Lecture 29: May 5

Lecturer: Satish Rao *Scribe: Kevin Lacker*

29.1 Gossip

The idea in this lecture is that we have n distributed databases and we want them to all have the same content. Every once in a while, one of them receives an update, and we want to make sure that all of them get updated. In general, the algorithms in this lecture were first published by Demers et al, and also analyzed by Karp et al.

The plan is for the databases to "gossip" amongst themselves in order to update as quickly and efficiently as possible. One simple possibility is for databases to call each other at random and synchronize the data they contain. However, it is somewhat inefficient to exchange the entire database constantly. The solution for this problem is to only exchange the most recent updates. This is known as "rumormongering." It is more fun to think of an algorithm as people spreading rumors rather than databases exchanging updates.

Let's think of one update occurring independently from all the other updates, and analyze how knowledge of this update spreads throughout the graph. Could we do anything structured outside of our call-people-at-random model?

- We could all get in a circle and tell our neighbors our rumors. This is efficient, but slow.
- We could build a tree with small diameter and communicate across the tree edges. This is fast and efficient, but not fault-tolerant. For example, if the root of the tree is knocked out, half of the graph will not receive our update.

There are two main ideas for propagating rumors.

- "Push." If we know a rumor, pick somebody and tell them.
- "Pull." Every once in a while, pick somebody and ask for rumors.

29.2 Push

To analyze the push algorithm, we break down the rumor's spread into two phases. First, the rumor must go from one person knowing it to a constant fraction of the people knowing it. Second, the rumor must go from a constant fraction of the people not knowing it, to zero people not knowing it.

Let s_t be the number of people who know the rumor after time step t. To analyze Phase 1, assume that $s_{t-1} < n/4$, and let us estimate $E[s_t]$. Imagine that it is now step t, and the s_{t-1} people who already know the rumor are in some order. They each pick somebody at random to call. Since $s_{t-1} < n/4$, at each step of the way most of the people have still not heard the rumor. Thus, each person who already knows the rumor expects to inform at least 1/2 people. So $E[s_t] \geq s_{t-1}$. Thus, we would expect a constant fraction of the people to learn the rumor in $O(\log n)$ time. This analysis is not exactly correct; that would require Chernov bounds. But let us worry about that later.

To analyze Phase 2, let u_t denote the number of people who have not heard the rumor as of time step t, and assume $u_{t-1} < n/2$. Since over $n/2$ people know the rumor, the odds of a given person who does not know the rumor at step $t-1$ remaining uninformed through step t are are most $(1-\frac{1}{n})^{\frac{n}{2}}$. Thus,

$$
\begin{aligned}
E[u_t] &\leq u_{t-1} - (1-(1-\frac{1}{n})^{\frac{n}{2}})u_{t-1} \\
E[u_t] &\leq u_{t-1} - (1-e^{-\frac{1}{2}})u_{t-1} \\
E[u_t] &\leq e^{-\frac{1}{2}}u_{t-1}
\end{aligned}
$$

Therefore Phase 2 should also take $O(\log n)$ steps.

As far as efficiency goes, during Phase 1 we are expanding exponentially. Therefore the last step dominates, and we send $O(n)$ messages. During Phase 2 we are sending $O(n)$ messages at every step. Thus the total number of messages sent is $O(n \log n)$, which is an inefficiency of $O(\log n)$.

29.3 Pull

Analyzing the pull algorithm one update at a time is a little strange. How do we know when to stop pulling? How do we know when to start pulling? And on this note, how do we know when to stop the push algorithm? This is fixable, but messy, so we will just not worry about it.

In Phase 1 of the pull algorithm, we bound the number of people who know the rumor, $s_{t-1} \leq n/2$. A person who does not know the rumor has exactly a s_{t-1}/n chance of learning the rumor in time step t. There are $n - s_{t-1}$ such people. So

$$
\begin{aligned}
E[s_t] &= s_{t-1} + \frac{s_{t-1}}{n}(n - s_{t-1}) \\
E[s_t] &= \frac{3}{2}s_{t-1}
\end{aligned}
$$

With the same caveats on Chernov bounds that dogged the push algorithm, we can conclude that Phase 1 of pull takes $O(\log n)$ time. However, each step requires $O(n)$ messages, because most of the people are actively pulling. Thus, there is an inefficiency of $O(\log n)$ here.

In Phase 2 of the pull algorithm, let's say that $(1-\epsilon)n$ people know the rumor, and ϵn do not, where $\epsilon < 1/2$. Each person who does not know the rumor has a $(1-\epsilon)$ chance of learning the rumor in the next step. So the number of uninformed people shrinks as ϵn, $\epsilon^2 n$, $\epsilon^4 n$, $\epsilon^8 n$, et cetera. This means that Phase 2 takes $O(\log \log n)$ steps. Also, since the number of messages at each step shrinks exponentially, Phase 2 only takes $O(n)$ messages.

29.4 Push/Pull

The push and pull algorithms are each bad, because neither of them uses the ideal $O(n)$ messages. Ideally, we'd like to get these random-phone-call algorithms to be just as efficient as the deterministic algorithms. It is interesting that the push algorithm is inefficient in Phase 2, whereas the pull algorithm is inefficient in Phase 1. In fact, it is more than interesting, it is the grounds for the push/pull algorithm.

The idea behind push/pull is that we will push for a while, and then pull for a while. How will we know when to start pulling?

Even assuming each node has an internal clock that tells it how many steps have passed since a new update was introduced, we will need to be very exact. The transition will happen after $O(\log n)$ steps. But if we just wait $O(\log n)$ steps without worrying about the constant, we might have spent $O(\log n)$ too many steps. At $O(n)$ messages per step, that would make our algorithm inefficient. So we need to be very precise about the time to switch from pushing to pulling.

There is an extra phase to the push/pull algorithm. Phase 0 is the time when less than $\log^4 n/2$ people know the rumor. At step $t-1$, $s_{t-1} \leq \log^4 n/2$ people have heard the rumor. Analyzing the same way as for Phase 1 of the push algorithm, there are at most $\log^4 n$ that any given person could call that already know the rumor. Given s_{t-1},

$$
\begin{aligned}
E[s_t - s_{t-1}] &\geq (1 - \frac{\log^4 n}{n})s_{t-1} \\
E[s_t] &\geq (2 - \frac{\log^4 n}{n})s_{t-1} \\
Pr[s_t &\geq (2 - 2\frac{\log^4 n}{n})s_{t-1}] \geq \frac{1}{2}
\end{aligned}
$$

Thus Phase 0 will be complete in $O(\log\log n)$ steps.

Let Phase 1 be the time when $s_{t-1} \leq n/\log n$. For each person who does not know the rumor, the odds are $(1-\frac{1}{n})^{s_{t-1}}$ that this person will still not know the rumor at step s_t. This is a random variable, so the Chernov bounds are applicable. To simplify,

$$
\begin{aligned}
E[s_t] &\geq n - (1 - \frac{1}{n})^{s_{t-1}}(n - s_{t-1}) \\
E[s_t] &\geq s_{t-1}(2 - \frac{1}{\log n})
\end{aligned}
$$

This lets us be very careful about the growth of s_t. After exactly $\log_2 n$ steps, we can stop pushing. The error in our estimate is $O(\log\log n)$, so we push at most $O(n \log\log n)$ messages, for an inefficiency of $O(\log\log n)$. Of course, in practice we can just stop pushing after we have called some threshold number of people who already know the rumor.

Phase 2 of push/pull falls to the same analysis as Phase 2 of the pull algorithm. Since we already have an inefficiency of $O(\log\log n)$, we do not really even have to stop pushing now. We do need to stop pushing at some point, however, so it might as well be now.

To summarize, the push/pull algorithm takes $O(n\log\log n)$ messages to spread a rumor in $O(\log n)$ time. It is a factor of $O(\log\log n)$ more inefficient than a deterministic algorithm would be.

29.5 Other Results

The same analysis basically holds for other distributions, where you cannot call somebody at random but can only access other people according to some distribution.

The push/pull algorithm is fault-tolerant; if there are F faults after $O(\log n)$ rounds of the algorithm, at most $O(F)$ people will be denied access to the rumor. This is the best possible fault-tolerance, and far better than the fault-tolerance of the tree.

In this model, you can only communicate with another person at random. You cannot remember the location of anything, you can only pick up the phone and get connected to somebody. In this model, $\Omega(n \log \log n)$ messages are necessary to spread a rumor in $O(\log n)$ rounds. So in a sense, the push/pull algorithm is the best possible. Of course, this bound is only valid in this model, where the tree strategy is not allowed.

We disliked the tree algorithm because a single fault could knock out half the graph. It is possible to have a graph that is fault-tolerant, however. A hypercube, a decent initial guess, does not work, since slicing it in half along the diagonal takes $o(2^n)$ nodes to eliminate $O(2^n)$ nodes from the graph. We need expander graphs, which are graphs that obey

$$\forall S \subset V : |S| < \frac{|V|}{2}, |N(S)| \geq (1+\beta)|S|$$

This is exactly the property necessary for the graph to be fault-tolerant. However, if a small area S has its boundary cut out of the graph, then rumors will still not be able to travel from S to the rest of the graph. In this sense, the push/pull algorithm still has superior fault-tolerance.