Paxson
Spring 2017

CS 161
Computer Security

Discussion 9

Week of April 3, 2017

Question 1 ***Lists and Trees of Hashes*** **(20 min)**

BitTorrent splits large files into small file chunks which are then transmitted between peers in such a way that each peer eventually ends up with the whole file. Commonly, chunks are of size 2^8 KiB = 256 KiB.

Because you cannot trust peers, you have to verify each chunk as you download them from a peer before you start providing them to other peers. Furthermore, you want to be able to do this as soon as possible and not wait for the whole file to be downloaded. You also want to be able to know which part of the file got potentially corrupted so that you do not have to re-download the whole file.

To achieve the above properties, BitTorrent uses a Torrent file. The file contains information describing the file (or files) to be transmitted, and their chunks. You must obtain this file from a trusted source.

(a) Initially, a Torrent file contained a list of SHA-1 hashes for each chunk. How large is such a list for a 10 GiB large file, if one SHA-1 hash takes 160 bits?

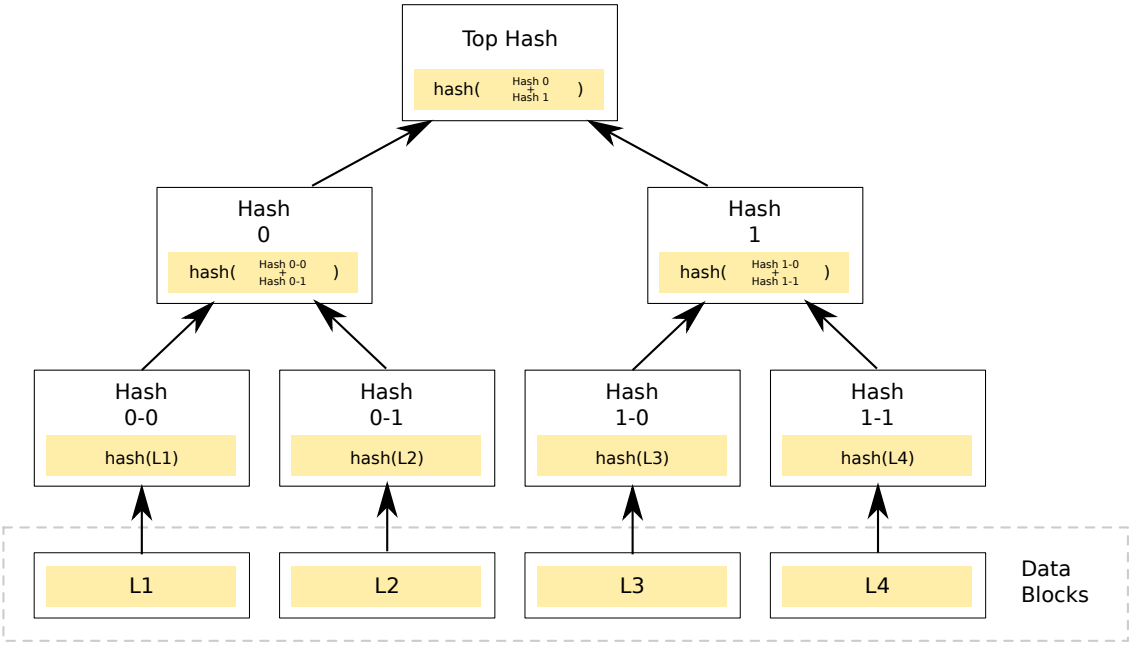
> **Solution:** Number of chunks: 10 GiB / 256 KiB = $10 \cdot 2^{30} / 2^{18} = 10 \cdot 2^{12} = 40 \cdot 2^{10}$
>
> Size of the list: $40 \cdot 2^{10} \cdot 20$ B = 800 KiB

(b) One way to make Torrent files smaller is to instead store only a hash of the hash list (top hash, or root hash) in the file and retrieve the hash list itself from a peer. Why would we want to make a Torrent file smaller? What is a downside of this approach?

> **Solution:** We want to reduce the size of Torrent files in order to minimize load on the servers hosting them.
>
> The downside is that we have to retrieve the whole list before we can verify any chunk, because we first have to verify the list itself against the top hash. But transmitting the hash list takes longer than transmitting the first chunk, since 800 KiB is larger than 256 KiB. Therefore, we cannot start providing the first chunk to others as soon as we have it. We also want to avoid transmissions larger than a chunk size.

Moreover, hashes are not very compressible, so compression does not help much.

(c) One approach to address the issue of the size of the hash list is to split it into chunks. However, you would then need a hash list of those chunks. A better approach is to generalize this idea and use a data structure called a hash tree or Merkle tree:



Now you do not need the whole hash list in advance to verify one chunk. Instead, you can ask your peer to provide you with some hashes along with the chunk just received.

Suppose you just received chunk L2 from a peer. Which and how many hashes do you need to verify if you correctly received chunk L2? How would you generalize which and how many hashes you need for each chunk?

Solution:

To verify if L2 was correctly received, we need hash 0-0, hash 1, and the top hash. Since the top hash is provided in the Torrent file, we need just two hashes from the peer: hash 0-0 and hash 1.

In general, we need the hash of the received chunk's sibling, the uncle of the received chunk's hash, and so on until the top hash. The number of required hashes is logarithmic in the total number of chunks.

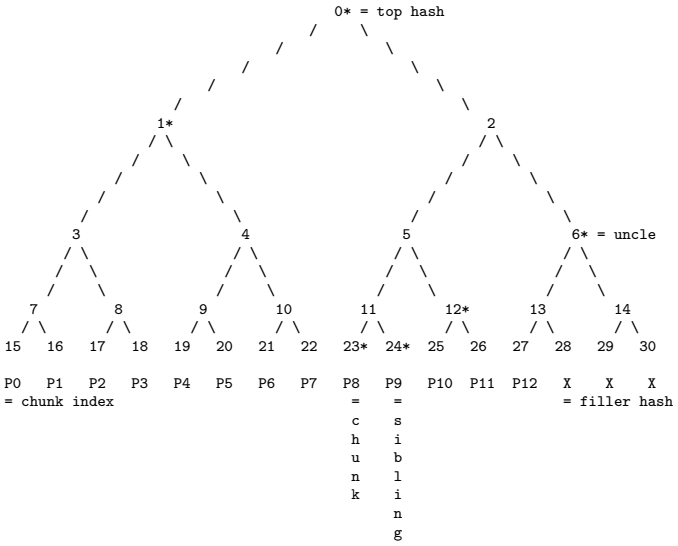
For the 10 GiB file, with each chunk we get the following size of hashes:

$$\lceil log_2 \left(40 \cdot 2^{10}\right) \rceil = 16$$

$$16 \cdot 20 \text{ B} = 320 \text{ B}$$

The downside is that now for all chunks overall we have to get $40 \cdot 2^{10} \cdot 16 \cdot 20$ B = 12.5 MiB of hashes.

Example from the BitTorrent specs: Hashes required for verifying chunk P8 are marked with *. You can compute the hash for the chunk P8 yourself, and you have the top hash provided in a Torrent file.



Hint: You might want to use something like this to implement efficient updates for part 3 of Project 2.

(d) You do not trust your peer with the contents of a chunk, but why you can use hashes provided by the peer for verifying the file?

Solution: We can use hashes provided by the peer for verifying the file because we receive the top hash from a trusted source, a Torrent file. Hence, if the peer provides us with an invalid hash or an invalid chunk, the top hash we compute will not match the expected top hash. In such cases, we do not use the chunk provided by the peer.

Question 2 ***Circumventing Network Policy Controls*** **(15 min)**

You are stuck at the Chicago O'Hare airport for at least 7 more hours, and the airport does not have free WiFi. How boring! Since you are unwilling to pay the offensive fees the hotspot provider demands to access the Internet, you start to explore what you can do with the limited connectivity. Quickly you find out that all connections to Internet hosts are blocked. You also discover that the DNS server of the hotspot answers queries for any hostname in the Internet.

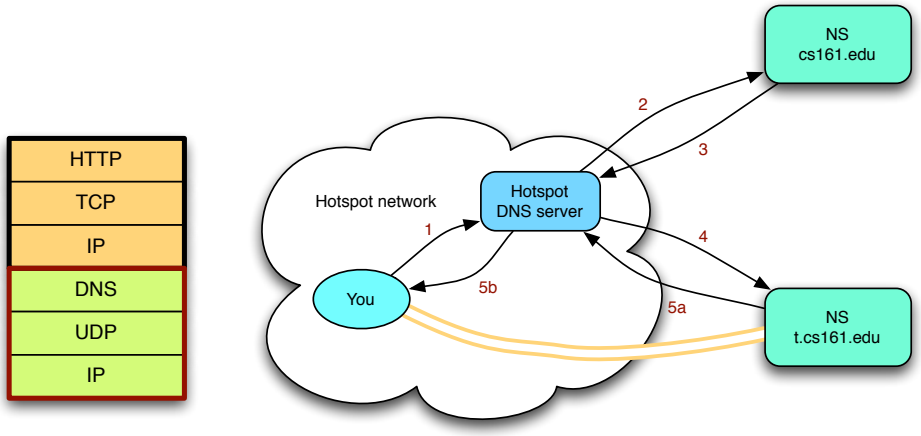
As an exceptionally studious and forward-looking CS 161 student, you were prepared for this scenario of Internet deprivation and in fact already set up your own DNS infrastructure prior to leaving on your trip. How is it possible to get free WiFi access in this scenario? What needed to happen prior to the trip? Sketch a diagram and describe any protocol you come up with.

Solution: In this scenario, you can **tunnel** Internet traffic through DNS. The key idea is to encapsulate your traffic into DNS queries that you send to a DNS server in the Internet under your control. This server decapsulates the DNS queries and relays the data to its intended destination, stuffs the response traffic into a DNS reply and ships it back to you.

For this to work, you must have set up a DNS server and a tunnel domain prior to leaving on the trip. Assume you own the domain `cs161.edu` and you would like to use `t.cs161.edu` as the tunnel subdomain.[1] Further, assume that the IP address of `t.cs161.edu` is 6.6.6.6 and you have deployed a custom DNS server at that address listening on port 53 for incoming requests.

The following graphic illustrates the necessary steps to establish a DNS tunnel. On the left hand side you see the network stack, where the bottom three layers correspond to the regular DNS communication in the hotspot network, depicted by the black arrows in the Figure on the right. The top three layers represent the *overlay network* that you create *inside* the DNS communication. In this case, assume that you would like to establish an HTTP connection to `192.150.187.12` and have already created an HTTP request with appropriate TCP/IP headers. For simplicity, let us refer to this bundled data as `encoded-request`.

[1] One often uses a separate subdomain for tunneling to be able to use the main domain also for other purposes. Another reason is that some (web-based) domain management tools only allow you to set NS entries for subdomains but not for the top-level domain itself.



1. To get the data to the server, you ask the local hotspot resolver to look up `encoded-request.t.cs161.edu`.
2. Assuming the hotspot resolver already knows the nameserver for `cs161.edu`, the resolver now needs to find out who is responsible for `t.cs161.edu`.
3. The authoritative nameserver for `cs161.edu` tells the hotspot resolver that `t.cs161.edu` is managed by 6.6.6.6.
4. Next, the hotspot resolver sends the query for `encoded-request.t.cs161.edu` to 6.6.6.6. Your custom nameserver at this address knows how to decode the incoming data, forward the HTTP request to `192.150.187.12` and encode the HTTP reply plus the TCP/IP headers.

5a. The encoded response is sent back as a TXT record, another form of DNS record we have not yet encountered.[2] This record allows you to associate arbitrary text with a DNS name.

5b. After having received the encoded HTTP response as a TXT record, the hotspot resolver forwards it you. At this point, you can decode the data in the TXT record and rebuild the overlay TCP/IP stack to extract the HTTP reply.

Note that steps (2) and (3) only occur once because the hotspot resolver will cache the nameserver for `t.cs161.edu`.

If you would like to read more about DNS tunneling, you can find a good introduction at `http://dnstunnel.de/`. Moreover, the software *iodine* (`http://dev.kryo.se/iodine/`) is a concrete example of a DNS tunnel implementation that ships both a client to be used on your laptop and a custom DNS server daemon.

[2] A question you might find interesting to explore: if the DNS server could only return A records (32-bit IP addresses), could you still get some sort of HTTP tunneling to work?

Question 3 ***Project 2 Rollback Prevention*** **(15 min)**

In Project 2, you were not required to prevent rollback attacks where the server reverted the state of a file to a previous value. In this problem you will design a scheme to prevent *partial rollback* attacks. (A *partial rollback* is one where the server rolls back the contents of one file, but not another.) Clients in your scheme can cooperate to work together against a malicious server, but may not keep state on the client.

(a) The simplest scheme has each user write to a shared *hash chain* that exists on the server. After an upload operation, the user who performed that operation adds a new entry to the chain and uploads it to the storage server. What should each entry contain so that users can verify the state of the server by examining the hash chain?

Solution: There are many valid answers to this question. We will go in to one of them.

The idea here is that we are imagining a shared global hash chain on the server. Each entry should contain enough information to reconstruct what should have happened.

So, our scheme will have a hash chain where whenever a user uploads a new file, they add a new hash to the chain containing the following:

- A random, fresh transaction ID. (Not strictly necessary, but a good practice.)
- The user who is performing the update.
- The key the file is stored at on the server.
- The asymmetric signature of the file data. (Could be also just a hash.)
- The previous most-recent hash that existed on the server.
- An asymmetric signature of the entire block.

Then, whenever we make an action on the server, it becomes simple (if inefficient) to check that the state at the server matches the hash chain correctly. We first download the chain, and then verify each hash starting from the top hash. If the server ever rolls back the state of a single file, they would also have to roll back that part of the hash chain. This is only possible if the state they are rolling back is the most recently updated file.

That is, the server can only *completely* revert state to a previously seen state, not just the state of a single file.

(b) Can we create a more secure scheme if we allow clients to maintain state? What additional guarantees can we provide?

Solution: In the previous scheme, the server can still roll back to a previously seen state by changing everything. If we allow clients to keep state, we can prevent the server from ever reverting state.

Again, there are many valid schemes to do this; we discuss one below.

We keep a hash chain for each file independently on the server. When we share a file with someone else, we also send them a link to the hash chain so they can update it with any updates they also make. Our hash chain entries contain the same information as above.

On the client, we keep a copy of the random transaction ID of the file for each file. This makes it so that the server cannot rollback any file because the ID will be invalid.

Another less efficient solution is to simply only remember the latest transaction ID of the global hash chain from the solution to the previous problem.