Simulation of the BubbleStorm Peer-to-Peer Network

Bachelor Thesis

Author:	Dimitar Mechev
Supervisor:	Christof Leng
Reviewer:	Prof. Alejandro Buchmann, PhD
Submission Date:	06.09.2008

Contents

List of Figures

List of Tables

Chapter 1

Summary

1.1 Zusammenfassung

Die Aufgabenstellung dieser Bachelorarbeit beinhaltet die Implementierung, Simulation und Auswertung eines unstrukturierten Peer-to-Peer Netzwerksystems fuer probabilistische Suche namens BubbleStorm. Die Hauptziele der Arbeit sind eine erweiterbare Implementierung von BubbleStorm anzubieten, das System innerhalb einer realistisch simulierten Umgebung auszuwerten und die gewonnenen Ergebnissen mit bestehenden Ergebnisse zu vergleichen.

Damit die oben gennanten Ziele erreicht werden, wird als Plattform PeerfactSim.KOM eingesetzt.

1.2 Summary

This bachelor thesis is about implementing, simulating and evaluating of an unstructured peer-to-peer network system for probabilistic search called BubbleStorm. The main objectives of this work are to provide extendable implementation of BubbleStorm, to evaluate the system within a realistic simulated environment and to compare the results with results of the BubbleStorm prototype.

In order to achieve the former goals an evaluation platform for large-scale peer-to-peer networks called PeerfactSim.KOM is used.

Chapter 2

Introduction

2.1 Peer-to-Peer Systems

Peer-to-Peer network architectures try to provide a simple solution to the problems, that client-server architectures tend to encounter trying to meet the evolving requirements of the Internet. In a peer-to-peer system each peer is equipotent. Peers are clients and servers at the same time and do not depend on conventional centralized resources. Peer-to-peer systems take advantage of cumulative resources provided by each peer. Although less reliable and secure, peer-to-peer systems provide cheaper and easier maintenance. Furthermore, peer-to-peer networks are an intuitive and natural approach for implementing human network-based organizations like social networks.

Each peer-to-peer network has specific overlay topology build on the top of the transport layer of the ISO/OSI model. In order to maintain the topology structure and share resources peers interact directly with each other by exchanging messages. Depending on the topology structure, peer-to-peer networks can be classified into two main types - hybrid and pure peer-to-peer networks.

In hybrid peer-to-peer systems peers are connected to a central sever, that processes peer requests. The role of the server is to provide an index of all shared resources. The actual information is transmitted directly between the peers. The central resource index provides fast and efficient search but it still remains a single point of failure. Well known hybrid peer-to-peer systems are Napster, Direct Connect, SoulSeek and BitTorrent.

In pure peer-to-peer systems all participants have the same role in the network. Thus single points of failure are avoided. In order to maintain self-organization and reliable resource sharing such system need more complex overlay topology. Pure peer-to-peer networks are divided into two subtypes depending on the overlay structure.

2.1.1 Structured

Structured pure peer-to-peer overlays typically bind content to hash keys, which are used as addresses. In contrast to hybrid peer-to-peer networks, where the index of the shared resources is kept on a central server, structured overlays use Distributed Hash Table (DHT). Each peer is responsible for some of the key-value pairs, so a message addressed to any key will incrementally route towards the overlay node responsible for this key and corresponding data. As each object within the overlay is uniquely identifiable it can be addressed directly. This approach is more efficient than unstructured searching, but an object can only be found if its unique name is known. The main disadvantage is the complex structure maintenance, especially after node failures. Well known structured overlays that use DHT are: Chord [13], Pastry [14], Tapestry [15], CAN [16] and Tulip [17]. There also exist structured overlays, like HyperCuP [10], that do not implement the DHT approach. HyperCuP relies on a deterministic organization of the peers in a hypercube graph for efficient broadcasting and searching.

2.1.2 Unstructured

In unstructured pure peer-to-peer networks the overlay links are established arbitrarily or randomly. Such networks can be easily constructed, as a joining peer can copy existing links of other already participating peers and establish its own links to some arbitrary peers. The construction technique can also use the random walk approach. In order to find specific data, a query is typically flooded through the network. Flooding wastes a lot of network and peer resources and it should be limited in order to prevent network overload. The limitation of the flood on the other hand dramatically affects the search success, as only part of the network processes the requests. Well known unstructured overlays that use flooding search technique are Gnutella [18] and FastTrack [19]. Another searching approach is the random walk search technique. In general a random walk based search algorithm can reduce the network traffic and enhance the system scalability. However it experiences longer search latency and its success rate depends to a great extend on the underlying topology. Random walks achieve improvement over flooding in the case of clustered overlay topologies and in the case of re-issuing the same request several times.

2.2 Evaluation of Peer-to-Peer Network Overlays

Evaluation methods for peer-to-peer overlays can be analytical, involve testing prototypes, measuring of real-world deployments or simulation. An analytical approach requires too many simplifications as the peer-to-peer systems are very complex. Although it is useful and many overlays originate as pure analysis, the analytical approach is not sufficient to guarantee a working real-world system. Running large scale experiments with prototypes in a testbed is difficult due to a lack of sufficiently large testbeds. It is also hard to simulate realistic conditions and to collect and analyze the testbed experiment output. The measurement of real-world deployments is not useful for overlays originated in research projects. However it is used as source for statistical data for comparison. The approximations which simulations provide are

much closer to reality than an analytical approach, and it is possible to simulate networks of hundred thousands of peers.

Chapter 3

The BubbleStorm system

3.1 Overview

Terpstra, Kangasharju, Leng, and Buchmann developed a peer-to-peer system called BubbleStorm. BubbleStorm is an unstructured pure peer-to-peer system, which uses randomized processes to probabilistically organize the nodes within the overlay and provides exhaustive search with probabilistic guarantees. It is specially designed to solve the rendezvous problem in heterogeneous network. Since BubbleStorm is an unstructured overlay, queries are evaluated at the peers that receive them, providing a useful separation between network topology and query evaluation. The network-level search strategy of the system enables easier design and optimization of distributed query languages. BubbleStorm provides application designers with the freedom to create more complex and sophisticated P2P applications using any existing libraries for query evaluation (full text, XPath, relational, etc.) and easily integrating Client/Server algorithms to the P2P environment.

3.2 Topology

Nodes are randomly placed in the network and follow local rules about who they connect with. BubbleStorm uses a self organizing algorithm, which is performed locally on every peer and aligns nodes in a random multigraph arranged as an Euler circle. The network is randomly incrementally created using a random walk of size$\lceil 3 * (1 + \log(n)) \rceil$, where n is the network size. The overlay topology exploits heterogeneity of peers by choosing the degree of a node proportional to its bandwidth and computing power. The degree of each node should be even in order to ensure that an Euler circle exists in the multigraph and should be greater or equal than four to provide connectivity with high probability. The circular structure is used only to maintain

the integration of joining nodes and it is not crucial if the circle is broken because of a node failure.

3.2.1 Locations

In order to implement the circular structure of the overlay multigraph the concept of locations is introduced. Every graph with nodes of even degree contains Euler circle. Each instance of a node in the Euler circle of the multigraph is called location. The circle in Figure 2 is the corresponding Euler circle of the overlay multigraph in Figure 1. The indexes of the nodes in the circle represent location IDs. Locations are stored in a routing table at each peer and every location has unique ID within a single routing table. There are at most two links bounded to every location. The first link points to a peer that is called **master** of the location and the second link points to a peer that is called **slave** of the location. Every connection between peers has its local and remote location. Observing the arc $C_3 - D_2 - B_1$ in Figure 2 we can say that peer D is connected on local location 2 with peers B at remote location 1 and C at remote location 3, where C is the master of the location and B is the slave. Nevertheless all nodes are equipotent, the concepts of master and slave nodes are introduced in order to serialize the join and leave operations so that race conditions are eliminated. Self loops are explicitly allowed and we say that peer C is connected on location 2 and the master of the location is the peer itself connected on location 1.

The number of locations of every peer is chosen proportionally to its bandwidth and computation power. For good robustness against peer failures a minimum of three locations per peer is recommended. The number of locations is calculated by the following formula:

$$L = \frac{L_d * min(U, D)}{S},$$

Where L_D is the desired number of locations per peer, U is the upload speed, D is the download speed and S is the desired speed per location.

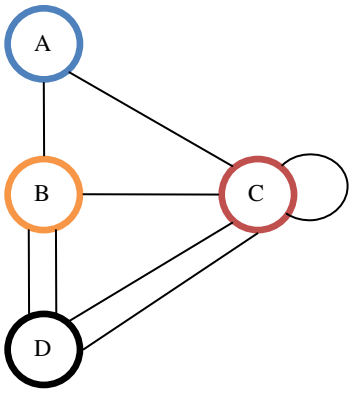


Figure 1: BubbleStorm overlay multigraph

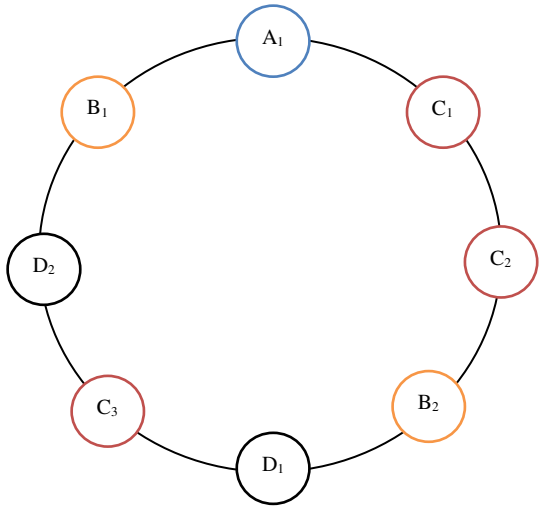


Figure 2: Overlay Euler circle

3.2.1.1 Construction and Organization

The topology uses only local information for self-organization. If a peer wants to join the BubbleStorm network it should connect to an already participating peer. A local or a web cache can be used for finding such peers. Then the joining peer creates a local location on which it wants to connect the network and sends this location to the participating peer. With the random walk technique a random edge is chosen and the joining node is placed between the nodes of the edge.

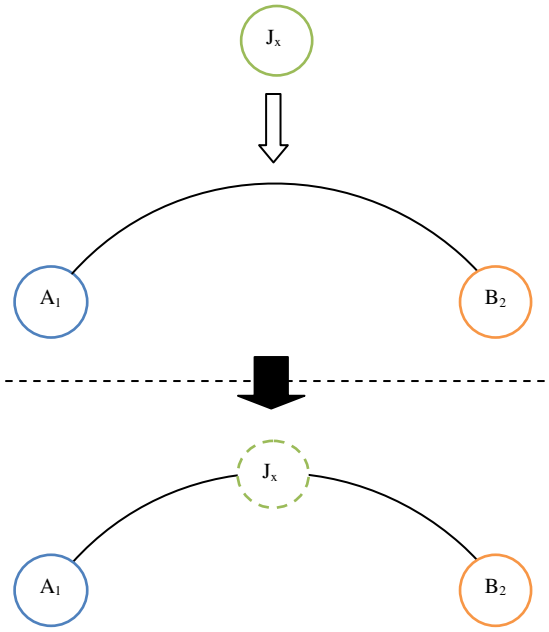


Figure 3: Joining node

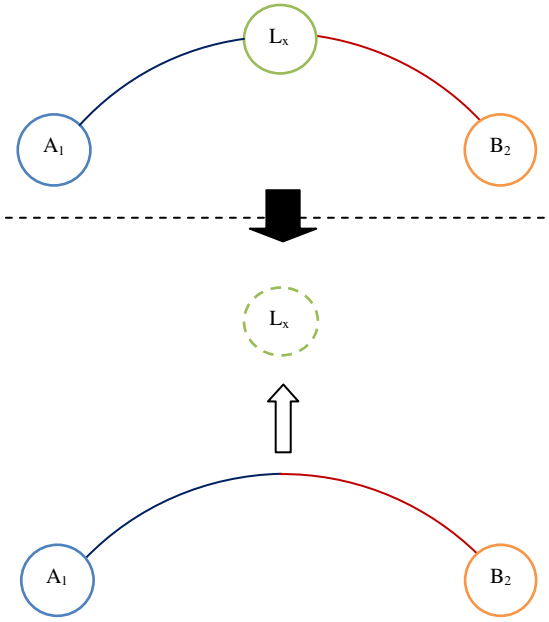


Figure 4: Leaving node

In Figure 3 peer J is joining the network. It creates locally unique location x on which it will connect. The edge $A_1 - B_2$ is randomly chosen. The edge is split and peer J on location x becomes slave of location 1 at peer A and master of location 2 at peer B. Peer A on location 1 becomes master of location x at peer J and peer B on location 2 becomes slave of location x at peer J. In Figure 4 peer L is leaving location x. Then the master and the slave link of this location are merged. A peer can be connected on multiple locations but only links of same location can be merged, otherwise the circle could be split in two separated circles.

3.3 Measurement

The protocol is used for measuring global system state. The three factors that describe the system state are the values D_0, D_1 and D_2. D_0 is the measurement of the network size and is needed for

calculation of the length of random walks performed by the join protocol. The length of the random walk is given by the formula:

$$random\ walk\ length = \lceil 3 * (1 + \log(D_0)) \rceil.$$

Values D_1 and D_2 are used by the Bubblecast algorithm in order to compute the number of query and data replicas, so that for each (query, data) pair there exists a rendezvous node. D_1 and D_2 are given by the following formula:

$$D_i = \sum_{v \in V} \deg(v)^i.$$

A precision of 5% of the calculated global state values is sufficient.

The measurement protocol distributes measurement data via keep-alive messages that are sent by each peer to all neighbors every 5 seconds.

3.3.1 Measurement Algorithm

The algorithm is based on the work by Kempe, Dobra and Gehrke and has been adapted by Terpstra, Leng and Buchmann. It converges to the sum of a variable over all nodes in O(log n) message rounds.

The algorithm is explained by the paradigm of a fisherman that puts water and a school of fish into a lake. After a while the fish is uniformly distributed over the lake. Then the fisherman can estimate the size of the lake by examining the amount of fish in a cubic meter of water. In our case we have three lakes to examine: D_0, D_1, D_2. Each peer acts as a fisherman and manages a region in each lake. Each peer picks a random fish size and puts $F = 1.0$ amount of fish in its regions. Every 5 seconds each peer sends keep-alive messages to its neighbors and mixes its regions of the lakes with the regions managed by the neighbors. When fish of different size is mixed into a region, larger fish eats smaller. After a while only the school of the biggest fish is alive and is uniformly spread in each of the three lakes. Then each peer can estimate the values D_i by dividing the measured water W_i in each lake by the amount of the fish inside the measured water.

3.3.1.1 Algorithm Initialization

The initial values for W_0 is 1.0 as W_0 is the measurement of the network size and in each region the initial number of peers is 1. The initial values of W_1 and W_2 are respectively d and d^2 where d is the degree of the peer.

3.3.1.2 Distribution of Measured Values

Every 5 seconds a peer should distribute the measured values W_i among the neighbors sending a keep-alive message to each neighbor. The distribution has two steps:

- The values W_i and F are divided into $d + 1$ parts, where d is the degree of the peer

- Each neighbor receives $\frac{1}{d+1}$ of the values and the last part stays at the peer overwriting W_i and F

3.3.1.3 Update of Measured Values

When peer receives a Ping message it should mix the incoming water that contains fish of some size with its current water and fish. The peer adds the incoming values W_i to the current ones. The updated amounts of measured water in each lake are given by the formula:

$$W_{updated_i} = W_{current_i} + W_{received_i}$$

Then it compares the size of the incoming fish with the size of its current fish.

- If the size of the incoming fish is greater than the current size, all smaller fish is eaten and F is set to the incoming amount of fish.
- If the size of the incoming fish is equal to the current size, the incoming amount of fish F is added to the current one.
- If the size of the incoming fish is less than the current size, all incoming fish is eaten and the current amount of fish F is not changed.

Let c be the size of the current fish and r the size of the received fish. Let F_j be the amount of fish of size j. The updated amount of fish is given by the following formula:

$$\tilde{F_c} = \begin{cases} F_r & r > c \\ F_r + F_c & r = c \\ F_c & r < c \end{cases}$$

3.3.1.4 Restart Measurement Round

Once the biggest fish is uniformly distributed and the measurement values are updated the protocol should restart. In order to implement this, the concepts of stability and measurement round are introduced. The measurement round lasts from the initialization of the algorithm to the time that a peer has received stable measurement information about the network. A peer considers received measurement information stable when the water/fish ratio calculated from the received values has been within a 1% tolerance for 5 keep-alive messages. When a peer receives stable measurement values it increments the stability counter. When stability counter reaches 3 the measurement round is incremented and protocol restarted. If a peer receives keep-alive message with a higher round it should save its current results, adapt its round and reinitialize its values. New peers do not add any information to the keep-alive messages until their round is incremented at least once. In order to avoid the limitation of the 32-bit values the updated round R_u is calculated by the following formula:

$$R_u = \begin{cases} R_m & G - R_c \leq 2^{16} \text{ ^} G \neq 0 \\ R_c & otherwise' \end{cases}$$

where R_m is the round of the sender of the keep-alive message, R_c is the current round of the receiver and G is calculated by:

$$G = \begin{cases} R_m + 2^{32} & R_m < R_c \\ R_m & otherwise \end{cases}$$

3.4 Bubblecast Protocol

Bubblecast is the technique used to replicate messages onto nodes. When a peer wants to insert data of type-A in the system it replicates the data onto random set of nodes sending type-A messages, so it creates a *bubble* of type-A. When another peer wants to query the data it creates a *bubble* of type-B sending messages of type-B and replicating the query onto another set of nodes. The main task of Bubblecast is to ensure the rendezvous between every type-A and type-B messages with some probability. It combines two techniques taking advantage of the controlled replication of random walks and low latency of flooding.

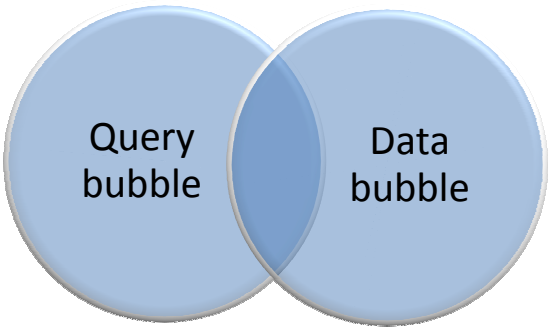


Figure 5 Overlapping bubbles.

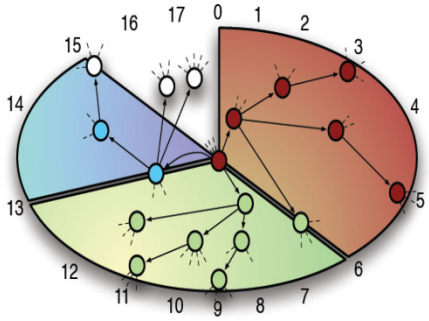


Figure 6 Bubble pies.

3.4.1 Bubblecast Algorithm

As input the Bubblecast algorithm takes the object that is to be replicated, the size of the bubble (the desired number of replicas) and value called split factor. The split factor indicates how many random neighbors are used to forward the Bubblecast message. In order to enable incremental

search of the bubbles, a range $[start, end)$ of the bubble pie slice can be defined, where *start* is the starting position of the slice and *end* is the ending position of the current bubble slice.

3.4.1.1 Bubble Size

The success of the Bubblecast depends on the existence of a rendezvous node, which has both query and data replicas. For a query-data pair(p, q), the probability r of success is given by the following formula:

$$r = 1 - e^{-s_q s_d / n},$$

where s_q is the number of query replicas is, s_d is the number of data replicas and n is the size of the network. The number of replicas depends on the required probability r. Let c be the *certainty factor* that controls this probability, defined as $c := \sqrt{-\ln(1-r)}$. The success probability as a function of c is given in Table 1. By this definition $s_q . s_d = c^2 . n$. While the product $s_q . s_d$ is determined, the ratio remains a free variable.

Table 1. Success probability as a function of c

c	1	2	3	4
$r = 1 - e^{-c^2}$	63.21%	98.17%	99.99%	99.99999%

The values s_q and s_d are calculated by the following formula:

$$s_i = c\sqrt{T . b_i}, \qquad where\ T = \frac{D_1^2}{D_2 - 2D_1}$$

The values $b_q$and b_d are called *balance factors* for messages of type q and d and are equal to the ratios R_d/R_q and R_q/R_d respectively, where $R_i$is the rate of bytes per second transmitted on average by messages of type i. In order to minimize the network traffic the sum $s_q R_q + s_d R_d$ should be minimized.

3.4.1.2 Starting a Bubblecast

When a peer wants to start a Bubblecast, it has to calculate the size of the bubble first as shown above. Then it should determine the range of the pie slice, setting the start and the end of the slice. If it doesn't want to do incremental search the start should be set to zero and the end to the overall bubble size.

3.4.1.3 Handling a Bubblecast

When a peer receives a Bubblecast message it checks if the start value of the bubble slice. If $start = 0$ then the message should be processed locally by informing the corresponding Bubblecast message handler and forwarded if needed. Otherwise the message is just forwarded as explained below.

3.4.1.3.1 Forward a Bubblecast Message

First a list of contact information of all neighbors is created and permutated. Since the message should not be sent back to the sender of the message, all occurrences of the sender's contact info are removed. Then all duplicate edges and self loops are removed and for each removed entry the counter for locally handled messages is incremented. Finally the size of the bubble and the start and the end of the bubble size are decreased by all locally handled messages. Here comes the role of the *split factor* which defines to how many neighbors the Bubblecast message is forwarded. Thus, the remaining size is divided by the split factor, the start and the end are fixed for every subslice and the messages are sent to random neighbors chosen from the list.

3.4.1.4 Matching

There exist two options to report a match to the originator of the Bubblecast message. The first option is to create a new connection to the originator and send a match message. The second is to send back the match message on the route of the Bubblecast message. Only the first option will be supported in the simulations but the second can be easily implemented if needed. Both alternatives have advantages and disadvantages which are explained below.

3.4.1.4.1 Direct Connection Match

The address of the originator is provided by the Bubblecast message, so no additional routing information is needed. The delay is lower as all results are immediately reported to the originator peer. The main disadvantage is that the originator of the Bubblecast message may be flooded with reports of redundant results which could be crucial for popular searched objects and big networks. Another problem is the lack of end-to-end connectivity of peers behind a firewall or a NAT-enabled router.

3.4.1.4.2 Routing Back Match

This alternative method is very useful for filtering redundant results. The disadvantages are that it has a higher delay and a possible failure of a peer near the sender will cause a loss of a big part of the sub-bubble when Bubblecast is in progress.

Chapter 4 BubbleStorm analysis

This chapter presents the theoretical analysis of the Bubblecast algorithm performed by Terpstra, Leng and Buchmann in [5]. The four theorems presented and proved in this chapter analyze the latency, the correctness probability, the maximal load per node and the optimality of BubbleStorm.

The random variable in the following proofs is the topology. Thus, the neighbors of a node and the content of a bubble ϕ_u^A are also random variables. On the other hand, the number of neighbors of a node and $|E|$ are fixed. The size of the bubble $|\phi_u^A|$ is fixed by the assumptions above.

4.1 Latency

The latency depends on the underlying network and since this information is not known to the system, it is measured in terms of overlay hops. The measurement algorithm provides a good estimation of the network size ($D_0 = |V|$). The time-out of a slice of bubble of type-A by incremental search is given by the following theorem.

Theorem 1. *A Bubblecast slice of type-A bubble has latency*

$$L \leq \lceil \log_2 |V| + \log_2 c + 1 \rceil = O(\log V + \log c)$$

Proof. Each Bubblecast has a latency of the longest path length. Since every node forwards the Bubblecast to at least two neighbors the Bubblecast terminates within $\leq \log_2 |A|$ hops.

$$L \leq \log_2 |A| \leq \log_2 cT \leq \log_2 (2c|V|)$$

4.2 Correctness Probability

The Bubblecast algorithm is correct for a type-A and type-B bubble if there exists a rendezvous node which received replicas of both types. In this subsection the failure probability is analyzed and proved to be $\leq e^{-c^2}$, where c is the certainty factor.

Throughout this subsection it is assumed that locations are organized in a circle possibly with broken edges. All permutations of the locations are equally likely. Thus, the location that is reached by exploring an edge is a uniform random sample chosen without replacement. Notice also that broken edges do not prevent access to specific location as the graph is a random variable.

In order to prove this we need to analyze and determine the expected size of the border of the bubble. Let ϕ_u^A is a bubble of type A around a node u. The size of the border of the bubble is

given by the degree function $deg(\phi_u^A)$ and is equal to the number of half-edges incident on the subgraph vertices excluding those within the subgraph. The next lemma shows that the size of the border is normal distributed and gives estimate of the expected value and the variance.

Lemma 2. *Let ϕ_u^A is a type-A bubble around a node u and d_v is the degree of some node v. Then the size of the bubble border given by the degree function $deg(\phi_u^A)$ is normally distributed with expected value $\mathbf{E}\big(deg(\phi_u^A)\big)$ and variance $\boldsymbol{\sigma}^2\big(deg(\phi_u^A)\big)$, where*

$$\mathbf{E}\big(deg(\phi_u^A)\big) \geq \frac{|\phi_u^A|}{2|E|}\sum_{v\in V} d_v(d_v-2) - \frac{|\phi_u^A|^2}{8|E|^2}\sum_{v\in V} d_v^3$$

$$\boldsymbol{\sigma}^2\big(deg(\phi_u^A)\big) \leq \frac{|\phi_u^A|}{2|E|-1}\sum_{v\in V} d_v^3 \approx \frac{|\phi_u^A|}{2|E|}\sum_{v\in V} d_v^3$$

Proof. Let start from node u and consequently add the nodes that belong to the bubble examining the change of the boarder size. At the beginning the border includes only node *u* and has size of d_u. Then consequently we examine the edges that connect an already examined node to newly reached node. Each newly reached node *n* increases the border size by $d_n - 2$, because the two half-edges which connect the node with the already examined nodes remain within the bubble and do not contribute to the border. Let $\boldsymbol{I}_{\phi_u^A}(v)$ is the indicator function of the subset ϕ_u^A of the set V, given by

$$\boldsymbol{I}_{\phi_u^A}(v) = \begin{cases} 1 & if\ v \in \phi_u^A \\ 0 & otherwise \end{cases}$$

Then the border size can be described as

$$deg(\phi_u^A) = \sum_{v\in V} d_v \boldsymbol{I}_{\phi_u^A}(v) - 2|\phi_u^A|$$

For indicator variables, $\mathbf{E}\left(\boldsymbol{I}_{\phi_u^A}^k(v)\right) = \mathbf{P}\big(\boldsymbol{I}_{\phi_u^A}(v)=1\big)$ for all k and therefore, $\mathbf{E}\left(\left(d_v \boldsymbol{I}_{\phi_u^A}(v)\right)^k\right) = \mathbf{P}\big(\boldsymbol{I}_{\phi_u^A}(v)=1\big)d_v^k$. The indicator variables are independent and identically distributed so we apply the central limit theorem. The theorem implies that $deg(\phi_u^A)$ is normally distributed with:

$$\mathbf{E}\big(deg(\phi_u^A)\big) = \sum_{v\in V} d_v \mathbf{P}\big(\boldsymbol{I}_{\phi_u^A}(v)=1\big) - 2|\phi_u^A|$$

$$\boldsymbol{\sigma}^2\big(deg(\phi_u^A)\big) = \sum_{v\in V} d_v^2 \mathbf{P}\big(\boldsymbol{I}_{\phi_u^A}(v)=1\big)\Big(1-\mathbf{P}\big(\boldsymbol{I}_{\phi_u^A}(v)=1\big)\Big)$$

Now we should find bounds on $\mathbf{P}\big(\boldsymbol{I}_{\phi_u^A}(v)=1\big)$. By examining all the $|\phi_u^A|$ directed edges, we mark every visited location as unreachable. Thus, at step i, only $|E|-1-i$ locations are reachable with equal probability. A node *v* has $\frac{d_v}{2}$ locations. A location is called unreached if and only if every exploration missed it.

$$\mathbf{P}\left(\boldsymbol{I}_{\phi_u^A}(v)=0\right)=\prod_{i=0}^{|\phi_u^A|-1}\left(1-\frac{\frac{1}{2}d_v}{|E|-i-1}\right)$$

Whenever $d_v > 2$ (which is by default the minimum degree),

$$1-\frac{d_v|\phi_u^A|}{2|E|-1}\leq\ \mathbf{P}\left(\boldsymbol{I}_{\phi_u^A}(v)=0\right)\leq 1-\frac{d_v|\phi_u^A|}{2|E|}+\frac{1}{2}\left(\frac{d_v|\phi_u^A|}{2|E|}\right)^2$$

Applying this inequality and the fact that $\mathbf{P}\left(\boldsymbol{I}_{\phi_u^A}(v)=1\right)=1-\mathbf{P}\left(\boldsymbol{I}_{\phi_u^A}(v)=0\right)$ we get that

$$\mathbf{E}\left(deg(\phi_u^A)\right)\geq\frac{|\phi_u^A|}{2|E|}\sum_{v\in V}d_v^2-\frac{|\phi_u^A|^2}{8|E|^2}\sum_{v\in V}d_v^3-\frac{|\phi_u^A|}{2|E|}\sum_{v\in V}2d_v$$

$$\boldsymbol{\sigma}^2\left(deg(\phi_u^A)\right)\leq\sum_{v\neq u}d_v^2\,\mathbf{P}\left(\boldsymbol{I}_{\phi_u^A}(v)=1\right)\leq\frac{|\phi_u^A|}{2|E|}\sum_{v\in V}d_v^3\quad\square$$

We will need two more definitions in order to present the main correctness theorem.

Definition 3. *The degree heterogeneity is measured by*

$$H:=\frac{\sum_{v\in V}d_v^3}{\left(\sum_{v\in V}d_v(d_v-2)\right)^{3/2}}$$

H describes the phase-transition between a distributed system and a centralized one. Nodes with relative capacity $< \sqrt{|V|}$ times the largest node are not useful participants and should connect as clients. For more uniformly distributed d_v, *H* decreases much more quickly.

Definition 4. *The extent to which a work-load's relative type-A and type-B traffic differ is measured with*

$$\boldsymbol{\Upsilon}:=\frac{1}{2}\sqrt{\frac{W_A}{W_B}}+\frac{1}{2}\sqrt{\frac{W_B}{W_A}}$$

Theorem 5. *For two arbitrary nodes, v and u, chosen independently from the connected network topology, the bubbles ϕ_u^A and ϕ_u^B fail to reach a common node with probability*

$$\mathbf{P}(failure)=\mathbf{P}(\phi_u^A\cap\phi_u^B=\emptyset)\leq e^{-c^2+c^3\Upsilon H}$$

where $H\to 0$ as $|H|\to\infty$ for $d_v=o\left(|E|^{\frac{1}{2}}\right)$.

Proof. We want to find a node that received both type of messages. By assumption $|\phi_u^A|$ is not a random variable, but nevertheless $deg(\phi_u^A)$ is. Thus, we first examine the conditional failure probability $F(\Delta) = \mathbf{P}(fail \mid deg(\phi_u^A) = \Delta)$. Later we will apply $\mathbf{E}\left(F\left(deg(\phi_u^A)\right)\right) = \mathbf{P}(fail)$ to eliminate this.

Let explore one edge in ϕ_u^B at a time while holding ϕ_u^A fixed. We are successful in finding a common node if we reach an edge in $E(\phi_u^A)$ or incident on $V(\phi_u^A)$. This must happen before step $|E| - |\phi_u^A|$. The step on which we succeed we denote by the random variable *S*.

Notice that $\mathbf{P}(S \geq 0 \mid deg(\phi_u^A) = \Delta) = 1$ and find

$$F(\Delta) = \mathbf{P}(|\phi_v^B| \leq S \mid deg(\phi_u^A) = \Delta) = \prod_{i=0}^{|\phi_v^B|-1} \mathbf{P}(S \neq i \mid S \geq i \cap deg(\phi_u^A) = \Delta)$$

$\mathbf{P}(S = 0 \mid deg(\phi_u^A) = \Delta) = \frac{|\phi_u^A|}{|E|} + 2\frac{\Delta}{2|E|} - \frac{\Delta^2}{(2|E|)^2} > \frac{\frac{1}{2}\Delta}{|E|}$ is the chance that the first added edge is a success. All subsequent edges must be incident to the first and so cannot be in ϕ_u^A, because the first one must cross the border.

Let make the following observations:

- The set ϕ_u^A contains directed edges each connecting two locations. Without crossing the border they are unreachable.
- As we explore ϕ_v^B, at step *i*, *i* edges have been explored, so *i* locations are also unavailable for a given direction. When $S \geq i$, these locations are disjoint from ϕ_u^A.
- The border of ϕ_u^A contains half-edges. Exactly half face clockwise. So, for a given exploration direction, $\frac{1}{2}\Delta$ border locations can be reach.

For $i > 0$, we explore an unbroken and unexplored edge which is incident on an explored edge. The probability that we are interested in is the chance that we have just reached a location on the border of ϕ_u^A : $\mathbf{P}(S = i \mid S \geq i \cap deg(\phi_u^A) = \Delta)$. As the circle of locations is filled by permutation all options are equally likely. Success is achieved for $\frac{1}{2}\Delta$ locations and $|\phi_u^A| + i$ of $|E|$ neighbors are not possible to select, having already assigned place in the circle. Therefore,

$$\mathbf{P}(S = i \mid S \geq i \cap deg(\phi_u^A) = \Delta) = \frac{\frac{1}{2}\Delta}{|\mathrm{E}| - |\phi_u^A| - \mathrm{i} - 1} > \frac{\frac{1}{2}\Delta}{|\mathrm{E}|}$$

Using the inequality above, we derive an estimation of $\mathbf{P}(S \neq i \mid S \geq i \cap deg(\phi_u^A) = \Delta)$. And the conditional failure probability $F(\Delta)$ is

$$F(\Delta) < \left(1 - \frac{\frac{1}{2}\Delta}{|E|}\right)^{|\phi_v^B|} < e^{-\frac{\frac{1}{2}|\phi_v^B|\Delta}{|E|}}$$

Let X the normal random variable $deg(\phi_u^A)$. For normal distributions $\mathbf{E}(e^{-kX}) = e^{-k\mu+k^2\sigma^2/2}$. If $k = |\phi_v^B|/2|E|$ then

$$\mathbf{P}(fail) = \mathbf{E}\left(F\left(deg(\phi_u^A)\right)\right) < \mathbf{E}(e^{-kX}) = e^{-k\mu+k^2\sigma^2/2}$$

$$\leq e^{-\frac{|\phi_u^A||\phi_v^B|}{4|E|^2}\left(\sum_{v\in V} d_v(d_v-2)-\frac{|\phi_u^A|+|\phi_v^B|}{4|E|}\sum_{v\in V} d_v^3\right)} = e^{-c^2+c^2\frac{|\phi_u^A|+|\phi_v^B|}{4|E|}\frac{\sum_{v\in V} d_v^3}{\sum_{v\in V} d_v(d_v-2)}} = e^{-c^2+c^3\gamma H} \quad \square$$

Subsequent attempts in the same graph are not independent; they succeed or fail depending on the previous result. Thus, there exists a BubbleStorm network that always succeeds. And the estimation of the probability that this happens is given by the following corollary.

Corollary 6. *A network always succeeds with probability*

$$\mathbf{P}(all\ pairs\ match) \geq 1 - |V|^2 e^{-c^2+c^3\gamma H}$$

where $H \to 0$ as $|E| \to \infty$ for $d_v = o\left(|E|^{\frac{1}{2}}\right)$.

Proof. The chance that opposite is true is

$$\mathbf{P}(\exists u, v \in V: \phi_u^A \cap \phi_v^B = \emptyset) \leq \sum_{u,v\in V} \mathbf{P}(\phi_u^A \cap \phi_v^B = \emptyset)$$

Now apply Theorem 5 inside the sum $|V|^2$ times. □

By setting $c = \lambda\sqrt{2\log|V|}$ as $|E| \to \infty$, we have that $\mathbf{P}(all\ pairs\ match) \geq 1 - e^{-\lambda^2}$. Thus, the system almost never fails, when $c = \boldsymbol{\Theta}\left(\sqrt{log|V|}\right)$.

4.3 Load per Node

It is assumed that the node is loaded proportionally to its degree. The message sources are independent and proportional to the edges. A workload consists of set M_A of type-A messages. The size of message $m \in M_A$ is $|m|$, thus the type-A workload is $W_A = \sum_{m\in M_A}|m|$. These messages are replicated vie Bubblecast to form bubbles ϕ_m^A.

Theorem 7. *An edge, e, chosen independently of the topology and load, carries traffic T_A of type-A with*

$$\mathbf{E}(T_A) = c\sqrt{\frac{W_A W_B}{\sum_{v\in V} d_v(d_v-2)}}$$

$$\boldsymbol{\sigma}^2(T_A) \approx \mathbf{E}(T_A)\frac{\sum_{m\in M_A}|m|^2}{W_A}$$

Proof. The traffic of type-A carried by an edge is the sum in bytes of all type-A transmitted messages.

$$T_A = \sum_{m \in M_A} |m| I_{e \in \phi_m^A}$$

The edge e is chosen independently form the source of the load, thus $\mathbf{P}(e \in \phi_m^A) = \frac{|\phi_m^A|}{|\mathrm{E}|}$ for $m \in M_A$. It follows that

$$\mathbf{E}(T_A) = \frac{|\phi^A|}{|E|} W_A = \frac{c}{|E|} \sqrt{T W_A W_B} = c \sqrt{\frac{W_A W_B}{\sum_{v \in V} d_v (d_v - 2)}}$$

The variance is $\sigma^2(T_A) = \mathbf{E}\left({T_A}^2\right) - \mathbf{E}(T_A)^2$ and it follows that

$$\boldsymbol{\sigma}^2(T_A) = \sum_{m_1, m_2 \in M_A} |m_1||m_2| \left(\mathbf{E}\left(I_{\phi_{m_1}^A} I_{\phi_{m_2}^A} \right) - \frac{|\phi_m^A|^2}{|\mathrm{E}|^2} \right)$$

If the sources of the two bubbles are different, then

$$\mathbf{P}\left(e \in \phi_{m_1}^A \cap e \in \phi_{m_2}^A\right) = \mathbf{P}\left(e \in \phi_{m_1}^A\right) \mathbf{P}\left(e \in \phi_{m_2}^A\right)$$

Unfortunately this is not true when the source is the same, but a smooth load distribution is assumed, thus

$$\boldsymbol{\sigma}^2(T_A) = \sum_{m \in M_A} |m|^2 \frac{|\phi_m^A|}{|E|} \left(1 - \frac{|\phi_m^A|}{|E|}\right) \leq \frac{|\phi^A|}{|E|} \sum_{m \in M_A} |m|^2 = \mathbf{E}(T_A) \frac{\sum_{m \in M_A} |m|^2}{W_A} \quad \square$$

Now we can calculate how well the load is distributed. Assume that $d_v \in o\left(\sqrt{|V|}\right)$ and W_A and W_b are proportional to $|V|$. Let the maximum message size is M_{max}.

Corollary 8. *Let e be an edge chosen independently of topology and load. Then* $T_A \leq k\mathbf{E}(T_A)$ *almost surely for* $k > 1$.

Proof. From Chebyshev's inequality follows

$$\mathbf{P}\left(|T_A - \mathbf{E}(T_A)| \geq (k-1)\mathbf{E}(T_A)\right) \leq \frac{\sigma^2(T_A)}{(k-1)^2 \mathbf{E}(T_A)^2} \leq \frac{\sum_{v \in V} d_v (d_v - 2)}{(k-1)^2 c \sqrt{W_A W_B}} \frac{\sum_{m \in M_A} |m|^2}{W_A}$$

$$\leq \frac{\sqrt{\sqrt{|V|}|V|}}{(k-1)^2 c \sqrt{|V||V|}} M_{max} = O\left(\frac{1}{\sqrt{\sqrt{|V|}}}\right)$$

As $|V| \to \infty$ the probability drops to zero. □

4.4 Optimality

The following lower-bound is proved in [3]. Let γ_v be the download capacity of a node.

Theorem 9. *Any system that guarantees rendezvous of every type-A and type-B message must have a node which spend relative load (load divided by capacity),*

$$t \geq 2\sqrt{\frac{W_A W_b}{\sum_{v \in V} \gamma_v}}$$

For comparison, Theorem 6 shows that when both types of traffic A and B are combined for all edges at a node

$$\mathbf{E}(T_u) = 2d_u c\sqrt{\frac{W_A W_B}{\sum_{v \in V} d_v(d_v - 2)}} \leq 2\sqrt{2}d_u c\sqrt{\frac{W_A W_B}{\sum_{v \in V} d_v^2}}$$

As the topology is designed to set node degree proportional to node capacity, $t = \frac{T_u}{d_u}$. Therefore, the result is within a constant factor of $\sqrt{2}c$ of optimal on most nodes.

Chapter 5 The PeerfactSim.KOM Simulator

PeerfactSim.KOM is a discrete event based simulator written in Java providing a benchmarking platform for peer-to-peer systems, especially for overlay networks. It is partly developed in the QuaP2P[1] and CONTENT[2] projects and released under the GNU General Public License.

5.1 Overview

PeerfactSim.KOM is a scalable, object-oriented, and light weight simulation framework. The simulator is designed modularly, extensible and scales to around 10^6 peers for simple overlays like Gnutella and 10^5 for more complex overlays like Kademlia. It provides a framework for efficient and accurate modeling of peer-to-peer protocols and applications. Moreover it is a unique evaluation platform for all kinds of overlays enabling a fair or even a less biased comparison between different overlays. The simulator also provides predefined benchmarking sets for various quality attributes with the appropriate user behavior models and output statistics. The peer-to-peer overlay developer is free to choose between various peer distributions and churn rates. The underlying network model considers geographical distances between peers, the processing delay of intermediate systems, signal propagation, congestions, retransmission and packet loss.

[1] DFG Research Project QuaP2P. www.quap2p.de

[2] CONTENT - Excellence in Content Distribution Research. www.ist-content.org

5.2 Properties

5.2.1 Modularity

Each functional part of the simulator is modularly designed in order to enable exchange with different implementations.

5.2.2 Underlay Network Model

Since the underlay network model highly influences the scalability of the simulator, all the impacts of an underlay network such as packet loss, propagation delay, congestion, etc. on peer-to-peer overlays are identified and modeled.

5.3 Architecture

In order to address effectively the complexity of peer-to-peer systems the distinction of the functional modules of the simulator is made clear and they are divided and organized in a four layer model. (Fig. 7)

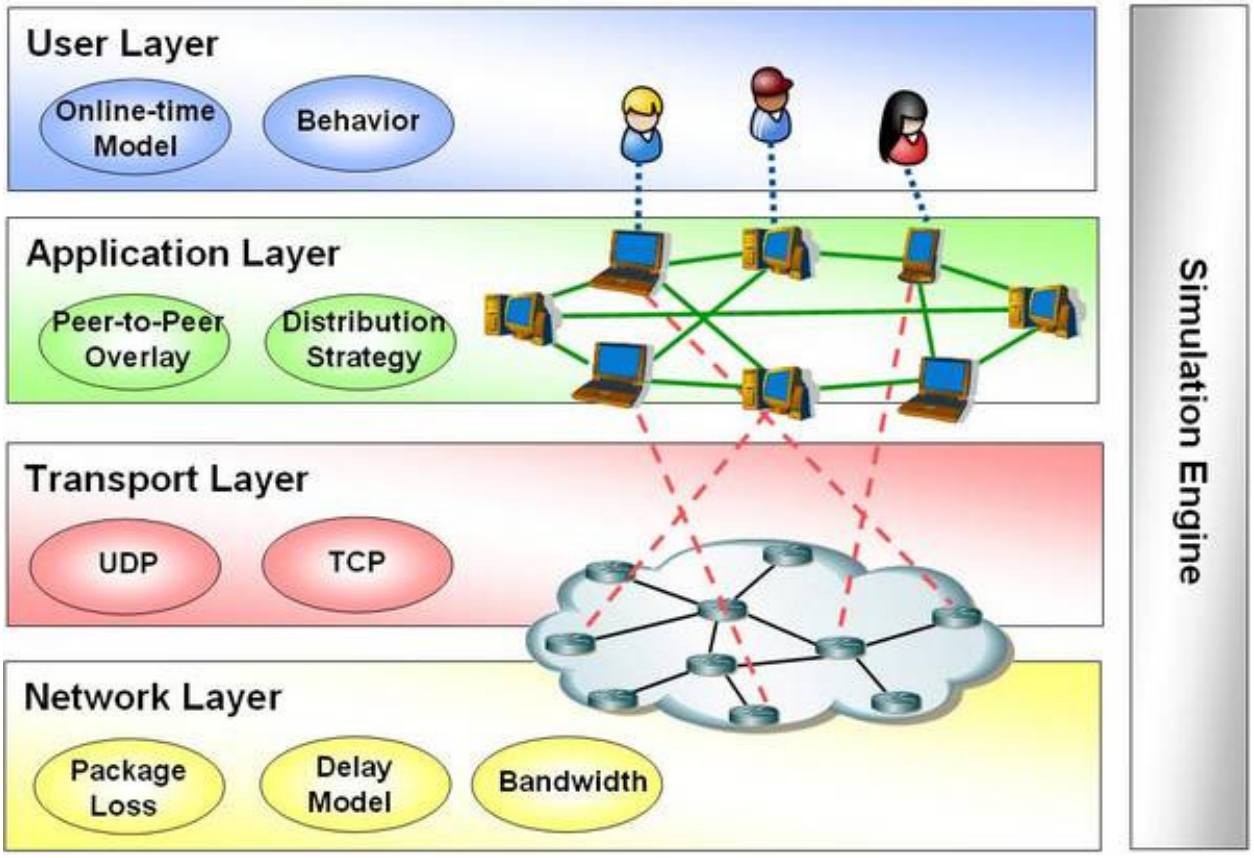


Figure 7 Functionality layers of the simulator

5.3.1 Network Layer

The focus of P2P system simulations is on the layers above the transport layer. Thus for the network layer the simulator provides a simple latency model that simulate message delivery times. The network layer takes into account most important network characteristics of end-to-end connections between peers like geographical distance, signal propagation, congestion, retransmission and packet loss. The latency model used by the simulator is described and proven valid in [9].

5.3.2 Transport Layer

The transport layer provides abstraction of UDP and TCP.

5.3.3 Application Layer

The application layer encapsulates the distribution strategy and overlay related algorithms and operations. This layer enables us to model p2p applications for content distribution, communication, and collaboration. The application layer is separated from the user layer because user behavior influences the performance of the entire system. Thus the same application model can be simulated with different user behavior models.

5.3.4 User layer

It is necessary to model the dynamic participation of peers as the stability and the performance of the overlay networks strongly depend on the churn model. The user layer is able to capture the behavior of each peer during a simulation scenario. PeerfactSim.KOM supports several important functionalities such as generating peers based on a density world map, the selection of variety of churn models and describing user behavior.

Chapter 6

Implementation

6.1 Basic Structure of the Overlay

Every overlay consists of different types of components. All needed components for the implementation of BubbleStorm within PeerFactSim.KOM are described here.

6.1.1 Node

The node is the most important component in the overlay. Nodes handle all incoming messages and message replies. For clearer structure a central class called *BSMessageHandler* is implemented. It handles the incoming messages implementing the interface *TransMessageListener* provided by the simulator.

6.1.1.1 Node Factory

The class *BSFactory* is the component factory responsible for node creation, overlayID generation and assigning hosts to nodes.

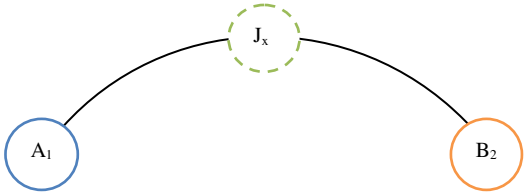
6.1.1.2 Node Identification

- overlayID

In BubbleStorm nodes don't have unique identifier within the overlay. However we need to assign a unique virtual label to every node within the simulator in order to easily track and evaluate the behavior of each node. *BSOverlayID* implements the interface OverlayID which is provided by the simulator. Currently the ID is represented by a primitive integer and the network can scale up to $2^{32} - 1 = 4, 294, 967, 295$, but the *BSOverlayID* can also easily be constructed with the universal integer storage type *BigInteger* provided by the simulator. The creation of the *BSOverlayID* is done in the *BSFactory* class, which is responsible for the node creation. A random generator provided by the simulator is used to generate unique overlay IDs.

6.1.2 Location

As it is said before the overlay multigraph is arranged as circle and the concept of location was introduced in order to implement this circular structure. The location can be understood as an instance of a peer in the Euler circle of the graph. Each peer in the overlay can have many locations in the overlay circle and each location should be unique within the routing table of the peer. The following figure presents a part of the overlay circle involving peers A, J and B connected on some locations. We will use it to understand the attributes of the location object.



The location object has the following attributes:

- locationID

Within the BubbleStorm topology nodes are identified by the *locations* they are connected on, respectively by their locations' IDs. Since the topology maintenance algorithms follow only local rules, the IDs of the locations should be unique only within the routing table of a peer. The locationID is of type Integer and is generated by *java.util.Random* class.

In the figure above *X* is the locationID of the location J_x managed by peer J.

- masterLink

The master link attribute contains the contact info (IP, port) of the master peer of the location.

In the figure above peer *A* is the master link of location J_x.

- masterRemoteLocationID

This is locationID of the location on which the master is connected.

In the figure above 1 is the masterRemoteLocationID of location J_x.

- slaveLink

The slave link attribute contains the contact info (IP, port) of the slave peer of the location.

In the figure above peer *B* is the slave link of location J_x.

- slaveRemoteLocationID

This is locationID of the location on which the slave is connected.

In the figure above 2 is the masterRemoteLocationID of location J_x.

- joinOperation

This is a reference to an ongoing join operation initiated for this location. It is null if the location is not joining.

- leaveOperation

This is a reference to an ongoing leave operation initiated for this location. It is null if the location is not leaving.

- expectsMaster

This boolean attribute indicates if the peer expects a master to connect on this location.

- expectsSlave

This boolean attribute indicates if the peer expects a slave to connect on this location.

6.1.3 Routing Table

The routing table of a peer in the BubbleStorm network consists of three collections:

- fullPeersTable (HashMap)

This collection stores information about the neighbors of a peer in the multigraph. In this data structure an *Integer* representing a *locationID* is mapped to an instance of the *Location* class. The map cannot contain duplicate keys or values. The *Location* object stores contact information about its Master and Slave peer, as well as information about the current state of the location (joining, leaving, expired, expecting master or slave, etc.)

- clients (ArrayList)

This collection stores contact information about peers that are connected as clients.

- uplinks (ArrayList)

This collection stores contact information about peers that the current peer is connected to as client.

6.1.4 Messages

Messages are used as commands from one peer to another and contain different objects that need to be exchanged between the peers. All messages extend the abstract class *BSOverlayMessage.* This class is parameterized, to concretize the type of the *OverlayID* used by all messages and some general methods are implemented. Here is the list of all used messages:

Message type	Description
ClientHello	When a peer wants to connect the BubbleStorm network it sends ClientHello to an already participating peer.
ClientOK	If a peer has received ClientHello and accepts the connection to the new peer it responds with ClientOK.
ClientDeny	If a peer has received ClientHello and rejects the connection to the new peer it responds with ClientDeny.
SplitEdge	In order to join the network on specific *location* a peer must send a SplitEdge message to a peer that it has already connection to.

MasterHello	By sending MasterHello message peer informs the receiver of this message, that the sender is the new master of an edge.
SlaveHello	By sending SlaveHello message a peer informs the receiver of this message, that the sender is the new slave of an edge.
Redirect	Redirect message is sent by the Master to its Slave in order to promote a peer as new Master on the corresponding location of the Slave
Cancel	This message is sent as reply of the Redirect message, if the Slave cannot connect to the newly promoted Master
TCPFin	This message informs the receiver that the connection is closed.
MergeEdge	MergeEdge is send by the Slave to its Master, when the Slave wants to leave a specific location
BreakEdge	BreakEdge is send by the Master to its Slave, when the Master wants to leave a broken location.
Ping	Ping message is used as keep-alive message and is send every 5 seconds to all pingable neighbors It also carries the measurement data about the network

6.1.5 Operations

An operation class, that implements the *AbstractOperation* interface, is created for every overlay-specific operation like *join, leave and ping*. Normally an operation class encapsulates all methods relevant to a certain overlay operation. However, for the join and leave operations in my implementation of BubbleStorm, this is not true. The methods responsible for joining and leaving of a node are distributed between the operation classes and the message handler class.

- JoinBSOperation

This operation is responsible for the joining of nodes to the BubbleStorm network. As input variables the operation needs the contact information of the joining peer, a list of bootstrap peers and a Boolean variable which indicates if the peer wants to connect only as client or as full peer. When the operation is started the joining peer sends ClientHello message to one of the bootstrap nodes and waits for ClientOK message as a reply. Then if the peer wants to connect to the BubbleStorm network as a full peer it should create a new location and send a SplitMessage to the bootstrap peer trying to connect on the newly created location. The operation is finished successful if the joining peer receives MasterHello or/and SlaveHello from third peers.

- LeaveBSOperation

This operation is responsible for leaving the BubbleStorm network. As input variables the operation needs the contact information of the leaving peer and the locationID of the location

that should be left. If the location has no master then the leaving peer sends BreakEdge message to its slave, otherwise it sends MergeEdge message to its master and waits for TCPFin message from the master and the slave (if any).

- PingBSOperation

The class *PingSender* schedules a ping operation every 5 seconds. The operation is responsible for distributing the measurements about the network to all neighbors of a peer using Ping messages.

- BubbleCastOperation

This operation is responsible for distributing query and data replicas and finding matches.

6.2 The BubbleStorm Protocol

This section provides a detailed definition and implementation approach of the protocol described in [4]. The protocol is built on TCP/IP and defines how the participants in the network communicate with each other. At first the topology operations like *join* and *leave* are described and afterwards the implementation approach of the Measurement and Bubblecast protocols is explained.

Since the current version of the simulator rather only simulate TCP connection with its in-order delivery then providing full TCP implementation I had to find a way to distinguish between different connections. Since locationID is unique within the routing table of a peer, the quadruple (sender IP, sender locationID, receiver IP, receiver locationID) is unique for every connection. Thus the sender and receiver locationID information is included in the messages without increasing their size.

6.2.1 The Join Protocol

A new peer has to follow the join protocol in order to connect to the BubbleStorm network. As a new peer joins the network on some new location the existing topology circle is extended. The protocol can be split into three parts. At first the new peer connects as client to some bootstrap peer, then an edge is chosen via a random walk and finally the actual joining process is performed. The methods which implement the join protocol are distributed between *JoinBSOperation* and *BSMessageHandler* class.

6.2.1.1 Connect as Client

At first the new peer tries to connect to some bootstrap peer. On execution of the JoinBSOperation the peer sends ClientHello message to the bootstrap peer and waits for reply. If the bootstrap peer can accept the new peer as client it sends back ClientOK message, if not replies with ClientDeny. If the joining peer receives ClientDeny message it should try to connect to another bootstrap node or the operation should be considered unsuccessful. When the peer receives the ClientOK message it is considered to be connected as client. The newly joined peer and the bootstrap peer should add the corresponding contact info respectively to the collection of uplinks and clients. If the operation class does not receive the expected reply the operation fails.

Once the peer is connected as client the operation can be considered as successful if or the client can proceed to connect as full peer splitting an edge in the circle of locations.

6.2.1.2 Random Walk to Find an Edge

If the client peer wants to connect as full peer it creates a new location in the routing table with unique locationID. Then it sends SplitEdge message to an uplink peer. The SplitEdge message contains the contact info of the joining peer, the new locationID and a hop counter. The hop counter is initialized with $\lceil 8 + 3\ln(network\ size) \rceil$ and if the network size is not yet known the counter is set to 0x7FFF FFFF hops and. The counter is adjusted by the first node in the forwarding process that is aware of the network size. When a peer receives SplitEdge message there are four options for handling the message depending on the hop counter of the message and the current state of the peer.

1. If the hop counter is greater than five the message is forwarded to a random neighbor peer.
2. If the hop counter is between zero and five and the peer is able to split an edge it connects to the originator of the SplitEdge message and continues with the last part of the join protocol. An edge can be split if it is not already splitting or waiting for SplitEdge to complete.
3. If the hop counter is between zero and five but the peer is not able to process a split it routes the message to a random neighbor.
4. If the hop counter reaches zero and the peer cannot process the split it discards the message. The *JoinBSOperation* started by the sender of the SplitEdge message will time out and the operation will finish unsuccessful.

6.2.1.3 Connect as Full Peer

For easier understanding of the protocol follow the references in the text to the figures below. After the joining peer (*J*) sends the SplitEdge message to find an edge to split it waits MasterHello and SlaveHello messages from the master (*A*) and the slave (*B*) (if any) of the splitting edge.

The peer (*A*) that processes the SplitEdge message is responsible for the integration of the joining peer and its new location within the virtual location circle. It chooses a random location (1) in its routing table which can be split. The peer (*A*) should become the master link of the newly created location (*x*) of the joining peer and the slave (*B*) (if any) of the splitting edge should become the slave link of the newly created location of the joining peer.

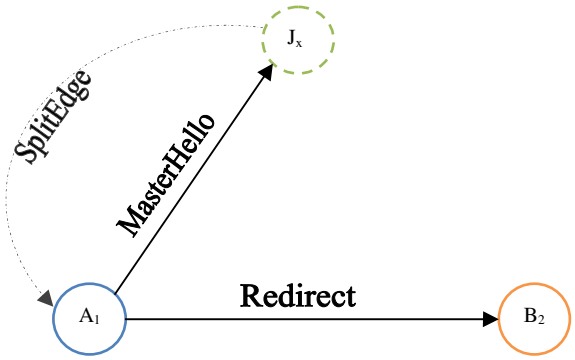


Figure 8 Connect as full peer

.

6.2.1.3.1 Master Behavior

The peer that received the SplitEdge message will be the master link of the joining location. It sends MasterHello message to the joining peer. This message tells the new peer that the sender of the message becomes master link of the new location and informs it whether the new peer should expect a new slave.

Simultaneously the master sends Redirect message to the slave (if any) of the split location in order to inform it that it should connect to the joining peer and become its slave. The Redirect message contains the contact information of the joining peer and the location it tries to connect on.

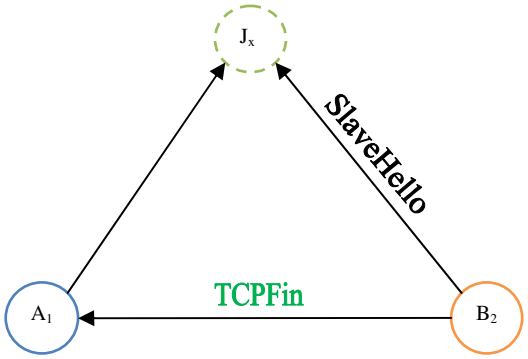


Figure 9 Successful join

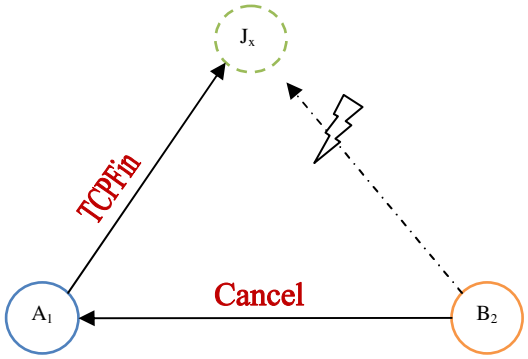


Figure 10 Unsuccessful join

6.2.1.3.2 Slave Behavior

The slave (*B*) of the splitting edge receives a Redirect message from its current master link (A_1) on some location (*2*). The message includes the contact information of the joining peer and the locationID it wants to join on. The slave peer should connect to the joining peer and send a SlaveHello message promoting itself as the slave link of the joining location. If it has successfully sent (Figure 9) the SlaveHello message the slave sets master link of the current location to be the joining node. Then the slave should send a TCPFin message to its current master. If the slave cannot connect to the joining peer (Figure 10) it sends Cancel message to its current master. Then the master closes the connection to the joining node and keeps the old slave (*B*) for its location.

6.2.2 Leave Protocol

Peers that want to exit the BubbleStorm network should follow the leave protocol. If a peer wants to leave it should merge the edge to its master and its slave for every location in its routing table.

6.2.2.1 Leaving Peer Behavior

The leaving peer sends a MergeEdge message to each master of its locations. The MergeEdge message contains contact information for the slave of the leaving peer at certain location. If the leaving peer has no slave for a location the IP, port and locationID is set to 0. On the other hand if the peer has no master for a certain location it sends a BreakEdge message to its slave. After sending the MergeEdge and/or BreakEdge messages the leaving peer waits for TCPFin message from the corresponding masters and slaves in order to successfully finish the leave procedure.

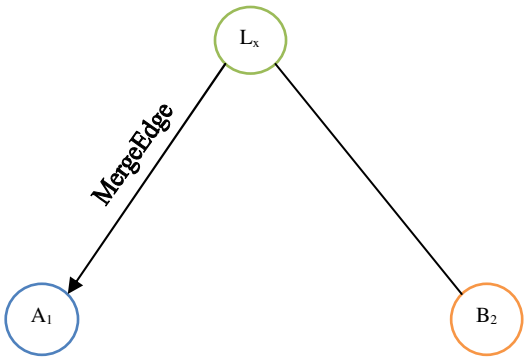


Figure 11 Receiving MergeEdge message

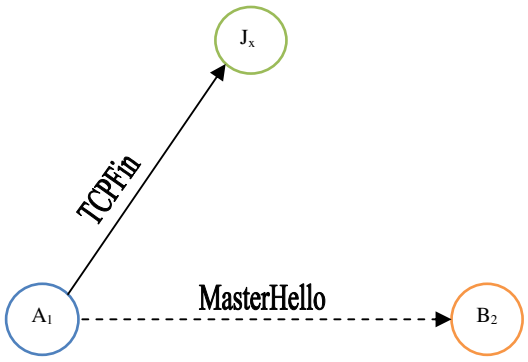


Figure 12 Sending TCPFin and MasterHello

6.2.2.2 Master Behavior

The master accepts the incoming MergeEdge message by sending back TCPFin message. It simultaneously contacts the slave of the leaving peer (taking the contact info from the MergeEdge message) and sends a MasterHello message with expectsSlave field set to *false*. Thus the master successfully replaces its slave node.

If the master receives a MergeEdge message with no information about the current slave of the leaving node this means that the leaving peer leaves a broken edge. So the master just sends TCPFin to the leaving peer and closes the connection.

6.2.2.3 Slave Behavior

When a peer receives a MasterHello message for a location with already existing master it should close the connection to the old master and promote the peer which sent the MasterHello message to be the new master.

6.2.2.4 Avoiding Race Conditions

There exist two cases where racing conditions could occur.

1. Figure 13 describes the following case. Two peers want to leave two consequent locations A_x and B_y. Peer *A* sends MergeEdge message for location *x* to peer *C*. Afterwards it receives MergeEdge message from peer *B* for the same location. Then it should simply ignore the MergeEdge message from *B*. After peer *A* has left location A_x and peer *C* has sent MasterHello to peer *B*, then peer *B* sends again the MergeEdge message this time to its new master *C*.

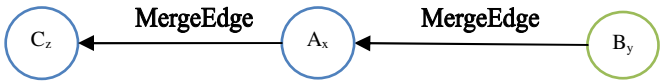


Figure 13 Race condition one

2. Figure 14 describes the second case where peer *B* wants to leave location *y* while its master *A* splits the location *x* where they are connected, because of the joining node *J*. In this case peer *A* should ignore the MergeEdge message and peer *B* should send SlaveHello message to the joining peer as defined in the join protocol. After that *B* should send the MergeEdge message to its new master *J*.

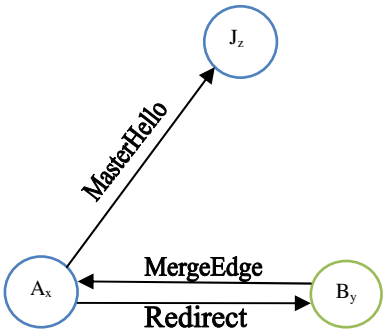


Figure 14 Race condition two

6.2.3 Measurement Protocol

Detailed description of the measurement protocol is given in section 3.3. The implementation of the protocol is distributed within the following classes.

- Pond class

This class implements the three lakes needed for our measurement.

- Measurement class

This class processes all incoming Ping messages, updates the water in the lakes and restarts the Measurement protocol if needed.

- PingBSOperation class

The ping operation class is responsible for distributing the measurement values to all neighbors vie Ping messages.

- PingMessage class

6.2.4 Bubblecast Protocol

Detailed description of the Bubblecast protocol is given in section 3.4. The methods *processBubbleCastMessage* and *forwardBubbleCastMessage* in the *BSMessageHandler* class implement the Bubblecast protocol. If a Bubblecast message has to be processed the corresponding handler is informed. Every host should have handler for each Bubblecast message type. If no handler for the received message exists then the message is simply forwarded.

Chapter 7 Evaluation

7.1 Testing

The tests are carried out on an eight processor Xeon server.

7.1.1 Configuration

The simulated network had size of about 1100 peers. Peers join the network subsequently in interval of 5 seconds following the join protocol. The network reached it size in about 9 minutes. Each peer generates and publishes a document of size 4 bytes every 110 seconds and search for some random available document every 30 seconds. Documents are considered available 20 seconds after publishing. The split factor is set to $s = 2$ and the certainty factor is set to $c = 2$. The expected success rate is $1 - e^{-c^2} \approx 98.2\%$.

7.1.2 Metrics

Four different metrics are used to evaluate the BubbleStorm network.

- **Network Size Measurement**

  The evaluation of the network size measurement will show the correctness of the measurement protocol.

- **Bubblecast Success Rate**

  The success rate of finding rendezvous node will be observed in different scenarios to prove the efficiency of the Bubblecast protocol.

- **Traffic**

  The traffic caused by different type of messages will be evaluated to be compared with the analytical predictions.

- **Latency**

  This metric shows the latency of the Bubblecast operation in different scenarios.

7.1.3 Scenarios

The network was simulated for four different scenarios.

- **Static**

This scenario provides result for a static network without churn, crash or any special events.

- **Pure Churn**

After the network reaches its size random peers are chosen to leave and join again in exponential rate.

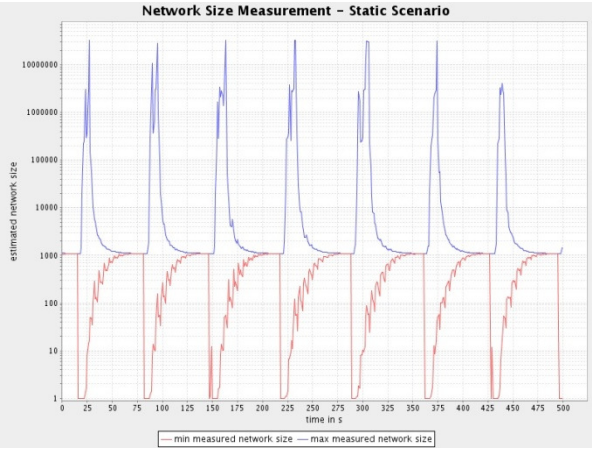
- **Massive Leave**

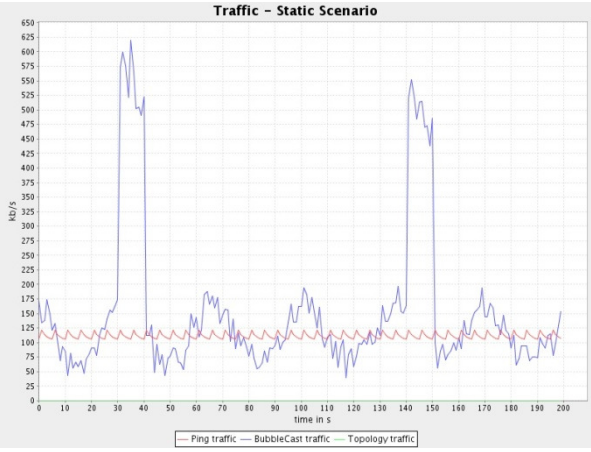
This scenario extends the pure churn scenario. In order to show the robustness of the BubbleStorm network about 50% of the peers in the network are forced to leave at some point in time.

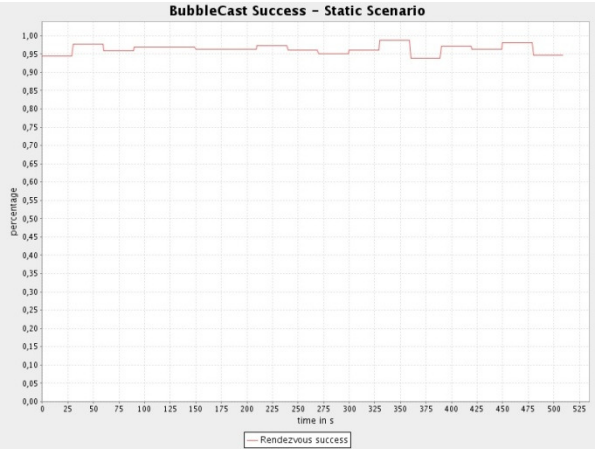
7.2 Results

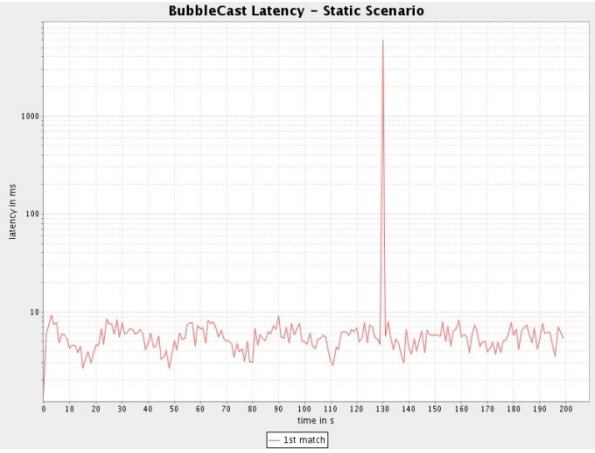
7.2.1 Static Scenario

The results showed that the measurement protocol works with good precision converging to the actual network size every 75 seconds, confirming the results of the BubbleStorm prototype. The Bubblecast success rate is under the expected 98.2%. Unsuccessful Bubblecast is caused by collisions and possible messages loss. The current implementation of the network layer of PeerfactSim.KOM showed some bugs during the simulation of BubbleStorm. The bugs of the TCP abstract implementation were fixed, but it is possible that there is still some message loss in some special cases. The Bubblecast success rate should be tested with the new implementation of the network layer which development was in progress at the time this evaluation was done. Peaks in the graph of the Bubblecast traffic are caused by massive data population every 110 seconds.



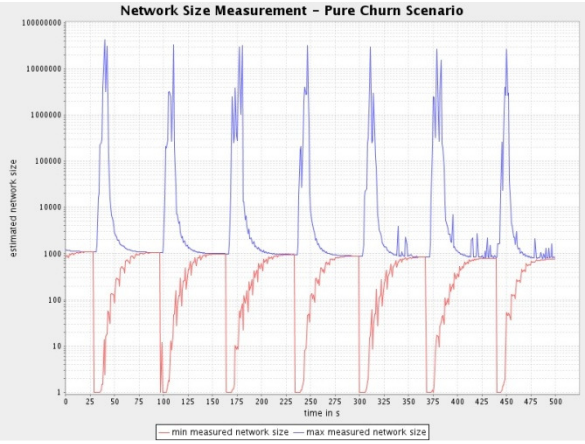


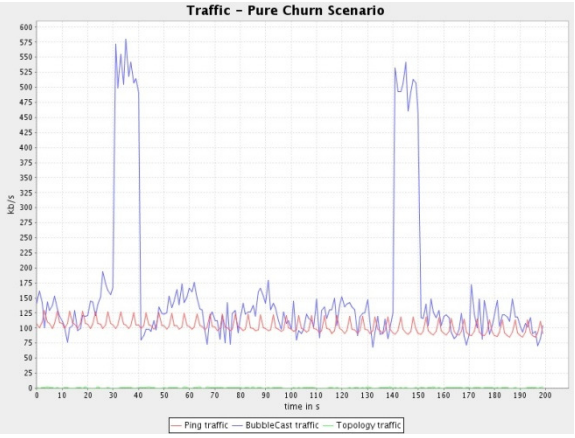


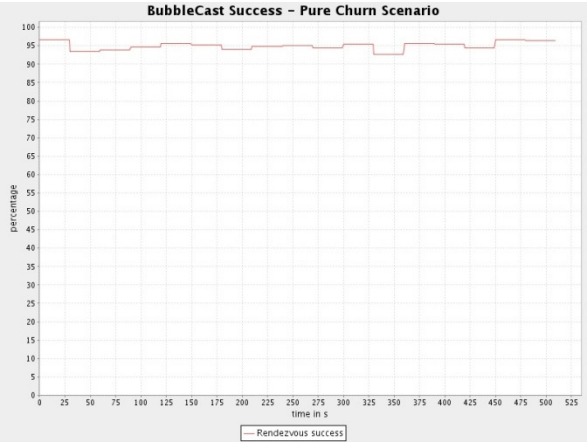


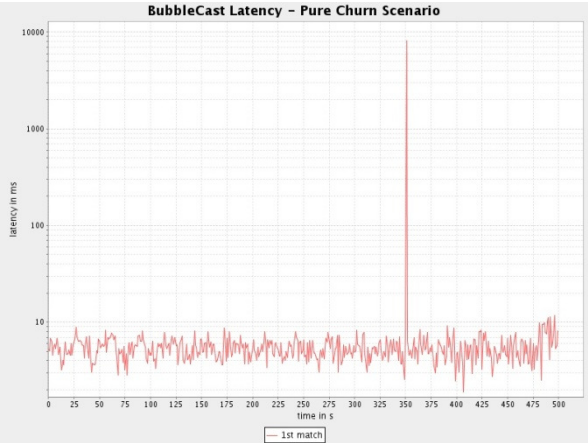
7.2.2 Churn Scenario

The exponential churn model provided by the simulator is used. The measurement protocol shows precise estimation of the network size even when peers actively join and leave the network. The Bubblecast latency remains low and the traffic results do not deviate from the results of the static scenario. The conclusion is that the system remains stable during active join and leave of peers.



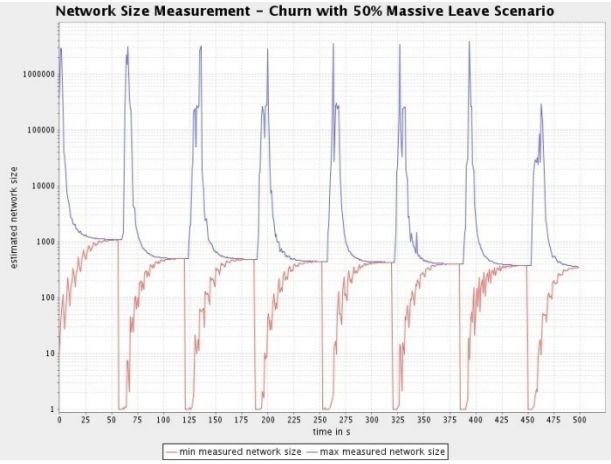


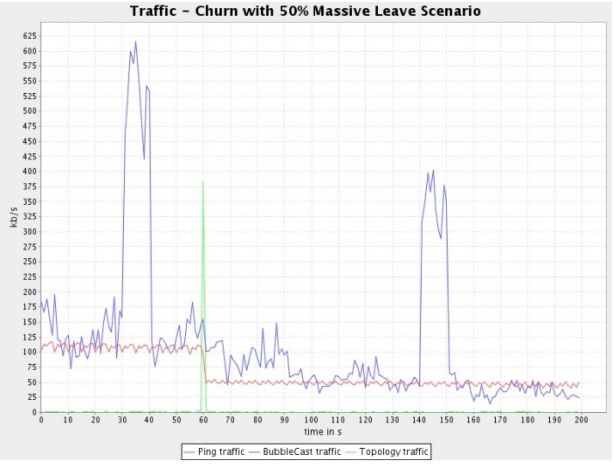


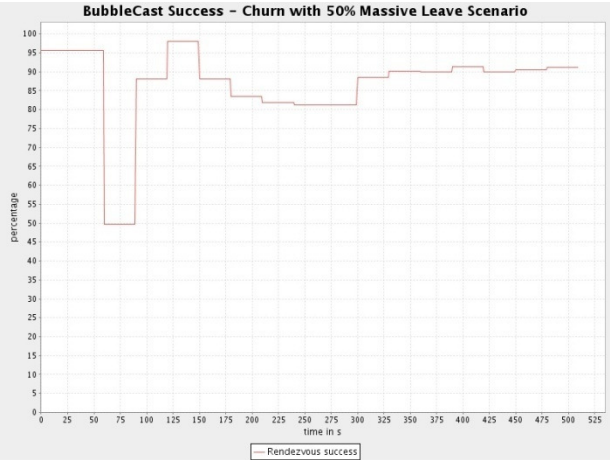


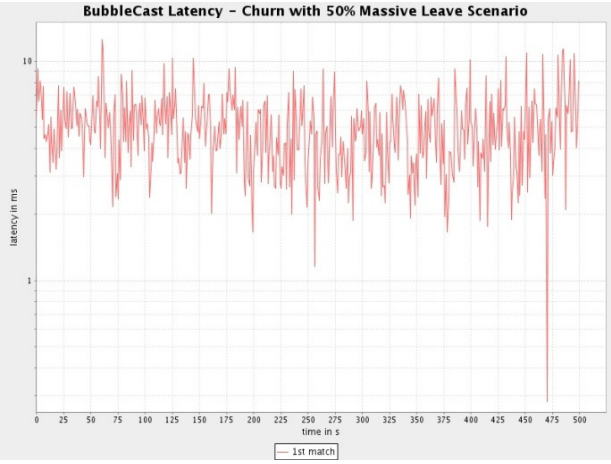
7.2.3 Leave Scenario

The exponential churn model provided by the simulator is used. In addition of the pure churn scenario, 50% of the participating peers are forced to leave the network 1 second after the start of the observation. The estimation of the network size remains precise. The success rate of the Bubblecast suffers a temporary decrease after the massive leave but in about 40 seconds the system shows quick stabilization. The traffic remains within the expected range and the latency remains low.









Chapter 8

Conclusion

This bachelor thesis presented detailed description, protocol definition, implementation and evaluation of the BubbleStorm system. In order to evaluate the system within a realistic simulated environment the BubbleStorm network is implemented within the PeerfactSim.KOM simulation framework. PeerfactSim.KOM provides an evaluation platform for efficient and accurate modeling of peer-to-peer protocols and applications and comparison between different overlays. BubbleStorm is decentralized peer-to-peer system for exhaustive search with probabilistic guarantee. The main task of this work was to provide an extendable implementation of BubbleStorm and to compare the evaluation results with the theoretical predictions and the results of the BubbleStorm prototype. The evaluated network had size of about 1100 peers and successfully handled 50% leave stabilizing quickly afterwards.

BubbleStorm seems to be ideal for complex search in heterogeneous networks. It provides a useful separation between network topology and query evaluation and the application designers

can easily create more complex and sophisticated P2P applications using any existing library for query evaluation and Client/Server algorithms. The evaluation results acquired by the simulations within PeerfactSim.KOM confirmed the theoretical results in [5] and the results provided by the prototype [11].

References

[1] Ralf Steinmetz, Klaus Wehrle. Peer-to-Peer Systems and Applications. Springer, 1st edition, 2005.

[2] Béla Bolloás. Random Graphs. Cambridge University Press, 2nd edition, 2001.

[3] Wesley W. Terpstra. Distributed Cartesian Product. Diploma thesis, Technische Universität Darmstadt, Darmstadt, Germany, May 2006.

[4] Wesley W. Terpstra, Jussi Kangasharju, Christof Leng, and Alejandro P. Buchmann. BubbleStorm: resilient, probabilistic, and exhaustive Peer-to-Peer search.

[5] Wesley W. Terpstra, Christof Leng, and Alejandro P. Buchmann. BubbleStorm: analysis of probabilistic exhaustive search in a heterogeneous Peer-to-Peer system. Technical Report TUD-CS-2007-2, Technische Universität Darmstadt, Germany, May 2007.

[6] Gerald Klunker. A Measurement-Based Approach for Realistic and Efficient Modeling of Transmission Times in Large Scale Peer-to-Peer Simulations. Diploma thesis. February 2008.

[7] Eser Esen. How to create an overlay in PeerfactSim.KOM. Technische Universität Darmstadt, Germany, February 2008.

[8] Aleksandra Kovačević, Sebastian Kaune, Nicolas Liebau, Ralf Steinmetz and Patrick Mukherjee. Benchmarking Platform for Peer-to-Peer Systems. Oldenbourg Wissenschaftsverlag, Print ISSN: 1611-2776, September 2007.

[9] O. Heckmann. A System-oriented Approach to Efficiency and Quality of Service for Internet Service Providers. PhD thesis, TU Darmstadt, 2004.

[10] Mario Schlosser, Michael Sintek, Stefan Decker, Wolfgang Nejdl. HyperCuP – Hypercubes, Ontologies and Efficient Search on P2P Networks. Computer Science Department, Stanford University. USA.

[11] Marco Swoboda. Implementation of a prototype for the BubbleStorm peer-to-peer network. Diploma thesis. Technische Universität Darmstadt, Germany, April 2008.

[12] Yatin Chawathe , Sylvia Ratnasamy , Lee Breslau , Nick Lanham , Scott Shenker. Making gnutella-like P2P systems scalable. SIGCOMM'03, August 25–29, 2003, Karlsruhe, Germany.

[13] Ion Stoica, Robert Morris, David Karger, and M. Francs Kaashoek. Chord: A scalable

peer-to-peer lookup service for internet applications. In Proceedings of COMM'01, pages 149{160, San Diego, California, United States, January 2001. ACM Press.

[14] Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. Lecture Notes in Computer Science, 2218:329, 2001.

[15] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant widearea location and routing. TR UCB/CSD-01-1141, U.C.Berkeley, CA, 2001.

[16] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. Technical Report, TR-00-010, U.C.Berkeley, CA, 2000.

[17] Ittai Abraham, Ankur Badola, Danny Bickson, Dahlia Malkhi, Sharad Malook, Saar Ron. Practical Locality-Awareness for Large Scale Information Sharing. In Proc. 4th Intl. Workshop on Peer-to-Peer Systems (IPTPS), 2005.

[18] Clip2. The gnutella protocol specification v0.4n. http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf, 2000.

[19] Fasttrack. http://www.fasttrack.nu.