Aggregated Hierarchical Multicast—A Many-to-Many Communication Paradigm Using Programmable Networks

Tilman Wolf, *Member, IEEE,* and Sumi Y. Choi, *Student Member, IEEE*

***Abstract*—Developments toward ubiquitous and pervasive computing have lead to application scenarios where a large number of sensors and computing devices are connected to the network. These devices might constantly send status information via multicast to a number of applications or users. One big challenge in this environment is the amount of data traffic generated by such sensors, which depends on the size of the data, the transmission frequency, and the number of senders and receivers. However, for certain applications it is sufficient to receive less accurate, aggregated data from a group of sources. This leads to the possibility of using programmable routers to perform such data aggregation inside the network.**

While the basic algorithm for data merging has been described in literature, we address how a large number of sources can be organized in a hierarchical structure to allow multiple users to get a view of all sensors at different levels of aggregation. With the control information that is provided by an aggregating node, the user can traverse the aggregation tree by joining different multicast sessions that transmit different levels of detail. This provides a novel communication paradigm that reduces the network overhead from continuously transmitting sources and organizes data in a way that it is useful to receivers.

We provide two detailed example scenarios: A battlefield information system, which aggregates geographic location data of units, and a conferencing application, which aggregates audio data. We describe the aggregation algorithm that is used and analyze its effect on delay and jitter of periodic transmissions by the sources. We describe the hierarchical control structure that provides multiple levels of aggregation and how real-time transport protocol (RTP) can be used to implement it. The performance of the proposed scheme is evaluated with measurements that were done on an implementation of the audio aggregation application.

***Index Terms*—Active network application, computer networks, data aggregation, many-to-many communication, programmable networks.**

I. Introduction

TECHNOLOGY and integration of semiconductor components have reached a level where pervasive and ubiquitous computing is becoming a reality. Typical examples are sensor networks that report status information, appliances and entertainment electronics in a household, and personal-area network devices. Using low-power wireless links, these devices form data networks of over which information is transmitted in an ad-hoc fashion or over a network infrastructure. One characteristic of devices in such networks is that they often continuously report status information by periodically transmitting data. Such data is then collected by a central node, processed, and presented to users.

The challenges that these networks pose lie not only in how to realize small, low-power devices that perform the desired functions, but also in their operation. Key aspects of operation are how to achieve connectivity on a link and on a network level between all components, how to deal with failure, and how to develop a scalable method of controlling information flow from these devices. The last point-managing the flow of information-is what we address in this work.

In a typical scenario, more than one user or "observer" wants to receive data from possibly a large number of sources. This can be seen as many-to-many or multisource communication. In a traditional many-to-many communication, each sender is connected to each receiver (be it over unicast or multicast). As a result, a receiver has to deal with as many connections as senders that he observes. As the amount of transmitted data and the number of sources increases, the receiving end-system has to process increasingly more data traffic. This does not scale well, neither in terms of bandwidth requirements nor in terms of computational demands, in particular for thin, mobile clients with little computational resources and low bandwidth network connections. Thus, the goals of an advanced communication paradigm for many-to-many communication are the following:

1) avoid overloading the network and data processing nodes with constantly transmitting sensors;
2) allow the aggregation of a possibly large number of sources to present data in a manageable fashion;
3) be adaptable to changes in user's interest.

We present aggregated hierarchical multicast (AHM), which provides a solution to each of these goals. The key observation is that for certain applications in many-to-many communications the data from all senders is not of equal importance to the receiver. Thus, it is sufficient to aggregate the information sent by most of the sources, while keeping the original data stream from a few selected sources. For this purpose AHM merges data streams inside the network using an active or programmable network infrastructure. To achieve a reduction in data traffic, the aggregation is typically associated with a loss of fidelity. Different levels of detail should be available to observers and therefore aggregates are arranged in a hierarchical fashion and dis-

Manuscript received August 29, 2002; revised March 25, 2003 and June 6, 2003. This work was published in part the *Proceedings of IEEE MILCOM* 2001. This paper was recommended by Guest Editors S. Karnouskos, A. Vasilakos, and W. Pedrycz.

T. Wolf is with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA, USA (e-mail: wolf@ecs.umass.edu).

S. Y. Choi is with the Department of Computer Science and Engineering Washington University, St. Louis, MO, USA (e-mail: syc1@arl.wustl.edu).

Digital Object Identifier 10.1109/TSMCC.2003.817352

tributed via multicast. Observers can navigate through the hierarchy to obtain the desired level of granularity. This approach allows a significant reduction of data traffic as shown in our evaluation, especially when receivers are only interested in high-level information of large groups of senders.

Section II briefly discusses relevant related work. Section III then describes the functionality of AHM in detail on the basis of two example scenarios: a battlefield information system and an audio conferencing application. Also, different information aggregation methods are discussed. Section IV describes the hierarchical structure of multicast sessions that provides various levels of aggregation detail to the user. The general-purpose aggregation algorithm for an AHM node is presented in Section V. Section VI then focuses on the analytic evaluation of the proposed aggregation process. Theoretical bounds for jitter and delay in periodic transmissions are derived. Measurement results from our testbed implementation of AHM are also presented. A summary of the contributions in Section VII concludes this paper.

II. Related Work

Sensor networks with large number of devices have been proposed and prototyped. Projects at UC Berkeley and Intel Research have recently demonstrated large-scale prototypes where sensors monitor the environment (light intensity) and communicate via an ad-hoc network to deliver this information. An extension that monitors wildlife habitats has been proposed [1]. Similar research has been supported by DARPA for its importance to military applications. The technical issues of building small, low power devices that can transmit via short-range radios have been addressed [2], [3]. Also, numerous routing protocols for such ad-hoc environment [4], as well as failure recovery have been presented. Our work is based on these results and addresses the operational issues of how to manage the data of such a system efficiently. We assume that an underlying network infrastructure is given.

The aggregation operation of data packets inside the network requires the support of the network infrastructure in terms of processing resources. With the development of active and programmable networks [5]–[7], it is possible to perform arbitrary computation in the data path of network nodes. An active node in such a network is capable of performing the aggregation processing of packets as they are being forwarded. Typical implementations of programmable network nodes range from workstations that act as active routers to high-performance switches that are augmented with per-port network processors. The definition of a unifying node operating system (NodeOS [8]) aims at making these systems interoperable. Aggregated Hierarchical Multicast is an application for active networks that illustrates the benefits of having a programmable network infrastructure. However, we do not go into the details of active networking architectures in this paper.

In the context of multicast, data transcoding has been proposed by Kouvelas [9] for receivers with bad reception. This "self organized transcoding" (SOT) adapts dynamically to changing network conditions. There is also abundant work on "active multicast" [10]–[12], which use active networks to improve the transmission quality of multicast sessions. This work is related to ours only insofar as we could use it to distribute the aggregated data streams.

In terms of performance, it is important to have sufficient processing resources on nodes to perform data aggregation. For this purpose, it is possible to use network processors. Commercial instances of such multiprocessors that are specialized for packet processing are the Intel IXP2800 [13] and the IBM PowerNP [14]. While the programming of such a system requires more effort than a general active network node, it provides better performance and scalability. It can be expected that as more sophisticated software tools become available for network processors, many active network architectures will be ported to such platforms.

The aggregation of data streams that we present is similar to "reverse multicast,"which has been introduced as "concast" in the context of active networking. Calvert *et al.* [15] introduce "simple concast" as an aggregation mechanism to suppress duplicate packets (for example to avoid NAK implosion in multicast). This work also provides a generic framework for defining the aggregation function to be used. In our work, we expand this concept and introduce a novel hierarchical organization of aggregating nodes that provides a structure for efficient management. We also show a detailed delay and jitter analysis of the our extended algorithm that considers periodic transmissions. With the presented result, we show that aggregated hierarchical multicast is scalable approach to many-to-many communication in programmable networks.

III. Information Aggregation and its Applications

As discussed above, the limited scalability of many-to-many communication lies in the demands on the network to deliver numerous data streams to the end system and the demands on the end system to process and display the information. We focus on two applications that implement such a mode of communication. One is a battlefield information system, which is characterized by a large number of "sensors" (i.e., soldiers) and a clear hierarchical structure that can be used for the aggregation. The other is a audio conferencing application, which is an example for a very intuitive aggregation of audio sources. Section III-A describe these applications in more detail and highlight the properties relevant to this work.

There are also a number of applications that require information aggregation. In the context of networking, ACK or NACK aggregation is necessary for multicast to avoid an implosion on the sender. This is often a simple Boolean function. In ATM networks, more complex functions are used to aggregate available bit rate (ABR) responses from point-to-multipoint connections [16].

A. Application I: Battlefield Information System

The purpose of a battlefield information system is to provide status information of a large number of soldiers and equipment to a group of commanders ("observers"). We assume that all soldiers are connected to a common interconnection network (e.g., via wireless links) and have the equipment to periodically transmit their status information (e.g., geographic location, vital

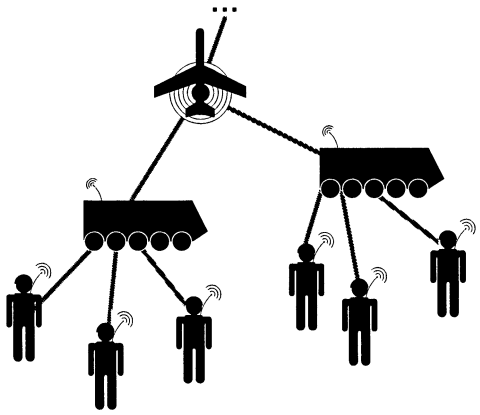

Fig. 1. Battlefield information application for aggregated hierarchical multicast. Each node sends its status information to its hierarchical parent, where it is aggregated.

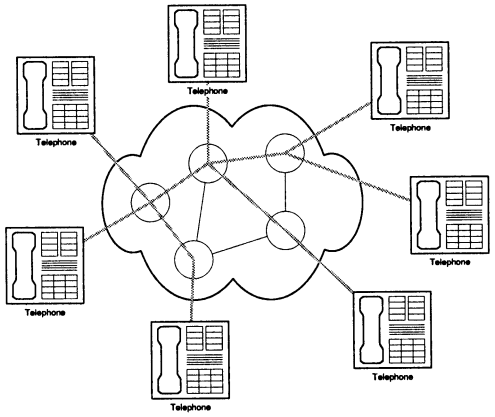

Fig. 2. Audio conferencing application for aggregated hierarchical multicast. Each end system sends audio that is mixed at nodes where multiple audio streams merge. The end system receives a single, mixed audio stream.

statistics, and other useful data). The observers can receive this data and display the status of all soldiers accordingly.

The challenge of this application lies in the large amount of data that is received by an observer. Considering only a few thousand soldiers who transmit every few seconds gives an average of several hundred data packets per second that have to be processed and displayed. Thus, such a system s limited in its scalability if all sources continuously sent multicast messages to the group of observers.

It is unlikely that an observer is interested in the exact status of every single soldier at all times. Instead, the overall status of a group might be more relevant. Thus, it can be considered to present aggregated status information of a group of soldiers to an observer and thereby reducing the number of data streams that are sent to a receiver. One way to aggregate geographic status information, for example, is to send only the location of a group instead of all individuals. The group could be represented by its centroid, the weighted geographic average, or the bounding box. This reduces the amount of detail in the representation, but it also reduces the amount of data that needs to be transmitted. Similarly, vital statistics can be aggregated. For example, the predicate "healthy" of each soldier could be aggregated using a Boolean "and" function. Repeating this aggregation on different levels allows further reduction of data transmission.

We can see that the clear hierarchical structure of a military organization lends itself well to an aggregation scheme, where the data of a group is aggregated by its parent (i.e., the node that is "in command"). An illustration of this is shown in Fig. 1. More details on how to tap this hierarchy at the right level to get the right detail of information is discussed in Section IV.

B. Application II: Audio Conferencing

An audio conferencing application, as illustrated in Fig. 2, is another typical example of a many-to-many communication. Each participant in the conference needs to be able to hear all other participants and therefore needs to receive their audio data stream. In a traditional scenario without network support, the end-system has to receive all these data streams, mix them together, and play the result to the user. In a programmable network, where data can be processed inside the network, it is possible to merge audio streams together on common nodes. This aggregation reduces the overall data transmission as well as the load on the end-system, which does not have to merge all data streams anymore. A dynamically adaptable audio mixing technique called distributed partial mixing (DPM) has been proposed by Radenkovic *et al.* [17]. DPM is TCP-fair and adjusts to varying number of participants and network conditions. While DPM can achieve better audio quality by delaying the mixing step, our aggregation mechanism achieves lowest bandwidth usage. Also, the AHM mechanism used here is simpler and illustrates the aggregation step more clearly.

Similar to the battlefield application, there are also several levels of aggregation. Depending on the application, it might be desirable to listen only to a smaller group of people. Such a scenario could occur when multiple people in two conference rooms use audio conferencing. If everyone has their own microphone and speaker, one might choose to receive only audio from the remote conference room and not from other participants in the local room (because direct audio communication and the conference application audio might interfere).

C. Data Aggregation

From these two applications, it can be seen that aggregation of data in the network is beneficial. For one, aggregation reduces the amount of data that is transmitted, which reduces the network load. Another effect is that the load on the receiving end-system is reduced, because aggregation steps that are necessary to display the data are moved from the end-system into the network.

In principle, there are three dimensions in which data aggregation can be performed:

1) transmission frequency: reduction in the number of packets transmitted;
2) number of senders: reduction in the number of data streams;
3) amount of data: reduction in the size of data that is transmitted.

These aggregation categories can possibly influence each other. Aggregating a set of senders in to a single data stream typically increases the amount of data that is sent on this stream (as compared to a single sender). However, the reduction in the

number of streams leads to an overall reduction of transmitted data.

There are several issues that have to be addressed in order to achieve efficient data aggregation. For one, it has to be possible to aggregate the data that is transmitted. In many cases, aggregation can be achieved by reducing the fidelity of the information (e.g., scale change for geographic information) or generating an overlay of different data streams (e.g., mixing of audio). Status information can often be aggregated by simple arithmetic and Boolean functions. However, certain data, e.g., text messages, cannot be scaled or aggregated effectively without losing crucial components of the data. In such a case, the information can be concatenated and transmitted unchanged, losing the benefits of reduction in bandwidth and processing requirements. For the remaining discussion, we assume that an effective aggregation algorithm is available.

The applications described above can use aggregation in several dimensions. For example, in the battlefield information system, it might be necessary to have high frequency updates for the status of an individual soldier who moves around quickly. The centroid of a group, though, moves at a slower rate and therefore needs to be updated less frequently. Also aggregating several individuals into a group reduces the number of senders. In the conferencing application, the number of streams is reduced at an aggregation node. However, the frequency of transmission and the data size remains the same due to the stringent requirements of audio applications. A qualitative analysis of the benefits of aggregation is presented in Section VI.

IV. Hierarchy of Source-Based Multicast Sessions

To make aggregation of data inside the network practical, two key issues need to be addressed. First, it is necessary to provide different levels of aggregation to different users. Only in a few applications it can be argued that all users will need information at one level of aggregation. Second, users have to be able to dynamically change the level of aggregation that they receive.

We propose a hierarchy of multicast session to accommodate these requirements. Each node in the hierarchy provides data streams with a certain level of aggregation. To allow dynamic adjustment of the level of aggregation, we use control information from a node that identifies other nodes, which provide data streams with higher and lower levels of aggregation. These ideas are elaborated in Sections IV-A–D.

A. Hierarchy Layout

Each network node that aggregates data streams needs to provide the results to possibly multiple receivers. These receivers can either be end-user applications or other nodes that aggregate that data further. To avoid the overhead from multiple unicast connections, we assume that a node provides its data stream in a multicast tree with the root being the node. This is illustrated in Fig. 3.

To obtain a hierarchical structure that can be used to navigate through different levels of aggregation, we arrange nodes in tree form. Thus, each node (other than the root of the tree) has a parent assigned that is always a receiver of the aggregated data stream (as illustrated in Fig. 3). The resulting tree structure is shown in Fig. 4. Each layer represents a different level of aggregation. The lowest layer, layer 0, contains the data sources (i.e., soldiers, telephones) that send unaggregated data. Each node in layer 1 aggregates multiple layer 0 sources to a new stream. This stream is sent upwards to layer 2, where it is aggregated with other layer 1 streams. This continues up to the root node. In general, a layer-i node aggregates streams from layer $i-1$ and sends it to layer $i+1$. Users can connect to nodes in any layer to receive data streams with different levels of aggregation. If more detail is required, the observer can connect to a child of the current node. If less detailed information is required, the observer can move up to the parent of the current node.

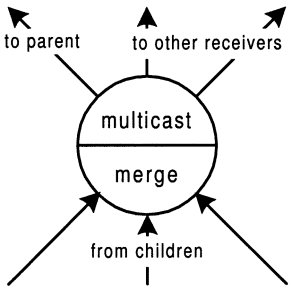

Fig. 3. Aggregating node. Multiple data streams are received, aggregated, and sent via multicast to a number of receivers.

This hierarchy maps each node of layer 1 and higher to a multicast group (each representing a different set of layer-0 sources or different levels of aggregation). In practice, it might not be feasible to use such a large number of multicast groups. To reduce the number of multicast sources, certain aggregation nodes can be connected to their parents via unicast and not provide their aggregated data stream of other observers. This however reduces the granularity at which an observer can receive data.

It is important to note that the efficiency of the proposed hierarchy is based on the assumption that it follows the underlying network topology. This is a reasonable assumption as sources that are closely co-located probably share the same networking infrastructure and transmit somewhat related data.

B. Session Control

On the control plane of aggregated multicast, observers have to be able to join and leave the various multicast sessions that they are interested in. For this purpose, there are three functions that need to be provided:

1) joining/leaving multicast groups;
2) initialization of the tree layout;
3) obtaining control information on parents/children of nodes.

For joining multicast groups, we assume all nodes and observers support standard multicast protocols (e.g., by using multicast routers that exchange IGMP [18] messages and route using MOSPF [19]). The initialization of the tree requires all nodes-other than leaves-to know who their children are, so they can subscribe to their multicast streams and further aggregate them. This can be achieved by static configuration or by using a naming scheme as described below. A scalable way for configuring is to have the children send the parent a control message that causes the parent to join the child's multicast group.

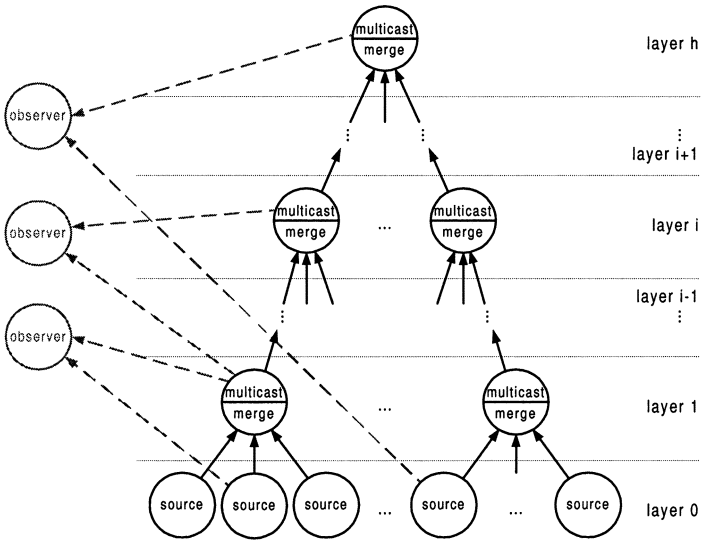

Fig. 4. Hierarchy of data aggregation. The parent of each node and possibly multiple observers can connect to the multicast session provided by an aggregating node.

Finally, the observer needs to obtain information on the name and multicast identifier of the children and the parent of a node. This is necessary so that the observer can change to a different level of aggregation if necessary. For this purpose, the data stream can be aggregated to include the multicast identifiers of children and parent of a node. The real-time transport protocol (RTP) used in our audio application supports "contributing source identifiers," which can be used for this purpose [20]. We assume that the multicast identifier for the root node is always known to all observers (or at least easily obtainable) as the starting point of navigation.

C. Naming Issues

For scalability reasons, the control data of a session cannot contain the names/identifiers of all children, grandchildren, etc. This would lead to exponential growth in control information and defeat the purpose of aggregation. This poses some difficulty if an observer that starts at the root node wants to "zoom in" on a particular node. Since the control information at the root node might only contain information on its children, it is not clear which path to go in order to reach a particular node on the lower layers.

To solve this problem, we add a naming scheme to the hierarchy. Each node is identified by its name and the concatenation of names of nodes in the path from the root to the node. This results in a hierarchical naming scheme that gives a one-to-one mapping from a name to a node and its multicast group. The concept is similar to that used in the domain name service [21] in the Internet or that of class names in object oriented programming languages, like Java [22]. Thus, the full name of leaf x contains the list of higher layer nodes that lead to x (i.e., the path from the root to x), which allows the observer to easily navigate through the tree. For convenience and to abbreviate long names, certain intermediary nodes can be aliased with unique names.

For our two example applications, such naming schemes can easily be derived. For the battlefield applications, it is intuitive that the hierarchical structure follows actual command chains. Thus, the naming scheme can be derived by concatenating the chain of command for a given node. In the conferencing application, the naming scheme could be the name of the company followed by the location of the building followed by the number of the conference room from where the call is made.

The hierarchical structure, the distribution of control information, and the naming scheme enables the observer to navigate through different aggregation levels of a large number of sensors and find a suitable level of detail. Next, we discuss the actual algorithm used for aggregation of data inside the network.

D. Limitations

The proposed hierarchical structure assumed a static tree structure and a fixed assignment of names to nodes. This can cause problems if any of the following dynamic conditions occur.

1) Nodes move physically. This can cause the underlying networking topology to change and cause inefficiencies in the aggregation process.
2) Nodes move logically in the hierarchy. This causes the node names to not match the hierarchical naming scheme.
3) Tree becomes unbalanced. This does not cause incorrect operation, but could cause performance degradation due to large delays.

For the first case, it is necessary to adjust the logical hierarchy to match with the underlying network topology. This change is equivalent to moving nodes within the hierarchy. Such a move requires that the names are adjusted accordingly and the aggregating parent nodes are reconfigured to reflect the change. The final case can also be solved by moving nodes to balance the tree.

The control issues of these hierarchy adjustments are not discussed in detail here. We are mainly focusing on changing user interest on a fixed hierarchy (i.e., more or less granularity of aggregation) and the performance issues of the aggregation process.

V. Aggregation Algorithm

The aggregation of data is the functionality that has to be provided inside the network. Here we discuss how packets can be efficiently aggregated while providing bounds on delay and jitter in case of packet losses. The focus is more on the issues of packet buffering and timeouts than the actual aggregation of the packet payload.

We assume all sources periodically transmit data in form of discrete datagrams. These datagrams are transmitted unreliably, as it is common for UDP in IP networks. Since it cannot be assumed that the sources can be synchronized, the basic aggregation has to buffer packets until data from all sources is available. Additionally, it is necessary to manage timer to ensure that lost packets do not cause excessive delay and jitter in the data stream.

Consider a router that has to merge k data streams, $s_1 \ldots s_k$, to a new, aggregated data stream s_a. Assume we are given an aggregation function F that generates a packet p_{s_a} from packets $p_{s_1}, \ldots p_{s_k}$ ($p_{s_a} = F(p_{s_1}, \ldots, p_{s_k})$). If a packet p_{s_i} has not been received during a period and the timeout is triggered after t_{timeout}, the merging function F can use either an older data packet of stream s_i or use the neutral element 0_F as a placeholder. Let us also assume for now that all sources send packets of the same size (same amount of information or samples) and with the same frequency. The resulting procedure for buffering and merging packets is shown in detail in Fig. 5. There are four parts to the algorithm.

1) *Part I (lines 7–12): A packet arrives and is stored in the buffer.* This requires that the buffer slot for that stream is not yet used.
2) *Part II (lines 13–26): A packet arrives and its buffer slot is already taken.* This happens, when the algorithm was waiting for packets from other streams that were lost or delayed. With the arrival of a second packet from a stream, we know that it is time to send the aggregated packet. Thus, empty buffer slots are filled with "null packets" and the data is aggregated and sent.
3) *Part III (lines 27–34): All buffer slots are filled.* In this case, one packet from each flow is available and we can aggregate the data and send out the result.
4) *Part IV (lines 36–46): A timeout occurred.* In this case, missing packets are replaced by "null packets" and the aggregated result is sent.

The packets are stored in the buffer array, b, that has n slots. The variable bcount keeps track of the number of valid buffer entries. The timer is set to t_{timeout} every time the first packet is put into the empty buffer. This way, no packet is ever stored longer than t_{timeout}. The timer is cleared (set to ∞) when the buffer is cleared.

This algorithm can also be extended to reduce the frequency of transmissions on the aggregated data stream. By buffering f packets from each stream, the frequency of the aggregated stream is reduced to $1/f$ the frequency of the sources. The effects on delay and jitter of periodic transmissions are discussed in Section VI-B.

```
 1: initialization:
 2: clear b
 3: bcount ← 0
 4: set timer to ∞
 5:
 6: receive packet p_{s_i} from stream s_i
 7: if b[i] is empty then
 8:    b[i] ← p_{s_i} {store packet in buffer}
 9:    bcount ← bcount + 1 {increase buffer counter}
10:    if bcount = 1 then
11:       set timer to t_timeout {set timer if this is the
                                  first packet in the buffer}
12:    end if
13: else
14:    {there is already a packet from source s_i}
15:    for j = 1 to k do
16:       if b[j] is empty then
17:          b[j] ← 0_F {fill empty buffer slots with neutral elements}
18:       end if
19:    end for
20:    p_{s_a} ← F(b) {merge packets}
21:    send p_{s_a}
22:    clear b
23:    b[i] ← p_{s_i} {store packet that was just received in buffer}
24:    bcount ← 1 {adjust buffer counter}
25:    set timer to t_timeout {set timer (since this is the
                               first packet in the buffer)}
26: end if
27: if bcount = n then
28:    {there is one packet from each source in the buffer}
29:    p_{s_a} ← F(b) {merge packets}
30:    send p_{s_a}
31:    clear b
32:    bcount ← 0 {adjust buffer counter}
33:    set timer to ∞
34: end if
35:
36: on timer = 0 do: {timeout occurred}
37: for j = 1 to n do
38:    if b[j] is empty then
39:       b[j] ← 0_F {fill empty buffer slots with neutral elements}
40:    end if
41: end for
42: p_{s_a} ← F(b) {merge packets}
43: send p_{s_a}
44: clear b
45: bcount ← 0 {adjust buffer counter}
46: set timer to ∞
```

Fig. 5. Packet merging algorithm.

VI. Evaluation

To show the effectiveness of aggregated hierarchical multicast, we first look at the reduction of link bandwidth that is achieved over a centralized solution. Second, we analyze the complexity and correctness of the aggregation algorithm and show its effects on delay and jitter. Third, we show measurements from a prototype implementation of the audio conferencing application that confirm the trends derived in the analysis.

A. Bandwidth and Computation

To analyze the benefits of aggregation in the network, we compare aggregated hierarchical multicast with traditional

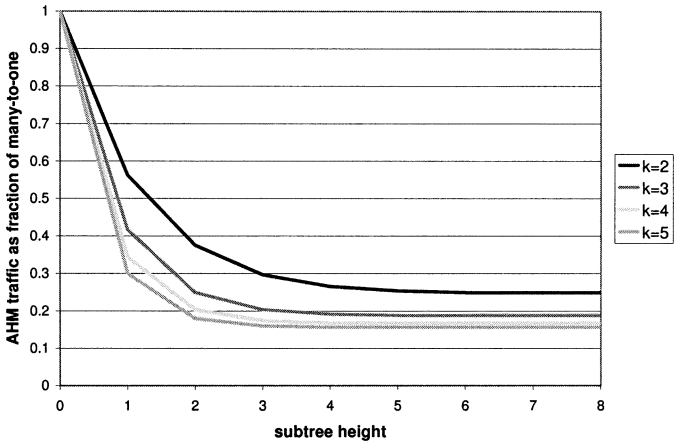

Fig. 6. Traffic reduction in aggregated hierarchical multicast over many-to-one communication.

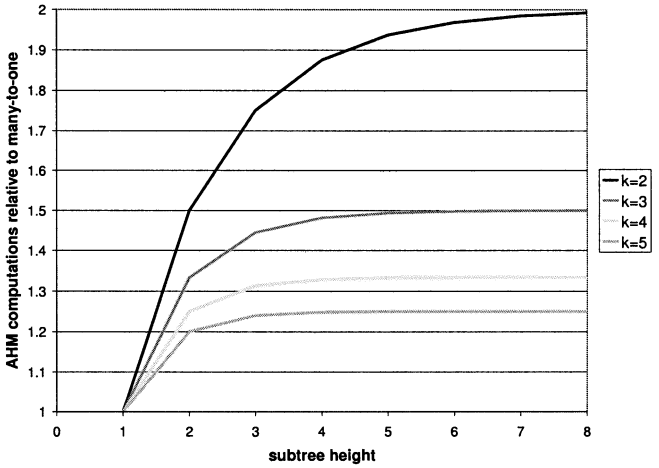

Fig. 7. Computational overhead for aggregated hierarchical multicast over many-to-one communication.

many-to-one communication, where the receiver aggregates all data. For this analysis we assume that the aggregation reduces the number of data streams on each node. The packet size and bandwidth for all streams is assumed to be equal. Assume a balanced tree of height h with nodes of degree k. Say the receiver is the root of the tree and it wants to observe all leaves of a subtree with height l. Also assume the tree maps to the network topology such that there is one link between nodes.

In aggregated hierarchical multicast, each leaf of the subtree sends one message to its parent, which is further aggregated until it reaches the root of the subtree. The total amount of bandwidth (in terms of messages times links used) in AHM is

$$b_{AHM} = \sum_{i=1}^{h} k^i.$$

In traditional many-to-one communication, all leaves have to send a message to the root. Thus, the amount of bandwidth used is

$$b_{MTO} = k^h \cdot h.$$

When a user wants to observe a subtree of a certain height, in AHM all messages are aggregated up to the root of the subtree. The result is then sent to the user. In the traditional approach, messages from the leaves are send individually to the receiver. Fig. 6 shows a comparison in terms the fraction of traffic that is necessary in AHM compared to traditional many-to-one communication. The graph shows results for different subtree sizes l in a tree of height $h = 8$. Even for small subtrees $(l \geq 2)$ (i.e., few nodes that get aggregated), the total traffic is 60–80% for AHM than for the traditional approach.

However, aggregation in the network has its price. Each data stream is aggregated possibly multiple times on the way to its destination. In one-to-many communication, each message is aggregated only once at the receiver. In a centralized system, the total number of aggregation computations is

$$p_{MTO} = k^h$$

because one packet from each leaf is processed once at the root (assuming each packet triggers an aggregation computation-even just one). For AHM, each node processes exactly one packet from each child, which adds up to the number of internal tree nodes times the degree of the nodes

$$p_{AHM} = (k^h - 1) \cdot \frac{k}{(k-1)}.$$

This results in the processing overhead illustrated in Fig. 7, which shows the factor of additional aggregation steps for AHM (p_{AHM}/p_{MTO}). Since the complexity of the aggregation depends on the number of streams that need to be aggregated, the total number of stream aggregations are counted. It shows that for small node degrees $(k = 2)$ potentially twice as many aggregation computations are necessary. Higher degree trees have less of a computational overhead. Considering that higher degree trees also require fewer transmissions, they are a more favorable choice for aggregated hierarchical multicast.

This evaluation shows that there is a tradeoff between bandwidth and computational overhead. AHM significantly reduces the number of messages sent on links, but it increases the amount of computation required up to a factor of two. In general, higher degrees of nodes lead to more favorable configurations.

B. Aggregation Algorithm

For the aggregation algorithm presented above, we evaluate both the computational complexity as well as its correctness.

1) Computational Complexity: Two parts of the aggregation algorithm contribute to its computational complexity. One is the per packet processing and the other is the aggregation of the stored packets. As for the per packet processing, the complexity is constant $[O(1)]$. The packet merging is $O(k)$, since it uses all stored packets. If we do an amortized analysis, though, we can add credit for each of the k received packets, which is used for the $O(k)$ computation. This results in a constant $O(1)$ amortized complexity. Note, that if we have to merge any set of packets with 0_F (due to packet loss or reordering), the amortized analysis does not hold, because no credit was added for null packets. In addition to the complexity of the aggregation, one must consider the complexity of the aggregation function F. It can be expected that for most practical applications, F requires significantly more processing than the buffering of the packets.

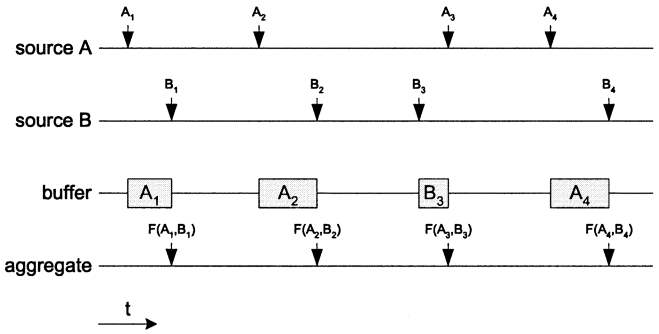

Fig. 8. Aggregation algorithm with no packet loss.

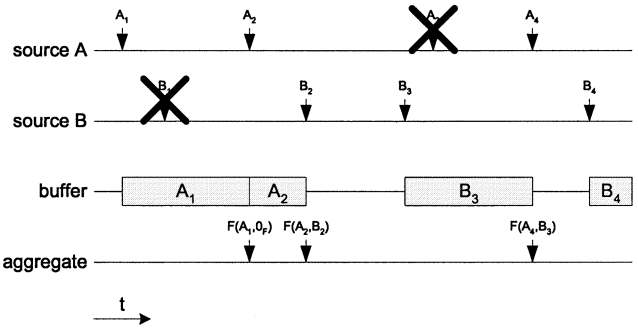

Fig. 9. Aggregation algorithm with packet loss and $\Delta t < t_{\text{timeout}}$.

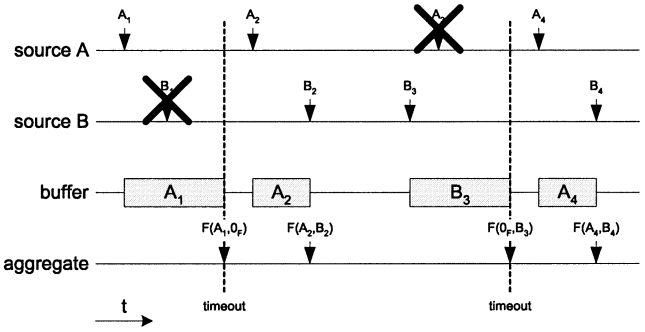

Fig. 10. Aggregation algorithm with packet loss and $\Delta t \geq t_{\text{timeout}}$.

2) Correctness: To show correctness of the aggregation algorithm, we show for different cases that it operates as desired and discuss the effects of jitter. The consideration of jitter is insofar important that it affects the proper operation of the algorithm. The analysis shows, though, that jitter is not increased by the aggregation step and thus, limited to the amount of jitter that is introduced by the data source.

Let p_j^i be the j^{th} packet sent by source s_i. This packet is received by the router at time t_j^i. The interarrival time of packets from one source be Δt. To account for jitter, we define the random variable J that is bound my the maximum jitter $j_{\max}$ $(P[-j_{\max} \leq J \leq j_{\max}] = 1)$. We also assume that $j_{\max} < 1/2 \cdot \Delta t$ to avoid reordering of packets and that the average jitter is zero $(E[J] = 0)$. Thus, the expression for packet arrival times is

$$t_j^i = i \cdot \Delta t_0^i + J.$$

The time for the first packet, t_0^i, can be set arbitrarily.

As for correctness, the objective of the algorithm is to merge one packet from each source and send out the aggregated result. In the case where there is no packet loss, no jitter, and packets from all streams have the same interarrival time, it is easy to see that within a time of Δt the buffer is filled with one packet from each stream (Part I). When the buffer is full, the packets are aggregated and sent out (Part III). If we allow packet loss, the algorithm needs to detect the loss and replace the missing packet with 0_F before merging. With jitter, the algorithm also has to make sure that a delayed packet is not wrongly assumed to be lost. Therefore we give an upper bound on the allowable jitter for which the algorithm still performs correctly. The following four scenarios discuss variations of packet loss for two streams, derive an upper bound for the jitter, and extend the results to three or more streams.

a) No Packet Loss: As shown in Fig. 8, the first packet received from stream A is buffered. As a packet arrives from stream B, both are merged and one aggregated packet is sent out. Due to jitter, the arrival order of packets can change (e.g., from $A-B$ for the first packets to $B-A$ for the third packets in Fig. 8). This does not affect the output of the algorithm, since output is only generated when packets from both streams are available. Even if the order of packets changes constantly (e.g., $A-B-B-A-A-B-B-\cdots$), the algorithm will still work correctly. This assumes, though, that temporally close packets are matched by the aggregation algorithm and patterns, like $A-B-A-A-B$ do not occur.

When three packets from one stream are received in a row without any packets from the other stream (e.g., $A-B-B-B-A$), the middle B packet has to be merged with 0_F, since no packet from the other stream is available. But this only occurs for significant jitter, because three packets have to be received in a shorter time that two packets from another stream can be spaced apart. Thus, we have to have

$$2 \cdot (\Delta t - j_{\max}) > \Delta t + 2 \cdot j_{\max}$$

which gives us an upper bound for j_{max} of $j_{max} < 1/4 \cdot \Delta t$.

b) Packet Loss and Large t_{timeout}: With the possibility of packets getting lost, the setting for the timeout value t_{timeout} becomes important. As with any timer, the goal is to set the timeout no larger than it is necessary to detect packet loss. But it should not be so small that a timeout occurs when packets are just delayed due to jitter. The first observation from this is that maximum time between packets that should be merged together is $\Delta t + 2 \cdot J_{\max}$. Thus, $t_{\text{timeout}} \leq \Delta T + 2 \cdot j_{\max}$.

If the timeout is set to $t_{\text{timeout}} = \Delta t + 2 \cdot j_{\max}$, it almost never times out on packet loss. As shown in Fig. 9, if a packet gets lost, say B_1, most likely the next packet of A (A_2 in this example) arrives before the timer expires and replaces A_1. It can also happen that packets are merged out of order. In Fig. 9, this happens for B_3 and A_4. For some applications, this is an undesired effect and can be avoided by reducing t_{timeout}.

c) Packet Loss and Small $t_{timeout}$: The packet loss scenario for a smaller timeout is shown in Fig. 10. Here, a timeout occurs before A_2 arrives, which causes A_1 to be merged with 0_F and sent. This is the same behavior as in Fig. 9, except that the

TABLE I
WORST-CASE, AVERAGE, AND BEST-CASE DELAY FOR AGGREGATION ALGORITHM

	worst case	average	best case
no packet loss	$\min(\Delta t + 2 \cdot j_{max}, t_{timeout})$	$\min((1 - \frac{1}{k}) \cdot \Delta t, t_{timeout})$	0
packet loss	$\min(\Delta t + 2 \cdot j_{max}, t_{timeout})$	$\min(\Delta t, t_{timeout})$	$\min(\Delta t - 2 \cdot j_{max}, t_{timeout})$

packet is delayed less, because the timeout happens before A_2 arrives. Also, it maintains the order of merged packets for this example. the loss of A_3 causes a timeout and B_3 to be merged with 0_F, not with A_4 as it happened above. A_4 and B_4 are then properly merged together.

If the timeout becomes too small (e.g., $t_{\text{timeout}} < 1/2 \cdot \Delta t$), the algorithm might not function properly, since it times out on almost every packet, merges it with 0_F, and effectively sends twice as many packets. The ideal combination of timeout and jitter is where packets from different streams are close enough together that they can be merged together without packets from the same stream replacing each other. The maximum time between two samples from different streams is $1/2 \cdot \Delta t + 2 \cdot j_{\max}$. The minimum time between two samples of the same stream is $\Delta t - 2 \cdot j_{max}$. Thus, the algorithm works best if

$$\frac{1}{2} \cdot \Delta t + 2 \cdot j_{\max} < \Delta t - 2 \cdot j_{\max}.$$

This leaves an upper bound on j_{max} of $j_{max} < 1/8 \cdot \Delta t$. The timeout setting for this case should be $t_{\text{timeout}} = 1/2 \cdot \Delta t + 2 \cdot j_{\max}$.

d) Three or More Streams: So far we have looked only at the case of two streams that are merged. For a larger number of streams, the above analysis changes slightly. For the lossless case, the worst case pattern becomes $A - B - B - C - C - A - \cdots$ (this pattern also applies for streams with more than three sources). the minimum distance between the first B and the second C is $2 \cdot (\Delta t - 2 \cdot j_{\max})$. the maximum distance between the two As is $\Delta t + 2 \cdot j_{\max}$. Thus, the jitter is limited by

$$2 \cdot (\Delta t - 2 \cdot j_{\max}) > \Delta t + 2 \cdot j_{\max}.$$

This gives a limit of $j_{\max} < 1/6 \cdot \Delta t$ for the jitter. The analysis for the timeout in the packet loss case also changes slightly. The maximum time between a set of samples from n different streams is $(1 - 1/k) \cdot \Delta t + 2 \cdot j_{max}$. The minimum between two samples from the same stream remains $\Delta t - 2 \cdot j_{\max}$. Thus, the bound for $j_{\max}$ is given by

$$\left(1 - \frac{1}{k}\right) \cdot \Delta t + 2 \cdot j_{\max} < \Delta t - 2 \cdot j_{\max}.$$

This solves to $j_{\max} < 1/4k \cdot \Delta t$ with the same timeout of $t_{\text{timeout}} = (1 - 1/k) \cdot \Delta t + 2 \cdot j_{\max}$. This result indicates that the tolerance for jitter decreases as the number of streams increases.

e) Delay and Jitter Propagation: The worst-case, average, and best-case delay introduced by the aggregation algorithm is shown in Table I. the delay from processing of F is not considered here, since it can be assumed to be constant and just added to the expressions in Table I. With a timeout setting of $t_{\text{timeout}} = (1 - 1/k) \cdot \Delta t + 2 \cdot j_{\max}$, as recommended above, the worst case is always bound by the timeout. Also, the average and best-case are bound by the timeout in case of packet loss. This shows that the delay per aggregation stage is roughly limited to Δt for small jitter.

Since it is important to keep a bound on jitter for the correctness of the algorithm, we now look at the jitter that occurs on the output. (Note, that we cannot just subtract the best case delay from the worst case delay to obtain jitter, since these cases assume different offsets between streams.) If no packet loss occurs, the jitter of the output remains $\pm j_{\max}$. This can be easily shown by the following argument. Assume there was no jitter and stream X sends the packet that fills the buffer and triggers the aggregation. Taking jitter into consideration, the aggregation cannot be delayed by more than $j_{\max}$, since no packet can arrive later that $j_{\max}$ after the expected time for the packet from X. Also, the buffer cannot fill earlier than $j_{\max}$ before the expected time for the packet from X. Thus, the total jitter is $\pm j_{\max}$. If we allow packet losses, a similar argument can be made. The aggregation cannot happen any earlier than $j_{\max}$ before the expected time for the packet from X. If a loss occurs, the aggregation might be delayed by t_{timeout}, which is typically $(1 - 1/k) \cdot \Delta t + 2 \cdot j_{\max}$. This introduces significant jitter, but packet loss should only occur infrequently.

f) Other Issues: Finally, we look at a few issues that have been neglected so far. In the above analysis, we assumed each packet contains the same amount of information. This might not be a realistic assumption. If packets contain different amounts of information, we need to buffer them in a way that we can receive several, possibly small data packets from one stream before we start aggregating. For this purpose per-stream ring buffers can be used. If data is received from a stream, it is put in the ring buffer (part I of the algorithm). If the buffer is filled, we treat it like part II of the algorithm. For part III, we change to condition when to start aggregating. We require that the average amount of packet data is available from each stream. Requiring less than that, can lead to a fragmentation effect and the generation of n packets per ΔT.

Another issue is maintaining the order of packets when aggregating. For some applications it is undesirable to merge packets with different "timestamps" (as shown for packets A_4 and B_3 in Fig. 9). For this purpose, we can also use the ring buffers described above. If a packet loss or reordering is detected, we just fill the buffer with 0_F up to where data is available again. When merging, we merge with these 0_Fs instead of the newer data.

C. Measurements

To evaluate the performance of the described algorithm in a real application, we have implemented a prototype of the audio conferencing application described above. The prototype aggregates PCM μ-law encoded data streams and RTP/RTCP [20] as the protocol that communicates control information (e.g., source identifier). Our implementation is limited only insofar that it uses unicast connections between nodes and does not implement all control features. All measurements were performed on

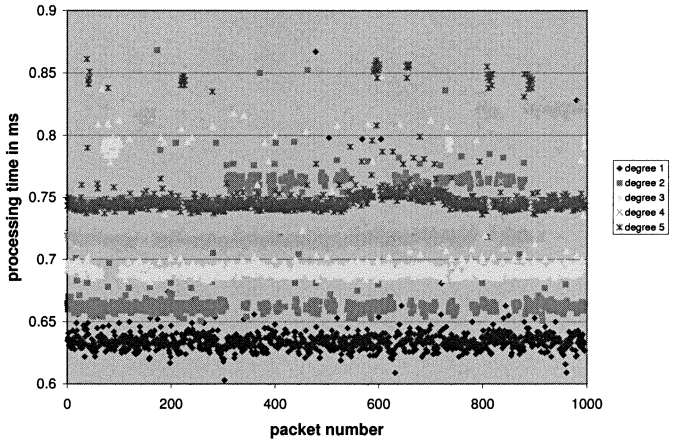

Fig. 11. Processing delay for 400 byte packets and node degrees of $k = 1 \ldots 5$.

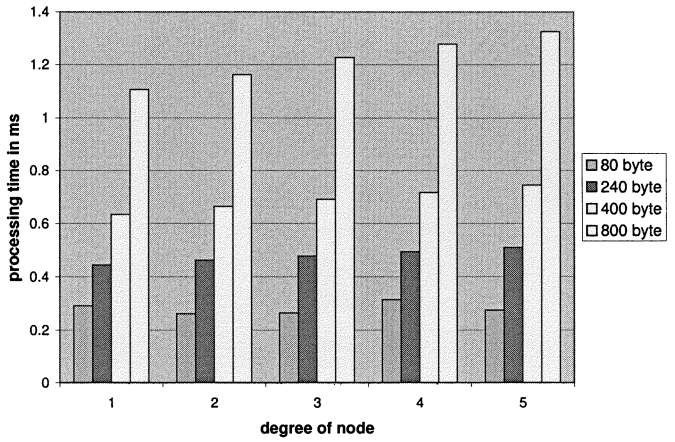

Fig. 12. Median processing delay for varying packet sizes and node degrees.

a heterogeneous set of machines with Pentium processors and NetBSD and Linux operating systems using and environment similar to the active network node [23].

1) Processing Delay: The processing delay is the time between the arrival of the packet that triggers aggregation and the transmission of the aggregated packet. It varies depending on the size of the packet, the aggregation complexity, and the number of packets in the buffer. Fig. 11 shows the processing times for 1000 aggregations for degrees of $k = 1 \ldots 5$ and a packet size of 400 bytes. The processing delay is relatively uniform over the number of packets. Higher degree aggregations require slightly more processing.

To show the effects of varying node degree and packet size, Fig. 12 shows the median delay for packets of sizes 80, 240, 400, and 800 byte. The degree of the node varies again from 1–5. The increase in processing time is proportional to the packet size, as should be expected. Over the range of data, there is also an increase in processing time due to higher node degrees, but it is not quite as significant.

For audio conferencing, a bounded one-way delay of packets is very important to achieve good user satisfaction. As Figs. 11 and 12 show, the delay on a single node depends on the aggregation degree as well as the packet size. This delay of a few milliseconds is small compared to the propagation delay on transcontinental links. Therefore it can be assumed that the

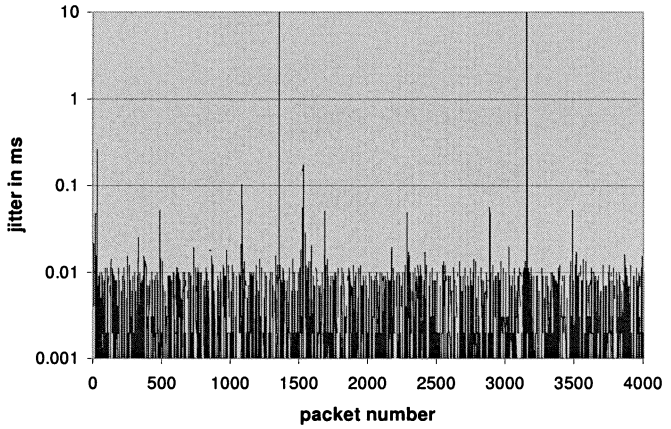

Fig. 13. Jitter for 400 byte packets at source.

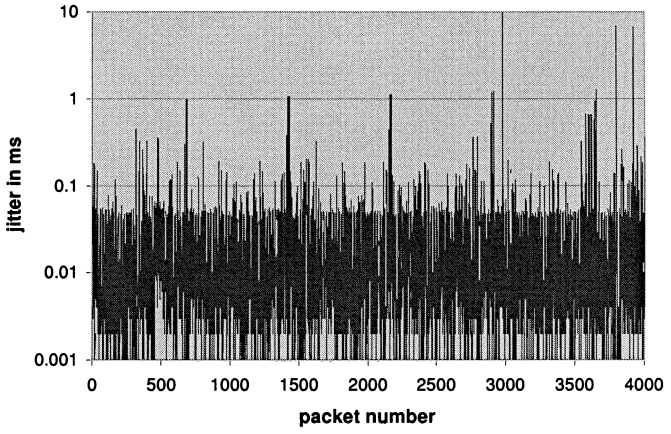

Fig. 14. Jitter for 400 byte packets after four aggregation steps.

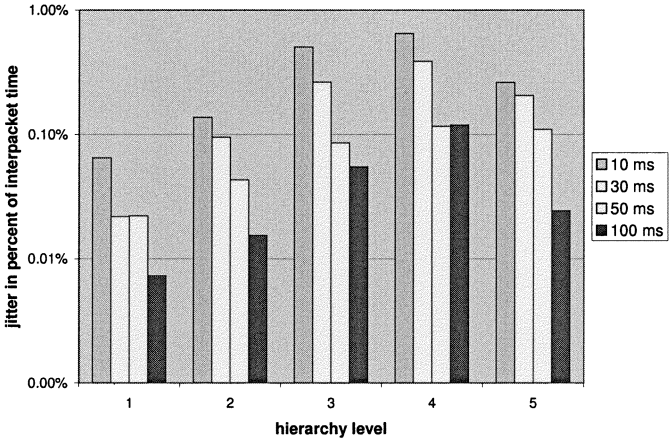

Fig. 15. Jitter for various transmission frequencies and aggregation layers expressed as fraction of Δt.

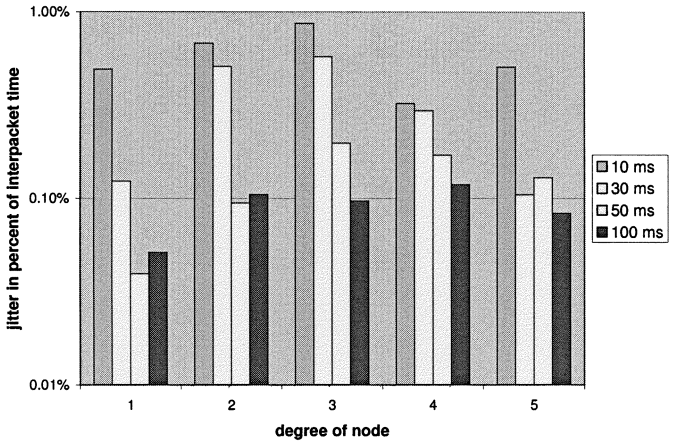

Fig. 16. Jitter for various transmission frequencies and node degrees expressed as fraction of Δt.

overall delay is dominated by the propagation delay when the number of aggregation steps is not excessive.

2) Jitter: The jitter that is introduced by the aggregation affects the correctness of the aggregation and needs to be bound, as discussed in Section VI-B. The jitter for an audio data source is plotted in Fig. 13. It can be seen that the average jitter is limited to 10 μs. Looking at the jitter after traversing five aggregation steps, Fig. 14 shows that it has increased to about 50 μs.

Next, we look at jitter over a range of transmission frequencies, aggregation layers, and node degrees. Fig. 15 shows the jitter as a fraction of the transmission time Δt on different layers of a hierarchy. While jitter increases on higher layers, it is still very small (less then 1%). Additionally, Fig. 16 shows that the jitter is not drastically increasing for larger node degrees. Altogether, jitter does increase on higher levels of the hierarchy and for larger degree nodes, but it remains in the order of 1% of the interpacket time and therefore does not pose any problems.

These measurements indicate that the aggregation algorithm is robust and performs well over a wide range of aggregation levels, node degrees, and packet sizes.

VII. Summary

Aggregated hierarchical multicast is a many-to-many communication scheme that can significantly reduce the amount of data that needs to be transmitted over the network. It is based on aggregating data flows inside the network and presenting a hierarchy of views to the observers. The aggregation is done scalably on programmable network nodes avoiding the need of a central processing facility.

The analysis of AHM shows that the bounds on jitter and delay con be guaranteed for periodically transmitting senders using the algorithm presented in this paper. Our prototype implementation of the audio conferencing application on a programmable router shows that aggregation of packet data is feasible and low jitter can be achieved in a real system.

As pervasive devices become smaller, more numerous and ubiquitous, it will become increasingly important to have a scalable communication paradigm to support such an environment. We believe aggregated hierarchical multicast is an important step in this direction as it demonstrates the power of a programmable network infrastructure and makes the operation of large-scale sensor networks feasible.

References

[1] A. Mainwaring, J. Polastre, R. Szewczyk, and D. Culler, "Wireless Sensor Networks or Habitat Monitoring," Intel Research, Berkeley, CA, Tech. Rep. IBR-TR-02-0006, 2002.

[2] J. M. Rabaey, A. M. Josie, J. L. da Silva Jr., D. Patel, and S. Roundy, "Picoradio supports ad hoc ultra-low power wireless networking," *IEEE Comput.*, vol. 33, pp. 42–48, July 2000.

[3] P. Bhagwat, I. Korpeoglu, C. Bisdikian, M. Naghshineh, and S. K. Tripathi, "BlueSky: A cordless networking solution for palmtop computers," in *Proc. 5th Annual ACM/IEEE Int. Conf. Mobile Computing Networking (MOBICOM)*, Seattle, WA, Aug 1999, pp. 69–76.

[4] E. M. Royer and C.-K. Toh, "A review of current routing protocols for ad hoc wireless networks," *IEEE Pers. Commun.*, vol. 6, no. 2, pp. 46–55, Apr. 1999.

[5] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden, "A survey of active network research," *IEEE Commun. Mag.*, vol. 35, pp. 80–86, Jan 1997.

[6] A. T. Campbell, H. G. De Meer, M. E. Kounavis, K. Miki, J. B. Vincente, and D. Villela, "A survey of programmable networks," *Comput. Commun. Rev.*, vol. 29, no. 2, pp. 7–23, Apr. 1999.

[7] K. Psounis, "Active networks: Applications, security, safety, and architectures," *IEEE Commun. Surveys*, vol. 2, no. I, p. Ql, 1999.

[8] L. Peterson, Ed., "NodeOS Interface Specification," AN Node OS Working Group, Tech. Rep, 2002.

[9] I. Kouvelas, V. Hardman, and J. Crowcroft, "Network adaptive continuous-media applications through self organized transcoding," in *Proc. Network Operating Systems Support Digital Audio Video*, Cambridge, U.K., July 1998.

[10] L.-W. Lehman, S. J. Garland, and D. L. Tennenhouse, "Active reliable multicast," in *Proc. IEEE INFOCOM*, San Francisco, CA, Apr 1998, pp. 581–589.

[11] S. K. Kasera, S. Bhattacharyya, M. Keaton, D. Kiwior, J. Kurose, D. Towsley, and S. Zabele, "Scalable fair reliable multicast using active services," *IEEE Network*, vol. 14, no. 1, pp. 48–57, Jan 2000.

[12] T. Harbaum, A. Speer, R. Wittmann, and M. Zitterbart, "AMnet: Efficient heterogeneous group communication through rapid service creation," in *Proc. Active Middleware Workshop*, Pittsburgh, PA, Aug 2000.

[13] (2002) Intel 1XP2800 Network Processor. Intel Corp. [Online]. Available: http://developer.intel.com/design/network/products/npfamily/ixp2800.htm

[14] (2000) IBM Power Network Processors. IBM Corp. [Online]. Available: http://www.chips.ibm.com/products/wired/communications/network_processors.html

[15] K. L. Calvert, J. Griffioen, B. Mullins, A. Sehgal, and S. Wen, "Concast: Design and implementation of an active network service," *IEEE J. Select. Areas Commun.*, vol. 19, pp. 404–409, Mar. 2001.

[16] X Zhang Jr., K. G. Shin, D. Saha, and D. D. Kandlur, "Scalable flow control for multicasr ABR services in ATM networks," *IEEE/ACM Trans. Networking*, vol. 10, pp. 67–85, Feb. 2002.

[17] M. Radenkovic, C. Greenhalgh, and S. Benford, "Deployment issues for multi-user audio support in CVE's," in *Proc. ACM Symp. Virtual Reality Software Technology*, Hong Kong, Nov. 2002, pp. 179–185.

[18] S. Deering, "Host Extensions for IP Multicasting," Stanford Univ., Stanford, CT, RFC 1112, 1989.

[19] J. Moy, "MOSPF: Analysis and Experience," Network Working Group, RFC 1585, 1994.

[20] H. Schulzrinne, S. Casner, R. Frederick, and Van Jacobsen, "RTP: A Transport Protocol for Real-Time Applications," IETF Network Working Group, RFC 1889, 1996.

[21] P. Mockapetris, "Domain Names—Implementation and Specification," Network Working Group, RFC 1035, 1987.

[22] B. Joy, G. Steele, J. Gosling, and G. Bracha, *The Java Language Specification*, 2nd ed. Norwell, MA: Addison-Wesley, 2000.

[23] D. Decasper, G. Parulkar, S. Choi, J. DeHart, T. Wolf, and B. Plattner, "A scalable, high performance active network node," *IEEE Network*, vol. 31, pp. 8–19, Jan. 1999.

Tilman Wolf received the Diploma degree in informatics from the Universität Stuttgart, Stuttgart, Germany in 1998. He received the M.S. degree in computer science, the M.S. degree in computer engineering, and the D.Sc. degree in computer science in from Washington Unversity, St. Louis, in 1998, 2000, and 2002, respectively.

He is currently Assistant Professor in the Department of Electrical and Computer Engineering at the University of Massachusetts, Amherst. His research interests are advanced computer networks, programmable routers, network processor design, and benchmarking.

Sumi Y. Choi (S'02) received the B.S. degree in mathematics from Yonsei University, Seoul, Korea, in 1994 and the M.S. degree in computer science from Brown University, Providence, RI. She is currently pursuing the Ph.D. degree in computer science and engineering at Washington University, St. Louis.

Her research interests include programmable networks, routing algorithms, resource management, and network design.