Distributed Systems Lab 14

Hashing-based data distribution

The previous labs focused largely on ensuring consistency of data in the face of concurrency and failures. This lab is in turn concerned with distributing data across machines comprising the system and efficiently localizing a machine responsible for a particular data item. It covers popular techniques known as *consistent hashing* and *distributed hash tables* and, as a running example, employs a solution named *Chord*. We will use files contained in this package, which you should download and extract locally.

Learning Section

Let us assume that we have a set of `N` machines, where `N` is potentially "large" and may change over time as machines come and go. Suppose also that we have a "huge" collection of data items, such that each data item has an associated *key* by which this item will be searched for. Our goal is to divide data items across the machines such that:

1. Each machine receives a similar number of data items and adding or removing a machine requires redistributing only a fraction of all data items.

2. Each data item can be efficiently looked up by any machine in the system, that is, given a transport-layer address of an arbitrary machine (i.e., its IP address and a port number) and the key of the data item, the system returns the transport-layer address of the machine that hosts the data item (if the item is present in the system).

Consistent hashing

With respect to Requirement 1., a natural way of dividing the data items among the machines would involve classic hashing. In this approach, each machine would constitute a separate hash bucket, uniquely identified by some Machine ID, `MID` in `[0..N - 1]`. It would be responsible for storing those data items whose keys hash to its bucket, that is, every item whose Data ID, `DID = hash(key) mod N`, equals `MID`.

Unfortunately, such an approach does not guarantee that changing the machine population slightly would require only minimal changes to the data item distribution. On the contrary, changing the number of machines by as little as one requires redistributing virtually all data items as a result of their `DID`s changing.

This problem is addressed by **consistent hashing**. In its popular variant, all IDs belong to a large numerical space `[0..2^B - 1]`, which conceptually loops around, thereby forming a ring modulo `2^B`, that is, the next ID after `2^B - 1` is `0`. Constant `B` denotes the length of an identifier in bits and is normally at least 128. Each machine is assigned its `MID` from this space uniformly at random, so that the machines are evenly dispersed on the ring. Likewise, the `DID` for a data item is obtained by hashing the key of the item to the ID space with a cryptographic hash function, so that `DID`s are also expected to be uniformly distributed across the ring. Importantly, however, in contrast to classic

hashing, each machine is responsible not only for those data items whose DIDs are equal to its MID, but for all those data items whose DIDs fall into a “specific range” on the ring around the MID. While the definition of “specific range” may vary between particular solutions, it ensures that adding or removing a machine affects only the two closest machines: the immediate successor and predecessor of the machine on the ring.

In Chord, which is our running example of an actual solution for this lab, the notion of “specific range” is defined as follows: a node with a given MID is responsible for all DIDs that are equal to the MID or fall between this MID and the one of the preceding active machine on the ring. For example, suppose that `B = 5` (i.e., the maximal ID is 31) and that machines with the following MIDs are active in the system: 0, 3, 8, 10, 13, 17, 19, 20, and 27. In such a setting, the machine with `MID = 19` (machine 19) is responsible for DIDs `{18, 19}` as it succeeds on the ring the machine with `MID = 17` (machine 17). Accordingly, machine 20 is responsible only for DID `{20}`, machine 27 for `{21, ..., 27}`, and machine 0 for `{28, ..., 31, 0}`. The complete mapping of DIDs to MIDs in the example is illustrated in this diagram.

As a side note, in practice, a single DID is mapped to multiple MIDs so that simultaneous failures of several machines can be tolerated. However, such redundancy rules are relatively straightforward and are thus immaterial for today’s lab.

Distributed hash table

Requirement 2., the ability to look up data items, can be ensured in a centralized manner: by having a dedicated (possibly replicated) server that monitors all machines, maintains the DID-to-MID mapping, updates it whenever the machine population changes, and propagates it back to each machine. In fact, this solution is quite common in practice.

In systems involving very large numbers of machines, however, a centralized solution may not scale. Instead, such systems may require maintaining the DID-to-MID mapping in a decentralized fashion. Such functionality is referred to as a **distributed hash table** (**DHT**).

Many DHTs have been proposed. Their common property is that they are implemented as **overlay networks**. Such a network is composed of *nodes* and *links*. In our case, a *node* corresponds to a machine (or actually an instance of the process implementing the DHT functionality on this machine). In any case, each node knows only about a small subset of all nodes. The knowledge that a node has about another node includes information on how the other node can be reached (e.g., its transport-layer address). This information can thus be thought of as a (logical) *link* in the overlay network: by having such a link, the node can send messages to the other node; in contrast, it is unable to initiate communication with a node it has no link to (no information about). A DHT uses its overlay links to route a message with a given key between its nodes, so that the message ultimately reaches a node responsible for this key (actually for the corresponding DID). This routing is done at the application layer of the OS network protocol stack, and hence is called **overlay routing** to emphasize that it is done on top of the regular Internet routing. The particular DHTs differ in their rules on selecting overlay links and routing messages. In particular, during the lecture, rules for the *Pastry* DHT were given; here, in turn, we explain the ones for *Chord*.

Overlay construction rules in Chord

In Chord, a link to a node comprises, among others, the MID and the transport-layer address of this node. Each node maintains locally three data structures with links to *other* nodes:

- *successor table* – Contains up to R links to nodes succeeding the present node on the ring (where $R \geq 1$ is a configuration parameter). The 0-th entry of the table is a link to the next

active node on the ring in the clockwise order (if it exists), the 1-st entry – to the second next node in this order, and so on.

- *predecessor table* – By symmetry, contains up to `R` links to nodes preceding the present node on the ring. The `0`-th entry of the table is a link to the next active node in the counter-clockwise order on the ring (if it exists), the 1-st entry – to the second next node in this order, and so on.

- *finger table* – Contains up to `B` links that are shortcuts in the ring. More specifically, the `i`-th element of the table is a link to the first active node on the ring in the clockwise order whose `MID` is at least `2^i` apart from the `MID` of the present node but less than `2^(i+1)` apart (if such a node exists).

The three tables constitute the *routing state* of a node. It can be observed that this state scales logarithmically with `N` (assuming a uniform `MID` distribution on the ring).

For illustration, consider the previous example of a ring. If `R` was `3`, then the successor and predecessor tables of machine `8` would be `[10, 13, 17]` and `[3, 0, 27]`, respectively. In contrast, if `R` were greater than `8`, then only the initial `8` entries would be present in the two tables of any machine (the other being `null`). When it comes in turn to the finger tables, for machine `8` it would be as follows: `[null, 10, 13, 17, 27]` because there is no machine with its `MID` in range `[8+2^0..8+2^1)`, the first machine in the clockwise order with its `MID` in range `[8+2^1..8+2^2)` is machine `10`, the first machine in range `[8+2^2..8+2^3)` is machine `13`, and so on. Accordingly, the finger table of machine `19` would be `[20, null, null, 27, 3]`. The routing states of all nodes in the example, assuming `R = 1`, are illustrated in this diagram.

Routing in Chord

The routing states allow individual nodes to forward messages for specific `DID`s over the overlay links so that the messages ultimately reach the nodes responsible for the `DID`s. To this end, each node receiving a message inspects the destination `DID` in the message and its local routing state and decides what to do with the message as follows:

1. If, based on its predecessor and successor tables and its own `MID`, the node is able to determine the target `MID` responsible for the `DID`, then it forwards the message using the transport-layer address associated with the link to that `MID` (if the target `MID` belongs to another node) or accepts the message as the destination node (if the target `MID` equals its own `MID`).

2. Otherwise, it uses its finger table greedily: it selects for forwarding the entry with the largest index in the table such that sending the message to the `MID` corresponding to the entry does not result in “jumping” in the clockwise order over the `DID` on the ring. (In other words, it forwards the message as far as it is possible but not to overjump the `DID`.)

As an example, consider again the aforementioned system (with `B = 5` and `R = 1`), depicted in this diagram. Routing a message for `DID = 3` by machine `8` can be done in one hop because, from its predecessor table, the machine can determine that machine `3` is responsible for the `DID`. Likewise, routing a message for `DID = 12` by node `10` can be done in one hop because, from its successor table, machine `10` can determine that machine `13` is responsible for the `DID`. In contrast, routing a message for `DID = 25` by machine `0` requires four hops:

- Machine `0` uses the 4-th entry in its finger table as it is the largest-index entry and does not cause jumping over `25` in the clockwise order on the ring. In effect, the message is forwarded to machine `17`.

- Machine 17 uses the 1-st entry in its finger table because the 2-nd entry is null while forwarding to the 3-rd entry would result in jumping over 25 in the clockwise order on the ring. As a result, the message is received by machine 19.
- Machine 19 uses the 0-th entry in its finger table for the same reasons, thereby sending the message to machine 20.
- Machine 20 determines, based on its successor table, that it is machine 27 that is responsible for DID = 25, so it forwards the message to that machine, which constitutes the last transmission as machine 27 accepts the message based on its predecessor table and its own MID.

In general, like routing state, the maximal number of hops scales logarithmically with N (assuming a uniform distribution of MIDs over the ring). Therefore, at least from the algorithmic perspective, Chord is scalable overall: if the system doubles, its asymptotic performance drops only by a constant factor.

Final remarks

The presented rules concern only a system in a stable state. In the real world, however, the state of particular nodes may be inconsistent, notably during changes in the node population. Moreover, there are many other issues a practical implementation has to consider. For more information, refer to the additional homework.

Small Assignment

Your task is to implement in Rust the link selection and routing rules described for Chord in this scenario. You shall complete the implementation of two functions provided in the template, following doc comments defined there.

This Small Assignment is worth **2 points**: 1 point if the link selection rules are implemented correctly and 1 point if routing is implemented correctly. To run the code you should use the executor system you developed as the first Large Assignment.

Additional Homework

In you are interested in Chord, you can start with the original Chord paper. The paper presents mostly theoretical concepts, while implementing them in practice requires further solutions, some of which are described in the following PhD dissertation.

As an alternative to Chord, you can look at Pastry, which was developed roughly at the same time and was more mature upon its introduction (Pastry was discussed during the lecture). In particular, the website of the project and of its key contributor may be a good start.

All in all, DHTs are an important concept in distributed systems, and hence there are myriads of publications concerned with them.

Authors: K. Iwanicki, W. Ciszewski, M. Banaszek.