

It Probably Works

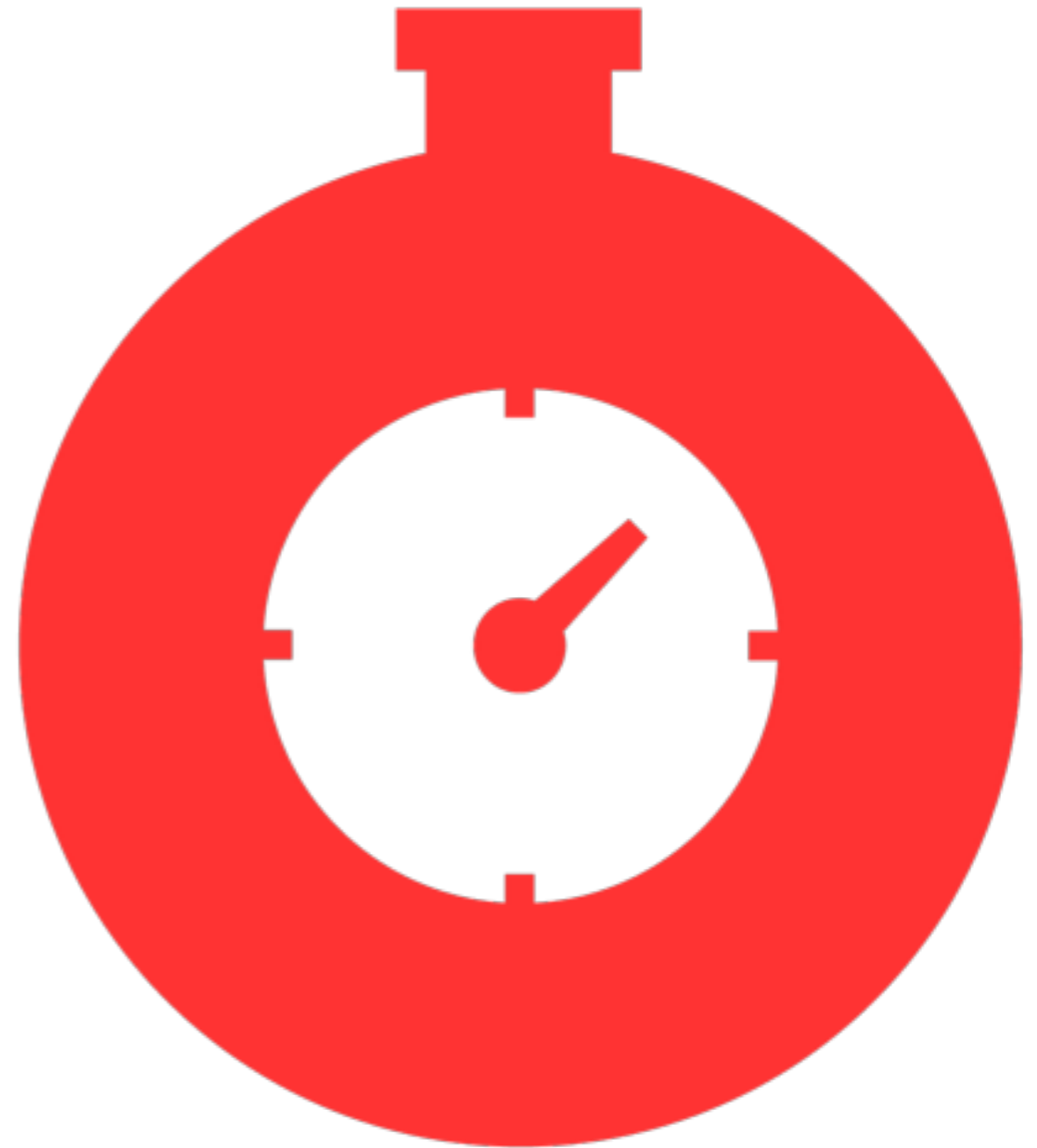
# Tyler McMullen

CTO of Fastly

@tbmcmullen

# Fastly

We're an awesome CDN.



What is a probabilistic  
algorithm?

Why bother?

“In testing primality of very large numbers chosen at random, the chance of stumbling upon a value that fools the Fermat test is less than the chance that cosmic radiation will cause the computer to make an error in carrying out a 'correct' algorithm. Considering an algorithm to be inadequate for the first reason but not for the second illustrates the difference between mathematics and engineering.”

–Hal Abelson and Gerald J. Sussman, SICP

Everything is  
probabilistic

Probabilistic algorithms  
are not “guessing”



Provably bounded  
error rates

Don't use them if  
someone's life depends  
on it.

~~Don't use them if  
someone's life depends  
on it.~~

When should I use  
these?

Ok, now what?

# The Count-distinct Problem

# The Problem

How many unique words are in a large corpus of text?

# The Problem

How many different users visited a popular website in a day?



# The Problem

How many unique IPs have connected to a server over the last hour?

# The Problem

How many unique URLs have been requested through an HTTP proxy?

# The Problem

How many unique URLs have been requested through an entire **network** of HTTP proxies?

```
166.208.249.236 - - [25/Feb/2015:07:20:13 +0000] "GET /product/982" 200 20246
103.138.203.165 - - [25/Feb/2015:07:20:13 +0000] "GET /article/490" 200 29870
191.141.247.227 - - [25/Feb/2015:07:20:13 +0000] "HEAD /page/1" 200 20409
150.255.232.154 - - [25/Feb/2015:07:20:13 +0000] "GET /page/1" 200 42999
191.141.247.227 - - [25/Feb/2015:07:20:13 +0000] "GET /article/490" 200 25080
150.255.232.154 - - [25/Feb/2015:07:20:13 +0000] "GET /listing/567" 200 33617
103.138.203.165 - - [25/Feb/2015:07:20:13 +0000] "HEAD /listing/567" 200 29618
191.141.247.227 - - [25/Feb/2015:07:20:13 +0000] "HEAD /page/1" 200 30265
166.208.249.236 - - [25/Feb/2015:07:20:13 +0000] "GET /page/1" 200 5683
244.210.202.222 - - [25/Feb/2015:07:20:13 +0000] "HEAD /article/490" 200 47124
103.138.203.165 - - [25/Feb/2015:07:20:13 +0000] "HEAD /listing/567" 200 48734
103.138.203.165 - - [25/Feb/2015:07:20:13 +0000] "GET /listing/567" 200 27392
191.141.247.227 - - [25/Feb/2015:07:20:13 +0000] "GET /listing/567" 200 15705
150.255.232.154 - - [25/Feb/2015:07:20:13 +0000] "GET /page/1" 200 22587
244.210.202.222 - - [25/Feb/2015:07:20:13 +0000] "HEAD /product/982" 200 30063
244.210.202.222 - - [25/Feb/2015:07:20:13 +0000] "GET /page/1" 200 6041
166.208.249.236 - - [25/Feb/2015:07:20:13 +0000] "GET /product/982" 200 25783
191.141.247.227 - - [25/Feb/2015:07:20:13 +0000] "GET /article/490" 200 1099
244.210.202.222 - - [25/Feb/2015:07:20:13 +0000] "GET /product/982" 200 31494
191.141.247.227 - - [25/Feb/2015:07:20:13 +0000] "GET /listing/567" 200 30389
150.255.232.154 - - [25/Feb/2015:07:20:13 +0000] "GET /article/490" 200 10251
191.141.247.227 - - [25/Feb/2015:07:20:13 +0000] "GET /product/982" 200 19384
150.255.232.154 - - [25/Feb/2015:07:20:13 +0000] "HEAD /product/982" 200 24062
244.210.202.222 - - [25/Feb/2015:07:20:13 +0000] "GET /article/490" 200 19070
191.141.247.227 - - [25/Feb/2015:07:20:13 +0000] "GET /page/648" 200 45159
191.141.247.227 - - [25/Feb/2015:07:20:13 +0000] "HEAD /page/648" 200 5576
166.208.249.236 - - [25/Feb/2015:07:20:13 +0000] "GET /page/648" 200 41869
166.208.249.236 - - [25/Feb/2015:07:20:13 +0000] "GET /listing/567" 200 42414
```

```
def count_distinct(stream):  
    seen = set()  
    for item in stream:  
        seen.add(item)  
    return len(seen)
```

Scale.

Count-distinct across  
thousands of servers.

```
def combined_cardinality(seen_sets):  
    combined = set()  
    for seen in seen_sets:  
        combined |= seen  
    return len(combined)
```



The set grows linearly.

Precision comes at a cost.

# HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm

Philippe Flajolet<sup>1</sup> and Éric Fusy<sup>1</sup> and Olivier Gandouet<sup>2</sup> and Frédéric Meunier<sup>1</sup>

<sup>1</sup>*Algorithms Project, INRIA–Rocquencourt, F78153 Le Chesnay (France)*

<sup>2</sup>*LIRMM, 161 rue Ada, 34392 Montpellier (France)*

---

This extended abstract describes and analyses a near-optimal probabilistic algorithm, HYPERLOGLOG, dedicated to estimating the number of *distinct* elements (the *cardinality*) of very large data ensembles. Using an auxiliary memory of  $m$  units (typically, “short bytes”), HYPERLOGLOG performs a single pass over the data and produces an estimate of the cardinality such that the relative accuracy (the *standard error*) is typically about  $1.04/\sqrt{m}$ . This improves on the best previously known cardinality estimator, LOGLOG, whose accuracy can be matched by consuming only 64% of the original memory. For instance, the new algorithm makes it possible to estimate cardinalities well beyond  $10^9$  with a typical accuracy of 2% while using a memory of only 1.5 kilobytes. The algorithm parallelizes optimally and adapts to the sliding window model.

## Introduction

The purpose of this note is to present and analyse an efficient algorithm for estimating the *number of distinct elements*, known as the *cardinality*, of large data ensembles, which are referred to here as *multisets* and are usually massive *streams* (read-once sequences). This problem has received a great deal of attention over the past two decades, finding an ever growing number of applications in networking and traffic monitoring, such as the detection of worm propagation, of network attacks (e.g., by Denial of Service), and of link-based spam on the web [3]. For instance, a data stream over a network consists of a sequence of packets, each packet having a header, which contains a pair (source–destination) of addresses, followed by a body of specific data; the number of distinct header pairs (the cardinality of the multiset) in various

# Loglog Counting of Large Cardinalities

## (Extended Abstract)

Marianne Durand and Philippe Flajolet

Algorithms Project, INRIA–Rocquencourt, F78153 Le Chesnay (France)

**Abstract.** Using an auxiliary memory smaller than the size of this abstract, the LOGLOG algorithm makes it possible to estimate in a single pass and within a few percents the number of different words in the whole of Shakespeare’s works. In general the LOGLOG algorithm makes use of  $m$  “small bytes” of auxiliary memory in order to estimate in a single pass the number of distinct elements (the “*cardinality*”) in a file, and it does so with an accuracy that is of the order of  $1/\sqrt{m}$ . The “small bytes” to be used in order to count cardinalities till  $N_{\max}$  comprise about  $\log \log N_{\max}$  bits, so that cardinalities well in the range of billions can be determined using one or two kilobytes of memory only. The basic version of the LOGLOG algorithm is validated by a complete analysis. An optimized version, super-LOGLOG, is also engineered and tested on real-life data. The algorithm parallelizes optimally.

## 1 Introduction

The problem addressed in this note is that of determining the number of *distinct*

“example.com/user/page/1”

“example.com/user/page/1”



1 0 1 1 0 1 0 1

**1 0 1 1 0 1 0 1**


1 0 1 1 0 1 0 1



$P(\text{bit } 0 \text{ is set}) = 0.5$




1 0 1 1 0 1 0 1



$P(\text{bit } 0 \text{ is set}$   
 $\& \text{ bit } 1 \text{ is set}) = 0.25$

1 0 1 1 0 1 0 1



P(bit 0 is set  
& bit 1 is set  
& bit 2 is set) = 0.125

1 0 1 1 0 1 0 1



$P(\text{bit } 0 \text{ is set}) = 0.5$

Expected trials = 2

1 0 1 1 0 1 0 1  
↑ ↑

P(bit 0 is set  
& bit 1 is set) = 0.25  
Expected trials = 4

1 0 1 1 0 1 0 1  
↑ ↑ ↑

P(bit 0 is set  
& bit 1 is set  
& bit 2 is set) = 0.125  
Expected trials = 8

We expect the maximum number of leading zeros we have seen + 1 to approximate  $\log_2(\text{unique items})$ .

Improve the accuracy of  
the estimate by  
partitioning the input data.

```
class LogLog(object):
    def __init__(self, k):
        self.k = k
        self.m = 2 ** k
        self.M = np.zeros(self.m, dtype=np.int)
        self.alpha = Alpha[k]

    def insert(self, token):
        y = hash_fn(token)
        j = y >> (hash_len - self.k)
        remaining = y & ((1 << (hash_len - self.k)) - 1)
        first_set_bit = (64 - self.k) -
            int(math.log(remaining, 2))
        self.M[j] = max(self.M[j], first_set_bit)

    def cardinality(self):
        return self.alpha * 2 ** np.mean(self.M)
```



# Unions of HyperLogLogs

# HyperLogLog

- Adding an item:  $O(1)$
- Retrieving cardinality:  $O(1)$
- Space:  $O(\log \log n)$
- Error rate: 2%

# HyperLogLog

For 100 million unique items,  
and an error rate of 2%,  
the size of the HyperLogLog is...

**1,500 bytes**

# Reliable Broadcast

# The Problem

Reliably broadcast “purge” messages across the world  
as quickly as possible.

Single source of  
truth

Single source of  
failures

Atomic broadcast



Reliable broadcast

# A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing

Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang  
to appear in IEEE/ACM Transactions on Networking, December 1997

*Abstract*—This paper<sup>1</sup> describes SRM (Scalable Reliable Multicast), a reliable multicast framework for light-weight sessions and application level framing. The algorithms of this framework are efficient, robust, and scale well to both very large networks and very large sessions. The SRM framework has been prototyped in *wb*, a distributed whiteboard application, which has been used on a global scale with sessions ranging from a few to a few hundred participants. The paper describes the principles that have guided the SRM design, including the IP multicast group delivery model, an end-to-end, receiver-based model of reliability, and the application level framing protocol model. As with unicast communications, the performance of a reliable multicast delivery algorithm depends on the underlying topology and operational environment. We investigate that dependence via analysis and simulation, and demonstrate an adaptive algorithm that uses the results of previous loss recovery events to adapt the control parameters used for future loss recovery. With the adaptive algorithm, our reliable multicast delivery algorithm provides good performance over a wide range of underlying topologies.

## 1 Introduction

Several researchers have proposed generic reliable multicast protocols, much as TCP is a generic transport protocol for reliable unicast transmission. In this paper we take a different view: unlike the unicast case where requirements for reliable, sequenced data delivery are fairly general, different multicast applications have widely different requirements for reliability. For example, some applications require that delivery obey a total ordering while many others do not. Some applications have many or all the members sending data while others have only one data source. Some applications have replicated data, for example in an  $n$ -redundant file store, so several members are capable of transmitting a data item while for others all data originates at a single source. These differences all affect the design of a reliable multicast protocol. Although one could design a protocol for the worst-case requirements, e.g., guaranteeing totally ordered

recognition. In 1990 Clark and Tennenhouse proposed a new protocol model called Application Level Framing (ALF) which explicitly includes an application's semantics in the design of that application's protocol [6]. ALF was later elaborated with a light-weight rendezvous mechanism based on the IP multicast distribution model, and with a notion of receiver-based adaptation for unreliable, real-time applications such as audio and video conferencing. The result, known as Light-Weight Sessions (LWS) [19], has been very successful in the design of wide-area, large-scale, conferencing applications. This paper further evolves the principles of ALF and LWS to add a framework for Scalable Reliable Multicast (SRM).

ALF says that the best way to meet diverse application requirements is to leave as much functionality and flexibility as possible to the application. Therefore SRM is designed to meet only the minimal definition of reliable multicast, i.e., eventual delivery of all the data to all the group members, without enforcing any particular delivery order. We believe that if the need arises, machinery to enforce a particular delivery order can be easily added on top of this reliable delivery service.

It has been argued [36, 34] that a single dynamically configurable protocol should be used to accommodate different application requirements. The ALF argument is even stronger: not only do different applications require different types of error recovery, flow control, and rate control mechanisms, but further, these mechanisms must explicitly account for the structure of the underlying application data itself.

SRM is also heavily based on the group delivery model that is the centerpiece of the IP multicast protocol [8]. In IP multicast, data sources simply send to the group's multicast address (a normal IP address chosen from a reserved range of addresses) without needing any advance knowledge of the group member-

# AN EFFICIENT RELIABLE BROADCAST PROTOCOL

*M. Frans Kaashoek  
Andrew S. Tanenbaum  
Susan Flynn Hummel  
Henri E. Bal*

Dept. of Mathematics and Computer Science  
Vrije Universiteit  
Amsterdam, The Netherlands

Email: kaashoek@cs.vu.nl

## ABSTRACT

Many distributed and parallel applications can make good use of broadcast communication. In this paper we present a (software) protocol that simulates reliable broadcast, even on an unreliable network. Using this protocol, application programs need not worry about lost messages. Recovery of communication failures is handled automatically and transparently by the protocol. In normal operation, our protocol is more efficient than previously published reliable broadcast protocols. An initial implementation of the protocol on 10 MC68020 CPUs connected by a 10 Mbit/sec Ethernet performs a reliable broadcast in 1.5 msec.

## 1. INTRODUCTION

Most current distributed operating systems are based on remote procedure call (RPC) [Birrell and Nelson 1984]. For many distributed and parallel applications, however, this sender-to-receiver-and-back communication style is inappropriate. What is frequently needed is *broadcasting*, in which an arbitrary one of the  $n$  user processes sends a message to the other  $n - 1$  processes. Although broadcasting can always be simulated by sending  $n - 1$  messages and waiting for the  $n - 1$  acknowledgements, this algorithm is slow, inefficient, and wasteful of network bandwidth. In this paper we discuss a new protocol that allows 100% reliable broadcasting to be implemented on unreliable networks in only two messages per broadcast.

# Gossip Protocols

## Epidemic Broadcast Trees \*

João Leitão

University of Lisbon

jleitao@lasige.di.fc.ul.pt

José Pereira

University of Minho

jop@di.uminho.pt

Luís Rodrigues

University of Lisbon

ler@di.fc.ul.pt

### Abstract

*There is an inherent trade-off between epidemic and deterministic tree-based broadcast primitives. Tree-based approaches have a small message complexity in steady-state but are very fragile in the presence of faults. Gossip, or epidemic, protocols have a higher message complexity but also offer much higher resilience.*

*This paper proposes an integrated broadcast scheme that combines both approaches. We use a low cost scheme to build and maintain broadcast trees embedded on a gossip-based overlay. The protocol sends the message payload preferably via tree branches but uses the remaining links of the gossip overlay for fast recovery and expedite tree healing. Experimental evaluation presented in the paper shows that our new strategy has a low overhead and that is able to support large number of faults while maintaining a high reliability.*

### 1. Introduction

Many systems require highly scalable and reliable broadcast primitives. These primitives aim at ensuring that all correct participants receive all broadcast messages, even in the presence of network omissions or node failures.

# Sprinkler — Reliable Broadcast for Geographically Dispersed Datacenters

Haoyan Geng and Robbert van Renesse

Cornell University, Ithaca, New York, USA

**Abstract.** This paper describes and evaluates Sprinkler, a reliable high-throughput broadcast facility for geographically dispersed datacenters. For scaling cloud services, datacenters use caching throughout their infrastructure. Sprinkler can be used to broadcast update events that invalidate cache entries. The number of recipients can scale to many thousands in such scenarios. The Sprinkler infrastructure consists of two layers: one layer to disseminate events among datacenters, and a second layer to disseminate events among machines within a datacenter. A novel garbage collection interface is introduced to save storage space and network bandwidth. The first layer is evaluated using an implementation deployed on Emulab. For the second layer, involving thousands of nodes, we use a discrete event simulation. The effect of garbage collection is analyzed using simulation. The evaluation shows that Sprinkler can disseminate millions of events per second throughout a large cloud infrastructure, and garbage collection is effective in workloads like cache invalidation.

**Keywords:** Broadcast, performance, fault tolerance, garbage collection

## 1 Introduction

Today's large scale web applications such as Facebook, Amazon, eBay, Google+, and so on, rely heavily on caching for providing low latency responses to client queries. Enterprise data is stored in reliable but slow back-end databases. In order to be able to keep up with load and provide low latency responses, client query results are computed and opportunistically cached in memory on many thousands of machines throughout the organization's various datacenters [21]. But when a database is updated, all affected cache entries have to be invalidated. Until this is completed, inconsistent data can be exposed to clients. Since the databases cannot keep track of where these cache entries are, it is necessary to multicast an invalidation notification to all machines that may have cached query results. The rate of such invalidations can reach hundreds of thousands per second. If any invalidation gets lost, inconsistencies exposed to clients may be

“Designed for Scale”

# Probabilistic Guarantees



# Bimodal Multicast

KENNETH P. BIRMAN

Cornell University

MARK HAYDEN

Digital Equipment Corporation/Compaq

OZNUR OZKASAP and ZHEN XIAO

Cornell University

MIHAI BUDIU

Carnegie Mellon University

and

YARON MINSKY

Cornell University

---

There are many methods for making a multicast protocol "reliable." At one end of the spectrum, a reliable multicast protocol might offer atomicity guarantees, such as all-or-nothing delivery, delivery ordering, and perhaps additional properties such as virtually synchronous addressing. At the other are protocols that use local repair to overcome transient packet loss in the network, offering "best effort" reliability. Yet none of this prior work has treated stability of multicast delivery as a basic reliability property, such as might be needed in an internet radio, television, or conferencing application. This article looks at reliability with a new goal: development of a multicast protocol which is reliable in a sense that can be rigorously quantified and includes throughput stability guarantees. We characterize this new protocol as a "bimodal multicast" in reference to its reliability model, which corresponds to a family of bimodal probability distributions. Here, we introduce the protocol, provide a theoretical analysis of its behavior, review experimental results, and discuss some candidate applications. These confirm that bimodal multicast is reliable, scalable, and that the protocol provides remarkably stable delivery throughput.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network

---

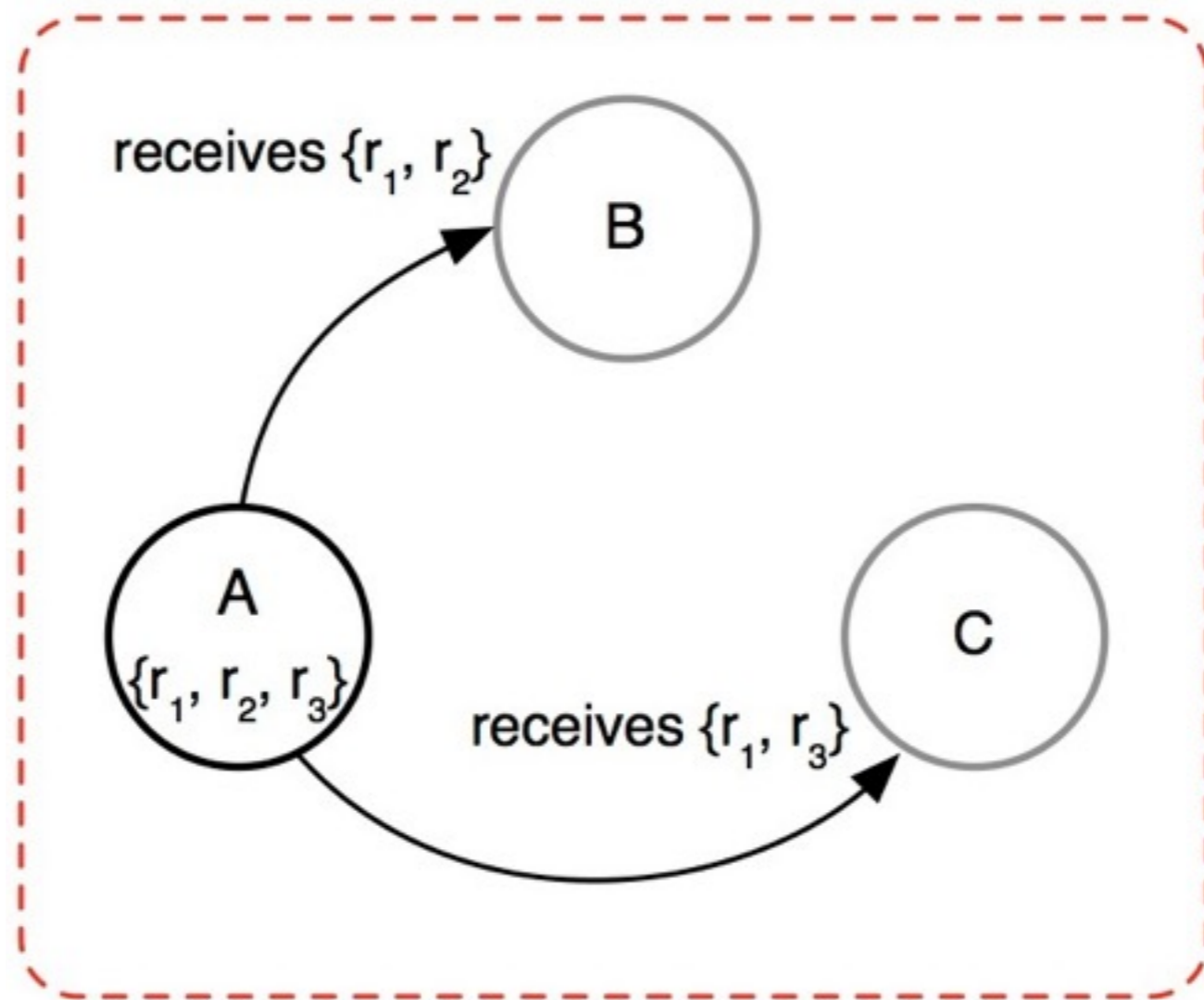
This work was supported by DARPA/ONR contracts N0014-96-1-10014 and ARPA/RADC F30602-96-1-0317, the Cornell Theory Center, and the Turkish Research Foundation.

Authors' addresses: K. P. Birman, Department of Computer Science, Cornell University, 4126 Upson Hall, Ithaca, NY 14853; email: ken@cs.cornell.edu; M. Hayden, Systems Research Center, Digital Equipment Corporation/Compaq, 130 Lytton Avenue, Palo Alto, CA 94301; email: hayden@src.dec.com; O. Ozkasap and Z. Xiao, Department of Computer Science, Cornell University, 4126 Upson Hall, Ithaca, NY 14853; email: ozkasap@cs.cornell.edu; xiao@cs.cornell.edu; M. Budiu, Department of Computer Science, Carnegie Mellon University, Ithaca, NY 14853; email: mihaib@cs.cmu.edu; Y. Minsky, Department of Computer Science, Cornell University, 4126 Upson Hall, Ithaca, NY 14853.

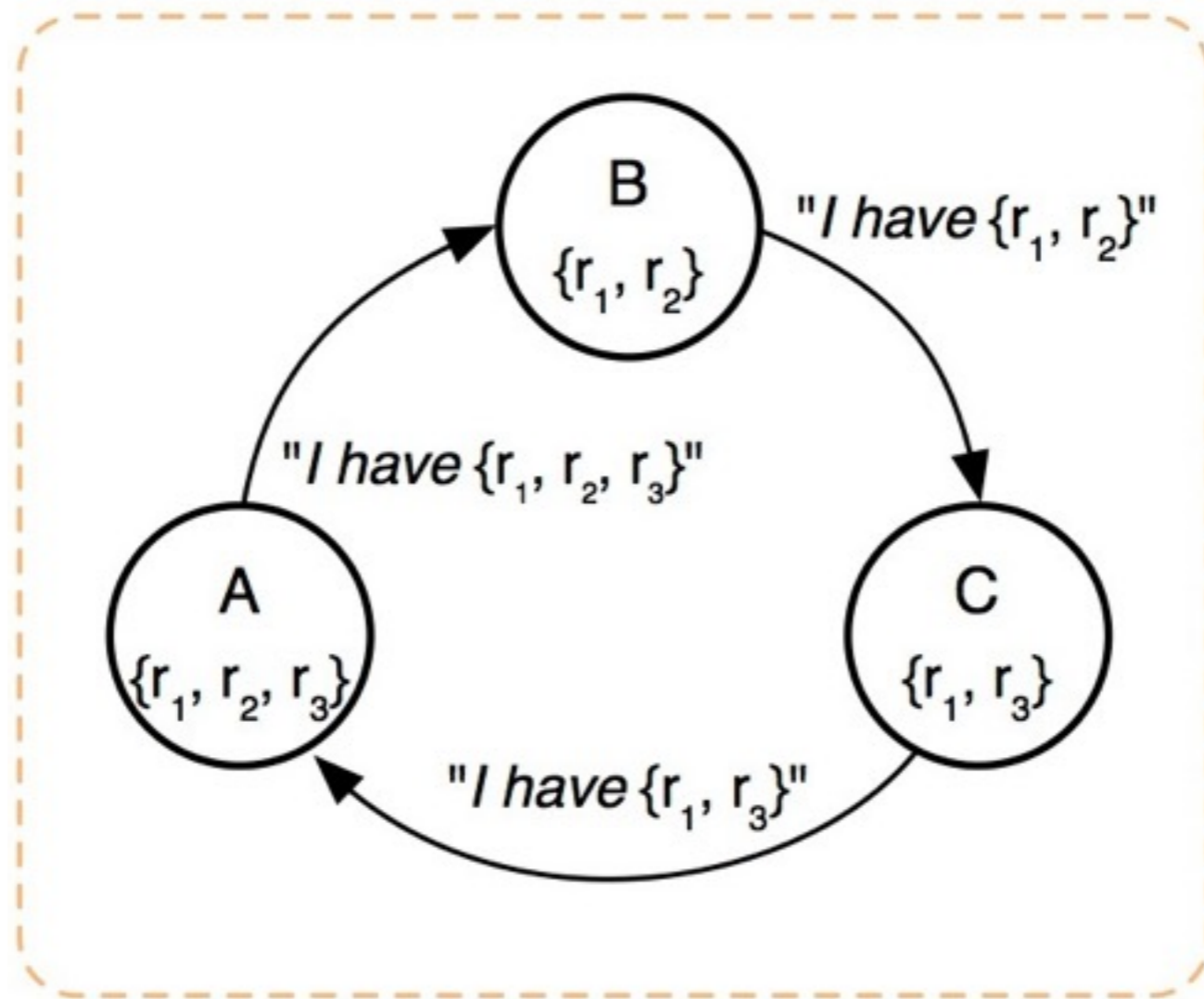
Permission to make digital/hard copy of part or all of this work for personal or classroom use

# Bimodal Multicast

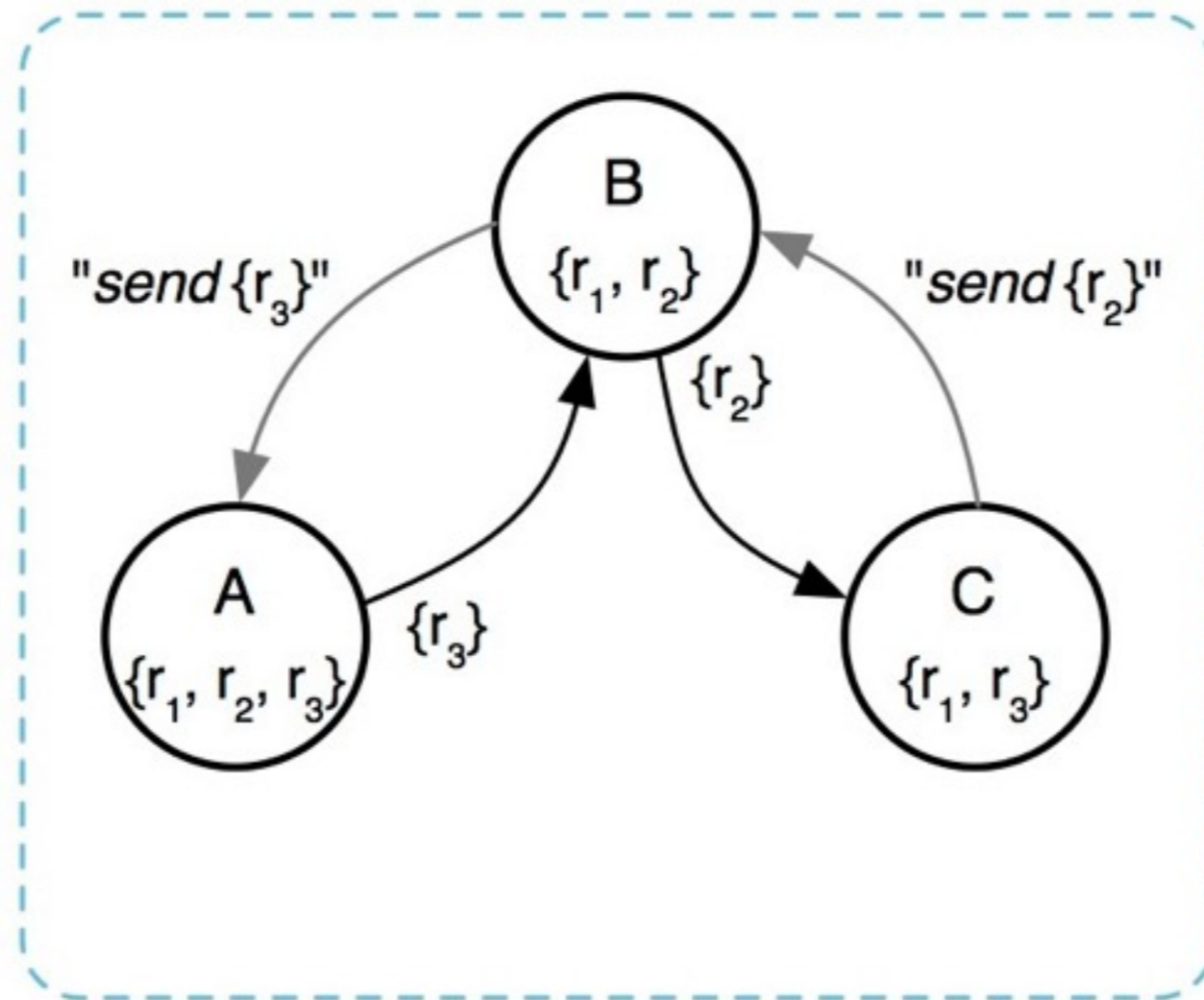
- Quickly broadcast message to all servers
- Gossip to recover lost messages



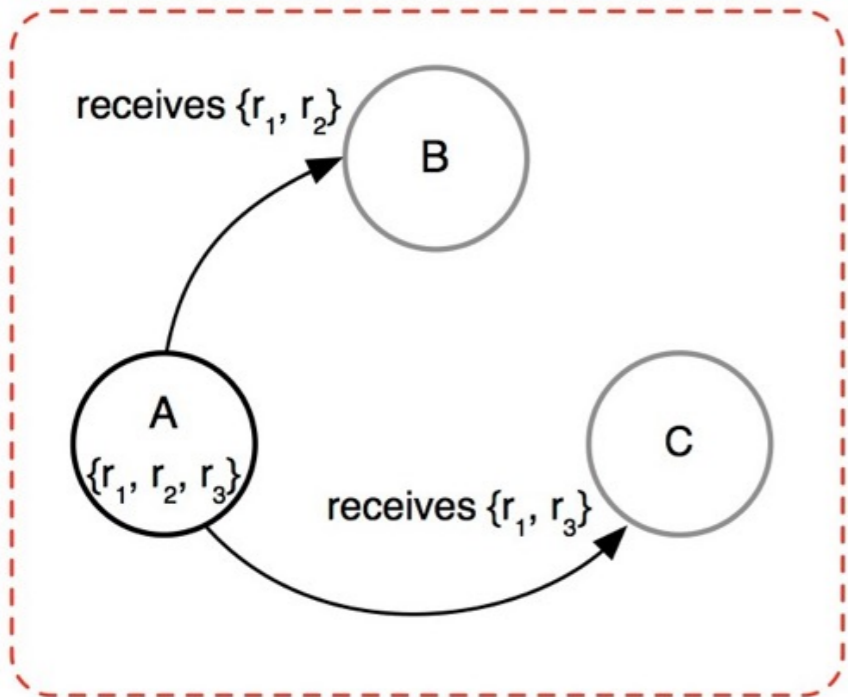
**Broadcast**



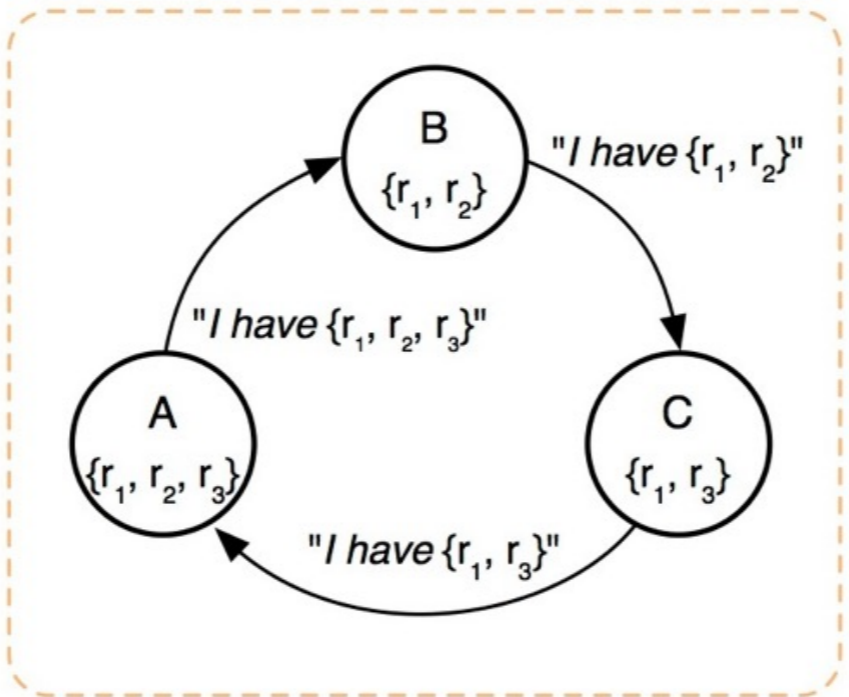
**Gossip**



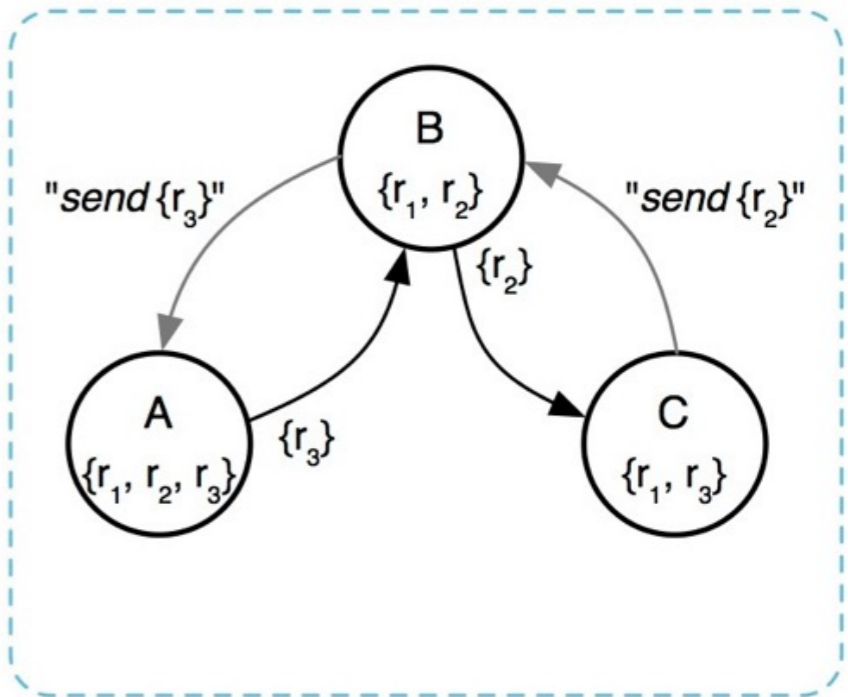
**Recover**



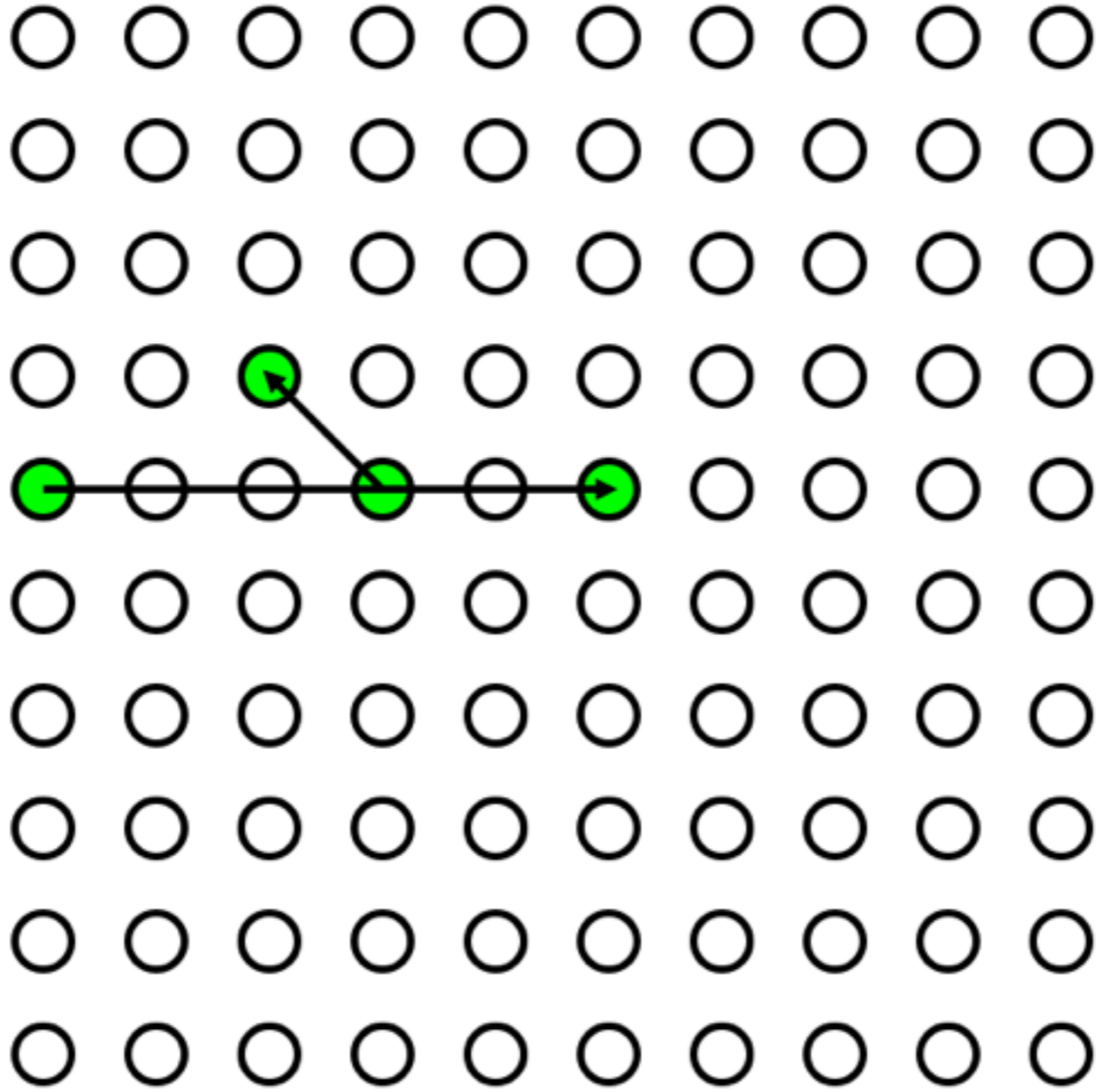
**Broadcast**



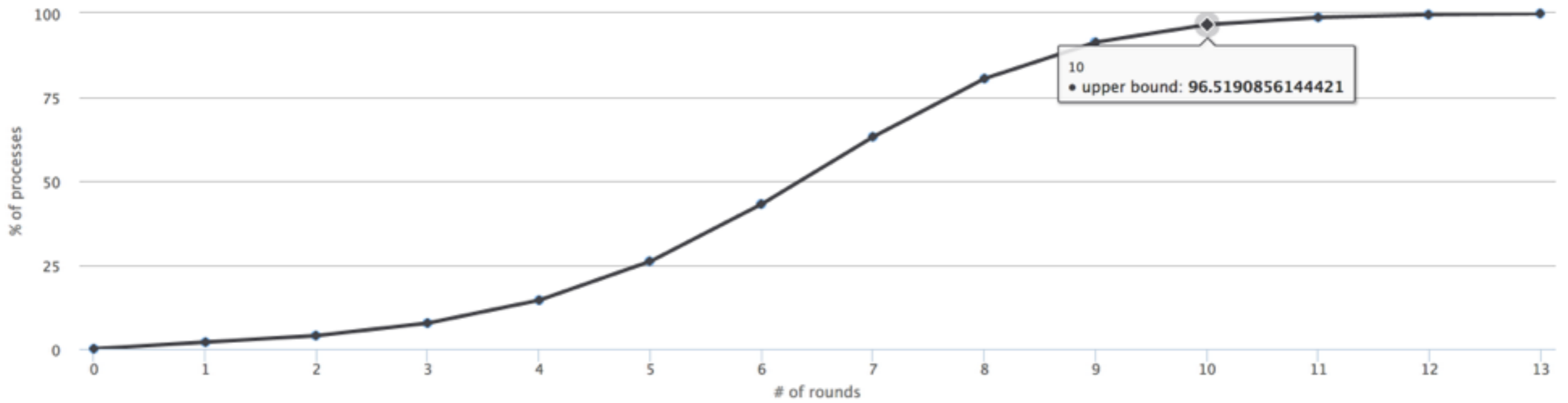
**Gossip**



**Recover**



Expected percent of infected processes after n rounds



● lower bound    ● upper bound

Highcharts.com

Number of rounds **12**

P(message loss)  $\leq$  **0**

% of processes who received initial broadcast **0.01**



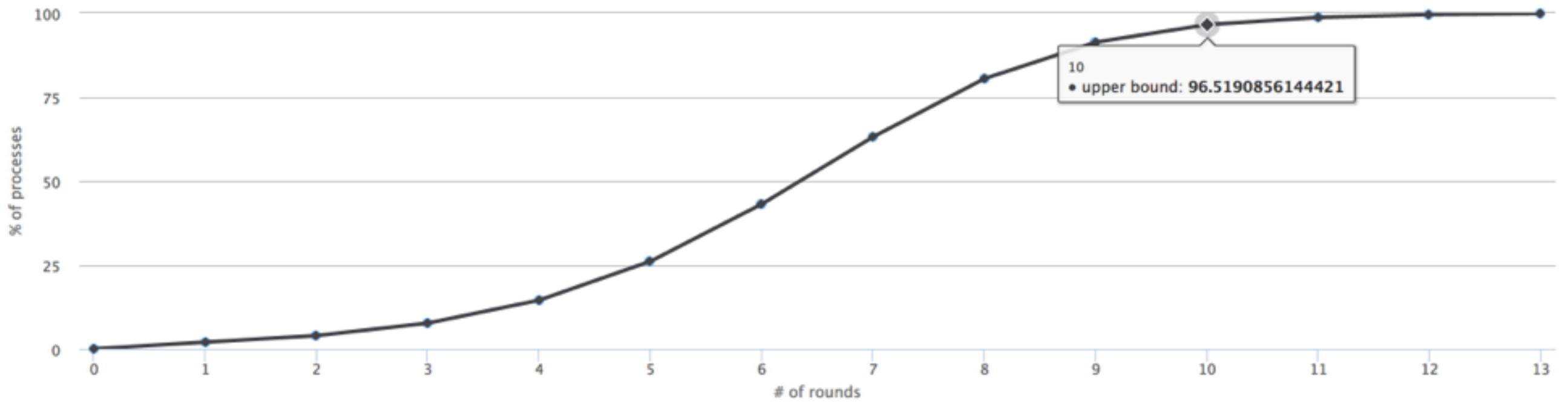


# One Problem

Computers have limited space

Throw away messages

Expected percent of infected processes after n rounds



● lower bound    ◆ upper bound

Highcharts.com

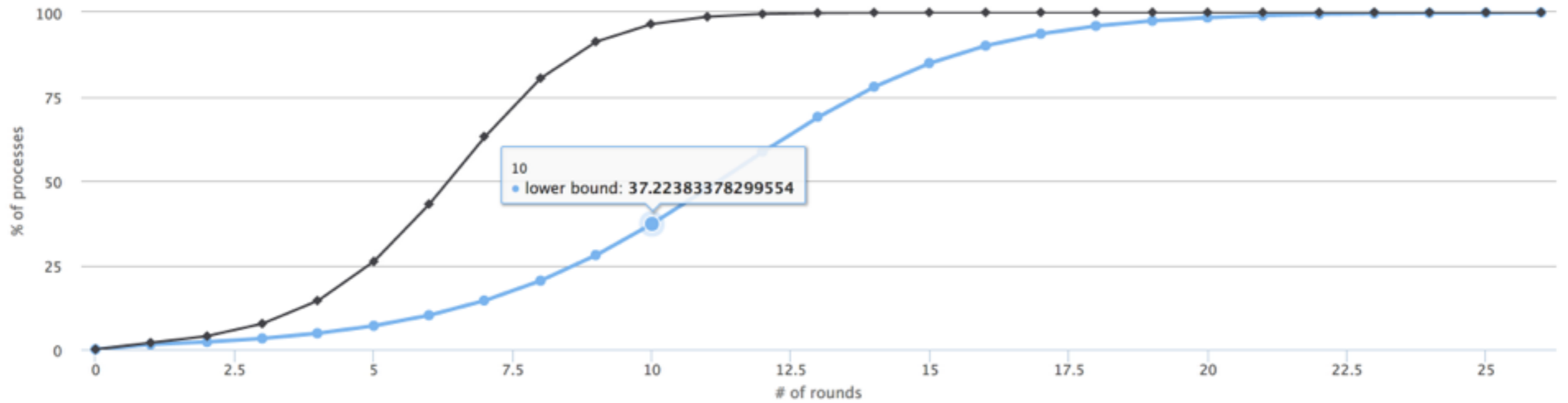
Number of rounds **12**

P(message loss)  $\leq$  **0**

% of processes who received initial broadcast **0.01**



Expected percent of infected processes after n rounds



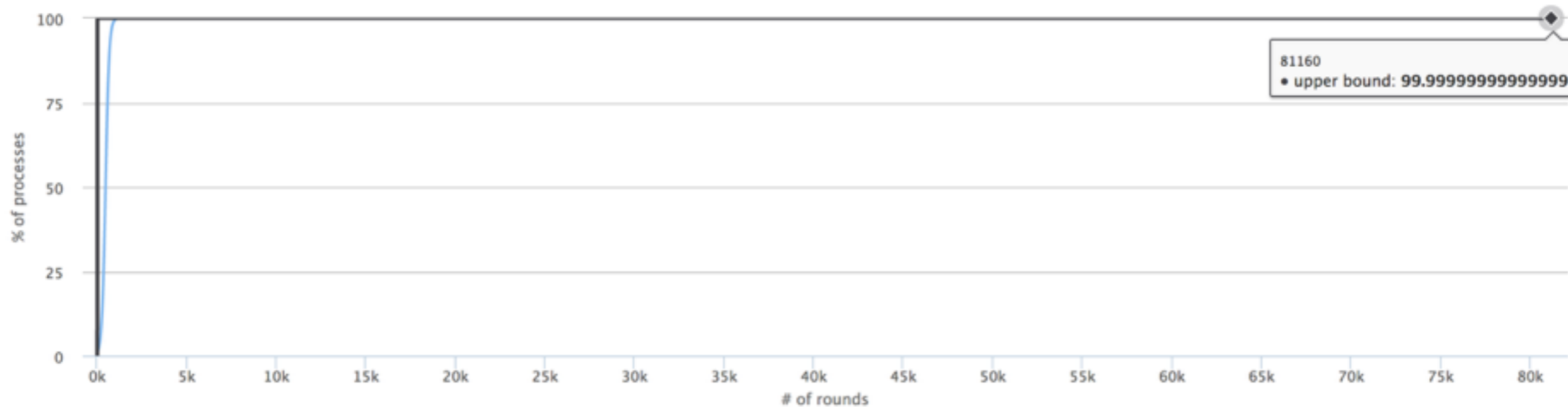
● lower bound    ◆ upper bound

Number of rounds 25

P(message loss)  $\leq$  0.51

% of processes who received initial broadcast 0.01

Expected percent of infected processes after n rounds



◆ lower bound    ◆ upper bound

Number of rounds **81283**

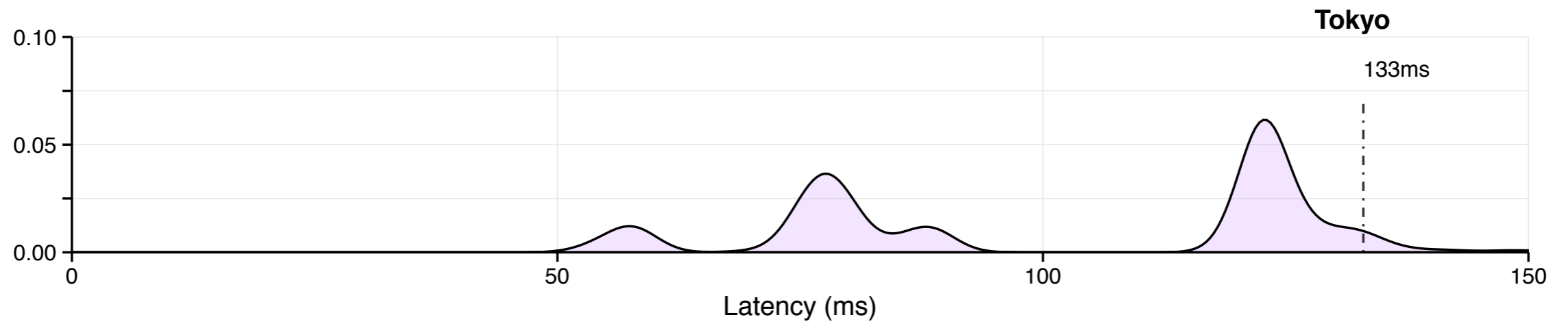
P(message loss)  $\leq$  **0.99**

% of processes who received initial broadcast **0.01**

“with high probability” is fine

Real World

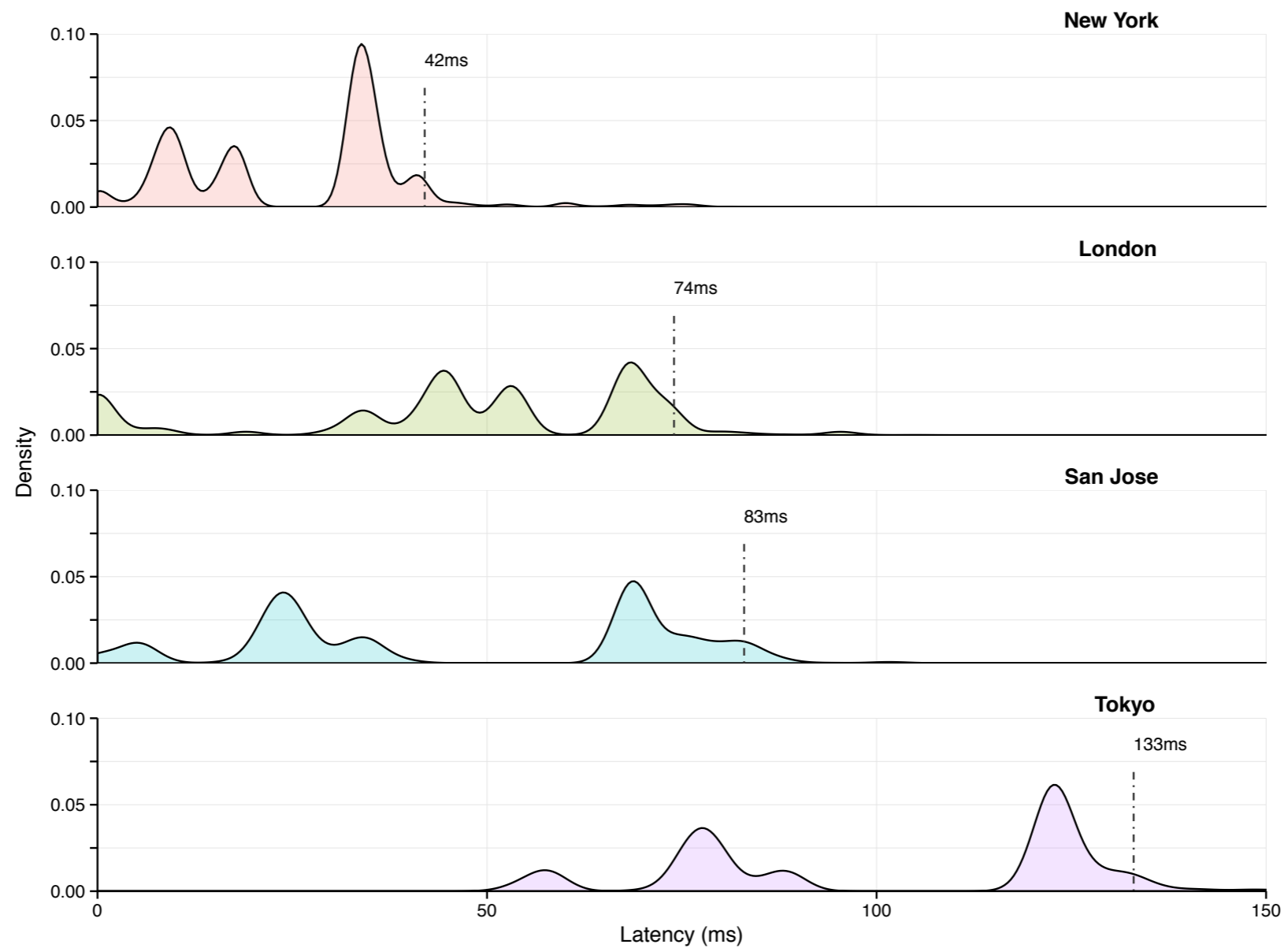
# End-to-End Latency





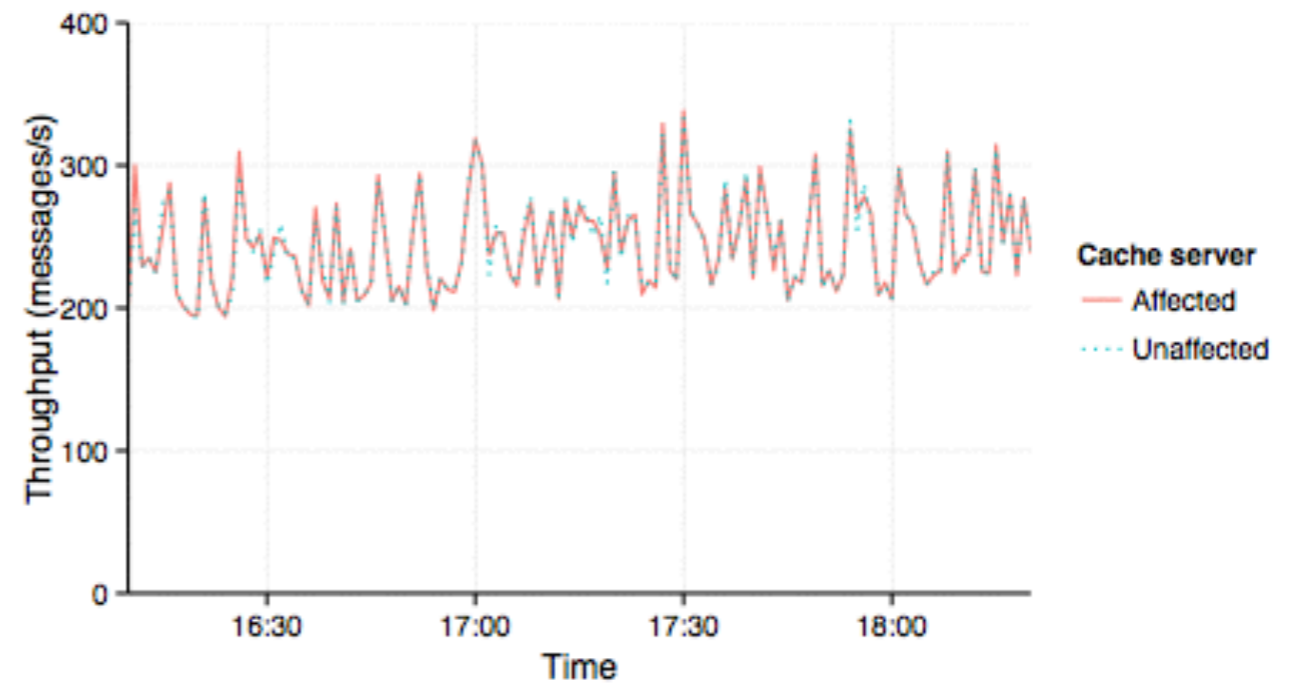
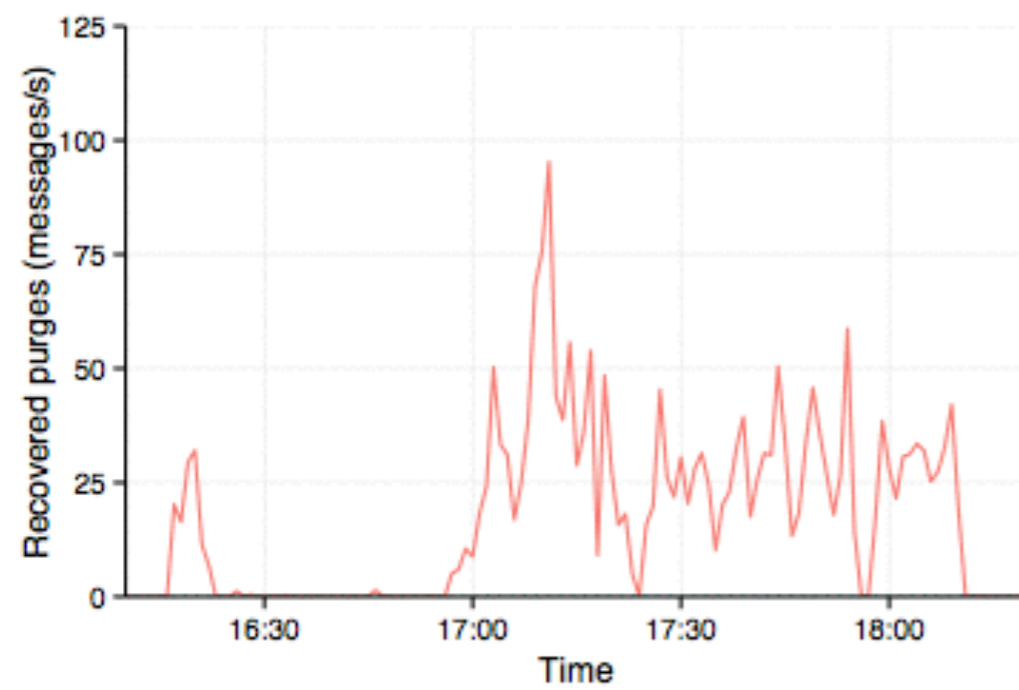
# End-to-End Latency

Density plot and 95<sup>th</sup> percentile of purge latency by server location



# Packet Loss

Purge performance



Good systems are boring

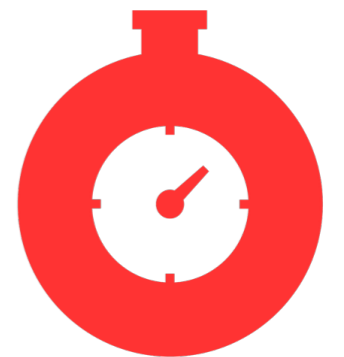
What was the point again?

We can build things that  
are otherwise  
unrealistic

We can build systems  
that are more  
reliable

You're already using them.

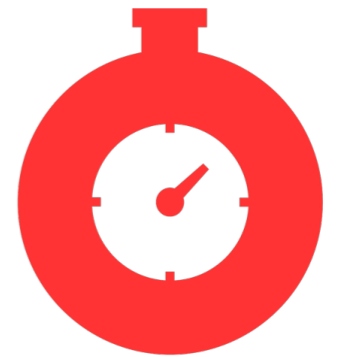
We're hiring!





# Thanks

@tbmcmullen



What even is this?

# Probabilistic Algorithms

# Randomized Algorithms

# Estimation Algorithms

# Probabilistic Algorithms

1. An iota of theory
2. Where are they useful and where are they not?
3. HyperLogLog
4. Locality-sensitive Hashing
5. Bimodal Multicast

“An algorithm that uses randomness to improve its efficiency”



Las Vegas





Monte Carlo

# Las Vegas

```
def find_las_vegas(haystack, needle):  
    length = len(haystack)  
    while True:  
        index = randrange(length)  
        if haystack[index] == needle:  
            return index
```

# Monte Carlo

```
def find_monte_carlo(haystack, needle, k):  
    length = len(haystack)  
    for i in range(k):  
        index = randrange(length)  
        if haystack[index] == needle:  
            return index
```

“For many problems a randomized algorithm is the simplest the fastest or both.”

– Prabhakar Raghavan (author of *Randomized Algorithms*)

# Naive Solution

For 100 million unique IPv4 addresses,  
the size of the hash is...

**>400mb**

# Slightly Less Naive

Add each IP to a *bloom filter* and keep a *counter* of the IPs that don't collide.

# Slightly Less Naive

```
ips_seen = BloomFilter(capacity=expected_size, error_rate=0.03)
counter = 0

for line in log_file:
    ip = extract_ip(line)
    if items_bloom.add(ip):
        counter += 1

print "Unique IPs:", counter
```

# Slightly Less Naive

- Adding an IP:  $O(1)$
- Retrieving cardinality:  $O(1)$
- Space:  $O(n)$  ← kind of
- Error rate: 3%



# Slightly Less Naive

For 100 million unique IPv4 addresses,  
and an error rate of 3%,  
the size of the bloom filter is...

**87mb**

```
def insert(self, token):
    # Get hash of token
    y = hash_fn(token)

    # Extract `k` most significant bits of `y`
    j = y >> (hash_len - self.k)

    # Extract remaining bits of `y`
    remaining = y & ((1 << (hash_len - self.k)) - 1)

    # Find "first" set bit of `remaining`
    first_set_bit = (64 - self.k) -
                    int(math.log(remaining, 2))

    # Update `M[j]` to max of `first_set_bit`
    # and existing value of `M[j]`
    self.M[j] = max(self.M[j], first_set_bit)
```

```
def cardinality(self):  
    # The mean of `M` estimates `log2(n)` with  
    # an additive bias  
    return self.alpha * 2 ** np.mean(self.M)
```

# The Problem

Find documents that are similar to one specific document.

# The Problem

Find images that are similar to one specific image.

# The Problem

Find graphs that are correlated to one specific graph.

# The Problem

Nearest neighbor search.

# The Problem

“Find the  $n$  closest points in a  $d$ -dimensional space.”



# The Problem

You have a bunch of things and you want to figure out which ones are similar.



“There has been a lot of recent work on streaming algorithms, i.e. algorithms that produce an output by making one pass (or a few passes) over the data while using a limited amount of storage space and time. To cite a few examples, ...”



```
{ "there": 1, "has": 1, "been": 1, "a":4,  
  "lot": 1, "of": 2, "recent":1, ... }
```

- Cosine similarity
- Jaccard similarity
- Euclidian distance
- etc etc etc

# Euclidian Distance

Metric space

kd-trees

# Curse of Dimensionality



Locality-sensitive hashing

# Similarity Search in High Dimensions via Hashing

ARISTIDES GIONIS \*      PIOTR INDYK<sup>†</sup>      RAJEEV MOTWANI<sup>‡</sup>

Department of Computer Science

Stanford University

Stanford, CA 94305

{gionis, indyk, rajeev}@cs.stanford.edu

## Abstract

The nearest- or near-neighbor query problems arise in a large variety of database applications, usually in the context of similarity searching. Of late, there has been increasing interest in building search/index structures for performing similarity search over high-dimensional data, e.g., image databases, document collections, time-series databases, and genome databases. Unfortunately, all known techniques for solving this problem fall prey to the “curse of dimensionality.” That is, the data structures scale poorly with data dimensionality; in fact, if the number of dimensions exceeds 10 to 20, searching in  $k$ -d trees and related structures involves the inspection of a large fraction of the database, thereby doing no better than brute-force linear search. It has been suggested that since the selection of features and the choice of a distance metric in typical applications is rather heuristic, determining an approximate nearest neighbor should suffice for most practical purposes. In this paper, we examine a novel scheme for approximate similarity search based on hashing. The basic idea is to hash the points

from the database so as to ensure that the probability of collision is much higher for objects that are close to each other than for those that are far apart. We provide experimental evidence that our method gives significant improvement in running time over other methods for searching in high-dimensional spaces based on hierarchical tree decomposition. Experimental results also indicate that our scheme scales well even for a relatively large number of dimensions (more than 50).

## 1 Introduction

A similarity search problem involves a collection of objects (e.g., documents, images) that are characterized by a collection of relevant features and represented as points in a high-dimensional attribute space; given queries in the form of points in this space, we are required to find the nearest (most similar) object to the query. The particularly interesting and well-studied case is the  $d$ -dimensional Euclidean space. The problem is of major importance to a variety of applications; some examples are: data compression [20]; databases and data mining [21]; information retrieval [11, 16, 38]; image and video databases [15, 17, 37, 42]; machine learning [7]; pattern recognition [9, 13]; and, statistics and data analysis [12, 27]. Typically, the features of the objects of interest are represented as points in  $\mathbb{R}^d$  and a distance metric is used to measure similarity of objects. The basic problem then is to perform indexing or similarity searching for query objects. The number

\*Supported by NAVY N00014-96-1-1221 grant and NSF Grant IIS-9811904.

<sup>†</sup>Supported by Stanford Graduate Fellowship and NSF NYI

# Similarity Estimation Techniques from Rounding Algorithms

Moses S. Charikar  
Dept. of Computer Science  
Princeton University  
35 Olden Street  
Princeton, NJ 08544  
moses@cs.princeton.edu

## ABSTRACT

A locality sensitive hashing scheme is a distribution on a family  $\mathcal{F}$  of hash functions operating on a collection of objects, such that for two objects  $x, y$ ,

$$\Pr_{h \in \mathcal{F}}[h(x) = h(y)] = \text{sim}(x, y),$$

where  $\text{sim}(x, y) \in [0, 1]$  is some similarity function defined on the collection of objects. Such a scheme leads to a compact representation of objects so that similarity of objects can be estimated from their compact sketches, and also leads to efficient algorithms for approximate nearest neighbor search and clustering. Min-wise independent permutations provide an elegant construction of such a locality sensitive hashing scheme for a collection of subsets with the set similarity measure  $\text{sim}(A, B) = \frac{|A \cap B|}{|A \cup B|}$ .

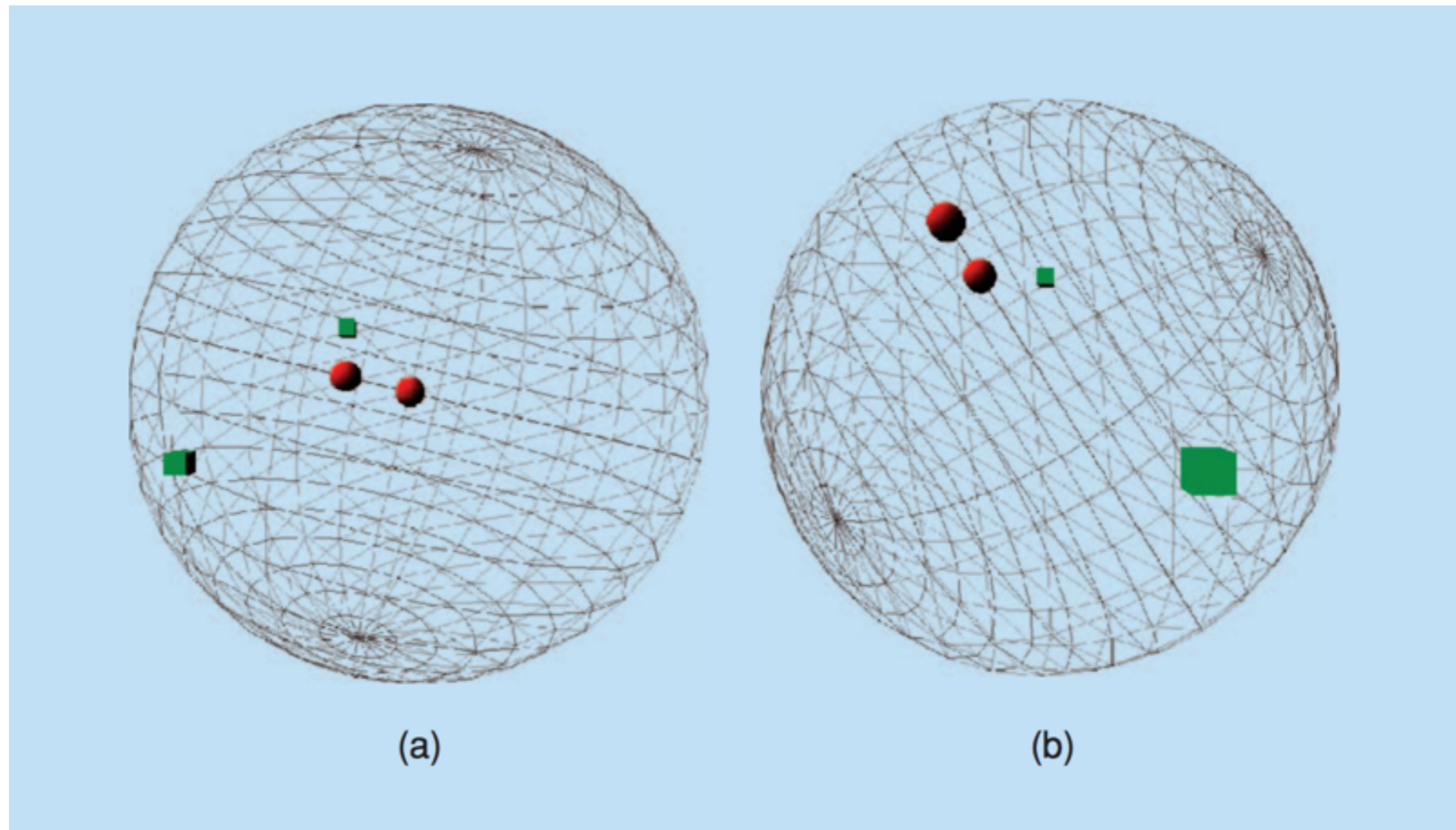
We show that rounding algorithms for LPs and SDPs used in the context of approximation algorithms can be viewed as locality sensitive hashing schemes for several interesting collections of objects. Based on this insight, we construct new locality sensitive hashing schemes for:

1. A collection of vectors with the distance between  $\vec{u}$  and  $\vec{v}$  measured by  $\theta(\vec{u}, \vec{v})/\pi$ , where  $\theta(\vec{u}, \vec{v})$  is the angle between  $\vec{u}$  and  $\vec{v}$ . This yields a sketching scheme for estimating the cosine similarity measure between two vectors, as well as a simple alternative to minwise independent permutations for estimating set similarity.
2. A collection of distributions on  $n$  points in a metric space, with distance between distributions measured by the Earth Mover Distance (**EMD**), (a popular distance measure in graphics and vision). Our hash func-

## 1. INTRODUCTION

The current information explosion has resulted in an increasing number of applications that need to deal with large volumes of data. While traditional algorithm analysis assumes that the data fits in main memory, it is unreasonable to make such assumptions when dealing with massive data sets such as data from phone calls collected by phone companies, multimedia data, web page repositories and so on. This new setting has resulted in an increased interest in algorithms that process the input data in restricted ways, including sampling a few data points, making only a few passes over the data, and constructing a succinct sketch of the input which can then be efficiently processed.

There has been a lot of recent work on streaming algorithms, i.e. algorithms that produce an output by making one pass (or a few passes) over the data while using a limited amount of storage space and time. To cite a few examples, Alon *et al* [2] considered the problem of estimating frequency moments and Guha *et al* [25] considered the problem of clustering points in a streaming fashion. Many of these streaming algorithms need to represent important aspects of the data they have seen so far in a small amount of space; in other words they maintain a compact sketch of the data that encapsulates the relevant properties of the data set. Indeed, some of these techniques lead to sketching algorithms – algorithms that produce a compact sketch of a data set so that various measurements on the original data set can be estimated by efficient computations on the compact sketches. Building on the ideas of [2], Alon *et al* [1] give algorithms for estimating join sizes. Gibbons and Matias [18] give sketching algorithms producing so called *synopsis data structures* for various problems including maintaining ap-



Locality-Sensitive Hashing for Finding Nearest Neighbors  
- Slaney and Casey

# Random Hyperplanes

```
{ "there": 1, "has": 1, "been": 1, "a":4,  
  "lot": 1, "of": 2, "recent":1, ... }
```



LSH



0 1 1 1 0 1 0 0 0 0 1 0 1 0 1 0 ...

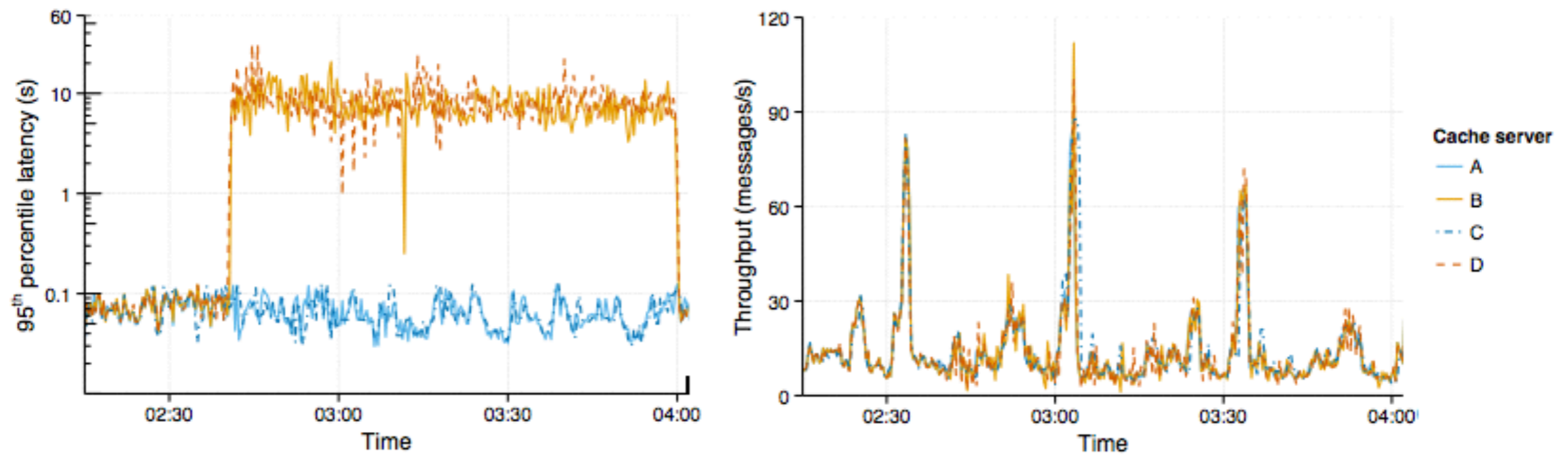
Cosine Similarity



LSH

# Firewall Partition

Purge performance under network partition





# DDoS

Purge performance under denial-of-service attack

