Evaluation of an Adaptive Transport Protocol

Benjamin Atkin and Kenneth P. Birman
Department of Computer Science
Cornell University, Ithaca, NY
{batkin,ken}@cs.cornell.edu

***Abstract*—Applications on mobile computers must adapt to high variability in wireless network performance. Extending the semantics of transport protocols to offer more control over communication to the user allows applications to adapt their behavior to bandwidth variability. We examine adding bandwidth notifications, priorities and timeliness guarantees to a network API as a method for achieving greater application control over bursty traffic. Experiments demonstrate that the extended API allows applications to adjust to bandwidth variations effectively. We also compare three different implementations of the API: two which run on top of TCP, and one new protocol, ATP, which performs comparably to the TCP extensions, but has better performance for some workloads, including a workload simulating remote file system traffic.**

I. Introduction

Wireless networks are characterised by high variability in network conditions: bandwidth and round-trip times can vary greatly depending on the distance of a host from a base station, and on local interference. The problem of adapting applications and network protocols which make use of stream-oriented communication, such as video playback, to variable wireless network conditions has been well studied. In this paper, we describe our work in providing support for applications which exhibit bursty communication patterns.

Many of the applications which one might wish to use on a wireless host send or receive data of multiple types, which are not of the same precedence, or sent in the same volume. For instance, FTP sends separate control and data messages, and a distributed file system client might send cache validations, receive new versions of files and write back modified files. In order to react to changing bandwidth available to the host, an application designed to operate in a wireless network might adjust its communication patterns, reducing the transmission of data of one type while increasing the priority of another type of data. Unfortunately, most existing interfaces to network protocols, such as the BSD sockets interface, do not provide much detail about network conditions to the user. Making up for this omission and supplying a greater degree of control over network communications to applications is a nontrivial undertaking: even the task of determining the bandwidth available to the application can be complicated if there are multiple applications concurrently using the network.

The authors were supported in part by DARPA/AFRL-IFGA grant F30602-99-1-0532, with additional support from Microsoft Research and from the Intel Corporation.

We have concentrated on bursty applications that send messages over the network intermittently or unpredictably, since stream-oriented protocols such as TCP have been highly optimised for wireless networks. Bursty communication makes tasks such as dividing bandwidth between different classes of data more difficult, since the bandwidth allocation between classes cannot be both constant and fair. Dealing with messages rather than streams also permits a more flexible interface, in which a message can be re-prioritised or have its transmission suspended, restarted, or cancelled. Web browsers, file systems and FTP, and X Windows are potential examples of applications which could benefit from our extended network API.

The structure of the rest of this paper is as follows. Section II describes the motivation for our API and identifies some applications which could benefit from it. Section III describes NAI, the Network-Aware Interface we have designed for bursty communication. Section IV describes the ATP implementation of NAI and its algorithms, as well as discussing some examples of execution. Section V compares the performance of ATP to TCP and implementations of NAI over TCP, in a series of experiments. Section VI summarises related work, while Section VII concludes and describes our plans for future work.

II. Motivation

To illustrate the scope of variability in wireless communication, we have measured network conditions in our own 802.11b wireless network. Figure 1 shows some representative measurements. The graph of bandwidth in Figure 1(a) is derived from packet-pair measurements [1] made by a receiving host as the sender moved in the vicinity of a base station. Each second, two packets were sent consecutively from the wireless host to the receiver, which was connected to the base station, to measure the inter-arrival time. Under ideal conditions, the inter-arrival time measurement corresponds to the time to send a single packet over the wireless network. The inverse of this number provides a rough indication of the bandwidth available at that instant. We note that since the 802.11b protocol incorporates its own packet retransmissions, high error rates are translated into reduced bandwidth and increased latency.

Both the bandwidth and round-trip time measurements shown in Figure 1 are highly variable. Applications which perform a large amount of network communication under these types of circumstances may have poor performance, unless they can

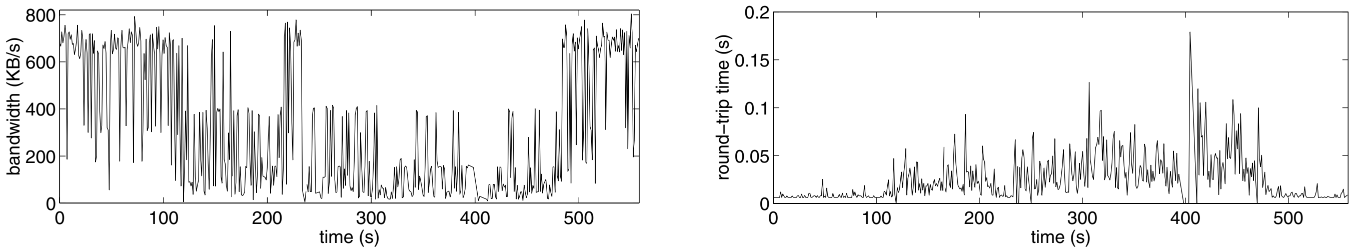


Figure 1: Time series of bandwidth and round-trip times over a 10 minute interval in a wireless LAN. *The graphs show values obtained by a combination packet-pair and round-trip time estimate for 1500-byte packets measured on our wireless network. A laptop communicated over a wireless link with a desktop machine attached to a base station. Variations are due to distance from the base station and local interference around the laptop as it moved. The regions from 0-100 and 500 seconds onwards correspond to the laptop being next to the base station.*

adapt to such a degree of variability. Underestimating the available bandwidth may underutilise the network, and overestimating bandwidth may leave an application unable to function. *Adaptive* applications are able to change their degree of communication or mode of operation to suit the currently available bandwidth. Examples of classic adaptive applications include streaming video playback and image retrieval during web browsing [2], [3]. In response to low bandwidth, a video player might reduce the quality of the frames it displays, or a web browser might retrieve degraded versions of images appearing in web pages, instead of the original high-quality versions. We call this *modal adaptation*, since the application has a set of modes it can operate in, each with an associated communication cost, and chooses its current mode based on the available bandwidth.

Considered as a whole, Web browsing is an example of a different class of applications, which we call *mixed-mode* applications. In contrast to video playback, these are typified by irregular communication of distinct units of data, which can be divided into several classes of importance. Common characteristics of mixed-mode applications might include:

(i) Bursty communication of discrete, application-specified data items
(ii) Different classes of data transfers, each with a different importance (priority), and potentially with different timeliness requirements and distributions of sizes
(iii) Possibility for fine-grained adjustment of behavior based on bandwidth availability

We have already listed some examples of potential mixed-mode applications. Two are particularly worth elaborating on:

Web browsing: A web browser retrieves text, images and other data types. Rather than uniformly degrading image quality, it can prioritise text over images, and then leave the remaining bandwidth to be used for image retrieval. Encodings such as progressive JPEG for images can allow optimistic retrieval of an image at high resolution, and an early abort of the request (or a partial result) if bandwidth turns out to be insufficient. Web browsing is an interactive activity, so the tradeoff of image retrieval delay versus available bandwidth can also be used to decide at which quality to retrieve an image.

Distributed file system: Distributed file systems are popular because of the convenience, security, fault-tolerance and data-sharing capabilities which they provide, in contrast to the file system local to a machine [4]. Unlike web browsing, the work done by a distributed file system client may be on behalf of several independent applications on a host (for instance, a word processor and compiler executing concurrently). A caching file system client also performs multiple types of communication with a file server: it has to fetch files in response to requests and write files back to the file server, and it may also validate files in the local cache in order to reuse them, and prefetch files. If a file is not shared with other users, then writing it back to the file server can be delayed until bandwidth is plentiful; prefetching files is only advantageous if it does not degrade the overall performance of the file system client. Once again, timeliness of communication is important, in order to prevent the user suffering long delays in file accesses. Cache validations should be performed quickly, but prefetches can take longer.

III. A NETWORK-AWARE API

TCP is a well-tuned protocol and the standard reliable communication protocol for the Internet, so the benefit of implementing a completely new, incompatible protocol is small. Since most of the mismatch between TCP and the mixed-mode class of applications we have identified stems from the narrow interface to TCP, the approach we have explored is to implement a protocol that runs on top of TCP or some other reliable transport protocol, but has enhanced semantics.

We use the name NAI (Network-Aware Interface) to refer to the API itself. In Section V we compare three implementations of the interface: ATP, NAI-TCP, and NAI-MTCP, of which ATP (the Adaptive Transport Protocol) is the most sophisticated. Our discussion of NAI is made with reference to its implementation in ATP, which includes all the features of NAI.

A. Sending and receiving messages

The basic operations of NAI conform to the BSD sockets interface. NAI provides the regular `socket`, `bind`, and `connect` calls, but augments the `sendmsg` and `recvmsg` calls with additional information. Unlike TCP, NAI is message-oriented: an application specifies its data units (files, images, RPCs, and so on) to NAI explicitly, and NAI preserves message boundaries at the receiver. NAI incorporates some ideas from Application Level Framing (ALF) [5] in the way that it handles messages. When an application makes a send call, it tells NAI how to process the message: what its priority is relative to other messages, and how to react if there is insufficient bandwidth to deliver the message. Priorities are strict, so that low-priority messages wait for the high-priority messages ahead of them. Messages can be sent synchronously or asynchronously, allowing a sender to inspect the state of a message as it is being sent, to see how much data remains to be transferred. Based on this information, it might decide to abort the transfer, defer it to make way for more important messages, or restart it if it was already deferred.

B. *Message queues*

A strict priority scheme for transmitting messages introduces the possibility of starvation of low-priority messages. This is particularly the case for multiple non-cooperating applications, since one application could undermine the priority scheme by putting all of its messages at the highest priority. Transmitting all messages of one priority serially would also make the transmission delay for a message highly unpredictable, so that an application might have trouble setting callback timers appropriately. To overcome these problems, NAI incorporates message queues. Each application initially has a separate message queue, and it can create additional message queues as it sees fit. NAI schedules messages from each message queue independently, dividing the available bandwidth between the queues according to the highest priority used by the queue. In this way, even an application which exclusively uses the lowest priority level will get some share of the bandwidth, though not as much as applications which use all the priority levels. We do not describe message queues in detail in this paper, since techniques for bandwidth division among concurrent streams are already well studied [6].

C. *Bandwidth estimation and timers*

In order to adjust its behavior based on network conditions, an application must first know how much bandwidth is available. The true bandwidth cannot always be determined accurately, so a bandwidth estimate must be derived. ATP incorporates a bandwidth estimator, but the two simpler implementations of NAI do not. The remainder of this section describes NAI over ATP.

The most straightforward way for an application to monitor how much bandwidth is available is to register a *bandwidth callback* and an associated bandwidth range (this is an idea borrowed from Odyssey [3]). A bandwidth callback is a function specified by the application, and called by NAI when the bandwidth moves outside the specified range. The bandwidth callback function may itself register a new callback, with a new range around the new bandwidth value.

A disadvantage of this mechanism is that if multiple applications are using the network, then they will all see the same bandwidth estimate, unless bandwidth is somehow shared between them (we discuss sharing bandwidth between applications later in this section). If the applications communicate in bursts, rather than over streams, then it is nontrivial to determine what share of the bandwidth each should be entitled to, and therefore, what estimate it should be given.

An alternative solution, which is also provided by NAI, is to allow applications to specify their bandwidth requirements implicitly, by using *callback timers* for individual messages. A callback timer specifies a timeout and a function to call if the timeout expires; it can also expire early, if the bandwidth estimate indicates that the message cannot be delivered in time, according to the current bandwidth. Using a callback timer, an application can specify how long it expects a message to take to transmit – implying a corresponding available bandwidth – and relies on NAI to invoke the callback if the timeout occurs. If the callback is invoked, NAI first suspends transmitting the message, and the application then decides whether to continue or defer transmission, or to cancel the message. Callback timers provide a finer degree of control than coarse modes based on the available bandwidth, since the application can speculatively send messages without knowing ahead of time that the bandwidth to deliver them is available. Using a callback timer also allows an application to ensure timely delivery for a message.

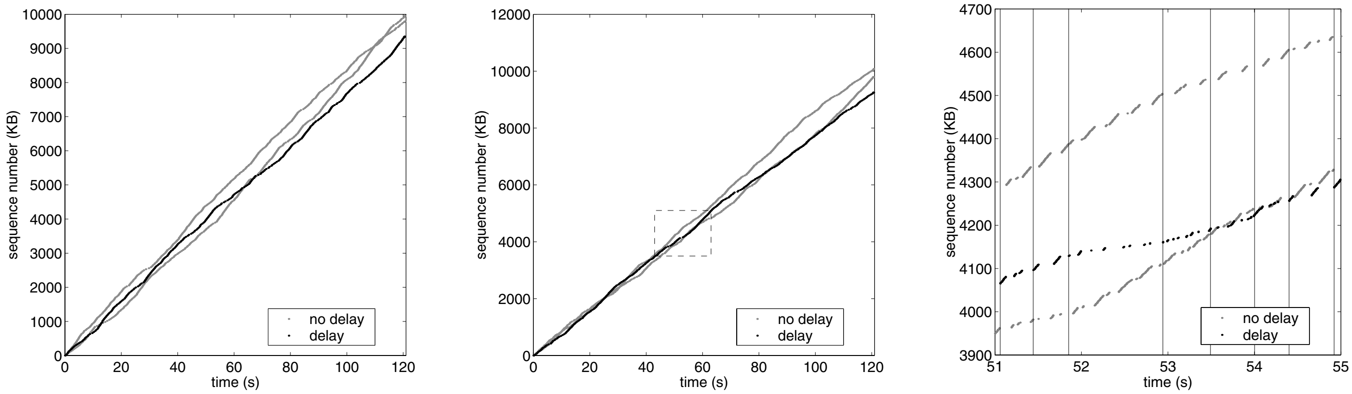
Implementing callback timers requires controlling the admission of messages. The ATP implementation does this by ranking messages by priority and then deciding if a new message can be added to the currently queued messages without jeopardising the timers of any existing messages (that is, without causing a message which was deliverable under the current bandwidth to exceed its timeout value). Within a priority level, ATP uses the Earliest-Deadline First scheduling algorithm [7], familiar from real-time scheduling, for delivering messages within their timeout intervals.

Of course, not all messages are sent with callback timers, in which case they are implicitly assigned an infinite timeout. An application which does not make use of callback timers at all could accumulate a large backlog of messages, if it sends messages at a faster rate than they can be transferred over the network. We are investigating implementing a backlog-based callback scheme to support these types of applications, which invokes callbacks based on the incoming and outgoing rates of messages.

IV. Implementation

Implementing NAI within the Adaptive Transport Protocol requires some effort, even with the aid of a reliable protocol such as TCP. Bandwidth estimation, managing timers, and providing message-oriented, rather than stream-oriented semantics, must be provided on top of the underlying kernel protocol. In addition, TCP's congestion-control scheme represents a potential obstacle to our mechanisms for flow control and assigning priorities to messages.

ATP, the complete implementation of NAI which we describe here, runs at user-level over TCP or UDP, and consists of approximately ten thousand lines of C code. The other two implementations of NAI are described in Section V.



(a) (b) (c)

Figure 2: Unequal divisions of bandwidth for concurrent TCP connections. *Each flow transmits 32 KB messages back-to-back for 2 minutes; sequence number plots for the connections are shown. In (a), the stream shown by the darkest line pauses for 100 ms between each send call; in (b), it pauses for 150 ms. Plot (c) is a magnification of the boxed region in (b). Vertical lines in (c) indicate instants when the stream with the delay between sends completes sending a message: this interval of plot (b) shows a high degree of variation in the bandwidth obtained by the delayed stream.*

A. Reliable transport subsystem

ATP is a reliable protocol, so it must run on top of a reliable datagram protocol or a reliable stream protocol. At first glance, TCP would seem to be perfectly adequate, since it has been highly optimised to perform well both in local-area networks and wide-area networks. TCP has also been adapted to cope with the peculiarities of wireless networks, such as high error rates and packet losses [8], [9].

However, implementing ATP on TCP requires considering a number of alternatives. Transmitting an entire message at once using TCP may result in the message being buffered in the kernel (if there is sufficient buffer space), preventing an application from deferring the send operation or aborting it. Once TCP has copied data into the kernel, it is not easy to determine how much of it has been sent. A final difficulty lies in deciding whether to use multiple streams to connect the sender to the receiver, and, if so, how to allocate messages to streams. Using a single TCP stream will result in all message transmissions being serialised, so that a high-priority message may have to wait for a low-priority message. Using multiple TCP streams may result in unpredictable competition for bandwidth, since TCP is a greedy protocol and most common implementations of TCP do not coordinate congestion control between streams [10].

In our initial version of ATP, we chose to allow message transmission over either TCP, or a reliable datagram protocol on top of UDP, which we will refer to as SPP (Sequenced Packet Protocol). The TCP implementation is the simpler of the two, and runs over a single TCP connection, but sends messages in fixed-size segments (1 MTU), rather than sending an entire message at a time. Padding is required for small messages to enable the receiver to detect segment boundaries. It has the natural advantage of being TCP-friendly. The SPP implementation performs its own buffering, retransmissions and duplicate suppression at user level. Since it uses UDP datagrams, it requires no padding to distinguish message boundaries, and is therefore more efficient for transmitting small messages. Unlike TCP, SPP is not optimised for WAN use, and has not been thoroughly tested to determine its fairness or behavior under high error rates. It is robust to the errors we have seen in our 802.11b network, and to packet drops caused by queue overflows in the sender's network stack. The ATP experiments in this paper use the version running over SPP.

For the purposes of comparison, we have also implemented a version of NAI using multiple TCP connections, one per priority level, which we refer to as MTCP (multi-stream TCP, described in more detail in Section V). We conducted some experiments to assess the behavior of concurrent TCP streams and determine if this design was suitable for an NAI implementation.

The graphs in Figure 2 illustrate some cases of competition between concurrent TCP streams which result in an unequal division of bandwidth. We conducted experiments to measure how three concurrent TCP flows compete for bandwidth over a link with a capacity of 256 KB/s. Each flow sends 32 KB messages in a loop for 120 seconds. One of the three senders was delayed for a constant amount of time between the completion of one send call and the start of the next, to investigate the effect of TCP's slow-start restart [11] for idle connections. The expected outcome of these tests would be for the two undelayed senders to receive a roughly equal share of bandwidth, and the delayed sender to receive a smaller share, decreased in proportion to the size of the delay. However, as Figures 2(a) and 2(b) show, the shares of bandwidth are not constant, and at some points the delayed sender outperforms the undelayed senders! In addition to looking at sequence number plots, we also calculated the times taken for individual send operations by the delayed sender. Figure 2(b) shows a sequence number plot where the delayed sender waits for 150 ms between send operations; measuring the time between completion of successive sends gives an average of 0.42 seconds, but a range from 0.22 to 1.09, almost a five-fold variation (a minimum of 0.22 is possible because the send can buffer data in the kernel and return fast, overlapping with the delay). This compares to an expected mean of 0.375 seconds if all the streams received equal bandwidth. Figure 2(c) shows the region of this test that the send operation taking 1.09 seconds lies in, with vertical bars indicating the completion times of sends. This type of unpredictable and uncontrollable contention effect argues that NAI over multiple TCP streams may ultimately be inferior to other options, particularly in settings where small and infrequent high-priority messages are intermixed with lower-priority bulk data transfers.

B. Bandwidth estimator

Estimating bandwidth is a necessary component of an adaptive transport protocol, since both the application and the protocol itself rely on this value in order to adapt appropriately to network changes. ATP requires a bandwidth estimate to fully implement callback timers, or else a message can never be reported as undeliverable before its callback timer expires.

Much work has been devoted to the general problem of estimating bandwidth for flows in a wide-area network. Wide-area bandwidth estimation schemes must arrive at an estimate of limiting bandwidth, which lies at some link along the path between a sender and receiver. In contrast, we assume that the rate of communication is principally limited by the bandwidth on the wireless link. Since all communication between the mobile host and remote hosts must be over this link, we can estimate the total bandwidth available on it, rather than deriving separate estimates for each connection or destination host. Our current estimator assumes that traffic from protocols other than ATP constitutes a negligible fraction of the total traffic, but this restriction would be removed in a kernel version of ATP.

A further important difference from wide-area bandwidth estimation is a side-effect of ATP semantics. Since ATP incorporates priorities for messages, an inaccurate estimate can cause a priority inversion.

The difficulty is that the device driver may buffer datagrams when it is unable to transmit as fast as the incoming rate. If an over-optimistic bandwidth estimate causes the kernel to buffer datagrams from low-priority messages, these will hold up the transmission of datagrams from high-priority messages until the send queue is free of low-priority datagrams. It is impractical to remove datagrams from the send queue, but some operating systems allow the length of the queue to be read by applications (for instance, by FreeBSD's `ifmib` feature). The ATP bandwidth estimator incorporates a heuristic to "back off" and reduce its estimate when it detects a backlog in the send queue.

The bandwidth estimation algorithm: A straightforward scheme for bandwidth estimation is to count how many send operations, and of what sizes, complete over an interval, and divide to find the estimate. This can be inaccurate because the kernel buffers data, both at the protocol level (in the case of TCP – UDP does no buffering), and at the network device driver. Additionally, the time required to derive an estimate depends on the amount of data being sent. Blocking on a send operation for a large message will delay the estimate until the send completes.

```
int curbw;
int staleness = 0;
int polldelay;  // configurable, >= 0

bandwidth_estimate(used, backlog) {
  used = maximum(used, filter(used));

  if (backlog > MAXIMUM_BACKLOG) {
    curbw = used; staleness = 0;
    return (used, used - backlog*MTU);
  }
  else if (used < curbw) {
    int probe = PROBE_SIZE;
    staleness++;
    probe *= staleness / polldelay;
    curbw = used;
    return (curbw, curbw + probe);
  }
  else {
    staleness = 0;
    return (curbw, curbw + PROBE_SIZE);
  }
}
```

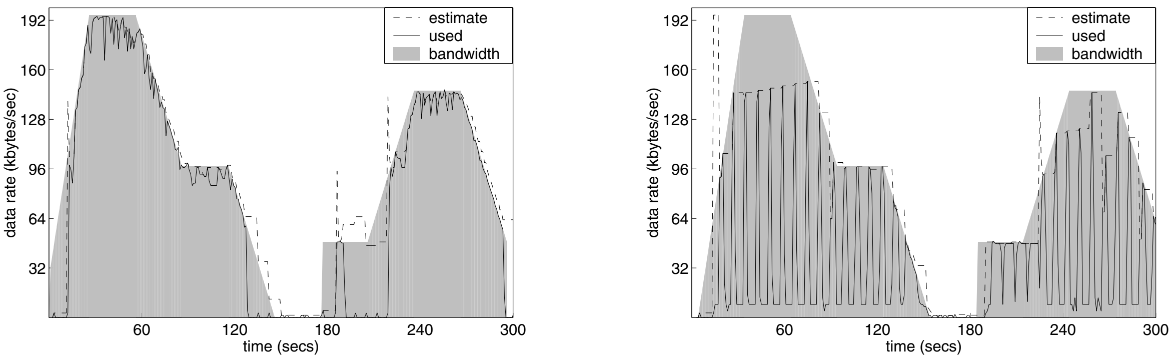
Figure 3: The bandwidth estimation algorithm. *Every second, ATP invokes the bandwidth estimator to determine a new estimate (this is a tuple of the internal estimate and the estimate advertised to the application). The bandwidth used over the preceding second is averaged with the values for the four preceding seconds, then the new estimate is generated, depending on the backlog and staleness of the current estimate.* `MAXIMUM_BACKLOG` *is set high enough to avoid transient backlogs (10 in our implementation), and* `PROBE_SIZE` *is half the minimum of the send queue capacity and the current estimate.*

Alternatively, the bandwidth used by a TCP connection can be derived from TCP's round-trip time estimate and the send window size. However, this value reflects how much data has been sent on the connection, and not the potential capacity of the connection.

ATP derives its estimate by polling the network card for the amount of data sent and received over the course of each second. This quantity is then used as a predictor of the bandwidth over the next second. Since the bandwidth reported by the network card depends on the amount of data which ATP tries to send, this simple estimate is inaccurate if the true bandwidth is higher than the send rate. Accordingly, the bandwidth estimator uses a probing scheme to speculatively increase the estimate.

Estimation relies on three statistics: the observed bandwidth, the length of the network interface send queue (backlog), and the staleness of its current estimate. Staleness measures the number of seconds since the last point at which the estimate changed, or since there was a "genuine decrease" in available bandwidth. A genuine decrease can be distinguished from a decrease in the count of bytes transmitted by detecting that the length of the send queue has increased (in fact, we check that it exceeds a threshold of 10 packets; the maximum length allowed is 50 in FreeBSD).

Figure 3 shows pseudocode for the bandwidth estimation algorithm. It runs once every second, and computes two values based on the statistics obtained over the previous second: `curbw` and `available`. The `available` value is the estimate of available bandwidth on the wireless link, which is supplied to the application. The `curbw` value is the amount of band-



(a) (b)

Figure 4: ATP behavior for two workloads. *Graph (a) shows bandwidth usage for a workload of 64 KB messages, entering the system every quarter of a second, with callback timeouts of one second. When bandwidth is less than 64 KB/s, no messages are admitted. Graph (b) shows a combination workload of high-priority 4 KB messages and low-priority 256 KB messages with long timers.*

width which ATP should restrict itself to using over the next second. This is an internal ATP statistic, which may be lower than `available` if the network interface send queue is nonempty and there is a consequent risk of priority inversion. The `available` value may also be higher if the estimator is probing the bandwidth. An averaging filter with a window size of 5 is used to smooth the estimates in order to make them less sensitive to transient spikes.

If the application has queued more messages than can be sent in a single second, then the estimator will automatically detect increases in available bandwidth, since the per-second usage will increase as the bandwidth increases. However, if the network is underutilised, then an increase may not be detected without an additional mechanism. For instance, an application using ATP may use the bandwidth estimate to determine its mode of operation, and not change into a mode using more bandwidth unless the estimate goes up. In this case the surplus bandwidth would never be exploited. ATP breaks this deadlock in two ways. Firstly, in the trivial case where the application is unwilling to send any data because it believes the bandwidth is zero, ATP polls the remote host to determine when bandwidth rises above zero, making a simple packet-pair estimate [12]. Secondly, ATP probes the bandwidth by speculatively increasing the estimate it gives the application. The intent behind this mechanism is that the application will eventually decide the estimate is high enough and attempt to exploit it by sending more data. Figure 3 shows the probe mechanism as part of the `bandwidth_estimate` routine.

C. Examples of ATP execution

To place the preceding algorithms in context, we examine some representative executions of ATP. Figure 4 shows bandwidth estimates for two examples of ATP execution. The actual bandwidth curves are synthetic, and were generated with the use of the Dummynet traffic-shaping module [13] (the bandwidth curves are explained in more detail in section V).

Graph (a) shows the bandwidth estimates when the system is saturated: every 0.25 seconds, a new 64 KB message enters the system, and must be delivered within a second. Between 120 and 220 seconds, almost no messages are admitted, since the bandwidth estimate is below 64 KB/s. Admission ceases when the estimate is in the vicinity of 64 KB/s, and resumes when the estimate has jumped to 140 KB/s from 48 KB/s. An anomaly is evident at 193 seconds, where the estimate jumps sharply due to a packet-pair measurement. Though this causes some messages to be incorrectly admitted, it is quickly rectified.

Graph (b) shows a mixed workload, with both high-priority and low-priority messages. Every 0.5 seconds, a high-priority, 4 KB message with a 1-second callback timeout enters the system; every 8 seconds, a low-priority, 256 KB message with a 16-second timeout enters. The spikes in the reported bandwidth curve indicate these larger, low-priority messages. The combination of smaller messages and looser timeouts allows ATP to make use of the period between 180 and 210 seconds when bandwidth is at 50 KB/s.

The use of the bandwidth estimate for message admission is illustrated in Figure 5, which shows part of an execution of the same workload as in Figure 4(b). Events at the sender are shown at the left, and at the receiver on the right. Arrows indicate the correspondence between the entry of messages into the system and their delivery to the receiver. The small, high-priority messages arrive every 0.5 seconds, while a large message arrives every 8 seconds. The decreasing bandwidth estimates correspond to the system entering the zero-bandwidth region of Figure 4(b). The interval between arrival of a small message and its delivery lengthens as bandwidth decreases, and some small messages are dropped due to their timers expiring. It is worth noting that some arrival-to-delivery intervals are greater than one second, the nominal timeout, but this is due to the timeout only being enforced at the sender, not at the receiver, in order to avoid requiring clock synchronisation. While the bandwidth is technically sufficient for delivery of all the small messages shown on the timeline, a reduction in the `available` value after 138 seconds leads to some messages being discarded when their callback timers expire. Insufficient bandwidth at 145 seconds causes a new 256 KB message to be rejected, and the 256 KB message

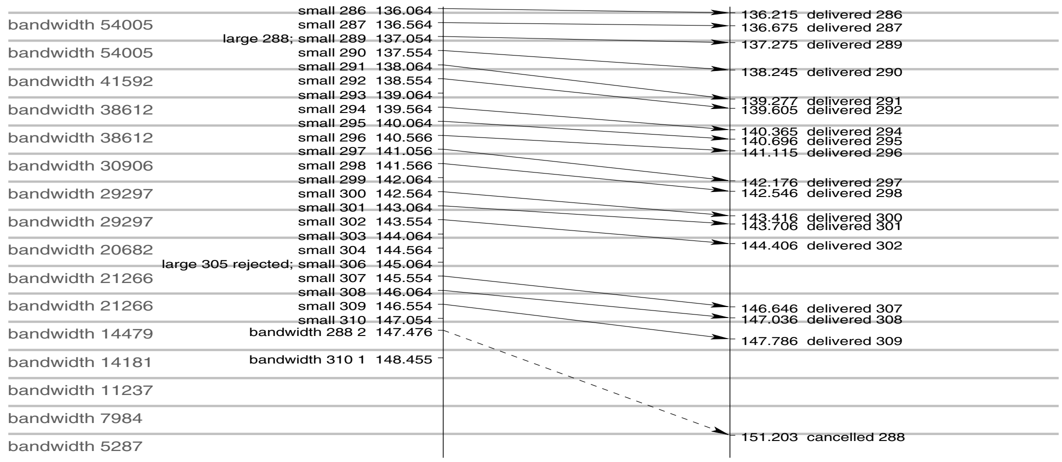


Figure 5: Timeline of ATP execution. *The section shown is from the execution of the same workload as shown in Figure 4(b). Entry points of messages into the system are denoted as "small" (high-priority, 4 KB) or "large" (low-priority, 64 KB), followed by a sequence number. The per-second bandwidth estimate is given in grey (each estimate is computed at the time marked by the line above it). Arrows indicate when sending an message commences at the sender and receipt occurs at the receiver. Messages after 148.5 seconds, and removal of messages due to timers, have been omitted – an message without an arrow was dropped due to a timer expiring.*

admitted at 137 seconds is discarded due to insufficient bandwidth at 147 seconds, before its timer has expired. The fact that the notification of cancellation arrives at the receiver four seconds later is due to the backlog in the device driver send queue. Slow delivery of cancellation notifications is tolerable because they serve only to free buffer space at the receiver.

D. Adaptation mechanisms

ATP incorporates two mechanisms by which an application can keep track of bandwidth availability: explicitly, through the use of bandwidth notifications, and implicitly, by relying on callback timers and callbacks to express its assumptions about the bandwidth. In this section, we compare the behavior of these two techniques with a mixed-mode workload and a bandwidth trace drawn from measurements of our wireless network (the trace is a subinterval of the trace shown in Figure 1). In addition, we show some effects of varying the poll delay for the bandwidth estimator, as described in Section IV-B.

The application we tested has four logical modes, corresponding to different levels of bandwidth usage. At each level, messages are sent with a 1-second timeout, of a sufficient size and frequency to match the target bandwidth usage, as follows:

level	*upper bound*	*usage*	*total usage*
1	100 KB/s	25 KB/s	25 KB/s
2	200 KB/s	100 KB/s	125 KB/s
3	400 KB/s	125 KB/s	250 KB/s
4	none	250 KB/s	500 KB/s

"Upper bound" refers to the upper limit of bandwidth for transmitting at that level; "usage" refers to the bandwidth usage if all messages at that level are delivered successfully; "total usage" gives the cumulative figures. Tests were conducted using two styles of adaptation:

Bandwidth notifications only. Each level corresponds to a mode of operation; the application registers the low and high ends of the bandwidth range for the current level, and ATP notifies it when the bandwidth estimate moves outside the range. When at mode n, the application sends all messages from levels 1 to n. All messages have the same priority.

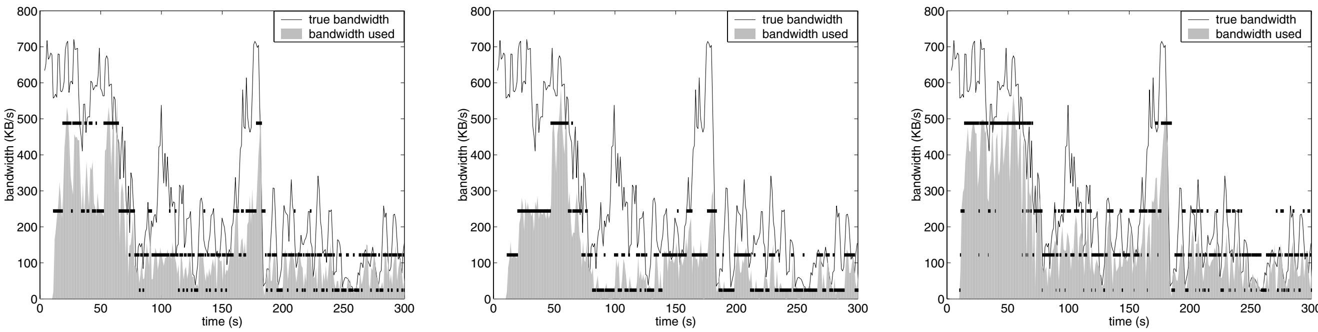
Callbacks+priorities. The application transmits without using modes, instead sending messages for all the levels concurrently, but with priorities to rank the messages (level 1 having the highest, level 4 the lowest priority).

algorithm	*poll*	*MB transferred by level*				*total*	*bw*
		1	*2*	*3*	*4*		
bandwidth	1 s	5.6	18.7	9.7	7.4	41.5	44.0
callbacks	1 s	6.2	19.9	12.2	10.2	48.5	54.1
bandwidth	3 s	5.8	14.5	6.9	3.2	30.4	32.4
callbacks	3 s	6.6	18.2	9.4	8.9	43.1	47.6

Table I: Results for adaptation tests. *Some results for the two adaptation schemes and poll delay values (the "poll" column) are shown. The "total" column gives the total throughput of successfully delivered messages, and the "bw" column gives the total bandwidth used (including incomplete messages).*

We ran experiments using these two adaptation schemes and a number of values for the poll delay parameter. Using a bandwidth trace results in a high degree of time-dependence in the behavior of the algorithms, which makes direct comparisons under identical circumstances problematic: for brevity, we only present a few representative results in Table I. Three of these test cases, showing the mode of operation of the application over time (or the highest level it sends at over time), appear in Figure 6.

Graphs 6(a) and 6(b) show that the poll delay has an appreciable effect on bandwidth utilization. Slowing down the rate at which the bandwidth estimate rises may reduce the instability of the mode the application operates in (see the mode lines for 250-260 seconds in the two graphs), but also leads to a slow reaction when the bandwidth rises suddenly, since the application



(a) (b) (c)

Figure 6: Effects of adaptation schemes on a mixed-mode application. *These graphs show two bandwidth curves: the upper curve is the actual available bandwidth from the trace shown in Figure 1, and the lower curve is the bandwidth used by the application. Additionally, horizontal lines indicate intervals when the application was operating in each of the four modes: a mode line corresponds to the bandwidth required to deliver all messages in that mode (for callback adaptation, this shows the highest level for which a message was delivered during each second). Graphs (a) and (b) show examples of the "bandwidth-only" adaptation scheme with poll delays of 1 and 3 seconds. Graph (c) shows an example of the "callbacks+priorities" adaptation scheme with a poll delay of 1 second. The true bandwidth curve is smoothed for clarity in these graphs, though not in the actual tests.*

waits until it gets a bandwidth notification before it starts exploiting the new level. As graph 6(c) shows, the callbacks+priorities scheme is less reliant on the bandwidth estimate, since it always has a large number of messages which it can potentially send. Comparing the amount of data transferred at each level reveals that the callbacks+priorities scheme sends more data successfully for both poll delay values, at the cost of a higher overhead in wasted bandwidth, while the modal scheme is more conservative. If the higher overhead can be tolerated, a mixed-mode adaptation scheme therefore appears more appropriate for bursty applications.

V. Experiments

We have compared the performance of three implementations of NAI, ATP and two simpler TCP-based implementations. Two sets of experiments were performed. First, ATP's performance for bulk data transfer was measured, without making use of NAI's extended semantics, and second, ATP and NAI-over-TCP were compared for a number of workloads incorporating callback timers and priorities. We describe the experimental setup and methodology before presenting the results of the experiments.

A. Experimental setup

We ran our experiments on an Aironet IEEE 802.11b wireless subnet with a single base station, which was attached to a Ethernet switch. A 1 GHz Celeron desktop computer running FreeBSD 4.5 served as the receiver. To minimise effects of contention on the wired network, it was attached directly to the switch. The sender was an 800 MHz Pentium III laptop, also running FreeBSD 4.5, and communicating through an Aironet wireless Ethernet card. The advertised throughput of the Aironet card is 11 Mbps, though in normal use we never saw more than 4.9 Mbps. When there were no obstructions or major sources of interference between the card and base station, data rates of between 5.6 and 7.4 Mbps were observed.

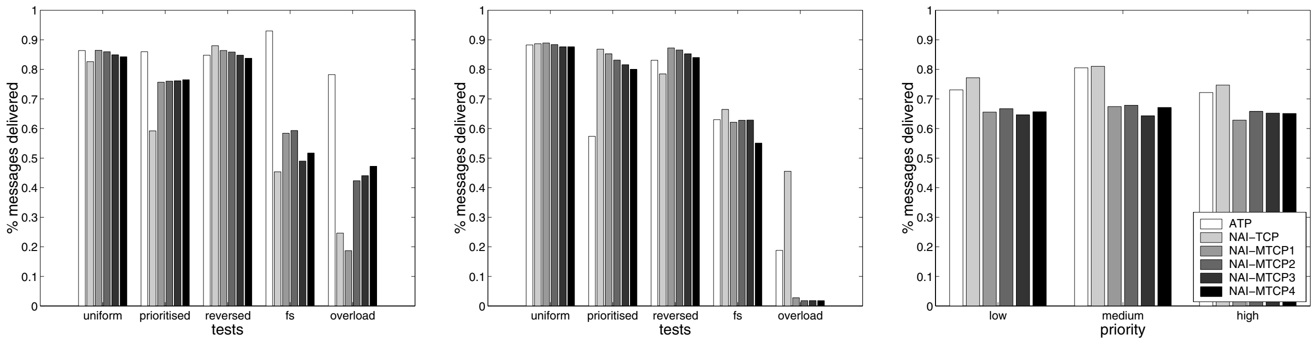
In order to achieve repeatable results for experiments, we used the FreeBSD Dummynet traffic shaping module [13] to control the bandwidth and round-trip time at the sender according to a trace file. As we have described in Section IV-D, we have found that using real traces of bandwidth in a very high variability in experimental results. To eliminate some of this variability, we used a simplified, synthetic bandwidth trace which was already shown in Figure 4.

B. Experimental methodology

We compared the ATP implementation of NAI with two TCP implementations, which incorporate priorities and callback timers, while excluding more complex features of ATP, such as the bandwidth estimator. Both TCP implementations of NAI run at the user level, but differ in their degree of sophistication:

TCP with timers ("NAI-TCP"). The sender opens a single, blocking connection to the receiver. New messages wait on a queue to be sent. Whenever a send call returns, the first queued message is sent if less than half of its timeout has expired, otherwise it is discarded (this heuristic was intended to protect against backlogs when bandwidth was very low). Send operations are irrevocable. Once a send completes, the time is compared against the timer to see if it was delivered within the timeout. The receiver sends a 1-byte user-level acknowledgement to the sender upon receipt of each message, so as to eliminate the effect of TCP sends returning early when data is still buffered in the kernel. NAI-TCP is implemented in two hundred lines of C code.

Multi-stream TCP ("NAI-MTCP"). Multiple messages can be sent concurrently: one connection is opened for each priority level, and the number of high-priority connections is additionally controlled by a parameter (we refer to NAI-MTCP1 for NAI-MTCP with one high-priority connection, NAI-MTCP2 for



(a) High-priority message delivery (b) Low-priority message delivery (c) Random workload

Figure 7: Performance of NAI implementations compared. *Graphs (a) and (b) show the relative proportions of high- and low-priority messages delivered by ATP, NAI-TCP and NAI-MTCP for each workload, normalised by the total number of messages of the appropriate priority in the workload. Graph (c) does the same for the three priority levels in the "random" workload. Each bar shows the average of five trials.*

two high-priority connections, and so on). Behavior is otherwise the same as TCP with timers (sends cannot be aborted), though MTCP can terminate transmission of a message early if its timer expires. NAI-MTCP consists of two thousand lines of C code, compared to ten thousand lines for ATP.

C. Bulk data transfer

The raw performance of ATP was measured by a series of throughput tests, using message sizes starting at 1 KB, and increasing by powers of two up to 1 MB. Inter-arrival spacing was negligible, and the duration of the test was set so that each test transferred a total of 64 MB. All the messages were given a uniform callback timeout long enough to ensure that they would be sent without the risk of being rejected, and bandwidth was set to 512 KB/s. The experiments were conducted using only synchronous or only asynchronous sends. For comparison, the same experiments were run using kernel TCP with a single connection.

The results of these tests were extremely uniform for both ATP and TCP. Both protocols had a peak throughput of 512 KB/s, with very little variation once the peak was achieved (differences of at most one IP datagram from one second to the next). The only significant difference between ATP and TCP was in startup time: TCP doubles its send window size after every round-trip, so it was able to reach the peak bandwidth within a second. ATP is more pessimistic, only increasing the amount of data it sends once per second; it took about ten seconds for the bandwidth usage to reach 512 KB/s. Because the sender and receiver have fast CPUs, the selection of synchronous or asynchronous sends made no difference to ATP performance, despite the fact that asynchronous calls allow pipelining of sends, while synchronous calls are blocking.

D. Priority and deadline workloads

To compare the three implementations of NAI, we used six workloads, each mixing messages of different priorities and callback timeouts, as shown in Table II. When workloads were tested over ATP and NAI-MTCP, asynchronous message transmission was used, but NAI-TCP supports only synchronous

test name	*priority*	*size*	*timer*	*delay*	*n*
uniform	high	4 KB	1 s	0.5 s	600
	high	32 KB	4 s	2 s	150
prioritised	high	4 KB	1 s	0.5 s	600
	low	256 KB	16 s	8 s	38
reversed	high	64 KB	16 s	8 s	75
	low	4 KB	1 s	0.5 s	600
filesystem	high	64 B	0.5 s	0.1 s	3000
	low	64 KB	1 s	1 s	300
overload	high	16 KB	1 s	0.25s	1200
	low	64 KB	1 s	0.5s	600
random	all	1-64 KB	0.5-4 s	0.5 s	600

Table II: Parameters for the priority and deadline tests. *Messages in each workload are divided by priority. The columns for "delay" and "n" give the inter-arrival spacing and the number of messages, respectively. Sizes and callback times for the random test are distributed uniformly within the ranges indicated.*

transmission. As a consequence, ATP and NAI-MTCP are able to preempt low-priority messages with higher-priority ones.

All of the workloads have at least two classes of messages, a high-priority and a low-priority class. The objective of these tests is to measure how many high-priority messages each protocol can deliver before their send timers expire. A secondary consideration is how many low-priority messages are delivered, since a trivial protocol might refuse to deliver all low-priority messages! The "uniform" test sends all messages at the high priority, while the "prioritised" test sends small messages at high priority and larger ones at low priority. The "reversed" test reverses these priority assignments. The "filesystem" workload is intended to model a mixture of large file chunk retrievals and cache validation calls, as might be encountered in a typical distributed file system. The "overload" test sends messages at a very high data rate. Finally, the "random" test is unusual in that characteristics for its messages were generated randomly according to a uniform distribution, and priorities were randomly selected from low, medium or high levels (resulting in 205, 191 and 204 messages respectively). All the tests use the trace of bandwidth described earlier, which varies bandwidth between 0 and 200 KB/s, and lasts for five minutes. While these workloads are not realistic, together they serve to provide an indication of how the three NAI implementations perform under various conditions.

Figures 7(a) and 7(b) plot the results of the first five tests, according to message priority. In the uniform and reversed tests, there is no prioritisation, or the benefit of prioritisation is small, due to large timer values for high-priority messages. Here the differences between the implementations is minor. However, in cases where there is significant contention for bandwidth (the filesystem and overload tests, and to a lesser extent, the prioritised test), ATP performs the best, since it is able to devote all available bandwidth to sending high-priority messages. In contrast, NAI-TCP ignores priorities, and NAI-MTCP always devotes one connection to low-priority messages (though the more high-priority connections it has, the less bandwidth low-priority messages will receive).

Comparing the performance of ATP and NAI-MTCP reveals the disadvantage of the MTCP design: it is subject to contention between the concurrent streams, and the best number of streams to assign to high-priority messages varies for different workloads. The prioritised test delivers roughly the same number of messages irrespective of the number of high-priority streams, but the performance for the reversed test degrades with more streams, while that of the overload test increases. The critical factor is whether transmitting multiple messages concurrently can result in all of them being dropped when their send timers expire. The file system and overload workloads demonstrate this phenomenon most strongly, as ATP significantly outperforms NAI-MTCP because it always sends messages serially. It can also vary the order of message transmission, and interrupt transmission of a low-priority message when a higher-priority message arrives. The poor performance of NAI-MTCP in delivering low-priority messages in the overload test is due to the fact that NAI-MTCP is unable to discard a message early if it is undeliverable before its timer expires: without estimating the available bandwidth, the protocol cannot avoid wasting bandwidth on such a message.

Figure 7(c) shows the results for the "random" workload. NAI-TCP slightly outperforms ATP at two priority levels: here the fact that it ignores priorities proves to be a benefit, since it does not preempt messages and so does not devote bandwidth to a low-priority message, only to discard it when a higher-priority message arrives. NAI-TCP outperforms ATP by 2.5% in delivering high-priority messages, due to ATP rejecting messages at points where it underestimates the current bandwidth. As in the filesystem test, NAI-MTCP's use of multiple streams results in multiple messages being transmitted concurrently.

To summarise, in most of the cases considered, both NAI-MTCP and ATP represent an improvement over NAI-TCP in the proportion of high-priority messages delivered. However, ATP is able to outperform NAI-MTCP by a factor of 30% or more for some workloads, including network communication typical of a distributed file system. ATP is also able to abort a message early when it discovers that insufficient bandwidth exists to deliver it before its callback timer expires. Additionally, as we have shown in Section IV-A, there are conditions under which TCP fails to provide a fair bandwidth division between concurrent streams, which may undermine the effectiveness of the callback timer implementation in NAI-MTCP.

VI. Related work

While there has been a great deal of research in adapting applications and protocols to variations in network characteristics, and in bandwidth-division algorithms, for the sake of brevity we will mention only a few related projects.

We have already proposed web browsing [2] and remote file systems [4] as specific applications which can adapt to bandwidth availability; several systems provide more general adaptation mechanisms to applications. Odyssey [3] allows applications on a mobile host to adapt to changes in availability of many kinds of resources; the bandwidth callback mechanism in ATP is copied from Odyssey's upcalls. Rover [14] focuses on placing components of mobile applications to control communication between mobile clients and servers. ATP has some similarities to Rover's Queued RPC. HATS [15] regulates the transmission of documents over a bandwidth-constrained link, dividing them into hierarchies of data units and allowing policies for scheduling retrieval of particular types of data units to be set for the entire system.

ATP incorporates a simple scheme for a host to determine the bandwidth available to it on a wireless link; more sophisticated schemes exist, particularly for estimating bandwidth along a path in a wide-area network [16], [17]. The Odyssey system incorporates a scheme for determining available bandwidth by comparing the expected time to transmit and acknowledge a message with its actual transmission time [18].

The problems with bandwidth allocation between multiple flows discussed in Section IV-A are not new. T/TCP [19] and TCP Fast Start [20] improve TCP performance for small data transfers by caching and reusing connection state. Henderson et al [21] have investigated the effects of TCP algorithms on bandwidth allocation between concurrent connections. Congestion Manager [10] and Ensemble-TCP [22] share state information between connections to a remote host and allow the aggregate bandwidth to be divided among state-sharing connections by a priority mechanism. TCP Nice [23] adjusts TCP's congestion control algorithm to ensure that "background" TCP flows have lower priority than regular flows. These systems improve performance without compromising TCP congestion control. In contrast, ATP aims to provide an mechanism which allows applications to adapt bursty network communication to bandwidth availability, and to express timing requirements. ATP over TCP allows ATP to be used in a WAN while remaining TCP-friendly.

VII. Conclusion

We have described NAI, a network-aware API for adaptive applications running on a wireless host, which allows an application to be informed of the state of the network and the messages it sends, and to adjust its behavior accordingly. We have also described our ATP implementation of NAI and how it adjusts to changes in bandwidth, as well as demonstrating that an application using ATP can accurately match its "mode of operation" to the available bandwidth. Finally, we have compared alternatives for implementing NAI: using a bandwidth estimator and a reliable datagram protocol over UDP (ATP), or alternatively, over multiple TCP channels (MTCP). We favor the ATP implementation because it has superior performance and predictability, but in settings where UDP is inappropriate, the MTCP implementation is an acceptable alternative, despite the contention effects described in Section IV-A.

As we have illustrated in Section IV-D, ATP enables a high quality of adaptation for applications, and adaptation using priorities can provide better performance than a traditional modal adaptation scheme. The fundamental premise motivating our work has been that this type of priority-driven adaptation is vital in developing more intelligent mobile applications, and ATP appears to be a suitable and effective basis on which to build them.

Future work on ATP includes comparing the appropriateness of using TCP against the advantages of SPP (our reliable datagram protocol), in order to adapt ATP to operate in a wide-area network. We also intend to investigate techniques for making ATP's bandwidth estimator more accurate and responsive in detecting bandwidth increases. ATP is currently being used in the development of an adaptive distributed file system for mobile clients.

ACKNOWLEDGEMENTS

We would like to thank Alan Demers, Emin Gün Sirer, Robbert van Renesse, Eva Tardos and Werner Vogels for many comments and suggestions over the course of this work. We also thank Venugopalan Ramasubramanian, Indranil Gupta, Ranveer Chandra and Rimon Barr for helpful discussions and comments on this and earlier versions of the text of this paper.

REFERENCES

[1] Brian D. Noble, M. Satyanarayanan, Giao T. Nguyen, and Randy H. Katz, "Trace-based mobile network emulation," in *Proceedings of the ACM SIGCOMM '97 Conference*, Cannes, France, Sept. 1997, pp. 51–62.

[2] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir, "Adapting to network and client variation via on-demand dynamic distillation," in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, Massachusetts, Oct. 1996, pp. 160–170.

[3] Brian D. Noble and Mahadev Satyanarayanan, "Experience with adaptive mobile applications in Odyssey," *Mobile Networks and Applications*, vol. 4, no. 4, 1999.

[4] James J. Kistler and M. Satyanarayanan, "Disconnected operation in the Coda file system," *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 3–25, 1992.

[5] David D. Clark and David Tennenhouse, "Architectural considerations for a new generation of protocols," in *Proceedings of the ACM SIGCOMM '90 Conference*, Sept. 1990.

[6] Alan Demers, Srinivasan Keshav, and Scott Shenker, "Analysis and simulation of a fair queueing algorithm," in *Proceedings of the ACM SIGCOMM '89 Conference*, Austin, Texas, 1989, pp. 1–12.

[7] K. Ramaritham, J. A. Stankovic, and P. Shiah, "Efficient scheduling algorithms for realtime multiprocessor systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 2, pp. 184–194, 1990.

[8] Hari Balakrishnan, Venkat Padmanabhan, Srinivasan Seshan, and Randy H. Katz, "A comparison of mechanisms for improving TCP performance over wireless links," *IEEE/ACM Transactions on Networking*, vol. 5, no. 6, pp. 756–769, Dec. 1997.

[9] A. Lahanas and V. Tsaoussidis, "Experiments with adaptive error recovery strategies," in *Proceedings of IEEE ISCC 2001*, July 2001.

[10] D. Andersen, D. Bansal, D. Curtis, S. Seshan, and H. Balakrishnan, "System support for bandwidth management and content adaptation in Internet applications," in *Proceedings of 4th Symposium on Operating Systems Design and Implementation*, San Diego, CA, Oct. 2000, pp. 213–226.

[11] Van Jacobson, "Congestion avoidance and control," in *Proceedings of the ACM SIGCOMM '88 Conference*, Aug. 1988.

[12] Srinivasan Keshav, *An Engineering Approach to Computer Networking*, pp. 424–426, Addison-Wesley, 1997.

[13] Luigi Rizzo, "Dummynet: a simple approach to the evaluation of network protocols," *ACM Computer Communication Review*, vol. 27, no. 1, Jan. 1997.

[14] Anthony D. Joseph, Joshua A. Tauber, and M. Frans Kaashoek, "Mobile computing with the Rover Toolkit," *IEEE Transactions on Computers: Special issue on Mobile Computing*, vol. 46, no. 3, pp. 337–352, Mar. 1997.

[15] Eyal de Lara, Dan S. Wallach, and Willy Zwaenepoel, "HATS: Hierarchical adaptive transmission scheduling for multi-application adaptation," in *Proceedings of the 2002 Multimedia Computing and Networking Conference*, San Jose, California, Jan. 2002.

[16] Manish Jain and Constantinos Dovrolis, "End-to-end available bandwidth: Measurement methodology, dynamics, and relation with TCP throughput," in *Proceedings of the ACM SIGCOMM 2002 Conference*, Aug. 2002.

[17] Kevin Lai and Mary Baker, "Nettimer: a tool for measuring bottleneck link bandwidth," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Mar. 2001.

[18] Brian D. Noble, *Mobile Data Access*, Ph.D. thesis, Carnegie Mellon University, May 1998.

[19] R. Braden, "T/TCP – TCP extensions for transactions functional specification," RFC 1644, Internet Engineering Task Force, July 1994.

[20] V. Padmanabhan and R. Katz, "TCP Fast Start: a technique for speeding up web transfers," in *Proceedings of Globecom 1998*, 1998.

[21] T. H. Henderson, E. Sahouria, S. McCanne, and R. H. Katz, "On improving the fairness of TCP congestion avoidance," in *IEEE Globecom Conference*, 1998.

[22] Lars Eggert, John Heidemann, and Joe Touch, "Effects of Ensemble-TCP," *ACM Computer Communication Review*, vol. 30, no. 1, pp. 15–29, Jan. 2000.

[23] Arun Venkataramani, Ravi Kokku, and Mike Dahlin, "TCP Nice: A mechanism for background transfers," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*. USENIX Association, Dec. 2002.