2-2004

Application-Controlled Loss-Tolerant Data Dissemination

Guanling Chen
Dartmouth College

David Kotz
Dartmouth College, David.F.Kotz@Dartmouth.EDU

Application-Controlled Loss-Tolerant Data Dissemination

Guanling Chen and David Kotz
Department of Computer Science
Dartmouth College
Hanover, NH, USA 03755

Dartmouth Computer Science Technical Report TR2004-488

Abstract

Reactive or proactive mobile applications require continuous monitoring of their physical and computational environment to make appropriate decisions in time. These applications need to monitor data streams produced by sensors and react to changes. When mobile sensors and applications are connected by low-bandwidth wireless networks, sensor data rates may overwhelm the capacity of network links or of the applications. In traditional networks and distributed systems, 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. Arbitrary data drops are not necessarily acceptable to the reactive mobile applications receiving sensor data. Data distribution systems must support application-specific policies that selectively drop data objects when network or application buffers overflow.

In this paper we present a data-dissemination service, PACK, which allows applications to specify customized data-reduction policies. These policies define how to discard or summarize data flows wherever buffers overflow on the dissemination path, notably at the mobile hosts where applications often reside. The PACK service provides an overlay infrastructure to support mobile data sources and sinks, using application-specific data-reduction policies where necessary along the data path. We uniformly apply the data-stream "packing" abstraction to buffer overflow caused by network congestion, slow receivers, and the temporary disconnections caused by end-host mobility. We demonstrate the effectiveness of our approach with an application example and experimental measurements.

1 Introduction

Adaptive mobile applications rely on awareness of their execution context, such as physical location, network condition, and state of their peers. To obtain such informa-tion, applications typically need to continuously monitor data streams produced by 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, (either senders or receivers).

All three situations involve buffers: *flow control* prevents overflow of receiver's buffer (such as by informing sender how much more data the buffer can hold); *congestion control* uses certain mechanisms to notify the sender either explicitly or implicitly when buffers of intermediate network elements are full; disconnection causes the buffer 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 buffer size because physical memory is finite and because latency may grow unacceptably large if data sits in the queue waiting to be processed.

As a buffer becomes full, it is inevitable that the buffer must drop some data or tell the sender to pause. As one example, consider TCP/IP. A TCP receiver explicitly informs the sender about the available space of its receiving buffer so the TCP sender adjusts its sending window accordingly to prevent overflow. IP routers drop packets as an implicit notification to TCP senders about congestion in their buffers; again, the sender should regulate its sending rate. Traditional transports like UDP/IP or TCP/IP, exposes little control to the endpoints; they cannot recognize the application's data semantics and priorities. Thus the application either gets best-effort data delivery (data dropped arbitrarily when buffers overflow), or end-to-end reliable transmission that may require the sender to slow down or pause. When the data sender is sensing the physical environment, however, a pause is equivalent to dropping data because a paused sensor no longer collects data.

We argue that we can take advantage of the middle ground. Some applications do not require reliable trans-

fer but need some control on what data to drop when a buffer reaches limit. These "loss-tolerant" applications are adaptive to informed or controlled data loss by, for instance, degrading accuracy or other performance aspects. The benefits of such strategic data reduction are: 1) it is friendly to "unstoppable" data sources such as sensors with limited buffering capability; 2) the reduced data stream preserves application-specific semantics; and 3) the transmission latency is kept low by dropping less-important data.

There are many examples of loss-tolerant applications in multimedia. We are mainly interested in non-multimedia applications. Consider a disaster scenario where victims and responders carry devices that report their vital signs and current location. As the monitoring device carried by a local commander (or its wireless network condition) becomes incapable of handling the aggregated data volume, data senders on sensing or monitoring devices may want to drop less-urgent data such as pulse rates in the "normal" range, duplicated values, and those that have not changed much since previous report. The applications may produce less accurate results without a complete complete sensor record, but the result may still be useful if application semantics influence the choice of data to drop.

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 buffer overflow. The summaries of dropped data serve as a hint to the receiver about the current buffering 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) [9], which in this paper we call *events*. Since PACK is able to separate the 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.

In addition to the policies at the end hosts, it is necessary to install data-reduction policies on the buffers of the intermediate forwarding nodes, so they can be triggered closer to congested links or disconnected clients. It is not practical and may not be efficient to inject PACK functionalities into a widely deployed protocol stack (such as IP). Instead, we implement packing policies at the application layer using the buffers above the networking stack. We assume forwarding nodes are strategically placed in the infrastructure to form a multicast overlay service capable of executing data-reduction policies at any node.

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

The rest of the paper is organized as follows. We present the data stream structures and policy specifications in Section 2. The overlay architecture, protocols and related issues are discussed in Section 3. We present the unified buffer management technique in Section 4. Then we discuss the system implementation issues (Section 5) and present evaluation results (Section 6). Section 7 demonstrates the effectiveness of our approach with some campus-wide WiFi monitoring applications. We discuss related work in Section 8 and conclude in Section 9.

2 Data-reduction policy

We begin by defining terms. A data *endpoint* is either a *sender* or a *receiver*. A *client* is a non-overlay host that, mobile or not, may host one or more endpoints. A sender produces a data stream, a sequence of events carrying application data, such as sensor readings. We model an event as a list of *attributes*: each contains a *tag* string and a *value* object. Currently we assume that all events from the same sender have same structure, namely, the same set of attribute tags. To receive a data stream, the receiver *subscribes* to some sender. The sender client, intermediate forwarding overlay 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 (we discuss the detail of buffer management in Section 4).

Receivers may attach a data-reduction policy (or simply *policy*) to their queues (on any node of the path), to specify how to shorten the queue when it becomes full, by discarding and summarizing certain events according to applications' needs. Figure 1 shows the overall structure of the PACK service, with two receivers subscribed to the same sender.

PACK puts all events, either from a local sender or from the network, into its internal queue waiting to be consumed by a local receiver or transmitted to next host on the path. If a queue becomes full, PACK triggers its associated policy to examine the events in the queue and
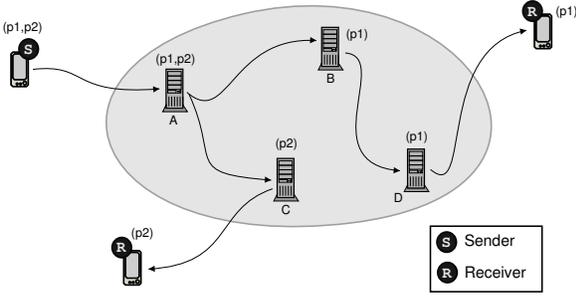
Figure 1: The PACK service consists of a set of overlay nodes, which cooperatively serve clients that host endpoints (either sender or receiver). This example shows two receivers subscribed to the same sender. Each receiver subscribes to the sender with a customized policy (*p*1 or *p*2). Policies are installed on all the hosts along the path from sender to receiver. Nodes on multiple paths contain multiple policies (node A contains both *p*1 and *p*2).

```
<policy attribute="PulseRate">
  <summary>
    <digester name="MEAN">
    <digester name="COUNT">
  </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 data stream. Depending on current buffer fullness, the policy either drops events whose value has not changed much (DELTA), drops events whose value is outside a certain range (WITHIN), or drops all previous events except last 10 ones (LATEST). All dropped events are summarized to compute the number and average value as supplementary information to the receiver.

determine which should be dropped. The policy may also specify how to summarize the dropped events into *digests*, which are placed in the resulting queue as well. On the receiver's client, PACK pulls events or digests from the queue and invokes different interface of the receiver. We now discuss what consists a policy specification and how PACK executes a policy.

Policy specification

A policy defines an ordered list of *filtering levels*, and each level contains a single *filter* or a chain of filters. The list of levels reflects receivers' willingness to drop events under increasingly desperate overflow conditions: more events should be dropped by filters at higher levels. the lower levels. The policy may contain arbitrary number of levels. Given an event queue to be reduced, PACK determines which level to use and then passes the queue through all the filters defined up to and including that decided level, starting from the lowest level.

A filter is instantiated with application-defined parameters and determines what events to keep and what to drop given an event queue as input. The filters are independent, do not communicate with each other, and do not retain or share state. Since an event may contain several attributes, the filter typically requires a parameter indicating which attribute to apply the filtering.

Filters drop some events. Optionally a policy may also specify how to summarize dropped events using a single or chain of *digesters*. The result of summarization, is a *digest* event injected into the event stream. Thus an event queue may contain a mixed set of events and digests. The digests give some rough feedback to the receiver about which events were dropped, and also serve as a buffer

overflow indication; the receiving application may take action such as switching to different sources 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 number and average PulseRate value of all dropped events. The example gives a single filter for each buffering level. The first-level filter drops events whose pulse rate has not changed much since the previous event; the second-level 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 buffering situations, all three filters are applied in sequence to each event in the queue.

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 (re-

tains 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, SUM, and MEAN, which have typical semantics as their name suggests.

As indicated in Figure 2, our approach is to allow applications to compose predefined filters into a customized policy. We could have used a general-purpose language to express more general policies or even more general filters. The trade-off is that as the language gets more powerful and more complex filters are supported, it is more likely that PACK will have more overhead on filter execution and eventually reduce system scalability [5]. Based on our experience so far, many loss-tolerant applications desire simple and straight-forward policies. Thus our strategy is to keep the filters simple and efficient, and to expand the filter repository as necessary.

Policy execution

Due to previous packing operations performed locally or at upper stream hosts, a queue may consists of a sequence of mingled digests (d) and events (e) as follows (the sequence number reflects the order in the queue instead of the original counter at the sender):

$$e_1, e_2, d_3, e_4, \ldots, e_5, d_6, e_7, e_8 .$$

Suppose a policy is executed on this queue and e_2 is to be dropped, then d_3 should be updated using e_2. On the other hand, if all the events between e_1 and e_8 are to be dropped, a new digest should be computed based on dropped events. In particular, d_3 and d_6 should be able to be combined.

Thus the digesters should be "associative" so they can be recursively executed on previous results. Note that, since the same policy exists on every host in a path, that this associativity applies across hosts as well as within a host (when a buffer must be packed again). All the digesters we mentioned above (such as MAX, MIN) satisfy this requirement. If we were to provide a digester that computes the number of unique values in dropped events, then the digests have to carry all the unique values so they can be merged or updated accurately. The number of unique values, however, may be unbounded and defeat the purpose of summarization. Although it is possible to use these digests as packing boundaries (so they do not have to be updated or merged), a queue may end up with many digests with little actual data and reduce the effect of filters applied later.
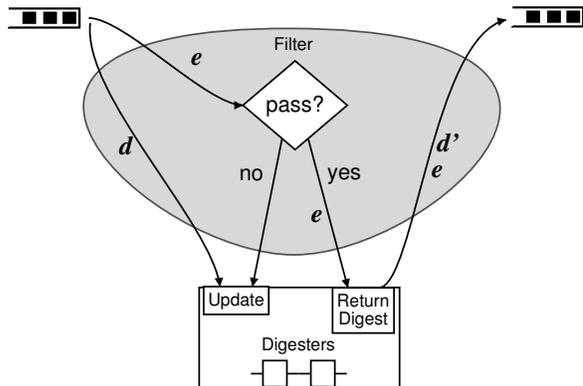

Figure 3: Packing a queue through a single filter. The filter gets an event e or a digest d from input queue in order. If e fails to pass the filter then it is taken by digesters, as are all the digests d from input. Otherwise a new digest d' is computed and put into output queue together with the satisfying e.

When a policy is triggered, PACK takes the input queue and forms a chain of filters up to the filtering level it has decided. PACK feeds the queue to the first filter, passes the resulting queue to next filter, and so on until the last filter. Figure 3 visualizes how a single filter executes the policy. For each event e in input queue, if it fails to pass then it is used by digesters to update current summary state, like previously computed digests d in input queue. If the e passes the filter, then a new digest d' is computed and placed in output queue together with the satisfying event e.

A design alternative is to take one event from the input queue and check it against all filters until it is either fails in middle or passes all. Only after the previous event has already run through all filters, the next event in input queue is admitted and follows the same procedure. Our approach, however, takes the input queue as a whole and feeds it through all filters. We believe the first approach limits what a filter can do since the event pass the filter only once and the filter does not know how many more events are coming. Our PACK filters, however, are able to perform tasks on the whole queue, such as LATEST and GUNIQ. The overhead of two approaches, however, should be comparable since each event has to be checked against all filters in sequence.

It is possible that PACK may not be able to reduce a queue at all even after applying the highest filtering level. The reasons might be that the policy does not apply well to current data values so all filters are not effective, or the link to next host is congested or disconnected and the queue has already been filtered at the highest level. In such cases, PACK drops all the events in queue and applies policy's digesters, or COUNT if the policy does not have one.

3 The service overlay

Strictly speaking, PACK could be implemented only on sending and receiving hosts, without an overlay infrastructure. PACK would only need to manage the input buffer of the receiver and the output buffer of the sender. A service overlay, however, is more attractive as the supporting infrastructure for several reasons: 1) the overlay node provides an ideal proxy for the mobile clients, such as handling subscriptions for sender and buffering data for disconnected receiver; 2) pack operations can be performed closer to where congestion and disconnection happens to improve scalability and responsiveness; 3) an overlay supports application-level multicast in absence of IP multicast to reduce network traffic [7]; and 4) other services can also be provided by the same overlay, such as naming and discovery of data sources [3].

Dissemination path

The clients, which host data endpoints (either senders or receivers), are not part of the PACK overlay. Instead, a client has to explicitly attach to an overlay node to request services for its endpoints. The node to which the client attached acts as the *proxy* for all the endpoints on the client. Each endpoint and overlay node has a unique ID from the same ID space. Each endpoint also has a *root* node, whose ID is closest to that endpoint's ID among all nodes. Note that an endpoint's root is not necessarily the same node as that endpoint's proxy node. All the overlay nodes are functionally equivalent and may play several roles simultaneously.

As shown in Figure 4, 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; the the events are multicasted to the root nodes of all subscribing receivers RR, hopping through a set of intermediate forwarding nodes MF; finally the events are forwarded to the clients hosting each receiver R via its proxy RP. Note the SR, set of intermediate forwarding MFs, and all subscribing RRs form an application-level multicast (ALM) tree for the event stream published by S. Castro et al. present and compare some of protocols that can be used to build ALM on peer-to-peer overlays [7].

This rendezvous-based approach is analogous to ROAM's Internet indirection architecture, which was designed to support host mobility [19]. Our approach, however, requires the client to explicitly attach to a proxy before its endpoints can send or receive data. The reason is that the clients typically need to engage in a soft-state protocol to periodically communicate with some infrastructure host so each other knows the other party's liveness. The frequency of the heartbeat is purposefully high, to improve responsiveness since mobile clients are less stable. The
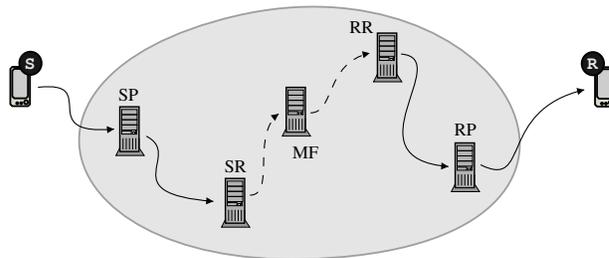

Figure 4: The data dissemination path in the PACK overlay. We show a single sender S and receiver R. The overlay nodes are functional equivalent, and each plays one or more roles at the same time. The SP and RP denote the proxies for the mobile client where S or/and R runs. The proxy is a bridge between the endpoints (S or R) and their root nodes (SR or RR). The root nodes, together with a set of intermediate forwarding nodes MF, form an application-level multicast tree.

client, however, may host many endpoints and it is expensive to repeat the heartbeat to all of the client's endpoints' roots, increasing the demand on computation and bandwidth, both of which might be limited on the mobile clients.

Mobile clients

Mobile clients may experience temporary disconnection caused by weak links or mobility hand-offs. During disconnection, a client may roam and change its network address (network mobility), and it may or may not choose the original proxy when it reconnects (host mobility). A client may voluntarily decide to change proxy if it finds a "better" overlay node, such as one that is closer or has a lighter load. The proxy may also make its own decision to disconnect a client, if proxy is about to shutdown or is too crowded, and force the client to select a different proxy.

Thus the client and its proxy engage in a protocol maintaining the state about each other. A client (thus the endpoints it hosts) may appear in three states to the proxy, *attached*, *detached*, or *departed*. State transitions from *attached* to *detached* are triggered either by an explicit requests or by missing several heartbeat signals. If the client has been detached longer than a timer threshold, the proxy assumes the client has departed (and will not re-attach). The proxy appears to the client in two states: either *attached* or *detached*, and transitions are managed in a way similar to the client state.

PACK starts to buffer data on the sender client for all receivers if it is detached from the proxy, or in the overlay if some receiver client has detached. If the receiver client has departed, PACK removes its subscription and all accumulated queues. Since a client may re-attach to a different proxy, PACK buffers the data for the detached receiver

on its root rather than on its proxy. The proxy explicitly notifies the root for each endpoints in its care When a client re-attaches to a new proxy, it contacts the old proxy first to retrieve any buffered events for its receivers, then it informs the new proxy it is ready to receive new events buffered at the receivers' roots. We discuss the details of buffer management on the dissemination path in the next section.

4 Buffer management

The queues on the data dissemination path may overflow for various reasons. For instance, the output queues of sender S and receiver's root RR grow during client disconnection. The input queue of receiver R grows if it is slow consuming arriving events. Network congestion in the overlay may also cause queue overflow. Instead of dropping the newest events when the queue fills, PACK uses the receiver's policy to discard and summarize portions of the queue.

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 receivers on the clients, 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 a client, there is only one local buffer for all resident receivers and one remote buffer for all senders. On an overlay node, there are several buffers to serve the different roles the node may play, but all are remote buffers. We discuss them in detail later in this section.

Both local and remote buffers adopt a two-level indexing structure (shown in Figure 5), where the first index is the sender's ID. The local buffer on a client uses the receiver's ID 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 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 due to, for instance congestion or disconnection, a private queue is created for each receiver and packed using individual policy.

On each client or overlay node, a dispatcher thread pulls events from the network, adding a reference (pointer to event object) for each event into one or more queues, based on the header of received event. The header contains the sender's ID sid, the receiver's ID rid, and the
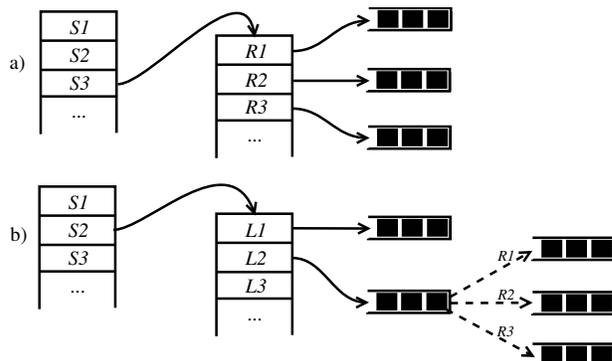

Figure 5: Two-level indexing structure of buffers, both having the sender's ID as first index. An input buffer (a) uses the receiver's ID as second index, while an output buffer (b) uses the destination/link address as the second index. Under normal conditions, all the receivers across the same link share a single queue. If the shared queue reaches its limit, private queues are created for each receiver and their policies are executed to shrink the queues.

destination *toward*. For instance, if *toward*="SP", then the event was just admitted into overlay from some sender client. If rid is empty, then the event is a multicast event destined to all subscribers of this sender. Otherwise, it is a unicast event destined to one specific receiver. We discuss how the event forwarding path is set up in Section 4 and how PACK triggers policy to reduce events in Section 4.

An alternative buffer design is to maintain a single queue for all the received events. Then, when packing is necessary, we need to scan the whole queue to find events for particular subscription to apply its policy. It may use less memory, since our approach may put multiple references of the same event into several queues if there are more than one subscriber. We believe memory usage is not likely to be a significant concern because the events are not replicated. The two-index structure for separate queues, however, gives us greater flexibility to choose queueing and packing policies and reduces a large amount of implementation complexity.

Event forwarding

There are several types of buffers in PACK system and we adopt the following naming convention for ease of discussion. A buffer has a name of capitalized letters ended with "B", and the prefix denotes the destination of the events in the buffer. For instance, a buffer named SRB contains all the events being forwarded to sender's root SR, where the SR depends on the sender's ID and may not be the same host. Buffer RCB contains events destined to receiver client RC.

A client has a local buffer RB (Receiver Buffer) for all

receivers, and a remote buffer SPB (Sender Proxy Buffer) for all senders. The first index of RB contains a list of sid and the second index contains a list of subscribing local rid. The second index of SPB contains only one entry, namely, the proxy's address. When a receiver makes a subscription, an entry is added to RB (extending the first- and second-level indices as necessary). When a client receives an event relayed from its proxy, it enqueues it into the RB. When a client receives a new subscription from its proxy, it adds an entry (extending the indices as necessary) to SPB. When a sender publishes an event, it enqueues it into SPB.

Since an overlay node may play several roles simultaneously, it maintains several remote buffers. One is the Sender Root Buffer (SRB) containing events being relayed to their senders' root. Another is the Receiver Root Buffer (RRB) containing events being forwarding down the multicast tree toward RR. The Receiver Proxy Buffer (RPB) contains events being forwarded to receivers' proxy. And finally, the Receiver Client Buffer (RCB) contains events that should be sent to directly connected clients. There is only one of each these buffers on a single overlay node.

If a receiver R is currently detached, the RPB at its root is notified to "suspend" the queue for R, which means the dispatcher may continue to put events in the queue but the scheduler is not allowed to pull events from the queue and send them to proxy RP. The queue is "resumed" when R is re-attached. Similarly, the SPB on the client may also be suspended and resumed when that sender client is detached or re-attached to its proxy.

A typical event flow is thus to traverse the named buffers as follows (the first and last buffer are on the clients while the middle four are on overlay nodes):

$$\boxed{\text{SPB} \rightarrow \text{SRB} \rightarrow \text{RRB} \rightarrow \text{RPB} \rightarrow \text{RCB} \rightarrow \text{RB}\,.}$$
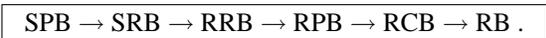
As PACK propagates a receiver's subscription request through overlay nodes in reverse direction, it adds a subscription entry in the two-level index structure, together with a PACK policy, to appropriate remote buffers along the path. The algorithm is described in Procedure 1. Note all the buffers mentioned here are remote buffers and the field *lasthop* is used to set up the second index of the buffer. The field *toward* indicates where to forward the subscription request. The proxy periodically probes the root to maintain a (proxy–root) address mapping so the requests can be forwarded correctly.

As mentioned above, the dispatcher receives events from the network and puts them into appropriate buffers based on the event header. An event header contains a sender ID

Procedure 1 Propagating subscription requests through the overlay nodes. The *request* originates from R whose *toward* is initialized to be RP.

```
 1: (sid, rid, toward, lasthop) ← request
 2: if toward is RP then
 3:     add subscription to RCB
 4:     request.toward ← RR
 5:     send request to rid's root directly
 6: else if toward is RR then
 7:     add subscription to RPB
 8:     request.toward ← SR
 9:     send request to sid's root directly
10: else if toward is SR then
11:     add subscription to RRB
12:     if local-node is sid's root then
13:         request.toward ← SP
14:         send request to sid's proxy
15:     else
16:         send request to sid's multicast parent
17:     end if
18: else if toward is SP then
19:     add subscription to SRB
20:     request.toward ← SC
21:     send request to sid's client
22: else
23:     error
24: end if
```

sid, a receiver ID rid, and a field *toward* indicating where to forward the event. The forwarding procedure is simple and similar to Procedure 1 in the reverse direction. When an overlay node receives an event, it checks the *toward* field. If the buffer leading to *toward* contains the sid as the first index, then *toward* is updated to be next stop and the event is enqueued.

Buffer packing

Each queue in a buffer has a limited size and may overflow if its consumer thread runs slower than the dispatcher adds events, for instance, because of a slow network link to a mobile device. 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 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.

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.

It may seem strange to split a single stream of events into multiple unicast streams when queues are overfull; the forwarding node may end up with more events to store than before! We rely on the pack policies to drop enough events. If the congestion or disconnection lasts long enough that no filters in a policy can drop any event, PACK triggers a built-in worst-case policy that drops all events in the queue and summarizes them using COUNT or the digester(s) specified in the policy.

When an event is first published, its header field rid is empty, indicating that it is a multicast event to be sent to all subscribers. A shared queue contains only multicast events. When a shared queue is packed by a receiver's policy into a private queue, the rid header is set to that receiver's ID,indicating that is now a unicast event. Any downstream node receiving unicast events will respect the rid and place them in the appropriate private queue... but first pushing any events in its shared queue over into private queues to preserve event order. If all the private queues are drained before the shared queue becomes full again, then the buffer switches to "shared" mode automatically.

As mentioned in Section 3, when a mobile client detaches, all its endpoints' subscriptions are suspended by suspending the consumer thread of the RPB and RCB. New events may continue to arrive, of course, and the queues may be packed while the client is detached. Once the client reattaches and the suspended queues are resumed, their consumer threads are resumed. If the client reattaches chooses a different proxy when reattaching, the new proxy will notify the RR to resume the buffer RPB; since the new proxy is on a different link, the RPB moves the suspended queue to the new index slot before resuming the thread.

5   Implementation

Our implementation is based on Java SDK 1.4.1. We chose Pastry as the overlay routing protocol [16] and used Scribe as the basic application-level multicast layer [7], although others would suffice. In the rest of this section, we discuss our implementation.
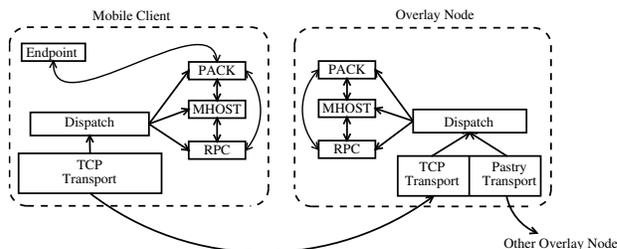

Figure 6: Service structure of clients and overlay nodes. PACK relies on the MHOST service to monitor mobile client (and its endpoints) state, while the endpoints use the PACK interface to publish events or make subscriptions. Communication between overlay nodes and clients uses a different transport service than the one used for inter-overlay protocol; clients do not participate in the PACK overlay.

Service architecture

Each node of the PACK system adopts a service-oriented architecture as shown in Figure 6. Each node contains the same set of services, such as a transport service, a simulated RPC service that blocks its caller until a reply is received, a mobility service ("MHOST") that manages the state between a client and its proxy (see Section 3), and a PACK service that allows endpoints to publish events, subscribe, and specify packing policy. Each service communicates with its peer on other nodes through the transport service. Note the transport used for client-overlay communication is separate from the one used for inter-overlay communication. Thus PACK excludes clients from participating in the peer-to-peer protocol. Inter-service communication within a client or overlay node uses local service interface invocation, masking the fact that a service is distributed across all participating nodes. For instance, PACK service registers a listener with MHOST to be notified about state change of the proxy.

Currently we connect overlay nodes using TCP/IP, which provides reliable, ordered transmission and congestion control of a single hop between overlay nodes and clients. We use Scribe [7] to maintain application-level multicast trees so PACK populates the subscription requests from leaves toward root as in Section 4. 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. Since PACK works on the queues accumulated above TCP (namely, after a sender's TCP buffer is filled), the events in TCP sending buffer are not accessible to PACK and they may be blocked until they get through to the other end. Rather than developing a customized protocol to replace TCP, we limit TCP's send buffer size (as 1024 bytes) to diminish TCP's overhead for now. Ultimately it may be best to replace TCP with UDP and extend our transport service to

handle event (packet) retransmission and congestion detection, which may also relieve the problem of lost events in TCP buffer as the client disconnects/moves without explicit request.

Ladder packing

When packing an event queue is necessary, PACK must determine which level of filters to apply. Packing at a high level may drop many important events. On the other hand, packing at a low level 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, initially one. 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 rational is that the change of t_l captures 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 a history of the turnaround time of all levels, t_l, smoothed using a low-pass filter with parameter $\alpha = 0.1$ (empirically derived) from an observation $\widehat{t_l}$:

$$t_l = (1 - \alpha)\widehat{t_l} + \alpha t_l .$$

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

$$\delta_l = (\widehat{t_l} - t_l)/t_l .$$

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.
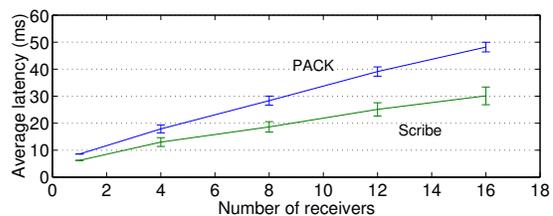

Figure 7: Average latency for one sender and multiple receivers, using a single-node PACK overlay. The sender publishes an event at the fixed interval of 200ms. Each receiver computes its own average delivery latency, and y axis shows the average of all receivers' averages and the bar indicates the standard deviation.

6   Evaluation

In this section we present some experimental results from the PACK service, using the Emulab testbed at Utah.[1] We constructed a network of five hosts interconnected by a switched 100Mbps LAN. The loss rate and latency on all links were set to zero. We turned off the just-in-time compiler and garbage collector in the Java VM. We measured the overhead of PACK buffering on the overlay nodes, examined the queueing behaviors when PACK polices were triggered, and the delay as clients disconnect and reconnect to proxies.

Buffering overhead

We set up a single data sender and a single overlay node on the same Emulab LAN host (although a normal PACK deployment would place data senders on client hosts that are distinct from overlay hosts, this configuration allows this experiment to focus on the cost of buffer operations rather than the network latency). We used 1 to 16 receiver clients, distributed evenly across the other four LAN hosts (again, a normal PACK deployment would place each receiver client on a separate client host, but for the purpose of this experiment four hosts were sufficient). The sender published an event every 200ms and all receivers subscribed to the same sender. The single overlay node played the role of proxy and root for all end points. We ran each experiment for 3 minutes and each receiver computed the average latency over all events it received. We compare two sets of tests, one using PACK and another using Scribe's own `join` and `multicast` interface. We plot the results in Figure 7.

As the number of receivers increased, the plot shows that both the delivery latency of PACK and Scribe grew linearly. PACK's larger slope indicates a non-trivial overhead compared to the baseline Scribe. The overhead mainly comes from buffering and thread synchronization.
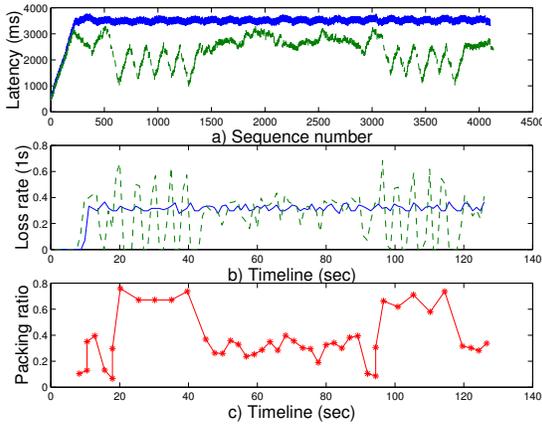
---

[1]http://www.emulab.net/

Figure 8: Queueing behavior of event reduction using Drop-Tail (steady line) and a three-level PACK policy (line with jig-saw variances). The first plot shows the running sequence of perceived latency of all events by receivers. The second plot shows the loss rate perceived by receivers over past 1 second window. And the last plot, derived from a trace of the overlay node, shows pack operations at various times with the ratio of event reduction.

Each event has to traverse four buffers on the single PACK overlay node (see Section 4), and another two on the clients. The buffer RCB contains one entry in the second index for each receiver since they all reside in different JVMs (or clients). Each receiver queue has a worker thread to pull events and transmit across corresponding TCP connection. We believe that a non-blocking network transport service should reduce the overhead [18].

Queueing tradeoff

To measure the queueing behaviors when 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-level PACK policy. Each level of the policy contains a single filter, randomly throwing out events (10%, 25%, and 50% respectively). We show the results in Figure 8.

Figure 8(a) shows the latency perceived by the receiver. After the buffer filled up, events in the DropTail queue have a (nearly constant) high latency because each event has to go through the full length of the queue before transmission. On the other hand, events in the queue managed by the PACK policy exhibit lower average latency because events may be pulled out of the middle of the queue, so other events have less distance to travel.

From these results it is clear that application designers should use filters that are more likely to drop events in the middle (such as EVERY, RANDOM, GUNIQ) rather than at the tail (such as LATEST).

Figure 8(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 8(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). The filtering level varied, despite the steady publication rate, because the RANDOM filter dropped varying amounts of events and our heuristic adapted to longer or shorter inter-packing intervals by adjusting the filtering level.

Client attach/detach

As a mobile client detaches from and re-attaches to its proxy, PACK suspends and resumes its event queue in the RPB buffer (located at RR). To measure how this operation scales with many moving clients, we again set up one sender and one overlay node on one LAN host in the Emulab topology, and varied the number of receiver clients (distributed evenly across the other four LAN hosts). Each client had one endpoint. Each client explicitly repeated the operations of attaching to and detaching from the overlay node 20 times, while waiting 5 seconds before each state transition. We measured the delay from the client-issued "attach" request until the first buffered event arrived, and Figure 9 shows that the delay was less than one second. This latency is important because it directly affects the user experience in many applications.

While the average delay clearly grew as the number of receiver clients increases, there was a large variance of the delay across receivers. We saw a similar wide variance across the 20 requests within a single receiver. We believe that this variance was due to thread scheduling and synchronization effects in the Java VM. The detach/attach requests, implemented by the RPC service, were handled by the proxy using a small thread pool. Also, the RPB buffer had one queue for each client; each had a consumer thread that transmitted events across the TCP connection to a client. The threads compete for the network
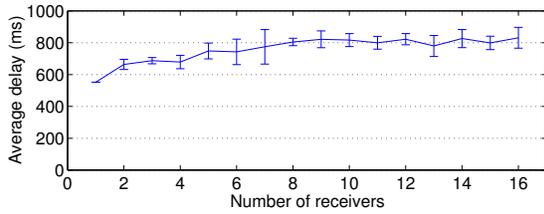
Figure 9: Detach/Attach delay measurements. Each receiver client attached to and detached from the overlay node, with 5 seconds interval between state transitions. The delay is measured as the time between the client issued an attach request until the first buffered event was released and arrived at the client. The points on the curve are the average delay of the 20 requests, and the bars are the standard deviation across the receivers.

since every queue in the RPB buffer had events buffered during client disconnection. This competition may be the more significant effect because Figure 7 shows little latency variation under a light load.

We could further improve PACK performance with several optimization techniques. A non-blocking network library could reduce the number of threads. Each thread could pull more than one event from its queue each time to reduce synchronization overhead. We could also study whether PACK fairly treats receivers that may subscribe to different sources with varying policies. All these topics are future research.

7 Applications

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 (AP), 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, leaves the network, and so on.

One of our goals is to provide an IP-based location 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. Figure 10 shows a Web proxy, modified from an open-source Java proxy [12], that is able to 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. Similarly, a location-prediction service could instruct a Guide application [8] on a mobile device
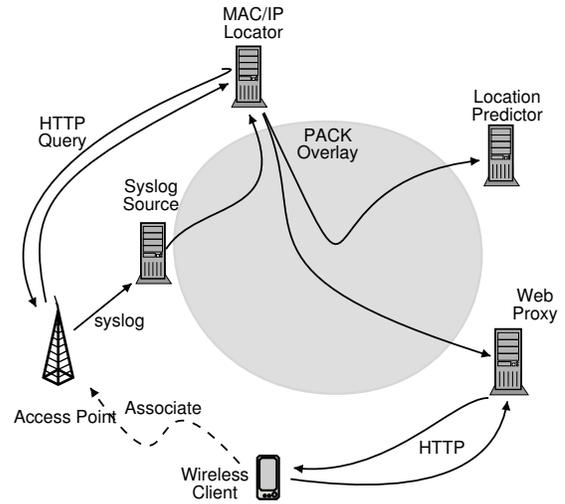

Figure 10: The MAC/IP locator monitors the syslog message stream and polls the AP for MAC-IP mapping. Then the locator publishes another stream with location updates of the mobile device. The Web proxy and a location-prediction service subscribe to the output of the locator to, for instance, push location-related content to clients.

to prefetch content based on next likely stop.

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.

MAC/IP locator

We focus our dicussion 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 6-level set of filters, some of which could be chained on the same level, but we chose to separate them for easier tracing. These filters are listed as follows (the policy is not shown to save space):

1. EQ: retain only events whose *message type* is "Info";

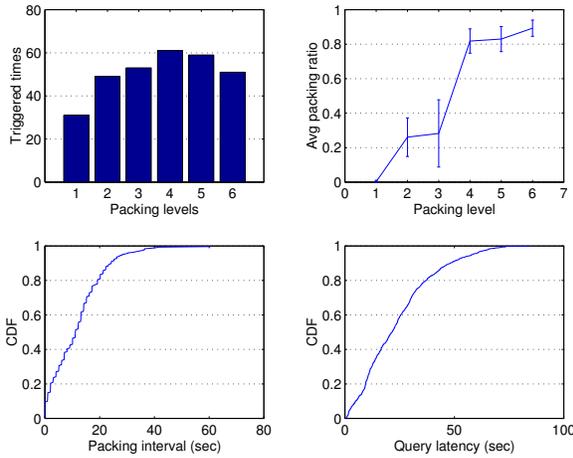2. INSET: discard certain events such as "Authenticated" or "roamed";

Figure 11: Statistics derived from the PACK trace collected on behalf the MAC/IP locator, who made a subscription to syslog source with a 6-level filtering policy.

3. MATCH: discard the events whose *host name* represents an AP instead of mobile clients;

4. FIRST: retain only the first event whose *action* is any of the four messages indicating the clients' departure from the network;

5. GUNIQ: remove all events with duplicated *AP name* except the first one (see the optimization discussed below);

6. EVERY: drop one event out of every three.

To accelerate the query performance, we made two optimizations to the locator. First, we do not query the AP if the syslog event already contains an IP address for the client. Second, when querying the AP we retrieved the list of all its associated clients and cached the results to speed up lookups for other clients. We collected the PACK trace for a hour-long run and Figure 11 shows some basic statistics.

The upper-left plot presents the distribution of the filtering levels triggered by PACK service. All filtering levels were triggered, varing 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 work load carefully to configure more efficient policies. The lower-left plot indicates that PACK triggered the policy rather frequently, with the median approximately 11 seconds. The lower-right plot shows the latency, derived by the time the query is resolved and the timestamp in the original syslog event. Although we set

the connection timeout to be 30 seconds for each poll, the longest delay to return a query was 84 seconds suggesting some AP was under heavy load and slow to return results even the connection was established.

Discussion

The locator could adapt to situations when level 6 is frequently triggered by creating multiple threads for parallel polling, so fewer events (which might be association messages) might be dropped. We are currently reluctant to take this approach since the downstream application may want in-order event delivery. The location predictor, for example, is sensitive to the sequence of moves.

We note that filters 1, 2, and 3 throw out events having no value to the locator service. If the source supports filtered subscription then none of those events need to be transferred across the network. The source, however, might become the bottleneck as the number filters to run increases. Rather than using PACK as a filtering system, we believe a more general infrastructure is necessary, such as a content-based event system with built-in (limited) filtering or a data composition network supporting a more powerful language [10]. PACK complements these systems to deal with buffer overflow issues.

8 Related work

The design choices made by PACK generally follow the principle of Application-Level Framing [9]. The data manipulation and transfer control are based on Application Data Units (ADU). In our case, pack operations are performed on the queued data units with a particular structure. On one hand, it is simplest to drop the recent ADU when a queue is about to overflow. On the other hand, this policy is inadequate or even incorrect for many applications with different requirements. Although this flexibility could be implemented at both sender and receiver, it is not easily deployable to intermediate IP routers. Thus an application-level overlay infrastructure is attractive since we can push the "packing" function closer to congestion and disconnection to improve scalability and responsiveness.

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 [11, 15]. 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 appropriate customized policy, a receiver is able to get critical data or summary information during the time of congestion or the recovery period. For many applications this outcome is better than a strict reliable service (TCP) or a random-loss (UDP) service.

Recent work using an overlay of event brokers to provide a content-based publish/subscribe service has been focused on routing and matching scalability and has largely ignored end-to-end flow control [4, 6]. Pietzuch and Bhola, however, study the congestion-control issues in the context of the Gryphon network during the course [14]. Congestion in the whole system can not be solved by simply interconnecting nodes with TCP because the overlay is constructed in application space above TCP. Their solution is to apply additional protocols for end-to-end reliability for guaranteed event delivery. The sender (or the broker serving the sender) then has the responsibility to store all the events during congestion for later recovery, such as using a database. From the application's point of view, their protocols are no different than traditional approaches and there is no explicit support for mobile clients.

Receiver-driven layered multicast (RLM) [13] leverages the fact that multimedia streams can be encoded in different layers (rates), each of which requires different bandwidth. The receivers then join only the multicast group (corresponding to layer or encoding rate) that best matches available network capacity. In a way, this idea is similar to the PACK service, which enforces receiver-specified policies. RLM, however, focuses only on multimedia applications, works at the packet level, and requires IP multicast. PACK is built in application space and requires no special capability in an IP network; it uses the same "packing" mechanism for flow control (managing the queues at end hosts) and congestion control (managing queues in overlay nodes); and it provides explicit support for mobile clients (either data sources or sinks). On the other hand, PACK needs a deployed overlay infrastructure, and PACK requires more applications effort since layer selection is transparent in RLM and requires no explicit application policies.

Researchers in the database community provide a query-oriented view on continuous stream processing. One of the goals is to design algorithms (or approximations) to realize SQL-like operators (such as join) over the data stream. It is desirable for these algorithms to use only limited memory and time so the system can keep up with the arriving data [2]. In the Aurora system, Tatbul et al. propose to reduce system load by dynamically injecting data-drop operators in a query network [17]. Choosing where to put the dropper and how much to drop is based on the "QoS graph" specified by applications. While this approach is analogous to PACK's policy-driven flow control, there are several differences. First, Aurora assumes a complete knowledge of the query network, using a pre-generated table of drop locations as the search space. Second, PACK works on all buffers along the delivery path and supports mobile clients while Aurora only works at the centralized query engine. Finally, their QoS function provides quantitative feedback when dropping data while PACK allows explicit summarization of dropped events.

ROAM uses indirection points in the infrastructure to support robust and efficient hand-off [19]. Our PACK service adopts a similar approach and retains the transport state on these rendezvous points. PACK excludes the (mobile) clients from participating in the overlay and PACK clients use a single proxy to help manage the endpoints it hosts. A quite different approach taken by the Intentional Naming System (INS) is to route messages to a name, whose destination is resolved as the message hops through the overlay network so the receiver is free to move in the mean time [1]. Currently PACK flow control may not be applied directly on top of INS since the messages from same sender may be sent to different receivers.

9   Summary and Conclusion

Ubiquitous computing applications often need context information to adapt to their changing environment. The flow of context information from sensors and other information sources to applications is highly dynamic. Traditional flow control and congestion control approaches either stop the sender or drop arbitrary data (packets) between sender and receiver. Our PACK service allows applications to specify data-reduction policies that selectively drop and summarize context events when the flow exceeds the capacity of wireless network links or an application's ability to consume the data, or when the mobile device is temporarily disconnected.

PACK enables customized data-reduction policies so loss-tolerant applications can trade data completeness for fresh data, low latency, and semantically meaningful data. PACK employs an overlay infrastructure to support mobile data end-points for temporary disconnection and hand-off. Finally, PACK's summaries allow receivers to meaningfully react to current buffering conditions.

References

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, pages 186–201, Charleston, South Carolina, United States, 1999.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the 21st ACM Symposium on Principles of Database Systems*, pages 1–16, Madison, Wisconsin, 2002.

[3] M. Balazinska, H. Balakrishnan, and D. Karger. INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. In *Proceedings of the First International Conference on Pervasive Computing*, pages 195–210, Zurich, Switzerland, August 2002.

[4] G. Banavar, M. Kaplan, K. Shaw, R. E. Strom, D. C. Sturman, and W. Tao. Information flow based event distribution middleware. In *the Middleware Workshop at the ICDCS 1999*, Austin, Texas, 1999.

[5] 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, Oregon, United States, 2000.

[6] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3), 2001.

[7] M. Castro, M. B. Jones, A.-M. Kermarrec, A. Rowstron, H. W. M. Theimer, 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*, San Francisco, CA, Apr. 2003.

[8] K. Cheverst, N. Davies, K. Mitchell, and A. Friday. Experiences of developing and deploying a context-aware tourist guide: the GUIDE project. In *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking*, pages 20–31, Boston, Massachusetts, United States, 2000.

[9] D. Clark and D. 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, Pennsylvania, United States, 1990.

[10] N. H. Cohen, H. Lei, P. Castro, J. S. Davis II, and A. Purakayastha. Composing Pervasive Data Using iQL. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, pages 94–, Callicoon, New York, June 2002.

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

[12] Mark R. Boyns, Muffin: World Wide Web Filtering System. http://muffin.doit.org/, Nov. 2003.

[13] 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, California, United States, 1996.

[14] P. R. Pietzuch and S. Bhola. Congestion control in a reliable scalable message-oriented middleware. In *Proceedings of the 2003 International Middleware Conference*, pages 202–221, Rio de Janeiro, Brazil, June 2003.

[15] 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, Tennessee, United States, 1994.

[16] 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.

[17] N. Tatbul, U. Cetintemel, S. 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*, Berlin, Germany, Sept. 2003.

[18] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable Internet services. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, pages 230–243, Banff, Alberta, Canada, 2001.

[19] S. Zhuang, K. Lai, I. Stoica, R. Katz, and S. Shenker. Host Mobility Using an Internet Indirection Infrastructure. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services*, San Francisco, CA, May 2003.