

Bubbles: Adaptive Routing Scheme for High-Speed Dynamic Networks

(Extended Abstract)

Shlomi Dolev* Evangelos Kranakis† Danny Krizanc† David Peleg‡

Abstract

This paper presents the first dynamic routing scheme for high-speed networks. The scheme is based on a hierarchical *bubbles* partition of the underlying communication graph. Dynamic routing schemes are ranked by their *adaptability*, i.e., the maximum number of sites to be updated upon a topology change. An advantage of our scheme is that it implies small number of updates upon a topology change. In particular, for the case of a bounded degree network it is proved that our scheme is optimal in its adaptability by presenting a matching tight lower bound. Our bubble routing scheme is a combination of a distributed routing data-base, a routing strategy and a routing data-base update. It is shown how to perform the routing data-base update on a dynamic network in a distributed manner.

*Department of Mathematics and Computer Science, Ben-Gurion University of the Negev, Beer-Sheva 84105, Israel. Part of this work was done while this author visited the School of Computer Science, Carleton University, and the Department of Computer Science, Texas A&M University. Email: shlomi@cs.bgu.ac.il. Part of this research was supported by NSERC (Natural Sciences and Engineering Research Council of Canada) grant.

†School of Computer Science, Carleton University, Ottawa, Ontario, K1S 5B6, Canada. Email: {kranakis,krizanc}@scs.carleton.ca. Part of this research was supported by NSERC (Natural Sciences and Engineering Research Council of Canada) grant.

‡Department of Applied Mathematics and Computer Science, The Weizmann Institute, Rehovot 76100, Israel. Email: peleg@wisdom.weizmann.ac.il.

1 Introduction

The advent of fiber-optic technology dramatically changes the characteristics of distributed networks. It also improves the capabilities of distributed networks because it gives them the potential of supporting new services such as multimedia and real-time applications. Traditional algorithms designed for the point-to-point classical model of distributed networks may neither fit the new characteristics of the high-speed network nor support the new tasks it is capable of achieving. The relation between the bandwidth of a fiber-optics cable (on the order of hundreds of Gigabit per second) and the speed of a processor implies a bottleneck in the process time as opposed to the communication time. Therefore, high-speed networks use a fast switching subsystem in order to utilize the power of the fiber optic cables.

Algorithms for basic tasks that match the new network structure are of interest. In (high-speed) distributed networks messages are used for communication between different sites. A message sent from one site to another is transferred through the network according to a *routing scheme*. The routing scheme ensures that the message is forwarded towards its destination. The routing scheme serves the basic communication primitive in the network — message delivery. Being such a basic component, the performance of the distributed network as a whole may be dominated by the quality of the routing scheme. Thus, finding an efficient routing scheme is one of the most important tasks in distributed networks.

Imagine a network in which users may be connected and disconnected upon request. Users may migrate from one geographical region to another causing a change in the demand for services at different parts of the network. Assume further that the network spans the entire world and a single user (or a network junction) changes its location from one street of New York to another: is it reasonable to update the *entire* net-

work with a new routing data-base? We would not like the *entire* distributed network to be updated upon each such dynamic change. In fact we would like to *minimize* the effect of a topology change as much as possible.

Beyond planned topology changes, such as users migration, some transient topology changes may take place due to a failure of communication links or processors. One would like the network to automatically change the routing data-base to reflect the new topology upon the change. The resources used by such distributed routing data-base update (messages and time) have an inherent relation with the number of sites that have to change their portion of the distributed routing data-base.

We distinguish between *static* and *dynamic* routing schemes. A *static* routing scheme is a combination of a distributed routing data-base and a fitting routing strategy. The routing data-base is tailored to the network topology. Whenever the network changes its topology, a new distributed routing data-base is assigned to the network possibly changing the routing data-base portion of each processor. A *dynamic* routing scheme has in addition a fitting routing data-base update. Upon a topology change (e.g., link addition or removal) this fitting routing data-base update would change the distributed data-base only in a *limited* number of sites. We rank the dynamic routing schemes by their *adaptability*, i.e., the maximum number of sites to be updated upon a topology change. By this definition static routing schemes are associated with adaptability that is in the order of the number of nodes in the network.

The efficiency of a dynamic routing scheme is measured not only by its adaptability. It is also measured by the time and memory complexities associated with it. The time performance is measured by a *super-hop count* — the maximum (over all possible origin-destination pairs) number of super-hops, or routing steps, needed. (We shall define precisely the routing process in high-speed networks and the notion of super-hops later on.) The memory complexity is the total number of bits used for the routing data-base¹.

Previous work: Many clever routing schemes and lower bounds for the resources required for routing in point-to-point networks were presented in the past. Following the pioneering work of [KK77, KK80], which

¹Note that there is an interesting relation between the adaptability and the memory requirement e.g., it is clear that the worst case in terms of adaptability and memory requirement is when the entire topology is stored in the memory of each processor. Roughly speaking, both the adaptability and the memory requirements gain when the processors have less information on the system.

originated the approach of using hierarchical clustering strategies for memory-efficient routing, came a number of papers which attempted to characterize and bound the resource tradeoffs involved. The first set of studies along this line were mostly designed for special classes of networks like trees [SK85], complete networks [vLT86], and grids [vLT87]. Then routing schemes for general networks were presented in [PU89], [AB+90] and [AP92]. These studies focused on the design of routing schemes with compact routing tables and low stretch factor. The *stretch factor* of a routing scheme is defined as the maximum ratio, over all pairs of nodes in the network, between the length of the route provided for them by the routing scheme, and the actual distance between them in the network.

Unfortunately, most of the success in this field is for static networks. Few papers consider the dynamic property of the network. The following quotation is taken from [SK85]: “In actual network the topology may vary in time; in particular, nodes or links may be added or deleted”. [SK85] present a partial solution for limited cases of topology changes that keep the network in a tree structure. In [AP+92], the routing scheme of [AP92] is extended to the dynamic case for general networks. However, as argued in [AGR89], network changes in static routing schemes (such as [PU89, AB+89, AP92] or the derived dynamic scheme [AP+92]) “require expensive pre-processing to reconstruct the routing scheme over the whole network. The newly constructed structure is used until the next change...”. In contrast, [AGR89] design a routing scheme for the restricted case of dynamic growing trees. The solution of [AGR89] can handle neither link nor processor failures, nor can it be applied to the case of general graphs.

The papers mentioned above all deal with the traditional point-to-point communication model. Those solutions are not applicable for fiber optic high-speed networks, since the stretch measure (based on actual distances) is no longer the predominant cost parameter relevant to such networks, given their specific characteristics. Furthermore, the issue of adaptability is not handled by any of the above papers. In particular, the solutions provided in the above papers for the dynamic version of the problem might require data-base updates in all the nodes of the network following a single topology change.

Recently, [GZ94, CGZ94] presented static routing schemes for high-speed networks. The schemes statically assign links along a path to act as a virtual long link. Both papers attempt to simultaneously optimize three cost parameters, namely, the super-hop count, the stretch factor and the link load. However,

	Super-hop count	Memory Required	Adaptability	Model
Multiple Trees	1	$O(n^2 \log n)$	$O(n)$	
Single Leader	2	$O(n^2 \log \delta)$	$O(n)$	
Basic Bubbles	k	$O(k\delta n^{1+1/k} \log \delta)$	$O(3^k \delta^2 n^{1/k})$	Node failures
Basic Bubbles	k	$O(k\delta n^{1+1/k} \log \delta)$	$O(3^k \delta n^{1/k})$	Link failures
Edge-Bubbles	k	$O(kn^{1+1/k} \log \delta)$	$O(3^k \delta n^{1/k})$	Node Failures
Edge-Bubbles	k	$O(kn^{1+1/k} \log \delta)$	$O(3^k n^{1/k})$	Link failures

Figure 1: Comparison of Routing Schemes

[GZ94, CGZ94] as well do not address adaptability explicitly, and in the resulting solutions a single topology change may in some cases require the entire routing data-base to be updated. Hence the solution might be impractical in a dynamically changing environment.

Contributions of this paper: In this work we present the first dynamic routing scheme for high-speed networks. We present a family of hierarchical *bubbles* schemes. The intuition behind this structure is the behavior of natural bubbles. Assume a partition of a space into bubbles. Whereas bubbles may shrink (members are disconnected) or expand and/or blow up (new members are inserted) we want to avoid total change of the bubble structure upon a change at a single bubble. We define an upper and lower threshold for the size of a bubble. When the upper threshold is reached the bubble is split into two bubbles. When the lower threshold is reached the bubble is combined with a single neighboring bubble which “swallows” the small bubble (and then is split if necessary).

The Bubbles scheme exhibits a tradeoff between the number of super-hops needed for routing on the one hand, and the adaptability and memory requirement parameters. Namely, the more super-hops are allowed, the less memory is needed to store the routing data-base, and the more efficient it becomes to adapt the scheme to dynamic changes. Denote the maximum number of neighboring nodes with which each node in the network may be directly connected by δ . For the case of constant δ we prove that our scheme is optimal in its adaptability by presenting a matching tight lower bound. Note that this is the case in many settings in reality, where the number of communication ports of a single processor is limited. Our bubble routing scheme is a combination of a distributed routing data-base, a routing strategy and a routing data-base update. We present a distributed routing data-base update strategy for dynamic changing networks.

The rest of the paper is organized as follows. The definition of the problem and two examples for basic rout-

ing schemes appear in Section 2. These two schemes are given as examples for our complexity measures and a base for comparison with the bubbles scheme, which is presented in Section 3. See Figure 1 for a summary of the comparison among the different schemes (note that we typically choose k to be a small constant with a value that is much less than $\log n$). Section 4 presents a lower bound on the adaptability of any routing scheme. The distributed update of the bubbles routing data-base is sketched in Section 5. Concluding remarks are given in Section 6. Most of the proofs are omitted from this extended abstract.

2 Definition of the Problem

2.1 High-Speed Dynamic Network

We consider the high-speed model of a communication network as described in [CG88]. The network is described by an undirected graph $G = (V, E)$. The nodes, $V = \{1, \dots, n\}$, represent the processors of the network (where n is essentially an upper bound on the number of processors in a connected component). The edges of the graph represent bidirectional communication channels between the processors. Each processor consists of two components, a *switching subsystem* and a *node control unit* that are connected by a virtual link. The *switching subsystem* is a fast and simple hardware device that switches arriving packets to the appropriate communication link or to its node control unit. Within the switching subsystem each communication link (including the virtual link to the switching subsystems’ node control unit) has a unique label. When a packet p arrives at a switching subsystem and the header of p contains at least one label, then the switching subsystem removes the first label, l , and the shortened packet is sent on the link with label l . The *node control unit* of each processor contains the processing hardware and software necessary to extract the information content of messages (delivered in the packets), do internal computation, and generate packets to be forwarded to other

nodes via the processor's switching subsystem.

In this model a packet consists of the entire path from the sender to the receiver. If a packet length is 50 bytes and δ is, say 5, then each processor might reach 5^{50} other processors. This is more than enough for every existing network. Moreover, note that the first packet in a connection may mark the path (by the connection identifier) for the next packets that are related to the same connection — thus not every packet contains the entire path. Due to the above network architecture, it is assumed that a message sent from any processor, P , to any destination, Q , in the network may arrive in one time unit provided the labels along the entire path from P to Q are known to P . We refer to such a routing step as a *super-hop*.² In case the entire path is not known to P , the message may be sent through a number of super-hops. In each super-hop, the sender packs the message in a packet whose header contains the route description to the next intermediate destination, and advancing the message to that destination. Since the overhead of preparing the packets dominates the transmission time, the cost of the entire routing process is proportional to the number of super-hops it involves.

The network is *dynamic* in the strong sense: processors and links may crash and recover arbitrarily. However, the number of edges connected to a node P , which we call the *degree* of P , does not exceed some predefined value. We denote the maximum degree of nodes in the network by δ .

2.2 Complexity Measures for Routing Schemes

The routing of packets in the network is done according to a *distributed routing data-base* maintained by the node control unit and a fitting *routing strategy* and *data-base update*.

Super-hop count: The *super-hop count* of a routing scheme bounds the maximum number of super-hops (or, packet generation and transmission steps) required in order to send a message from one node control unit to another. Obviously, a single packet transmission is sufficient when every processor knows the entire topology (including the link labels). Thus, the super-hop count can be thought of as measuring the ratio between the maximum number of packets *generated* by the routing scheme and the optimal number of packets that must be generated in order to deliver a message sent from one node control unit to another³.

²Certain routing models devised for ATM networks use more elaborate mechanisms, and distinguish between several types of “super-hops”, e.g., routing along virtual channels or virtual paths. Here, we follow the simple, unified model of *Automatic Network Routing (ANR)* of [CG88].

³It is interesting to note that in that sense, the super-hop

Memory: The memory complexity is the total number of bits maintained by the node control units for the routing data-structure.

Adaptability: A single topology change \mathcal{C} occurs when a single processor or a single link joins (or recovers) or leaves (or crashes) the system. Note that for any two topologies G_1 and G_2 there exist a finite sequence of single topology changes $\mathcal{C}_1, \mathcal{C}_2, \dots$ that transfer G_1 into G_2 . For example the sequence can start with adding all the processors that do not appear in G_1 but do appear in G_2 , one processor at a time. Then links are added in a similar fashion. Finally, links and then processors are removed to form G_2 . Ideally a topology change causes only a very limited number of processors to change their portion of the routing distributed data-base. Thus, we choose the *adaptability measure* to be the maximum number of processors that have to change their portion of the routing distributed data-base upon a single topology change. We first ignore the issue of *how* the routing data-base is updated upon a topology change and only count the number of processors that have to change their routing data-base. Then we present a distributed update for the bubbles routing scheme which we call *bubbles update*.

Distributed adaptability: is the maximal number of node control units that have to participate in the distributed routing data-base update upon a single topology change. Clearly, the distributed adaptability is greater than the adaptability since every node control unit that has to change its portion of the routing distributed data-base participates in the distributed data-base update. In addition, other node control units may participate in the distributed update without changing their routing data-base portion.

2.3 Examples — Basic Routing Schemes

In this section we bring two simple examples for routing schemes that demonstrates the power of the computation model and the importance of our performance measures. The first routing scheme is called *multiple spanning trees* the second is the *single leader scheme*. Both schemes have adaptability $O(n)$.

The simplest routing scheme for our network is the *multiple spanning trees* routing scheme. The routing distributed data-base of the multiple spanning tree scheme is a description of a spanning tree of the entire topology at each processor. The routing strategy is to use the entire path given by the routing distributed data-base to each destination. The routing update has to change the distributed routing data-base upon every

count of a routing scheme in the high-speed model essentially plays a similar role to that played by the *stretch factor* measure in routing schemes for traditional computer networks.

processor addition and removal as well as upon removal of a link used as part of a spanning tree by some processor. The update would change the spanning trees representation in an obvious manner. The super-hop count of this scheme is clearly one, since a single packet is generated for the delivery of a message. The memory requirement is $n^2(\log n + \log \delta)$ since $n(\log n + \log \delta)$ bits are required in order to describe a spanning tree with link labels for every processor (the description is by the use of parentheses form). The adaptability is n because a single recovery of a processor or a link requires the update of the tree of every processor.

The second example is the *single leader* routing scheme. In this scheme only a single processor, L , maintains the description of a spanning tree of the entire network. Every other processor, P , has the path description to L , $path_L$. $path_L$ is a list of link labels that defines a path from P to L . The routing strategy for delivering a message m to a processor Q is to send a packet $(path_L, Q, m)$ to L and then L sends a packet $(path_Q, m)$ to Q . A single failure of a link (or the leader) may require n processors ($n - 1$ processors, respectively) to update their portion of the routing data-base. For the single leader routing scheme the super-hop count is 2 and the memory requirement is $n(\log n + \log \delta) + n(n - 1)\log \delta$.

3 The Bubbles Routing Scheme

In this section we present our main result which is a novel routing scheme for dynamic high speed networks. We first introduce a new type of hierarchical partitioning scheme for general graphs, with certain desirable properties. We then present a graph partitioning algorithm that constructs such a partition. Then we explain how to maintain the properties of the partition (most notably, the size bounds on connected components) upon a dynamic change. Finally, we present our routing scheme, which is based on the hierarchical partition.

3.1 The Bubbles Partition

Definition 3.1 *Given a graph G and parameters $0 < a < b$, an $[a, b]$ -bubbles partition of G is a partition of the nodes of G into disjoint connected components called bubbles, $\mathcal{P} = \{\mathcal{B}^1, \dots, \mathcal{B}^r\}$, such that the size of each bubble \mathcal{B} (i.e., the number of nodes it contains), satisfies $a \leq |\mathcal{B}| \leq b$.*

Let us next describe a partitioning algorithm based on a technique presented in [Va81]. Given a n -node graph G with maximum degree δ and a parameter

$1 < x \leq n$, the algorithm produces an $[x, \delta x]$ bubble partition for G . As a preprocessing step, choose an arbitrary node of the graph, R , and construct a spanning tree, \mathcal{T}_R , rooted at R with edges directed towards the leaves. The algorithm will partition this tree into bubbles as described next (in Fig. 2). Theorem 3.1 is implied by the existence of the partition algorithm.

Theorem 3.1 *For every n -node graph G with maximum degree δ and parameter $1 \leq x \leq n$, G has an $[x, \delta x]$ bubble partition. ■*

Next we define special hierarchies of bubbles partitions.

Definition 3.2 *Given a graph G and a list of k integer parameters $\bar{x} = (x_1, x_2, \dots, x_k, x_{k+1})$, where $x_1 = n$, $x_i > \delta x_{i+1}$ and $x_{k+1} = 1$, a \bar{x} -hierarchical bubbles partition of G is a collection of k partitions \mathcal{P}_i , $1 \leq i \leq k$, with the following properties. (1) For every $1 \leq i \leq k$, \mathcal{P}_i is an $[x_i, \delta x_i]$ -bubbles partition. (2) For every $1 < i \leq k$, \mathcal{P}_i is a refinement of \mathcal{P}_{i-1} , i.e., each bubble of level i is fully contained in some bubble of level $i - 1$ (or equivalently, each bubble of level $i - 1$ is partitioned into bubbles of level i).*

The hierarchical bubble partition of a graph is constructed level by level, where the level 1 partition consists of a single bubble covering the entire graph. The partitions are constructed by the following algorithm. First, apply the preprocessing step used before algorithm PARTITION, and construct the spanning tree \mathcal{T}_R for the graph. Then execute Algorithm HIERARCHY(\bar{x}) described in Fig. 3.

Theorem 3.2 *Every graph has a \bar{x} -hierarchical bubbles partition. ■*

In the sequel, we choose the parameters \bar{x} as $x_i = \lceil n^{(k-i+1)/k} \rceil$. This choice satisfies the requirements of the definition, so long as $n^{1/k} \geq \delta$. We use \mathcal{B}_i to denote a bubble at level i . For every bubble \mathcal{B}_i let $\mathcal{T}(\mathcal{B}_i)$ be a representation of the subtree of the spanning tree \mathcal{T} that spans the bubble \mathcal{B}_i . (In the sequel we refer to both the description of the spanning tree and the spanning tree itself by $\mathcal{T}(\mathcal{B}_i)$.) Thus each of the k partitions \mathcal{P}_i defines a forest $\mathcal{F}_i = \bigcup_{\mathcal{B} \in \mathcal{P}_i} \mathcal{T}(\mathcal{B})$. We note the following properties of the hierarchical partition. Since each bubble of level i is composed of the union of some bubbles of level $i + 1$, the spanning tree $\mathcal{T}(\mathcal{B}_i)$ is a combination of the spanning trees of those bubbles. Hence $\mathcal{F}_{i+1} \subseteq \mathcal{F}_i$ for every $1 \leq i \leq k - 1$, and \mathcal{F}_1 , corresponding to the single bubble containing the whole network, is

⁴We say that $\mathcal{F}_{i+1} \subseteq \mathcal{F}_i$ when every tree in \mathcal{F}_{i+1} is a subtree in \mathcal{F}_i .

the entire tree \mathcal{T} . These properties will be maintained as invariants of the hierarchical partition throughout the updates performed upon topology changes, and are important for the analysis of the scheme, as will be shown in the sequel.

3.2 Dynamic Maintenance of the Partition

In order to use the Bubbles partition over time in a dynamically changing environment, it is necessary to be able to maintain a legal partition in the presence of link and node failures and recoveries. One point that we need to address first when dealing with the dynamic case is that of failures that disconnect the network. Our policy in such a case is to treat each connected component separately, but still using the global parameters \bar{x} . This has the implication, that certain connected components cannot be partitioned in some of the lower levels of the hierarchy, since they are too small. In particular, if a certain connected component G' of the graph has less than x_i (but at least x_{i+1}) nodes, then on levels 1 through i there is a single bubble encompassing the entire connected component, and this bubble is said to be *illegal*, since it is smaller than the allowed lower bound. The implication of such situation is that once a connecting link recovers, it is necessary to reorganize the partition in order to “legalize” the bubble structure.

We now present the strategy for handling each topology change. For now, the strategy we describe is centralized, as if an outside operator changes the distributed routing data-base. Later, in Section 5, we discuss a distributed implementation of our maintenance strategy. We use the following definition.

Definition 3.3 *Two bubbles \mathcal{B}'_l and \mathcal{B}''_l both at level l are said to be tree-neighboring w.r.t. the level $l-1$ forest \mathcal{F}_{l-1} if there exists an edge of \mathcal{F}_{l-1} that connects them. We refer to this tree link as the connecting link of $\mathcal{T}(\mathcal{B}'_l)$ and $\mathcal{T}(\mathcal{B}''_l)$.*

Our maintenance activities are based on employing two basic operations, or “procedures”, $\text{COMBINE}(\mathcal{B}'_l, \mathcal{B}''_l)$ and $\text{SPLIT}(\mathcal{B}_l)$, described in Figures 4 and 5, responsible for combining and splitting bubbles, respectively. These two basic procedures are used as building blocks for our update operations. Let us next describe another procedure, that handles the removal of a link e from the forest of spanning trees of the level l bubbles partition, \mathcal{F}_l . This removal can be the result of either the link’s failure, or the restructuring of bubbles at the next lower level, $l-1$, that caused the removal of this link from \mathcal{F}_{l-1} . In any case, it is assumed that e does not appear in \mathcal{F}_{l-1} . The computation of the new forest \mathcal{F}'_l , that no longer contain e , is based on the current forest for level $l-1$, \mathcal{F}_{l-1} , in the

sense that the only edges that are added to \mathcal{F}'_l during the procedure’s execution are also edges of \mathcal{F}_{l-1} . The procedure $\text{REMOVE}(e, \mathcal{F}_l)$, described next in Figure 6, relies on the assumption that both \mathcal{F}_{l-1} and \mathcal{F}_l span legal bubbles partitions (of level $l-1$ and l , respectively). However, it is not assumed that \mathcal{F}_l is a refinement of \mathcal{F}_{l-1} .

It is important to understand that because of the way procedure REMOVE removes a link (namely, trying to merge the resulting portions of the broken bubble with neighboring bubbles, and then splitting the combined bubbles to regain the size bound), it might happen that a single link removal at level l can cause more than one link removal at the next level $l+1$ (in fact, up to three). Hence, a single topology change of the form of a link failure might result in up to 3^{i-1} link removal operations in each level i . Consequently, whenever we need to remove an edge from the partition of some level i , we typically need to immediately fix the partitions of all higher levels, $i+1$ through k , as well. This is done via procedure REMOVE_ALL .

The procedure fixes the Bubbles partitions \mathcal{P}_l at each level $i \leq l \leq k$, one at a time, starting from level i up. At each level l , the procedure outputs a corrected forest \mathcal{F}'_l . The computation of the new bubble partition of level l , \mathcal{P}_l , and the corresponding forest \mathcal{F}'_l , is based on the recently computed forest for level $l-1$, \mathcal{F}'_{l-1} , and the old forest \mathcal{F}_l of level l before the topology change. Suppose the procedure already computed the new partition up to and including level $l-1$. As a result, there is a number of edges that appear in \mathcal{F}_l and do not appear in \mathcal{F}'_{l-1} (since they have already been removed on that level). The procedure removes each tree link of $\mathcal{F}_l - \mathcal{F}'_{l-1} = \{e_1, e_2, \dots, e_m\}$ from \mathcal{F}_l . This is done sequentially, edge by edge, and the forest \mathcal{F}_l is modified at each edge removal step to reflect the change. Hence the sequence of link removals creates a sequence of forests $\mathcal{F}_l = \mathcal{F}_l^{(0)}, \mathcal{F}_l^{(1)}, \dots, \mathcal{F}_l^{(m)} = \mathcal{F}'_l$, where the update for removing e_j leads from $\mathcal{F}_l^{(j-1)}$ to $\mathcal{F}_l^{(j)}$.

It remains to describe the modification of the forest $\mathcal{F}_l^{(j-1)}$ due to the removal of e_j . If e_j is not in $\mathcal{F}_l^{(j-1)}$, then nothing need be done. Otherwise, if the removal of e_j causes the partition of some bubble, then we invoke procedure $\text{REMOVE}(e_j, \mathcal{F}_l^{(j-1)})$ to remove the edge and correct $\mathcal{F}_l^{(j-1)}$ into $\mathcal{F}_l^{(j)}$. Again, it is assumed that e does not appear in \mathcal{F}_{l-1} , and all edges added to \mathcal{F}'_l or higher partitions during the procedure’s execution belong to \mathcal{F}_{l-1} . The procedure relies on the assumption that initially, \mathcal{F}_l (for every $i-1 \leq l \leq k$) spans a legal bubbles partition. Procedure REMOVE_ALL is described formally in Figure 7.

Note that the lemma above holds for the case of legal

bubbles i.e., the connected component includes at least x_2 nodes. Similar arguments hold for the case in which the connected component is small. Till this stage we described algorithm to construct the bubble partition and basic procedures for maintaining the partition. Now we detail how those procedures are invoked in every of the four possibilities for topology changes.

Link Removal: If the removal of the link e does not partition the spanning tree of any bubble then no change of the hierarchical bubbles partition is required. If the spanning tree of a bubble is disconnected by removing e , then the link removal is handled at each level starting from level one up. At level one, if the communication graph is still connected then a non-tree link is promoted to be a tree link in order to retain the connectivity of the entire spanning tree, \mathcal{T} . The removal itself is then performed by applying procedure $\text{REMOVE_ALL}(e, 1)$.

Link Addition: An addition of a link that does not connect two previously separated connected components does not change the routing data base. An addition of a link that does connect two previously disconnected components G' and G'' is handled as follows. Let $\mathcal{T}(\mathcal{B}'_1)$ and $\mathcal{T}(\mathcal{B}''_1)$ be the spanning trees of G' and G'' , respectively, before the link addition. Connect $\mathcal{T}(\mathcal{B}'_1)$ with $\mathcal{T}(\mathcal{B}''_1)$ by invoking procedure $\text{COMBINE}(\mathcal{B}'_1, \mathcal{B}''_1)$ using the new link, to form a spanning tree $\mathcal{T}(\mathcal{B}_1)$ of the new connected communication graph.

Continue with levels 2 up to k one at a time. For level $2 \leq i \leq k$, if the new link connects two legal bubbles then no update operations are triggered by level i . Otherwise, when the new link connects at least one illegal bubble, then combine the illegal bubble with the new tree neighboring bubble (using procedure COMBINE), and split the combined bubble (using procedure SPLIT) if necessary. The split operation may result in the need to remove a link e at the next level. This is handled by invoking the link removal procedure $\text{REMOVE_ALL}(e, i)$ described above. Then the update for level $i + 1$ may be started.

Node Addition or Removal: Both the addition and the removal of a node can be described in terms of addition and removal of links. The addition is handled by first adding a link that connects two separated connected components one of which is the single node. Then adding the rest of the links one by one. Node removal is done by removing one link at a time and then removing the node.

In order to argue about the correctness of the dynamic update procedures, two main claims need to be established, namely, that at the end of each update operation, each partition \mathcal{P}_i is a legal $[x_i, \delta x_i]$ -bubbles par-

tion, and that the entire hierarchy is a hierarchical bubbles partition. These claims are naturally implied by the properties established earlier regarding procedures REMOVE and REMOVE_ALL .

3.3 The Routing Scheme

Next we describe the distributed routing data-base by the use of the bubble partition.

Routing Distributed Data-base: Start by constructing a hierarchical bubbles partition for the network. For every bubble \mathcal{B}_k at level k , define the nodes that reside in \mathcal{B}_k as its *members*, and choose one of these nodes to be the *leader* of the bubble, denoted $L(\mathcal{B}_k)$. In general, for level $i < k$, define the *members* of a bubble \mathcal{B}_i to be all the leaders of level $i + 1$ bubbles that reside in the connected component of \mathcal{B}_i , and choose one of these members to be the bubble's leader, $L(\mathcal{B}_i)$. (The single bubble of level one, \mathcal{B}_1 , will have no use for a leader.) The tree $\mathcal{T}(\mathcal{B}_i)$ is known to every member of \mathcal{B}_i ⁵. Note that in particular, every member of \mathcal{B}_1 maintains in its memory $\mathcal{T}(\mathcal{B}_1)$, which is a spanning tree of the entire communication graph.

Routing Strategy: P delivers a message m to a processor Q by the following procedure:

Let $i_1 \geq k$ be the lowest level in which P is a member, and let \mathcal{B}_{i_1} be its home bubble. Then P searches for Q in the spanning tree description $\mathcal{T}(\mathcal{B}_{i_1})$. If Q is not found in $\mathcal{T}(\mathcal{B}_{i_1})$ then P sends the packet $(\text{path}_{L_1}, Q, m)$ to $L_1 = L(\mathcal{B}_{i_1})$, the leader of \mathcal{B}_{i_1} .

This leader, L_1 , repeats the process. I.e., let $i_2 \leq i_1$ be the lowest level in which L_1 is a member, and let \mathcal{B}_{i_2} be its home bubble. Then L_1 searches for Q in the spanning tree description $\mathcal{T}(\mathcal{B}_{i_2})$, and if Q is not found then L_1 sends the packet $(\text{path}_{L_2}, Q, m)$ to $L_2 = L(\mathcal{B}_{i_2})$.

This process must terminate (at a member of \mathcal{B}_1 in the worst case) since every member of \mathcal{B}_1 has a spanning tree description of the entire communication graph, namely, $\mathcal{T}(\mathcal{B}_1)$.

Routing Update: Whenever we apply procedure $\text{COMBINE}(\mathcal{B}'_i, \mathcal{B}''_i)$, the members of the combined bubble has to be updated regarding the new tree. Likewise, whenever we apply $\text{SPLIT}(\mathcal{B}_i)$, the members of the two split bubbles have to be updated. Finally, upon adding a new link for reconnecting the tree \mathcal{T} , every member of \mathcal{B}_1 (namely, every level 2 leader) is updated with the new tree \mathcal{T} , similar to any other COMBINE operation.

Theorem 3.3 *The bubble routing scheme has the following properties: (1) Super-hop count = k . (2)*

⁵For load balancing reasons, the representation of the spanning tree can be extended to include the full topology of the bubble by increasing the memory requirement by a factor of δ .

Memory requirement = $k\delta n^{1+1/k}(\log n + \log \delta)$. (3)
Adaptability = $3^k \delta^2 n^{1/k}$ in the *node-failure model*, and
 $3^k \delta n^{1/k}$ in the *link-failure model*. ■

3.4 Improved Partitioning — Edge-Bubbles Partition

The Bubbles routing scheme as described above is efficient when δ is small, and in particular, for bounded degree networks there is a little significance to this factor. However, if δ is large (close to n), and we consider only link failures, the Bubbles scheme may not compare favorably even with the simple Multiple Trees scheme, because of the occurrences of δ in the expression for adaptability (and in the expression for memory). For such a case it is desirable to get rid of the dependence of the complexities on δ . Note that a δ factor for the adaptability is inherent to the problem if we consider a model that allows node failures and recoveries. This δ factor is based on the fact that a node addition can cause the connection of δ previously disconnected components.

We now sketch a modified partitioning scheme called the *edge-bubbles partition*, that eliminates a δ factor (occurring in both the bound of adaptability and memory) at the cost of using a somewhat more complex partition and algorithm. In order to partition the communication graph into node disjoint connected components, the bubble scheme presented above allows the range of the bubble size to be $[x, \delta x]$. It turns out that it is possible to form a bubble-like partition of a tree into *edge-disjoint* trees (i.e., with bubbles possibly sharing nodes), such that the size of each tree is between x and $3x$.

Definition 3.4 *Given a graph $G = (V, E)$ and parameters $0 < a < b$, an $[a, b]$ edge-bubbles partition of G is a (possibly node-overlapping) cover of V by a collection of edge-disjoint connected trees called bubbles, $\{\mathcal{B}_1, \dots, \mathcal{B}_q\}$, where $\mathcal{B}_i = (V_i, E_i)$, with the following property: $E_i \cap E_j = \emptyset$ for every $1 \leq i < j \leq q$.*

We next present an overview a partitioning algorithm, that given a n -node graph G and a parameter $1 \leq x \leq n$, produces an $[x, 3x]$ edge-bubbles partition for G . The new algorithm is similar in nature to the PARTITION algorithm of Figure 2. The essential differences between the two algorithms are that (i) M_P is the total number of edges in the subtree \mathcal{T}_P , and (ii) The treatment of a node P found in step (2) of the algorithm, namely, a node P satisfying $M_P \geq x$, such that all of P 's children Q satisfy $M_Q < x$, is different: For a particular edge e from P to one of its children, Q , we

define a subtree, \mathcal{T}_e , rooted at e to be the subtree rooted in Q together with e . Recall that the previous algorithm simply makes the subtree \mathcal{T}_P rooted at P into one bubble, containing, in particular, all the subtrees rooted at P 's children. In contrast, the new algorithm will select *some* of the subtrees \mathcal{T}_e rooted at P 's outgoing edges of total size greater than x but not exceeding $2x$. The selected subtrees are now merged into an edge-connected bubble. This bubble is removed from the tree⁶. This modified definition of a bubble partition creates some difficulties also at the stage of the dynamic maintenance of the partition. We again defer the details of handling these complications to the full paper.

Theorem 3.4 *The edge-bubbles routing scheme has the following properties: (1) Super-hop count = k . (2) Memory requirement = $k n^{1+1/k}(\log n + \log \delta)$. (3) Adaptability = $3^k \delta n^{1/k}$ in the node-failure model, and $3^k n^{1/k}$ in the link-failure model. ■*

4 Lower Bound

In this section we give the main ideas for our lower bound on the adaptability for graphs with bounded degree δ and super-hop count k . Given any source-oblivious routing scheme with super-hop count k we build a spanning tree \mathcal{T}_k as follows. The root of the spanning tree is a processor P . Assume that every processor is to send a message m to P , and connect each processor Q with the destination of its first (super-hop) packet when Q sends m to P . The *depth* of a tree is the maximal number of edges from the root to a leaf. Clearly, because the super-hop count is not greater than k the depth of \mathcal{T}_k is no more than k . Thus, there exists at least one node, Q , in \mathcal{T}_k with at least $(n/2)^{1/k}$ children. The node Q is connected to the rest of the processors in G by at most δ links. Thus, by the pigeon-hole principle there must be a link, (Q, R) , used by a set C of at least $(n/2)^{1/k} / \delta$ processors in sending their first packet to carry the message to P . It is easy to see that a removal of this link (and addition of at most one other link to maintain connectivity if necessary) implies $\Omega(n^{1/k})$ adaptability.

Theorem 4.1 *In a bounded degree network, any source oblivious routing scheme with super-hop count k has adaptability $\Omega(n^{1/k})$. ■*

⁶Note that the resulting bubbles are of size $[x, 3x]$ since the partition of the last bubble into two bubbles of size in the range $[x, 2x]$ might not be possible in the case P has three outgoing edges for which the subtree rooted at each of them is of size less than x but greater than say, $3x/4$.

5 Distributed Bubbles Update

So far we have not been concerned with the issue of how to distribute the bubbles update algorithm. Still the above upper bound for the adaptability of the bubble routing scheme is useful for the case of manually adding and removing links and nodes. This is the case for telephone networks where new users may be connected and existing users may be disconnected. However, our results are made stronger by handling automatically topology changes while keeping the low adaptability. For the sake of keeping a low adaptability we introduce a distributed bubbles update algorithm that can cope with transient failures and recoveries as well as permanent disconnection and connections.

The main idea for the distributed bubble update is to have the processors neighboring a topology change to be the *monitors* of the change. When several monitors exist for the same connected component then all but the monitor with the highest identifier stop being monitors. The task of the (left) monitor is (1) to collect information on the tree structure it belongs to and on the existing routing data-base, (2) to try to merge with other neighboring trees and (3) to modify the existing routing data-base to fit the current topology. The modification should be performed in a way that involves the least number of node control units. The monitor uses additional features of the switching subsystem to support a propagation with feedback procedure that collects information on the existing data-base of the connected component to which the monitor belongs. Then the collected data-base is used for determining the corrections required. Further details are deferred to the full version.

6 Concluding Remarks

In this paper we defined new measures for the efficiency of routing schemes for high-speed dynamic networks. We presented the concept of bubble-partition of a graph to support the routing in high-speed dynamic networks. Our routing scheme is efficient in terms of the number of routing steps (super-hop count), memory, and adaptability. The routing scheme also support load balancing of traffic by allowing each processor to forward a message along a randomly chosen path within its bubble (the path does not necessarily belong to the bubble tree). The scheme had been proven optimal in its adaptability for bounded degree networks by a matching lower bound.

References

- [AGR89] Y. Afek, E. Gafni, and M. Ricklin. "Upper and lower bounds for routing schemes in dynamic networks," *Proc. 30th Symp. on Foundations of Computer Science*, pp. 370-375, 1989.
- [Aw85] B. Awerbuch. "Complexity of network synchronization," *JACM*, 32(4):804-823, 1985.
- [AB+89] B. Awerbuch, A. Bar-Noy, N. Linial, and D. Peleg. "Compact Distributed Data Structure for Adaptive Routing," *Proc. 21th Symp. on Theory of Computing*, pp. 479-489, 1989.
- [AB+90] B. Awerbuch, A. Bar-Noy, N. Linial, and D. Peleg. "Improved routing strategies with succinct tables," *J. of Algorithms*, 11:307-341, 1990.
- [AP92] B. Awerbuch and D. Peleg. "Routing with polynomial communication-space tradeoff," *SIAM J. on Discrete Math*, 5(2) pp. 151-162, 1992.
- [AP+92] B. Awerbuch, B. Patt-Shamir, D. Peleg, and M. Saks. "Adapting to asynchronous dynamic networks," *Proc. 24th Symp. on Theory of Computing*, pp. 557-570, 1992.
- [CG88] I. Cidon and I. Gopal. "PARIS: An approach to private integrated networks," *Journal of Analog and Digital Cabled Systems* 1(2), pp. 77-86, 1988.
- [CGZ94] I. Cidon, O. Gerstel and S. Zaks. "A Scalable Approach to Routing in ATM Networks," *Proc. 8th Int. Workshop on Distributed Algorithms*, 1994.
- [GZ94] O. Gerstel and S. Zaks. "The Virtual Path Layout Problem in Fast Networks," *Proc. 13th ACM Symp. on Principles of Distributed Computing*, 1994.
- [KK77] L. Kleinrock and F. Kamoun, "Hierarchical routing for large networks," *Computer Networks* 1, (1977), 155-174.
- [KK80] L. Kleinrock and F. Kamoun, "Optimal clustering structures for hierarchical topological design of large computer networks," *Networks* 10, (1980), 221-248.
- [PU89] D. Peleg and E. Upfal. "A tradeoff between size and efficiency for routing tables," *JACM*, 36:510-530, 1989.
- [SK85] M. Santoro and R. Khatib. "Labeling and implicit routing in networks," *The Computer Journal*, 28:5-8, 1985.
- [Va81] L. G. Valiant. "Universality consideration in VLSI circuits," *IEEE Transactions on Computers*, 30, 135-140, 1981.
- [vLT86] J. van Leeuwen, and R.B. Tan. "Routing with compact routing tables," *The Book of L*, G. Rozenberg and A. Salomaa, eds. Springer-Verlag, New York, 1986, pp. 259-273.
- [vLT87] J. van Leeuwen, and R.B. Tan. "Interval Routing," *The Computer J.*, 30 (1987), 298-307.

- (1) Mark every node P in \mathcal{T}_R by M_P , the total number of nodes in the subtree \mathcal{T}_P rooted at P .
If $M_R \leq \delta x$ then make \mathcal{T}_R into a bubble and terminate.
- (2) (a) Find a node P with $M_P \geq x$, but all of whose children Q satisfy $M_Q < x$.
(b) Make \mathcal{T}_P into a bubble.
(c) Remove this bubble and the edge connecting P to its parent from the graph.
- (3) Assign \mathcal{T}_R to be the connected tree rooted at R following (2), and goto (1).

Figure 2: Algorithm PARTITION(x).

For $i = 2$ to k **do**:

- (1) Apply algorithm PARTITION(x_i) separately to every bubble of the previous level partition \mathcal{P}_{i-1} . Use the portion of \mathcal{T}_R that spans the bubble at level $i - 1$.
- (2) Add the resulting bubbles to the current level partition \mathcal{P}_i .

Figure 3: Algorithm HIERARCHY(\bar{x}).

Input: Two tree-neighboring level l bubbles \mathcal{B}'_l and \mathcal{B}''_l

Action:

Connect $\mathcal{T}(\mathcal{B}'_l)$ and $\mathcal{T}(\mathcal{B}''_l)$ using their connecting link.

Output: A combined bubble \mathcal{B}_l .

Figure 4: Procedure COMBINE($\mathcal{B}'_l, \mathcal{B}''_l$).

Input: A (typically oversized) bubble \mathcal{B}_l .

Action:

Select one of the nodes of \mathcal{B}_l to be the root R .
Direct $\mathcal{T}(\mathcal{B}_l)$ from R towards the leaves.
Execute steps (1) to (3) of Algorithm PARTITION(x_l).

Output: An $[x_l, \delta x_l]$ -bubbles partition of \mathcal{B}_l .

Figure 5: Procedure SPLIT(\mathcal{B}_l).

Input: Edge e for removal, forest \mathcal{F}_l spanning a legal partition.

Action:

The removal of e_i causes the partition of a single bubble, say \mathcal{B}_l , into two portions, \mathcal{B}'_l and \mathcal{B}''_l .

For each of these two portions $\tilde{\mathcal{B}}$ **do**:

1. If the size of $\tilde{\mathcal{B}}$ is still greater than x_{l+1} , then make it into an independent bubble without change.
2. Otherwise, if it is too small, then do the following:
 - (a) Find some bubble $\tilde{\mathcal{B}}$ of \mathcal{F}_l that is a tree neighbor of $\tilde{\mathcal{B}}$ w.r.t. \mathcal{F}_{l-1} .
 - (b) Merge $\tilde{\mathcal{B}}$ and $\tilde{\mathcal{B}}$ through invoking procedure COMBINE($\tilde{\mathcal{B}}, \tilde{\mathcal{B}}$).
 - (c) If the combined bubble includes more than δx_{l+1} nodes, then split it using procedure SPLIT.

Output: Forest \mathcal{F}'_l .

Figure 6: Procedure REMOVE(e, \mathcal{F}_l).

Input: Edge e for removal, forests \mathcal{F}_l for $i \leq l \leq k$ spanning legal partitions.

Action:

Let $\mathcal{F}'_i \leftarrow \text{REMOVE}(e, \mathcal{F}_i)$.

For $l = i + 1$ to k **do**:

1. Let $\mathcal{F}_l - \mathcal{F}'_{l-1} = \{e_1, e_2, \dots, e_m\}$.

2. Let $\mathcal{F}_l^{(0)} \leftarrow \mathcal{F}_l$.

3. **For** $j = 1$ to m **do**:

If e_j is in $\mathcal{F}_l^{(j-1)}$, then invoke $\mathcal{F}_l^{(j)} \leftarrow \text{REMOVE}(e_j, \mathcal{F}_l^{(j-1)})$.

4. Let $\mathcal{F}'_l \leftarrow \mathcal{F}_l^{(m)}$.

Output: Forests \mathcal{F}'_l for $i \leq l \leq k$.

Figure 7: Procedure REMOVE_ALL(e, i).