CREW: A Gossip-based Flash-Dissemination System

Mayur Deshpande, Bo Xing, Iosif Lazardis, Bijit Hore, Nalini Venkatasubramanian & Sharad Mehrotra
Donald Bren School of Information and Computer Sciences
University of California, Irvine
Email: {mayur,bxing,iosif,bhore,nalini,sharad}@ics.uci.edu

Abstract

In this paper, we explore a new form of dissemination that arises in distributed, mission-critical applications called Flash Dissemination. This involves the rapid dissemination of rich information to a large number of recipients in a very short period of time. A key characteristic of Flash Dissemination is its unpredictability (e.g., natural hazards), but when invoked it must harness all possible resources to ensure timely delivery of information. Additionally, it must scale to a large number of recipients and perform efficiently in highly heterogeneous (data, network) and failure prone environments. We investigate a peer-based approach based on the simple principle of transferring dissemination load to information receivers using foundations from broadcast networks, gossip theory and random networks. Gossip-based protocols are well known for being stateless, scalable and fault-tolerant; however, their performance degrades as content size increases, because of the propagation of redundant gossip messages. In this paper, we propose CREW (Concurrent Random Expanding Walkers), a smart gossip protocol designed to maximize the speed of dissemination by transmitting data only as needed, and by exploiting both intra and inter node concurrency. CREW is designed to support both content and network heterogeneity and deal with transmission failures without sacrificing dissemination speed. We implemented CREW on top of a scalable middleware environment that allows for deployment across several platforms and developed optimizations without compromising on the stateless nature of CREW. We evaluated CREW empirically and compared it to optimized implementations of popular gossip and peer-based systems. Our experiments show that CREW significantly outperforms both traditional gossip and current large content dissemination systems while sustaining its performance in the presence of network errors.

Keywords: Gossip Broadcast, Peer-to-Peer, Content Delivery, Fault Resilience, Autonomic Adaptation, Middleware

1 Introduction

Dissemination consists of the transmission of a data object from a source to a group of intended recipients. In this paper, we deal with a particularly useful (and often ignored) form of dissemination that arises in time-critical applications called *Flash Dissemination*. Such a scenario consists of a rapid dissemination of varying amounts of information to a large number of recipients in a very short period of time. We motivate flash dissemination through examples from the emergency management domain. Consider "Shake-Cast", a service from the Advanced National Seismic System[1] which aims to provide accurate and timely information about seismic events. Sensor data about the earthquake is collected in real-time and then processed to generate a "Shake-Map": this is a GIS file that can be 'layered' on a city map, for example, to assess which structures might be most affected. This information is sent to various subscribers, e.g., city, county and state emergency management organizations, for immediate assessment of the impact of the earthquake and to support triaging, co-ordination and resource allocation decisions. This information may also be regionally disseminated instantaneously to participating organizations including non-governmental agencies (Red Cross) and private entities (utility companies, hospitals, schools, etc.) hence resulting in a potentially large number (tens of thousands) of subscribers. Subscribers register a machine ahead of time to receive the information; such machines may use widely different networks (T1, DSL, Microwave, etc).

In such a setting, speedy delivery of information is critical because this will enable more informed and timely decision making resulting in better response. A flash dissemination scenario entails the following characteristics:

- *Unpredictability:* The dissemination events (e.g. disasters) are unpredictable, so the time when flash dissemination will be needed cannot be known. A flash dissemination system, must therefore be ready to work well at very short notice and cannot be scheduled or optimized in advance. Furthermore, the availability of infrastructure upon which to disseminate may be unpredictable. The disaster at hand may be responsible for many unpredictable faults, e.g., severing optic fibers, destroying hard disks, power outages, etc.

- *Scalability:* The number of end receivers may vary from thousands to hundreds of thousands depending upon the nature of flash dissemination and the receivers that must be contacted.

- *Network and Content Heterogeneity:* When end receivers are geographically distributed, heterogeneity in the network, in terms of latency, is natural. Added to this, different receivers may possess different capabilities in bandwidth leading to additional heterogeneity. Content Heterogeneity arises since rich information such as pictures, small voice/video

[1] http://www.anss.org/

clips, GIS files etc. range in size from hundreds of KB to a couple of MB.

A naive solution for the problem of flash dissemination would be to dedicate substantial resources (e.g., large network pipes and fast servers) on a continuous basis. Such a solution is not cost-effective because these resources will be wasted except in the infrequent and unpredictable event of a disaster. A more pragmatic solution can be achieved, if we recast the dissemination problem to a peer-based setting, where recipients participate in the dissemination process. The basic idea is to tap the resources of the end receivers and shift dissemination load to the set of clients organized as a large Peer-To-Peer (P2P) dissemination system.

Dissemination systems today are tailored to two ends of a spectrum: dissemination of small data (events) and dissemination of large (possibly streaming) content. For small data, the focus is usually on low-latency delivery of data in the range of tens of kilobytes; for example, delivery of stock prices or updates in a multiplayer online game [18]. It is not obvious how these systems would scale with data size because they don't exploit high-bandwidth nodes: conversely, such nodes are exploited in large content delivery systems, which can thus sustain high throughput, and are able to deliver content of the order of hundreds of MB to GB. We explore the latter systems in more detail in Sec-2. Again, it is not entirely obvious if large content delivery systems can achieve very fast dissemination for medium amounts of data. Additionally, these systems are designed assuming 'normal' network and host behavior, e.g., a constant rate of faults ('churn' rate [22, 26]).

On the other end of the spectrum, gossip-based broadcast systems are designed to accommodate unpredictable faults. But most gossip systems do not usually take into account variation in node bandwidths. For small amounts of data, this is usually not much of a concern. However, for medium and large content, the overhead due to the redundant messages makes traditional gossip based approaches considerably slower.

Our goal in building CREW (for Concurrent Random Expanding Walkers) is to take the best of both worlds – fast dissemination of content over heterogeneous networks *and* under unpredictable conditions. CREW is a new, fully decentralized, gossip-based protocol, designed from the ground up, with a focus on reducing the data overhead and on increasing both system wide and within node concurrency. We implemented CREW using a scalable middleware platform and added optimizations without compromising its stateless nature. Increased concurrency and reduced overhead allows CREW to disseminate data extremely fast and to scale in terms of both network and content size. Additionally, CREW adapts to network heterogeneity while degrading gracefully in the presence of heterogeneous packet losses.

The primary contributions of this paper are:

1. Design, implementation and evaluation of CREW, a decentralized, stateless, gossip-based protocol for fast dissemina-

tion of rich information. CREW is almost twice as fast as current, optimized dissemination systems (such as BitTorrent and Bullet) for flash dissemination and imposes around 400% less data overhead than traditional gossip.

2. A thorough and systematic evaluation of CREW as well as various dissemination systems demonstrating the effectiveness of CREW for flash dissemination.

3. A gossip protocol that has a deterministic termination property and autonomically adjusts to fault rates at runtime.

4. A new approach to gossip sampling service using random walks on overlays. This approach reduces data overhead of gossip messages and provides for near real-time view management.

The rest of the paper is as follows. In Sec-2 we outline the rationale for the CREW protocol. In Sec-3 we describe the full CREW protocol. Implementation of CREW is described in Sec-4 and we analyze its real world performance in Sec-5. Finally, we conclude in Sec-6

2 Rationale for CREW

At an abstract level, flash dissemination is the canonical broadcast problem in networks – how to distribute data, split into M chunks, from one source to N other receivers, as fast as possible. This problem was originally studied in the context of routers [11] and has been revisited in the P2P context [12]. The optimal solution to the problem is composed of two main steps: (1) Getting one chunk to each peer as fast as possible and then, (2) Optimally partitioning the peers into equal sets of givers and receivers, until all peers have all chunks. We name the first phase as the *Ramp-up Phase* and the second phase as the *Sustained Throughput Phase*. The optimal number of time-steps for dissemination of M chunks to N nodes is $Log(N) + 2M - 1$ [11, 12].

The optimal solution assumes homogeneous network bandwidth and latency between nodes. Introducing heterogeneity immediately makes the problem NP-hard [16]. Different heuristics and data structures have been used to tackle heterogeneous bandwidth nodes, along with special P2P overlays explicitly designed for dealing with heterogeneity [29]. A bandwidth optimized tree seems a logical data structure to use; high-bandwidth nodes are placed near the root and support many children nodes. A forest based approach extends this idea even further so that leaf nodes can not only be receivers but also data givers. Splitstream [10] is one such sophisticated approach that uses a Distributed Hash Table (DHT) overlay to construct a bandwidth optimal forest. However, both trees and forests suffer from scalability problems in the presence of faults.

Mesh based approaches relax the rigid overlay structure and provide for better fault tolerance by constructing multiple paths between nodes. Additionally, mesh-based approaches seem to provide higher throughput than tree based approaches [20, 30, 24]. BitTorrent [1], the current defacto protocol for distributing large content is also a mesh-based system. The popular nature of BitTorrent has led many researchers to examine it from a more theoretical perspective [7, 26] and show why BitTorrent works so well in practice in distributing large content. Mesh-based systems, however, still maintain some 'state'. In Bullet, a tree is used to disseminate control data. In BitTorrent, each node maintains state about its current neighbors: the chunks that are in the pipeline for download and upload. Additionally, these systems are tailored for delivering large content to a large number of receivers. When the content is large, the sustained throughput phase clearly overshadows the ramp-up phase. When content to be disseminated is relatively small (hundreds of KBs to couple of MBs) and the number of receivers still large, then the ramp up phase contributes significantly to the total dissemination time. Additionally, delivering large content can easily extend into hours, providing ample time for system reconfiguration and fine-tuning. By contrast, in the case of flash dissemination, the time available for exploiting high-bandwidth nodes is significantly less.

During disasters systems and networks become unstable and unpredictable; therefore, a primary objective is to achieve dissemination in less-than-perfect network conditions. Gossip [15] based broadcast protocols are an almost perfect fit for this scenario. They trade redundancy for scalability and simplicity. Gossip-based broadcast is extremely effective and scalable for various dissemination scenarios. However, they suffer from various deficiencies when applied to flash dissemination. We revisit a well known gossip based broadcast protocol and examine these deficiencies in more detail.

Gossip Based Broadcast Revisited

In gossip based broadcast, every node that receives a message, buffers it, and then forwards it (i.e. gossips it) a certain number of times, each time to a randomly selected subset of processes [27]. In effect, most gossip protocols, for e.g. lpbcast [25], are implemented as 'push-based' mechanisms. The number of processes to push to is also usually fixed and is known as *fanout*. A fanout of 3-5 is usually sufficient, with normal Internet packet loss rate, to guarantee reliable delivery of a message to all nodes in the system. Selecting random nodes (also known as the *sampling service*[14]) to push to, is a core challenge. Each node selects random nodes from its current 'view' (a local cache of addresses of other nodes). In lpbcast, a node maintains a 'partial view' of the system which consists of addresses of L other random nodes. This view is then attached and forwarded in each gossip message. A receiving node, then combines its own view with the view contained in the gossip message. To keep view size scalable various intelligent policies are used to truncate the view back to a constant

size [25]. Views are forwarded in each message so that information about nodes that join/leave the system is updated, over time, in all nodes. lpbbast is an elegant and simple gossip-based broadcast system that achieves scalability and fast delivery of events: for N nodes in a system, it takes $Log(N)$ time to deliver the message to all nodes, with high probability (w.h.p.) of events. However, two factors impede the use of lpbcast for fast delivery of medium and large size content: high data overhead leading to slow dissemination and lack of adaptation to heterogeneity.

Data Overhead and Dissemination Time:

The fanout of gossip directly impacts the dissemination time, detrimentally. Consider content of a certain size to be disseminated among N nodes. Let it take M unit of time to send the content fully between two nodes using their full bandwidth (Assume all nodes have equal bandwidth). Sending content as one gossip message would then require $O(M * Log(N))$ time. Ideally, we would like the time bound to be $O(M + LogN)$. If we split the content into M chunks, and if the fanout is K, it can be shown that the total time to disseminate *cannot be lower than* $K(M + Log_K(N))$ (proof omitted). In analysis of most gossip protocols, it is implicitly assumed that the time to send many events is the same as the time to send one event. While this is true if the events are small, this assumption does not hold when the chunks are large and the bandwidths of nodes dictate the time of delivery. The fanout, therefore, directly affects the best case completion time, with a larger fanout invariably leading to a longer completion time. Apart from the overhead due to sending redundant messages, each message also carries the 'view', adding further to the overhead and increasing the dissemination time. The longer dissemination time due to the large overhead was also shown empirically in [20]. In [31], the authors examine how the impact of fanout can be reduced by adapting each node's fanout dynamically. However, the focus on their work was on reducing the latency of delivery of a message and did not tackle the issue of multiple messages or node bandwidths explicitly.

Lack of Adaptation to Heterogeneity:

Fixed fanout in nodes makes gossip not easily adaptable to heterogeneity in bandwidth. For example, how to assign more data transfer work to high-bandwidth nodes and how to prevent low bandwidth ones from getting overwhelmed. Additionally, if gossip is implemented over UDP, as is natural to do, congestion at the network layer must also be taken into consideration [9]. In [27], the authors examine the issue of adaptation and congestion control. However, they adapt to congestion at the level of application level buffers and not directly at the network level. Estimating and exploiting bandwidth is a more involved problem as we describe in Sec-3.2.2. Another drawback of having fixed fanout is lack of adaptation to dynamic fault rates. During flash dissemination, it is hard to predict beforehand the network conditions and fault rates. While a fanout of 3-5

works quite well for normal packet loss rate (1-2%), a higher fanout may be needed when faults are high. However, a fixed fanout introduces an unappealing tradeoff – too large a fanout would imply extremely slow dissemination whereas a small fanout would lead to some node not getting the information at all.

3 The CREW Protocol

Our goal is to maintain the inherent stateless, scalable and fault-resilient properties of gossip while achieving (1) fast dissemination (2) over heterogeneous networks. We employ two main techniques to make CREW fast, namely, reducing redundant data and providing low-overhead concurrency in the system. To tackle heterogeneity, we introduce concurrency within a node and adapt it to local and global bandwidth availability. This helps keep nodes "busy" at their optimum bandwidth usage; thereby making dissemination even faster. We begin by describing the techniques to support fast gossip in CREW, followed by introducing the extensions to support heterogeneity.

3.1 Basic CREW Protocol: Making Gossip Fast

As noted in Sec-2, the basic bottleneck of gossip for fast dissemination is the high data overhead which leads to decreased throughput and slow dissemination time. The overhead for gossip arises from the redundant gossip messages. To tackle redundant messages, we use a metadata-based pull mechanism to give nodes "content awareness". Nodes use the metadata to pull only messages that they do not have. Further the metadata is broadcast as fast as possible, so that all nodes can be "up and pulling" in the shortest time, leading to very high concurrency. We also provide a low overhead mechanism for concurrency, based on random walks on overlays. This low overhead mechanism is essential to prevent the concurrency from initially congesting the system. Decentralized construction of good overlays is a challenge in itself. We address this by designing *Bounce*, a protocol that can efficiently construct overlays with good properties, requiring no state maintenance at any node.

3.1.1 Reducing Redundant Messages

First, we introduce the concept of "content awareness". The original content is divided into multiple chunks and each chunk is assigned a unique chunk-id[2]. The list of all chunk-ids is termed as metadata[3]. Metadata information about the chunks (and their ids) are known by all nodes before they start gossiping (we will describe how this is achieved shortly). Next, we invert the "fanout push" logic of traditional gossip into a "pull-based" mechanism. A pull-initiator node sends out the list of

[2] A discussion on optimal chunk size can be found in [11]

[3] similar in concept to a ".torrent" file in BitTorrent that has the metadata for the actual file

the ids of the chunks that it has already received to a target node, selected uniformly at random. The target node then sends, one chunk at random, that the initiator does not have. If the target node has no "missing" chunks, it sends an error message. Thus, *nodes never pull duplicate chunks*. This basic protocol is described in Fig-1. Once a node receives all chunks that are listed in the metadata, it immediately stops gossiping. Thus, CREW has a *deterministic termination-delivery property* – when all nodes terminate (stop gossiping), all nodes have all chunks. This is unlike push-based gossip that guarantees only probabilistic delivery at termination.

The analogy for traditional gossip is that of 'rumor spreading', wherein a person randomly calls another person and tells them about the rumor (message). We can extend this analogy to that of content-aware pull gossip. Imagine a 'large' rumor or news item. Initially, everyone knows that there is some rumor going around (metadata) but doesn't know everything about it. When a person hears that the rumor is going around, she tries to collect all information about the rumor. She calls another person at random and tells him all that she knows at the current moment. The other person then tells her some part of the rumor that she is missing (if he knows something extra). Over time, everyone gets the 'full picture'.

Metadata is initially broadcast on the overlay. When a node gets the metadata for the first time, it forwards the metadata to all its neighbors, except the one from which it was received. This ensures that all nodes get the metadata. A natural question is what happens if a node fails to get the metadata in the first place? This can happen if the network of a node goes down at the point of broadcast and comes back up again later. In this case, if the node reconnects back into the overlay, then it will hear some pull messages from other nodes. Pull messages contain a metadata identifier. If a node sees an unknown metadata identifier, it can then, explicitly pull for this metadata, before pulling for the chunks.

3.1.2 Enabling Low Overhead Concurrency

Metadata is small in size and is received by all nodes very fast. Thus, in a very short time all nodes are active and trying to pull chunks aggressively. Since, initially, very few nodes have chunks to give, most nodes receive error messages in the pulls and immediately seek other nodes to contact. To support this high level of concurrency at a low cost, the pull messages must be as small as possible. The list of received-ids is close to zero, so this is not too much of an overhead. However, if we employ traditional gossip mechanism of sending a node's view in each message, each gossip message unnecessarily increases in size. We therefore designed a new approach for implementing view maintenance and sampling service. Our sampling service is based on the theory of random walks on overlays. [13] showed that the nodes visited during a random walk of X steps on an expander network, is an approximation of a random subset of size X (with a larger X leading to a better approximation).

INITIALIZE:

$RecvdChunksIds \leftarrow \{\emptyset\}$

$RecvdChunks \leftarrow \{\emptyset\}$

$ChunksToGet \leftarrow \{c_1.id, c_2.id, ...c_M.id\}$

BEGIN

1) **While** $|ChunksToGet| > 0$
2) Node $X \leftarrow$ get next random node
3) Chunk $ck \leftarrow$ RPC[a] $(X, GossipPull, RecvdChunksIds)$
4) $RecvdChunks \leftarrow RecvdChunks \cup ck$
5) $RecvdChunksIds \leftarrow RecvdChunksIds \cup ck.id$
6) $ChunksToGet \leftarrow ChunksToGet - ck.id$

END

Figure 1. Basic CREW Protocol

[a] A generic Remote Procedure Call of type (RemoteNode, RemoteMethod, Parameters)

BEGIN

1) **While** $|ChunksToGet| > 0$
2) **While** Spare bandwidth exists
3) Node $X \leftarrow$ get next random node
4) **Do Concurrently With Main Thread:**
5) ChunkId $id \leftarrow$ RPC$(X, IntentToPull, RecvdChunksIds)$
6) **Acquire Mutex Lock**
7) **If** $(id \in RecvdChunksIds)$
8) **Release Mutex Lock**
9) **Else**
10) $RecvdChunksIds \leftarrow RecvdChunksIds \cup id$
11) $ChunksToGet \leftarrow ChunksToGet - id$
12) **Release Mutex Lock**
13) Chunk $ck \leftarrow$ RPC $(X, GetChunk, id)$
14) $RecvdChunks \leftarrow RecvdChunks \cup ck$

END

Figure 2. CREW Protocol Loop for heterogeneous Networks

Finding the next random node to gossip with, can now be as trivial as getting the next random node in a random walk. The overhead for each gossip is now one extra node address. The target node returns the address of one of its random neighbor, in the pull reply message. Thus, the overhead is one instead of "view size" for each gossip message. In CREW, we maintain an explicit overlay among the nodes (using open TCP connections) for doing the random walk. Next, we describe how we construct and maintain this overlay in a decentralized stateless manner.

3.1.3 Bounce: Low Cost Overlay

An ideal overlay for random walks is an expander overlay ([21, 13]). However, in these overlays, a node must keep explicit state information about its neighbors (for e.g., which Hamiltonian cycle [21] the neighbor belongs to and whether it is a predecessor or successor). This runs counter-intuitive to the stateless nature of gossip. Therefore, we designed a new decentralized protocol, Bounce, that creates a sparse, low-cluster [17] and small-diameter overlay. Though we have not analyzed the theoretical properties of this overlay yet, experimental results show that CREW performance using this overlay is comparable to that of using an ideal sampling service[4]. Additionally, Bounce requires only small number of message exchanges to add a new node and the average number of messages required to build the Bounce overlay grows sub-linearly with the network size.

In Bounce, to acquire a new neighbor, a node does a random walk on the overlay asking nodes if they will accept a new

[4] which returns a node, uniformly at random, from the set of all nodes in the system

```
INITIALIZE:
    CurrNode ← random neighbor
    Tries ← 0
BEGIN
1)  While (TRUE)
2)    Boolean accept ← RPC (CurrNode, AcceptAsNeigh, Tries)
3)    If (accept)
4)      Break Loop
5)    CurrNode ← RPC (CurrNode, GetRandomNeighbor)
6)    ++Tries
END
```

Figure 3. Bounce Protocol

```
AcceptAsNeigh(Tries)
Output: Boolean: Node accepted as neighbor or not
BEGIN
1)  ProbAccept = BounceFormula (self.degree, Tries)
2)  dice = Rand (0, 1)
3)  If ProbAccept > dice
4)    return True
5)  return False
END
```

Figure 4. Accept Logic of Bounce

neighbor (the walking node). If yes, a new neighbor link is formed between the walking node and the acceptor node. Else, the walking node continues with its random walk until some other node accepts it as its neighbor. The walking node keeps track of how many times it has been 'rejected' so far. This parameter (called *Tries*) is used by acceptor nodes (along with their own degree) to decide whether to accept the walking node or not. The crux of the problem here is to prevent some nodes from acquiring too many neighbors while also keeping the number of retries (rejections) low. The exact protocol is shown in Fig-3 and Fig-4. The "Bounce Formula" of the accept method that we use is:

$$ProbAccept = \frac{1}{self.degree} + rand(0,1) * Log(Tries)$$

The probability of acceptance is therefore directly proportional to the number of rejects and inversely proportional to a node's degree. Simulations showed that for constructing a 10,000-node network, with each node having at least 4 neighbor links, a node required, on average, 1.664 messages to acquire a neighbor link. The maximum number of messages needed by a node to acquire 4 neighbors was 32. The clustering coefficient of the network (as defined in [17]) was 0.144 and node degree was approximately normally distributed.

3.2 Extending CREW for Heterogeneous Networks

Wide area networks are seldom homogeneous. There is varying latency and nodes have varying bandwidths, sometimes in the order of magnitudes. For example, inter-node latency can vary between 2 - 700 milliseconds and bandwidth can vary from 64Kbps to 10Mbps. This raises both challenges and opportunities. In particular, (1) How to reduce the detrimental effects of high latency? (2) How can high bandwidth nodes be exploited? and (3) How to adapt high bandwidth nodes, at runtime, from overwhelming (and congesting) low bandwidth nodes? We explore these questions and propose additions to

the basic CREW protocol to tackle these issues.

3.2.1 Latency Amortization

In the basic CREW protocol, a node waits for the current pull to finish before starting on the next one. When a node initiates a pull message to another random node, it must wait at least for Round Trip Time (RTT), between the two nodes, before hearing back any reply (error or chunk reception). If the RTT between two nodes is 500ms, for example, then nothing useful happens for almost half a second. During this time, a node "wastes" its bandwidth entirely. If the reply was an error message, the node has to start again. Moreover, to preserve the gossip-nature of CREW, there is no straightforward way to amortize this long setup time – a node moves away to another random node after a pull. In other protocols (such as BitTorrent, Bullet, SplitStream, etc.), connections once open, are used to transfer multiple chunks. Changing CREW to do multiple transfers with one node would be against the basic gossip model. This, therefore, seems like a fundamental clash between theory and practice – sticking to pull-based gossip would make CREW extremely slow in any network where nodes had large latencies.

However, high latency cost can be amortized in another way – not by transacting multiple chunks with a node, but by transacting a single chunk with multiple nodes, concurrently. We call this the **concurrent pull optimization**. CREW protocol enhanced to deal with concurrency is shown in Fig-2. Doing concurrent pulls naively, however, may result in a node receiving duplicate chunks. For example, two gossip pulls initiated by concurrent pulls at the same time, may download the same chunk. To prevent this, we split the gossip step into two phases. In the first phase an "intent to pull" message is sent to the target node (Fig-2 Line 5). The target node replies with the chunk-id of the chunk, which it would have actually given back had this been basic CREW. The received id is then compared to check if some other concurrent pull is already trying to get this chunk. If not, the chunk is really pulled in the second phase (Fig-2 Lines 7-14).

Since nodes are contacted at random, some of the contacted nodes have low latency while others may have high latency. Chunk transfers from lower-latency nodes can overlap with the setup to higher-latency nodes – thereby masking the setup cost. The problem is then deciding what would be a good concurrency factor. Too low a factor might result in under-utilized bandwidth and too high a factor results in bandwidth being unnecessarily split across many transactions, thus delaying all the transactions and increasing the dissemination time. Additionally, we would like the concurrency factor to be autonomic and dynamically adaptive at runtime. To achieve this, a node keeps track of its "spare bandwidth". Whenever spare bandwidth exists, a node immediately starts a new pull (Fig-2, line-2).

3.2.2 Bandwidth Estimation

Estimating the spare bandwidth of a node is not trivial, with the very notion of bandwidth being tricky to define precisely. Specially, in the 1-to-many case, what is the maximum bandwidth of a node, say A, with respect to a target set of nodes, N? For example, a node may have 100Mbps bandwidth to its local area router but only 200Kbps bandwidth to another faraway router. If the target set of nodes all fall behind the faraway router, then the node's maximum bandwidth (w.r.t. to target set) can only be less than 200Kbps. If we extend this example to multiple nodes being behind multiple routers, then estimation of maximum bandwidth becomes a combinatorial problem. The maximum bandwidth of a node, A, can then be defined as the maximum throughput achieved by communicating simultaneously with some subset of nodes in N. A way to do this would be to compute all possible subsets of N and then test which subset gives the maximum bandwidth. This is an NP-hard problem.

Current systems, like BitTorrent and $Bullet'$ use heuristics to calculate a node's maximum bandwidth. For example, in BitTorrent, a node connects at random, to a subset of the target nodes and measures the bandwidth. At regular intervals, a new node is chosen, usually at random, and a connection is opened to it. If the bandwidth increases, then one of the old connections which had the least bandwidth is dropped. In general, the idea is for each node to slowly calculate and evolve towards the subset of nodes that give it its maximum bandwidth. When the content to be disseminated is large, there is significant time for nodes to stabilize and maximize their bandwidth utilization. In flash dissemination, the content is usually small and hence there is relatively little time to evolve to maximum utilization. The gossip nature of CREW, however, allows us to leverage the fast moving connection setup to estimate maximum bandwidth rapidly.

In CREW, each node starts with an initial value of zero as its maximum bandwidth. Two pull connections are allowed to progress concurrently at any time, irrespective of spare bandwidth. The initial maximum bandwidth is estimated from the first two pulls. After this, with every new connection that is opened (for either initiating a pull or transferring a chunk), the maximum bandwidth estimate is updated, if current bandwidth utilization exceeds the current maximum bandwidth estimate. This simple scheme is highly effective in estimating the maximum bandwidth of a node rapidly. Once maximum bandwidth is calculated, calculating of spare bandwidth is straightforward. Nodes also use the estimate of maximum bandwidth to also decide whether to allow other peers to download chunks from them. If a peer is using up all its bandwidth, then it'll return an error message for all pull requests. This is used by the puller node to estimate global congestion as we explain next.

3.2.3 Congestion Adaptation

If a low bandwidth node is already at its peak bandwidth utilization, then it rejects any new pull requests, irrespective of whether it has missing chunks or not. In the pathological case where most nodes have no spare bandwidth, we would like nodes with spare capacity not to contact these "busy" nodes. If nodes with spare bandwidth try to do pulls, they end up generating redundant data (in the form of pull requests) and slowing down the dissemination process. The gossip nature of CREW, however, allows us to elegantly tackle this problem. When a node makes a pull, the target node estimates if it has spare bandwidth. If not, it replies back with a special error message, saying that it is "busy". If the initiator hears many such "busy" messages in a short period of time, then it can be fairly certain that most nodes are near capacity (and can then take appropriate action like backing off). This is due to the uniform random property of gossip. The replies from the target nodes are representative of the replies of a random sample from the total population. Thus, if most nodes in the random sample are busy, then most nodes in the total population will also be busy. More generally, the reply message from the target node may contain any local state and the initiator can quickly glean global state information from these individual replies. *Pull replies are therefore, a powerful mechanism that can be used to estimate global properties about the system*.

3.3 Fault Tolerance in CREW

Changing push-based gossip to a metadata-based pull model offers many benefits from a fault tolerance point of view, but it also introduces a new challenge. We describe the benefits first followed by the challenge.

The pull logic of CREW completely eliminates the need for deciding optimum fanout. A node does as many pulls as necessary to get all chunks. If faults occur when it is pulling, it just pulls more number times. This simple mechanism therefore leads to an elegant, autonomic fault-tolerance property – *depending upon the fault rate, nodes do less or more pulls, automatically*. The simplicity of this property is hard to overstate.

CREW also benefits from a near real-time view management property. The "view" of a node in CREW, is its list of neighbors. If a node dies, its neighbors remove it from their neighbor-list, and do not forward any random walks to it. Thus, the dead node *vanishes from all nodes' view immediately*. Thus, using random walks in overlay for view management allows for near real-time updates to views of all nodes. Additionally, nodes do not spend resources trying to contact dead nodes and this in turn speeds up the dissemination process.

Content-aware pulling in CREW introduces a fault tolerance challenge that is absent in push-based gossip. The list of chunk-ids that a node sends to the target pull node may get lost, in which case the target node will never reply back.

Additionally, if chunks are sent as smaller data packets, then, even if one data packet is lost, the entire chunk is "corrupted". When packet loss rate increases, the performance of CREW can degrade exponentially fast. This challenge can be addressed by using an underlying transport protocol that does packet loss detection and recovery. Thus, we use TCP as the underlying transport for all inter-node communication in CREW. TCP provides an efficient ACK-based recovery and retransmit protocol. Using a ACK-based retransmit approach (such as TCP) instead of a naive fire-and-forget policy, allows CREW's performance to degrade linearly, instead of exponentially, as a function of packet loss rate[23]. Thus, using TCP not only alleviates this weakness, but also provides other important benefits – such as automatic congestion control at the network level. Using TCP, however, introduces other challenges such as higher setup cost (due to 3-way handshake) and dealing with slow-starts. These are addressed by the concurrency extensions (as described in Sec-3.2) and the optimizations in CREW implementation (Sec-4).

4 CREW: Implementation

Our goal was to design and implement CREW so that it would perform well in real world heterogeneous networks. The design and implementation was an iterative process with valuable insights provided by the Modelnet testbed (we describe the testbed setup in Sec-5). In building the actual system, our overriding philosophy was to make the system as modular and easy to maintain as possible. Rather than develop it from scratch, we choose an Object-based middleware, ICE [3], as our fundamental software platform. As we'll describe, this choice considerably eased and simplified our implementation. Additionally, developing CREW using ICE allows us to leverage all the benefits of a cross-platform middleware platform. We have Java and C++ versions of CREW running on Windows XP, Linux and FreeBSD.

CREW is implemented as a set of interacting modules, as shown in Fig-5. We provide a brief overview of these modules and then describe them in detail. The actual CREW protocol is executed by the Pull/Push threads. A Pull/Push thread uses various supporting modules. The Bandwidth Manager calculates and estimates spare bandwidth on a node and the Pull thread uses this to figure out if it should do more concurrent pulls. The Random Walker is responsible for traversing the overlay and collecting random nodes to gossip with. The Random Walker is in turn dependent upon the Neighbor Manager which makes sure that a node is always connected into the overlay. We now describe the modules in greater detail.

- *Pull Manager:* The Pull manager is initialized as soon as Metadata is received through a neighbor and remains alive until all chunks (for particular content) are collected. Depending upon spare bandwidth (information that is got from the Bandwidth Manager), the pull manager initiates gossip pulls. Concurrent pulls can be naturally handled in their own

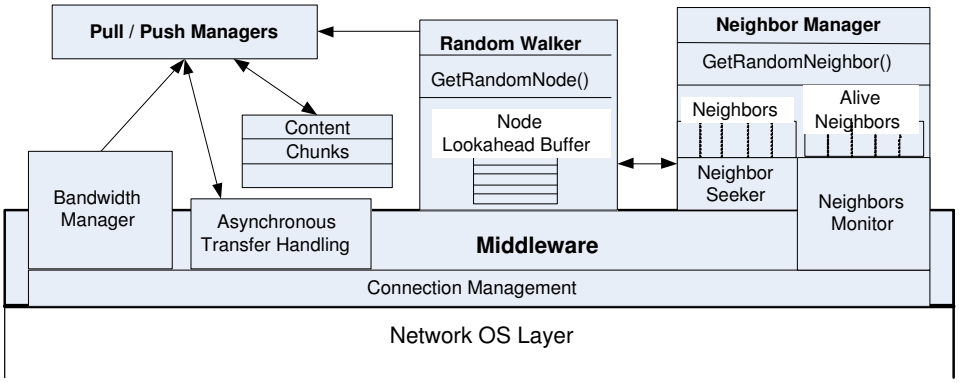


Figure 5. CREW System: Main Modules

separate thread but ICE allows for a more efficient mechanism called Asynchronous Method Invocation (AMI)[8]. AMI allows the application to register a callback method with a Remote Procedure Call (RPC). When the RPC is complete, the middleware 'calls back' the registered method. In our case, we execute every gossip as an AMI call and register a 'chunk handler' with it. When the chunk handler is called back, we check to see if we received a chunk in that particular gossip, and if so, do appropriate processing on it. The advantage with AMI is that a single application thread can initiate multiple concurrent RPCs. Further, the ICE subsystem handles the concurrent AMI calls efficiently using a leader-followers[28] thread pool based on the select I/O system call.

- *Push Manager:* Similar to the pull manager, we also developed a Push Manager in which a node pushes chunks to other nodes (again after executing a handshake to make sure that duplicate chunks are not transferred). In networks where high-bandwidth nodes exist, we would expect these nodes to complete faster than low bandwidth nodes. To exploit these nodes, we introduced the 'push' mechanism in CREW. After a node finishes pulling all chunks, it switches to a Push mode. In this mode, a node randomly visits other nodes and enquires if the target node needs any chunk. If so, a chunk is pushed to the target node. A problem with the push protocol, as compared to the pull protocol, is the termination criterion – when does a node decide to stop pushing? We implement termination in a probabilistic way. The probability that a node will stop pushing is directly proportional to the number of consecutive 'rejects' that a node encounters. A reject is an error in the handshake which indicates that the node visited does not need and chunks from the visiting node. In our implementation, we use the following formula to calculate the probability of stopping: $Prob(stop) = 1 - \frac{1}{rejects}, rejects \geq 1$. Usually, a push walk stops after it hears four consecutive rejects. $rejects$ is reset to 1 as soon as a valid chunk-id is received in the push random walk.

- *Bandwidth Manager:* ICE provides facilities for statistic gathering, one of which includes the bytes transferred in/out through the middleware. We use this to calculate the current bandwidth usage, on every epoch of 300 milliseconds. Maximum and spare bandwidth calculation is done as described in Sec-3.2.2.

- *Random Walker:* By implementing Random Walker as a separate module, we abstracted out the sampling service functionality. This allowed us to make an interesting optimization that is fully transparent to the pull thread. The Random Walker visits a certain number of nodes *ahead of time* and maintains open connections to them in a data structure called the *Node Lookahead Buffer* (NLB). When the pull thread asks for the next random node, the Random Walker returns one open connection from the NLB (and removes it from the NLB). Having a connection already open saves on TCP connection setup time. While the pull thread is busy setting up gossip pulls, the Random Walker is concurrently preopening connections. Connections are opened until a high 'water mark' is hit and the Random Walker is then stopped. When the NLB size falls below a 'low water mark', the Random Walker is restarted. The Random walker is initially started from a random neighbor of the node. During a random walk on the overlay, if there are any network failures or timeouts, the Random Walker resets back to a random neighbor and continues. Connection management is crucial for CREW since many connections are opened and 'discarded' (not needed) rapidly. Here again, the Automatic Connection Management (ACM) feature of the middleware comes in handy. ACM can be thought of as a garbage collector for socket connections. If there is no traffic (in/out) on a socket for a certain period of time, the middleware automatically closes the connection, freeing up OS resources. Thus, CREW does not need to worry about managing socket connections explicitly.

- *Neighbor Manager:* The Neighbor Manager is primarily responsible for neighbor fault detection and recovery. The Neighbor Manager periodically (1 second) 'pings' one neighbor selected at random from the neighbor list. In CREW, each node maintains a neighbor list of size five. If a neighbor fails to respond to the ping, and if the number of neighbors are below five, then the Neighbor Manager initiates the Bounce protocol. The Bounce protocol itself involves a random walk on the overlay. The Neighbor Manager thus uses the Random Walker to get random nodes to try and acquire as neighbors. The open connections in the NLB further speeds up the tryouts and the recovery phase.

5 Performance Evaluation

In this section, we quantitatively analyze the performance of CREW and compare it to other systems and approaches.

5.1 Experimental Framework

In our experiments, we test CREW in terms of (1) How fast it can disseminate information to a set of receivers over spread across a wide area network, (2) How it scales with increasing system size and increasing content size, (3) What is its data overhead, (4) How well it adapts and exploits heterogeneity in the networks and (5) How gracefully it scales in presence of heterogenous network errors.

To measure these factors, and be confident that the results would be a good indication of what one could expect in a real deployment, we setup a testbed using Modelnet [6], which is a real-time network traffic shaper and provides an ideal base to test various systems without modifying them. Further, Modelnet allows for customized setup of various network topologies[5]. Using Modelnet, we compare CREW with actual optimized implementations of BitTorrent, Bullet, SplitStream and Asynchronous TCP Gossip under different conditions. Next, we describe our experimental testbed and the network topologies that we used.

5.1.1 Testbed

The testbed consists of a FreeBSD machine as an emulator and four Debian Linux hosts. All machines support Gigabit ethernet interfaces and are connected by a dedicated Gigabit router. The emulator is a dual processor 2.6Ghz machine with 2GB of RAM while the hosts are single processor machines running at 2.8Ghz with 500MB of RAM. The emulator machine runs a custom FreeBSD Kernel configured with a system clock running at 1000Hz (as required by Modelnet). The hosts run Linux with a customized 2.6 version kernel [6]. The hosts support Java version 1.5, Python version 2.3.5 and GCC version 3.3.5. All hosts are synchronized to within two milliseconds through NTP (Network Time Protocol).

To model the vagaries of the underlying Internet, we used the *Inet* [4] topology generator tool to generate Internet router topologies of 5000 routers. Inet generates topologies on a XY plane which Modelnet then uses to emulate inter-router (and hence inter-node) latencies. Bandwidth constraints and network packet loss rates are specified separately in Modelnet. Primarily, we used two main network topologies: (1) a homogeneous network where all end nodes have equal bandwidth of 200Kbps and (2) a heterogeneous network with end nodes at three levels of bandwidth: 200Kbps, 800Kbps and 3200Kbps. Additionally, we generated homogeneous networks with varying packet loss rates, from 1% to 20%. For all network topologies, however, the latency between nodes is always heterogenous, as dictated by the router backbone generated by Inet. Our

[5] Another choice of testbed that we considered was PlanetLab (http://www.planet-lab.org/) but most nodes there have high bandwidth. Secondly, designing custom overlays with heterogenous bandwidths and loss rates would have been very difficult.

[6] This version supports NPTL (New Posix Threading Library), to efficiently support multiple threads.

choice of bandwidths for nodes requires some explanation since the testbed imposes certain restrictions. First, the maximum bandwidth generated in the testbed cannot exceed 1 Gbps (bottleneck of emulator NIC card and router). Second, to keep the emulator from being overloaded, we did not want to generate data at such a rate that the emulator CPU usage went above 10%. Third, while Modelnet provides for running many virtual nodes in one physical host, we did not want to create so many processes that the swap space was being used. Under these constraints, we would still like to simulate reasonable bandwidth assumptions. In our use-case of Shake-Cast, many emergency organizations connect via basic DSL. Choosing 200Kbps as the default value, therefor seemed reasonable. As noted, we also introduce nodes with higher bandwidths in certain tests. These, again reflect possible DSL bandwidths. However, we hasten to add that the bandwidth on all emulated nodes is fully bidirectional.

5.1.2 Comparison Systems

Our choice of comparison systems is not to exhaustively compare CREW to all dissemination systems but to compare it to well-known "sample points" in the application-layer broadcast/multicast systems space. The primary motivation is to test if CREW, and hence a gossip-based approach, can perform comparably to optimized overlay dissemination systems. BitTorrent is the current defacto system for distributing large content in the Internet today. Moreover, it is a fully mesh-based system. Bullet is a hybrid tree/mesh system, while SplitStream is primarily a tree/forest based system (for content delivery paths). Slurpie [30] (for cooperative mesh-based dissemination) and Chainsaw [24] (for mesh-based streaming) are examples of other systems that are known to have good performance. However, at this time, we are unable to obtain the source code of Chainsaw, and we are still in the process of setting up the test framework for Slurpie. Hence, experimental results for these two systems are not included in this paper. To compare these various systems, we ran actual implementations of them over Modelnet. It should be noted that some of the comparison systems are not designed for fixed size content delivery. However, for these systems, we have given optimistic interpretation of how they would disseminate fixed size content.

Specifics of the comparison systems are given below.

- **BitTorrent:** We downloaded and used the python source code for BitTorrent version 4.0.2. We changed the source code so that we could instrument the total bytes that were sent/received by a BitTorrent client.

- **Bullet:** For bullet, we used the source code of Macedon version-1.2.1 [5]. This version did not contain $Bullet'$[19], an optimized system designed explicitly for large content distribution. Bullet is inherently a streaming protocol. To compare it to other content dissemination systems, we made a minor change to the source code of the *appmacedon*

driver file. During streaming, a Bullet node logs the time when it first receives data, to the time when it receives data that corresponds to a particular file/content size. This is a simplification because there is no explicit logic in each node to get 'missing data'. This simplification is representative of the best case scenario if Bullet were used for dissemination content. Again, we emphasize that our comparison point is not $Bullet'$, which has explicit logic (and optimizations) to deal with heterogenous bandwidth nodes and content reassembly[7].

- **SplitStream:** The Macedon framework provides built-in support for various P2P protocols, including SplitStream. We used the same *appmacedon* driver as before but changed the underlying protocol to SplitStream. Again, a SplitStream node notes the time taken to receive data of a particular content size.

- **Asynchronous TCP Gossip:** We also developed a sophisticated Gossip system based on lpbcast. Our primary goal was to test the dissemination speed and hence we removed the sampling service logic of lpbcast, replacing it with an "ideal" sampling service. Each node is supplied with the list of all other nodes and thus does not need to send its view. Therefore, the overhead from sampling service is zero and is a best case scenario. Gossip is implemented asynchronously with each node sending every unique gossip message, as soon as it gets it, to 4 other nodes. Further, we send the gossip message via TCP due to the problems of congestion with UDP. A sophisticated communication substrate was also designed for sending the gossip messages. Each node maintains a thread-pool (of size 10) to send gossip messages. Many gossip messages can therefore overlap, if necessary, for increased concurrency. We added error handling as well. A gossip send is tried multiple (4) times (with increasing backoff time), in case the receiver is currently overwhelmed. This was designed as a primitive means of congestion control. Our goal was to take the basic idea of lpbcast and then implement concurrency, heterogeneity and congestion adaptation into it.

5.1.3 Testing Methodology and Metrics

Each of our experimental runs consists of one "server" and multiple peers. The server is a node in that particular system that initially has all the content. A test starts when the first peer receives the first piece of content and the test ends when all peers have all the content. The different nature of the systems introduces slight variations to the tests. Before a test starts, we want all nodes to be "up" and already started. In BitTorrent, the .torrent file (metadata) is already present in each node. We start the seeder last in a BitTorrent run so that there is no node startup latency. When the seeder enters the system, all nodes have already formed the BitTorrent mesh. For Bullet and SplitStream, we wait 30 seconds before streaming, so that

[7]Private communication with the authors of Bullet' reveal that the source code for Bullet' will be released soon and we will evaluate it then.

any optimization that they need to perform can take place. For CREW, we introduce the "server" last. Unlike BitTorrent, though, a run in CREW includes the metadata broadcast time as well. The server is always a 200Kbps node, irrespective of the network topology. We run each experiment five times and plot the average value of the five runs. In our experiments, we measure three major metrics:

- *Complete Dissemination Time* (or *Completion Time* in short). Completion time is the amount of time from when the dissemination process is started at the seeder until all (100% of) the nodes in the network receive all the content.
- *Dissemination Coverage Speed* (or *Coverage Speed* in short). Coverage speed captures how fast data dissemination proceeds over the network. It indicates how many nodes have received all the content at a certain point of time.
- *Dissemination Data Overhead Percentage* (or *Data Overhead* in short). Data overhead measures the average extra data bytes that are transmitted at each node for dissemination. It is defined as in the following equation.

$$Data\_Overhead = \frac{total\_data\_bytes\_transmitted}{num\_nodes \times file\_size} - 1$$

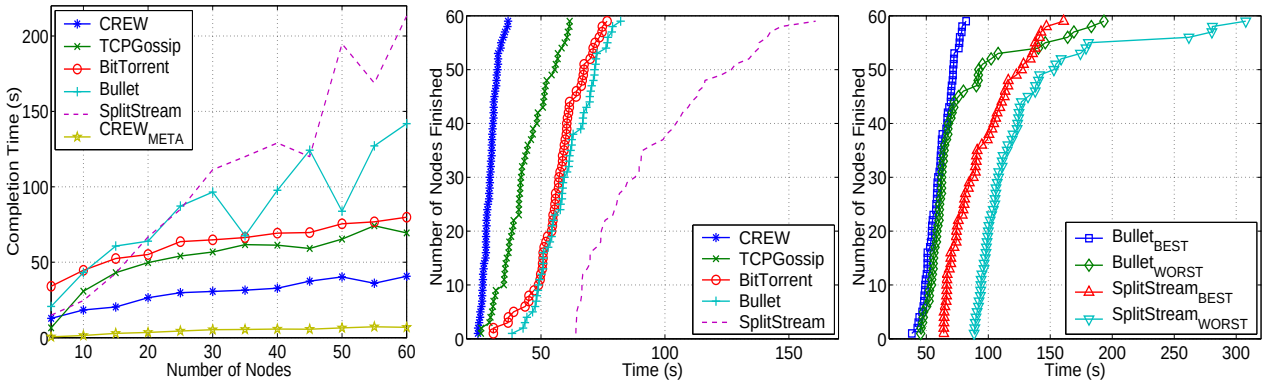
5.2 Experimental Results

The experimental results are presented in several aspects: network size scalability, content size scalability, adaptability to both bandwidth heterogeneity and lossy links. Unless otherwise specified, the default settings for the experiments are (1) homogeneous networks, (2) 1% upper loss rate, (3) 100K content size and (4) 60 nodes. We use an optimized version of CREW when comparing with other systems. At the end of the section, we present results that show why we selected this particular version of CREW.

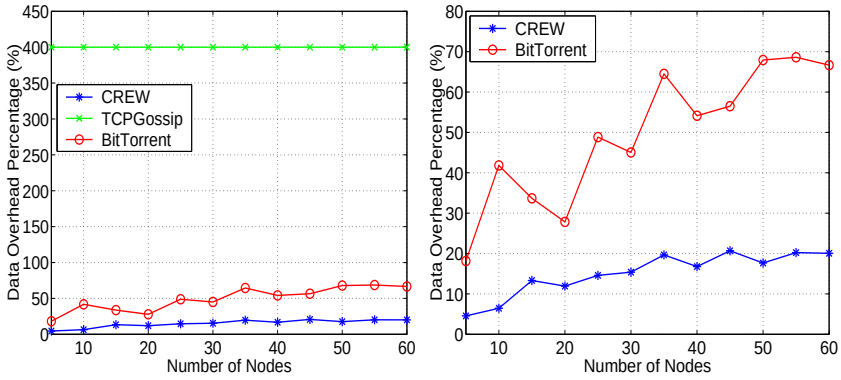
5.2.1 Network Size Scalability

We first analyze the time and data overhead to disseminate a 100K file among an increasing number of recipients. A homogeneous network with each node at 200Kbps is used and the total number of recipients is varied.

As shown in Fig-6(a), when the number of recipients is greater than 10, CREW disseminates faster than all the other systems and for 60 nodes, CREW is almost twice as fast the next best system, BitTorrent. CREW is therefore able to achieve extremely rapid dissemination. As previously stated (Sec-3.1), metadata propagation is extremely fast and initiates all nodes into the dissemination process. The $CREW_{META}$ line in Fig-6(a) shows this. Also, metadata propagation is extremely fast even as the network scale increases.



(a) Dissemination Time Vs Network Size (b) Coverage Speed (c) Coverage Speed (Best and Worst Case)

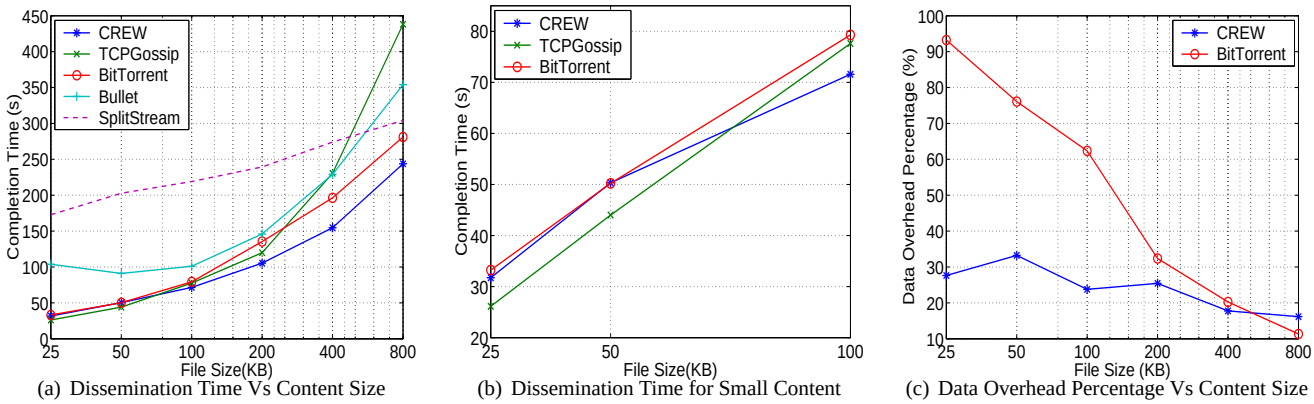


(d) Data Overhead Vs Network Size (e) Data Overhead: CREW and BitTorrent

Figure 6. Network Size Scalability in Homogeneous Networks (We were unable to instrument Bullet and SplitStream to get data overhead)

TCPGossip also scales well, with dissemination time close to that of BitTorrent. Bullet and SplitStream, however, seem to scale poorly and rather erratically. To examine why this was so, we plotted the dissemination spread of the various systems, as shown in Fig-6(b). The figure plots the completion times of 60 nodes for a particular run of the 5 systems. As an example, in SplitStream, after 100 seconds, around 38 nodes have received all the disseminated content. In Bullet, it takes a very long time for the last fraction of nodes to get all the data; a worst case is plotted in Fig-6(c). We conjecture that this may be because, Bullet and SplitStream take longer to stabilize and involve all nodes in the dissemination process. While disseminating large content, this is masked but becomes apparent when disseminating small amounts of data. Fig-6(b) also shows that at any given point of time, nodes in CREW get the content faster than any of the other systems. The fast ramp-up speed of CREW and significant concurrency contribute to its superior performance.

Fig-6(d) plots the comparison of data overhead with varying number of nodes for BitTorrent and CREW. TCPGossip



(a) Dissemination Time Vs Content Size (b) Dissemination Time for Small Content (c) Data Overhead Percentage Vs Content Size

Figure 7. Content Size Scalability in homogeneous Network

incurs a constant 400% data overhead (due to the fanout of 4, every node transmits each chunk 4 times), and hence is not plotted. For Bullet and SplitStream, the API provided did not allow us to instrument data transmitted and received and hence we were unable to measure their overhead. Hence, they too have not been plotted.

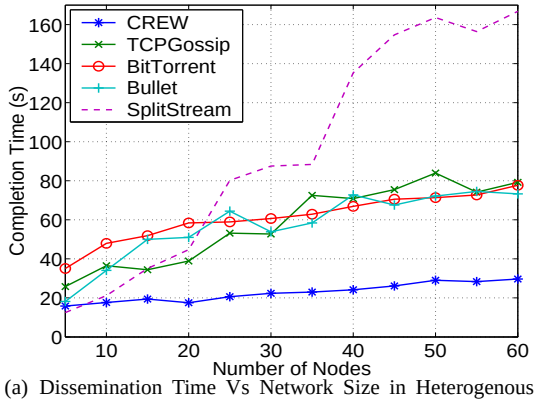
As Fig-6(d) shows, the overhead for CREW is much lesser than that of BitTorrent (and both are orders of magnitude less than TCPGossip). Additionally, the overhead in CREW seems to grow more slowly than that of BitTorrent, with increasing network size.

5.2.2 Content Size Scalability

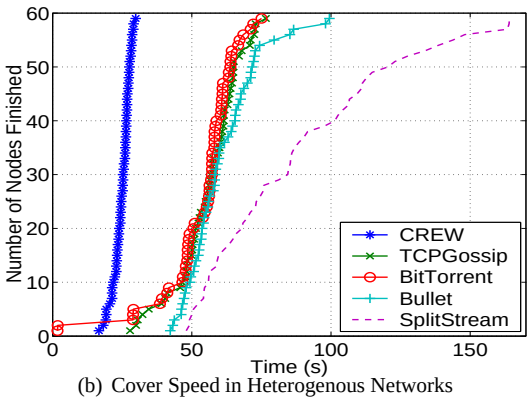
In this experiment, we examine the time and the data overhead to disseminate content of varying size, from 25K to 800K, among 60 peers with homogeneous bandwidth. The results are shown in Fig-7.

The dissemination time increases almost linearly for all systems. However, the different systems display interesting and characteristic behavior depending upon the content to be disseminated. TCPGossip does extremely fast dissemination when the content is small (as seen in Fig-7(b) but the time for complete dissemination increases more rapidly than other systems, when content size increases. Thus, it takes the longest time to disseminate 800K. This is characteristic and shows why gossip-based protocols are well suited for fast dissemination of small content but unsuitable for large content. SplitStream has the highest dissemination time for small content but scales extremely well. The remaining three systems (CREW, BitTorrent and Bullet) all exhibit behavior similar to one another but the gap between CREW and the other two systems seems to widen as the content size grows – suggesting that CREW may in fact perform quite well with very large content too.

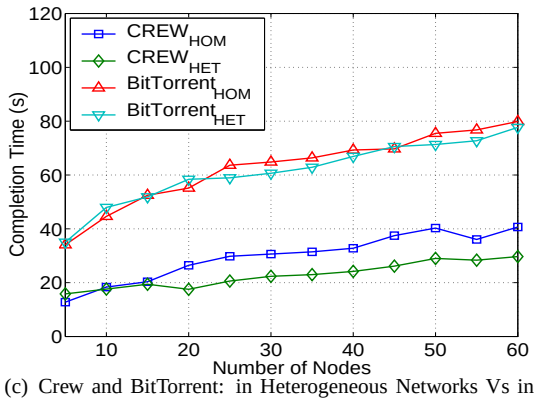
Fig-7(c) shows CREW's data overhead in disseminating different content size compared to BitTorrent. Both CREW and BitTorrent use less extra data to disseminate larger content with data overhead of BitTorrent decreasing more than that of



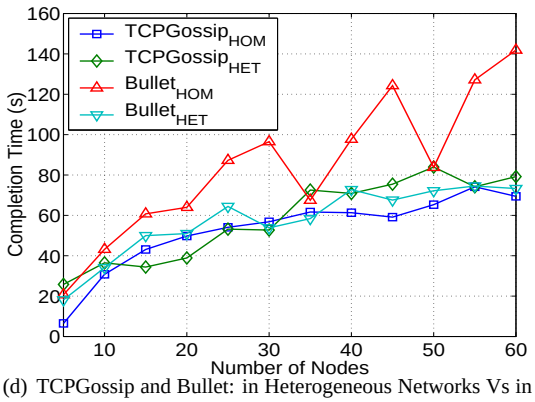
(a) Dissemination Time Vs Network Size in Heterogenous Networks



(b) Cover Speed in Heterogenous Networks



(c) Crew and BitTorrent: in Heterogeneous Networks Vs in Homogeneous Networks



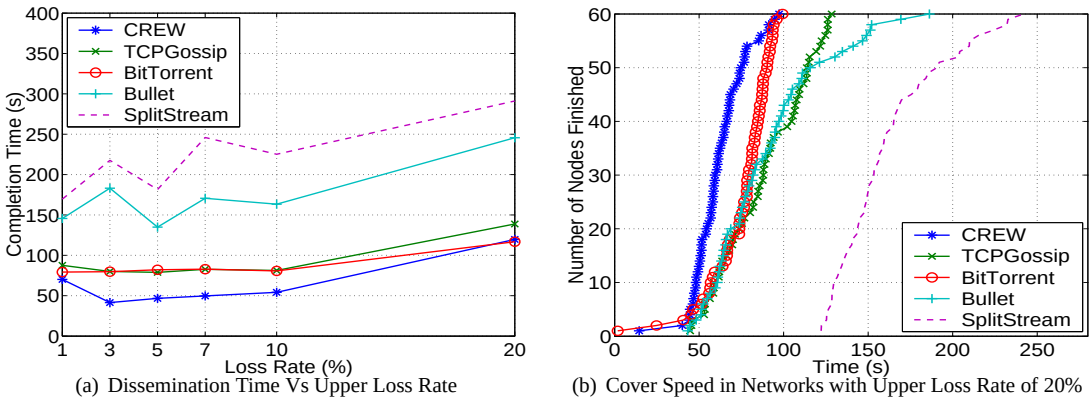
(d) TCPGossip and Bullet: in Heterogeneous Networks Vs in Homogeneous Networks

Figure 8. Network Size Scalability with Heterogenous Bandwidth Adaptation

CREW. However, overhead in CREW is quite low in the absolute sense varying between 18% to 35% (in the data sizes tested).

5.2.3 Heterogeneity Testing

We now evaluate how well the different systems are able to adapt to and exploit varying node bandwidths. In this experiment we maintained a constant ratio of high-bandwidth nodes to low bandwidth nodes; for every 4 nodes of 200Kbps, there is a high-bandwidth node of 800Kbps. Thus, while testing the dissemination time for 40 nodes, there are 32 low-bandwidth nodes and 8 high-bandwidth nodes. Additionally, when there are more than 45 nodes present, we introduce an even higher-bandwidth node, that of 3200Kbps. We manually changed the homogeneous network topology file of Modelnet to generate this heterogenous network. The network latencies are the same as the homogeneous network. The dissemination time of the various systems in heterogenous network are plotted in Fig-8(a). The spread times are shown in Fig-8(b).



(a) Dissemination Time Vs Upper Loss Rate

(b) Cover Speed in Networks with Upper Loss Rate of 20%

Figure 9. Adaptability to Network Faults

CREW, Bullet and SplitStream are all able to exploit heterogeneity to achieve faster dissemination time (as the comparisons show in Figs-8(c)(d)). However, BitTorrent seems unable to exploit heterogenous bandwidths and the dissemination time is not reduced as compared to that in a homogeneous network. This is probably due to the small content size and BitTorrent nodes do not get enough time to form a "good mesh". The time for BitTorrent to ramp-up to a good mesh therefore seems to affect its ability to exploit heterogeneity fast enough. TCPGossip displays a seemingly counterintuitive behavior – it performs worse in a heterogenous network as compared to a homogeneous network. This can be explained by conjecturing that high-bandwidth nodes overwhelm the low-bandwidth nodes, thus making chunk transfers to low-bandwidth take longer time. The effect of introducing high bandwidth nodes for Bullet is quite striking, with dissemination time reducing considerably. The performance of Bullet is now on par with that of BitTorrent and TCPGossip.

5.2.4 Adaptability to Network Faults

We now analyze the effect of packet loss rate on dissemination time. Our aim is to emulate an unpredictable network whose fault rate is not known in advance. To emulate this, we generated various topologies with Modelnet by specifying lower and upper bound packet drop rates. For example, by specifying an upper loss rate of 5% and a lower loss rate of 0%, Modelnet assigns a packet loss rate at random from 0-5% to each of the 5000 routers. The packet loss between any two end nodes is therefore different and heterogenous. We generated 6 different topologies with upper loss rates varying from 1% to 20% and lower loss rates fixed at 0%. The 20% loss rate topology is particularly pathological and extremely heterogenous in terms of the packet loss rates. The throughput of the systems are plotted in Fig-9.

CREW uses TCP for all its communication and intuition suggests that its performance must degrade sharply as the packet loss increases (causing TCP to perform poorly). However, as can be seen in Fig-9(a), the degradation, in reality, is graceful.

The concurrency in CREW is an extremely powerful mechanism that prevents rapid degradation of throughput under unstable network conditions. The degradation, however, still seems sharper as compared to BitTorrent. This is true if one considers 100% completion time of all peers. If the actual finish times of the various peers are compared, as in Fig-9(b), it is clear that most peers using CREW actually finish much faster than those in BitTorrent. It is the "tail", the last 10-15 peers, that actually make the total completion time for CREW longer. These peers are the ones that connect over very lossy network links and when they try to download chunks, they often "timeout" resulting in slow and repeated downloads. This is not an inherent property of CREW but an artifact of our implementation. We implement all RPC calls with a timeout. If an RPC doesn't complete in a particular time period, it is cancelled irrespective of whether data is still being exchanged. We are currently working to resolve this 'bug'. Fig-9(b) also shows that the peers over less lossy links are not penalized as much because of other lossy peers. In contrast, all nodes in BitTorrent are affected by lossy nodes resulting even in 'good' nodes finishing later.

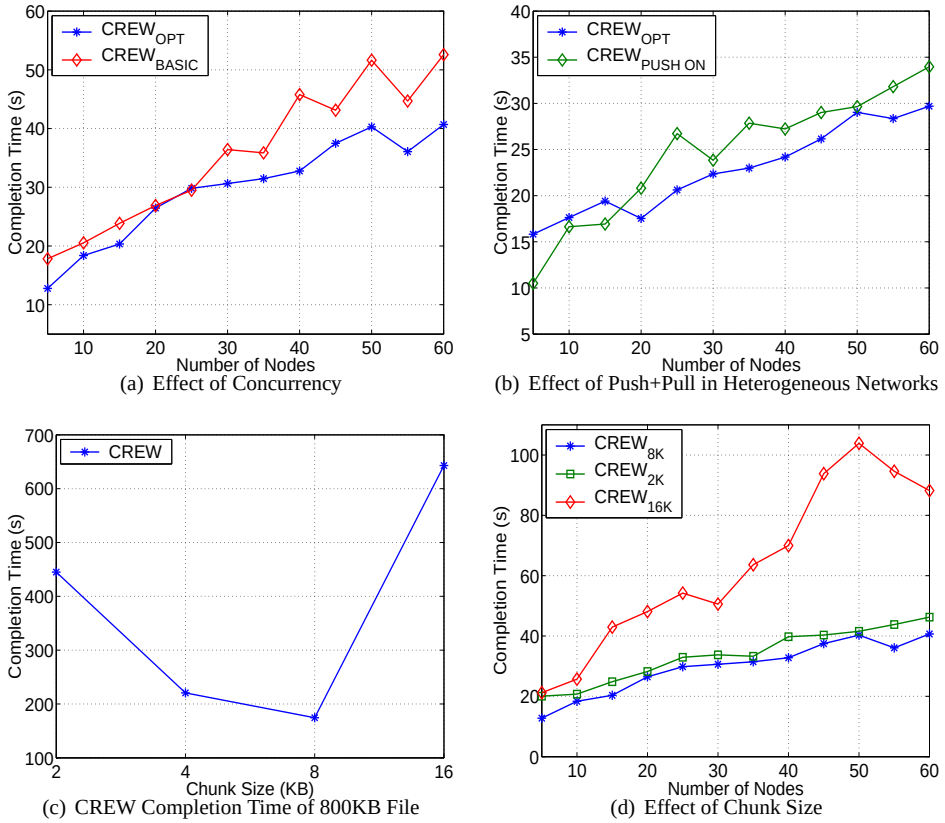
5.2.5 CREW Specific Optimizations

We now evaluate the performance of various variations of CREW to analyze what works best. In particular, we want to evaluate whether intra-node concurrency is a good feature, whether Push+Pull exploits heterogeneity better and finally, study the effect of chunk size on dissemination time.

Fig-10(a) plots the dissemination time for two variations of CREW: $CREW_{OPT}$ (Fig-2) has intra-node concurrency turned on while $CREW_{BASIC}$ (Fig-1) does not have any concurrency. The difference in dissemination time is quite noticeable, especially as the network of nodes grows in size. This reemphasizes the value of intra-node concurrency.

Next, we evaluate if Push along with Pull (PUSH_ON) leads to decreased dissemination time. Intuitively, Push+Pull should lead to increased concurrency and hence reduced dissemination time. However, as Fig-10(b), shows. PUSH_ON has slightly longer dissemination time as compared to a only-pull approach. We conjecture that this is due to push messages congesting the network without useful benefit; pull based concurrency seems enough to achieve good performance.

Finally, we evaluate the effect of chunk size on the dissemination time. Figs-10(c)(d) show quite clearly that a chunk size of 8KB results in the best performance. First, we evaluate the performance using a 800KB file for 60 nodes. The result is shown in Fig-10(c), which shows 8KB chunk to perform the best. The question then, is whether using other chunks results in loss of scalability. Fig-10(d) shows that a larger chunk (16KB) is quite detrimental to performance. A large chunk implies less number of chunks and also increased time to transfer a chunk. This results in decreased concurrency, hence slowing the



(a) Effect of Concurrency (b) Effect of Push+Pull in Heterogeneous Networks

(c) CREW Completion Time of 800KB File (d) Effect of Chunk Size

Figure 10. CREW Specific Optimizations

dissemination time. Making the chunk too small, however, results in too many pulls, again resulting in poorer performance.

5.3 Performance Evaluation Summary

Despite the optimizations and concurrency, CREW is still a gossip protocol and counter intuitive from a systems perspective. A node contacts another nodes at random, does gossip, and then, immediately moves away to another node. Thus, there is very little scope to amortize the connection setup cost. Making chunk sizes larger only increases the dissemination time (Fig-10(d)). Further, combined with the fact that we use TCP, we cannot achieve immediate full usage of bandwidth due to TCP slow-start. By the time full usage is reached, the chunk transfer may be over. However, as the results show, CREW not only performs well, but clearly outperforms the other dissemination systems. High intra and inter- node concurrency, combined with fast estimate of bandwidth by each node makes CREW an extremely fast protocol.

6 Concluding Remarks

Gossip based broadcast is extremely appealing for flash dissemination because it is scalable and resilient to faults. However, because gossip may entail redundant transmission of messages, its performance becomes poorer as the size of the disseminated content increases. In this paper, we introduced CREW, a new gossip-based protocol for flash dissemination that scales extremely well, achieving fast dissemination irrespective of network or content size. While current experimental results show CREW to be highly scalable, we would like to verify this both analytically and experimentally for an even larger number of nodes. Currently, we are setting up a testbed to test for thousands of nodes using Emulab [2].

Even though we designed CREW with flash dissemination in mind, its good performance may be useful for other applications as well. For example, CREW could be used for dissemination of software patches or updates to millions of machines. All machines should receive the required patches as soon as possible; otherwise they will be vulnerable for an extended period of time to suffer failures or attacks by hackers. In a different example, podcasts, RSS feeds, or news bulletins must be propagated very quickly to subscribers, because their utility and relevance degrades over time. CREW, is however, currently designed for nodes that are fully cooperative. Adapting CREW for a barter approach [12] is extremely challenging due to the gossip nature of the protocol; nodes cannot establish long standing 'barter agreements'.

References

[1] Bittorrent: http://bitconjurer.org/bittorrent/.

[2] Emulab: http://www.emulab.net/.

[3] Ice middleware: http://www.zeroc.com/.

[4] Inet: http://topology.eecs.umich.edu/inet/.

[5] Macedon: http://macedon.ucsd.edu/.

[6] Modelnet: http://issg.cs.duke.edu/modelnet.html.

[7] ARTHUR, D., AND PANIGRAHY, R. Analyzing the efficiency of bittorrent and related peer-to-peer networks. In *SODA* (2006).

[8] ARULANTHU, A. B., O'RYAN, C., SCHMIDT, D. C., KIRCHER, M., AND PARSONS, J. The design and performance of a scable orb architecture for cobra asynchronous messaging. In *International Conference on Distributed systems platforms* (2000).

[9] BRAHAMI, M., P.TH.EUGSTER, GUERRAOUI, R., AND HANDURUKANDE, S. Bgp-based clustering for scalable and reliable gossip broadcast. In *Proceedings of the LNCS Global Computing Workshop* (2004).

[10] CASTRO, M., DRUSCHEL, P., KERMARREC, A.-M., NANDI, A., ROWSTRON, A., AND SINGH, A. Splitstream: High-bandwidth multicast in a cooperative environment. In *SOSP* (2003).

[11] FARLEY, A. M. Broadcast time in communication networks. In *SIAM Journal on Applied Mathematics* (1980), vol. 39.

[12] GANESAN, P., AND SESHADRI, M. On cooperative content distribution and the price of barter. In *ICDCS* (2005).

[13] GKANTSIDIS, C., MIHAIL, M., AND SABERI, A. Random walks in peer-to-peer networks. In *IEEE INFOCOM* (2004).

[14] JELASITY, M., GUERRAOUI, R., KERMARREC, A.-M., AND VAN STEEN, M. The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. In *5th International Middleware Conference* (2004).

[15] KARP, R., SCHINDELHAUER, C., S.SHENKER, AND VOCKING, B. Randomized rumor spreading. In *IEEE Symposium on Foundations of Computer Science (FOCS) 2000.* (2000).

[16] KHULLER, S., AND KIM, Y.-A. On broadcasting in heterogeneous networks. In *ACM-SIAM Symposium on Discrete algorithms* (2004).

[17] KLIENBERG, J. Navigation in a small world. *Nature* (2000).

[18] KNUTSSON, B., LU, H., XU, W., AND HOPKINS, B. Peer-to-peer support for massively multiplayer games. In *IEEE INFOCOM* (2004).

[19] KOSTIC, D., BRAUD, R., KILLIAN, C., VANDEKIEFT, E., ANDERSON, J. W., SNOEREN, A. C., AND VAHDAT, A. Maintaining high bandwidth under dynamic network conditions. In *USENIX Annual Technical Conference* (2005).

[20] KOSTIC, D., RODRIGUEZ, A., ALBRECHT, J., AND VAHDAT, A. Bullet: High bandwidth data dissemination using an overlay mesh. In *Usenix Symposium on Operating Systems Principles (SOSP)* (2003).

[21] LAW, C., AND SIU, K.-Y. Distributed construction of random expander networks. In *IEEE INFOCOM* (2003).

[22] LIBEN-NOWELL, D., BALAKRISHNAN, H., AND KARGER, D. Analysis of the evolution of peer-to-peer systems. In *PODC* (2002), pp. 233–242.

[23] PADHYE, J., FIROIU, V., TOWNSLEY, D., AND KUROSE, J. Modelling tcp throughput: A simple model and its empirical validation. In *SIGCOMM* (1998).

[24] PAI, V., KUMAR, K., TAMILMANI, K., SAMBAMURTHY, V., AND MOHR, A. E. Chainsaw: Eliminating trees from overlay multicast. In *IPTPS* (2005).

[25] P.TH.EUGSTER, GUERRAOUI, R., HANDURUKANDE, S., KERMARREC, A.-M., AND P.KOUZNETSOV. Lightweight probabilistic broadcast. In *DSN* (2001).

[26] QIU, D., AND SRIKANT, R. Modeling and performance analysis of bittorrent-like peer-to-peer networks. In *SIGCOMM* (2004).

[27] RODRIGUES, L., HANDURUKANDE, S., PEREIRA, J., GUERRAOUI, R., AND KERMARREC, A.-M. Adaptive gossip-based broadcast. In *Conference on Dependable Systems and Networks* (2003).

[28] SCHMIDT, D. C., O'RYAN, C., PYARALI, I., KIRCHER, M., , AND BUSCHMANN, F. Leader/followers a design pattern for efficient multi-threaded event demultiplexing and dispatching. In *PLoP* (2000).

[29] SHEN, K. Structure management for scalable overlay service construction. In *NSDI* (2004).

[30] SHERWOOD, R., BRAUD, R., AND BHATTACHARJEE, B. Slurpie: A cooperative bulk data transfer protocol. In *IEEE INFOCOM* (2004).

[31] VERMA, S., AND OOI, W. T. Controlling gossip infection pattern using adaptive fanout. In *ICDCS* (2005).

APPENDIX: CREW Analysis

We analyze CREW with a simplified model in which all nodes have equal bandwidth and latency. Let the content to be disseminated be divided into M chunks, each of which, if transferred between two nodes using their full bandwidth, takes 1 unit of time. Let the total number of nodes be N. Assume that at time T_0, all nodes have received information on the list of

chunks to get and start their gossip loops. We model CREW as progressing in synchronous time steps and analyze the time complexity in terms of the time steps. Assume that each time step can be further sub-divided into three smaller time steps:

SubStep 1: Each node chooses another node, uniformly at random, and sends it its list of received chunks.

SubStep 2: Each node chooses to honor, uniformly at random, only one request. Other requests are rejected.

SubStep 3: A node sends out a chunk, if possible

Notice, that after the first time step, a node may have multiple 'incoming' requests from other nodes. Plus each node has one 'outgoing' request. Therefore in the second sub-step, a node chooses at random from all the incoming and the one outgoing request. If a node, say req chooses an outgoing request, then in the third time step, it simply waits to hear from the requested node. On the other side, the requested node, say rep, may have decided to honor some other request, in which case req will not get any data in the third time step. Only if rep also chooses req and if rep has some chunk that req does not have, will req get anything in the third sub-step. We also assume that the second sub-step is long enough so that the node can receive all the incoming requests. Additionally, we assume that the third substep is the one that takes the longest time. If a node does not get (or give) anything in the third substep, it still waits the amount of time that it would have taken had it sent out a chunk. In effect, nodes are fully synchronized at each step and also at each substep.

Dissemination Time:

To model CREW, we started with an approximate, analytical model. We assumed that for the first M steps, the seeder picks one node at random and injects it with a chunk (thus giving out all M chunks in M time). Thereafter, we used an approximate expected analysis to model the number of chunks possessed by each node, at each time step. This analysis led us to the following theorem:

Theorem 6.1 *The expected average number of chunks in a peer is* $N_{k+1} = N_k + \frac{1}{4}(1-(1-\frac{N_k}{M}(1-\frac{N_k}{M}))^M)$, *with* $N_0 = \frac{M}{N}$. *where* k *is the number of time steps after all* M *chunks have been injected by the root into the network.*

Proof: Omitted

This recurrence relation has no closed form solution and hard to analyze. Therefore, we decided to build a simulator for the simplified model of CREW (where the seeder does not play any special part in the first M steps, as in Theorem-6.1).

The simulator simulates the simplified CREW protocol for given N and M values. The output is the number of time steps needed for all nodes to have all chunks. Due to the random nature of CREW, the time steps (even for fixed M and N values) vary over different iterations. Therefore, for each combination of M and N, we ran 50 runs. The results are shown in Fig-11.

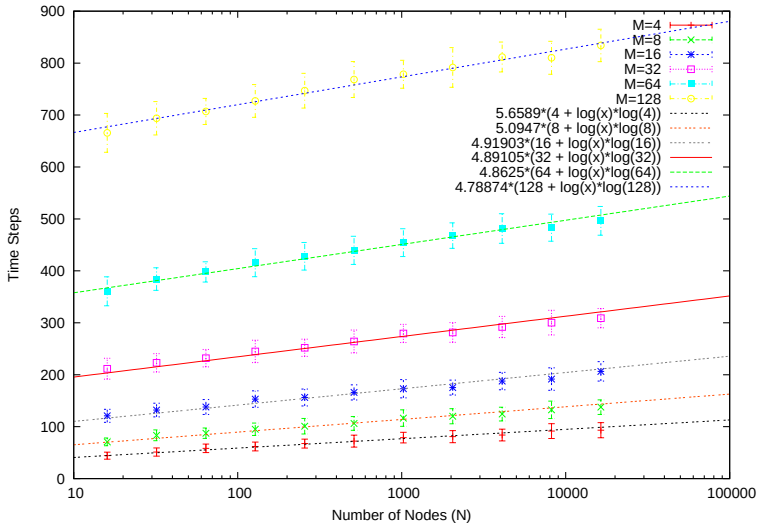


Figure 11. Dissemination Time Steps using Simulator

The X-axis is logarithmic and represents the number of nodes (N). The Y-axis is the total time steps for N nodes to receive all M chunks. We tested for different values of M and these are plotted in the same graph. Each data point represents the average value of these 50 runs along with the standard deviation. We also plot 6 curves that best fit the data points.

We tried a wide variety of sample functions to test their 'fit' to the data; among others we tried $O(M + LogN)$ and $O(M * LogN)$. However, as Fig-11 shows, $a * (M + LogN * LogM)$ is a very good fit. The reduced Chi-square for each of the fitted lines is less than 0.1 (indicating an extremely good fit). The value of a for each of the fitted lines decreases quite slowly (with logarithmic increase in M) indicating that this is both a conservative and good approximation. For other sample functions, the reduced Chi-square was either quite large or a increased with M. Thus, we believe that the dissemination time for CREW can be modeled as $O(M + LogN * LogM)$. This is suboptimal with respect to the optimal solution of $O(M + logN)$. However, the constant factor for CREW seems to be low (around 5). Additionally, the suboptimality of CREW increases only logarithmically with number of chunks (or file size). For low number of chunks, the difference between CREW and an optimal solution would be quite small.

Chunk Size versus Dissemination Time: Let the number of steps required by CREW for dissemination be $K * (LogN * LogM + M)$[8]. Assume that a for chunk size of 1, it takes one unit of time for an entire step to finish. Further let the time step be directly proportional to the chunk size. With chunk size 1, the time required is $K * (LogN * LogM + M) * 1$.

Let each chunk now be B times bigger. If originally there were M messages of unit size, we now have M/B messages of

[8]We had previously assumed a dissemination time of $K * (M + LogN)$ for the analysis. We now present the analysis with the updated model for dissemination time.

size B, and each transfer now takes B times longer. The time needed for CREW is now:

$$\begin{aligned} Time &= K * (LogN * Log(M/B) + M/B) * B \\ &= K * (B * LogN * LogM - B * Log(B) + M) \end{aligned} \tag{1}$$

Therefore as chunk size increases (from 1), dissemination time increases (for $LogN * LogM > Log(B)$).

Data Overhead: In CREW, no duplicate chunks are ever sent out. Hence, the overhead is entirely due to nodes sending out the list of chunk-ids; let this list be called the ***HandshakeMessage***. If we assume that all nodes perform pulls until all nodes get the entire content (a worst case scenario), then the total data sent out in CREW is: $TimeSteps * N * Size(HandshakeMessage) + N * M * Size(Chunk)$. Further, let's define data overhead as $(sent - min)/min$ where $sent$ is the total data sent into the network and min is the minimum amount of data that needs to be sent out (by any scheme). It is easy to see that min is $N * M$ (all nodes have to get M chunks). For simplicity and without loss of generality, let size of the chunk be 1 and the size of handshake be h (some fraction of the chunk size).

The overhead in CREW can now be calculated as:

$$\begin{aligned} Overhead &= \frac{K * (LogN * LogM + M) * N * h + (N * M * 1) - (N * M * 1)}{N * M * 1} \\ &= \frac{K * (LogN * LogM + M) * h}{M} \\ &= \frac{hKLogNLogM}{M} + hk \\ &= hk(1 + \frac{LogNLogM}{M}) \end{aligned} \tag{2}$$

The overhead, therefore, increases (sublinearly) with network size and decreases with increasing content size. This is because, when the content is made bigger, the sustained throughput phase is much longer and the overhead in this phase is much smaller (nodes are not wasting their pulls).