Policy-Driven Data Dissemination for Context-Aware Applications

Guanling Chen and David Kotz
Institute of Security Technology Studies
Department of Computer Science, Dartmouth College
Hanover, NH 03755, USA
{glchen, dfk}@cs.dartmouth.edu

Abstract

Context-aware pervasive-computing applications require continuous monitoring of their physical and computational environment to make appropriate adaptation decisions in time. The data streams produced by sensors, however, may overflow the queues on the dissemination path. Traditional flow-control and congestion-control policies either drop data or force the sender to pause. When the data sender is sensing the physical environment, however, a pause is equivalent to dropping data. Instead of arbitrarily dropping data that may contain important events, we present a policy-driven data dissemination service named PACK, based on an overlay-based infrastructure for efficient multicast delivery. PACK enforces application-specified policies that define how to discard or summarize data flows wherever queues overflow on the data path, notably at the mobile hosts where applications often reside. A key contribution of our approach is to uniformly apply the data-stream "packing" abstraction to queue overflow caused by network congestion, slow receivers, and temporary disconnection. We present experimental results and a detailed application study of the PACK service.

1 Introduction

Adaptive pervasive-computing applications rely on awareness of their execution context, such as physical location, network condition, and state of their peers. To obtain such information, applications typically need to continuously monitor data streams produced by distributed sensors so that they can react to events quickly. Due to the potential large data volume, however, it is necessary to control the data flow from sender (sensor) to receiver (application) so that the data rate does not exceed the receiver's consumption rate or exceed the network's transmission capability. We must also support *disconnected operation* for mobile clients, whether senders or receivers.

All three situations involve queues: *flow control* prevents overflow of a receiver's queue (such as by informing the sender how much more data the buffer can hold); *congestion control* uses certain mechanisms to notify the sender either explicitly or implicitly when queues of intermediate network elements are full; disconnection causes the queue at the sending side of the broken link grow until the link is restored. In each case, it is necessary to have a limit on the queue size because physical memory is finite and because latency may grow unacceptably large as the queue builds up.

If a queue becomes full, it must either drop the new data (best effort) or tell the sender to pause (reliable delivery). UDP and TCP are transport protocols representing such approaches. From an application's point of view, however, the best-effort approach may lose important events and pausing a sender may incur larger latency. If the sender is a sensor, asking it to pause is equivalent to dropping data arbitrarily due to its limited buffering capability. It makes sense, then, for the queue to drop some data when it becomes full, but only according to the application's semantics.

We observe that many context-aware applications are loss-tolerant, which means that they can adapt to occasional data loss and often do not require exact data delivery. Many multimedia applications are loss tolerant in nature, but we focus on non-multimedia applications in this paper. For instance, an application that maintains a room's temperature for current user(s) will likely be able to function correctly even if it misses several sensor readings. Similarly, an ActiveMap application can adapt to loss of location-change updates by fading the object at its current location as a function of time since the last update [7]. One reason these applications are able to tolerate data delivery loss is that they are designed to cope with unreliable sensors, which also may lead to data loss and inaccuracy.

In this paper, we present a data-dissemination service, PACK, that allows applications to specify data-reduction policies. These policies contain customized strategies for discarding or summarizing portions of a data stream in case of queue overflow. The summaries of dropped data serve as a hint to the receiver about the current queueing condition; the receiver may adapt by, for example, choosing a different data source or using a faster algorithm to keep up with the arriving data. Unlike congestion control in the network layer, which makes decisions based on opaque packets since it does not recognize the boundaries of application-level data objects, the PACK policies work at the granularity of Application Data Units (ADU) [4], which in this paper we call *events*. Since PACK works with events that follow a common structure, PACK can get the *values* inside the event object enabling a much more flexible and expressive policy space for receivers.

Our PACK service presents three contributions. First, it enables customized data-reduction policies so context-aware applications can trade data completeness for fresh data, low latency, and semantically meaningful data. Second, it employs an overlay infrastructure to efficiently multicast data and to support mobile data endpoints for temporary disconnection and hand-off. Finally, it provides an adaptation mechanism so receivers may react to current queueing conditions.

The rest of the paper is organized as follows. We present the data dissemination mechanism over the PACK overlay in Section 2. We then show the policy specifications in Section 3 and our queue-reduction technique in Section 4. The experimental results and application studies are presented in Section 5. Finally, we discuss related work in Section 6 and conclude in Section 7.

2 Data dissemination

An overlay-based infrastructure gives us a fully-distributed and self-organized system. We can build a multicast facility with an overlay for efficient data dissemination even if IP multicast is not available [3]. An application-level overlay also permits incremental deployment of customized functionality, such as PACK operations, without modifying networking protocol stacks in the operating systems.

The PACK service consists of a set of PACK nodes and some PACK clients. PACK *nodes* are functionally equivalent and peer with each other to form a service overlay using an application-level peer-to-peer routing protocol. A PACK *client* is not part of the PACK overlay, because we expect many clients will be mobile devices with limited capability and bandwidth. The client uses a a library to explicitly attach to a PACK node, which acts as the *proxy* for that client.

A data *endpoint* on the client is either a *sender* or a *receiver*. A sender produces a sequence of *events* carrying application data, such as sensor readings. To receive an event stream, the receiver *subscribes* to some sender. The sender client, intermediate forwarding PACK nodes, and the receiver client form a dissemination path for that subscription. We allow many receivers to subscribe to a single sender, or a single receiver to subscribe to multiple senders.

Conceptually there is a FIFO *queue* on each host of the path for a particular subscription, temporarily holding the events in transition. A *buffer* consists of multiple queues for multiple subscriptions (Section 4). Receivers specify a data-reduction policy (or simply *policy*) that is deployed to their queues on any host of the path. The policy defines how to shorten the queue when it becomes full, by discarding and summarizing certain events according to the applications' needs (Section 3).

Each endpoint and overlay node has a unique numeric key randomly chosen from the same key space. The subscriptions of a sender S are managed by S's *root* node, whose key is closest to S's key among all nodes. Note that a sender's root is not necessarily the same node as S's proxy. All the overlay nodes are functionally equivalent and may play several roles simultaneously.

As shown in Figure 1, a data dissemination path is constructed as follows: the client hosting a sender S forwards all its published events to the sender's root SR via the proxy SP; then the events are multicasted to the proxy nodes of all subscribing receivers RP, hopping through a set of intermediate forwarding nodes MF; finally the events are forwarded to the clients hosting each receiver R. Note the SR, set of intermediate forwarding MFs, and all subscribing RPs form an application-level multicast (ALM) tree for the event stream published by S. A policy propagates in the overlay with the receiver's subscription request so the policy embeds in every node of the dissemination path, and multiple receivers' requests incrementally construct the multicast tree. Castro et al. present and compare some of protocols that can be used to build ALM on peer-to-peer overlays [3].

Due to the lack of space, we only briefly discuss how PACK handles host mobility here. PACK starts to buffer data on the sender client for all receivers if it is detached from the proxy, or on the receiver proxy if some receiver client has detached. If the receiver client departs, PACK removes its subscription and all accumulated queues. If a receiver R re-attaches to a proxy RP' different than its previous proxy RP, it first asks RP' to join the multicast tree and cancel its subscription at RP. Then R asks RP for all the buffered data before requesting data from RP'. A sequence number in the data units is used to prevent duplicate delivery.

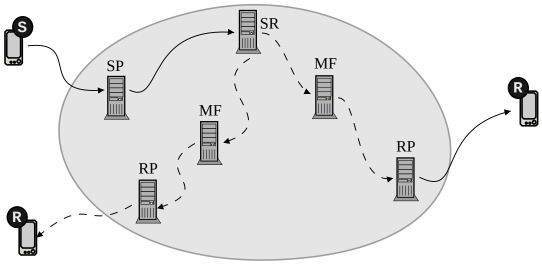

Figure 1. Multiple data dissemination paths in the PACK overlay form an application-level multicast tree.

3 Data-reduction policy

A policy defines an ordered list of *filters*, which reflects the receiver's willingness to drop events under increasingly desperate overflow conditions. Given filters 1 to n and an event queue for a subscription to be reduced, PACK determines k so the events in the queue pass through filters 1 to k. Thus the higher k is, the more filters are applied and the more events will be dropped. The algorithm to determine k, given the current queueing condition, is separate from policies (Section 4).

An event contains a list of *attributes*, and a filter determines what events to keep and what to drop given an event queue as input, based on events' attribute values. The filters are independent, do not communicate with each other, and do not retain or share state. Optionally a policy may also specify how to summarize dropped events using a *digester*. The result of summarization is a *digest* event appended to the output event stream. Thus an event queue may contain a mixed list of events and digests. The digests give rough feedback to the receiver about which events were dropped, and also serve as a queue overflow indication; the receiving application may take action such as switching to different sensors or using a faster algorithm to consume events.

We show an example policy in Figure 2 using XML syntax (although it is not the only possible specification language). First the policy specifies that all the filters apply on the attribute with tag "PulseRate". It is also possible to specify a different attribute for each filter. All dropped events are summarized to inform receivers about the maximum and minimum PulseRate values of all dropped events. The first filter drops events whose pulse rate has not changed much since the previous event; the second filter drops all events that have pulse rate inside of a "normal" range (since they are less important); and the last filter simply keeps the latest 10 events and drops everything else. In urgent queueing situations, all three filters are applied in sequence to each event in the queue.

```
<policy attribute="PulseRate">
 <summary>
  <digester name="MAX">
  <digester name="MIN">
 </summary>
 <level>
  <filter name="DELTA">
   <para name="change" value="5"/>
  </filter>
 </level>
 <level>
  <filter name="WITHIN">
   <para name="low" value="50"/>
   <para name="high" value="100"/>
  </filter>
 </level>
 <level>
  <filter name="LATEST">
   <para name="window" value="10"/>
  </filter>
 </level>
</policy>
```

Figure 2. An example of PACK policy that is applied to monitor a patient's pulse rate.

Currently PACK supports basic comparison-based filters, such as GT ($>$), GE ($\geq$), EQ ($=$), NE ($\neq$), LT ($<$), LE ($\leq$), MATCH ($=\sim$), and WITHIN ($[k1,\ k2]$). We also provide some set-based operators such as INSET ($\in$), CONTAIN ($\ni$), SUBSET ($\subset$), SUPSET ($\supset$), FIRST (retains only the first value in a set), and LAST (retains only the last value in a set). More advanced filters include UNIQ (remove adjacent duplicates), GUNIQ (remove all duplicates), DELTA (remove values not changed much), LATEST (keep only last N events), EVERY (keep only every N events), and RANDOM (randomly throw away a certain fraction of events). The digesters for summarization are MAX, MIN, COUNT, and SUM, which have semantics as their name suggests.

4 Buffer management

A PACK host puts all events, either from a local sender or from the network, into its buffer waiting to be consumed by a local receiver or transmitted to the next host on the path. A buffer is a data structure containing multiple subscriptions, or queues for receivers. We distinguish two kinds of buffers: one is the *local buffer* for endpoints on the same host, and the other is the *remote buffer* containing events to be transmitted to clients or some overlay node. Events in a local buffer are consumed locally by the receivers' event handlers, while the events in a remote buffer are transmitted across a network link. While there might be multiple endpoints on

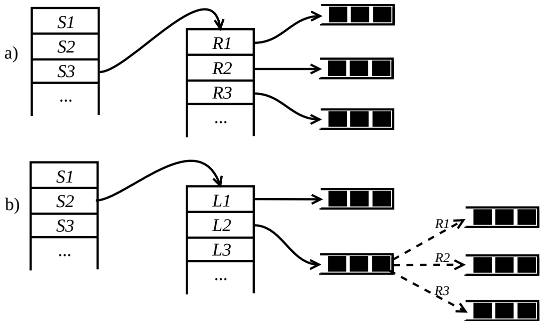

Figure 3. Two-level indexing structure of local and remote buffers.

a client, there is only one local buffer for all resident endpoints and one remote buffer for all destinations.

Both local and remote buffers adopt a two-level indexing structure (shown in Figure 3), where the first index is the sender's key. The local buffer on a client uses the receiver's key as the second index, while a remote buffer uses link address as the second index. An entry for a given link address means there is at least one receiver subscribing to the corresponding sender across that (overlay) link. The two indexes in a local buffer point to a queue for a single receiver. On the other hand, the two indexes in a remote buffer point to a shared queue for all receivers across the same link under normal conditions. As the shared queue reaches its limit, a private queue is created for each receiver and packed using its individual policy.

4.1 Queue reduction

Each queue in a buffer has a limited size and may overflow if its consumption rate is slower than the event arrival rate. Whenever a new event arrives to a full queue, PACK will trigger its PACK policy to reduce the number of events in the queue. For a local buffer, this operation is straightforward, since the second index of the buffer points to a single queue with an individual receiver. The second index of a remote buffer, however, is the link address that points to a queue shared by several receivers over that link. When PACK decides to pack a shared queue, it runs all the events in the queue through each receiver's policy, placing each policy's output in a private queue for that receiver. Note all the event duplication is based on references, not object instances. Figure 3 shows private queues in the lower right.

All newly arrived events are added to the shared queue, which is now empty. The buffer's consumer thread always pulls events from the private queues first and uses the shared queue when all private queues are empty. It is possible that another pack operation is necessary if the shared queue fills up and adds more events to private queues before they are completely drained.

Note a queue may be associated with multiple policies from receivers subscribed to the same sender. During queue overflow, all policies will be executed and the results are kept separated to avoid conflicts. This means that the amount of buffer state increases as the number of policies increases, posing a potential scalability limitation on PACK buffer and preventing a wide-area deployment with hundreds of thousands receivers. We are willing to pay this price to have expressive policies since most of our current applications are targeted at a campus-wide deployment with a limited number of subscribers for individual data sources. It is possible, however, to increase scalability by limiting the policy flexibility [2].

4.2 Ladder algorithm

When packing an event queue is necessary, PACK must determine which filters to apply. Packing with too many filters may unnecessarily drop many important events. On the other hand, packing with too few filters may not drop enough events, and the time spent packing may exceed the time saved processing or transmitting events. Unfortunately there is no straightforward algorithm for this choice, because there are many dynamic factors to consider, such as the event arrival rate, current network congestion, the filter drop ratio (which depends on values in events), and the receiver consumption rate.

PACK employs a heuristic adaptive approach in which each queue is assigned a specific filtering level k (initially one). Once k is determined given a packing request, all events in the queue pass through filters 1 to k in sequence. The heuristic changes the filtering level up or down one step at a time (like climbing up and down a ladder), based on the observed history and current value of a single metric. We define that metric, the *turnaround time* t, to be the amount of time between the current packing request and the most recent pack operation (at a particular level l). The rationale is that changes in t_l capture most of the above dynamic factors. An increase in t_l is due to a slowdown in the event arrival rate, an increase in the departure rate, or an increase in the drop rate of filters up to level l, all suggesting that it may be safe to move down one level and reduce the number of dropped events. A decrease of t_l indicates changes in the opposite direction and suggests moving up one level to throw out more events.

PACK keeps historical turnaround time of all levels, t_l, smoothed using a low-pass filter with parameter $\alpha = 0.1$ (empirically derived) from an observation $\hat{t}_l$:

$$t_l = (1 - \alpha)\hat{t}_l + \alpha t_l \; .$$

We define the change ratio of the turnaround time at a particular level l as:

$$\delta_l = (\hat{t}_l - t_l)/t_l \ .$$

To respond to a current event-reduction request, PACK chooses to move down one filtering level to $l-1$ if δ_l exceeds a positive threshold (0.1), or to move up one level to $l+1$ if δ_l exceeds a negative threshold (-0.1). Otherwise, PACK uses the previous level.

5 Implementation and evaluation

Our implementation is based on Java SDK 1.4.1. We chose Pastry [10] as the overlay routing protocol, but PACK uses its own TCP transport service to disseminate events rather than Pastry's transport library, which has a mixed UDP/TCP mode and its own internal message queues. We used Scribe [3] to maintain application-level multicast trees for PACK to populate the subscription policies.

We first present some experimental results using the Emulab testbed at Utah, in which we focused on measuring the performance of the PACK buffers inside the infrastructure. Next we give an application study of the PACK buffer on a client that tracked a large number of wireless devices on campus.

5.1 Queueing tradeoff

To measure the queueing behavior when a policy is triggered, we used Emulab to set up two hosts connected by a 50Kbps network link. We placed a single receiver on one host, and a single sender and an overlay node on the other. The sender published an event every 30ms, and the events accumulated at the overlay node due to the slow link to the receiver. We compared two approaches to drop events when the queue fills: one is to drop the new event, simulating "drop-tail" behavior, the other is to use a three-filter PACK policy, each filter randomly throwing out events (10%, 25%, and 50% respectively). We show the results in Figure 4. In all the tests we turned off the just-in-time compiler and garbage collector in the Java VM.

Figure 4(a) shows the latency perceived by the receiver. After the buffer filled up, events in the DropTail queue had a (nearly constant) high latency because each event had to go through the full length of the queue before transmission. On the other hand, events in the queue managed by the PACK policy exhibited lower average latency because events were pulled out of the middle of the queue, so other events had less distance to travel. From these results it is clear that PACK reduced latency by dropping data according to application's semantics, and it is desirable for applications to use filters that are more likely to drop events in the middle (such as EVERY, RANDOM, GUNIQ) rather than at the tail.

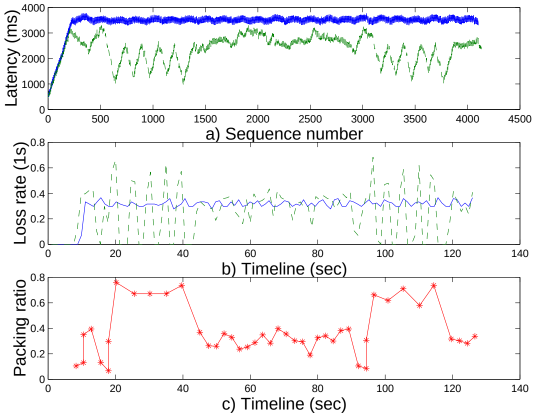

Figure 4. Comparison of queueing behavior using DropTail (solid line) and a three-level PACK policy (dashed line).

Figure 4(b) plots a running sequence of the event loss rate for each 1 second window at the receiver. We see that the DropTail queue's loss rate was about 30% because the arrival rate was one third more than the bottleneck link could handle, and after the queue filled it was always saturated. The loss rate of PACK was high during intervals when the queue was packed, and zero in intervals when the queue was not packed. The loss rate depended on which level pack operation was performed. Figure 4(c) shows a trace from the overlay node denoting when the queue was packed and what fraction of events were dropped. It shows that most pack operations were performed at the second level, dropping events at rate of $0.1 + 0.9 * 0.25 = 0.325$, which fit well with this event flow because the arrival rate was one third higher than the consumption rate (link bandwidth). Our heuristic algorithm worked reasonable well, although the filtering level varied despite the steady publication rate. The reason is that the RANDOM filter dropped varying amounts of events and our ladder algorithm adapted to longer or shorter inter-packing intervals by adjusting the filtering level.

5.2 Application study

As an example application, we use PACK to monitor a campus-wide wireless network. Our campus is covered by more than 550 802.11b access points (APs), each configured to send its syslog messages to a computer in our lab. We run a data source on that host to parse the raw messages into a more structured representation and to publish a continuous event stream. By subscribing to this syslog source, applications can be notified when a client associates with an AP, roams within the network, or leaves the network.

One of our goals is to provide an IP-based location

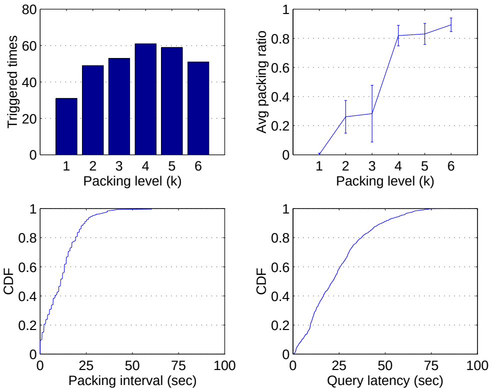

Figure 5. Statistics derived from an one-hour trace collected by the MAC/IP locator.

service: given a wireless IP address, the service can identify the AP where the device is currently associated. This enables us to deploy location-based applications, often without modifying legacy software. For instance, we modified an open source Web proxy so it can push location-oriented content to any requesting Web browser on wireless devices based on the IP address in the HTTP header. Currently we insert information about the building as a text bar on top of the client requested page.

To provide this kind of service, a locator subscribes to the syslog source and monitors all devices' association with the network. The association message contains the device's MAC address and associated AP name, but does not always include the IP address of that device. In such cases, the locator queries the AP for the IP address of its associated clients using a HTTP-based interface (SNMP is another choice, but appears to be slower). The query takes from hundreds of milliseconds to dozens of seconds, depending on the AP's current load and configuration. We also do not permit more than one query in 30 seconds to the same AP so our queries do not pose too much overhead over normal traffic. As a result, we frequently find that the locator falls behind the syslog event stream, considering the large wireless population we have.

We focus our discussion on the subscription made by the locator to the syslog source, where the events tend to overflow the receiver's queue RB. The locator uses a PACK policy consisting of six filters (we skip their description due to space limitation). We collected the PACK trace for an hour-long run and Figure 5 shows some basic statistics.

The upper-left plot presents the distribution of the filtering levels triggered by the PACK service. All filtering levels were triggered, varying from 31 times to 61 times, out of 304 pack operations. The upper-right plot shows that the filters had a wide variety of packing ratios over that one-hour load. It seemed that the filter 2 and 4 discarded most of the events while filters 1, 3 and 5 did not help much. This suggests strongly that an application programmer should study the workload carefully to configure efficient policies. The lower-left plot indicates that PACK triggered the policy rather frequently, with the median interval approximately 11 seconds. The lower-right plot shows the latency distribution, derived from the time the AP query is resolved and the timestamp in the original syslog event. Although we set the connection timeout to be 30 seconds for each query, the longest delay to return a query was 84 seconds; some AP was under heavy load and slow to return results even after the connection was established.

6 Related work

Traditional congestion and flow-control protocols concern both unicast and multicast. They are typically transparent to applications and provide semantics such as reliable in-order data transport. When computational and network resources are limited, these protocols have to either regulate the sender's rate or disconnect the slow receivers [5, 9]. The usual alternative, UDP/IP, has no guarantees about delivery or ordering, and forces applications to tolerate any and all loss, end to end. Our goal, on the other hand, is to trade reliability for quicker data delivery and service continuity for loss-tolerant applications. Our PACK service applies to data streams with a particular structure. This loss of generality, however, enables PACK to enforce receiver-specified policies. The PACK protocol does not prevent or bound the amount of congestion, which is also dependent on cross traffic. But with an appropriate customized policy, a receiver is able to get critical data or summary information during congestion. For many applications this outcome is better than a strict reliable service (TCP) or a random-loss (UDP) service.

Performing application-specific computation, including filtering, inside networks is not a new idea. In particular, it is possible to implement our PACK service using a general Active Network (AN) framework [12]. We, however, chose an overlay network that is easier to deploy and has explicit support for multicast, mobility, and data reduction. Bhattacharjee and others propose to manage congestion by dropping data units based on source-attached policies [1]. Receiver-driven layered multicast (RLM) [6], actively detects network congestion and finds the best multicast group (layer) to which the multimedia application should join. Pasquale et al. put sink-supplied filters as close to the audio/video source as possible to save network bandwidth [8]. Our work, however, aims at broader categories of applications and has to support sink-customized policies since the source typically cannot predict how the sinks want to manipulate the sensor data. PACK policies thus need to be more expressive than the filtering operations on multimedia streams.

Researchers in the database community provide a query-oriented view on continuous stream processing. One of the focus is to formally define a SQL-like stream-manipulation language, which has the potential to replace PACK's current "ad hoc" XML-based interface. In particular, the Aurora system reduces the load by dynamically injecting data-drop operators in a query network [11]. Choosing where to put the dropper and how much to drop is based on the "QoS graph" specified by applications. Aurora assumes a complete knowledge of the query network and uses a pre-generated table of drop locations as the search space. The QoS function provides quantitative feedback when dropping data while PACK allows explicit summarization of dropped events.

7 Conclusion and Future Work

We present a novel approach to solve the queue overflow problem using application-specified policies, taking advantage of the observation that many context-aware pervasive-computing applications are loss-tolerant in nature. Our PACK service enforces data-reduction policies throughout the data path when queues overflow, caused by network congestion, slow receivers, or temporary disconnection. Our sink-based approach and the expressiveness of PACK policies poses a scalability limitation, but provides more fine-grained results tailed to applications' needs as a trade off. Our experimental results show that PACK policies also reduce average delivery latency for fast data streams, and our application study shows that the PACK service works well in practice.

We plan to evaluate PACK service in larger-scale settings, in particular, to investigate how policy-driven data reduction affects the fairness across multiple subscriptions with intersected dissemination paths. PACK currently assumes static policies, and it is another research challenge how to allow and enforce dynamic updates of policy throughout the data path while preserving application semantics. It would also be interesting to see whether PACK can dynamically (re)-configure the appropriate filters to minimize a programmer's effort to specify filters in exact sequence. Finally, we plan to extend this policy-driven data reduction approach to an infrastructure-free environment, such as an ad hoc network.

References

[1] S. Bhattacharjee, K. L. Calvert, and E. W. Zegura. An architecture for active networking. In *Proceedings of High Performance Networking (HPN'97)*, April 1997.

[2] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an Internet-scale event notification service. In *Proceedings of the 19th Annual Symposium on Principles of Distributed Computing*, pages 219–227, Portland, OR, 2000.

[3] M. Castro, M. Jones, A.-M. Kermarrec, A. Rowstron, M. Theimer, H. Wang, and A. Wolman. An evaluation of scalable application-level multicast built using peer-to-peer overlays. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 1510–1520, San Francisco, CA, 2003.

[4] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the 1990 ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 200–208, Philadelphia, PA, 1990.

[5] V. Jacobson. Congestion avoidance and control. In *Proceedings of the Symposium on Communications Architectures and Protocols*, pages 314–329, Stanford, CA, 1988.

[6] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven layered multicast. In *Proceedings of the 1996 ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 117–130, Palo Alto, CA, 1996.

[7] J. F. McCarthy and E. S. Meidel. ActiveMap: A Visualization Tool for Location Awareness to Support Informal Interactions. In *Proceedings of the First International Symposium on Handheld and Ubiquitous Computing*, pages 158–170, Karlsruhe, Germany, September 1999. Springer-Verlag.

[8] J. C. Pasquale, G. C. Polyzos, E. W. Anderson, and V. P. Kompella. Filter propagation in dissemination trees: Trading off bandwidth and processing in continuous media networks. In *Proceedings of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 93)*, pages 259–268, 1993.

[9] S. Pingali, D. Towsley, and J. F. Kurose. A comparison of sender-initiated and receiver-initiated reliable multicast protocols. In *Proceedings of the 1994 ACM Conference on Measurement and Modeling of Computer Systems*, pages 221–230, Nashville, TN, 1994.

[10] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the 2001 International Middleware Conference*, pages 329–350, Heidelberg, Germany, November 2001.

[11] N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 309–320, Berlin, Germany, September 2003.

[12] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications*, 35(1):80–86, January 1997.