Distributed Systems Lab 13

Eventually consistent gossip-based aggregation

During the previous labs, we discussed eventual consistency and, more specifically, its op-based variant. In that variant, by means of reliable broadcast, processes constituting the system exchange client operations that are subsequently applied to their local states, so that in effect those states converge to the same values. This lab, in turn, focuses on state-based eventual consistency, in which process states themselves are being exchanged. As an application example, we will adopt gossip-based aggregation. We will use files contained in this package, which you should download and extract locally.

Learning Section

In-network aggregation

Aggregation describes functionality of summarizing information. There are many functions that can be utilized for producing various types of summaries, but during this lab we will be interested in basic ones, namely `MIN`, `MAX`, `COUNT`, `SUM`, and `AVG`. Without loss of generality, we assume that each process `i` holds a single value, `vi`, and the goal is to compute the value of such a function over the set comprising the values of all processes, `V = {v1, v2, ..., vN}`. In our distributed setting, a process will correspond to a network node, and the nodes communicate only by exchanging messages.

A straightforward approach to aggregating set `V` can be as follows: all network nodes send their raw values to a distinguished node, and the distinguished node computes the desired aggregation function on the entire value set, possibly disseminating the result back to the individual nodes if necessary. Such a centralized approach, however, may not be appropriate in some scenarios, notably when the values, `vi`, are large or somehow bound to their nodes. In such cases, distributed **in-network aggregation** may be required.

Distributed in-network computation of the aforementioned aggregation functions relies on so-called *partial aggregates*. Partial aggregates summarize information from subsets of nodes, can be combined to represent summaries of larger subsets, and final aggregate values are computed from them. More formally, the following operations are defined on partial aggregates:

- `init(v) -> P` – given the value, `v`, of the executing node, initializes a partial aggregate, `P`, which will represent a singleton subset containing the value of the node.
- `merge(P, P) -> P` – merges two partial aggregates for two subsets of node values into a partial aggregate for the subset being the union of the two subsets.
- `eval(P) -> a` – returns the final result, `a`, of the computed aggregation function for the subset of node values corresponding to the partial aggregate, `P`.

For example, when computing `MAX`, the partial aggregate would be simply an element of set `V`, and the three operations would be defined as follows: `init(v) = v`, `merge(vx, vy) = max(vx, vy)`, `eval(v) = v`.

Decentralized gossip-based aggregation

A common approach to performing in-network aggregation entails organizing the nodes into a logical spanning tree. Each node in the tree `init`ializes its own partial aggregate based on its local value. Then, the leaf nodes send their partial aggregates to their parent nodes in the tree. Such a node `merge`s the partial aggregates received from its children into its local partial aggregate, which it then forwards to its parent node in the tree, and so on, up to the root node of the tree. Finally, the root node `eval`uates the final aggregation result from its local partial aggregate and, if necessary, disseminates this value to the individual nodes.

Such an approach is well suited when the node population and connectivity are fairly stable. In contrast, if they are dynamic, maintaining the tree can become a major problem. In such scenarios, more flexible communication algorithms are employed, which in essence offer multiple paths for information to flow from one node to another. A class particularly popular in large-scale highly dynamic systems is **epidemic algorithms**, also known as **gossip-based algorithms** or **gossiping** for short.

In such an algorithm, nodes operate in rounds, each round lasting some fixed time interval. In every round, each node selects at random another node in the network. This is frequently a task of a so-called peer sampling service, which, interestingly, need not be aware of the entire node population. The node then either sends its local information to that node (*push-based gossiping*), fetches information from that node (*pull-based gossiping*), or exchanges information with that node (*push-pull-based gossiping*). It has been shown that such algorithms propagate information in the network exponentially fast, like epidemics or gossip, and hence their name. They have numerous applications, in-network aggregation being one of them.

In particular, a distributed computation of an aggregation function using gossiping can proceed roughly as follows. Each node `init`ializes its own local partial aggregate based on its local value. The nodes then repeatedly gossip their local partial aggregates, such that upon reception of a partial aggregate from another node in a given round, the recipient `merge`s it into the local one, to be used in the next round, and so on. The gossiping is in principle an infinite process but should eventually converge, and thus, if sufficiently many rounds have passed, any node (not just some specific one) can `eval`uate its local partial aggregate into the final aggregation result. In practice, the evaluation can happen in each round to give an additional indication of whether the results have converged sufficiently.

Probabilistic counting sketches

Such a gossip-based in-network aggregation algorithm can be directly applied to `MIN` and `MAX`. In contrast, employing it for the other considered aggregation functions is more involved. This is due to the multi-path propagation of information during gossiping, which may cause the value of a given node to contribute multiple times to the final aggregate, thereby distorting the counts, sums, and averages. This problem can be addressed by having partial aggregates, and notably, their `merge` operation, *order- and duplicate-insensitive*. In other words, partial aggregates can be made monotonic and the `merge` operation can be made:

- idempotent – `merge(P1, P1) = P1`,
- commutative – `merge(P1, P2) = merge(P2, P1)`, and
- associative – `merge(P1, merge(P2, P3)) = merge(merge(P1, P2), P3)`.

A straightforward way of ensuring these properties could be by having partial aggregates correspond to their respective value subsets and the `merge` operation to a set union. However, in effect, the state exchanged between the nodes during gossiping would be linear in the number of nodes, thereby impairing scalability. Therefore, we want partial aggregates to somehow summarize information as

well. To explain how this can be done, let us focus on the `COUNT` function, as other functions can be implemented using this function.

Order- and duplicate-insensitive counting can be realized by so-called **probabilistic counting sketches/synopses**. They do not produce exact values but only estimates. Nevertheless, in a dynamic system, estimates are typically sufficient, as exact counts could be difficult to obtain anyway. There are many algorithms for computing such approximate counts, notably linear counting and hyper-log-log are widely recognized ones. Here, we focus on a variant of probabilistic counting.

In this approach, a partial aggregate comprises one or more *instances* of a same-sized *sketch*. A sketch is a bitmask logarithmic in the number of counted values. The three operations on such bitmasks are defined as follows:

- `init(_) -> P` – Produces a bitmask that has `0` on all positions but one. The position of the sole `1` is selected at random from a geometric distribution with parameter `1/2`, that is, position `0` is selected with probability `1/2`, position `1`, with probability `1/4`, position `2`, with probability `1/8`, and so on.
- `merge(P, P) -> P` – Returns a bitmask that is a bit-wise OR of the two bitmasks given as parameters.
- `eval(P) -> a` – Yields value `a` equal to `c * 2^Fz`, where `Fz` is the position in the bitmask of the first `0` bit and `c = 1.29281` is a scaling factor. Value `a` represents the result of counting.

All in all, rather than an exact count, a sketch aims to estimate its order of magnitude (base 2), that is, its base-2 logarithm. This may result in large errors. They can be alleviated by using multiple sketch instances per partial aggregate. The three operations extend to such a vector of sketch instances as follows:

- `init` uses the previous algorithm independently for each instance in the vector, thereby setting one bit to `1` in each instance.
- `merge` also performs the bitwise OR independently for each element of the vectors of instances constituting the partial aggregates: instance `0` in the first aggregate is ORed with instance `0` in the second aggregate, instance `1` with instance `1`, and so on.
- `eval` yields a geometric average of the values computed using the previous algorithm, that is, a value equal to `c * 2^((Fz0 + Fz1 + ... + FzM)/(M+1))`, where `M+1` is the number of sketch instances in a partial aggregate, and `Fzi` is the index of the first `0` bit in instance `i`.

Small Assignment

Your task is to implement gossip-based aggregation using probabilistic counting sketches as discussed hitherto. The implementation must be based on the supplied template, which assumes a system consisting of `Node`s and clients that operate as follows.

A client can contact any node and request it to install an aggregation query, which is done by sending to the node a `QueryInstallMsg`. The goal of the query is estimating the number of nodes in the system that satisfy the associated `predicate`. The query also specifies the number of probabilistic sketch instances (`num_instances`) and bits in each instance (`bits_per_instance`) that should be utilized when computing the aggregate.

Nodes use gossiping to propagate among each other the information necessary for executing such queries. A node initiates gossiping whenever it receives a `SyncTriggerMsg`. It is the system that controls when those messages are sent; you must never send them in your implementation. Upon reception of such a message, the node requests its associated peer sampling service (`pss`) to obtain a random node with which it will gossip. Subsequently, it sends the necessary information to that

node within a single `SyncGossipMsg`. The other node does not reply with any message, that is, the communication scheme is push-based.

At any moment in time, a client can contact any node asking it to provide the current estimate of the result for any query. The client does this by sending to the node a `QueryResultPollMsg`. The message carries the identifier of the node on which the query was originally installed (`initiator`) and a callback to be executed by the receiving node with the query result estimate as the parameter (`callback`). If the node is not aware of the query, it executes the callback with value `None`; otherwise, it executes the callback with its current estimate of the node count satisfying the predicate of the query.

To recap, multiple queries can be executing in the system at the same time. However, a next query installed by a client at a node overrides the previous query installed at that node by the same or another client: the execution of the previous query should eventually cease. In other words, for every initiator, each node executes only the most recent query of which it is aware. Finally, a `SyncGossipMsg`, sent by a node to its peer, should contain information on all queries executed by the node.

Your solution must implement probabilistic counting by means of the `ProbabilisticCounter` structure. As the source of random values (`RandomnessSource`) for selecting sketch bits, the dedicated `rs` object of the node must be utilized. The object generates pseudo-random values from a uniform distribution. To convert them into ones from the desired geometric distribution, your solution must use the provided `uniform_u32_to_geometric` method. Moreover, `ProbabilisticCounter` must ensure special handling of two corner cases. First, `evaluate` on a counter in which no sketch instance has any bit set to `1` should return `0` instead of the value implied by the previous formula. Second, `evaluate` on a counter in which at least one instance has all bits set to `1` should return infinity (`u64::MAX`) instead of the value from the formula.

This Small Assignment is worth **2 points**. To run the system you should use the executor system you implemented as the first Large Assignment.

Additional Homework

When you have solved the assignment, think how to use partial aggregates for `COUNT` to implement `SUM` and `AVG`.

Finally, gossiping and aggregation are large and important areas in distributed systems. If you are interested, explore the cited papers as well as others concerned with the topics.

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