BubbleStorm: Analysis of Probabilistic Exhaustive Search in a Heterogeneous Peer-to-Peer System

TUD-CS-2007-2

Wesley W. Terpstra* Christof Leng† Alejandro P. Buchmann

Technische Universität Darmstadt
D-64283 Darmstadt, Germany
{terpstra,cleng,buchmann}@dvs1.informatik.tu-darmstadt.de

ABSTRACT

Exhaustive search in large-scale peer-to-peer systems is complicated by heterogeneity, crashes, node churn, hotspots, and weakly structured data. While existing approaches solve some of these problems, the BubbleStorm system offers a naturally integrated and simple solution based on random multigraphs. The simplicity of this mathematical structure allows us to rigorously analyze even the heterogeneous case.

We present a new communication primitive, named bubblecast, which induces subgraphs (bubbles) of controlled size. It can incrementally enlarge a bubble, but operates in parallel. Further, the underlying topology deals easily with crashes and node churn as maintenance operations are local, atomic, and minimally disruptive. In spite of this, it preserves the global random structure of a simple permutation. When combined into the BubbleStorm system, bubblecast on this topology performs exhaustive search with adjustable probabilistic guarantees and no hotspots. If every query must rendezvous with every datum, this approach has optimal per-node bandwidth complexity. Indeed, rather than suffering from heterogeneity, it is exploited fully.

1. INTRODUCTION

Peer-to-peer systems typically execute queries in large and heterogeneous networks. Most scalable approaches thus far—distributed hash tables—can be categorized as key-value indexes [14, 16]. Although keyword search, ranged search, and fuzzy matching can be implemented via an index [19], it is not clear how these scale on distributed indexes. Even if these scale, a designer must still fix which queries can be executed and then implement his routing strategy according to these application-specific queries. Peer-to-peer searches are predominantly run over weakly structured data; a single search might simultaneously include keywords, range restrictions, and nested types. What is missing is an efficient distributed version of the flexible linear-scan. Linear-scan can be formulated as a *rendezvous* problem where the query must meet every datum.

Furthermore, if data popularities are variable, hotspots may arise. Popularity distributions in online applications[1] are often Zipf-like [4]. But, if the popularity of the most popular item is independent of the number of items (Zipf with exponent > 1), then the accesses will grow linearly with the system size. Unfortunately, popular items are both more frequently updated and more frequently queried. If the system supports publish-subscribe, every open subscription for the popular item must be informed of every update. Also, the more popular the item, the more associated records each query must retrieve. These are rendezvous subproblems present within the hotspot. Therefore, even in situations where a key-value index makes sense, access frequencies might suggest that the more important factor is solving the rendezvous problem for popular data efficiently.

As Godfrey and Stoica observe, heterogeneity, rather than being an issue, can be an asset [10]. Due to constraints on bandwidth, memory, and processing speed, nodes in a network have very different capacities. However, most attempts start with a naturally homogeneous system and then graft in heterogeneity [8, 10]. While some systems [5, 13, 15] (typically unstructured) are by nature heterogeneous, they neither anticipate nor rigorously analyze the benefit.

A better approach is a naturally heterogeneous system engineered with foreknowledge of the potential asymptotic gains. For the rendezvous problem, when a node receives double the queries and datum, it quadruples its usefulness to the system, as it serves as rendezvous for four times as many pairs. A related analytic bound [17] shows that the total capacity of any rendezvous system can benefit at most from the sum of squared downlink bandwidths. A system built on these theoretical foundations could leverage the full potential of heterogeneity.

*Supported by the DFG Graduiertenkolleg 492, Enabling Technologies for Electronic Commerce.

†Supported by the DFG Research Group 733, Quality in Peer-to-Peer Systems (QuaP2P).

[1] In file sharing the distributions are quite similar, but differ by a fetch-at-most-once behaviour, likely because downloads are resource-intensive [11].

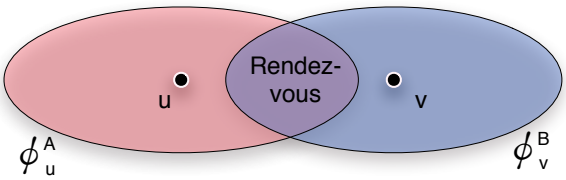

Figure 1: The intersection of a bubble originating from u with a bubble from v is the rendezvous

BubbleStorm is such a system, designed specifically to solve the rendezvous problem in a heterogeneous network. It performs efficient exhaustive search, introduces no hotspots, and is fully peer-to-peer. The specific contributions covered in this paper include

- Bubblecast, a communication primitive which induces subgraphs (bubbles) of controlled size. It can incrementally enlarge a bubble and operates in parallel.
- A peer-to-peer topology based on random multigraphs, modelled as a circular permutation. Its maintenance operations are local, atomic, and minimally disruptive.
- An equation for choosing bubble sizes to control the probability that bubblecast solves the rendezvous problem and exhaustive search.
- Rigorous, mathematical analysis proving the correctness, latency, load, and optimality for bubblecast run on this randomized topology.

Each of these contributions are tailored for heterogeneity.

2. OVERVIEW

To solve the rendezvous problem, BubbleStorm takes inspiration from the birthday paradox. One consequence of the birthday paradox is that two surprisingly small groups of people (type-A and type-B) are likely to have a birthday common to both groups. If children are more often born in the spring, the chance of a common birthday goes up.

Applying this to networking, whenever a node inserts new data (type-A), it replicates that data onto a random set of nodes by sending them type-A messages. Each replica is a person and the node storing that replica is the birthday. Similarly, each query (type-B) is replicated onto another set of nodes. Thanks to the birthday paradox, it is likely that some node received both and can match the query against the data. In the heterogeneous setting, powerful nodes are like spring days, increasing the chance of a match, or rendezvous.

Bubblecast is the communication primitive used to replicate a message onto nodes; it creates a message *bubble*. Match evaluation happens on nodes in the intersection of type-A and type-B bubbles—the rendezvous shown in Figure 1. Bubblecast blows these bubbles by flooding the message within a subgraph, whose size is specified as a parameter. Like a group of people has random birthdays, a bubble contains random nodes, thanks to the randomized topology.

We will model the randomized multigraph[2] as a circular permutation. It is modified solely by join, leave, and crash events. Heterogeneity is incorporated into the topology by setting node degree proportional to capacity. Thus, nodes have a fixed size routing table and locally control their desired workload. Furthermore, the join and leave operations modify no more edges than the node's eventual degree.

In heterogeneous networks, the bubble size required for a match is smaller, reflecting the effect of spring days. We will present an explicit equation, the match threshold, which captures this relationship in Section 4.2. The unit bubble size will be derived from this match threshold.

When a bubble is c times larger than the unit bubble, it intersects bubbles of opposing type with probability at least $1-e^{-c^2}$. Analysis will prove this key result, that controlling the bubble size controls the chance of intersection. Finally, we will meet the theoretical lower bound for the rendezvous problem by balancing the type-A and type-B bubble sizes.

3. TOPOLOGY

For a connected multigraph where every node has even degree, Euler proved that a tour of all edges is possible. By writing down nodes each time they are crossed in the tour, one can untangle the multigraph into a circle. An example of this correspondence is shown in Figure 2.

In our model, every node appears in $\frac{\deg}{2}$ *locations* on the circle, and every edge appears exactly once. Naturally, for a given multigraph there are many such circles, but a given circle describes exactly one multigraph. As mentioned in Section 2, a node sets its desired locations proportional to its capacity. However, every node must have minimum degree of 4, in order to keep the network well connected.

The combined global state of a BubbleStorm network describes such a circle. This circle includes direction; a node's location has an edge connecting to it's clockwise (CW) neighbour and another to it's counter-clockwise (CCW) neighbour. These edges represent the TCP connections that link the network together. With self-loops it is possible that the neighbour of a node's location is the node itself.

Each node stores neighbour state in a routing table. A node v labels its locations, $\ell(v)$, on the circle with *location identifiers*, for example $\ell(v) = \{v_1, v_2, v_3\}$. Keyed by location identifier, the routing table stores the network address and remote location identifier of the CW and CCW neighbours.

The BubbleStorm graph is the random set G. As usual, we write $V(S)$ to mean the vertexes in subgraph S. For brevity, V will be used as short-hand for $V(G)$. Similarly, $E(S)$ denotes the edges in a subgraph and $E = E(G)$. The circle C is a permutation of the node locations, $\ell(V)$. $(u_1, v_1) \in C$ if and only if v_1 is the clockwise successor of u_1. The edges in directed multiset $E(G)$ simply drop the location identifiers from C. Denote the degree of v by d_v. The observant reader will notice that $|E| = |C| = |\ell(V)|$.

[2]Although potential self-loops and multi-edges are not useful to bubblecast, they are rare and necessary for the model.

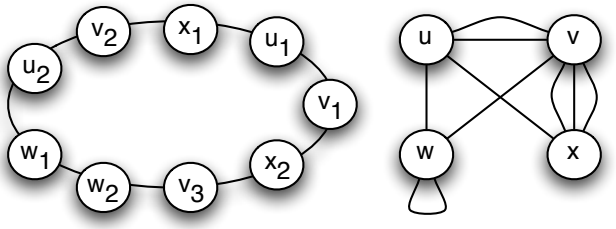

Figure 2: A network viewed as circle and multigraph

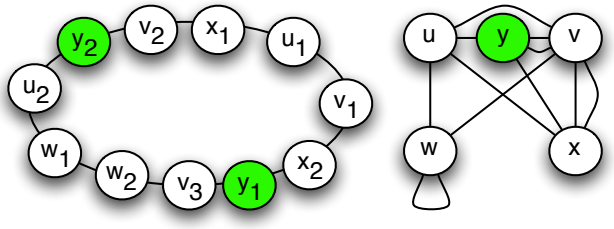

Figure 3: Both views of the network after y joins

3.1 Membership Protocol

The very first node creates the network. It forms a permutation of its desired number of locations and then connects them cyclically to form the circle. For example, $v_2 \to v_1$, $v_1 \to v_3$, and $v_3 \to v_2$.

The join and leave algorithms are executed independently (and in parallel) for each location a node has on the circle. Conceptually, the join algorithm picks a random edge and inserts the joining location into the middle of that edge. Figure 3 shows the result of y joining Figure 2 twice. Once inserted, the joining node has gained two neighbours at the new location, without affecting the degree of other nodes.

The edge to split is chosen by a biased random walk starting from a known participant. Thus, joining nodes must know the identity of a node participating in the network. Our prototype keeps a disk-backed cache of the most recently seen nodes for this purpose.

In the simplified join protocol shown below, a_x refers to the network address of node x and l_x refers to the relevant location in the routing table of node x. `SplitEdge`(a_y, l_y) messages are routed via a random walk of logarithmic length. The eventual receiver, u, randomly picks location l_u where it will splice the sender in as a CW neighbour. Then u connects to the new node y. The `Hello`(l_y, CCW, l_u) messages tell receiver y to set $l_y[CCW] = u$, where l_u is the sender's location identifier. If successful, the `Redirect`(a_y, l_y) message tells the receiver v to cease communication on this edge and connect instead to y. If v cannot connect to y, it cancels the complete operation. Otherwise, `shutdown`, the usual TCP closure of writing, signals u that no more messages will be sent, and the other end can safely `close`. After u closed the connection, the join operation is complete. A cancelled operation is restarted by the joining node by sending a new `SplitEdge` message.

The join protocol
(y joins between u and v)

y	$u \leftarrow \ldots \leftarrow$ `SplitEdge`(a_y, l_y)	
u	$y \leftarrow$ `Hello`(l_y, CCW, l_u), $v \leftarrow$ `Redirect`(a_y, l_y)	
y	l_y[CCW] $= u$	
	v can connect to y	otherwise
v	$y \leftarrow$ `Hello`(l_y, CW, l_v)	$u \leftarrow$ `Cancel`
	$u \leftarrow$ `shutdown`, l_v[CCW] $= y$	
y	l_y[CW] $= v$	
u	l_u[CW] $= y$, $v \leftarrow$ `close`	$y \leftarrow$ `close`
v	$u \leftarrow$ `close`	

One invariant of these protocols is that only the location CCW of an edge can modify that edge. This makes it relatively easy to serialize concurrent joins and leaves. To this end, the join protocol only sends `Redirect` messages in the CW direction. Similarly, in the leave algorithm below, `MergeEdge` messages (used by a leaving node to inform neighbours about its departure) are sent to the CCW neighbours. The CCW neighbour is responsible for the edge and reconnects to the CW neighbour of the leaving node's location. If the CCW neighbour is already changing the edge (splitting it or leaving itself), `TryLater` is used to tell the node to wait for the change before retrying to leave.

The leave protocol
(y leaves between u and v)

y	$u \leftarrow$ `MergeEdge`(a_v, l_v)	
	u can process `MergeEdge`	u will change
u	$y \leftarrow$ `shutdown`, l_u[CW] $= v$	$y \leftarrow$ `TryLater`
	$v \leftarrow$ `Hello`(l_v, CCW, l_u)	
v	$y \leftarrow$ `shutdown`, l_v[CCW] $= u$	
y	$u \leftarrow$ `close`, $v \leftarrow$ `close`	

THEOREM 1. *The membership protocol maps circles to circles. Or, the network is always a circle.*

PROOF. The first node establishes a circle. When a node joins, it adds itself to a location on the circle. Therefore the graph remains a circle. When a node leaves, it splices together the edges at its location, preserving the circle. □

THEOREM 2. *Every permutation of $\ell(V)$ on the circle is equally likely. There are $(|E|-1)!$ permutations.*

PROOF. The proof proceeds by induction on join/leave events. The first node forms a permutated circle uniformly at random, of which there are $(|E|-1)!$. Due to the random walk, each new location splits one of the $|E|$ edges, chosen uniformly at random. As the network previously had $(|E|-1)!$ permutations all equally likely, the resulting $|E|!$ permutations are also equally likely. When a location leaves, it could have been in any of the $|E-1|$ slots between other locations. Thus, each resulting permutation occurs $|E-1|$ times, or with probability $\frac{1}{(|E|-2)!}$. □

3.2 Crashes

As long as no nodes crash, the circle stays connected. Unfortunately, in a large system, nodes will malfunction. As is typical for peer-to-peer systems, we consider only crash-failures. When a node crashes, it quits without executing the leave algorithm and creates gaps in the circle.

While it might be tempting to repair the circle, the circle is only a mathematical model that fits our join/leave algorithm. It turns out that the properties needed are also provided by a broken circle. Therefore, we don't bother repairing the circle, as this would be quite costly and pointless. Instead, we change the model to circles with broken edges.

Although we do not repair the circle, we must repair node degree; those nodes adjacent to a broken edge have had their degree reduced. If left alone, this would violate the invariant that node degree is proportional to node capacity. Recall that nodes only choose a *desired* number of locations; we allow the actual degree to be one below the desired value. Whenever the actual degree is two or more below the desired degree, run the join algorithm to add two neighbours. Also, shrink the routing table by dropping locations which have both neighbour edges broken. However, two half-broken locations may *not* be merged, as this could make two circles.

Edges split by the join algorithm—including broken edges[3]—must be chosen uniformly at random. Without any bias, a random walk would choose a node proportional to the number of its faultless edges. We should adjust this probability via techniques similar to [1], so that the random walk selects a node proportional to the number of its locations. In practise one might just omit this, as it has little effect.

Before we allowed broken edges, the network was always connected. As we start breaking edges, the network could become disconnected. In fact, normal random graphs are only almost surely connected. The more edges that are broken, the closer we come to the usual model of random graphs.

THEOREM 3. *When as many edges in the circle are broken as possible, the graph is a random perfect matching.*

PROOF. As a location with two broken neighbours is removed, every location must have one neighbour. Let them all have exactly one neighbour (the most broken possible), then every second edge in the circle is broken. It would look something like $l_1 \to l_2$, $l_3 \to l_4$, ... Let the connected pairs of locations be matches. Every permutation describes a matching of all the locations. The same perfect matching can be arranged on the circle $\frac{|\ell(V)|}{2}!$ ways with $2^{\frac{|\ell(V)|}{2}}$ orientations for each pair. Thus they are all equally likely. □

COROLLARY 4. *The graph is almost surely connected.*

PROOF. Bollobás [2] proved almost sure connectivity for regular graphs using perfect matchings. As the actual degree can only be 1 less than the minimum desired degree of 4, we have deg ≥ 3. Thus, his proof also applies to our graph. □

4. BUBBLECAST

Within BubbleStorm, the job of bubblecast is to ensure a rendezvous between every type-A and type-B message. It does this by replicating both message types into the network, where they will probabilistically meet. The replication subgraph for a given message is called its bubble, defined as ϕ_u^A for a type-A message replicated from node u. For two message bubbles with non-empty intersection, there is a rendezvous node, $r \in V(\phi_u^A) \cap V(\phi_v^B)$, which received both. When r exists, bubblecast has succeeded for this pair; otherwise it has failed.

The process of blowing a bubble is similar to flooding. The origin node sends the message to his neighbours, who send it to their neighbours recursively until the desired replication is achieved. The major advantage of this approach is that has low latency, due to the extreme parallelism. Furthermore, as small subsets in a random graph are very tree-like [2], the process reaches many different nodes quickly.

Unlike flooding, bubblecast precisely controls the number of edges in the bubble subgraph. This corresponds exactly to the number of messages sent. We define this as the bubble size, and denote it as $|E(\phi_u^A)|$ or just $|\phi_u^A|$. For all type-A messages, the replication factor $|\phi^A|$ is the same. Similarly, all type-B messages have the same bubble size, which likely differs from type-A. The probability of a rendezvous is directly related to the number of edges explored; it is therefore important to carefully control this quantity (Section 4.2).

For this reason we reject the usual TTL or hop count that limits the recursion depth; for example, see Gnutella. Increasing the TTL exponentially increases the nodes reached and edges explored. Worse, in heterogeneous networks, the TTL method reaches a wildly different number of nodes depending on the interior node degrees. For these reasons, bubblecast includes the bubble size in each message, specifying exactly how many edges remain to be explored.

Not only can bubblecast control the total edges explored, it can control when those edges are explored. If a search has many hits, it might be a self-inflicted denial-of-service attack to perform an exhaustive search. The traditional TTL approach would be to repeatedly reissue the query with a growing TTL. Unfortunately, this approach wastes bandwidth on interior nodes and produces inaccurate bubble sizes.

Bubblecast solves this with resumable queries, viewing the bubble as a pie. Each partial search explores only a pie-slice, with a given angle. By varying the angle of the pie slice, one can query with a controllable size (pie area). By keeping the pie slices disjoint, very little traffic is incurred on interior nodes. Naturally, once the entire pie is explored, this method ceases to permit incremental search. Therefore, the pie size should be chosen to be a bit larger than needed to reach the desired probabilistic bound. This approach will also be useful in dealing with collisions (Section 4.3).

Node *split* is the maximum number of messages forwarded to neighbours. The split must be at least 2, but can be as large as $d_v - 1$. In Figure 4 all nodes have split = 3. A system-wide split ensures that outgoing traffic stays proportional to degree, rather than squared degree as seen in flooding.

Also, in today's Internet, available uplink/downlink bandwidth are not always symmetric (e.g. ADSL). Symmetric nodes should use the fixed, system-wide split. Asymmetric nodes can halve their split to send roughly half as much as

[3]When a broken edge is split, the processing node connects to the joining node. Thereby, the broken edge is moved to the joining node, who receives only one neighbour.

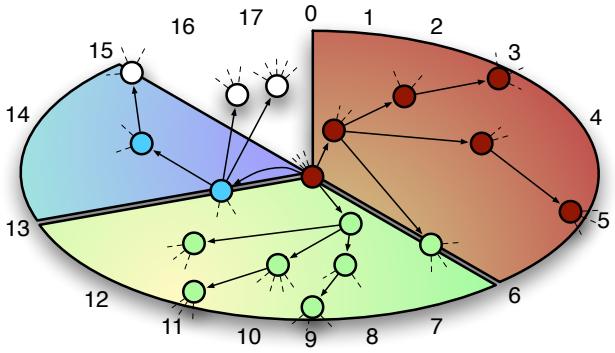

Figure 4: A size 18 bubble cut into three slice

received[4]. This marginally increases the uplink requirement of many other nodes in the system.

4.1 The Algorithm

The header of a bubblecast message includes the desired bubble size and the [start, end) pie interval of the slice to fill. Each node divides the bubble into smaller bubbles and forwards these to some of its neighbours. This is very similar to building a search tree. The process is illustrated in Figure 4. In that example the first pie-slice had [start, end) = [0, 6) and size = 18. Users of bubblecast guarantee to fill the pie clockwise, with no gaps, starting from 0.

The edges explored are chosen by taking the first 'split' neighbour edges out of a permutation. In this way, every edge is equally likely to receive a sub-bubble. The permutation of neighbour edges is chosen randomly, seeded by the address of the original message sender. This ensures that a future pie-slice in the same bubble will use the same neighbours. Conceptually the message-id specifies a directed subgraph of the network for exploration.

The correctness of the bubblecast algorithm only requires that $|\phi|$ edges be explored. The exponential division of the bubble size is solely for low latency. Therefore, when there is a double-edge, we can send the same size as for any other neighbour. However, if we explored both halves of the double-edge, we need to reduce the bubble size by one (for the extra edge explored). Similarly, if we explore a self-loop, the bubble size should be reduced. The whole algorithm is shown in Figure 5.

Once a bubble has explored $\sqrt{|E|}$ edges, it becomes quite likely to have found a cycle. This can pose a problem as a short cycle means that a node received the same bubble twice. As both messages will explore the same subgraph, the entire bubble size of the smaller sub-bubble is lost. Also, if the permutation in Figure 5 picked only self loops (very short cycles), bubblecast will not replicate the message at all. For these reasons, collision counter-measures are important, and will be discussed in Section 4.3.

4.2 Bubble Size

The larger the bubble, the more likely it will rendezvous with all messages of the opposing type. For this reason, bubble size is proportional to the controllable certainty factor c. However, the required size of a bubble clearly depends on the size of the network and its expansion factor. These aspects are captured by the match threshold.

[4]To our knowledge, ours is the first P2P search system to support asymmetric upstream/downstream bandwidth.

```
bubblecast(size,start,end,msg) {
 // The arrival edge comes first
 if (start == 0) match(msg);
 local = 1;
 // Establish the neighbours we send to
 out = permute_neighbours(origin_of(msg));
 out = remove_sender(out, sender_of(msg));
 out = subarray(out, 0, split);
 out = remove_self_loops(out);
 out = eliminate_duplicate_edges(out);
 // Remove local edges from remaining size
 local += split - out.length();
 start = min(start - local, 0);
 size -= local;
 end -= local;
 if (size <= 0) return;
 // Split the size amongst chosen neighbours
 outlen = out.length();
 pos = 0;
 for (i = 0; i < outlen; i++) {
  size_i = size/outlen + (i<size%outlen)?1:0;
  start_i = max(0, start - pos);
  end_i = min(size_i, end - pos);
  if(start_i < end_i)
   out[i].bubblecast(size_i,start_i,end_i,msg);
  pos = pos + size_i;
 }
 assert(pos == size);
}
```

Figure 5: The bubblecast algorithm

DEFINITION 5. *The match threshold, T, is defined as*

$$T := \frac{\left(\sum_{v\in V} d_v\right)^2}{\sum_{v\in V} d_v(d_v-2)} = \frac{(2|E|)^2}{\sum_{v\in V} d_v(d_v-2)} \leq 2|V|$$

Intuitively, the match threshold is the bubble size required to reach an arbitrary node with probability $\geq 1-\frac{1}{e}$. In a homogeneous network $T = |V|\frac{d}{d-2}$. As $d \to \infty$, bubble exploration becomes $|V|$ independent choices out of V. Thus, successful match probability is $\geq 1-\frac{1}{e}$ as expected.

As bubblecast works on heterogeneous systems, T must be a bit more sophisticated. When $|E|$ is held fixed, denominator $\sum_{v\in V} d_v(d_v-2)$ has a minimum in the homogeneous case (easy proof using Lagrange multipliers). Therefore, T is maximized by the homogeneous case and heterogeneity reduces the match threshold. This is the core of how BubbleStorm benefits from heterogeneity.

Naturally, in real systems we are not interested in flooding the whole network. Therefore, instead of one giant bubble for type-A messages, we trade off type-A and type-B bubble sizes while keeping their product equal to T; Section 5.2 shows this is the right relationship. If the type-A/B workloads (measured in bytes) are W_A and W_B, then we want to minimize $W_A|\phi^A| + W_B|\phi^B|$, the bandwidth cost after replication. With $|\phi^A||\phi^B|$ fixed, $W_A|\phi^A| = W_B|\phi^B|$ is the best trade-off. It is shown optimal for rendezvous in Section 5.4.

OBJECTIVE 6. *For a given certainty factor c and workload W_A, W_B, the bubbles sizes should be*

$$|\phi^A| := \left\lceil c\sqrt{T\frac{W_B}{W_A}} \right\rceil \qquad |\phi^B| := \left\lceil c\sqrt{T\frac{W_A}{W_B}} \right\rceil$$

except where this would exceed cT, where it is $\lceil cT \rceil$.

While c is a system-wide parameter, T, W_A, and W_B must be known to determine the correct bubble sizes. They are found using an epidemic measurement protocol based on [12]. Alternately, the workload ratio can be a system-wide parameter based on projected usage. As the measurement algorithm can only calculate averages and sums, T needs to be rewritten as $T = \frac{D_1^2}{D_2 - 2D_1}$, where $D_i = \sum_{v \in V} d_v^i$.

4.3 Collisions

When a bubble's message is sent over the same edge twice, that is a collision. Collisions can only occur if there is a cycle contained in the bubble subgraph. Their effect is to reduce the effective size of a bubble, thus decreasing success probability. For this reason, if probabilistic guarantees are required[5], collisions must be prevented.

In order to detect collisions, every node remembers recently seen bubbles. For each bubble, it stores a unique bubble ID and the end of slice (if it differs from incoming size). Receiving a bubble with a previously seen ID could lead to a collision. However, when the start of slice equals the stored end of slice, this is just a subsequent pie slice and should be processed normally. Otherwise, there is a cycle in the bubble subgraph. A cycle only causes a collision if receiving node forwards the message over the same edge twice. To prevent this, the receiving node must now take counter-measures.

When a collision is detected, the receiving node can report the incoming size back to the original sender. The receiver then ceases further processing. The sender can then make an additional pie slice with the sum of reported sizes. This requires that the sender's first bubblecast is a partial slice. It also increases the latency. Fortunately, most of the time the problem can be fixed locally where the collision is detected.

If bubblecast uses only a fraction of a node's neighbours for forwarding (split $<$ degree -1), a number of spare edges are available. For example, with degree 10 and split 4, bubblecast uses 5 edges (one for the incoming message). So, there are enough edges available to bubblecast *twice* from this node without re-using an edge. When the ratio of degree to split is higher, even more collisions are preventable.

Only when insufficient spare edges are available on a node must collisions be reported back to the original bubblecaster. As a generalization of the Birthday Paradox, the probability of x collisions on node v is below $\binom{|\phi|}{x}\left(\frac{d_v}{2|E|}\right)^x$. Therefore, triple collisions are very unlikely for typical bubble sizes. High degree nodes have a higher chance of seeing multiple collisions, but their higher edge to split ratio compensates—more collisions are needed to exhaust their spare edges.

[5]As cycles are quite rare, it is reasonable for applications with failure tolerant requirements to do nothing.

5. ANALYSIS

While the algorithm used for rendezvous is relatively simple, it was chosen to allow for rigorous analysis. There are four important theorems about BubbleStorm. They relate to its latency, correctness, maximal per-node load, and optimality.

In order to prove the results below, we will assume that the bubble size is specified by Objective 6. That is, some technique, perhaps one from Section 4.3, has been implemented to guarantee the correct number of edges are explored.

Keep in mind that the random variable in our proofs is the topology. Thus, a node's neighbour is a random variable, as is the contents of a bubble like ϕ_u^A. However, the number of neighbours a node has, and thus $|E|$, are fixed. The size of a bubble, $|\phi_u^A|$, is also fixed by our assumption above.

5.1 Latency

When doing an incremental search using bubblecast, one must know when a slice has finished. If the size of the network is known, the following theorem can be used to calculate the time-out for a slice. The measurement algorithm provides the network size ($D_0 = |V|$). The latency in milliseconds depends on the underlying network topology and technology. As we do not possess this information, we instead calculate latency in overlay hops.

THEOREM 7. *A bubblecast slice in ϕ has latency*

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

PROOF. Each bubblecast invocation has a latency of the longest path length. From nearly every node, the number of neighbours reached is ≥ 2. Thus, bubblecast terminates with depth $\leq \log_2 |\phi|$.

$$L_A \leq \log_2 |\phi| \leq \log_2 cT \leq \log_2(2c|V|)$$

This worst-case dropped the usual $\frac{1}{2}$ from bubble balance. □

5.2 Correctness Probability

The algorithm is correct, or succeeds, for a type-A and type-B bubble if there is some node which received messages of both. The main result of this section is that failure probability depends on the certainty factor, c, as $\mathbf{P}(\text{fail}) \leq e^{-c^2}$.

Throughout this section, we will assume a circle possibly with broken edges. How these broken edges are formed was discussed in Section 3.2. However, when one explores an intact edge, as all permutations are equally likely, the location reached is a uniform random sample chosen without replacement. The broken edges do not prevent access to a specific location when the graph is a random variable.

In order to prove the main result, we will first need a technical result about the expected size of the border or surface of a bubble. To measure this quantity we will generalize the degree function to operate not only on nodes, but also on a subgraph. For a single node v, $\deg(v)$ the number of half-edges incident on the node. This means that self-loops get counted twice and other edges once. For a subgraph, we also count the half-edges incident on subgraph vertexes, excluding those in the bubble. This is illustrated in Figure 6.

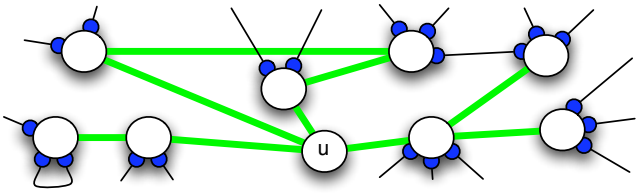

Figure 6: A bubble's degree/border as blue dots

LEMMA 8. *For a node $u \in V$ chosen independently from the connected graph, the border is normally distributed with*

$$\begin{aligned}\mathbf{E}(\deg(\phi_u^A)) &\geq \frac{|\phi_u^A|}{2|E|}\sum_{v\in V} d_v(d_v-2) - \frac{|\phi_u^A|^2}{8|E|^2}\sum_{v\in V} d_v^3 \\ \sigma^2(\deg(\phi_u^A)) &\leq \frac{|\phi_u^A|}{2|E|-1}\sum_{v\in V} d_v^3 \approx \frac{|\phi_u^A|}{2|E|}\sum_{v\in V} d_v^3\end{aligned}$$

PROOF. The border starts with node u and degree $= d_u$. Then, at each step, we explore an edge to reach nodes. When a node is *first* added, it contributes its degree to the border. Naturally, each edge explored connects two nodes, one that was in the border before, and one on the reached node. Thus, every exploration decreases the border by 2. Let I_v indicate that $v \in \phi_u^A$ (ie: $\mathbf{P}(I_v = 1) = \mathbf{P}(v \in \phi_u^A)$).

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

For indicator variables, $\mathbf{E}(I_v^k) = \mathbf{P}(I_v = 1)$ for all k and therefore, $\mathbf{E}((d_v I_v)^k) = \mathbf{P}(I_v = 1)d_v^k$. As the indicator variables are independent, we apply the central limit theorem:

$$\begin{aligned}\mathbf{E}(\deg(\phi_u^A)) &= \sum_{v\in V} d_v \mathbf{P}(I_v = 1) - 2|\phi_u^A| \\ \sigma^2(\deg(\phi_u^A)) &= \sum_{v\in V} d_v^2 \mathbf{P}(I_v = 1)(1 - \mathbf{P}(I_v = 1))\end{aligned}$$

We now find bounds on $\mathbf{P}(I_v = 1)$ by running the algorithm. We explore $|\phi_u^A|$ *directed* edges, and each edge added makes one location unreachable CW and another CCW. Thus, at step i, only $|E| - 1 - i$ locations are reachable, all equally likely thanks to the permutation. A given v has $\frac{d_v}{2}$ locations. It is unreached if and only if every exploration missed it.

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

Whenever $\frac{1}{2}d_v > 1$ (which it is by our minimum degree),

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

Expand the expectation using $\mathbf{P}(I_v = 1) = 1 - \mathbf{P}(I_v = 0)$.

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

$$\sigma^2(\deg(\phi_u^A)) \leq \sum_{v\neq u} d_v^2 \mathbf{P}(I_v = 1) \leq \frac{|\phi_u^A|}{2|E|}\sum_{v\in V} d_v^3$$

For the variance, notice that $(1 - \mathbf{P}(I_u)) = 0$. □

DEFINITION 9. *We measure degree heterogeneity with*

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

While H seems to have no direct physical interpretation, it somehow describes the phase-transition between a distributed system and a centralized one. Nodes with relative capacity less than $\sqrt{|V|}$ times the largest node are not useful participants; they should be clients. This is because a node which has a $\sqrt{|V|}$ times larger degree serves as a rendezvous point for $|V|$ times more than the smaller node. Even if the remainder of the network consisted solely of these smaller nodes, they would still only serve as a rendezvous for half of the load. The system would be have perfect rendezvous success and be at worst twice slower if all the other nodes just acted as clients to the larger node. This corresponds to the fact that if $d_v = o(\sqrt{|E|})$, then $H = o(1)$.

The above is a worst-case bound, and for more uniformly distributed d_v, H decreases much more quickly. In fact, if the degree relationship between nodes are fixed fractions[6], then $H \approx \Theta(|V|^{-0.5})$. In our experience, H is an overestimate in homogeneous and near-homogeneous systems.

DEFINITION 10. *We measure the extent to which a workload's relative type-A and type-B traffic differ with*

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

THEOREM 11. *For two arbitrary nodes, u and v, chosen independently from the connected network topology, the bubbles ϕ_u^A and ϕ_v^B fail to reach a common node with probability*

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

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

PROOF. We want to find a node which received both messages. One sufficient condition for this is that an edge in ϕ_v^B reaches the border of ϕ_u^A. That would mean that this edge transmitted the type-B message and is incident on a node which saw the type-A message. That node must therefore have received both.

We assumed that $|\phi_u^A|$ is not a random variable, but nevertheless $\deg(\phi_u^A)$ is. Thus, we first examine the conditional failure probability $F(\Delta) = \mathbf{P}(\text{fail} \mid \deg(\phi_u^A) = \Delta)$. Later we will apply $\mathbf{E}(F(\deg(\phi_u^A))) = \mathbf{P}(\text{fail})$ to eliminate this.

Imagine exploring one edge in ϕ_v^B at a time while holding ϕ_u^A fixed. We stop exploring edges and declare victory once we add reach an edge in $E(\phi_u^A)$ or incident on $V(\phi_u^A)$. This must happen before step $|E| - |\phi_u^A|$, at which point there are no more edges. The *first* step where we succeed and stop, we denote by the random variable S.

[6]This appears to be the case for Gnutella [5].

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

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

$\mathbf{P}(S = 0 \mid \deg(\phi_u^A) = \Delta) = \frac{|\phi_u^A|}{|E|} + 2\frac{\Delta}{2|E|} - \frac{\Delta^2}{(2|E|)^2} > \frac{\frac{1}{2}\Delta}{|E|}$ is the chance that the first edge added was instant victory[7]. All subsequent edges must be incident to the first, and therefore can not be in ϕ_u^A, because first one must cross the border.

Let us make a few observations about the involved sets:

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

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

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

Filling in the values of $\mathbf{P}(S \neq i \mid S \geq i \cap \deg(\phi_u^A) = \Delta)$,

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

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

$$\begin{aligned} \mathbf{P}(\text{fail}) &= \mathbf{E}(F(\deg(\phi_u^A))) < \mathbf{E}(e^{-kX}) = e^{-k\mu + k^2\sigma^2/2} \\ &\leq e^{-\frac{|\phi_v^B||\phi_u^A|}{4|E|^2}\left(\sum_{v\in V} d_v(d_v-2) - \frac{|\phi_u^A|+|\phi_v^B|}{4|E|}\sum_{v\in V} d_v^3\right)} \\ &= e^{-c^2 + c^2\frac{|\phi_u^A|+|\phi_c^B|}{4|E|}\frac{\sum_{v\in V} d_v^3}{\sum_{v\in V} d_v(d_v-2)}} = e^{-c^2+c^3\Upsilon H} \end{aligned}$$

Recall the bounds on H to complete the theorem. □

The preceding proof bounded the chance that messages from two nodes do not rendezvous. Subsequent queries between those two nodes either succeed or fail depending on the previous result. One might be interested in the probability that a BubbleStorm network *always* provides rendezvous for every pair. Subsequent attempts in the same graph are not independent, so such networks exist.

[7]The squared Δ is simply due to self-loops on the border.

COROLLARY 12. *A network always succeeds with chance*

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

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

PROOF. We bound the chance that the opposite is true

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

Apply Theorem 11 inside the sum $|V|^2$ times. □

By setting $c = \lambda\sqrt{2\log|V|}$ as $|E| \to \infty$, we can say this happens with $\mathbf{P} \geq 1 - e^{-\lambda^2}$. So, by making $c = \Theta(\sqrt{\log|V|})$, we can build systems which almost surely never fail.

5.3 Load

We will assume that load is injected at a node proportionally to its degree. This assumption makes the message sources independent and proportional to the edges, helping us derive the load variance. However, this is also a fairness criteria, as individual nodes contribute resources proportional to their consumption. If this assumption is unacceptable, bubblecast can be preceded by a simple random walk of logarithmic length. The final probability distribution of a random walk is proportional to node degree, as required.

A workload consists of a set M_A of type-A messages. $|m|$ for $m \in M_A$ is the size of the message. The type-A workload is, as before, $W_A = \sum_{m\in M_A} |m|$. These messages are replicated via bubblecast to form bubbles ϕ_m^A in the system. Similar definitions apply for type-B workload.

THEOREM 13. *An edge, e, chosen independently of the topology and load, carries type-A traffic T_A, with*

$$\begin{aligned} \mathbf{E}(T_A) &= c\sqrt{\frac{W_A W_B}{\sum_{v\in V} d_v(d_v-2)}} \\ \sigma^2(T_A) &\approx \mathbf{E}(T_A)\frac{\sum_{m\in M_A}|m|^2}{W_A} \end{aligned}$$

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

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

As e was chosen independently from the source of the load, $\mathbf{P}(e \in \phi_m^A) = \frac{|\phi_m^A|}{|E|}$ for $m \in M_A$.

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

The variance $\sigma^2(T_A) = \mathbf{E}(T_A^2) - \mathbf{E}^2(T_A)$, or

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

We would like the cross-term $\mathbf{P}(e \in \phi_{m_1}^A \cap e \in \phi_{m_2}^A)$ to equal $\mathbf{P}(e \in \phi_{m_1}^A)\mathbf{P}(e \in \phi_{m_2}^A)$ by independence. Unfortunately,

this is not true when the source of two bubbles is the same, but we assumed a smooth load distribution. So,

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

Substitute in $\mathbf{E}(T_A) = \frac{|\phi^A|}{|E|} W_A$. □

Now that we have both the expected load and the variance, we can calculate how well distributed the load is. As usual, $d_v \in o(\sqrt{|V|})$. We will also assume a maximum message size $M_{\max}$ and that W_A, W_B are proportional to $|V|$.

COROLLARY 14. *For edge e chosen independently of topology and load, $T_A \leq k\mathbf{E}(T_A)$ almost surely, for any $k > 1$.*

PROOF. We apply Chebyshev's inequality, to find

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

Therefore, as $|V| \to \infty$ the probability drops to zero. □

5.4 Optimality

We cite here a lower-bound result proven in [17], with adjusted notation. γ_v is the download capacity of a node.

THEOREM 15. *Any system which guarantees rendezvous of every type-A and type-B message must have a node which spent relative load (load divided by capacity),*

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

For comparison, Theorem 13 showed that when both type-A/B traffic are combined for all edges at a node,

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

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

Static systems of the same complexity can be built without a failure probability. However, our system is highly dynamic, and the failure probability is controllable. If a static system were scaled to the same number of participants as BubbleStorm, we doubt that its success probability would still be 100%, due to downtime, etc.

6. RELATED WORK

Often understood as the ancestor of unstructured P2P systems, the original Gnutella [13] lacks scientific design. Its simple hop-limited query flooding without any data replication is clearly not scalable and prompted widespread criticism. Nonetheless, Gnutella demonstrated the astounding potential of peer-to-peer systems and inspired most of the research in unstructured P2P systems.

Gia [5] is a Gnutella-inspired system that combines biased random walks for queries with one-hop data replication, topology adaption, and flow control. Delivering promising simulation results for non-exhaustive searches, unfortunately the authors do not provide mathematical analysis. When dissatisfied with its current neighbour set, a node actively searches for new neighbours and connects to them. This may disconnect other nodes, possibly making them dissatisfied, and thus leading to a chain reaction, causing non-local change in the topology.

Cohen and Shenker [6] analyze replication in unstructured P2P systems. When queries terminate after the first match, they show optimal replication is proportional to the square root of an item's popularity. Adjusting BubbleStorm to this approach would be easy: use different bubble sizes for different type-A messages. However, we do not take the popularity into account, because exhaustive search does not benefit from the extra replication of popular data.

Sarshar et al [15] enable exhaustive search on a Gnutella topology with sub-linear complexity. They combine random walk data replication with a two-phase query scheme. Queries are first installed along a random walk and then flooded with a probabilistic algorithm based on bond percolation. Traffic cost and success probability are analyzed. However, the only heterogeneity permissible corresponds to power-law graphs. While their system has $O(\sqrt{|V|}\log^2|V|)$ query complexity, it does so by introducing nodes of degree $\sqrt{|V|}$. They also consider nodes of degree $O(|V|)$, well beyond the point where centralization is superior. Even then, their system is $\log^2|V|$ off the optimum of the lower-bound met by BubbleStorm. It remains unclear how close they come to the bound under useful degree constraints.

An approach quite similar to BubbleStorm is described in [9]. Instead of bubblecasting, Ferreira et al use random walks of length $O(c\sqrt{|V|})$ to sample a random set of nodes. In contrast to our system they do not exploit the heterogeneity of node capacities, nor balance the random set sizes.

While most of the related approaches use random walks for rendezvous, we argue that random walks have high latency and are unreliable. Random walks do not exploit the natural parallelism of a distributed system. The latency of the random walk is proportional to its length, which may be unacceptably high for interactive applications. Furthermore, if a single node in the random walk crashes while processing the message, the operation fails completely. In contrast, bubblecasting offers a latency logarithmic to the bubble size. Occasional message loss reduces bubble sizes and thus success probability, but does not mean complete failure. In BubbleStorm we do use random walks for join, but these walks are short and not time-critical.

The topology maintenance of BubbleStorm has a join algorithm similar to that used in SWAN [3]. However, Bourassa et al are not trying to solve exhaustive search. Cooper et al [7] proved that repeated application of this algorithm converges to a random regular graph in the usual sense. This proof is based on switchings caused by leaves which connect

different neighbours and can disconnect the graph. BubbleStorm does not do this and is analyzed on the basis of an entirely different model. Other differences are that our joins/leaves are atomic, we don't need to repair crashed links, and we add heterogeneous degree.

Bit Zipper [18] solves the rendezvous problem using distributed hash tables (DHTs). Because the correctness is deferred to the underlying DHT, Bit Zipper itself has no failure probability. However, the system is not able to adaptively balance query and storage traffic, nor does it exploit heterogeneity. The usual, non-uniform allocation of DHT key space will also likely induce hotspots. This aside, it does meet the same lower-bound in the homogeneous case.

7. CONCLUSION

In this paper we presented the bubblecast algorithm and a random multigraph topology which, when combined, enable exhaustive search with adjustable probabilistic guarantees in large-scale heterogeneous systems.

Bubblecast is a new low latency, communication primitive that enhances traditional TTL-limited flooding by a precise size parameter. Additionally, the bubble can be searched incrementally slice by slice. Bubblecast could be easily used in cases where TTL-limited flooding is used today.

Although modelled as a circle permutation, the BubbleStorm topology is extremely robust against node failures. Yet to achieve this, only two maintenance operations are needed—join and leave—and they modify just a single edge. Still, the topology sustains the modelled randomness at all times.

By bubblecasting data and queries over the BubbleStorm topology the two components are combined into a search system. The probabilistic success guarantee $(1 - e^{-c^2})$ of the algorithm can be traded-off linearly with bandwidth consumption (c) by adjusting bubble sizes. Furthermore, the algorithm meets the lower-bound on per-node bandwidth.

All elements of the system are heterogeneity-aware. The potential benefit from heterogeneity in node capacity is analyzed and exploited. As every edge sees the same traffic, a node can choose its degree according to its capacity; hotspots are avoided. Thus, the combined algorithms offer a scalable approach to exhaustive search in distributed systems for cases where key-value indexes are not appropriate.

8. REFERENCES

[1] A. Awan, R. A. Ferreira, S. Jagannathan, and A. 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] B. Bollobás. *Random Graphs*. Cambridge University Press, 2nd edition, 2001.

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

[4] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of INFOCOM'99*, pages 126–134, 1999.

[5] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making gnutella-like P2P systems scalable. In *Proceedings of SIGCOMM'03*, pages 407–418, New York, NY, USA, 2003. ACM Press.

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

[7] C. Cooper, M. Dyer, and C. 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.

[8] V. Darlagiannis, A. Mauthe, and R. Steinmetz. Overlay Design Mechanisms for Heterogeneous, Large Scale, Dynamic P2P Systems. *Journal of Network and Systems Management*, 12(3):371–395, September 2004.

[9] R. A. Ferreira, M. K. Ramanathan, A. Awan, A. Grama, and S. 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.

[10] P. B. Godfrey and I. Stoica. Heterogeneity and Load Balance in Distributed Hash Tables. In *Proceedings of INFOCOM'05*, 2005.

[11] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, Modeling and Analysis of a Peer-to-Peer File-Sharing Workload. In *In Proceedings of SOSP'03*, Bolton Landing, NY, June 2003.

[12] D. Kempe, A. Dobra, and J. Gehrke. Gossip-Based Computation of Aggregate Information. In *Proceedings of FOCS'03*, page 482, Washington, DC, USA, 2003. IEEE Computer Society.

[13] T. Klingberg and R. Manfredi. Gnutella, June 2002. http://rfc-gnutella.sourceforge.net/developer/testing/.

[14] P. Maymounkov and D. Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *IPTPS'01: Revised Papers from the First Intl. Workshop on P2P Systems*, pages 53–65, London, UK, 2002. Springer-Verlag.

[15] N. Sarshar, P. O. Boykin, and V. 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.

[16] I. Stoica, R. Morris, D. Karger, and M. F. Kaashoek. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of COMM'01*, pages 149–160, San Diego, California, United States, January 2001. ACM Press.

[17] W. W. Terpstra. Distributed Cartesian Product. Master's thesis, Technische Universität Darmstadt, Darmstadt, Germany, May 2006.

[18] W. W. Terpstra, S. Behnel, L. Fiege, J. Kangasharju, and A. Buchmann. Bit Zipper Rendezvous—Optimal Data Placement for General P2P Queries. In *EDBT'04 Workshop on Peer-to-Peer Computing and DataBases*, March 2004.

[19] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 2nd edition, 1999.