Master's Thesis

Approximate Algorithms for Distributed Systems

Author:	Marie Hoffmann
Institute:	Institut für Informatik, Freie Universität Berlin
Primary Referee:	Prof. Dr. Katinka Wolter
Secondary Referee:	Dr. Thorsten Schütt
Date:	4th of February, 2013

Zusammenfassung

Peer-to-Peer-Systeme (P2P) sind eine spezielle Klasse von verteilten Systemen. In einem typischen P2P-System sind alle Knoten gleichberechtigt und teilen sich die gleichen Aufgaben. In der vorliegenden Arbeit wurden drei typische Probleme in P2P-Systemen behandelt: dem Speichern von Datenkopien, der Quantilberechnung auf verteilten Datenströmen und der Bestimmung der Ausfallrate von Knoten. Das Anfertigen von Datenkopien ist eine der ältesten Techniken um gespeicherte Daten in einem P2P-System zu verwalten und Leseanfragen effizient zu bearbeiten. Zum Beispiel verwenden verteilte Datenbanken diese Technik. Sie gehören einem Overlay-Netzwerk an, welches von der zugrundeliegenden Netzwerk-Topologie aus Hardwareknoten abstrahiert. Die Herausforderung besteht darin, eine Menge von Datenkopien so zu verteilen, dass die Antwortzeiten und Ausfallwahrscheinlichkeiten minimiert werden, ohne dass zuvor die Netzwerk-Topologie bekannt ist. Wir zeigen wie man mithilfe von agglomerativem Clustering dieses Ziel erreicht. Heute gebräuchliche Methoden zur Zusammenfassung verteilter Daten oder Datenströme erfordern entweder einen Synchronisationsschritt oder kommunizieren und vereinen Aggregate hierarchisch, welches dem Prinzip der flachen Hierarchie in P2P-Systemen widerspricht. Wir testen, ob randomisiertes Senden und Verschmelzen von Aggregaten die gleichen Ergebnisse produziert. Die resultierenden Datenaggregate dienen schließlich zur Bestimmung von Quantilen. Um ein P2P-Overlay-Netzwerk zu erzeugen und seine Infrastruktur instand zu halten ist es notwendig regelmäßig Nachrichten zu senden. Da Bandbreite eine knappe Ressource ist, welche sich das Overlay-Netzwerk mit Anwendungen anderer Schichten teilt, ist es erstrebenswert, die Zahl von Verwaltungsnachrichten so gering wie möglich zu halten. Die untere Schranke für deren Frequenz ist offensichtlich gegeben durch die Abwanderungsrate der Peers. Wir zeigen wie Netzwerkknoten die mittlere Lebensdauer ihrer Nachbarn schätzen können und mit dieser die Nachrichtenfrequenz optimieren ohne die Infrastruktur des P2P-Overlays zu destabilisieren.

Abstract

Peer-to-peer (P2P) systems form a special class of distributed systems. Typically, nodes in a P2P system are flat and share the same responsabilities. In this thesis we focus on three problems that occur in P2P systems: the storage of data replicates, quantile computation on distributed data streams, and churn rate estimation. Data replication is one of the oldest techniques to maintain stored data in a P2P system and to reply to read requests. Applications, which use data replication are distributed databases. They are part of an abstract overlay network and do not see the underlying network topology. The question is how to place a set of data replicates in a distributed system such that response times and failure probabilities become minimal without *a priori* knowledge of the topology of the underlying hardware nodes? We show how to utilize an agglomerative clustering procedure to reach this goal. State-of-the-art algorithms for aggregation of distributed data or data streams require at some point synchronization, or merge data aggregates hierarchically, which does not accompany the basic principle of P2P systems. We test whether randomized communication and merging of data aggregates are able to produce the same results. These data aggregates serve for quantile queries. Constituting and maintaining a P2P overlay network requires frequent message passing. It is a goal to minimize the number of maintenance messages since they consume bandwidth which might be missing for other applications. The lower bound of the frequency for maintenance messages is highly dependent on the churn rate of peers. We show how to estimate the mean lifetime of peers and to reduce the frequency for maintenance messages without destabilizing the infrastructure of the constituting overlay.

Acknowledgements

I would like to thank Dr. Thorsten Schütt for supervising me as a tutor at the ZIB for many years. I would also like to thank Prof. Dr. Katinka Wolter for agreeing to supervise my thesis on short term, and Prof. Dr. Artur Andrzejak for inducting me into distributed systems.

Contents

Chapter 1

Introduction

Would you rather plow a field with
two strong oxen or 1024 chicken?

Seymour Cray

As the costs of the computational power of a processing unit do not scale linearly, distributed systems represent an alternative for applications that are computationally intensive. The computing units of a distributed system are not located in one place, but might be distributed across a building, a town or the whole world. The units do not communicate via a common primary storage, but through sending messages. In the last decades many algorithms, e.g. for sorting, matrix multiplication, or clustering have been rewritten to run partially in parallel.

Peer-to-peer (P2P) systems form a subset of distributed systems. Clients (or nodes) in a P2P system are flat – they all run the same routines and may act as clients or as servers. Most P2P systems have transient peers, which share duties and benefits for a short period of time. Beside the loss of availability, peers might display a large heterogenity concerning computing power, storage space and connection speed. It is an inherent property of P2P systems that peers are a long way away from each other.

Computing data aggregates in a completely decentralized P2P system is much more challenging, since we can not put the task and responsability to a single node, which collects and merges aggregates from all nodes of the network. We must give up the demand of exactness, since P2P networks might increase to arbitrary sizes and clients storing data of interest are transient. Most parallelized algorithms, which compute exact results, require at some point synchronization. For example, in the parallel version of K-means a root node has to be determined and the labeling and re-estimation steps have to be performed synchronously [1]. Even the approximate version of K-means for P2P systems requires synchronization in terms of iteration rounds [4].

Giving up the demand for an exact solution offers the opportunity to try new approaches, which might be simpler, faster, and at the same time exhibit a high degree of robustness, scalability and distribution. In chapter 2 an approximate algorithm for agglomerative clustering is developed which is able to answer questions on the topology of a network. By assigning two-dimensional network coordinates to nodes, we are able to detect locations and sizes of data centers. Furthermore, the same procedure can be used to find locations for storing

a fixed number of replicated objects, such that lookup times and failure probabilities in a distributed hash table (DHT) are minimal.

Another challenge in field of distributed systems or sensor networks is how to compute efficiently quantiles or histograms on data or data streams that are distributed over a network. For example, assume that each node stores the response times for database queries and an application or administrator would like to have an overview of their distribution. The question is how to aggregate distributed data efficiently? We evaluate in chapter 3 a method that computes a histogram-like data summary on each node. These summaries are merged randomly and queried for quantiles. Histograms represent a robust way to reflect data distributions.

In order to maintain the infrastructure of an overlay network, each peer sends regularly messages that control or repair entries of its routing table. These messages consume bandwith which might be lacking for other applications. It is a common goal to minimize their number with the constraint that the infrastructure is kept alive. In chapter 4 we show how to estimate the mean lifetime of peers and adapt the frequency for maintenance messages to achieve this goal.

In general, all methods presented in this thesis can be utilized by higher applications or by the overlay network itself to manage its maintenance and to reduce the need for human administration.

1.1 Gossiping

A common goal of algorithms computing quantitive, administrative information in P2P systems is that each node should end up with the similar amount of information, since in a pure P2P environment nodes must remain replacable. The methods of the first two chapters use the gossiping protocol to spread data aggregates. The central idea of gossiping is that for each client the communication partner is chosen randomly with equal probability from the set of all clients. The randomized manner circumvents many problems concerning net size, unreliability of peers, or an inconvenient structure of the underlying network. Gossiping is the communication principle that accompanies best the idea of P2P systems in terms of equal rights. A gossip-based protocol which forms an *unstructured* overlay network is CYCLON [20].

On top of unstructured overlay networks, we can build *structured* overlays, like Chord, Pastry, or CAN, that may query underlying gossiping routines to establish communication between arbitrary parties with negligible costs.

Chapter 2

Approximate Distributed Clustering over a P2P Network

The technique of clustering is a well-known tool for data analysis. The aim is to group a set of objects X according to their reported features. Objects that show high similarities among each other should end up in the same class. The clustering approach is data driven, we learn clusters only by computations on the data. We control the quality of clusters by prescribing their number or defining a distance dimension.

In P2P systems X can not be accessed at the whole, but is distributed over a network. Each node N^i holds a subset $X^{(i)} \subseteq X$ in its local memory. The data X might be any feature for which a distance dimension can be defined. In this chapter we will concentrate on network coordinates. Thus, we restrict the attribute space of X to $\mathbb{R}^{n \times d}$. The real spatial positions of nodes in a network are unknown, since overlay networks hide the underlying network topology. Algorithms, like *Vivaldi* [6], use the fact that network latencies between nodes in general correspond to spatial distances. Vivaldi was published by Dabek et al. in 2004. The gossip-based algorithm iteratively computes a solution to a spring relaxation problem. Each node sends its own coordinates $y \in \mathbb{R}^2$ and an error estimate $e_y \in \mathbb{R}^+$ to a randomly assigned neighbor along with the round trip time (RRT) rtt. The error estimate reflects the node's confidence about its own coordinates. Depending on the error estimates of both nodes and their euclidean distance in relation to the real RRT, the neighbor adjusts its own two-dimensional network coordinates x. Either in direction to or away from the sending node (see Algorithm 1 below). Initially, the node's coordinates are taken randomly from $\mathbb{R}^2$ and e_x is set to one.

Based on network coordinates, the primary objective is to detect data centers $C \in \mathbb{R}^{k \times d}$ by the use of clustering. But there are many file storing or sharing applications that do not only require the identification of data centers. Potential storage locations should be distributed equally in the communication space, such that transmission latencies and failure probabilities become minimal. Since node failures within a cluster are correlative, replicates should be stored in different clusters. It is also important to identify and avoid outlying nodes for replicate storage, since we probably will receive few requests by nearby nodes. Figure 2.1 shows a number of nodes, that store replicates in an unwanted and wanted manner.

Let k_{algo} refer to the number of replicates to store in a P2P network and k_{data} the real number

Algorithm 1 Vivaldi: Assignment of Network Coordinates.

proc vivaldi(**float** rtt, **float**[ ] y, **float** e_y) $\equiv$
  comment: weight balances local and remote error
  $w := e_x/(e_x + e_y)$
  comment: relative error of incoming sample
  $e_s := \big|\|x - y\| - rtt\big|/rtt$
  comment: update weighted moving average of local error
  $e_x := e_s \times c_e \times w + e_x \times (1 - c_e \times w)$
  comment: update local coordinates
  $\delta := c_c \times w$
  $x := x + \delta \times \big(rtt - \|x - y\|\big) \times u(x - y)$
end

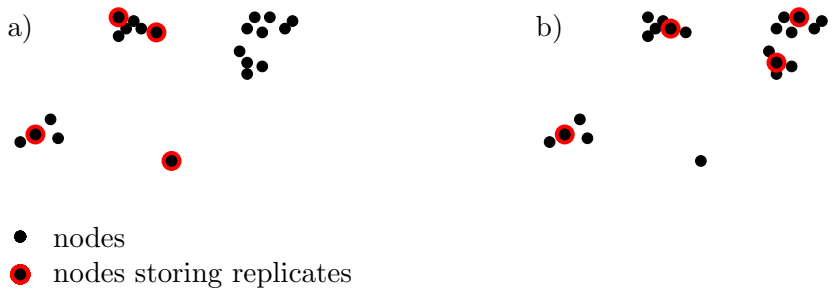


Figure 2.1: Distribution of four replicates in a network. (a) 'Bad' distribution: two replicates are stored in the same cluster and a singleton is used for storage. (b) 'Better' distribution: one replicate per cluster, the outlier is left out.

of clusters, optimal choices for replica placements are summed up in Table 2.1.

$k_{algo} < k_{data}$	$k_{algo} == k_{data}$	$k_{algo} < k_{data}$
Detect clusters and select the k_{algo} largest clusters. Store replicates on nodes that are nearby their centroids.	Detect clusters and store replicates on nodes nearby their centroids.	Detect clusters and if necessary, place replicates into the same cluster such that the storing nodes represent approximately $1/k_{algo}$-th of all nodes.

Table 2.1: Optimal strategies for replica storage.

On a typical data set with 'well' separated and non-concentric clusters, most clustering algorithms will find the exact positions as long as the input k_{algo} matches the latent k_{data}, or some other threshold is set appropriately. But what happens if they differ, like in most use cases for replica storage?

We either have to skip centroids or have to determine centroids that subdivide a single cluster. For the latter case, technically their coordinates do not correspond to centroids any more. Nevertheless, they should be placed as far as possible from each other for two reasons. First, location and breakdown of nodes are correlative, and second, uniform distribution minimizes the overall latency of messages, if we assume to have no prior knowledge about the underlying network. Another reason is that replicates might be too large to be stored on one node, thus, they are distributed over a cluster (or fraction). On the one hand, we need to detect real clusters, on the other hand, we would like to receive positions whose neighborhood represent equal fractions, which do not necessarily correspond to centroid positions. We will later see how to optimize these counteractive goals.

If the demand of exactness is given up, we can give *approximate* versions for agglomerative and K-means clustering for P2P networks. An approximate version of agglomerative clustering was given in my Bachelor thesis [2] in 2009. In the same year Datta et al. proposed an approximate version of K-means (LSP2P K-means) [4]. Both approaches are able to deal with topology changes and loss of data. When using approximate, agglomerative clustering via gossiping, nodes in a network receive a view that converges arbitrarily close to a global one within a few communication steps [3].

For agglomerative clustering the process of data agglomeration is highly unsupervised. There is no control over k, the final number of centroids, nor their relative sizes, nor their separation. In a worst case scenario we might end up with a number of centroids with small distances among each other (but still above a threshold below which we would merge).

For applications that request a fixed number of centroids, K-means had been the first choice, so far. It offers user control over the final number of centroids. Additionally the centroids show a high degree of separation, due to minimization of the squared error between data and centroids over several iterations and disregard of densities.

In this part we extend the approximate, agglomerative clustering algorithm to gain control over k and the degrees of how equally sized and well separated final clusters become. With respect to Datta et al., we will call the new algorithm *2SP2P K-agglomerative clustering*,

since it consists of two stages and is intended for the usage within P2P networks. As we will later see, 2SP2P K-agglomerative clustering is easier to implement, completely decentralized and has lower space and communication costs in comparison to LSP2P K-means.

In general both clustering algorithms work for arbitrary data. But since the improvements of agglomerative clustering aim at cluster localization, we assume that the distributed data set corresponds to network coordinates which are known to each node, e.g. by running a coordinates assigning routine like *Vivaldi*.

One word about exactness, clustering is NP-hard and all *state-of-the-art* clustering approaches are greedy. Thus we have no guarantee for a globally optimal solution. Additionally, we will use an approximate approach to deal with P2P networks.

In the following, $k \in \mathbb{N}^+$ refers to the number of centroids. If necessary, we differ between k_{data}, the latent number of clusters in X, and k_{algo}, the input variable of a clustering algorithm.

2.1 Clustering Approaches

Figure 2.2 shows a taxonomy of clustering techniques for non-distributed systems. Partitional clustering approaches try to optimize the partitioning of a data set without considering former partitions. Data can arbitrarily often switch between different partitions. Hierarchical techniques split or merge consecutively subsets until a desired number of subsets is reached or another termination criterion is fulfilled.

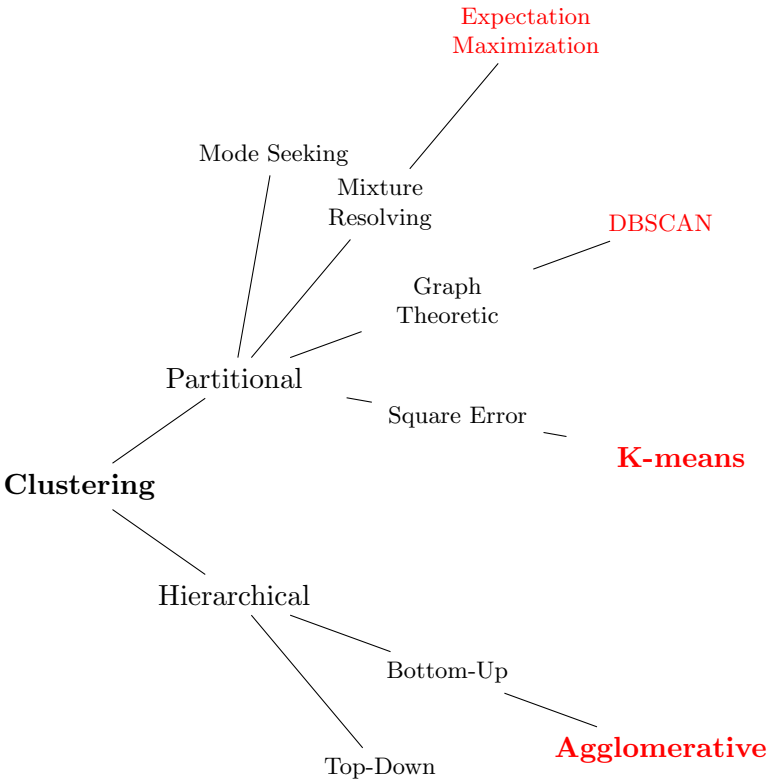


Figure 2.2: A taxonomy of clustering approaches and examples.

2.1.1 K-Means Clustering

One of the most widespread clustering algorithms is K-means. It uses the partitional approach. In detail, K-means partitions the data into k disjoint, exhaustive groups, where k is a user-specified parameter. The goal is to find a partition that minimizes the *squared error* between centroids and data. Initially, k centroids are chosen randomly, their positions are improved iteratively until a termination criterion is met. Each iteration consists of an *expectation* (E-step) and a *maximization* step (M-step). In the E-step data is assigned to their closest centroids. The subsequent M-step recomputes the centroids. For the assignment between data and centroids, a vector of labels has to be stored. Centroids from previous iterations are dropped.

Since K-Means tries to minimize the distances between data and centroids. For most data sets it will never happen that we end up with two centroids that are close neighbors. Loosely

speaking, there will always be enough data between two centroids. This is an important criterion for replica storage locations.

K-means is a greedy algorithm for a NP-hard problem, we might miss the global minimum. By varying the set of initial centroids, we increase the chance of finding centroids that are globally optimal. In practice, if the number of centroids is unknown, K-means is run with different settings for k.

K-means has structural similarities to the Expectation Maximization (EM) algorithm. Instead of labels, indicating the membership to clusters, the EM algorithm computes posterior probabilities[1]. Figure 2.3 shows several iterations of the EM algorithm. The centers of the ellipses represent the means and the radii the deviations of the clusters. The classification is illustrated by the coloring. Later on, we will use K-means only for optimizing the center positions, not their deviations.

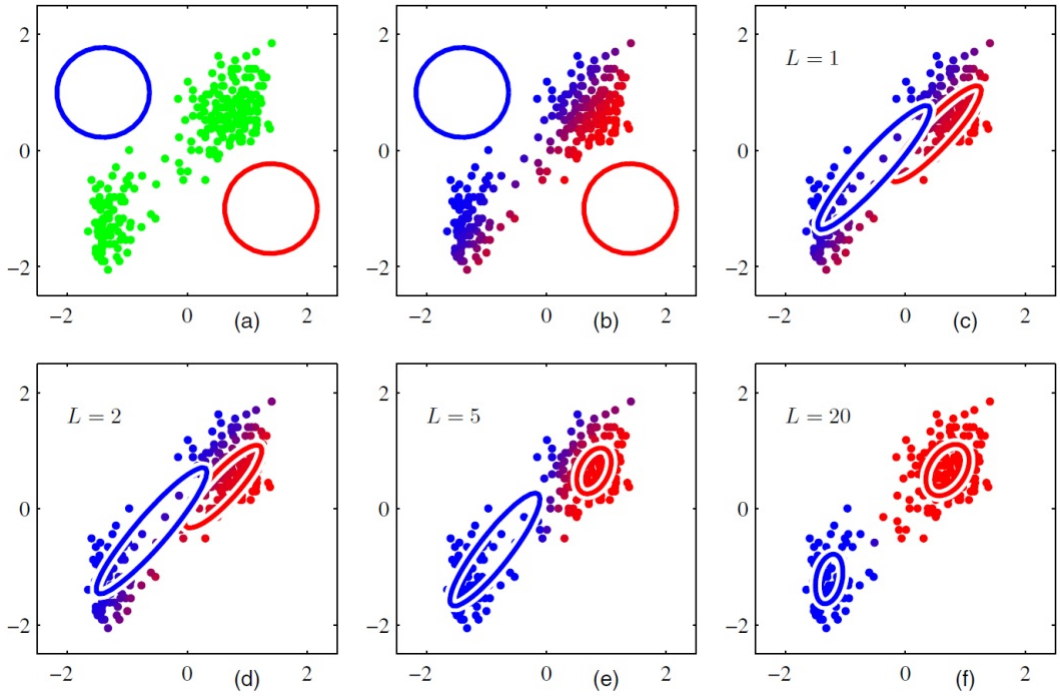


Figure 2.3: Iterations of the EM algorithm. (a) Unclassified data (green) and initial centroids given by their means μ and deviations σ. (b) After E-step, (c)-(f) after 1, 2, 5 and 20 iterations. Taken from Bishop [5].

The pseudocode of K-means is given in Algorithm 2. Note that $[k]$ and $[n]$ are shorthand for $[1..k]$ and $[1..n]$, respectively, and $\xleftarrow{\$}$ refers to a randomized assignment. As a termination criterion the labeling of two subsequent iterations is compared. If no data point changes its membership to a centroid, the algorithm has converged to a locally optimal solution. Another common approach is to compare the squared error between cluster centers and respective

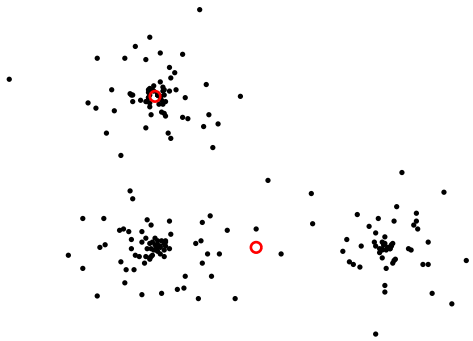
[1] $p(X|\theta)$ – probability of the observed data X given model parameters θ

data, and to stop if the difference of two subsequent errors falls below a threshold γ. This is useful if we have constraints on the number of iterations. Note that one can construct cases for which data items flip arbitrarily often their labels. Therefore it is not recommended to formulate a stopping criterion based solely on labels.

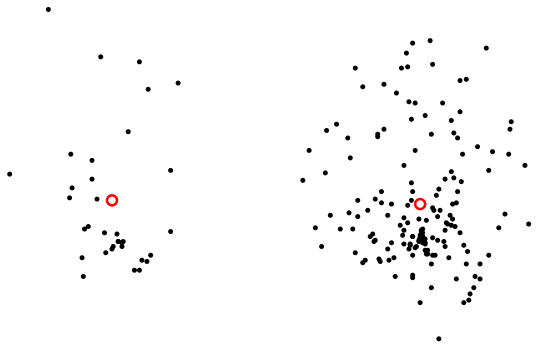
Algorithm 2 K-Means Clustering

proc KMeans(**float**[ ][ ] X, **int** k, **float** γ) $\equiv$
  $n := |X|$
  $c_l \overset{\$}{\leftarrow} X \quad \forall l \in [k]$
  $label_i := \arg\min_{l \in [k]} \|X(i) - c_l\| \quad \forall i \in [n]$
  $c^{old} := c$
  comment: M-step, re-estimate centroids
  $c_l := \frac{1}{|\{i | label_i == l\}|} \sum_{label_i == l} X_i \quad \forall l \in [k]$
  while $\max_j \{\|c_j^{old} - c_j\|\} \geq \gamma$
    $c^{old} := c$
    comment: E-step, compute expected label for each datum
    $label_i := \arg\min_{l \in [k]} \|X_i - c_l\| \quad \forall i \in [n]$
    comment: M-step, re-estimate centroids
    $c_l := \frac{1}{|\{i : label_i == l\}|} \sum_{label_i == l} X_i \quad \forall l \in [k]$
  end
end

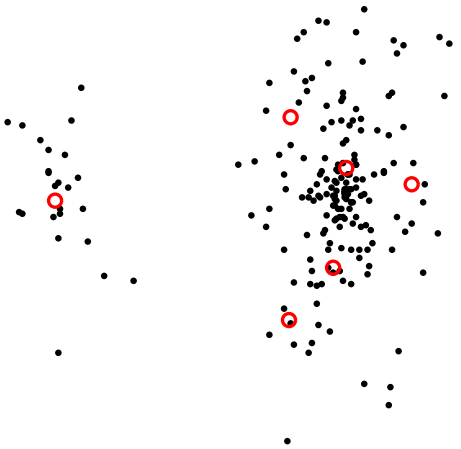
What is the behavior of K-means if the requested number of centroids does not meet the latent number of clusters? Figure 2.4 shows typical results. If the input k_{algo} falls below k_{data}, at least one centroid will be placed between two centroids to minimize the squared error to the data of both clusters. To store a replicate on a nearby node would be a bad choice. For $k_{algo} == k_{data}$, K-means is expected to find the latent centroids. If k_{algo} exceeds k_{data}, K-means places several centroids into one cluster, such that they represent almost equally sized fractions of it.



(a) $k_{algo} < k_{data}$. One centroid is placed in between two clusters with few nodes nearby.



(b) $k_{algo} == k_{data}$. Centroids perfectly meet latent centroids.



(c) $k_{algo} > k_{data}$. Centroids in right cluster are distributed uniformly.

Figure 2.4: Data (black) and centroids (red) estimated by K-means clustering.

2.1.2 Agglomerative Clustering

Agglomerative clustering is a hierarchical clustering technique that works bottom-up. Initially each data point represents a cluster – a singleton. Iteratively the closest clusters are merged. The merging stops as soon as there are no two clusters whose distance falls below some given threshold $\theta \in \mathbb{R}$ (see Algorithm 3 and subroutine 5). The proceeding can be visualized by dendrograms (see Figure 2.5). Branch points correspond to centroids.

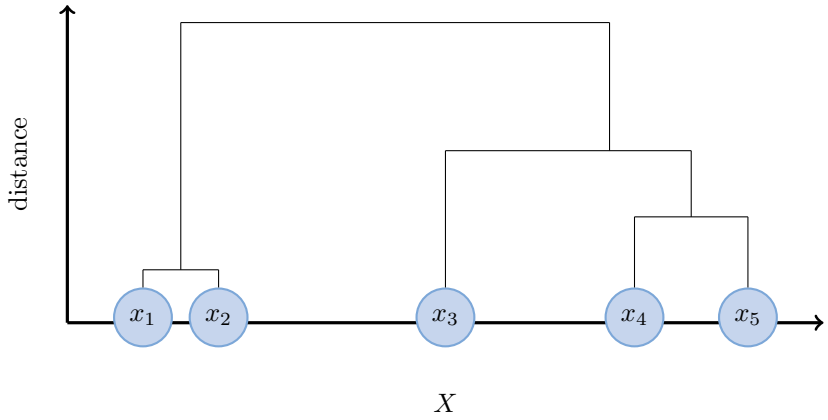


Figure 2.5: Proceeding of agglomerative clustering on 1D data.

There are many ways to define distances between two clusters, for example:

Single-Link distance between the *closest* pair of nodes from different clusters

Complete-Link distance between the *most remote* pair of nodes from different clusters

Centroid method distance between the centroids of two clusters

Looking at the results, they differ in sensitivity for noise or the ability to detect concentric clusters [7]. Note that for the last method which uses only centroids, the algorithm does not need to keep track of data points. For merging and recomputation of new centroids it suffices to store the centroids' coordinates and their relative sizes. This fact is exploited by the approximate version for P2P systems [3]. The pseudocode is given in Algorithm 3.

Another distinction from K-means is that we do not need to give the number of centroids. If θ is chosen properly, agglomerative clustering (from now on θ-agglomerative clustering) is able to detect the natural number of clusters. But what happens if we simply stop after the amount of centroids has become k to receive a particular number of centroids? This first approach is shown in Algorithm 4, the K-agglomerative clustering routine. We only changed the stopping criterion. In the following, we will examine whether K-agglomerative clustering produces meaningful results with respect to the optimal choice of replica storage locations (see Table 2.1).

2.1.2.1 Case Distinction

On a typical data set ('well' separated and non-concentric clusters), with high probability θ-agglomerative clustering will find the exact positions. But what happens, if we force the

Algorithm 3 Agglomerative Clustering

proc θ-AggloClustering(**float**[ ][ ] $data$, **float** θ) $\equiv$
  $n := |data|$
  $C := data$
  $w_i := 1/|data| \quad \forall i \in [n]$
  $(i, j) := \arg\min_{\substack{i,j \\ i \neq j}} \|c_i - c_j\|$
  while $\|c_i - c_j\| < \theta$
    $(C, w) :=$ Merge(C, w, i, j)
    $(i, j) := \arg\min_{\substack{i,j \\ i \neq j}} \|c_i - c_j\|$
  end
  return (C, w)
end

Algorithm 4 K-Agglomerative Clustering

proc KAggloClustering(**float**[ ][ ] $data$, **int** k) $\equiv$
  $n := |data|$
  $C := data$
  $w_i := 1/|data| \quad \forall i \in [n]$
  $(i, j) := \arg\min_{\substack{i,j \\ i \neq j}} \|c_i - c_j\|$
  while $|C| > k$
    $(C, w) :=$ Merge(C, w, i, j)
    $(i, j) := \arg\min_{\substack{i,j \\ i \neq j}} \|c_i - c_j\|$
  end
  return (C, w)
end

Algorithm 5 Weighted Merging of two Centroids

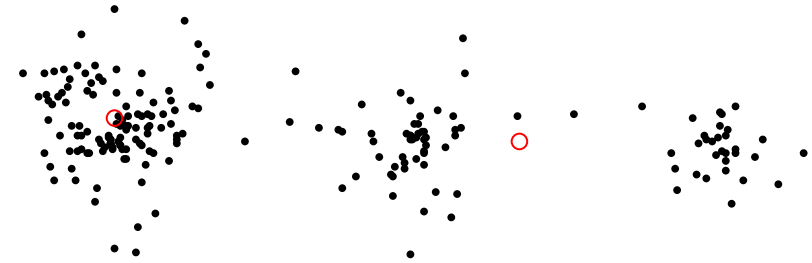
proc Merge(**float**[ ][ ] C, **float**[ ] w, **int** i, **int** j) $\equiv$
  $C := C +\!\!+ [(c_i \cdot w_i + c_j \cdot w_j)/(w_i + w_j)]$
  $w := w +\!\!+ [w_i + w_j]$
  comment: delete original entries
  delete$(C, [i, j])$
  delete$(w, [i, j])$
  return (C, w)
end

agglomerative clustering procedure to stop after the number of centroids reaches k_{algo}? Figure 2.6 shows typical results for different settings for k_{algo} and k_{data}.

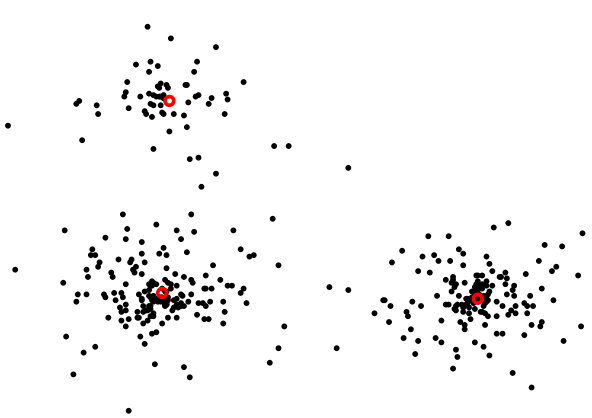
If k is equivalent to the latent number of centroids, we expect to detect latent centroids (see Figure 2.6(b)). The result is the same as if we would have applied θ-agglomerative clustering. If the input k does not meet the latent k_{data}, the results are rather unusable. For k_{algo} less than k_{data}, centroids might be placed between two clusters. They represent an arithmetic mean, although any nodes are nearby. It would make more sense to discard the smaller cluster and to return the centroids of the k largest clusters respectively. We can use the standard procedure of agglomerative clustering to detect latent centroids.

For k_{algo} exceeding k_{data}, we can not avoid to end up with centroids placed close to each other within a single cluster. But they should neither be placed side by side, nor represent only two or three nodes, since location and node failures are correlative. Centroids within one cluster should represent a significant proportion of it, ideally $1/k_{algo}$.

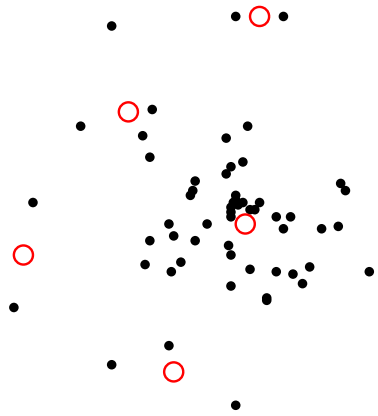
With respect to the optimal criteria to replica placement, K-means provides better partitions for k_{algo} larger than k_{data} (see Figure 2.4(c) on page 10). To overcome the disadvantages of K-agglomerative clustering, we will turn the standard routine into a three-step routine which is triggered by the relationship of k_{algo} and k_{data}. For the estimation of the latent k_{data}, we will use θ-agglomerative clustering and return the k_{algo} largest centroids, if k_{algo} less or



(a) $k_{algo} < k_{data}$: Centroids are placed in between two clusters in few nodes nearby.



(b) $k_{algo} == k_{data}$. Centroids perfectly meet latent centroids.



(c) $k_{algo} > k_{data}$. One centroid represents the latent centroid of the cluster, the others represent small fractions of the outskirts.

Figure 2.6: Data (black) and centroids (red) estimated by K-agglomerative clustering.

equal to k_{data}. The handling of the case $k_{algo} > k_{data}$ will be more challenging. We want to receive several centroids that are placed into a single cluster and represent $1/k$-th of the whole data set. To prevent the standard agglomerative approach to focus solely on dense regions, we will modify the merging criterion. The next section describes a measure against slow agglomeration of nodes in the outskirts.

2.1.2.2 Boost Agglomeration below $1/k$

To force that centroids within single clusters represent equally sized subsets of it, we modify our merging criterion by incorporating relative cluster sizes. Merges that result in new clusters with relative sizes below $1/k$ are favored by adding a weighted sigmoid function with input parameters w, k and 'sharpness' z. The sigmoid function $\sigma : \mathbb{R} \mapsto [0;\ 1]$ with $\sigma(x, z) = \frac{1}{1+e^{-z \cdot x}}$ is a strictly monotonic increasing function with an infimum at 0 and a supremum at 1. Its inflection point is at $(0, 1/2)$. Our input parameter w comes from $[0;\ 1]$, we therefore shift the input right by $1/k$.

The effect is that two candidates for merging are strongly favored if their summed up sizes are below $1/k$. The significance of cluster weights is controlled by the coefficient δ and the sharpness z. Increasing both will raise the chance to end up with equally sized centroids at the expense of higher position errors. The relationship of δ and z, and different kind of qualities, like equally sizing or accuracy, will be examined in section 2.1.4.2.

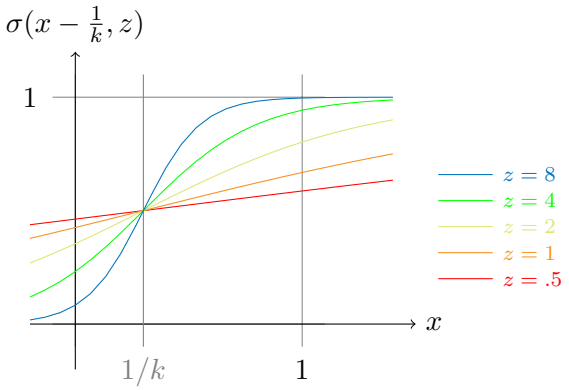


Figure 2.7: Sigmoid function with varying sharpness z.

If k_{algo} is greater than k_{data}, we replace the merge criterion used in θ-agglomerative clustering, namely

$$(\hat{i}, \hat{j}) = \arg\min_{\substack{i,j \\ i<j}} \{\|c_i - c_j\|\} \tag{2.1.1}$$

by a new one that incorporates relative sizes

$$(\hat{i}, \hat{j}) = \arg\min_{\substack{i,j \\ i<j}} \left\{\|c_i - c_j\|/D + \delta \cdot \sigma(w_i + w_j - 1/k, z)\right\} \tag{2.1.2}$$

with D being the diameter – the largest euclidean distance between two nodes of the network.

2.1.2.3 Identifying Singletons

According to our motivation of finding representative nodes for cluster centers, it is not eligible to receive singletons. If outliers are present or k is very large, the agglomerative clustering approach tends to keep outliers as centroids. Why? Due to their large distances to other centroids, they will never be chosen for being merged into another centroid. But singletons can easily be identified by their low relative sizes. Since we do not know how many outliers are present and will survive the clustering procedure, we apply agglomerative clustering and stop early enough to keep a larger set of centroids.

Roughly spoken, we apply k'-agglomerative clustering with $k' > k$ during the distributed computation step. Only on request we reduce the list of centroids by cleaning the centroids from singletons and applying k-agglomerative clustering on the cleaned set. Reducing the list from k' to k centroids can be done instantaneous and does not require communication in a distributed version.

A centroid c_i is regarded as being a singleton, if for its relative size w_i holds $w_i \leq \epsilon$. For the evaluation part ϵ was set to $2/n$, since we might have computing errors on w_i or the estimated net size n. The routine of separating singletons from latent cluster centroids should be executed as late as possible. In early stages current singletons might get absorbed by larger centroids that are not known yet to a single node. On the other hand, if we start identifying singletons on a strongly shrunk set, we might obtain less than k valid centroids.

If we have some prior knowledge about the probability p of nodes being singletons, the latest moment for an outlier removal is when $|c| = k + \lceil pn \rceil$ holds. For the sake of completeness the pseudocode is given in Algorithm 6.

Algorithm 6 Singleton Separation

```
proc SeparateSingletons(float[ ][ ] C, float[ ] w, int k, float ε) ≡
  while |C| > k
        i := arg min_{j∈[|C|]} w_j
        do if w_i > ε then return (C, w, S) fi;
        S := S ∪ {C_i}
        delete(C, [i])
        delete(w, [i])
        i := arg min_{j∈|C|]} w_j
  end
  w := w/(Σ w)
  return (C, w, S)
end
```

2.1.2.4 2SP2P K-agglomerative Clustering

Applying both modifications, we end up with a two-step agglomerative clustering routine. In a first step k'-agglomerative clustering with $k' > k$ is applied on the data. In a distributed, approximate version, a node receives this data indirectly by gossiping its currently estimated centroids (see chapter 2.2.2).

After having received the request to return $k \in [1;\ k']$ centroids, a node executes again θ-agglomerative clustering to detect the latent number of clusters k_{data}. If the requested k does not exceed k_{data}, the k largest centroids are returned. The procedure of separating singletons (see Algorithm 6) is incorporated into θ-agglomerative clustering the same way as below in procedure 2SKAggloClustering (see Algorithm 8 on page 17), but not explicitly written here.

If k exceeds k_{data}, we perform k-agglomerative clustering on the k' centroids and drop the results from θ-agglomerative clustering. We use the new merge criterion to receive centroids representing equally sized partitions of clusters. As soon as $|c| = k + \lceil pn \rceil$ holds, we separate the singletons from the set of centroids. We continue to agglomerate until k' reaches k and we are done.

Algorithm 7 2S K-agglomerative Clustering

```
proc 2SKAggloClustering(float[ ][ ] data, int m, int k, float ε, float δ, float z, float θ) ≡
  (C, w) := 2SKSub1(data, m, k)
  (C, w, S) := 2SKSub2(C, w, k, ε, δ, z, θ)
  return (C, w, S)
end

comment: the 1st subroutine requires access to the data

proc 2SKSub1(float[ ][ ] data, int m, int k) ≡
  C := data
  w[i] := 1/|C|   ∀i ∈ [|data|]
  comment: mk-agglomerative clustering with euclidean distance only
  (C, w) := KAggloClustering(C, w, m · k)
end

comment: the 2nd subroutine for reduction operates only on the centroids

proc 2SKSub2(float[ ][ ] C, float[ ] w, float ε, float δ, float z, float θ) ≡
  (C_latent, w_latent, S) := ThetaAggloClustering(C, w, θ)
  comment: if k_data ≥ k return k largest centroids
  do if |C_latent| ≥ k
       C := C_latent[findKLargest(w_latent, k)]
       w := w_latent[findKLargest(w_latent, k)]
       return (C, w, S)
     fi
     comment: else agglomerate with new merge criterion
     (C, w, S) := KAggloPlusClustering(C, w, k, ε, δ, z)
     return (C, w, S)
  end
```

Algorithm 8 K-agglomerative Clustering with Singleton Separation

proc KAggloPlusClustering(**float**[ ][ ] C, **float**[ ] w, **int** k, **float** ϵ, **float** δ, **float** z) $\equiv$
  $sepFlag := false$
  $S = \{\}$
  while $|C| > k$
    do if $\neg sepFlag \wedge |C| <= k + \lceil p \cdot n \rceil$
      $S :=$ SeperateSingletons(C, w, k, ϵ)
      $sepFlag := true$
    fi
    $(i, j) := \arg\min_{\substack{i,j \\ i \neq j}} \left\{ \|c_i - c_j\|/D + \delta \cdot \sigma(w_i + w_j - 1/k, z) \right\}$
    $(C, w) :=$ Merge(C, w, i, j)
  end
  return (C, w, S)
end

Be aware that we moved the assignment of the initial centroids and size vector outside all clustering procedures. For the standard agglomerative clustering (Algorithm 3) with distance threshold θ we did the same. An example is given below in Algorithm 9. In this way, the core routine will be utilized for the distributed version.

Algorithm 9 K-Agglomerative Clustering with new Merge Criterion and extracted centroid assignment

proc MainClustering(**float**[ ][ ] $data$, **int** k, **float** δ, **float** z) $\equiv$
  $C := data$
  $w_i := 1/|data| \quad \forall i \in$ [data]
  $(C, w) :=$ KAggloClustering(C, w, k) comment: or any other
  output(C, w)
end

comment: clustering without data access
proc KAggloClustering(**float**[ ][ ] C, **float**[ ] w, **int** k, **float** δ, **float** z) $\equiv$
  $(i, j) := \arg\min_{\substack{i,j \\ i \neq j}} \|c_i - c_j\|$
  while $|C| > k$
    $(C, w) :=$ Merge(C, w, i, j)
    $(i, j) := \arg\min_{\substack{i,j \\ i < j}} \left\{ \|c_i - c_j\| + \delta\sigma\big(w_i + w_j - 1 + 1/k, z\big) \right\}$
  end
  return (C, w)
end

2.1.3 Error of Global Clustering

Forcing the agglomerative clustering routine to return $k_{algo} \neq k_{data}$ centroids obviously counteracts the strength of agglomerative clustering – detection of dense regions. We express this by separating the error:

1. **correctness** of centroid **positions**
2. **equally sized** centroids
3. **well-separation** of centroids

According to these qualities we will record three different errors: err_{pos}, err_{eq}, and err_{sep}. $\widehat{c}$ and $\widehat{w}$ denote the estimated values for c and w. For being able to compare estimated and latent values, we apply agglomerative clustering on the larger set, if $|c| \neq |\widehat{c}|$ holds.

We define the position error between real c and estimated centroids $\widehat{c}$ as the product of their euclidean distance and their difference in size. We assume that $\widehat{c_i}$ approximates c_j if the latter minimizes the expression $\|\widehat{c_i} - c_j\| \cdot |\widehat{w_i} - w_j|$. We normalize the error by the diameter D of the network and the number of counted centroids.

$$err_{pos} = \frac{1}{kD} \sum_i \underbrace{\arg\min_j \{\|\widehat{c_i} - c_j\| |\widehat{w_i} - w_j|\}}_{\text{closest centroids with respect to position and size}} \tag{2.1.3}$$

On the other hand, if we are forced to place several centroids into one cluster, we want to reward **equally distributed** cluster centers of **equal sizes** that have rather large distances between each other. Since both qualities might counteract, we separate their errors. err_{eq} measures the discrepancy between estimated cluster sizes and an ideal uniform distribution of nodes over clusters, which is $1/k$.

$$err_{eq} = \underbrace{\frac{1}{k} \sum_i (\widehat{w_i} - 1/k)^2}_{\text{deviation from equally sized clusters}} \tag{2.1.4}$$

For the separation error, we penalize centroids that are close to each other by summing up $(1 - \|c_i - c_j\|/D) \in [0;\ 1]$ for all unique pairs of centroids.

$$err_{sep} = \underbrace{\frac{2}{k(k-1)} \sum_{\substack{ij \\ i<j}} \left(1 - \|c_i - c_j\|/D\right)}_{\text{penalty for low distances between centroids}} \tag{2.1.5}$$

2.1.4 Experimental Setup and Results

2.1.4.1 Data Generation

All of the data is sampled independently from k normal distributions with fixed means $c_{j\in[k]} \in \mathbb{R}^{2\times 1}$ and deviations. The number of centroids k is drawn from the interval $[2;\ 8]$ with relative sizes taken from $[0.2;\ 0.7]$, both distributed uniformly. Since the latent number of centroids k_{data} merely meets the number of centroids k_{algo} requested by an application, both variables are generated independently. They are equal with a probability of $1:7$. In these cases the position errors are expected to be the best. The latent centroids C and sizes wd are stored to compute the errors.

Algorithm 10 Data Generation

proc DataSetGen(**int** n, **float** p) $\equiv$
  comment: choose latent and required k independently
  $k_{data} \xleftarrow{\$} [2;\ 8]$
  $k_{algo} \xleftarrow{\$} [2;\ 8]$
  comment: generate latent centroids plus singletons along a grid
  $C :=$ meshGrid$(k_{data} + \lceil n \cdot p \rceil)$;
  comment: generate k_{data} relative cluster sizes
  $wc_j \xleftarrow{\$} [.05;\ .7] \quad \forall j \in [k_{data}]$
  $wc := wc/(\sum wc)$;
  comment: for each centroid C_j generate $n \cdot wc_j$ distributed normally
  $data \leftarrow \mathcal{N}(\mu = C_j, \sigma^2 = .4) \quad \forall j \in [k_{data}] \forall i \in [wc_j \cdot n]$
  $data := data +\!\!+ C_{k+1:n}$ comment: append singletons
  $wd_i := 1/n \quad \forall i \in [n]$
  return $(C, wc, data, wd, D)$
end

2.1.4.2 Results for δ and z in new merge criterion

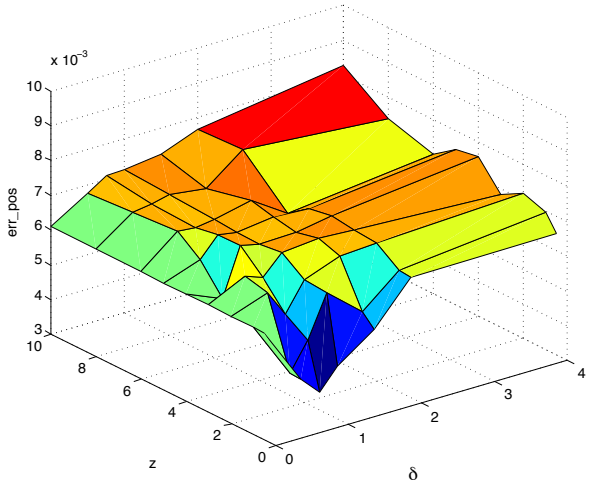
To find optimal settings for the parameters δ and z in the new merge criterion, combinations of settings for the sharpness z and the importance δ were tested (see Figure 2.8).

$$(i,j) := \arg\min_{\substack{i,j \\ i<j}} \Big\{ \|c_i - c_j\| + \delta\sigma\big(w_i + w_j - 1 + 1/k, z\big) \Big\} \qquad \text{(new merge criterion)}$$

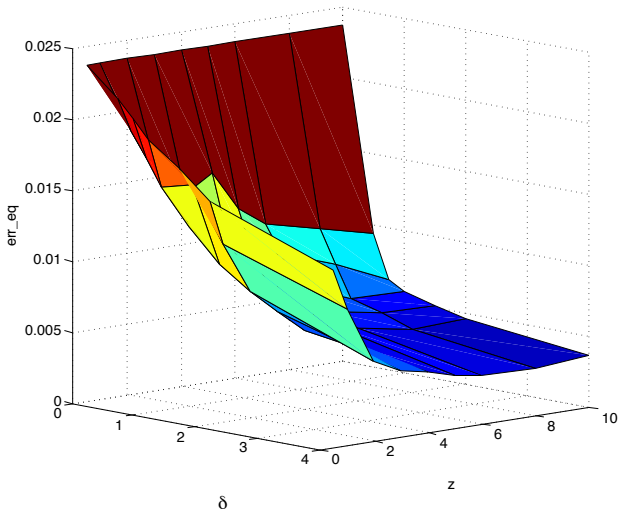
	$\sigma(err_{pos}) \cdot 10^{-3}$	$\sigma(err_{eq}) \cdot 10^{-3}$	$\sigma(err_{sep}) \cdot 10^{-3}$
δ	0.0004	0.0385	0.3472
z	0.0005	0.0158	0.1998

Table 2.2: Deviation for δ and z if other parameter is fixed.

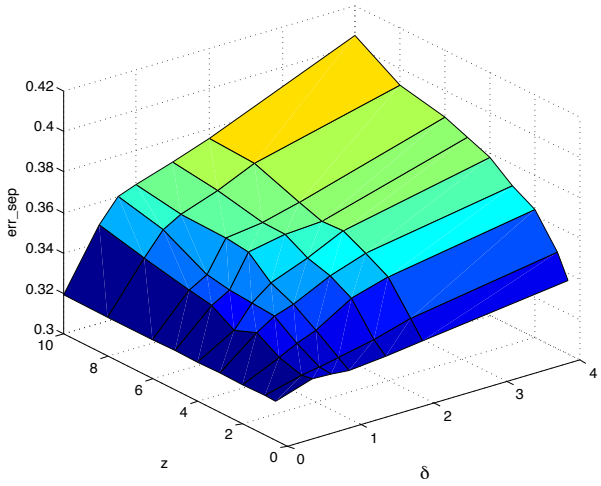
The influence of the second term of the merging criterion, that is $\delta\sigma(w_i + w_j - 1/k, z)$, becomes larger with the increasing of δ and z. Increasing δ and z will worsen the position



(a) err_pos, $min : \delta = .75, z = .5$



(b) err_eq, $min : \delta = 2, z = 8$



(c) err_sep, $min : \delta = 0, z = \emptyset$

Figure 2.8: δ versus z with $\delta \in [0, .5, .75, 1, 1.5, 2, 4]$ and $z \in [.5, 1 - 6, 8, 10]$

and separation errors. On the other hand, we will improve the equality error by more than four times. Apparently the control parameters δ and z are correlated with respect to the errors. They always should be increased or decreased together.

Table 2.2 shows the deviations of δ and z[2]. The influence of δ is slightly larger for the equality and separation errors. With regard to their correlation, one should consolidate both into a single formula with one input parameter $u \in [0;\ 4]$ and optimal coefficient vector β derived from linear regression analysis.

$$f(u) := \begin{pmatrix} (\delta_{min} + u) \cdot \beta_1 \\ (z_{min} + u) \cdot \beta_2 \end{pmatrix} = \begin{pmatrix} \delta \\ z \end{pmatrix} \tag{2.1.6}$$

2.1.4.3 1S versus 3S Agglomerative Clustering

To test the effect of the introduction of the case distinction (relation of k_{algo} and k_{data}), these configurations where tested against each other:

T1 K-agglomerative clustering with old merge criterion

$$(\hat{i}, \hat{j}) = \arg\min_{\substack{i,j \\ i<j}} \|c_i - c_j\|$$

T2 K-agglomerative clustering with new merge criterion and input parameters δ, z

$$(\hat{i}, \hat{j}) = \arg\min_{\substack{i,j \\ i<j}} \left\{ \|c_i - c_j\|/D + \delta \cdot \sigma(w_i + w_j - 1/k, z) \right\}$$

T3 3S K-agglomerative clustering with case distinction and input parameters δ, z

T4 K-means clustering

T2 and T3 use the new merging criterion, thus they were run with three parameter settings for δ and z. Table 2.3 shows the errors.

error	**K-agglo-merative** (old crit.)	**K-agglomerative**			**3S K-agglomerative**			**K-means**
		$\delta = .75$ $z = .5$	$\delta = 1$ $z = 4$	$\delta = 2$ $z = 8$	$\delta = .75$ $z = .5$	$\delta = 1$ $z = 4$	$\delta = 2$ $z = 8$	
pos	1.0000	0.8521	0.5706	0.8331	0.1359	0.3953	0.4790	0.8672
eq	1.0000	0.7582	0.5292	0.1151	0.5949	0.2653	0.1887	0.3129
sep	0.4663	0.5201	0.6567	0.9192	0.8220	0.9515	1.0000	0.8878

Table 2.3: Position, equality and separation errors for 1S/3S K-agglomerative clustering with three parameter sets and K-means, rows are divided by their maxima.

Position error. The position error is the best for 3S K-agglomerative clustering which returns the real centroids if $k_{algo} \leq k_{data}$ (in 57% of all test cases). In these cases we expect tiny position errors between estimated and latent centroids. Within 3S K-agglomerative

[2] more precise: mean of deviations, averaged for all settings of the fixed parameter

clustering the position error decreases if the second term of the merging criterion becomes less important (low values for δ and z). As we saw, K-agglomerative and K-means produce large position errors for $k_{algo} < k_{data}$, since they place at least one centroid between two or more clusters. This increases their overall position error.

Equality error. The deviation of centroid sizes from $1/k$ is the highest for the standard K-agglomerative clustering and very low for K-means. This is because K-means minimizes squared errors between centroids and assigned nodes. The squared error is lower if centroids subdivide data equally. Whereas K-agglomerative clustering simply concentrates on regions with high densities. Outskirts are neglected, since they contain fewer nodes that show low distances. This phenomenon is damped by amplifying the second term in the merge criterion: $\delta \cdot \sigma(w_i + w_j - 1/k, z)$. The higher δ and z are set, the more we force merging in sparse regions. With the incorporation of relative sizes, K-agglomerative clustering even outperforms K-means.

Separation error. The standard agglomerative clustering algorithm which exclusively considers position errors outperforms all the other approaches. Agglomerative clustering ensures that no centroids survive that are close to each other. Out of all approaches this seems to be the best strategy to maximize the overall distances *between* clusters. Looking at K-agglomerative and 3S K-agglomerative using the new merge criterion, the separation of centroids is reduced with the gain of δ and z. It is the worst for 3S K-agglomerative clustering. One reason is that for the case $k_{algo} \leq k_{data}$ it returns the k_{algo} largest centroids no matter how big or small their distances between each others are.

2.2 Adaption to P2P Systems: Approximate Clustering

After having built an algorithm for a global instance having access to all data, we now switch over to P2P systems. Algorithms that strictly follow the P2P paradigm are completely decentralized. There are no supervising instances. All participating computational units or nodes are equally privileged. In real live networks, one has to cope with nodes that emerge or disappear at any moment. If we compute statistics that incorporate data from single nodes, we must ensure on the one hand that the influence of failing nodes fades away, and on the other hand, that data of newly emerging nodes will quickly spread over the network. It's hard to design exact clustering algorithms for non-static networks, and, most important, these solutions are not scalable. As soon as data changes or a node joins, all nodes would have to be updated. Beside, all exact clustering algorithms require to parse several times the whole data set.

More appropriate are economic algorithms which are robust concerning data and topology changes and that are fast. The approximate versions of agglomerative clustering and K-means exchange locally estimated centroids and parse only local data. With every iteration or communication step estimated centroids get closer to the global ones.

2.2.1 Approximate K-Means Clustering

An approximate version of K-means for P2P systems has been proposed by Datta et al. in 2009 [4]. A node N^i stores two kinds of centroids: locally estimated centroids with corresponding cluster counts $C^i = \{(c^i_j, w^i_j) : 1 \leq j \leq k\}$ and centroids $V^i = \{v^i_j : 1 \leq j \leq k\}$ that are a weighted average of local and remote centroids. The E-step (labeling) is always done with respect to V^i. The centroids and counts derived from the subsequent M-step (re-estimation according to the labeling) are stored in C^i.

Algorithm 11 on page 24 is a simplified version from their paper. Periodically a node sends poll messages to a fixed set of neighbors Γ. After all polled neighbors have responded with their centroids and cluster counts, the node will update its centroid set V^i.

Polls that can not be answered yet are stored in a poll table. As soon as a new iteration is passed, namely, new centroids and cluster counts have been computed, the poll table is processed for requests related to the current iteration. A node can only process centroids from neighbors that are or have already passed the same iteration. In a P2P network nodes might be in any iteration state. Thus nodes have to wait for requested data from their neighbors, and have to store historical data in order to be able to reply requests of nodes that resides in lower iteration states.

The authors built in a switch, namely, nodes whose centroids don not change significantly can change from an 'active' state to a 'terminated' state. Terminated nodes will not send any polls or update V, but will respond to polls with their latest centroids and cluster counts. If data changes or a node is added to whom a terminated node is neighboring, the node has to check for becoming active again. For the sake of simplicity the state handling is left out in the pseudocode.

During implementation and testing of this approximated version of K-means for P2P systems, I noticed one problem. If the local data is not sufficiently large or not mixed (local data is

Figure 2.9: Framework for Approximate K-Means Clustering in P2P Systems

Algorithm 11 LSP2P K-Means

```
proc Initialize(C_1^0, w_1^0) from N_0 ≡
  (C_1, w_1) := (C_1^0, w_1^0)
  historyTable := [(C_1, w_1)]
  Wait := {}
  pollTable := [ ]
end
proc Timer ≡
  for N^j ∈ Γ do
    sendTo N^j : Poll(i, l)
  od
end
proc Poll(k, l̂) from N^k ≡
  if l̂ ≤ l then
        sendTo N^k : PollResp(historyTable[l̂])
      else
        pollTable := pollTable +++ [(k, l̂)]
  fi
end
proc PollResp(C_l^k, w_l^k) from N^k ≡
  Wait := Wait ∪ {(C_l^k, w_l^k)}
  if |Wait| == |Γ| then
        v_{j,l+1} := (Σ_{N_k∈Wait} c_{j,l}^k w_{j,l}^k) / (Σ_{N^k∈Wait} w_{j,l}^k)
        Wait := {}
        if max{‖v_{j,l} − v_{j,l+1}‖ : 1 ≤ j ≤ K} > γ then Iterate
        fi
  fi
end
proc Iterate ≡
  l := l + 1
  labels := E-Step(data, V)
  (C_l, w_l) := M-Step(data, labels)
  for (k_m, l) ∈ pollTable do
    sendTo N^{k_m} : PollResp(C_l, w_l)
  od
end
```

derived from less than k_{data} clusters), the carrying out of one iteration with respect to V^i might result in centroids that have a cluster count of zero. Thus the M-step drops the corresponding centroids if we do not implement a special behavior. The authors of the paper tested their framework in a simulation of up to 2,000 nodes and 130,000 to 1.3 million 10-dimensional data points (at least 65 data points per node). They set the number of clusters k to 8 for all tests. The chance of having no members from a cluster is very low. Thus, this exception did probably not occur in their setting.

A common critique on K-means is its non-robustness for outliers. The worst scenario is if an

outlier is chosen as one of the initial centroids by chance. Its position will not change, since no data will be assigned in the E-step. The outlier will survive every iteration. Even if an outlier has not been chosen as an initial centroid, it attracts its closest centroid and therefore skews its coordinates. This effect is strengthened if the local data is not sufficiently large. Therefore it is common to preprocess data and to remove outliers before applying K-means.

2.2.1.1 Behavior in a Dynamic Environment

In a dynamic P2P environment nodes might become unavailable due to node failures or topology changes. In order to cope with these challenges, the authors Datta et al. propose this behavior for LSP2P K-means:

Node failure or topology change. Each node N^i having a failing node N^j as a neighbor (detected by monitoring Γ^i), has to stop to wait or send messages from or to N^j by removing it from its $Wait^i$ and $pollTable$ lists.

Node addition. The newly joined node N^i inherits the centroids and cluster counts of the neighbor with the lowest iteration number. N^i starts iterating as described in Algorithm 11. Each terminated node having N^i as a neighbor, has to test whether it becomes active again. It sends a poll message, receives centroids and counts of N^i, and carries out one iteration of K-means. If the centroids changed significantly, the neighboring node becomes active again. Its immediate neighbors now have to do the same test.

Data change. If the node is still active, nothing must be done. If it has entered the terminated state, the node recomputes (C_l^i, w_l^i) and polls its immediate neighbors. If after the update of V^i and subsequent recomputation of (C_{l+1}^i, w_{l+1}^i) the centroids have changed significantly, the node becomes active again. N^i sends a data change message to its neighbors. If they are in the terminated state, they have to determine whether to become active using the same technique as described in the 'Node addition' case.

2.2.2 Approximate Agglomerative Clustering

We now apply the same approach for approximate agglomerative clustering presented in my Bachelor thesis for 3S K-agglomerative clustering from section 2.1.2.4. Initially (or if the local data changes), a node assumes its local data to be the centroids with equal relative sizes. Periodically the `Timer` function is called and the node becomes active. To a randomly selected peer from the network it sends its currently estimated centroids and sizes. Upon receipt of remote centroids and sizes, another node sends its own, but calls `ShuffleResp` which only computes the update. Otherwise the message passing would not stop. Subsequently local and remote centroids and sizes are concatenated and the `Update` procedure is called which performs mk-agglomerative clustering. Only if requested, the number of centroids is reduced further and singletons are separated in the subroutine `3SKAgglo` given by Algorithm 7 on page 16.

Figure 2.10: Framework for Approximate Clustering in P2P Systems

Algorithm 12 2SP2P K-agglomerative Clustering

proc Initialize $\equiv$
  $(C, w) := ([self.data], \frac{1}{|self.data|}[1.0]_{|self.data|})$
end
proc Timer $\equiv$
  $peer :=$ SelectRandomPeer()
  sendTo $peer$: Shuffle(C, w)
end
proc Shuffle(C_{rmt}, w_{rmt}) **from** p $\equiv$
  sendTo p : ShuffleResponse(C, w)
  $(C, w) :=$ Update$(C +\!\!+ C_{rmt}, w +\!\!+ w_{rmt})$
end
proc ShuffleResp(C_{rmt}, w_{rmt}) **from** p $\equiv$
  $(C, w) :=$ Update$(C +\!\!+ C_{rmt}, w +\!\!+ w_{rmt})$
end
proc Update $\equiv$
  $(C, w) :=$ KAgglomerative$(C, w, m \cdot k)$
  $(C, w) :=$ Normalize(C, w)
end
proc Request **from** p $\equiv$
  $(C_{req}, w_{req}, S) :=$ 2SKSub2$(C, w, k, \epsilon, \delta, z, \theta)$
  sendTo p : RequestResponse(C_{req}, w_{req})
end

2.2.2.1 Behavior in a Dynamic Environment

We now propose how nodes should behave on typical events in a P2P environment:

Node failure or topology change. Since the 2SP2P K-agglomerative clustering is designed to rely on a node independent gossiping routine, a node just needs to wait until a new peer is assigned in the next time slice.

Node addition. Any newly joined node first calls the `Initialize` procedure. After a few data exchanges the node will have similar centroids like its peers.

Data change. Local data is only read in the initialization step. Thus we drop the current centroids and call the `Initialize` procedure from Algorithm 12 as if the node had just been added to the network.

To save computational resources one could introduce an active and a terminated state like in LSP2P K-means. Since nodes do not store immediate neighbors, a node does not know whether an assigned peer has just joined the network. Thus, nodes would have to store and send their age in terms of iterations or passed time. If the age falls below a threshold, an assigned peer determines the new centroids and decides whether to become active (same behavior as described in section 2.2.1.1).

2.2.3 Error of Distributed Approximate Clustering

In section 2.1.4.3 we examined for the global versions of K-means and 3S K-agglomerative clustering how the position, equality and separation errors changed. We now have to show that 2SP2P K-agglomerative clustering converges towards the solution of the global algorithm. For comparison LSP2P K-means clustering has been implemented also.

2.2.4 Experimental Setup and Results

The data for the LSP2P K-means and 2SP2P K-agglomerative clustering was generated the same way as described in section 2.1.4.1. But to accommodate LSP2P K-means, no outliers had been interspersed and a node stores not a single data point, but at least $\max\{k_{data}\} = 8$ data points. Again the data is assumed to be partitioned disjunctively and exhaustively over the network. Instead of the weaker assumption that only a couple of neighbors are available to whom a node sends poll messages, we assume that we can connect to any neighbor from the whole network. A node independent service provides a gossiping partner chosen randomly from the whole network. In one time slice each node carries out successfully one data exchange with one gossiping partner. Therefore, we do not need to handle history or poll tables for LSP2P K-means. We assume a static environment – no node addition or change of data after the network has been established.

As a convergence criterion we consider the position error. For each round k and data were generated newly and distributed over N node structures establishing a network. The network was duplicated to run LSP2P K-means and 2SP2P K-agglomerative clustering with the same prerequisites. A node structure stores local data, centroids, counts and other algorithm specific parameters. After each time slice the position errors between local $C^i := \{c_l^i : 1 \leq l \leq k\}$ and global centroids $C := \{c_j : 1 \leq j \leq k\}$ for all nodes were averaged. Global centroids for LSP2P K-means were received from global K-means, and global centroids for 2SP2P K-means from an application of 3S K-agglomerative clustering. The evaluation was run 100 times. The position error after the t-th gossiping iteration is:

$$err_{pos}^{P2P}(t) \quad = \quad \frac{1}{R}\sum_{r=1}^{R}\frac{1}{N}\sum_{i=1}^{N}\frac{1}{kD}\sum_{l=1}^{k}\arg\min_{j}\{\|c_l^i(t) - c_j(t)\| \cdot |w_l^i(t) - w_j(t)|\}$$

2.2.4.1 Results

The results of err_{pos}^{P2P} and $t \in [1;\ 25]$ are given in Table 2.4 and Figure 2.11.

t	1	2	3	4	5	6	7	8	10	20
err_{pos}^{KAgglo}	0096	.0064	.0041	.0026	.0018	.0014	.0013	.0012	.0013	.0015
err_{pos}^{KMeans}	.0186	.0159	.0148	.0141	.0138	.0134	.0133	.0131	.0127	.0122

Table 2.4: Position errors for 2SP2P K-agglomerative clustering and LSP2P K-means

The error plot in Figure 2.11 shows that 2SP2P clustering converges relatively fast. After 6 exchanges with random peers, the error does not improve further. The gossiping routine was implemented as a *pull* gossiping, which means, that a node polls for remote centroids, but

does not send them. LSP2P K-means clustering has a lower speed of convergence and the error remains higher.

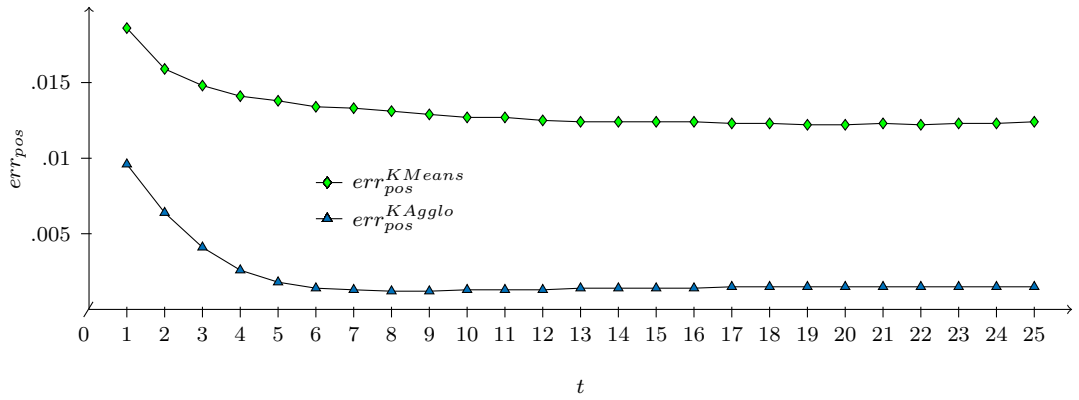


Figure 2.11: Position errors for 2SP2P K-agglomerative clustering and LSP2P K-means with $|data^i| = 8 \quad \forall i \in [n]$.

2.3 Discussion

The results of section 2.2.4.1 show that 2SP2P K-agglomerative clustering quickly converges toward the global solution in a static network. We conclude that the approximated version has the same properties as 3S K-agglomerative clustering, its global counterpart. We observe too, that LSP2P K-means works as a gossiped version, which does not poll a fixed neighborhood, but a single random peer from the whole network. Table 2.5 outlines the features of 2SP2P K-agglomerative clustering and LSP2P K-means. We will now discuss them in more detail.

Simplicity. First, one notices that the framework of 2SP2P K-agglomerative clustering is much simpler than the one of LSP2P K-means. The centerpiece is the same – the diffusion of locally estimated centroids. LSP2P K-means immediately averages them, whereas 2SP2P K-agglomerative clustering concatenates first and merges only if two centroids are close enough.

LSP2P K-means:

$$v_{j,l+1}^i = \frac{\sum_{N_k \in Wait^i} c_{j,l}^k n_{j,l}^k}{\sum_{N_k \in Wait^i} n_{j,l}^k}$$

2SP2P K-agglomerative:

$$\begin{aligned} c &= c +\!\!+ c_{rem} \\ w &= w +\!\!+ w_{rem} \\ c &= c \backslash \{c_i, c_j\} \cup \frac{c_i w_i + c_j w_j}{w_i + w_j} \quad \text{if } c_i, c_j \text{ being close enough} \end{aligned}$$

For the next iteration LSP2P K-means does not forward the weighted average V^i, but the centroids and counts carried out after one iteration. Hence, V^i is moved into direction of the local data. Whereas 2SP2P K-agglomerative clustering simply forwards C^i and averages cluster counts w^i. Hence, centroids are spread with higher speed, which can be seen by the fast convergence of 2SP2P K-agglomerative clustering. After approximately 5 exchanges the locally estimated centroids are the same as the global ones.

Communication costs. Let I denotes the maximum number of iterations carried out and L the maximum number of neighbors to whom a node will send its centroids and counts. In each iteration a node sends at most k centroids to L neighbors. The communication costs

for both approaches are $\mathcal{O}(kIL)$.

Computational costs. Let D denotes the maximum size of a data set stored at a node. For initialization the local data set is reduced from D data points to mk centroids. This requires $\mathcal{O}(D^2)$ comparisons of pairwise distances. This is done only once. During the communication phase mk remote centroids are concatenated with the local ones. The set of centroids is reduced from $2mk$ to mk by comparing pairwise distances which is the most expensive part. For applications asking for k centroids, the set is reduced further until $\mathsf{c} == k$. The number of comparisons is $\sum_{i=k}^{2mk} i = \sum_{i=1}^{2mk} i - \sum_{i=1}^{k-1} i = \frac{(2mk+1)2mk}{2} - \frac{(k-1)k}{2} = 2m^2k^2 + mk - k^2/2 + k/2 = \mathcal{O}(m^2k^2)$ per node. For the whole lifetime of a node we finally have $\mathcal{O}(Im^2k^2 + D^2)$ computations. The computational costs for LSP2P K-means depend on the size of the local data set. The averaging of local and remote centroids is cheap: $\mathcal{O}(k)$. The subsequent K-means iterations requires to parse the data at most two times. One time for labeling – each data point is assigned to its closest centroid and centroid re-estimation. For LSP2P K-means we end up with $\mathcal{O}(IDk)$ per node. Concerning data set size D, 2SP2P clustering is expensive when initialization is executed and LSP2P K-means does not scale in the 'active' state when centroids are re-estimated.

Space costs. 2SP2P K-agglomerative clustering does not partially synchronize like LSP2P K-means. Hence, no counters, history or poll tables have to be stored. But we need a buffer to separate singletons and being flexible for an arbitrarily requested k. We store during the computational phase $k \cdot m$ centroids and for requests additionally a reduced set of size k. Whereas LSP2P K-means needs to store history tables for nodes residing in older stages and poll tables for requesting nodes. $\mathcal{O}(kI)$ are the costs for storing a history table and $\mathcal{O}(IL)$ the space costs for a poll table [4].

Convergence. The experimental evaluations above show (at least) empirical convergence. In the paper of Datta et al. [4] analytical bounds on the accuracy for the distributed K-means clustering algorithm are shown. For the proof the authors only had to assume that communicating parties are sampled uniformly at random. This corresponds exactly to the testing conditions under which we sampled peers randomly from the whole network instead of taking a fixed neighborhood.

Properties	**2SP2P K-agglomerative**	**LSP2P K-means**
synchronization of iterations required	no	yes
minimal number of data points per node required	no	yes
k must be fixed during computing phase	no	yes
order of centroids and counts must be kept fixed	no	yes
robustness for outliers	yes	no
communication costs	$\mathcal{O}(kIL)$	$\mathcal{O}(kIL)$
computational costs	$\mathcal{O}(Im^2k^2 + D^2)$	$\mathcal{O}(IkD)$
memory costs	during exchange phase: $\Omega(mk)$	history and poll table: $\mathcal{O}(I(k+L))$
quality depends on sampling service	yes	yes
quality depends on degree of entropy	no	yes

Table 2.5: comparison of 2SP2P K-agglomerative clustering and LSP2P K-means

Pros. 2SP2P K-agglomerative clustering returns either the natural centroids or places several equally sized centroids into a single cluster similar to K-means. But K-means fails to detect all latent centroids if k is less than k_{data} and might return centroids that reside between two clusters. The reason for keeping the centroid set size large was to have a buffer for removing singletons. The buffer has a second advantage – applications are able to request arbitrary k's[3] as opposed to K-means. K-means has no other chance than repeating the whole computations for a new input k.

2SP2P K-agglomerative clustering was intended to estimate coordinates of centroids. Therefore, it is able to deal with data sets of size one or even zero. As long as the list of concatenated centroids does not exceed mk or k, respectively, the node just accumulates data. LSP2P K-means, as presented by Datta et al., is intended to work on large data sets distributed over a P2P network. The algorithm has to be modified to handle nodes storing data sets with sizes less than k.

Cons. We need some estimates over the network to set variables like θ, the upper bound for standard agglomerative clustering or ϵ the upper bound for singleton sizes. For appropriate settings of θ we need an estimate of the standard deviations of clusters, and for ϵ an estimator of the system size.

[3]We simply replace mk in the clustering routine by $mk_{max}+pn$, where k_{max} is an upper bound for requested values of k, p an estimate for nodes being a singleton, and n the estimate for the net size

Chapter 3

Quantiles and Histograms on Distributed Streams

In large, unstructured data sets it is difficult to detect trends or patterns. There is a need for data summaries in terms of quantiles or histograms. Quantiles characterize distributions in ways that are less sensitive to outliers than simpler approaches as the mean and variance [10]. In many settings data is distributed across multiple sources, like servers in a web application or measuring devices in a sensor network. One would like to analyze the performance of websites or distributed applications, or summarize the distribution of measured values in a sensor network.

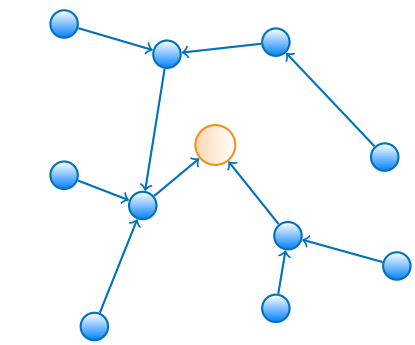
Due to space and temporal limitations, it is impractical to set up a supervising unit which collects all the data and computes a quantile or histogram on it. Instead, local summaries are computed and merged into a final structure. Currently all known approaches for quantile computation on distributed data streams are not distributed itself. Data summaries are merged along a routing tree (see Figure 3.1(a) [14] or collected by a single node which combines all nodes' summaries to finally compute quantiles [16].

A histogram can be constructed from a series of quantiles or reverse quantiles (namely ranks). It can be used for answering different kinds of queries such as range queries, since we only need to determine the set of buckets which lie within the user specified ranges [9].

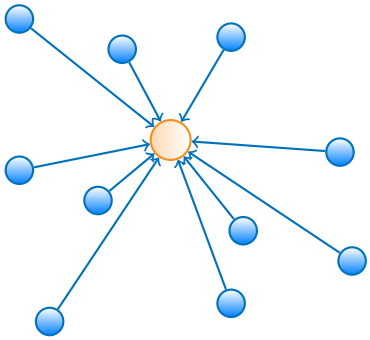
In this chapter we will examine whether quantile summaries can be distributed in a gossiping manner without losing upper bounds of the error. We evaluate experimentally the *q-digest* presented by Shrivastava et. al in 2004 [14]. A q-digest is a tree-like summary of an input data stream. Q-digests can be merged, compressed, and need only constant memory size. It can be used to answer queries for quantiles, ranks or histograms.

3.1 Quantiles

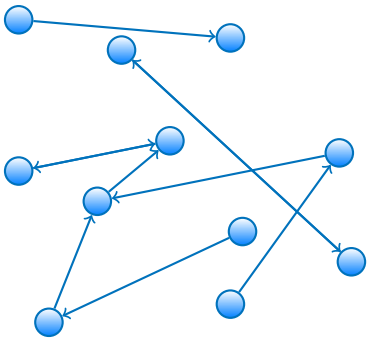
Given a set of n observations $S = \{x_1, x_2, ..., x_n\}$ the ϕ-quantile of a distribution is defined as the value x below which ϕn elements of S lie. In other words, the ϕ-quantile is the value whose rank is ϕn in the ordered set. For example, the .5-quantile is the median of a set, and the 0- and 1-quantile the minimum and the maximum of a set.

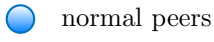


(a) Routing tree. Data is summarized by parent nodes along the route.



(b) Super-peer. A single node collects data summaries from all nodes of the network.





(c) Gossiping. Each node communicates with a randomly chosen neighbor.

Figure 3.1: Strategies of communicating information in a network.

A straightforward method to compute a ϕ-quantile is to sort the data in increasing order and to return the ϕnth element. This implies the storage of all observations $X := \{x_1, x_2, ..., x_n\}$ and to pass the data more than once due to sorting. For determining a given quantile one can do better.

The rank determination belongs to the group of selection problems. Munro and Patterson [12] gave lower bounds on the selection and sorting problem with limited storage. They showed that if an algorithm passes the input data p times, it needs to store at least $n^{1/p}$ items.

Theorem 1. *Any p-pass algorithm to solve the selection problem on a stream of n elements requires* $\Omega(n^{1/p})$ *space.*

Munro and Patterson described an algorithm that almost achieves this lower space bound. A left and a right filter are maintained between which the candidate elements lie. In every pass the gap between the filters is tightened until a small group of candidates remain. In a final pass the ϕ-quantile is selected.

A consequence of theorem 1 is that if we have space limitations that are typically much smaller than n, any p-pass algorithm solving the sort and selection problem and storing less than $n^{1/p}$ items is merely approximate.

3.1.1 Frequency Distributions and Histograms

The cumulative distribution function *cdf* describes the probability that a random variable X with a given probability distribution is less or equal than a given $x \in X$.

$$cdf_X(x) = P(X \leq x)$$

Quantiles are related to frequency distributions and histograms in the following way. The *cdf* is the inverse of the ϕ-quantile (see Figure 3.2).

Histograms are a way to summarize data. They divide the value range into buckets or bins and store for each bucket a counter – the number of elements within the bucket. Depending on the type of bins used, there are two kinds of histograms: *equi-width* and *equi-probable* (or *equi-depth*) histograms. For equi-width histograms the range is subdivided into equally sized bins. The equidistant bin edges are queried to compute the heights.

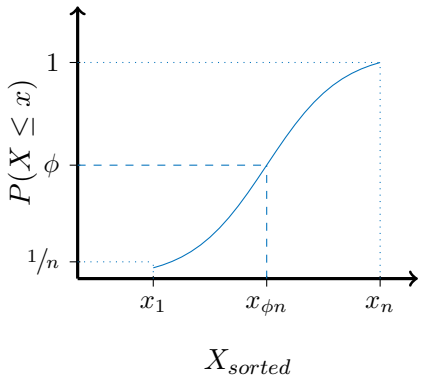


Figure 3.2: ϕ-quantile on a cumulative distribution function of normally distributed X.

The key source of inaccuracy in the use of histograms is that the distribution of data points within a bucket is not retained. If the bucket edges are fixed, like for equi-width histograms, we might lose precision if the bin width is chosen to big or the data deviation is very small. Otherwise some bins contain no or very few elements if the bin width is chosen to be too small.

It is less erroneous to store equi-probable buckets, where the bucket widths are flexible and it is guaranteed that no bucket represents 'too many' (inaccuracy!) or 'too few' (space limitation!) items.

When histograms are combined, the absolute error of equi-width histograms grows with the number of observations, whereas for equi-probable histograms we can guarantee an error reduction, if we simply increase the number of bins (shown by Piatetsky-Shapiro and Connel [13]). The structure that is experimentally analyzed and presented in section 3.2 is similar to an equi-probable histogram, except that the bins are overlapping.

A structure that answers ϕ-quantiles and ranks, can be used to compose both kinds of histograms. An equi-probable histogram can be built from a series of quantiles (see Figure 3.7 and Algorithm 16), and an equi-width histogram from a series of *inverse quantiles* (see Algorithm 17). Given an observation, the inverse quantile is its relative rank in the sorted sequence.

3.1.2 Data Streams and Loss of Information

If the input data is a stream, we have $n \to \infty$ on the one hand, and a memory space that is strongly limited on the other hand, like in a typical sensor network environment. Thus, the approach described in 3.1 is impractical, we are not able to parse the input data several times. Algorithms on data streams are therefore one-pass algorithms – each data point is scanned once, and if not explicitly stored, it is irretrievably lost. Since we can not store all seen data, we will definitely lose information.

In the following, we do not use n for the complete data set size (which is unknown in a data stream), but for the number of input items that are buffered locally beside the computed data summary or statistics. Generally, a data summary can be computed either on the whole data stream or on recent items. Recent items can be modeled by two *sliding window* types. The *count-based sliding window* which includes the last n items seen, and the *time-based sliding window* which stores all items arriving within the time window [16]. The count-based sliding window can be seen as a special case of the time-based sliding window with one item arriving at a time.

As concluded from Theorem 1, any single-pass algorithm for quantile estimation, storing a fixed amount of data, can only be approximate. To get an impression of how big the error can get if no information is given about the data distributions connected to each sensor, an example is given. Assume there are three sensors A, B, and C. Sensors A and B know the exact median of their own data sets X_A and X_B of a fixed size n – say med_A and med_B. Now, sensor C queries the medians of A and B, and calculates a median estimate by taking the maximum (or minimum) of med_A and med_B. No matter of how med_A and med_B are combined, in the worst case the estimated median is $n/2$ positions afar from its real location in the whole data set $X_C := X_A \cup X_B$ (see figure 3.3). The problem is that we do not know the rank of med_A in X_B and vice versa. Thus, communicating quantiles without additional information does not work. The best one can do is to compute data summaries that are small enough to be sent over a network and which can be utilized to compute quantiles and other queries on it.

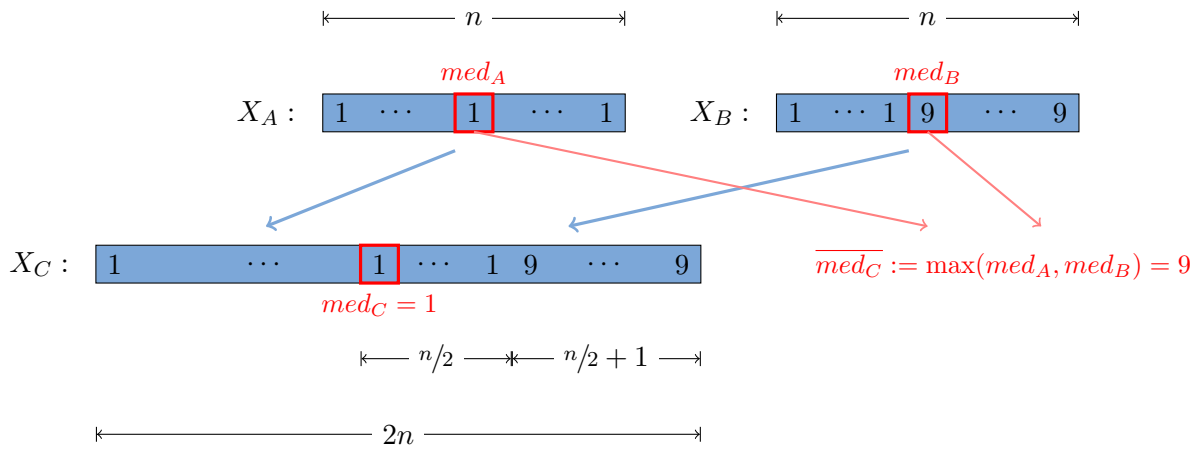


Figure 3.3: Combination of Medians. (left) exact: merge, sort and find position $\lfloor n \rfloor$ in $X_A \cup X_B$, (right) approximate: maximum of med_A and med_B, for (right) the error corresponds to a right shift of the median of about $n/2$ positions

3.2 Quantile Digest

A way to summarize observations is to compute a quantile digest (q-digest) which can incorporate arbitrary many observations. A q-digest is a binary tree whose nodes represent overlapping and almost equi-probable bins of a histogram with a value range from 1 to σ. Given the decompression parameter k, at most $3k$ nodes of the complete binary tree are stored. The variable *count* represents the bucket height, the value range is given by the left and the right most leaves in a complete binary tree.

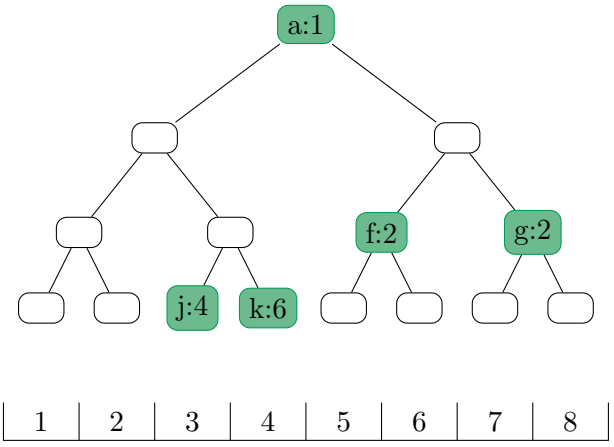


Figure 3.4: Q-digest: Complete binary tree on range $[1..\sigma = 8]$. Only green nodes are stored. Node a represents a bin covering the whole range with one item, node f covers the range 5-6 and contains two items.

Algorithm 13 Compression – establishing the q-digest property

```
proc Compress (Q, n, k) ≡
  l := log_2 σ
  while l > 0 do
      for v in level l do
          if v.count + v_s.count + v_p.count ≤ ⌊n/k⌋
            v_p.count := v_p.count + v.count + v_s.count
            Q := Q\{v, v_s}
          fi
      od
  od
  return Q
end
```

3.2.1 Construction and Compression

A q-digest holds almost equi-probable bins, by requiring that all subtrees satisfy the *q-digest property*:

$$v.count \leq \lfloor n/k \rfloor \tag{3.2.1}$$

$$v.count + v_p.count + v_s.count > \lfloor n/k \rfloor \tag{3.2.2}$$

where v_p is the parent and v_s the sibling of node v. All inner nodes must not incorporate more than $1/k$-th of all n observations. At the same time, we force inner nodes to represent at least $1/k$-th of all observations *together with their sibling and parent count*. Excluded are the root and the leaf nodes. The root node is allowed to violate against property 3.2.2 and the leaf nodes against property 3.2.1. The reason is, that the counter of the root can not be compressed further, while for leaf nodes, with relatively high counts, it is not necessary to compress them. Nodes with count zero are not stored.

For initially building a q-digest, a simple histogram $H = \{(i, f_i) : 1 \leq i \leq \sigma\}$ on a series of n observations is built. The observations and their frequency counts constitute the leaf nodes of a q-digest. In the bottom-up procedure of Algorithm 13, nodes violating property 3.2.2 are removed together with their sibling and their counts are added to the parent node. As a result, bins representing only a 'small' fraction of observations are merged and will never be reconstituted again. With each merge the precision of the resulting node gets bisected.

3.2.2 Merging

Q-digests can be merged such that they represent the input data of many sensors. This is done by adding the counts of a all nodes (see Figure 3.6). Afterwards, the q-digest property is reconstituted by applying the compress procedure.

The maximum error in count of any node is $\frac{\log \sigma}{k} \cdot n$. This relative maximum error does not

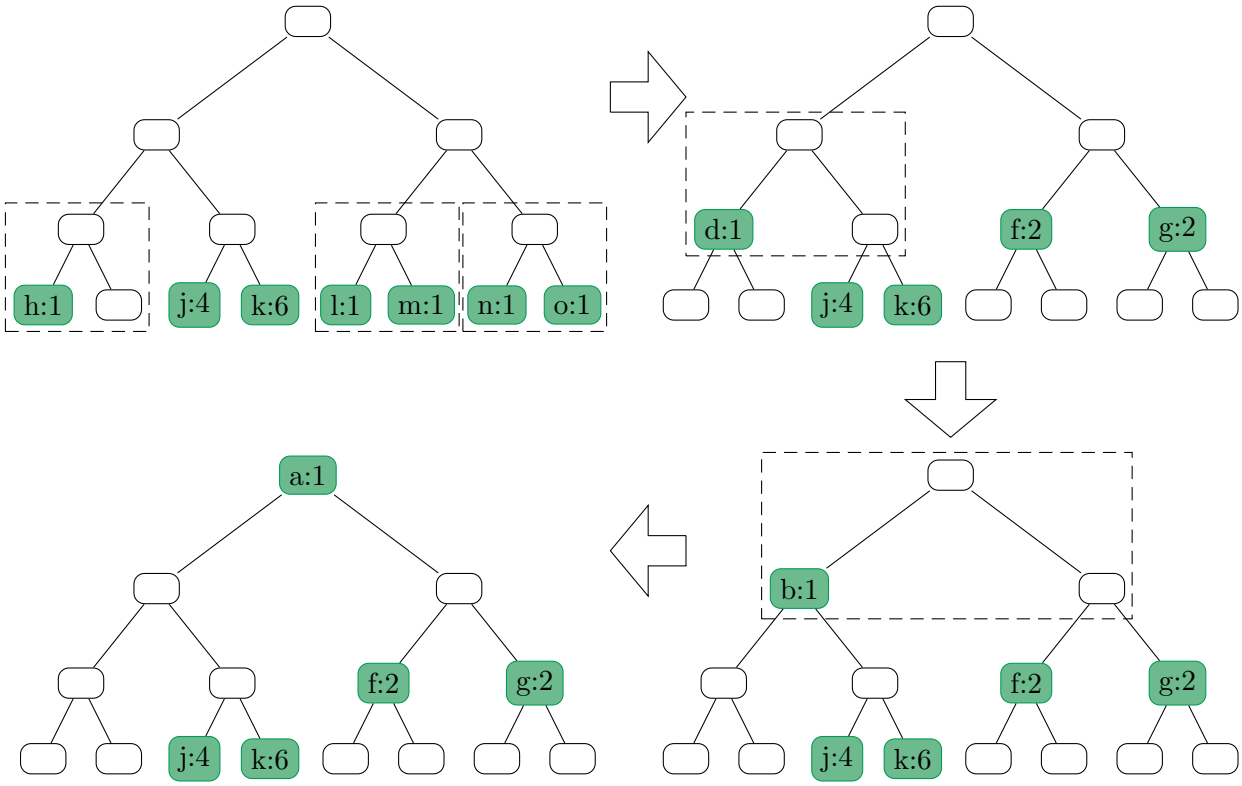


Figure 3.5: Compression. $n = 15$, $k = 5$, $\sigma = 8$, parts violating the q-digest property are marked by a dashed box.

Algorithm 14 Combining two q-digests

```
proc Merge (qdigest Q_1, int n_1, qdigest Q_2, int n_2, int k) ≡
  Q := Q_1 ∪ Q_2
  Compress(Q, n_1 + n_2, k)
  return Q
end
```

increase if q-digests are merged. Proof:

$$error(v) \leq \sum_i error(v_i) \leq \sum \frac{\log \sigma}{k} n_i \tag{3.2.3}$$

$$= \frac{\log \sigma}{k} \sum_i n_i = \frac{\log \sigma}{k} n \tag{3.2.4}$$

Intuitively it should be clear, that if two q-digest are merged and compressed again, the precision can not be better than the precision of the worst q-digest, since at most $3k$ nodes of the q-digest now represent $n_1 + n_2$ data items. Moreover, merging several q-digests into a single one, results in absolute node errors of at most $\frac{\log \sigma}{k} \sum_i n_i$ (see section 3.4 in [14]).

Concerning space complexity, we have:

Lemma 1. *A q-digest constructed with compression parameter k has a size of at most 3k (nodes).*

3.2.3 Quantile Computation

Given a q-digest and $\phi \in [0, 1]$, the goal is to determine a leaf node whose predecessors represent at least ϕn items in a postorder traversal. The source of inaccuracy are inner nodes, whose counts are distributed arbitrarily between subjacent leaf nodes. The postorder traversal sorts the nodes according to their right endpoints. Nodes with smaller ranges appear first. The counts are summed up until the expected position ϕn is reached. One of the subjacent leaf nodes of the current node corresponds to the correct ϕ-quantile. Since there might be more counts of subsequent postordered nodes than we have summed up so far, the rightmost leaf is returned (see Algorithm 15). To keep in mind, only leaf nodes represent single bin values.

3.2.4 Equi-probable Histogram Computation

Histograms are composed of series of quantiles or ranks. For an equi-probable histogram of step size τ, the ϕ-quantiles for $\phi \in [0 : \tau : 1]$ are computed. Between two neighboring quantiles lie τn elements. Figure 3.7 shows an equi-probable histogram with ϕ assigned to the x-axis. All bins have the same width and a height corresponding to the quantile value from the ordered set of observations.

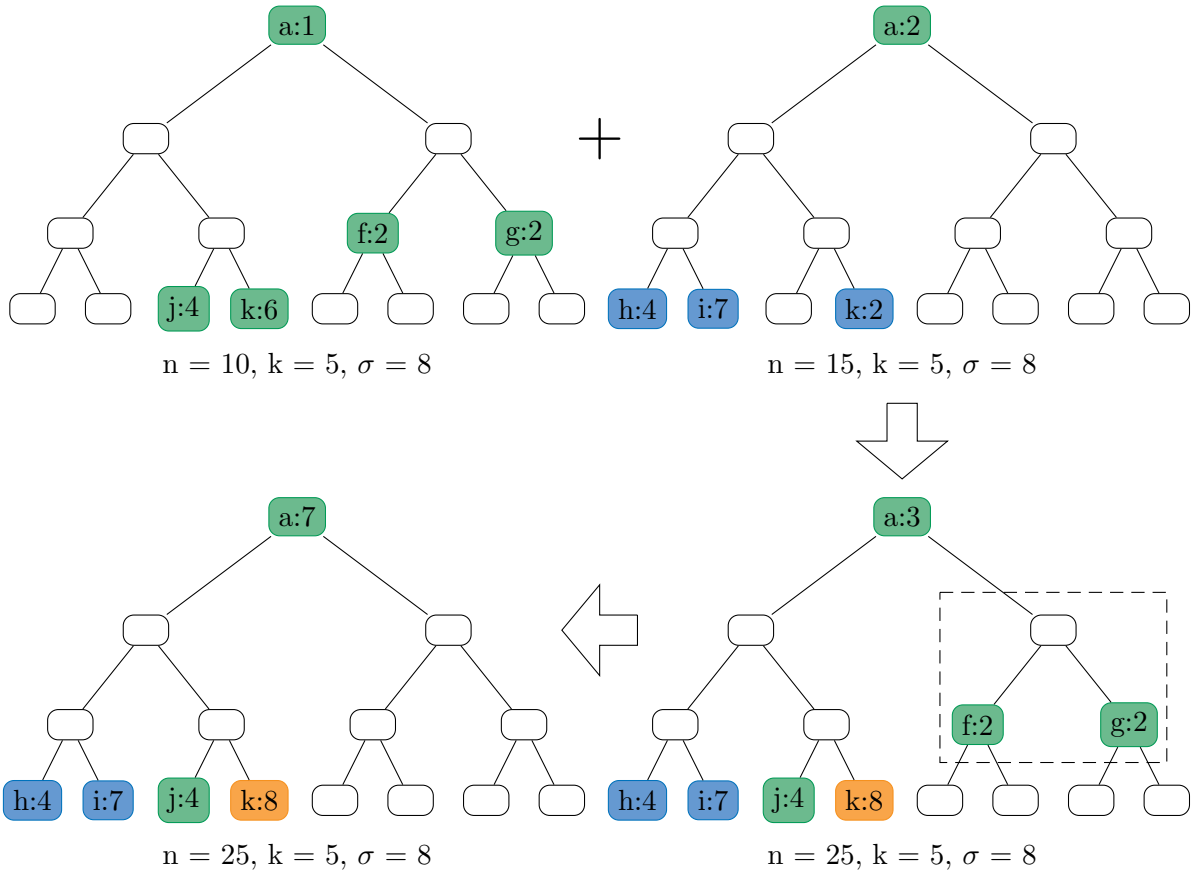


Figure 3.6: Merging two q-digests. First the counts are added (bottom right), second the q-digest property is reconstituted.

Algorithm 15 Quantile computation on a q-digest

```
proc quantile (qdigest Q, int σ) ≡
  L := postorder(Q.tree, σ)
  s := 0
  for v ∈ L do
    s := s + v.count
    comment: expected position reached?
    if s >= p · Q.n
      return rightLeaf(v, σ)
    fi
  end
  return rightLeaf(L.end, σ)
end
```

Algorithm 16 Computing an equi-probable histogram

```
proc histogramEquiProb (qdigest Q, int σ, float τ) ≡
  φ := [0 : τ : 1]
  comment: determine 1/τ + 1 quantiles
  q_i := quantile(Q, φ_i)   ∀i ∈ [1..|φ|]
  return q
end
```

3.2.5 Equi-width Histogram Computation

In order to compute an equi-width histogram with a given bin width on the range $[1..\sigma]$, we proceed as follows: for each bin we query the ranks for the upper and lower edges. The rank difference gives us the number of elements that lie in the bin and corresponds to the bin height (see Algorithm 17).

3.2.6 Distributed Combination of Q-Digests

A framework for gossiping q-digests in a P2P system is given below. In this version each call of `Initialize` resets the current q-digest to a new one, computed on the local data set. First, the leaf nodes of the bottom level are constituted. They form an exact histogram of the input data. The leaf nodes are finally compressed into a q-digest. With every call of the `Timer`, a randomly selected neighbor from the network receives the local q-digest and its count, and sends its own q-digest back (see `Shuffle`). To implement a push version, `Shuffle` must not send the local q-digest to the sender, and for a pull version the input arguments of `Shuffle` must be void.

The `Request` function could be a wrapper for any statistical query that has been discussed

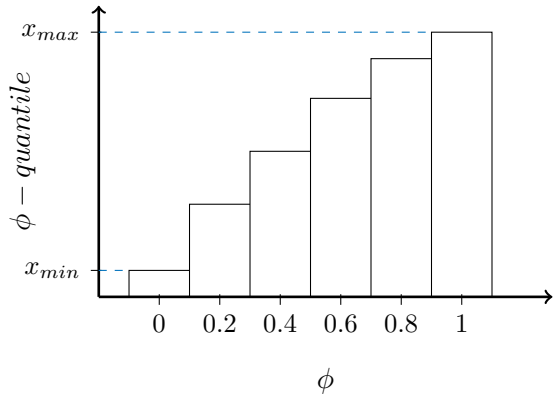


Figure 3.7: Equi-probable histogram from 7 ϕ-quantile queries with $\phi \in [0 : .2 : 1]$. Quantiles are assigned to the axis of ordinates.

Algorithm 17 Computing an equi-width histogram

```
proc histogramEquiWidth (qdigest Q, int σ, int bwidth) ≡
  b := [1 : bwidth : σ + 1]
  comment: determine ranks of all bin edges
  invQ_i := inverseQuantile(Q, b_i)   ∀i ∈ [1..|b|]
  h_i := invQ_{i+1} − invQ_i   ∀i ∈ [1..|invQ| − 1]
  return h
where
proc inverseQuantile(qdigest Q, int σ, int x) ≡
  comment: list of nodes in postorder
  L := postorder(Q.tree)
  rank := 1
  for i = 1 : length(L)
      comment: add node count if x is not reached yet
      if rightLeaf(L(i, 1), σ) >= x then exit fi
      rank := rank + L(i, 2)
  end
  return rank
end
```

so far – single quantiles, ranks, equi-width or equi-probable histograms. For the subsequent experimental evaluation, equi-probable histograms were computed.

3.3 Evaluation

To measure the quality of q-digests, a series of quantile queries for $\phi \in [0 : 0.1 : 1]$ was computed and compared to the exact approach[1]. Let $Q_j := \{q_1, q_2, ..., q_m\}$ be a set of

[1]By sorting the whole data set and selecting the positions $\phi \cdot n$ for all settings of ϕ.

Figure 3.8: Framework for Gossiped Merging of Q-Digests in P2P Systems

Algorithm 18 Distributed, Gossiped Merging of Q-Digests

```
proc Initialize(data, k, σ) ≡
  leaves := buildtree(data, σ);
  n := |data|
  QDigest := Compress(leaves, n, k)
end
proc Timer ≡
  peer := SelectRandomPeer()
  sendTo peer : Shuffle(QDigest, n)
end
proc Shuffle(QDigests_rmt, n_rmt) from p ≡
  sendTo p : ShuffleResponse(QDigest)
  QDigest := Merge(QDigest, n, QDigest_rmt, n_rmt)
end
proc ShuffleResp(QDigest_rmt, n_rmt) from p ≡
  (QDigest, n) := Merge(QDigest, n, QDigest_rmt, n_rmt)
end
proc Request(τ) from p ≡
  H := histogramEquiProb(QDigest, σ, τ)
  sendTo p : RequestResponse(H)
end
```

quantiles. The error between two quantile sets (or equi-probable histograms), representing the same quantile queries, is defined as the squared, average difference between two corresponding quantiles relative to their range σ.

$$e_{hist}(Q_1, Q_2) \quad := \quad \frac{1}{\sigma^2 m} \sum (q_{1,i} - q_{2,i})^2 \tag{3.3.1}$$

The data was generated randomly according to a normal distribution $\mathcal{N}(\mu, \sigma_{\mathcal{N}})$. The mean μ and deviation $\sigma_{\mathcal{N}}$ were taken randomly with equal probabilities from the intervals $[\sigma/4, 3\sigma/4]$, and $[1, \sigma/4]$, respectively. Either all sensor data followed a common normal distribution or the distribution parameters were tossed separately.

3.3.1 Compression Error

The size of a q-digest in terms of number of nodes can be controlled by the parameter k. The q-digest property ensures that at most $3k$ nodes are stored. Thus, if k is decreased, the tree gets more compressed and quantile queries are less precise, because node counts are associated to wider value ranges. Plot 3.9 shows the effect of varying k.

The results are the average of 100 repetitions. For each repeat the newly generated data set had the size 1024. In comparison to $k = 5$, the error for $k = 20$ is reduced by two third. But setting k to 40 in comparison to 20, does not provide a smaller error.

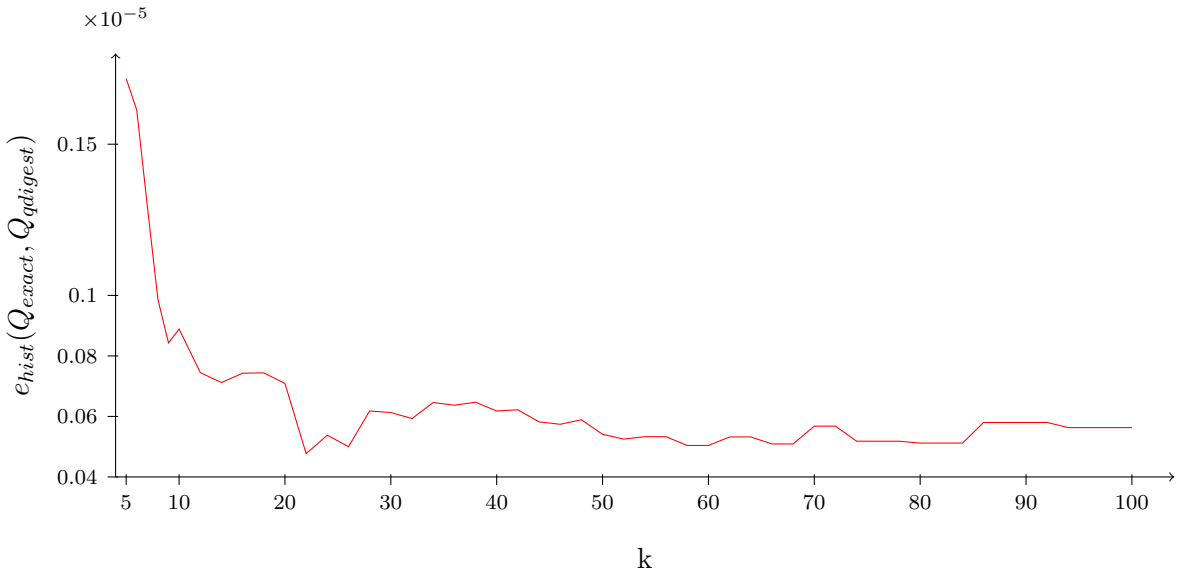


Figure 3.9: Error between exact equi-probable histogram and histogram computed on a single q-digest. $\mu \in [\sigma/4, 3/4\sigma]$, $\sigma_{\mathcal{N}} \in [1, \sigma/4]$, $\phi \in [0 : .1 : 1]$.

3.3.2 Gossiping versus Routing

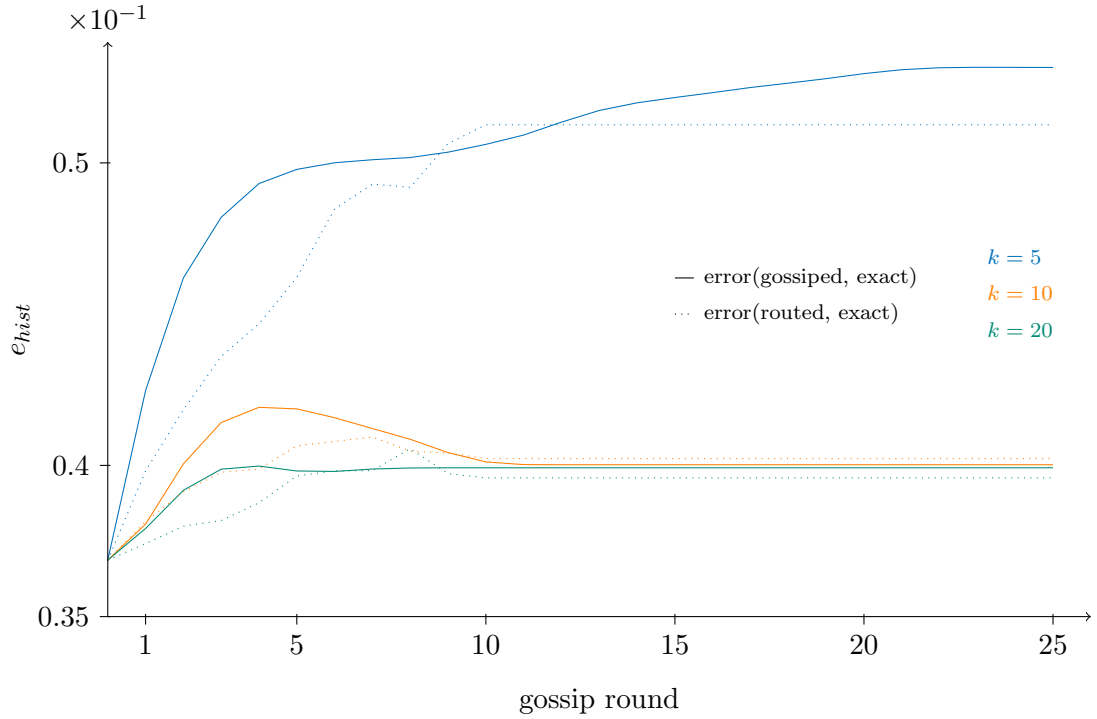
For the subsequent tests, a network of 1024 sensors was simulated. Each sensor was equipped with a buffer to store the last 1024 observations. Given the compression factor k, the buffer content was converted into a q-digest and cleared. Two different strategies of distributing q-digests are compared – unsupervised gossiping and deterministic routing.

For gossiping in each round a gossiping partner was chosen randomly from the whole network for each node and queried for its q-digest (see Algorithm 18). The querying node then merged its own q-digest with the remote q-digest according to Algorithm 14. The resulting q-digest replaced the original one. This corresponds to a pull gossiping, which is expected to have a lower convergence rate than push-pull gossiping.

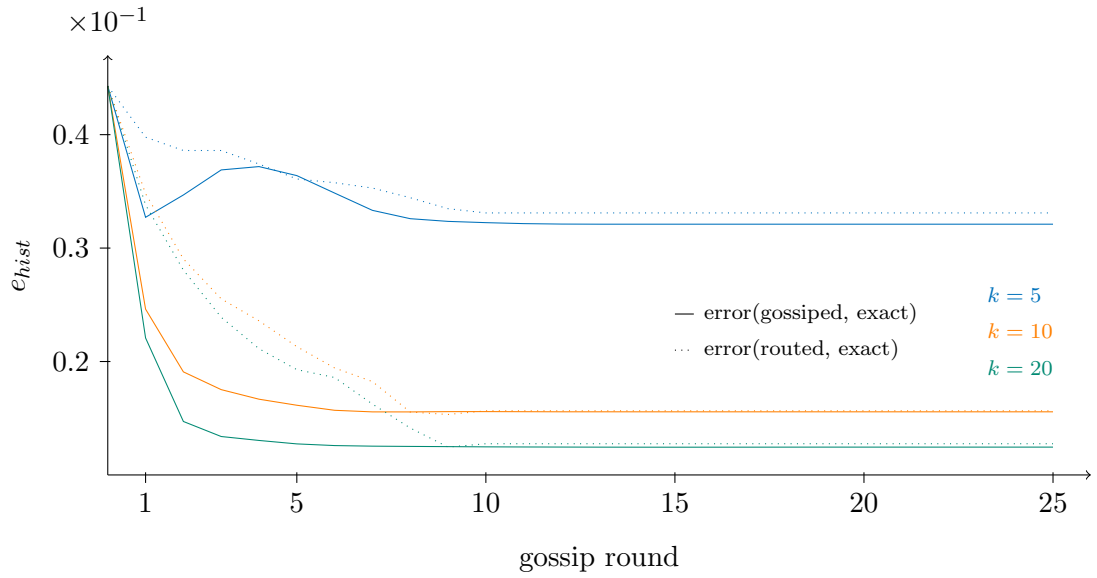
To simulate routing, a random routing tree was generated as shown in Figure 3.1(a). Q-digests had been merged bottom-up to a root node. It was assumed that currently updated nodes send their q-digest to their parents simultaneously. The quantile error for each stage was computed on currently updated nodes, the root node and all nodes in between. In the last phase the root node holds a q-digest representing the compressed data of all nodes' buffers.

For the first test, all local data sets followed a common normal distribution (Figure 3.10(a)). In the second test, mean and deviation had been drawn individually for each sensor (Figure 3.10(b)).

We observe, that if the distribution of the local data sets follows a common global distribution, each sensor has a summary that represents the global data set relatively well without any communication (round = 0). The subsequent exchange and merging of q-digests slightly worsens the error. Especially for strong compressed q-digests ($k = 5$). The quantile error converges late (after 20 rounds) for strong compressed and quickly for low compressed q-digests.



(a) One normal distribution for all sensor data.



(b) Different normal distributions for each sensor.

Figure 3.10: Error between exact quantiles $\phi = [0 : .1 : 1]$ and quantiles computed on q-digests for different compression parameters $k = [5, 10, 20]$. For comparison the error between exact quantiles and quantiles computed on q-digests of a routing tree are plotted also. Number of sensors = 1024, local data set sizes = 1024, $\sigma = 128$.

The routing tree is traversed up to its root after approximately $\log_2(1024)$ steps. Thus, it always takes a constant number of rounds to communicate the final q-digest to all nodes, e.g. by broadcasting the root's q-digest.

If sensor data is biased, the first quantiles computed on local summaries are highly erroneous. But they quickly improve through summary exchange. After five merging steps the quantile error is almost constant and slightly below the one for routing trees. Again, the compression factor k determines the final height of the average quantile error independently from the communication method.

3.4 Experimental Results

The experiments in section 3.3.2 showed that the error computed on gossiped q-digests is only slightly worse than the error computed on the q-digest of the root node of a routing tree, or even better. For all settings of the compression parameter k and different data distribution dependencies, the error converges towards a k-dependent constant. One would expect that merging data summaries along a routing tree is always less erroneous than gossiping, which induces a high degree of redundancy. Redundancy in this case means that the same q-digest is (indirectly) processed several times.

Although, the structure of a routing tree is sampled randomly, the subsequent digestion is deterministic and not redundant. But this seems to be of no advantage with respect to the quantile error.

Generally we observe, that the relative error for gossiping and routing converges towards a constant term. Locally computed quantiles will always differ substantially from globally computed quantiles, no matter of how many q-digests are exchanged during a node's lifetime.

3.5 Discussion

If local data sets do not follow a global data distribution, there is a necessity for exchanging data summaries. The quantile errors in Figure 3.10(b) for sensor nodes communicating in a gossiping manner meet the ones of routing trees after approximately five gossiping rounds. The quick convergence complies with the requisites of transient networks where nodes are short-living and/or nodes have a high data throughput. For the above sensor network a binary routing tree has depth $\log_2 1024 = 10$. It takes ten communication rounds until a single node – the root – has the similar amount of information as any node in a gossiping network.

Besides, forwarding along a routing tree can be considered as a special case of gossiping. In each step, the pool of communicating sensors is diminished by those sensors, who received q-digests from all their children and have sent their summary to parent sensors. This corresponds to a real world scenario in which we can not force all sensors to communicate in each time interval. Sensors might be temporary unavailable.

From the fact, that the quantile error converges towards a k-dependent constant, we can conclude, that supervision and tracking of the number of gossiping steps is not necessary. On the other hand, we can not improve the quality of q-digests by introducing more communication

steps.

Greenwald and Khanna [15] showed that merging quantile summaries deterministically, leads to a final summary that is $h(n)/(2B)(+\max_i(\epsilon_i))$-approximate. With $h(n)$ being the maximum height of the algorithm tree, B the buffer size for storing observations and $\max_i(\epsilon_i)$ the worst precision of any algorithm subtree. Assuming that the local buffer size can not be increased, one consequence is, that we get the highest precision for hierarchical merging, if the algorithm tree is flat, thus the branching factor is high. In routing trees, the message forwarding scheme is fixed, but for nodes organized in a ring-like overlay network, like Chord, it is relatively simple and cheap to implement higher branching factors.

For frameworks using a randomized merging approach, one might conjecture, that the error is unbounded if the algorithm runs for an infinite number of rounds, because this is corresponding to an algorithm tree of arbitrary height h. Although, one has to be careful with drawing conclusions from deterministic, hierarchical to randomized, unhierarchical merging, this is an important aspect, that has to be examined for any approach running an infinite number of steps. In the experimental setup above, the error for q-digests always converged. One explanation might be, that using gossiping, results relatively quick in local q-digests that are close to the 'average' q-digest. Merging similar q-digests neither gives new information, nor does it increase the error.

The table below shows the average quantile errors of all nodes in a algorithm tree (hierarchical) and nodes in a network communicating via the gossip protocol (gossiping) with different branching factors a. For hierarchical merging, the error after the merging of the leaves improves with the height of a. But for $a > 4$, either the subsequent merging introduces an error that exceeds the fact that knowledge is shared, or the intermediate q-digests are that close to the global one, that merging only reinforces their error.

Whereas for gossiping, the error of different branching factors always converges towards a constant between $[0.23, 0.25]$. For higher branching factors, the error only slightly deteriorates, but has the same speed of convergence like hierarchical merging. We conclude that for gossiping the branching factor only determines the speed of convergence, but not the error itself. The quantile error, in turn, depends solely on the choice of the compression parameter k given a data set.

a		**error**										
2	hierarchical	0.677	0.414	0.299	0.272	0.260	0.244	0.236	0.224	0.227	0.228	0.218
	gossiped	0.677	0.377	0.292	0.274	0.260	0.250	0.244	conv	conv	conv	conv
4	hierarchical	0.677	0.265	0.213	0.226	0.220	0.218					
	gossiped	0.677	0.258	0.250	0.247	conv	conv	conv	conv	conv	conv	conv
8	hierarchical	0.677	0.186	0.210	0.227	0.218						
	gossiped	0.677	0.230	0.242	0.242	conv	conv	conv	conv	conv	conv	conv
16	hierarchical	0.677	0.177	0.219	0.236							
	gossiped	0.677	0.229	0.236	conv	conv	conv	conv	conv	conv	conv	conv
32	hierarchical	0.677	0.158	0.232								
	gossiped	0.677	0.232	0.237	conv	conv	conv	conv	conv	conv	conv	conv

Table 3.1: Deterministic, hierarchical merging of q-digests and gossiped merging with varying branching factors a. Data set size = 1024, number of sensors = 1024, for each sensor data set μ and $\sigma_{\mathcal{N}}$ had been drawn randomly from $[\sigma/4, 3\sigma/4]$ and $[1, \sigma/4]$, respectively.

Summed up, we conclude that the q-digest is an appropriate structure for aggregating ob-

servations in data streams. It is not only restricted to represent the last n observations. The merging procedure (see Algorithm 14) provides a mechanism to compute q-digests for arbitrary window sizes[2]. Together with the gossip protocol, we can circumvent an increasing error, that is introduced by merging q-digests permanently.

3.5.1 Efficient Implementation of Higher Branching Factors

For applications running on top of Chord-like overlay networks, which require higher speeds of convergence, there are ways to implement any branching factor a with $\Theta(\log_a N)$ time for look-ups and $\Theta(a \log_a N)$ space complexity of the routing table with N being the identifier space of network nodes[17].

We assume that the identifier space has size $N = a^L$, where L is some integer. To achieve an a-ary look-up, a routing table of $\log_a N = L$ levels is stored at each node. Each level corresponds to a different degree of graininess of the view of the identifier space. Each level divides the closest partition of the preceding level into a equally sized more closed-grained intervals of the identifier space. This is repeated for the subsequent levels, until the subdivisions correspond to consecutive identifiers. All nodes store in their routing tables contact nodes to each of the intervals. If p is the identifier of a node, then the l-th entry of its routing table is $[(p \oplus i \cdot a^{L-l})\%N]$ for $i \in [0, ..., a-1]$. Contact nodes must not be the first numeric node of an interval.

Figure 3.11 gives an example of a fully populated network with an identifier space of $N = 64$, and branching factor $a = 4$. Node 0 stores a routing table with $a \log_a N = aL$ contact nodes. For example, three routing table look-ups are necessary to pass a message from node 0 to node 46. First, node 0 contacts the closest, preceding node, which is node 32. Node 32 has the address of node 44 in the second level of its routing table. Node 44 stores in the third level of its routing table the address of node 46. Thus, the message is passed from node 0 to node 44, and then further to node 46 (see blue path in Figure 3.11).

In an a-ary routing scheme, hierarchical, deterministic merging needs $\Theta(N)$ time steps to digest the data of all nodes. We proceed as follows: a random node is chosen to be the root node. The root sends to all its first level contact nodes the message to send a q-digest back. We state, that a node can answer with a q-digest only if it either has been contacted as a last level node (no further refinement possible) or it has received all q-digests from nodes belonging to its view. In the second case, the node merges its own and the received q-digests into a single one before sending it back via the network.

In hierarchical, deterministic merging, the root sends $a-1$ messages to its first level contact nodes. The root and its first level routing neighbors send altogether $a(a-1)$ messages to their second level neighbors, and so on. At the bottom level, we have $a^{L-1}(a-1)$ sent messages. Hence, we have $\Theta(a^L) = \Theta(N)$ communication costs, which is not surprising, since each node of the network has to be contacted once. But in an a-ary routing scheme, we can parallelize most of the message passing. A node is contacted after at most L time steps to send a q-digest back, or at most $L-1$ time steps to contact $a-1$ further neighbors. Therefore, any node of the network forwards a query for only $a-1$ successors and will be reached within L time steps. If a node can send messages only sequentially, then the slowest messages take $\Theta(aL)$ time steps, otherwise $\Theta(L)$.

[2]By merging q-digests representing older observations with the q-digest on the last n observations.

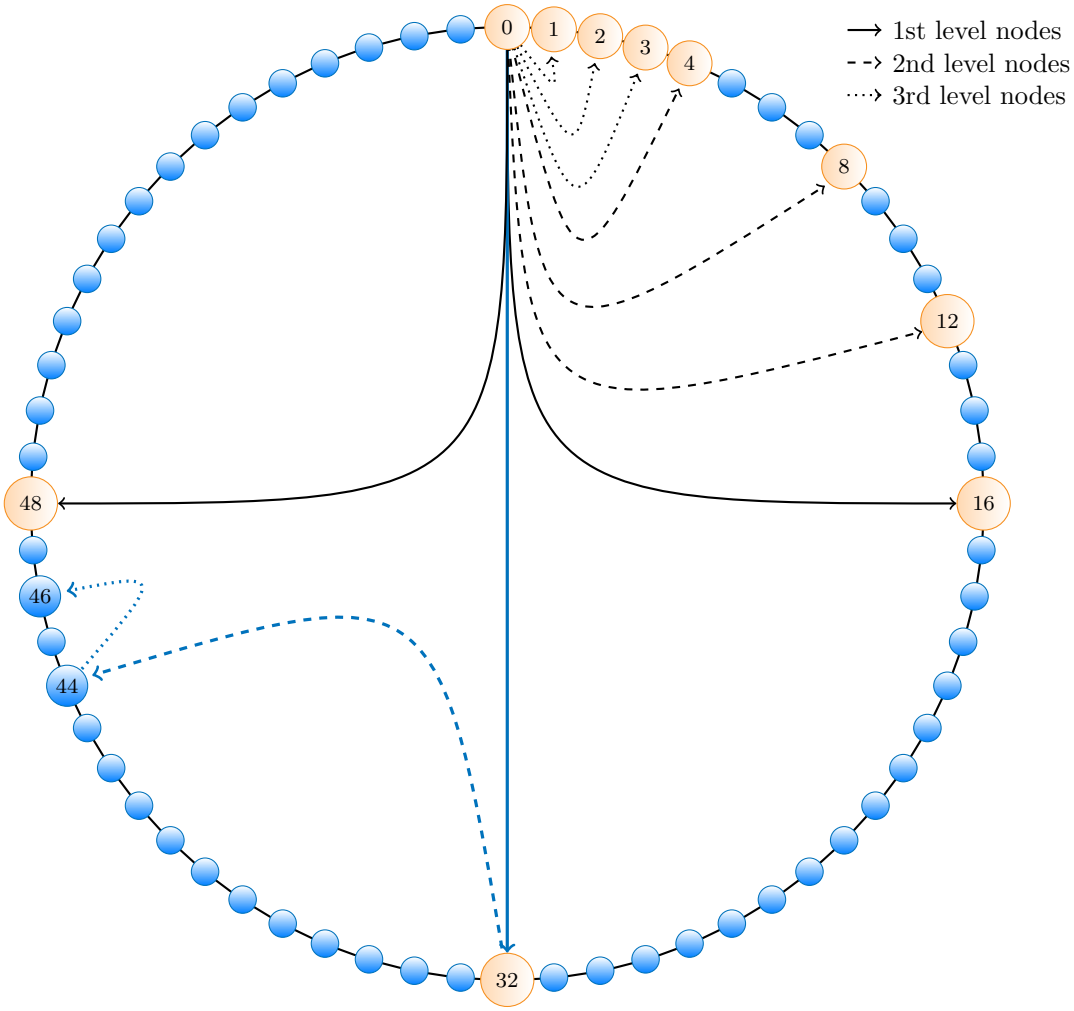


Figure 3.11: Contact nodes (orange) in node 0's routing table with $N = 64 = 4^3$, $a = 4$ and $L = 3$. The blue path shows routing from node 0 to node 46. The last two look-ups are performed by the nodes with ids 32 and 44.

Chapter 4

A Lifetime Estimator for P2P Networks

A major challenge in P2P networks is to maintain the infrastructure of the overlay with the least possible effort. The overlay infrastructure is maintained by routing tables which contain addresses of other nodes in the network. The Chord protocol, for example, involves *fixfinger messages*, a type of message that verifies or repairs entries in a routing structure called finger table. The problem is to find the optimal frequency with which fixfinger messages are sent. Sending with a high frequency is unnecessarily bandwidth consuming, whereas sending with a too low frequency might result in a disrupted infrastructure. The most intuitive way to adjust the frequency for fixfinger messages is to make it dependent from the mean lifetime of nodes in the network. From the mean lifetime we are able to deduce the turnover or *churn rate* – the probability that a node leaves (or enters) the network.

In this chapter we show a method for computing the mean lifetime based on a small set of samples that is collected while the network has already been established. This method fits the requirements of P2P networks – it is completely decentralized and can be implemented at each node. To model lifetimes we use the Weibull probability function whose latent parameters are estimated from the sample set using linear regression. The complexity is linear in terms of the sample set size. The locally estimated mean lifetime will be used to trigger the frequency for fixfinger messages.

4.1 Weibull Distribution

The Weibull probability distribution was first identified by Maurice Fréchet in 1927. It was described in detail by Waloddi Weibull [18], who showed the wide range of its application. Whether it is the breaking strength or fatigue life of material, the size distribution of particles, or the failure probability of electronic devices – they all can be modeled by the Weibull distribution function. The reason is that the Weibull distribution is able to assume the shape of an exponential, normal or Rayleigh distribution, depending on the parameter settings.

We first give the *cumulative distribution function* F for the Weibull function, and then derive its frequency distribution. The cumulative distribution function is the probability P choosing

at random an individual $X \in \mathbf{X}$ being less or equal to any $x \in \mathbf{X}$.

$$P(X \leq x) = F(x)$$

Any distribution function can be expressed as

$$F(x) = 1 - e^{-\phi(\alpha)}$$

The cumulative distribution function for one-dimensional Weibull distributed data is defined as

$$F(x;\ k, \lambda, \theta) := \begin{cases} 1 - e^{-(\frac{x-\theta}{\lambda})^k} & , x \geq \theta \\ 0 & , x < \theta \end{cases}$$

where $k > 0$ is the shape parameter, $\lambda > 0$ the scale parameter and θ the location. Differentiating F with respect to x results in the *frequency* or *probability density function* f

$$f(x;\ k, \lambda, \theta) = \frac{dF}{dx} = \begin{cases} \frac{k}{\lambda}\left(\frac{x-\theta}{\lambda}\right)^{k-1} e^{-(\frac{x-\theta}{\lambda})^k} & , x \geq \theta \\ 0 & , x < \theta \end{cases} \tag{4.1.1}$$

Figure 4.1 shows the probability density function with different settings of the shape. For $k = 5$, the shape of the probability density function becomes similar to the normal distribution.

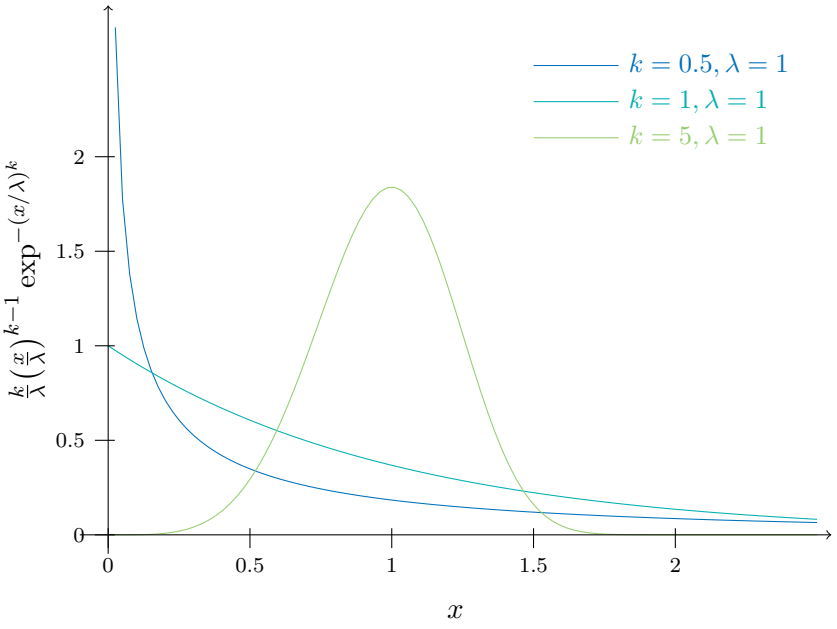


Figure 4.1: Probability density function for Weibull distributed data with varying shape parameters.

4.2 Parameter Estimation

Given a set of sampled lifetimes $\mathbf{X} \in \mathbb{R}^{n \times 1}$, the goal is to determine the parameters k and λ from linear regression, such that the data fits the Weibull distribution function best. The first moment μ can be derived from the estimated shape k_{est} and the scale parameter λ_{est}:

$$\mu = \lambda_{est}\Gamma(1 + 1/k_{est}) \tag{4.2.1}$$

with Γ being the Gamma function, an extended factorial function shifted down by 1.

$$\Gamma(x) := \int_0^\infty t^{x-1} \exp^{-t} dt$$

4.2.1 Linear Regression

Given a sample set $X \in \mathbb{R}^{n \times m}$ with a corresponding set $Y \in \mathbb{R}^{n \times 1}$ for which we assume a linear relationship: $y_i = \alpha_0 + \alpha_1 x_{i,1} + \alpha_2 x_{i,2} + \cdots + \alpha_m x_{i,m}$, $\forall i \in [1..n]$. Our goal is to determine the coefficient vector α such that the squared error E between samples and regression line becomes minimal. We prepend the column vector 1_n to X such that we can multiply X and α without the need to add the extra α_0.

$$E = \frac{1}{2}(Y - X\alpha)^T(Y - X\alpha) \quad \rightarrow \min! \tag{4.2.2}$$

We minimize E by determining the root of the derivation for α.

$$\begin{aligned}
\frac{\partial E}{\partial \alpha} &= 0 \\
\frac{1}{2}(-X)^T(Y - X\alpha) + \frac{1}{2}(Y - X\alpha)^T(-X) &= 0 \\
-X^T(Y - X\alpha) &= 0 \qquad (\text{ due to } A^TB = B^TA) \\
-X^TY + X^TX\alpha &= 0 \\
\alpha &= (X^TX)^{-1}(X^TY)
\end{aligned} \tag{4.2.3}$$

4.2.2 Application to Weibull Distributed Data

To make use of Formula 4.2.3, we have to linearize the Weibull function. We can not linearize it directly, but its cumulative distribution function. When the double logarithm of F is plotted versus the logarithm of x, the dependencies may be described as a linear equation of the form $y = mx + c$. Thus, by changing the variables, F can be linearized and utilized for linear regression.

$$\begin{aligned}
F(x;\, k, \lambda) &= 1 - \exp^{-(x/\lambda)^k} \\
-\ln(1 - F(x;\, k, \lambda)) &= (x/\lambda)^k \\
\underbrace{\ln\big(-\ln(1 - F(x;\, k, \lambda))\big)}_{y} &= \underbrace{k \ln x}_{mx} - \underbrace{k \ln \lambda}_{c}
\end{aligned}$$

We can calculate easily y and $\ln x$ for all $x \in X$ of our samples. The remaining parameters k and c can be estimated by solving the linear equation system:

$$\begin{pmatrix} \ln(-\ln(1 - F(x_1))) \\ \ln(-\ln(1 - F(x_2))) \\ \vdots \\ \ln(-\ln(1 - F(x_n))) \end{pmatrix} = \begin{pmatrix} \ln x_1 & 1 \\ \ln x_2 & 1 \\ \vdots & \vdots \\ \ln x_n & 1 \end{pmatrix} \begin{pmatrix} k \\ c \end{pmatrix} \tag{4.2.4}$$

Plugged into result (4.2.3), we get:

$$\begin{pmatrix} k \\ c \end{pmatrix} = (X^T X)^{-1} X^T Y \tag{4.2.5}$$

The computation of the inverse of $X^T X$ is trivial, since it is in $\mathbb{R}^{2\times 2}$. Let $A \in \mathbb{R}^{2\times 2}$ with $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$, then the inverse of A is:

$$A^{-1} = \frac{1}{\det(A)} adj(A) = \frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

The case $\det(A) = 0$ (singular or non-regular matrix) represents are rare case, but has to be checked in an implementation.

After having estimated c and k_{est}, λ can be inferred by its relation to k and c:

$$\lambda_{est} = \exp^{-c/k_{est}}$$

Finally, k_{est} and λ_{est} are plugged into Equation 4.2.1 to compute the mean of the sampled lifetimes. Let n be the number of samples, then the expected running time is $\Theta(n)$.

4.3 Evaluation

4.3.1 Linear Regression Error

In order to measure the quality of the mean μ_{LR} estimated by linear regression, it was compared to Matlab's built-in function `wblfit`, which iteratively approximates k and λ. Given the shape and the scale parameter, formula 4.2.1 gives the mean μ_M. Both approximated means were plugged into error Formula 4.3.1. The estimated mean μ_{LR} was compared to the exact, but latent mean μ also (see error plot 4.2).

$$\begin{aligned} e(\mu_{LR}, \mu) &= \left(\frac{\mu_{LR} - \mu}{\mu}\right)^2 \\ e(\mu_M, \mu) &= \left(\frac{\mu_M - \mu}{\mu}\right)^2 \end{aligned}$$

Figure 4.2 shows the result of 10,000 repetitions. The scale parameter was fixed, the shape was chosen randomly from the interval $[1, 5]$. The exact mean μ is the result of equation 4.2.1. The error decays exponentially. Depending on the robustness of the application or protocol, 5, 10, or 20 samples are sufficient to compute an almost-exact mean. For the subsequent tests in Chord, most peers collected not more than 5 samples.

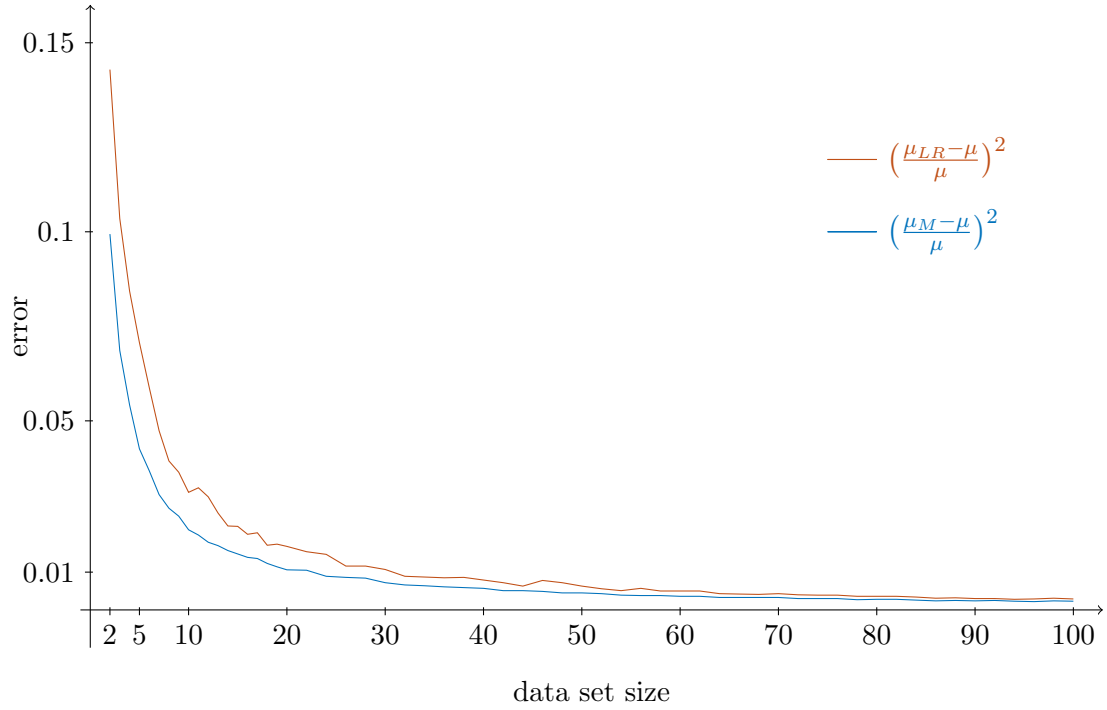


Figure 4.2: Comparison between Matlab's built-in wblfit and parameter estimation with linear regression.

4.3.2 Message Reduction in Chord

We now show how to use the lifetime estimator in an overlay network using the simulation framework Oversim[1]. Oversim is an open-source framework for simulating P2P networks and different layer types. Among the overlay protocols there is an implementation of Chord. Nodes in an overlay network using the Chord protocol are arranged in a ring-like structure (see Figure 3.11 in the previous chapter). For ring maintenance there are essential messages which are send if an internal timer has expired or as a response:

`fix_finger`	Called periodically to refresh finger table entries. If enlisted nodes have become unavailable, new successors have to be determined.
`stabilize`	Called to check whether the predecessor of a node's successor is the same.
`notify`	Sent by node who might be the predecessor of the receiver.
`ping`, `pingResponse`	Helper routine to check for aliveness.

For our purpose we added a new version of the Chord protocol which contains the following modifications:

1. We added the function `churnRateEstimator` in `ChordFingerTable.cc` which proceeds like described in section 4.2.2 to estimate the mean lifetime. The function receives a

[1] `http://www.oversim.org/`

vector of pairs of node IDs and simulation time points from nodes that did not respond to ping messages. See code in Appendix 5.3.

2. The function `handleFixFingersTimerExpired` in `ChordNew.cc` calls `churnRateEstimator` and adjusts the frequency f of the finger table update according to the estimated μ. We do not trigger the frequency of the function call of `handleFixFingersTimerExpired` directly, but implemented a switch using the formula:

   `fixFingerCalls = (fixFingerCalls + 1) % frequency`.

   Each time `fixFingerCalls` equals zero `fix_finger` messages are sent. For all other cases the procedure collects lifetimes by sending ping messages to all nodes in its routing table (see lines 991 to 1003 in Appendix 5.3) and storing the current simulation time. If a node does not respond, it is considered to be dead and its lifetime is computed by taking the difference of the former system time and its time of creation [2].

There are different strategies possible to adjust the frequency of fixfinger messages.

i) Simply switching between a high and a low frequency:
$$\text{frequency} = (\mu < c) \ ? \ \alpha_{low} \ : \ \alpha_{high}$$

ii) The frequency depends linearly from the estimated churn rate:
$$\text{frequency} = \lceil \alpha \cdot \mu \rceil$$

iii) The frequency depends logarithmically from the estimated churn rate:
$$\text{frequency} = \log^2 \mu$$

4.3.2.1 Results with OverSim

All three strategies for adjusting the frequency of fix finger messages resulted in a smaller total number of sent messages compared to the standard Chord implementation with constant frequency. Figure 4.3 shows the results for the third strategy, which has the advantage that no parameters have to be tuned. The blue line refers to the standard implementation, the green line to the modified version 'ChordNew'. The plotted values are mean values, averaged for 100 repetitions of the simulation. Different mean lifetimes from the interval [200; 800] have been modeled, they are assigned to the x-axis.

Each experiment starts with a short *initial phase* in which the specified number of nodes is created for the overlay, here 1024. The node creation phase is followed by the *transition phase* – a temporal buffer to give the nodes time to establish the infrastructure of the overlay. Finally the *measurement phase* starts – the only interval in which statistical data is collected. Among these are:

- The **Number of maintenance messages** is the mean of the number of all messages needed to maintain Chord's infrastructure. Decreasing the number of fixfinger messages does not automatically decrease the total number of messages in the network. In our case, we have to send more ping messages to test for aliveness of nodes.

[2] see line 331 in function `churnRateEstimator` in Appendix 5.3

- The **number of fixfinger messages** is the mean of the number of sent fixfinger messages per second.
- The **number of ping messages** and **ping response messages** are the mean of the numbers of sent ping messages and their responses per second.
- **Packets dropped** is the mean of the total number of dropped UDP-packets due to unavailable destination. This characteristic reflects the health of the infrastructure. Packets are dropped if finger table entries are out of date.
- **One-way hop count** is the mean of hops a packet spends in traveling from one node to the target node.

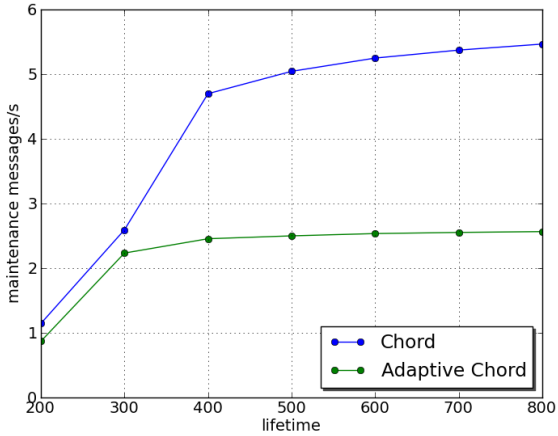
Figure 4.3(a) shows that the number of messages needed for ring maintenance gets halved for nodes with mean lifetimes of at least 400s. One reason for the decreased number of maintenance messages is that the adjusted frequency for fixfinger messages is lower than the one of the standard implementation (see Figure 4.3(b)). The price for a reduced number of fixfinger messages is that more ping and ping response messages need to be sent. But, expressed in number of hops, they are much cheaper than fixfinger messages.

Most important – a lower fixfinger frequency does not harm the infrastructure. We need up to two more hops for packet delivery (see Figure 4.3(f)), but less UDP packets of the transport layer get lost (see Figure 4.3(e)).

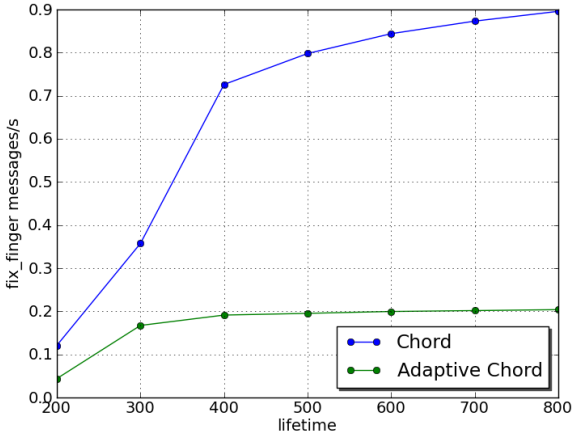
4.4 Discussion

We showed how to utilize the lifetime estimator to reduce the number of messages that serve for ring maintenance in the Chord protocol. There are many more applications of the lifetime estimator, which are not restricted to the overlay. Mean lifetimes can be transmitted to the application layer, where applications may alter their behavior depending on the churn rate of clients.

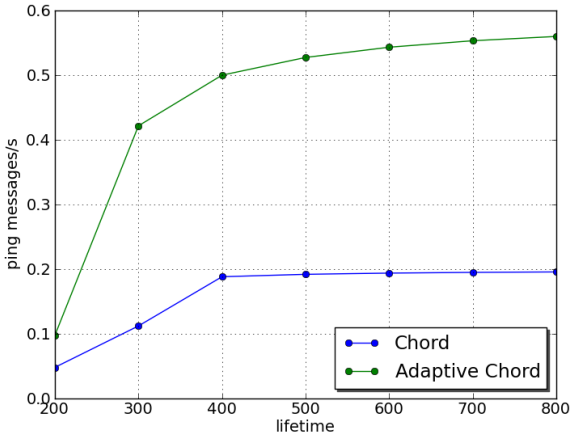
In our implementation lifetime data was collected by sending ping messages to neighbors from the finger table. Since Chord guarantees communication costs of $\Theta(\log N)$ between two arbitrary nodes, one could also implement a gossiping-based version for sending ping messages, like Pruteanu et al. [19] did in their churn rate detecting algorithm. The main focus of this chapter was not to develop an efficient framework for the distributed churn estimation problem, but to show how to determine the parameters of a Weibull distribution based on a small set of samples.



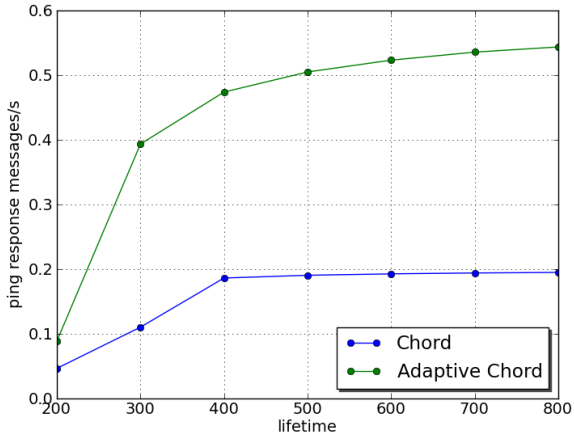
(a) Number of maintenance messages.



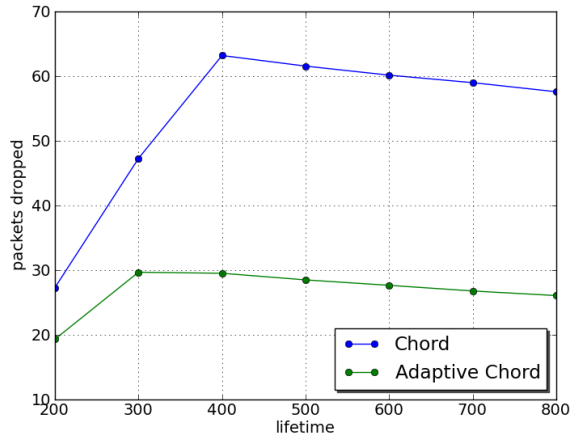
(b) Number of fixfinger messages.



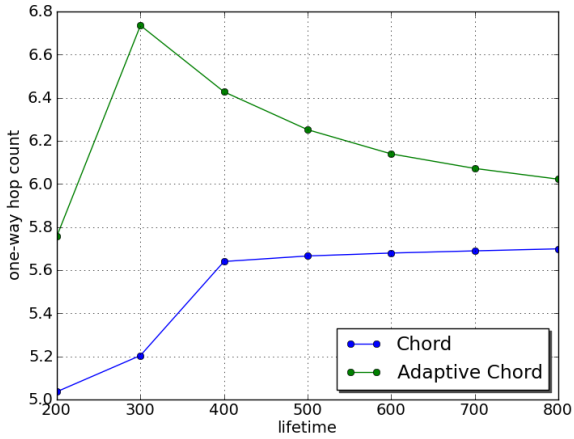
(c) Number of ping messages.



(d) Number of ping response messages.



(e) Packets dropped due to unavailable destination.



(f) One-way hop count.

Figure 4.3: Strategy: frequency = $\log^2 \mu$, steady-state net size: 1024, measuring time: 5000s, number of repetitions: 100.

Chapter 5

Summary

5.1 Approximate Distributed Clustering over a P2P Network

We modified the unsupervised learning technique of agglomerative clustering to produce meaningful coordinates for the placement of a fixed number of replicas. The constraints for optimal replica placement are avoidance of outliers, and replica spreading into different clusters. During the last unification steps of agglomerative clustering these constraints are potentially violated. By stopping the clustering routine much earlier, we keep enough centroids to reject outliers and merge the remaining centroids with a new merging criterion which takes into account the relative cluster sizes. If the designated number of centroids is less or equal to the real number of centroids, the modified agglomerative clustering routine is able to detect the latent centroids and will reject the smaller ones for replica placement. Whereas K-means would compute centroid coordinates that lie between two or more latent clusters.

On the other hand, if several replicas have to be stored in the same cluster, we reduce failure probabilities by maximizing the spatial distances between them. K-means meets this criterion best, since it minimizes the distances between centroids and assigned nodes. The agglomerative clustering procedure does not label nodes. This fact offered the opportunity to design a very simple version for P2P systems which gossips locally estimated centroids and does not require any kind of synchronization. But for ensuring large distances between replicas in the same cluster, we had to take into account the current relative sizes of centroids in a final merging phase to force agglomeration towards $1/k$-th of the total net size.

5.2 Quantiles and Histograms on Distributed Streams

In histogram computation, equi-width histograms are the ones with unbounded errors. If the bucket width is not adaptive, arbitrary many or few items might be counted into a single bin. A q-digest is a structure that corresponds to an equi-probable histogram, except that the buckets are overlapping. By ensuring that a bucket count does not fall below or exceeds a threshold, we can guarantee some degree of exactness and an upper bound for the storage space. With the constraint that the storage space is not increased, q-digests can be merged in a deterministic way, such that the root node receives a q-digest aggregating the data from all sources. In chapter 3 we showed, that the error bounds are preserved if q-

digests are merged randomly for arbitrary many iterations. Therefore, we can use the robust gossiping-protocol which meets the requests of P2P systems much better than supervised or deterministic approaches.

5.3 A Lifetime Estimator for P2P Networks

Knowing the churn rate of web clients is not only interesting for commercial applications, but also for the underlying overlay protocol. Maintaining the P2P overlay structure requires frequent passing of messages for detecting dead or joined nodes and correcting routing table entries at peers. Since messages dedicated for maintenance also consume prized bandwidth, it is a goal to reduce their frequency without disrupting the infrastructure of the overlay. Obviously, the lower bound of needful maintenance messages correlates with the mean lifetime of peers. In chapter 4 we showed how nodes can sample a few lifetimes of peers and use linear regression to estimate the latent parameters of the Weibull distribution function. The Weibull function is flexible enough to model exponential decay or normally distributed data. We modified the Chord protocol and charged the update frequency for `fix_finger` messages with the estimated mean lifetime. Thereby, we reduced the number of maintenance messages significantly without degrading the infrastructure.

Appendix

Code

Listing 1: ChordNewFingerTable::churnRateEstimator

```
313 /* estimating the churnRate with linear regression */
314 double ChordNewFingerTable::churnRateEstimator(double fixfingersDelay, std::map
        <OverlayKey, std::pair<int, simtime_t> > *pingSent)
315 {
316     double k = -1;  // shape parameter
317     double lambda = -1; // scale parameter
318     double mu = -1;     // estimated mean lifetime (churn rate)
319     double MIN_DOUBLE = -50;
320     double MAX_DOUBLE = 10000;
321
322     std::vector<double> data;
323     std::map<OverlayKey,simtime_t>::iterator it;
324     std::map<OverlayKey,std::pair<int, simtime_t> >::iterator it1;
325
326     OverlayKey key;
327     /* use age of not ping responding nodes */
328     for (it1 = pingSent->begin(); it1 != pingSent->end(); it1++){
329         it = insertTocByKey.find((OverlayKey)(*it1).first);
330     if (it != insertTocByKey.end() && ((*it1).second.first != 1))
331             data.push_back(SIMTIME_DBL( (*it1).second.second) - SIMTIME_DBL((*
                it).second) - fixfingersDelay/2);
332     }
333     /* else: use age of vivid fingers */
334     if (data.size() == 0) {
335         for (uint p = 0; p < fingerTable.size(); p++){
336             if (fingerTable[p].first.isUnspecified())
337                 continue;
338             key = (OverlayKey) fingerTable[p].first.getKey();
339             if (isDuplicate(p))
340                 continue;
341             it = insertTocByKey.find(key);
342             if (it != insertTocByKey.end()){
343             data.push_back(SIMTIME_DBL(simTime()) - SIMTIME_DBL(insertTocByKey[
                key]));
344             }
345         }
346     }
347
348     /* no data collected - return */
349     if (data.size() == 0)
350     return 0;
```

```
351     else if (data.size() == 1)
352         return data.at(0);
353
354     std::vector<double> y_vec;
355     std::vector<std::pair<double, double> > x_mat;
356
357     int y_del = 0;
358     double y_val;
359     for (uint i = 0; i < data.size(); i++){
360         y_val = log(-log(1-edf(data, i)));
361         /* remove outliers */
362         if (y_val > MAX_DOUBLE || y_val < MIN_DOUBLE){
363             ++y_del;
364             continue;
365         }
366         y_vec.push_back(log(-log(1-edf(data, i))));
367         x_mat.push_back(std::make_pair<double, double>(log(data[i]), 1));
368     }
369
370     /* size_d = valid entries */
371     uint size_d = data.size() - y_del;
372     if (size_d == 0)
373         throw new cRuntimeError("ChordNewFingerTable::churnRateEstimator: all
            data out of range!\n");
374
375     if (size_d == 1)
376         data.at(0);
377
378     uint d2 = 2, d1 = 1;
379     double **y;
380     double **x;
381     double **alpha;
382     double **x_trans;
383     double **xx;
384     double **xx_inv;
385     double **xy;
386
387     /* allocate memory */
388     y = new double*[size_d];
389     x = new double*[size_d];
390     alpha = new double*[d2];
391     x_trans = new double*[d2];
392     xx = new double*[d2];
393     xx_inv = new double*[d2];
394     xy = new double*[d2];
395
396     for (uint i = 0; i < size_d; i++){
397         y[i] = new double[d1];
398         x[i] = new double[d2];
399     }
400
401     for (uint i = 0; i < d2; i++){
402         alpha[i] = new double[d1];
403         x_trans[i] = new double[size_d];
404         xx[i] = new double[d2];
405         xx_inv[i] = new double[d2];
406         xy[i] = new double[d1];
407     }
408
```

```
409     /* assign data */
410     for (uint i = 0; i < size_d; i++){
411         y[i][0] = y_vec.at(i);
412         x[i][0] = x_mat.at(i).first;
413         x[i][1] = x_mat.at(i).second;
414     }
415
416     /* solve using least squares, alpha = (x'x)^{-1}(x'y) */
417     trans(x_trans, x, size_d, d2);  // x'
418     mult(xx, x_trans, d2, size_d, x, size_d, d2); // X'X
419
420     /* (X'X)^-1 */
421     inv(xx_inv, xx);
422
423     mult(xy, x_trans, d2, size_d, y, size_d, d1);
424     mult(alpha, xx_inv, d2, d2, xy, d2, d1);
425
426     k = alpha[0][0];
427     lambda = exp(-alpha[1][0]/k);
428     mu = lambda*tgamma(1+1/k);
429
430     /* free memory */
431     for (uint i = 0; i < size_d; i++){
432         delete [] y[i];
433         delete [] x[i];
434     }
435     delete [] x;
436     delete [] y;
437     for (uint i = 0; i < d2; i++){
438         delete [] alpha[i];
439         delete [] x_trans[i];
440         delete [] xx[i];
441         delete [] xx_inv[i];
442         delete [] xy[i];
443     }
444     delete [] alpha;
445     delete [] x_trans;
446     delete [] xx;
447     delete [] xx_inv;
448     delete [] xy;
449
450     return mu;
451 }
```

Listing 2: ChordNewFingerTable::Helper Functions

```
460 /* calculate the transposal of a given matrix */
461 void ChordNewFingerTable::trans(double** A_trans, double** A, uint height, uint
        width){
462     for (uint i = 0; i < width; i++){
463         for (uint j = 0; j < height; j++)
464             A_trans[i][j] = A[j][i];
465     }
466 }
467
468 /* multiplicate two matrices A of size kxl and B of size mxn */
469 void ChordNewFingerTable::mult(double** AB, double** A, uint k, uint l, double
        ** B, uint m, uint n){
470     if (l != m || !(k&&l&&m&&n))
```

```
471             throw new cRuntimeError("ChordNewFinger::mult(): Wrong dimension for
                    matrix multiplication\n");
472
473         for (uint idx1 = 0; idx1 < k; idx1++){
474             for (uint idx2 = 0; idx2 < n; idx2++){
475                 AB[idx1][idx2] = 0;
476                 for (uint idx3 = 0; idx3 < l; idx3++)
477                     AB[idx1][idx2] += A[idx1][idx3]*B[idx3][idx2];
478             }
479         }
480     }
481
482     /* empirical distribution function edf(t) = (rank(t)-0.3)/(n+0.4)*/
483     double ChordNewFingerTable::edf(std::vector<double>v, uint idx){
484         uint countLEQ = 0;
485         for (std::vector<double>::size_type i = 0; i != v.size(); i++){
486             if (v[i] <= v[idx])
487             countLEQ++;
488         }
489         return ((double)countLEQ-0.3)/((double)v.size() + 0.4);
490     }
491
492     /* inverse of 2x2 matrix: 1/det(A)*adj(A) */
493     void ChordNewFingerTable::inv(double** A_inv, double** A){
494         double det = A[0][0]*A[1][1] - A[0][1]*A[1][0];
495         if (det == 0)
496             throw new cRuntimeError("ChordNewFingerTable::inverseD2: determinant of
                    matrix is zero\n");
497         A_inv[0][0] = A[1][1]/det;
498         A_inv[0][1] = -A[0][1]/det;
499         A_inv[1][0] = -A[1][0]/det;
500         A_inv[1][1] = A[0][0]/det;
501     }
```

Listing 3: ChordNew::handleFixFingersTimerExpired

```
941     void ChordNew::handleFixFingersTimerExpired(cMessage* msg) {
942         /* estimate churn rate only if at least one node didn't respond */
943         std::map<OverlayKey, std::pair<int, simtime_t> >::iterator it;
944         for (it = pingSent.begin(); it != pingSent.end(); it++){
945             if ((*it).second.first != 1){ // 0 no response, 1 ping response, 2 ping
                    timeout
946                 churnRate = fingerTable->churnRateEstimator2(fixfingersDelay, &
                        pingSent);
947                 if (churnRate <= 0) break;
948
949                 /* adjust frequency for table stabilization */
950                 if (churnRate < 100000){
951                     churnRates.push_back(churnRate);
952                     frequency = std::max((float) 1, (float) pow(log10(churnRateMean
                            (churnRates)), 2));
953                 }
954                 break;
955             }
956         }
957         fixFingerCalls = (fixFingerCalls+1) % frequency;
958         uint32_t nextFinger;
959         OverlayKey offset, lookupKey;
960         if ((state != READY) || successorList->isEmpty())
```

```
961             return;
962
963         /* send FixfingersCalls according to the frequence */
964         if (fixFingerCalls == 0){
965             for (nextFinger = 0; nextFinger < karyLength(thisNode.getKey().
                    getLength()); nextFinger++) {
966                 offset = offsetFunc(nextFinger);
967                 lookupKey = thisNode.getKey() + offset;
968                 /* send message only for non-trivial fingers */
969                 if (offset > successorList->getSuccessor().getKey() - thisNode.
                        getKey()) {
970                     // call FIXFINGER RPC
971                     FixfingersNewCall* call = new FixfingersNewCall("
                            FixfingersNewCall");
972                     call->setFinger(nextFinger);
973                     call->setBitLength(FIXFINGERSNEWCALL_L(call));
974                     sendRouteRpcCall(OVERLAY_COMP, lookupKey, call, NULL,
                            DEFAULT_ROUTING, fixfingersDelay);
975                 }
976                 /* delete trivial fingers (points to the successor node) */
977             else
978                 fingerTable->removeFinger(nextFinger);
979             }
980         }
981
982         /* send ping messages to nodes in fingertable */
983         else {
984             /* check aliveness of fingers */
985             if (!pingRespFingerSet.empty()){
986                 pingRespFingerSet.clear();
987                 pingRespTime['s'].clear();
988                 pingRespTime['r'].clear();
989             }
990             fingersUnique.clear();
991             for (nextFinger = 0; nextFinger < fingerTable->getSize(); nextFinger++)
                    {
992                 if (fingersUnique.find(fingerTable->getFinger(nextFinger).getKey())
                        == fingersUnique.end()){
993                     fingersUnique.insert(fingerTable->getFinger(nextFinger).getKey
                            ());
994                     pingNode(fingerTable->getFinger(nextFinger), -1, 1, NULL, "PING
                            ", NULL, 2, UDP_TRANSPORT);
995                     pingRespTime['s'].push_back(simTime());
996                     std::pair<std::map<OverlayKey, std::pair<int, simtime_t> >::
                            iterator,bool> ret;
997                     ret = pingSent.insert(std::pair<OverlayKey, std::pair<int,
                            simtime_t> > ( fingerTable-> getFinger(nextFinger).getKey()
                            , std::make_pair<int, simtime_t>(0, simTime())));
998                     if (ret.second == false){
999                         pingSent.erase(ret.first);
1000                        pingSent.insert(std::pair<OverlayKey, std::pair<int,
                                simtime_t> > ( fingerTable-> getFinger(nextFinger).
                                getKey(), std::make_pair<int, simtime_t>(0, simTime())
                                ));
1001                    }
1002                }
1003            }
1004        sentAlivenessCheck = true;
1005        }
```

```
1006        /* schedule next finger repair process */
1007        cancelEvent(fixfingers_timer);
1008        scheduleAt(simTime() + fixfingersDelay, msg);
1009    }
```

Bibliography

[1] Dhillon, Inderjit S. and Modha, Dharmendra S.: *A Data-Clustering Algorithm on Distributed Memory Multiprocessors*. Revised Papers from Large-Scale Parallel Data Mining, Workshop on Large-Scale Parallel KDD Systems, SIGKDD, 2000, Springer-Verlag, London, UK.

[2] Marie Hoffmann: *An Agglomerative, Gossip-based Clustering Approach for Distributed Systems*. Bachelor's Thesis, FU Berlin, Institut für Informatik, 2009.

[3] Thorsten Schütt, Alexander Reinefeld, Florian Schintke, and Marie Hoffmann: *Gossip-based Topology Inference for Efficient Overlay Mapping*. Peer-to-Peer Computing, 2009

[4] Souptik Datta, Chris R. Giannella, and Hillol Kargupta: *Approximate Distributed K-Means Clustering over a Peer-to-Peer Network*. IEEE Transactions on Knowledge and Data Engineering, Volume 21, October 2009

[5] Christopher M. Bishop: *Pattern Recognition and Machine Learning* Springer 2006.

[6] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris: *Vivaldi: A Decentralized Network Coordinate System*. SIGCOMM'04, Aug. 30-Sept. 3, 2004, Portland, Oregon, USA.

[7] A. K. Jain, M. N. Murty, and P. J. Flynn: *Data Clustering: A Review*. ACM Computing Surveys, Vol. 31, No. 3, September 1999.

[8] `http://entwickler.com/itr/psecom,id,84,nodeid,.html`
entwickler.com - Lexikon.

[9] edited by Charu C. Aggarwal: *Data Streams – Models and Algorithms*. Springer 2007.

[10] Chiranjeeb Buragohain and Subhash Suri: *Quantiles on Streams*.

[11] Raj Jain and Iimrich Chlamtac: *The P^2 Algorithm for Dynamic Statistical Calculation of Quantiles and Histograms Without Storing Observations*. Communications of the ACM, Vol. 28, No. 10, October 1985.

[12] J. I. Munro and M. S. Paterson: *Selection and Sorting with Limited Storage*. Theoretical Computer Science, pp 315–323, 1980.

[13] G. Piatetsky-Shapiro and C. Connell: *Accurate Estimation of the Number of Tuples Satisfying a Condition*. ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD), 1984.

[14] N. Shrivastava, C. Buragohain, D. Agrawal, S. Suri: *Medians and Beyond: New Aggregation Techniques for Sensor Networks*. SenSys '04, November 3-5, 2004, Baltimore, Maryland, USA.

[15] M. B. Greenwald and S. Khanna: *Quantiles and EquiDepth Histograms over Streams*. Chapter 2 of Part II of "Data Stream Management: Processing High-Speed Data Streams", edited by M. Garofalakis, J. Gehrke, and R. Rastogi, Springer, 2005.

[16] H.L. Chan, T.-W. Lam, L.K. Lee and H.F. Ting: *Continous Monitoring of Distributd Data Streams over a Time-based Sliding Window*. Symposium on Theoretical Aspects of Computer Science 2010 (Nancy, France). pp. 179-190.

[17] Ali Ghodsi: *Distributed k-ary System: Algorithms for Distributed Hash Tables* Dissertation submitted to the Royal Institute of Technology (KTH), December 2006, School of Information and Communication Technology, Department of Electronic, Computer, and Software Systems Stockholm, Sweden.

[18] Waloddi Weibull: *A Statistical Distribution Function of Wide Applicability* Contributed by the Applied Mechanics Division for presentation, by title, at the Annual Meeting, Atlantic City, N.J., November 25 – 30, 1951, of The American Society of Mechanical Engineers.

[19] Pruteanu, Andrei and Iyer, Venkat and Dulman, Stefan: *ChurnDetect: A Gossip-Based Churn Estimator for Large-Scale Dynamic Networks* Euro-Par 2011 Parallel Processing, Lecture Notes in Computer Science, pp 289–301, Springer Berlin Heidelberg, 2011.

[20] Spyros Voulgaris and Daniela Gavidia and Maarten Van Steen: *CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays* Journal of Network and Systems Management, volume 13, 2005.

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich diese Abschlussarbeit selbständig verfasst, noch nicht anderweitig für andere Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Berlin, den 04. Februar 2013