MIT 18.996: Topic in TCS: Internet Research Problems **Spring 2002**

Lecture 13 — May 1, 2002

Lecturer: Michael Mitzenmacher *Scribe: Omar Aftab, Omar Bakr*

13.1 Introduction

The distribution of content to large, world-wide audiences presents major challenges to content providers. The data to be delivered ranges from large files shared infrequently to smaller files that are continually accessed. Data distribution applications span from sharing critical enterprise data to downloading large software and popular games. Popular streaming applications include on-demand downloads, and the live streaming of high-bandwidth video and audio. All these applications require a combination of reliability, speed and scalability.

As we have seen in previous lectures, the early Internet was designed for slow connection speeds and small file transfers. Technological improvements have led to faster equipment and greater bandwidth, but the Web is still plagued with network congestion, packet loss and latency issues.[1] These problems are growing as more and more users access increasingly large files - and as more and more content providers start to host them.

Even with improved technology, and Akamai's Edge Delivery systems that do much to mitigate these issues, a new paradigm for data-delivery may eventually be needed. This lecture discusses novel data delivery techniques that exploit advances in coding theory to optimize the delivery of data.

13.2 Current Delivery Solutions

There are currently two broad alternatives for delivering popular content over the Internet. Neither is ideal, each having its own set of strengths and weaknesses.

13.2.1 Point-to-Point Solutions

In the traditional point-to-point solution for content delivery, each user initiates a TCP connection with the server, which creates an unique, independent data stream just for him. The user then proceeds to download the file at his own rate, prompting the server to retransmit any packets that get corrupted or lost. This simple scheme allows users to:

- **Download On Demand:** A major advantage of this scheme is that it allows users to access content on demand, making downloads easy and convenient
- **Resume Downloads:** Users can seamlessly continue downloads after temporary interruptions. They simply have to reconnect to the server and request a stream that starts where they left off

[1]For an overview of the problems that face the Net today, see Lecture I.

- **Tolerate Packet Loss:** TCP's adaptive retransmission allows users to tolerate some packet loss. If packets are missing, the server is transparently requested to retransmit them

Because of these virtues, this scheme is almost universally used in practice today. However, the approach also has a number of major shortcomings. In fact, it invariably results in:

- **High Server Load:** One major disadvantage of the scheme is that the server must deliver and manage thousands of individual TCP streams simultaneously. If too many users attempt to access the content, the server can be rapidly bogged down. An example of this problem is the Victoria's Secret Online Fashion Show, which was heavily advertised at the 1999 Super Bowl. When more than a million subscribers attempted to access the live webcast simultaneously, the central servers could not cope, swamping the site and leaving nearly all viewers unable to view the webcast.
- **High Network Load:** The network, too, is overloaded since it has to handle thousands of simultaneous streams
- **Severe Scalabilty Issues:** Conventional protocols, such as TCP, deal with packet-loss by requesting the sender to retransmit the missing packets. While this approach works for unicast networks, its multicast analog is known to be unscalable. Consider, for example, a server distributing large files to thousands of clients. As clients lose packets, their requests for retransmission can quickly overwhelm the server - a process commonly referred to as *feedback implosion.* Adaptive retransmission, then, is clearly problematic for the distribution of popular content: the approach is simply not scalable

13.2.2 Broadcast Solutions

An alternative solution that eliminates the need for retransmission and allows receivers to access data asynchronously is the *broadcast* solution. In this approach, the server repeatedly loops through the transmission of all data packets in the file. Receivers may join the stream at any time, then listen until they receive all distinct packets for the file. This approach has some obvious disadvantages, including:

- **No Download On Demand:** Users can not initiate downloads at their discretion, rather they must wait and possibly receive unnecessary repititions until they finally get the data they want.
- **No Resumed Downloads:** Users can not continue downloads seamlessly after temporary interruptions. They have to wait for the receiver to cycle through the entire file.
- **Packet Loss:** Packet loss is a major problem. Since there are no individual streams or retransmissions, if you lose a packet you have to wait till the next cycle to get it.

Despite these shortcomings, the approach offers some advantages:

- **Low Server Load:** The server only handles a single stream, and continuously cycles through it. It does not have to handle a stream per connection: there is no state per user, nor are there any retransmissions. Thus server load is nominal.
- **Low Network Load:** There is low load on the network as well, since there is only one copy of the stream.
- **Scales Well:** Since new users don't add any overhead, and there are no retransmissions, the approach is highly scalable.

13.3 Coding Solutions

Clearly, what is needed is a way to combine the advantages of both paradigms - while mitigating their shortcomings. This can be accomplished by using techniques from coding theory.

We all think of data as a series of ordered packets. A message is taken, broken into small sequential chunks and transmitted over the network to the receiver, where it is re-assembled and finally read. There are two fundamental sources of error in this process: packet loss, and corruption.

In conventional protocols, each packet has a distinct header that identifies its position within the original file. This header allows the receiver to reassemble the file in the correct order, and to detect missing packets. Each packet is also equipped with a standard checksum which allows the receiver to detect corrupted packets, which are then dropped and treated as missing.

13.3.1 Dealing with Packet Loss: Forward-Error-Correction

As we have seen, traditional protocols such as TCP deal with packet-loss by requesting the sender to retransmit the missing packets - an approach which can quickly become unscalable. An alternative solution is to attempt to *reconstruct* the original data in the face of losses.

This is the idea behind mechanisms such as Forward-Error-Correction, which aim to recover from corruption by adding *redundancy* to the original information. The idea is simple: transmit the original source data, in the form of a sequence of packets, along with additional, redundant packets. This redundant data can be used to recover lost source information at the receivers.

For example, consider transferring a file through a network facing severe capacity constraints. In our troubled network, one out of every nine packets is dropped. A simple way of addressing this is as follows:

1. Divide the file into *blocks* of eight packets each.
2. Append to each block a ninth packet that contains the XOR of the eight original packets

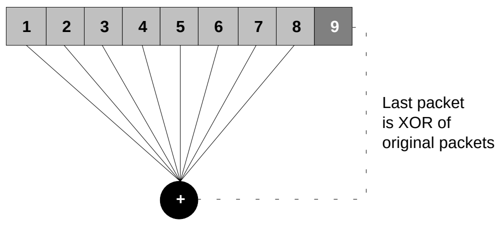

Figure 13.1. A Simple Parity-Check Code

The payload of the ninth packet is *redundant*, since it can be derived from the other eight. Assume, now, that one of the packets is lost. If this happens to be the ninth packet, nothing needs to be done - it was redundant anyway. If, however, it was one of the original packets it can be reconstructed simply by taking the XOR of all received packets[2]. This simple XOR 'code' is called a parity-check-code. In general, the class of Eraser Codes allow recovery of data given incomplete information: we can recover from errors without having to request retransmissions.

Issues with FEC

Note that our simple scheme fails if more than one packet is dropped in a group of nine. In general, more sophisticated FEC schemes add redundant packets to original ones in the hope of recovering information in the face of losses: in doing so, they must strike a delicate balance. On the one hand, adding too much redundancy wastes resources such as bandwidth and CPU computation time. On the other, adding too little may make recovery in the face of corruption impossible.

Finally, the encoding and decoding times for traditional FEC schemes quickly become unmanagable for all but the smallest segments of data. This has severely limited the scope of such schemes in the past.

13.3.2 A Digital Fountain

Using a recently developed class of codes known as Tornado Codes, it has become possible to add redundancy to the original data in a manner such that the receiver can reconstruct the original data after receiving a sufficient number of packets - regardless of order or sequence.

In other words, we can think of data as drops of *water*: you don't care which drops you get, and you don't care if you lose some drops - as long as you get enough drops to fill your cup[3]. Using such a coding scheme, as long as we receive a sufficient number of packets - any packets - we can reconstruct the original file. One could even get these packets from multiple sources, without having to worry about arranging coordination between them.

[2]Note that the XOR of a packet with itself is zero

[3]We can also view these encoded packets as random linear equations over the original content. Once we have enough independent equations, we can solve for the original file.

This attractive scheme relies on two fundamental properties made possible by Tornado Codes:

- We can take a file of n packets, and encode it into cn encoded packets.
- Given *any set* of n encoded packets, the original file can be recovered[4]. This means that encoded packets are completely interchangable: it doesn't matter which packets the client receives, and in what order: once a sufficient number of packets have been received the original content can be recovered. Thus, if some encoded packets are lost in transit, any equal amount of encoded packets is just as useful for reconstructing the original file.

Given these two properties, we can combine the advantages of the two traditional methods of content delivery - while avoiding their limitations. Consider a system that sits in a loop, continuously transmitting a stream of encoded packets. Clients come in and receive these packets, and as soon as they have heard enough to recover the file, they disconnect.

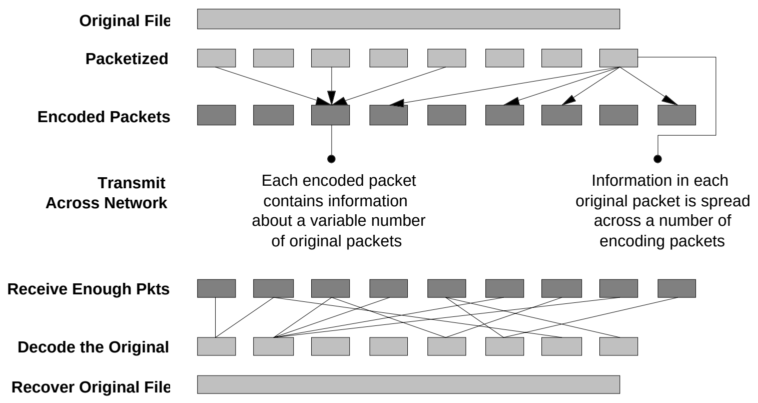
Original File
Packetized
Encoded Packets
Transmit Across Network
Each encoded packet contains information about a variable number of original packets
Information in each original packet is spread across a number of encoding packets
Receive Enough Pkts
Decode the Original
Recover Original File

Figure 13.2. Generation of the Encoded Packets

Unlike traditional broadcast solutions, this ensures that content is available *on demand.* When a user requires data, he simply connects to the server, receives n packets, and disconnects. The server does not need to create a new connection for each individual user, it just continues to serve the same stream.

Downloads can also be resumed seamlessly. If a user gets disconnected after receiving only $k < n$ packets, he can simply reconnect, get any $n - k$ packets and complete his download. Packet loss, too, is no longer a problem: if a packet is dropped, the next one can simply take its place.

This scheme combines the low-load and scalability of the broadcast solution with the reliability and flexibility of the point-to-point solution, and is thus very attractive.

[4]In reality there is some overhead to every coding scheme, we would actually require $(1 + \epsilon)n$

Issues with Codes

Encoding and decoding tend to be slow, time-consuming processes. While the file only needs to be encoded once on the server, decoding is problematic, since each client must individually decode the file on their home machines. There are two metrics generally used to measure the efficiency of coding schemes:

- **Time Overhead:** The time to encode and decode, expressed as a multiple of the encoding length.
- **Reception Efficiency:** The ratio of packets in the original message to the packets needed to decode the message. The optimal is 1.

There is an engineering trade-off between these two metrics. Codes that exhibit optimal reception efficiency - you only need to get exactly n packets to recover a file n packets long - tend to be slow. A famous example are the standard Reed-Solomon codes - their running time is quadriatic in the length of the message.

Tornado Codes

By slightly relaxing our demands on reception efficiency, we can decode the file far more quickly. We buy this speed at a price - instead of needing only n packets to reconstruct the original file, we now need $(1+\epsilon)n$. Tornado codes exhibit exactly such properties: In order to decode a file of 100K, the user has to download 105K - but he can decode the file very rapidly. In fact, the time overhead for Tornado Codes is independent of the length of the file:

- Reception Efficiency is $1/(1+\epsilon)$
- Time Overhead is $\ln(1/\epsilon)$

13.3.3 Digital Fountain: How It Works

Digital Fountain, Inc is currently putting just such a scheme into practice, providing the ability to scale a single IP Multicast data stream to reach tens of thousands or even millions of users, without massive server or cache farms or Internet infrastructure upgrades.

A traditional FEC scheme takes a file and appends a number of coding packets to it. In contrast, the Fountain scheme can generate a potentially infinite number of such coding packets for a single file. The original file can be reconstructed using *any combination* of a sufficient number of these packets - and the process is fast and efficient. A server that is hosting a file generates a continuous stream of such encoded packets. Users connect to the server, hear a sufficient number of packets, and disconnect. There are no retransmissions nor any need for coordination. Indeed, downloads can be performed in parallel if multiple servers are available. The codes used are Tornado codes, and the coding technology that generates the infinite stream of unique packets is called the Luby Transform.

Digital Fountain refers to each encoded packet as Meta-Content$^{\text{TM}}$. The file to be served passes through an engine, which encodes it into a stream of Meta-Content$^{\text{TM}}$packets. Note

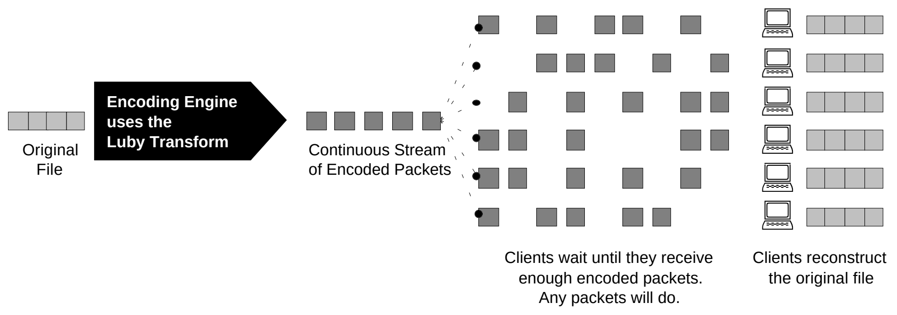
Original File
Encoding Engine uses the Luby Transform
Continuous Stream of Encoded Packets
Clients wait until they receive enough encoded packets. Any packets will do.
Clients reconstruct the original file

Figure 13.3. The Digital Fountain Scheme

that each packet is independently useful - ALL clients requesting the data can use it. It may be useful to think of each packet as a random linear equation that contains some information about the file. Each equation provides additional information, and one equation is as good as another. Once a sufficient number of these equations have been collected, one can solve for the original content.

Clients on a broadcast or multi-cast network can receive these packets directly. Unicast clients, however, must use an independent UDP stream to get these packets. Note that there is no need to use TCP, since packet-loss is now irrelevant - once enough packets are received, the file will be complete[5]

13.4 Bloom Filters

We now move to a new topic. Imagine that you are connected to Napster, and wish to download a song. For you to do so, you need to know which computers are currently sharing it. This, in fact, is a very common network task: discovering which pieces of data reside at a given node. The most common way of dealing with this is to exchange complete lists that summarize each node's holdings. Such lists can be extremely long and unwieldy, using up a lot of network bandwidth and slowing down transfers.

An alternative way of answering such queries is to use a data structure called the *Bloom Filter*. This provides small, approximate lists that can answer such queries with a low probability of error.

More formally, we can express the problem as follows:

Given a set $S = \{x_1, x_2, x_3 ... x_n\}$ on a universe U, we wish to answers queries of the form: Is $y \in S$?. For example, we may be considering a set of songs - say everything by the Beatles - from the universe of all possible song titles. Such queries can be answered by using a Bloom Filter in:

[5]Note that an independent stream still has to be generated for each Unicast client, except that it uses UDP instead of TCP. Thus server load is reduced because there are no retransmissions or state, while network load is reduced since UDP has less overhead. The use of broadcast or multicast with this technology, is obviously more efficient.

- Constant time
- Small space
- But with some small probability, p of being wrong

13.4.1 The Data Structure

Given the set $S = \{x_1, x_2, x_3 ... x_n\}$ we create a bloom filter, B that represents the set as follows: We begin with an m bit array, filled with 0s:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

We hash each item x_j in S, k times using different hash functions each time. We use the result of each hash as an *index* to the array, and set the bit at that index to one. Formally, if the result of a hash $H_i(x_j) = a$, then we set $B[a] = 1$. After doing so for each item, using each hash function, our array looks like this:

0	0	0	1	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0

To check if y is in S, we hash y using each hash function, using the results to index into the array as before. If, at *each* position we find a 1, we conclude that y is indeed in S. Otherwise it definitely is *not*. Formally: $y \in S$, if $B[H_i(y)] = 1$ for all i such that $0 < i \leq k$

0	0	0	*1*	0	*1*	0	0	*1*	0	0	0	0	0	*1*	0	0	0	0	*1*	0	0

Note that each hash function can point to an entry containing 1, even though y is not actually present in S. It is thus entirely possible to have false positives. False negatives, however, are not possible: if y is in S, the query will always return true. Errors in a bloom-filter, thus, are *unidirectional.* This lends to their use as queries: if a node has data that you want, you will always access it. You only run the risk of checking and finding that it is not there. Such false-positives, however, are not particularly troublesome: they can result from cache changes and old lists anyway.

13.4.2 Optimal Bloom Filters

An ideal bloom filter is one that minimizes the probability of error. Given m bits for the filter, and n elements in the set S, we wish to choose the number k of hash functions to minimize false positives[6].

- Let p be the probability that a cell is empty. Since the hash is random, the probability that a given hash function maps to a particular cell is evenly distributed, in other words is $1/m$. Thus the probability of a given hash function *not* hitting a particular cell is $(1-1/m)$. Since we have k hash functions for each of n elements, the probability that a cell is empty, $p = (1 - 1/m)^{kn} \approx e^{-kn/m}$

[6]We assume that our hash functions look truly random.

- Let f be the probability of a false positive. This is the probability of the cell not being empty for *all* k hashes. Thus $f = (1-p)^k \approx (1-e^{-kn/m})^k$

Recall that a false positive can occur if each hash function maps y to a cell containing 1, even though y does not actually belong to the set. An optimal bloom filter is one that minimizes the chance of this happening. Consider an element y that is *not* a member of the set S. As the number of hash functions, k, increases we have a greater chance of running into a 0 for at least one of them; this would tend to *reduce* the probability of false positives. On the other hand, more hash functions simply mean more 1's in the array[7] - which tends to *increase* the probability of error.

In fact, calculus tells us that the optimal bloom filter uses $k = (\ln 2)m/n$ hash functions.

13.4.3 Compressed Bloom Filters

Our original motivation was to exchange compact filters between network nodes, instead of lengthy lists. Within this context, we know that a Bloom filter is not just a data structure: it is also a *message*. Since transmitting any message consumes bandwidth, it may be worthwile to compress it.

From one perspective, a Bloom filter looks just like a bit vector, and compressing such vectors is easy. Arithmetic coding, for example, is a particularly well-suited technique for compressing such strings. The more fundamental question is can Bloom filters be compressed at all?

Optimization & Compression

We know that:

- The probability of a cell being empty, $p = (1-1/m)^{kn} \approx e^{-kn/m}$
- The probability of a false positive, $f = (1-p)^k \approx (1-e^{-kn/m})^k$
- The optimal number of hash functions, $k = (\ln 2)m/n$

But at this optimum $k = (\ln 2)m/n$, and substituting back in, $p = 1/2$ - the probability of a cell being empty is the same as of it being 1. This would seem to imply that the Bloom filter is a true random string, which as we all know, cannot be compressed. This, however, is a fallacy.

With compression, the optimal number of hash functions, k *changes*. The original trade-off involved only three parameters:

- Size: m/n bits per item
- Number of hash functions: k
- False positive probability: f

[7]Recall how Bloom filters are set up. Each hash function contributes a '1' for each element in the set.

With compression, however, there are now four parameters to optimize over:

- Compressed (transmission) Size: z/n bits per item
- Decompressed (stored) Size: m/n bits per item
- Number of hash functions: k
- False positive probability: f

Transmission cost, which depends on the *compressed* size of the filter is what is important. The storage size, which is what our previous optimum was based on, is no longer as relevant. Machine memory is cheap and readily available: bandwidth is not. Our goal, thus, is to reduce the false positive rate by increasing the decompressed size, while keeping the transmission cost constant.

Optimal Compressed Filter

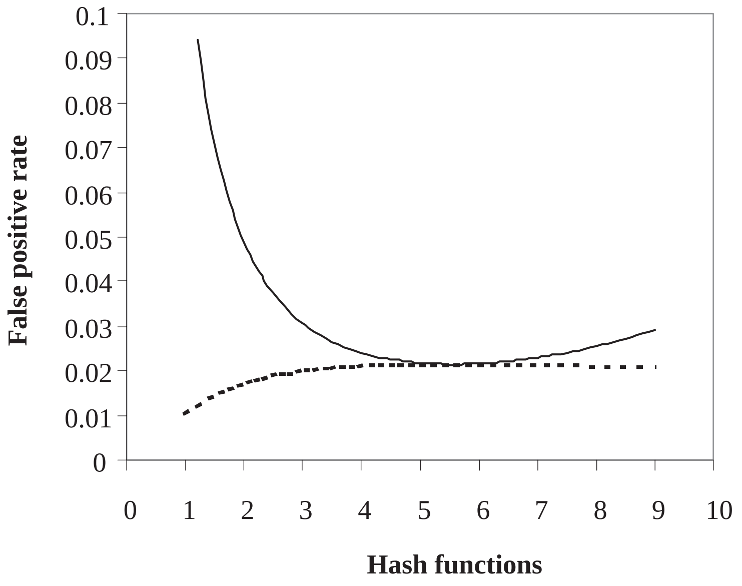

Figure 13.4. False positive rate as a function of k when $z/n = 8$

The graph above depicts the false positive rate as a function of the number of hash functions used. The solid curve represents the tradeoff in the original, uncompressed bloom filter, while the dotted line represents the tradeoff for a compressed filter. We see from the graph that at $k = (\ln 2)m/n$, false positives are minimized for an uncompressed filter. However, at this value of k, they are actually *maximized* for a compressed one. The best case without compression is the *worst* case with compression: compression always helps!

Note that using a compressed filter reduces the false positive rate *while using fewer hash functions*. Since computing hash functions for each query takes up time, this may also significantly improve performance.

Array bits per element	m/n	8	12.6	46
Trans. bits per element	z/n	8	7.582	6.891
Hash functions	k	6	2	1
False positive rate	f	0.0216	0.0216	0.0215

Table 13.1. Improvement with Compressed Filter given a Fixed Probability Rate

We can see from the table that a compressed filter can reduce transmission size from 8 bits/element to 7.5 bits/element, at the cost of increasing the uncompressed size to 12.6 bits/element. In doing so, we also manage to reduce the number of has functions needed from 6 to 2 - while keeping the false positive probability completely unchanged.

Array bits per element	m/n	8	14	92
Trans. bits per element	z/n	8	7.923	7.923
Hash functions	k	6	2	1
False positive rate	f	0.0216	0.00177	0.0108

Table 13.2. Improvement with Compressed Filter given a Bounded Transmission Size

In this table, we can see how increasing the uncompressed size while keeping the transmission size bounded improves the false positive rate. For example, increasing the uncompressed size from 8 to 14 (while the compressed size remained constant at 8) reduced the error rate considerably - while cutting the number of hash functions used by a third.

The moral of our story, then, is clear: bloom filters should be compressed whenever possible.

13.5 Bringing It All Together: Informed Content Delivery

Imagine nodes in a network downloading the same file from a source. Assuming that a pair of nodes have not received *exactly* the same content, they should be able to benefit from this fact. Each can do better by sharing his data with his neighbour and combining the two data-sets intelligently. This simple idea is called Informed Content Delivery.

The essential idea is that on a multicast, or overlay Peer-to-Peer network, there are often untapped communication paths available for data to traverse. Nodes can use these additional paths to share data amongst each other and benefit from the exchange. For example, my neighbour and I may be missing different parts of a file if packet losses are independent. By sharing or *reconciling* information with each other, we can each reconstruct the complete file.

Reliable multicast uses tree-based networks. The diagram depicts such a logical tree, the links involved, and the additional paths that can be exploited.

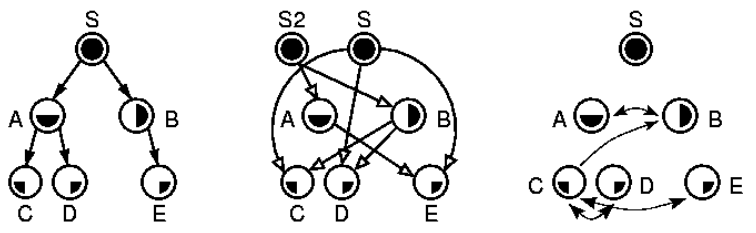

Figure 13.5. Taking Advantage of Extra Communication Paths

Though the idea behind Informed Content Delivery is simple, exchanging information between additional links consumes bandwidth. As such, the idea will be more applicable in the future, when bandwidth will be cheap and the goal will be to receive data as quickly as possible.

13.5.1 Challenges For Informed Content Delivery

The Internet is a highly mutable environment. Connections are continuously being made and broken, and their speeds and loss rates vary widely. Sessions are preemptable, and at any given moment routers, hosts and links can come up or go down. Couple this with an enormous client population, and the challenges faced by Informed Content Delivery seem almost intractable.

To complicate matters, adaptive overlays that build on top of the Internet work by reconfiguring network topologies and can actually exacerbate these problems. In the face of all this, what is to become of our idea that two neighbours with disparate data-sets can benefit each other? Any scheme that purports to do so must be able to cope with frequent reconfiguration, provide preemption support, and be scalable.

13.5.2 The Digital Fountain Solution

The Digital Fountain provides just such a scheme. It is:

- **Stateless:** Servers can produce encoded packets continuously.
- **Time-Invariant:** Encoding is memoryless - one packet is as good as another
- **Tolerant:** The scheme can tolerate different client speeds and network characteristics - all that matters is water in your cup.
- **Additive:** Fountains can be accessed in parallel - data from one is as good as data from another.

Using the Digital Fountain scheme, the fluidity that was a problem for Content Delivery can become an asset. The ever-changing environment of the Internet results in different working-sets of data, even amongst peers receiving identical content. The receiver with the higher rate, or the one that started earlier will have more content. Similarly, receivers suffering from uncorrelated packet losses, will have gaps in *different portions* of their working sets.

This means that there is an opportunity for peers to exchange fountain packets: the ones they have not received can be put to use. This however, presents a challenge. Reconciliation using traditional, ordered sequential packets is relatively simple. You connect to your neighbour and ask him for packets that you are missing. In the fountain scheme, each node has a set of packets from a potentially infinite pool of unique packets. Discovering which ones your neighbour has that are of actual use to you is quite a knotty problem.

The Alexandrian solution is to use our old friend, the Bloom Filter. We can coordinate between peers by exchanging a (compressed) Bloom filter that represents a list of all encoding packets that a node has. The neighbouring node receives the filter, and can start sending all the packets its peer did not possess. The occasional false positive is not harmful, since coding already contains redundancy. Besides, you want useful packets as soon as possible, and an occasional extraneous one does no harm.

As we have discussed, Bloom filters require only a small number of packets, thus nodes can be kept appraised of their neighbours holdings at minimum cost. If it is known that the number of discrepencies between two nodes is limited, then enhanced data structures called Approximate Reconciliation Trees can be used. These structures are also based on Bloom Filters[8].

There are many other potential applications that can exploit the unique features of the Digital Fountain approach to data delivery. The Informed Content Delivery described above is only one of them, and the area is an active field of research.

[8]These reconciliation techniques can also have other practical uses in applications such as databases, or for synchronizing handhelds. In deciding between the approximate reconciliation provided by Bloom Filters, and the exact reconciliation provided by lists, the main criterion remains communication complexity and the cost of bandwidth.

Bibliography

[1] Digital Fountain White Paper: “Digital Fountain’s Meta-Content Technology - A Quantum Leap in Content Delivery” 2001, Fremont, CA.

[2] Byers, J; Luby, M; Mitzenmzher, M; & Rege. A. “A Digital Fountain Approach to Reliable Distribution of Bulk Data.”, SIGCOMM pgs 56-67, 1998.

[3] Mitzenmacher, Michael. Presentation: “Codes, Bloom Filters, and Overlay Networks.” 2002, Cambridge, MA.