Lecture 13

Clock Distribution

For most of this lecture series, we have focused on fully connected topologies. As discussed, this has some justification, as high resilience to (permanent) faults requires high connectivity. However, in many cases it may simply not be practical to have a fully connected system — e.g., having $n(n-1)/2$ links on a chip means that we quickly run out of (physical) space for all these wires!

Instead, we can use the fault-tolerance techniques provided so far to build a reliable clock source, and strive for a fault-tolerant distribution scheme. Here, we need to assume that the (permanent) faults in the system are not distributed in a worst-case fashion, but reasonably "spread out." With this assumption, we can hope for smaller degrees, as we won't end up with a situation in which the neighborhood of a correct node is "taken over" by a majority of faulty ones, effectively cutting the node off from the rest of the network. In the following, we assume that each neighborhood contains at most f faults (for some constant or at least small f).

Definition 13.1 (Local Faults). *We say that there are at most f* local faults *in a given graph (V, E) with correct nodes $V_g \subseteq V$, if each $v \in V_g$ has at most f neighbors in $V \setminus V_g$. If the graph is directed, we require that at most f in-neighbors are faulty.*

This lecture is mostly about discussing this fairly open problem. We give no definite solutions, and we will not formalize the presented ones carefully.

13.1 First Attempt: Clock Trees

We could build a fault-tolerant version of a clock distribution tree. Using the techniques we discussed in previous lectures, we can set up a highly robust clock source serving as root of the tree, and we replace each node in the tree by a *cluster* of $2f + 1$ nodes, whose (logical) clocks will just mirror what happens at the parent "node" in the tree. Concretely, this means to locally trigger and forward a clock flank — which we will refer to as *pulse* — when receiving the $(f+1)^{th}$ signal from the parent cluster. We end up with a very simple structure and node degrees of $\mathcal{O}(f)$, which is the best we can hope for. Self-stabilization is essentially for free, as we have a simple master-slave relationship; once the parent cluster pulses in synchrony and with the right frequency, its easy for the child cluster to start following suit.

Lemma 13.2. *If each node is faulty with uniform and independent probability of $p \in o(1/(fn^{f+1}))$ and (in-)degrees are at most $2f+1$, there are at most f local faults with probability $1-o(1)$.*

Proof. W.l.o.g., we assume that all nodes have in-degree $2f+1$, as the probability for having more than f faulty neighbors decreases with the degree (and all random choices are independent). For a single node, the probability that more than f of its neighbors are faulty is bounded by

$$\sum_{f'=f+1}^{2f+1} \binom{2f+1}{f'} p^{f'}(1-p)^{2f+1-f'} \leq (f+1)\binom{2f+1}{f+1} p^{f+1}$$
$$\leq ((2f+1)p)^{f+1} \in o\left(\frac{1}{n}\right).$$

By the union bound, the overall probability of having more than f local faults is thus bounded by $\sum_{i=1}^{n} o(1/n) = o(1)$. □

Corollary 13.3. *Assume that each node is faulty with uniform and independent probability of $p \in o(1/(fn^{f+1}))$ and other system parameters (like delay and uncertainty) are identical to the clock tree from which the "fat tree" described above is derived. Then, with probability $1-o(1)$, each correct node produces clock pulses within the same worst-case time bounds as the corresponding node in the original tree.*

Proof. By Lemma 13.2, with probability $1-o(1)$ there are at most f local faults. Thus, any (non-root) node will disregard faulty nodes' signals unless they lie within the interval spanned by correct nodes' signals. As delays, etc. match those of the original tree, the resulting worst-case bounds are identical. □

Is the resulting solution good? It depends on the the system, but we can argue that it doesn't scale well. We know from the first lecture that the (worst-case) skew between tree nodes is proportional to their distance in terms of the considered graph. Sticking to (traditional) computer chips, a clock tree needs to provide the clock signal to a roughly quadratic area, where we can expect that at the very least physically close-by parts of the chip need the signals provided to them to be well-synchronized. At least for some nodes, a tree must fail to do so.

Lemma 13.4. *For $k \in \mathbb{N}$, consider a $k \times k$ grid in $\mathbb{R}^2$ in which adjacent grid nodes have distance 1. For any tree spanning the grid points, there are adjacent grid nodes that are in distance $\Omega(k)$ in the tree.*

Proof. Observe that if a tree node v has degree larger than 3, we can reduce its degree by inserting an additional node arbitrarily close to it and attaching 2 or more of the children of v to the new node instead. This changes tree distances by an arbitrarily small amount. Accordingly, we can w.l.o.g. assume that the tree is binary.

In any tree T of maximum degree 3 (of at least 4 nodes), there is some edge so that the two components resulting from removing this edge have size at least $(n-1)/3 \in \Omega(n)$. To see this, pick an arbitrary node edge, delete it, and look at the resulting components T_1 and T_2. If $|T_1|, |T_2| \geq (n-1)/3$, we're done.

Otherwise, assume w.l.o.g. that $|T_1| < (n-1)/3$, i.e., $|T_2| \geq n - (n-1)/3 = 2(n-1)/3 + 1$. Let w be the endpoint of the deleted edge that lies in T_2. Deleting w from T_2 results in (at most) two components of T_2, as w has degree 3 (and thus at most 2 in T_2). One of these components must have size at least $(|T_2|-1)/2 \geq (n-1)/3$. Consider the edge connecting w to this component and delete it from T, resulting in components T_1' and T_2'; let's say $w \in T_1'$. By the previous considerations, we have that $|T_1'| > |T_1|$, while $|T_2'| \geq (n-1)/3$. Now either $|T_1'| \geq (n-1)/3$ and we're done, or we can repeat the argument, resulting in an edge for which one component is even larger than T_1' and the other remains of size at least $(n-1)/3$. Thus, repeating this argument inductively, we must eventually reach an edge satisfying the claim.

From the above claim, we can infer that we can partition the nodes into two sets such that (i) each set contains $\Omega(k^2)$ nodes and (ii) each set induces a subtree. We call a node a *boundary node* if it has a neighbor in the other set. From (i) we can infer that the boundary must contain nodes in distance $\Omega(k)$ from each other, as any area of size $\Omega(k^2)$ must have a boundary of size $\Omega(k)$ (even when not considering nodes at the boundary of the entire grid). Fix two such nodes v and w in the same subtree. As they are in distance $\Omega(k)$, the path in the subtree connecting them is of that length. This in turn means that at least one of them is in distance $\Omega(k)$ of the root of its subtree. Finally, we conclude that this node is in distance $\Omega(k)$ from its neighbor(s) in the other set within the tree. □

Remarks:

- If uncertainties are proportional to the length of a link, we can immediately conclude that the worst-case skew between adjacent nodes is $\Omega(uk)$, cf. Theorem 1.6.

- One can rely on probabilistic guarantees instead. However, even if things behave "nicely," we still end up adding up variances in link delays, resulting in standard deviation $\Omega(u\sqrt{k})$ (if a unit length link has standard deviation u).

- There are quite a few tricks electrical engineers came up with in order to deal with the (few) long links of an H-tree (and relatives) in a better way, but ultimately physics gets in the way of scaling tree topologies arbitrarily.

- The above bound is tight up to constants, which is shown by an H-tree. For $k+1$ (the "width" of the grid) being a power of 2, an H-tree is constructed recursively as follows. Place the root in the center of the grid. Then connect it to two children by going $k/2$ to the right and left, respectively. Each of these children also has two children, which are in distance $k/4$ going up or down respectively. The four nodes in depth two of the tree are now exactly in the center of four disjoint subgrids of $k/2 \times k/2$ nodes, and the construction is applied recursively for $\log k$ steps. In the end, each grid point with both x- and y-index being odd (or even, depending on indexation) is occupied by a leaf.

- The construction from the lemma actually shows that there must be $\Omega(k)$ adjacent grid nodes that are in distance $\Omega(k)$ in the tree. More generally,

one can show that $\Omega(2^i k)$ adjacent grid nodes are in distance $\Omega(k/2^i)$ in the tree.

- An H-tree matches this bound, too: Cutting the square in half horizontally and vertically, one gets four subsquares, each of which hosts a smaller H-tree. Where we cut the grid, we have $2k$ node pairs that end up being in distance (almost) $2k$ in the tree. All other node pairs are in the same of one of the four sub-H-trees, so they are by factor 2 closer to each other.

- All this is only useful when faults do not "cluster" within, well, clusters. The assumption that faults are completely independent is unrealistic, but it's crucial to design the system to make this an approximate reality. Of course, this also applies to the techniques from earlier chapters — a system-wide power failure will bring down any algorithm, regardless of how many node failures it can tolerate.

- Fault-tolerance and self-stabilization (under the assumption of at most f local faults) become easy due to the fact that we don't have any *cyclic* dependencies. In other words, we don't have to restrict ourselves to trees — DAGs are perfectly fine!

13.2 Second Attempt: Lynch-Welch on DAGs

The same strategy we used in Chapter 5 can be extended to directed acyclic graphs. Assuming f-local faults, we make sure that each node (except "source" nodes, which are supplied with a clock signal), have at least $3f+1$ in-neighbors. We then get the following property.

Corollary 13.5. *For $v, w \in V_g$, assume that both of them have the same set N of in-neighbors, where $|N| \geq 3f+1$ and $|V_g \cap N| \geq |N| - f$. Suppose each node $x \in N_g := V_g \cap N$ pulses once at time p_x and v, w interpret the respective messages as in Line 5 and that T and $\mathcal{S}$ are sufficiently large. Then v and w pulse at times p_v and p_w such that $|p_v - p_w| \in \mathcal{O}(\|\vec{p}\|/2 + (\vartheta - 1)T + u)$.*

- We can use this to adapt the earlier approach of fat trees to guarantee that local skews *with respect to the tree topology* and any given pulse remain bounded, regardless of the depth of the tree. However, this does not fix the problem of large skews between different branches of the tree.

- As stated, we can use this on more general DAGs. We could, e.g., take two clusters and let them have a "common" child. In this case, the $3f+1$ nodes of the child cluster may exhibit a larger skew, though, as the bound on the skew between the two parent clusters may be much larger than within each parent cluster.

- This leads to the following open problem: Is there a clever choice of a DAG that avoids the issues of trees, i.e., it can clock a two- (or three-)dimensional area with small local skews (with respect to physical distances)?

13.3 Third Attempt: Fault-tolerant GCS

Trying something else, we could seek to simulate the (non-fault-tolerant) GCS algorithm from chapter 2 on an arbitrary topology. In fact, node degrees of 3 are sufficient to cover any structure without "misrepresenting" physical distances too much.

The general idea is to have clusters of size $3f+1$, which each may have up to f faulty nodes, that we synchronize internally with the Lynch-Welch algorithm. However, these clocks will not be the hardware clocks of the GCS algorithm—they will be the logical clocks! Concretely:

- The output of the Lynch-Welch algorithm at a (correct) node can be seen as (up to the skew) accurate representation of the cluster's logical clock. The node will communicate this clock to all adjacent clusters.
- At the same time, it is the node's logical clock of the Lynch-Welch algorithm, which is set up to handle the increased clock drift of a logical clock of the GCS algorithm, i.e., it assumes that the underlying "hardware" clock has drift $\vartheta(1+\mu)$.
- The GCS algorithm also takes into account a larger "hardware" clock drift, as it needs to account for the clock corrections of the Lynch-Welch algorithm within clusters. By amortizing these changes over $\Theta(T)$ time, this implies a "hardware" clock drift of $\mathcal{O}(\vartheta(1+\mu)+u/T)$, and by choosing T large enough and recalling that $\mu \in \mathcal{O}(1)$, we can bound this by $\mathcal{O}(\vartheta)$. In other words, if ϑ is small enough, both algorithms are able to handle each other's clock manipulations.
- Now each node can read the own cluster clock (by looking at its own clock) and that of adjacent clusters (by looking at all nodes there and, e.g., taking the median value) with an error of $u+\mathcal{S}$ (plus possibly a term for refreshing this information only infrequently), where $\mathcal{S} \in \mathcal{O}((\vartheta-1)d+u)$ is the skew of the Lynch-Welch algorithm within each cluster.
- This yields a $\delta \in \mathcal{O}((\vartheta-1)d+u)$ for the GCS algorithm, i.e., the same asymptotic guarantee as for the original GCS algorithm.
- To control the global skew, one could, e.g., use a tree structure as discussed above, employing the strategy from Task 1 of the second exercise sheet.

Remarks:

- Take this description with a grain of salt; it's an idea for an approach that has not been proven correct yet.
- The failure condition of the system (more than f faults in a cluster) is slightly different from f-local faults, but the asymptotics in terms of the failure probability that can be sustained are the same.
- Even for $f = 1$ and a maximum degree of 3 of the original graph, the minimum node degree is $3f + 3(3f + 1) = 12f + 3 = 15$, where all these links are bidirectional. One could reduce this to $3f + 3(2f + 1) = 12$ by arguing that $2f + 1$ values suffice to "read" a cluster clock (the median

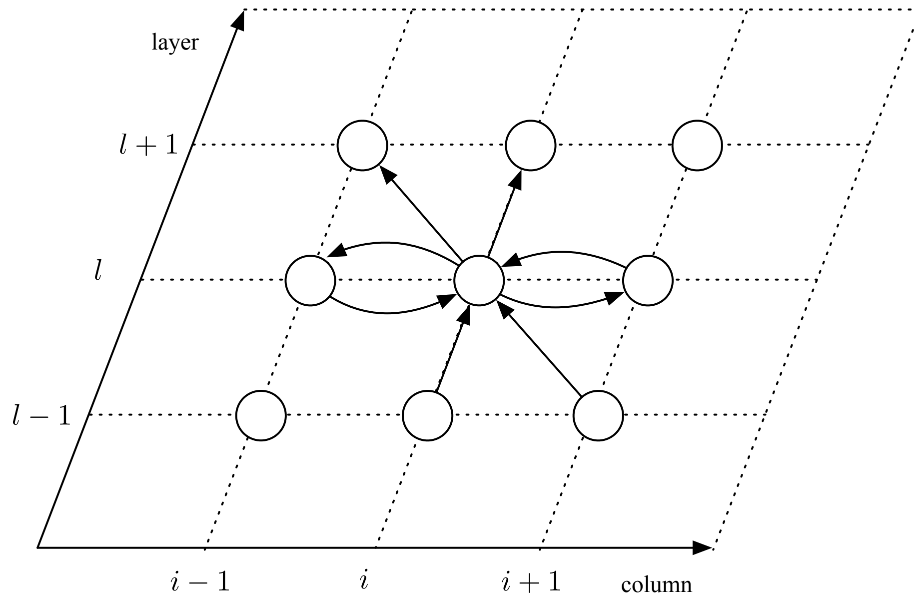

Figure 13.1: Structure of the HEX grid.

value will be in the range of correct nodes' values), but this comes at the cost of having less accurate readings.

- If all of this works out, efficiently adding self-stabilization is still an open problem. We have discussed how to do this for the Lynch-Welch algorithm, but we also need to make sure that clock values are communicated correctly between clusters. At least when considering a hardware implementation, communicating (and performing computations with) encoded clock values, as opposed to just sending clock "ticks," might be rather expensive.

- When all of this is done, despite the good asymptotics, we end up with awfully complicated nodes. Even if we can afford the energy and area for this, it means that our nodes are more likely to fail than simpler ones. The gains in reliability may be small or we may even end up with a *less* reliable system! We may need much simpler solutions!

13.4 Fourth Attempt: HEX

With this issue in mind, we looked for a very low-degree (for convenience also planar) topology that can tolerate one local fault, with an extremely simple algorithm.

The idea is that nodes are organized in layers, where the nodes in the initial layer are provided with a clock signal by some other means. A node locally triggers and forwards a pulse when it has received messages from two in-neighbors. Byzantine nodes cannot trigger spurious faults if we have 1-local faults. And they also can't prevent nodes from triggering their pulse, as an in-neighbor in

the same layer will provide a pulse signal in case an in-neighbor in the previous layer is faulty (this is not completely trivial, but easy to show).

With reasonable effort, the approach can also be made self-stabilizing, by ensuring via some timeouts that the system will recover layer by layer once the clock source works correctly. There are also elaborate (laborious?) proofs showing that HEX has a worst-case skew of $d + \mathcal{O}(\min\{L, W\} \cdot u^2/d)$ between neighbors in the same layer, where L is the number of layers and W the "width" of the grid, respectively, granted that skews at initial layer are 0 (non-zero skews are accounted for by the bounds as well). Moreover, the same bound holds between neighbors from different layers when shifting the output clocks by an additive d per layer to account for delays. Despite all this, HEX suffers from a rather basic flaw.

Observation 13.6. *Even with a single crash fault, perfect input (i.e., skew* 0 *on the initial layer), and* $u = 0$*, correct neighbors in the same layer may exhibit a skew of* d*.*

Proof. The crashing node will cause its out-neighbors on the next layer to trigger a pulse d time later, yet all other nodes on this layer will pulse at the same time as they would without the fault. □

You might argue now that the bound we had already has an additive d in it, and you would be correct — from the point of view of a worst-case analysis. In contrast, things look very different when assuming that delays do not behave in a worst-case fashion.

Getting a better understanding of this is an art. First, one needs to determine how delays behave if they are not chosen adversarially (meaning that we just make sure that this does not matter). As a first stab at the task, one may be optimistic: we assume that each delay is chosen independently and uniformly at random from $(d - u, d)$. Second, the resulting distributions appear to be very hard to analyze mathematically. This is owed to the fact that they are the result of operations involving both taking the minimum and maximum, disqualifying straightforward application of concentration bounds and other basic probabilistic tools.

We ended up simply study the grid under the above assumptions by computer simulation. The result was that, if there are no faults (and skews at the initial layer are small), the grid performs astoundingly well, without ever needing any of the communication links between nodes in the same layer, see Figure 13.2. We observe skews of $\mathcal{O}(u)$, where $u \ll d$ implies that the worst-case bounds are not all that informative. In a way, the grid performed *too well* for the approach to fault-tolerance to make sense!

Remarks:

- The name "HEX" originates in the hexagonal structure of a node's neighborhood in what appears to be the most natural physical layout.

13.5 Fifth Attempt: TRIX

The above insights suggest an even simpler topology, in which each node has 3 in- and outneighbors each, and triggers a pulse when receiving the second pulse on its incoming links.

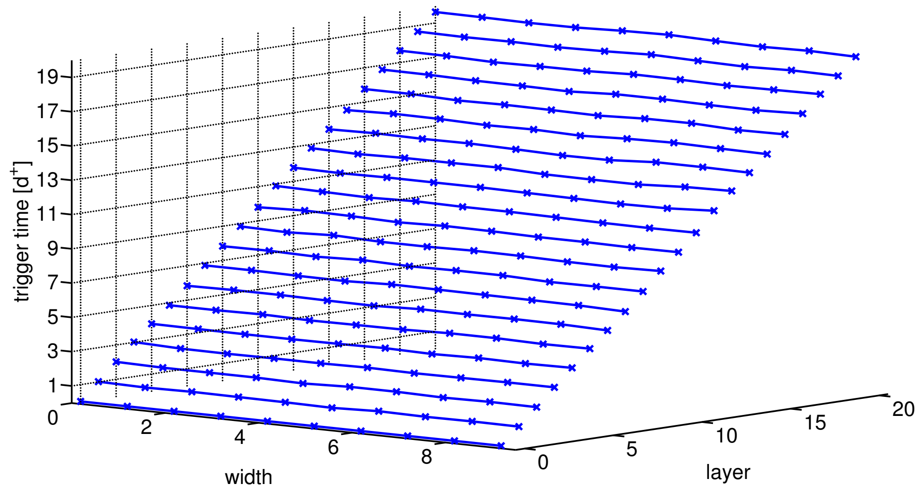

Figure 13.2: A pulse of a fault-free HEX grid with uniformly random delays ($d^+ = d$). It's easy to see that skews between neighbors remain far smaller than d, meaning that the links within a layer never contribute to triggering pulses.

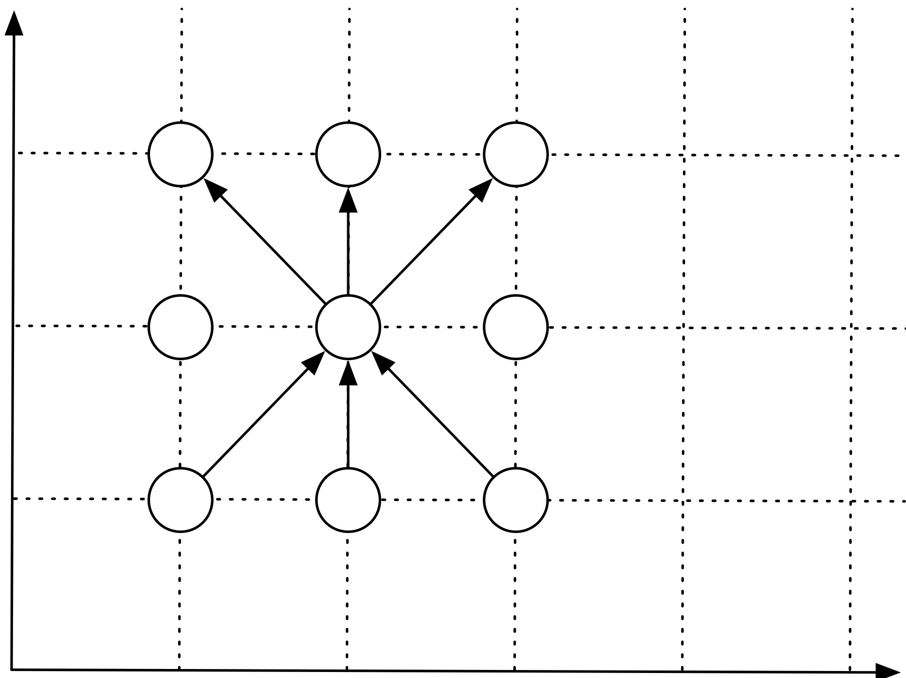

Figure 13.3: Structure of the TRIX grid.

Note that from the point of view of a worst-case analysis, this solution is rather bad: If all message delays are small on one "side" of the grid and large on the other, a skew of up to u per layer can be built up, which is by a factor of (up to reasonably small constants) d/u worse than HEX — except that HEX suffers from an additive d in its bound. Up to d/u layers, HEX thus doesn't really look better on this front, requiring at least 100 layers before these asymptotics could kick in.

But these are worst-case bounds, which the above considerations suggest to not matter as much as one might think. An important advantage of TRIX over HEX is that even when a node fails, this does not cause any of its out-neighbors to be triggered significantly later. As under the assumption of 1-local faults any node has two in-neighbors on the previous layer, it will still trigger its pulse between $d - u$ and d time after (at least) one of its correct in-neighbors.

In some ways, TRIX is much easier to analyze: self-stabilization is almost trivial, as there are no interactions between nodes in the same layer. The worst-case skew bounds are essentially obvious. The effect of faults on skews can be easily bounded by u for any given fault, and one can easily suspect that faults have no significant effect on far-away parts of the system (so long as they are 1-local). But when it comes to the behavior of skews under u.i.r. link delays, the situation is equally embarrassing as for HEX.

Here are some results from computer experiments, where for simplicity link delays are either 0 or 1 with (independent) probability of 1/2 each, and there is no skew in the first layer.

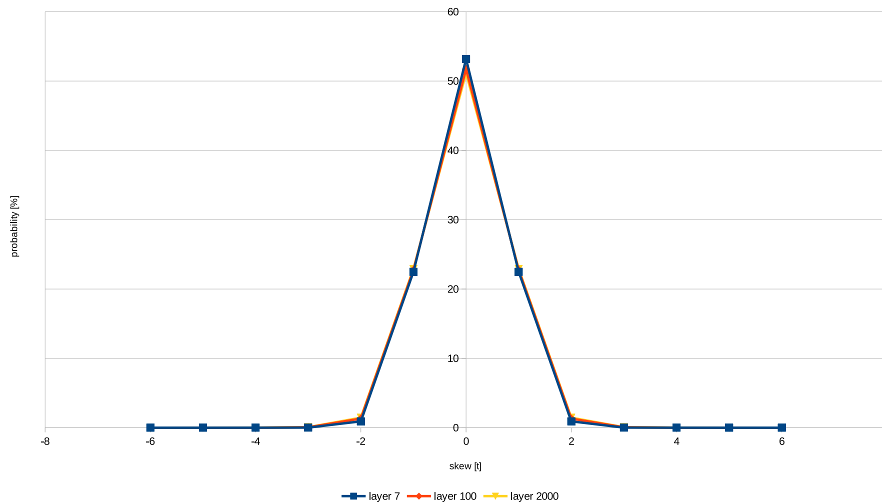

Figure 13.4: Distributions of skews between neighbors in layers 7, 100, and 2000 for TRIX with random 0-1-delays. This suggests a binomial (or related) distribution with extremly small standard deviation, where the number of layers has very limited influence.

Remarks:

- The name "TRIX" indicates that in- and outdegrees are three (and plays on "HEX").

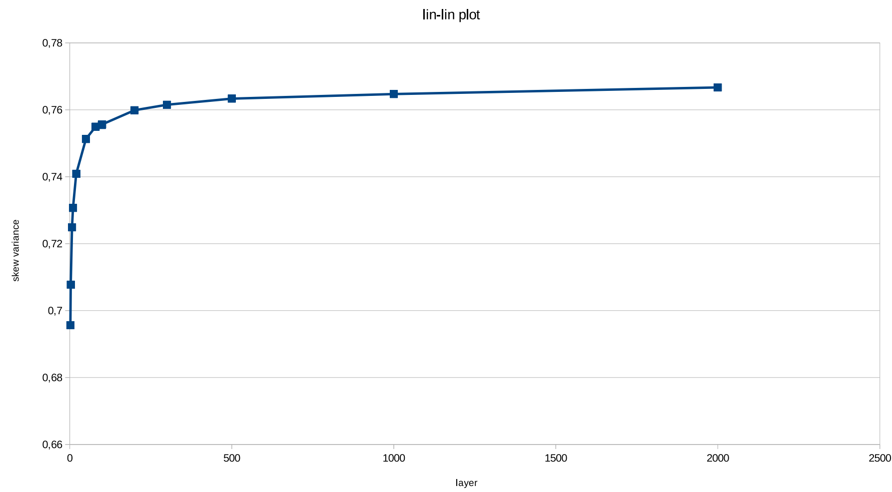

Figure 13.5: Plot of the variance of the above distribution as function of the number of layers. Does it grow very slowly, but is unbounded, or does it converge to a fixed value corresponding to a limit distribution?

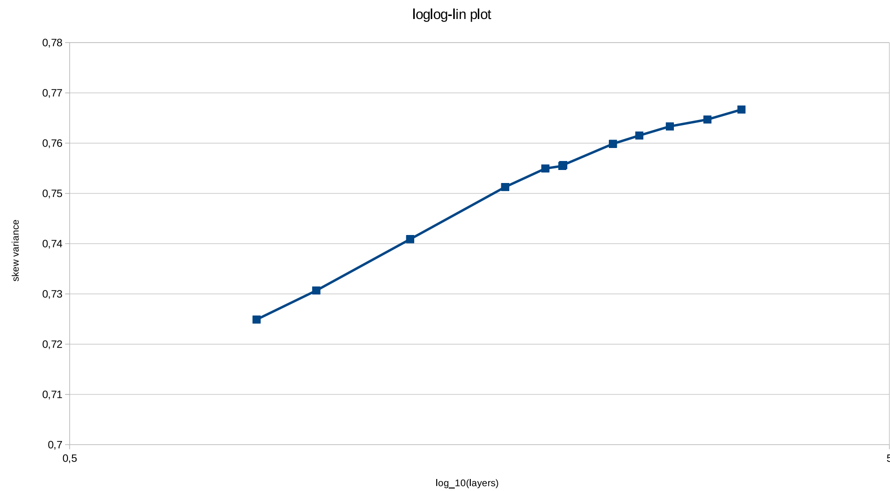

Figure 13.6: The same plot, but with a doubly logarithmic x-scale. One may suspect that the growth is bounded by $\log \log L$, where L is the number of layers. However, the considered range of values is rather small for this (especially given the very small change of the actual y-values), and the number of experiments may be insufficient for a reliable assessment.

- We do not understand why the TRIX grid appears to work so extraordinarily well with randomized link delays.
- Do the skews just grow very slowly, or converge to a limit distribution?

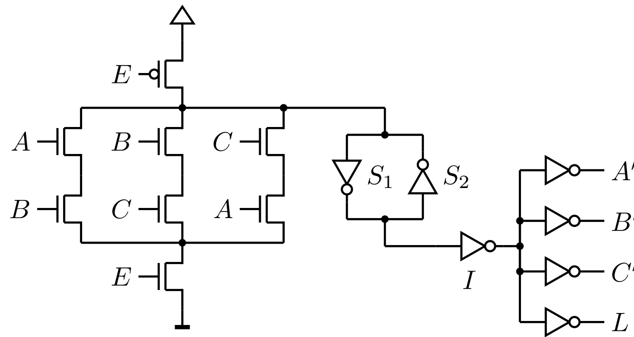

Figure 13.7: Circuit implementing the critical path in triggering a pulse. The "gates" connected to signals A, B, C, and E are transistors; you can think of them as switches that are "open" (i.e., conduct) if the input is high (a.k.a. 1) and "closed" (i.e., have almost infinite resistance) if the input is low (a.k.a. 0). E is an enable signal that is controlled in a non-time-critical manner by additional (local) circuitry. At the beginning of a cycle, enable is high, i.e., the connection to ground (in the bottom) is open and the connection to the supply voltage (top) is closed; the storage loop formed by the inverters S_1 and S_2 has low output, meaning that the outgoing (A', B', and C') and local (L) output signals are low. As soon as at least two out of the three incoming signals (A, B, and C) become high, this pulls the storage loop input to low (i.e., ground), and the outputs switch to high. The inverter I has a high threshold, meaning that S_1, S_2, and I essentially form a masking register. Thus, no matter what a faulty node provides as input, the output transition will be clean and occur within the interval spanned by the correct in-neighbors' signal transition times (plus delay). E is then controlled via L in a way that holds the output signals high for long enough for the outneighbors to react and for the incoming correct signals to go low again, and then is set to low in order to "reset" the circuit to the initial state, readying it for the next pulse.

- Is the grid even reducing larger skews effectively?
- For a practical realization, one has to further adapt the topology. Concretely, the clock source should not be a (wide) intial layer, but rather a few nodes. This suggests a layout with layers being nested circles, adding a constant number of nodes per layer. These and other concerns will affect a final design, requiring further studies.

TRIX and Metastability

TRIX nodes are extremely simple, but that doesn't mean that we can ignore metastability issues. Given that we assume Byzantine faults, we should better make sure that they don't break the abstract pulse triggering logic we've been thinking about. Synchronizers are not an option here, as (forcibly) aligning the incoming pulses to some local clock would significantly increase skews. There are unclocked circuit elements with similar function, but we can take a more direct approach by designing a suitable circuit element for the (time-)critical path on the transistor level.

Remarks:

- This is conceptually very similar to the Strategy shown in Figure 7.6, except for the addition of the loop capturing the transition. The storage loop ensures that a Byzantine node cannot, e.g., wait until the first correct signal transition arrived, then pull the output high (using its own input), and then pull it down again before the second correct transition arrives.

Bibliographic Notes

Clock trees have been the standard clocking method for decades, and they still are for many systems or at least subsystems. They have been heavily engineered, and in fact most clock "trees" are not real trees any more, for instance due to a grid-like structure connecting the leaves to reduce skews. It's difficult to find comprehensive and up-to-date literature, as most developments nowadays occur within companies, which are not eager to share their know-how. See [X09] for some introductory material.

To my knowledge, neither have variants of the Lynch-Welch algorithm on not fully connected graphs been studied, nor have attempts at making GCS algorithms fault-tolerant been made in the literature; fault-tolerant clock distribution methods — at least in terms of mathematical proven guarantees — appear to be essentially unchartered territory. For results on HEX see [DFL+16]; no publications discussing TRIX exist at the time of writing.

Bibliography

[DFL+16] Danny Dolev, Matthias Függer, Christoph Lenzen, Martin Perner, and Ulrich Schmid. HEX: Scaling Honeycombs is Easier than Scaling Clock Trees. *J. Comput. Syst. Sci.*, 82(5):929–956, 2016. Preprint available at https://people.mpi-inf.mpg.de/~clenzen/pubs/DFLPS16scaling.pdf.

[X09] Thucydides Xanthopoulos. *Clocking in modern VLSI systems*. Springer Science & Business Media, 2009.