CREW: A Gossip-based Flash-Dissemination System

Mayur Deshpande, Bo Xing, Iosif Lazardis, Bijit Hore, Nalini Venkatasubramanian & Sharad Mehrotra
University of California, Irvine
{mayur,bxing,iosif,bhore,nalini,sharad}@ics.uci.edu

Abstract

In this paper, we explore a new form of dissemination called Flash Dissemination that involves dissemination of fixed, rich information to a large number of recipients in as short a time as possible. Key characteristics of Flash Dissemination include unpredictability in its need, scalability to large number of recipients and autonomic performance in highly heterogenous and failure-prone environments. Previous work either addresses large content delivery in heterogenous networks or fault-tolerant dissemination of (streaming) events. We investigate a peer-based approach using foundations from broadcast networks, gossip theory and random networks. In this paper, we propose CREW (Concurrent Random Expanding Walkers), a scalable, lightweight, and autonomic gossip-based protocol. CREW is also explicitly designed to maximize the speed of dissemination using adaptive and intelligent intra and inter node concurrency. We implemented CREW on top of a scalable middleware environment and compared it to optimized implementations of popular gossip and peer-based systems. Our experiments show that CREW outperforms both traditional gossip and current large content dissemination systems, across a wide range of comparative metrics, even though its design is counter-intuitive from a systems perspective.

Keywords: Gossip, Broadcast, Peer-to-Peer, 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 rapid dissemination of varying amounts of information to a large number of recipients in a very short period of time. We motivate flash dissemination with an example from the emergency management domain. Consider "Shake-Cast", a service from the Advanced National Seismic System (http://www.anss.org) 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. 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: Flash Dissemination events (e.g. disasters) are unpredictable and are not known in advance. A flash dissemination system, must be ready to work at very short notice and cannot be scheduled or optimized in advance. Further, the underlying network infrastructure may also be unpredictable.

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, network heterogeneity in latency is natural. Additionally, different receivers may possess different bandwidth capacities resulting in bandwidth heterogeneity. Content heterogeneity arises since rich information such as pictures, small voice/video 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. 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 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 [16]. 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 to deliver content of the order of hundreds of MBs to GBs. We explore the latter systems in more detail in Sec-2. Again, it is not obvious if large content delivery systems can achieve very fast dissemination for medium amounts of data. Additionally, these systems are not designed to handle unpredictable faults but are tailored assuming certain network and host behavior, e.g., a constant 'churn' rate ([19, 22]). On the other hand, gossip-based broadcast systems are designed to accommodate unpredictable faults. However, gossip-based protocols face scalability issues on many fronts and do not usually take into account large network heterogeneity. 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 over heterogeneous networks *and* under unpredictable conditions. CREW is a new, fully decentralized, gossip-based protocol, designed from the ground up, focussed on reducing data overhead and increasing both inter and intra 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 very 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 dissemination of rich information. CREW is almost twice as fast as current, optimized dissemination systems (such as BitTorrent and Bullet) for flash dissemination and imposes order of magnitude 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 both heterogenous bandwidth and 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 a lightweight, scalable, and 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

[1] 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. An optimal solution[11] exists for homogenous network but this problem is NP-hard for a heterogenous network [15]. Approaches for fast dissemination in heterogenous networks center around identifying and exploiting the high-bandwidth nodes without overwhelming the low-bandwidth nodes [9, 18, 1, 25]. Common overlay data structures used to implement these systems include trees/forests [9], pure meshes [1] and hybrid tree/mesh [18]. Empirical evidence suggests that meshes, in general, offer higher throughput and better fault tolerance. These systems are designed for content that is fixed and large or long-lasting (10s of mins and more) streams. Thus, either time is spent in pre-optimizing the overlay [9] or moving the overlay towards high throughput [18, 1]. For flash dissemination, we cannot pre-optimize since the network can change dramatically after a disaster. Neither is there time to move towards a good overlay at runtime since flash dissemination is usually concerned with medium amounts of data (hundreds of KBs to couple of MBs) and thus dissemination finishes within a couple of minutes (at most).

Additionally, during disasters, systems and networks become unstable and unpredictable; therefore, a primary objective is to achieve dissemination in less-than-perfect network conditions. Gossip [14] based broadcast protocols are an almost perfect fit for this scenario due to their stateless and fault-tolerance properties. They can be roughly divided into 'pure' or 'hybrid' approaches. In hybrid approaches [7], the primary dissemination is in a non-gossip manner (e.g., along an overlay tree) with gossip being used to deal with faults and lost messages. In pure gossip-protocols, all or most of dissemination occurs via gossip. For e.g. in lpbcast [21], each message (or data chunk) is gossiped (usually blindly forwarded in a 'fire and forget' manner) to 'fanout' number of other nodes, chosen at random. This blind forwarding is key to gossip's fault-tolerance property but adds a significant overhead resulting in slow dissemination speed (as shown in experiments in [18]). Dissemination speed can be improved by reducing the fanout but his results in decreased reliability. Additionally, fixing a constant fanout under-utilizes high-bandwidth nodes and may also overwhelm low-bandwidth nodes in a heterogenous setting. Recently researchers have started to examine how fanout can be autonomically and dynamically changed at runtime [26, 20]. Apart from autonomic adaptation in heterogenous networks, gossip protocols also face other challenges with regard to decentralized and scalable 'view' and memory buffer management [13, 23] . Finally, the use of UDP as underlying primitive creates problems of network congestion, especially in wide-area heterogenous networks and has to be dealt with explicitly [8].

The deficiencies listed above are not intrinsic to gossip behavior. As noted, pieces of research exist that (partially) address one or more of these issues individually. However, there is no work that addresses all these challenges in a simple, unified manner for the purpose of flash dissemination. The rationale for CREW is to show that gossip based protocols can in fact overcome these deficiencies and achieve extremely fast flash dissemination of medium sized data in unpredictable and heterogeneous environments.

[1] A more detailed examination can be found in [10]

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: reducing redundant data and providing low-overhead concurrency. Then, to tackle heterogeneity, we introduce concurrency within a node and adapt it to local and global bandwidth availability. We begin by describing the techniques to support fast gossip in CREW, followed by introducing the extensions to support heterogeneity.

3.1 Basic CREW: Making Gossip Fast

As note in Sec-2, the basic bottleneck in gossip is the high data overhead due to redundant messages leading to decreased throughput and slow dissemination time. 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. Bounce protocol is described in detail in [10].

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 as-

```
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 ($X, GossipPull, RecvdChunksIds$)
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

```
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 for Heterogeneous Networks

signed 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 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.

3.1.2 Enabling Low Overhead Concurrency

Metadata is small and is received by all nodes very fast. Thus, all nodes are active rapidly and trying to pull chunks aggressively. 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, 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. 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. [12] 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). Finding the next random node to gossip with, is now 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.

[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

3.2 Extending CREW for Heterogeneity

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, how to (1) reduce the detrimental effects of high latency? (2) exploit high bandwidth nodes? and (3) 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 which, 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 may result in a node receiving duplicate chunks. 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 nodes have may low latency while others have high latency. Chunk transfers from lower-latency nodes can overlap with the setup to higher-latency nodes – thereby masking setup cost. The problem then is deciding 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. 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 "relative maximum bandwidth" being tricky to define precisely (See [10] for more details). Current systems, like BitTorrent and $Bullet'$ use heuristics to calculate a node's maximum bandwidth. In general, the idea is for each node to 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 estimate maximum bandwidth rapidly. In CREW a node is constantly establishing new connections; this allows it to guess its relative maximum bandwidth very fast. In contrast, say BitTorrent, nodes evaluate other nodes in the network quite slowly. More details on CREW's bandwidth estimation can be found in [10]. Once maximum bandwidth is calculated, calculating of spare bandwidth is straightforward. Nodes also use the estimate of maximum bandwidth to decide whether to allow other peers to download chunks from them. If a peer is using up all its bandwidth, then it will 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 Autonomic 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, autonomically*. 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 which allows CREW's performance to degrade linearly, instead of exponentially, as a function of packet loss rate. Using TCP also provides other important benefits – such as automatic congestion control at the network level. However, using TCP 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 which we describe next.

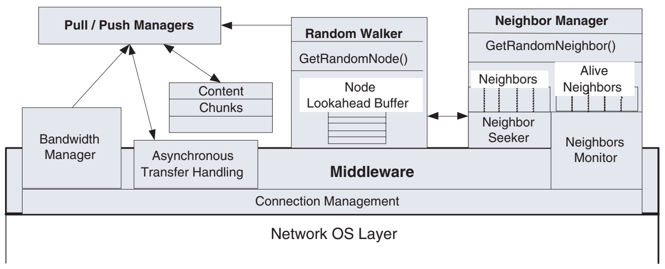


Figure 3. CREW System: Main Modules

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 [2], as our fundamental software platform which allows us to leverage all the benefits of a cross-platform middleware platform. CREW is implemented as a set of interacting modules, as shown in Fig-3. 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 two of the modules in greater detail. For a description of the other modules please see [10].

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 received from the Bandwidth Manager), the pull manager initiates gossip pulls. Concurrent pulls can be naturally handled in separate threads but ICE allows for a more efficient mechanism, Asynchronous Method Invocation (AMI)[6]. 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[24] thread pool based on the select I/O system call.

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.

5 Performance Evaluation

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 [5], 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. 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 (emulator) and four Linux hosts. All machines have Gigabit ethernet and are connected by a dedicated Gigabit router. The emulator is a dual 2.6Ghz machine with 2GB RAM while the hosts are single processor (2.8Ghz) machines with 500MB RAM. The emulator machine runs a custom FreeBSD Kernel with a system clock at 1000Hz (as required by Modelnet). The hosts run Linux with a customized 2.6 version kernel [4]. The hosts support Java version 1.5, Python version 2.3.5 and GCC version 3.3.5. All hosts are synchronized to within 2 msec through NTP (Network Time Protocol).

To model the vagaries of the underlying Internet, we used the *Inet* [3] 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. 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.

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, fully mesh-based system, is the current defacto system for distributing large content in the Internet today. Bullet is a hybrid tree/mesh system, while SplitStream is primarily a tree/forest based system (for content delivery paths). 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 (ver-4.0.2). We changed the source code to instrument the total bytes sent/received by a BitTorrent client.

Bullet: For bullet, we used the source code of Macedon version-1.2.1 [4]. This version did not contain $Bullet'$[17], 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. 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

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

is a simplification because there is no explicit logic in each node to get ‘missing data’. This is representative of the best case scenario if Bullet were used for disseminating fixed content.

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.

Asynchronous TCP Gossip: We also developed a sophisticated Gossip system based on lpbcast. Our primary goal was to test dissemination speed and hence we removed the sampling service logic of lpbcast, replacing it with an “ideal” one. Each node is supplied with the list of all other nodes and does not need to send its view. Hence, 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, sending a gossip message multiple (4) times (with increasing backoff time), in case the receiver is currently overwhelmed. This was designed as a 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 that initially has all the content. A test starts when the first peer receives the first piece of content and 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 so that there is no node startup latency. When the seeder enters the system, all nodes have already formed the mesh. For Bullet and SplitStream, we wait 30 sec before streaming, so that 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. We measure three major metrics:

1. 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.

2. 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.

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

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

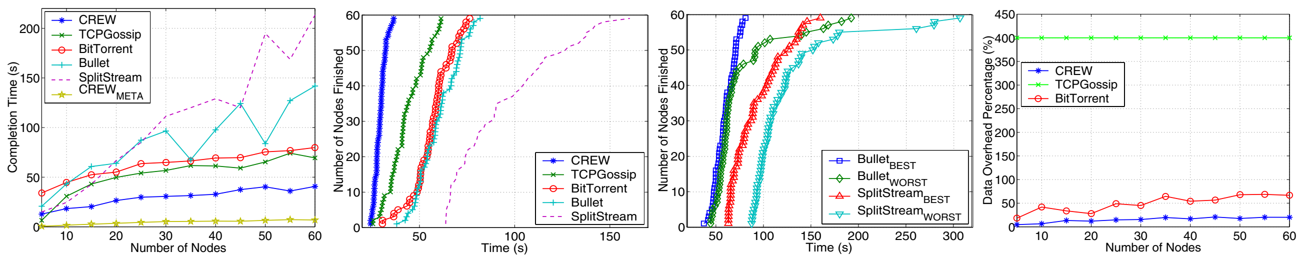
We first analyze the time and data overhead to disseminate a 100K file among an increasing set of recipients. A homogeneous network of 200Kbps nodes (latency is heterogenous) is used and the total number of recipients is varied. As Fig-4(a) shows, when the number of recipients is greater than 10, CREW disseminates faster than all the other systems; and for 60 nodes, CREW is almost twice as fast the next best system, BitTorrent. CREW, therefore achieves extremely rapid dissemination. As previously stated (Sec-3.1), metadata propagation is extremely fast and initiates all nodes almost simultaneously into the dissemination process ($CREW_{META}$ line in Fig-4(a)).

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-4(b) where the completion times of 60 nodes for a particular run of the 5 systems is plotted. 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-4(c). We conjecture that 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-4(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-4(d) plots the comparison of data overhead with varying number of nodes for BitTorrent and CREW. TCPGossip 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-4(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.

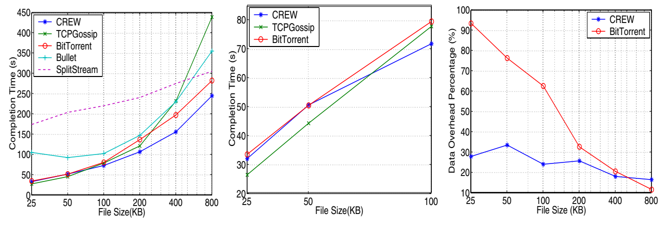
5.2.2 Content Size Scalability

In this experiment (Fig-5), we examine the time and data overhead to disseminate content of varying size, from 25K to 800K, among 60 peers with homogeneous bandwidth. 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-5(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 systems (CREW, BitTorrent and Bullet) exhibit similar behavior – suggesting that CREW may



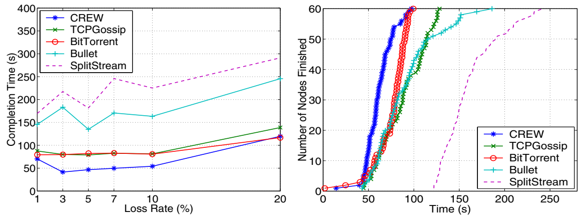
(a) Time Vs Network Size (b) Coverage Speed (c) Coverage (Best and Worst Case) (d) Data Overhead Vs Network Size

Figure 4. Network Size Scalability in Homogeneous Networks



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

Figure 5. Content Size Scalability



(a) Time Vs Pkt. Loss Rate (b) Coverage @ 20% Loss Rate

Figure 7. Adaptability to Network Faults

in fact perform quite well with very large content too. Fig-5(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 CREW.

5.2.3 Adaptation to Heterogeneous Networks

We evaluate how different systems can adapt to, and exploit varying node bandwidths. We maintained a constant ratio of one high-bandwidth node of 800Kbps per 4 low-bandwidth nodes of 200Kbps. Thus, while testing the dissemination time for 40 nodes, there are 32 low-bandwidth nodes and 8 high-bandwidth nodes. Additionally, when more than 45 nodes are present, we introduce an even higher-bandwidth node – 3200Kbps. We manually changed the homogeneous network topology file of Modelnet to generate this heterogenous network, using the same latencies as the homogeneous network. The dissemination times are plotted in Fig-6(a). The spread times are shown in Fig-6(b).

CREW, Bullet and SplitStream are all able to exploit heterogeneity to achieve faster dissemination time (as shown in Figs-6(c)(d)). However, BitTorrent seems unable to exploit heterogenous bandwidths and the dissemination time is not reduced 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's behavior is seemingly counterintuitive – it performs worse in a heterogenous network. We conjecture that this is due to high-bandwidth nodes overwhelming low-bandwidth ones, thus making chunk transfers to low-bandwidth take longer time. The effect of introducing high bandwidth nodes for Bullet is striking, reducing dissemination time considerably.

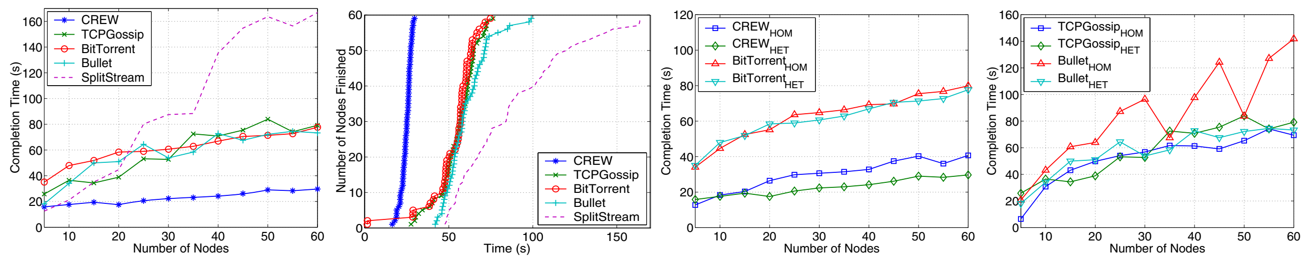
5.2.4 Adaptability to Network Faults

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-7.

CREW uses TCP for all its communication and intuitively its performance must degrade sharply as the packet loss increases However, as can be seen in Fig-7(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-7(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 (An explanation of this anomaly can be found in [10]).

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



(a) Time Vs Network Size (b) Coverage Speed (c) Crew & BitTorrent: Heterogeneous Vs Homogeneous (d) TCPGossip & Bullet: Heterogeneous Vs Homogeneous

Figure 6. Network Size Scalability with Heterogenous Bandwidth Adaptation

scope to amortize the connection setup cost. Making chunk sizes larger only increases the dissemination time. 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, pure gossip entails redundant transmission of messages and 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. Even though we designed CREW with flash dissemination in mind, its good performance may be useful for other applications as well. For example, it could be used to make web-servers scalable. Images or large html pages fit perfectly with the data size that CREW is very good at disseminating fast.

Acknowledgements

Our work was supported by NSF (Award Numbers 0331707 and 0331690). We are grateful to the Los-Angeles Emergency Preparedness (LA-EPD) department for allowing one of the authors to observe and learn about emergency preparedness; from which the primary motivation for this work emerged. We would also like to thank Prof. Kathleen Tierney of U. Colorado and Prof. Carter Butts for their guidance and insights.

References

[1] Bittorrent: http://bitconjurer.org/bittorrent/.
[2] Ice middleware: http://www.zeroc.com/.
[3] Inet: http://topology.eecs.umich.edu/inet/.
[4] Macedon: http://macedon.ucsd.edu/.
[5] Modelnet: http://issg.cs.duke.edu/modelnet.html.
[6] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, M. Kircher, and J. Parsons. The design and performance of a scable orb architecture for cobra asynchronous messaging. In *International Conference on Distributed systems platforms*, 2000.
[7] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. In *ACM TOCS*, 1999.
[8] M. Brahami, P.Th.Eugster, R. Guerraoui, and S. Handurukande. Bgp-based clustering for scalable and reliable gossip broadcast. In *Proceedings of the LNCS Global Computing Workshop*, 2004.
[9] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in a cooperative environment. In *SOSP*, 2003.
[10] M. Deshpande, B. Xing, I. Lazardis, B. Hore, N. Venkatasubramanian, and S. Mehrotra. Crew: A gossip-based flash-dissemination system, http://www.ics.uci.edu/∼mayur/crew_ics_tr_2005.pdf, 2005.
[11] A. M. Farley. Broadcast time in communication networks. In *SIAM Journal on Applied Mathematics*, volume 39, 1980.
[12] C. Gkantsidis, M. Mihail, and A. Saberi. Random walks in peer-to-peer networks. In *IEEE INFOCOM*, 2004.
[13] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. In *5th International Middleware Conference*, 2004.
[14] R. Karp, C. Schindelhauer, S.Shenker, and B. Vocking. Randomized rumor spreading. In *IEEE Symposium on Foundations of Computer Science (FOCS) 2000.*, 2000.
[15] S. Khuller and Y.-A. Kim. On broadcasting in heterogeneous networks. In *ACM-SIAM Symposium on Discrete algorithms*, 2004.
[16] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *IEEE INFOCOM*, 2004.
[17] D. Kostic, R. Braud, C. Killian, E. Vandekieft, J. W. Anderson, A. C. Snoeren, and A. Vahdat. Maintaining high bandwidth under dynamic network conditions. In *USENIX Annual Technical Conference*, 2005.
[18] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Usenix Symposium on Operating Systems Principles (SOSP)*, 2003.
[19] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the evolution of peer-to-peer systems. In *PODC*, pages 233–242, 2002.
[20] J. Pereira, L. Rodrigues, and A. Pinto. Low latency probabilistic broadcast in wide area networks. In *IEEE SRDS*, 2004.
[21] P.Th.Eugster, R. Guerraoui, S. Handurukande, A.-M. Kermarrec, and P.Kouznetsov. Lightweight probabilistic broadcast. In *DSN*, 2001.
[22] D. Qiu and R. Srikant. Modeling and performance analysis of bittorrent-like peer-to-peer networks. In *SIGCOMM*, 2004.
[23] L. Rodrigues, S. Handurukande, J. Pereira, R. Guerraoui, and A.-M. Kermarrec. Adaptive gossip-based broadcast. In *Conference on Dependable Systems and Networks*, 2003.
[24] D. C. Schmidt, C. O'Ryan, I. Pyarali, M. Kircher, , and F. Buschmann. Leader/followers a design pattern for efficient multi-threaded event demultiplexing and dispatching. In *PLoP*, 2000.
[25] K. Shen. Structure management for scalable overlay service construction. In *NSDI*, 2004.
[26] S. Verma and W. T. Ooi. Controlling gossip infection pattern using adaptive fanout. In *ICDCS*, 2005.