Implementation of a prototype for the BubbleStorm peer-to-peer network

Marco Swoboda

August 23, 2007

Contents

Chapter 1

Introduction

At first I categorize the different kinds of peer to peer systems. After the reader got a rough idea about peer to peer systems the motivation why BubbleStorm has been developed is given. In the following the idea behind BubbleStorm system is explained. In the end of this chapter I summarise the related work which has helped to develop BubbleStorm.

1.1 Peer-to-Peer Systems

Peer-to-peer computer networks rely on the computing power and bandwidth of the participating peers. There is no need for central servers. Each peer is client and server simultaneously. The network is typically an overlay network, placed on top of an existing one, for example the Internet or a LAN. Such a network has many purposes and advantages. In contrast, a client of the common client-sever model always communicates with a central server. Servers are expected to only reply to requests of clients. There is no exchange between other clients. This makes the server an expensive hot-spot. Peer-to-peer networks take the idea to share the load between the nodes participating. This reduces hot-spots and costs for example. Possible applications, that can be build on peer-to-peer networks, could be a distributed wiki, file sharing, peer-to-peer telephony, real-time data streaming like radio and television, instant messaging, chat or multiplayer gaming.
Typically peer-to-peer systems are used to exchange messages with the peers participating in the network. In order to be scalable a peer node sends messages only to a part of the network[1] or to servers that arbitrate requests[2].

[1]This applies to decentralised strategies
[2]This is the case for centralised peer-to-peer systems

E.g. Gnutella[3] uses a limited flooding of requests. This primitive approach controls only the depth of the flood, therefore the number of hosts reached is dependent of the degree of the single nodes encountered in the flood.
Peer-to-peer systems are differentiated by their topology. Either a peer-to-peer network is centralised (hybrid) or it is decentralised (pure). Decentralised networks can be split up in structured and unstructured systems.

Types of peer-to-peer systems:

- **hybrid peer-to-peer**
  Requests of peers are processed by central servers[4]. The server acts like an index for content and arbitrates the requester to a peer that provides the requested content. The requested information is then transferred directly from peer to peer.

- **pure peer-to-peer**
  This kind of network does not have any clients and dedicated servers but only equal nodes which are called peers. These peers act as client and server at the simultaneously to the other nodes in the network.

  - **unstructured**
    In unstructured peer-to-peer networks each node is connected arbitrarily or randomly, which is the case for BubbleStorm, with nodes in the network. The construction of such networks is very easy, because a new peer can simply copy the neighbours of a known participating peer or send join messages over a random walk to a participating peer, which connects to the new peer. Most approaches in such networks use limited flooding as a means to distribute messages or requests. It has to be limited, because complete flooding would overload and dramatically slow down the entire network. On the other hand may limited flooding not always yield a satisfying result, because only a small part of the peers can process the request. This makes rare data hardly to find.

  - **structured**
    A structured peer-to-peer network typically assigns keys to data items that are mapped to the nodes that provide this content. In contrast to centralised networks the index is not kept at a specific server, but is distributed over all nodes. Each node is responsible

[3] Gnutella is a well-known decentralised unstructured network. See [11]
[4] An example for this kind of network is Napster. See [9]

for some of the key - value pairs. This improves the performance of finding items, but has also weaknesses when a peer crashes or if it is not powerful enough to handle a popular key.

1.2 Motivation

Probably the most important property of peer-to-peer systems is the search for services provided by other peers. Centralised systems are mostly only scalable under special conditions and do not fully exploit the heterogeneity of the peers. The decentralised solutions work on either structured or unstructured topologies[5]. Some systems provide exhaustive search by utilising hash tables for keywords, others allow range queries renouncing exhaustive search.
Although structured peer-to-peer systems only provide *exact match* queries, some are modified to provide *keyword* search[6]. This is only possible in an even more complex manner. The advantage is that it provides an exhaustive search, so that it allows to find even rare items if they exist in the peer-to-peer overlay network.
However, unstructured peer-to-peer systems usually don't provide exhaustive search nowadays. E.g. Gnutella, which is a well known unstructured peer-to-peer system, uses a limited flooding in order to find content. It is clear that this approach is not scalable and usually does not find rare items, if these items are not in the direct neighbourhood of the searching peer. But Gnutella-like systems have the advantage that they naturally provide keyword search and range queries, because every peer searches locally stored data and does not use a distributed hash table (DHT) like structured systems.
BubbleStorm is an unstructured peer-to-peer system, which provides exhaustive search with probabilistic guarantees, while simultaneously minimising the message overhead. Since BubbleStorm is unstructured, messages have to be evaluated a the peer that received it. Developer can create customised applications that provide keyword search and/or range queries at each node. Although analytical work has already been done [18] and the system has already been simulated without an underlying network [17], a prototype for testing BubbleStorm under real conditions. So the theoretical and simulated results can be reflected.

[5] See 1.1 for differences between these systems
[6] E.g. [14] uses keyword with a distributed hash table (DHT)

1.3 The BubbleStorm approach

Terpstra, Leng, and Buchmann developed a peer-to-peer system called BubbleStorm. This work is based on the results they present in [17, 18]. BubbleStorm is a heterogeneous peer-to-peer system that was designed to solve the rendezvous problem. It provides exhaustive search by sequential scan. Hot-spots are avoided by the topology that incorporates heterogeneity. The BubbleStorm system is a pure peer-to-peer system. It is unstructured in the sense that the placement of a node in the network is random and the nodes have no special function depending on their position in the network. This means that the arrangement of the nodes do not need a certain structure. BubbleStorm uses a self organising algorithm which align nodes in a circular connected graph in order to retain a connected graph while keeping a predefined amount of neighbours and simultaneously providing a place for new nodes.
The topology of BubbleStorm uses a random walk for establishing a multigraph. This multigraph is arranged as a circle, where the nodes are randomly permuted. Topology maintenance is simple, because its operations are local, atomic and minimally disruptive. It supports heterogeneity by choosing the individual node degree proportional to bandwidth and computing power of each peer. The degree of a node should be $\geqslant$ four in order to ensure connectivity with high probability even if some nodes fail. Euler proved that a circle exists in a graph if every node has an even degree which means that each node in the BubbleStorm multigraph should have an even degree.
BubbleCast is a communication primitive, that is built on top of the topology. It induces subgraphs (so-called bubbles) of controlled size. The BubbleCast distribution forms a bubble of nodes, that received this particular BubbleCast message. It exploits the birthday paradox for doing an exhaustive search by creating a random bubble for each meta-data and request. If a node gets both, it acts as a mediator for requester and provider. BubbleCast bubbles can be incrementally enlarged, if the current results are not satisfying yet.

1.4 Related work

In this section I abstract the work that helped to develop the BubbleStorm peer-to-peer system. At first the most important properties, efficient search and scalability, for BubbleStorm are examined. Afterwards peer-to-peer systems are researched refering to heterogeneity and user behaviour. The ideas of the last part are the basis for the measurement part of BubbleStorm.

1.4.1 Efficient Search in peer-to-peer networks

Finding information and provided services is probably the most important use of a network of distributed peers, that provide content. Therefore efficient methods for finding even rare content have been investigated.
In 2002 Cohen and Shenker [5] researched replication strategies in unstructured peer-to-peer systems in order to improve the efficiency of search in unstructured peer-to-peer networks. They assume that the access frequency is known for each item. The conclusion of this work was that the optimal replication strategy is a replication proportional to the square root of the popularity of the item. Such a system balances the load optimally. On the other hand is this system not designed for exhaustive search because such a system should not distinguish objects by their popularity.
Search and replication strategies are extensively studied in the work of Lv et al. [12] in June 2002. Besides of that they proposed a search method based on random walks and active replication combined with uniform random graph topology in order to improve unstructured peer-to-peer systems like e.g. Gnutella[11]. Instead of limited flooding they use random walks. This way they reduce the network traffic by two orders of magnitude. On the other hand a single random walk increases the response time by two orders of magnitude. They broke down the response time by using k random walks in parallel by a factor of $\frac{1}{k}$. They showed that the used random walks provide better results if they check periodically with the original requester if the content has already been found before it proceeds with the next node. They conclude that active replication[7] and uniform sampled graphs yield better results than passive replication and/or power law graphs. Simulations and analytical results are presented in their work. A objection could be that they made gross simplifications relating to a fixed query distribution and a fixed network topology and do not take exhaustive search into account.
2004 in [16] Sarshar et al. present an efficient exhaustive search for unstructured peer-to-peer systems like e.g. Gnutella[11]. A random walk is used for data replication. As a means of doing a search, queries are at first installed along a random walk followed by partially flooding of the installed queries. The flooding utilises a probabilistic broadcast scheme, which is called percolation search. The system was simulated and analysed and is designed to perform scalable searches in random power-law networks with a very high probability of finding even rare items.
The problem of locating any object independent of its access frequency (*ex-*

[7] Active replication means that meta information on content, which is available at certain nodes, is stored at a set of arbitrary nodes. While passive replication means that data is replicated at a certain node only if the node requested this content.

haustive search) in an efficient manner was examined by Ferreira et al. [7]. Here they tried to make use of the birthday paradox by locating object references on a set of $O(\gamma\sqrt{n})$ randomly selected peers, where n is the network size and γ is a free definable system parameter. They guarantee a success probability of $e^{-\gamma^2}$. For distributing the references they used a random walk of length $\Omega(\log n)+$ *k more hops*. The random walk uses an adaptation of the Metropolis-Hastings algorithm [1, 8]. Ferreira et al. showed that it is possible to give probabilistic guarantees for a decentralised search. They used multiple parallel random walks, so the network does not tend to overload. Unfortunately, they did not take heterogeneity into account.

1.4.2 Providing Scalability

A big problem of most peer-to-peer systems is that most of them scale only under special circumstances. This means that the network overloads if too many peers are connected. There were proposed improvements that seem to work for now, but it is only a question of time until these improvements will fail, if the networks keep growing. There exist some ideas to exploit heterogeneity, which appears to be a good idea in order to make peer-to-peer networks scalable.
In March 2002 Lv, Ratnasamy, and Shenker proposed a study on heterogeneity of the Gnutella peer-to-peer system [13]. This work incorporated capacity awareness with Gnutella peers. For doing a search, a parallel random walk on power-law random graphs was used, replacing limited flooding because of its inefficiency. An attempt was made to let the highest capacity nodes also carry the most traffic. The Gnutella protocol was extended by a flow control and a topology adaptation facility. The flow control is solved by letting a $node_a$ check periodically whether it is overloaded. If it is overloaded, a $node_b$ with a high incoming rate will be selected and then redirected to another $node_c$ with much spare capacity left. If all neighbours are unavailable for redirecting the overloaded node will request $node_b$ to reduce the output rate. They examined only a passive replication strategy combined with a biased random walk that sends a request to preferably high capacity nodes, that have not already seen these requests. An implicit weak hierarchy is built, where weak nodes are placed aside powerful ones, because the probability of a hit of a certain request increases, if they send requests to high capacity nodes. The main conclusion was that a decentralised peer-to-peer system, like Gnutella, can be significantly improved, if it exploits heterogeneity.
In [4] Chawathe, Ratnasamy, Breslau, Lanham, and Shenker proposed a decentralised and unstructured peer-to-peer network called Gia. Gia is mo-

tivated by the Gnutella system. It uses one-hop replication and a dynamic topology adaptation in order to exploit heterogeneity. A token based active flow control avoids overloading of nodes. Finally, they use a biased random walk for queries. The walk is preferable directed toward high capacity nodes. The nodes in Gia do a one-hop replication, which exchanges meta-information about owned content with direct neighbours, which improves the probability that an item can be found. The topology adaptation uses a level of satisfaction for each node. If it is not fully satisfied, it tries to connect to more nodes until it is satisfied. High capacity nodes are chosen a higher minimum level of satisfaction while weak nodes use a lower level. So weak nodes are implicitly placed next to high capacity nodes, which can act as directory servers for its neighbourhood. The disadvantages of this approach is that the dynamic topology adaptation may induce a domino effect, because if $node_a$ gets a request to add a new $node_b$ to his neighbours, but the maximum number of neighbours is already reached, $node_a$ drops a random $node_c$ of its current neighbours in order to add $node_b$. As a consequence $node_c$ may become unsatisfied and connects to new $node_d$ himself. The evaluation of Gia is based on simulation and doesn't provide any analytical results.
2003 Bourassa and Holt [3] present the Swan network, which is based on random regular graphs. Bourassa and Holt claimed that their network has many desirable characteristics of random regular graphs. Among these are high connectivity, logarithmic diameter, self-healing, robustness, easy adaptation, massively scalable, and high performance. They delete nodes at random, which corresponds to a failing node, and then add another random node, in order to create a uniform sampled network structure. The Swan networks are self-healing after the crash of some nodes, by completing a d-regular graph among the remaining nodes. Swan networks have two processes for dealing with lost neighbours. First an inexpensive process that sends messages to connected nodes, and second a more expensive process that contacts nodes that are not directly connected to the node that wants to increase its amount of neighbours. If the graph remains connected, the repairs can safely use the first repair mechanism. Otherwise the more expensive method has to be used. The advantage of modelling the network as a random graph is that this method is very robust even against massive node failures.
Cooper, Dyer, and Greenhill [6] use a Markov chain modelling Swan networks, described by Bourassa and Holt in [3], which yield its analytical results. Further Cooper, Dyer and Greenhill showed that the Markov chain converges with polynomial speed for the switch operation. Finally, they give an upper bound of the convergence rate.

1.4.3 Empiric Studies

In order to build an effective peer-to-peer network, one has to know how the network is used. There also exist significant differences in the behaviour of users in peer-to-peer networks.
Saroiu et al. made a study on peer-to-peer file sharing systems in January 2002. In [15] the peers participating with Napster and Gnutella were characterised. Therefore a significant amount of peers have been observed and profiled. They measured how often peers connect and disconnect from the system, their bandwidth, latency and availability. Other than that mentioned, they analysed how much peers are willing to cooperate. The conclusion of the research was first that in peer-to-peer systems exist a significant amount of heterogeneity. Second there exist groups of peers that act more as a client and other act more as a server, although these systems were designed for symmetry of responsibility. Third, peers misreport information if the misreport will yield a significant advantage. In order to get accurate information of the peers, a peer-to-peer system must directly measure the interested information or have a method to verify the information that a peer has entered.

1.4.4 Monitoring in peer to peer networks

Measuring network characteristics can be used to optimise network usage.In [10] Kempe, Dobra and Gehrke researched into decentralised algorithms for aggregate computations like sums and averages. This algorithm may be used by peer-to-peer and sensor networks. In order to handle node count fluctuations the measurements have to be constantly repeated after the current measurement stabilised. The values that should be measured are uniformly distributed to the neighbours and the node self. If two consecutive distributed values (d_1, d_2) are nearly equal, $|d_1 - d_2| \leq \varepsilon$, then the measurement is assumed stabilised and a new round is started. Although guarantees of the results from ‘gossipping‘ are usually probabilistic, the robustness and simplicity is astounding. Each node measures the desired aggregate locally. The latest results can be spread by existing nodes to new nodes that have not yet seen any measured results. There are a few drawbacks like a higher bandwidth usage and a higher latency until the measurement stabilises compared to a central collector. Latency is only a problem in a very fast changing network, since the measurement converges with exponential speed. Results afford a weak consistency. This means, that exact values can be only obtained if there is no change in the network structure within a complete round.

1.4.5 Conclusion

Efficient Search in peer-to-peer networks

In order to improve the load and efficiency of a peer-to-peer system Cohen and Shenker investigated different replication strategies. They came to the result that the number of replications should be proportional to the square root of the popularity of that item. This approach can indeed improve the load distribution, but it is impractical because the popularity of an item can not be easily determined or calculated. This method also can not provide an exhaustive search, which may be desirable for participating peers. Lv et al. [12] tried to exchange the limited flooding search by using random walks and active replication within uniform random graphs. Unfortunately they don't take exhaustive search into account. A drawback to their approach is that random walks have a high latency. Sarshar et al. [16] use a similar approach utilising random walks and active replication for an exhaustive search. The exhaustive search method used by Ferreira et al. [7] makes use of the birthday paradox and is also based on active replication and a random walk for searching, like Lv et al. [12] and Sarshar et al. [16]. Unfortunately they do not consider heterogeneity in their work. Network reorganisation due to node failures and permanent heavy load should be solved locally at each node in order to minimise the effect on other nodes.

Scalability

Lv, Ratnasamy, and Shenker [13] proposed an enhancement to the Gnutella protocol that significantly improved the network by exploiting heterogeneity. The shortcomings are that they only use a passive replication, which leads to lower query hits and their method of building the network induces a weak formed two level hierarchy, where weak nodes should send queries to high capacity nodes. This approach is in very-large networks not scalable, because if the network grows and many high capacity nodes are on the same level, the system falls back to the original operation mode.
The idea of using a hierarchy was also taken up by Chawathe et al., who tried to build a scalable decentralised peer-to-peer system by creating an implicit hierarchy. They used an active one hop replication and a random walk for queries, in contrast to Lv et al. [13], and a dynamic topology adaption. This system is quite complex and since the dynamic topology adaption changes

the topology of the surrounding neighbours of a node, capacity shortage problems may be just forwarded, which leads to a change of the topology, started at a single node, that may involves multiple nodes. Therefore a node should be able to solve capacity shortages locally without influencing other nodes. Bourassa and Holt went an other direction, relating to the topology structure, in their SWAN [3] system. The Swan technology uses a topology based on random graphs. The random graph is created by doing a random insertion of new nodes. The changes are only local and minimally disruptive. As random graphs have good characteristics related to robustness and high connectivity, these should be implemented in peer-to-peer systems. Overall one can say that active replication with some kind of random walk is currently the only possibility to do a scalable and exhaustive search in an unstructured peer-to-peer system. Overall one can say that the structure of a peer-to-peer network as a random graph is a good choice as a structure for generic networks with heterogeneity and customisable applications, since this yields some desirable advantages. Nevertheless in case of specific applications and specific network parameters a certain structure may be exploited and therefore provide particular advantages.

Empiric Studies

Saroiu et al. presented in their work on peer-to-peer network characteristics [15] a classification of the peers incorporated into the specific networks. They showed that peer-to-peer system have to deal with heterogeneity and that a peer-to-peer network should not trust the information, that peers provide about their capabilities. Instead these systems should verify the entered information or measure the desired information themselves.

Measurements in peer to peer networks

The measurement methods, described by Kempe et al. in [10], provide good approximations for aggregates of network information. E.g., the number of nodes participating in the network can easily be measured. The main disadvantages are a very high latency and exact results can only be obtained, if the network is stable. For peer-to-peer networks, where peers often connect and disconnect, the measurements are never exact, but they have only a small deviation, that may be tolerated.

Chapter 2

The BubbleStorm System

2.1 The BubbleStorm P2P technology

2.1.1 Overview

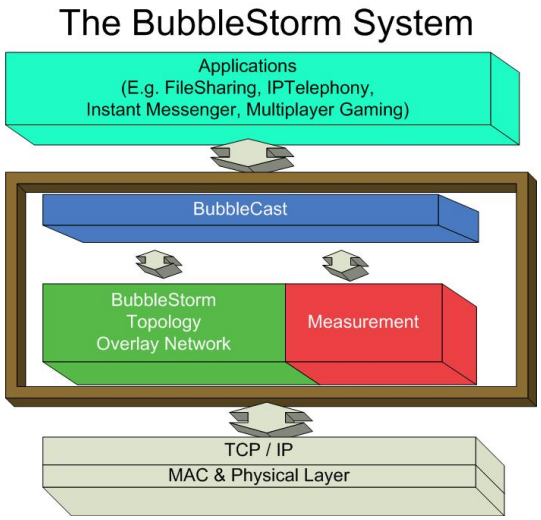


Figure 2.1: BubbleStorm overview

BubbleStorm is a peer-to-peer technology that provides a probabilistic exhaustive search in a decentralised and unstructured way. It scales by exploiting heterogeneity on random graphs.
BubbleStorm is build on TCP/IP and forms an overlay network that is randomly created. It can be used in LANs or over the Internet. The BubbleStorm system consists of three base parts plus one part for the applications

that are built on top. These parts are:

1. Topology
2. Measurement
3. BubbleCast
4. Applications building on BubbleCast

2.1.2 Locations

An important concept introduced by the topology part are locations, that are used to keep the network connected. Locations are stored in a routing table and at most two links are bound to it. Location IDs are randomly chosen, if a new peer wants to join the network. Each location has two places for links. One place is used for the master link and the other is used for the slave. So every node is in the middle of two other nodes per location. If there haven't been any crashes, these connections form a circle. A master node does not have any privileges. This concept is only used in order to serialise join and leave processes so that race conditions are eliminated. See figure 2.2 for an example of some peers connected to each other and the corresponding circle.

On the left hand in figure 2.2 is the circle that is equivalent to the con-

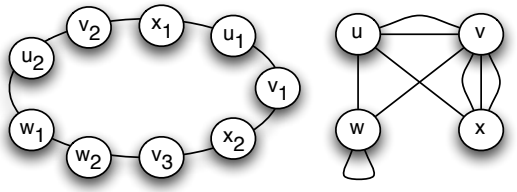


Figure 2.2: Locations in a circle

nections on the right. On the left side each index means the location of the node. So node w is connected on location 1 with node u at location 2, where node u_2 is the slave. So w is the master of the edge $w_1 \rightarrow u_2$. Also a self loop can be observed at the edge $w_1 \rightarrow w_2$, which is associated to the self loop on the right.

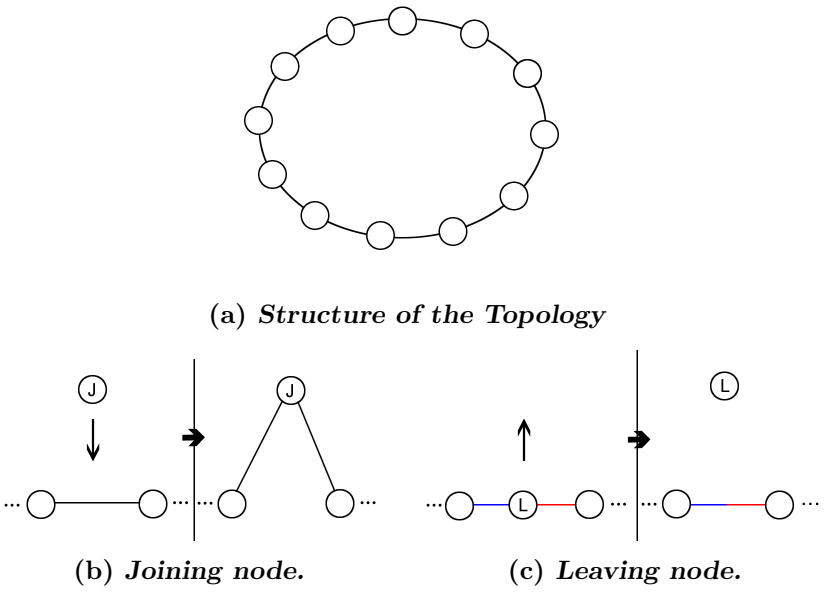
The amount of locations should be chosen by a peer proportional to its bandwidth and computation power. The absolute minimum count for locations is two, although a minimum of three is recommended to increase robustness against crashes. The locations count is calculated using the following

formula:

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

where L is the amount of locations that should be chosen, L_d is the desired minimum number of locations, U is the upload speed, D is the download speed and S is the desired speed per location, so $\frac{S}{2}$ is the estimated bandwidth per edge.

2.1.3 Topology



(a) ***Structure of the Topology***

(b) ***Joining node.***

(c) ***Leaving node.***

Figure 2.3: Topology of BubbleStorm

The topology is simple and uses only local information for self-organisation. The overlay network is incrementally randomly created using a random walk of length $3 * (1 + log(n))$, where n is the network size. The nodes are aligned in a circle like in figure 2.3(a). Each node has at least four neighbours with a tolerance of ± 1. These neighbours should be connected to some other nodes but self loops are not forbidden. E.g. see node w on the right side in figure 2.3(a). Every connection has a certain local and remote location[1]. In order to participate in the BubbleStorm peer-to-peer network, a node has to proceed the join protocol[2]. Therefore it needs to connect to a peer that

[1] for details of locations see (2.1.2)

[2] See the join protocol definition in section 2.2.3 on page 26

already participates in the BubbleStorm network. Such peers can be found using a local cache or a web cache. When a node found a peer and joins, it is placed in between a random edge. This means that the new node gets two new neighbours. Each of these two neighbours are connected to the same location, that was randomly created by the joining peer. Nodes that want to leave should proceed the leave protocol[3]. This node has to merge the two edges of each location, like in figure 2.3(c). It is important, that only edges of the same locations are merged, otherwise the circle could be separated into two circles. Using this method the graph keeps connected, which is the only reason of the circular structure.
The number of edges each peer has depends on its power. So this is the point, where heterogeneity comes into play. A more powerful peer should have more edges than a weak peer. If a peer is constantly congested, it may dynamically reduce its locations, if it is idle most of the time, it tries to increase its degree. The number of neighbour nodes must not vary in more than 2 * chosen locations ± 1. If this happens as a result of a crash, then the number has to be corrected. A SplitEdge message should be sent, if the node has less neighbours, respectively a MergeEdge is sent if we have too many neighbours.

2.1.4 Measurement

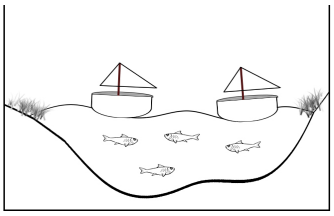


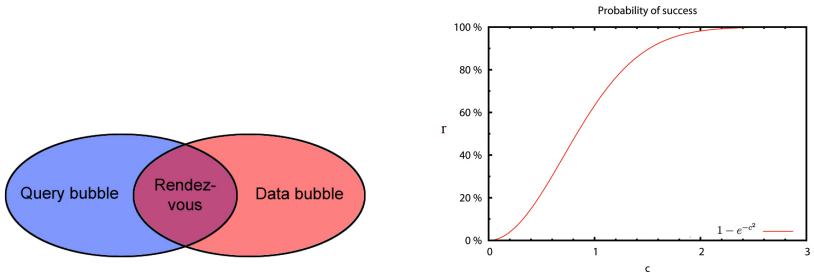
Figure 2.4: Measurement analogy

The measurement has two tasks. First it is used to estimate the network size, which is important for determining the length of random walks. Second it estimates two additional values that are important for the BubbleCast algorithm in order to choose the right bubble size. These values are D_1 and D_2, where $D_i := \sum_{v \in V} \deg(v)^i$ and V is the set of nodes that participate in the network. A precision of approximately 5% of the measured values is

[3]See the leave protocol definition in section 2.2.4 on page 36

sufficient for the needed calculations by BubbleCast and the random walk. The measurement is based on the results of the work on epidemic algorithms of Kempe, Dobra and Gehrke[10]. However, their algorithm was changed by Terpstra, Leng and Buchmann[18] in a way that no designated leader is needed anymore. A detailed description of the measurement protocol can be found in the measurement protocol definition in section 2.2.5 on page 41.

2.1.5 BubbleCast



(a) *Overlapping bubbles.* **(b) *Probabilities of success.***

Figure 2.5: BubbleCast bubbles and the probability of a success

BubbleCast is a new communication primitive that replicates information, which could be data, meta-data or queries. It combines the advantages of both random walks and flooding by controlling the complete amount of replicas with exponential incremental parallelism for distributing it. The emerging subgraph is called a bubble. The required information for building this subgraph by BubbleCast is the weight w, which corresponds to the bubble size, and the split factor s, which regulates the exponential distribution. When a BubbleCast message has been received, its weight is decreased by one and the message is locally processed by an application that builds on BubbleCast. If w is still greater than zero then s pseudo random neighbours are chosen. The weight w of the BubbleCast message is almost equally split by s so that $w = \sum_{i=1}^{s} w_i$ with $w_i \approx \frac{w}{s}$ and w_i is a whole number. The new BubbleCast messages are then sent to the previously chosen s neighbours.
If a node receives the data and a corresponding query, it can act as a mediator. The probability r that a mediator is found for any data/query association is $1 - e^{-c^2} = r$. Let d be the data bubble size and q the query bubble size. We set the failure probability $e^{-c^2} = e^{-dq/n} \Rightarrow dq = c^2 n$. Then the probability of two bubbles overlapping is $1 - e^{-dq/n}$, where n is the total number of nodes. Note that only the product of q and d is important for the success probability.

This makes it possible to do a trade-off of the query and data bubble sizes in order to reduce overall network traffic.

Choosing the Bubble size

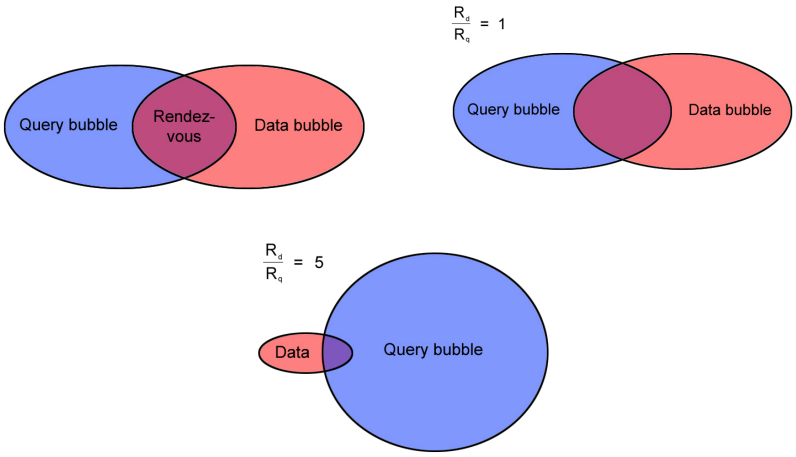


Figure 2.6: Overlapping bubbles

Let R_d be the estimated average injection rate for data bubbles and R_q the average injection rate for query bubbles. The total system traffic for BubbleCast messages, that is $dR_d + qR_q$ subject to $qd = c^2 n$, should be minimised. This leads to a balanced bubble size of $q = \sqrt{nR_d/R_q}$ and $d = \sqrt{nR_q/R_d}$[4]. So each application should provide a *balance factor* that is R_q/R_d. For now this should be a constant, but in the future this factor may be changed dynamically depending on the measured injection rate. BubbleCast samples proportional to node degree, so a powerful node that has twice the degree gets twice as many query and data messages. So the node provides a rendezvous point for four times as many query/data pairs. This means, that the bubble sizes should not depend on the number of nodes in the network, but on the number of edges. The following formula, derived in [18], gives a threshold T, that is used to determine bubble sizes, and also takes heterogeneity into account:

$$T = \frac{D_1^2}{D_2 - 2D_1}.$$

[4]See [17] for a proof of this equation

In the equation of the bubble size the node amount n should be replaced by the threshold T, so that $q = \sqrt{TR_d/R_q}$ and $d = \sqrt{TR_q/R_d}$.

Handling Collisions

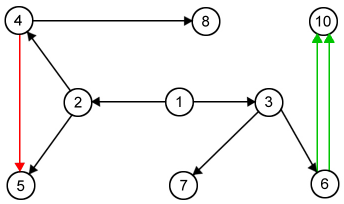


Figure 2.7: Bubble distribution.

Collisions in BubbleCast only occur if a node receives two BubbleCast messages for the same bubble[5]. Preventing collisions is important, if probabilistic guarantees are required.
A requirement in order to detect collisions is that each bubble is assigned a unique bubble ID or a hash on the message payload. If a node receives two BubbleCast messages with the same bubble ID, respectively the same has value for the payload, the node has to decide, if it is a collision or if it is only a part of the next sub-bubble, which should be processed as usual. Sub-bubbles correspond to a new pie slice in figure 2.7. No collision happens if the end of the first sub-bubble is the start of the next sub-bubble. Therefore the node must also store the end of each sub-bubble. When a real collision is detected, the node has some possibilities to react. First it may report the size to the originator of the BubbleCast and ceases further processing. The originator can then incrementally add a pie slice by sending the BubbleCast message to neighbours, that have not received this BubbleCast message yet. Unfortunately this increases latency. The second method solves the collision locally, where the collision happened. If BubbleCast uses only a small number as the split factor ($s < degree - 1$, one edge for the incoming message), it has enough spare edges to forward the BubbleCast message to.

2.1.6 Applications building on BubbleCast

Applications can be developed independently building on BubbleCast. Customised query algorithms on locally stored data can simply be implemented.

[5]See 2.1.8 for the probability of a collision to take place

Possible applications could be file sharing, distributed wiki, distributed forum, Internet telephony, instant messaging, chat, video conferencing and multi-player gaming. For details on the interface between applications and BubbleStorm see section **??**.

2.1.7 Properties of BubbleStorm

Powerful Search

BubbleStorm provides an exhaustive search, that gives probabilistic guarantees, that a rendezvous point of a query and the corresponding data is found. Since the evaluation of a query is done only on locally stored data, customised search methods (e.g. XPath or full text) on application level may be used. This is the most important advantage over DHTs, because the methods used with DHTs are very restrictive.

Scalability and Heterogeneity

By exploiting heterogeneity BubbleStorm provides a massive scalable topology, that considerably improves performance.

Optimality

To my knowledge no other system, that provides exhaustive search, has asymptotically less traffic than BubbleStorm. See [17] for a proof sketch on the optimality of BubbleStorm.

Robustness

For a peer-to-peer system node failures (crashes) are often encountered. Node failures occur when peers leave the BubbleStorm network without executing the leave protocol (see 2.2.4). This produces gaps in the created circle. These gaps are not important in order to obtain the desired properties, so the circle does not need to be repaired. In fact, the circle heals itself, if two broken edges collide. This means the a location of a peer suddenly has both edges broken, so this location can be dismissed. This makes the topology management a lot easier. It is important that two half-broken locations must not be merged because this may create two circles. The edges that are split by a node while processing a SplitEdge message should be randomly chosen. If more broken location a graph has could lead to the opinion that the nodes in the graph could be split in two or more separate graphs. In fact the more locations

break, the more looks the graph like the usual model of a random graph. Such graphs are reputed to be almost always surely connected[6].

Load Balancing

Unlike most peer-to-peer systems BubbleStorm does not introduce any hot-spots. All peers are equal super peers which may have clients. There is no keyword or hash specific destination for querys, like in DHTs. Since traffic is equally spread over the links, congested peers can dynamically reduce traffic by merging a location. On the other hand unchallenged nodes can increase their links by joining.

2.1.8 Properties of BubbleCast

Collisions

The probability of x collisions at node v is less than $\binom{size}{x}\left(\frac{d_v}{2|E|}\right)^x$, where $size$ is the total bubble size, d_v is the node degree of v and $|E|$ is the amount of all edges in the BubbleStorm network. See [18]. So the probability of two collisions on one node is very low. For high degree nodes the probability is higher, but especially these nodes should have more spare edges, due to ratio of the split factor and the node degree, which then compensates.

Latency

Since the bubbles are created exponential with exponent *split factor* s the longest path is approximately the logarithm to base s. Since $s \geq 2$ the worst case would be $s = 2$. Due to processing latency and queueing effects at the nodes, values greater than three have a negative effect. In fact a value of two seems optimally subject to decreasing latency. If we only consider the longest path of a BubbleCast message, the maximum number of overlay hops, is $L_A \leq \log_2(|size|) \leq \log_2(cT) \leq \log_2(2c|V|)$ [18], where $|size|$ is the bubble size, the threshold T is derived in 2.1.5 and $|V|$ is the number of nodes in the network.

[6] See Bollobás[2] for a proof of almost surely connected graphs

2.2 The BubbleStorm Protocol

BubbleStorm has a protocol for building an overlay network topology. It is built on TCP/IP and provides primitives for distributed searching and publishing content to the participants of the network. This mechanism is called BubbleCast. Every participant is both client and server at the same time, the so-called *peers*. BubbleStorm has the responsibility to route messages to random neighbours, which are the nodes that are directly connected to a peer. The topology forms a random multigraph[7] that is internally arranged as a connected circle. In this circle, each node can appear more than once.

2.2.1 Protocol Definition

Introduction

The BubbleStorm protocol defines how the participants in the peer-to-peer network have to communicate with each other. It consists of a set of messages. The currently defined messages are described in table 2.1. The application of these messages are explained in the appropriate sections.
A peer that wants to participate in the peer-to-peer network, has to connect to a peer, that is currently in the network. The acquisition of another peer is done via a host backed cache and a web cache. The exact location of the web cache is not yet specified and may be a configuration parameter. The first node creates self loops in order to create a circle.

2.2.2 Generic Message Properties

The following properties are valid for the message definitions below:

- All fields use **big-endian** byte order unless otherwise specified.
- All fields use **signed** 2-complement integer values unless otherwise specified.
- All 32-bit IP addresses use IPv4 format, e.g. see table 2.2 on page 26.
- All single-precision floating point numbers have 32 bit. They have the common structure, that has been defined in ANSI/IEEE Std 754-1985.

[7] In contrast to a digraph a multigraph can have more than one edge between two nodes and also self loops are allowed.

Message type	**Description of the provided messages**
SplitEdge	The SplitEdge operation is used to join BubbleStorm and to increase the amount of neighbours.
MasterHello	A peer informs the receiver of this message, that the sender is the new master of an edge.
SlaveHello	A peer sends a SlaveHello to a joining node, telling it that the sender is the slave of an edge.
Redirect	Redirect is used by a Master in order to let his slave know, that he has to redirect its message flow to another node.
Cancel	This message is used if a node fails to connect after receiving a Redirect form its master in order to tell the master to abort the joining process of the joining node.
MergeEdge	The MergeEdge message is used for an orderly leave of the network.
BreakEdge	A BreakEdge is used if a node wants to leave the network, but its master isn't available any more, then the edge between the leaving peer and the slave of this location has to be broken.
Ping	A Ping message has two purposes. First it is used as a sign that the connection is still alive. Second it is used to carry measurement data. Therefore Pings have to be sent in an equal interval by all peers. It is not required and also not desirable that the peers are synchronised, since this would lead to traffic bursts.
BubbleCast	BubbleCast is the primary communication primitive for application that are built on BubbleStorm.
ClientHello	When a new client connects it send a ClientHello to the host it connected to.
ClientOk	If a client has connected and has already send a ClientHello, the host should answer with a ClientOk if it accepts.
ClientDeny	If a client has connected and has already send a ClientHello, the host has to answer with a ClientDeny if it cannot accept any more clients.
UserConnect	This message is sent when a user connection should be established.
UserAccept	This message is the answer if the user connect request is accepted.

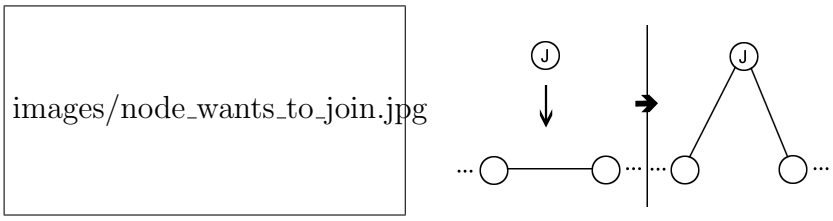
Table 2.1: Message types

Byte value	0x0A	0xAB	0x0E	0x8C
Byte offset	0	1	2	3

Table 2.2: Representation of the IPv4 address "10.171.14.140"

2.2.3 The Join Protocol

As said earlier, the peers in the BubbleStorm network are aligned in a circle. A new node has to proceed the join protocol in order to extend the circle[8]. Whether a node wants to join the network or wants to get new neighbour nodes, this node and all involved nodes will also have to proceed the join protocol. The join protocol can be split up into two parts. A random walk for choosing an edge and the joining process itself.



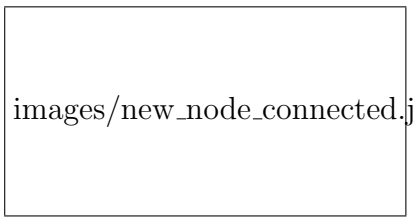
(a) ***Random walk.*** **(b)** ***Joining node.***

Figure 2.8: Joining the BubbleStorm network.

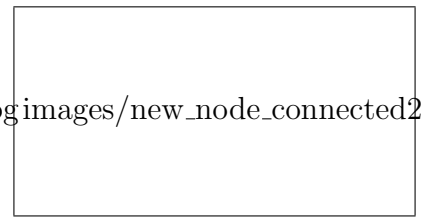
If a node isn't yet connected to network it will have to use some kind of host cache to initially connect to a host that already participates in the BubbleStorm network. The SplitEdge message must include the field location that has to be unique to the host that starts the SplitEdge message. This field is recommended to be randomly chosen, although this is not required. Then the SplitEdge message is sent over a random walk of length $8 + 3 * ln(network\ size)$ through the network. The SplitEdge message reaches the end of the random walk if the remaining hops are ≤ 5. If the node can not process the split request and the remaining hops are above zero, it handles the message like a SplitEdge messages that has a remaining hop count ¿ 5. Otherwise the message is dropped. If a node can process a SplitEdge and the hop counter is between zero and five it will choose a random available location to split. This node will become the master of the joining node. The master node and the joining node perform a TCP Handshake, that is initiated by the master. After the connection is established, the master sends a MasterHello to the joining node.

[8]See section 2.1.2 for details on the circular structure of BubbleStorm networks

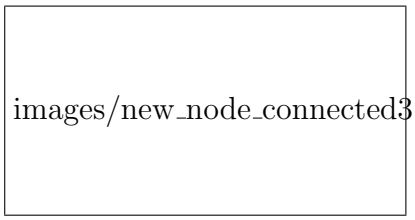
Now we have two possibilities. Either we the master has already a slave on the location that should be split or not. In the first case (figure 2.9(a)),



(a) *Split edge with neighbour.*



(b) *Redirect and MasterHello.*

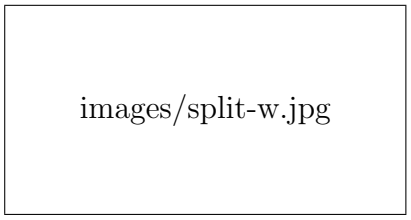


(c) *SlaveHello sent.*

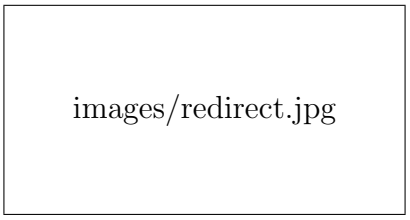
(d) *Communication ceased.*

Figure 2.9: Handling a SplitEdge 2nd phase with slave.

the field expect slave, that is contained in the MasterHello message, is set to true. Then in figure 2.9(b) the master sends a Redirect message to its slave with the information about the joining node, which it received with the SplitEdge message and the MasterHello message to the joining node. On receiving the MasterHello message, the joining node sets his new master for its location and waits for the SlaveHello from its future slave. If the future slave succeeds in connecting to the joining node, it shuts down the link to its current master and sends a SlaveHello to its new master (see figure 2.9(d)). If the future slave cannot connect to the new slave, it sends a Cancel to its



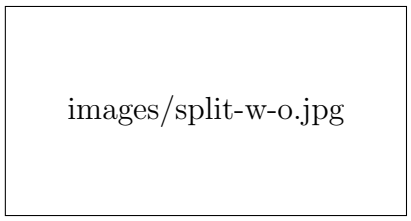
(a) *Connect to new node fails.*



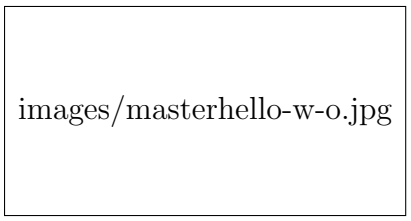
(b) *Restore original state.*

Figure 2.10: Cancelling a join process.

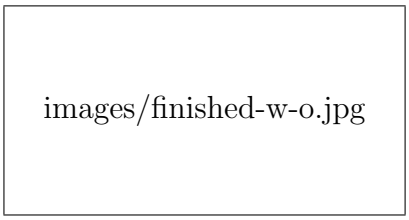
current master, which in turn closes the link to the joining node and keeps the old slave current. In the second case, see figure 2.11(a), the master sets the field expect slave to false. In this case, only one message is sent. That is



(a) *Split edge without neighbour.*



(b) *MasterHello sent to new node(II).*

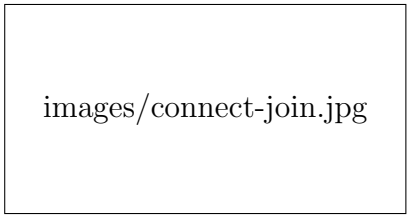


(c) *Join finished 2nd case.*

Figure 2.11: Handling a SplitEdge 2nd phase without slave.

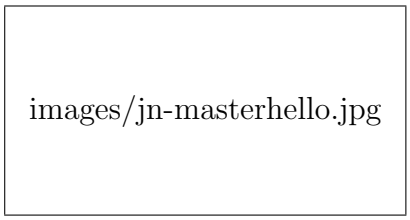
the MasterHello message, which is sent to the joining node. The join process is finished, when this message has been processed (see figures 2.11(b) and 2.11(c)).
In all MasterHello and SlaveHello messages the approximated values of the current measurement are included. Since a new node does not have any information about the network, these values are used to bootstrap the measured values. SplitEdge messages that are started without knowing the network size, which is important to choose the correct length of the random walk, have to be initialised with the value $2^{31} - 1$, which corresponds to a maximum signed 32 bit integer. The first node, that receives a SplitEdge with a number of remaining hops, that is beyond the expected value, should correct the value according to the formula mentioned above.
The needed messages for joining are:

1. SplitEdge
2. MasterHello
3. Redirect
4. SlaveHello
5. Cancel

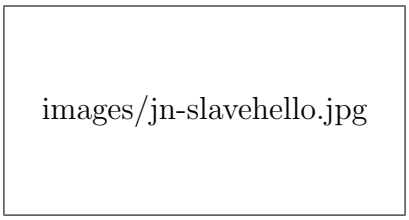


(a) *Connect to node.*

(b) *Send SplitEdge message.*



(c) *Receive MasterHello.*



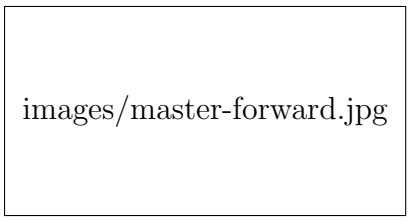
(d) *Receive SlaveHello.*

Figure 2.12: Joining BubbleStorm.

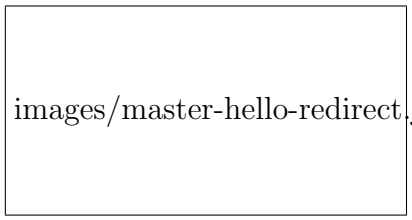
Algorithm of the corresponding nodes

Joining node At first the joining node connects to a node, that already participates in the BubbleStorm network. The remaining hop count for the SplitEdge message should be initialised with $\lceil 8+3*\ln(network\,size)\rceil$, where ln is the natural logarithm to base e. If the network size isn't known yet, the hop count should be initialised with 0x7FFF FFFF hops. For each SplitEdge a unique random location has to be generated. The value of the location has to be included in the SplitEdge message.
Then it sends the SplitEdge messages to the node to which it is connected, see figure 2.12(b). After that the node can stay connected as a client node while waiting for incoming connections. This means that this node can immediately begin searching the BubbleStorm network.
On receiving the MasterHello message in figure 2.12(c) the node sets the sender of the MasterHello as his master for the location, that was given in the MasterHello message. The location of the receiver has to exists, otherwise the master should be dropped. On receiving the SlaveHello message the sender is set as the slave for this edge, like In figure 2.12(d). If the SplitEdge message is already timed out the additional neighbours should be accepted but after the join is complete the location should be merged. If the location has not been yet been created then the sender of the message should be dropped. After that the node is fully integrated into the network.

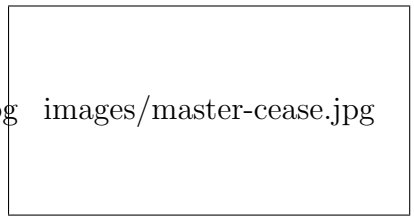
(a) ***Receiving the SplitEdge.***



(b) ***Establish connection.***



(c) ***Sending MasterHello and Redirect messages.***



(d) ***Ceasing connection to former slave.***

Figure 2.13: Master view of join.

Master Every node that gets a SplitEdge message is a potential master for the joining node. The first host that receives the SplitEdge message (See figure 2.13(a)) and reads an unexpected high hop count should correct it to a reasonable value. For example, if the network size is 1.000.000 nodes the hop count should be less than $\lceil(8 + 3 * ln(1000000))\rceil = \lceil(8 + 3 * 13.82)\rceil = 50$. So the node that knows the network size and gets a SplitEdge message with a too high value as the hop count should sets the hop count to 50 and then process the message. See section 2.1.4 for details of the calculation of the network size. In any case the hop counter is reduced by one. There are four possible ways to handle a SplitEdge depending on the remaining hop counter and the current state of the receiving node.

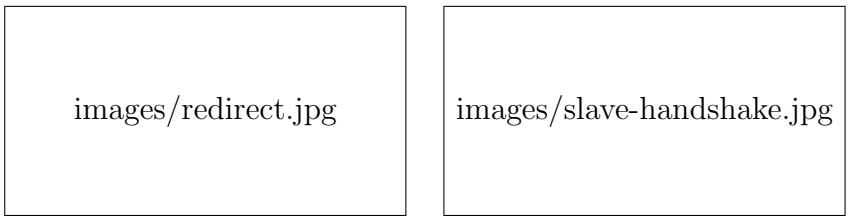
1. If the hop count of a SplitEdge message is greater than five it forwarded to a random neighbour node, like in figure 2.13(b).

2. If the node is able to split an edge, it connects to the originator of the SplitEdge message. An edge can be split, if it isn't already splitting.

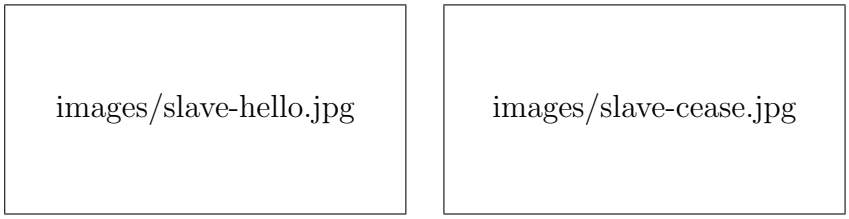
3. If the node is not able to process a split and the hop counter is still greater than zero it routes the message to a random neighbour node.

4. If the hop counter reaches zero and the node can't process a SplitEdge, it should silently discard the message. The loss will be discovered by a time out of the joining node.

The Master processes the SplitEdge by creating a connection with the joining host. Therefore it sends a TCP SYN and the other host replies with a TCP SYN ACK. After the connection is established the master sends a Master-Hello to the joining node with the corresponding locations set. This message tells the new node that it is the slave for the connection and the sender is the master. Simultaneously the master sends a Redirect message to the current slave of the edge that is currently splitting, see figure 2.13(c). The Redirect message contains the needed information about the joining node.
If the connection between the master and the current slave is shutdown with a TCP FIN signal, it should send a FIN back and close the connection, like in figure 2.13(d). After that the master sets the future slave as the current slave.



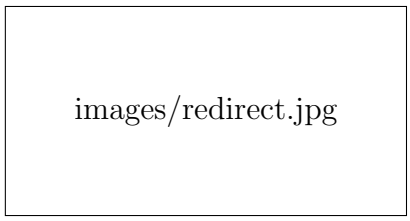
(a) ***Receiving the Redirect.*** **(b)** ***TCP Handshake.***



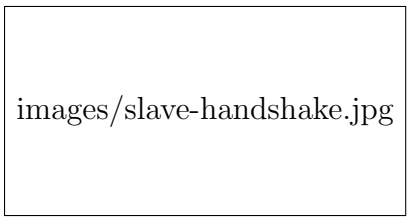
(c) ***Sending SlaveHello.*** **(d)** ***Ceasing connection.***

Figure 2.14: Slave view of join.

Slave First, in figure 2.14(a), the slave receives a Redirect message from its master. This message instructs it to change the current edge to its master to a new edge between it and the node given in the Redirect message. The new node is the future master of the edge that is going to be created.
Afterwards in figure 2.14(b), the slave tries to do a TCP handshake. If it succeeds, the slave sends a SlaveHello (figure 2.14(c)) to the future master and sets it as its master. Then, in figure 2.14(d) the slave sends a FIN to the old master. After this the connection to the old master should be closed and the current master replaced by the node that just joined.
If the connection isn't possible because of firewalls or NATs the slave should



(a) ***Send Cancel to master.***



(b) ***Restore connections.***

Figure 2.15: Slave cancels join.

send a Cancel message to the master, like in figure 2.15(a). The master should then close the connection to the joining node and keep the old slave for its location.

Clients that want to join

Connecting as client is useful for hosts that don't have full accessibility (firewall, NAT) and it is recommended for very low capacity nodes. Also nodes that want to join can stay as a client until they are promoted to a full participating node. So they can already send BubbleCast messages and don't need to experience any delay.
When a client is connected it has to send a SplitEdge, in order to participate in the network as full node, or a ClientHello in order to apply for an client connection. The client may receive a ClientOk if the peer node lets the new node stay as a client. Otherwise the client will receive a ClientDeny. SplitEdge messages from a client should be handled like a SplitEdge bundled with a ClientHello if the client hasn't been accepted yet. After the client received a ClientOk it may send SplitEdge messages, BubbleCast messages for doing a query or to post available content or meta-content. Clients usually don't send Ping messages. The only exception is when a full peer wants to leave the BubbleStorm network. Then he has end all communication with its neighbour peers and stay connected as a client or it reconnects as a client in order to send a Ping message, that contains all measurement data, which would be lost if the node simply left.
Clients may also initiate user connections which are handled in the last part about applications that are built on top of the BubbleStorm core. Other messages are currently not allowed for client nodes.
Note: A node that connects to a host should be handled like a client until it is promoted. This means that a client, which is the new master, can send a MasterHello message and a client that is the new slave can send a SlaveHello

message. Other messages except the messages mentioned above won't be sent from any host that is connected as a client.

SplitEdge

Fields	Message type	Origin IP address	Origin TCP port	Location of origin	Hops
Byte offset	0..3	4..7	8..9	10..13	14..17

Table 2.3: Fields of the SplitEdge message

Purpose This message is used by nodes that want to get new neighbours. This happens when a node joins the network and when neighbours crash. One SplitEdge sent should result in two new neighbours for the given location. It is possible due to node crashes that only one neighbour is returned or the message gets lost on the random walk through the BubbleStorm network. Therefore a node should set a timeout for SplitEdge messages and send additional messages if the timeout expires. Locations should not be reused, since this could lead to conflicts, when the a large delay is experienced.

Message type Fixed value := 0x5E000080(hex) = 1577058432 (dec)

Origin IP address The IP address of the new node that wants to join the BubbleStorm network.

Origin TCP port The TCP port on which the new host is listening for incoming connections.

Location of origin The location of the joining host that should be filled.

Hops This field contains the remaining hops of the random walk.

MasterHello

Fields	Message type	Sender location	Receiver location	Expect slave	Listen port	Measurement bootstrap
Byte offset	0..3	4..7	8..11	12	13..14	15..26

Table 2.4: Fields of the MasterHello message

Purpose A MasterHello message is used by a sender A , that wants to tell a receiver B that A is the master of the edge between A and B for the given locations. B should set A as the master for this edge and A should set B as the slave of the connection according to the locations passed in the MasterHello message.

Message type Fixed value := 0x5E000001(hex) = 1577058305(dec)

Sender location The location of the sender of this message.

Receiver location The location of the receiver of this message.

Expect slave A value of `0` sets this field to false, any other value sets this field to true. This should tell the receiver if a slave is going to connect to it.

Listen port The TCP port on which the master is listening for incoming connections. This is important since an incoming connection does not tell what port it is listening on.

Measurement bootstrap This field contains three Single-precision floating point numbers, that are used to bootstrap the values of the network size and the parameters for BubbleCast at the new node. If a new node joins, it has to use the latest stable values that were measured by a node that has already participated in the network.

Redirect

Fields	Message type	Remote IP address	Remote TCP port	Remote location
Byte offset	0..3	4..7	8..9	10..13

Table 2.5: Fields of the Redirect message

Purpose This message is used by the master of a connection, in order to instruct the slave to redirect the edge between the master and slave to the joining node. On receiving this message the slave should send a SlaveHello to the joining node.

Message type value := 0x5E000002(hex) = 1577058306(dec)

Remote IP address The IP address of the joining host to which this edge should be redirected.

Remote TCP port The TCP port on which the joining node listens.

Remote location The location that the joining node wants to allocate.

SlaveHello

Fields	Message type	Sender location	Receiver location	Listen port	Measurement bootstrap
Byte offset	0..3	4..7	8..11	13..14	15..26

Table 2.6: Fields of the SlaveHello message

Purpose A SlaveHello message is the similar to the MasterHello message. The SlaveHello message is used by a sender C that sends it to the receiver D. So D can set C as the new slave for the local location and itself as the master for the edge between C and D according to the locations passed in the SlaveHello message.

Message type value := 0x5E000041(hex) = 1577058369(dec)

Sender location The location of the sender of this message.

Receiver location The location of the receiver of this message.

Listen port The TCP port on which the slave is listening for incoming connections. This is important since an incoming connection does not tell what port it is listening on.

Measurement bootstrap This field contains three Single-precision floating point numbers, that are used to bootstrap the values of the network size and the parameters for BubbleCast at the new node. If a new node joins, it has to use the latest stable values that were measured by a node that has already participated in the network.

Cancel

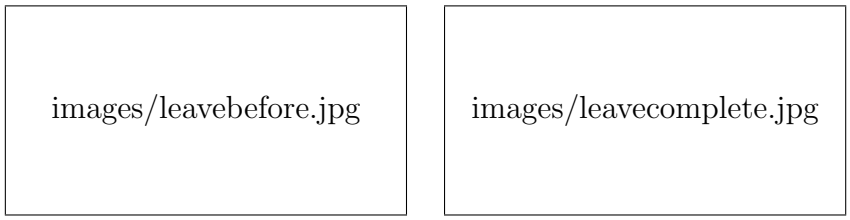
Fields	Message type
Byte offset	0..3

Table 2.7: Fields of the Cancel message

Purpose If a slave gets a Redirect message and tries to connect to the node whose data was given in the message and fails, it replies its master with a Cancel message, that in turn restores the state before the join process began.

Message type value := 0x5E000043(hex) = 1577058371(dec)

2.2.4 The Leave Protocol



(a) ***Before leave.*** **(b)** ***Merge complete.***

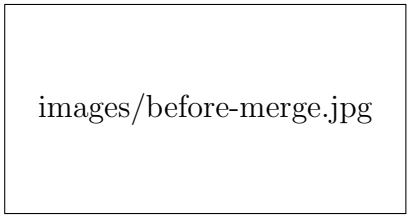
Figure 2.16: Leaving BubbleStorm.

Hosts, that want to leave the BubbleStorm network, should not just quit. Instead they should behave according to the leave protocol in order to preserve the circular structure of the network. We said earlier that each location has a master and a slave, where the node itself is placed in between these two nodes. This is the situation in figure 2.16(a). If this node wants to leave, then it should merge the edge between it and its master and the edge between it and its slave. So the master communicates with the slave and the leaving node isn't involved anymore. See figure 2.16(b) for the state after the merge operation. Edges of different location must not be merged, otherwise the ring could be transformed into two separated rings. This is especially true for broken edges.
The needed messages for leaving are:

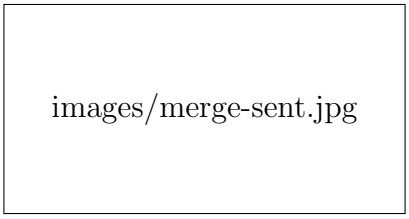
1. MergeEdge
2. MasterHello
3. BreakEdge

Algorithm for the corresponding nodes

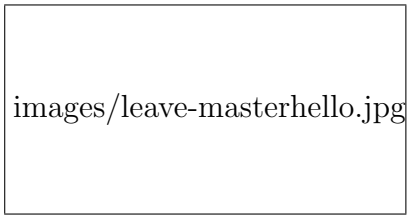
Leaving node In figure 2.17(a) we have the situation before the leave process. The nodes represented in the circular structure. At first the leaving



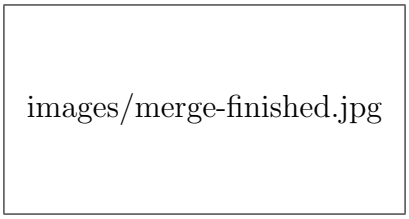
(a) ***State before leave.***



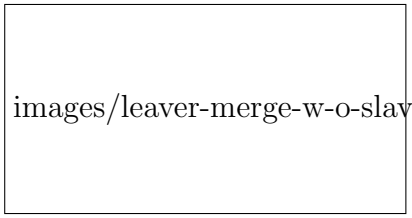
(b) ***Sending a MergeEdge to the master.***



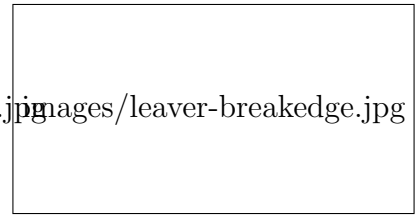
(c) ***Receiving a FIN from the master.***



(d) ***Merge complete.***



(e) ***Send MergeEdge without slave.***

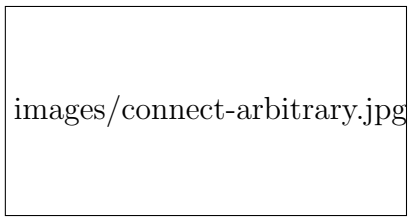


(f) ***Send BreakEdge.***

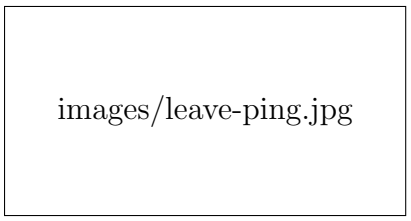
Figure 2.17: Leaving the BubbleStorm network (leaving node).

node sends for each location a MergeEdge to the master node of that location in order to request the master to let the node leave the network, like in figure 2.17(b).
The leaving node then waits for a TCP FIN signal from the master and the slave. Each FIN is replied by a FIN sent back so that the nodes are now disconnected. The final result of this procedure is shown in figure 2.17(d).
If the location has no slave to merge with, the MergeEdge message should contain the IP address 0.0.0.0 with TCP port 0 and location 0, figure 2.17(e) shows the situation when such a MergeEdge message has to be sent. If a location has no master to send a MergeEdge message to, it will send a BreakEdge message to the slave, see figure 2.17(f).

A node that wants to leave the network should send its remaining measurement information to a node that still participates in the network, see figures 2.18.



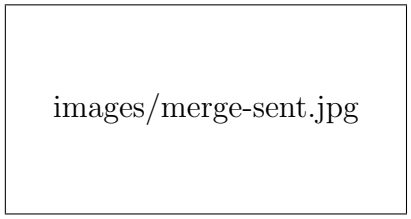
(a) *Connecting to an arbitrary node.*



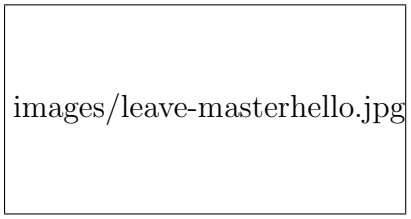
(b) *Sending a Ping.*

Figure 2.18: Saving measurement information.

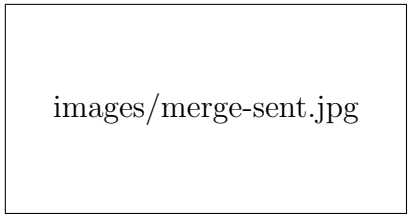
Note 1: If a node wants to leave a single location, it should preferably choose a location that has a broken edge. This should be done in order to let holes in the circle run into each other until they eliminate themselves.
Note 2: Before the clients are closed, a clients should be redirected or at least informed about the intention to leave.



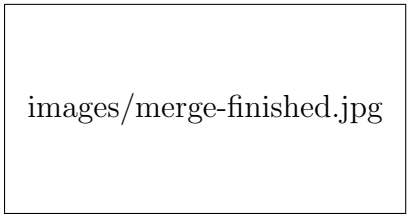
(a) *Receiving the MergeEdge.*



(b) *Sending a FIN to the leaver.*



(c) *Sending a MasterHello to the new slave.*

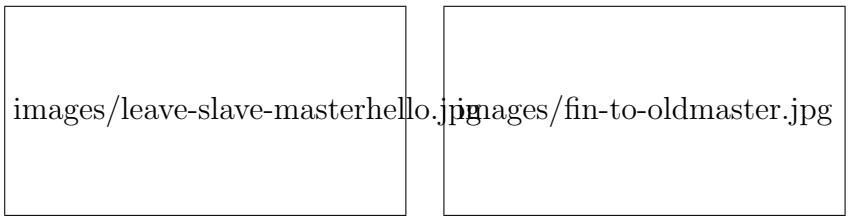


(d) *After MergeEdge with IP and TCP set to 0.*

Figure 2.19: Leaving the BubbleStorm network (master).

Master node The master accepts the received MergeEdge message (figure 2.19(a)) by sending a FIN to the leaving node (2.19(b)) that in turn responds with a FIN and closes the connection. Simultaneously, the master establishes a connection with the new slave, whose information is given in the MergeEdge message. They perform a TCP handshake. After this the master sends a MasterHello to the new slave with the expect slave field set

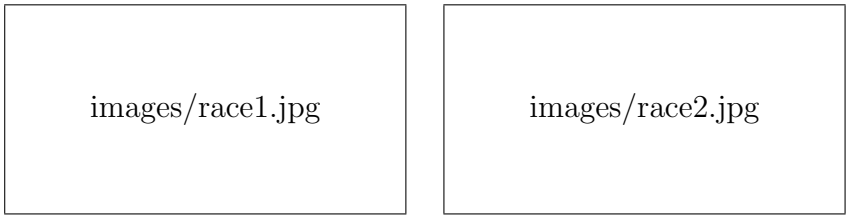
to false, see figure 2.19(c). Now the master node has successfully replaced its slave node.
If the master received a MergeEdge without any information about the replacement slave (all fields set to 0) then it just disconnects from the slave by sending a TCP FIN and awaiting the FIN replied. See figure 2.19(d) for this situation.



(a) ***Receiving a MasterHello while already having a master.*** **(b)** ***Sending FIN to old master.***

Figure 2.20: Leaving the BubbleStorm network (slave).

Slave node When a node receives a MasterHello message for a location, where the master is already present, see figure 2.20(a), it should cease the communication with the old slave and replace it with the node that has just sent the MasterHello message.



(a) ***Race condition one.*** **(b)** ***Race condition two.***

Figure 2.21: Race conditions when leaving.

Avoiding race conditions when a node leaves There exist two basic cases where race conditions could occur.
Figure 2.21(a) shows the situation for the first case, where two nodes, $node_L$ and $node_S$, want to leave simultaneously. $Node_L$ sends a MergeEdge for a certain location i to its master $node_M$ and afterwards it receives a MergeEdge message for the same local location i from its slave $node_S$. In this case $node_L$ should simply ignore the MergeEdge message from $node_S$. When $node_S$ receives a MasterHello message for the location it wanted to leave it sends a

MergeEdge message to the new master $node_M$.
Figure 2.21(b) shows the situation for the second case, where a node $node_L$ wants to leave while its master $node_M$ is splitting the location where they are connected. A node $node_L$ that sends a MergeEdge message for a certain location receives a Redirect message from its master $node_M$. Then $node_M$ should ignore the MergeEdge message and its slave $node_L$ should send a SlaveHello message to its new master $node_{NM}$ followed by a MergeEdge message for this location. If the leaving node cannot connect to $node_{NM}$ it should send a Cancel message followed by the MergeEdge back to $node_M$.

MergeEdge

Fields	Message type	Remote IP address	Remote TCP port	Remote location
Byte offset	0..3	4..7	8..9	10..13

Table 2.8: Fields of the MergeEdge message

Message type value := 0x5E000042(hex) = 1577058370 (dec)

Remote IP address The IP address of the host that should be the new slave after the merge completes. Set this field to 0 if no slave is present.

Remote TCP port The TCP port the new slave listens on. This field uses an unsigned short. Set this field to 0 if no slave is present.

Remote location The remote location of the new slave that should be replaced. Set this field to 0 if no slave is present.

Purpose This message is used by a node that wants to leave the location on which the message is sent. This message contains the address and location of the replacement for the sending node. All fields must be set to 0, if and only if the location of the leaving node has no slave.

BreakEdge

Fields	Message type
Byte offset	0..3

Table 2.9: Fields of the BreakEdge message

Message type value := 0x5E000003(hex) = 1577058307(dec)

Purpose The BreakEdge message is used if a node has no master for the location it wants to leave. As a master itself for the slave of the location it can break the edge by sending a BreakEdge to its slave.

MasterHello

The description of this message can be found in the join protocol part on page 33.
*Note: the field **expect slave** should be set to false, because no slave should connect when merging.*

2.2.5 The Ping and Measurement Protocol

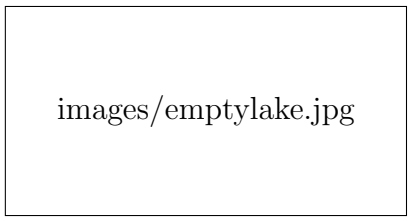
The Ping message is used for two purposes. First it is used as a keep alive signal for connected neighbours and second it is used to carry the measurement data used by the measurement protocol.

Pinging nodes

The Ping messages should be send in equal intervals of five seconds. A host is assumed dead if it hasn't sent several typicallymore than three ping messages in a row. If this happens the connection to the host should be removed from the route table and if the number of neighbours falls below the minimum acceptable amount of neighbours a new SplitEdge message has to be sent to a random remaining neighbour.

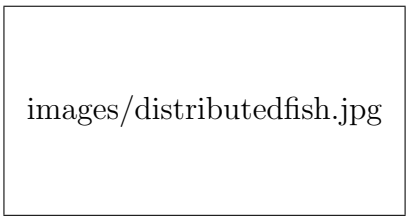
Theory of the measurement protocol

The measurement part is based on the work of Kempe et al. [10] and has been modified to eliminate the need for a single probe distributor.
The measurement uses a paradigm of fishermen that throw water and a school of fish into a lake, see figures 2.22(a) and 2.22(b). After a while when the fish is uniformly spread, they examine the water and amount of the school of fish under the fishing boat. All kinds of fish in the lake are predators that eat smaller fish but never larger ones. They are no cannibals and dislike each other. So they distribute themselves uniformly over the lake, see figure 2.22(c) for an example for such a distribution. Then the water under a boat divided by the fraction of the only fish swarm that is still left, these will be the largest fish, will be a good estimation of the size of the lake.
The size of the fish should be a randomly chosen 64 bit signed long value,



(a) ***Empty lake.***

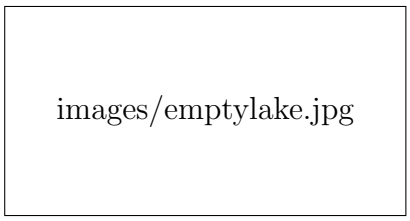
(b) ***Filled lake with different fish.***



(c) ***Equally distributed fish.***

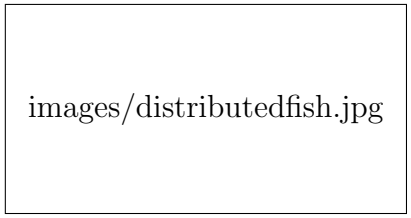
Figure 2.22: Measurement paradigm.

that is newly generated every round. Fish size collisions will hardly occurr because the highest value (biggest kind of fish) has to be randomly chosen by two independent nodes in the same round.
In our case the lake is the interested value D_0, D_1 or D_2. So we need to examine three lakes simultaneously. Fortunately we can reuse the fish. Each fisherman has to put an equal number of fish into the lake. Let this be one fish swarm $f = 100\% = 1.0$. The initial values of D_0, D_1 and D_2 have to be set according to the following rule. Since we are interested in $\sum n$, where $\sum n$ is the amount of nodes in the network, an inital value of D_0 of 1.0^9 is set. For the value of D_1 (respectively D_2) the initial values should be d (respectively d^2) where d is the current degree of the node.
In order to calculate the interested values, D_0, D_1 and D_2 have to be divide by f. So $\frac{W_i}{f} \approx \sum d^i = D_i$, for $i\epsilon\{0, 1, 2\}$, where d_i are the actual measured values.
For example, if the fish is nearly equally distributed[10] and a node gets W_0 (network size) $= 2$ and f (fraction of fish) $= 0.2$, the network size $D_0 = \frac{2}{0.2} = 10$. This means that the network has 10 nodes.



(a) ***Network.***

(b) ***Sending parts to the neighbours 1.***



(c) ***Sending parts to the neighbours 2.***

Figure 2.23: Measurement distribution.

Distribution algorithm

The distribution works as follows:

1. Every five seconds the measurement values are distributed.
2. The values f, D_0, D_1 and D_2 are divided into $d + 1$ parts.
3. Every edge of the node receives a part and one part stays at the node.

Calculating the amount of fish

The fact that big fish eat smaller fish has to modelled somehow. So let f_a be the amount of fish of size a and let r be the received fish size respectively s the stored fish size. Then the following cases may occur when receiving a Ping message.

1. The fish size in the Ping message is greater than the current fish size of the stored fish. Then the current fish is completely eaten by the larger ones.

[9] We put one node in the network.
[10] This means if $f_t - f_{t-1} \approx 0$

2. The fish size in the Ping message is equal to the fish size of the currently stored fish. Then the fish in the Ping message will be simply added to the fish that is stored.

3. The fish size in the Ping message is less than the fish size of the fish stored at the receiving node. In this case the fish of the Ping message is completely eaten and the old amount of fish will stay the current amount.

$$\hat{f}_s = \begin{cases} f_r & r > s \\ f_s + f_r & r = s \\ f_s & r < s \end{cases}$$

New nodes that enter the network in the middle of a round

The nodes that join should not put any additional values into the Ping message, instead they should just distribute the values that they receive using the algorithm described above.

Round switch

If the measured values have stabilized a new round is started. Therefore the first node that found a stable value overwrites the old network information with the newly found, then increments the round by one and reinitialises its values, that will be measured. If a node receives a Ping message with a higher round contained than the current one, it should save its current results, adapt the round and reinitialise the measured values. In order to deal with the limitation of 32 bit values, the round that should be chosen has to be adapted. A value of 0 is not allowed, since it is used by nodes that do not have any information about the network. This only used for the first few ping messages, which the receiving nodes must simply ignore with respect to the measurement. Let K be the actual round and G the seen round from an incoming Ping message. Then the following equations have to be used to calculate the current round $\hat{K}$:

$$\hat{G} = \begin{cases} G + 2^{32} & G < K \\ G & otherwise \end{cases} \qquad \hat{K} = \begin{cases} G & \hat{G} - K \leq 2^{16} \wedge G \neq 0 \\ K & otherwise \end{cases}$$

The needed messages for the measurement part are:

1. Ping

Ping

Fields	Message type	Round	Fish size	Measurement	Fish count
Byte offset	0..3	4..7	8..15	16..27	28..31

Table 2.10: Fields of the Ping message

Purpose This message is used as a keep alive signal and piggybacks measurement data.

Message type Fixed value := 0x5E000081(hex) = 1577058433 (dec)

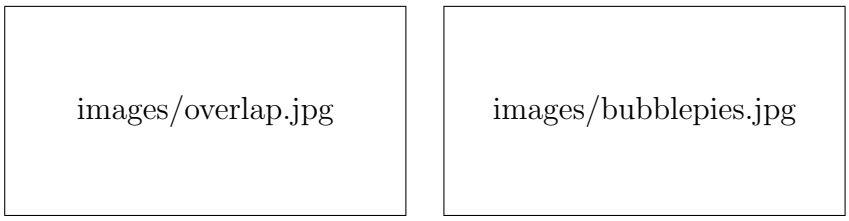
Round This is the current round for measurement

Fish size A randomly created fish size. Big fishes eat smaller ones. This field uses a signed long(8 bytes).

Measurement The measurement values. This field consists of three single-precision floating point numbers.

Fish count The fraction of (all biggest seen) fish that we want to distribute to this receiver. This field is a single-precision floating point number.

2.2.6 The BubbleCast Protocol



(a) ***Overlapping bubbles.*** **(b)** ***Bubble pies.***

Figure 2.24: BubbleCast description.

BubbleCast is used as the primary distribution method for application specific information. This distribution method gives probabilistic guarantees for a rendezvous of a query message and a matching data message. The actual match has to be done on the application level, so the matching can be adapted to the application needs.

Some of the values included in a BubbleCast message are the size, start, and end. The size describes the size of the sub-bubble that is created at each node that receives the BubbleCast message. Bubbles may be searched incrementally. Each incremental search is a pie slice, similar to figure 2.7. Each slice has a range of $[start..end)$. The start is the starting position of the slice, while the end is the ending position of the slice in the current bubble. When a host receives a BubbleCast message it processes the message locally. If the bubble size is still greater than zero, the remaining size is divided by the *split factor* and forwarded to an equal number of randomly chosen neighbours. The *split factor* is application dependent. A value of two or three is recommended in order to avoid queueing effects. A *split factor* of two would mean that each host that received a BubbleCast message would forward the message to two neighbours. The peers, to which the BubbleCast messages are sent, are pseudo randomly chosen with the origin of the message as the seed. This makes it possible to do incremental search and may be used as a countermeasure for collisions. A message is never sent back to the host from which it was received, since this would induce a sure collision.
When starting a BubbleCast two additional values are needed, which are a *certainty factor* c and A *balanced factor* b. These values are used to specify the initial size of a BubbleCast message. The *certainty factor* c controls the probabilistic guarantee r for a rendezvous of two messages. The probability r of a match is:

$$r = 1 - e^{-c^2}$$

The bubble size

The size s of the bubbles are calculated by the following formula using the values D_1 and D_2 gained from the measurement protocol[11]:

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

Let the overlap probability e^{-c^2} be equal to $e^{-s_a s_b/n}$. This means that the certainty factor $c = s_a s_b/n$ has to satisfy this condition, where n is the network size. Therefore s_a and s_b should be chosen sophisticated, so that the overall traffic can be minimised. In order to choose the optimal size the *balanced factor* has to be approximated. The good *balanced factor* is chosen using the rate of bytes per second that are transmitted on average by type a and type b messages. Let rate R_a be the rate of messages of type a and R_b the rate of type b messages. A minimal network traffic can be achieved

[11] See section 2.2.5 on page 41

by minimising $s_a R_a + s_b R_b$ subject to $s_a s_b = c^2 n$. The balanced factor for message type a equals actually the ratio $\frac{R_b}{R_a}$. A detailed analysis of the bubble size can be found in [17, 18].
The needed messages for BubbleCast are:

1. BubbleCast
2. Match

Algorithm for the involved nodes

Starting a BubbleCast If a peer wants to start a BubbleCast, it has to calculate the size according to the rule above. If it does not want to do an incremental search, then it should choose zero for the start and the end should equal the size. Each application has to define its global unique payload type. An application may use some payload types, e.g. one type for a query message and one type for data respectively meta-data messages. It has to set its TCP/IP address, which isn't changed anymore once it is set.
After all parameters have been set and the payload is attached, the BubbleCast message should be handled as described in the next paragraph.

Handling a BubbleCast The algorithm in figure 2.25 shows the handling of a BubbleCast message. It will be described here step by step. Whenever a node receives a BubbleCast with $start = 0$ it has to match the message content locally by the application that is responsible for the received bubble type, otherwise the BubbleCast message is only routed according to the following algorithm. Matching is handled later in this chapter.
The variable *local* counts the size of the BubbleCast messages that are locally handled. There are some steps that have to be done before the BubbleCast message can be forwarded.At first all neighbours have to be permuted, then the sender of the BubbleCast message is removed since we do not want to send the message back. A message sent back would cause a sure collision which should be avoided. The application specific *split factor* 'split' defines to how many neighbours the BubbleCast message is forwarded. This is shown in line 7 of the algorithm. A *split factor* value of one would BubbleCast distribute the messages over a single random walk. After the hosts have been chosen, self loops and duplicate edges are removed from the array of hosts. The removed neighbours are redundant and so they are handled at the sending peer in order to minimise network traffic.
Finally the size is reduced by all locally handled BubbleCast messages. The same is done for the start and for the end in line 13 to 15. At last the remaining size is split up among the remaining neighbours to which the message

has to be forwarded. See line 20 to 30 for the splitting of the size, the recalculations for the start and end of each message, and the forwarding of the BubbleCast messages. The recalculation is described in the next part about the provided incremental search.

Incremental BubbleCast Searches A very useful property of BubbleCast is that it can be used for incremental search. So for a probability of success of 99.99% a certainty factor of 3 has to be used. Since a certainty factor of 0.7 results in a success probability of more than 50%, a match may be found very early in the searching phase. This is especially true for popular data with several redundant bubbles. So message overhead may be reduced by incrementally search the peer-to-peer network. This method uses the fields start and end, which have not yet been described thoroughly. Each part of the incremental search can be seen as a slice of the whole bubble. Figure 2.24(b) shows a bubble that is incrementally searched. If enough results have already been received then any remaining slices should be dismissed.
The start holds the value of the start of the slice while the end holds the exclusive end of the bubble slice. Before the starts and ends of the slices can be determined, the sub-bubble sizes have to be calculated. The overall size is split by the number of hosts to which the message is actually sent. Let *mod* be the remainder of the division and $div := \lfloor size/split \rfloor$. Therefore each message has a size of *div* and the additional size is distributed among the first *mod* neighbours, which can be observed in line 22 of the code. The position in the current bubble is set to zero in the beginning. The start of a slice is the the maximum of the start of the current bubble reduced by the position in the current bubble and the locally handled edges, and zero. So the start for slice i $start_i = \max(start - local - position, 0)$. The end of a slice is the minimum of the size of that slice and the overall end reduced by the current position of the slice and the locally handled edges. This means that for slice i $end_i = \min(size_i, end - local - position)$.
Let us make an example of a slice in the middle of a bubble. Let the bubble size be 9, the start is 3, and the end is 6, while the split factor is 2. Let's assume that no duplicate edges and self loops were chosen for forwarding. The first message that is handled has the values (9, 3, 6) as (size, start, end). This message is not matched locally since the start does not equal zero. The new size is calculated as $\frac{9-1}{2} = 4$, the start equals $\max(0, 3 - 1 - 0) = 2$, and the end equals $\min(4, 6 - 1 - 0) = 4$. The message sent has the values $(4, 2, 4)$, which means that we asked only peer two and peer three to handle this part of the BubbleCast sub-bubble of size four.
Since we have another neighbour left, we add $size_i = 4$ to the position. The

```
1  bubblecast(int size, int start, int end, byte* msg) {
2    if(start == 0) process(msg);
3    int local = 1;
4    // find neighbours
5    out = permutate_neighbours(origin_of(msg));
6    out = remove_sender(out, msg);
7    out = subarray(out, 0, split);
8    out = remove_self_loops(out);
9    out = remove_duplicate_edges(out);
10   // remove local edges from remaining size
11   int outlen = out.length;
12   local += split - outlen;
13   start = min(start - local, 0);
14   size -= local;
15   end -= local;
16   if(size < 0) return;
17   // split the size among the chosen neighbours
18   int pos = 0;
19   int div = size / outlen;
20   int mod = size % outlen;
21   for(int i=0; i<outlen; i++){
22     int size_i = div + ((i < mod) ? 1 : 0);
23     int start_i = max(start - pos, 0);
24     int end_i = max(min(size_i, end - pos), 0);
25     if (start_i < end_i) {
26       out[i].bubblecast(size_i, start_i, end_i, msg);
27     }
28     pos += size_i;
29   }
30   assert(pos == size);
31 }
```

Figure 2.25: BubbleCast algorithm

start is calculated as $\max(0, 3 - 1 - 4) = 0$, and the end of the second slice equals $\min(4, 6 - 1 - 4) = 1$. The message sent to the second neighbour has the values $(4, 0, 1)$. So actually only one peer in the sub-bubble of size four is asked to handle the message. The receiving nodes follow the same procedure.

Matching Currently there a two alternatives of reporting matches to the originator of a BubbleCast. The first is to create a new connection to the originator and send him a match message. The second would be to send the message back on the same way the BubbleCast has been forwarded. Both methods have advantages and disadvantages, that have to be examined. At the time this is written only the first method is supported. So let us take a look into those two procedures.

1. **Directly connecting to the originator** The advantage of directly connecting to the originator is that the delay decreases because positive results are immediately reported back. No routing information have to be kept since the address of the originator is transmitted along with the BubbleCast message.
   The main disadvantage is that the originator may be SYN flooded with connection requests of many redundant results. If the network is large and a popular item is searched, the node may experience an incoming connection flood of over a thousand requests.

2. **Routing Match messages back to the originator** This method is very good to filter redundant results. It does not stress the network by establishing many short living connections.
   The disadvantage is that this method has a higher delay. Also if a node near the sender crashes while the BubbleCast is in progress, a large portion of a sub-bubble may be lost.
   A possible solution may be that the peers that were connected to the crashed peer, connect directly to the originator and send the filtered results to it. After a timeout they may terminate the connection.

BubbleCast

Purpose This message is used as the primary communication primitive. All applications building on BubbleStorm have to use this message type when distributing information.

Message type Fixed value := 0x5E000082(hex) = 1577058434 (dec)

Payload length The length of the payload

Fields	Message type	Payload length	Origin IP address	Origin TCP port	Bubble size
Byte offset	0..3	4..7	8..11	12..13	14..17

Fields cont'd	Total Size	bubble start	bubble end	bubble type	payload
Byte offset	18..21	22..25	26..29	30..33	33..(33+length)

Table 2.11: Fields of the BubbleCast message

Origin IP address The IP address of the peer that initiated this BubbleCast message.

Origin TCP port The port on which this host is listening for incoming connections.

Bubble size The size of the (sub-)bubble.

Total size The complete size of the bubble. This field is constant within each bubble.

Bubble start The start of the bubble slice.

Bubble end The end of the bubble slice.

Bubble type This field is application dependent. It is used as an identifier for different message payloads and should be unique in the BubbleStorm network.

Payload The actual payload, that the receiver has to process.

2.2.7 The User Connection Protocol

In order to allow nodes to send data over a user defined dedicated connection, an user defined application creates a connection according to the following protocol.
When the TCP/IP connection is established the initator sends the ASCII string

BubbleStorm UserConnect/<type> \n\n

where <type> should be the name of the application specific connection type. A node that whishes to accept the connection must respond with the ASCII string

BubbleStorm OK \n\n

Any other response is handled as a deny for the connection request. A connect rejection may have various reasons. E.g. the number of incoming connection slots is exhausted or the application type is not supported.
After the connection is established the transferred content has to be completely specified by the used user application.

Chapter 3

Implementation

The beginning of this chapter gives an overview of the prototype followed by the API description for applications that are build on top of the prototype. In the end a summary of the problems and decisions made in order to solve these are presented.

3.1 Overview

Figure 3.1: Overview of the Prototype

This overview shows of which parts the prototype consists. Figure 2.1 shows the parts of the BubbleStorm prototype. The prototype consist of the three parts 'topology', 'measurement', and 'BubbleCast' and builds on TCP/IP as the underlying network layer. The interface between the prototype and the TCP/IP networking ability is provided by Java NIO, which allows to do non blocking I/O operations. This part of the implementation is based on the LimeWire implementation, that can be found at 'www.limewire.org'. The topology part and the application developer's interface are the only parts that may create or close TCP/IP connections. All parts transmit messages over the created overlay network. There exist two special connections: the uplink, which is used as an initial connection to join the BubbleStorm network and the fishlink which is used to save measurement data when a node leaves. These connections can be the same connection if this connection is kept until a node decides to leave. But if the uplink node left while the new node is still connected then the new node has to find a new fishlink in order to duly leave the BubbleStorm network.

3.2 Design

BubbleStorm is based on TCP/IP as the underlying infrastructure. TCP/IP was chosen as the networking protocols because they are connection oriented which made it easier to create the overlay structure.

3.2.1 I/O Handling

Structure

The *bubblestorm.router.RouterService* class concentrates the core peer behaviour which is the topology maintenance, measurement and BubbleCast handling. All timeout checks and ping distributions are controlled by the RouterService. It also handles all incoming messages ,except for user connections, and is responsible to follow the protocol. The RouterService has a RouteTable which contains the information of all established connections.
Each socket is encapsulated in a *bubblestorm.nio.connection.Connection*. Connections have information about their location and the role they are playing e.g. master, slave. Each connection has also a message queue that is linked to the role and location in the route table.
The I/O event system is concentrated in the NIODispatcher class. Since the Singleton pattern has been mostly avoided in the prototype multiple instances can be run independently of each other. It is also possible to have more than one NIODispatcher for a BubbleStorm node although it is not needed, because if a task needs much processing time a new thread should be started and the NIODispatcher should resume the dispatching process.

Socket Creation

A socket may be created by accepting an incoming request or by initiating a connection. The socket input and output is piped through objects which are the bandwidth throttle, the message writing objects. It is also possible to pipe the message through a de-/obfuscator and deflater/inflater. This makes it very flexible for processing the byte stream on a lower layer. In figure 3.2.1 is shown how sockets prepared for input output operations.

Server

The main server functionality is contained in the **bubblestorm.nio.Acceptor** class. The accepting channel is registered with the NIODispatcher after it is created. Server sockets are created like the usual sockets by using the factory methods of the bubblestorm.nio.SocketFactory class, so the actually returned

socket may be a customised socket which simply extends java.net.Socket respectively java.net.ServerSocket. The NIODispatcher notifies the Acceptor by calling handleAccept(Socket socket) from the BubbleStormAcceptObserver object which is registered at the ServerSocket. If the server should be shut down then the port has to be set to zero. After this has been done no further incoming sockets can be accepted. Since all server functionality is bundled into the Acceptor class it is easy to make changes of the server or to add UPnP support.

Connecting

New socket are created by calling SocketFactory.newSocket(NIODispatcher) in the 'bubblestorm.nio' package. The newly created bubblestorm.nio.NonBlockSocket can then be connected by calling the provided connect methods. A comfortable way to react on a successful connect is to pass a ConnectObserver object to the connect method which executes user defined code when the connection has been established. This makes the connecting process event based which makes the post connect process more flexible and simpler. It allows an initialisation of objects that wait on the established connection which is better than giving each connection a type and when a connect event occurs, a onConnect method has to handle each type differently.

Sending and receiving messages

After the connection has been established, the connection has to be initialised for reading and writing. Reading is done by a MessageReader which reads data from the underlying socket and then encloses the data into the appropriate message. Before the message is passed to the RouterService the Connection handles the message itself and sets its attributes as needed, e.g. the topological location passed by a MasterHello or SlaveHello.
Writing messages is similar to reading messages. Here a MessageWriter is created which processes messages before they are sent and puts them into a ByteBuffer object in order to transmit the message. The MessageWriter takes a ThrottleWriter as its direct underlying channel. So the upload bandwidth may be controlled. the next part describes the functionality of the bandwidth throttle.

3.2.2 Bandwidth Throttle

The bandwidth throttle works as follows. Whenever a channel wants to read or write something it has to interest in the underlying channel. The under-

lying channel uses the handleRead() or handleWrite() method in order tell the channel that it may now read or write. When e.g. a ThrottleWriter receives a handleWrite() call it requests bandwidth from a common bandwidth throttle. It reserves the minimum of the desired bandwidth and the available bandwidth that the throttle granted the ThrottleWriter. Then the ThrottleWriter informs the upper channel that it is now able to write to the socket. The ThrottleWriter limits the possible transmission to the value that has earlier been determined. After the write process has been finished any left bandwidth is returned to the throttle so it may continue to distribute bandwidth to ThrottleWriters.
After some time has passed the NIODispatcher informs the bandwidth throttle about the new time. When enough time has elapsed the bandwidth throttle resets the available bandwidth to the configured value.

3.2.3 Node Cache

The node cache can be used for local files and for web based caches. The format of the files are exactly the same for the local cache and web cache. It only contains ip:port pairs that are seperated by a new line. The node cache retrieval is bound to the class PlainHostCache. The format for the cache can be changed e.g. to XML by exchanging the PlainHostCache by a XMLNodeCache. Since the data that is stored at the node cache is very simple a XML file would be too intricate. A HostCache provides two methods for retrieval. The first one is getAddresses(java.lang.String filename) which tries to retrieve the node cache from a locally stored file. The second method is to call getAddresses(java.net.URL url) which tries to obtain the addresses provided at the given url. In my opinion these methods are adequate for a peer to peer system.

3.2.4 Message Overview

BubbleStorm specific messages don't contain string values but bit fields. This makes it easier to extract values and saves bandwidth. Since the messaging system should handle different message types all messages must have a common super class which is bubblestorm.messages.Message. This class defines a getMessage(ByteBuffer) method which the specific message types have to implement. For each message type exist additional to an own message class a parser which is able to decode the message that has been received. The decoding of a message works like a simple finite state machine which decides on the first 4 bytes, the message type, what to do next. The length of each

message is provided by the appropriate parser. An overview of the messages provided can be seen in figure **??**.

Message priorities

Messages are stored in priority queues in order to send the most important messages first. The priorities are set as follows:

Priority 0
- MasterHello
- SlaveHello
- Cancel

Priority 1
- Redirect
- MergeEdge
- BreakEdge

Priority 2
- SplitEdge
- Ping

Priority 3+
- BubbleCast (3)

Topology messages have priority 0 and 1 except for the SplitEdge message. The hello messages have to be sent before any other messages because these messages determine the location of the connection and are important to the topology. The messages with priority one makes only sense to the receiver if a hello message has been received first. The Cancel message has also a priority zero because it may interfere with a MergeEdge message. If the connection to new node should be canceled but the MergeEdge message has been sent before the Cancel, then the MergeEdge will be ignored because the receiving node assumes the MergeEdge will be retransmitted to the new node. After the Cancel has been sent the sender, who wants to leave, waits for a shutdown as a reaction on the MergeEdge it sent. Since the MergeEdge had been ignored the leaving node will wait for an undefined time.
Messages with priority 3 are the SplitEdge message and the Ping message. These message types are important to integrate new nodes and to measure important network sizes. These messages should also be send after the topology messages. Ping messages have to be redistributed to the neighbours of each node. Therefore the neighbours have to be set before Ping messages are sent which makes it necessary to send topology specific messages before Ping messages.

3.2.5 Congestion Control

The congestion control is only used for BubbleCast messages and works as follows. Whenever a BubbleCast message has to be processed by the router a check is done how long the message queues are. If the sum of the bytes to be transferred divided by the possible upload rate is greater than four seconds all further BubbleCast messages are dropped. If the length is less than two seconds all messages are processed and routed to, random neighbours. In the interval between two and four seconds messages with a low remaining distribution amount are more likely to be dropped.

$$relativeSize = \frac{s}{bubbleSize}$$

$$oneSize = \frac{1.0}{relativeSize}$$

$$window = 4s - \frac{\log(relativeSize)}{\log(oneSize)}) * (4s - 2s)$$

The window is calculated as follows. At firs the remaining bubble size s is divided by the complete bubble size $bubbleSize$, which is determined by the formula given in the protocol definition in section 2.2.6 on page 46. If the calculated window is smaller than the current time a message would have to wait in order to be transmitted then it will be dropped. This is a very simple and effective method and causes only small reduction of the resulting bubble.

3.2.6 Routing

All messages are passed to the RouterService which decides how to react on incoming messages. It manages the messages that have to be sent in order to keep the network running. It also is responsible to create connections when needed e.g. in case of an incoming MergeEdge request. All messages have a predefined target except SplitEdge messages and BubbleCast message which are distributed to a random neighbour.

3.2.7 BubbleCast

BubbleCast messages are handled in the way it was described in the protocol definition. Every host in the BubbleStorm network should have a handler for each BubbleCast message type. If no for a received BubbleCast message exist then an error message is written to a log and the message is forwarded if it is not filtered by the congestion control. The certainty factor is assumed

three which corresponds to a 95% hit probability and the balanced factor is assumed 0.5.

3.2.8 Utilities

The implementation is based on Java NIO. Because NIO does not block threads when waiting for incoming or outgoing traffic a very small amount of threads can be used to do all I/O operations. It introduces channels that can be registered with one ore more selectors. Buffers are used to read or write from Channels that are writable respectively readable. I used the Limewire implementation of the I/O system for the prototype because it is mostly debugged, works very effective and it was easy to integrate. An important point for me to use a part of limewire was that it provides bandwidth throttles for Java NIO Sockets.
For details of Java NIO see the official references at "http://java.sun.com/j2se/1.4.2/docs/guide/nio/"
The Limewire implementation can be found at "www.limewire.org/limewire.zip"
For logging the package LOG4J from "http://logging.apache.org/log4j/docs/"
is used.

3.3 API description

This section describes the application programming interface for applications that are build on top of the BubbleStorm prototype. It is shown how to run a BubbleStorm node, how to integrate applications into the prototype, how to send BubbleCast messages, how to create user defined connections. Along with the prototype comes the javadoc documentation. This documentation describes all important methods for a developer who wants to update and expand the prototype.

Creating a new BubbleStorm node

A new node can simply be created by creating a new instance of the **bubblestorm.router.RouterService** class. The RouterService class provides many features like joining and leaving a BubbleStorm network, starting BubbleCast messages and creating user connections.

Joining a BubbleStorm network

After a node is created it has to join an existing BubbleStorm network. Therefore the method RouterService.join() has to be called. When joining one can choose how to get a first connection. If join() is called with no

arguments then the default local host cache is used ('./hostcache.txt'). The join method also may be called with a String as the argument where the String value is interpreted as a filename to a local host cache. The third possibility is to call join(url) with a java.net.URL as the argument. In this case a web cache is used which should be located at the given URL. The last possibility to call join is to give a java.net.InetSocketAddress as the argument which instructs the router to join by the given address. A node may create a new BubbleStorm network by calling RouterService.createSelfLoop() which can only be done if the node has not already joined any existing BubbleStorm network.

Leaving the BubbleStorm network

When the user wants to quit he should call the RouterService.leave() method.

Registering applications

In order to use the BubbleStorm network for applications, each used message type has to be registered at the router. Therefore a message handler has to be created which implements bubblestorm.bubblecast.IBSTypeHandler. A sample implementation of a message handler in the figure 3.3. The methods getBalanceFactor() and getCertaintyFactor() are message specific. The return values have to be chosen by the application developer according to the protocol definition. The method getBubbleType() should return a unique value for the kind of BubbleCast message. When messages of a certain type are received then the method processMessage(...) of the registered message handler is called. As a parameter the transmitted payload is given. The payload is the raw byte stream that has been received. The decoding of the payload should be handled by each application so the application messages experience a high flexibility and platform independence. Additional to the payload the address of the original sender of the BubbleCast message is provided. This is useful if the originator provides a file list of shared files for example in order to let other people know who provided the message.

Sending BubbleCasts

BubbleCast messages can be sent by calling RouterService.bubblecast(bubblestorm.bubblecast
msg). The content of each BubbleCast message is application specific. Each message type should have its own bubble type. E.g. a query message may have bubble type 3 and a publish message may have a bubble type 4. It is important that the corresponding message handler, that were described above,

have the same bubble type as the messages in order to correctly handle the messages.

Creating user connections

Connections to other BubbleStorm nodes may also be created by calling methods from the RouterService class. The RouterService class provides the method **connect(InetAddress address, int port, String connectionType, AppConnectObserver observer)**. The bubblestorm.applications.AppConnectObserver interface defines the method 'void handleConnect(AppConnection connection)' which is called when the connection is established.

Accepting user connections

In order to accept incoming user connections an acceptor handler has to be registered. Therefore an AcceptObserver has to be registered at the RouterService. The RouterService can then dispatch the created connection to the appropriate application when the connection has successfully been established.

Reading from connections

In order to read from sockets, a bubblestorm.nio.channel.ChannelReadObserver has to be implemented and AppConnection.setReader(ChannelReadObserver o) has to be called. The ChannelReadObserver interface defines a method handleRead() that is called when the connection has data to be processed. The application may also implement an interface like MessageReceiver bubblestorm.nio.connection which is a callback to the application for the events: a mesage has been completely received and the other side closed the connection.

Writing to connections

When an application has to send a message it has to call setWriter(ChannelWriter o) once on the new created AppConnection object. Whenever something has to be written the method 'channel.interestWrite(this, true)' should be called on the underlying channel. When the channel is interested in writing data it has to wait until it is able to actually send the data. When it becomes available the method handleWrite of the ChannelWriter object is called.

3.4 Conclusion

The I/O handling of the prototype is flexible and allows to extend the prototype very easily. It uses the java new IO for improving the performance. The most functions of the prototype are event based. The general problem of throttling the bandwidth was solved transparently to upper layers which allows the programmer to pay less attention when programming an application or expanding the prototype.
For handling a BubbleCast message by an application, the application has to register a handler for each message type. These handlers can be added dynamically at runtime and run application specific code. This code may be a matching algorithm for example. This seems a good solution because each application can define own bubble types and the payload of a BubbleCast message can be handled individually and platform independent.
The messages of BubbleStorm are fixed length and variable length. Variable length messages have to be used for BubbleCast messages since the payload length may vary. The messages have priorities because important messages, like topology and measurement messages, have to be sent unconditionally and in the correct order.
The congestion control only dismisses BubbleCast messages which create only a small sub-bubble compared to the overall bubble size. So the damage to a bubble should be minimally and the congested node should be relieved. Since users do not update the BubbleStorm software regularly it may take a long time, the nodes of the network may work with different versions of which some may not know every possible application. So a total bubble size field in the BubbleCast message has to be used for the congestion control to work properly and doing less damage to the bubbles. It's not desirable to have the new features only working properly when the majority of the nodes has been updated as this may take at lot of time.
The API defines some interfaces for an application developer. The developer may create user defined connections and reuse the IO subsystem of the BubbleStorm prototype. In order to differentiate user connections from BubbleStorm messages the application has to send at first an ASCII string that describes the connection type.

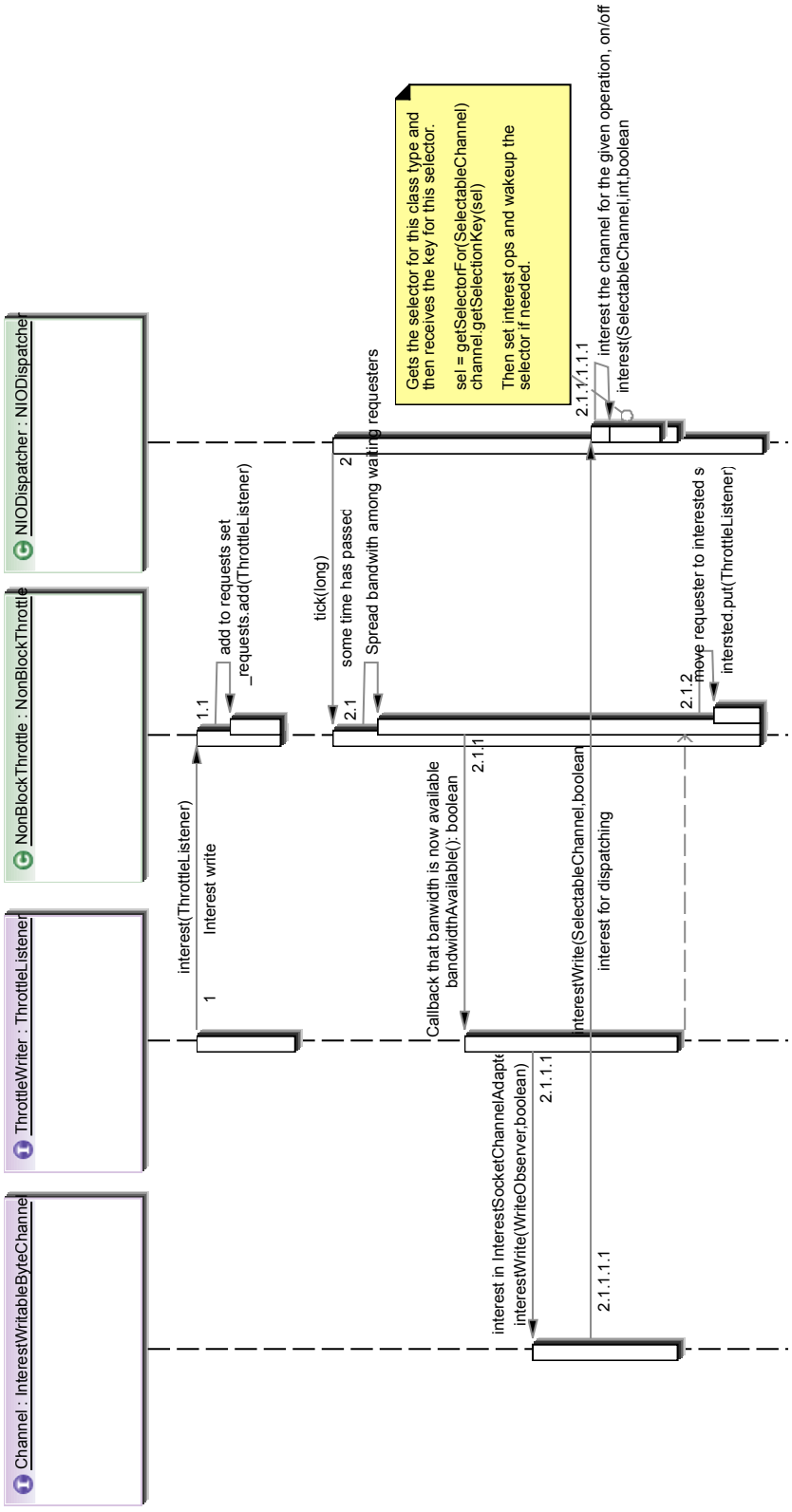


Figure 3.2: Registering channels for I/O

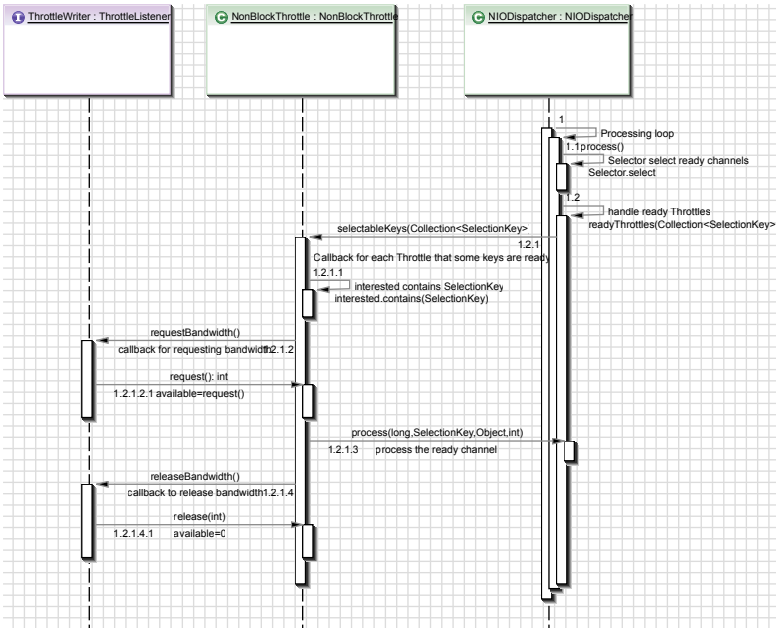


Figure 3.3: Processing ready channels

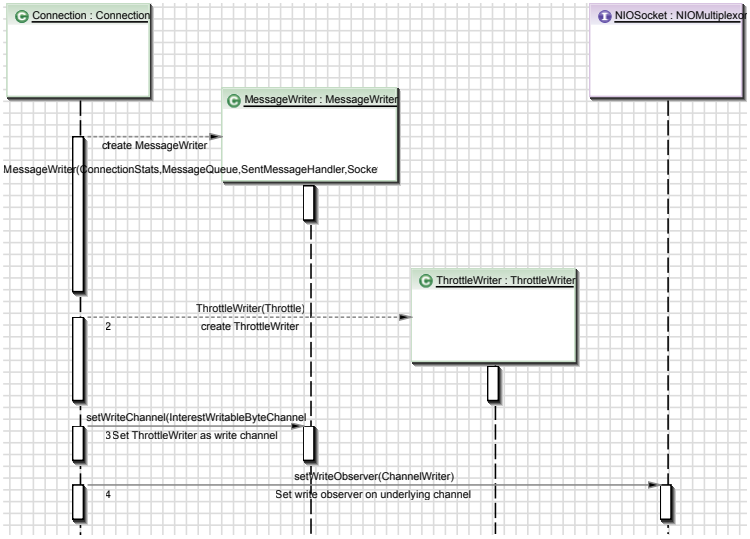


Figure 3.4: Writer creation

```
int listenport = 1489;
RouterService rs = new RouterService(listenport);

try {
  // uses the default local node cache './hostcache.txt'
  rs.join();
} catch (IOException e) {
  // do some exception handling
}
// uses the specified file as the node cache
// rs.join(String filename);
// uses the url as a host cache
// rs.join(java.net.URL url);
// connects to the specified address
// rs.join(java.net.InetSocketAddress address);
```

Figure 3.5: Creating a BubbleStorm node

```
public class XMessageHandler implements IBSTypeHandler {
  public float getBalanceFactor() {
    return 2.0f;
  }
  public int getBubbleType() {
    return 17; // the unique message type
  }
  public float getCertaintyFactor() {
    return 1.5f;
  }
  public void processMessage(InetAddress originAddress,
                             int originPort, byte[] payload) {
    // process the BubbleCast message
    // e.g. app.process(originAddress, originPort, payload);
  }
}
```

Figure 3.6: Creating BubbleCast message type handlers

```
IBSTypeHandler handler = new XMessageHandler
try {
  rs.register(handler);
} catch (IOException e) {
  // do some exception handling
}
```

Figure 3.7: Registering BubbleCast a message handler

```
rs.bubblecast(new IBubbleCastMessageType(){
  public byte[] getData() {
    Charset c = Charset.forName("UTF-8");
    return c.encode("query string").array();
  }
  public int getType() {
    return 3;
  }
});
```

Figure 3.8: Sending a BubbleCast message

```
public class XConnectObserver implements
        bubblestorm.bubblecast.AppConnectObserver {
  public void handleConnect(AppConnection connection)
                        throws IOException{
    ... handle established connection here ...
  }
  public void handleIOException(IOException iox) { ... }
  public void shutdown(){ ... }
}
```

Figure 3.9: Connecting to a host

```
RouterService rs = ...;
rs.connect(ip, port, "MyAppConnectionType", new XConnectObserver())
```

Figure 3.10: Connecting to a host

```
1 RouterService rs = ...;
2 rs.accept("MyAppConnectionType", new AppAcceptObserver(){
3   public void handleAccept(AppConnection connection)
4         throws IOException{
5   ... handle connection here ...
6   }
7 });
```

Figure 3.11: Accepting a connection

```
1 ByteBuffer buffer = ...;
2 AppConnection c = ...;
3 c.setReader(new XReadObserver());
4 c.setWriter(new XChannelWriter());
```

Figure 3.12: Registering for read and write

```
1 public class XReadObserver implements ChannelReadObserver {
2   /** the source channel */
3   private InterestReadableByteChannel channel;
4   /** the receiver of the messages, which
5    * should be the corresponding connection */
6   private MessageReceiver receiver;
7   public XReadObserver(MessageReceiver receiver){
8         this.receiver = receiver;
9   }
10  public void handleRead() throws IOException {
11    ... read implementation here ...
12  }
13  public void shutdown() {
14    synchronized(this) {
15      if(shutdown)
16        return;
17      shutdown = true;
18    }
19    receiver.messagingClosed();
20  }
21  public InterestReadableByteChannel getReadChannel() {
22    return channel;
23  }
24  public void setReadChannel(InterestReadableByteChannel newChannel
25    if(newChannel == null){
26      throw new NullPointerException("Cannot set channel to null");
27    }
28    channel = newChannel;
29  }
30 }
```

Figure 3.13: Creating a ChannelReadObserver

```
1  public class XWriteObserver implements ChannelWriter {
2    /** The sink channel we write to & interest ourselves on.*/
3    private InterestWritableByteChannel channel;
4    /** A callback for handlers who wish to process
5     * messages we succesfully sent. */
6    private final SentMessageHandler sendHandler;
7    public XWriteObserver(SentMessageHandler handler){
8          this.handler = handler;
9    }
10   public boolean handleWrite() throws IOException {
11     ... write implementation here ...
12         /** returns true if writing is not yet finished */
13   }
14   public synchronized InterestWritableByteChannel getWriteChannel() {
15     return channel;
16   }
17   public synchronized void setWriteChannel(
18               InterestWritableByteChannel newChannel) {
19     if(newChannel == null){
20       throw new NullPointerException(
21                       "Cannot set channel to null");
22     }
23     channel = newChannel;
24     channel.interestWrite(this, true);
25   }
26   public synchronized void shutdown() {
27     /** if nothing is left to write then shutdown output */
28   }
29   /** a method for sending messages */
30   public void send(message){ ... handle message to be sent ...}
31 }
```

Figure 3.14: Creating a ChannelWriter

Chapter 4

Evaluation

In this chapter the results of the test of the prototype are presented. At first the configuration of the test system is given. After that I give a overview of the results of the tests and in the end I evaluate the results particularly with regard to the results of the simulator [1]

4.1 Evaluation of BubbleStorm

performance of crash recovery, bubblecast match, bubblecast distribution

4.2 Evaluation of Prototype

Performance of I/O, message processing, Algorithms used

4.3 Evaluation of Simulation

[1] The results of the simulator has been published in [17]

Chapter 5

Future Work

1. BubbleCast packet lifetime
2. remember seen BC messages for incremental search (Bubble ID, end)
3. Redirect Client
4. Match messages
5. dynamic location count
6. dynamic balanced factor adaption
7. extend measurement to measure node churn
8. UPnP
9. Proxy support
10. SSL / TSL support
11. compression
12. real applications e.g.
    - wiki
    - blog
    - file sharing
    - chat
    - video conferencing / live streaming
    - VoIP
    - multiplayer gaming

Chapter 6

Conclusion

Bibliography

[1] Asad Awan, Ronaldo A. Ferreira, Suresh Jagannathan, and Ananth Grama. Distributed Uniform Sampling in Unstructured Peer-to-Peer Networks. In *Proceedings of HICSS'06*, pages 223–233, Washington, DC, USA, 2006. IEEE Computer Society.

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

[3] Virgil Bourassa and Fred B. Holt. SWAN: Small-world wide area networks. In *Proceedings of SSGRR'03*, 2003.

[4] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making gnutella-like P2P systems scalable. In *Proceedings of SIGCOMM'03*, pages 407–418, New York, NY, USA, 2003. ACM Press.

[5] Edith Cohen and Scott Shenker. Replication strategies in unstructured peer-to-peer networks. In *Proceedings of SIGCOMM'02*, pages 177–190, New York, NY, USA, 2002. ACM Press.

[6] Colin Cooper, Martin Dyer, and Catherine Greenhill. Sampling regular graphs and a peer-to-peer network. In *Proceedings of SODA'05*, pages 980–988, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[7] Ronaldo A. Ferreira, Murali Krishna Ramanathan, Asad Awan, Ananth Grama, and Suresh Jagannathan. Search with Probabilistic Guarantees in Unstructured Peer-to-Peer Networks. In *Proceedings of P2P'05*, pages 165–172, Washington, DC, USA, 2005. IEEE Computer Society.

[8] W. Hastings. Monte carlo sampling methods using markov chains and their application. In *Biometrika*, pages 57(1):97–109, 1970.

[9] Napster Inc. The napster homepage, 2007. http://www.napster.com/.

[10] David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-Based Computation of Aggregate Information. In *Proceedings of FOCS'03*, page 482, Washington, DC, USA, 2003. IEEE Computer Society.

[11] Tor Klingberg and Raphael Manfredi. Gnutella, June 2002. http://rfc-gnutella.sourceforge.net/developer/testing/.

[12] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and Replication in Unstructured Peer-to-Peer Networks. In *Proceedings of 16th ACM International Conference on Supercomputing ICS'02*, New York, NY, USA, June 2002.

[13] Qin Lv, Sylvia Ratnasamy, and Scott Shenker. Can heterogeneity make gnutella scalable? In *Proceedings of the 1st International Workshop on Peer-To-Peer Systems (IPTPS '02)*, 2002.

[14] Patrick Reynolds and Amin Vahdat. Efficient Peer-to-Peer Keywork Searching. In *Proceedings of the International Middleware Conference*, Department of Computer Science, Duke University, June 2003.

[15] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *In Proceedings of MMCN'02*, 2002.

[16] Nima Sarshar, P. Oscar Boykin, and Vwani P. Roychowdhury. Percolation Search in Power Law Networks: Making Unstructured Peer-to-Peer Networks Scalable. In *Proceedings of P2P'04*, pages 2–9, Washington, DC, USA, 2004. IEEE Computer Society.

[17] Wesley W. Terpstra, Jussi Kangasharju, Christof Leng, and Alejandro P. Buchmann. BubbleStorm: resilient, probabilistic, and exhaustive Peer-to-Peer search. In *Proceedings of the 2007 ACM SIGCOMM Conference*, August 2007. to appear.

[18] 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.