582615 - Overlay and P2P Networks, Spring 2016

Exercise 2 model answers

Considering the following points in the answer granted points in the assignment.

Assignment 1 - BitTorrent (5 points)

BitTorrent relies on two key algorithms: *Choke algorithm* for peer selection, and *Rarest first algorithm* for piece download strategy. Describe:

What does the *Rarest first algorithm* do?

The rarest first algorithm is a piece selection strategy, where a peer maintains a list of which pieces are the most rare in its neighborhood, and attempts to get them before the other pieces. The goal is to prevent rare pieces from becoming unavailable on the network as nodes leave when their download is finished.

What do the three added policies do? When are they used and what is their goal?

The three added policies are called the *random first* policy, the *strict priority* policy, and the *end game mode*. Random first policy is used to bootstrap the download by requesting any four pieces at random, in order to get something to contribute to the network. The strict priority policy states, that when the first block of a piece is requested, the rest of the blocks of that piece are requested at highest priority. Only complete pieces can be contributed back to the network, so it is more urgent to collect complete pieces, than to collect individual blocks here and there. The third policy is the end game mode. When a client sent a request for the last missing blocks, it will send that request to all the peers it knows that have those blocks. As the client receives the blocks it will cancel the respective requests. The purpose if this is to get the last pieces and finish the download as quickly as possible, without having to wait for a selected peer to grant a slot.

What does the *Choke algorithm* do? How are new joining peers with nothing to give taken into account?

The choke algorithm is used for giving turns to peers that express an interest toward some blocks. Every ten seconds, the three peers with the most download bandwidth are unchoked, i.e. allowed to download, and every 30 seconds a random peer is unchoked. The first part is based on the assumption that those with the highest bandwidth are also likely the most effective to disseminate the content further. The second part, called the opportunistic unchoke, serves to allow those who just joined the network to get those first four parts, and may also help the end game mode.

Starting from mainline BitTorrent 4.0.0, clients that have finished downloading and decide to stay seeding the torrent use a modified version of the *Choke algorithm*. How is this modified version different?

The modification changed the focus from bandwidth to time. Instead of ordering peers by their download rate, the peers are now ordered based on how long ago they were last unchoked, causing also the peers with lower bandwidth to get equal chance to download. For peers that have not been unchoked by the seed before, the opportunistic unchoke remains.

Assignment 2 - Bloom Filters (5 points)

What are Bloom filters and what are they used for?

Bloom filters are probabilistic data structures that can be used to answer the question, *'is it possible that this element might be part of the set?'* They are a relatively compact, and queries to Bloom filters are relatively cheap. For example, a system that tracks content dissemination in a P2P system might have a Bloom filter for each peer that contains hashes of files as their elements; a quick check from the bloom filter can answer whether it's worth asking this peer for some file.

What is the difference between a Bloom filter and a regular hash table?

Both Bloom filters and hash tables map elements to indexes in an array based on the output of hash functions. Regular hash tables usually run a key through a hash function that returns an index, and make that index point to a value. Bloom filters run the key through a collection of hash functions to get a collection of indexes, and make the value at those indexes 1, i.e. Bloom filters exhibit the properties of a set instead of a map.

Describe the steps of inserting an item into a Bloom filter, and querying the presence of that item.

When an element is inserted into a Bloom filter, its key is run through a set of hash functions, their outputs are mapped to indexes on an array and the value at those indexes is set to 1.

When an element is queried in a Bloom filter, its key is run through the same set of hash functions, their outputs are mapped to indexes on the array, and if the value in all of the indexes is 1, the element may be part of the set. If the value at any index is 0, the element is not in the set.

State	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Result
Empty filter	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Insert A hash1(A) = 4 hash2(A) = 11 hash3(A) = 1 hash4(A) = 7	0	**1**	0	0	**1**	0	0	**1**	0	0	0	**1**	0	0	0	0	
Query A hash1(A) = 4 hash2(A) = 11 hash3(A) = 1 hash4(A) = 7	0	**1?**	0	0	**1?**	0	0	**1?**	0	0	0	**1?**	0	0	0	0	True
Query B hash1(B) = 2 hash2(B) = 7 hash3(B) = 1 hash4(B) = 11	0	**1?**	**0?**	0	1	0	0	**1?**	0	0	0	**1?**	0	0	0	0	False

Can elements be removed from a standard Bloom filter? Explain why.

No. The only way to remove an element from the standard bloom filter is to reconstruct it without the element to remove. This is due to the probabilistic nature of the bloom filter and hash collisions. Remember that the Bloom filter does not remember for certain whether an element has been inserted or not. The 'trivial solution' to run the element through the hash functions, and set the value at the returned indexes to zero would effectively remove all elements for which any of the indexes overlap with those of the removed element. This would invalidate the invariant that only positives can be false.

Bloom filters sometimes return false results. Doesn't that make them useless? What *do* we know about the results?

The Bloom filters sometimes return **false positive** results. That is, if the Bloom filter says that an element is part of the set, it might be part of the set. If a Bloom filter says that an element is not part of the set, it is definitely not part of that set. A typical use for a Bloom filter is to perform a quick check of whether it's worth conducting a more expensive query; it's a way to cheaply weed out most of the queries that would have certainly come out empty anyways.

Assignment 3 - Distributed Hash Tables (DHTs) (5 points)

Hash tables are *key-value-stores* where a key is mangled through a hash function, the result is mapped to a position in some address space, and a pointer to the value is stored in that position. A Distributed Hash Table is essentially the same, but the address space and storage are divided in parts and the parts are distributed among a collection of hosts instead of having everything on one host.

What are typical operations that a DHT provides?

The typical operations provided by a DHT are those provided by a regular hash table as well: insertion and query. Arguably also deletion, but that can be considered equivalent with insertion of an empty value.

What challenges does spreading a hash table over multiple nodes impose?

When a hash table is distributed among multiple nodes, each node will know only about part of the address space. There needs to be a way to map positions in the address space to nodes in the DHT. When adding or removing nodes from the DHT, the mappings need to be changed, and the objects transfered to the new nodes responsible for that part of the address space.

A DHT may grow to be so big, that it's not reasonable to have all nodes maintain entries about all other nodes. In such a case, the DHT needs a mechanism for routing the insertions and queries towards the correct destination through other nodes. A mathematical model, such as the XOR-distance metric may be used.

In a regular hash table, if the node that has it fails, the hash table is gone. In a DHT, the number of nodes is typically larger, and so is the probability of a node failure; redundancy needs to be implemented. There are other typical challenges related to all distributed systems, such as consistency and availability, to be considered as well.

Security may also be a concern when designing a DHT. This may not be an issue for an open file-sharing network, but it may be a huge concern for a database of business information, where access to the data needs to be restricted and coordinated to ensure integrity and confidentiality.

Choose a DHT (e.g., Kademlia, CAN, Tapestry or Chord) and describe the steps of inserting a (key, value) pair into that DHT. What can you say in general about retrieving that value from the DHT?

Let's use Kademlia as the example. In Kademlia, all nodes have a node id, that is a random 160-bit number. All (hashes of) keys are also 160-bit numbers. Kademlia uses the XOR-metric (number produced by the bitwise exclusive or-operation) to compute the distance between a node id and a key; the longer the matching prefix between a key and a node id, the shorter the distance in XOR-metric.

Kademlia nodes maintain lists of (IP address, UDP port, Node id) triplets, known as k-buckets; one list for each possible common prefix length, up to k nodes in each list. When inserting an object in Kademlia, a node starts by locating the k nodes with the smallest XOR-distance with the key. This is done by querying the closest nodes in the node's own routing table for locating a node whose node id would be the same as the key. They should return with lists of k nodes that they know closest to the key. The node then selects the k nodes closest to the key, and stores the object on those nodes.

Retrieving a value in general involves the same steps: hashing the key, mapping it to a location or distance, finding the node or nodes responsible for that location, and requesting the value for that key. In Kademlia, the find_value operation asks for the value from the (by XOR-metric) closest nodes, which either return a list of nodes that are even closer, or if they had it, the value that was requested.

Assignment 4 - Consistent hashing (5 points)

How does consistent hashing work?

In consistent hashing, the hash table is modeled as a ring. The address space of that ring is set to [0,1[. The ring is then divided into buckets. A bucket is technically a container for an address range along the ring. To ensure a uniform distribution of buckets along the ring, a well balanced random number generator should be used. Each node that participates in the DHT is assigned one or more buckets, i.e. responsibility for objects that are placed at those assigned address ranges along the ring. The idea of distributing the buckets and objects evenly along the ring is to spread the load among the nodes as evenly as possible.

When an object is inserted into a DHT that uses consistent hashing, its key is entered to a well balanced hash function, whose output is mapped to a position on the ring. That position belongs to some address range (a bucket), and that bucket belongs to a node. The object is stored on that node.

For a query, the key is hashed again and mapped to the same location on the ring, which maps to some bucket, which again maps to the node that has the object.

What happens when you add/remove a node in a system that uses consistent hashing?

When a node joins, it is given (or chooses) one or more buckets at random points on the ring. They may be buckets that previously belonged to another node, or they may be completely new buckets, meaning that the ring gets partitioned into even smaller parts. The objects for the effected part of the ring will maintain their position on the ring, but will be stored on the new node that is responsible for those buckets.

When a node parts, its buckets are joined with the clockwise next buckets, and the objects are moved to the nodes now responsible for those buckets. Typically only the affected objects need to be transfered between nodes.

Why is it important to use a well balanced hash function?

A well balanced hash function distributes the buckets and keys evenly along the ring, and hence evenly along the nodes. Poorly balanced functions spread the objects unevenly, and will burden some nodes significantly more than others, causing the DHT to have bottlenecks and unused resources at the same time, degrading performance.

How can replication be done with consistent hashing?

Replication is typically done to provide redundancy in case of node failure. Some strategies for replication in consistent hashing:

- Placing of copies of the objects in some number of buckets following the bucket where object was placed. If the responsible node fails, copies of the objects will already be available in the buckets that would become responsible for the bucket if the node left.
- Another strategy is to add a suffix for the key, compute multiple positions on the ring, and store copies of the object to those locations.
- One strategy is also to simply map each bucket to multiple nodes, so that if one node fails, there are other nodes that already know everything about that bucket.

Turbo challenge (5 points)

How does Cassandra partition the data?

Cassandra uses consistent hashing to partition the data. See assignment about consistent hashing for more.

What does Cassandra do when it notices that the data is distributed unequally among the nodes?

Cassandra analyzes the load of each node periodically, and moves the lightly burdened buckets along the ring to balance the load. This requires a minimal number of object transfers between nodes and distributes the load evenly along the whole cluster.

What techniques does Cassandra use achieve scalability and redundancy?

The load balancing strategy is also Cassandra's main strategy for scalability. The periodic balancing of load is also taken into account when adding new nodes; their position on the ring is mutually chosen so that they share the burden of the currently most burdened node.

Redundancy is achieved through replication. Cassandra supports configurable replication strategies, namely the SimpleStrategy and NetworkTopologyStrategy. The simple strategy simply places the replicas on the next (replication_factor - 1) nodes along the consistent hashing ring. Using this strategy, the keys are already replicated on the nodes that would become responsible for them should the node responsible for that key fail. NetworkTopologyStrategy allows smarter replica placement that can distribute the replicas in different racks or data centers, depending on the configured topology.

Cassandra's membership strategy also plays its role for both features. In Cassandra, every node knows about all other nodes. It is always easy to find a node that has the item and requests never need to travel many hops. As a downside however the routing tables can be big.