18. Algorithmic Aspects of Overlay Networks

Danny Raz (Technion, Israel Institute of Technology)

18.1 Background and Motivation

This chapter studies basic algorithmic tasks over overlay networks. The main idea is to explore both the communication and the computation needed to perform a task. Unlike many of the work in this area (and other chapters of the book) we mainly consider in this chapter tasks related to information gathering or dissemination, as they can be used as building blocks in many overlay applications.

Overlay networks are logical networks that are created on top of the "real" physical network. Recently, various forms of such overlay networks have been deployed in the Internet. The most popular ones are file sharing Peer-to-Peer (P2P) networks and Content Distribution Networks (CDN). As well explained in other chapters of this book, in a Peer-to-Peer network, users voluntarily contributed their computers to form an overlay network that is then used mainly for file swapping. CDNs are overlay networks that are used to distribute Web context to distribution points of context delivery service providers, aiming at bringing content closer to the end users.

While the specific characterizations of these networks (and other overlay networks) vary, the main features are similar. In overlay networks we are dealing with two layers: a fast underlying network layer, and a much slower application layer. In this application layer, arbitrary complex computations (mostly related to data) are being done. In such an environment it is no longer true that the inter-node computations are fast with respect to the communication delay.

In order to study the algorithmic aspects of such networks one has to consider both the propagation delay through the network, and the computation delay at the overlay nodes. The well known distributed models [37] assume in many cases a global bound on the propagation delay and on the computation delay, in order to bound the overall time complexity, and therefore we need to adopt a different model in our case. The two-layer structure described above is very similar to the situation in an Active Network Node [587], and thus we can adopt the model from [507], based on a similar model for high speed networks [121].

In this model, the underlying networking layer handles packets with minimal delay and thus we assume a fixed constant C units of delay per packet, which includes the propagation delay along the link. In the upper layer, more complex computation is done and thus the time complexity depends on the

R. Steinmetz and K. Wehrle (Eds.): P2P Systems and Applications, LNCS 3485, pp. 289-321, 2005.
© Springer-Verlag Berlin Heidelberg 2005

exact algorithm and the size of the data. We denote this complexity by $P(k)$, where k is the size of the input to the computation. In this chapter we follow [507] and assume that $P(k) = P \times k$ for some constant P, since we at least need to copy all the data from the fast layer to the computation layer.

Recall that a link in the overlay network is a virtual link and it may be a long path in the underlying network. Figure 18.1 depicts such a network. The big circles represent overlay nodes and the wide dotted lines represent links between nodes in the overlay network. The translation of overlay links to physical links depends on the routing in the underlying (IP) network. For example a packet going from node C to node D may be routed through node E, even though E is not connected to C in the overlay network.

There are two different methods in which overlay networks can handle routing. In the first one, the overlay network takes care of routing, and thus packets are forwarded by the overlay layer to the appropriate virtual neighbor. For example, if node C needs to send a message to node G, it may route it through node B, thus the actual packet may travel more than once over several links of the underlying networks. The second routing method uses the underlying routing, i.e., if node C needs to send a message to node G, it will get its actual address (say IP address) and will send the message directly to this address. The actual performance of the algorithms depends, of course, on the routing methods, and we will carry out the analysis for each of the methods.

Now, under this model, the time complexity (i.e. the time it takes to complete the task) of data collection is no longer just a function of the length of the paths along which the data is collected. In fact, this time complexity also depends on the number of overlay network nodes in which the data is processed, and on the complexity of the processing algorithm itself.

In particular, it is no longer true that the popular solution of collecting data along a spanning tree as described in [37] is optimal. Consider for ex-

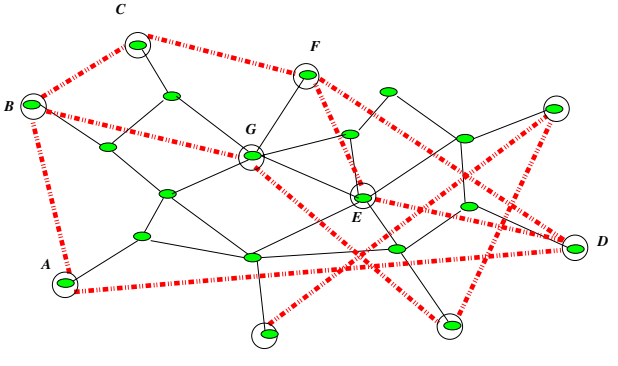


Fig. 18.1: An Example of an Overlay Network.

ample the tree described in Figure 18.2. Since the amount of information passed along each of the nodes on the long path is $O(n)$, the overall time complexity is $\Omega(Pn^2)$. Another naive solution for this problem is to iteratively query each node in the network, however, this solution will require that (for the same tree) n separate messages will arrive to the root, and an overall $\Omega(n^2)$ messages if the overlay network takes care of the routing. The goal is to develop algorithms that will have both a linear time and message complexity.

We start with a simpler problem, where data is collected along a given path in the overlay network. We study different simple algorithms for this case (similar to the one presented in [507] for the active networking case) and analyze their performance in the two overlay routing methods. Then we turn to the algorithm, called *collect-rec* in [507], that uses recursion to collect data. When the algorithm collects data from a path, it partitions the path into two segments, and runs recursively on each segment. The data collected from the second segment is sent to the first node in the path. The complexity analysis of this algorithm shows that the algorithm *time complexity* is $O(nP + nC)$ and the *message complexity* is $O(nlog(n))$ in the first routing method, as in the active networking case. If routing in the overlay network is done using the second method then the algorithm *time complexity* is $O(nP + nd_nC)$ and the *message complexity* can be bounded by $\min O(nd_nlog(n)), O(\bar{d}log(n))$, where d_n is the average distance (in underlying network hops) between overlay neighbors, and $\bar{d}$ is the average distance (in underlying network hops) between any two overlay nodes.

The more general problem is the problem of efficiently collecting data from a general overlay network. We are given a network with a specific node called *root*, where the number of nodes in the overlay network is n, its diameter is D, the number of logical links is m, and each node holds one unit of information. The *root* intends to collect the data from all the nodes in the overlay network. We assume that the network topology is arbitrary and no global routing information is available to the nodes. That is, a node in the overlay network may know the name (or IP address) of its logical neighbors, but no global information about the structure of the overlay network is known. Our aim is to develop an algorithm that solves the problem defined above, with minimal time and message complexity.

The need for such data collection depends on the specific network. For the popular Peer-to-Peer networks, one can think of obtaining a snapshot of the network at a given time, or finding the number of copies of a very popular file. For CDNs, obtaining usage statistics is always a need, and doing so with minimal delay and network overhead is very desirable. As shown before, the naive implementation of collecting data along a given spanning tree may perform badly. One possible approach is to extend Algorithm collect-rec([507]) from a path to a general graph. We explain this method, and generalize the data collection algorithm to an algorithm that collects an arbitrary amount

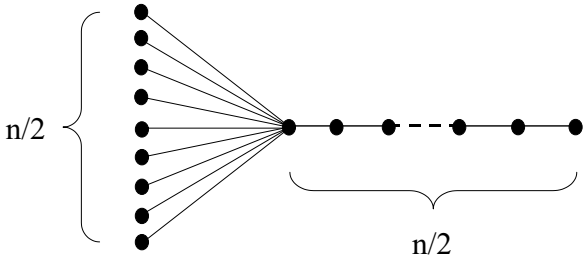


Fig. 18.2: Worst case broom.

of data from any link on a path. This last algorithm is then used as a building block in the more general algorithm that collects data in an almost optimal way from any given spanning tree.

For many overlay applications (such a CDN), assuming that the overlay network maintains a spanning tree is a very natural assumption as some information should be sent (or collected) from all the nodes. For other applications, like a Peer-to-Peer file sharing application, maintaining a global spanning tree may be too expensive since the amount of users that join (or leave) the Peer-to-Peer network each time unit is too big. Nevertheless, in all non-trivial proposals for a structural Peer-to-Peer network, maintaining such a tree requires at most a very small change to the existing infrastructure. If a spanning tree does not exist, one will have to create a spanning tree, and run the algorithm on top of it (assuming the amount of data collected is big enough). In order to create such a tree, one can use the well known algorithm, [37] in which every node that receives a message indicating the creation of the tree, sends this message to all its neighbors. The creation of such a tree in our model will take $O(CD + mP)$ time and $O(m)$ messages. This can be naturally included in the number assigning step of Algorithm collect-rec, resulting in a message complexity of $O(m + nlog(n))$ and a time complexity of $O(mP + nC)$.

The rest of the chapter is organized as follows. We start with the formal definition of the model, then in section 18.3.1 we describe algorithms that collect information from a path in the overlay network. In Section 18.4 we describe a data collecting algorithm that works on a single path but when the amount of data in each node is not fixed. Apart from being an interesting problem by itself, this algorithm presents the basic building block for the more general algorithm, weighted collect on trees, described in Section 18.5. This algorithm collects information from a tree that spans the overlay network. We deal with the creation of such a spanning tree in section 18.6 and with general functions that can be computed from the nodes' data in section 18.7.

Then, in section 18.8 we present a simulation study done to evaluate the performance of these algorithms in an Internet like setting.

18.2 Model Definition

Normally, the nodes of an overlay network are not interior nodes of the underlying IP network. However, in general they may be interior, and recent works such as [503] address "Topologically Aware" overlay networks. Thus, in order to be able to describe the system in a consistent way we assume that all nodes reside inside the network. In such a case, each internal node of an overlay network (see Fig. 18.3) consists of two parts: the *Fast Networking Layer* (FNL), and the *Application Layer* (AL). The FNL is an IP forwarding mechanism that in addition to transmitting the main data traffic, can filter packets destined to the AL from all packets passing through the node and redirect these packets to the AL. The AL provides means for executing the received programs.

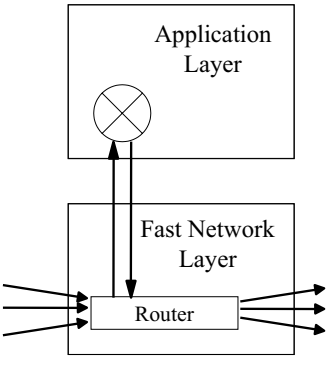


Fig. 18.3: A node structure.

This model is similar to the one introduced in [507] for active networks. It distinguishes between two types of delay: the delay of the message that only passes through the FNL module and the delay of the message that also triggers a computation in the AL. A message that passes only through the FNL suffers a constant delay. We bound this delay by the constant C; note that in practice the propagation time between neighbors (in the overlay network) may vary from 1Ms (in a LAN) to 80Ms (In a WAN). Since in most systems, messages are exchanged using the TCP protocol over links that have enough bandwidth (say more than 144Kbs), the propagation delay of messages is almost constant, which justifies this constant delay assumption.

A message that performs computations in the AL suffers an additional delay of kP time units, where P is a given constant, and k is a measure for the size of the input and output of the local computation. Note, that k is at least linear in the message size, since the message must be copied from the FNL to the AL. Our network in unsynchronized but throughout the chapter we do assume reliability, i.e., messages are never lost.

As explained before, we consider two different methods in which routing is done in the overlay layer. In the first one, the overlay network takes care of routing, and thus packets are forwarded by the overlay layer to the appropriate virtual neighbor. The second routing method uses the underlying routing, i.e., if a node needs to send a message to another overlay node it will get its actual address (say IP address) and will send the message directly to this address. The actual performance of the algorithms depends, of course, on the routing methods, and we will carry out the analysis for each of the methods. We use d_n to denote the average distance (in underlying network hops) between a node and its neighbors, and $\bar{d}$ to denote the average distance (in underlying network hops) between any two nodes in the network. Note that if the overlay network does not use any localization these distances may be equal. However, in many cases (see for example [503]) proximity in the underlying network plays a major part in the creation of the neighboring relation in the overlay network. In such cases d_n may be significantly smaller than $\bar{d}$. In the extreme case where the overlay network is exactly the underlying network, $d_n = 1$ and $\bar{d}$ is the average distance between nodes in the underlying network.

We consider two cost functions. The first one is the *time complexity* that describes the time it takes to accomplish the task. The second cost function we consider is the *message complexity* that counts the number of messages that pass between neighbors in the underlying network during the algorithm execution.

Throughout this chapter we say that a message passes between nodes through the *fast track* if it is not processed by any of the AL's on its way from the source to the destination. Otherwise we say that the message passes through the *slow track*. Note, that a fixed size message that passes through the *fast track* between two nodes of distance n in the overlay network, suffers an average delay of nd_nC time units in the first routing method and an average delay of $\bar{d}C$ in the second routing method. When the same message passes through the *slow track* it suffers an average delay of $nd_nC + nP$ time units, in both methods since the slow track forces the overlay network to send the message from one overlay node to another.

18.3 Gathering Information Along a Path

The problem can be stated as follows. A node v seeks to learn the ids of the nodes along the route (in the overlay network) from itself to another node u. Recall that in our model, v only knows the id of the next hop node along this route.

18.3.1 Basic Algorithms

In a naive implementation (naive), node v queries its next hop node for the id of the second hop node. Then it iteratively queries the nodes along the route until it reaches the one leading to the destination. This method resembles the way the `traceroute` program works, but it does not use the TTL field which is not part of the model. The delay of the naive algorithm is comprised of n activations of an AL level program plus the network delay. The average network delay in the first routing method is $2id_nC$ for $i = 1, 2, \ldots, n-1$ hops, which sums up to $O(n^2)d_nC$ time units. The message complexity in this case is given by $\sum_{i=1}^{n-1} 2id_n = O(n^2)d_n$. Using the second routing method, each message travels on the average $\bar{d}$ underlying network hops, and thus the average delay of the algorithm is $O(n(P + \bar{d}C))$, and the average message complexity is $2n\bar{d}$.

Next we describe two simple algorithms, collect-en-route and report-en-route, (presented in [507]) that improve the above solution to the route exploration problem, and analyze their performance using our model. Following this discussion we turn to more sophisticated solutions that achieve near optimal performances.

collect-en-route

In algorithm collect-en-route the source initiates a single packet that traverses the route and collects the host ids from each node. When the packet arrives at the destination node, it sends the data directly (using only the FNL) back to the source. The packet contains the source node id, the destination node id, and a list of ids it traversed. The source starts the algorithm by sending the message $MSG^*(s, d, \{s\})$ towards the destination, and outputs the list it receives from the destination.

Clearly, the message complexity of this algorithm for a node at distance n is $2d_nn$ in the first routing method, and $d_nn + \bar{d}$ using the second routing method.

The communication delay for this algorithm is $2nd_nC$ and $nd_nC + \bar{d}C$ for the two routing methods since exactly one message traverses the route in each direction. The execution delay at a node at distance i is iP since the message length is increased by one unit each hop. The total delay is given

```
collect-en-route
    1. for MSG*(s, d, list)
    2.     if i == d
    3.         send Report(list|i) to s
    4.     else
    5.         send MSG*(s, d, list|i) to d
```

Fig. 18.4: collect-en-route for an intermediate node i

by $2nd_nC + \sum_{i=1}^{n} iP = 2nd_nC + \frac{n(n+1)}{2}P$ for the first routing method, and by $nd_nC + \bar{d}C + \sum_{i=1}^{n} iP = nd_nC + \bar{d}C + \frac{n(n+1)}{2}P$ for the second routing method. Note that this algorithm is somewhat more sensitive to packet loss than the previous (and the following) one since no partial information is available at the source before the algorithm terminates. Furthermore, the time-out required to detect a message loss here is significantly larger than with the other algorithms presented here.

report-en-route

In algorithm report-en-route the source sends a request packet downstream the path. When a request packet arrives at a node, it sends the required information back to the source and forwards the request downstream to the next hop. This design minimizes the time of arrival of each part of the route information, while it compromises communication cost. The algorithm uses two message types: a forward going message, MSG^*, that contains the source node id, the destination node id, and a hop counter; and a backward going *Report*. The source starts the algorithm by sending the message $MSG^*(s, d, 0)$, each node increases the hop counter by one, forwards the message towards the destination, and sends a *Report* towards the destination with its id and the hop counter value. The source uses the hop count to order the list of nodes in its output.

```
report-en-route
    1. for MSG*(s, d, c)
    2.     send Report(id, c + 1) to s
    3.     if i ≠ d
    4.         send MSG*(s, d, c + 1) to d
```

Fig. 18.5: report-en-route for an intermediate node i

The message complexity using the first routing method is clearly $O(d_n n^2)$, and using the second routing method t becomes $O(n(d_n + \bar{d}))$. The communication delay for this algorithm using the first routing method is $2d_nnC$ since

exactly one message traverses the route in the forward direction until the destination, and this message is then sent back to the source. If we use the second routing method, the communication delay of the message sent from the destination back to the source is $C\bar{d}$. The execution delay in all the nodes is P since the message length is exactly one unit. The total delay is given then by $n(2Cd_n + P)$ for the first routing method, and $n(Cd_n + P) + C\bar{d}$ for the second routing method.

Algorithm collect-en-route features a linear message complexity with a quadratic delay, while algorithm report-en-route features a linear completion delay with a quadratic message complexity. Combining these two algorithms we can achieve tradeoffs between these two measures. In particular, if both measures are equally important we may want to minimize their sum.

Report-Every-l

An algorithm that enables us to optimize the two measures combined works as follows. First step is to obtain n, the length of the route between the two endpoints. This may be known from previous executions of the algorithm or can be obtained by an algorithm which is linear both in time and message complexity (see next sub-section). Next we send a fixed size message to initiate collect-en-route in n/l segments each of length l. This can be done using a counter that is initialized to l at the beginning of every segment, and decreased by one at every intermediate node. Thus, for the first routing method, the execution of collect-en-route in the segment i starts after at most $(i-1)(Cd_n + P)$ time units, and the overall time complexity is $(n-l)(Cd_n + P) + \sum_{i=1}^{l}(Cd_n + iP) = O(nd_nC + (n + l^2)P)$. The message complexity is $O(nd_n) + \sum_{i=1}^{nd_n/l}(l + il) = O(\frac{n^2}{l}d_n)$. Choosing $l = \sqrt{n}$ results in a linear time complexity while using $O(n\sqrt{n})$ messages. The requirement to balance the two measures (up to a constant factor of P), translates to $l^2 = \frac{n^2}{l}$ which gives $l = n^{2/3}$. For this l value both time and message complexity are $O(n^{4/3})$.

18.3.2 collect-rec

A different approach, however, is needed in order to reduce both the time and message complexity. The following algorithm, collect-rec, achieves an almost linear time and linear message complexity. The main idea is to partition the path between the source and the destination into two segments, to run the algorithm recursively on each segment, and then to send the information about the second segment route from the partition point to the source via the FNL track. In order to do so on the segment (i, j), in each recursive step one needs to find the id of the partition point, k, and to notify this node, k,

that it has to both perform the algorithm on the segment (k, j) and report to i. In addition, i has to know that it is collecting data only until this partition point, k, and it should get the rest of the information via the fast track. The partition can be done, naively, in two passes. First we find the segment length. Then sending the segment length and a counter in the slow track allows k to identify itself as the partition node.

The idea behind the algorithm, as described above is very simple. However, the detailed implementation is somewhat complex. A pseudo-code implementation of this algorithm is given below.

```
main(i, d, l)
1.  if l = 0
2.      return({val_i})
3.  send Reach^1_port(i, d, ⌊l/2⌋, ⌈l/2⌉) to d
4.  L_1 ← main(i, d, ⌊l/2⌋)
5.  L_2 ← receive_port()
6.  return(L_1|L_2)
```

Fig. 18.6: Function main for some node i.[1]

```
1. For Reach(s, d, count, l)
2.   if count > 0
3.       send Reach(s, d, count − 1, l)
4.   else
5.       L ← main(i, d, l)
6.       send Report(L) to s
```

Fig. 18.7: Reaction of collect-rec for receipt of message Reach()

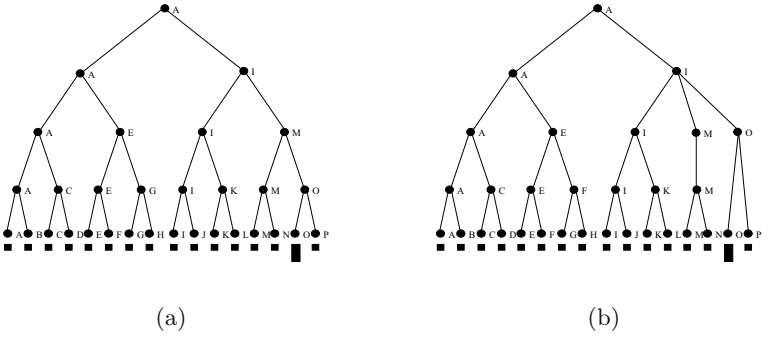
```
collect(d)
1.  l ← getlength(s, d)
2.  L ← main(s, d, l)
3.  return(L)
```

Fig. 18.8: Algorithm collect-rec for a source node, s.

[1] Subindexing with *port* is to indicate a possible implementation where the port number is used to deliver incoming messages to the correct recursive instantiation.

As explained before, finding the id of the partition point k, and sending the information to this node is the most difficult technical part of the implementation. This is done by sending the Reach() message with a counter that reaches 0 at node k. The id of the node who performed the partition, s, is part of the packet data, and thus once the information regarding the segment is available at node k it can send it directly via the fast track, to s.

The implementation of $getlength(s, d)$, which finds the hop length of the route between s and d, is similar to collect-en-route. The only difference is that the source sends a counter initialized to zero as the third parameter instead of an empty list. Intermediate nodes increase the counter by one instead of concatenating the next hop. Using the first routing method, this requires an average message complexity of $2d_n n$, and time complexity of $(n+1)P+2nCd_n$. If the overlay network uses the second routing method then the average message complexity is $d_n n + \bar{d}$, and the average time complexity is $(n+1)P + C(nd_n + \bar{d})$.



(a) (b)

Fig. 18.9: Example for the logical tree for $n = 16$

In order to analyze the time and message complexity of the algorithm, we need a better understanding of the properties of collect-rec algorithm. Consider the logical tree that is built by collect-rec algorithm (see Figure 18.9.a). This logical tree is built above the logical structure of the overlay network, and it defines the partition of the path into *segments* such that node i is responsible for collecting the information from segment $[i, i + segmentsize_i]$. The tree represents the hierarchy among the segments in the recursive execution of the algorithm. Note, that the same node can appear several times continuously in the logical tree but all the appearances must be on the path from its leaf appearance to the root. The main properties of this logical tree in our case (when each node has one unit of data) are:

1. The number of nodes with height[2] h is bounded by $n/2^h$.
2. At level l, the amount of data that a node sends to its father is smaller than 2^l and greater than or equal to 2^{l-1}. This data is copied at most $log(n) - l$ times during the algorithm run.
3. The distance on the path from a node with height l (a leaf has height 0) to its father in the logical tree is bounded by 2^l.

Using the above properties, one can conclude that the time complexity for a segment of length n is bounded by the time complexity of a segment of length $n/2$ plus a processing time of $n/2$ data units plus sending the Reach() message in the slow track along a (logical) path of length $n/2$, plus sending the data via the fast track back along a path of length $n/2$. All together we get the following recursive formula for the first routing method,

$$TC(n) \leq TC(n/2) + Pn/2 + (P + Cd_n)n/2 + Cd_n n/2,$$

and for the message complexity

$$MC(n) \leq 2MC(n/2) + nd_n.$$

For the second routing method we get the following recursive formulae,

$$TC(n) \leq TC(n/2) + Pn/2 + (P + Cd_n n/2 + C\bar{d}),$$

and for the message complexity

$$MC(n) \leq 2MC(n/2) + d_n n/2 + \bar{d}.$$

By solving these equations and adding the time complexity of $getlength(s, d)$, we can prove the following theorem.

Theorem 18.3.1. *Algorithm collect-rec solves the route detection problem with time complexity of $O(n(P + Cd_n))$, and message complexity of $O(nd_n \log n)$ for the first routing method, and time complexity of $O(n(P + Cd_n) + C\bar{d} \log n)$, and message complexity of $O(n(\bar{d} + d_n + \log n))$ for the second routing method.*

Tables 18.1 and 18.2 summarize the time and message complexity of the algorithms we described for collecting uniform information along a logical path for the two routing methods.

18.4 weighted collect-rec Algorithm

In this section we consider the problem of collecting data along a path where each node has an arbitrary amount of data. If we simply collect the information along the path, the time complexity may be quadratic if all the data is

[2] The height of a leaf in the logical tree is 0, and the height of the root is $\log n$.

algorithm name	time complexity	message complexity
naive	$O(nP + d_n n^2 C)$	$O(d_n n^2)$
collect-en-route	$O(n^2 P + d_n nC)$	$O(d_n n)$
report-en-route	$O(nP + d_n nC)$	$O(d_n n^2)$
report-every-l	$O((n + l^2)P + d_n nC)$	$O(d_n n^2/l)$
collect-rec	$O(nP + d_n nC)$	$O(d_n n \log n)$

Table 18.1: Summary of route exploration algorithms - first routing method.

algorithm name	time complexity	message complexity
naive	$O(nP + \bar{d}nC)$	$O(\bar{d}n)$
collect-en-route	$O(n^2 P + C(d_n n + \bar{d})$	$O(d_n n + \bar{d})$
report-en-route	$O(n(Cd_n + P) + C\bar{d})$	$O(n(d_n + \bar{d}))$
report-every-l	$O((n + l^2)P + C(d_n n + \bar{d})$	$O(d_n n + \bar{d}n/l)$
collect-rec	$O(nP + C(d_n n + \log n\bar{d}))$	$O(d_n n \log n + n\bar{d})$

Table 18.2: Summary of route exploration algorithms - second routing method.

concentrated at the end of the path. collect-rec algorithm is also not optimal for this problem, since when all the data is concentrated in the last node the data is copied $log(n)$ times during its transmission to the first node and thus the *execution complexity* of collect-rec will be $O(nlog(n)P)$. Thus, we developed a new algorithm that improves collect-rec and solves this problem more efficiently.

Our problem can be formally defined as follows: We are given an overlay network with two designated nodes i and j. Node i intends to collect data from all the nodes along the path from itself to j, where the amount of data in each node may vary. Let the length of the path be n and the total amount of data in all the nodes along the path be $\bar{n}$. We assume that no global routing information is available to the node, and thus every node knows only the *id*s of its neighbors. Our goal is to develop an algorithm that solves the described problem while minimizing the *message* and *time complexity*.

18.4.1 Algorithm Description

Our algorithm, called algorithm weighted collect-rec, follows the steps of the collect-rec algorithm. In order to collect the data along the path from node i to node j the algorithm partitions the path into two segments, runs itself on each segment recursively and then sends data from the second segment to i via the Fast Network Layer. However, in our case we cannot assume that the amount of data in each segment is proportional to the segment size since nodes may hold an arbitrary amount of data. Thus we need to adjust the destination of each node's data according to the amount of data that the node sends, and the total amount of data in the system.

Consider the logical tree properties of algorithm collect-rec described in the previous section. Item 2 is not valid for this logical tree if the data is distributed arbitrarily along the path. In the example on Figure 18.9.a node O has 4 units of data while each other node has one unit of data. Since O has extra data, the amount of data processed by node M is not bounded by 2^2. In this case algorithm weighted collect-rec adjusts the logical tree in such a way that O sends its data directly to node I (see Figure 18.9.b). Let $\bar{n}$ be the total amount of data in the tree; we assume for simplicity that $\bar{n} \geq n$ and term the value $\bar{n}/n$ the *amount ratio*. The adjusted tree has the following properties:

1. The number of nodes with height h is bounded by $O(n/2^h)$.
2. If the amount of data that a node sends to its father is greater than $\bar{n}2^l/n$ and smaller than or equal to $\bar{n}2^{l-1}/n$, then the data is copied at most $log(n) - l$ times during the algorithm run.
3. If the amount of data that a node sends to its father is greater than $\bar{n}2^l/n$ and smaller than or equal to $\bar{n}2^{l+1}/n$, then the distance on the path from the node to its father in the logical tree is bounded by $O(2^{log(n)-l})$.

These properties allow the weighted collect-rec algorithm to reach good complexity. weighted collect-rec algorithm uses two parameters to determine the distance of the data transmission. These parameters are the node's position in the basic logical tree, and the amount of data that the nodes in its logical subtree have.

18.4.2 Detailed Algorithm Description

The algorithm consists of three phases. During the first phase the algorithm computes the path parameters, i.e., the path length and the total amount of information along the path, assigns *length level* to each node and defines the destination for each node according to the *basic logical tree*, i.e. a tree that does not consider the amount of data in each node. The purposes of the second phase are to adjust the length of transporting data when required,

and to allow each node to know the amount of data it should receive during the last phase and the id of the parent node in the *adjusted logical tree*. In the third phase, all the nodes send their own data and the data they received from other nodes to their parent in the adjusted logical tree. At the end of this phase, the root (the first node in the path) receives all the data, and the algorithm accomplishes its task.

Phase 1

In this phase the algorithm builds the basic logical tree (of collect-rec algorithm) and computes the *amount ratio* which indicates the total amount of data in the path. Later, in the next phase, this information is used to modify the logical tree as needed. Recall that the logical tree defines the partition of the path into *segments* such that node i is responsible for collecting the information from segment $[i, i + segmentsize_i]$. The same idea is deployed in the weighted collect-rec algorithm. The weighted collect-rec algorithm builds a logical tree (see for example Figure 18.9.a), in which links indicate the responsibility of nodes for segments.

At the beginning of the algorithm the *root* sends a message with two counters towards the last node in the path. The counters are the *length* and *total data amount* of the path. The message passes through the Application Layer until it reaches the last node in the path. Each node that is traversed by this message, increments the length and adds its local data amount to the *total data amount*. The last node sends the message to the *root* through the Fast Network Layer.

When the *root* receives the message containing the path parameters, it initiates a recursive partition process. During this process each node i sends partition messages to nodes with numbers $(i + \lfloor segmentsize_i/2^k \rfloor)$, where k changes from 0 to i's *length level*-1. The messages are delivered through the Application Layer. Each such message, destined to node i, contains the address of its sender (i.e., i's parent in the logical tree who is also called i's destination), i's *length level* that describes the length of i's segment, and the *amount ratio*. Partition messages are sent via the Application Layer using a counter in the same way this process is done in algorithm collect-rec. Upon receiving the partition message each node stores all received parameters, it sends a hello message containing its id to its parent in the logical tree, and initiates a partition process on its segment. At the end of this phase, each node knows the ids of its parent and of its children in the basic logical tree. Note, that the *root*'s *length level* is $log(n)$ and the size of its segment is n.

Phase 2

As mentioned above, the goal of the second phase is to compute the *adjusted logical tree*. This includes determining the ids of the parents and children in

the new tree, and computing the amount of data that will be collected. This is done by collecting information regarding the amount of data in each segment, and disseminating this information along the hierarchy toward the root. This can be viewed as running algorithm collect-rec when the data collected is the amount of data at each node, and the length of its segment. This information is sufficient in order for each node to determine which of the nodes in its subtree are its children in the adjusted tree, and how much information each one of them sends.

This phase starts with a message *Comput-Adjusted-Tree (n,$\bar{n}$,idList)*, sent by the root using the fast track along the basic logical tree. Upon receiving this message, each node in the tree stores locally the idList that contains the ids of all the logical nodes from the root to it, adds its id to the idList, and forwards the message to the children in its basic logical tree. When a leaf gets this message, it computes the id of its destination according to the idList, and the amount of data it has according to the following rule. If the amount of data is greater than $\bar{n}2^l/n$ and smaller than or equal to $\bar{n}2^{l-1}/n$, then the destination is the $(log(n) - l)$'s element in the idList. It then sends an *AdjustTree(DataList)* message to its parent in the basic logical tree. Upon receiving an AdjustTree(DataList) message from all children, each intermediate node computes the amount of data it should receive, the ids of its children in the adjusted tree, and by adding its own data amount, the amount of data it needs to send up the tree. Using this data and the rule specified above, the node can use the idList it stored to compute the id of its adjusted destination (i.e. its parent in the adjusted tree). It then appends all lists it received from its children, adds its own amount and dest-id, and sends an Adjust() message to its parent in the basic logical tree.

This phase ends when the root gets an Adjust() message from all its children. At this point each node in the tree knows how much information it should get, and the ids of the nodes that will send it information, and also the id of its destination, i.e., its parent in the adjusted logical tree.

Phase 3

In this phase, data is sent along the links of the adjusted logical tree (as described in Figure 18.9). Each node sends data to its destination. The nodes, that receive data from other nodes, must wait until receiving all the data and only then they transmit this received data together with their local data to their *destination*. This is done through the *fast track*. At the end of the phase the *root* receives all the data of the path and the algorithm accomplishes its task.

```
weighted collect-rec-Phase2-3(i)
1.  if received ComputeAdjustTree(n, n̄,idList)
2.      if(segmentsize_i > 1)
3.          idList = id | idList
4.          send ComputeAdjustTree(n, n̄,idList) to Child_L
5.          send ComputeAdjustTree(n, n̄,idList) to Child_R
6.      else
7.          dest = parent
8.          if (local data amount >0 )
9.              l= log n + 1 - ⌊log(n/n̄· local data amount)⌋
10.             dest = idList_l
11.         DataList = (i,local data amount,dest)
12.         send AdjustTree(DataList) to perent
13.         send report(Data) to dest
14. i=0
15. while (i < number − of − children)
16.     if received AdjustTree(DataList)
17.     i++
18.     List = List | DataList
19.
20. dataAmount = processAdjustTree(List)
21. dataAmount+=local data amount
22. dest = parent
23. if (local data amount >0 )
24.     l= log n + 1 - ⌊log(n/n̄· local data amount)⌋
25.     dest = idList_l l=⌊log(n̄/n· dataAmount)⌋
26. DataList = (i,dataAmount,dest) | DataList
27. send AdjustTree(DataList) to perent
28. collectedAmount = dataAmount - local data amount
29. L ← collectData(collectedAmount)
30. L ← L|(localdata)
31. send Report(L) to dest
```

Fig. 18.10: Function main (phases 2 and 3) for node i.

18.4.3 Complexity Analysis of weighted collect-rec Algorithm

As explained above, the first and second phases of algorithm weighted collect-rec can be viewed as an execution of algorithm collect-rec, where the data collected is the triple $(i, amount(i), dest(i))$ and not the real data of each node. The only change here is that we added at the beginning of Phase 2 the message *AdjustTree(idList)* which goes down the logical tree. However, since the amount of data in the idList is bounded by the tree hight ($\log n$), the complexity of these phases is the same as the complexity of collect-rec, which can be formally stated as follows.

Lemma 18.4.1. *The time complexity of Phase 1 and Phase 2 is* $O(n(P + Cd_n))$*, and the message complexity is* $O(n(d_n + \log n))$ *for the first routing*

```
collectData(n)
    1.  L ← empty
    2.  while( n > 0)
    3.      if received Report(data)
    4.          L ← L|data
    5.          n− = sizeof(data)
    6.  return L
```

Fig. 18.11: Function collectData().

```
processAdjustTree(List)
    1.  amount = 0
    2.  for each entry in List
    3.      if( entry.dest == i) amount+=entry.amount
    4.  return amount
```

Fig. 18.12: Function processAdjustMessage.

method, and time complexity is $O(n(P + Cd_n) + C\bar{d}\log n)$*, and the message complexity is* $O(n(\bar{d} + d_n + \log n))$ *for the second routing method.*

The more difficult part is to analyze the complexity of the third phase in which data is actually sent along the logical links of the adjusted tree. We need to prove the following lemma.

Lemma 18.4.2. *The time complexity of Phase 3 is* $O(n(P + Cd_n))$*, and the message complexity of* $O(n(d_n + \log n))$ *for the first routing method, and time complexity is* $O(n(P + Cd_n) + C\bar{d}\log n)$*, and the message complexity is* $O(n(\bar{d} + d_n + \log n))$ *for the second routing method.*

Proof. When the algorithm collects data all the messages are sent towards the first node in the path, therefore the *communication delay* of phase 3 is $O(Cd_n n)$ using the first routing method and $O(C\bar{d}\log n)$ using the second routing method.

Define the *length level* of a node to be l if its *segment* size is smaller than 2^{l+1} but bigger than or equal to 2^l. A node has an *amount level* l if the amount of data it sends (this includes its own data and the data received from the other nodes during the algorithm execution) is greater than or equal to $(\bar{n}/n)2^l$ and smaller than $(\bar{n}/n)2^{l+1}$. Finally we define the node's *level* to be the maximum between its *length level* and its *amount level*.

According to the algorithm every node sends its data to a node with a higher *level*. The *amount level* of the node determines the *execution delay* in the node. The *execution delay* in the node with *amount level* l_a equals at worst to the amount of data it might process without increasing its *amount level*, hence it equals to $\bar{n}/n(2^{l_a-1})$. There are $log(n)$ possible *amount levels* and processing in the nodes with the same *level* is done simultaneously, hence the *execution delay* of Phase 3 is:

$$\sum_{l=0}^{log(n)} \frac{\bar{n}}{n}(2^l) \leq O(\bar{n}).$$

Therefore the *time complexity* of Phase 3 is $O(n(P+Cd_n))$ for the first routing method, and $O(n(P + Cd_n) + C\bar{d}\log n)$ for the second routing method.

During the data collection phase each node sends one message with data to its destination. The distance (in terms of overlay hops) between the node and its destination is bounded by 2^l where l is the node's *level*. The number of nodes with *level* l is bounded by $n/2^{l-1}$ since there are at most $n/2^l$ nodes with amount level l and at most $n/2^l$ nodes with length level l. Hence, when using the first routing method, the total number of messages passing during the third phase is bounded by

$$\sum_{l=0}^{log(n)} \frac{nd_n}{2^{l-1}}(2^l) \leq O(nlog(n)).$$

If we use the second routing method, the average message complexity is simply $\bar{d}\log n$.

Combining all three phases together we can prove the following theorem.

Theorem 18.4.1. *The time complexity of the algorithm* weighted collect-rec *is* $O(n(P + Cd_n))$*, and the message complexity is* $O(n(d_n + \log n))$ *for the first routing method, and the time complexity is* $O(n(P + Cd_n) + C\bar{d}\log n)$*, and the message complexity is* $O(n(\bar{d} + d_n + \log n))$ *for the second routing method.*

18.5 Gathering Information from a Tree

In this section we deal with a more general problem where we need to collect information from a general graph, and not from a specific path. First, we assume the existence of a spanning tree rooted at the root, and we want to collect information from all leaves of this tree along the paths to the root. As shown in the introduction to this chapter, the naive solution of collecting data along a spanning tree as described in [37] is not optimal in our model.

One can show that in the first routing method, it is impossible to gather all information with a time complexity lower than $\Omega(DC + nP)$, where D is the diameter of the overlay network. This is true because a message cannot arrive at the most remote element faster than DC time units, and the algorithm must spend at least P time units to copy the message from every element in the network. The message complexity cannot be lower than $\Omega(n)$, since every element in the network must send at least one message, thus the root must process at least n units of data. Moreover, if no global structure such as a spanning tree is available, the message complexity is bounded by $O(m)$,

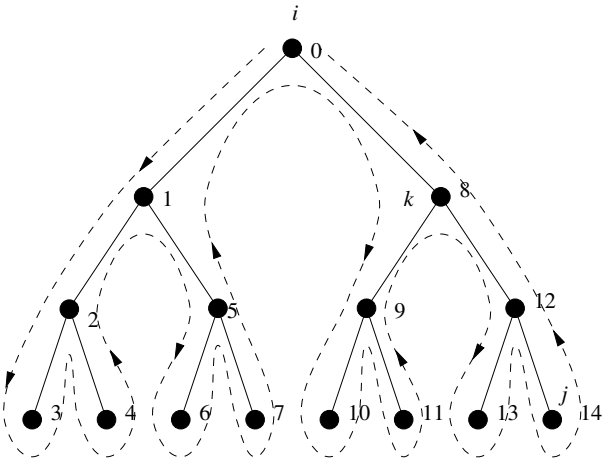


Fig. 18.13: A simple extension of collect-rec.

where m is the number of logical links in the overlay network, since every link in the graph must be tested in order to insure that we cover all nodes.

A first step towards developing algorithms that will have both a linear time and message complexity is to modify collect-rec to work on trees. This can be done (see Figure 18.13) by assigning a number to every node in the tree according to a pre-order visiting starting at the root. We now consider the path that goes from node 1 (the root) to node n, according to this order. Note that node i and node $i+1$ may not be neighbors in the overlay network (for example nodes 11 and 12 in Figure 18.13), and therefore this path is not a simple path. However, the total length of the path in terms of overlay nodes is bounded by $2n$.

Creating such a path requires assigning numbers to the nodes in the tree according to the pre-order visiting order, and allowing nodes in the tree to route messages according to this node number. For this, each node should know its own number, and the number range of each of its subtrees. This can be easily done by a bottom up pass on the tree that collects the sizes of the subtrees, followed by a top down pass on the tree in which every node assigns ranges to each of its subtrees. Since the size of the messages is constant, the time complexity of this process can be bounded by $O(Cd_nD + Pn)$, and the message complexity is $O(n)$.

After this phase we can run Algorithm collect-rec on this path and obtain the message and time complexity of collect-rec. All together we get the following theorem.

Theorem 18.5.1. *One can collect data from any given spanning tree with time complexity of $O(n(P + Cd_n))$, and message complexity of $O(n(d_n +$*

$\log n))$ *for the first routing method, and time complexity of* $O(n(P + Cd_n) + C\bar{d}\log n)$*, and message complexity of* $O(n(\bar{d} + d_n + \log n))$ *for the second routing method.*

However, in practice n might be very big (100,000 nodes or more in a typical Peer-to-Peer network) while the diameter of the network (and thus the height of the spanning tree) is much smaller (typically 10). It is thus both practically important and theoretically interesting to reduce the complexity of the data collecting algorithm in this model.

In the rest of this section we describe an algorithm, called weighted collect on trees, for the general data collection problem. As indicated before, we assume the existence of a spanning tree rooted at the root, and we want to collect information from all leaves of this tree along the paths to the root. Our algorithm follows the ideas presented in the previous section for the weighted collect-rec algorithm. We start with the given spanning tree (as we started with he basic logical tree in algorithm weighted collect-rec, and we modify it by assigning new destinations to some nodes. This is done in a way that data is always sent up the tree towards the root, balancing between the amount of data that a node sends and the length of path along which the data is transported.

In order to do so, we assign what we call a *length level* to each node. These length level values should have the following two properties: The number of the nodes with *length level* l must be greater than or equal to the total number of the nodes with *length level* greater than l, and the distance between a node with *length level* l and the nearest node towards the *root* with *length level* greater than l must be bounded by $O(2^l)$. Finding an appropriate assignment that has the desired properties is not easy. As will be proven later, the following algorithm assigns length level to the nodes of the tree with the desired properties. All leaves have *length level* 0. The *length level* of each internal node is the minimum between the maximum of all the *length level*s in the node's sub-tree plus 1, and the position of the first 1 in the binary representation of the distance of the node from the *root*. Once *length level*s are assigned, each node should send its data to the first node, on the way to the root in the tree, that has a *length level* that is bigger that the node's *length level*.

18.5.1 Detailed Algorithm Description

Phase 1

As explained above, the goals of the first phase are to assign a *length level* to each node, to find the destination of each node, and to compute the number of the adjust messages that each node needs to receive during the second phase.

Definition 18.5.1. *Let* $length_{bin}(i)$ *be the binary representation of the distance between the* root *and node* i*.*

Definition 18.5.2. *Let* $\eta(i)$ *be the position of the least significant bit that equals to* 1 *in* $length_{bin}(i)$*.*

The algorithm assigns *length level* 0 to all leaves. The *length levels* of all other nodes are defined by the formula $min\{\eta(i), l_m(i) + 1\}$ (where $l_m(i)$ is the maximal *length level* in the sub-tree of i excluding i).

In order to find the distance from each node to the *root*, the *root* disseminates among its children a message with one counter that is equal to 1. Each node that receives such a message stores the value of the counter as the distance to the *root*, increments the counter, and disseminates the message to its children.

Then, the algorithm computes the maximal *length level* in the sub-tree of each node and assigns the *length level* to each node. During this computation each node receives one message from each of its children. This message contains the maximal *length level* in the child sub-tree. Upon receiving this message the node computes both its *length level* and the maximal *length level* in its sub-tree (considering its own level as well), and sends the message with the computed value to its parent. This computation is done bottom up, and it starts in the leaves immediately after receiving the message with the distance to the *root*.

Once a node knows its *length level* it starts searching for its destination. this is done by creating a request that contains the node's *length level* and sending it towards the *root* via the Application Layer. Each node that receives such a request stores the address of the child who sent it. Then, the node forwards the request to its parent iff the node's *length level* is smaller than or equal to the received *length level* and the node did not send a request before with the same *length level*.

If the node's *length level* is bigger than the requested *length level*, it sends a reply with its address and *length level* to the appropriate child. When a node receives a reply, it disseminates the reply to all children who had sent the appropriate requests before, and stores the address from the reply if this is also its own destination.

In order to move on to the last task of the phase, it is important that each node will know that it has received all messages from its subtree. This is done by sending notification messages up the tree. Each leaf sends such a notification immediately after sending its request. Other nodes send their notification when they receive notification from all of their children and finished processing all the received requests. Note, that a node delays the dissemination of replies to its children until it sends a notification towards the *root*. Only after the node notifies its father, it is guaranteed that it knows about all its children who are waiting for the delayed replies.

The last task of the phase is to figure out the number of *length sources* of each node. Recall, that node j is the *length source* of node i if j received i's

address as a destination address during the first phase. Denote by $V_l(i)$ the set of the *length sources* of i with the *length level* l. The number of *length sources* equals the sum of $V_l(i)$, for all l smaller than i's *length level*. Next we describe how the algorithm computes $V_l(i)$ for a specific l. Consider the virtual tree that contains all the paths through which the replies with i's address passed. When replies reach their destinations this virtual tree is well-defined, since each node has the addresses of the children that received these messages, and the node will not receive new requests since its children finished sending requests. The root of this sub-tree is i. Note, that two such virtual trees, that are built from the paths passed by the replies with the same *length level*, have no common edges if their roots are distinct. The algorithm then computes the sum of the values of all the virtual tree nodes in a bottom up way, where each node adds the value 1 iff the node's *length level* is l.

Phase 2 and Phase 3

Once the destination of each node is known, the algorithm proceeds exactly as in algorithm weighted collect-rec since the balancing between the amount of data and the *length level* is exactly the same, and sending the data is of course the same.

18.5.2 Complexity Analysis of weighted collect on trees Algorithm

For simplicity, we do not explicitly state the pseudo code that implements the algorithm. The main difference between algorithm weighted collect on trees and algorithm weighted collect-rec is in the first phase and the definition of the *length level*s. Once the logical tree on top of the overlay tree is created, the algorithm and the analysis is very similar to the one in algorithm weighted collect-rec. We begin with two definitions.

Definition 18.5.3. *Let V_l be the group of nodes whose* length level *assigned by the algorithm is l. Denote by n_x the size of V_x.*

Lemma 18.5.1. *The size of V_l is equals to or greater than the sum of sizes of V_i, for all $i > l$, i.e. $n_l \geq \sum_{i=l+1}^{max\ length\ level} n_i$.*

Proof. Consider the group $V_{>l} = \bigcup V_i$ such that $i > (l+1)$. Denote by v_i a node from $V_{>l}$, and by $M(v_i)$ the nearest to v_i node in its sub-tree with *length level* l. Such a node always exists, because under each node with *length level* x there are nodes with all *length levels* less than x. If there are two or more such nodes we can arbitrarily choose one of them.

Consider two distinct nodes v_1, v_2 from $V_{>l}$. There are two cases. The first case is when none of these two nodes is in the sub-tree of the second node. The second case is when one node is in the sub-tree of the second node.

$M(v_i)$ is always in the sub-tree of v_i. Thus, in the first case $M(v_1)$ and $M(v_2)$ are distinct nodes. Consider now the second case. Assume, that v_2 is in the sub-tree of v_1. $M(v_2)$ is in the sub-tree of v_2. Since the *length levels* of the considered nodes is greater than l, both $\eta(v_1)$ and $\eta(v_1)$ are greater than l. Hence, the path from v_2 to v_1 contains at least one node with *length level* equals to l. Henceforth, $M(v_1)$ and $M(v_2)$ are distinct nodes.

In both cases we proved that $M(v_1) \neq M(v_2)$, hence for each v_i there is $M(v_i)$ that is distinct from all other $M(v_j)$ if $v_i \neq v_j$. Thus, $n_l \geq \sum_{i=l+1}^{max\ length\ level} n_i$, and the lemma follows.

Lemma 18.5.2. *The number of the nodes with length level l is bounded by $n/2^l$.*

Proof. In order to prove the lemma it is sufficient to prove the following equation

$$\sum_{i=l}^{max\ length\ level} n_i \leq \frac{n}{2^l} \tag{18.1}$$

since $n_i \geq 0$ for every i.

The proof of Equation 18.1 is obtained using induction on *length level*.

Base case. The *length level* equals to 0. The lemma holds, because the total number of nodes in the tree is n.

Inductive step. Suppose that the lemma holds for all *length levels* less than or equal to l. We prove that the lemma holds also for *length level* $l+1$. Equation 18.1 equals to:

$$n_l + \sum_{i=l+1}^{max\ length\ level} n_i \leq \frac{n}{2^l}. \tag{18.2}$$

According to Lemma 18.5.1 $n_l \geq \sum_{i=l+1}^{max\ length\ level} n_i$. Thus, Equation 18.3 follows from Equation 18.2.

$$2 \sum_{i=l+1}^{max\ length\ level} n_i \leq \frac{n}{2^l}, \quad \sum_{i=l+1}^{max\ length\ level} n_i \leq \frac{n}{2^{l+1}} \tag{18.3}$$

The lemma follows.

Lemma 18.5.3. *The distance from node i with length level l to the nearest node towards the root with a bigger length level is bounded by $2^{(l+1)}$.*

Proof. When the distance from i to the *root* is equal to or smaller than $2^{(l+1)}$ the lemma follows, since the *root* has the highest *length level* among other nodes. Suppose now that the distance to the *root* is greater than $2^{(l+1)}$. There are two cases: the first case is when the first 1 in $length_{bin}(i)$ is placed at position l, and the second case is when the first 1 in $length_{bin}(i)$ is placed at a position greater than l.

Consider the first case. Since the distance from i to the *root* is greater than $2^{(l+1)}$ there is 1 in $length_{bin}(i)$ at position k which is greater than l. Let $\bar{i}$ be the node that lies on the path from i to the *root* and $length_{bin}(\bar{i})$ is the same as $length_{bin}(i)$ except in position l. The $\bar{i}$'s *length level* cannot be smaller than $(l+1)$ because the first 1 in $length_{bin}(\bar{i})$ is at a position greater than l and the maximal *length level* in $\bar{i}$'s sub-tree is at least l since i is in this sub-tree. The distance from i to $\bar{i}$ is 2^l, thus the lemma follows in this case.

Consider the second case. Let k be the position of the first 1 in $length_{bin}(i)$. Let $\bar{i}$ be the node that lies on the path from i to the *root* and the distance between i and $\bar{i}$ is 2^{l+1}. There are two sub-cases, when $k > (l+1)$ and when $k = (l+1)$. When $k > (l+1)$ $length_{bin}(\bar{i})$ has no 1 at positions less than $l+1$. When $k = (l+1)$ $length_{bin}(\bar{i})$ is the same as $length_{bin}(i)$ except in position k. Since the distance from i to the *root* is greater than $2^{(l+1)}$, $length_{bin}(i)$ has 1 at a position greater than k. In the two sub-cases the first 1 in $length_{bin}(\bar{i})$ is in a position greater than l. Since i is in the $\bar{i}$'s sub-tree the maximal *length level* in this sub-tree is at least l. Therefore, $\bar{i}$'s *length level* cannot be smaller than $(l+1)$. The lemma follows.

Before we will prove the next lemma we must note that the maximal *length level* assigned by the algorithm is $log(D)$.

Lemma 18.5.4. *The time complexity of phase 1 is* $O(Dd_nC + nP)$ *and its message complexity is* $O(d_n n \log(D))$.

Proof. During the first phase the algorithm uses the following types of messages: a message with distance to the root, a message with maximal length level in the sub-tree of each node, a message that notifies the node's father that the node finished sending requests, a message with request for the address of the destination, and a message with reply that contains the destination address. The complexities related to the first three types of the messages are the same. The complexity related to the last two types are also the same. Hence the complexity of the second phase is determined by the complexity related to the messages with distance to the *root* and by the complexity related to the messages with destination address.

Consider the messages with distance to the *root*. Each node receives one such message, thus their *message complexity* is $O(n)$. The messages are disseminated in one direction, hence their *communication delay* is $O(DC)$. In order to evaluate the *execution delay* related to these messages consider a critical path $\{s_0, s_1, ..., s_k\}$, where messages are sent from s_{i+1} to s_i and s_0 is the *root*. The *execution delay* of these messages is equal to the sum of *execution delays* at each node, denoted by t_i:

$$\sum_{i=0}^{k} t_i, \tag{18.4}$$

The node spends a constant time when it sends such a message to each of its children, hence t_i equals to the number of node's children. Since the sum of children in the tree cannot exceed the number of the nodes of the tree, the *execution delay* of the messages with distance to the *root* is bounded by $O(nP)$. The *time complexity* related to the messages with distance to the *root* is $O(DC + nP)$.

Consider now the messages that contain the addresses of the destinations. These messages are always sent in one direction, hence their *communication delay* is $O(DC)$. Since the paths, which messages with the same level but iwhich are sent by the different nodes pass, have no common edges, the *execution delay* is the $\sum_{i=0}^{log(D)} t_i$, where t_i is the *execution delay* of processing messages with level i. t_i consists of the *execution delay* of receiving the message from the node's father and retransmitting the message to the children. According to Lemma 18.5.3 the first component is bounded by 2^{i+1}. The second component is bounded by the number of nodes with *length level* i. According to Lemma 18.5.2 this number is bounded by $n/2^i$. Hence, the *execution delay* is:

$$\sum_{i=0}^{log(D)} 2^{i+1} + \frac{n}{2^i} \leq O(n) \tag{18.5}$$

The *time complexity* of these messages is $O(DC+nP)$. Consider now their *message complexity*. Each node sends one such message to a distance bounded by 2^{i+1}, where i is the node's *length level*, according to Lemma 18.5.3. The number of nodes with *length level* i is bounded by $n/2^i$ according to Lemma 18.5.2. Hence, the *message complexity* of these messages is:

$$\sum_{i=0}^{log(D)} 2^{i+1} \frac{n}{2^i} \leq O(nlog(D)) \tag{18.6}$$

The lemma follows.

As explained, the continuation of the algorithm is very similar to algorithm weighted collect-rec and thus we omit the detailed description and analysis. The overall complexity of the algorithm is stated in the following theorem.

Theorem 18.5.2. *The time complexity of the algorithm weighted collect on trees is* $O(n(P+Cd_n))$*, and the message complexity is* $O(n(d_n+\log D))$ *for the first routing method, and the time complexity is* $O(n(P + Cd_n) + C\bar{d}\log D)$*, and the message complexity is* $O(n(\bar{d} + d_n + \log n))$ *for the second routing method.*

18.6 Gathering Information from General Graphs

In the previous sections we assumed that the overlay network maintains a spanning tree, in which each node knows its parent and its descendants. For many overlay applications (such a CDN) this is a very natural assumption as some information should be sent (or collected) from all the nodes. For other applications, such as Peer-to-Peer file sharing applications, maintaining a global spanning tree may be too expensive since the amount of users that join (or leave) the Peer-to-Peer network each time unit is too big.

Nevertheless, in all non-trivial proposals for a structural Peer-to-Peer network, maintaining such a tree requires at most a very small change to the existing infrastructure. If a spanning tree does not exist, one will have to create a spanning tree, and run our algorithm on top of it (assuming the amount of data collected is big enough). In order to create such a tree, one can use the well known algorithm, [37] in which every node that receives a message indicating the creation of the tree, sends this message to all its neighbors. Each node replies to the first creation message with an "I am your child" message, and to all other messages (i.e. not the first one) with an "already in tree" message. This phase can be combined with the first phase of the algorithms described in the previous section, as they both start with a message being sent from the root down the spanning tree.

The complexity of creating such a tree is independent of the routing method since messages are exchanged only among overlay neighbors. The message complexity is of course $O(d_n m)$, where m is the number of links in the overlay network, and the time complexity is $O(Cd_n D + S_n P)$, where $S_n = \max_{all\ paths\ \pi\ in\ tree} \sum_{n \in \pi} degree(n)$.[a]

In general the only bound for the value of S_n is $2m$, but in many cases one can build a tree with much better values (see [93] for a discussion on this subject in a different model). If, however, the outdegrees of the overlay network is bounded by $\log n$ and the height of the spanning tree is also logarithmic in the number of nodes, as indeed is the case in most practical scenarios, then the time complexity of creating the tree becomes $O(Cd_n D + P \log n^2)$, and the overall time complexity of gathering information from the entire overlay network without assuming a spanning tree is the same as the one stated in Theorem 18.4.1, namely $O(n(P + Cd_n))$ for the first routing method, and $O(n(P + Cd_n) + C\bar{d} \log D)$ for the second routing method.

18.7 Global Functions

In many cases, exploring the path between two nodes is just an intermediate step towards computing some function along this path. A typical example is bottleneck detection: we want to detect the most congested link along a path. Another typical example is the need to know how many copies of a certain file are available in a Peer-to-Peer file sharing system. In both cases

the computation can be done using a single pass on the data (path in the first example, and tree in the second one) using constant size messages.

Bottleneck detection is a special case of a a global sensitive function [121] which we term succinct functions. These functions, e.g., average, min, max, and modulo, can be computed with a single path on the input, in any order, requiring only constant amount of memory. For such functions we can prove the following theorem.

Theorem 18.7.1. *Every succinct function on a path can be computed with time complexity* $\Theta(nd_n(P+C))$ *and a message complexity of* $\Theta(n)$*, and these bounds are tight.*

In a similar way, we can define succinct-tree functions as functions from a set of elements $\{X_i\}$ to a single element x, such that if $f(\{x_i\}) = x$ and $f(\{y_i\}) = y$ then $f(\{x_i\} \cup \{y_i\}) = f(\{x\} \cup \{y\})$. Such functions can be computed on a tree using a single bottom up pass with fixed length messages, and thus the following theorem holds.

Theorem 18.7.2. *Every succinct-tree function can be computed on an overlay network with time complexity of* $\Theta((Cd_nD+S_nP)$*, and message complexity of* $\Theta(m)$ *(*$\Theta(n)$ *if a spanning tree is available), and these bounds are tight.*

Note that since we only use messages between neighbors in the overlay network, the same results hold for both routing methods.

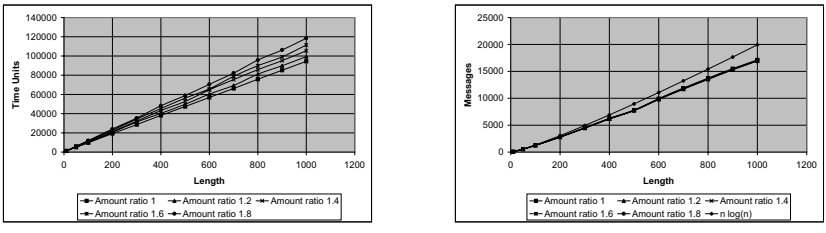
18.8 Performance Evaluation

In order to verify the practicality of the algorithms presented in this chapter, we evaluated their performance on large networks that contain thousands of internal nodes. The results were obtained using a specially built simulator that allows us to simulate runs on arbitrary big networks. To make the presentation clear, we only considered the first routing scheme, with $d_n = 1$.

18.8.1 weighted collect-rec Algorithm Performance

In this section we present the results of running the weighted collect-rec algorithm. Figure 18.14 depicts the results of collecting data that is distributed uniformly among all the nodes. The X axis is the path length. In these runs we checked the *time* and *message complexity* of the algorithm for different values of *amount ratio*, from 1 up to 1.8. Remember that the *amount ratio* is the ratio between the amount of the data in all the nodes and the path length. The results show that the message complexity is independent of the amount of transported data. For clarity of the presentation, we plotted in Fig.(18.14.b) the line for the theoretical bound $O(nlog(n))$.

As for time complexity, there is a clear difference in the running time with different *amount ratio*s. Note, however, that this difference is not proportional to the difference in the *amount ratio*. This can be explained by the fact that the running time of the first two phases depends only on the path length. The difference in the running time of the algorithm for different *amount ration*s, is introduced by the third phase, when the actual data collection is done.

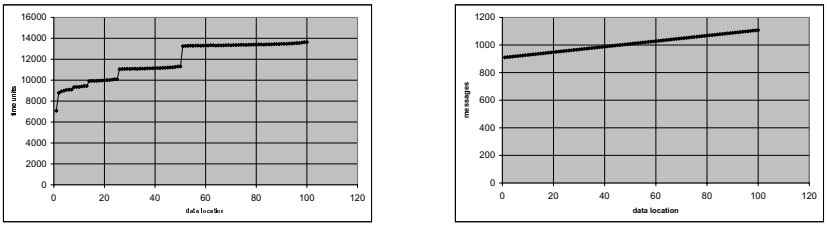


(a) Time (b) Messages

Fig. 18.14: weighted collect-rec performance

Figure 18.15 depicts the results of collecting data from a path where all the data is concentrated in one node. Note, that in this case the node that contains all the data must send the data directly to the *root* during the last phase. In each chart, the X axis describes the distance from the single node that contains data to the *root*. The number of nodes in the path is 100 in charts a and b. Both, the number of messages and the running time increase when data is located further away from the *root*. The initial values show the minimal time and number of messages required to accomplish the first two phases (i.e. the *time complexity* or the *number of messages* when the data is actually located at the *root*). The number of messages increases proportionally to the distance from the node to the *root*. As the node is further away from the *root* the message with the data is transmitted over a greater distance, and the number of messages grows. The growth of the running time looks like a step function. The figures show that the steps appear at values of $n/2^i$. Since the node sends the data directly to the *root* during the last phase, the algorithm must deliver the address of the *root* to this node during the second phase. Considering the path of the address delivering. The message with the address may be processed by other nodes and each time when the maximal *length level* of the nodes that participate in the delivering increases a new step starts. This is because the amount of work done by the node during the second phase is proportional to $n/2^i$. Inside each step (that is better seen for the small values of x) there is a weak increase of the running

time. This increase is caused by the growth of the *communication delay* that a message with data suffers when the distance to the *root* increases. The presented results were obtained by running one experiment for each value of x. A larger number of experiments here is useless, since the simulator gives the same results for exactly same starting parameters.

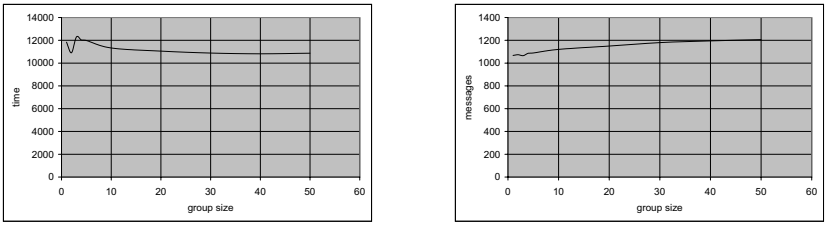


(a) Time (b) Messages

Fig. 18.15: weighted collect-rec performance. All data in one node

Figure 18.16 depicts the results of collecting data from a path where an arbitrary group of nodes contains all the data in the system. The number of nodes in the path is 100 in charts a and b. In each chart the X axis describes the size of the set of the nodes that contains all data. Note that the data is distributed uniformly among all the nodes in the set. The charts show that the running time decreases when the group size grows from $4-5$ up to approximately 20% of n. When the size of the group increases the data processing becomes more parallel and this causes the processing time to decrease. When the size of the group continues to grow, the time required for processing the data in each node increases since there are more data packets. This process balances the effect of the parallel processing and the total time required is the same.

As can be seen from Figure 18.16 (b), the number of messages increases logarithmically with the increasing of the group size. The charts show that as the group grows the average cost (in messages) of adding new nodes decreases. The average value of the standard deviation for the running time is 13% and for the number of messages is 3%. The standard deviation has higher values for the small group and it decreases as the group size increases.



(a) Time (b) Messages

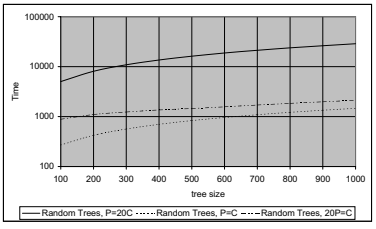
Fig. 18.16: weighted collect-rec performance. All data in a set of nodes

18.8.2 Performance of weighted collect on trees Algorithm

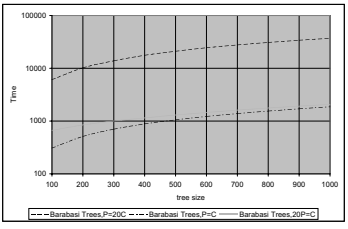
In order to evaluate the weighted collect on trees algorithm we used two network models. The networks created according these models have different topology properties. The first model, denoted by *random tree*, is the network topology where probability that a new node will be linked to an existing node is the same for all existing nodes (also known as $G(n, p)$). However, recent studies indicate that the Internet topology is far from being a random graph [201]. In order to capture spanning trees of such model the described algorithm were also tested on *Barabasi trees*. In this model the probability that a new node will be linked to an existing node with k links is proportional to k [637]. As it will be shown below, the differences in the network models affect the performance of the algorithm. The results of evaluating the performance of weighted collect on trees algorithm is depicted in Figures 18.17. In both charts, the x axis describes the size of the tree. Figure 18.17 (a) shows the results of running the algorithm on *random trees* and Figure 18.17 (b) shows the results of running the algorithm on *Barabasi trees*. The plotted results reflect an average cost taken from 1000 runs per each tree size, and the values of standard deviation do not exceed 4%.

One can observe that both the time and the number messages grow linearly, [3] this agrees with the theoretical analysis. The running time of the algorithm is smaller on *random trees* than on *Barabasi trees*. This is explained by the fact that *Barabasi trees* have a small group of nodes with a larger number of children (in *random trees* the distribution of children among the internal nodes is more uniformly) and these nodes perform a lot of work. The number of messages is greater when the algorithms run on the *random trees*. This can be explained by the fact that the diameter of *Barabasi trees* is less than the diameter of *random trees*.

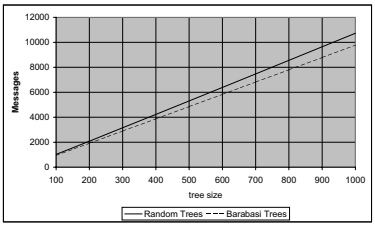
[3] Note that the time scale is logarithmic.



(a) Time for random trees



(b) Time for Barabasi trees



(c) Messages

Fig. 18.17: weighted collect on trees algorithm performance

Note that the actual value of the time complexity depends on the ratio between C and P. In practical scenarios this ratio depends on the network RTT (since C also present the propagation delay), the type of processing, the architecture of the overlay network, and the efficiency of data handling at the nodes. However, in a Peer-to-Peer network, where distant hosts can be neighbors in the overlay layer, this ratio could indeed be small (i.e. ≤ 1), while in local area overlay networks (as in Active Networks prototypes) this ratio could be as big as 20.

When creating the graphs in Figure 18.17, we used $p = c = 1$ to calculate the case $c = p$, $c = 1, p = 20$ to calculate the case $p = 20c$, and $p = 1, c = 20$ to calculate the case $c = 20p$. It is clear then, that the fastest case is when $c = p$. However, as indicated by the graphs the affect of increasing p is much more severe than increasing c. If we look at the Barabasi tree, we see that it takes about 2000 time units to collect information from a 1000 node tree (where $c = 20p$). Assuming $p = 0.1$Ms, we can infer that a 100,000 node graph could be collected in 20 seconds assuming RTT of 2Ms. For more realistic RTTs, and

assuming an open TCP connection between peers, one can collect information from 100,000 nodes in less than a minute, using algorithm weighted collect on trees.