Adaptive Counting Networks

Srikanta Tirthapura
Department of Electrical and Computer Engineering
Iowa State University, Ames, IA 50011
snt@iastate.edu

Abstract

Counting networks are well studied parallel and distributed data structures, which are useful in synchronization applications such as distributed counting and load balancing. However, current constructions of counting networks are static*, since their width (the degree of parallelism), and hence the size of the network, have to be fixed in advance. This present an obstacle in efficiently implementing them in a large distributed system whose size may be changing, due to nodes joining and leaving the network.*

We present an adaptive construction of the bitonic *counting network. Our network tunes its width to the system size in a distributed and local way.*

- *With high probability, the effective "width" of the network is $\Omega(N/\log^2 N)$, where N is* the number of nodes currently in the system*, and the effective "depth" of the network is $O(\log^2 N)$. In contrast, a static implementation would have the same width irrespective of the system size.*

- *When the system size changes, the network adapts by splitting or merging its components. All decisions and actions are decentralized: these include the decision of when to split and merge the components, and the action of splitting and merging them.*

Our construction is layered on an overlay network which provides an efficient peer-to-peer lookup service, and uses the recursive structure present in the bitonic network to adapt its implementation. Though we discuss the bitonic network, our technique could be applied to build an adaptive implementation of any distributed data structure which can be decomposed in a recursive way.

1. Introduction

Counting networks [AHS94] are well studied distributed data structures which are useful in distributed counting, load balancing, and other synchronization applications. Counting networks route tokens from input to output wires, through many layers of simple computing elements called balancers. They ensure that irrespective of the distribution of tokens on the input wires, the tokens are always uniformly distributed across the output wires to the maximum extent possible. The construction and properties of counting networks have been studied in detail [AHS94, AVY94, HT03, KP92, Wat98].

However, all currently known constructions of counting networks are *static*. The degree of parallelism (the "width" of the network), and the number of processing elements (balancers) required to implement have to be fixed in advance, and cannot change with time. In a distributed system, however, the set of processors can change over time. A small value of the width might be efficient for a small system size, but will not provide enough parallelism if the system becomes larger. Conversely, choosing a large width will lead to unnecessary overhead for small system sizes, due to the large number of processing elements that would be needed. Ideally, the degree of parallelism of the counting network should adapt to the system size and load.

We present such an adaptive construction of counting networks, which can scale with the system size. The basic units of our construction are variable width *components* as opposed to fixed width *balancers*. When the system size increases, the number of components implementing the network increases by "splitting" currently existing components, and when the system size decreases, the number of components decreases, by "merging" different components together.

We layer our implementation of counting networks on top of an overlay network which provides an efficient lookup service, such as provided by many of the current peer-to-peer systems[PRR99, HKRZ02, SMK$^+$01, RFH$^+$01].

Our adaptive counting network has the following properties. Let N denote the current number of nodes in the system.

- With high probability, the effective width (degree of parallelism) of the counting network is $\Omega(N/\log^2 N)$, and the effective depth (a measure of the latency) is $O(\log^2 N)$.

- With high probability, the total number of components in the counting network is $O(N)$. The expected number of components that are mapped to a node is $\Theta(1)$ and the maximum number of components on any node is $O(\log N/\log\log N)$ with high probability.
- When the system size changes, the counting network adapts by splitting or merging its components. All decisions and actions are decentralized: these include the decisions of splitting and merging the components, and the actions of splitting and merging them.

In contrast, the effective width and depth of a static implementation does not change with the system size, and is thus independent of N. For a width of w, the bitonic and periodic networks have depths of $O(\log^2 w)$, and use a total of $O(w \log^2 w)$ balancers.

1.1. Counting and Balancing Networks

Counting networks have been built out of simple computing elements called *balancers*. A *balancer* is an asynchronous switch with two input wires and two output wires. The two input wires are labeled 0 and 1 and the two output wires are labeled 0 and 1. A balancer accepts a stream of tokens on its input wires, and the i-th token entering the balancer leaves on output wire $i \bmod 2$. A *balancing network* is an acyclic network of balancers where output wires of some balancers are linked to input wires of others. Counting networks are balancing networks with the following *step property* w.r.t. the distribution of the tokens across the output wires.

Let w denote the number of output wires of the balancing network. Let $X = x_0 x_1 \ldots x_{w-1}$ denote the sequence of the numbers of tokens that are emitted out of output wires labeled $0, 1, \ldots (w-1)$ respectively. A network is said to be in a *quiescent* state if every token that has entered the network has also left it. A balancing network is a *counting network* if in every quiescent state of the network, the sequence X satisfies the following condition: for every $0 \leq i < j \leq (w-1)$, it is true that $0 \leq x_i - x_j \leq 1$.

Applications A counting network can be used in a scalable implementation of a distributed counter object. In a large scale distributed system, a counting network can be used to generate consecutive token numbers on demand in a parallel and distributed manner.

A counting network can also be used to build a distributed data structure which can match producer resources with consumer resource requests. In this application, consumers may asynchronously generate "request tokens" requesting use of a resource, and producers may asynchronously generate "supply tokens" whenever resources are available for use. The word "resource" is used in a general sense: for example, they could be CPU cycles, or a permission to access a service. Synchronization is required to ensure that each request token is matched with exactly one supply token (if enough supply is available), and vice-versa. As illustrated in [AHS94], this producer-consumer matching problem can be solved by using two back to back counting networks, one for producers and the other for consumers.

1.2. Overview of Solution

We focus on the *bitonic* counting network [AHS94, Bat68], but the same technique can be used for any distributed data structure which can be decomposed recursively. Our solution has two parts.

Firstly, we use the inductive structure present in the bitonic network to increase or decrease the number of components implementing the network, whenever the nodes decide to. Initially, the entire bitonic network resides on one node, as a single component. If the system size increases, and more parallelism in necessary, then the bitonic network splits into "components" and these components are re-mapped to different nodes and connected by (application level) wires. These components can further split into smaller components, and so on. When the nodes sense that the system size has decreased, then many smaller components can be merged together to form a larger component, and this is also a local action involving only those components being merged. The mapping of the components to the nodes is done using the distributed hash function provided by the underlying layer.

The second part of our solution is the distributed decision about when to split and merge components. Each node estimates the appropriate level of parallelism for the network using an estimate of the number of nodes in the system. Knowing the system size exactly is infeasible in a large system. However, it is possible to obtain good estimates in a local way, if the nodes in the system have random identifiers, as is the case in most structured peer-to-peer systems. Each node in the system uses its local estimate of the network size in making a decision on splits and merges.

Combining the two, we find that the decentralized splitting and merging decisions of the different nodes are coordinated enough so that we get a network whose effective width and depth closely match the width and depth of the "ideal" counting network that one could statically construct for the current system size.

1.3. Related Work

Diffracting Trees [SZ96] are another class of balancing networks, where the balancers are organized as a tree, and whose outputs also have the step property. The original diffracting trees also had the problem that no single set of parameters (such as the the size of the tree) worked well across a range of system loads. To remedy this, *reactive diffracting trees* [DLS00] were proposed, which can grow and shrink as the system load varies. Recently, *self-adjusting trees* [HHPT03] have been proposed, which can react faster to changes in system size, and which need less manual configuration.

The above work is focused on the shared memory model, while we assume a message passing distributed system. Diffracting trees depend on efficient "diffraction through a prism" for ensuring that there is no contention at the root of the tree. This involves the choice of a good waiting strategy at the nodes implementing the prism. The counting networks that we consider do not have a single root node. Tokens can enter the counting network at one of many input nodes and hence counting networks do not face the problem of contention at the root.

In order to construct efficient distributed hash tables (DHTs), many systems and algorithms overlay parallel interconnection architectures over peer-to-peer networks. The Viceroy system [MNR02] overlays a Butterfly network, while the Chord system [SMK$^+$01] and the algorithm of Plaxton et. al. [PRR99] use Hypercubes in their overlays. Other work in building overlay networks include [AS03, MBR03, KK03]. Our counting network construction differs in its goal from the above. The above are built for purposes of routing in a DHT, while we assume the presence of a routing facility, and aim to build distributed structures on top.

Many counting network constructions have been proposed. The bitonic and periodic networks proposed in [AHS94] are isomorphic to the bitonic sorting network[Bat68] and the periodic sorting network[DPRS89] respectively, and both have $O(\log^2 w)$ depth (where w is the width). Constructions of smaller depth for the same width were proposed in [KP92], who showed the existence of counting networks of depth $O(\log w)$ and gave explicit constructions of networks of depth $O(c^{\log^* w} \log w)$ (where c is a constant) using deterministic balancers. The constructions in [KP92] are however not practical, since the hidden constants are very high. The above constructions do not consider the problem of reconfiguration for changing system sizes.

1.4. Model

We assume an underlying routing service which provides efficient routing to an object given the object's name, through a distributed hash table [SMK$^+$01, PRR99, RFH$^+$01, MNR02] . We also assume that the processing nodes are assigned random identifiers, as is the case in most structured peer-to-peer networks; for example, [SMK$^+$01, RFH$^+$01] make this assumption. The adaptive counting network is overlaid on top of this routing layer.

The adaptive counting network is a directed acyclic graph whose vertices are components (a component is defined precisely in the next section) and whose edges are sets of wires between components. The input layer of the network is the set of components which accept tokens and have 'input wires pointing into them, and the output layer of the network is the set of components from which the tokens are emerge. The width of a counting network is the number of input wires, which equals the number of output wires for the bitonic network.

The width of the network is a limit on the maximum parallelism that the network can achieve, but says nothing of the current level of parallelism in the network. To capture the current degree of parallelism and latency in using the adaptive counting network, we define the *effective width* and the *effective depth* of an adaptive network.

Definition 1.1 *The* effective width *of an adaptive network is the number of vertex disjoint paths from the input layer components to the output layer components.*

Definition 1.2 *The* effective depth *of an adaptive network is the length of the longest path from a component in the input layer to a component in the output layer.*

If N is the number of nodes in the network, we show that the expected number of components per node of the network is $O(1)$, the effective width of the network is $\Omega(N/\log^2 N)$, and the effective depth is $O(\log^2 N)$ (the above two also hold with high probability).

How to use this network Each of the input components of the bitonic network has a unique name (the exact naming scheme is described in the next section), and the client can send tokens to any input component. Due to network reconfiguration, it is possible that the set of input components will change with time, but we show in Section 3 that the client has to try no more than $\log w$ different names before it can find an input component which currently exists in the network (assuming that the structure of the counting network does not change during the lookup process). The client tokens are routed through the network and exit some output component with a correct counter value.

Road map In Section 2, we show how we use the inductive structure of the bitonic network in building our adaptive network. In Section 3, we describe how the nodes make distributed decisions about splitting and merging components, and prove the depth and width properties of our adaptive network.

2. The Bitonic Network Implementation

We will first consider a simple implementation of the bitonic network on a peer-to-peer system. We first decide upon a width $w = 2^k$, for some $k \geq 1$, based on an estimate of the parallelism necessary for the network. A bitonic network of width w has $\frac{w \log w(\log w+1)}{4}$ balancers. In the simple approach, we implement each balancer as a separate object. Each balancer object is assigned a unique name which can be constructed from its coordinates in the bitonic network; all input balancers get names in the range $(0,0),(0,1)\ldots(0,w-1)$, and all components at depth i get names in the range $(i,0)\ldots(i,w-1)$. The balancer with name b is mapped to node $h(b)$ where h is the distributed hash function that is provided by the underlying system.

Each balancer knows the names of its two out-neighbors, and can obtain their locations through applying the hash function. Tokens arriving into the balancer are sent alternately to the out-neighbors, with the first token going into the "top" out-neighbor.

The problem with this implementation is that it always uses $O(w \log^2 w)$ objects irrespective of the underlying system size. Suppose we had set the width $w = 100$, expecting the system to grow to up to 500 nodes. There would be about 1000 balancer objects implementing this network. If the actual number of nodes currently in the system is 50, then a centralized low parallelism implementation might be the best choice, and a distributed implementation might be unnecessary overhead.

2.1. Components

We now describe our adaptive bitonic network, whose size and effective width can change with time. Let w denote the width of the counting network, which is a limit on the maximum possible parallelism the network can achieve. We will construct an adaptive implementation of the BITONIC$[w]$ network.

We first inductively define the notion of a *component* of the network. A component of width k has k input wires and k output wires. A BITONIC$[w]$ is a component. From the recursive structure of BITONIC$[w]$, as shown in Figure 1, we arrive at a recursive decomposition of a component into smaller components.

The BITONIC$[k]$ component ($k = 2^i, i > 1$) can be divided into six smaller components, the top and bottom BITONIC$[k/2]$ components, the top and bottom MERGER$[k/2]$ components and the top and bottom MIX$[k/2]$ components. The connections between the various components are as follows.

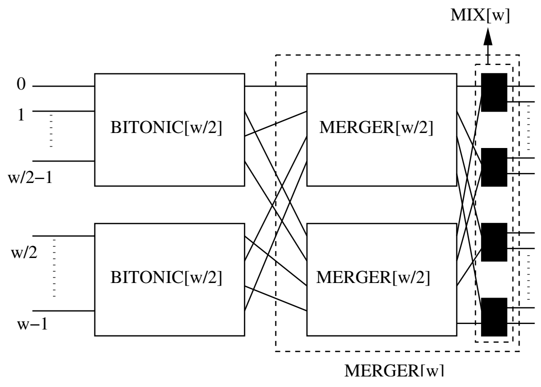

Figure 1. The recursive structure of the BITONIC$[w]$ network

- The input wires to BITONIC$[k]$ are split between the top BITONIC$[k/2]$, and the bottom BITONIC$[k/2]$, which get the top and bottom halves of the input respectively.

- Even numbered outputs of the top BITONIC$[k/2]$ go into the topmost $k/4$ inputs of the top MERGER$[k/2]$, and the odd outputs go into the topmost $k/4$ inputs of the bottom MERGER$[k/2]$.

- The outputs of the bottom BITONIC$[k/2]$ are split similarly. The even numbered outputs become the bottommost inputs of the top MERGER$[k/2]$ and the odd numbered outputs become the bottommost inputs of the bottom MERGER$[k/2]$.

- The top $k/4$ outputs of the top MERGER$[k/2]$ form the even numbered inputs into the top MIX$[k/2]$ and the bottom $k/4$ outputs form the even numbered inputs into the bottom MIX$[k/2]$. The corresponding outputs of the bottom MERGER$[k/2]$ make up the odd numbered inputs of the two MIX$[k/2]$ components.

- The output wires of the network are the output wires of the top MIX$[k/2]$ and the bottom MIX$[k/2]$ networks (in order).

A MERGER$[k]$ component can be divided into four components, two MERGER$[k/2]$ components and two MIX$[k/2]$ components. The connection of the wires between components is similar to the connections between the MERGER[] and MIX[] networks in the decomposition of BITONIC$[k]$.

A MIX$[k]$ component can be divided into two MIX$[k/2]$ components in the natural way, by dividing the balancers into two sets of $k/2$ balancers each. There are no connections between the two MIX[] components.

The smallest components are the individual balancers. All the components can be organized into a natural hierarchy, as shown in Figure 2. The decomposition tree of all the components starting from BITONIC$[w]$ is denoted by T_w.

Each component has a name which is assigned as follows: the name of a component is its position in a pre-order traversal of T_w.

2.2. Adaptive Implementation using T_w

Definition 2.1 *A* cut *on T_w is another tree formed by pruning away a set of subtrees from T_w (except for the complete subtree T_w itself).*

Two example cuts of T_8 are shown in Figure 2. The adaptive bitonic network is based on the following observation. *Given any cut of T_w,* BITONIC$[w]$ *can be implemented by the set of components at the leaves of this cut.* There are many possible cuts for a tree, each leading to a different set of components for implementing the bitonic network, and the distributed system selects that cut which fits its current parameters. Moreover, the system can change to an implementation based on a different cut in a decentralized way by splitting and merging components, as we describe further. An example implementation based on *cut1* in Figure 2 is shown in Figure 3.

Implementing a Component The strength of the above decomposition is that each component, whether BITONIC$[k]$ or MERGER$[k]$ or MIX$[k]$, can be implemented in a very simple way on a *single node*. At first glance, it might seem necessary that to implement BITONIC$[k]$, it is necessary to simulate all balancers and wires in the component. However, it is not necessary to do so. In fact, the implementation of all three types of components is the same, and is described below.

The component of width k has k input and k output wires, numbered from $0 \ldots k-1$. Each component has a single local variable x, which can take values ranging from $0 \ldots k-1$. The current value of x denotes wire on which the next input token is to be output; initially $x = 0$. The next token entering the component exits it on wire x, and x is incremented modulo k.

We first claim that a network constructed in the above manner counts. We omit the proof from this paper due to space constraints, and note that the proof is very similar to the proof that BITONIC$[w]$ counts [AHS94].

Theorem 2.1 *Given the above implementation of the* BITONIC[]*,* MERGER[] *and* MIX[] *components, the network constructed by any* cut *of T_w is a counting network of width w.*

Splitting a Component A component is always implemented at a single node. The node can split the component recursively if it estimates that greater parallelism is necessary.

A node splits component c as follows. (1)Insert new components with names equal to the children of c in T_w. (2)Initialize the new components appropriately, based on the current state of c, and form connections between the components as described in the recursive decomposition and (3)Remove component c from the system.

Merging Components The merging of smaller components into a component of a larger width is initiated by the node which had split the component in the first place. When a node v estimates that the network size has decreased, it might decide to initiate the merging of components. To implement this, v must keep a list of all the components that it has previously split, and which have not yet been merged.

When v decides that a component c (that it had split earlier) has to be merged, the procedure is as follows. Let C_c denote the set of the children components of c.

First, v contacts all the nodes which are hosting the components in C_c. If any of these components have been further split, then these components are recursively merged. Now, suppose $C_c = \{c_1, c_2 \ldots c_k\}$ (note that $k \leq 6$) and these components are residing at nodes $v_1, v_2, \ldots, v_k$ respectively. The merging of C_c into c is done as follows. (1)Temporarily stop routing tokens through $c_1 \ldots c_k$. i.e. if node v_j receives a token for component c_j, then v_j stores the tokens until the merging has completed, and will then forward it to v. (2)Construct a new component c whose state is initialized based on the current states of $c_1, \ldots c_k$. (3)Remove components $c_1, \ldots c_k$ from the system and insert the newly constructed component c at node v.

2.3. The Network Properties

Consider any cut, T, of the tree T_w. A counting network can be formed by the nodes at the leaves of T. The nodes of T are organized into *levels*: the root is at level 0, and the children of each node are at one level greater than the node itself. Note that we use the word "depth" to mean the depth of the overlay counting network, and that this is different from the level of the components in the decomposition tree T. Different leaves of T can be at different levels, so that the components of the network could be of different sizes.

Lemma 2.2 *If every leaf of T is at a level at most k, then the effective depth of the resulting network is at most $(k+1)(k+2)/2$.*

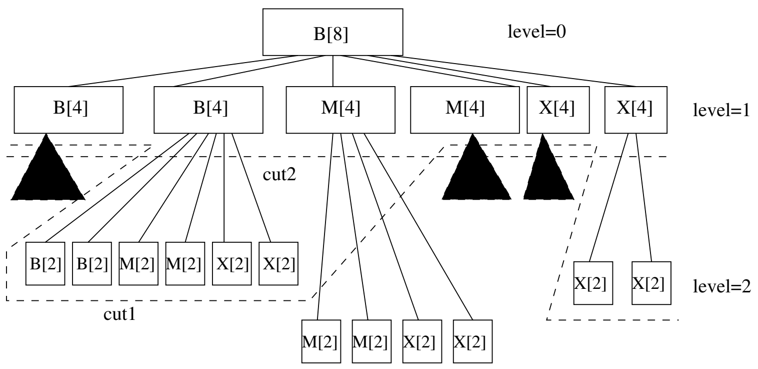

Figure 2. T_8**: The hierarchical decomposition of** $\text{BITONIC}[8]$
Each node in this tree is a component. The tree is not completely shown, as indicated by the solid subtrees. The boxes labeled $B[w], M[w], X[w]$ represent $\text{BITONIC}[w]$, $\text{MERGER}[w]$, $\text{MIX}[w]$ components respectively. Two example cuts, *cut1* and *cut2* are shown in the figure.

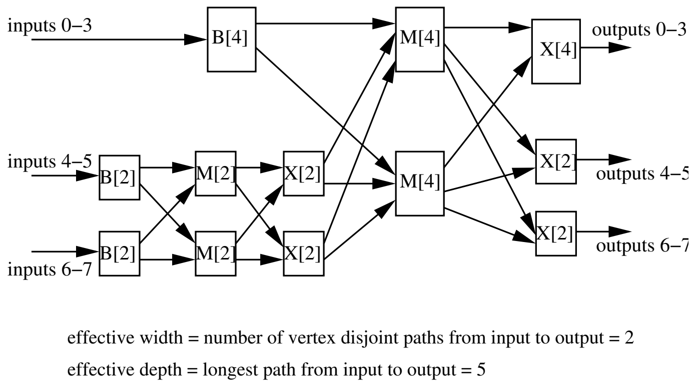
inputs 0–3
inputs 4–5
inputs 6–7
B[4]
B[2]
B[2]
M[2]
M[2]
X[2]
X[2]
M[4]
M[4]
X[4]
X[2]
X[2]
outputs 0–3
outputs 4–5
outputs 6–7
effective width = number of vertex disjoint paths from input to output = 2
effective depth = longest path from input to output = 5

Figure 3. A network of width 8 constructed out of 12 components of varying widths, from *cut1* **in Figure 2. The arrows between components represent groups of wires.**

Proof: Each vertex v of T represents a subnetwork that is formed by the subtree of T rooted at v. Some of these are bitonic networks, some merger and others mix networks. Let d_i^B denote the maximum effective depth of a bitonic subnetwork formed by a vertex at level i. We are interested in upper bounding d_0^B. Similarly, let d_i^M and d_i^X respectively denote the maximum effective depths of a merger and a mix network formed by a vertex at level i.

Consider an internal vertex which represents a bitonic subnetwork. Its children are two bitonic, two merger and two mix networks. Any token entering this bitonic subnetwork has to pass through exactly one of the smaller bitonic networks, one of the smaller merger networks and one of the smaller mix networks. Similarly, the effective depth of a merger subnetwork is equal to the sum of the depths of a smaller merger and a mix network. The effective depth of a mix network is always 1. Thus, we have the following inequalities:

$$\begin{aligned} d_i^B &\leq d_{i+1}^B + d_{i+1}^M + d_{i+1}^X \\ d_i^M &\leq d_{i+1}^M + d_{i+1}^X \\ d_i^X &= 1 \end{aligned}$$

In addition, we know $d_k^B = d_k^M = d_k^X = 1$, since no leaf is at a level greater than k. Solving the above recurrences, we obtain $d_0^B \leq (k+1)(k+2)/2$, which proves the lemma. ■

Lemma 2.3 *If the level of every leaf of T is at least k, then the effective width of the network is at least 2^k.*

Proof: If every leaf of T is at a level of exactly k, then the network formed by the components is isomorphic to a bitonic counting network of width 2^{k+1}, and hence the effective width of the network is exactly 2^k. According to our definition of effective width, a single balancer, which has two input and output wires, is a counting network of effective width 1, since the number of vertex disjoint paths from input to the output is 1.

Next, we note that the effective width of a network does not decrease if any of its components split, due to the following reason: vertex disjoint paths which existed before the components split are still vertex disjoint paths after the split. Thus, if every leaf of T is at a level of k or greater, then the effective width of the network is at least 2^k. ■

3. Distributed Decisions: When to Split/Merge?

We use the word "node" to mean a physical (computing) node in the distributed system. Each component is mapped to a node through a distributed hash function h. We assume that the nodes all have random identifiers in the range $[0, 1]$, and that the underlying distributed hash function is provided by the Chord system[SMK+01].

Notation: For integral $k > 0$, let $succ_k(v)$ denote the kth (clockwise) successor of node v on the Chord ring. Let $d(u, v)$ denote the distance between nodes u and v on the circumference of the Chord ring. We assume that the circumference of the ring is 1 unit. For each $\ell = 0 \ldots (\log w - 1)$, let $\phi(\ell)$ denote the number of components at level ℓ of T_w, the decomposition tree of a bitonic network of width w. Thus, $\phi(0) = 1$, $\phi(1) = 6$, and $\phi(2) = 24$ and so on.

Fact 1 *For integral $0 \leq k \leq \log w - 2$, we have: $\phi(k+1) \leq 6\phi(k)$ and $\phi(k+1) \geq 2\phi(k)$.*

3.1. System Size Estimation

An important component of the algorithm is a distributed way to estimate the size of the system, so as to decide the degree of parallelism that is appropriate. Size estimation has been studied in previous works on peer-to-peer DHT construction, including Manku[Man03], the Viceroy system[MNR02] and Horowitz and Malkhi[HM03]. Specifically, Manku gives an algorithm which gives an accurate estimate of the system size by measuring the size of the interval spanned by $\Theta(\log N)$ successive nodes. Our size estimation scheme is similar to that of Manku. Let N be the system size. We show that the size estimates of all the nodes are within a factor of 10 of the actual size N with high probability. A node v estimates N (locally) in two steps.

Step 1: v first obtains an estimate e_v of $\log N$ as follows: $e_v = \log \left\{ \frac{1}{d(v, succ_1(v))} \right\}$

Step 2: v uses e_v to get n_v, an estimate of N as follows. Let $k = 4\lceil e_v \rceil$. The estimate is: $n_v = \frac{k}{d(v, succ_k(v))}$. The above step is accomplished by stepping through k nodes on the Chord ring.

We show the following bounds on the size estimates obtained by the nodes. The proofs use Chernoff bounds.

Lemma 3.1 *For each node v in the system,* $\Pr\left\{e_v > \frac{\log N}{2}\right\} \geq 1 - \frac{1}{N^3}$ *and* $\Pr\left\{N/10 \leq n_v \leq 10N\right\} \geq 1 - \frac{3}{N^3}$

Proof:

$$\begin{aligned} \Pr[e_v < \frac{\log N}{2}] &= \Pr[\frac{1}{d(v, succ_1(v))} < \sqrt{N}] \\ &= \Pr[d(v, succ_1(v)) > 1/\sqrt{N}] \end{aligned}$$

The latter probability means that in a region of the circumference of length $1/\sqrt{N}$, no other node was mapped in. Since each node chooses a random point on the circle of circumference 1, this probability is: $[1 - 1/\sqrt{N}]^N \leq \exp -\sqrt{N} \leq \frac{1}{N^3}$ for large N.

Node v's estimate of the system size is $n_v = \frac{k}{d(v, succ_k(v))}$.

$\Pr[n_v > \alpha] = \Pr[d(v, succ_k(v)) < k/\alpha]$. Let X denote the number of nodes falling in a region of length k/α of the circle. The probability of each node falling in this region is k/α (since the circumference is 1 unit). The above probability is $\Pr[X > k] \leq \binom{n}{k}(k/\alpha)^k \leq (ne/\alpha)^k$. By plugging in $\alpha = 3Ne$ and $k = 2\log N$, the above is bounded by $1/N^3$.

$\Pr[n_v < \beta] = \Pr[d(v, succ_k(v)) > k/\beta]$. Let Y denote the number of nodes falling in a region of the circle of length k/β. The above probability is $\Pr[Y < k]$. $E[Y] = Nk/\beta$. Choosing $\beta = N/10$, we get $\Pr[Y < k] = \Pr[Y < E[Y](1 - 9/10)] \leq e^{-E[Y](9/10)^2/3} = e^{-2.7k}$, where we have used the Chernoff bounds. By plugging in $k = 2\log N$, the above probability is less than $1/N^3$.

Summing the probabilitites over all the nodes, and using the union bound, we obtain the desired result. ■

Henceforth, by the word "with high probability", we mean "with probability at least $1 - 1/N^\alpha$ where $\alpha > 0$. The above lemma immediately leads to the following lemma, using the union bound on probabilities:

Lemma 3.2 *With high probability, the size estimates of all the nodes in the system are within $N/10$ and $10N$.*

Local Level Estimates: Given its estimate of the system size, each node v determines a level ℓ_v as follows: ℓ_v is the largest integer k such that $\phi(k) < n_v$. If N is the actual size of the system, then define ℓ^* as follows: ℓ^* is the largest integer k such that $\phi(k) < N$.

The intuition is that if every node v's size estimate matched N, then the "best" implementation of BITONIC$[w]$ (which has as many components as the number of nodes in the system) would use the components at level ℓ^* of the decomposition tree T_w.

Lemma 3.3 *With high probability, the level estimates (ℓ_v's) of all the nodes in the system are in the range $[\ell^* - 4, \ell^* + 4]$.*

Proof: By the definition of ϕ, we have $2\phi(k) \leq \phi(k+1) \leq 6\phi(k)$. Consider any node v in the system. If $\ell_v > \ell^* + 4$, then $\phi(\ell_v - 4) > N$ and $\phi(\ell_v) < n_v$, which implies that $n_v > 16N$, which is not possible (with high probability) due to Lemma 3.2.

Similarly, for some node v, if $\ell_v < \ell^* - 4$, then $\phi(\ell_v + 5) < N$ and $\phi(\ell_v + 1) > n_v$, which implies that $n_v < N/16$, which is not possible (with high probability) from Lemma 3.2. ■

3.2. Splitting and Merging Rules

Node v maintains the following local invariant: *all components residing on v must be at level ℓ_v of T_w or greater.*

Splitting Rule Split all the components in v whose level is less than ℓ_v.

Merging Rule A node v reconsiders its earlier splitting decisions every time its level estimate ℓ_v decreases. Node v has a list of all components that it has currently split, but have not been merged yet. For each component i in this list, it checks to see if the level of i is still less than the (new) level of the node, ℓ_v. If not, then it initiates a merging of the component using the procedure described in Section 2.

The following lemma gives a characterization of the type of components that are finally implementing BITONIC$[w]$.

Lemma 3.4 *If the level of every node in the system was in the range $[\ell_1, \ell_2]$, then the level of every component in the network is also in the range $[\ell_1, \ell_2]$.*

Proof: First, we note that there cannot be any component whose level is less than ℓ_1. Regardless of which node this was mapped to, the component would be split.

We now show that there cannot be any component whose level is greater than ℓ_2. Suppose there was such a component i. Suppose v was the node which split a component (say j) to form component i (other components are also formed along with i). Since v's level is less than the level of component i, v will initiate a merge, which merges component i into j. ■

3.3. Properties of the resulting network

The final state of the network, resulting from the above splitting and merging rules, have the following properties.

Lemma 3.5 *The total number of components in the network is $O(N)$ with high probability. The expected number of components per node is $O(1)$ and the maximum number of components at any node is $O(\frac{\log N}{\log\log N})$ with high probability.*

Proof: From Lemmas 3.3 and 3.4, we have that all the components in the network have levels in the range $[\ell^* - 4, \ell^* + 4]$, with high probability. If all the components are at level $\ell^* - 4$, then the network would have the minimum number of components. This number is at least $N/6^5$, since $\phi(\ell^* - 4) \geq \phi(\ell^* + 1)/6^5 > N/6^5$ (where we have used Fact1).

Similarly, if all the components were at level $\ell^* + 4$, we would have the greatest number of components, and this number is not more than $6^4 N$, since $\phi(\ell^* + 4) \leq 6^4\phi(\ell^*) \leq 6^4 N$ (where we have used Fact1). Thus, with high probability, the number of components in the network is in the range $[c_1 N, c_2 N]$, where $c_1 = 1/6^5$ and $c_2 = 6^4$.

All these components are mapped to N nodes uniformly at random by the distributed hash function. Thus, the expected number of components per node is $O(1)$ and the maximum number of components per node is $O(\frac{\log N}{\log \log N})$, by the balls and bins analysis (Page 45 in [MR95]). ■

Theorem 3.6 *For the final network formed, the following are true with high probability:*
(1)The effective depth of the network is $O(\log^2 N)$.
(2)The effective width of the network is $\Omega(N/\log^2 N)$

Proof: From Lemma 3.3 and Lemma 2.2, we have that the depth of the network is less than $(\ell^* + 5)(\ell^* + 6)/2$. Since $\phi(\ell^*) < N \le \phi(\ell^* + 1)$, and $\phi(x)$ lies between 2^x and 6^x (from Fact 1), we have $\ell^* = O(\log N)$, and thus the depth of the network is $O(\log^2 N)$.

Similarly, from Lemma 3.3 and Lemma 2.3, we have that the width of the network is at least $2^{\ell^*-4} = \frac{2^{\ell^*}}{16}$. Since $\phi(\ell^*) \le N \le \phi(\ell^* + 1)$, the number of components at level ℓ^* is $\Omega(N)$.

We can write the number of components of the counting network formed at level ℓ^* as the effective width times the effective depth. This gives: $2^{\ell^*} \cdot \log^2 N = \Omega(N)$. Thus, the effective width of the network is $\Omega(N/\log^2 N)$. ■

3.4. Node Joins and Leaves

Node Joins the p2p Network No change is needed for the state of the counting network, other than the actions required to maintain the distributed routing state of the p2p network. If the system size increases significantly, then the size estimation mechanism will alert the nodes, and they will split individual components.

Node v Leaves the Network Before leaving, the node has to move all the components it currently holds to the new home of those objects in the p2p network. In the case of Chord, this new home (say w) is just v's successor node in the ring. In addition, v has to transfer the state of all components that it has split, but which have not been merged yet. Node w takes over the responsibility of merging nodes that v has earlier split.

Node Crashes When a node crashes, all the state of the counting network that is contained in it is lost along with the components that the node contains. One approach to recovering from such faults is through self-stabilization [Dij74]. We first assume that the underlying p2p routing layer (such as Chord) is self-stabilizing. On top of this, we can layer a self-stabilizing counting network.

The state of the balancing network is the union of the state of all its components. The state of each component is an integer, which denotes which output wire the next token should go out on. In recent work [HT03], we have shown how to make balancing networks self-stabilizing. If the network was reset to an illegal state by a fault, then it will recover to reach a legal state, through local stabilization actions. Though the algorithm in [HT03] is intended for balancers, it can be easily extended to the more general components.

3.5. Routing Efficiency

Each component needs to know the location of all its out-neighbors (the components that its output wires lead to) so that it can route tokens appropriately. We can show that the expected number of out-neighbors of any component is a constant, and since the expected number of components mapped to a node is $O(1)$ (Lemma 3.5), the expected number of out-neighbors that a node has to keep track of is also $O(1)$.

Further, a node does not need to recompute the address of the out-neighbors every time that it has to route a token. If we assume that changes in the structure of the counting network are infrequent when compared to the rate of token arrivals, then the addresses of the out-neighbors can be cached and tokens routed directly to them. These addresses have to recomputed whenever the locations of the out-neighbors change, either due to nodes joining and leaving the network, or due to component splits or merges.

Finding an Input Component Finally, we return back to the question of finding an input component to send tokens to. Suppose a node v chooses an input balancer's identifier to send tokens to (it could choose any of the w possible input balancers). This balancer is a leaf in T_w. Either the balancer currently exists as a component, or one of its ancestors in T_w exists as a component. Since there are $(\log w - 2)$ ancestors of any leaf in T_w, node v has to lookup at most $(\log w - 1)$ names in the worst case before it can find an input component which currently exists in the network, assuming that the structure of the counting network does not change during the lookup. In a typical case, v might need to lookup far fewer names, if it remembers the component that it had sent its previous tokens to.

References

[AHS94] J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, 1994.

[AS03] J. Aspnes and G. Shah. Skip graphs. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2003.

[AVY94] W. Aiello, R. Venkatesan, and M. Yung. Coins, weights and contention in balancing networks. In *Proceedings of the annual ACM symposium* *on Principles of Distributed Computing*, pages 193–205, August 1994.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference*, volume 32, pages 338–334, 1968.

[Dij74] E. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.

[DLS00] G. Della-Libera and N. Shavit. Reactive diffracting trees. *Journal of Parallel and Distributed Computing*, 60(7):853–890, 2000.

[DPRS89] M. Dowd, Y. Perl, L. Rudolph, and M. Saks. The periodic balanced sorting network. *Journal of the ACM*, 36(4):738–757, October 1989.

[HHPT03] P. Ha-Hoai, M. Papatriantafilou, and P. Tsigas. Self-adjusting trees. Technical Report 2003-09, Chalmers University of Technology and Goteborg University, 2003.

[HKRZ02] K. Hildrum, J. Kubiatowicz, S. Rao, and B. Zhao. Distributed data location in a dynamic network. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, 2002.

[HM03] K. Horowitz and D. Malkhi. Estimating network size from local information. *Information Processing Letters*, 88(5):237–243, December 2003.

[HT03] M. Herlihy and S. Tirthapura. Self stabilizing smoothing and counting. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS)*, pages 4–11, 2003.

[KK03] F. Kaashoek and D. R. Karger. Koorde: A simple degree-optimal hash table. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, February 2003.

[KP92] M. R. Klugerman and C. G. Plaxton. Small-depth counting networks. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 417–428, 1992.

[Man03] G. S. Manku. Routing networks for distributed hash tables. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, July 2003.

[MBR03] G. S. Manku, M. Bawa, and P. Raghavan. Symphony: Distributed hashing in a small world. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 127–140, 2003.

[MNR02] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, August 2002.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995. page 45.

[PRR99] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems*, 32:241–280, 1999.

[RFH$^+$01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM*, August 2001.

[SMK$^+$01] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM*, pages 149–160, August 2001.

[SZ96] N. Shavit and A. Zemach. Diffracting trees. *ACM Transactions on Computer Systems*, 14(4):385–428, 1996.

[Wat98] R. Wattenhofer. *Distributed Counting: How to Bypass Bottlenecks*. PhD thesis, Swiss Federal Institute of Technology (ETH) Zürich, 1998.