EECS 121: Coding for Digital Communication & Beyond **Fall 2013**

Lecture 22 — December 3

Lecturer: K. V. Rashmi *Scribe: Ajay Shanker Tripathi*

Introduction

22.1 Context

Distributed storage is a deeply relevant and rapidly-growing field. In this new age, cloud storage and parallel computing on multiple servers are incredibly important.

22.2 The Problem

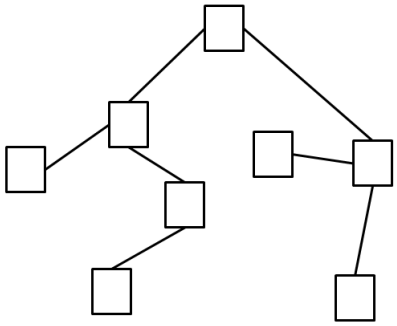


Figure 1: a sample distributed network. Each box represents a node, or machine, which stores some data. The lines represent links between nodes across which information can be sent.

A large scale distributed storage system may consist of multiple thousands (e.g., 5k-10k) of data "nodes" (machines), which collectively store tens (or even hundreds!) of petabytes of data. Each individual machine is unreliable because getting so many reliable machines wouldn't be nearly economically feasible. Time to time, several machines can go down, so one must deal with failure as a common occurrence, as opposed to a rare one. How can we ensure that failures can be recovered from? That is to say, how can we ensure that data is lost and unrecoverable with extremely low probability? The answer that should come to mind is that we should add some redundancy somehow. So how can we add this redundancy, and how effective and efficient will it be?

Different Methods of Adding Redundancy

22.3 Method 1: Replication

This is the most obvious form of adding redundancy: simpy store the same file to multiple nodes. How effective is this? Let's look at a simple example.

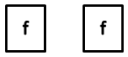


Figure 2: one file f stored on two nodes

Consider the system in figure 2, where we have some file f stored on two machines for reliability. Let each machine have a probability of failure of p. Then what is the probability that the file is unrecoverably lost? The answer is p^2 (both nodes must be lost). Furthermore, this system can withstand any 1 machine failure.

22.4 Method 2: Splitting the file and replicating the parts

A little more clever is to split up the file you're trying to store into separate parts, and try to protect those separate parts by replication. The idea is that we are trying to spread out the damage that a machine failure can inflict. Again, we analyze a simple example.

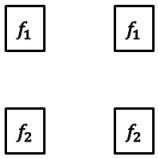


Figure 3: the two parts of f, f_1 and f_2, stored twice on two different machines each

In the example in figure 3, we take the same file f, and we split it into two files f_1 and f_2. Each of these parts are replicated to be stored on their own machine. Note that the same amount of data is being stored, just across 4 nodes instead of 2. Again, let each machine fail with probability p. What is the probability the file is unrecoverably lost? After a bit of work, one should arrive at $1 - (1 - p^2)^2$ (each file part must survive for the whole file to survive). Observe that this system can withstand any 1 machine failure as well, though it can handle up to 2 machine failures for particular combinations of failures. Still, by creating more ways in which the file can be lost, the probability of the file being lost is worse in this system, as shown in figure 5. However, this is a key step to making the problem admit more clever solutions, as we'll see next.

22.5 Method 3: Coding

Finally, we reach a method worthy of EE 121. The previous methods are forgetting one very important fact: we are dealing with information. Information isn't like physical baseballs. Information can be mixed and coded together, and as we've seen before, information is stronger with solidarity when mixed and combined together. Let's analyze the following example of a coded distributed storage scheme.

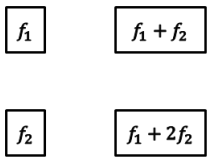


Figure 4: the two parts of f, f_1 and f_2, are stored in linear combinations on two of the nodes. Here, all operations are performed modulo the appropriate finite field size (e.g., mod 8 for $|f_1| = 1$ byte).

In the example in figure 4, we take the two parts of the file, f_1 and f_2, and attempt to mix them as linear combinations on two of the nodes. Again, the same amount of information as Method 1 is being stored in total. Here, however, we are taking advantage of the fact that information can be coded. How good does this system work? First off, note that any 2 machine failures can be tolerated, which is a cut above the other two methods. After a bit of analysis, one may find the probability of error to be $4p^3(1-p) + p^4 = p^3(4-3p)$. As can be seen in figure 5, this performs considerably better than the rest. One particular feature of this particular scheme to take notice of is that it is in "systematic form". This means that the actual files (in this case, f_1 and f_2) exist somewhere in uncoded form, so that the whole file can be downloaded and reconstructed without any decoding necessary. This systematic form is highly desirable for a system to avoid computation during the usual read operations.

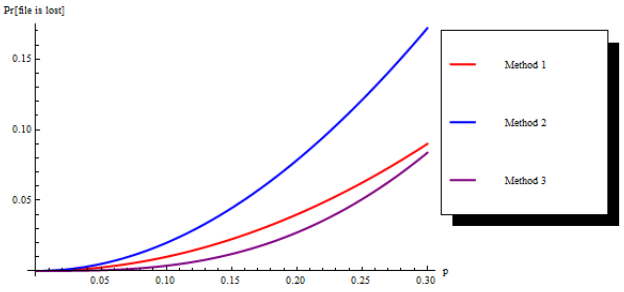


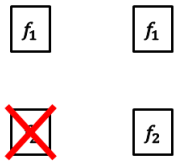
Figure 5: A comparison of failure probabilities. Note that Method 1 performs better.

It can be seen fairly simply that coding can do better than repetition with even less overhead. For example, consider a file f split into 7 parts $f_1, \ldots, f_7$. If you compare 3 times repetition to an $(n = 14, k = 7)$ RS code, it becomes obvious how much better coding is. Repetition would use a 3 times redundancy overhead, whereas the RS code would only use

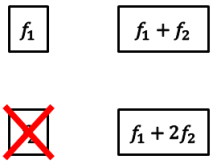
2 times redundancy to yield better protection. Analyzing the probabilities of error would also yield further proof of coding's superiority. So does this mean we're done? Simply using RS codes would solve our distributed storage problem? Well, the answer is no. Distributed storage is a special type of protection problem that requires more than being fail-safe, and hence we need to go beyond the standard RS codes, as we will see in the next section.

Central Distributed Storage Problem: the Cost of Repair

To introduce this special problem unique to distributed storage, let's consider two of the previous schemes.

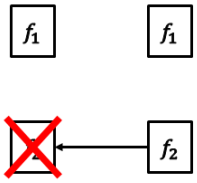


Method 2

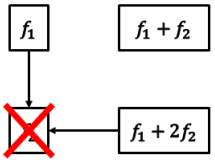


Method 3

As previously noted, method 3 completely outperforms method 2 in terms of reliability. But now, let's analyze what the cost of a machine failure is. Say the bottom-left node which contains f_2 were to go down, and is in need of recovery. How would the systems for each method repair the data in the failed node?



Method 2



Method 3

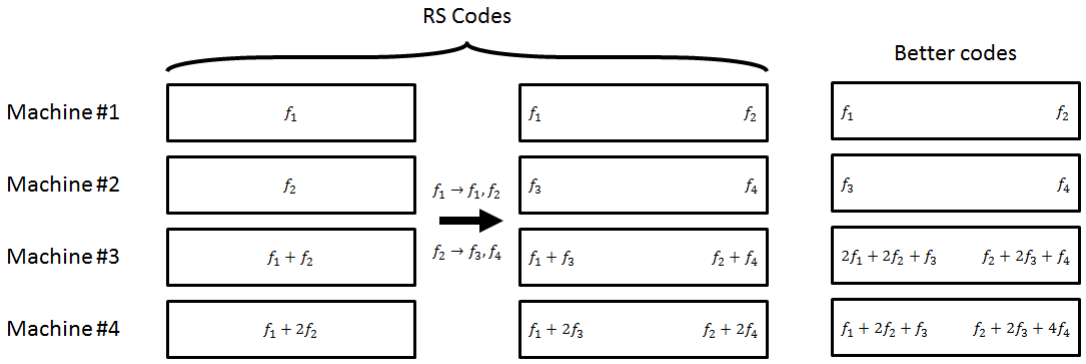
For method 2, the failed node need simply download the data from one machine, its neighboring f_2 node. For method 3, however, the failed node needs to download data from 2 machines to reconstruct its data. This highlights one of the problems in distributed storage: repair cost. Even though method 3 is more reliable, method 2 has less download overhead for repairing, and depending on the system, the download for repair is a precious resource that one wants to optimize.

To further show how big a problem the cost of repair can be, consider an (n, k) RS code. Recall that these RS codes provide much better protection over straight repetition. However, consider the cost of repair. If any of the n nodes fail, they have to download the data from k nodes in order to reconstruct their own data. That is to say, a node has to download the entire file to repair its small share of the file! This is obviously a large overhead that real systems have been facing.

We saw measurements from the data-warehouse cluster at Facebook from Reference 1 (see end of notes for references), showing the impact that coding has on network bandwidth consumption. It seems that we need better codes than RS to achieve the same reliability, but with much better repair costs. But how can we do this? Is it even possible? And if so, how good can we go?

22.6 Codes that Consider the Cost of Repair

It turns out that it is possible to get method 3's reliability but for much lesser cost of repair. We can see this in the following example.



In this example, we have a file f stored on 4 machines. On the very left, we see our old RS scheme with f split into f_1 and f_2 and each machine can only hold 1 unit of information. For the sake of understanding the new codes, we will look at a slightly modified RS scheme where f is split into $f_1, \ldots, f_4$, and let each machine hold two units of information. You can see the modified RS codes are very much like the original ones, with $f_1 \to f_1, f_2$ and $f_2 \to f_3, f_4$. You will notice that the same properties hold for these RS codes. Any two machines can fail and all four units of data that comprise f can be recovered (check to convince yourself of this). Furthermore, if a machine dies (like machine #1), then 4 units of data must be downloaded to repair the 2 units of data stored on the machine (i.e., the entire size of f must be downloaded to repair a machine). However, now let's look at the alleged better codes on the right.

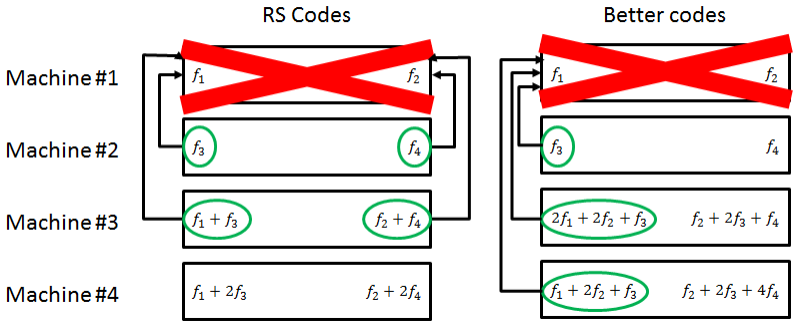


Figure 6: An example showing better repair-cost than RS codes for the same fault-tolerance.

Looking at the new codes, you should see that they have the same fault-tolerance as the RS codes: any 2 machine failures can be recovered from. However, looking at the repair cost, one sees that only 3 units of data need to be downloaded to repair all 2 units of data

stored in any one failed machine, as can be seen in figure 6. So this gives us some hope: it is possible to acheive the same reliability with greater effeciency in repair cost. All we need is better codes. Before we tackle what kind of code constructions we can use, we should first ask the question: what's the best we can do in terms of downloads needed to repair?

22.7 The Limits of Codes for Distributed Storage

Let's analyze the general problem to find the hard bounds on the best we can hope to do in terms of repair cost. These bounds were derived in Reference 2 (see end of notes for references). The construction for the problem is as follows.

There are n nodes (machines). Each can store α units of data. The total size of the file being stored is B units. We want two properties from this system:

1. Data-Collection (Reconstruction in the original paper): Connecting to any k nodes is sufficient to recover all B units of data (RS property).

2. "Repair" (Regeneration in the original paper): A single node can be repaired by connecting to **any** d other nodes and downloading β units of data from each.

To give a quick example of these parameters, here's what they are with respect to the "better codes" in the previous section: There are $n = 4$ machines that store $\alpha = 2$ units of data each to store a file of size $B = 4$ units. Connecting to any $k = 2$ nodes is enough to collect all the data, and connecting to any $d = 3$ nodes and downloading $\beta = 1$ units of data is enough to repair any failed node. There are some observations and constraints we can make from our parameters.

1. $k\alpha \geq B$: One can't recover the whole file unless they download at least the size of the file.

2. $d\beta \geq \alpha$: One can't recover all the data on a node unless one downloads at least that much data.

3. $d \geq k$: If this were not true (if say, $d = k - 1$), then one need only connect to $k - 1$ nodes to reconstruct all the data (repair a k^{th} node and then you have k nodes which give you all the data by the RS property). But this contradicts the fact that you should need at least k nodes to reconstruct the data.

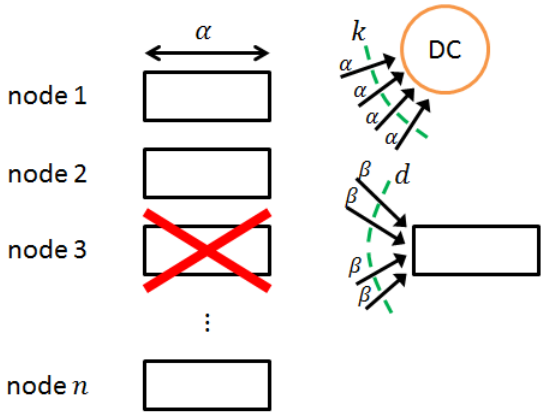


Figure 7: A depiction of the general problem. "DC" means data collector, a unit trying to get the entire file.

One last point to note is that, in our analysis, we have a relaxed constraint for repair that we only need a fundumentally equivalent file, as opposed to the exact same one. For example, if we have the files f_1, f_2, $f_1 + 2f_2$, $f_1 + 3f_2$ and f_1 gets erased, then it can be repaired as $f_1 + f_2$, and the system will still be effectively the same with the same data recoverable. The reason for this relaxed constraint is that the analysis we will use doesn't easily support exact reconstruction. Furthermore, because we are looking for an upperbound, it's alright to relax constrainsts a little.

With the problem now set up, let's look at a particular example: we will use an information flow graph to represent a distributed network system with $n = 4$, $k = 2$, $d = 3$, and a general α, β and B.

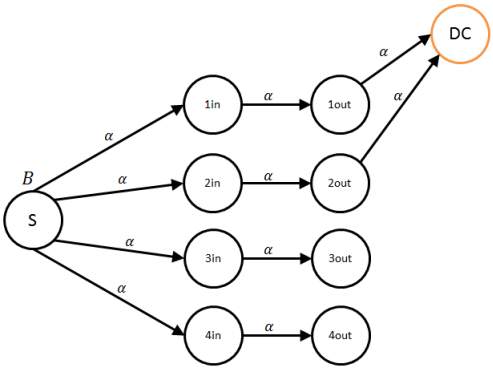
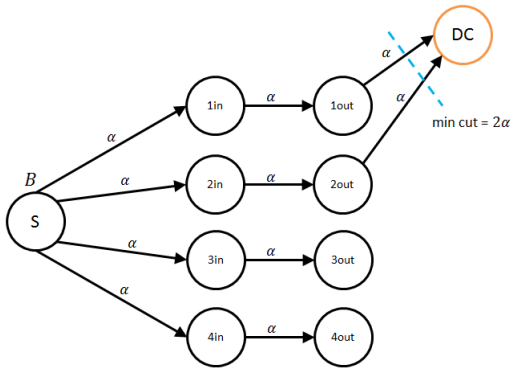


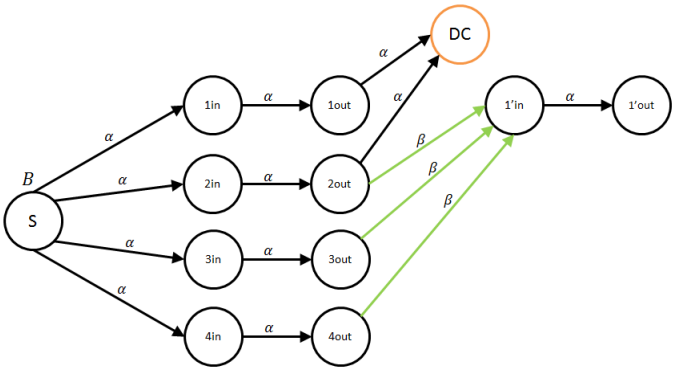
Figure 8: An example scenario of what might happen in a distributed storage system. $i$in $\to$ $i$out is how a node is represented in this networking style. S is the original source file.

Figure 8 shows an example scenario of something that could happen in a distributed storage system. An initial source file (represented as the S node) of size B gives α units of data to 4 different nodes, and then a data collector comes along and reads the contents

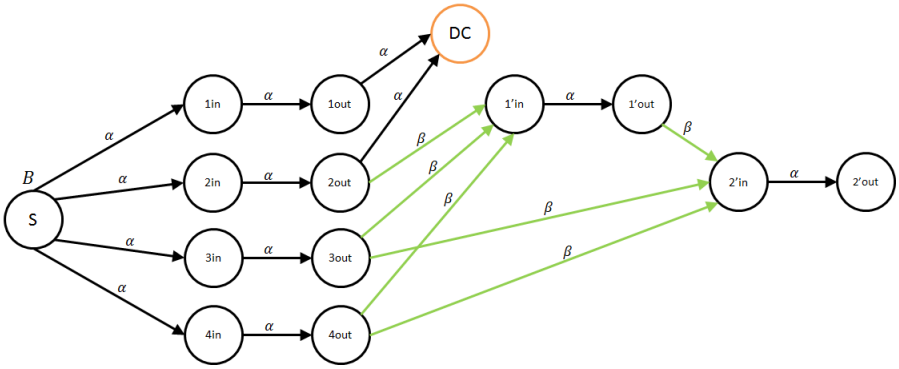
of two of the nodes to get the file of B units. Now notice the networking style that this scenario was represented with. What is the maximum amount of data that can flow through in this scenario? Well, the min cut is 2α, right before the data collector. So the maximum file size B is 2α. Any larger than that, and the network simply wouldn't allow that much information to flow through.



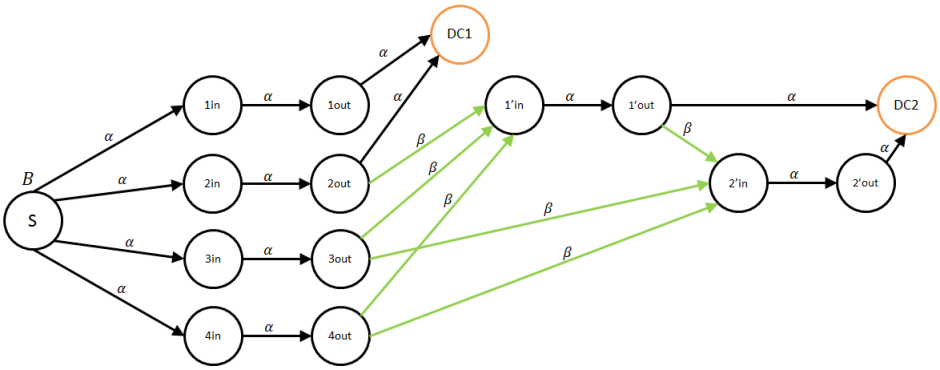
So what else could happen in this system? Well, one node can fail, and a new node would have to be constructed through a repair process of connecting to $d = 3$ nodes and downloading β units from each.



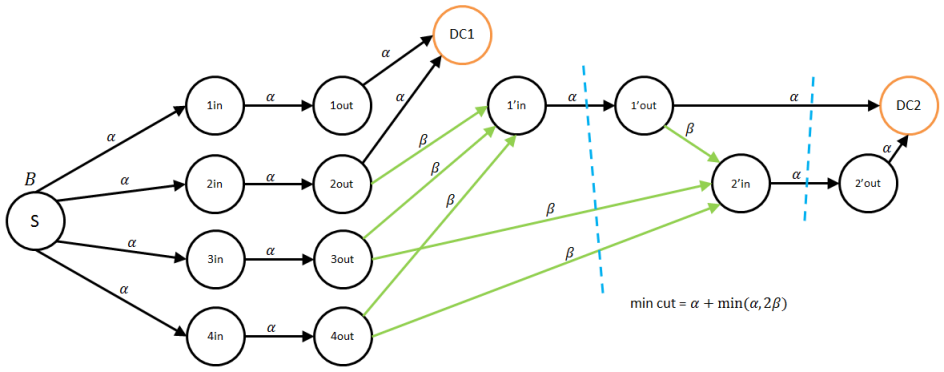
Next, let's let another node fail and require repair.



Lastly, let's have a data collector come in and try to aquire the file from our two probably weakest nodes, the two that had to be repaired.



Again, we analyze what is the maximum amount of information that can flow through this network, and use this to find a bound on B. If we look through the graph, we see that there could potentially be two min cuts. Again, we have the cut near the data collector of size 2α. But there is also a cut of size $\alpha + 2\beta$. Because we don't know how α and β are relative to each other, we write down the general expression for the maximum flow: $B < \alpha + \min(\alpha, 2\beta)$. It turns out this is the tightest informational bound that can be found for this example.



Let's analyze why B got constrained beyond 2α to be bound by $\alpha + \min(\alpha, 2\beta)$. Essentially, one of the β's which was part of the 3β going into the 2' node was pushed onto the DC side of the cut, so only 2β was left to be part of the min cut. We will see the same thing happen when we look at the fully general case, and this will gives us the fully general bound. So let's try to see how this sort of thing comes into play for the case with general n, k, and d as well as general α, β, and B.

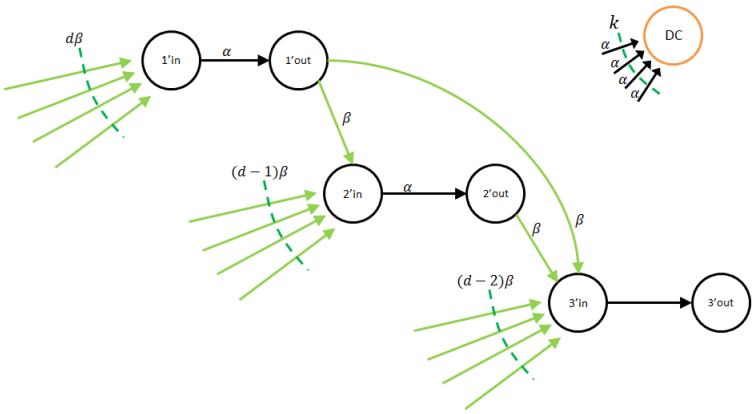


Figure 9: A slightly abstract view of a scenario where the min cut gets reduced

From figure 9, one should see that we can keep pushing an extra β onto the DC side of the cut if we want. So for the min cut, we could choose α or $d\beta$ (α always is the smaller of these two). Next, we could choose between α and $(d-1)\beta$. Then, we can choose between α and $(d-2)\beta$, etc.. This leads to the following formula for the mincut:

$$\text{min cut} = \min(\alpha, d\beta) + \min(\alpha, (d-1)\beta) + \cdots + min(\alpha, (d-(k-1))\beta)$$

$$\text{min cut} = \sum_{i=0}^{k-1} \min(\alpha, (d-i)\beta)$$

Recall that the min cut tells the maximum information that can flow, and we want this to be at least B. So this gives the final bound for the general problem.

$$\sum_{i=0}^{k-1} \min(\alpha, (d-i)\beta) \geq B$$

What insights does this final bound give us? Well, firstly, note that as α increases, B roughly increases. Also, as β increases, B roughly increases as well. This leads to what is called the "storage-bandwidth tradeoff". To store a file of size B, you can either prioritize α (the amount each node stores) and make β large, or prioritize β (the bandwidth or amount that must be downloaded for repair) and let α be large. This makes sense because if each node stores a large amount (α large), then it's easier to find the exact data you need to repair a node. If nodes store a minimal amount (α small), then a node seeking repair can't find exactly the data it wants in each node, and so must download redundant and not 100% helpful data.

The next observation is that, if $(d-k+1)\beta > \alpha$, then all the terms will be α anyways, so β is being wasted. Also, remember that $d\beta \geq \alpha$ because a node needs to download at least as much data as it has to be repaired. This tells us that the values we are about satisfy the bound

$$(d-k+1)\beta \leq \alpha \leq d\beta$$

Seeing how α is bounded leads to our next analysis. Let's represent α as $(d-p)\beta - \theta$ where $\theta \in [0, \beta)$ is the "remainder". And the values p can take on are $0, \ldots, k-1$. The maximum α has $p = 0$ and $\theta = 0$, where the minimum α has $p = k-1$ and $\theta = 0$.

The above representation is borrowed from Reference 3 (see end of notes for references). Using this representation, let's look once more at the bound we derived. $\alpha = (d-p)\beta - \theta$ is less than or equal to $(d-i)\beta$ for $i = 0, \ldots, p$. So the limit becomes

$$B = \underbrace{\alpha + \cdots + \alpha}_{(p+1) \text{ terms}} + \underbrace{(d-(p+1))\,\beta + \cdots + (d-k+1)\beta}_{k-(p+1) \text{ terms}}$$

$$B = (p+1)\alpha + f(p)\beta$$

Here, $f(p) = \sum_{i=p+1}^{k-1} (d-i)$ is some function of p. But the point is, the last expression shows that, for a fixed p and B, there is a linear relationship between α and β (the storage and the bandwidth). So to find the picture for the final bound, we pick p's from 0 to $k-1$, and vary θ from 0 to β eachtime, forming a piecewise linear graph, as shown in figure 10. The endpoint value where β is a minimum has "minimum bandwidth requirement" (MBR). The endpoint balue where α is a minimum has "minimum storage requirement" (MSR). Next time, we will discuss these points, as well as potential codes to acheive them.

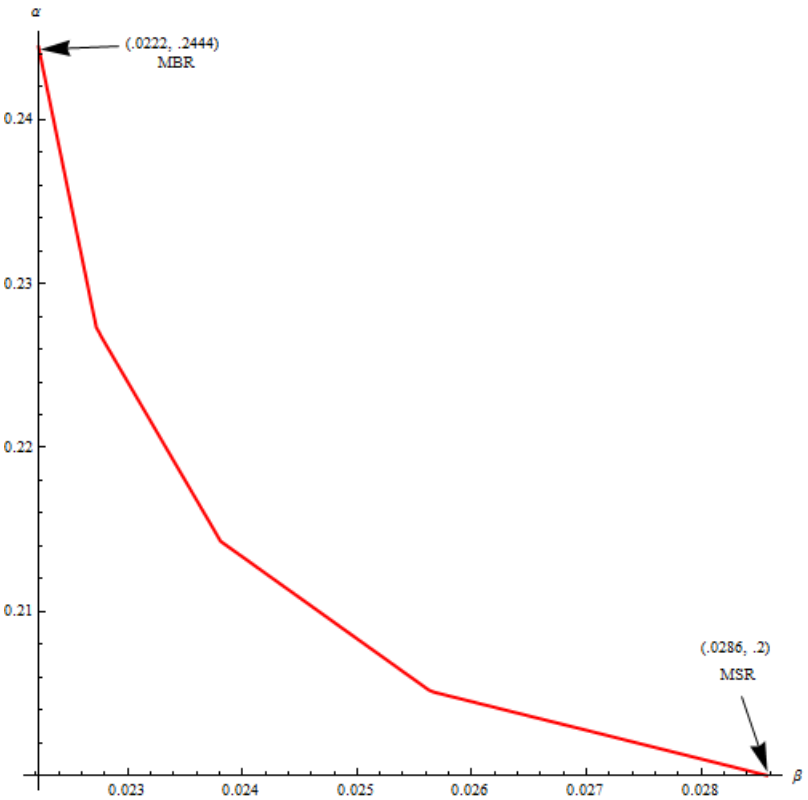


Figure 10: A plot of the bound for $k = 5$, $d = 11$, and $B = 1$.

22.8 References

1. "A solution to the network challenges of data recovery in erasure-coded distributed storage systems: a study on the Facebook warehouse cluster", Rashmi, K. V., Shah, N. B., Gu, D., Kuang, H., Borthakur, D., & Ramchandran, K. in Usenix Hot Storage, June 2013.
   (This paper presents measurements from the Facebook data-warehouse cluster regarding machine unavailability and the impact of coding on network bandwidth consumption. This motivates the problem of repair cost that we are studying in the next few lectures.)

2. "Network Coding For Distributed Storage Systems", Dimakis, A. G., Godfrey, P. B., Wu, Y., Wainwright, M. J., & Ramchandran, K. in IEEE Transactions on Information Theory 2010.

3. "Distributed Storage Codes with Repair-by-Transfer and Non-achievability of Interior Points on the Storage-Bandwidth Tradeoff", Shah, N. B., Rashmi, K. V., Kumar, P. V., Ramchandran, K. in IEEE Transactions on Information Theory 2012.