BubbleStorm: Resilient, Probabilistic, and Exhaustive Peer-to-Peer Search

Wesley W. Terpstra[*] Jussi Kangasharju Christof Leng[†] Alejandro P. Buchmann

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

University of Helsinki
jakangas@cs.helsinki.fi

ABSTRACT

Peer-to-peer systems promise inexpensive scalability, adaptability, and robustness. Thus, they are an attractive platform for file sharing, distributed wikis, and search engines. These applications often store weakly structured data, requiring sophisticated search algorithms. To simplify the search problem, most scalable algorithms introduce structure to the network. However, churn or violent disruption may break this structure, compromising search guarantees.

This paper proposes a simple probabilistic search system, BubbleStorm, built on random multigraphs. Our primary contribution is a flexible and reliable strategy for performing exhaustive search. BubbleStorm also exploits the heterogeneous bandwidth of peers. However, we sacrifice some of this bandwidth for high parallelism and low latency. The provided search guarantees are tunable, with success probability adjustable well into the realm of reliable systems.

For validation, we simulate a network with one million low-end peers and show BubbleStorm handles up to 90% simultaneous peer departure and 50% simultaneous crash.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed Applications; C.4 [**Performance of Systems**]: Design Studies, Performance attributes; G.3 [**Probability and Statistics**]: Probabilistic Algorithms

General Terms

Algorithms, Experimentation, Performance, Reliability

Keywords

Peer-to-Peer, Exhaustive Search, Simulation, Resilience

1. INTRODUCTION

A recent Internet trend is toward online communities centred around user-generated content. Examples include Wikipedia, MySpace, and Flickr. So far, these communities have been built using the client-server paradigm, requiring a backing organization capable of providing hosting. The costs involved are enormous, as evidenced by Wikipedia's recent funding drive [19]. Furthermore, the organization might fail or provide service only under onerous conditions. Economically unviable communities are simply never built. The alternative, peer-to-peer based hosting, shifts the cost (and responsibility) to the community served. However, this presents several challenging technical problems.

Besides the usual challenges of scalability, adaptability, and robustness, these applications require flexible query languages. Recent DHT-based solutions to keyword search [13, 20] build another layer on top of the underlying DHT. In addition to increasing bandwidth and latency, their customized routing is specifically designed for a single query algorithm. We believe that query algorithms will continue to evolve, and that coupling them to the network is not a wise long-term solution. Hence, our system's goal is to allow an application designer to implement his query evaluator for one machine, with the assurance that queries could (in principle) be evaluated against all documents. For key-value search, DHTs remain the appropriate complementary solution.

In this paper, we present BubbleStorm, a system separating query evaluation from network topology. It simplifies the design of distributed query languages by providing a network-level search strategy which does not compromise scalability, adaptability, or robustness. Key features include:

Powerful Search: BubbleStorm performs an exhaustive search, which probabilistically guarantees that the application's query evaluator runs on a computer containing the sought data. Since query evaluation is performed by a computer holding the document, the application developer is free to use any powerful query evaluator (XPath, full text, relational, etc). This is in stark contrast to DHTs (e.g. [12]), where the query evaluator is key-value lookup and search algorithms are hand-crafted on top of this primitive.

Scalability: Our results show that BubbleStorm easily handles networks built from one million 10kB/s nodes, approaching the scale of popular public peer-to-peer networks.

Fast Search: For a 100-byte query to reach the single matching document in a million nodes, BubbleStorm needs 600 ± 200ms and 115kB total network traffic, irrespective of query language. Our simulated network has poor physical locality; performance in a real deployment is likely better.

[*] 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).

SIGCOMM'07, August 27–31, 2007, Kyoto, Japan.
Copyright 2007 ACM 978-1-59593-713-1/07/0008 ...$5.00.

Heterogeneity: BubbleStorm exploits mixed bandwidth distributions to considerably improve its performance. Under reasonable assumptions, total query traffic drops by a factor of 3.6 compared to a homogeneous network. In our simulated application, this brings search traffic to parity with periodic overlay maintenance traffic. Most existing work on overlay-based search concentrates only on query traffic, assuming periodic maintenance traffic is negligible.

Optimality: When powerful search is required (defined above), BubbleStorm has optimal traffic. No system can exhaustively search with asymptotically less traffic (Section 3).

Load Balancing: With Zipf-distributed keyword popularity [3] or similar file-sharing workloads [9], structured systems suffer from hot-spots. BubbleStorm avoids this problem—keywords have no specific destination.

Robustness: Due to its unstructured and randomized topology, BubbleStorm survives catastrophic failure. Even during 90% simultaneous peer departure, queries continue to complete successfully. If 50% of peers crash simultaneously, the topology heals within one minute and successful search function is restored within 15 seconds. Although queries do fail during the 15 second crash recovery, the effect depends heavily on the crashed percentage, see Section 5.4.

This paper evaluates BubbleStorm by simulating a one million node network (Section 5). In our simplified application, users can publish documents and execute searches in *any* query language. We explain the intuition behind BubbleStorm in Section 2 and then provide an overview of BubbleStorm's three protocols: randomized multigraph maintenance (Section 2.1), epidemic sum calculation (Section 2.2), and query/data replication (Section 2.3). Section 4 highlights BubbleStorm's robustness by including torturous evaluation scenarios and we compare against Gnutella and a random walk system [7] in Section 6.

2. SYSTEM

To achieve exhaustive search, we must ensure that a query could be evaluated on every datum. This implies that for every (query, datum) pair, some node can perform the evaluation. In a DHT, where every query and datum is mapped to a key, the node responsible for the matching key performs the evaluation. However, BubbleStorm is designed for more difficult queries which resist such simple binning. The alternative taken by Gnutella is to replicate the query onto many nodes, which then evaluate the query on all the data they store. While this works, it does not scale; exhaustive search would require replicating the query onto every node. Publish-Subscribe systems [6] typically operate in the opposite manner, replicating the datum onto the nodes (guided by the subscriptions). BubbleStorm takes a hybrid approach: both queries and data are replicated.

Though BubbleStorm performs exhaustive search, it does so probabilistically—it can fail. Let the replicas of a query be red balls and the replicas of a document's meta-data green. Over n nodes, distribute r red balls and g green balls uniformly at random. The chance that no node has both a green and red ball is less than $e^{-rg/n}$. Notice that only the product rg matters; one can increase g while decreasing r and retain the same probabilistic bound. For example, if $rg = 4n$, the chance of a query finding the only matching document is greater than $1 - e^{-4} \approx 0.9817$. The optimal trade-off between r and g is derived in Section 2.4.

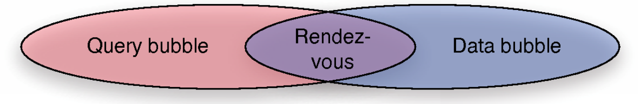


Figure 1: Large enough bubbles likely intersect

To exploit this, we must replicate a query to a set of independently chosen nodes with given set size. Contacting all nodes within h hops has low latency, but h cannot control the number of nodes reached precisely. For this purpose, we use a new communication primitive, called *bubblecast*. Its execution imposes a subgraph or *bubble* on the network of specified size—in the absence of cycles. When every query is replicated within a query bubble, and every datum in a data bubble, then the nodes receiving both are rendezvous nodes which evaluate the query on that datum (Figure 1). What remains is to ensure that explored nodes are independent and bubbles are cycle-free.

Fortunately, random graphs have these properties. Every edge leads to a new random sample. Within a small area, there is almost surely at most one cycle [1]. Certainly, one must prove that two bubblecasts over our particular random graph intersect with probability $1 - e^{-rg/n}$. The full derivation can be found in [18], but our focus in this paper is validating performance and robustness by simulation.

2.1 Topology

Because BubbleStorm's overlay is a random multigraph, finding random nodes is as easy as exploring edges. Random graphs are also resilient; even if a large proportion of nodes crash, a giant component remains [8]. Our graph is not regular, but assigns degree proportional to bandwidth. As edges in BubbleStorm carry the same average traffic, this choice spreads relative load evenly over a mixed bandwidth network, facilitating the use of high bandwidth nodes.

Preserving fixed node degree requires that joins and leaves only affect the degree of the executing node. From a high-level point-of-view, our join algorithm interposes a peer in the middle of a random existing connection. Thus the joining peer acquires two neighbours, while leaving their degrees unchanged. When leaving, the process is reversed and this same pair of neighbour connections is spliced back together. A node executes the join or leave algorithm multiple times in parallel to reach the degree dictated by its bandwidth.

The initial node in a BubbleStorm network connects once to itself, forming a circle of one edge. All subsequent joins operate as explained, interposing the joining peer into this circle multiple times. This circle makes our network a multigraph (self-loops and double-edges are allowed). We interpose joining nodes on random edges so that all node permutations on this circle are equally likely. As leaves never break the circle, without crashes it is impossible for the network to disconnect. While multi-edges and self-loops may seem redundant, they greatly simplify system design and implementation and are rare and easily avoided by bubblecast.

To keep the network topology consistent, we require that all join and leave operations are serializable. A conflict can occur between (multiple) join or leave attempts on the same edge. To prevent this, we use TCP for in-order delivery and designate one peer on every connection as its master, responsible for serializing these operations. The other peer

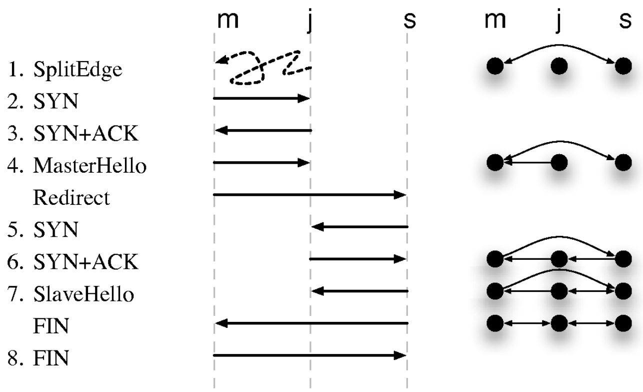


Figure 2: j joins between master m and slave s

is a slave, which must ask the master's permission to leave. When combined with careful link shutdown, in the absence of crashes, messages are never lost.

To bootstrap, a node must know the address of an existing peer. A persistent cache of previously seen peers can be used for this purpose. Next, the node sends a SplitEdge message on a random walk of length $3(1+\log n)$ to select an edge uniformly at random [2, 7]. Figure 2 illustrates the subsequent error-free message exchange of the join algorithm.

Peers should strive to execute the leave algorithm, shown in Figure 3. Although peer crashes almost surely do not disconnect the network, they will reduce the degree of neighbours. For this reason, peers tolerate one neighbour less than their desired degree. They join once if two neighbours crash. BubbleStorm makes no attempt to heal edges broken by crashes. System churn slowly eliminates them. The tail of Figure 6(h) illustrates this steady reduction.

BubbleStorm is implemented as a super-node network. Super-nodes (aka peers) are responsible for routing bubblecast traffic, measuring the network, and maintaining graph structure. Nodes not powerful enough for these activities become clients to a peer, issuing queries by proxy. In fact, all nodes start as clients while joining the topology as a peer.

2.2 Measurement

BubbleStorm includes a protocol for measuring global system state. It measures the number of query and datum replicas needed to ensure that for each (query, datum) pair there exists a rendezvous node. Section 2.4 computes this replication factor from D_1 and D_2 where

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

The join algorithm makes random walks of length $3 \log n$, $n = D_0$, so we measure D_0, D_1, and D_2. While measurements converge to arbitrary precision, only $\approx$5% is needed.

The measurement protocol piggy-backs on the periodic keep-alive messages used by the overlay to detect crashed neighbours. Keep-alive messages in BubbleStorm are sent to all neighbours every five seconds and include an extra 24 bytes for the measurement algorithm.

Our algorithm is based on work by Kempe, Dobra, and Gehrke [10]. They prove their algorithm converges to the sum of a variable over all nodes in $O(\log n)$ message rounds. Unfortunately, their algorithm requires a designated leader. We extend their work by eliminating this requirement [17].

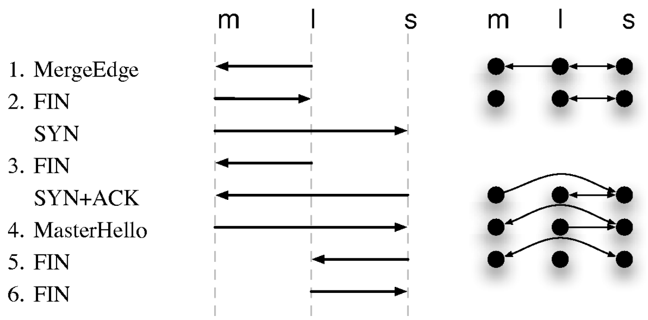


Figure 3: l leaves between master m and slave s

A useful analogy is measuring a lake's volume. By letting loose a school of fish into the lake, and assuming they spread-out uniformly, one can measure the lake's volume by counting the fish in a cubic metre of water. Peers manage lake regions, whose initial volumes are the variables to sum. Each peer picks a random fish size and puts 1.0 fish into its region (fish are infinitely divisible). When sending keep-alives, peer v mixes its lake region with its neighbours and itself; everyone gets $\frac{1}{\deg(v)+1}$ of the water and fish. Bigger fish in a region consume all smaller fish. Thus the biggest fish swims/diffuses throughout the lake until only it remains and is uniformly distributed. After $O(\log n)$ rounds, every peer's ratio of water divided by fish equals the sum.

By using a random fish size, exactly one fish will remain, without needing a leader to release it. However, once the biggest fish has spread uniformly throughout the lake, the protocol should update the measured values and restart. Our solution includes a measurement counter that increases whenever stability is reached. If a peer receives fish and water with a future counter, it moves its counter forward to match. When a peer has not seen the water/fish ratio differ in any messages (within a 1% tolerance) for five keep-alives, it increases the measurement counter itself. New peers do not add fish or water until their counter is increased once.

2.3 Bubblecast

Bubblecast replicates queries and data onto peers in the network. It hybridizes random walks with flooding, combining the precisely controlled replication of random walks with the low latency of flooding. Although bubblecast can be used on any graph with good expansion, it works especially well on our random topology.

As input, bubblecast takes the item to replicate, the desired number of replicas (weight), and s, the system *split factor*. After decreasing the weight by one, bubblecast processes the replica locally, e.g. storing the data or executing the query. Since evaluation is local, the query algorithm can be as sophisticated as desired. If the weight is not zero, bubblecast distributes the remainder between s neighbours, randomly chosen for forwarding (see Figure 4). Bubblecast never forwards to the neighbour who sent it the message.

Like a random walk, bubblecast weight controls the exact number of nodes contacted. Unlike a random walk, it reaches an exponentially growing number of nodes per step. While flooding is similar, it uses a hop counter and contacts all neighbours. Therefore, flooding only controls the logarithm of replication and is affected by the degree of nodes.

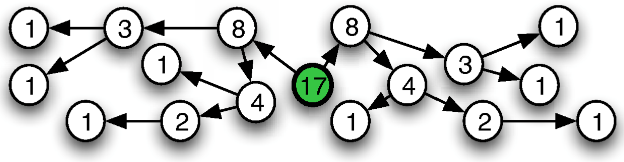


Figure 4: Bubblecast 17 replicas with $s = 2$

Bubblecast forwards to s neighbours so that all edges will carry the same average traffic. In a flooding system, connections from high degree nodes carry more traffic because those nodes receive more traffic. By fixing the split factor, the Markov stationary distribution of traffic under load is the same as for a random walk (all edges equally probable). Although bubblecast messages travel further than in flooding, reduced congestion makes each bubblecast hop faster.

For BubbleStorm searches to succeed, bubblecast must create the required replicas reliably. Compared to random walks, bubblecast handles packet loss better. When a packet is lost, both approaches lose replicas equal to the packet's current weight. Due to exponential division of the initial weight, w_0, bubblecast packets have weight $\approx \log_s w_0$ on average as opposed to $\frac{w_0}{2}$ for random walks.

Another concern is that bubblecast might reach the same node twice, reducing the replication. For our multigraph topology, bubblecast is careful not to follow self-loops and only forwards to a neighbour once (even if multiple connections exist). Larger cycles can not be detected locally, and *do* reduce bubblecast's replication. However, random graphs have mostly large cycles [1], so this is rare and has little effect as simulation in Section 5.1 shows.

2.4 Bubble Size

Correctness in BubbleStorm depends on the number of query and data replicas (q, d). For a given (query, datum) pair, BubbleStorm is successful if there exists a rendezvous node receiving a replica of both. For independently and uniformly placed replicas, this fails to happen with probability less than $e^{-qd/n}$. Assume for the moment this holds true.

The entire network computes the same values for q and d, so that all queries and data have the same replication factor. These values depend on the required reliability, r, and the query/data traffic balance, which an application designer estimates. Define the certainty factor as $c := \sqrt{-\ln(1-r)}$, a real-valued system parameter shown in Table 1. By this definition, $e^{-c^2} = 1 - r = e^{-qd/n} \implies qd = c^2 n$. While we have determined the product qd, the ratio remains a free variable. BubbleStorm exploits this freedom to balance bubble sizes. Let R_q and R_d be the rates, in bytes/second, at which queries and data are injected into the system—before replication. The total system traffic is then $qR_q + dR_d$ subject to $qd = c^2 n$. Minimizing traffic leads to balanced bubble sizes of $q = c\sqrt{nR_d/R_q}$ and $d = c\sqrt{nR_q/R_d}$. The ratio R_q/R_d is the traffic balance which the developer estimates.

The equation $e^{-qd/n}$ is a simplification. Query replicas are not placed independently from data replicas, due to the shared network topology. Furthermore, bubblecast does not sample uniformly from the graph, but proportional to node degree, like a random walk. This non-uniform sampling is actually an advantage; a node with twice the degree receives twice as many replicas of both queries and data. Therefore, it serves as a rendezvous for four times as many pairs. For our network, when the match threshold, T, is defined as,

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

Table 1: Match probability as a function of c

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

and n is replaced by T in the definitions of q and d,

$$q := \left\lceil c\sqrt{TR_d/R_q} \right\rceil, \qquad d := \left\lceil c\sqrt{TR_q/R_d} \right\rceil$$

the corrected failure probability [18], is bounded by

$$\mathbf{P}(\text{match failure}) < e^{-c^2 + c^3 H\Upsilon}, \text{ where } H \to 0 \text{ as } n \to \infty$$

The technical error term $H\Upsilon$ is small enough to ignore when node degree is $o(\sqrt{n})$, a condition BubbleStorm enforces.

As the network size and degree distribution change, the measurement protocol provides estimates on D_1 and D_2, and thus T. From T, bubblecast computes q and d. However, an intuitive understanding of T is hard. When the network is homogeneous, $T = n\frac{\deg}{\deg - 2}$; the -2 reflects the interdependence of query and data replicas. The denominator includes squared degrees (D_2), which reduce T as the heterogeneity increases. Smaller T means less replicas needed per bubble.

2.5 Congestion Control

As BubbleStorm strives for robustness, it must not succumb to congestion overload. A congested peer has higher latency when communicating with its neighbours. If this delays SplitEdge routing or keep-alives, the timeouts guarding against crashed peers might expire. The associated recovery traffic increases congestion, leading to more delays.

Generally speaking, the bottleneck in BubbleStorm is the uplink. Consider a peer receiving a bubblecast message. It potentially forwards that message over several of its TCP connections. Therefore, although each TCP connection sees average traffic, that traffic is correlated in time on the uplink wire. This effect amplifies the natural load burstiness. For peers with more download than upload bandwidth, this problem is even more pronounced. When uplink bandwidth is managed, we have not seen peers with congested downlink, so BubbleStorm's congestion control manages transmission only. We defer downlink congestion control to TCP.

Our congestion control strategy assigns priorities to all messages. Each message type has a priority measured in uplink congestion seconds. A peer models its output queue according to its bandwidth constraint. For example, if a peer with a 10kB/s uplink has 20kB in its queue, its congestion is 2 seconds. When the queue is empty, the congestion is 0 seconds. Messages are sent when their priority is larger than the link congestion. Topology maintenance and keep-alive messages have highest priority and are always enqueued. Bubblecast message priority depends on weight. A bubblecast query packet with weight w has priority

$$2 + 2\frac{\log(w)}{\log(q)} \text{ seconds}$$

When a bubblecast packet has too low a priority to be sent, it is simply discarded. This policy prevents dropping bubblecast packets which must still be replicated to many nodes.

2.6 Replica Maintenance

After bubblecast has installed replicas, the system must maintain the replication against churn and possibly update the data's value. Because the replicas are scattered throughout the network, updating values in-place seems practically impossible. However, installing new versions of the data is a reasonable alternative; querying nodes can distinguish new values from old ones based on a version number. Alternatively, one could replicate patches to old values for applications with a version history, like a wiki. To delete out-dated replicas, one can add to replicas a time-to-live, after which they are deleted. Which approach (and parameters) to use is application-specific and not covered here.

The second problem, maintaining the desired replication, can be solved elegantly in BubbleStorm. Using churn statistics derived from the measurement protocol, we are still evaluating several possible solutions for this topic.

3. ANALYSIS

Under Section 1's match requirement, that a query is evaluated on every datum by some node, BubbleStorm is nearly optimal. In this proof sketch, we consider λ, the largest percentage of bandwidth used on any node. When $\lambda = 1$, at least one node is completely saturated. As an objective function, minimizing λ forces load balance in a heterogeneous distributed system. Only minimizing packet count would ignore load balance and variable packet size. Define B_v and T_v as the bandwidth and traffic at node v. Then,

$$\lambda := \max_{v \in V} \frac{T_v}{B_v}$$

To compute λ for BubbleStorm, recall from Section 2.4 that the total bubblecast traffic is $qR_q + dR_d$. In the steady state, traffic is distributed equiprobably between links, so node v sees $\frac{\deg(v)}{D_1}$ of it. Recalling that $\deg(v)$ is proportional to B_v, set $b = \frac{B_v}{\deg(v)}$. Then, for BubbleStorm,

$$\begin{aligned} \lambda &= \max_{v \in V} \frac{\deg(v)}{D_1 B_v}(qR_q + dR_d) = \frac{2c}{D_1 b}\sqrt{\frac{D_1^2 R_q R_d}{D_2 - 2D_1}} \\ &= 2c\sqrt{\frac{R_q R_d}{\sum_{v \in V}(B_v^2 - 2B_v b)}} \approx 2c\sqrt{\frac{R_q R_d}{\sum_{v \in V} B_v^2}} \end{aligned}$$

To compute the lower-bound on λ, we require that every query meet every datum on some node. A necessary condition for this is that every query byte must meet every datum byte. After t seconds, there are $(R_q t)(R_d t)$ byte pairs to be matched. A node v downloads at most $\lambda B_v t$ bytes in this time. Nodes can only match as many byte pairs as the product of query and data bytes they downloaded. This product is maximized when the downloaded query and data bytes are equal, so $(\lambda B_v t/2)^2$ are the most pairs node v can match. Utilizing all nodes equally, the lower bound on λ is

$$\sum_{v \in V} (\lambda B_v t/2)^2 \geq (R_q t)(R_d t) \implies \lambda \geq 2\sqrt{\frac{R_q R_d}{\sum_{v \in V} B_v^2}}$$

Therefore BubbleStorm is optimal to within the factor c.

4. SIMULATION

We opted to use a custom simulator with BubbleStorm. As BubbleStorm is TCP-based, we knew that we would need a light-weight approximation. Previous analytic work had shown that upload and download queueing effects would be important, so our simulator includes ingress and egress congestion. We wanted to simulate BubbleStorm with 1 million peers, approaching the size of real peer-to-peer networks. This required very tight memory management to fit the replica databases, message queues, and event heap.

The simulated topology places every node at a position on Earth, chosen uniformly at random, simulating a worst-case scenario for latency. Each node has an upload and download link bandwidth and end-point specific delay. The total flight-time for a message includes:

- the queueing delay while the message waits in line to be uploaded
- the sender's last-hop link delay
- twice the time required by light to travel the arc-length between the nodes on earth
- a normal random variable with 5 ± 5ms
- the receiver's last-hop link delay
- the queueing delay while the message waits in line to be downloaded
- waiting for previous, out-of-order TCP segments

Our simulation of TCP is rather simple and disables the Nagle algorithm. The main purpose of TCP in our system is reliable in-order delivery. We simulate this and the usual SYN, SYN+ACK, ACK sequence and tear-down via FINs.

Nodes are implemented in C++ with statically allocated memory, except for their message buffers. They hook TCP callbacks (onConnected, onRejected, onAccept, onReceive) and interact with the system solely via sending messages and connecting to addresses. The BubbleStorm 'database' is a simple bit vector recording documents received. Simulations took 20-50 hours per run on an Opteron 2.0GHz, consuming from 7.6GB of RAM to 13GB for heterogeneous scenarios.

4.1 Configuration Parameters

We simulate a one million peer distributed wiki subjected to different 8 minute failure scenarios. The simulator first goes through a warm-up phase to reach the target size. After the measurement protocol has stabilized (3 minutes), we begin injecting bubblecast traffic. This marks the start of the 8 minute test. One minute later, the simulated event occurs, and the following 7 minutes record the effect.

To push the system close to its limits, we use very weak peers. As reasonable bandwidth distribution causes nearly a four-fold bubblecast traffic reduction (Section 5.5), we take a homogeneous case as our control scenario. In this setup, every peer has only 10kB/s upload capacity, 100kB/s download capacity, and a 40ms local link delay. This corresponds to an ADSL-1000 node with poor line quality. Every peer has a degree of 10 and thus joins 5 times. We could certainly simulate more powerful peers, but then the load and failure scenarios would not stress the network.

Unless otherwise specified by the scenario, peers have a pre-calculated lifetime taken from the exponential distribution with mean 60 minutes. At the end of their lifetime they either crash or leave. A more accurate lifetime would be

$$\mathbf{P}(\text{lifetime} \leq t) = \frac{1 - e^{-\frac{\alpha t}{\alpha t + 1}}}{1 - e^{-1}}, \text{where } \alpha = \frac{1}{72}$$

which fits the session duration plot from [15]. However, our exponential random variable chooses shorter lived nodes, which is a good worst-case assumption. Using an exponen-

	Homogeneous		Heterogeneous	
n	Query	Data	Query	Data
10k	328	153	163	76
100k	1036	483	327	152
1M	3276	1527	950	443

Table 2: Number of replicas for $c = 2$

Link type	Pop	Upstream	Downstream	Last hop
ADSL-1000	60%	16kB/s	128kB/s	30ms
ADSL-2000	25%	32kB/s	256kB/s	20ms
1Mbit	10%	128kB/s	128kB/s	1ms
10Mbit	5%	1.28MB/s	1.28MB/s	1ms

Table 3: Heterogeneous scenario bandwidths

tial lifetime has the marked advantage that we need not worry about the lifetime distribution after reaching our target network size; exponential distributions are memory-less.

To quickly grow a BubbleStorm network to the desired size, there is a phase before every test where no bubblecast traffic is transmitted, but the network receives 10% of its size in new peers every 10 seconds. The normal join algorithm is used. Once the target one million peers have been added, we switch to a Poisson arrival rate that balances the exponential leave rate. As exponential random variables have no memory and are symmetric in time, we assign every living peer an age taken from the lifetime distribution. Thus the age distribution is correct even though all peers are in reality no older than a few minutes. Once the network has correct age and lifetime distributions and size, we can inject bubblecast traffic derived from peer uptime.

To simulate a wiki, we assume that article meta-data is 2kB and new articles are created every 30 user minutes. Similarly, queries are 100 bytes (before headers) and are injected on average every 5 user minutes. However, the actual injection distribution follows the 80/20 rule. The first 20% of a peer's lifetime is used to inject 80% of its traffic; this rule is applied recursively. The bubblecast split and certainty factors were set to $s = 2$ and $c = 2$ for a success rate of $1 - e^{-c^2} \approx 98.2\%$. A real system would probably choose $c = 3$ for 99.99% at a cost of 50% more traffic. However, this failure rate is too small to measure via simulation with any accuracy. The resulting bubble sizes ($c\sqrt{TR_d/R_q}$ and $c\sqrt{TR_q/R_d}$) for this wiki query/data traffic ratio are shown in Table 2. The heterogeneous T is derived from Table 3, the bandwidths used in the heterogeneous simulation scenario.

To measure bubblecast success, we colour some query and article bubblecasts at 100Hz. Wherever a coloured article bubblecast goes, it sets a bit in the peer's bit vector database. Twenty seconds after an article was coloured green, a query for green is bubblecasted. Queries report success if any of the replicated queries arrive on a green peer. Only marked bubblecasts report latency and number of peers contacted.

4.2 Scenarios

We investigate the performance of our system in a variety of scenarios which demonstrate the efficiency and robustness of the system. The scenarios represent both realistic operating conditions and failure scenarios which test robustness. Every scenario is run 11 times for reliable comparison.

Pure Churn: This scenario provides a base line for comparison to other scenarios. Peers join with Poisson arrival rate and leave the system with exponential departure rate. The test starts with the population already at equilibrium. We assume orderly departure, i.e., no peer crashes. Injected load has 80/20 distribution.

Massive Leave: This is an extension of the Pure Churn scenario. In addition to normal churn, a large number of peers leave the network simultaneously after one minute. These departures are again only normal leaves, not crashes. We simulate 50% and 90% simultaneous departure in this scenario. While these percentages are extremely high for realistic scenarios, they clearly demonstrate the robustness of our system. Furthermore, simultaneous leaves highlight the atomic nature of master-slave link management.

Churn with Crashes: This is the most realistic homogeneous scenario we investigate. It is identical to the Pure Churn scenario, with the exception that some peer departures are crashes, not following the leave protocol. We set the ratio of crash departures to 10%.

Massive Crash: This scenario builds on Churn with Crashes, but crashes a large percentage of peers simultaneously. We investigate cases where 5%, 10%, and 50% of the peers die. This scenario shows how quickly BubbleStorm is able to recover from catastrophic failure.

Heterogeneous Network: BubbleStorm is not a homogeneous peer-to-peer system. This scenario extends Churn with Crashes to include peers with different network capabilities. Because BubbleStorm is a super-node network, peers providing less than 16kB/s upstream bandwidth are excluded. Modelled peers and their associated link parameters are summarized in Table 3. In this scenario peer degree is proportional to uplink bandwidth.

5. RESULTS

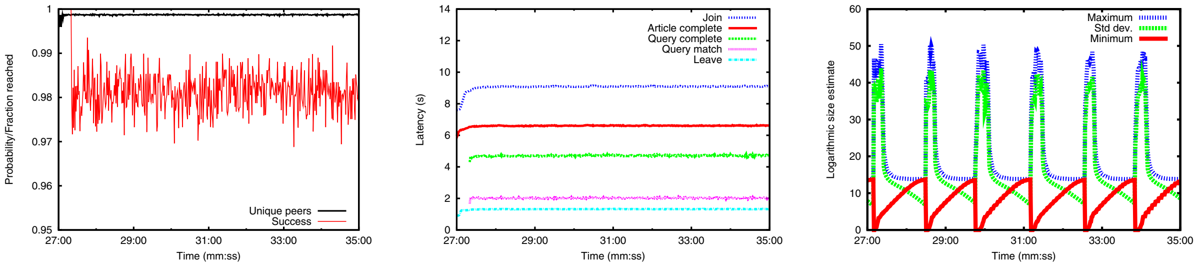
We measure BubbleStorm's query behaviour, topological integrity, and measurement accuracy. For bubblecast, the simulator provides information on success probability and the size of the imposed query/data subgraphs. The latency of query and topology operations are measured, as are the percentage of nodes with full or nearly full degree—this helps measure peer assimilation rate. Measurement (Section 2.2) convergence affects the success of BubbleStorm; knowing when a new T is used helps to explain the sudden transitions in search performance. Most key operations are affected by congestion, so we also analyze the maximum and average uplink backlog and traffic break-down.

In all plots, the x-axis shows the time since the beginning of the simulation in minutes. For nearly all plots, the y-axis represents an average of 11 independent runs. The initial 27 minutes include the warm-up exponential network growth and an additional three-minute delay to let the system reach equilibrium. After one minute of bubblecasting, we introduce the simulated anomaly at 28 minutes. The contents and y-axis for the different plots are as follows:

Bubblecast: Shows how often queries find the single matching datum and the percentage of unique peers reached.

Latencies: Shows the latencies of completing query and data bubblecasts, joining and leaving the network, and the time till a query first finds its match,

Measurements: Shows the maximum and minimum estimates on network size and standard deviation of all peer measurements on a logarithmic scale. Because the protocol switches rounds at different times, these plots cannot be av-



(a) Bubblecast (b) Latencies (c) Measurements

Figure 5: Pure churn

eraged. If the plots are not qualitatively the same, the most interesting plot is examined and the discrepancy explained.

Node degrees: Shows the percentage of nodes with a given number of edges. Figure 6(h) also shows the fraction of nodes executing the leave algorithm.

Congestion: Shows the maximum and average node uplink congestion in seconds. A node with nothing in its uplink queue has 0 seconds of congestion. The average includes standard deviation error bars. The right-hand y-axis shows the average amount of traffic seen by a node.

Traffic breakdown: Shows bubblecast, keep-alive, and topology traffic measured in GB/s for the entire network.

5.1 Pure Churn

Pure Churn is unremarkable. Success rate hovers in the expected 98% range. For the requested replication, approximately 99.9% of the replicas arrive on unique/distinct peers. As there are no crashes and congestion is low, this 0.1% loss is from cycles in the graph. Still, the reduction is small enough to justify omission of a cycle-avoiding algorithm.

While searches take on average about 4.5 seconds to complete (Figure 5(b)), surprisingly most successful matches happen within 2 seconds. This is related to the c factor; we look further than needed for 50% confidence. As queries and data are propagated further than necessary, they probably intersect in advance of complete bubble exploration. This causes the fast search phenomenon. As average match time is what the user sees, this is a very attractive property. Keep in mind that nodes are uniformly distributed on the earth and 80ms is added by the end-point links. Real deployments would probably see even better latency.

Joins are routed via a random walk of length $3\log(n) \approx 60$ hops in a one million node network. Even though the join delay is about 9 seconds, BubbleStorm is a super-node network allowing new nodes to execute searches immediately.

The measurement protocol periodically converges in Figure 5(c). Each time the standard deviation drops to an acceptable level, the measurement protocol begins a new round, resulting in the sudden peaks. After the biggest fish (from Section 2.2) has spread throughout the lake, the standard deviation drops linearly in our graph. Thus it is improving exponentially (the scale is logarithmic). The minimum corresponds to the peer which had produced the largest fish; it initially has an unnaturally high concentration of fish. As the deviation drops exponentially, it is clear that the maximum and minimum also converge quickly. Oddly the maximum estimate converges especially fast, while the minimum converges exponentially as expected.

5.2 Massive Leave

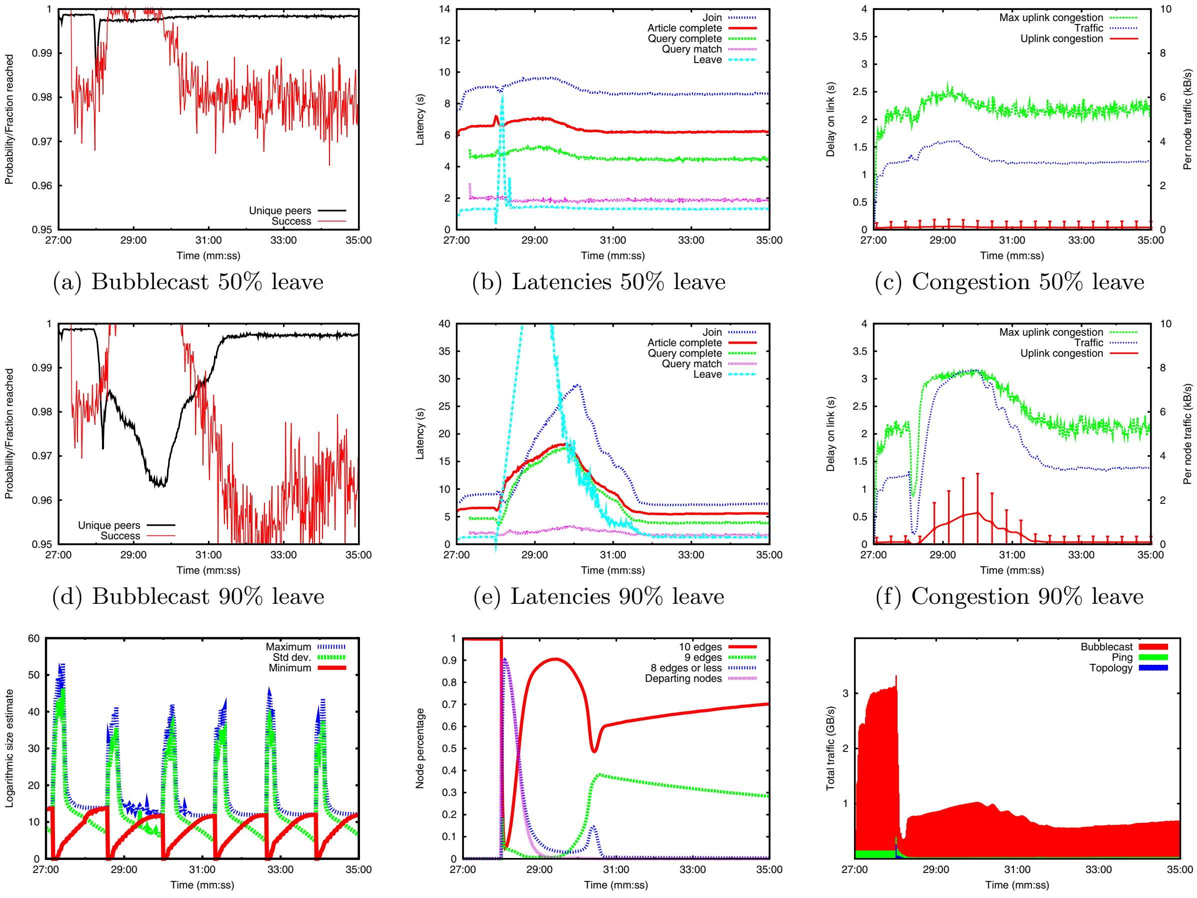
Peer exits should not decrease search success. Figures 6(a) and 6(d) show an initial success rate of 98% as expected. After the departure, both scenarios jump up to nearly 100% success. This is because the measurement protocol (shown for 90% leave in Figure 6(g)) does not provide a new network measurement until 30:25. Prior to then, the bubble sizes are still being calculated with a match threshold appropriate for a much larger network. Thus, the bubbles are bigger than required, resulting in an artificially higher c value. Once the measurement protocol calculates a new value, the success returns to the expected range, except for the 90% leave case which has only 96%, discussed at the end of this section.

The master-slave link protocol serializes operations. If a master is leaving, its slave cannot leave until the master has finished. This creates leave dependency chains in the network. With only 50% of the network departing, the chains are not long. A chain of length greater than two is only 25% likely, three is 12.5%, and so on. In contrast, with 90% departure, chain lengths greater than five long are still 60% likely. Therefore, the effect on leave times is far more extreme in the 90% case (figures 6(b) and 6(e)). Nevertheless, after four minutes even the extremely long chains in the 90% scenario have been flushed—see Figure 6(h).

In both leave scenarios, the Poisson arrival rate remains unchanged. As the exponential leave rate times the current population no longer balances the arrival rate, the system begins growing. The system would eventually re-stabilize at one million nodes. Effectively, in the 50% case, the system is coping with twice the previous relative arrival rate versus ten times the relative arrival rate in the 90% case. While the topology handles this growth rate and the measurement protocol similarly converges, the 80/20 traffic is a problem.

When a peer joins, the simulator injects most of its data and queries immediately. As the relative arrival rate is significantly increased and the bubble sizes are artificially inflated, this leads to a traffic explosion. The 50% case is not so extreme, and Figure 6(c) shows that the traffic undergoes a brief increase that stops at 30:00 when the new threshold value kicks in. This traffic causes an average congestion bump from 40ms to 60ms. As congestion affects link latency, this accounts for the slightly slower operation times shown by Figure 6(b) during the event recovery.

The same effect is much more pronounced for the 90% leave scenario. Significantly higher per-node traffic hits the system, resulting in a bubblecast reachability drop (Figure 6(d)) to cope with congestion. Yet the success rate remains undamaged (actually it improves) due to congestion



(a) Bubblecast 50% leave

(b) Latencies 50% leave

(c) Congestion 50% leave

(d) Bubblecast 90% leave

(e) Latencies 90% leave

(f) Congestion 90% leave

(g) Measurements 90% leave

(h) Node degrees 90% leave

(i) Traffic breakdown 90% leave

Figure 6: Massive leave

control, which drops only low weight packets. Thus, oversized bubbles, reduced in size, still remain larger than required and successfully intersect. Once a new measurement kicks in, reachability shoots back up as congestion clears.

Not all is roses, however. The transient congestion is much higher—a full 650ms average! This increases the latency on all operations, but SplitEdge must be routed for ≈ 50 hops. The average congestion results in the 30s SplitEdge timeout firing. This in turn leads to broken edges when SplitEdge is retransmitted and causes two splits where only one can be satisfied. Therefore, 30 seconds after the congestion peak, Figure 6(h) shows a rapid increase in degree 9 nodes due to broken edges. However, our system is designed to cope with broken edges and this presents only an aesthetic problem.

Another artifact of simulating the 90% scenario is that the subsequent success is 2% below what it should be. The network is growing by approximately 21% every 90 seconds, due to the unchanged Poisson arrival rate. Although this aftereffect is unrealistic, it is interesting. As the measurement protocol takes 90 seconds to resolve new threshold values, on average the network size is underestimated by 10%. This reduces c, so that $1-e^{-(0.9c)^2} \approx 0.96$ which accounts for the missing 2%. Normally, a network would not be expanding this quickly, but if this is important for a reliable application, increasing traffic by 10% (and thus increasing c) would retain the desired probability during this growth.

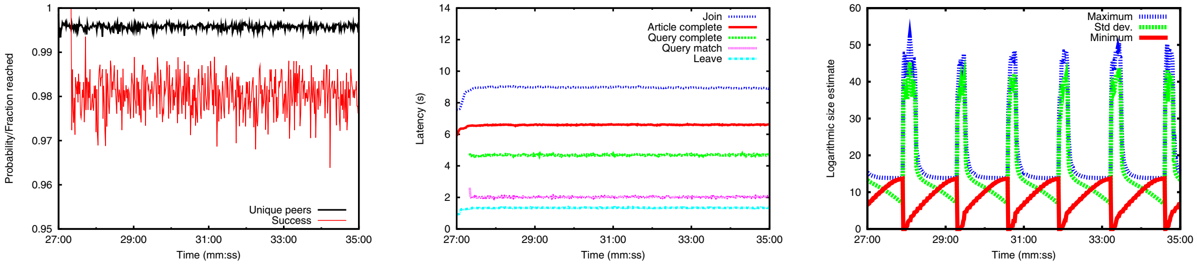
5.3 Churn with Crashes

Churn with Crashes is similar to the Pure Churn scenario, except that 10% of the peer departures are crashes not following the leave protocol. The only difference between Churn with Crashes (Figure 7) and Pure Churn (Figure 5) is a 0.4% drop in unique peers reached. However, search success is not appreciably affected, confirming that occasional peer crashes do not hurt the system.

5.4 Massive Crash

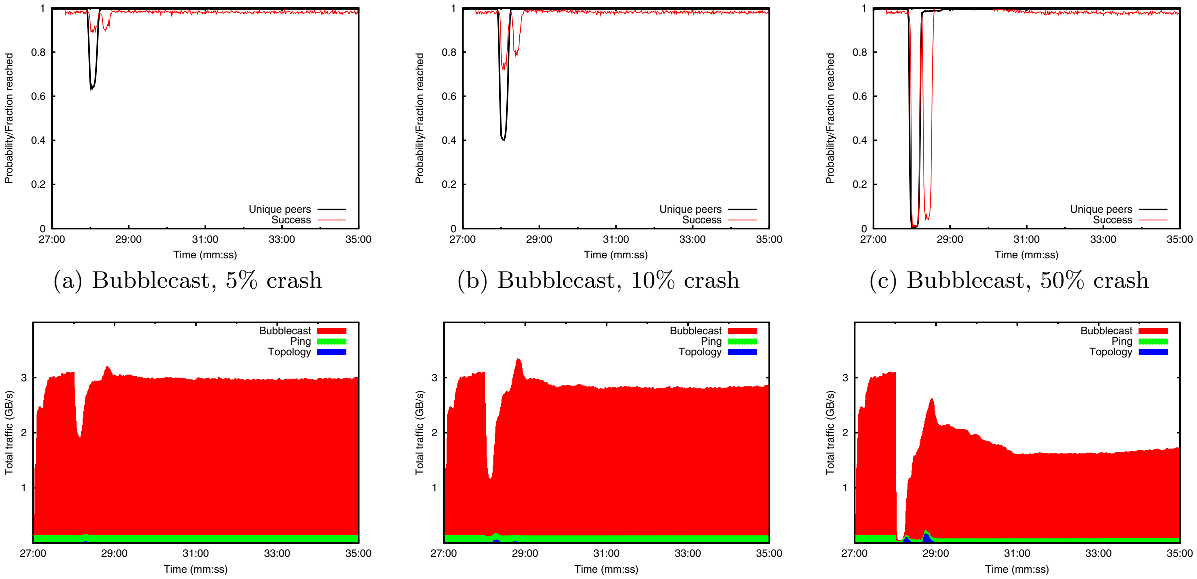
A simultaneous crash killing a large fraction of the network is most interesting (failure of an intercontinental line or a large Internet provider). Crashed peers cannot be detected without a timeout. Until the timeout expires, these peers act as bubblecast black holes, absorbing all packets, including weight > 1, causing massive correlated failure. Yet, amazingly, not only does BubbleStorm's topology recover within a minute from a 50% crash, but in-progress searches also continue to succeed during the 5% and 10% crash timeout window. Even for a 50% crash event, within 15 seconds the system has achieved *higher* success than before the crash.

To make sense of these results, we need to look at what a crash does. The dead peers act as open wounds, bleeding bubblecast packets out until the crash is detected and the wound closed. If there are enough such cuts, nearly all the bubblecast traffic bleeds out as shown for the 50% crash



(a) Bubblecast (b) Latencies (c) Measurements

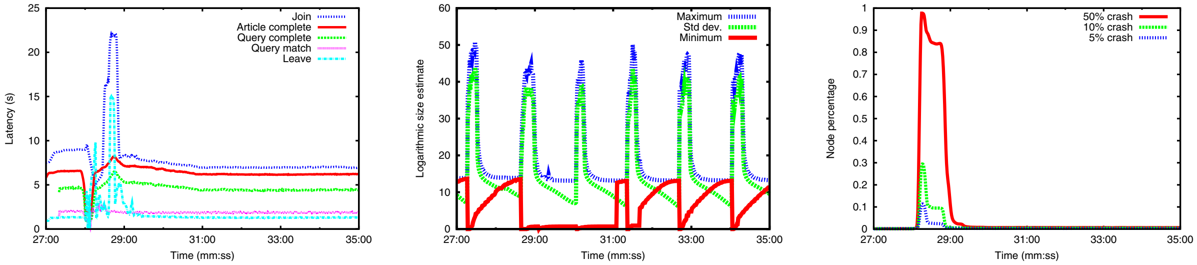
Figure 7: Churn with crashes



(a) Bubblecast, 5% crash (b) Bubblecast, 10% crash (c) Bubblecast, 50% crash

(d) Traffic breakdown, 5% crash (e) Traffic breakdown, 10% crash (f) Traffic breakdown, 50% crash

Figure 8: Massive crash: Success and traffic breakdown

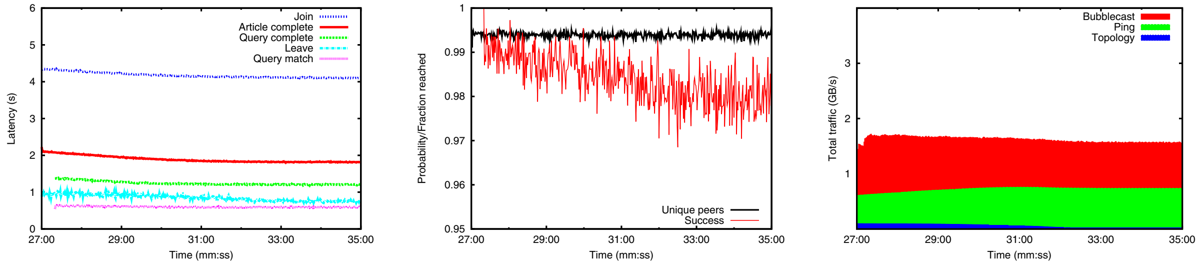


(a) Latencies, 50% crash (b) Measurements, 50% crash (c) Nodes with 8 edges or less

Figure 9: Massive crash: Latency, measurements, node degrees

traffic break-down in Figure 8(f). However, once the leaks are detected and the neighbour connections closed, bubblecast packets immediately stop being lost. Bubblecast does not need any particular degree sequence to work, so once the packet loss ceases, it operates as usual, but with *too large a bubble size*. This is why the success is immediately restored, but the congestion increases slightly (not shown). This inherent and automatic stabilization in BubbleStorm is the pillar of the system's robustness.

The topology can quickly recover its previous degree sequence because the congestion just fell due to bled out bubblecast messages. The corresponding latency drop in Figure 9(a) occurs at the critical moment when topology restoration begins. As the join operation is atomic and only commits to a joining neighbour on slave connection verification, no bubblecast traffic is lost during recovery. Therefore, only bubblecasts that were actually inflight during the 15 second window before crash detection suffer any effect from even the 50% crash. The spike in join latency comes from Split-



(a) Latencies (b) Bubblecast (c) Traffic breakdown

Figure 10: Heterogeneous network

Edges that bled out through unhealed cut links. At 29:00, a minute after the crash, the network degree sequence in Figure 9(c) has mostly healed, just in time to deal with the resurgence of bubblecast traffic. As soon as a new measurement stabilizes, the network is back to business as usual.

Random graphs like BubbleStorm's retain a giant component plus a small number of isolated components after large crash events [8]. For the run plotted in Figure 9(b) there were three pairs of nodes that were partitioned from the network. All the other nodes rejoin the network, but these stay isolated and report their local minimum.

Finally, let's examine the success probability. Depending on the proportion of bubblecast traffic bled out, we can determine the effect. At 50% crash, effectively all the traffic in Figure 8(f) is bled out and all inflight queries in Figure 8(c) fail. Shortly thereafter, the large query bubbles easily find matches. Twenty seconds later, the simulator looks for the articles created during the crash event, so the success drops again, but this is only an echo of the crash event which already healed. At 10% failure, about half the traffic in Figure 8(e) bleeds out. This results in a reduction of c by factor 2. Thus, $1 - e^{-(c/2)^2} \approx 0.63$ bounds the success of Figure 8(b) during the crash timeout window. Finally, for 5% crash, only 33% went missing from Figure 8(d), and so $1 - e^{-(c2/3)^2} \approx 0.83$ bounds Figure 8(a). Thus, a system which wants no degradation during a 5% crash would need to increase it's regular traffic by 50% to cancel the 33% loss.

5.5 Heterogeneous Network

The heterogeneous network scenario is the most realistic. BubbleStorm exploits heterogeneity to improve its performance, so we expect better results than Churn with Crashes.

Figure 10(c) shows a marked increase in keep-alive traffic. Peers take degree proportional to uplink bandwidth and ADSL-1000 peers have degree 10. Each connection requires 68 bytes (with TCP/IP headers) of keep-alive traffic every 5 seconds. As the network still contains a million peers, the higher average degree increases traffic.

The next difference is the bubblecast traffic reduction. BubbleStorm benefits from heterogeneity due to the match threshold equation. As the heterogeneity goes up, the match threshold goes down. For the homogeneous one million peer scenarios, $T = 1.25M$. However, in this scenario $T \approx 106.5k$, reducing the bubble sizes by about 3.4 fold. Figure 10(c) contrasted with 8(d) clearly demonstrates this.

High bandwidth peers in BubbleStorm do not reach their target degree immediately; this would be disruptive to the network. Instead, their degree increases linearly through time. Due to exponential warm-up growth, most nodes in the system joined recently (although they believe they are older). Thus, high bandwidth nodes must still reach their degree when the simulation starts, so topology traffic in Figure 10(c) is high. As the degree increases, the match threshold, T, *decreases*; BubbleStorm uses the heterogeneity. This can also be seen in the steadily decreasing bubblecast traffic. Since measurements of T lag behind the actual value, bubbles are slightly larger than necessary, resulting in an initially higher than expected success (Figure 10(b)).

Higher degree peers process more traffic. In fact, ADSL-1000 users in this scenario only need to process 288B/s, whereas the larger 10MB/s peers carry 23kB/s. When summed across all peers, this equals the 1.7GB/s traffic plotted.

Powerful peers have very positive effects on latency. They have lower link latency themselves and as they have most edges, most traffic crosses them. The combined effect yields successful search match times of 600 ± 200ms even though peers are uniformly distributed across the planet.

6. COMPARISON

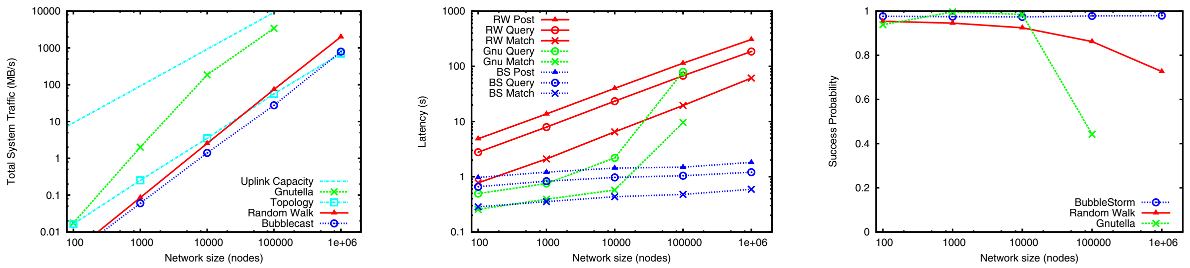
An apples to apples comparison of BubbleStorm to other systems is difficult. Structured approaches implement specific query algorithms and their performance depends on completely different parameters than unstructured systems. Unstructured systems like Gia [4] do not solve exhaustive search. Two systems solving nearly the same problem are Gnutella and the random walks of Ferreira et al [7].

Gnutella might be called the great grandfather of all unstructured systems, and it shows its age. We take it only as a base line for comparison. Gnutella searches by flooding queries with a hop counter and does not replicate data. With a large enough hop counter, it can search exhaustively.

The random walk system follows similar intuition to BubbleStorm. It replicates both queries and data to $c\sqrt{n}$ nodes in a (possibly) unstructured network. As the title suggests, random walks are used for replication, but these walks have biased probability to eliminate the effects of heterogeneity.

In our simulator, all systems must find the single matching document. We evaluate them on network sizes ranging from 10^2 to 10^6 peers, with system parameters from Section 4.1 and the heterogeneous bandwidth distribution from Table 3. All systems exclude clients; peers are run as super-nodes.

Our implementation of Gnutella uses a min degree of four. When joining, it is allowed to ask the simulator for running nodes chosen uniformly at random, improving its expansion when compared to real clients. It also asks the simulator for the network size, in order to compute a hop count of $\log_{(2\times4-1)} n$, which reaches nearly all nodes. Our implementation of Gnutella has no Ping/Pong or QueryHit traffic.



(a) Traffic (b) Latencies (c) Success

Figure 11: Comparison of Gnutella, Random Walks, and BubbleStorm

The random walk system omits a specific protocol, topology, and method for computing n. For fair comparison, we give it BubbleStorm's overlay and measurement protocol.

To evaluate the systems, we compare traffic, latency, and success averaged over 11 runs. As the x-axis is logarithmic in network size, the y-axis for latency and traffic are also logarithmic. The traffic plot includes the total bandwidth available to the systems, which grows linearly. BubbleStorm and the random walk system share topology, so topology traffic is split from the bubblecast and random walk traffic.

Gnutella's total traffic is known to grow as $\Theta(n^2)$. Thus, the logarithmic Figure 11(a) shows Gnutella with slope 2. Up to 10^4 nodes, Gnutella has good success and latency, but its bandwidth is quickly approaching the system limit. At 10^5 nodes, it requires more bandwidth than is available and saturates many peer connections. However, Gnutella does not fully exploit the available bandwidth; it uses only 35%. Without a split factor, flooding has poor load balancing. As the links are not utilized uniformly, congestion causes packet loss in advance of system saturation. This loss impacts success, which falls to 44%, and latency. Search completion takes 80s, although matches are found in 9.6s. Of minor note, the TTL is rounded down for 100 nodes, but up for other sizes, accounting for the lower success at this size.

The problem with the random walk system is that the walks are too long. A walk's latency (Figure 11(b)) grows linearly with its length, $\ell = c\sqrt{n}$. At 10^6 nodes, queries take 190s to complete and 66s to succeed compared to BubbleStorm's 1.4s and 0.6s. While the overlay only loses packets when nodes crash, the forwarding success, p, is 99.95%, for the simulated parameters. Once packet loss is included, random walks only reach $\frac{1-p^\ell}{1-p}$ nodes. In a 10^6 node network, this means 37% of replicas are never created, causing the success rate to fall as the network grows. Success is also lower, because random walks often step backwards, decreasing hops without creating a replica. Finally, as the random walks are biased to avoid heterogeneity and don't balance query/data replication, traffic is higher than for bubblecast.

Of the three systems, BubbleStorm is the most efficient, responsive, and reliable. Figure 11(a) shows that up to 10^6 nodes, the keep-alive traffic (constant per node) plus join/leave traffic (logarithmic, but smaller) dominates bubblecast traffic. Thanks to the split factor, link congestion is averted and latency is logarithmic in size. Up to the limits of our simulator, success seems stable as the network grows. Certainly, bubblecast traffic will eventually exceed the total uplink capacity, but it appears that much room remains.

7. RELATED WORK

The advent of DHT systems enabled efficient key-value peer-to-peer search. Yet, this mechanism alone is insufficient for many real-world problems. More sophisticated queries must be implemented atop DHTs, tailored to a specific task. Keyword search, in particular, has been studied extensively.

It has been shown that traditional inverted-indexes are not feasible for large scale keyword search [11]. Reynolds and Vahdat [13] use Bloom Filters to reduce the size of transferred lists while recursively intersecting the matching document set. Although being fairly bandwidth-efficient, the approach increases the latency experienced by the user. The Proof search system [20] reduces bandwidth further by applying pre-computed PageRank scores and content summaries to the index entries. The distributed computation of PageRank [14] adds additional cost to index maintenance.

Furthermore, Reynolds [13] and Yang [20] acknowledge that Zipf-distributed keyword popularity causes load balance problems in DHTs. This is an inherent issue for key-value indexes. Due to our completely different approach, keyword hotspots are avoided by BubbleStorm.

Gia [4] is an unstructured system combining biased random walks with one-hop data replication. While it delivers promising simulation results, it is not designed to return all matches in the system. Its topology uses several heuristics to put most nodes within short reach of high capacity nodes.

Sarshar et al [16] 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 installed along a random walk and then flooded with a probabilistic algorithm based on bond percolation. Traffic cost and success rate are analyzed. However, the only heterogeneity permitted corresponds to power-law graphs and introduces nodes of unrealistic degrees beyond $\sqrt{n}$.

While most 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; their latency is proportional to their length. Worse, any message lost in the walk stops its action completely. In contrast, bubblecasting offers latency logarithmic in bubble size and loss only slightly reduce that size, which does not usually make the search fail. While BubbleStorm does use random walks for join, these walks are short, time-insensitive, and resubmittable.

BubbleStorm has a join algorithm similar to that used in SWAN [2]. However, Bourassa et al are solving multicast, not exhaustive search. Cooper et al [5] proved that repeated application of their join and leave algorithms converges to a random regular graph in the usual sense. Compared to SWAN our joins and leaves are serializable, we don't need to repair crashed links, and we allow heterogeneous degree.

Yang et al [21] compared the performance of three different P2P keyword search systems: a Bloom Filter DHT-based system, flooding with super nodes, and random walks without data replication. The DHT performed best in terms of latency, but had worst publishing cost. Random walks were the opposite, also failing to provide full recall of matches. Their tests evaluated networks with 1000 peers and searches with two keywords on average. Our simulations feature networks 1000 times larger and 100 byte queries (more than two keywords). Documents in their scenario appear to be twice the size of our replicated meta-data. We conjecture that under the same conditions, BubbleStorm would outperform all three systems' search latency and compete with the DHT on bandwidth, especially in heterogeneous networks.

8. CONCLUSION

In this paper, we presented BubbleStorm, a probabilistic platform for exhaustive search based on random multigraphs. It is robust even under extreme conditions with poor resources. Our simulated network consisted of one million 10kB/s nodes. We subjected this network to simultaneous 90% node departure and 50% simultaneous crash failure. Yet, BubbleStorm survives because: a) unstructured networks do not have invariants to restore and can thus resume normal operation immediately after detecting crashes, and b) the system is self-stabilizing; when many nodes depart, search success improves immediately, no action required.

In addition, heterogeneity benefits BubbleStorm, leading to search latency of 600 ± 200ms—under the worst-case assumption of nodes distributed uniformly across the planet. This stems partly from parallelism, but mostly from trading bandwidth for latency. In contrast, most previous work takes the opposite approach, even though bandwidth is always improving and latency is approaching its physical limit. Nevertheless, when required to support all query languages, BubbleStorm's bandwidth utilization per node is asymptotically minimal (Section 3). It needs only 115kB to exhaustively search one million nodes with a 100B query.

We believe BubbleStorm is ideal for complex searches over static or versioned data. For example, file sharing systems could use it for searching file meta-data and wikis for full-text search. By leaving the query language to the application designer, we anticipate that more uses will be found.

9. ACKNOWLEDGEMENTS

We thank the Center for Scientific Computing of the University Frankfurt for contributing significant CPU time to our simulations. We would also like to thank Aditya Akella and all our reviewers for their helpful comments.

10. REFERENCES

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

[2] V. Bourassa and F. B. Holt. SWAN: Small-world wide area networks. In *SSGRR*, 2003.

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

[4] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making Gnutella-like P2P Systems Scalable. In *SIGCOMM*, pages 407–418, 2003.

[5] C. Cooper, M. Dyer, and C. Greenhill. Sampling regular graphs and a peer-to-peer network. In *SODA*, pages 980–988, 2005.

[6] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[7] R. A. Ferreira, M. K. Ramanathan, A. Awan, A. Grama, and S. Jagannathan Search with Probabilistic Guarantees in Unstructured Peer-to-Peer Networks. In *P2P*, pages 165–172, 2005.

[8] C. Greenhill, F. B. Holt, and N. Wormald. Expansion properties of a random regular graph after random vertex deletions. European Journal of Combinatorics, 2007 (to appear).

[9] 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 *SOSP*, pages 314–329, 2003.

[10] D. Kempe, A. Dobra, and J. Gehrke. Gossip-Based Computation of Aggregate Information. In *FOCS*, pages 482–491, 2003.

[11] J. Li, B. Loo, J. Hellerstein, F. Kaashoek, D. Karger, and R. Morris On the Feasibility of Peer-to-Peer Web Indexing and Search. In *IPTPS*, 2003.

[12] P. Maymounkov and D. Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *IPTPS*, pages 53–65, 2002.

[13] P. Reynolds and A. Vahdat. Efficient Peer-to-Peer Keyword Searching. In *Middleware*, pages 21–40, 2003.

[14] K. Sankaralingam, S. Sethumadhavan, and J. C. Browne. Distributed Pagerank for P2P Systems. In *HPDC*, pages 58–68, 2003.

[15] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *MMCN*, 2002.

[16] N. Sarshar, P. O. Boykin, and V. P. Roychowdhury. Percolation Search in Power Law Networks: Making Unstructured Peer-to-Peer Networks Scalable. In *P2P*, pages 2–9, 2004.

[17] W. W. Terpstra, C. Leng, and A. P. Buchmann. Brief Announcement: Practical Summation via Gossip. In *PODC*, 2007.

[18] W. W. Terpstra, C. Leng, and A. 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, 2007.

[19] Wikimedia Foundation. What we need the money for. http://wikimediafoundation.org/w/index.php ?title=What_we_need_the_money_for&oldid=18704.

[20] K.-H. Yang and J.-M. Ho. Proof: A DHT-based Peer-to-Peer Search Engine. In *Conference on Web Intelligence*, pages 702–708, 2006.

[21] Y. Yang, R. Dunlap, M. Rexroad, and B. F. Cooper. Performance of Full Text Search in Structured and Unstructured Peer-to-Peer Systems. In *INFOCOM*, 2006.