# KELP: An Architecture for Understanding Global System Behavior in Massively Scalable Distributed Systems

by

Arthur Yung

B.Sc., University of British Columbia, 1998

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

**Master of Science**

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standard

**The University of British Columbia**

April 2001

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date April 26, 2001

# Abstract

Current techniques do not scale distributed systems to millions of nodes because they cannot handle global behavior description and global coordination to such massive sizes. KELP addresses these problems with a loose, decentralized system of nodes that are connected together with a small-world network. Viewed as a network of randomly connected clusters, a small-world network supports massive scalability with its random connectivity while still supporting locality within its clusters. KELP uses two key properties of randomness to scale. First, there is a short typical distance of separation between any two nodes in the system. This is used to quickly infer global behavior. Second, nodes have relatively little knowledge of the overall system, which helps provide looser semantics for global coordination. Finally, on top of the small-world infrastructure, KELP provides massively scalable data structures to make building massive scale systems less ad-hoc.

# Contents

# List of Tables

# List of Figures

# Acknowledgements

I'd like to thank the National Sciences and Engineering Research Council (NSERC) for financially supporting this thesis with a postgraduate scholarship. I'd also like to acknowledge the invaluable contribution of Stephan Gudmundson who acted like a partner for most of this project. Thanks to Mike Feeley and Norm Hutchinson for their guidance and thanks to Lisa Streit for proofreading this thesis.

ARTHUR YUNG

*The University of British Columbia*
*April 2001*

To mom. I am nothing without you.

# Chapter 1

# Introduction

The Internet provides the infrastructure to support distributed systems comprised of hundreds of thousands or even millions of nodes. To date, the main focus on scalability has been building centralized servers that handle large numbers of clients; however, this model of computing does not scale to massive sizes because centralization is a bottleneck. Also, this approach does not support more peer-to-peer applications such as peer-based file-sharing system with many users and decentralized distributed virtual environments. This thesis describes a system designed to scale past current limitations and to support a wider range of applications.

Massively scalable distributed systems that support millions of users on a global scale are challenging to build because massive amounts of global information are difficult to manage in a timely fashion. Scaling to such large sizes also requires looser consistency and lower coherence, which makes global coordination impractical. Furthermore, there are no general or reusable methods for writing large scale systems; up to now, the loose semantics that support massive scale have been followed in an ad-hoc manner.

Some massive scale systems have been built, but only in system-specific ways.

For example, DNS [14] scales well using hierarchy because it takes advantage of infrequent name changes and caches names at various hierarchical levels to avoid the root from becoming a bottleneck. The WWW [6] also scales well because it is comprised of millions of different web servers that each scale independently. There is no global information shared between web servers, so the millions of web servers do not have the overhead of coordinating with one another.

Our goal is to provide a mechanism for massively scaling a range of distributed systems. In particular, we wish to massively scale non-hierarchical distributed systems that share global knowledge among all their nodes and we wish to do so without using an ad-hoc approach.

Our strategy to achieve our goal is to follow the looser, scalable semantics in a serverless and decentralized manner; every node knows about a few other nodes in the system and global knowledge is partitioned among all nodes. Nodes within the system interact in a peer-to-peer manner and clients can interact with any node in the system.

Following this strategy, we have designed a massively scalable system called KELP. To achieve massive scale in a system where every node only knows a few other nodes, node connectivity in KELP is modeled after natural systems that grow to large sizes. Natural systems like the neural network of the worm Caenorhabditis elegans, the power grid of the western United States [17], and hyperlinks on the World Wide Web [3] are all connected via a *small-world network*. Small-world networks allow these natural systems to grow to large sizes because they take advantage of locality while providing a short separation distance between any two nodes in the system.

A small-world network is a collection of tightly coupled clusters that are

loosely and randomly connected to a few other clusters. The tightly coupled clusters provide locality while the loose connectivity of the clusters allow information to be propagated from one end of the system to the other in just a few hops. For example, infectious disease is predicted to be spread extremely quickly among organisms connected via a small-world network [17].

Since clusters are a well-known technology [15], we focus our research on the random connectedness of the small-world graph. To investigate this, we devised an algorithm for building a decentralized distributed system comprised of millions of nodes connected via a random graph. The random graph connects every node to a few other randomly-chosen nodes and provides key properties for scaling our system: a short distance between any two nodes that allows for quick information propagation and minimal knowledge at each node that allows for loose global coordination. Coupling these properties with global knowledge partitioned among all nodes provides a method for quick dissemination of global information throughout the system.

In order to evaluate our system, we built two distributed data structures on top of KELP's random graph connection: a set and a queue. In order to scale these data structures, their commonly-accepted semantics had to be loosened. For example, a dequeue operation on the queue returns an element that is very close to the oldest instead of strictly the oldest. Applications using these data structures to achieve massive scalability must tolerate these loosened semantics.

This thesis is organized as follows. Chapter 2 describes the graph theory used to address the problems of achieving massive scalability and also describes how KELP uses a random graph as its infrastructure. Chapter 3 explains how the infrastructure can be used to describe global behavior. Chapter 4 describes the

data structures built on top of the graph infrastructure. Chapter 5 describes the implementation of KELP's simulation. Chapter 6 shows the results of evaluations that show the properties of the constructed graphs and the behavior of the data structures. Chapter 7 describes related work. Finally, Chapter 8 concludes the thesis and describes future work.

# Chapter 2

# Infrastructure

KELP's infrastructure provides connectivity between nodes and is designed to address scalability by providing services that support the inference of global behavior. To support these operations, the nodes' connectivity is modeled after large-scale natural systems that connect nodes via a small-world network.

This chapter begins by describing the general problems in attempting to achieve massive scalability. Then, the motivation of modeling KELP's infrastructure on small-world networks is explained. Next, the chapter focuses on the random connectedness of the small-world network and explains how random properties help provide massive scalability. Finally, this chapter ends by explaining how KELP's infrastructure is built and how nodes failures are handled.

## 2.1 Problems with Achieving Massive Scalability

Current techniques do not scale distributed systems to massive sizes because global information management and global coordination are bottlenecks. There is also no general method of creating a scalable system, which makes building these systems

difficult.

Global information management is a difficult problem to handle efficiently because there are large amounts of information dispersed throughout the many nodes in a large-scale system. Global coordination suffers from similar difficulties due to the need for a large number of coordination messages. Finally, even though there are general scalability guidelines to follow (see [16]), there is no general solution for building massively scalable systems, so most systems are created in an ad-hoc manner.

A system has to handle these problems in order to achieve massive scalability and KELP's infrastructure does so by modeling its connectivity after a small-world network.

## 2.2 Modeling After Small-World Networks

A distributed system's connectivity is modeled as a graph to define key system properties. KELP's connectivity is modeled after a small-world network because small-world networks have been shown to allow some naturally occurring systems to grow to large sizes [17, 3].

A small-world network is a graph that combines the benefits of a regular graph and a random graph [17]. A regular graph is a graph whose nodes can be connected locally and whose nodes all have the same *degree* (where degree, $d$, is the number of *neighbors* that each node is connected to). Regular graphs can provide the benefit of locality through clustering. A random graph is a graph whose nodes are randomly connected to each other. Random graphs provide the benefit of a short typical separation between nodes and a small $d$ relative to $n$ (the number of nodes in the system). A small-world network provides the benefits of both regular

and random graphs: a high degree of clustering and a short typical separation.

KELP's infrastructure is thus modeled after small-world networks to address the problems outlined in Section 2.1.

Global information is handled by partitioning the information throughout all nodes in the small-world network. Each node has its own part of the global information that can be quickly shared with any other node because of the short typical distance of separation.

Global coordination is handled in two ways. Locally connected nodes coordinate tightly while randomly connected nodes coordinate loosely. The loose semantics are facilitated with every node's small $d$ because each node doesn't have to maintain much information about other nodes in the system.

Finally, KELP addresses the challenge of building ad-hoc scalable systems by allowing massively scalable systems to be built on top of its small-world-based infrastructure. KELP also provides massively scalable data structures to make building massively scalable systems less ad-hoc.

## 2.3 Properties of KELP's Graph

Nodes in KELP are connected via a directed $d$-regular random graph. Directed means that a node's incoming and outgoing edges are clearly defined and $d$-regular means that all nodes have $d$ neighbors. Also, the number of incoming and outgoing edges for each node is fixed.

This random graph connectivity provides the benefits of small-world networks (except for locality) and also provides the benefit of *rapid mixing* [5]. Rapid mixing means that a message sent from one node will have an equal probability of reaching all other nodes in a very small number of steps (relative to $n$). The random graph

provides two operations that use the rapid mixing property: random walks and expanders.

### 2.3.1 Random Walk

The first operation to exhibit the rapid mixing property is the *random walk* operation. A random walk is used to randomly find any node with equal probability.

A random walk starts at any node in the system. At each node in the walk, a random neighbor is chosen as the next node to step to. If an adequate number of steps is taken, the random walk will end up at a random node.

The number of steps ($h$) required to make the chosen node random is relatively small and depends on two factors. The first factor is the number of nodes in the system ($n$). To remain random, $h$ must grow logarithmically with $n$ [8]. The second factor is the system's degree, $d$. A higher $d$ allows for a shorter $h$ and vice versa. An analysis of the relationship between $n$, $k$, and $h$ is provided in Section 6.1.

### 2.3.2 Expander

The second operation to use the rapid mixing property is the *expander* operation. The expander is used to to quickly propagate information from one node to a given percentage of other nodes in the system.

An expander starts at one node and is forwarded to each of the starting node node's neighbors. Nodes receiving the expander may also forward it to all of their neighbors. The number of times an expander is forwarded is called the expander's *depth* and is denoted with $e$.

The expander depth ($e$) required to cover the graph is relatively small compared to $n$ because each level of the expander results in a large factor of new nodes

receiving the expander. At each level, the number of new nodes receiving the expander can be up to $d$ times the number of nodes who have already received the expander. Hence, the expander can spread information at an exponential rate. However, once half the nodes in the system receive the expander, the rate of expansion slows down because the expander will be propagated to nodes that it has already visited. In conclusion, expanders can propagate information to all nodes in the system, but are quickest when only propagating information to less than half the nodes in the system.

## 2.4 Building KELP's Random Graph

Building KELP's random graph is difficult because of two problems. First, random graphs are typically constructed by randomly choosing a node's neighbors from the set of all nodes already in the graph; however, this cannot be done in KELP because there is no global knowledge of all nodes. Second, KELP is a distributed system that must support the dynamic addition and removal of nodes (e.g. node failure and recovery), but dynamic graph membership is not addressed by random graph theory. KELP's random graph building technique addresses these problems and maintains randomness by being an incremental algorithm that chooses a new node's neighbors from the set of all nodes already in the graph.

KELP's random graph building algorithm starts by bootstrapping the system with $d+1$ nodes to form a complete graph of degree $d$. After bootstrapping, new nodes join the graph via any node already in the graph. To select each of its $d$ neighbors, the new node ($o$) performs a random walk to choose a random node ($x$). The chosen node, $x$, randomly picks one of its neighbors ($v$) to evict from its neighbor list. Then $x$ replaces $v$ in its neighbor list with $o$ and $o$ gets $v$ as one of

its neighbors (see Figure 2.1). If the randomly chosen node $x$ is the original node $o$, another random walk is performed to chose another neighbor.

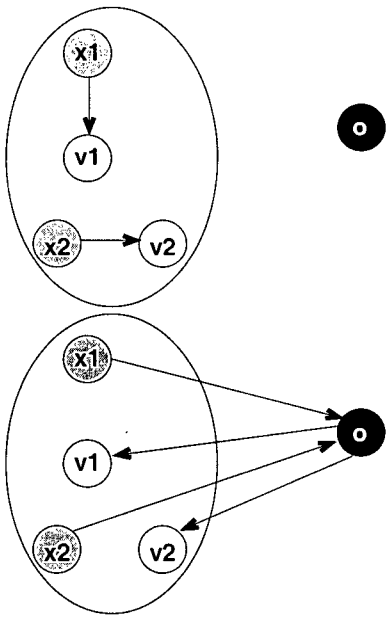


Figure 2.1: Node $o$ joins KELP

A node gracefully leaving KELP must be removed from the graph. A leaving node ($l$) notifies both its incoming and outgoing neighbors. Each of the incoming neighbors lose an outgoing edge to $l$ and each of the outgoing neighbors lose an incoming edge from $l$. To maintain connectivity and to balance the number of incoming edges and the number of outgoing edges, each of $l$'s incoming neighbors is connected to one of $l$'s outgoing neighbors. Nodes not gracefully leaving the system (i.e. due to node failures) are discussed in Section 2.5.

Both node addition and removal maintain fairness by ensuring that every node has the same number of neighbors and outgoing and incoming edges. This balance helps ensure randomness because every node will have an equal probability of being chosen at every step in a random walk. However, this balance is difficult

to maintain in the event of node failures.

## 2.5 Handling Node Failures

Unreachable nodes due to node failures or downed network links cause random walk failures and non-randomness and can also indicate a network partition. A random walk failure occurs when the random walk does not return from a step to an unreachable node. Random walk failures will never finish and will never return to the originator, so there must be a mechanism to detect and handle the failure. Random walk failures are also an indication of non-randomness in the graph because they indicate a node failure. A node failure results in the node's neighbors having fewer references to them (a lower incoming degree), thus the failed node's neighbors will then have less chance of being in a random walk, resulting in non-randomness. Another cause of non-randomness is when nodes reference unreachable nodes. These *phantom references* cause nodes to effectively have lower outgoing degrees, which biases random walks towards reachable nodes. Random walk failures can also indicate a network partition because only random walks that stay entirely within the partition are successful, thus biasing these random walks to return only nodes within the partition.

KELP must be able to detect unreachable nodes and repair the graph in a way that maintains the graph's randomness. Failed random walks and unreachable nodes are detected via a combination of timers, heartbeats, and pings while randomness is ensured by repicking seemingly non-random nodes.

To fix the problem of random walk failures, random walks are time limited and restarted by the originator if they do not return within the time limit. In case the random walk gets delayed (i.e. because of congestion), heartbeat messages

are sent back to the random walk originator to reset the random walk time limit. Specifically, the timestamp of the random walk start time is sent along with the random walk. Nodes receiving the random walk use a global clock to check whether or not a heartbeat message needs to be sent back to the originator.

Restarted random walks may be an indication of bias because some nodes have fewer references to them than others. To counteract this bias, nodes chosen via restarted random walks are flagged to be repicked later when the graph is no longer biased. Each node periodically tries to repick its flagged neighbors and only replaces them and clears the flag when a random walk successfully completes without restarting.

To reduce random walk failures in the first place, references to failed or unreachable nodes need to be detected and repaired. These phantom references are detected by having every node periodically ping all of its outgoing neighbors. Detected phantom references are then replaced with a new node chosen via a random walk.

Random walk failures can also indicate a network partition, but KELP cannot determine whether a partition exists or not because the number of nodes in the system is unknown. However, KELP can use failed random walks to probabilistically indicate a partition. Random walks only return nodes within its own partition and fail when trying to reach nodes in other partitions, so many failed random walks may indicate a partition. (Small partitions will probabilistically have more random walk failures than large partitions because they have more references to unreachable nodes outside the partition.) Once a partition is detected, the network can merge back into one system if one partition knows about any node in the other partition. Nodes in the detected partition repick their flagged neighbors by starting a ran-

dom walk at the node in the other partition. Doing so creates connections between the two partitions. Eventually, all flagged neighbors in the other partition will be repicked as well and the repicked neighbors will be from the set of all nodes in the system.

# Chapter 3

# Determining Global Behavior from KELP's Infrastructure

This chapter describes how KELP's infrastructure facilitates the identification of global behavior and the sharing and discovery of information. First, a generic method of searching is described followed by a description of how specializing this method results in faster searches.

## 3.1 Color

KELP's most generic method for searching is called *color*. Nodes have properties called color and the system provides an operation to find any node with a specified color. For example, if a user wanted to find a game server to play on, the user would search for the game's name and the color search would return a node in the system that is serving that game.

To propagate color information throughout KELP, each node periodically uses an expander to advertise its color to a given percentage of other nodes in the

system. Nodes receiving this expander keep track of the advertising node and its color as hints, so each node has a list of other nodes and their corresponding colors. These hints expire to account for node failures and the periodic expanders ensure that the hints are fairly up-to-date.

A node uses a *color search* to find a node of a particular color. The color search contains a search predicate that is evaluated against every hint in a node's list. If a hint satisfies the search predicate, then the corresponding node is checked to see if the hint is still valid. If the hint is still valid, then the color search is successful; otherwise, the color search continues with a random probe to any number of the node's neighbors.

Two parameters affect the speed and effectiveness of the color search: the depth of the expander used to propagate hints and the number of neighbors that the color search is propagated to at each node.

The percentage of nodes that have a particular hint is determined by the depth of the expander used to advertise the hint. Deeper expanders spread hint knowledge to a greater percentage of nodes, which results in shorter searches. On the other hand, deeper expanders increase search overhead in two ways. First, hints take longer to propagate because of higher resource usage (e.g. network and CPU). Second, more hints are stored by each node, which increases the local search time and storage required at each node.

There is also a tradeoff between the number of messages and search time when deciding on the *width* of the color search (width is the number of neighbors that a color search continues onto at each node). Propagating the search to more neighbors can result in quicker search times, but increases the number of messages because unfinished probes will continue searching even though a matching node has

already been found (probes are independent of one another). Typically, the width of a color search is related to the percent of nodes that have a certain hint; greater percentages allow for narrower searches and vice versa.

Another factor of search width that affects the speed of a color search is whether the search width is fixed or variable. A fixed search width means that a hint is propagated to exponentially more neighbors with every additional expander depth. For example, a search width of 7 expands to 7, 49, 343, and 2401 neighbors for expander depths 1, 2, 3, and 4. To reach different numbers of neighbors in between the exponential leaps, search widths can be variable based on expander depth. As the expander depth increases, the search width can decrease. For example, if the search width in the previous example was decreased to 2 for the third depth, then the number of neighbors reached would be 7, 49, 98, and 196.

Picking a good search width and expander depth is key for fast, effective color searches; however, there is still a probability that a color search will fail to find its target. A color search fails if the color being searched for doesn't exist or if the search's random probe doesn't happen to find a relevant hint. In either case, the search cannot be allowed to continue indefinitely, so a color search's length is limited. Failed searches are retried several times with longer and longer length limits. After several failed retries, the system can state with high probability that the color being searched for doesn't exist.

## 3.2 Heat

To support directed searches, KELP provides an operation similar to color called *heat*. Heat advertises a numeric value as its property, which allows searches to be directed instead of random. Like color, heat propagates information via expanders,

but unlike color, a *heat search* uses a hill-climbing algorithm to get closer to its target on every step.

Every node has two numeric values: a *node heat* that indicates how hot it is and a *path heat* that indicates how hot its paths are. A node's path heat is based on $\alpha$% of its node heat and (100-$\alpha$)% of the maximum of its neighbors' path heats (where $\alpha > 50$ so a node's own node heat has the greatest effect on its path heat). In addition, each node maintains and periodically updates a list of its neighbors' path heats. This list is used as a hint to direct heat searches towards hotter neighbors.

When a node's path heat changes by more than some threshold, it propagates its new path heat to its outgoing neighbors. Nodes receiving the new path heats recalculate their own path heats and continue propagating the heat change information if necessary. At each step away from the original node, the effect of the original path heat decreases because only (100-$\alpha$)% of a neighbor's path heat is used in a node's path heat calculation. This decrease in path heat effect is called *heat decay* and determines how far heat hints are propagated.

A heat search is a directed search that always goes towards hotter nodes until a node satisfying some predicate is found. On every step of the search, the heat search checks if the current node satisfies the predicate. If it does, the search ends and a callback is made to the heat-search originator; otherwise, the search continues to one of the node's hotter neighbors via a weighted-random choice. The random choice is weighted by each of the neighbor's path heats so that similarly heated neighbors both have a chance of being chosen and that hotter neighbors have a higher probability of being chosen. This weighted-random choice also prevents simultaneous searches from all finding the same node, which is useful for load balancing and in cases where a search results in the destination node's heat to change.

The main parameter that affects a heat search's speed and effectiveness is the decay value (100-$\alpha$). The decay value affects the number of nodes that heat dissipates to. For example, a smaller decay value means that more nodes have knowledge about the heat, but at the cost of longer times to propagate and update path heats.

Given that the decay value affects the range of heat dissipation, it is possible for a heat to not reach certain areas of the graph. If these cold areas do not have any heat knowledge at all, then heat searches starting from those areas would not find anything. To prevent these searches from failing, heat searches from cold areas first use a random walk to find a heated area. Once in a heated area, the heat search continues as usual.

# Chapter 4

# Example Data Structures

Using KELP's random graph connectivity as the base layer and color and heat for searching, we have designed some massively scalable data structures that provide loose semantics: a set and a queue.

## 4.1 Set

The set is a data structure for finding nodes that belong to a certain group. Nodes within groups are connected such that all nodes in a group are reachable once any node in the group is found. This set can be used, for example, by applications that require a large-scale, decentralized discovery service. The set could also be used as the basis for a peer-based file-sharing system like Gnutella [1]. The set could also be used to connect and find nodes in the same areas of a distributed virtual environment. [1] Here, groups would contain all nodes that are virtually co-located together. Similarly, the set could be used to group together and find nodes in an IRC-like chat facility [10] or a match-making service (e.g. to find people to play

[1] A distributed virtual environment allows multiple users to interact on the same objects in the same virtual environment over many, interconnected computers.

networked games against). Also, the set could be used as part of a naming service where nodes can belong to many groups and where groups represent attributes of the node such as organization, geography, etc.

The set uses color and adds the concept of groups. The color being advertised is the group identity and color searches continue onto one random neighbor until they find a node in the group being searched for. The set leaves the responsibility of connecting nodes within groups to specific group implementations because different applications require different types of connectivity.

A set's search speed and effectiveness not only depend on the parameters that affect color searches, but also on group sizes. The size of groups affect the percentage of nodes that have hints for that group. Each node in the group propagates hint information via expanders, so bigger groups will propagate more hint information, which makes it quicker to search for the group.

## 4.2 Queue

KELP's queue loosens queue semantics in order to scale to larger sizes. Instead of strictly returning the oldest element in the queue, KELP's dequeue returns an element that is close to the oldest. Also, since there is no global knowledge, the aggregate number of elements in the queue cannot be found. This loose, decentralized queue is ideal for massive scale distributed worker queues that serve job tasks to many clients. For example, this worker queue could be used to distribute the processing tasks of the SETI@home project [7].

The queue partitions its data amongst all nodes in KELP's infrastructure and uses heat to share queue elements among nodes. Each node stores queue elements in its own local *queue part*. Enqueue and dequeue requests can be directed to any

node in the system. Enqueues are always handled locally with enqueued elements inserted into the back of the local node's queue part. Dequeues are handled locally if the node receiving the dequeue request has local elements that can be dequeued. In this case, the dequeue returns the oldest local element. If there are no local elements, a heat search is used to find a node with excess elements (a node with more elements than it needs). The number of excess elements is the number of elements that a node has in excess of what it needs to satisfy its current request rates. Some of these excess elements are then transferred to the local node and the dequeue returns the oldest of these elements. If the heat search doesn't find anything, the dequeue request returns empty.

The queue uses heat to transfer elements from nodes that have excess elements (hot nodes) to nodes that need elements (cold nodes). Hot nodes are nodes that have more elements than they need and are indicated by a growing local queue size. For example, nodes with enqueue rates greater than dequeue rates would be hot nodes. Cold nodes request the number of elements needed to satisfy its needs for the time between heat information updates ($t$) and hot nodes give as many elements that it can spare up to a maximum of the number of elements requested. A dequeue from a cold node that cannot find elements using heat returns nothing to the requester and requires the requester to handle the situation.

The hotness and coldness of nodes caused by an uneven distribution of queue elements among nodes raises issues of staleness and fairness. Also, node failures raise the issue of handling unrecoverable data loss.

Elements in the queue may become stale if a node containing elements does not receive any dequeue requests. In this case, the stale elements would become the oldest in the aggregate queue, but never get dequeued. To ensure liveliness and

progress, stale elements get exponentially hotter with every heat update so heat searches will eventually find stale elements before excess elements.

Since many nodes can be hot simultaneously, there needs to be a mechanism for ensuring fairness when a data request needs to decide between similarly heated nodes. This fairness is provided by heat's weighted-random choice that is used when deciding the next node to continue the search to.

Finally, a node failure results in the loss of all elements in that node's queue part. The queue does not provide a method for recovering this data; instead the responsibility of detecting and handling the data loss is left up to the application using the end-to-end argument [9]. For example, the application would have to detect a lost element and then reinsert it into the queue.

# Chapter 5

# Implementation

To evaluate KELP's ideas for massive scalability, a simulation of KELP was implemented using Java [11]. The simulation builds a network of nodes connected via a random graph and also builds the set and queue data structures on top. Every node in the random graph is represented by a Java object that is made into a distributed object using ObjectSpace Voyager[1].

This chapter describes three aspects of the implementation: the random graph, the set and color, and the queue and heat.

## 5.1 Implementing the Random Graph

The random graph building algorithm is described in detail in Sections 2.4 and 2.5. This section describes three specific implementation details: architecture, bootstrapping, and random walk parameters.

[1] http://www.objectspace.com/voyager/prodVoyager.asp (24 Aug 00)

### 5.1.1 Design Architecture

The simulation provides a base random graph infrastructure that can be extended to provide data structures and services. A *RandomGraph* object is used to build the random graph and to keep track of all nodes in the graph. Nodes are represented with *Node* objects that keep track of their neighbors. Each *Node* can also be coupled with any number of *Overlay* objects to extend its functionality. For example, a *Node* can be coupled with a *SetOverlay* to provide set functionality.

The two random graph operations, random walk and expander, are implemented by mobile agents. These two operations are represented by *RandomWalk* and *Expander* objects that contain all the data and logic needed to perform their work. Like a mobile agent, these objects are shipped in their entirety from node to node to perform their work. These objects can also be specialized (subclassed) to provide specific functionality; for example, there are specialized *RandomWalk* objects for building the random graph and for searching the set.

### 5.1.2 Bootstrapping and Node Addition

For the simulation, a well-known location called the *Registry* is used for bootstrapping the system. Joining nodes use the *Registry* to find an initial node to contact. The *Registry* contains references to the first $d+1$ joining nodes for bootstrapping purposes. After bootstrapping, joining nodes can contact any node in the system; however, the simulation uses the *Registry* both during and after bootstrapping because it provides a single, convenient method for all nodes to join the system.

A node $o$ joining the random graph starts off by registering with the *Registry*. If the system is bootstrapping, the *Registry* will store a reference to the node; otherwise, nothing is done. Next, $o$ will ask the *Registry* for a node at which to start

a random walk. If the system is bootstrapping, no node is returned; otherwise, the *Registry* randomly chooses one of its *d+1* node references to return.

If no node is returned, $o$ knows that it is part of the bootstrapping process, so it will ask the *Registry* for a list of all nodes in the system so far. For every node $x$ in the list, $o$ makes $x$ a neighbor and also contacts $x$ to make $o$ a neighbor. This part of the algorithm creates a complete graph for the first *d+1* nodes joining the graph.

If a node reference is returned from the *Registry* request for a random walk start point, the node knows that it can join the graph normally, so it will asynchronously start a random walk using the random graph building algorithm. The joining node also asynchronously starts *d-1* other random walks in the same way to find all $d$ of its neighbors.

This implementation for node addition is simple but has a serious drawback in that the *Registry* is a centralized object in an otherwise decentralized system. However, the *Registry* can be removed since it is only used for bootstrapping. Nodes themselves can keep track of whether the system has been bootstrapped or not. Joining nodes initially contact any node already in the system. If the initial node is already bootstrapped, then the joining node joins as normal. Otherwise, the joining node participates in the bootstrapping process. As long as joining nodes can find any node already in the system, no centralized *Registry* is needed.

### 5.1.3 Random Walks

Each random walk used to build the graph is represented with a *RGRandomWalk* object that is a specialization (subclass) of a *RandomWalk* mobile agent. The *RandomWalk* object keeps track of the starting node while the *RGRandomWalk* object

keeps track of the number of remaining steps and a *TimeStamp* object that indicates when the walk started.

The *RandomGraph* object starts all random walks for all nodes and stores a timer for each of the walks. Each of the timers is uniquely identified by a *TimeStamp* object. If a timer expires, its corresponding *TimeStamp* object is invalidated and a new *RGRandomWalk* is sent out. The time given to a random walk before it expires is denoted with the constant *TIMER_RANDWALK*.

At each step of the random walk, *RGRandomWalk* checks whether or not it has to send a heartbeat back to the *RandomGraph* to reset its timer. A heartbeat is sent back if the time difference between the current time (given by a global clock) and the *TimeStamp* is greater than a *TIMER_HEARTBEAT* constant. (In our implementation, *TIMER_HEARTBEAT*'s value is three-quarters of *TIMER_RANDWALK*'s value - the heartbeat value must be less than the random walk timer value so that heartbeat messages can reach the original node before the random walk timer expires.) If the random walk has more steps to go, the number of steps remaining is decremented and a random neighbor is chosen as the next node for *RGRandomWalk* to step to.

Several checks must be made once a random walk finishes all its steps. If *RGRandomWalk* ends up back at the starting node, the corresponding timer is stopped and a new *RGRandomWalk* is sent out. (Note, this case is unlikely to occur with a large graph.) If *RGRandomWalk* ends up at a node that already has the walk's source as a neighbor, the number of steps is reset to the original number of steps and the walk is continued. This continued walk has the same length of as all other random walks so that its results will be random. If *RGRandomWalk*'s *TimeStamp* is no longer valid, then the corresponding timer expired and a new *RGRandomWalk*

was sent out, so this *RGRandomWalk* ends without doing anything.

Once a random walk passes all these checks, the walk ends properly and neighbors are swapped per the random graph building algorithm in Section 2.4.

## 5.2 Implementing the Set using Color

Color and set are implemented together. Set functionality is added to each node by coupling a *SetOverlay* object with every *Node* object. The set implementation is straight forward and follows the description given in Sections 3.1 and 4.1. The only thing to note is how the implementation reuses and specializes the *RandomWalk* and *Expander* mobile agents from the random graph layer of the implementation.

Each set search is a specialized *RandomWalk* object called *SetSearchWalk*. The *SetSearchWalk* object stores several things: a search predicate, a listener, a count of the number of steps taken, and a hop limit. The search predicate is used to determine whether or not a node satisfies the search criteria. The listener is a reference and callback to the requesting node that is used when a node satisfies the search predicate. Finally, the count of the number of steps and the hop limit are used to determine when a search has taken too long. On every step, the count is incremented and a random neighbor is chosen as the next node to step to. If the count exceeds the limit, the search returns unsuccessfully back to the requester.

Each hint-propagating expander is a specialized *Expander* object called *NodePropertiesExpander*. The *Expander* object keeps track of the current expander depth, the maximum expander depth, and, if necessary, the parameters for varying the expander width. The *NodePropertiesExpander* object stores the color being propagated. On every level, the expander adds its color to the current *SetOverlay*'s collection of hints, increments its expander depth and then continues propagation

if the current depth is less than the maximum depth. The parameter for variable expander width is an ordered list of widths. The order of the elements corresponds to the expander depth and the elements' values correspond to each depth's width. This width determines the number of randomly-chosen nodes to continue propagating the expander to. For example, a width of 4 at position 3 in the list means that the expander is propagated to only 4 neighbors on the third level of the expander.

## 5.3 Implementing the Queue using Heat

Heat and the queue are described in detail in Sections 3.2 and 4.2. This section describes three specific implementation details: heat and queue overlays, heat propagation, and how dequeue uses heat searches to find and transfer queue elements.

### 5.3.1 Overlays

Heat and the queue were implemented as separate layers on top of the random graph infrastructure. The queue layer lies on top of the heat layer, which lies on top of the random graph layer. Heat functionality is provided with a *HeatOverlay*, which is specialized with a *QueueOverlay* to provide queue functionality.

The *HeatOverlay* keeps track of a node's current and previous node heat and path heat as well as a list of its outgoing neighbors' path heats. A node's current node heat is denoted with *NODE_HEAT* and its current path heat is denoted with *PATH_HEAT*. These values are used in heat propagation and heat searches. The previous node heat and path heat values are stored because heat is only propagated if the change in heat from the last update is greater than some threshold.

The *QueueOverlay* is a specialization of the *HeatOverlay* that has access to all the values described above. In addition, *QueueOverlay* keeps track of the elements in

the local queue part, the number of elements (*QUEUE_SIZE*), the current enqueue and dequeue rates, and the high and low thresholds for the number of elements stored in the local queue part. The dequeue rate is denoted with *DQR*, the enqueue rate is denoted with *EQR*, the high threshold is denoted with *THRESHOLD_HI*, and the low threshold is denoted with *THRESHOLD_LOW*. These values are used to determine whether a node is hot or cold.

### 5.3.2 Heat Propagation

Heat value propagation is handled via a *TimerHeat* object that has a list of all nodes in the system. *TimerHeat* periodically and synchronously updates heats by getting every node to recalculate its node heat and path heat. The frequency of these heat updates is set via a compile-time constant called *TIMER_HEAT* (this constant can be changed for different runs of the simulation). If a node's new path heat changes by more than a threshold from its previous path heat, the path heat is propagated with heat decay to the node's outgoing neighbors. This subsection specifies how *NODE_HEAT* and *PATH_HEAT* are calculated and also describes how path heats are propagated with decay throughout the system.

#### Node Heat

Node heat starts out as zero and is recalculated during a heat update if one of two conditions holds. Either the node has excess elements or the node has old elements which are becoming stale.

A node with excess elements is a hot node if it satisfies two conditions. First, it has more elements than *THRESHOLD_HI*. Second, it has an $EQR >= DQR$.

The high threshold, *THRESHOLD_HI*, is recalculated every heat update

period and determines the number of elements needed to indicate an excess. The threshold is calculated with:

$$THRESHOLD\_HI = (DQR \times TIMER\_HEAT) \times THRESHOLD\_MULTIPLIER$$

This threshold ensures that the node has enough elements for local dequeuing in one heat update period at the last recorded dequeue rate (this number of elements is given by the product in parentheses: $DQR \times TIMER\_HEAT$). *THRESHOLD_MULTIPLIER* is a constant multiplier that is used to pad the threshold in case the *DQR* increases between heat updates. The constant used in this implementation was 1.25.

A node with excess elements only becomes hot if the $EQR >= DQR$ because such rates mean that the number of elements will increase or stay the same. Nodes whose $EQR < DQR$ do not become hot because local dequeuing requires elements in the local queue part for dequeuing.

If both of the above conditions are met, *NODE_HEAT* is calculated with:

$$NODE\_HEAT = HEAT\_EXCESS\_BASE + ((QUEUE\_SIZE - THRESHOLD\_HI) \times HEAT\_EXCESS\_MULTIPLIER)$$

The above equation shows that *NODE_HEAT* is a direct multiple of the number of excess elements ($QUEUE\_SIZE - THRESHOLD\_HI$) that a node has. The base heat value, *HEAT_EXCESS_BASE*, and the constant multiplier, *HEAT_EXCESS_MULTIPLIER*, are both used to make the *NODE_HEAT* value larger, which allows heat to be propagated further. In this implementation, we choose only to use the

multiplier, so *HEAT_EXCESS_BASE* has value 0 and *HEAT_EXCESS_MULTIPLIER* has value 10.

A node with stale elements becomes hot if it satisfies two conditions: the *QUEUE_SIZE* must be greater than zero and the *DQR* must be zero. These two conditions mean that there are elements in the local queue part that will not be locally dequeued. These elements will eventually become stale and will need to be removed.

To remove these elements, the node becomes hotter and hotter with time. If the age of the oldest element (denoted with *AGE_OF_OLDEST*) is greater than a threshold (set to 45 time periods for this implementation), then *NODE_HEAT* is calculated with:

$$NODE\_HEAT = HEAT\_STALE\_BASE + AGE\_OF\_OLDEST^{HEAT\_STALE\_EXPONENT}$$

*NODE_HEAT* grows superlinearly as elements become staler because *AGE_OF_OLDEST* grows linearly as elements get older and *HEAT_STALE_EXPONENT* is a constant (set to 1.4 for this implementation). *HEAT_STALE_BASE* is meant to provide larger heat values for stale nodes, but is set to zero in this implementation.

### Path Heat and Propagation

Path heat is recalculated every heat update and then is propagated with heat decay if the change in path heat is greater than *THRESHOLD_PATH_HEAT*.

To calculate path heat, a node first gets all its outgoing neighbors' path heats to find their maximum. Then path heat is calculated with the following formula:

$$PATH\_HEAT = (\alpha \times NODE\_HEAT) +$$

$$((1 - \alpha) \times max(\ neighbor's\ PATH\_HEAT's))$$

Here, $\alpha$ is the heat decay given as a decimal number. Its value is 0.7 so that a node's own *NODE_HEAT* has the most effect on its *PATH_HEAT* and only a small percent of a neighbor's *PATH_HEAT* is used.

Path heat is propagated to all incoming neighbors if the difference between the new and previous *PATH_HEAT*s is greater than *THRESHOLD_PATH_HEAT*. The threshold determines how far heat is propagated to and is set to 1 in order for heat changes to be propagated as far as possible.

### 5.3.3 Dequeue and Heat Search

Dequeuers start heat searches when they need more data to satisfy incoming dequeue requests. In other words, the dequeuer is cold and it needs to find a hot node to get queue elements from. This subsection specifies when heat searches are started, how the search is conducted, and the amount of data transferred from a hot node to a cold node.

#### Starting a Heat Search

Heat searches are started in one of two ways: when the local queue part becomes empty or when the number of elements falls below *THRESHOLD_LOW*.

When there are no local elements to return, the dequeuer starts a synchronous heat search to find elements from a hot node. If a hot node exists, some elements are transferred from the hot node to the cold node and then the oldest of these elements is returned. If no hot nodes are found, the dequeue request returns empty.

A synchronous heat search delays the time for a dequeue request to return a value. Therefore, an asynchronous heat search is used to find and transfer elements

to the cold node before it runs out of elements. The asynchronous nature of the heat search allows dequeues to continue unhindered while more elements are being found. An asynchronous heat search is started when the number of queue elements falls below *THRESHOLD_LOW* and the $DQR > EQR$.

The low threshold, *THRESHOLD_LOW*, is recalculated every heat update period (but only when $DQR > EQR$ to avoid negative threshold values). This threshold is calculated with:

$$THRESHOLD\_LOW = (DQR - EQR) \times TIMER\_HEAT$$

This value predicts the net decrease in the number of elements for the next heat update interval, which is useful in determining when a local queue part will run out of elements.

A node also needs a $DQR > EQR$ before starting an asynchronous heat search because the node would not need more elements otherwise. A $DQR <= EQR$ means that a local queue size stays the same or increases whereas a $DQR > EQR$ means that the local queue size is decreasing. A decreasing local queue size means that the local queue part will eventually run out of elements.

An asynchronous heat search is started when both the above conditions are met. The $DQR > EQR$ coupled with a local queue size smaller than *THRESHOLD_LOW* indicate that a node will run out of elements soon.

### Heat Search

A heat search is a hill-climbing algorithm that starts from a cold node and steps to hotter nodes until it finds a node hot enough to get queue elements from.

A heat search starts off by checking its list of outgoing neighbors' path heats. If all of the outgoing neighbors have zero path heat, then a random walk is used

to try to find a heated area (an area with non-zero heat values). If the random walk doesn't find a heated area within $h$ steps, the heat search returns empty. The number of steps, $h$, is the length of the random walk used to build the graph because it is the number of steps required to make a walk random. If the random walk finds a heated area, the heat search continues normally.

Each node visited by a heat search is first tested to see if it is hot enough to transfer elements from. If the node isn't hot enough, the heat search continues to a hotter node.

A node must satisfy two conditions in order be to hot enough for an element transfer. Both conditions are related the *NODE_HEAT* calculation. The first condition has two parts: the local queue size must be larger than *THRESHOLD_HI* and the *EQR* must be greater than or equal to the *DQR*. The second condition also has two parts: the local queue size must be greater than zero and the oldest element must be older than *THRESHOLD_STALE_DATA* (where *THRESHOLD_STALE_DATA* is set to 1.5 times *TIMER_HEAT*).

If either of these conditions is true, elements are transferred from the hot node to the requesting cold node. The number of elements transferred is described in the next section.

If neither condition is true, the heat search must continue to another node. The next node chosen comes from the set of outgoing neighbors that have a higher *PATH_HEAT* than the current node. Among these nodes, the next node is chosen via a weighted-random choice. Each node is weighted with its path heat as a percentage of the sum of all nodes' path heats.

### Transferring Elements from Hot to Cold

Once a heat search finds a node that is hot enough, it must decide on the number of elements to transfer back to the cold node. Each hot node has a maximum number of elements that it can give away and each heat search includes the maximum number of elements that the cold node wants. The number of elements transferred is the minimum of these two amounts.

The maximum number of elements that a hot node is willing to give away depends on whether the hot node has excess elements or stale elements. In the excess element case, the maximum number is the number of excess elements, which is given by the formula:

$$QUEUE\_SIZE - THRESHOLD\_HI$$

In the stale element case, the maximum number is all the stale elements plus half the remaining elements since those elements probably won't be dequeued anytime soon. The number of elements to transfer in this stale case is given by the formula:

$$NUM\_STALE + \frac{(QUEUE\_SIZE - NUM\_STALE)}{2}$$

where $NUM\_STALE$ are the number of stale elements (the number of elements older than $THRESHOLD\_STALE\_DATA$).

The maximum number of elements requested by a cold node is the predicted number of elements that the cold node needs to satisfy dequeue requests for the next heat update period. This number is $THRESHOLD\_LOW$.

Once the minimum of these two maximums is calculated, the hot node takes out this number of its oldest queue elements and gives them to the cold node.

# Chapter 6

# Evaluation

KELP's evaluation did not strictly measure performance because the evaluation focused on validating KELP's global behavior and because workload patterns for the data structures are unknown.

This chapter describes the three areas that KELP's evaluation focused on: the randomness of graphs built, the expected search length for the set, and the expected order of dequeued elements from the queue.

## 6.1 Random Graph Evaluation

The main purpose of the random graph evaluation is to ensure that KELP's incremental graph building algorithm is indeed random. In particular, the constructed graph must allow random walks starting at any node to end up at a random node.

To evaluate the graph, the simulation outputs a list of all nodes and their neighbors in the form of an *adjacency matrix*. The adjacency matrix is a sparse $n \, x \, n$ matrix where each row represents a node. Each row has only $d$ non-zero values. A 1 represents a neighbor with the column number as the neighbor's node number and

a 0 represents a non-neighbor.

These adjacency matrices are evaluated in two ways to show the randomness of the constructed graphs: an experimental approach where the results of many random walks were counted and a mathematical approach that shows the probability of random walks ending at each node.

### 6.1.1 Experimental Analysis

The experimental analysis uses the produced adjacency matrices to run and record the results of many random walks starting from node 1. The number of random walks started was such that the expected number of times that random walks ended at each node is 2000. This test was done on systems with different $n$, $d$, and $h$. Here, $h$ denotes both the length of random walks used to build the graph and the length of random walks used to test the graph.

The results are shown in Table 6.1. The table shows the resultant standard deviation to mean percentage (*stdev/mean*) for every $n$, $d$, and $h$ tested. Smaller *stdev/mean* values are more random because the results of the random walks are closer to a perfectly random, uniform distribution. The table shows that a relatively small $d$ and $h$ are sufficient for a high degree of randomness; for example, a $d$ of 9 and a $h$ of 8 are sufficient to have a *stdev/mean* of 5.41that, for each $n$ and $d$ combination, random walks get more random as $h$ increases (because the *stdev/mean* decreases).

Longer random walks (larger $h$'s) provide more random results; however, there is a point where more steps does not make the results much more random. We have chosen this point to be the number of steps it takes for the *stdev/mean* percentage to drop to approximately 5%. This threshold was chosen because the percentage is sufficiently small for a high probability of randomness and because

| n | d | stdev/mean (%) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | h=4 | h=5 | h=6 | h=7 | h=8 | h=9 | h=10 | h=11 |
| 100 | 7 | | 6.13 | **3.36** | 3.25 | 2.16 | | | |
| | 9 | 10.62 | **4.15** | 2.57 | 1.91 | | | | |
| 1000 | 7 | | | 11.16 | **5.13** | 4.34 | 2.22 | | |
| | 9 | | 12.89 | **4.90** | 2.64 | 2.25 | | | |
| 10000 | 7 | | | | 11.33 | **4.67** | 2.72 | 2.39 | |
| | 9 | | | 14.08 | **5.10** | 2.68 | 2.28 | | |
| 100000 | 7 | | | | | 13.44 | **5.44** | 2.93 | 2.34 |
| | 9 | | | | 14.63 | **5.41** | 2.75 | 2.31 | |

Table 6.1: Experimental analysis of random graph

the percentage does not decrease much more with more steps. Using a system with 10,000 nodes ($n$=10,000) and degree 7 ($d$=7) as an example, Figure 6.1 shows that *stdev/mean* decreases as $h$ increases and then levels off once the threshold is reached (the threshold is shown as a shaded triangle).

In Table 6.1, the threshold *stdev/mean* percentage for each $n$ and $d$ pair is highlighted in bold. These thresholds are graphed out in Figure 6.2. The solid parts of the lines come from the experimental results while the dotted parts of the lines are extrapolations of the data for larger $n$'s. The figure shows that threshold $h$ increases logarithmically as $n$ increases ($n$ is shown logarithmically on the y-axis). The figure also shows that $h$ decreases as $d$ increases (the higher degree line ($d$=9) is left of the lower degree line ($d$=7)). These results show that KELP successfully creates random graphs with the desired random walk properties.

### 6.1.2 Mathematical Analysis

The mathematical analysis uses adjacency matrices to count the number of times a random walk can end up at each node and presents those numbers as probabilities. This analysis is an iterative process that begins with node 1 as the current location

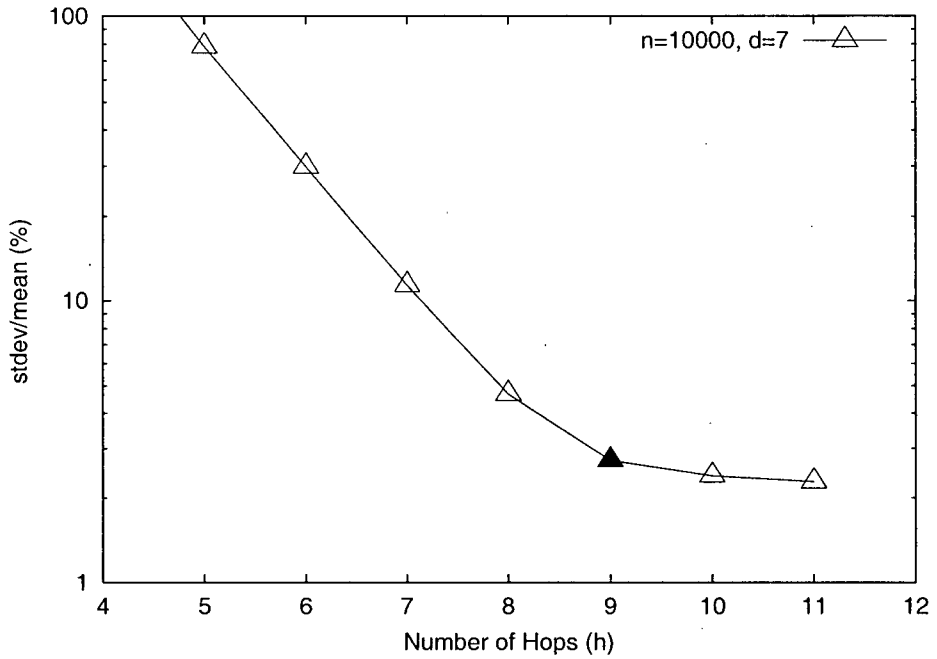


Figure 6.1: Increasing $h$ for randomness

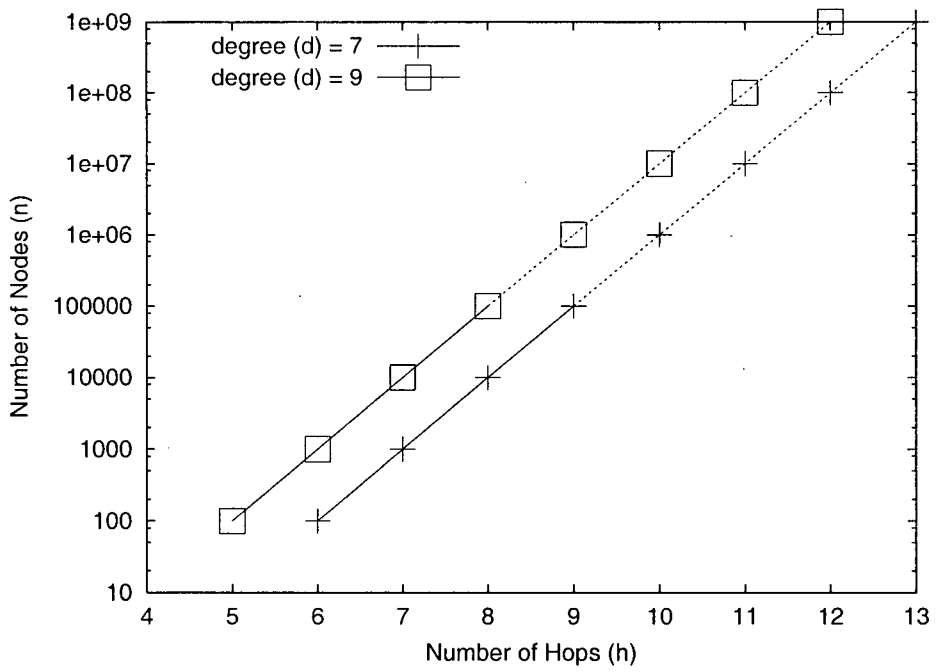


Figure 6.2: Threshold $h$'s for randomness

```
1   for i = 1:num_hops
2     curr_loc = M*curr_loc;
3     walk_count = walk_count*degree;
4     W = [W curr_loc/walk_count];
5   end
```

Figure 6.3: Mathematical Analysis of Random Graph

of all possible random walks. Each iteration of the process represents a step in the random walk where all nodes reachable from the current locations become the new current locations. The number of times a node is reachable is accounted for and the probabilities that each node will be reached is calculated by dividing each node's count by the total number of possible endpoints.

This mathematical analysis is done with a matlab program whose main iterative loop is shown in Figure 6.3. The variable *num_hops* is the number of random walk steps to test. Vector *curr_loc* begins as [1 0 ... 0] to represent starting at node 1. Matrix $M$ is the transpose of the adjacency matrix. Line 2 is how *curr_loc* gets updated with the locations of all possible random walks at a particular step. Variable *walk_count* stores the total number of possible random walks up to the current step, which is why it grows by *degree* power on each iteration. Finally, matrix $W$ is augmented with a vector containing the probabilities that each node is a random walk endpoint.

Formally, this analysis is known as a *Markov chain.* In [4], Aldous states that the analysis will clearly show that $W$ eventually converges to an uniform distribution. Mathematically, this means that the average variance of $W[i] - u$ will decrease as $i$ increases. Once uniform distribution is reached, the average variance stays approximately the same even though $i$ increases. Here, $i$ is the column number, which represents the number of steps taken so far, and $u$ is the uniform location

vector $[1...1]/n$, which indicates that every node has the same probability of being a random walk endpoint.

Figure 6.4 shows an example of the Markov chain analysis converging to the uniform distribution for system with $n$=10000, $d$=9, and $h$=7. The x-axis represents the number of steps and the y-axis is logarithmically scaled to represent the average variance. The decreasing average variance with the increasing number of steps indicates that random walks get more random with more steps. The flattening out of the average variance indicates that more steps after this point do not make the random walk any more random.

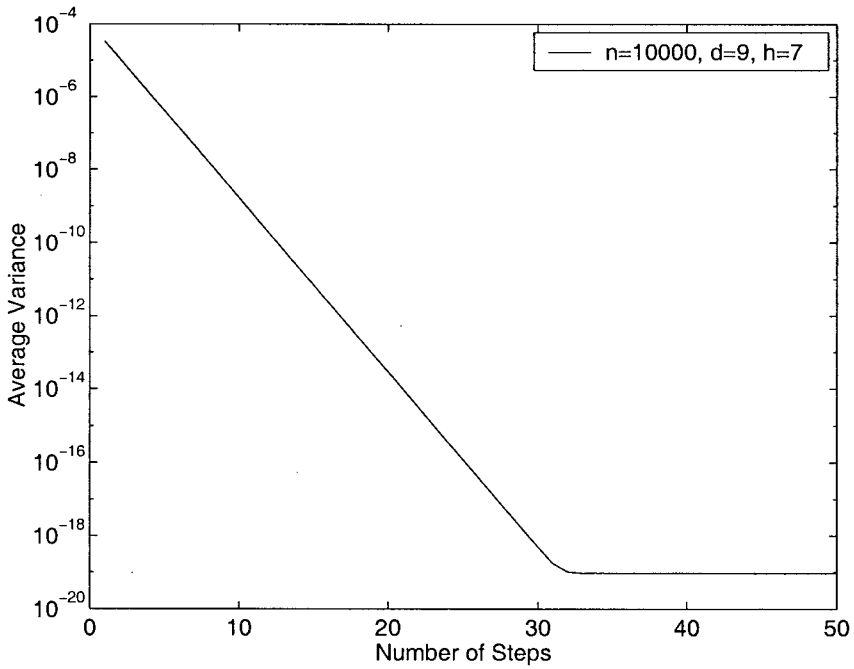


Figure 6.4: Example of Markov chain analysis

This mathematical analysis shows that graphs built by KELP exhibit desired random graph properties. The example in Figure 6.4 shows that more than 30 steps are needed to converge to an uniform distribution, but the experimental analysis

shows that only 8 steps are needed for fairly good randomness with this system configuration. Additional steps after the experimental numbers do not make the results much more random because the average variance is already very small ($1.50 \times 10^{-8}$ for 8 steps). Hence, the threshold number of steps given by the experimental analysis is sufficient for KELP's randomness requirements.

### 6.1.3 Random Graph Conclusions

Our simulation shows that KELP's random graph building algorithm allows random walks to find random nodes quickly for system sizes up to 100,000 nodes. These results predict that the random graph will scale to much larger sizes because the only component of the random graph that depends on $n$ is the length of the random walk. The length of the random walk, $h$, grows logarithmically with $n$, but the relative cost of $h$ actually decreases as $n$ increases (mathematically speaking, $\log n/n \rightarrow 0$). Furthermore, $h$ can be made smaller if the system has a larger $d$.

## 6.2 Set Evaluation

The purpose of the set evaluation is to compare the length of simulated set searches to their expected lengths. To evaluate this, the simulation runs tests for different samples of 1000 nodes that vary the hint-propagation expander depth and the number of evenly-sized groups in each set. Each test sends out 10,000 searches with each group being searched for the same number of times. Each node is also the starting point for the same number of searches. The lengths of all the set searches were recorded and the average was compared to the expected set search lengths.

The expected search set search length is given by the following equation and probabilities:

$$
\begin{aligned}
E(Set\ Search\ Length) &= \frac{1}{Pr(x)} \\
Pr(x) &= 1 - Pr(\overline{x}) \\
&= 1 - (1 - gsp)^{E(Number\ of\ Hints\ in\ List)}
\end{aligned}
$$

where

$$
\begin{aligned}
x &= node\ contains\ hint \\
E(Number\ of\ Hints\ in\ List) &= \%\ Coverage \times n
\end{aligned}
$$

Table 6.2 shows the expected set search lengths for the different search parameters: *group size percentage* (*gsp*) and *% coverage* along with its corresponding $e$ and $p$. The symbol $e$ denotes the expander depth used and the symbol $p$ denotes the partial expansion, which is the number of neighbors to propagate to after the expander has reached $e$ levels deep.

| gsp | % Coverage (e,p) | | | | | | |
|---|---|---|---|---|---|---|---|
| | 0.1% (0,0) | 0.8% (1,0) | 5.7% (2,0) | 10.6% (2,1) | 20.4% (2,3) | 30.2% (2,5) | 40.0% (3,0) |
| 20.0% | 5 | 1.20 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| 10.0% | 10 | 1.76 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| 2.0% | 50 | 6.70 | 1.46 | 1.13 | 1.02 | 1.00 | 1.00 |
| 1.0% | 100 | 12.94 | 2.29 | 1.53 | 1.15 | 1.05 | 1.02 |
| 0.2% | 500 | 62.94 | 9.27 | 5.23 | 2.98 | 2.20 | 1.81 |
| 0.1% | 1000 | 125.44 | 18.04 | 9.94 | 5.42 | 3.83 | 3.03 |

Table 6.2: Expected set search lengths

Group size percentage is the percent of nodes in the system that each evenly-sized group takes up. A gsp of 20% means that a group contains 20% of all nodes

in the system, so for a system with 1000 nodes, a gsp of 20% means that a group consists of 200 nodes that each have the same color.

Percent coverage (*% coverage*) is the percent of nodes in the system that have a hint about a particular color. Percent coverage depends on the hint-propagating expander depth ($e$) and the partial expansion ($p$). The formula for the calculation is:

$$\%\,Coverage \;=\; \frac{(\sum_{i=0}^{e} d^i) + (d^e \times p)}{n}$$

Percent coverage denotes the theoretical maximums for hint propagation because some nodes will receive the same hints more than once.

As expected, Table 6.2 shows that higher % coverage and higher group size percentage cause shorter search lengths. For example, if groups consisted of only one node (gsp 0.1%) and no hints were propagated (% coverage 0.1%), then the search is expected to query every node in the system as shown with the spike in Figure 6.5 (expected length 1000). This example seems bad, but is a worst-case scenario. With just a few levels of hint propagation and slightly larger groups, the expected search length quickly becomes relatively short. This is shown in Figure 6.6 where the expected set search length decreases quickly with a logarithmic scale on the y-axis.

Results from our experiments of running 10,000 searches for every % coverage and gsp pair had at most a 14.84% relative error when compared to the calculated expected values; however, the average relative error was 4.57%. These results have a high relative error because the percentages that we deal with are so small, which results in a high variance. Variance is calculated with:

$$Variance \;=\; \frac{1 - Pr(x)}{Pr(x)^2}$$

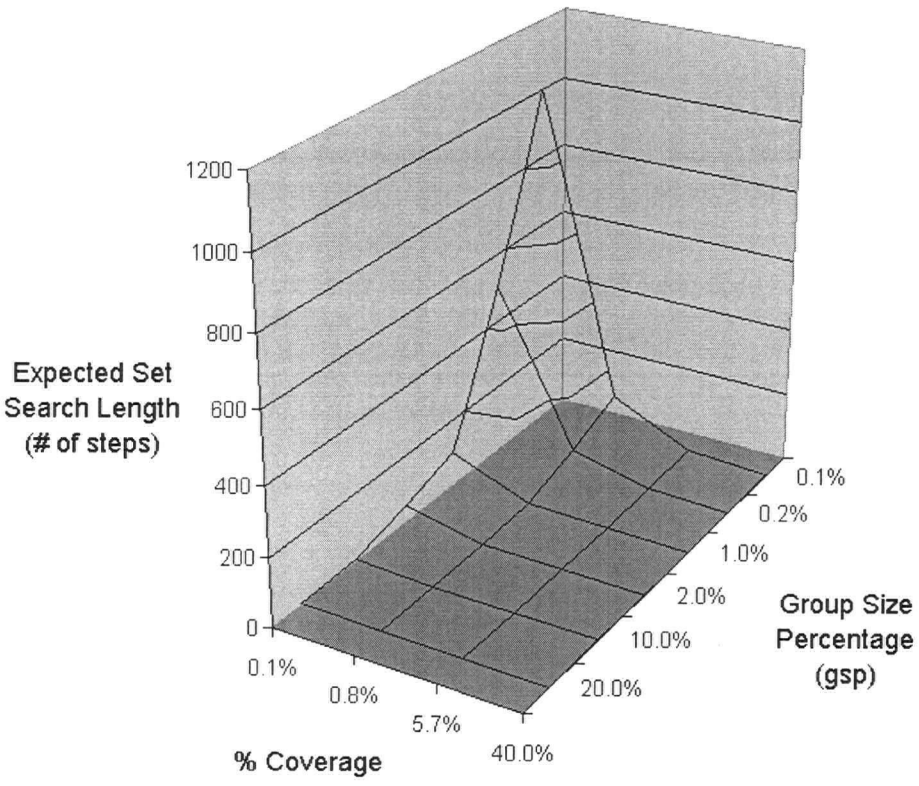


Figure 6.5: Expected set search lengths (3D)

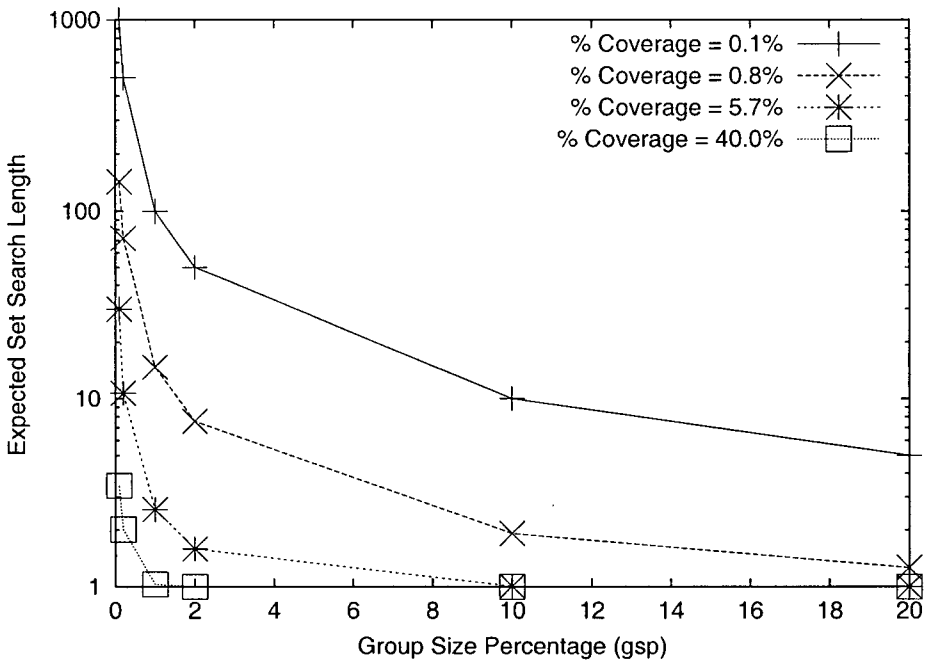


Figure 6.6: Expected set search lengths (2D)

For example, a gsp of 0.02% and a % coverage of 0.8% results in a variance of approximately 300 (which is 30% for $n$=1000).

These results show that the set works as desired and that the search length will be relatively small if % coverage and gsp are high enough. If these two percentages remain constant as $n$ increase, then the set will scale perfectly. Moreover, the global behavior of the set would be well known.

## 6.3 Queue Evaluation

The main purpose of the queue evaluation is to show that dequeues return elements that are close to the oldest in the overall queue. The queue was tested by having many different enqueuers and dequeuers transfer data at constant rates from random nodes. Dequeue output was then analyzed to evaluate the relative order of dequeued

elements.

The main property measured by the queue evaluation is called *rank*. Rank is a percentage that determines the order-wise oldest element in the queue (order-wise based on the FIFO order in which elements are enqueued). A rank of 100% means that a dequeue returned the oldest element in the queue while 0% means that a dequeue returned the newest element. Rank does not measure time, just the order in which elements are taken out of the queue.

Table 6.3 shows the results of the main queue test, which has 10 randomly-chosen enqueuers and 10 randomly-chosen dequeuers (enqueuers and dequeuers can be the same node). The test varies the number of nodes and the time between heat updates, $t$, which is measured in terms of the number of *periods* in between updates. In each period, each enqueuer and each dequeuer performs one operation each. Each row of the table shows the results of one test run for 12000 periods. The third column of the table contains the average rank of all dequeued elements with the standard deviation ($s$) shown in parentheses. The rank percentile columns of 80+ and 90+ show the percentage of dequeued elements that are within the 20% or 10% oldest ranked respectively.

| | | rank | rank percentile | |
|---|---|---|---|---|
| n | t | avg($s$) | (80+) | (90+) |
| 100 | 300 | 80%(19) | 62.67% | 51.90% |
| | 600 | 82%(17) | 68.89% | 56.98% |
| | 1200 | 83%(17) | 52.46% | 37.84% |
| 1000 | 300 | 84%(16) | 71.45% | 58.54% |
| | 600 | 72%(23) | 70.36% | 57.36% |
| | 1200 | 79%(19) | 61.27% | 52.81% |

$s$ - standard deviation

Table 6.3: Queue evaluation

The queue evaluation shows several properties for the given workload pattern. On average, dequeues return the 20% oldest ranked elements in the queue more than 60% of the time. Also, more than 50% of dequeues return the top 10% oldest ranked elements. Increasing $n$ in this test did not affect rank because the absolute number of enqueuers and dequeuers did not change. Changing $t$ also did not affect rank because the amount of data transferred between nodes is dependent on $t$: a higher $t$ means more data is transferred.

The results shown in Table 6.3 are workload dependent for 10 hot nodes and 10 cold nodes all transferring data at a constant rate. All other nodes in the system are assumed to be *locally balanced*, which means that they are neither hot nor cold. Locally balanced nodes do not interact with other nodes, so adding more locally balanced nodes shouldn't affect the queue's performance.

This evaluation shows that the queue performs fairly close to FIFO for any system size given this constant workload pattern. The evaluation also shows that the queue's transfer of excess data via heat works well to balance hot and cold nodes. The tested workload pattern may be unrealistic, but the queue cannot be tested for performance until realistic workload patterns are known. Once realistic workload patterns are known, the queue and heat parameters can be tuned to maximize the queue's performance.

# Chapter 7

# Related Work

This section describes and compares three other projects and ideas that are related to KELP. First, a system that is very similar to KELP's set called Gnutella is discussed. Second, a project with the same goal of massive scalability called Globe is discussed. Finally, other uses of small-world networks related to the Internet are discussed.

## 7.1 Gnutella

Gnutella [1] is a decentralized, peer-to-peer file-sharing system. Nodes join Gnutella by first connecting to any node in the system. Then the joining nodes try to discover other neighbors by periodically sending out pings. Each Gnutella node can then share some of its own files and also search for other nodes' files by sending out expanders to query other nodes. Each expander is limited with a TTL (time-to-live).

Gnutella is very similar to KELP's set except that Gnutella does not scale well. Both are decentralized, both are peer-to-peer, and both use expanders to

propagate and discover information. Both systems also rely on probability when searching for information. However, KELP imposes a structure on the system's connectivity graph that provides characteristics of the graph, whereas Gnutella's graph structure is completely arbitrary so no characteristics can be found from it.

The main difference between Gnutella and KELP is that Gnutella does not scale massively. Scalability in Gnutella is limited to about 10,000 nodes because Gnutella tries to maintain a global property about the system - Gnutella periodically tries to discover the number of nodes in the system via pings. These pings take up approximately 50% of network traffic in the system, which limits Gnutella's scalability.

## 7.2 Globe

The Globe project (GLobal Object Based Environment) [13, 12] from Vrije University is a wide-area distributed system that uses an object-based framework and distributed shared objects for developing massively scalable distributed applications. Globe's distributed shared object framework is meant to provide programmers with a standard mechanism for scaling and replication by providing a standard framework that all implementations conform to instead of having many different ad-hoc solutions such as different server and proxy caches. A scalable WWW service called GlobeDocs was prototyped[1].

Globe shares two main goals with KELP: achieving massive scalability and removing ad-hoc approaches to scalable solutions. First, both projects are meant to massively scale worldwide via the Internet. Second, both projects are meant to remove the ad-hocness in building scalable systems. Globe provides a general pro-

[1]See http://www.cs.vu.nl/ steen/globe/ (1 Sep 00)

grammatic framework that separates policy from mechanism while KELP provides an extensible, scalable infrastructure.

Other than these goals, Globe and KELP differ significantly. Globe is meant for many different services within the system (such as web servers) whereas KELP is meant to support a global service that is aggregately provided by all nodes in the system. Also, Globe is not meant to support a decentralized, peer-to-peer service and Globe does not deal with global information.

## 7.3 Small-World Networks

Small-world networks have been related to the Internet in a couple ways. First, WWW hyperlinks connecting related sites have been shown to be a small-world network [3]. Second, it has been suggested that the Internet can be connected at the router level via a small-world network [2].

Currently, the Internet has small-world networks connecting computers at the web site level, but not at the router level. In [2], Summerfield suggests that the Internet could be connected via a small-world network by clustering local Internet service providers together and then randomly linking each of these clusters with a few others. KELP also needs to be expanded to use small-world networks in this manner. This future work is discussed in the next chapter.

# Chapter 8

# Conclusions and Future Work

KELP's key to massive scalability is to describe global behavior in a loosely coupled, decentralized system. Our simulations have shown that KELP's random graph provides a scalable method of connecting nodes in the system such that any node can reach any other node in a relatively few hops. Coupling this randomness with the locality provided by clusters gives us a small-world network that massive scale systems can be built on top of. KELP also provides set and queue data structures on top of the small-world network to make the process of building scalable applications less ad-hoc. These data structures facilitate large scale systems by providing loose semantics that require less coordination.

Overall, KELP has been shown to massively scale systems with its small-world network connectivity. As the Internet becomes more ubiquitous, one can foresee the need for large scale distributed systems that can take advantage of KELP's massive-scale, decentralized system.

Thus far, the KELP project and simulation have verified that a small-world network facilitates scalability by providing the system with a means of determining some global behavior of the system. The next step in this project is to expand on

the infrastructure and data structures of KELP and then implement a practical, highly-scalable system that uses KELP.

Our analysis has shown that random graphs provide the properties that systems can use to massively scale. Now, locality needs to be integrated into the system to provide the benefits of small-world networks.

More data structures need to be designed. For example, a spanning tree for broadcasts could be useful in reducing the overhead of expanders.

Also, the data structures need to be tested against a real workload. Given a particular application, the data structures and their parameters can be modified to achieve optimal performance. For example, it may be better to transfer every second element from a hot node to a cold node instead of just the oldest few.

Finally, KELP's capabilities need to be demonstrated with a practical application. One possible application is to design a massively scalable peer-based file-sharing system for the Internet. This system would be like Gnutella, which allows peer-to-peer computers to share files, except KELP's peer-based file-sharing system would be able to scale to millions of computers while Gnutella only scales to 10,000 nodes.

Another application that would demonstrate KELP's capabilities is a name service that facilitates millions of mobile devices with dynamic network addresses. Such a system would be useful because mobile, network-capable devices are becoming more popular and DNS cannot handle such frequent name changes.

# Bibliography


[1] Gnutella. http://gnutella.wego.com (25 Jul 00).

[2] How telecommunications is making it a smaller world after all. Presentation found at http://www.ee.mu.oz.au/staff/summer/smallworld/tsld001.htm (25 Jul 00).

[3] Lada Adamic. The small world web. ECDL'99 (European Conference on Research and Advanced Technology for Digital Libraries), 1999. http://www.parc.xerox.com/istl/groups/iea/www/smallworldpaper.html (25 Jul 00).

[4] David Aldous. Random walks on finite groups and rapidly mixing markov chains. In *Seminaire de Probabilités XVII.* Springer-Verlag.

[5] Andrei Broder and Eli Shamir. On the second eigenvalue of random regular graphs (preliminary version). In *28th Annual Symposium on Foundations of Computer Science*, pages 286–294, Los Angeles, California, 12–14 October 1987. IEEE.

[6] Tim Berners-Lee et al. The world-wide web. *CACM*, 37(8):76–82, 1994.

[7] W. T. Sullivan III et al. A new major seti project based on project serendip data and 100,000 personal computers. In *Astronomical and Biochemical Origins and the Search for Life in the Universe: proceedings of the 5th International conference on bioastronomy, IAU colloquium no. 161, Capri, July 1-5, 1996*, 1997.

[8] Martin Hildebrand. Random walks on random simple graphs. *Random Structures and Algorithms*, 8(4):301–318, July 1996.

[9] D.P. Reed J.H. Saltzer and D.D. Clark. End-to-end arguments in system design. *ACM Transaction on Computer Systems*, 2(4):277–288, November 1984.

[10] C. Kalt. Rfc 2810: Internet relay chat: Architecture, April 2000.

[11] James Gosling Ken Arnold and David Holmes. *The Java Programming Language*. Addison Wesley, 3rd edition, 2000.

[12] I. Kuz M. van Steen, A.S. Tanenbaum and H.J. Sips. A scalable middleware solution for advanced wide-area web services. *Distributed Systems Engineering*, 6(1):34–42, March 1999.

[13] Philip Homburg Maarten van Steen and Andrew S. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, pages 70–78, January-March 1999.

[14] P.V. Mockapetris. Rfc 1034: Domain names - implementation and specification, Nov 1987.

[15] H. Levy N. Kronenberg and W. Strecker. Vaxclusters: A closely coupled distributed system. *ACM Transactions on Computer Systems*, 4(2):130–146, 1986.

[16] B. Clifford Neuman. *Scale in Distributed Systems*. IEEE Computer Society Press, 1994.

[17] Duncan J. Watts and Steve H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, June 1998.