Scalable Computing: Practice and Experience
Volume 6, Number 3, pp. 67–84. http://www.scpe.org
ISSN 1895-1767

AN ADAPTIVE FILE DISTRIBUTION ALGORITHM FOR WIDE AREA NETWORK

TAKASHI HOSHINO* , KENJIRO TAURA* , AND TAKASHI CHIKAYAMA*

Abstract. This paper describes a data distribution algorithm suitable for copying large files to many nodes in multiple clusters in wide-area networks. It is a self-organizing algorithm that achieves pipeline transfers, fault tolerance, scalability, and an efficient route selection. It works in the presence of today's typical network restrictions such as firewalls and Network Address Translations, making it suitable in wide-area setting. Experimental results indicate our algorithm is able to automatically build a transfer route close to the optimal. Propagation of a 300MB file from one root node to over 150 nodes takes about 1.5 times as long as the best time obtained by the manually optimized transfer route.

Key words. Self-stabilizing distributed algorithm, fault tolerance, scalability, wide-area network

1. Introduction. This paper describes a practical algorithm for copying large data (typically in a file) from a source node(s) to many destination nodes in parallel. We seek a scalable solution suitable both within a cluster and across many clusters in wide-area. By suitable within a cluster, we mean that it fully utilizes the available bandwidth of LAN/cluster interconnect. For example, assuming 32 nodes are connected via a sufficiently high-throughput switch, it should be able to copy a single large file to the 32 nodes in not much more than the time it takes to copy the file to a single node. Such an algorithm must at least perform many one-to-one transfers in parallel. By suitable in wide-area, we mean it makes a good choice in selecting transfer routes. If many nodes in a cluster retrieve data from another cluster, a link across the two easily saturates. Thus such an algorithm should have a mechanism to transfer data within a cluster where possible.

To be practical, it should work with a simple and small manual configuration that may not be very accurate. It won't be practical to assume, for example, that the user gives a complete and accurate information about physical network topology, desirable paths for transferring data, or even logical network connectivity (i.e., network settings such as firewall and Network Address Translation (NAT)). Assuming such information is not practical not only because the user may not want to write them, but also because such information may change over time due to such events as node/network failures. The system therefore must tolerate inaccurate information and adapt to the conditions observed at runtime. Such an adaptive system naturally supports fault tolerance in the sense that even if some nodes fail, remaining nodes accomplish their work and nodes that once failed can join the transfer again.

We believe such a fault-tolerant and adaptive file replicator is a mandatory building block for cluster and Grid computing. It may be used for installing large program/data to many nodes. It may also be used in file synchronizers [5] so they support synchronizing data among a large number of nodes in parallel. Perhaps most important, replicating a large data to many nodes will be a practical technique to "reset" a distributed computation; it simply reinitializes all the involved nodes, so as to recover from some broken/inconsistent states. This observation accords with recent practices in large-scale cluster management, where reinstalling operating systems from scratch is considered as a normal operation, rather than the last resort, to fix broken clusters [13].

To get an intuitive idea about how a good transfer route typically looks, consider a network in Figure 1.1.

There are two local area networks (LANs) named A and B, each including three clusters (A_1, A_2, and A_3 in A and B_1, B_2, and B_3 in B). Assume nodes can connect to each other via the TCP layer.

Suppose the data is on a node in cluster A_1 and should propagate to all other nodes. In the figure, a small circle is a node, a rectangle a switch, and a line connecting a node and a switch a network cable that can transfer data with 100Mbps.[1]

Intuitively, the best strategy is to form a transfer route like the one shown in Figure 1.2. Figure 1.3 represents the same route in the physical topology. Specifically, the following two properties are important.

- The number of connections that cross a LAN/cluster boundary is small; there is only one connection across the two LANs and five connections across the six clusters.
- The entire transfer route forms a list. That is, no nodes serve data to two or more nodes.

The reason why the first property is important will be clear. A simple calculation will reveal that if nodes are randomly connected without any effort to connect nodes close to each other, links across LANs/clusters will

*University of Tokyo, {hoshino,tau,chikayama}@logos.t.u-tokyo.ac.jp
[1]Of course, this limit may not be due to the capacity of the cable *per se*, but due to NIC or switch.

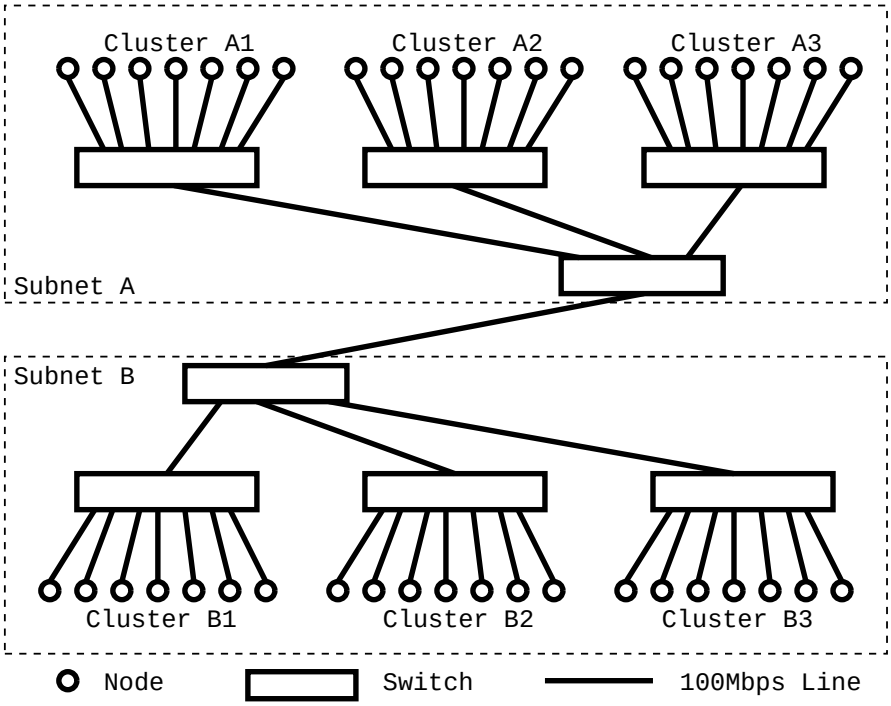

FIG. 1.1. *Typical network environment for which our solution is suitable*

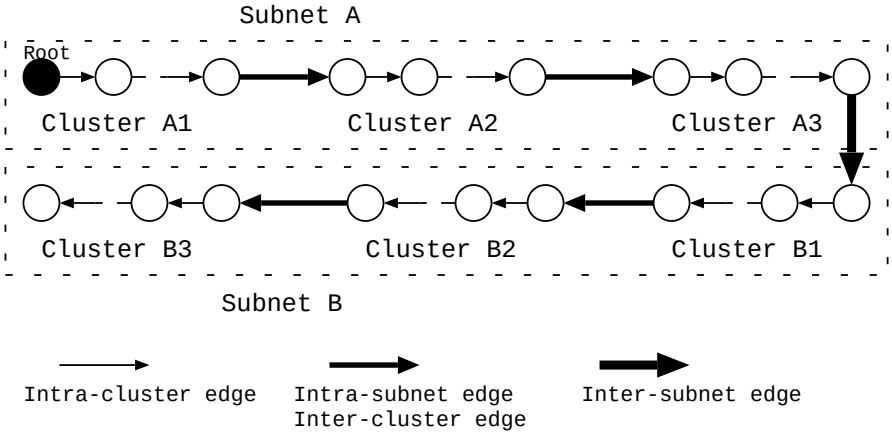

FIG. 1.2. *Best transfer route in application layer*

easily become a bottleneck. This is especially true in today's typical network configuration where capacity of long links (corporate-/campus-/wide- area) is similar to or at best only an order of magnitude larger or so than typical local area links. For example, let us assume for the purpose of discussion that we have two 100Mbps switched LANs connected via a 1Gbps link. In such settings, we should be able to transfer data among all the nodes in the two LANs approximately at the LAN bandwidth (100Mbps), but if connections are randomly chosen, a link across the two LANs can sustain only 10 such connections at best. Thus the 1Gbps link won't be enough for supporting 10 or more nodes in each side of it.

The second bullet may be less obvious. It is important for reducing the bottleneck in NICs. Suppose three nodes A, B, and C are linked via a 100Mbps switch. If data go from A to B to C, the throughput will be close to 100Mbps. If, on the other hand, A sends data both to B and C simultaneously, it can emit data at 50Mbps to each. Note that we assume A must send data to B and C separately, which we believe is a reasonable assumption because B and C may want different portion of the entire data stream. This is important especially

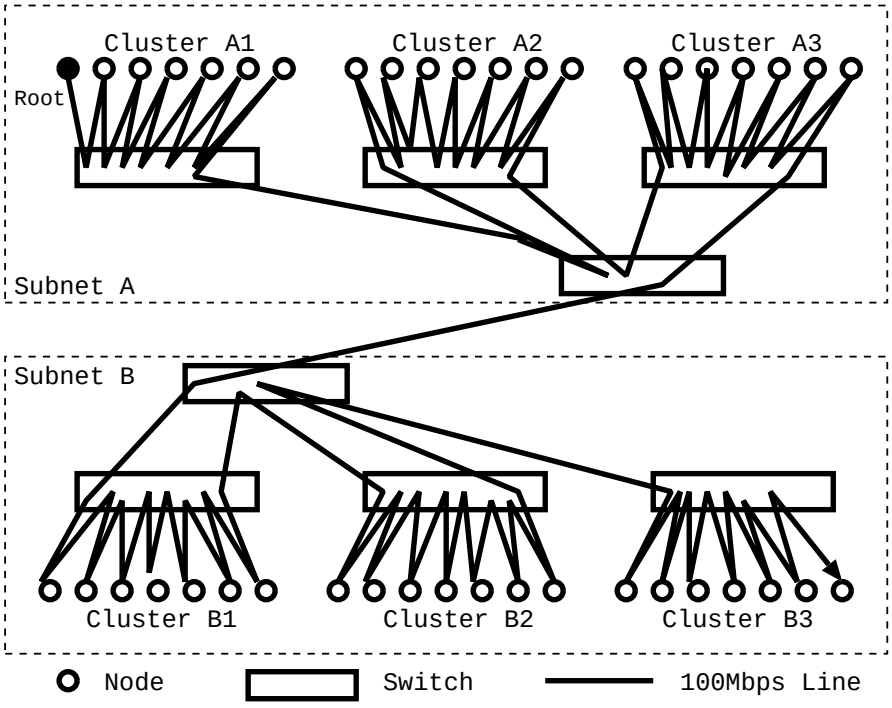

FIG. 1.3. *Best transfer route according to our guidelines in typical network*

when links across LANs are sufficiently powerful, so they won't become bottlenecks as long as we maintain the first property.

Our algorithm tries to build a transfer route close to such best routes. Note that it is not always possible to connect all nodes in a list. For example, if firewalls do not allow some connections, it may be unavoidable for some nodes to serve data to two or more children. Thus, our algorithm in general forms a transfer *forest*, with some heuristics to connect nodes close to each other and to make the tree deeper. It may be a forest, rather than a single tree, because there may be multiple nodes that have complete data in the beginning. In such cases, a separate tree will be formed rooted at each source node.

The paper is organized as follows. Section 2 describes a model of the network and the goal of this research. Then, we propose our algorithm and proof of efficiency in Section 3. And validation and evaluation are shown in Section 4. In Section 5, we explain related work. Finally, we conclude and summarize this research and remark to future work in Section 6.

2. Problem Description.

In this section, we define goals of the algorithm and formalize the problem.

2.1. Goals.

Tolerate faults and adapt to resource conditions: Copying a large file to many nodes takes a long time. Therefore our solution must tolerate temporal/permanent network faults and node crashes. When a node crashes, nodes receiving data from the crashed node must find a substitute so that the remaining nodes finish their tasks. When a node recovers, it must be able to join the transfer network and continue its job, without waiting for the ongoing operation to finish and then restarting from scratch. In addition to being fault-tolerant, it must adapt to changes in network conditions; it should change the transfer route depending on changes of conditions.

Both of these requirements preclude a simplistic solution that statically constructs a route in the beginning and tries to retain the same route until they finish. Nodes must continuously search for a better transfer route.

Make an efficient transfer route automatically: As motivated in Section 1, our general criteria for "good" transfer route are (1) using a small number of "long" connections (i.e., connections that travel a large number of hops, such as inter-subnet connections), and (2) having a small number of nodes that serve

data to multiple (more than one) children. This is based on our assumption that a bottleneck is typically caused by an inter-subnet edge or a node. Examples for the latter are disks and network interfaces. Our algorithm tries to optimize the number of long connections and the number of children for each node, with a very simple local search heuristics.

Work on today's typical network configurations: Today's typical network configurations do not allow each node to connect to all other nodes. Firewalls may block connections between LANs. Inside a LAN, it is common to place all cluster nodes but one (a master node) behind a NAT router, so that accesses to clusters need go through the master. With DHCP, it may even be impractical to assume all nodes to have persistent names.

In short, we must model the network as a general graph where allowed connections are represented by its edges. Yet it is impractical to assume such a graph is given by the user (or the administrator) either offline or in the beginning of the algorithm. Altogether, we must design an algorithm that begins with a minimum amount of global information (e.g., participating nodes) and a local knowledge of the network (e.g., neighbors) in each node.

Do not assume physical network topology: Knowing physical network topology would help us to optimize transfer routes. Designing the algorithm assuming a complete knowledge about it is, however, impractical for many reasons discussed so far. First it is cumbersome for the user or the administrator to maintain such information. We may be able to obtain such information by using tools such as traceroute, but such tools tend to be unavailable these days for security considerations. It is also difficult to obtain the topology of the network behind a single router with traceroute. Second, even if topology information is available, dynamically probing the network is always necessary to make the algorithm fault-tolerant and adaptive. Algorithms based on probing connectivity and proximity at runtime naturally work without detailed knowledge about network topology.

Of course, we could always use physical topology as hints to our algorithm, among many other hints such as IP address prefix, latency, and observed throughput.

To achieve these goals, each node involved in our algorithm continuously seeks a parent, a node that serves data to the node. When it faces such events as parent crashes or disconnections, it tries to find a new parent. Even without such events, they continuously search for a better parent to optimize the transfer route. The criteria for a better parent are that (1) the closer a node is to itself the better, and (2) the fewer children a node has the better.

Our algorithm is a simple local search algorithm that converges to a satisfactory transfer in typical network configurations of today. Ideally, we desire an algorithm to find a globally optimal solution for any given network. A plausible definition of the optimal would be to minimize the sum of selected edge weights and the number of branches (or equivalently, the number of leaves) in the graph. The two criteria may conflict for general weighted graphs and even if they do not, they will require a complex global optimization algorithm (e.g., fault-tolerant MST construction) whose practical importance may not be very clear. In the following, we formulate our problem and prove our simple algorithm has a property which translates to "a sufficiently good" transfer route in typical real network configurations.

2.2. Problem Formulation. As usual, we model the network by a directed graph $G = \langle V, E \rangle$, where V is a set of nodes participating in the replication. E represents possible connections between nodes; $(a, b) \in E \iff$ a knows b's name and the current network status allows a to connect to b.

The graph is for modeling purposes only; in practice, the network status may change over time, so each node cannot know the complete status of the network. It may even be impractical to assume each node knows all the neighbors it can connect to. In our implementation, each node begins with knowing information about a few of its neighbors and receiving a command that instructs it to participate in the replication of a file. They learn other node names on the fly by propagating information along established connections. This way, they learn other connections they may be able to make. They learn whether a particular connection is allowed or not by trying to establish a connection only when necessary. Nodes never maintain information about edges they are not adjacent to.

Below, we prove our optimization algorithm eventually reaches a transfer forest that has some desirable properties, assuming that the graph is fixed at some point. Note that our algorithm correctly finishes its job without this assumption. The assumption is essential only for stating the property of the forest our algorithm converges to.

To define the "goodness" of a transfer forest, we must introduce a notion of distance between nodes. One plausible formulation would be to give edges arbitrary weights, and to aim at reducing the total weights of selected edges (i.e., minimum spanning forest). We do not use this formulation but introduce a stronger assumption about the distances between nodes which we believe is a practical approximation of real networks, and show a simpler local search obtains sufficiently good results.

We assume nodes can be decomposed into groups so that nodes close to each other constitute a group. Our optimization algorithm does not assume that each node knows the decomposition explicitly, but only assumes that each node can somehow compare relative distances from the local node to other nodes. We show in Section 3.3 such a comparison induces a decomposition. It is such a decomposition for which our algorithm tries to reduce the number of inter-group edges. Again, the replication correctly finishes with inaccurate information, thus an implementation can use any sufficiently accurate measurement. Our current implementation is given in Section 3.2.1.

We say a decomposition is *complete* if nodes in each group form a clique (a complete subgraph) of G. That is, nodes inside a group can connect to each other without being blocked by, e.g., firewalls. For any decomposition which may or may not be complete, one can derive a complete decomposition by dividing its incomplete group into a number of groups so each of them is a clique. We call such a complete decomposition a *complete subdivision* of the original decomposition. Given a decomposition D, a complete subdivision that has the minimum number of groups is called the *coarsest* subdivision of D.

Given a decomposition, the goal would be to make a transfer forest close to the following *best desirable*, which has

1. the minimum number of edges connecting nodes in different groups, and
2. the minimum number of branches.

Our algorithm converges to the optimal if each node can connect to any other node (i.e., the entire graph is complete, or in practical terms, firewall, NAT, or DHCP do not deny any connection against us). In more general graphs, our algorithm has the following property. Let D the decomposition induced by a heuristics used to measure the relative distance between nodes, and $\overline{D}$ the coarsest complete subdivision of D. Our algorithm achieves (1) the number of inter-group connections $\leq \overline{N} - F$ and (2) the number of branches $\leq \overline{N} - 1$, where $\overline{N}$ is the total number of groups in $\overline{D}$ and F the number of groups in $\overline{D}$ containing at least one *finished* node, a node which has received the entire data.

Our claim that the above property translates to a good result in practice is based on the following observations.

- A simple measurement can reasonably approximate the "closeness" between nodes. For example, given a node in the same LAN as the local node and another not in the same LAN, it will be relatively easy for the local node to judge if one node is closer to the other, thus should be preferred. Therefore, one can obtain a decomposition each group of which has nodes close to each other.
- In typical network configurations, nodes close to each other tend to be allowed to connect to each other. Most typically, nodes within a LAN can connect to each other. Making a group of nodes close to each other thus tends to yield a subgraph that is nearly complete.

The first bullet implies that, if we group nodes based on a reasonably accurate measurement of distances between them, we will have groups each of which consists of nodes close to each other. Each such group will be nearly complete (bullet #2), therefore $\overline{N}$ will be close to N. Together, the number of connections crossing a group boundary will be close to $N - F$, and the number of branches close to $N - 1$.

3. Algorithm. The algorithm has several features that we should remark.

A simple, self-stabilizing distributed algorithm: Each node works based on information about its neighbors and optimizes transfer routes with a small amount of local information. Each node *continuously* seeks a closer node that may serve data faster. This mechanism naturally makes our algorithm fault-tolerant and allows nodes to join or leave computation at any time.

Parallel and pipelined transfer: Transferring data from node A to B and from C to D can occur in parallel. Moreover, transferring a piece of data from A to B and transferring another piece of data from B to C can also take place in parallel (pipelined transfer). This is especially important for replicating large files in switched networks.

A simple transfer loop avoidance: The algorithm naturally avoids deadlock due to a transfer loop simply by letting each node become a parent of another only when it has more data than others. This mechanism,

together with the self-stabilizing nature of the algorithm, is enough to make it deadlock-free; when a node crashes, its children will eventually learn there is no progress for a long time, in which case they try to connect to another node that is ahead of it.

```
01: /* Starting or After Recovered */
02: offset = current filesize on disk;
03: parent = invalid; /* the node self is getting data from.
*/
04: candidate = null;
05: is_sending_giveme = false;
06: children = none; /* nodes self is giving data to */
07: siblings = none; /* used for Tree2List Suggestion */
08: neighbors = list of neighbors (dead or alive);
09: while (true) {
10:   /********** Searching for Parent **********/
11:   (candidate == null && parent == invalid) ⇒
12:     candidate = a node in neighbors;
13:     send(candidate, ask(id, offset));
14:   /* NearParent Heuristics */
15:   (candidate == null && a node in neighbors satisfies
16:   is_closer(self, node, parent)) ⇒
17:     candidate = node;
18:     send(candidate, ask(id,offset));
19:   /* Tree2List Heuristics */
20:   (candidate == null && a sibling in siblings satisfies
21:   !is_closer(self, parent, sibling)) ⇒
22:     candidate = sibling;
23:     send(candidate, ask(id,offset));
24:   received(ask(wid, woffset)) ⇒
25:     if ((offset > woffset) &&
26:       (MAX_NODE > number of children)) {
27:       add this node (wid, woffset) to children;
28:       send(wid, ok(id, offset));
29:     } else {
30:       send(wid, ng(id));
31:     }
32:   received(ok(wid, woffset)) ⇒
33:     if (woffset > offset) {
34:       parent = wid; candidate = null;
35:     }
36:   received(ng(wid)) ⇒
37:     if (wid == candidate) {
38:       candidate = null;
39:     } else if (wid == parent) {
40:       parent = invalid;
41:     }
42:   /********** Data Transfer **********/
43:   (parent != invalid && offset < filesize &&
44:   !is_sending_giveme) ⇒
45:     is_sending_giveme = true;
46:     send(parent, giveme(id, offset));
47:   received(giveme(child, woffset)) ⇒
48:     if (offset > woffset) {
49:       size = max(BLOCKSIZE, offset—woffset);
50:       buf = load(filename, woffset, size);
51:       send(child, data(id, woffset, size, buf));
52:     } else {
53:       send(child, ng(id));
54:     }
55:   received(data(wid, woffset, size, buf) ⇒
56:     if (woffset == offset) {
57:       is_sending_giveme = false;
58:       save(filename, woffset, size, buf);
59:       offset += woffset;
60:     }
61:   (offset == filesize && parent != null) ⇒
62:     if (parent != invalid)
63:       send(parent, disconnect(id));
64:     parent = null;
65:   received(disconnect(child)) ⇒
66:     delete the child from children;
67:   /********** Tree2List Suggestion **********/
68:   (having more than one child) ⇒
69:     foreach child in children {
70:       send(child, suggestion(id, children));
71:     }
72:   received(suggestion(parent, new_siblings)) ⇒
73:     siblings = new_siblings;
74:   /********** Fault Handling **********/
75:   (timeout(data, ng) from parent) ⇒
76:     parent = invalid;
77:   (timeout(giveme, disconnect) from child) ⇒
78:     delete the child from children;
79:   (timeout(ok, ng) from candidate) ⇒
80:     candidate = null;
81: }
```

FIG. 3.1. *Pseudo-code of our algorithm*

Figure 3.1 shows the local algorithm running on each node. Prior to running this algorithm, each node knows its neighbors (*neighbors*) and the size of the file each node must eventually have (*filesize*). In actual implementation, each node may begin with an incomplete list of neighbors. Nodes propagate their neighbors to other and learn from others.

Inside the main **while** loop (line 9–81) is written as a list of the following form:

condition ⇒ *action*

where *condition* is a precondition (or a *guard*) in which the *action* can take place. The predicate **received(**X**)** evaluates to true if a message that matches X is in the incoming message queue of the node. Each iteration of the loop waits for at least one guard to become true, and executes the corresponding action. If multiple guards

are true, any one of them is chosen arbitrarily.

First, we explain the base part of this algorithm in Section 3.1. We continue with the route optimization heuristics in Section 3.2

3.1. The Base Algorithm. Each node repeats the following until it gets the entire data.

- It seeks a node that is ahead of itself (i.e., has more data than itself). Let us call such a node *its parent*. A parent may change over time.
- Once it finds a parent, it asks the parent to send the data that should come next to the data it currently has. For example, if a node has the first 1000 bytes of a file, it will ask the parent to send some amount of data from offset 1000.
- In addition,
  - Each node, except ones that have obtained the entire data, seeks a node that is closer to its current parent. Details are in Section 3.2.1.
  - Each node having two or more children tries to resolve this situation, by suggesting children to connect to one of its siblings.

When a node receives an instruction to participate in a replication, each node checks how much data it has (line 2), searches for a candidate node that has data grater than itself by connecting to some nodes in its *neighbors* list. Variable *offset* indicates the size of data at that time, and satisfies the inequality $0 \leq$ *offset* $\leq$ *filesize*. During data transfer, the invariant *child's offset* $\leq$ *parent's offset* is maintained (line 25, 33, and 48).

A node searching for a parent sends an `ask` message carrying its *offset* (data size) to a candidate (line 11–13). If the receiver has more data than the sender, it sends an `ok` message to the node sender (line 24–28, 32–35). At that time, the relation between parent-child is established. After that, the child sends a `giveme` message to the parent (line 43–46) and the parent sends a chunk of data to the child (line 47–51). This repeats until the child either catches up the parent in data size (line 52–54), finds a better candidate than the current parent, or receives an error. If the receiver of `ask` does not have more data than the sender, it sends an answer `ng` (line 29–31) to the sender. Receiving an `ng` message (line 36–41), the node continues to search for a parent.

A node can be a parent of some nodes and a child of another at the same time. In effect, we achieve a pipeline transfer through all nodes.

When a parent becomes unreachable from its child (due to a parent crash or a network failure), the child merely searches for a new parent. When a node recovers, it can participate in the transfer from the offset at the time it has failed. Hence, this algorithm is fault-tolerant (line 74–80).

3.2. Adaptive Transfer Route Optimization. Now, we explain optimizing heuristics on top of the base algorithm (line 14–23, 67–73).

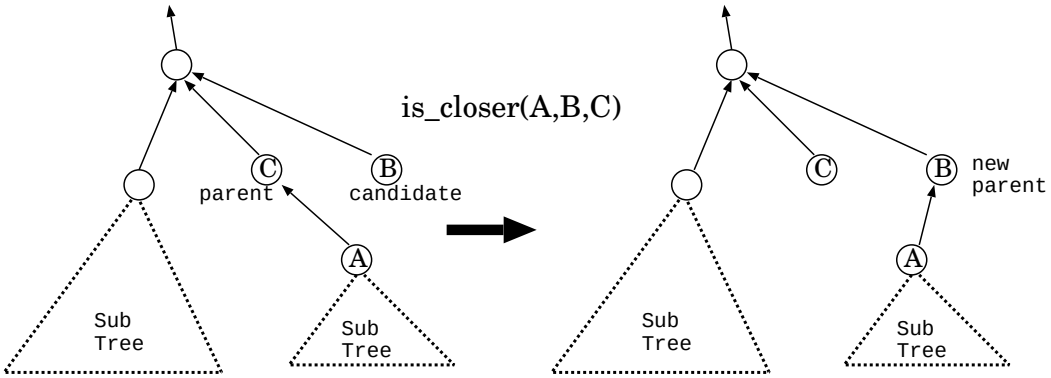

FIG. 3.2. *NearParent Operation*

3.2.1. NearParent Heuristics. Each node periodically tries to connect to a node that is closer to its current parent (*candidate* in Figure 3.2, line 15–18 in Figure 3.1). If the candidate node turns out to have more data than the local node (line 32–35), it selects the new node as the new parent. Figure 3.2 shows how this heuristics modifies a part of the transfer tree.

Note that even if each node has connected to its parent, it searches for an even closer candidate periodically. We have not conducted an extensive study about the best frequency. Frequent measurements will allow us to find a good transfer route fast at the cost of increased network traffic. Our current implementation guarantees that there is at most one traffic from each node for the measurement. It also guarantees each node performs a measurement at most once every 100ms. This will hardly affects CPU or network load.

The predicate to judge if a node B is closer than C from the local node A, is_closer(A, B, C), currently uses the following criteria in the listed order.

Throughput observed in the past: Each node records throughput from each of the nodes that have been chosen as its parent. If A has chosen both B and C as its parent before, whichever produced a better throughput is considered closer.

Observed latency: The above criterion is not applicable when either B or C has never been chosen one as A's parent. In this case A uses latencies it takes to connect to B and C.

The length of the matching IP address prefix: When observed latencies are too close to discriminate, we use IP addresses of A, B, and C. We compare the lengths of the common prefixes of IP addresses of A and B to that of A and C.

For the purpose of proving the theoretical property of the algorithm mentioned in Section 3.3 (also stated as Theorem 3.7), is_closer can be any predicate that satisfies the following properties.

- is_closer(A, B, C) and is_closer(A, C, B) do not become true at the same time.
- For a given A, the binary relation:
  $$\mathrm{R}_A(B, C) \stackrel{\text{def}}{=} \text{is\_closer}(A, B, C)$$
  is transitive. That is,
  $$\text{is\_closer}(A, B, C) \wedge \text{is\_closer}(A, C, D) \Rightarrow \text{is\_closer}(A, B, D)$$
- is_closer$(A, B, C) \Rightarrow$ is_closer(B, A, C)

It will be clear that any reasonable definition of relative distance and an accurate measurement of it, including the ones listed above, will satisfy the first two bullets. The third property may not sound very obvious. Examples that satisfy the property include:

- A definition based on the bottleneck edge on trees. That is, assume nodes are connected via a weighted tree and let is_closer(A, B, C) be true iff the minimum weight on the path between A and B is larger than that on the path between A and C.
- A definition based on the distance on trees. That is, assume nodes are connected via a tree and let is_closer(A, B, C) be true iff the path between A and B is shorter than A and C.
- A definition based on address prefixes. That is, assume nodes are assigned integer addresses and let is_closer(A, B, C) be true iff the length of the matching address prefix between A and B is larger than that between A and C.

Therefore we expect that our current implementation of is_closer based on measured bandwidths between nodes, measured latencies between nodes, and the length of IP address prefixes, will satisfy the third property provided measurements are accurate.

Note that implementing such a predicate does not require any *a priori* notion of groups. Just defining/measuring the relative closeness between nodes will suffice, as long as such a definition/measurement satisfies the above properties. In Section 2.2, we show such a predicate in general implicitly induces a distance between nodes, which in turn induces a decomposition of nodes based on the distance. Our algorithm reduces the number of inter-group edges for a decomposition derived this way.

3.2.2. Tree2List Heuristics. NearParent heuristics reduces the number of edges that cross group boundaries. It however is not useful for reducing the number of branches. Another optimization, called Tree2List heuristics, comes into play to make the transfer route closer to a list.

A node that has two or more children sends its children list to every child (line 68–71). When a node receives a suggestion message, which effectively contains its current siblings, it chooses one in the list as the next candidate if the current parent is not closer to it (lines 72–73, 20–23). Figure 3.3 shows how Tree2List heuristics modifies a part of the transfer tree. Intuitively, Tree2List pushes branches in a transfer tree downwards, hoping the tree eventually becomes a list.

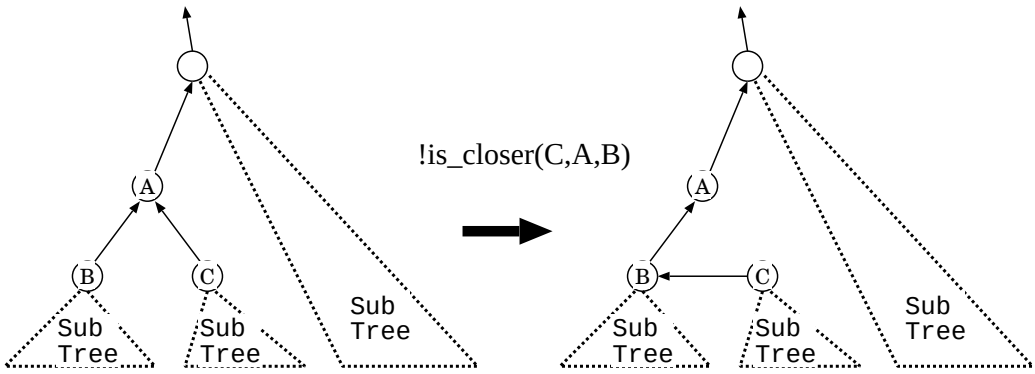

FIG. 3.3. *Tree2List Operation*

An important property about Tree2List, proved in the next section, is that it never increases the number of inter-group edges. This guarantees that applying Tree2List does not impede the NearParent's effort of reducing the number of inter-group edges. In the next section, we state and prove properties of transfer forests after applying both heuristics in an arbitrary order.

3.3. Properties of the Route Optimization Algorithm. Let is_closer satisfy the properties stated in Section 3.2.1. We first show the following, that says is_closer(A, B, C) is equivalent to comparing a distance between A and B and between A and C, for some definition of a distance.

LEMMA 3.1. *For* is_closer *satisfying the property stated in Section 3.2.1, there exists* a distance function d *that satisfies the following.*

- *For all nodes A and B, $d(A, B) = d(B, A)$.*
- *For all nodes A, B, and C,*

$$\text{is\_closer}(A, B, C) \iff d(A, B) < d(A, C).$$

Proof: See Appendix A.1.

The following Lemma is important for guaranteeing Tree2List is applicable when we have many branches.

LEMMA 3.2. *For any d satisfying the condition in Lemma 3.1,*

$$\max(d(A, B), d(A, C)) \geq d(B, C)$$

is true for all nodes A, B, and C. Proof: See Appendix A.2.

A distance function d and a threshold t define a natural decomposition of a graph. That is, we remove all edges (x, y) such that $d(x, y) > t$ from the original graph, and let a group be a connected component of the graph. We call such a decomposition is *derived* from is_closer. Many decompositions can be derived from a single definition of is_closer, depending on the choice of d and t.

We model our route optimization heuristics as a process of rewriting the transfer forest according to NearParent, Tree2List, or finishing the transfer to a node.

DEFINITION 3.3. *A state of computation is a forest among participating nodes, induced by their* `parent` *pointers. Let S and S' be states. We define relations $\rightarrow_n$, $\rightarrow_t$, $\rightarrow_f$, and $\rightarrow$ by:*

1. *$S \rightarrow_n S' \overset{def}{\iff}$ S' is obtained by applying NearParent to S (Figure 3.2),*
2. *$S \rightarrow_t S' \overset{def}{\iff}$ S' is obtained by applying Tree2List to S (Figure 3.3),*
3. *$S \rightarrow_f S' \overset{def}{\iff}$ S' is obtained by finishing a node and making its* `parent` *pointer null, and*
4. *$\rightarrow \overset{def}{=} \rightarrow_n \cup \rightarrow_t \cup \rightarrow_f$. That is,*

$$S \rightarrow S' \overset{def}{\iff} (S \rightarrow_n S') \text{ or } (S \rightarrow_t S') \text{ or } (S \rightarrow_f S').$$

Next, we define some quantities of states. Below, we fix a decomposition D derived by is_closer, and let $\overline{D}$ be the coarsest subdivision of D. Let d and t the distance function and the threshold that induced D. Let $\overline{N}$ be the number of groups in $\overline{D}$. When we say a group, it always means a group of $\overline{D}$. Nodes in a single group by definition form a clique.

Definition 3.4.

- *Let $w(S)$ be the number of edges in forest S that cross group boundaries. For technical convenience, we consider an invalid* `parent` *pointer to cross a group boundary, and a null* `parent` *pointer* not *to cross any group boundary.*
- *Let $f(S)$ be the number of finished nodes (having* `parent` $=$ *null) and $F(S)$ be the number of groups that have at least one finished node. We say such a group is* finished. *Note there may be unfinished nodes in a finished group.*
- *Let $l(S)$ be the number of leaves (i.e., nodes that are not pointed to by any* `parent` *pointer).*

Lemma 3.5. *Transition paths are bounded. That is, the length of a path $S_0 \to S_1 \to S_2 \to \cdots$ is bounded.*
Proof: Define $SUMDIST(S)$, $SUMDEPTH(S)$, and $Q(S)$ as follows.

$$SUMDIST(S) = \sum_{x\,:\,\text{node}} d(x, x\text{'s } \texttt{parent}),$$
$$SUMDEPTH(S) = \sum_{x\,:\,\text{node}} depth(x), \text{and}$$
$$Q(S) = (f(S), -SUMDIST(S), SUMDEPTH(S)),$$

where $depth(x)$ is the number of hops from the root of the tree x belongs to. $d(x, x\text{'s } \texttt{parent})$ is the distance between x and its parent. Again for technical convenience, if x's `parent` pointer is `invalid` we consider it has a value larger than any other $d(y, z)$ for $z \neq$ `invalid`. Similarly, if x's `parent` is `null`, it takes a value smaller than any other $d(y, z)$ for $z \neq$ `null`.

If we introduce a lexicographical order among triples $Q(S)$, it is easy to see $Q(S)$ strictly increases by a single transition step. That is,

$$S \to S' \Rightarrow Q(S) < Q(S').$$

In fact, $\to_f$ increases $f(S)$, $\to_n$ does not change $f(S)$ and increases $-SUMDIST(S)$, and $\to_t$ does not change $f(S)$, never decreases $-SUMDIST(S)$, and increases $SUMDEPTH(S)$.

Since all quantities of the triples are clearly bounded above, we have proved transition paths are bounded. □

Lemma 3.6.

1. *If S satisfies $w(S) > \overline{N} - F(S)$, then $\to_n$ is applicable to S. That is, there exists S' such that $S \to_n S'$.*
2. *If S satisfies $l(S) - f(S) \geq \overline{N}$, $f(S) \geq 1$, and $\to_n$ is not applicable to S, then $\to_t$ is applicable to S.*

Proof:

1. If $w(S) > \overline{N} - F(S)$ (= the number of unfinished groups), either of the following must hold.
   - There is an unfinished group having more than one outgoing inter-group edges.
   - There is a finished group having an outgoing inter-group inter-group edge.

   An *outgoing edge* is a `parent` pointer pointing to a node outside the group. In the former case, let two of such edges be (A, B) and (C, D). A and C belong to one group, say X, while neither B nor D belong to X. Thus, a transition by $\to_n$ that either makes A one of C's children or vice versa, is applicable. In the latter case, let one such edge be (A, B) and one finished node in the group be P. Thus, a transition by $\to_n$ that makes A one of P's children is applicable.
2. We split the proof into two cases, (i) $l(S) - f(S) > \overline{N}$, and (ii) $l(S) - f(S) = \overline{N}$.
   (i) $l(S) - f(S) > \overline{N}$:
   We have at least one group X that satisfies:

   $$l - f > 1$$

   where l and f denote the number of leaves in X and the number of finished nodes in X, respectively. Let $a_1, a_2, \cdots a_l$ be the leaves in X ($l \geq 2$). Let $a_{i,1} = a_i$ and $\vec{a}_i = (a_{i,1}, a_{i,2}, \cdots, a_{i,n_i})$ ($i = 1, \cdots, l$) be chains of `parent` pointers starting from a_i. That is, $a_{i,j}$ is a child of $a_{i,j+1}$) for all i and j ($1 \leq i \leq l$, $1 \leq j \leq n_i - 1$).
   We argue by contradiction that all but one of such chains must be entirely in X. Let us assume w.o.l.g. neither of $\vec{a}_1$ nor $\vec{a}_2$ are in X. Then there are j and k ($1 \leq j \leq n_1 - 1$ and $1 \leq k \leq n_2 - 1$) such that $a_{1,j}$ and $a_{2,k} \in X$, and $a_{1,j+1}$ and $a_{2,k+1} \notin X$. Then a transition by $\to_n$ that connects $a_{1,j}$ and $a_{2,k}$ should be applicable. This contradicts the assumption that $\to_n$ is not applicable in S.
   Now we have $l - 1$ chains entirely in X. Since $l - f \geq 2$ ($\Rightarrow l - 1 \geq f + 1$), at least two of them must merge at some node in X. Let a node at which two merges be A, and B and C the children of A on the

two chains. It remains to show we have either ($\neg$is_closer(B, A, C)) or ($\neg$is_closer(C, A, B)), so either B or C can trigger $\rightarrow_t$. By Lemma 3.2, we have

$$\max(d(A,B), d(A,C)) \geq d(B,C),$$

from which we can derive:

$$\begin{array}{ll} & \max(d(A,B), d(A,C)) \geq d(B,C) \\ \Leftrightarrow & d(A,B) \geq d(B,C) \text{ or } d(A,C) \geq d(B,C) \\ \Leftrightarrow & d(B,A) \geq d(B,C) \text{ or } d(C,A) \geq d(C,B) \\ \Rightarrow & \neg\text{is\_closer}(B,A,C) \text{ or } \neg\text{is\_closer}(C,A,B). \end{array}$$

(ii) $l(S) - f(S) = \overline{N}$:
If we have one group X that satisfies:

$$l - f > 1,$$

then the same discussion as (i) applies. In the remaining case all the groups satisfy:

$$l - f = 1.$$

Let X be any group. As in (i), consider the l chains starting from a node in X. If all the l chains are entirely in X, two of them must merge in X, and the following argument is the same as (i). Therefore each group has exactly one chain outgoing from the group. Then we have $\overline{N}$ inter-group edges, i.e., $w(S) \geq \overline{N}$. This implies, however, $\rightarrow_n$ is applicable because $f(S) \geq 1 \Rightarrow F(S) \geq 1 \Rightarrow w(S) \geq \overline{N} > \overline{N} - F(S)$.

□

THEOREM 3.7. *Along any path of state transitions starting from any state I, we reach within finite steps a state S_∞ satisfying:*

1. $w(S_\infty) \leq \overline{N} - F(S_\infty)$, *and*
2. $l(S_\infty) - f(S_\infty) \leq \overline{N} - 1$.

Proof: From Lemma 3.5, any transition path $I = S_0 \rightarrow S_1 \rightarrow \cdots$ is bounded, therefore reaches a state S_∞ in which neither $\rightarrow_n$ nor $\rightarrow_t$ (or $\rightarrow$, for that matter) is applicable. Lemma 3.6 shows in this state we have both of the above properties. □

Remark 1:. As a special case where $D = \overline{D}$ (i.e., no edges are blocked inside a group of D), we have $N = \overline{N}$. In this case the theorem implies that, for sufficiently long transfers, the number of edges between groups reaches the optimal $N - F(S)$. Replicating a file from $F(S)$ groups to the rest will clearly need $N - F(S)$ inter-group edges. For being close to a list, the second bullet of the theorem implies that the number of branches, effectively calculated by $l(S) - f(S)$, is the optimal $N - 1$. To see this is optimal in general, consider a network configuration shown in Figure 3.4, which forces inter-group edges to form a star.

Remark 2:. Recall that the theorem applies to *any* decomposition derived from is_closer. If the network has multiple levels of hierarchies, (e.g., inside a cluster, clusters inside a LAN, LANs in a campus/corporate area, and LANs in wide area), and is_closer can descriminate all of them, our algorithm simultaneously optimizes all the levels. For example, let us say we have N_1 LANs and N_2 clusters and $f(S) = 1$ as the usual case. If we assume is_closer can descriminate intra-cluster, inter-cluster but intra-LAN, and inter-LAN edges, and the network configuration allows all connections, our algorithm converges to a state in which we have $N_1 - 1$ inter-LAN edges and $N_2 - 1$ inter-cluster edges.

4. Evaluation.

4.1. Implementation. We have implemented the described algorithm in Java. This is executable on common computers supporting Java and TCP/IPv4 protocol. We confirmed the program runs on Solaris (sparc), Linux (x86), Windows (x86), and Tru64Unix (Alpha). Stopping some nodes in the middle of a distribution task did not prevent any of the remaining nodes from finishing the task, confirming its fault-tolerance.

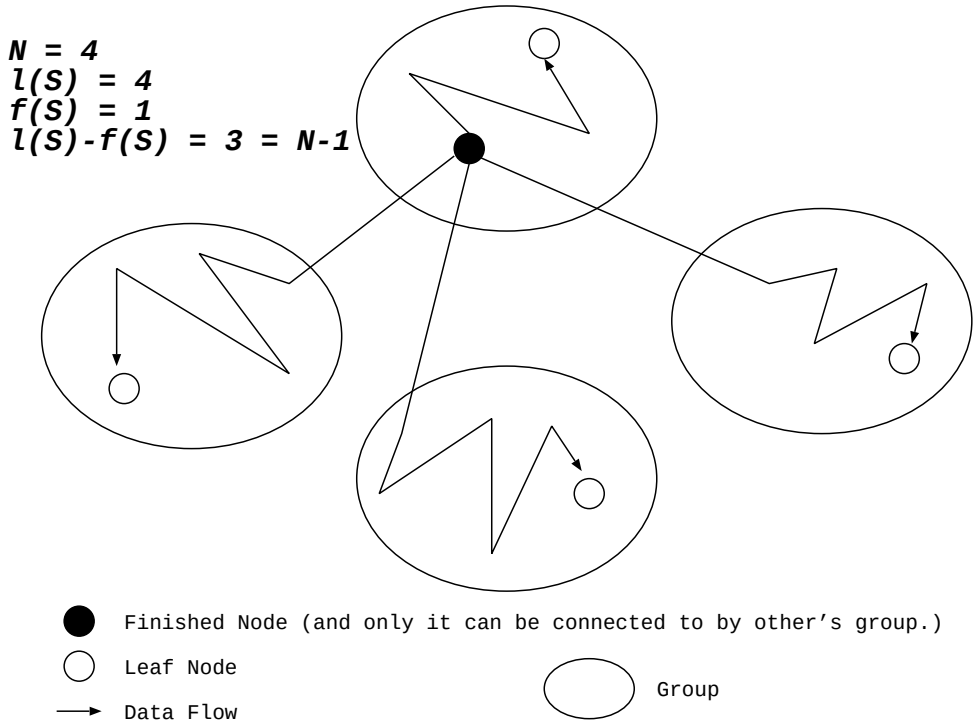

FIG. 3.4. *An example where the optimal value of* $l(S) - f(S)$ *is* $N - 1$

4.2. Single Cluster Experiments. First, we ran some experiments in a single cluster. The cluster consists of 16 nodes. Each node has two Alpha CPUs and a local hard-disk. Network cables of nodes are connected to a 100Mbps switch. Local disk bandwidth is faster than network, so it does not reveal as a bottleneck. CPU is also fast enough.

We initially let one node have 500MB file, and others have no data. Since there is only a single cluster, NearParent optimization does not play any role in this experiment. So this experiment is to see the effect of Tree2List. In addition to Tree2List, we ran the base algorithm without any optimization, changing the maximum number of children each node can serve, from one to five. They clearly demonstrate how important is it to make the transfer tree close to a list.

The time which the distribution tasks spent is shown in Figure 4.1.

In this result, it is clear that the average distribution time increases as the maximum number of children increases. The graph also indicates that, in this particular experiment, limiting the number of children to one yields the best result. That is, restricting the shape of the transfer tree to a list in the first place is better than our Tree2List strategy which first forms an arbitrary tree and then tries to develop it to a list. We believe, however, our strategy has several advantages. First, nodes may not be able to form a list in the presence of firewalls etc. In such cases, one must fall back to a tree. Second, forming a list in the beginning may take much longer than forming a tree, especially when the number of nodes becomes large, since a list can only grow one node at a time.

4.3. Multiple Cluster Experiments. Next, we made experiments in seven clusters illustrated in Figure 4.2. They are all placed in the campus of University of Tokyo.

- An IBM Linux cluster called "istbs" contains 70 nodes. We used all of them for the experiment. Nodes within a cluster are connected via 1Gbps links. A node in this cluster is the source node in this experiment. Bandwidth from/to other clusters below is poor 100Mbps.
- A SunFire15K SMP called "istsun" has 70 CPUs, of which we used 20. We used this machine as if it were 20 separate nodes. It has a 100Mbps NIC shared by all CPUs. Replication of 300MB data among 20 nodes inside istsun takes about 70 sec, where the throughput is about 34Mbps. This seems due to disk I/O bandwidth.

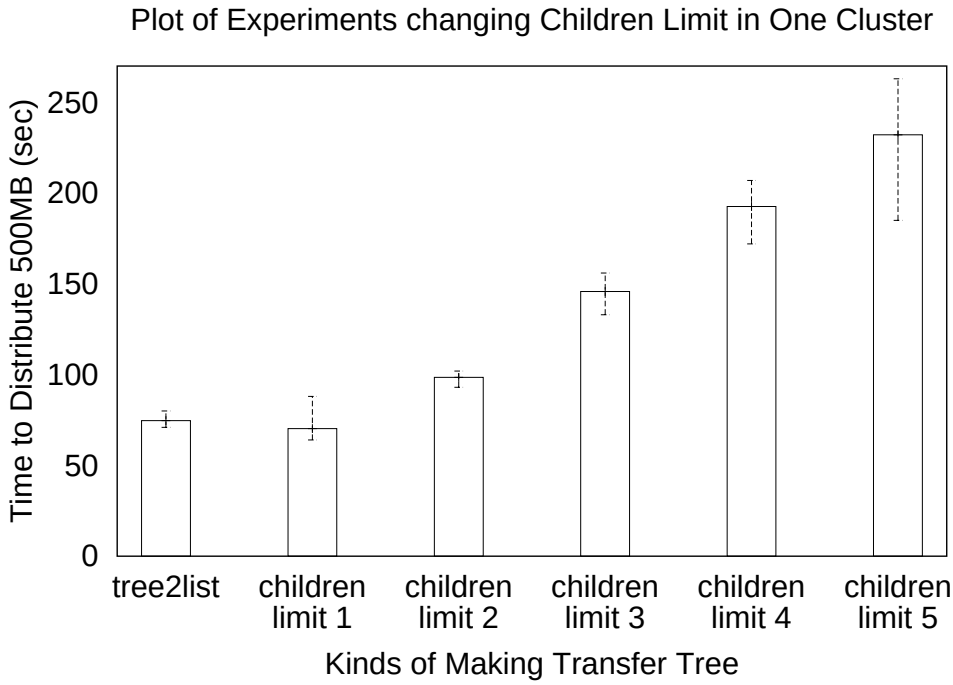

FIG. 4.1. *Performance in a single cluster*

- A cluster of clusters called "kototoi" contains three cluster each having 16 nodes. Network speed is 100Mbps inside each cluster. Throughput between two of the three is several hundreds Mbps. Having more than one connection to a single cluster easily saturates the link. No nodes outside kototoi cannot directly connect to inside it.
- An HP Alpha cluster called "oxen" contains 16 nodes, which is the same cluster in Section 4.2. There are two (and only two) gateway nodes that can connect to and can be connected from outside the cluster.
- A Linux cluster called "marten" each of which runs Linux inside VMWare. Its configuration is almost the same as a cluster in kototoi.
- For connectivity, any node can connect to istsun nodes and the gateways of oxen. Also, istsun and istbs are in the same virtual LAN, so nodes in the two clusters can directly connect to each other. Connections to remaining nodes from other clusters are blocked.

We compared the following algorithms.

Random tree: The base algorithm without any heuristics, with no limit on the number of children for each node.

NearParent only: The base algorithm + NearParent. No Tree2List.

Tree2List only: The base algorithm + Tree2List. No NearParent.

NearParent + Tree2List: Use both Tree2List and NearParent.

Manual: Fix the transfer route that we consider will be the best, as follows; istbs connects to istsun via one inter-cluster edge. It is branched into three inside istsun. They go to kototoi, oxen, and marten. Inside clusters, there are no branches. The throughput should be close to 100Mbps / 3 = 33Mbps, determined by the three outgoing edges from istsun, which share a single 100Mbps NIC.

In Figure 4.3, the results are presented. Not surprisingly, "Manual" is the fastest. NearParent + Tree2List achieved an overhead of 50-100% to the manually tuned transfer and more than four times faster than the random tree.

Figure 4.4 shows that the number of inter-cluster edges and distribution time have a strong correlation.

This result confirms that reducing inter-cluster (and inter-subnet) edges strongly affects performance of replication among many nodes.

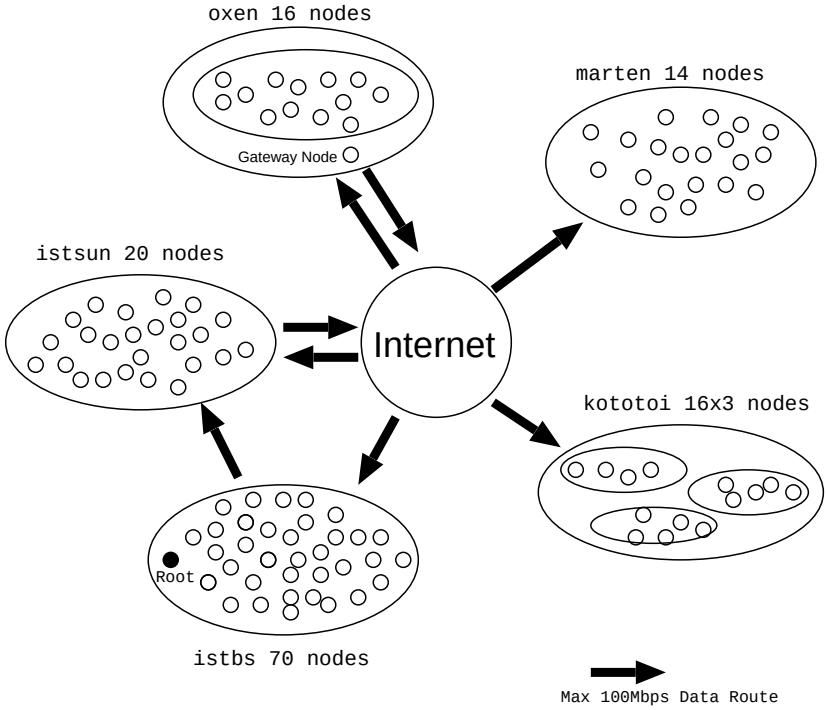

FIG. 4.2. *Condition of 7 clusters*

5. Related Work.

5.1. Minimum Spanning Tree Construction. MST construction is a commonly used technique for optimizing flows in networks. There have been a number of published algorithms and their applications [2, 6, 1]. It is compelling to model our problem by a general weighted graph, with the goal being a tree that has a small weight and a small number of branches.

We considered approaches along this line and then abandoned them for several reasons. First, from theoretical point of view, minimizing the two criteria at the same time is impossible for general weighted graphs, so we must make a difficult (and somewhat arbitrary) decision about how to trade one for the other. From the practical side, building an MST for general weighted graph in fault-tolerant and self-stabilizing manner is already complex to implement. Finally, typical real networks have a relatively simple structure we can (and should) exploit. That is, nodes close to each other in terms of physical proximity can logically connect to each other at some level and below. Therefore these nodes should be able to form a list entirely within the clique. We have shown this is in fact possible with a very simple hill-climbing with fault-tolerance and adaptiveness.

5.2. Application-Level Multicast and CDN. Our work is in spirit similar to a number of work on application-level multicast and content distribution networks (CDN). Our optimization criteria are different from them, particularly in that we try to reduce the number of branches.

ALMI [9] uses a centralized tree management scheme and makes MST for good performance. End System Multicast [7] takes both latency and bandwidth into account when making a tree of end-hosts. In [12], CAN [11] is used for the infrastructure of multicast. Bayeux [15] uses Tapestry [14] that is also content-addressable network. Overcast [8] is a multicasting system that achieves both small latencies and high throughput. The main application of these systems is multimedia streaming to widely distributed nodes. In such settings, it is important to bound latencies because the application may be an interactive multimedia application. Also in CDNs, the main criteria are latencies and traffic load balancing, rather than delivering as much bandwidth as possible. So researches about CDN such [10, 4, 3] mainly concern how to allocate replicas of contents, and how to redirect user requests to appropriate replicas. On the other hand, it is less important for such applications to squeeze the available bandwidth of local area networks, because there are typically a small number of participating nodes within each network. In contrast, our file replication does not have to optimize

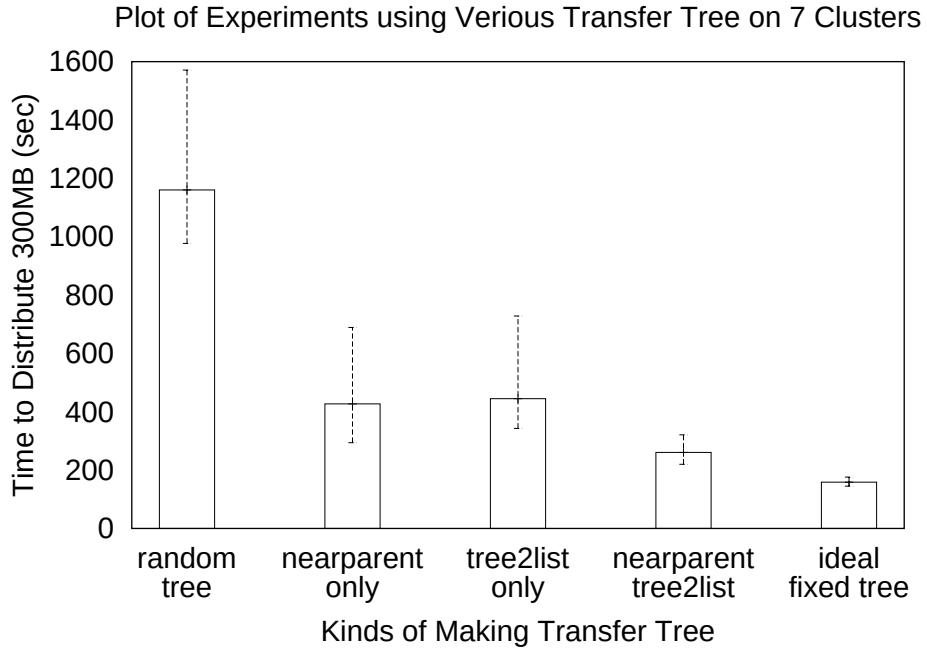

FIG. 4.3. *Performance on 7 clusters*

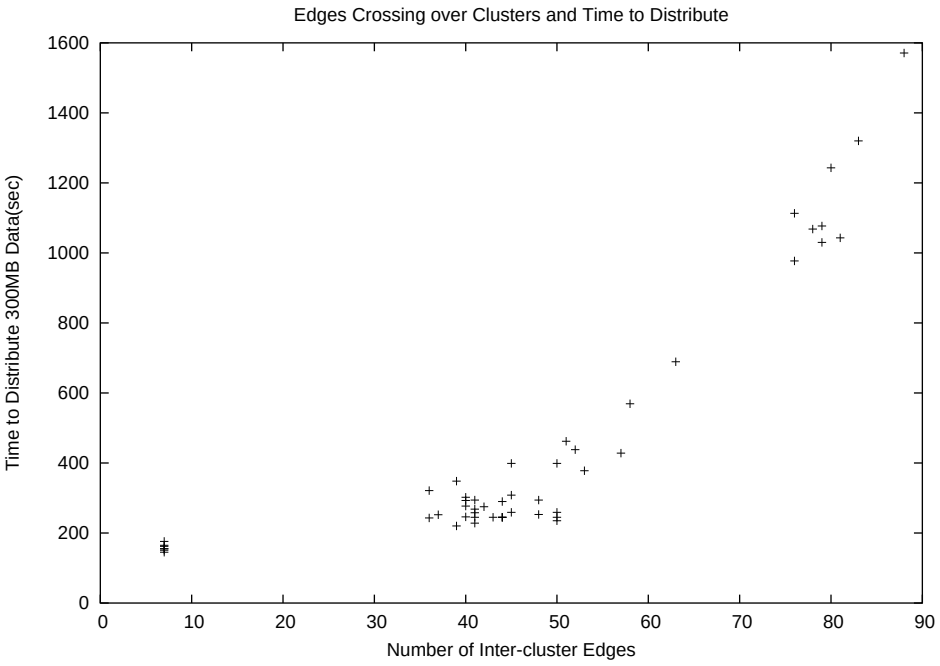

FIG. 4.4. *Correlation between number of inter-cluster edges and distribution time*

latencies aggressively, because the first priority is on the completion time of transferring large files. It is also very important to utilize LAN bandwidth as much as possible, as the typical usage will be to copy large files to many nodes in clusters. These differences lead them to different optimization criteria, with ours including a unique Tree2List heuristics.

6. Summary and Future Work. We have described a large file distribution algorithm that realizes scalability, adaptiveness, fault-tolerance, and efficient use of bandwidths. It is based on a simple distributed algorithm with simple local heuristics to optimize transfers. We formalized and proved the properties of our algorithm and argued that this gives a good result in practical settings. Our system will be useful for setting up a number of clusters and preparing wide-area distributed computations with a large data. Evaluations show that our implementation is effective in real environment consisting of over 150 nodes across seven clusters campus-wide.

Our current implementation of the protocol is not secure. Any malicious node can participate in the replication and breaks the integrity. To be a useful tool for distributed computing, we must use a suitable authentication when nodes connect to each other. While introducing secure authentications is possible, this may increase the cost of deploying such tools, whose very purpose will be to help maintain a large number of nodes easily. We must study how to maintain ease of installation and use of this tool while achieving a reasonable level of security.

REFERENCES

[1] Abhishek Agrawal and Henri Casanova. Clustering Hosts in P2P and Global Computing Platforms. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, pages 367–373, 2003.
[2] F. Bauer and A. Varma. Distributed Algorithms for Multicast Path Setup in Data Networks. Technical Report UCSC-CRL-95-10, University of California at Santa Cruz, August 1995.
[3] A. Biliris, C. Cranor, F. Douglis, M. Rabinovich, S. Sibal, O. Spatscheck, and W. Sturm. CDN brokering. In *Proceedings of WCW'01*, June 2001.
[4] Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97)*, Monterey, CA, 1997.
[5] CVS home. http://www.cvshome.org/.
[6] Lisa Higham and Zhiying Liang. Self-Stabilizing Minimum Spanning Tree Construction on Message-Passing Networks. In *Proceedings of the 15th Conf. on Distributed Computing, DISC, LNCS 2180*, pages 194–208, 2001.
[7] Yang hua Chu, Sanjay G. Rao, Srinivasan Seshan, and Hui Zhang. Enabling Conferencing Applications on the Internet Using an Overlay Multicast Architecture. In *ACM SIGCOMM 2001*, San Diago, CA, August 2001. ACM.
[8] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James W. O'Toole, Jr. Overcast: Reliable Multicasting with an Overlay Network. In *Proceedings of the Fourth Symposium on Operating System Design and Implementation (OSDI)*, pages 197–212, October 2000.
[9] Dimitris Pendarakis, Sherlia Shi, Dinesh Verma, and Marcel Waldvogel. ALMI: An Application Level Multicast Infrastructure. In *Proceedings of the 3rd USNIX Symposium on Internet Technologies and Systems (USITS '01)*, pages 49–60, San Francisco, CA, USA, March 2001.
[10] Lili Qiu, Venkata N. Padmanabhan, and Geoffrey M. Voelker. On the placement of web server replicas. In *INFOCOM*, pages 1587–1596, 2001.
[11] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM 2001)*, pages 161–172. ACM Press, August 2001.
[12] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Application-Level Multicast Using Content-Addressable Networks. *Lecture Notes in Computer Science*, 2233, 2001.
[13] Yasuhito Takamiya, Atsushi Manabe, and Satoshi Matsuoka. Lucie: A fast installation and administration tool for large-scaled clusters (in Japanese). In *SACSIS 2003*, pages 365–372, May 2003.
[14] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.
[15] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John D. Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-area Data Dissemination. In *Proceedings of the Eleventh International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 2001)*, June 2001.

Appendix A. Omitted Proofs. In this section we abbreviate is_closer to $\mathcal{C}$.

A.1. Lemma 3.1. Let V be the set of all nodes. We introduce an unknown x_{AB} for each $A, B \in V$. For each triple (A, B, C) such that $\mathcal{C}(A, B, C)$ is true, we generate a constraint $x_{AB} < x_{AC}$. We then unify x_{AB} and x_{BA} for all $A, B \in V$, replacing all occurrence of one with the other. We are going to show there are no loops of constraints $x_{AB} < x_{CD} < \cdots < x_{AB}$, thus the constraints are satisfiable. When we have proved this, we let $d(A, B) = x_{AB}$, for all $A, B \in V$.

To begin with, we show the following:

$$x_{AB} < \cdots < x_{YZ}$$
$$\Rightarrow \mathcal{C}(A, B, Z) \text{ or } \mathcal{C}(A, B, Y),$$

by induction on the length (the number of inequalities) of the lefthand side n.

1. $n = 1$:
   Observe we must have $A = Y$, $A = Z$, $B = Y$, or $B = Z$ since this constraint was generated from $\mathcal{C}$. When $A = Y$, $x_{AB} < x_{YZ} \Rightarrow x_{AB} < x_{AZ} \Rightarrow \mathcal{C}(A, B, Z)$. Other cases are similar.
2. Assume the claim holds up to $n - 1$ and now we have

$$x_{AB} < x_{CD} < \cdots < x_{YZ}$$

   of length n. By induction hypothesis, we either have:
   (a) $\mathcal{C}(C, D, Z)$, or
   (b) $\mathcal{C}(C, D, Y)$.
   By $x_{AB} < x_{CD}$, we either have:
   (i) $A = C$ and $\mathcal{C}(A, B, D)$,
   (ii) $A = D$ and $\mathcal{C}(A, B, C)$,
   (iii) $B = C$ and $\mathcal{C}(A, B, D)$, or
   (iv) $B = D$ and $\mathcal{C}(A, B, C)$.
   Since (a) and (b) are similar we only prove the case (a) by analyzing the four cases (i)–(iv).
   (i) $\mathcal{C}(A, B, D)$ and $\mathcal{C}(A, D, Z)$
   $\Rightarrow \mathcal{C}(A, B, Z)$
   (ii) $\mathcal{C}(A, B, C)$ and $\mathcal{C}(C, A, Z)$
   $\Rightarrow \mathcal{C}(A, B, C)$ and $\mathcal{C}(A, C, Z)$
   $\Rightarrow \mathcal{C}(A, B, Z)$.
   (iii) $\mathcal{C}(A, B, D)$ and $\mathcal{C}(B, D, Z)$
   $\Rightarrow \mathcal{C}(B, A, D)$ and $\mathcal{C}(B, D, Z)$
   $\Rightarrow \mathcal{C}(B, A, Z) \Rightarrow (A, B, Z)$.
   (iv) $\mathcal{C}(A, B, C)$ and $\mathcal{C}(C, B, Z)$
   $\Rightarrow \mathcal{C}(B, A, C)$ and $\mathcal{C}(B, C, Z)$
   $\Rightarrow \mathcal{C}(B, A, Z) \Rightarrow (A, B, Z)$.

Now we prove by contradiction there are no loops:

$$x_{AB} < \cdots < x_{YZ} < x_{AB}.$$

By the above induction, we either have:
(a) $\mathcal{C}(A, B, Z)$ or,
(b) $\mathcal{C}(A, B, Y)$.
By $x_{YZ} < x_{AB}$, we either have:
(i) $Y = A$ and $\mathcal{C}(A, Z, B)$,
(ii) $Y = B$ and $\mathcal{C}(B, Z, A)$,
(iii) $Z = A$ and $\mathcal{C}(A, Y, B)$, or
(iv) $Z = B$ and $\mathcal{C}(B, Y, A)$.
We see combining any of (a)–(b) and any of (i)–(iv) will lead to contradiction. We only prove case (a) since (b) is similar.
(i) $\mathcal{C}(A, B, Z)$ and $\mathcal{C}(A, Z, B)$
$\Rightarrow$ false.
(ii) $\mathcal{C}(A, B, Z)$ and $\mathcal{C}(B, Z, A)$
$\Rightarrow \mathcal{C}(B, A, Z)$ and $\mathcal{C}(B, Z, A)$
$\Rightarrow$ false.
(iii) $\mathcal{C}(A, B, Z)$ and $Z = A \Rightarrow$ false.
(iv) Same as (iii).

A.2. Lemma 3.2. Analyze the three cases, (i) $d(A, C) < d(A, B)$, (ii) $d(A, B) < d(A, C)$, and (iii) $d(A, B) = d(A, C)$. Prove each case by contradiction.
(i) Let us assume $d(A, C) < d(A, B) < d(B, C)$. Then,
$d(A, C) < d(A, B)$ and $d(A, B) < d(B, C)$
$\Rightarrow d(A, C) < d(A, B)$ and $d(B, A) < d(B, C)$
$\Rightarrow \mathcal{C}(A, C, B)$ and $\mathcal{C}(B, A, C)$
$\Rightarrow \mathcal{C}(A, C, B)$ and $\mathcal{C}(A, B, C)$
$\Rightarrow$ false.

(ii) Similar to (i).

(iii) Let us assume $d(A,B) = d(A,C) < d(B,C)$. Then,
$d(A,B) = d(A,C)$ and $d(A,C) < d(B,C)$
$\Rightarrow d(A,B) = d(A,C)$ and $d(C,A) < d(C,B)$
$\Rightarrow d(A,B) = d(A,C)$ and $\mathcal{C}(C,A,B)$
$\Rightarrow d(A,B) = d(A,C)$ and $\mathcal{C}(A,C,B)$
$\Rightarrow d(A,C) = d(A,B)$ and $d(A,C) < d(A,B)$
$\Rightarrow$ false.

Edited by: Wilson Rivera, Jaime Seguel.
Received: July 3, 2003.
Accepted: September 1, 2003.