a *Network Information plane*, developed at Princeton and Bekeley. Like other P2P systems, Sophia was evaluated on PlanetLab. A network information plane is a "conceptual" plane that cuts horizontally through the whole network and thus is able to expose all system-state. Consequently, every system described here is an information plane.

While the sophia project does not explicitly address the dynamic structuring of an aggregation operation, its function as a distributed query and aggregation engine is relevant to PWHN. Sophia does not qualify as a P2P system, from what can be told from [50]. I decided to include it here because it is a system that takes a completely different approach to expressing queries.

Sophia supports three functions: aggregation, distributed queries and triggers. Queries and triggers can be formulated using a high-level, Prolog-based logic language or using the (underlying) low-level, functor-based instruction set. The authors chose a subset of the Prolog-language because both a domain-tailored description and a declarative query language incorporate a-priori assumptions about the system.

Every statement has an implicit part that evaluates to the current NodeID and Time. Thanks to this, Sophia has the ability to formulate statements involving time, as well as caching results. Queries can explicitly state on which nodes to evaluate particular computations. Sophia's query unification engine will expand a set of high-level rules into lower-level explicit evaluations that carry explicit locations. This also gives rise to the ability to rewrite those rules on the fly, thereby allowing in-query re-optimization. In addition, Sophia's implementation of Prolog has the ability to evaluate and return partial results from "incomplete" statements, i.e. expressions in which some subexpressions cannot be evaluated. An example of this being due to non-reachable nodes.

The run-time core is extendable through loadable modules that contain additional rules. Security within Sophia is enforced through *capabilities*. Capabilities are just aliases for rules that start with the string `cap` and end with a random 128-bit string. To be able to use a particular rule, a user must know its alias because the run-time system makes sure its original name cannot be used. The system also prevents enumerating all defined aliases. To avoid caching aliases in place of the real results, caching is implemented in a way that all capability-references are resolved before storing an entry.

The current implementation on PlanetLab runs a minimal local core on each node, with most of the functionality implemented in loadable modules. All terms are stored in a single, flat logic-term DB. Sensors are accessed through interfaces that insert "virtual" ground terms into the terms-DB, thereby making sensor-readings unifiable (processable by queries).

Summary

D-DBMS are a good approach to expressing arbitrary queries on a distributed set of nodes. Commonly they focus on keeping as much of the ACID semantics as possible intact, while still allowing the execution of distributed queries on a network of nodes. For all intents and purposes of a monitoring system, keeping the ACID semantics is not as important as having greater expressiveness for queries. This is not given in either PIER or Sophia because the former uses SQL whereas the latter uses Prolog to express queries.

2.6.3 Aggregation Overlays

Aggregation overlays are designed to provide a general framework for collection, transmission and aggregation of arbitrary data from distributed nodes. Generally, they are constructed to allow data that belongs to a specific group, like the group of relational data. This section introduces four aggregation systems.

Astrolabe

Astrolabe [1, 47] is a zoned, hierarchical, relational DB. It does not use an explicit RO as in a DHT. Instead, it relies on the administrator setting up reasonably structured zones according to the topology of the network by assigning proper DNS names to nodes. For example, all computers within UTEPs Computer Science (C.S.) department might start with "utep.cs." Following the protocol they will all be within the zone named "utep.cs," providing locality advantages over a KBT/FTT approach. Bootstrapping works by joining a hardcoded multicast group on the Local Area Network (LAN). In case that does not work Astrolabe will send out periodic broadcast messages to the LAN, for other nodes to pick up.

Locally available (read: published) data is held in Management Information Bases (MIBs), a name that is borrowed from the Simple Network Management Protocol (SNMP) [3]. The protocol keeps track of all the nodes in its own zone and of a set of *contact nodes* in other zones which are elected by those zones. Astrolabe uses the *gossip protocol* introduced in the introduction.

Each node periodically runs the gossip protocol. It will first update all its MIBs and then select some nodes of its own zone at random. If it is a representative for a zone, it will also gossip on behalf of that zone. To that end it selects another zone to gossip with and picks a random node from its contact list. If the node it picked is in its own zone it will tell that node what it knows about MIBs in their own zone. If the node is in another zone it will converse about MIBs in all their common ancestor zones. These messages do not contain the data; they only contain timestamps to give the receiver a chance to check their own MIBs for stale data. They will send a message back asking for the actual data.

Aggregation functions, introduced in the form of signed certificates, are used to compute aggregates for non-leaf zones. They can be introduced at runtime and are gossiped around just like everything else. Certificates can also be sent that change the behavior of the system. After updating any data, including mere local updates, Astrolabe will recompute any aggregates for which the data has changed.

Summarizing, this means that Astrolabe does not use routing at all. All the information one might want to query about has to be “gossiped” to the port-of-entry node. If one wants to ask a question, one has to install the query and wait until its certificate has disseminated down to all nodes (into all zones) and the answer has gossiped back up to him or her. This makes its *Gossip Protocol* the AO. Thus, Astrolabe does not have an RO according to my definition.

Astrolabe incorporates security by allowing each zone to have its own set of policies. They are introduced by certificates that are issued and signed by an administrator for that zone. For that purpose each zone contains a Certification Authority, which every node in that zone has to know and trust. PublicKey cryptography, symmetric cryptography, and no cryptography at all can all be used in a zone. Astrolabe at present only concerns itself with integrity and read/write Access Control Lists (ACLs), not with secrecy.

Astrolabe is an interesting project which has some compelling features, like security, but has been superceeded by Willow (cf. 2.6.3).

Willow

Willow [48] is a DHT-based, aggregating overlay-tree (a KBT). It uses its own Kademlia-like [26] DHT implementation. It seems to be an implicit successor to Astrolabe since it, according to the authors, inherits a lot of functionality from Astrolabe, at the same time choosing a more well-treaded path. Willow uses TCP and is currently implemented in roughly 2.3k lines of Java code (excluding SQL code).

Willow defines its *domains*, much like the zones in Astrolabe, to be comprised of all the nodes that share the same prefix, i.e. that are in the same subtree of the KBT. Following this definition, every node owns its own domain, while its parent domain consists of a node and its sibling.

Like the zones in Astrolabe, every domain elects both a candidate and a contact for itself. The contact is the younger of the two child nodes (or child contacts), whereas the candidate is the older. Thus Willow uses a dynamic election scheme based on age. The contact of a domain is responsible for letting its sibling know about any updates which will then disseminate them down its own subtree.

Willow comes equipped with a tree-healing mechanism but did not inherit Astrolabe's security features.

SDIMS

The ***S****calable* ***D****istributed* ***I****nformation* ***M****anagement* ***S****ystem* [52, 53] is a system based on FTTs, which is developed at the University of Texas at Austin. It is implemented in Java using the FreePastry framework [41] and has been evaluated on a number of departmental machines as well as on 69 PlanetLab nodes.

The authors state that they designed it to be a basic building block for a broad range of large-scale, distributed applications. Thus, SDIMS is meant to provide a "distributed operating system backbone" to aid the deployment of new services in a network. Flexibility in the context of SDIMS means that it does not assume anything about properties of your data a-priori, for example the update-rate of variables. Instead, up-k and down-j parameters are given while registering a query.

SDIMS respects administrative domains for security purposes by using what is called an Autonomous DHT (ADHT). It does so by introducing superfluous internal nodes into the FTT whenever the isolation of a domain would otherwise have been violated. Running a query is driven by three functions; install, update and probe.

- *Install* registers an aggregate function with the system. It has three optional attributes: Up, down, and domain.
- *Update* creates a new tuple.
- *Probe* delivers the value of an attribute to the application. It takes four optional arguments: Mode $\in [continuous, one - shot]$, level, up, and down.

SDIMS does not, however, split the aggregation function further into three smaller operators, like in Tiny AGgregation service (TAG).

The current prototype neither implements any security features nor restricts resource usage of a query. Future implementations are planned to incorporate both.

SOMO

The ***S**elf-**O**rganized **M**etadata **O**verlay* [54], which is developed in China, is a system-metadata and communication infrastructure to be used as a *health monitor*. A health monitor can be used to monitor the "health" of a distributed system, meaning it provides information about the system's state (good or bad).

It relies on a so-called *Data Overlay* that allows overlaying arbitrary data on top of any DHT. It facilitates hosting of any kind of data structure on a DHT by simply translating the native pointers into DHT keys (which are basically just pointers to other nodes). To work on a data overlay a data structure has to be translated in the following way:

1. Each object must have a key, and
2. for any pointer A store the corresponding key instead.

Self-Organized Metadata Overlay (SOMO) also stores a last known host along with the key to serve as a routing shortcut. This data overlay on a fairly static P2P network has the ability to give applications the illusion of almost infinite storage space.

SOMO builds this structure, which is very similar to a KBT, on top of an already existing DHT in the following way. The ID space is divided into N equal parts, with $N = Num(Nodes)$. A SOMO node is responsible for one of these parts. From this range a key will be derived in a deterministic way. The default algorithm in SOMO is to take its center. The SOMO node will then be hosted by the DHT node which owns this key.

The SOMO KBT starts out with only one DHT node which will host a SOMO node responsible for the whole ID-space. As soon as a function periodically executed by each SOMO node detects that the ID range, for which its hosting DHT node is responsible, is smaller than the range it feels responsible for, it will assume that new DHT nodes have joined and spawn new SOMO nodes for them. Thus the tree will grow. The root node will remain responsible for the whole space on level 0 but in lower levels there might be other DHT nodes responsible for certain parts, depending on their location in the ID-space. This scheme does almost, but not exactly, resemble my description of a tie-breaking scheme, with a static algorithm set to the center of the range.

This approach has the disadvantage that it might not react to membership changes as fast as a normal KBT algorithm would.

To gather system metadata each SOMO node will periodically collect reports from its children. If it is a leaf it will simply request the data from its hosting DHT node. Once the aggregate arrives at the root, it is trickled down towards the leafs again.

SOMO can be used for a variety of purposes. It is interesting in its novel approach of layering another abstraction on top of a DHT, which allows it to be somewhat independent of the DHT and have a much simpler design.

Summary

Aggregation overlays are the closest ancestors to what I set out to do. Astrolabe is more a D-DBMS than an aggregation overlay but allows the easy installation of aggregation functions which is why I put it in this section. It uses a gossip-style protocol which is the oldest and most ineffective approach by modern standards.

The available information about Willow is too sparse to be able to even tell if it could be used as a monitoring system, and even less so, how. SDIMS seems to be suitable for this purpose, but still assumes certain attributes about the data that an application needs to aggregate, for example that it has a primitive type and can be aggregated by a function from a small set of prefix functions. SOMO is a much more broader toolkit that can be used to store basically anything in a DHT that can be expressed by reference-types.

2.6.4 Sensor Networks

Sensor Networks are designed to provide a "surveilance network" built out of commonly small, unmanaged, radio-connected nodes. The nodes have to be able to adapt to an unknown (possibly hostile) territory, organize themselves in a fashion that allows routing and answer queries about or monitor the state of their environment and trigger an action on the occurrence of a specific condition. This section introduces the two most complete research projects in this field.

IRISNet

The ***I**nternet-scale **R**esource-**I**ntensive **S**ensor **Net**work Services* [27] project at Intel Research is one of the many projects under the hood of IRIS [34]. It is closer to a Sensor Network than the other systems in this survey, which are more comparable to traditional monitoring systems. Prior research of Sensor Networks [2, 21, 25, 33] has primarily focused on the severe resource constraints such systems traditionally faced. IrisNet broadens the definition to include richer sensors, such as internet-connected, powerful, commodity

PCs. It provides software infrastructure for deploying and maintaining very large (possibly-planetary-scale) Sensor Networks adhering to this new definition.

IrisNet is a 2-tier architecture. It decouples the agents that access the actual sensor from a database for those readings. Agents that export a generic interface for accessing sensors are called Sensing Agents (SAs). Nodes that make up the distributed database that stores the service specific data are called Organizing Agents (OAs). Each OA only participates in a single sensing service. However, a single machine can run multiple OAs. IrisNet's authors chose eXtensible Markup Language (XML) for representing data because it has the advantage of self-describing tags, thus carrying the necessary metadata around in every tuple. Queries are represented in XQuery because it is the most widely adopted query language for XML data.

Some definitions of P2P demand that a system be called P2P only if it has an address-scheme independent from DNS. According to this definition, IrisNet (as well as Astrolabe) is not P2P since its routing scheme relies on DNS. It names its nodes according to their physical location in terms of the real world. Each OA registers the names of all the SAs that it is responsible for with DNS. Thus it achieves a certain level of flexibility because node ownership remapping is easy, but, on the other hand, is dependent on a working DNS subsystem.

Routing queries to the least common ancestor OA of the queried data is not hard because that name is findable by just looking in the XML hierarchy. The OA that owns this part of the name space can then be found by a simple DNS lookup. If that OA cannot answer all parts of the query, it might send sub-queries to other OAs lower in the XML hierarchy. For parts they can answer, OAs also use partially matching cached results. If this is unwanted, a query can specify freshness constraints.

IrisNet lets services upload and execute pieces of code that filter sensor readings dynamically, directly to the SAs. This is called a *senselet*. Processed sensor readings are sent by the SA to any nearby OA, which will route it to the OA that actually owns this SA. By decoupling the SA from the OA, "mobile" sensors are made possible.

IrisNet's contributions lie more in the field of Sensor Networks, but there is an application of IrisNet to System Monitoring. *IrisLog* runs an SA on each PlanetLab node which uses Ganglia Sensors to collect 30 different performance metrics. Users can issue queries for particular metrics or fully-fledged XPath queries using a web-based form. The IrisLog XML schema describes which metrics should be gathered and to which OA they have to be sent. This is why IrisNet is included here.

TAG

The ***T****iny* ***AG****gregation service* for ad-hoc sensor networks [25], developed at UC Berkeley, is an aggregation system for data from small wireless sensors. It draws a lot of inspiration from Cougar [2] which argues towards sensor database systems. These so called *motes*, also developed at UC Berkeley, come equipped with a radio, a CPU, some memory, a small battery pack and a set of sensors. Their Operating System (OS), called TinyOS, provides a set of primitives to essentially build an ad-hoc P2P network for locating sensors and routing data. The mote wireless networks have some very specific properties that distinguish it from other systems in this survey. A radio network is a broadcast medium, meaning that every mote in range sees a message. Consequently messages destined for nodes not in range have to be relayed.

TAG builds a routing tree through flooding: The root node wishing to build an aggregation tree broadcasts a message with the level set to 0. Each node that has not seen this message before notes the sender ID as its parent, changes the level to 1 and re-broadcasts it. These trees are kept in soft-state, thus the root has to re-broadcast the building message every so often if it wishes to keep the tree alive.

The sensor DB in TAG can be thought of as a single, relational, append-only DB like the one used by Cougar [2]. Queries are formulated using SQL, enriched by one more keyword, `EPOCH`. The parameter `DURATION` to the keyword EPOCH, the only one that is supported so far, specifies the time (in seconds) a mote has to wait before aggregating and transmitting each successive sample.

Stream semantics differ from normal relational semantics in the fact that they produce a stream of values instead of a single aggregate. A tuple in this semantic consists of a $< group\_id, val >$ pair per group. Each group is timestamped and all the values used to compute the aggregate satisfy $timestamp < time\_of\_sample < timestamp{+}DURATION$.

Aggregation in TAG works similarly to the other P2P systems that use trees in this survey: The query is disseminated down the tree in a *distribution* phase and then aggregated up the tree in a *collection* phase. The query has a specific timeout in the form of the EPOCH keyword. Consequently, the root has to produce an answer before the next epoch begins. It tells its children when it expects an answer and powers down for the remaining time. The direct children will then subdivide this time range and tell their children to answer before the end of their timeout, respectively.

For computing the aggregates internally that are specified externally using SQL, TAG makes use of techniques that are well-known from shared-nothing parallel query processing environments [43]. These environments also require the coordination of a large number of independent nodes to calculate aggregates. They work by decomposing an aggregate function into three smaller ones:

- an initializer i,
- a merging function f, and
- an evaluator e.

i run on each node will emit a multi-valued (i.e. a vector) *partial-state record* $\langle x \rangle$. f is applied to two distinct partial-state records and has the general structure $\langle z \rangle = f(\langle x \rangle, \langle y \rangle)$, where $\langle x \rangle$ and $\langle y \rangle$ are two partial state records from different nodes. Finally, e is run on the last partial-state record output from f if it needs to be post-processed to produce an answer.

Another unique contribution of this paper is the classification of aggregate functions by four dimensions:

1. *Duplicate sensitive vs. insensitive* Describes a functions' robustness against duplicate readings from one sensor.

2. *Exemplary vs. summary* Exemplary functions return some representative subset of all readings, whereas summary functions perform some operation on all values.

3. *Monotonic* - Describes an aggregate function whose result is either smaller or bigger than both its inputs.

4. *State* - Assesses how much data an intermediate tuple has to contain in order to be evaluated correctly.

What makes this project interesting is not its description of TAG, because I think that its tree building and routing primitives are inferior to others presented in this survey, but its groundwork that is important in many ways to aggregation systems. The addition of an EPOCH concept to SQL is one such example, while the classification of aggregation functions along four axis is another.

Summary

Whereas both aggregation overlays and distributed DBMS can be used alongside an application that is to be monitored, sensor networks are designed to run by themselves and commonly focus on heavily resource-restricted hardware platforms. While IrisNet is an exception to this, it has no extension mechanism and can only be used to collect Ganglia data through IrisLog. The TAG paper is valuable because of its contribution but otherwise not usable as a monitoring framework.

Table 2.1: Overview of Related Systems

Name	Type	RO	AO	Deployed on	Language	Notes
PIER	query exec.	DHT	FTT	PlanetLab	UFL	Java Type System
Sophia	query exec.	static	static	PlanetLab	Prolog	
Astrolabe	hier. aggr. DB	DNS	gossip	-	SQL	Security
Willow	query exec.	KBT	KBT (dyn:age)	-	SQL	
SDIMS	hier. aggr. DB	ADHT	FTT	Dept. + PlanetLab	aggreg. : C	
SOMO	meta-data	DHT	KBT (stat:center)	-		host any ADT
IrisNet	SensorNet	DNS	OA	PlanetLab	XPath	“rich” sensors
TAG	query exec. (SN)	broadcast	simple tree	PlanetLab	enriched SQL	*mote*

2.6.5 Conclusion

Out of the systems presented in this section that are deployed on PlanetLab, only PIER and SDIMS (and maybe Sophia) are suited to do application level monitoring, albeit they are not making it easy. Both PIER and SDIMS were not designed with users of PlanetLab in mind. PIER was designed to be as close to a local relational query engine as a distributed one could be by giving up some of the ACID (atomicity, consistency, independence and durability) constraints. Its main purpose is therefore not aggregation, nor application-level monitoring. SDIMS is meant to be an aggregating overlay for system meta-data. As such it can be used for application-level monitoring by explicitly inserting a process' logs into the system and setting up an aggregation function to reduce the amount of data. Neither of these two systems (PIER and SDIMS) can be modified easily to execute user specified programs to make aggregation simple. Sophia is not running on PlanetLab any more and generally seems an unusual choice for system monitoring.

None of these choices currently concern themselves with security.

D-DBMS solve item 1 and some of them even solve item 2 of the problem. Aggregation systems obviously solve item 1, and Sensor Networks solve the third section of the problem (item 3).

Chapter 3

Approach

This chapter describes my approach from a high level, whereas the next chapter details the implementation and design and gives a short usage manual. My approach is structured after the problem delineation, as given in section 1.2.

3.1 General

To solve the problem of distributed collection, selection and aggregation of logged statistics of a P2P application the three problems given in section 1.2 have to be solved. To cope with the amount of data that has to be collected, it is unavoidable to reduce the data before it arrives at its destination to prevent saturation of resources.

To leave as much power in the hand of the user, I follow the approach of MapReduce and let the user specify his or her own collection, aggregation and evaluation functions. This can only work properly in an aggregation tree, however, if these operators implement associative functions, i.e. prefix functions.

For non-prefix functions, like for example subtraction, this will alter the result. PWHN copes with churn by using the data-structure we invented. It uses a Key-based MapReduce (KMR) to effectively and efficiently distribute the query to all nodes and uses the same tree for aggregation.

3.2 Distributed selection

As described earlier, centralized data collection is frequently unsuitable for monitoring the system state. Thus, as much processing of the data as possible, especially data selection, should be pushed out into the network towards the sources.

The question that remains is how to do this without assumptions about the data's structure a-priori. Traditional languages that are written for this, such as SQL, assume that the data fits into a relational schema. Others, that are used but not written specifically for data selection, like Prolog, are complicated and have a steep learning curve.

I decided to take an approach that leaves the choice of the language to the user. The operators in PWHN that are responsible for selection, aggregation and evaluation have the only restriction of being runnable programs. Thus, they can be programmed in the user's favorite programming, scripting, querying or logic language.

In this case this does not pose a security threat because the researcher using PlanetLab has complete control over his slice anyway and there is no point in uploading harmful code. There is one caveat: Since client-server communication is done without any encryption or security at the moment, anyone could connect to a running pwhn-server (provided he knows the right port number) and upload executable code. This should be addressed in future versions by introducing certificates and encryption.

3.3 Distributed aggregation

For the same reason as above, as much reduction of the data as possible should be pushed out into the network. We make the simplifying assumption that most of the operators are going to implement prefix operations, and thus can easily be distributed.

PWHN combines the strengths of both MapReduce (see section 2.2) and the ideas of shared-nothing, parallel query-processing environments (and thus such systems as TAG, cf. 2.6.4; and SDIMS, cf. 2.6.3).

It incorporates the strength of MapReduce by realizing that by restricting its operations to associative (i.e. prefix) functions, it does not lose much expressiveness; but on the other hand, gains an important advantage, that its operators can be pushed out into the network. While MapReduce requires its operators to map into and then do the reduction in a domain-reduced tuple-space (that is, each tuple has an explicit key), my system supports but does not require this.

It learns from the ideas of systems such as TAG by splitting the aggregation into three smaller, distinct operators; initialize, update, and evaluate. It has an advantage over those systems, in that it does not restrict those operators in any way (e.g. to SQL), besides the fact that they need to be executable programs.

3.4 Churn

A system that has to work in a P2P environment has to be able to cope with rapidly changing membership. It has to be able to heal its structures, such as the aggregation tree, when members leave; and adapt them, when new members join. It needs to reliably distribute the data (i.e. the programs) necessary for a query to all participants, even if the set of those participants changes during or after the initial distribution phase.

DHTs that are designed to find data (keys in this case), despite high churn, can be used to amend the problem of knowing which participants have left and which have joined. Our approach, however, does not attempt to adapt a statically built aggregation tree to a changing membership, but instead rebuilds the tree on every query. The underlying assumption is that the rate of churn is above the point that marks the equilibrium between healing and rebuilding the tree.

Therefore, our approach leverages DHTs to build a self-healing aggregation tree upon dynamic networks. I looked at several related work in this area and designed a data-structure that combines the strengths of both FTTs and KBT but tries to avoid some of their weaknesses. FTTs might introduce a hot-spot at the root node because they do not try to build an aggregation tree that is structured according to the current set of active nodes, but instead use the DHT to route; whereas KBTs only construct one global tree at start time and thus have the need for complicated healing algorithms. Moreover, a high amount of requests might also overload the global root.

After the initial distribution phase, my system transmits only hashes instead of the actual content, to be able to let new members detect that some content has changed.

Chapter 4

Implementation and design details

4.1 Design details

4.1.1 Data Structure

We seek a data-structure that combines the FTTs load-balancing feature with the determisn of a KBT. Since the FTT is defined by the entries of all participating nodes' fingertables, it is not predictable and may fluctuate frequently. This stems from the fact that the DHT needs some flexibility to be able to keep routing in the presence of churn.

A DHTs flexibility in routing means that it can pick the next hop from all those that have one more matching prefix bit. Thus the length of the prefix is inversely proportional to the number of candidate nodes. Consequently, the length of the *suffix* (the remaining bits after the prefix) is directly proportional to the number of eligible nodes. For k suffix bits the number of candidates is 2^k.

This leads to a solution which consists of a KBT rooted at a particular key. Since the number of candidates is directly proportional to the length of the suffix and we would like this number to be exactly one, we need to reduce the length of the suffix. This can be done by "fixing" it to the current node's suffix.

For example, in a "normal" DHT, a node with ID$=000$ that wants to route a message to Key(111) has four choices for the next hop, those that match $1XX$. Fixing the prefix and suffix to those of the current nodes, while still flipping the next bit (i.e. adding one more prefix bit) essentially reduces the number of routing candidates to one. Continuing the example, node(000) will have to route the message to node(100) because that corrects the next bit of the (currently empty) prefix and keeps the same suffix. This ensures that the resulting tree will be balanced and have a strong upper bound on the branching-factor of $len(ID)$ for a fully populated ID-space. This data-structure bears strong similarity to MapReduce, albeit keyed with a certain ID, thus I call it Key-based MapReduce (KMR).

A KMR is a subset of a KBT because a KMR is fully described by the set of nodes and the root key, whereas the KBT can have any permutation of the nodes with the constraint that all nodes in a subtree have the same bit at the respective level of the subtree's root. For n levels $a = 1 \cdot 2^n + 2 \cdot 2^{n-1} + \cdots + 2 \cdot 2^2 + 2 \cdot 2^1$ different KBTs are possible. Specifically, this means that a KBT with a set of nodes has a chance of $p = \frac{1}{a}$ to look like the KMR with the same set of nodes.

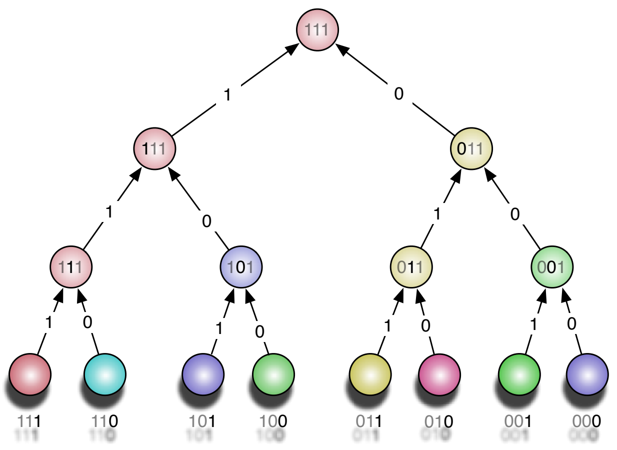


Figure 4.1: Example of a KMR (the flipped bit is highlighted at each level) for a 3-bit wide ID-Space, all nodes present, for Key(111)

(For homogeneity I will always represent KMRs such that the root node is the left child, thus the right child always has the corresponding bit of the root flipped. This structure is sometimes called a *Left-Tree*.)

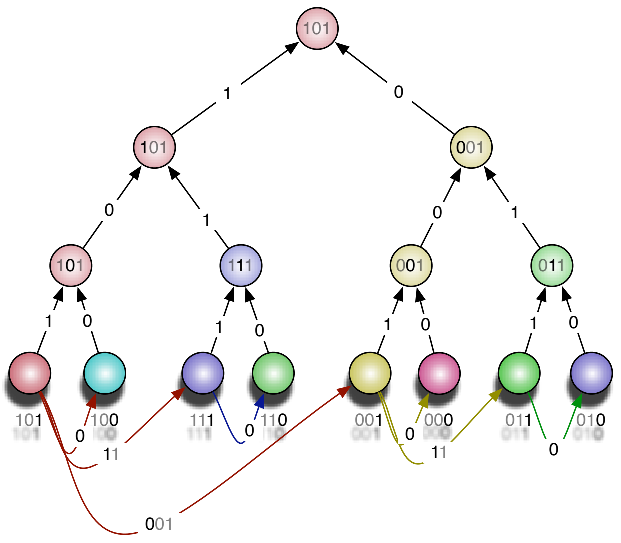


Figure 4.2: Explanation of the dissemination phase of a KMR rooted at Key(101)

The long way down

In the dissemination phase, a message has to be routed from the root of the tree to all leafs. It is done recursively. This is particularly easy due to the characteristic of a KMR that each internal node is its own parent if it has the same suffix as the root at the corresponding level, which in my representations will always be the left child. Each parent (root of a subtree) will be its own left child all the way down, and thus only needs to send the message to each of its (right) siblings. Since the length in bits of the suffix of a node will determine its level in the tree, this is the number of messages it needs to send. Thus, every node sends exactly P messages, where P =len(suffix), to other nodes according to the following algorithm.

Listing 4.1: Algorithm for the dissemination phase of a KMR

```
for(k = len(suffix); k >= 0; k--)
        route( OwnID ^ (1 << k) );
```

The algorithm starts at the node with the longest matching prefix and flip bits in the prefix “from the front,” thus doing the opposite from the aggregation phase: subtracting bits of the matching suffix. The algorithm sends as many messages as there are bits in the prefix.

Unfortunately, most DHTs will not have a fully populated ID-space. To prevent nodes from trying to route to a large number of non-existent nodes, which in a normal DHT would end up storing the same value over and over under different keys at a few nodes, an application building a KMR on a DHT would try to get access to the underlying fingertable and KBR Layer. There are two ways to solve this dilemma.

First, adding a single function to the DHTs interface allows an application to build a KMR on it. Since the application needs to find nodes without actually storing data on them, which would be the (unwanted) side-effect of a `put`, the function takes a key and returns the closest node to that key. Thus, it is called FIND_NODE.

After the parent of a subtree has determined the closest node to one of its children it was looking for, it just sends the message to that node instead (let us call that node A). This node has to assume the role of its non-existent parent and resends the message to all the children that its parent would have sent it to. Fortunately, due to the fact that the node, by virtue of getting this message for its parent, now knows that it is the closest ancestor to its parent, it is able to discern which of its siblings cannot be there. These are all the nodes who would have been closer to the original parent. Thus, A only has to resend the message to all of the direct children (of its parent) whose IDs are further from their parent's ID. In the representation I use in this paper, these happen to be all of A's parent's children to A's right.

The aforementioned leads to the second solution. If the application has access to the KBR and has the ability to route messages without side-effects it does not need FIND_NODE. All the messages that the node tries to send to non-existent nodes will either return to the sender or the next closest node to its right. An application keeps track of received messages and disregards messages that it had already seen. The node that receives messages destined for a non-existent node not send by itself knows that it has to assume its parents role and continues as outlined above. Should a node get the message that it itself has sent to one of its children, it can immediately deduce that there can be no more nodes in that subtree because all nodes in a subtree share the characteristic of having the same next bit; thus *any* of these nodes would have been closer to any other.

A KMR only *looks* binary in a certain representation, but on the other hand, it provides a strong upper-bound on its *arity* for each node, whereas KBTs/FTTs do not. This *arity* is the width of the ID-space - 1 for the root of the tree and decreases by 1 at each level. The first approach previously outlined has the ability to find existent nodes and will thus only send necessary messages. The second approach has to blindly send the maximum number of messages, many of which will end up at the same node. As soon as the next node down the chain determines all those nodes that cannot be there, this cuts down the number of messages that need to be sent. It still has to send messages to all of its parent's children that are to its right.

The stony way up

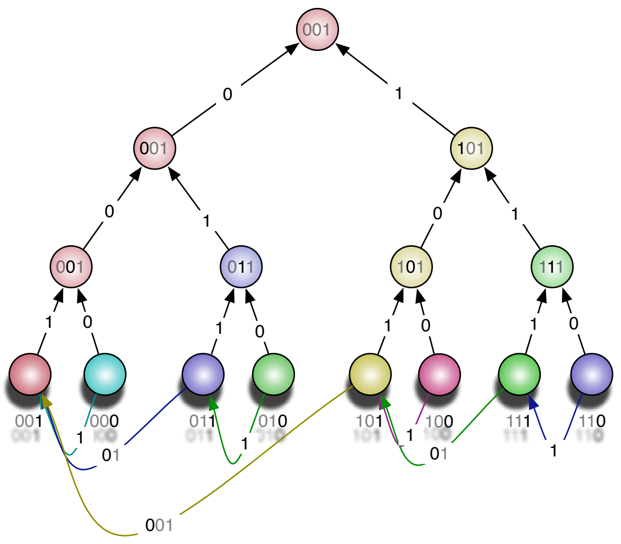


Figure 4.3: Graphic explanation of the aggregation phase of a KMR rooted at Key(001)

The aggregation phase works exactly opposite to the dissemination phase. To work the way up the tree, every node only needs to send one message but has to wait for as many messages as it sent in the dissemination phase to arrive. An estimate of the number of messages can be made using the global number of nodes as a statistical hint, if that number is unknown using the distance to the root as a rough estimate, or by using the fingertable. Then, each node sends one message to its immediate parent in the tree according to the following algorithm.

Listing 4.2: Algorithm for the aggregation phase of a KMR

```
route( OwnID ^ (1 << len(suffix) );
```

In a KMR routing "up" the tree is done by fixing both pre- and suffix and flipping bits in the suffix "from the rear," i.e. adding bits to the front of the matching suffix.

Since, as already mentioned above, the ID-space will most likely not be fully populated, this algorithm will again end up trying to route a lot of messages to non-existent nodes. This can be avoided in the following way.

A naive way is to remember where the message in the dissemination phase came from and just assume that the parent is still good while aggregating.

4.1.2 Systems leveraged

FreePastry

FreePastry [9] is an open-source implementation of the Pastry DHT [41] in Java. Pastry is very similar to the original Chord with added locality and more freedom in choosing its fingers. Its routing can be seen as passing a message around a circular ID-space (a ring) consisting of 160-bit IDs. Nodes store pointers to specific locations on the ring relative to their own, commonly half a revolution away, a quarter revolution away and so on or their immediate successors if those nodes do not exist. Pastry chooses as its fingers the same set of nodes that Kademlia DHTs choose from; all those that match the next bit flipped.

I choose to implement two trees on FreePastry because it is open-source and easy to use.

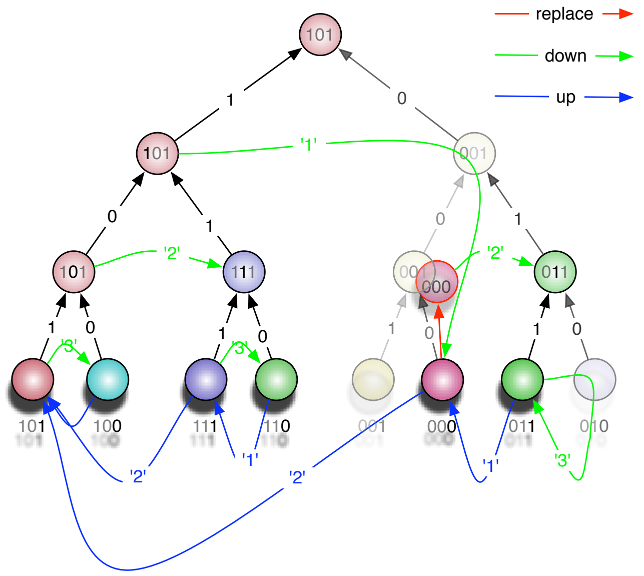


Figure 4.4: Summary of the both the aggregation phase and the dissemination phase of a KMR rooted at Key(101)

Furthermore, the fact that it exports the common Application Programming Interface (API) [6] comes in very handy when implementing these types of trees.

For a more extensive specification of FreePastry please refer to Appendix C.

4.2 Implementation

4.2.1 PWHN

PWHN is the name of my implementation of a MapReduce-like system for PlanetLab.

System requirements

To run PWHN the client and server need to be able to run java programs. The required java version is at least 1.4.2. All required java libraries are included in the distribution.

For conveniently starting and stopping the server using the client Graphical User Interface (GUI), sshd is required; to be able to use the automatic syncing feature if a node's files are out of date, rsync needs to be installed (no running rsync server is required). If execution of the user supplied scripts need a runtime environment or script interpreter, obviously this needs to be installed, also. The client uses both the *JFreeChart* [18] and *jgraph* [19] libraries for rendering the plots and graphs. Both are included in the distribution as binary libraries. Their respective source codes can be downloaded from the project websites.

Licenses

Both JFreeChart and jgraph are released under the Lesser General Public License (LGPL) version 2.1.

System description

Like other systems, for example SDIMS (2.6.3) and TAG (2.6.4), PWHN splits the aggregation into three smaller operators:

1. Init,
2. Update, and
3. Eval.

In contrast to other systems, these can be executable programs such as scripts with a working shebang. Only the Initializer is really necessary. In case the Aggregator is not given, the results are concatenated and if no Evaluator is given, the current result set is returned.

Data is passed from stage to stage in the following way. The Initializer writes a byte stream to its `stdout`. The Aggregator is called and the outputs from two distinct Init runs are given to it on its `stdin`. The data of each run starts with its length in bytes as an ASCII string on a line by itself and is followed by a newline. This would be an example of a valid input to the Aggregator.

```
11\n
(1,3.0,2.0)\n
12\n
(1,1.65,0.8)\n
```

The Aggregator is expected to deliver the aggregated form of its two inputs on `stdout`. The Evaluator should expect the last output of the Aggregator, or all outputs of the Init concatenated on its `stdin` and print its output to `stdout`.

PWHN scans all outputs from Init and will discard empty strings, which means it will not call the Aggregator of Evaluator with an empty input. If the scripts print and accept their in- and outputs with a trailing newline, however, is totally up to the user; PWHN does not make any assumptions about the form of the data. If, for example, the Initializer uses the python `print` command for output, care has to be taken in the Aggregator to delete the trailing newline using the `rstrip()` function.

To test if this might work quickly and efficiently, I implemented two different versions.

Script Version The first version is centralized and is written in python. Centralized, because it does not use a tree. Instead, it connects to all hosts using Secure SHell (SSH), remotely executing the requested Init file. After they all have terminated, it executes the Aggregator and Evaluator locally, if given.

All necessary options are supplied through the command line. It will do two things: rsync the files with the given set of IPs and then execute the Init file remotely and the others locally. It allows specifying a directory rather than single files which will get rsynced. If a directory was specified all three files have to be in that directory. The commands (rsync and SSH) are executed on all servers in parallel using vxargs, which is a script written in python specifically to execute one command on many PlanetLab nodes. Thus, a lot of the command-line arguments of PWHN are handed directly to vxargs. For a complete description of all arguments and some screenshots see Appendix E.

Java Version The Java version consists of two parts:

1. The tree surrogate that runs on my PlanetLab slice forms a FreePastry ring, and
2. the local client tool that connects to the tree requests a query and presents the results graphically.

The client allows the user to you specify all necessary options in a nice GUI and is able to run the centralized PWHN script, its own "native" Java implementation of the script using threads instead of vxargs, or the three different tree versions. The tree versions use the first given IP as a *entry-point* to connect to the FreePastry ring.

The GUI is described in the next section, whereas the server and tree architecture are outlined in the following two sections.

4.2.2 Client GUI

A superset of this description is available at [32].

The GUI mirrors the design of the script by letting the user select the three executables/scripts. The first tab selects the IPs to work with, a username for SSH, the method (script, native, or one of the three tree types), and a name for the run. The last tab displays the results in various formats. When started it looks like Figure 4.5.

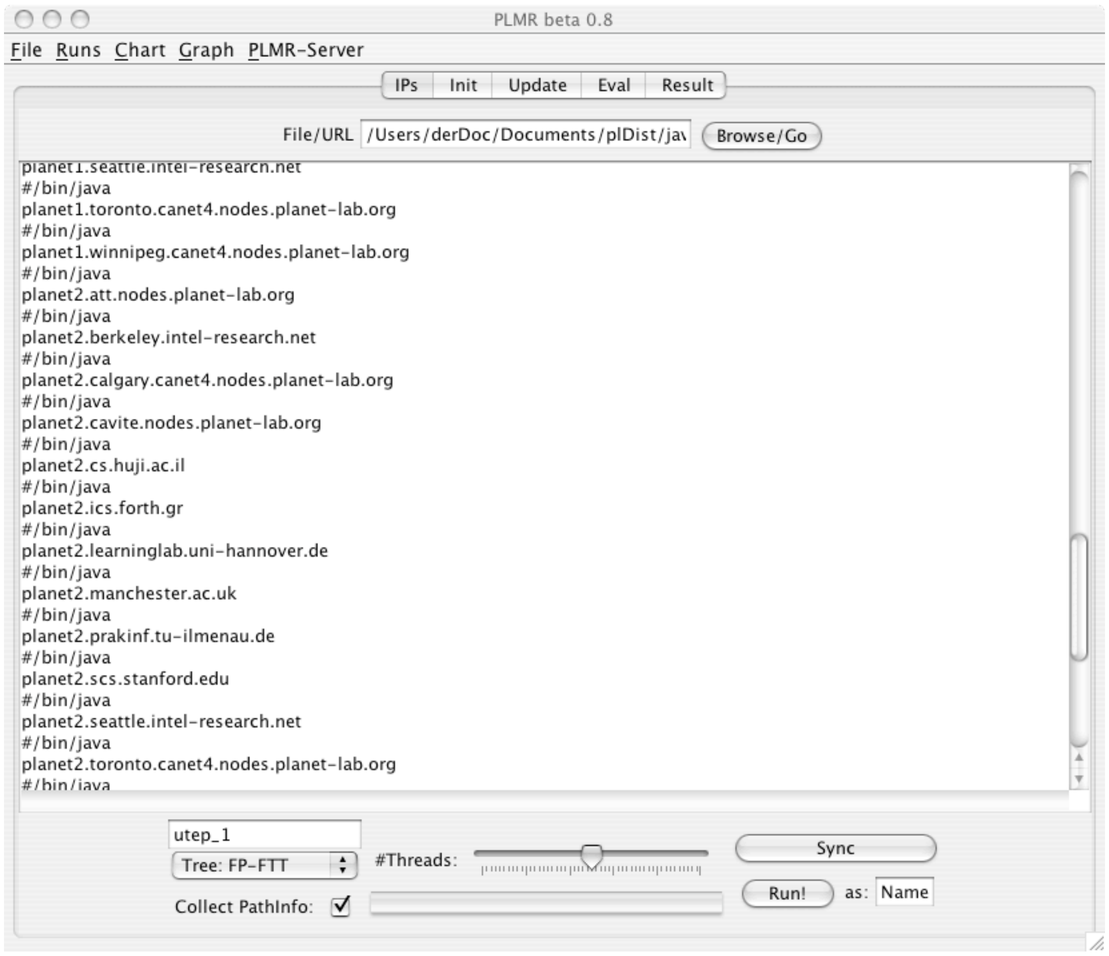
The "File" menu allows the user to store and later reload a full project including all files, names, and runs.

The "Runs" menu allows one to delete selected runs, save them as plain text files and change the run with the category.

The "Chart" menu allows one to reset the zoom on the chart and save the chart/graph as an image.

The "Graph" menu allows one to zoom the graph in and out.

The "PLMR-Server" menu allows one to sync the files needed for the PWHN server, start and stop it. It operates on the currently selected list of IPs. The *start* command


PLMR beta 0.8
File Runs Chart Graph PLMR-Server
IPs Init Update Eval Result
Browse/Go
Sync
Run!

File/URL /Users/derDoc/Documents/plDist/jav

```
planet1.seattle.intel-research.net
#/bin/java
planet1.toronto.canet4.nodes.planet-lab.org
#/bin/java
planet1.winnipeg.canet4.nodes.planet-lab.org
#/bin/java
planet2.att.nodes.planet-lab.org
#/bin/java
planet2.berkeley.intel-research.net
#/bin/java
planet2.calgary.canet4.nodes.planet-lab.org
#/bin/java
planet2.cavite.nodes.planet-lab.org
#/bin/java
planet2.cs.huji.ac.il
#/bin/java
planet2.ics.forth.gr
#/bin/java
planet2.learninglab.uni-hannover.de
#/bin/java
planet2.manchester.ac.uk
#/bin/java
planet2.prakinf.tu-ilmenau.de
#/bin/java
planet2.scs.stanford.edu
#/bin/java
planet2.seattle.intel-research.net
#/bin/java
planet2.toronto.canet4.nodes.planet-lab.org
#/bin/java
```

utep_1
Tree: FP-FTT
#Threads:
Collect PathInfo: ✓
as: Name

Figure 4.5: The client GUI after startup

starts PWHN on the first IP first, waits five seconds and then starts it on all the other IPs given, supplying them with the IP of the first node to connect to.

Note that PlanetLab-MapReduce (PLMR) was an earlier name for this project and has been discontinued in favor of PWHN.

Since all four tabs for selecting files look very much like the IP/Servers tab, I am going to explain them all at once. Instead of selecting three distinct files for the phases PWHN supports 3-in-1 files which are similar to unix startup-scripts and contain all phases in one file. They expect to be told their role on the commandline, by supplying one of [gen,agg,eval].

All four tabs display the contents of the currently selected file, let the user change them, or accept typing in new content which gets saved as a new file suffixed with the current UNIX-timestamp in milliseconds. The box above the text filed will always reflect the current filename. The IP tab also allows entering full Uniform Resource Locators (URLs) as filenames.

If the filename field is empty, a click on the adjacent button opens a new file chooser dialog to select a filename, otherwise it will try to open the entered file or fetch the URL.

IP/Servers tab

In this tab the user specifies a list of servers for the flat algorithms. The tree versions technically do not need this list of IPs any more to execute a query, but it is nice-to-have to be able to start the server on all the nodes through the GUI. The current implementation sends this list around with every query to be able to select on which nodes to execute a query, but this is not strictly necessary and could be easily changed.

The “sync” button executes an rsync with the current set of servers to transfer either the chosen directory or both the Init and Agg files. The rsync can use either the script or the native implementation. A *BuildMessage* (cf. subsection 4.3.3) for building a tree can only accommodate transporting single files, and thus can only execute non-synced runs if no directory has been selected. Warnings are issued if the user attempts otherwise.

Once a sync has been executed and none of the files or the IP list have changed, the GUI considers itself in sync with this set of IPs. In synced state, a flat or native run can be started and tree runs will include hashes rather than the contents of the files itself. In case a directory is selected, it is hashed recursively. Hashing is done with the help of a python script called *hasher.py* (see subsection 4.3.1).

Result tab

The result tab displays the results of all runs in various formats. In its pristine state it looks like Figure 4.6.

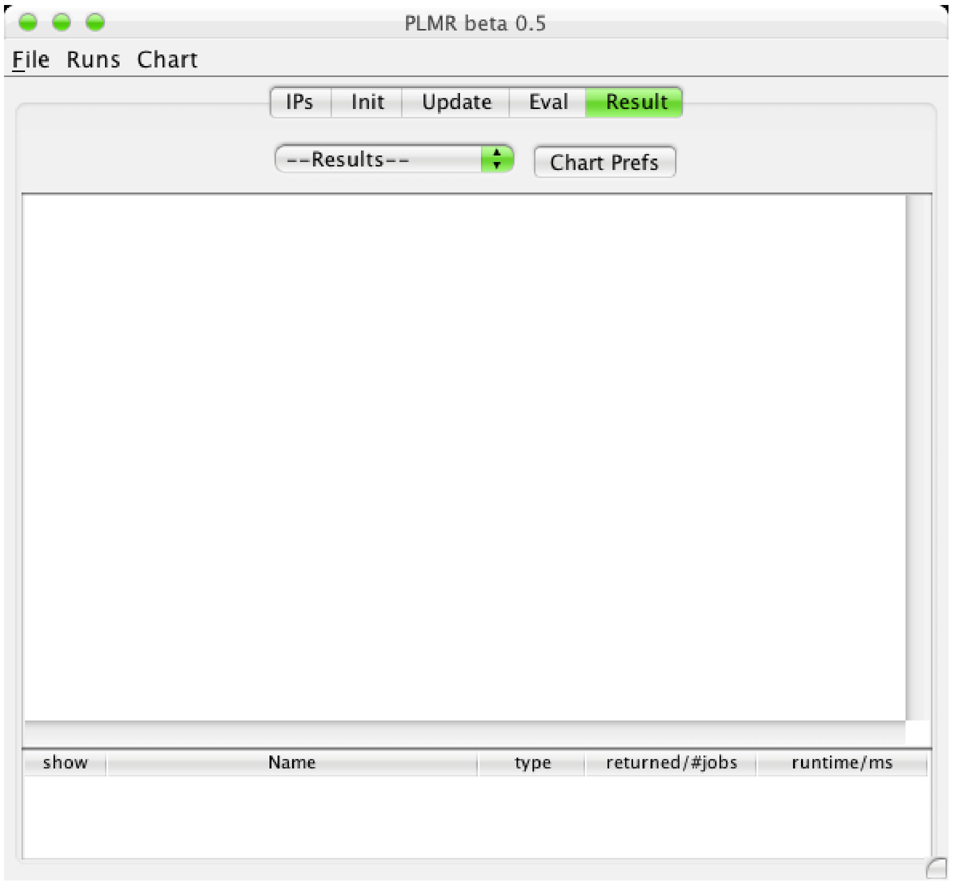
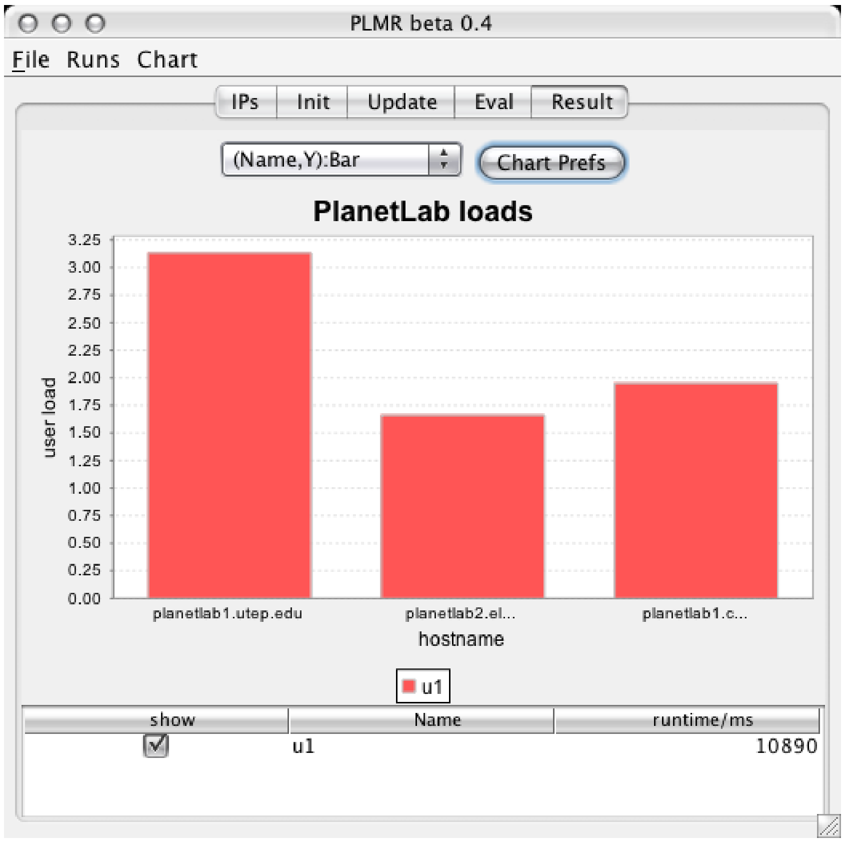

PLMR beta 0.5
File Runs Chart
IPs Init Update Eval Result
--Results--
Chart Prefs
show Name type returned/#jobs runtime/ms

Figure 4.6: The client GUI displaying the result tab

The combo-box at the top lets the user select one of 10 different ways of displaying the current result or one of three ways of displaying the extra data that this run is annotated with (information about the tree). At the bottom, information about the runs is displayed including a name if it had one or just the time of the run, the type, the runtime and the number of jobs returned vs. number of jobs requested. It also allows hiding of runs.

Three main type of result sets are supported for plotting, all of which have to be tuples enclosed in "()." They are:

- $(Name\langle string\rangle, Y - Val\langle real\rangle)$ which can be plotted as a bar chart, a Line-, a Dot- and an Area-chart. Figure 4.7 is an example of a Bar-Chart.

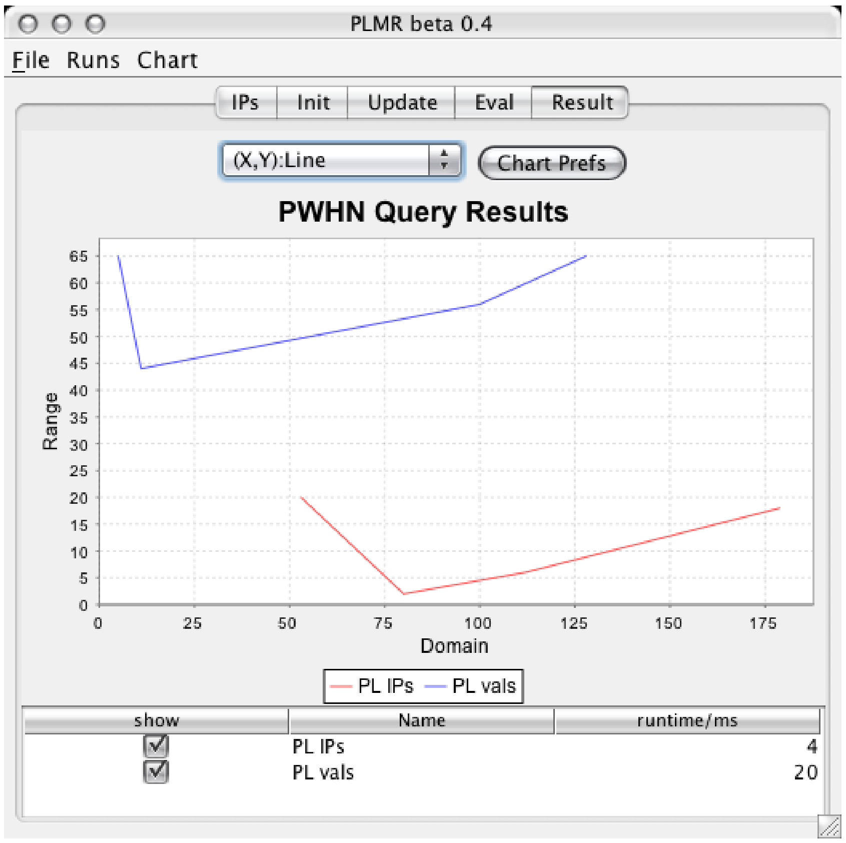


show	Name	runtime/ms
✓	u1	10890

Figure 4.7: The result tab showing a Bar-Chart

- $(X - Val\langle real\rangle, Y - Val\langle real\rangle)$ which can be plotted as a bar chart, a Line-, a Dot- and an Area-chart. Figure 4.8 is an example of a Line-Chart.

- $(X - Val\langle real\rangle, Y - Val\langle real\rangle, Z - Val\langle real\rangle)$ can be plotted as bubble chart, in which the size of the bubble is proportional to the z-value, or as color-chart where the color represents the z-value, ranging from green to red. Figure 4.9 is an example of a Bubble-Chart.



show	Name	runtime/ms
✓	PL IPs	4
✓	PL vals	20

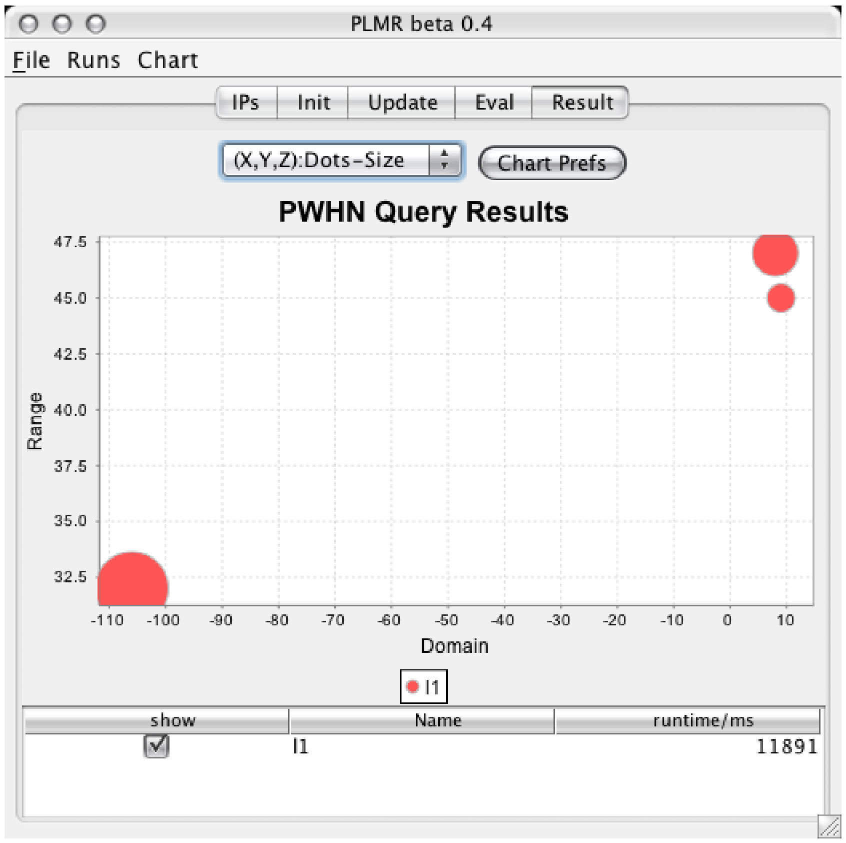
Figure 4.8: The result tab showing a Line-Chart

The chart preferences, which are accessible through the button with the same name, allow setting all the names and captions of the chart, the ranges for both the x- and y-axis, as well as a background picture.

4.2.3 Path Info

See 4.3.3. The optionally collectable path-infos are used to calculate some general statistics about a run, as well as to draw the paths on a 2d-world map and to visualize the shape of the tree. The results tab also allows to look at them in plain text, which simply prints all the information in the root PathInfo class and then its children recursively.

The general stats, which are also calculated for native runs, contain the following infor-



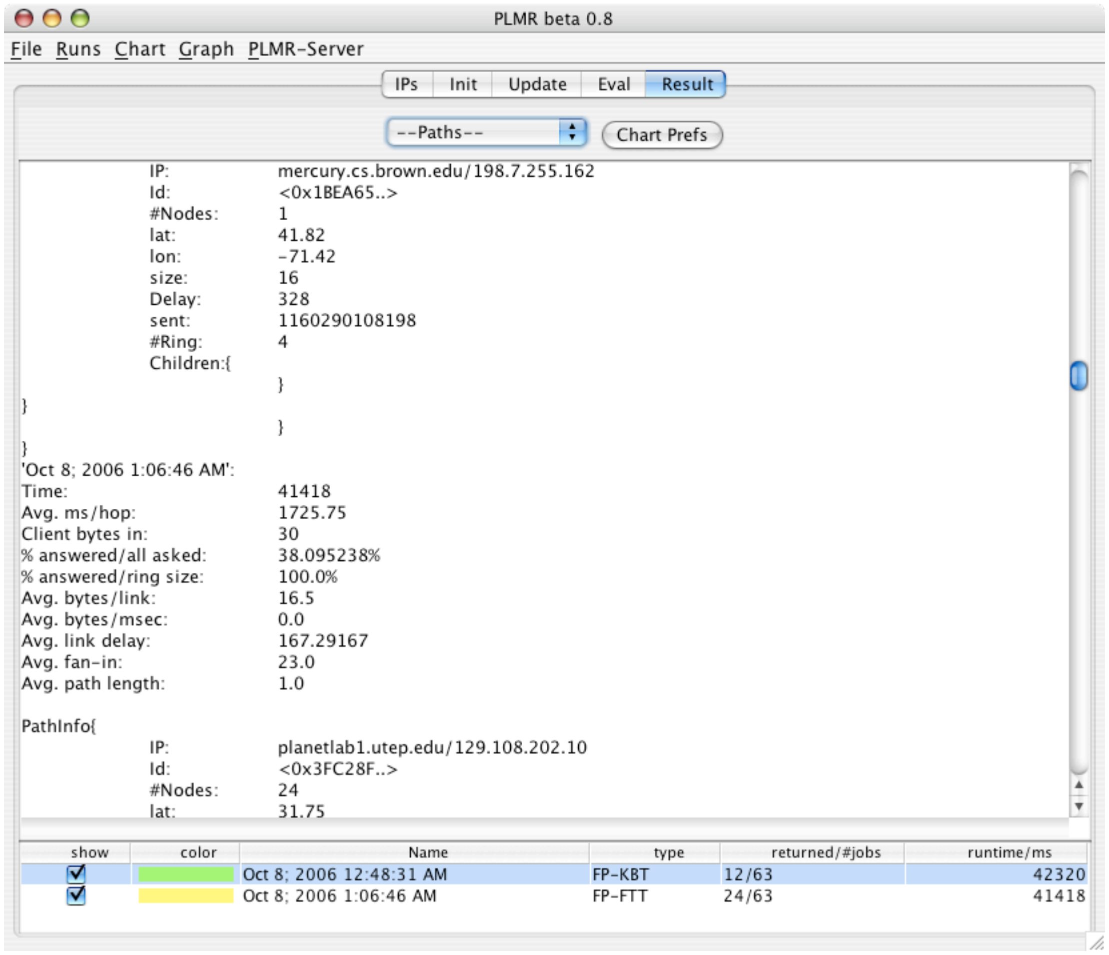
show	Name	runtime/ms
✓	I1	11891

Figure 4.9: The result tab showing a Bubble-Chart

mation (stats from native run of subsection 5.3.1)

```
'Oct 1; 2006 2:50:00 PM':
Time:                  138681
Avg. ms/hop:           602.9609
Client bytes in:       3673
% answered/all asked:  100.0%
Avg. bytes/link:       15.969565
Avg. bytes/msec:       0.001758027
Avg. link delay:       9083.8
Avg. fan-in:           230
Avg. path length:        1

(8.033870,9.247825)
```

The information stored within the PathInfo class can be displayed in textual form as shown in Figure 4.10.


PLMR beta 0.8
File Runs Chart Graph PLMR-Server
IPs Init Update Eval Result
--Paths--
Chart Prefs

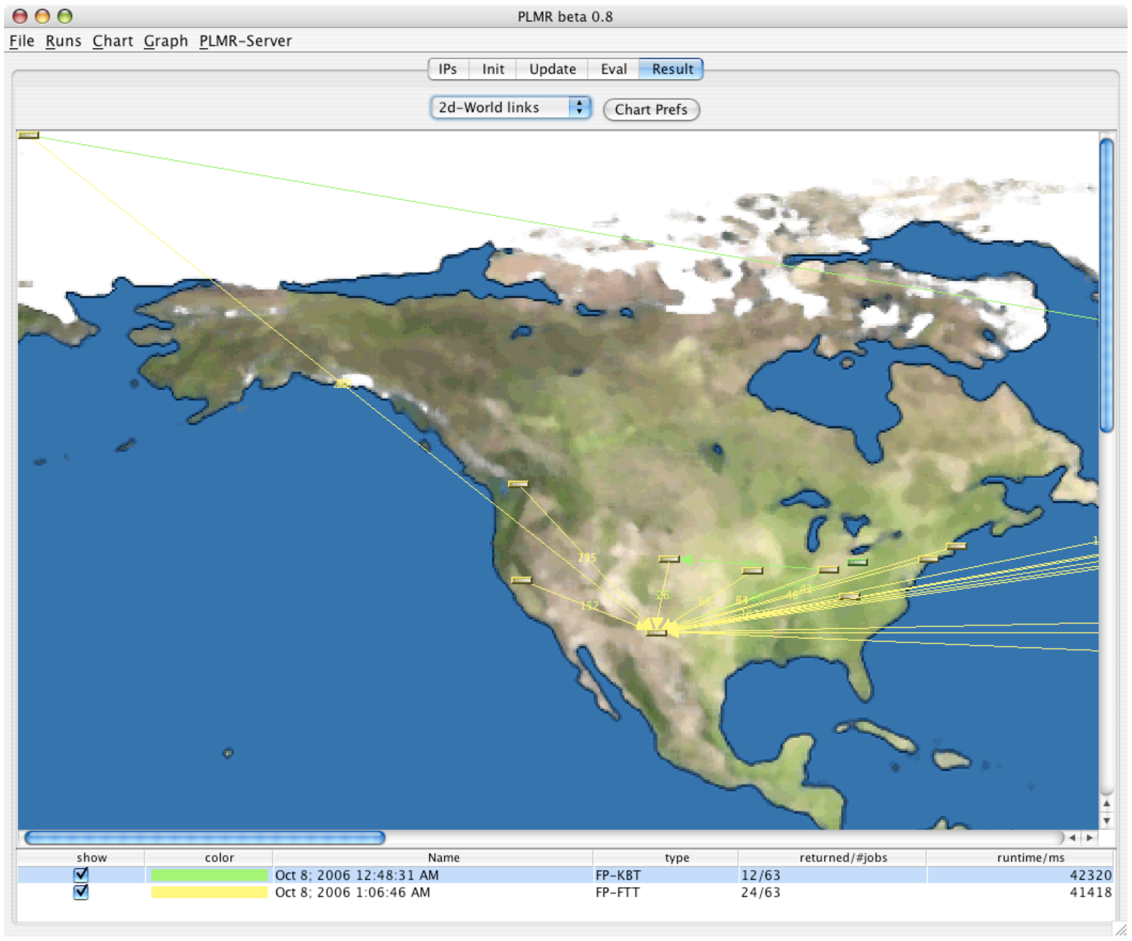
```
                IP:           mercury.cs.brown.edu/198.7.255.162
                Id:           <0x1BEA65..>
                #Nodes:       1
                lat:          41.82
                lon:          -71.42
                size:         16
                Delay:        328
                sent:         1160290108198
                #Ring:        4
                Children:{
                              }
}
                              }
}
'Oct 8; 2006 1:06:46 AM':
Time:                         41418
Avg. ms/hop:                  1725.75
Client bytes in:              30
% answered/all asked:         38.095238%
% answered/ring size:         100.0%
Avg. bytes/link:              16.5
Avg. bytes/msec:              0.0
Avg. link delay:              167.29167
Avg. fan-in:                  23.0
Avg. path length:             1.0

PathInfo{
                IP:           planetlab1.utep.edu/129.108.202.10
                Id:           <0x3FC28F..>
                #Nodes:       24
                lat:          31.75
```

show	color	Name	type	returned/#jobs	runtime/ms
✓	🟩	Oct 8; 2006 12:48:31 AM	FP-KBT	12/63	42320
✓	🟨	Oct 8; 2006 1:06:46 AM	FP-FTT	24/63	41418

Figure 4.10: Result tab showing the PathInfo in textual form

Figure 4.11 is an example of the paths drawn on a world-map. Note that the nodes in the upper left corner have no coordinate information associated with them yet.

The tree can be drawn in its actual form as shown in Figure 4.12, which should be used


PLMR beta 0.8
File Runs Chart Graph PLMR-Server
IPs Init Update Eval Result
2d-World links
Chart Prefs

show	color	Name	type	returned/#jobs	runtime/ms
✓	🟩	Oct 8; 2006 12:48:31 AM	FP-KBT	12/63	42320
✓	🟨	Oct 8; 2006 1:06:46 AM	FP-FTT	24/63	41418

Figure 4.11: Result tab showing the PathInfo drawn on a 2d-world map

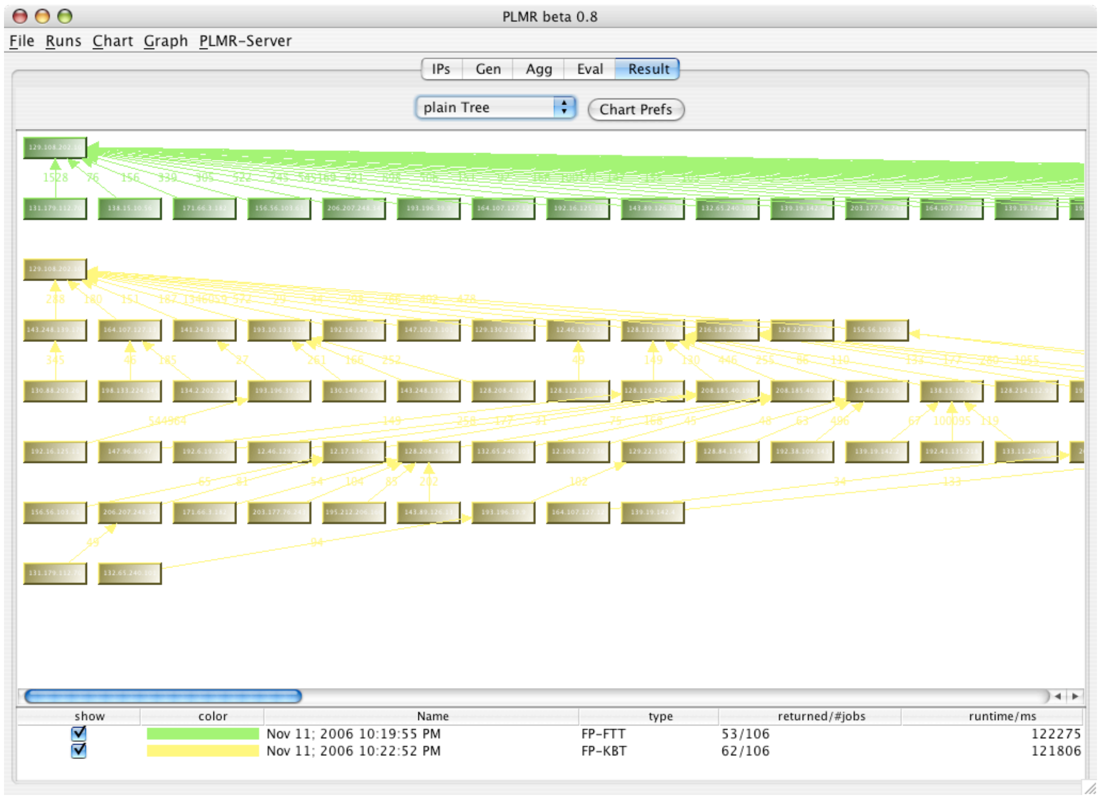
with FTTs.

4.2.4 Sample experiment

This sample shows how to run an experiment using the first set of scripts from subsection 5.3.1 on a random set of 10 nodes using a KMR.

The scripts calculate the average load over all servers, and the load's standard deviation. Let us assume that the file nodes10.txt contains the following lines:

```
planet2.prakinf.tu-ilmenau.de
```



show	color	Name	type	returned/#jobs	runtime/ms
✓	🟩	Nov 11; 2006 10:19:55 PM	FP-FTT	53/106	122275
✓	🟨	Nov 11; 2006 10:22:52 PM	FP-KBT	62/106	121806

Figure 4.12: Result tab showing the PathInfo drawn as a tree showing the aggregation tree's structure

```
planet2.ottawa.canet4.nodes.planet-lab.org
planet2.scs.stanford.edu
planet2.toronto.canet4.nodes.planet-lab.org
planet3.cs.huji.ac.il
planet3.pittsburgh.intel-research.net
planet4.berkeley.intel-research.net
planet4.cs.huji.ac.il
planet5.berkeley.intel-research.net
planet6.berkeley.intel-research.net
```

Start a shell and make sure that it is able to log into the machines remotely without asking for a password. This can be achieved by using public-key cryptography to log into the machines remotely using SSH. If the private key is secured with a passphrase, which is advisable, an *ssh-agent* needs to be running to supply the key to SSH. If it is not running, start it by typing:

```
$ eval ‘ssh-agent’
Agent pid 259
$ ssh-add
Enter passphrase for /Users/derDoc/.ssh/id_rsa:
Identity added: /Users/derDoc/.ssh/id_rsa (/Users/derDoc/.ssh/id_rsa)
```

If asked for the passphrase, enter it, and the key will be unlocked for use in this shell.

Now, the PWHN client can be started in this shell, because all subshells can also access the environment variable which names the ssh-agent pid. To start PWHN just type:

```
$ java -jar PLMR-1.0.jar
```

In the IP/servers tab select a list of IPs that you want to have the server started on, enter the username for ssh-logins and select “Sync server files” from the menu named “PLMR-Server.” In the opening dialog name the folder that contains your server files, it should be the contents of the plmr-server.zip archive of the distribution. After the sync has finished select the menu entry named “Start server” and select the *start-plmr.sh* script in the folder that has just been synced. The first server in the IP list will be started first and will be the one subsequent join requests will be directed at, so a reasonably lightly loaded and (for best results) close (in terms of Round Trip Time (RTT) space) node should be chosen.

Now go to the *Gen* tab and select the script or program that executes your initializer. If you have a 3-in-1 file click the checkbox. In our example we have three separate python scripts, so they are selected in the appropriate tabs. Since a tree run can only transport the Initializer and Aggregator as separate files, you cannot select a folder. If you try otherwise you will be warned, and have the choice to sync, ignore or cancel. If you sync before using native sync, you can safely ignore the warning and execute anyway, since the files exist on the nodes already. After all the files are selected and the server is started, you can click on *Run!* and the query message (BuildMessage class) will be built and send to the first node to start the build process and execute the query on it. In this case the first server was “planet2.prakinf.tu-ilmenau.de”, so a BuildMessage containing the Initializer and Aggregator as well as their names was built and send to it. This message is parsed and processed by this server, then sent down the tree. After forwarding the message, the node executes the initializer, stores the result, and waits for more results to arrive. Every BuildMessage includes a timeout which specifies the time in milliseconds that a node will wait for results to arrive. When they do, the Aggregator is executed. When the timeout fires, a *SendResMessage* is send to the local node, and it builts the ResultMessage and sends it to its parent.

When the query is finished (arrived at the root), a message (ResultMessage class) is be sent back to the client, containing the results you requested and some more information about the tree. Go to the result tab to look at the result. It contains some extended information about the the run, as well as the result itself.

```
KMR; 'Nov 27; 2006 11:52:43 PM':
Time:       122896
Avg. ms/hop:        40965.332
Client bytes in:        24
% answered/all asked:   30.0%
% answered/ring size:   60.0%
Avg. bytes/link:    18.666666
Avg. bytes/msec:    0.0
Avg. link delay:    96.333336
Avg. fan-in:        2.0
Avg. path length:   1.0
```

```
(3.223333,2.082458)
```

Note that due to reasons which are beyond my understanding only a third of the nodes join the ring and thus send an answer. I chose the timeout of 120 seconds to give each level of the tree enough time (8 seconds) to produce an answer which still is not enough for highly loaded nodes. This leaves time for 15 levels in the tree which is more than enough even for large networks. *% Answered vs. all asked* is the ration of the number of servers in the list and the number of received results. *% Answered vs. ring size* is the ratio of the result count versus ring-size as perceived by the root node. This "perceived ring-size" simply increments each time a new node joins the neighborhood set and does not decrement. This is underestimating the real size because not all nodes join the neighborhood even if they all join at the bootstrap node.

4.2.5 Application to Fern

Fern is a distributed publish-subscribe system for security credentials which are referenced by hashes. It is implemented using a trie in which leaves are actual credentials and internal nodes are just pointers along the way. Internal nodes store both a link and the hash of both child nodes. Fern uses plain files to store all nodes in the trie and publishes them using Coral, so that getting a credential simply comprises requesting some files through Coral. The current version is implemented in python, it consists of both a server that can publish and revoke certificates and maintains the repository; and a client that subscribes to some random subset of all available certificates. It is being developed by Eric Freudenthal and David Herrera at UTEP.

Each client logs a number of interesting statistics, one of them being the average time in seconds it takes for the trie to stabilize, i.e. the time the trie is not in sync with the server after a root certificate change. This is expressed as the number of events that occurred since the last log time and the times it took for certain thresholds of nodes in the trie to be updated (for a histogram); currently these thresholds are: 0.5, 0.75, 0.8, 0.9, 0.95, 0.98, 0.99.

The fern log format is designed to be easily parsable with python and looks like this:

```
['963896C357B6EC33F995E6C26D2DBFBDEE7C212F-
0000000000000000000000000000000000000000-142-0', 55, 55, 0, [(0.5,
10.809647083282471, 1), (0.75, 24.35219407081604, 1), (0.80000000000000004,
26.243300914764404, 1), (0.90000000000000002, 29.295248031616211, 1),
(0.94999999999999996, 31.379647970199585, 1), (0.97999999999999998,
38.444088935852051, 1), (0.999, 42.49633002281189, 1)]]
```

First is a unique name, then the following numbers since the last log period: Number of new nodes received, number new nodes published, number missed, and then histogram values. These are lists of "(threshold, time in secs, count for avg.)." *Threshold* signifies the percentage of nodes, and *time in seconds* is the time it took for that percentage of the nodes to be updated, while the count is just for averaging.

We wrote a 3-in-1 script that parses these log files, aggregates them and finally evaluates them to produce a format that is suitable for PWHN to produce a histogram. This format is simply a list (separated with newlines) of "(X,Y)" tuples to draw a line. This is the script.

Listing 4.3: 3-in-1 script for global-Distribution histogram in Fern

```python
#! /usr/bin/env python

import sys, os
import autoDict

def tupleSum(lt, rt):
    result = []
    for l, r in zip(lt, rt):
        result.append(l+r)
    return result

def readAggInput():
    #read inputs
    f = sys.stdin.readline().rstrip()
    first = sys.stdin.read(int(f)).rstrip()
    sys.stdin.read(1)
    s = sys.stdin.readline().rstrip()
    second = sys.stdin.read(int(s)).rstrip()
    return (first, second)
```

```
if sys.argv[1] == "gen":
    names = os.listdir("fernClientLogs/142/")
    fn = os.path.join("fernClientLogs/142/", names[0])
    fn = os.path.join(fn,"globalDist.log")

    f = open(fn)
    l = f.readlines()[-1]
    f.close()

    l = list(eval(l))
    print l

elif sys.argv[1] == "agg":
    dataMap = autoDict.AutoDict(lambda x : (0, 0))

    for input in readAggInput():
        inp = eval(input)
        name, numRcvd, numEvents, numMissed, data = inp
        for frac, time, count in data:
            dataMap[frac] = tupleSum(dataMap[frac], (time, count))
    outList = []

    for frac in sorted(dataMap.keys()):
        time, count = dataMap[frac]
        outList.append((frac, time, count))
    result = [name, numRcvd, numEvents, numMissed, outList]
    print str(result)

elif sys.argv[1] == "eval":
    name, numRcvd, numEvents, numMissed, data = eval(sys.stdin.readline().rstrip())
    outData = []
    for frac, sum, count in data:
        print (frac, sum / count)
```

After starting Fern on PlanetLab which automatically generates some events, I executed this script on the following servers:

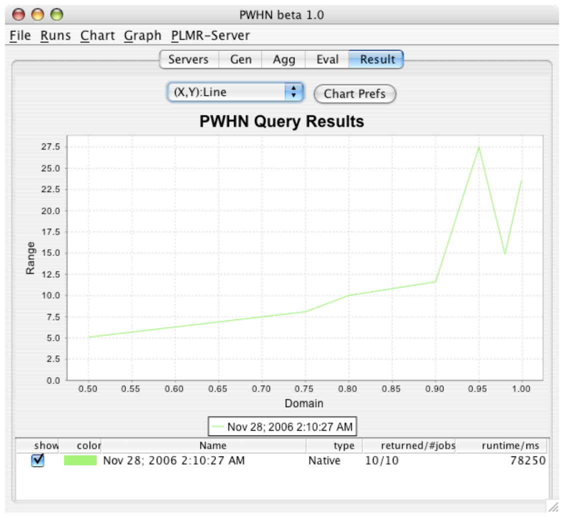
```
planetlab1.utep.edu
pl1.csl.utoronto.ca
pl1.ucs.indiana.edu
pl1.unm.edu
pl1a.pl.utsa.edu
pl2.cs.utk.edu
pl2.ucs.indiana.edu
pl2.unm.edu
plab1-itec.uni-klu.ac.at
```

```
plab1.cs.ust.hk
```

Eight servers answer, producing the following output.

```
(0.5, 5.1013276917593817)
(0.75, 8.0900618689400812)
(0.80000000000000004, 10.013715028762817)
(0.90000000000000002, 11.617396252495903)
(0.94999999999999996, 27.510209832872665)
(0.97999999999999998, 14.874251484870911)
(0.999, 23.537552714347839
```

Plotted as a line the histogram looks like ref Figure 4.13.



(X,Y):Line

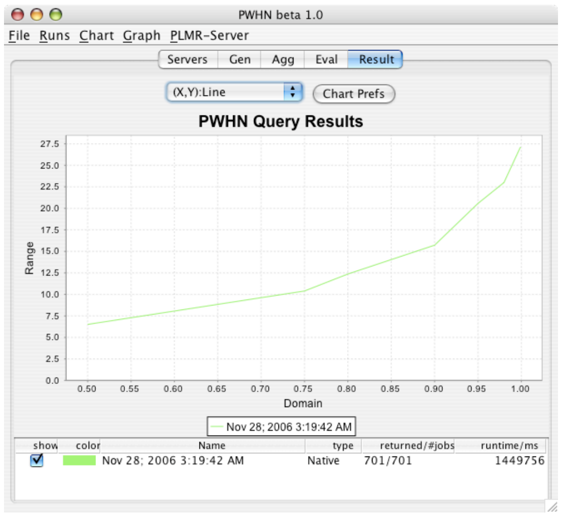
show	color	Name	type	returned/#jobs	runtime/ms
✓	🟩	Nov 28; 2006 2:10:27 AM	Native	10/10	78250

Figure 4.13: Global Distribution Histogram in Fern for 10 nodes

I also executed this script on all 712 nodes. This is the result, and the resulting Histogram looks like Figure 4.14.

```
(0.5, 6.5107764885539101)
(0.75, 10.395761886274958)
```

```
(0.8000000000000004, 12.369233447385122)
(0.9000000000000002, 15.719600200653076)
(0.9499999999999996, 20.572705371512306)
(0.9799999999999998, 23.004060913966253)
(0.999, 27.136558944528755)
```



show	color	Name	type	returned/#jobs	runtime/ms
✓	🟩	Nov 28; 2006 3:19:42 AM	Native	701/701	1449756

Figure 4.14: Global Distribution Histogram in Fern for all 701 nodes

Note that this curve looks a lot more like a histogram.

4.3 Architecture

The PWHN server surrogate is layered on top of FreePastry [9] (see Appendix C). FreePastry by itself takes care of maintaining the ring, i.e. the KBR part of a DHT, and all necessary communication links.

An application that runs on top of FreePastry's KBR only needs to implement the Common-API as defined in FreePastry and start a new node, supplying a bootstrap node if necessary. As soon as it registers itself with the newly created FreePastry node, it will get called whenever something happens and is allowed to route messages to other nodes or

IDs. This could be a message arriving for the application, a message being forwarded by FreePastry or a new neighbor joining.

Messages are defined in the application and are simply regular Java classes that implement the `Message` interface.

Listing 4.4: FreePastry Message Interface

```java
public interface Message extends Serializable {
  public final static byte MAX_PRIORITY = 0;
  public final static byte HIGH_PRIORITY = 5;
  public final static byte MEDIUM_HIGH_PRIORITY = 10;
  public final static byte MEDIUM_PRIORITY = 15;
  public final static byte MEDIUM_LOW_PRIORITY = 20;
  public final static byte LOW_PRIORITY = 25;

  public byte getPriority();
}
```

Messages are serialized by FreePastry before being sent over the wire using the standard Java object serializer which will preserve all non-transient fields. On every node along the path, the application's forward method is called with the de-serialized message. Once it arrives at its destination it is de-serialized and the application's deliver method is called.

PWHN uses the services offered by FreePastry in two ways:

1. In the dissemination phase to reliably spread the word to all nodes using a KMR-tree on top of the FreePastry KBR, and
2. to send messages to other nodes, for example those containing the results in the aggregation phase or those requesting an rsync.

4.3.1 Hasher.py

This is the code for the hasher script:

Listing 4.5: Code of hasher.py

```python
import sys, os, sha
#---------------------------------
def hashit(pth):
    if os.path.isfile(pth):
        ret = []
        ret.append(pth)
        return ret
    ret = []
    if os.path.isdir(pth):
        for fn in os.listdir(pth):
            if ( (not (fn==".")) and (not(fn=="..")) and (not(fn.endswith(".sha1"))) ):
                ret.extend(hashit(os.path.join(pth,fn)))
    ret.sort()
    return ret

def hashlist(l):
    h = ''
    for f in l:
        fs=open(f,'r').read()
        h += sha.new(fs).hexdigest()
    return sha.new(h).hexdigest()
#-------------------------------
if len(sys.argv) < 2:
    print "Error:␣No␣file␣given!"
    sys.exit(1)

hahs = 0
have_sha1 = 0
c_hash = 0
hashf = sys.argv[1]
if (hashf.endswith(os.sep)):
    hashf,t = os.path.split(hashf)
if not(os.path.exists(hashf)):
    print "False"
    sys.exit(0)
f = sys.stdin.readline().rstrip()
if (len(f) == 40):
    hahs = f
if os.path.isfile(hashf+".sha1"):
    h = open(hashf+".sha1",'r').read().rstrip()
    if (len(h) == 40):
        c_hash = h
        have_sha1 = 1
if not(have_sha1):
    c_hash = hashlist(hashit(hashf))
    open(hashf+".sha1",'w').write(c_hash)
if (hahs):
    s = "%s" % (c_hash == hahs)
    sys.stdout.write(s)
else:
    sys.stdout.write(c_hash)
```

`Hasher.py` takes only one argument which is the name of the file or directory to hash. It creates a sorted list of all filenames given, which are all files in the directory and its subdirectories if it were a directory, or just the given filename. Then it will iterate over that list, replacing the filenames with the sha1-hash of their contents, and finally produce another hash of that string. If there is a file called $< filename > .sha1$ it assumes that the file contains the previously created hash and reads it. If that file is not there, it will re-create the hash and store it in that file.

If a hash has been given to the script by means of its `stdin` it will compare the produced hash with the given one, and return either the string “True” or “False.” Since it waits for something to arrive on `stdin`, at least one byte needs to be available.

4.3.2 Unified Modeling Language (UML) Diagrams

plmr-client

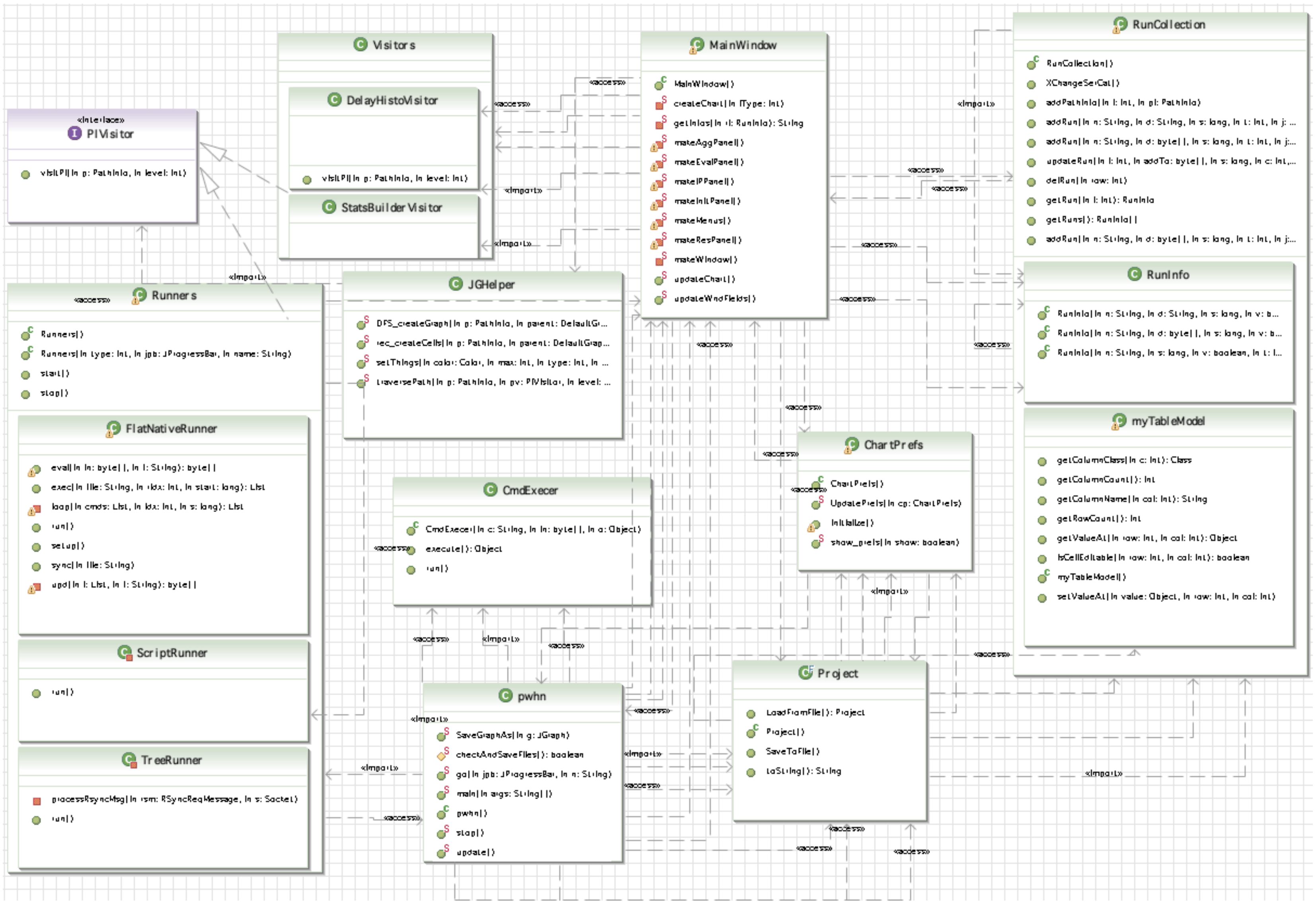

«interface»
PIVisitor
visitPI(in p: PathInfo, in level: int)
Visitors
DelayHistoVisitor
visitPI(in p: PathInfo, in level: int)
StatsBuilderVisitor
MainWindow
MainWindow()
createChart(in fType: int)
getInfos(in ri: RunInfo): String
makeAggPanel()
makeEvalPanel()
makeIPPanel()
makeInitPanel()
makeMenus()
makeResPanel()
makeWindow()
updateChart()
updateWndFields()
RunCollection
RunCollection()
XChangeSerCat()
addPathInfo(in i: int, in pi: PathInfo)
addRun(in n: String, in d: String, in s: long, in t: int, in j: ...
addRun(in n: String, in d: byte[], in s: long, in t: int, in j:...
updateRun(in i: int, in addTo: byte[], in s: long, in c: int,...
delRun(in row: int)
getRun(in i: int): RunInfo
getRuns(): RunInfo[]
addRun(in n: String, in d: byte[], in s: long, in t: int, in j:...
RunInfo
RunInfo(in n: String, in d: String, in s: long, in v: b...
RunInfo(in n: String, in d: byte[], in s: long, in v: b...
RunInfo(in n: String, in s: long, in v: boolean, in t: i...
myTableModel
getColumnClass(in c: int): Class
getColumnCount(): int
getColumnName(in col: int): String
getRowCount(): int
getValueAt(in row: int, in col: int): Object
isCellEditable(in row: int, in col: int): boolean
myTableModel()
setValueAt(in value: Object, in row: int, in col: int)
Runners
Runners()
Runners(in type: int, in jpb: JProgressBar, in name: String)
start()
stop()
FlatNativeRunner
eval(in in: byte[], in l: String): byte[]
exec(in file: String, in idx: int, in start: long): List
loop(in cmds: List, in idx: int, in s: long): List
run()
setup()
sync(in file: String)
upd(in l: List, in l: String): byte[]
ScriptRunner
run()
TreeRunner
processRsyncMsg(in rsm: RSyncReqMessage, in s: Socket)
run()
JGHelper
DFS_createGraph(in p: PathInfo, in parent: DefaultG...
rec_createCells(in p: PathInfo, in parent: DefaultGrap...
setThings(in color: Color, in max: int, in type: int, in ...
traversePath(in p: PathInfo, in pv: PIVisitor, in level: ...
CmdExecer
CmdExecer(in c: String, in in: byte[], in o: Object)
execute(): Object
run()
ChartPrefs
ChartPrefs()
UpdatePrefs(in cp: ChartPrefs)
initialize()
show_prefs(in show: boolean)
pwhn
SaveGraphAs(in g: JGraph)
checkAndSaveFiles(): boolean
go(in jpb: JProgressBar, in n: String)
main(in args: String[])
pwhn()
stop()
update()
Project
LoadFromFile(): Project
Project()
SaveToFile()
toString(): String
«access»
«import»

Figure 4.15: UML diagram of the plmr-client

Note: In the client UML-diagram I omitted all variables for better readability.

The client uses both the JFreeChart [18] and jgraph [19] libraries for rendering the plots and graphs.

The main controlling class of the client is *MainWindow* which defines the window and most of the actions that happen after user interaction. There is a helper class called *Charting* which is responsible for creating and updating the charts and graphs, and is called from MainWindow.

pwhn only is only responsible for controlling, like saving all files before starting an experiment, and contains the static main class.

The *Project* class holds all information about the currently loaded and displayed project (all filenames, results etc.) and can be serialized for permanent storage.

The *jfree* namespace contains three classes that I created for the XYZ (2d-world) view of 3d data, i.e. the renderer and the data-set classes for XYZ sets. *MyXYPlot* implements a Plot in which the background image zooms with the data displayed. This plot is also used for the 2d-world display.

RunCollection holds information about the runs, such as the results, type and all information that is displayed in the table. It exports the interface necessary for the JTable which is called *myTableModel*. The *RunInfo* class that is part of RunCollection holds all information about one run. *ColorRenderer* is the component that renders the color of the run in the table.

PIVisitor, *Visitors* and *JGHelper* are used for parsing and displaying of pathinfo information.

Runners and its subclasses contains all the code necessary for executing and controlling all four types of runs. It uses the *CmdExecer* class for execution of tasks which takes a commandline and its `stdin` input as a byte array, executes the command in a subshell, waits for it, copies the command's `stdout` into a byte buffer, optionally notifies someone and terminates.

ChartPrefs contains a window for viewing and setting all options of a chart like the title, axis names, axis width, background picture and zoom level.

plmr-server

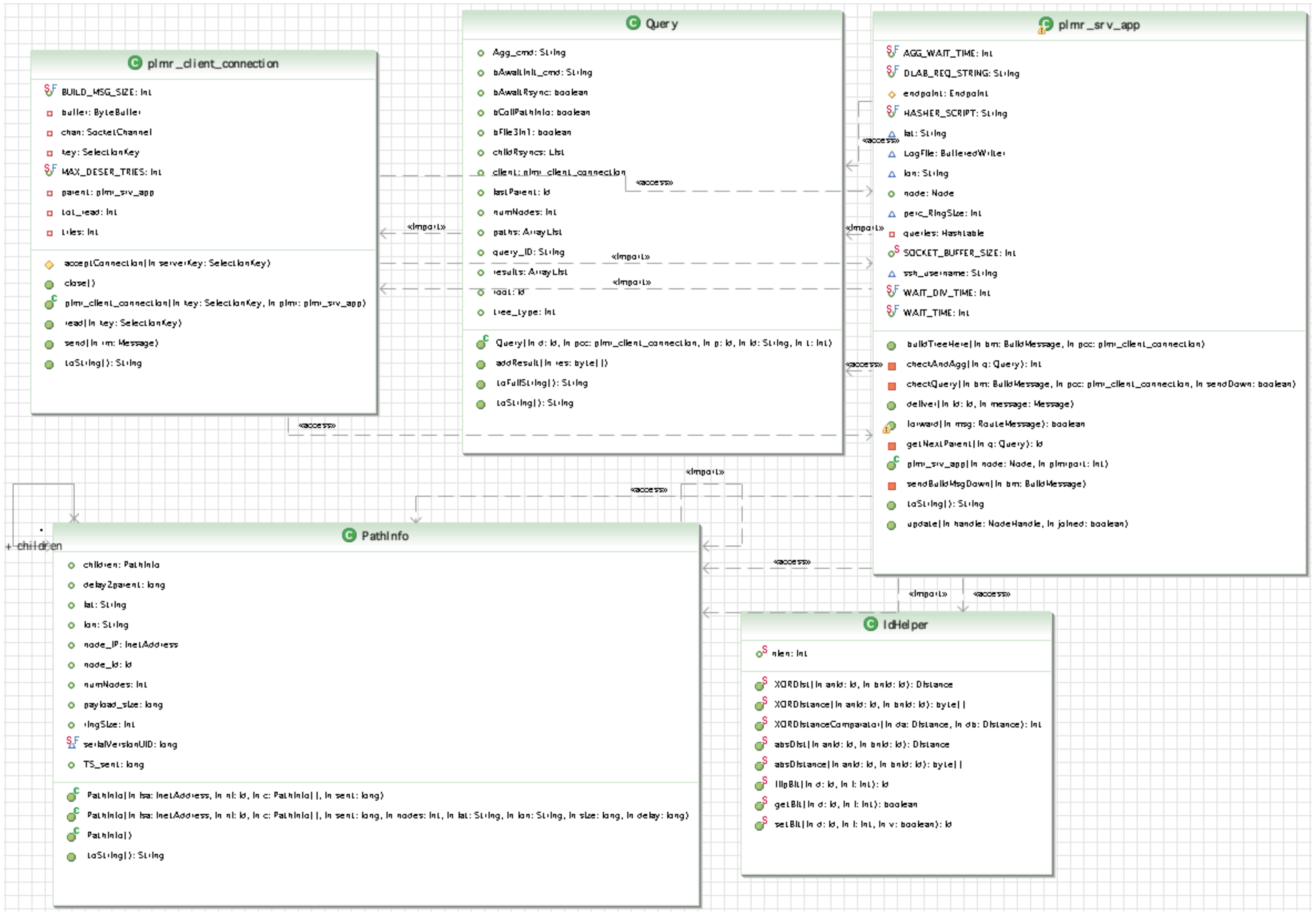

plmr_client_connection
BUILD_MSG_SIZE: int
buffer: ByteBuffer
chan: SocketChannel
key: SelectionKey
MAX_DESER_TRIES: int
parent: plmr_srv_app
tot_read: int
tries: int
acceptConnection(in serverKey: SelectionKey)
close()
plmr_client_connection(in key: SelectionKey, in plmr: plmr_srv_app)
read(in key: SelectionKey)
send(in rm: Message)
toString(): String
Query
Agg_cmd: String
bAwaitInit_cmd: String
bAwaitRsync: boolean
bCallPathInfo: boolean
bFile3In1: boolean
childRsyncs: List
client: plmr_client_connection
lastParent: Id
numNodes: int
paths: ArrayList
query_ID: String
results: ArrayList
root: Id
tree_type: int
Query(in d: Id, in pcc: plmr_client_connection, in p: Id, in id: String, in t: int)
addResult(in res: byte[])
toFullString(): String
toString(): String
plmr_srv_app
AGG_WAIT_TIME: int
DLAB_REQ_STRING: String
endpoint: Endpoint
HASHER_SCRIPT: String
lat: String
LogFile: BufferedWriter
lon: String
node: Node
perc_RingSize: int
queries: Hashtable
SOCKET_BUFFER_SIZE: int
ssh_username: String
WAIT_DIV_TIME: int
WAIT_TIME: int
buildTreeHere(in bm: BuildMessage, in pcc: plmr_client_connection)
checkAndAgg(in q: Query): int
checkQuery(in bm: BuildMessage, in pcc: plmr_client_connection, in sendDown: boolean)
deliver(in id: Id, in message: Message)
forward(in msg: RouteMessage): boolean
getNextParent(in q: Query): Id
plmr_srv_app(in node: Node, in plmrport: int)
sendBuildMsgDown(in bm: BuildMessage)
toString(): String
update(in handle: NodeHandle, in joined: boolean)
«access»
«import»
+ children
PathInfo
children: PathInfo
delay2parent: long
lat: String
lon: String
node_IP: InetAddress
node_Id: Id
numNodes: int
payload_size: long
ringSize: int
serialVersionUID: long
TS_sent: long
PathInfo(in isa: InetAddress, in nI: Id, in c: PathInfo[], in sent: long)
PathInfo(in isa: InetAddress, in nI: Id, in c: PathInfo[], in sent: long, in nodes: int, in lat: String, in lon: String, in size: long, in delay: long)
PathInfo()
toString(): String
IdHelper
nlen: int
XORDist(in aId: Id, in bId: Id): Distance
XORDistance(in aId: Id, in bId: Id): byte[]
XORDistanceComparator(in da: Distance, in db: Distance): int
absDist(in aId: Id, in bId: Id): Distance
absDistance(in aId: Id, in bId: Id): byte[]
flipBit(in d: Id, in i: int): Id
getBit(in d: Id, in i: int): boolean
setBit(in d: Id, in i: int, in v: boolean): Id

Figure 4.16: UML diagram of the plmr-server

The main class of the server is called *plmr_srv_main* it just contains code to parse the commandline and to start the FreePastry node. The class *plmr_srv_app* is the class that implements the necessary FreePastry code to operate a node. It has all function necessary for a FreePastry application, such as for receiving and forwarding messages and for neighborhood-set change notifications. Moreover, this class implements most of the routing algorithm logic and all of the query processing and management code. It uses the *Query* class for representing a query, and the *plmr_client_connection* class for a connection from a client which is responsible for serializing and deserializing the messages to and from the client.

The *CmdExecer* class from the client namespace implements the necessary interfaces for execution by the FreePastry background worker thread and is used for all computations and aggregations.

The java classes in the "messages" namespace are used for representing the messages that are sent over the wire and *PathInfo* is the container-class for the pathinfos. The class *IDHelper* contains static algorithms for simple ID calculations that FreePastry either does not implement or implements "private".

Activity chart

Figure 4.17 shows a rough picture of the activities, messages and classes involved when the user executes a query and it gets disseminated down the tree.

Figure 4.18 details the activities, classes and messages involved when the leafs send their results up the tree and they get aggregated; and what happens when they arrive at the client.

4.3.3 The Tree

The following section defines the messages sent in both phases and lists their important fields. It explains the actions involved when a node receives either of them and details the

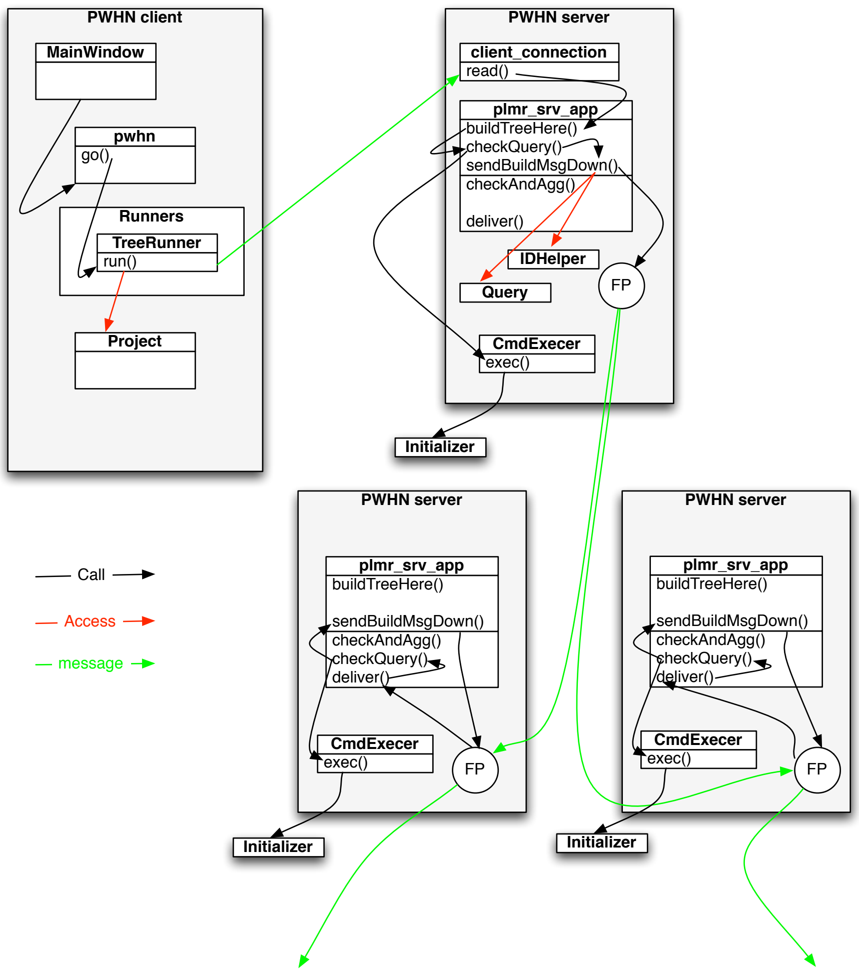


Figure 4.17: Activity chart for the dissemination phase of PWHN

algorithm some more. Both of these messages are sent using FreePastry message sending feature. All classes that implement FreePastry's Message interface can be sent to other nodes by calling upon FreePastry. They are serialized before being sent and deserialized before being delivered automatically. This is done in FreePastry by using the standard Java object serializer.

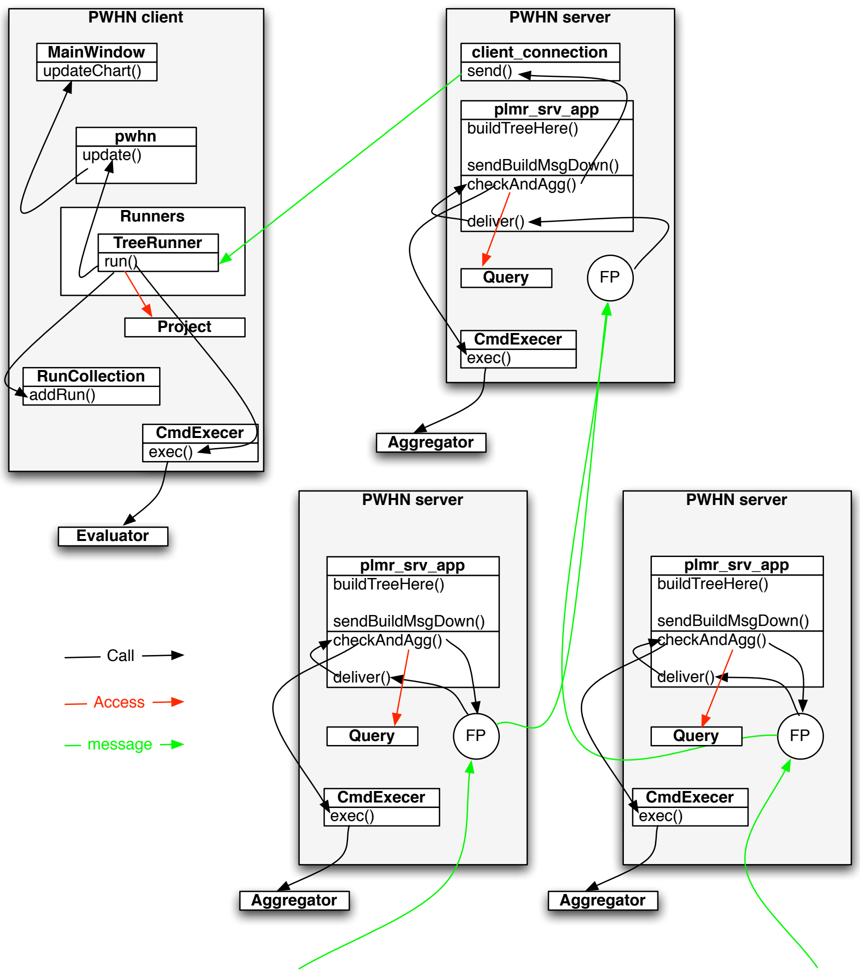


Figure 4.18: Activity chart for the aggregation phase of PWHN

Dissemination

The dissemination phase for all three types of trees uses the same algorithm, i.e. the "dissemination" algorithm of a KMR. The respective message is called *BuildMessage*. It carries information about the root of the tree, sender and receiver (in terms of FreePastry IDs), the type requested, a list of IPs on which to run Init, and the contents of the Initializer and Aggregator if not synced. Otherwise their hashes are present.

It also carries a *ProxyFor* field which gives the ID of the node for which the current sender is acting. This field is important in the case where a node sends the BuildMessage in place of another, non-existent node. Then the receiver has to use this field rather than the sender to determine his prefix. The sender still needs to be present to be able to retrace the path of the message up the tree for aggregation.

I had to add a *timeout* field later, to specify the current timeout, that is the time this node has to wait before sending an aggregated result to its parent. The first approach to estimate the timeout by a node ID's absolute numeric distance to the tree root did not work as well as statically specifying a wait time for each level. A value of 80 seconds for the root node and a delta of 4 seconds per level seems to be the best trade-off between speed and completeness.

Once a *BuildMessage* arrives at a node, a Query class will be created for it, which basically holds all necessary information about the query. The program will inspect the filenames and save the contents as the given filename if they do not look like a hash. If they do look like a hash it will execute `hasher.py` with the given hash. Should the hasher return "False" the application sends a *RequestRSyncMessage* to the node where the *BuildMessage* originally came from to initiate an rsync.

The PWHN client opens a connection to the first IP in the list by connecting to the PLMR-port which was specified on start-time to each plmr-server. Then the PWHN client sends a BuildMessage to the root node to initiate the tree building, albeit without all the FreePastry specific fields filled in. From the fact that the FreePastry node got this message over a client connection it knows that it is supposed to start a new tree, so it builds the query class, sets itself root and continues to send the message down the tree.

If the PWHN-server finds its own IP in the list, it will schedule a run of the Initializer in the dedicated FreePastry worker-thread if its copy of the file is fresh. Otherwise it will wait for the rsync to finish, after which the Initializer will be run. Then the node will schedule a message to let itself know that it is time to send the aggregated results up to its parent, and proceed to send the *BuildMessage* down the tree. To do this the node first determines its common prefix with its parent, and then sends one message for every bit in the suffix flipped. Should the message have been delivered to it because it owns the closest ID to a non-existent one, it will send the message to all those IDs that are further away from the non-existent one than its own and set the ProxyFor field to the original recipient. Then it will send the BuildMessage down, according to the algorithm outlined in subsection 4.1.1.

The FreePastry routing protocol has two phases. The first phase uses the routing table to correct digits of 4 bits at a time. When the next digit cannot be corrected any more, i.e. there is no node that has a 4-bit longer prefix then the current node's, FreePastry switches to the second phase. In the second phase all IDs that share the current prefix are checked for absolute closeness in numeric distance. The node that is numerically closest is chosen. For numeric closeness the direction, i.e. positive or negative, does not matter. This means that the found node is the closest in numeric distance to the target ID in either direction on the ring. Since numeric distance is different from XOR distance, this node is not necessarily the closest in XOR distance, which the routing algorithm from subsection 4.1.1 needs.

To accommodate for this, my implementation keeps searching all nodes in the fingertable for the closest node in XOR distance until there is no closer node. This is done on each node the message is delivered to by FreePastry. Once the closest node is reached, the algorithm continues as outlined above. This approach also prohibits using the second phase of FreePastry's two phase routing protocol.

My code checks the fingertable for an entry before routing to a specific next bit and only sends the message if there is an entry. This is routing in XOR space and ensures that the first phase is used. If there is no corresponding entry in the figertable, second-phase routing would take effect, and thus my code does not send a message at all, but uses XOR distance checking instead. This finds the closest node in XOR space instead of in numeric space and then sends the message directly to this node.

Note that the following two listings only show variables since functions do not get serialized.

Listing 4.6: PWHN-server BuildMessage

```
public class BuildMessage implements Message {
    public transient static final int FP_FTT    = 2;
    public transient static final int FP_KMR    = 3;
    public transient static final int CO_KMR    = 4;

    // serial UID for serialization
    static final long serialVersionUID      = 32780l;

    // stuff for FreePastry
    public Id          from;
    public Id          to;
    // type of tree requested
    public int         tree_type;
    // the root of tree
    public Id          root;
    // Where the msg should have been coming from.
    public Id          senderProxyFor;

    //the unique ID of this query per root,
    //used to identify the query at each node
    //this is the key into a HT for all outstanding queries,
    //normally just hostname + UnixTS
    public String          query_ID;

    // List of IPs to include result from
    public String[]        IPList;

    // if init is a local cmd, this is its path
    public String          Init_cmd;

    // else if its included here, this is it, or its hash if synced
    public byte[]          Init_content;

    // if agg is a local cmd, this is its path
    public String          Agg_cmd;

    // else if its included here, this is it, or its hash
    public byte[]          Agg_content;

    //and the rsync base if it is a dir
    public String          RSyncBaseDir;

    //wheter to collect path Info data
    public boolean         bCollPathInfo;

    //3-in-1 file
    public boolean         bFile3in1;

    //timeout for this level in tree
    public long           timeout;

}
```

When a node receives a *ResultMessage* containing some (possibly aggregated) results from another node, it adds it to the query and checks if a run of the Aggregator is necessary.

That check will run the Aggregator, if one is given, in the background thread, or concatenate the results with newlines.

When it is time to send the results up the tree, the node will determine its parent, depending on the type, and send a *ResultMessage* to it. FTTs simply route the message towards the tree-root, KMRs use the node where the *BuildMessage* came from as its parent, and the Coral-KMR calls the Coral overlay using Open Network Computing (ONC)-Remote Procedure Call (RPC) to determine the parent. If a node happens to receive a *ResultMessage* for an unknown query, it will try to route it towards its tree root, or drop it if it is the root.

Listing 4.7: PWHN-server ResultMessage

```
public class ResultMessage implements Message {

    /** Where the Message came from. */
    public Id         from;

    /** Where the Message is going. */
    public Id         to;

    //content of msg
    /** the unique ID of this query per root, used to identify the query at each node*/
    public String     query_ID;

    /**the result*/
    public byte[]     result   = new byte[0];

    /**and info about path of result*/
    public PathInfo path;

    /**the root for everybody who doesnt know that query*/
    public Id         root;
```

Path Info

In addition to the results of the query, a *ResultMessage* optionally carries some annotations about the path that it took. This information is filled-in by each node along the way, containing the node ID, its IP, its coordinates, the size in byte sent, the number of children, a recursive list of those children, and the send time as a time-stamp. The receiving parent fills in the difference between sending and current time as the delay which, of course, is not as exact as a ping but should suffice for my purposes. This information is used later by the client to recreate the tree graphically (see subsection 4.2.3). This is PathInfo's source:

Listing 4.8: PWHN-server PathInfo class

```java
public class PathInfo implements Serializable {

    static final long   serialVersionUID       = 327821;

    // IP,hostname of this node
    public InetAddress  node_IP;

    // FP ID of node
    public Id          node_id;

    //number of items in path, for easy displaying...
    public int         numNodes;

    //lat of node, so that client does not have to ask dlab for everything
    public String        lat;

    public String        lon;

    // children, that this node got results from, null if leaf
    public PathInfo[]    children;

    // delay to parent (filled in by parent), null if root
    //uses TS_sent - not very exact I know...but what the heck
    public long        delay2parent;

    //serialized length in bytes of the ResMsg that contained this PathInfo,
    //ie. size of payload (result) send over link with roughly above latency
    public long        payload_size;

    // time this was sent to parent (as Unix TS)
    public long        TS_sent;
}
```

Chapter 5

Evaluation

5.1 Objectives and general methods

The objective of the evaluation was to assert the research goals experimentally. This entails to make sure that PWHN is useful and usable as a measurement toolkit, does not perturb the inspected system too much; as well as confirming that KMRs are indeed more efficient in structuring an aggregation tree than FTTs.

For evaluation PWHN needed to be deployed on a set of nodes, an application for instrumentation needed to be selected and supporting scripts for measurements had to be written. Six different tests were developed by writing scripts. They all use CoMon [5] to collect statistics about the node and digest this information in a specific way to produce a useful answer (see section 5.3).

To argue in favor of the former we used PWHN to take measurements of the fern system which is being developed concurrently at UTEP by Eric Freudenthal and Ryan Spring [**?**]. As perturbation in a distributed system is not easily expressed in a metric, I looked at the bytes that are sent around the network, in particular the number of bytes that arrive at the client.

To validate that my data-structure is more efficient, I look at the structure of the constructed tree and calculate the average fan-in over all the nodes in both the FTT and the KMR trees. This should give me a picture of the tree's regularity.

5.2 Experimental setup

Experiments described in this section were conducted upon 500 PlanetLab nodes (see Appendix A).

Since slivers do not come with anything pre-installed, so as to not coerce researchers to a certain development environment, I had to install Java in order to run PWHN. Even with the help of vxargs, this would have meant rsync'ing the Java-RPM-file with every node of our slice, and installing it with the help of a shell script. The jdk-Redhat Package Manager (RPM) is about 50 MegaBytes (MBs) in size, which copied over slow internet connections to the farthest locations on the earth, would have taken a while, without guaranteeing success.

Java was installed upon our PlanetLab slice using RPM and stork [46].

After installing Java everywhere, I made a directory containing all the files needed for the server portion of PWHN. It also contained a copy of the jar file for FreePastry, and shell-scripts to start and stop the PWHN server.

The PWHN client can be used to sync this directory (called the "plmr-server" directory) with all the nodes, and then to start the ring by selecting the start-script from this directory. After some time the ring should have stabilized and you can start making queries against planetlab1.utep.edu. I chose this node just because it is located at UTEP, and thus has the fastest link to me.

Lessons learned Table 5.1 shows the results of the following tests for random sets of 10, 50, 250 and 500 nodes, all averaged over 3 days and different node sets (except the 500 set).

a) *ssh* - ssh login successful,

b) *java* - java is installed on the node (tested by executing `which java`), and

c) *ring* - node joined the ring.

Table 5.1: Node counts for different sets

Count	ssh	java	ring
10	7.33	6.67	6/5/5
50	36.33	30.67	12/28/7
250	160.67	116	65/56/13
500	316.33	227.67	215/64/20

Since it is very hard to determine the absolute number of participants in a FreePastry ring, even though they all joined the same bootstrap node (this does not mean that they all are in its routing table), I am using three different metrics.

I) `ps -Cjava` detects the absolute number of nodes still running FreePastry after 5 minutes,

II) `nodes joined` is the number of nodes joined as noted by the bootstrap node, and

III) `fingers` is the count of foreign nodes in the bootstrap node's fingertable.

Remarks

I): If a node cannot contact the bootstrap node it will start its own ring, and still be running java. If it cannot successfully complete joining the ring within 5 minutes with all the messages and state involved, as noted by the FreePastry-flag "ready", the PWHN node will terminate, and thus no java is running.

II): A node remarks changes in its neighborhood set. This does not give a complete picture of all nodes joined since joining nodes not in the neighborhood set will not be announced, not even though all nodes join at the bootstrap node.

III): This is the absolute count of foreign node IDs in the bootstrap node's fingertable, which does not need to be complete for the same reasons as above.

To find out why on average only half the nodes running java end up joining the ring, I tried to rotate the bootstrap node among a list consisting of the first nodes, rather than having all contact a single point. I did this because we thought that it might be due to a socket or filepointer limitation on the virtual machines. This introduced two other problems: 1) Starting the ring took longer since my algorithm can only start a number of clients at a time and has to wait for all SSH's to return, before starting the next set. 2) If one of the bootstrap nodes is to slow to come up in time or did not come up at all, all connecting nodes will start their own ring, and not be part of the global ring. This could of course be alleviated by having a list of bootstrap nodes all of which a starting node tries to connect to until one works. This is also the reason why the first node in the list (the global bootstrap) should be reasonably lightly loaded.

I observe that the set of PlanetLab systems available at any particular time varies widely and, to support experimentation, I need to routinely probe the set of available hosts. Generally only 10% of the PlanetLab nodes have sufficient resources available to run java and thus PWHN experiments.

I can only guess that that is due to the high average load of PlanetLab nodes (average of 9 on 150 nodes, with a standard deviation of 12), and I have seen loads of up to 700!

It also might be due to the small amount of incoming bandwidth that a sliver gets, because the bandwidth is evenly shared between all slices. Assuming that most nodes have a full 100MBit connection, the maximum throughput in Kbps should be around 10,000. The transmit rate average of 258 nodes is 1300.123 with a standard deviation of 1200.158. The maximum I have seen so far is 13,487. The receive rate is generally close to the transmit rate, although slightly lower in most cases.

Even though I wanted to avoid (by using stork) copying 50 MBs from my PC to all nodes, I had no other choice in the end than to do that. Of 270 nodes that I copied the Java-RPM onto, 244 succeeded in installing Java. Generally, a third of the nodes on which PWHN is started eventually join the FreePastry ring. As mentioned above, it is not clear to what this could be accredited. All those nodes do not completely boot into the ring within 5 minutes and terminate themselves. Even activating debugging could not shed any more light on the reason.

5.3 Script sets

My first intention was to use PWHN to instrument Coral. But after looking through its log files, I realized that they do not contain a lot of interesting information. The only useful piece of information in its log file is the number of hits that the Coral-webserver has seen since its last log-cycle. Since I do not have access to the live Coral installation, I had to run my own Coral network. Unfortunately, every Coral-proxy not running on our UTEP PlanetLab-node crashed when I made a request through it. Thus, the number of hits I could have queried of Coral would have been only from the UTEP node.

CoMon collects slice-centric and node-centric statistics on each node using CoTop and exports them through a sensor which is accessible on the local port 3121. Thus, these stats can be accessed from all slices. They contain information such as the current load, number of live slices, number of used ports and file-descriptors, outgoing and incoming network speeds, and top hogging slices for network-throughput, CPU and memory utilization.

Using that information, I designed the following five scripts to collect aggregates from CoMon. The first script reports the global state as the current average load and its standard deviation. The next two scripts can be used to find nodes on which to deploy experiments by looking for nodes that have low network load and those five that have the lowest number of live slices. The last two scripts can be used to find problems in PlanetLab. They report the top most hogging slice name as well as the hostnames and the amount of gigabytes free of nodes whose harddisks are more than 90% used.

5.3.1 Script set one

I constructed a set of scripts to collect and aggregate the current load over all nodes and its standard deviation. To calculate the average, the sum of all averages is needed along with the number of nodes. In addition, for the calculation of the standard deviation, the sum of the squares of all averages is needed. Therefore, the Init script, which is written in perl, emits a 1 along with the load and its square.

Listing 5.1: Init script for set one

```perl
#!/usr/bin/env perl
$u = `uptime`;
chomp($u);
%@s = split(/ /,$u);
@s = split(/,/,$s[-3]);
$v = $s[0];
$q = $v**2;
print "(1,$v,$q)";
```

The python-Aggregator script reads both its inputs, simply adds them up, and outputs them again.

Listing 5.2: Agg script for set one

```python
#!/usr/bin/env python
import sys
#read inputs
f = sys.stdin.readline().rstrip()
first = sys.stdin.read(int(f)).rstrip()
sys.stdin.read(1)
s = sys.stdin.readline().rstrip()
second = sys.stdin.read(int(s)).rstrip()
fir = first[1:-1].split(",")
sec = second[1:-1].split(",")
#add them
v = int(fir[0]) + int(sec[0])
u = (float(fir[1]) + float(sec[1]))
q = (float(fir[2]) + float(sec[2]))
print "(%i,%f,%f)" % (v,u,q)
```

And, last but not least, the python Evaluator reads the input, calculates the average and standard deviation, and prints both.

Listing 5.3: Eval script for set one

```python
#!/usr/bin/env python
import sys, math

f = sys.stdin.readline().rstrip()
fir = f[1:-1].split(",")

#assign some vars
sumx = float(fir[1])
sumx_squ = float(fir[2])
```

```
num = float(fir[0])

#get the avg.
avg = (sumx / num)

#and get the std dev.
std_dev = math.sqrt( ( sumx_squ - ((sumx**2)/num) ) / (num-1) )

if str(std_dev) == "nan":
        std_dev = 0

# and print avg, and std dev.
print "(%f,%f)" % (avg,std_dev)
```

5.3.1 enumerates the results of an execution of these scripts over 230 nodes. Also shown is the extra information that is calculated for each run.

```
230 nodes; native; 'Oct 1; 2006 2:50:00 PM':
Time:                    138681
Avg. ms/hop:             602.9609
Client bytes in:         3673
% answered/all asked:    100.0%
Avg. bytes/link:         15.969565
Avg. bytes/msec:         0.001758027
Avg. link delay:         9083.8
Avg. fan-in:             230
Avg. path length:        1

(8.033870,9.247825)
```

5.3.2 Script set two

Script two reads the CoMon stats by connecting to “localhost:3121” because the CoMon sensor is running on this port, and extracts the current transmit and receive throughputs. It outputs a tuple consisting of the hostname and the two extracted values only if they are both below the cutoff of 500. Since I simply want all those tuples concatenated, there is no Aggregator.

Listing 5.4: Init script for set two

```python
#!/usr/bin/env python
import os, urllib

#define cut-off for tx, rx
cut_off = 500
comon = "http://localhost:3121"
h = os.popen('hostname').read().strip()
lines = urllib.urlopen(comon).readlines()

tx = float(lines[31].split(":")[1].rstrip())
rx = float(lines[32].split(":")[1].rstrip())

if (tx < cut_off) and (rx < cut_off):
        print "(%s,%f,%f)" % (h,tx,rx)
```

A sample output from 75 nodes looks like this.

```
75 nodes, native; 'Sep 29; 2006 2:44:56 AM':
Time: 85167
Avg. ms/hop: 1135.56
Client bytes in: 626
% answered/all asked: 100.0%
Avg. bytes/link: 8.346666
Avg. bytes/msec: 6.5535464E-4
Avg. link delay: 12736.106
Avg. fan-in: 75
Avg. path length: 1

(planetlab-02.ipv6.lip6.fr,171.000000,343.000000)
(ait05.us.es,300.000000,215.000000)
(dragonlab.6planetlab.edu.cn,403.000000,119.000000)
(planetlab1.snva.internet2.planet-lab.org,63.000000,117.000000)
(planetlab2.atcorp.com,223.000000,182.000000)
```

```
(planetlab2.chin.internet2.planet-lab.org,247.000000,206.000000)
(planetlab1.ucb-dsl.nodes.planet-lab.org,227.000000,329.000000)
(planetlab2.csg.unizh.ch,410.000000,344.000000)
(pli2-pa-2.hpl.hp.com,263.000000,252.000000)
(planetlab2.ucb-dsl.nodes.planet-lab.org,167.000000,429.000000)
(sjtu1.6planetlab.edu.cn,264.000000,253.000000)
(plnode01.cs.mu.oz.au,90.000000,123.000000)
```

5.3.3 Script set three

Script number three extracts the number of live slices from CoMon and extracts only the top five with the lowest live slices using the Aggregator. It keeps a list of at most five (hostname,live) tuples and checks every line from both inputs against this list. If the number of live slices is lower than the current list-element, it is replaced. At the end it emits the list with five elements.

Listing 5.5: Init script for set three

```python
#!/usr/bin/env python
import os,urllib

comon = "http://localhost:3121"
lines = urllib.urlopen(comon).readlines()
h = os.popen('hostname').read().strip()
live = int(lines[34].split(":")[1].rstrip())

print "(%s,%i)" % (h,live)
```

Listing 5.6: Agg script for set three

```python
#!/usr/bin/env python
import sys
#function that checks i - a (h,#live) string - against all 5 elements in the list l
#the list h contains the corresponding hostnames
def check(i,l,h):
        if (i.startswith("(")):
                val = i[1:-1].split(",")
                #find if it is in list
                for j in range(5):
                        if (int(val[1]) < l[j]):
                                l[j] = int(val[1])
                                h[j] = val[0]
                                return

#read inputs
f = sys.stdin.readline().rstrip()
first = sys.stdin.read(int(f)).rstrip()
sys.stdin.read(1)
s = sys.stdin.readline().rstrip()
second = sys.stdin.read(int(s)).rstrip()
#we have a bunch of lines now, split on newline
f = first.split("\n")
s = second.split("\n")
l = [10000,10000,10000,10000,10000]
h = ["","","","",""]
#then make list
for i in f:
        check(i,l,h)
for i in s:
        check(i,l,h)
# and now print
for j in range(5):
        if (h[j]):
                print "(%s,%i)" % (h[j],l[j])
```

This is a sample from an experiment with 150 nodes.

```
150 nodes; native; Oct 1; 2006 1:29:07 AM':

Time: 123979

Avg. ms/hop: 826.5267

Client bytes in: 4643

% answered/all asked: 100.0%

Avg. bytes/link: 30.953333

Avg. bytes/msec: 0.0033714727

Avg. link delay: 9180.953

Avg. fan-in: 150

Avg. path length: 1


(mercury.cs.brown.edu,0)

(plab1.eece.ksu.edu,0)

(cs-planetlab1.cs.surrey.sfu.ca,1)

(edi.tkn.tu-berlin.de,2)
```

```
(dragonlab.6planetlab.edu.cn,2)
```

5.3.4 Script set four

This set of scripts was designed to extract the top hogging slice name from each node and summarize this information by emitting slice names and the number of nodes it hogs.

Listing 5.7: Init script for set four

```python
#!/usr/bin/env python
import os,urllib

comon = "http://localhost:3121"
lines = urllib.urlopen(comon).readlines()
h = os.popen('hostname').read().strip()
c = lines[26].split(":")[1].rstrip()
cpuhog = c.split("␣")[2]
print "(1,%s)" % (cpuhog)
```

Listing 5.8: Agg script for set four

```python
#!/usr/bin/env python
import sys
#checks the hash h, if the name given in the tuple i is already there, and adds its number
#creates new key otherwise
def check(i,h):
        if (i.startswith("(")):
                i = i[1:-1].split(",")
                if h.has_key(i[1]):
                        h[i[1]] += int(i[0])
                else:
                        h[i[1]] = int(i[0])
#read inputs
f = sys.stdin.readline().rstrip()
first = sys.stdin.read(int(f)).rstrip()
sys.stdin.read(1)
s = sys.stdin.readline().rstrip()
second = sys.stdin.read(int(s)).rstrip()
#we have a bunch of lines now, split on newline
f = first.split("\n")
s = second.split("\n")
#now just build a hash with the names and the times they appeared
h = {}
for i in f:
        check(i,h)
for i in s:
        check(i,h)
#and print
for k,v in h.iteritems():
        print "(%i,%s)" % (v,k)
```

Listing 5.9: Eval script for set four

```python
#!/usr/bin/env python
import sys, math

def check(i,h):
        if (i.startswith("(")):
                i = i[1:-1].split(",")
                if h.has_key(i[1]):
                        h[i[1]] += int(i[0])
                else:
                        h[i[1]] = int(i[0])

f = sys.stdin.read().rstrip()
e = f.split("\n")
h = {}
for i in e:
        check(i,h)
#and print
for k,v in h.iteritems():
        print "(%i,%s)" % (v,k)
```

Sample of experiment with 255 nodes.

```
255 nodes; native; 'Oct 1; 2006 1:42:40 AM':

Time: 134477

Avg. ms/hop: 527.3608

Client bytes in: 3784

% answered/all asked: 100.0%

Avg. bytes/link: 14.839215

Avg. bytes/msec: 0.001807337

Avg. link delay: 8210.541

Avg. fan-in: 255

Avg. path length: 1


(2,princeton_adtd)

(5,cernettsinghua_zhanghui)

(2,ucin_galaxy)

(1,ethz_kangoo)

(1,urochester_robinhood)
```

```
(5,rice_epost)

(5,utep_1)

(1,nyu_oasis)

(2,tsinghua_wwy)

(1,cornell_anycast)

(7,ucb_bamboo)

(63,nyu_d)

(4,rice_jeffh)

(15,princeton_codeen)

(2,utah_svc_slice)

(3,unimelb_vmn)

(2,epfl_kornfilt)

(10,princeton_coblitz)

(2,northwestern_remora)

(1,colorado_tor)

(3,uw_ah)

(6,umn_dcsg)

(2,idsl_kspark)

(11,irb_snort)

(1,utah_elab_27274)

(4,iitb_dita)

(8,arizona_stork)

(15,root)

(33,utah_elab_24927)

(10,passau_crawler)

(12,hplabs_oasis)
```

5.3.5 Script set five

Experiment five extracts the usage-percentage of the hard-disks from CoMon and emits a (hostname,disc-free/GB) tuple if the hard-drive is more than 90% full. Since a list of all nodes that meet this criteria is sought, no Aggregator is given.

Listing 5.10: Init script for set five

```
#!/usr/bin/env python
import os, urllib
cutoff = 90

comon = "http://localhost:3121"
lines = urllib.urlopen(comon).readlines()
h = os.popen('hostname').read().strip()
c = lines[14].split(":")[1].rstrip()
c = c.split("␣")
perc = int(c[1][0:-1])
free = float(c[2])

if (perc > cutoff):
        print "(%s,%f)" % (h,free)
```

My run of this script did not reveal any problematic nodes, so no sample output is given.

5.4 Results

5.4.1 Goal - useful tool

To argue that PWHN is useful, I will compare the steps needed for measuring and debugging a P2P system with and without PWHN. Since no other system, except MapReduce which is not designed for P2P, that splits its aggregation into smaller operators lets the user specify arbitrary programs for those operators, a developer would have to build such a system herself.

These steps are basically the ones that were taken to build a monitoring system for Coral. So, let us consider Coral as an example. The requirements and implementation of Coral did not include instrumentation infrastructure. After Coral was finished, deployed on PlanetLab, and running well, the desire to know exactly how well it was running arose. Code for collecting and storing basic state about the system (logging code) has to be present in any system, regardless of PWHN. Logging code was added to Coral, and a collection system was devised.

This system is centralized and works as follows. The Coral monitoring agent, called the *Coral-monitor*, is a process that runs on each Coral node and periodically connects to all three locally running Coral processes to request logging data. That data is written out to disk by the monitor and each of the three processes resets its counters and logging status. Another program, called the *Coral-crawler*, running on a node outside of Coral, is started by cron periodically. This programs' task is to connect to all specified coral-monitors and request their log files to save them locally. A minimal set of nodes serves as a starting point because the crawler learns of new nodes while crawling.

These log files are parsed and inserted into a mySQL database by a set of shell and mySQL of scripts. This approach did not work for long and was finally discontinued for two reasons. First, because it connected to all nodes and requested unaggregated data from them. Second, because this data was inserted into the database unprocessed which resulted in about 100 rows added to the DB every 30 minutes. The reason that this did not work (a DB should be able to handle millions of rows) was that all processing and analysis was done inside the DB using SQL. This was because the queries involved execution of a complex select, joining a number of tables, each containing a couple of million rows.

Obviously, this approach did not scale. To address this, an infrastructure is most suitable if it constructs an aggregation tree tailored to the specific P2P system it is targeting and does its selection and aggregation of relevant data as close to the sources as possible. If it is desired to make the system as general and at the same time as powerful as possible, a framework very close to PWHN will result. This framework will have to be able to construct an aggregation tree on top of the original system's network and be able to make sure that all participants agree on the programs that are used as operators for selection and aggregation. It will have to take care of executing the Init operators on the leafs as well as the Aggregate functions on internal nodes and pass the results up the tree to the right parent. In the case of P2P networks, this is not an easy task because of their highly dynamic membership characteristic.

PWHN offers a toolkit that builds the tree and takes care of the execution of all relevant operators and, furthermore, is able to present the results graphically. To use it, Java needs to be present on all nodes and scripts that obtain (select) the desired data and aggregate it, need to be written. After the PWHN-server part and accompanying files have been copied to all nodes using the client, it can be started to form the FreePastry ring. When the ring has stabilized, queries can be made against the participants. PWHN takes care of transmitting and refreshing files on nodes that have joined later, as well as detecting nodes that have left.

5.4.2 Goal - minimize disruption

To get a picture of the perturbation of an instrumentation framework on the system being measured, I looked at the number of bytes that the collection point (the client node) is receiving for a query.

If the results are simply concatenated instead of being aggregated, there is obviously no difference between a flat and a tree approach; the number of bytes transmitted increases linearly with the number of participants. If the data is being aggregated en-route, however, aggregation trees should have an advantage. The expected number of bytes in a flat approach increases linearly with the number of nodes because all aggregation is done locally. In an aggregation tree approach, I would expect the number of incoming bytes to stay the same for any number of nodes because all data is aggregated inside the tree and only the final result is transmitted to the client. This difference can be seen in Figure 5.1.

I ran three experiments with 10, 50 and 250 nodes for all five script sets and both flat and tree algorithms. The number of bytes that the collecting client receives for both the flat and the KBT algorithms is shown in Figure 5.1.

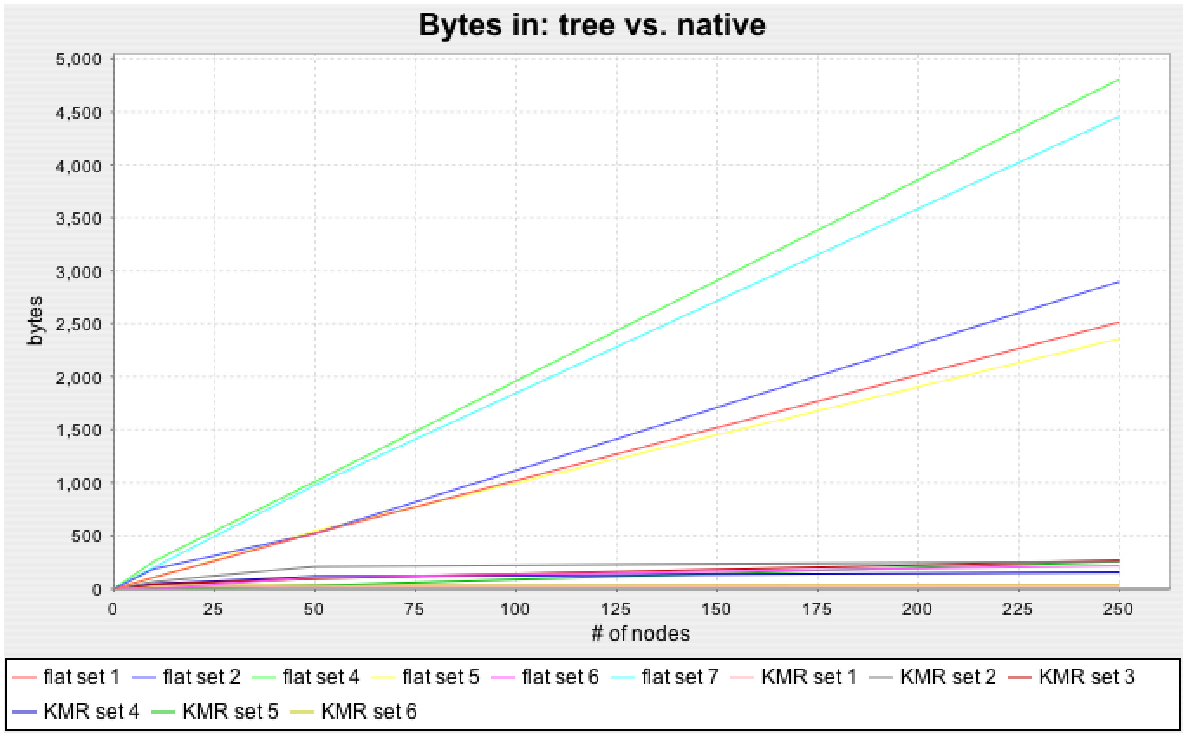


Figure 5.1: Bytes in: tree vs. flat

Despite the many colors, it is still discernible that the transmission size of flat approaches in general increases linearly with the number of nodes, whereas the incoming transmission size for tree approaches stays the more or less the same regardless of the actual number of participants. The variations are due to results that only appear in the output of bigger node sets which can happen for example in set two (see subsection 5.3.2).

Thus, it is to be expected that an aggregation tree minimizes perturbation of the measured system.

5.4.3 Hypothesis - KMR suitable for P2P

To determine if our KMR tree builds a more efficient aggregation tree on top of a DHTs structure than a FTT, I looked at the average fan-in in the tree.

Since a FTT sends all aggregated data towards the root of the tree using the DHT, I expected that most of the nodes send their result directly to the root, resulting in a higher fan-in at the root. A KMR uses the structure of the tree to send results up the tree, so I would expect the average fan-in to be lower.

Experiments with around 10, 20 and 40 answering nodes, of which the average fan-in is shown in Figure 5.2, show that this is indeed the case.

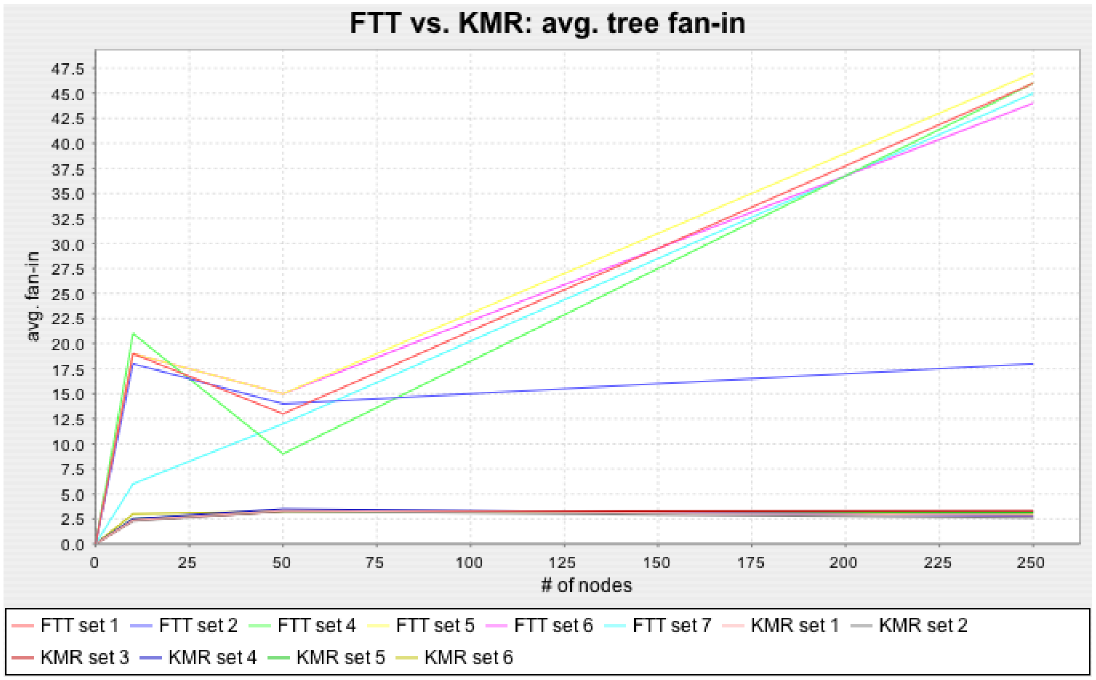


Figure 5.2: Avarage fan-ins in tree: FTT vs. KMR

Again despite the many colors, it is visible that FTTs in general have a linear fan-in in the number of nodes, whereas KMRs have an almost flat fan-in regardless of the number of nodes. I expected that FTTs would build an additional level if not all nodes join the same bootstrap node, because this ensures that not all nodes have that node's key in their routing table, but this was not the case. It seems that even FreePastry rings with over a hundred nodes have not enough ID's to cause two nodes to share the same prefix at a specific level. Thus, for each digit in a given target key there is only one node in every fingertable, which causes FTT-messages always to be routed in one hop.

Figure 5.3 and Figure 5.4 show the links between nodes, as noted by the PathInfo, plotted on a 2d representation of the world.

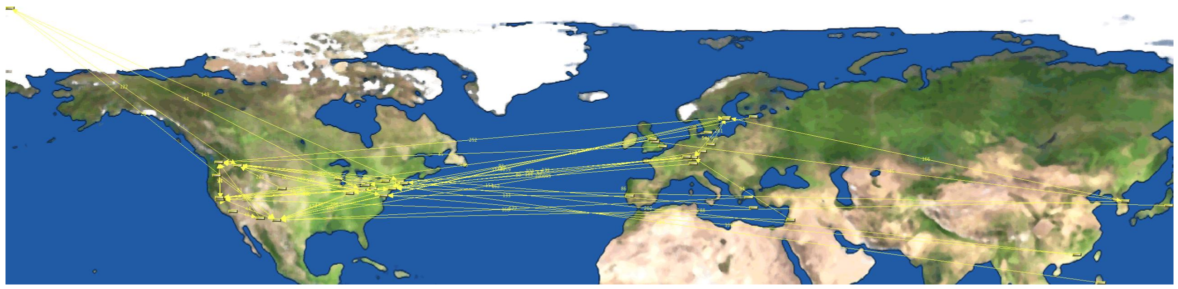


Figure 5.3: KMR-tree links drawn on a world map

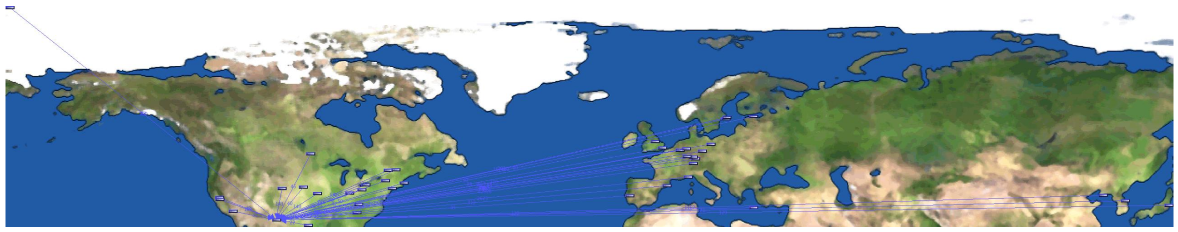


Figure 5.4: FTT-tree links drawn on a world map

Figure 5.5 shows the structure of a FTT tree and a KMR tree with around 20 nodes side-by-side. In that picture as well as in Figure 5.4, it is clearly visible that the FTT nodes all report to the root, whereas the KMR tree has more levels, thus reducing the load on the root.

In conclusion, the KMR builds a more regular tree with a lower fan-in at the root which helps distribute the load more evenly in the system.

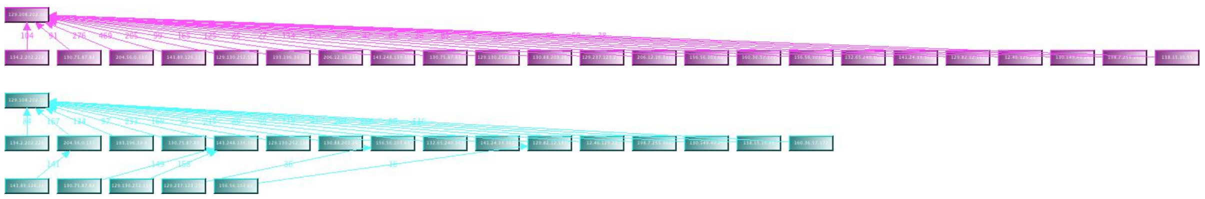


Figure 5.5: Structures of FTT and KBR trees for around 20 nodes side by side

Chapter 6

Future work

As the currently implemented version does not concern itself much with security, this is left for future additions.

First and foremost, this means introducing security certificates to the client-server communication, as this is done unencrypted as of now. Requiring clients to supply certificates with which the executables need to be signed before their execution is another worthwhile security consideration.

An additional security feature could be to require clients that specify executables that already exist on the servers to supply a cryptographic hash of the executables or a public key with which the executables need to be able to answer a challenge correctly before accepting any answers from them.

The ability to name unix domain sockets (pipes) or sockets instead of filenames on which data is to be accepted from already running daemons is another possible future addition.

Regarding algorithms, I have not implemented everything that is described in the introductory chapters due to time constraints. All current approaches just wait for a time inversely proportional to the node's IDs distance to the root ID for answers from its children. Instead, one of the other solutions from 4.1.1 could be implemented, for example estimation using the fingertable.

Additionally, the parent from which the BuildMessage came from is assumed to still be good and the result is sent to it in both FreePastry implementations. Instead, a more complicated approach of using the DHT to look up the correct parent, because it could have changed in the meantime, could be devised.

I did not implement the KBT algorithm because it seemed to much work to keep the tree structure in FreePastry intact and to chose the candidates. Remember that there is only one KBT which has to be kept up-to-date under churn, and cannot be built on each query like in the current approach. Moreover to be able to argue that a KMR is more efficient than a KBT, the system has to be put under considerable load from a number of clients with different roots. A future version could include the algorithmic foundations for a KBT.

Some more possible additions are outlined in the following paragraphs.

Allow input to the Init operators that is not static but dependent on the number of overall participants and the current node number. That is, give the user the ability to specify an input range over an arbitrary domain which will get partitioned between all live nodes by PWHN and supplied to each Init. For example, if the user wanted to be able to distribute fractal generation, he would specify the domain to be natural numbers and the ranges for the x and y coordinates be 0 to the desired image size. PWHN would then take these ranges, divide them evenly among all nodes and supply the respective ranges to the Init functions running on those nodes. These Init functions could either produce partial images which would be overlayed by the Aggregator to form the final image, or simply supply color values to the evaluator which would generate an image from them.

Give the user the ability to create sessions. A session would set up an aggregation tree including all currently selected nodes without specifying any operators. Then, a query could be disseminated down that tree causing the operators to be executed any time during the lifetime of a session without the overhead of re-creating the tree every time. Once a session is not needed anymore it can be deleted, thus causing the tree to be disintegrated. These sessions could be used to only specify a subset of the active nodes to be included in the query, thereby making queries more efficient if a complete picture is not desired. Rather than building the tree over all nodes and just having those nodes execute the Init function that are to be included in the query (which is the way it is done in the current implementation), the tree could instead be built spanning only the relevant nodes. This technique is used in MON [23].

Employ techniques that are known as *online querying* [10, 11, 12, 13, 14, 35, 36]. Systems that support online querying start displaying results from a query as soon as the first result data arrives, and continuously update the display over the lifetime of the query to include more detail. This enables the user to anticipate the outcome of the query and stop it early if he made a mistake in the specification; or it allows him to refine the query, e.g. to query a smaller subset of the original domain. This prohibits the use of aggregation, however, since a characteristic of aggregation is that the complete result is available at only one time, at the end. Moreover, steps have to be taken to guarantee that incoming results are randomly spread over the whole result set. If they do not happen to be randomly distributed, early renderings of the final result will be skewed.

Another possible addition is the storage, and thus replication of query results at different levels of the tree. Firstly, this enables the system to include older data of nodes that have failed recently if consistency bounds are specifiable. Secondly, it allows the inclusion of historic data in queries if those queries are regularly executed to store their results. For this, *up-k* and *down-j* parameters are useful. They let the writer of a query choose the replication parameters, i.e. how far up and down the tree results of single nodes are to be stored, depending on the update rates of the underlying variables. Adding replication abilities to PWHN poses a problem because replication and later retrieval inherently assume knowledge of meta-data (i.e. schemas) about the queried variables which PWHN does not posses. The knowledge of this meta-data in PWHN only exists in the operators themselves, which is why storage and retrieval logic would have to be included in the operators. This, in turn, can make the operators unnecessarily complex.

6.1 Future work using Coral

Theoretically, the Coral overlay could be used in the dissemination phase in the following way.

The challenge lies in knowing which nodes are members of sub-clusters, that is smaller clusters that are fully contained in the current one. Once those nodes are known it is trivial to chose one representative deterministically for each cluster that is responsible for further spreading the word in that cluster, and the same steps are taken for each smaller cluster. It might, for example, just be the node that is closest in its cluster to an ID that is assigned to each query on creation. Since this is different for each query-ID though, the list will change constantly. Another way is to have the node in each cluster that is closest to the cluster-ID be the representative for that cluster. Since this node will only be used in the dissemination phase, there is no danger of overloading it with too many queries.

Solution 1 Since there exists no central instance in Coral that knows who is a member of which cluster, or even who is a member at all, the challenge is still to find those who are representatives of sub-clusters. This can be solved by the addition of a second DHT to Coral. This DHT will have no levels, i.e. it will be accessible at each node uniformly and will contain only the representatives of each cluster indexed under their cluster ID. Each node will need to periodically look up all its cluster-IDs in that same cluster and if it is the closest one to its ID, it will insert itself in the global representative DHT.

This still leaves the problem of knowing which clusters exist. Since passively listening for cluster information in normal traffic is not likely to yield a complete picture, a list needs to be built. This could be done by the representatives who will be responsible for inserting information about themselves and their clusters at a well known place in the global (level-0) DHT or the second (representative) DHT. This will result in a list being built at that place containing information about all existing clusters. Single entries in that list have to be soft-state since clusters can vanish, like anything else in a DHT. Since this list contains all the information of the representative DHT, the list would make the representative DHT obsolete.

Solution 2 This leads to another solution. The information about its sub-clusters is only needed in a cluster itself. Once a node determines that it is the representative for a cluster, it inserts itself and its cluster information under the ID of its next higher cluster in the DHT of that cluster. This will end up building a list of all representatives of sub-clusters exactly where it is needed - at the representative of the next higher cluster. This even releases a node from the burden of having to look itself up in all its clusters, since it can infer from the simple fact that a list is stored at itself that it must be the representative for the cluster under which ID this list is stored.

However, this approach comes with a trade-off. If the number of members is small, the second solution is faster and more optimal than the first. When the number of participants crosses a certain threshold (which needs to be determined) then building a huge list at one node can overload that node, thus introducing a second representative-DHT is clearly the better and more optimal solution.

Chapter 7

Conclusion

This thesis describes PWHN, an infrastructure that is useful for instrumenting, debugging a monitoring of distributed systems. For large-scale P2P systems, hierarchical aggregation is a fundamental abstraction for scalability.

PWHN makes two unique contributions. Firstly, it allows the user to specify arbitrary programs for the three operators that the collection and aggregation are split into, much like MapReduce does. These three phases have been used internally by traditional DBMS for SQL formulated aggregation queries for a long time.

Secondly, it extends ideas from related systems to construct aggregation trees on top of DHTs to achieve higher efficiency.

An additional contribution is the naming and taxonomy of algorithms for building aggregation-trees upon KBR systems (KBTs and FTTs).

I argue that the algorithms used by others are not optimal and show how they can be ameliorated. I ran measurements of PWHN itself which show that my hypotheses is correct.

I implemented the PWHN toolkit to help researchers conduct, monitor and debug distributed experiments on PlanetLab. This toolkit is useful because distributed software is generally designed to be efficient, which could be compromised by an included measurement infrastructure. Conversely, if it was not designed with instrumentation in mind, it might not be able to support the needed operations. In this manner, PWHN offers a "bridging" architecture which permits a system not carefully designed to instrument itself to be effectively instrumented.

References

[1] K. P. Birman, R. van Renesse, and W. Vogels. Navigating in the storm: Using astrolabe for distributed self-configuration, monitoring and adaptation. In *Proc. of the Fifth Annual International Workshop on Active Middleware Services (AMS)*, 2003. 2.6.3

[2] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. *Lecture Notes in Computer Science*, 1987:3–??, 2001. URL `citeseer.ist.psu.edu/article/bonnet01towards.html`. 2.6.4, 2.6.4

[3] J. Case, M. Fedor, M. Schoffstall, and J. Davin. RFC 1157 - simple network management protocol (SNMP), May 1990. URL `http://www.faqs.org/rfcs/rfc1157.html`. 2.6.3

[4] B. Chun, J. M. Hellerstein, R. Huebsch, P. Maniatis, and T. Roscoe. Design considerations for information planes. In *Proc. of the First Workshop on Real, Large Distributed Systems (WORLDS)*, Dec. 2004. URL `citeseer.ist.psu.edu/chun04design.html`. 2.6.2

[5] CoMon. `http://comon.cs.princeton.edu/`. Valid on 13.9.2006. 5.1

[6] F. Dabek, B. Zhao, P. Druschel, and I. Stoica. Towards a common API for structured peer-to-peer overlays, Feb. 2003. URL `citeseer.ist.psu.edu/dabek03towards.html`. (document), 2.3.2, 4.1.2, B.1

[7] Distinct ONC/RPC Toolkits. `http://www.onc-rpc-xdr.com`. Valid on 13.9.2006. D.2.2

[8] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with coral. In *Proceedings of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, Mar. 2004. 1.1.1

[9] FreePastry. http://www.freepastry.org/. Valid on 13.9.2006. 4.1.2, 4.3

[10] P. J. Haas. Large-sample and deterministic confidence intervals for online aggregation. In *Proceedings of the Ninth International Conference on Scientific and Statistical Database Management (SSDBM)*, 1997. 6

[11] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, June 1999. 6

[12] J. M. Hellerstein. Control: Providing impossibly good performance for data-intensive applications. In *Proceedings of the Seventh International Workshop on High Performance Transaction Systems (HPTS)*, Sept. 1997. 6

[13] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, June 1997. 6

[14] J. M. Hellerstein, R. Avnur, A. Choua, C. Hidber, C. Olston, V. Raman, T. Roth, and P. J. Haas. Interactive data analysis with CONTROL. *IEEE Computer*, 32(8), Aug. 1999. 6

[15] R. Huebsch, B. Chun, and J. Hellerstein. Pier on planetlab: Initial experience and open problems. Technical Report IRB-TR03 -043, Intel Research Berkeley, Nov. 2003. URL citeseer.ist.psu.edu/huebsch03pier.html. 2.6.2

[16] R. Huebsch, J. M. Hellerstein, N. L. Boon, T. Loo, S. Shenker, and I. Stoica. Querying the internet with PIER. In *Proceedings of 19th International Conference on Very Large Databases (VLDB)*, Sept. 2003. URL citeseer.ist.psu.edu/huebsch03querying.html. 2.5.2, 2.6.2

[17] R. Huebsch, B. Chun, J. M. Hellerstein, B. T. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A. R. Yumerefendi. The architecture of PIER: an internet-scale query

processor. In *The Second Biennial Conference on Innovative Data Systems Research (CIDR)*, Jan. 2005. URL citeseer.ist.psu.edu/huebsch05architecture.html. 2.5.2, 2.6.2

[18] JFreeChart. http://www.jfree.org/jfreechart/. Valid on 25.11.2006. 4.2.1, 4.3.2

[19] jgraph. http://www.jgraph.com/. Valid on 25.11.2006. 4.2.1, 4.3.2

[20] M. F. Kaashoek and D. R. Karger. Koorde: A simple degree-optimal distributed hash table. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, Mar. 2003. URL citeseer.ist.psu.edu/kaashoek03koorde.html. 2.6.1

[21] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: Mobile networking for 'Smart Dust'. In *Proc. of the International Conference on Mobile Computing and Networking (MOBICOM)*, pages 271–278, 1999. URL citeseer.ist.psu.edu/kahn99next.html. 2.6.4

[22] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27: 831–838, 1980. 2.2

[23] J. Liang, S. Y. Ko, I. Gupta, and K. Nahrstedt. Mon: On-demand overlays for distributed system management. In *Proc. of the 2nd USENIX Workshop on Real, Large Distributed Systems (WORLDS)*, Dec. 2005. 6

[24] V. Lorenz-Meyer and E. Freudenthal. The quest for the holy grail of aggregation trees: Exploring prefix overlay(d) treasure-maps. Technical report, UTEP, Texas, Mar. 2006. URL http://rlab.cs.utep.edu/~vitus. Unpublished paper. 2.5

[25] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, 2002. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/844128.844142. 2.6.4, 2.6.4

[26] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, Mar. 2002. URL http://www.cs.rice.edu/Conferences/IPTPS02. 2.6.1, 2.6.3

[27] S. Nath, Y. Ke, P. B. Gibbons, B. Karp, and S. Seshan. Irisnet: An architecture for enabling sensor-enriched internet service. Technical Report IRP–TR–03–04, Intel Research Pittsburgh, June 2003. 2.6.4

[28] L. Peterson, D. Culler, T. Anderson, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. In *Proceedings of the 1st Workshop on Hot Topics in Networks (HotNets-I)*, Princeton, New Jersey, Oct. 2002. URL citeseer.ist.psu.edu/peterson02blueprint.html. 2.4, 2.6.2

[29] PIER. http://pier.cs.berkeley.edu/. Valid on 13.9.2006. 2.5.2, 2.6.2

[30] PlanetLab. http://www.planet-lab.org. Valid on 13.9.2006. 2.4, 2.6.2

[31] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997. URL citeseer.ist.psu.edu/plaxton97accessing.html. 2.5.1, 2.6.1

[32] PLMR - The PlanetLab MapReduce Project. http://www.robust.cs.utep.edu/~vitus/plmr/ or http://derdoc.xerver.org/pub/plmr/. Valid on 13.9.2006. 4.2.2, E

[33] G. J. Pottie and W. J. Kaiser. Wireless integrated network sensors. *Commun. ACM*, 43(5):51–58, 2000. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/332833.332838. 2.6.4

[34] Project IRIS. http://www.project-iris.net. Valid on 13.9.2006. 2.6.4

[35] V. Raman and J. M. Hellerstein. Partial results for online query processing. In *Proceedings of the ACM SIGMOD international conference on Management of data (SIGMOD)*, pages 275–286, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-497-5. doi: http://doi.acm.org/10.1145/564691.564723. 6

[36] V. Raman, B. Raman, and J. M. Hellerstein. Online dynamic reordering for interactive data processing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Sept. 1999. 6

[37] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. Technical Report TR-00-010, University of California, Berkeley, Berkeley, CA, 2000. URL `citeseer.ist.psu.edu/ratnasamy02scalable.html`. 2.3.2, 2.6.1, 2.6.2

[38] S. Ratnasamy, J. Hellerstein, and S. Shenker. Range queries over DHTs. Technical Report IRB-TR-03-009, Intel Research, 2003. 2.6.2

[39] Remote Tea - Java ONC/RPC. `http://remotetea.sf.net`. Valid on 13.9.2006. D.2.2

[40] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proc. of USENIX Technical Conference*, June 2004. URL `citeseer.ist.psu.edu/rhea03handling.html`. 2.6.1, 2.6.2

[41] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218: 329–??, 2001. URL `citeseer.ist.psu.edu/rowstron01pastry.html`. 2.3.2, 2.6.1, 2.6.3, 4.1.2, C.1

[42] Self-certifying File System. `http://www.fs.net`. Valid on 13.9.2006. D.2

[43] A. Shatdal and J. F. Naughton. Adaptive parallel aggregation algorithms. In *Proceedings of the ACM-SIGMOD international conference on Management*

of data (SIGMOD), pages 104–114, May 1995. URL citeseer.ist.psu.edu/shatdal95adaptive.html. 2.6.4

[44] R. Srinivasan. RFC 1831 - RPC: Remote procedure call protocol specification version 2, Aug. 1995. URL http://www.faqs.org/rfcs/rfc1831.html. D.2.2

[45] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001. URL citeseer.ist.psu.edu/stoica01chord.html. 2.3.2, 2.6.1, 2.6.2, C.1

[46] Stork Project. http://www.cs.arizona.edu/stork/. Valid on 26.9.2006. 5.2

[47] R. van Renesse and K. P. Birman. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining, 2001. URL citeseer.ist.psu.edu/article/robbert01astrolabe.html. 2.6.1, 2.6.3

[48] R. van Renesse and A. Bozdog. Willow: DHT, aggregation, and publish/subscribe in one protocol. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS)*, Feb. 2004. URL citeseer.ist.psu.edu/642279.html. 2.5.1, 2.6.3

[49] PlanetLab VNET. http://planet-lab.org/doc/vnet.php. Valid on 13.9.2006. A.1

[50] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An information plane for networked systems, Nov. 2003. URL citeseer.ist.psu.edu/wawrzoniak03sophia.html. 2.6.2

[51] M. D. Welsh. *An Architecture for Highly Concurrent, Well-Conditioned Internet Services*. PhD thesis, University of California, Berkeley, 2002. URL citeseer.ist.psu.edu/532228.html. 2.6.2

[52] P. Yalagandula. *A Scalable Information Management Middleware for Large Distributed Systems*. PhD thesis, University of Texas at Austin, Aug. 2005. 2.6.3

[53] P. Yalagandula and M. Dahlin. A scalable distributed information management system, 2003. URL citeseer.ist.psu.edu/yalagandula03scalable.html. 2.5.2, 2.6.3

[54] Z. Zhang, S. Shi, and J. Zhu. Somo: Self-organized metadata overlay for resource management in p2p DHT. In *Proc. of the Second Int. Workshop on Peer-to-Peer Systems (IPTPS)*, Berkeley, CA, Feb. 2003. URL citeseer.ist.psu.edu/zhang03somo.html. 2.5.1, 2.6.3

[55] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, Apr. 2001. URL citeseer.ist.psu.edu/zhao01tapestry.html. 2.3.2, 2.6.1

All references include links to the respective citeseer (http://citeseer.ist.psu.edu/) page if applicable, as well as links back to the sections where they appeared. Additionally, web links note the date of validity.

Glossary

A

ACL	Access Control List
ADHT	Autonomous DHT
ADT	Abstract Data Type
AO	Aggregation Overlay
API	Application Programming Interface

C

CDN	Content Distribution Network
C.S.	Computer Science

D

DBMS	Database Management System
DHT	Distributed Hash Table (see section B.2)
DNS	Domain Name System
DSHT	Distributed Sloppy Hashtable

F

FC	Fedora Core
FTP	File Transfer Protocol

FTT	FingerTable-based Tree (see subsection 2.5.2)
G	
GPL	General Public License
GUI	Graphical User Interface
I	
ICMP	Internet Control and Message Protocol
IDL	Interface Definition Language
IP	Internet Protocol
ISEP	International Student Exchange Program
K	
KBR	Key-Based Routing layer (see section B.1)
KBT	Key-Based Tree (see subsection 2.5.1)
KMR	Key-based MapReduce (see subsection 4.1.1)
L	
LAN	Local Area Network
LGPL	Lesser General Public License
M	
MAC	Media Access Control
MB	MegaByte
MIB	Management Information Base

N

NAT	Network Address Translation
NMSU	New Mexico State University

O

OA	Organizing Agent
ONC	Open Network Computing (see subsection D.2.2)
OS	Operating System

P

P2P	Peer-to-Peer
PHP	PHP Hypertext Preprocessor
PIER	P2P Information Exchange & Retrieval
PLC	PlanetLab Central (see Appendix A)
PLMR	PlanetLab-MapReduce
PWHN	PlanetenWachHundNetz

R

RADC	Research and Academic Data Center
RFC	Request for Comments
RO	Routing Overlay
RPC	Remote Procedure Call
RPM	Redhat Package Manager

RTT	Round Trip Time
S	
SA	Sensing Agent
SDIMS	Scalable Distributed Information Management System
SNMP	Simple Network Management Protocol
SOMO	Self-Organized Metadata Overlay
SQL	Structured Query Language
SSH	Secure SHell
T	
TAG	Tiny AGgregation service
TCP	Transmission Control Protocol
TLD	Top Level Domain
TTL	Time-To-Live
U	
UDP	Universal Datagram Protocol
UML	Unified Modeling Language
URL	Uniform Resource Locator
UTEP	University of Texas at El Paso
V	
VM	Virtual Machine

X

XDR	eXternal Data Representation
XML	eXtensible Markup Language

Appendix A

PlanetLab

This gives a short but sufficient description of the of the PlanetLab execution environment. PlanetLab is a distributed testbed intended for research in P2P. It consists of 707 nodes scattered around the world donated by research and academic institutions. All nodes are remotely managed by a management consortium in Princeton.

The next section gives a short overview of the linux environment that slices provide researchers with followed by some terminology.

A.1 Environment

After joining PlanetLab and donating at least two nodes, an institution can request a slice. When this slice is instantiated, PlanetLab Central (PLC) creates a VM on all the participating nodes and transmits the public keys of allowed users to them.

When a sliver is created, it is in a pristine state. It has litle more than an SSH daemon installed. This allows researchers to install whatever execution environment they prefer, be it Java, plain C or Mono. To be able to log into the respective slice each node runs an administrative slice with one global SSH daemon. When a user logs into this daemon it forwards the connection to the appropriate slice and *chroots* into its VM.

VMs are implemented by a Virtual Machine Monitor (VMM), of which only one runs on each machine. The current VMM is linux-based and only supports one type of VM at the moment: Linux *vservers*. VMs are created and controlled by a *node manager* on each node. It has a well-known interface that can be called upon from the outside world, either by PLC or by other infrastructure services, and responds by creating VMs and binding resources to it.

Vservers ensure separation between slivers by simulating a complete, dedicated linux machine. Slivers look like a single linux install on a physical machine and know nothing of the other slivers. In fact, they have no way of interfering with each other, except through an interface the node-manager exports. This interface allows slivers to “share” directories and other slivers to “mount” those. The only other way to pass data from one sliver to another is through sockets.

All remote calls in PlanetLab are made through XML-RPCs. Calls that come from PLC are signed using the PLC private key whose corresponding public key is hardcoded into every node.

PlanetLab allows slices to create and bind normal and raw sockets through the use of VNET [49], the PlanetLab virtualized network access module. Packets that pass through VNET are tracked and delivered to the slice that created the socket. Incoming packets that do not belong to an already bound socket are delivered to the administrative slice. Slices may bind sockets to an unbound port or send out packets on a previously unbound port which results in this port being bound to the slice. This is sometimes called *lazy binding*.

PlanetLab implements raw sockets as what is called *safe raw sockets*. The kernel suppresses all TCP and UDP replies related to raw sockets in order to not interfere with it. Raw sockets have to be bound to the `VNET` virtual ethernet interface by the slice. Since raw packets are actually re-routed through the IP-stack of the kernel, slices may include Media Access Control (MAC) and Internet Protocol (IP) headers, but destination and sender MAC addresses are ignored. The IP-header has to be well-formed and routable, however, and the socket has to be the one owned by the slice. Only certain Internet Control and Message Protocol (ICMP) messages are allowed. Furthermore, the packet is only passed through if the port number is above 1024.

A.2 Terminology

"Node"

A PlanetLab *node* is a machine dedicated to PlanetLab and running the PlanetLab flavor of Linux (a modified Fedora Core (FC) 2). It is connected to the internet and preferably not located behind any kind of firewall or Network Address Translation (NAT).

"Site"

An institution that is a member of PlanetLab has a distinct *site*, i.e. a location in the 3-d world where its nodes reside. Thus, a site runs multiple nodes in its location in the world. Each site has to donate at least 2 nodes to PlanetLab.

"Slice"

A *slice* is the "virtual" abstraction of a dedicated part of the resources on a set of nodes. It is therefore like a crosscut across multiple nodes. Each node runs one VM per slice that is assigned to it. A slice is typically used for one experiment only.

"Sliver"

The VM on one node that is part of a particular slice is sometimes called a *sliver*.

Appendix B

Distributed Hash Tables (DHTs) und their Key-Based Routing layers (KBRs)

This appendix gives a description of a DHTs interface and its corresponding KBR.

B.1 The KBR

The KBR guarantees to route a message to the owner of a given key in $O(log_k(n))$ hops for an n-bit wide ID-space. To do this, the KBR basically employs a k-ary search in the ID-space. The arity depends on the base that the KBR defines its IDs to be. Without loss of generality I assume them to be base 2 in my examples, like most DHT implementations do today.

IDs can be seen as long numbers in the base they are defined in. To change any number to any other, all that needs to be done is to "correct" digits in the source number starting from the beginning until the target number is reached. For this, at most "number-of-digits" steps are needed. Since the number of digits of any number depends on the base, this would take at most $O(log_k(n))$ for base k.

Thus, to send a message from any ID to any other we only have to "correct" one digit of the ID at each step. For IDs defined in base 2 this is basically employing a distributed binary-search by making a routing decision at each node. It checks in which range (subtree of the search-tree) the target ID falls and forwards the message to a node in that subtree.

The DHT specification has various dimensions of freedom in this process.

First, digits can be corrected strictly in order or out of order, which essentially results in *hypercube* routing.

Second, digits that have an absolute order (like in most numbering systems), can be corrected in only one direction; for example up, or in both. This does not make any difference in base-2 routing, because a bit can only have one of two states.

Of course digits are fixed $mod(log_k(n))$, thus correcting "upwards" eventually leads to the right digit, it just takes longer. This also has to do with the distance-metric used for the ID-space. If the metric allows negative distances, but the implementation only allows to route in positive direction, it has to go "around," whereas if the metric used is symmetric by essentially taking the absolute value of the distance, this would not happen. The XOR metric used by Kademlia, for example, has this property.

Thirdly, an implementation has the choice of only allowing messages to be forwarded to the exact node that has the next digit corrected and shares the digits after it (the *suffix*) with the current node (like in my KMR), or its immediate successor if that node is not there (like in Chord). Instead, the DHT could allow the message to be forwarded to any node that has the correct next digit. If digits are corrected upwards this would only work as long as the next hop is still smaller. The case when digits can be corrected in both directions allows the DHT the freedom to forward to the whole sub-space (or sub-tree).

This process can be visualized in at least two ways.

The top parts of Figure B.1 and Figure B.2 represent it as a circle, which is commonly used for Chord-like DHTs because they are defined to only be able to route upwards and use the exact finger or its immediate successor. In the circle this means that messages are always passed around clockwise if IDs increase in that direction.

The bottom parts of Figure B.1 and Figure B.2 show the representation as a decision-tree; that is, routing to ever smaller neighborhoods (sub-trees) of the target key, which is more commonly used for Kademlia-like implementations.

Figure B.1 shows the fingers for the Chord DHT at the top, and for Kademlia-style DHTs at the bottom. In this case, I assume a 4-bit wide ID-space, which is common to both and thus shown in the middle. Note that Kademlia chooses nodes of the other half-tree as fingers and keeps them in a totally ordered list using an implementation specific

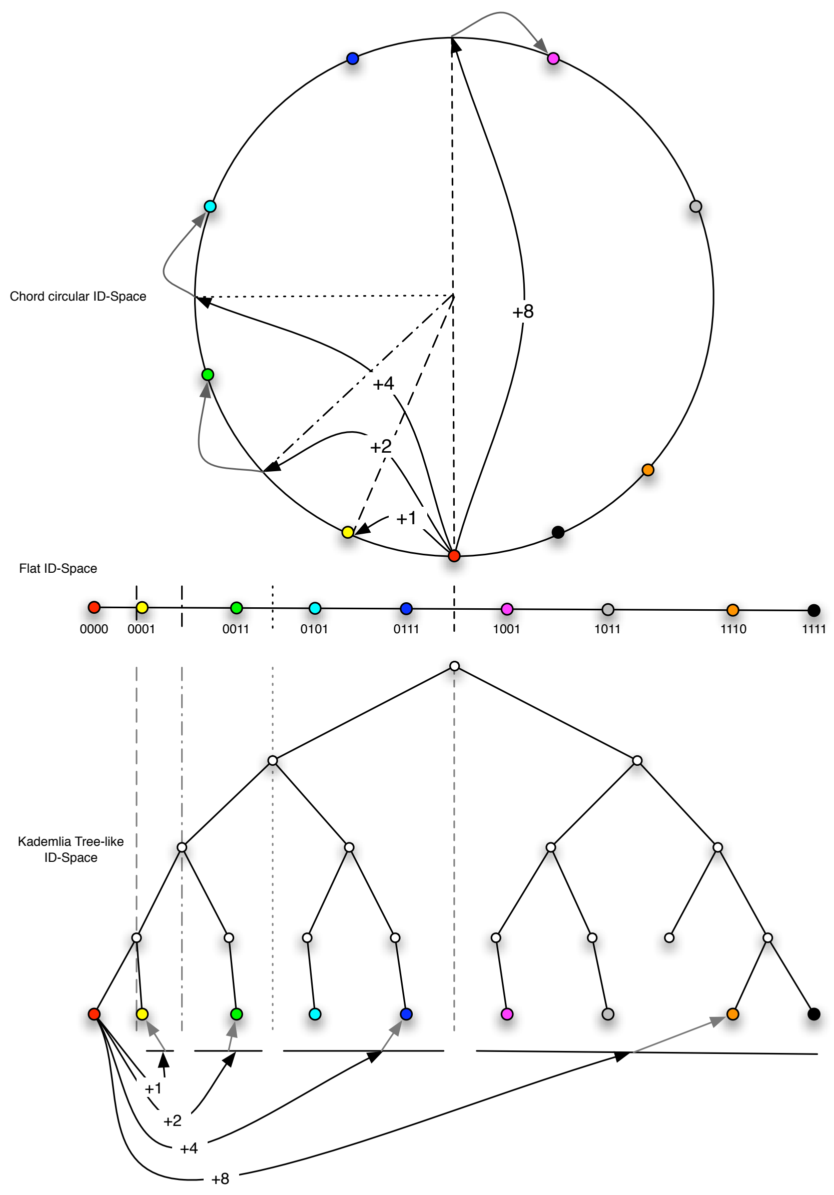


Figure B.1: Example fingers in Chord and Kademlia

distance-metric (commonly RTT). Also note that while in Chord the dividing lines lie on specific nodes; that is, the representation allows drawing a node on the line, whereas the representation as a tree necessitates infinitely small lines that do not actually go through any nodes, although theoretically they would.

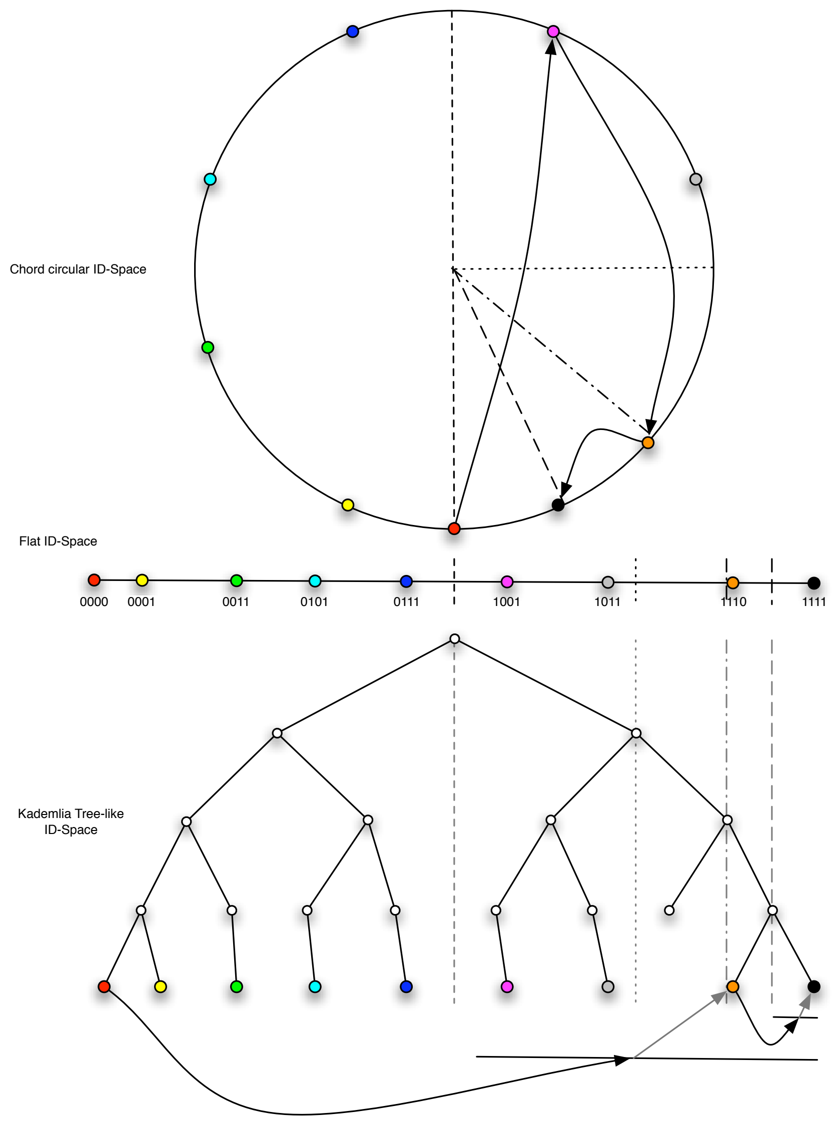


Figure B.2: Example of a lookup in Chord and Kademlia

Figure B.2 shows a lookup of Key15(1111) starting at Node0(0000). Kademlia termi-

Table B.1: Interface of the KBR according to the Common API [6]

Function	Description
$route(key, msg, hint)$	Routes *msg* to the node that is closest to the given *key* using *hint* for the first hop
$forward(\&key, \&msg, \&nextHop)$	Function in the application that is called on each intermediate hop while routing towards *key*
$deliver(key, msg)$	Delivers a *msg* to the application on the node that is closest to the *key*
$node[]\ localLookup(key, num, safe)$	Produces a list of up to *num* nodes that can be used as a next hop for routing towards *key*
$node[]\ neighborSet(num)$	Returns up to *num* nodes from the local neighbor-set
$node[]\ replicaSet(key, maxRank)$	Returns nodes from the replica-set for *key* of nodes up to *maxRank*
$update(node, joined)$	Function in the application that is called whenever the local neighborhood changes

nates one step earlier than Chord because the first finger for the other half-tree happens to correct three bits in one step, but it could have just as well been the same finger as in Chord, thus using the same number of steps.

B.2 The DHT

Once the KBR is in place, all the DHT has to do is to build a layer above it that will implement the well-known hashtable interface.

Table B.2: Interface of a DHT according to the Common API

Function	Description
$put(key, val)$	Stores *val* in the DHT under *key*
$remove(key)$	Removes the value for given *key*
$val = get(key)$	Retrieves *val* for *key* from the DHT

For that, it needs a way to map keys into the ID-space. Most DHTs use a hash function that distributes IDs statistically even, like SHA-1. To implement the familiar *put* and *get* after the key is mapped into the ID-space, the DHT only needs to route a message towards the hash of the key ($h(key)$), asking whoever gets it to store the value. Because everybody has to agree on the key to be able to look it up, everybody can use it's hash to route a message towards it. A simple *get* message routed to the owner of $h(key)$ will retrieve the value again.

Appendix C

FreePastry

PWHN uses FreePastry because it provides the services of a KBR; that is, message routing. This appendix provides a brief introduction to FreePastry and its functionality.

C.1 Pastry

The Pastry DHT [41] defines its ID-space to be a circular, 128-bit ring. IDs are assumed to be base 4 but can be defined in any base. Like any DHT, Pastry guarantees correction of at least one digit in any base at each hop. Pastry defines its proximity as clockwise distance on the ring, like Chord; thus, it is not symmetric. The advantages over Chord [45] lie in its locality properties, as well as the ability to route to any node whose next bit differs (Chord restricts its fingers to the nodes at exact places on the ring or their successors, e.g. the node exactly half-way around the ring).

Pastry nodes keep track of nodes at certain positions relative to their own in their *routing table*. These positions are in increasing distance to those nodes that share the current nodes' prefix and have a different next digit. In the circular ID-space, these end up being the nodes one-half revolution (or more) away, between one-quarter and one-half revolution away, and so on. Nodes also keep track of nodes that are closest to themselves in the ID-space from both directions in their *leaf-set*, and of those nodes that are closest in terms of the proximity-metric in their *neighborhood-set*.

The leaf-set is the first that is checked while routing a message to see if the requested key falls within the ownership of one of its entries. If it does not fall within the leaf-set, the routing table is used to find a node that is closer by at least one digit. This essentially

Table C.1: Interface of Pastry

Function	Description
nodeID=Init(Cred,Appl)	Joins a Pastry ring and registers *Application* with Pastry
route(msg,key)	Forwards *msg* to the closest node of *key*

bypasses the restriction of only forwarding clockwise around the ring. Once the message is close enough to the target node, the algorithm is switched from distance-metric to numeric difference routing.

The neighborhood set is used to help joining nodes find the closest nodes in the proximity space for their routing table.

The *Application* has to export the following operations.

C.2 FreePastry

FreePastry is Princeton's implementation of Pastry's KBR in Java. It exports Pastry's API, as well as the Common API (see Appendix B). In FreePastry, IDs are 160-bit long.

FreePastry defines the entries in its fingertable in base 16. That means that the fingertable has 40 rows with 16 columns each. 40 rows for each digit in the ID, and 16 columns for each of the 16 possible states of a digit. Each cell contains a list of nodes (if there are any) that match the first digits of the current node's ID and differ in the *rowth*-digit,

Table C.2: Interface that an application using Pastry has to export

Function	Description
deliver(msg,key)	Delivers *msg* to the application running on the closest node to *key*
forward(msg,key,next)	Called while forwarding *msg* towards *key*, giving the *next* hop
newLeafs(LeafSet)	Called whenever the *LeafSet* changes

having *column* as the digit. Thus, FreePastry tries to correct 4 bits at a time. If there is no node in that cell, FreePastry still tries to get closer to the target key by checking all node IDs for numeric closeness. The node that is absolutely closest in plain numeric space to the target ID is chosen for routing.

It also includes PAST, which is a DHT built on top of Pastry; SCRIBE, a publish/subscribe system; SplitStream, which is a CDN; and Glacier, another DHT implementation.

Appendix D

Coral

This appendix gives an overview of Coral and the Kademlia-DHT it uses.

D.1 Kademlia

Kademlia is a DHT that defines its ID-space to be a flat, 160-bit space, that uses a novel "XOR" distance-metric. It assumes its IDs to be base 2, for simplicity of routing. In contrast to Pastry it uses the XOR distance-metric, which is symmetric. Thus, it is easier to visualize the routing algorithm of Kademlia as a tree. Symmetric distance metrics have the added advantage of learning of peers by passively monitoring the routing traffic because it comes from both directions, whereas asymmetry inhibits this because forwarded messages come from the wrong direction.

Because of the fact that Kademlia uses only one algorithm for lookups all the way until the end, the necessary node state is reduced. For every i out of $0 \leq i < 160$ it keeps a list called k-bucket with up to k pointers to nodes, which are exactly i bits away. That is, for each ith k-bucket the first $i-1$ bits (the prefix) are the same, and the ith bit is flipped. k is typically a very small number, like 6.

The k-bucket lists implement a least-recently seen eviction policy, i.e. newer nodes are purged first, except that live nodes are never removed. This is based on the observation in Gnutella networks that older nodes are likely to stay longer.

The Kademlia Protocol consists of four RPCs.

PING merely pings a node to see if answers. FIND_NODE takes an ID as an argument and returns the k closest node to that ID known to the called node. FIND_VALUE does

Table D.1: RPC Interface of Kademlia

Function	Description
bool=PING()	Pings the callee
STORE(key,val)	Stores *val* under *key* at called node
nodes[]=FIND_NODE(ID)	Returns up to k known closest nodes for *ID*
nodes[]=FIND_VAL(key)	Like FIND_NODE, except returns the *val* when stored at called node

the same with one exception; if the recipient of FIND_VALUE has seen a STORE for that key, it will return the value instead. STORE stores a value under the key at the recipient.

To store and retrieve (key,value) tuples a Kademlia node needs to find the k closest nodes to a key. For that, it employs a recursive algorithm as follows. The local node starts with calling FIND_NODE in parallel on the α closest node it knows of, which it picks out of the appropiate k-bucket. Some of those will return nodes that are closer. Without waiting for all calls to return, the node will continue to call FIND_NODE on the nodes returned by the first iteration. When the k closest nodes seen so far have all returned without yielding any closer nodes, the lookup terminates.

For storing the value, the node will then call STORE on those k closest nodes. A lookup uses FIND_VALUE instead of FIND_NODE, which will return as soon as it finds a replication of the value and terminates the lookup.

Because all stored values are soft-state, they must be renewed ever so often. Replication nodes re-publish values they store every hour. The original owner is required to re-publish its content every 24 hours, otherwise it will expire after a day. Furthermore, each node will replicate the content locally after a successful lookup.

D.2 Coral

Coral-CDN uses its DHT which is a modified Kademlia to store content that its clients request. Whenever a client requests a file, Coral issues a GET to the DHT and delivers that content if found in the DHT. The Coral-CDN consists of three parts:

- *Coral*, the DHT itself;
- *WebProxy*, the webserver that client browsers issue requests to; and
- *DNSProxy*, the DNS daemon that resolves "coralized" URLs to the nearest WebProxy.

A URL is *coralized* by adding the domain-suffix "*.nuyd.net:8090*" to it. When a browser tries to resolve the Top Level Domain (TLD) *.nuyd.net*, the request is sent to the Coral-DNS-Proxy which tries to map the client's location to the nearest WebProxy. The answer returned will cause the browser to connect to the WebProxy on that address and issue its request. The WeProxy in turn looks up the file in Coral and either returns it if found, or requests it from the origin server, returns that, and stores it into Coral.

Currently, Coral is implemented in roughly 22.000 lines of C-code, which makes heavy use of C++ templates. It uses the *libasync* library, that is part of SFS [42], for asyncronous IO as well as ONC RPC for remoting.

For added efficiency, Coral makes a few additions to the original Kademlia.

Firstly, in an effort to keep traffic within a local neighborhood, Coral implements so-called *clusters*. Coral nodes will automatically partition themselves into clusters according to some distance-metric in the proximity space. The only currently supported distance-metric is RTT. A node will attempt to join a cluster when the distance to 90% of the nodes is below a certain diameter, or threshold. For the thresholds, values of 20, 60, and ∞ ms were found to be most useful.

All nodes that are members of a cluster will form their own Kademlia DHT. Thus, each node is a member of as many DHTs as there are cluster-levels. This is globally defined and should be set to some small number, for example 3. In each of those DHTs, each node assumes the same respective ID. Lookup, store, and get operations are augmented by a level parameter which specifies the target cluster-level for the operation.

When doing a lookup operation, a node will first attempt to find the closest node to the key in its smallest (level-3) cluster. Since that node has the same ID in the next higher-level cluster, it can be used as the first *hint* for that cluster. Consequently, after FIND_NODE in a higher-level cluster returns, a node will continue calling FIND_NODE with the level decreased by one on the last returned (closest) node.

Instead of storing values only once in the global DHT, a publisher has to store it in each cluster's DHT it is a member of. Lookups start in the smallest cluster and continue to travel outwards. This guarantees that the value will be found in the smallest neighborhood where it is stored, thus keeping traffic from unnecessarily traversing high-latency links.

Secondly, instead of allowing only one value per key, Coral allows multiple values per key and requires a get to only return a subset of those. This is advantageous because of the way the web-cache is implemented, Coral has multiple readers and writers for tuples. The Coral-CDN does not store the actual content of URLs, it only stores pointers to nodes that have it. Every Coral node that retrieves the content inserts a pointer to itself under the key of that content. Lookups will return a subset of those nodes, which are possibly in the smallest neighborhood (level-3 cluster) of the querying node. The authors call this augmented DHT a Distributed Sloppy Hashtable (DSHT).

D.2.1 libasync

Libasync is built around the insight that heavy threading does not necessarily make an application faster because of the incurred context switching overhead. This is especially true in linux environments which do not support light-weight threads at the OS-level. If everything is run in only one thread instead, the overall performance should benefit. This design prohibits the use of blocking calls, however. In lieu of using complicated event-notification techniques, `select` can be used to wait for an event. Libasync has been built to make this process easier.

An application interested in an event on any kind of file descriptor calls libasync with the descriptor, the event it is interested in, and a callback. As soon as the event takes place, the callback is called.

D.2.2 ONC RPC

Remote Procedure Call (RPC) was invented in 1976 by Sun and was later adopted and extended by the ONC foundation. It is a standard for calling procedures remotely, according to the client/server paradigm. Before it is sent out over the wire, data is transformed into a common format, known as eXternal Data Representation (XDR). RPC services listen on random ports above 1024 that are assigned on run-time. To solve the problem of knowing which service runs on what port, RPC uses the portmapper that listens for queries on the well-known TCP and UDP port, 111. ONC RPC is described in RFC 1831 [44].

Interfaces are written in an Interface Definition Language (IDL) very similar to C with the file-suffix ".x." These files are compiled by "rpcgen" into C files that can be used in programs to set up the server and client, respectively. They take care of transforming all data back and forth between XDR. Implementations, as well as the necessary "rpcgen" programs, exist for other languages such as Java [39] and .NET [7] (commercial).

Appendix E

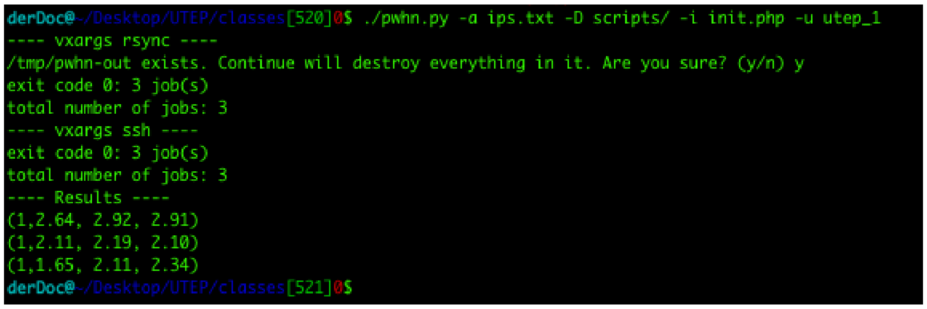
PWHN script tutorial

An initial version of PWHN was constructed as a set of python scripts that implement much of PWHN's final semantics. This appendix is a tutorial on its usage. Most of this is also available as a webpage at [32].

E.1 Screenshots

E.1.1 Example 1

Example run with only the Init script given in the directory scripts.



```
derDoc@~/Desktop/UTEP/classes[520]0$ ./pwhn.py -a ips.txt -D scripts/ -i init.php -u utep_1
---- vxargs rsync ----
/tmp/pwhn-out exists. Continue will destroy everything in it. Are you sure? (y/n) y
exit code 0: 3 job(s)
total number of jobs: 3
---- vxargs ssh ----
exit code 0: 3 job(s)
total number of jobs: 3
---- Results ----
(1,2.64, 2.92, 2.91)
(1,2.11, 2.19, 2.10)
(1,1.65, 2.11, 2.34)
derDoc@~/Desktop/UTEP/classes[521]0$
```

Figure E.1: Example run of a script that emits load averages for each server

The script emits a 4-tuple for each server it is run on, consisting of the number 1 and the load average. of the `uptime` command in the form $(1, 1min - load, 5m, 15m)$. This form gives the loads of each server it is run on.

script The Init script looks like this:

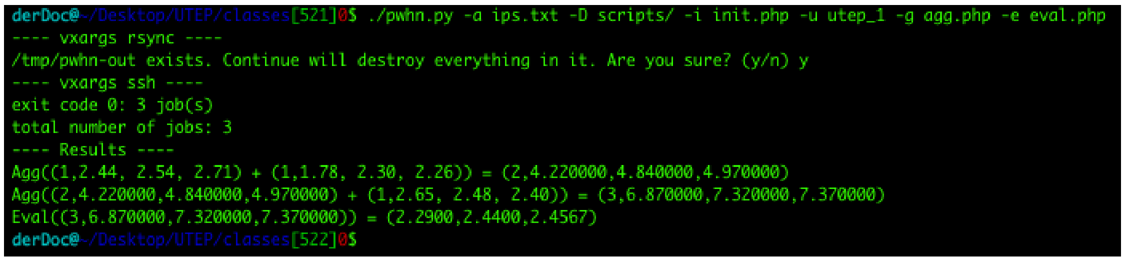
Listing E.1: Init script for example 1

```
#!/usr/bin/php -f
<?php
        $u = `uptime`;
        $load = trim(substr($u,strrpos($u,':')+1));
        $vals = explode( (strpos($load,',')>0)?',':' ',$load);
        printf("(1,%s,%s,%s)",$vals[0],$vals[1],$vals[2]);
?>
```

It just runs ‘uptime‘, gets the rest of the output after the “:” and splits the numbers at either “,” or “ ”, depending on which is there. Then it emits the tuple to `stdout`. It needs to be executable and thus has an appropriate shebang.

E.1.2 Example 2

To compute the average load over all the servers, the number 1 from example 1 is needed since the average function is **not** a prefix funtion. This means that $avg(a+b) \neq avg(a) + avg(b)$.



```
derDoc@~/Desktop/UTEP/classes[521]0$ ./pwhn.py -a ips.txt -D scripts/ -i init.php -u utep_1 -g agg.php -e eval.php
---- vxargs rsync ----
/tmp/pwhn-out exists. Continue will destroy everything in it. Are you sure? (y/n) y
---- vxargs ssh ----
exit code 0: 3 job(s)
total number of jobs: 3
---- Results ----
Agg((1,2.44, 2.54, 2.71) + (1,1.78, 2.30, 2.26)) = (2,4.220000,4.840000,4.970000)
Agg((2,4.220000,4.840000,4.970000) + (1,2.65, 2.48, 2.40)) = (3,6.870000,7.320000,7.370000)
Eval((3,6.870000,7.320000,7.370000)) = (2.2900,2.4400,2.4567)
derDoc@~/Desktop/UTEP/classes[522]0$
```

Figure E.2: Example run of a script that emits the load average over all the servers

Thus, you need to push the division to the end of the execution into the evaluator. In the intermediate results, the overall sum of all elements seen so far is carried. To know how many elements have been seen, each one is emitted with a leading 1 which is summed in the intermediate results. An example run of the script with all the map, reduce and eval scripts looks like E.2.

script The Init script looks like E.2.

Listing E.2: Init script for example 2

```
#!/usr/bin/php -f
<?php
        $u = `uptime`;
        $load = trim(substr($u,strrpos($u,':')+1));
        $vals = explode( (strpos($load,',')>0)?',':'␣',$load);
        printf('(1,%s,%s,%s)',$vals[0],$vals[1],$vals[2]);
?>
```

The aggregator looks like E.3.

Listing E.3: Aggregate script for example 2

```
#!/usr/bin/php -f
<?php
        fscanf(STDIN, "%d\n", $l1);
        $first = fread(STDIN,$l1);
        fread(STDIN,1);
        fscanf(STDIN, "%d\n", $l2);
        $second = fread(STDIN,$l2);
        $v1 = explode(',',substr($first,1,strlen($first)-2));
        $v2 = explode(',',substr($second,1,strlen($second)-2));
        printf("(%u,%f,%f,%f)",
                        (round($v1[0]+$v2[0])),
                        ($v1[1]+$v2[1]),
                        ($v1[2]+$v2[2]),
                        ($v1[3]+$v2[3])
        );
?>
```

This script takes two runs from Init on `stdin`, reads them into `$first` and `$second`, deletes the surrounding parentheses and $first and $second on ",". It then adds all values and emits a new tuple.

And finally, E.4 is the evaluator.

Listing E.4: Eval script for example 2

```
#!/usr/bin/php -f
<?php
        $in = trim(stream_get_contents(STDIN));
        $v = explode(',',substr($in,1,strlen($in)-2));
        printf("(%.4f,%.4f,%.4f)",
                        ($v[1]/$v[0]),
                        ($v[2]/$v[0]),
                        ($v[3]/$v[0])
        );
?>
```

This script takes the output of the last reduce run on `stdin`, deletes the surrounding parentheses and splits it on ",". It then divides all fields by the first number and prints the output to `stdout`.

E.2 ManPage

```
Usage:

NAME
  plmr.py - simple map-reduce script with arbitrary executables.

DESCRIPTION
  Syncs a file or directory with a remote server and executes init
  script remotely using vxargs.
  Most of the options are passed directly to vxargs.
  For vxargs to work the shell has to know where to find an ssh-agent
  that has the right identity added.

SYNOPSIS
  plmr.py [OPTIONS] -a <file> -i <file>

OPTIONS
  --file=filename, -f file
    Sources a file that specifies all options.
    Certain options may be overriden on the commandline.
    File is in Python syntax and should look like this (default settings):
        syncdir =   ""
        initfile =  ""
        aggfile =   ""
        evalfile =  ""
        outdir =    "/tmp/plmr-out"
        username =  ""
        no-sync =   0
        synconly =  0
        quiet =     0
        vxargs =    [] #list of single strings to give to vxargs, ex: [ "-a ips.txt" ]

  --nosync, -s
    If every server is already up to date, syncing can be switched of by
    specifying --no-sync to speed up execution.

  --synconly, -l
    To only sync, on the other hand, without executing, use synconly.

  --quiet, -q
    Supresses any output including output from vxargs, the only thing written
    to stdout is the last result.
    Implies -p and -y for vxargs.

  --help
    Print a summary of the options   and exit.

  --init=filename, -i file
    Specifies the name of the initializer script to run remotely.
    If <dir> is not specified only this file will be synced.
  --agg=filename, -g file
    Specifies the name of the aggregator script.
    If <dir> is specified this file will be assumed to be in <dir>.

  --eval=filename, -e file
    Specifies the name of the evaluator script.
    If <dir> is specified this file will be assumed to be in <dir>.

  --dir=dirname, -D dir
    The directory to syncronise. If specified all files have to be in that directory.

  --user=username, -u username
    The username to use for remote login, if different from current logged in user.

  --version
    Display current version and copyright information.


 Options passed to vxargs
  --max-procs=max-procs, -P max-procs
    Run up to max-procs processes at a time; the default is 30.

  --randomize, -r
    Randomize the host list before all execution.

  --args=filename, -a filename
    The arguments file.

  --output=outdir, -o outdir
    output directory for stdout and stderr files
    The default value is specified by the environment variable VXARGS_OUTDIR.
    If it is unspecified, both stdout and stderr will be redirected
    to a temp directory.
    Note that if the directory existed before execution, everything
    inside will be wiped without warning.

  --timeout=timeout, -t timeout
    The maximal time in second for each command to execute. timeout=0
    means infinite.  0 (i.e. infinite) is the default value. When the time is up,
    vxargs will send signal SIGINT to the process. If the process does not
    stop after 2 seconds, vxargs will send SIGTERM signal, and send SIGKILL
    if it still keeps running after 3 seconds.

  --noprompt, -y
    Wipe out the outdir without confirmation.

  --no-exec, -n
    Print the commands that would be executed, but do not execute them.

  --plain, -p
    Don't use curses-based output, but plain output to stdout
    instead. It will be less exciting, but will do the same job
    effectively.
    By default, plmr.py uses the curses-based output from vxargs.

SCRIPTS
```

```
 Initialiser
    The init executable is run remotely.
    It has to be an executable file, but could also be a script that is
    executable and has a working shebang.
    It is expected to write its output to stdout.
    An output could look like this, for example:
        (1,3.0,2.0)
    If this is the only script given, the output of every run will be printed
    to stdout seperated by newlines.

 Aggregator
    The agg executable is run locally.
    It has to be an executable file, but could also be a script that is
    executable and has a working shebang.
    It will receive the outputs of 2 init runs on stdin and is expected
    to write one aggregate to stdout.
    The inputs will both start with the exact length in bytes, written
    in ASCII on a line by itself.
    Then the input followed by a newline which is not counted as part of the input.
    The input could look like this, for example:
        11\n
        (1,3.0,2.0)\n
        12\n
        (1,1.65,0.8)\n
    Then the output could look like this:
        (2,4.65,2.8)
    If no aggregator is given, the input to the evaluator will be all outputs
    of the init script,
    seperated by the lengths in ASCII on lines by themselves like the input to
    the aggregator,
    except possibly consisting of more than two records.

 Evaluator
    The eval executable is run locally.
    It has to be an executable file, but could also be a script that is
    executable and has a working shebang.
    It will receive the last aggregate value on stdin and is expected to
    deliver the final result to stdout.
    Continuing the above example, the final result for the input:
        (2,4.65,2.8)
    could be:
        (2.325,1.4)
    If no evaluator is given, the output of the last aggregator will be
    written to stdout.


EXAMPLE
    Suppose the following conditions hold (the files have the stated contents).
    $ cat ips.txt
    192.41.135.218
    #planetlab1.csg.unizh.ch
    129.108.202.10
    #planetlab1.utep.edu
    131.175.17.10
    #planetlab2.elet.polimi.it

    $ cat scripts/init.php
```

```
#!/usr/bin/php -f
<?php
    $u = `uptime`;
    $load = trim(substr($u,strrpos($u,":")+1));
    $vals = explode( (strpos($load,",")>0)?",":" ",$load);
    printf("(1,%s,%s,%s)",$vals[0],$vals[1],$vals[2]);
?>

$ cat scripts/agg.php
#!/usr/bin/php -f
<?php
    fscanf(STDIN, "%d\n", $l1);
    $first = fread(STDIN,$l1);
    fread(STDIN,1);
    fscanf(STDIN, "%d\n", $l2);
    $second = fread(STDIN,$l2);
    $v1 = explode(",",substr($first,1,strlen($first)-2));
    $v2 = explode(",",substr($second,1,strlen($second)-2));
    printf("(%u,%f,%f,%f)",
                    (round($v1[0]+$v2[0])),
                    ($v1[1]+$v2[1]),
                    ($v1[2]+$v2[2]),
                    ($v1[3]+$v2[3])
    );
?>

$ cat scripts/eval.php
#!/usr/bin/php -f
<?php
    $in = trim(stream_get_contents(STDIN));
    $v = explode(",",substr($in,1,strlen($in)-2));
    printf("(%.4f,%.4f,%.4f)",
                    ($v[1]/$v[0]),
                    ($v[2]/$v[0]),
                    ($v[3]/$v[0])
    );
?>

Then executing
    ./plmr.py.py -D scripts/ -i init.php -g agg.php -e eval.php -a ips.txt
will output something similar to
    exit code 0: 3 job(s)
    total number of jobs: 3
    exit code 0: 3 job(s)
    total number of jobs: 3
    Agg((1,2.57, 2.64, 2.64) + (1,1.73, 1.84, 1.84))
     = (2,4.300000,4.480000,4.480000)
    Agg((2,4.300000,4.480000,4.480000) + (1,1.55, 1.40, 1.42))
     = (3,5.850000,5.880000,5.900000)
    Eval((3,5.850000,5.880000,5.900000)) = (1.9500,1.9600,1.9667)
```

Appendix F

CD contents

Contents of the included CD-ROM.

The folder "/tex" is the root of all *tex* sources. The main files for the thesis as well as the makefile is in the folder 2006-vitus-pwhn-thesis, all other included folders contain various files that the thesis includes. The root contains a modified version of the acronym package's *acronym.sty* that allows the possessive form of acronyms to be used by typing " aco{...}" in the source. This adds an "ś" to the output. Images are separated in figures, screenshots, logos and graphs.

"/src" contains all sources of PWHN as java files. The Namespace utep.plmr.* is in the folder "/utep.plmr" its subfolders. Helper functions for graphing of PathInfos are in "grapher", "jfree" contains my subclassed classes for the JfreeChart library, and server contains all server classes. "/thesis-scripts" contains all example scripts from section 5.3. "/Fern" contains some example log files from fern and the script used to collect and aggregate these logs.

"/dist" contains the complete binary distribution. All necessary ".jar" Files are included in the respective archives. "PLMR-server" contains the server jar, as well as the parameter file for FreePastry which enables debugging. Also included are sample output files that the server produces while running, in "/logs". These are the log of the fingertable (every minute), the stdout output of the server, and the debugging output produced by FreePastry itself. "start-plmr.sh" and "stop-plmr.sh" are shell scripts to start and stop the server. They have to be executed in the home directory, and can be used with the PWHN client. Start only starts the server if not already running, allows specifying a bootstrap IP and port which defaults to *planetlab1.utep.edu:9091*. The client includes two sample runs with fictitious data to simply test the charting functions. These do not include any PathInfo annotations.

"/javadoc" contains the complete javadoc for the whole project. "pwhn-site" is a copy of the website created for the PLMR project for PlanetLab. To be able to see all the pages, it has to be parsed by PHP Hypertext Preprocessor (PHP), which is why it has to be put into a webservers *cgi* directory, for example into apache's wwwroot. "geshi" is a source code highlighter for PHP. It is released under the General Public License (GPL).

```
\
    \tex                                - root of tex sources
    acronym.sty                         - modified version of acronym package
    Makefile                            - makes all papers
        \2006-vitus-pwhn-thesis         - thesis
            Makefile                    - makes the thesis.pdf file
        \2006-dht-pp_soat_i-techreport  - includes for data structures section (and tech report)
        \2006-p2pmon-survey             - includes for survey of related sys section (and survey)
        \bib                            - includes for bibliography, moacros & acros
        \figs                           - figures
        \graphs                         - graphs
        \img                            - logos
        \screens                        - screenshots

    \src                                - source of PWHN
        \utep
            \plmr                       - client-files in NS utep.plmr
                plmr.uml                - UML diagram in XML format (from Eclipse UML)
                \jfree                  - JfreeChart classes
                \server                 - NS .server (classes for server)
                    server.uml          - UML diagram of server in XML format
                    \messages           - NS .server.messages (messages for serialisation)
        \thesis-scripts
            \3-8                        - script sets 1-4
        \fern
            globalDist               - python global Distribution 3-in-1 file for Fern
            autoDict.py              - auto dictionary needed for globalDict
            \logs                    - some example logs from Fern


    \dist
        licence-LGPL.txt             - required LGPL licence
        FreePatry License.txt        - FP's license
        \PWHN-1.0                    - client: complete binary dist.
            PWHN-1.0.jar             - client GUI
            vxargs.py                - needed vxargs script
            PLMR.py                  - PWHN script version
            jcommon-1.0.4.jar        - JCommon lib for JFreeChart
            jfreechart-1.0.1.jar     - JFreeChart lib
            jgraph.jar               - JGraph lib
            FreePastry.jar           - also needed for client
        \PLMR-server-1.0             - server: complete binary dist.
            FreePastry-20b.jar       - FreePastry jar library
            PLMR-server-1.0.jar      - the server jar file
            freepastry.params        - parameter file for FP (enables debugging)
            start-plmr.sh            - script to start the server on current node
            stop-plmr.sh             - script to stop
            \logs                    - sample output of server
                fingertable.log      - log of FP's fingertable
                plmr-server.out      - stdout output of server
                pwhn_*.fp_log        - debugging output of FP (FINER)

    \javadoc                         - the javadoc for whole NS utep.plmr


    \pwhn-site                       - copy of PLMR-project website
        \geshi                       - source code highlighter for PHP
        \plmr                        - PLMR website root
            \files                   - all downloadable files
            \img                     - images
```

```
\inc                          - include files that contain all subpages
```

Curriculum Vitae

Vitus Lorenz-Meyer was born in Hamburg, Germany on July 31, 1980 as the second child to Monika and Dr. Georg-Christian Lorenz-Meyer. While Vitus' brother grew up going to elementary school in Hamburg itself, his parents moved to the outskirts soon after his birth, and so he went to the *Fürstin Ann-Marie von Bismarck* elementary school in Aumühle just east of Hamburg. At the age of ten he finished elementary school and transfered to the *Hansa-Gymnasium Hamburg-Bergedorf*, from which he graduated in 2000.

As civil or military service is mandatory in Germany, he chose to do something for the greater good, and worked in a church for 11 months.

In the winter semester of 2001 he started his studies of C.S. at the *Technische Universität Braunschweig*, in Braunschweig, Germany. After four semesters, in the fall of 2003, he finished his *Vordiplom*, which loosely translates to "preliminary exam", as specified by the study plan, shy only of one exam which he completed in Spring 2004.

In the summer prior, he had already started making plans for going abroad. The final admission paperwork was due at the International Office by the end of October. By the end of spring he received word that he was accepted. For the International Student Exchange Program (ISEP) program, which he chose because it allowed him to go to North America (where he had always wanted to go on an exchange to - but never came around to during high school) and of which his university is a member, participants have to make up a list of universities they would like to attend. One of his choices was New Mexico State University (NMSU) in Las Cruces, New Mexico.

For the fall semester of 2004, he transfered to NMSU as a full-time exchange student. His stay was going to be one full academic year. In the first semester of his stay, however, he had already started to like the southwest and decided to stay. On December 1st, 2004 he submitted his full application as an international student to the Graduate School. However, the educational differences between Germany and the requirements of the NMSU Graduate School denied him entrance.

In the beginning of May, a friend of his, an old professor who taught photography at NMSU for 25 years, suggested to consider applying to University of Texas at El Paso (UTEP). The next day they drove down to El Paso together and visited the Computer Science (C.S.) Department. There he met Eric Freudenthal, a professor of the C.S. Department, who would later become his advisor. What was planned to be a short visit turned out to be a four hour talk which left both sides with a positive impression.

The final decision for admission into the graduate program at UTEP lies within the Department, whereas at NMSU it lies within the graduate school. The night before his scheduled returning flight to Europe on July 1st, Eric called, and said: “You are in.”

By August 11th he had returned to UTEP and moved into Miner Village, the campus residences. Towards the end of his first semester he started reading most of the material concerning to what would later become his Master’s thesis topic. During the middle of his second semester he pin-pointed his topic and began writing his thesis. Six months later, the final product was finished.

Permanent address: Mortagneweg 12
21521 Aumühle
Germany

This thesis was typset with LaTEX2$_e$ by the author.