Mobility-aware adaptation for transport protocols

Benjamin Atkin and Kenneth P. Birman
Department of Computer Science
Cornell University, Ithaca, NY
{batkin, ken}@cs.cornell.edu
February 1, 2002

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 variability. We examine adding priorities and deadlines to data objects as a method for achieving greater application control. Priorities allow the application to control the transmission order of objects, while deadlines permit the transport protocol to inform the application if there is insufficient bandwidth for delivery. We compare three implementations of priorities and deadlines, two extending TCP at the application level, and one new protocol, ATP, which runs on top of UDP. Experiments show that ATP and the TCP extensions perform comparably in many cases, but ATP outperforms TCP for some workloads, including a workload simulating remote filesystem traffic.

1 Introduction

Mobile hosts in wireless networks can frequently experience variable network performance, due to local interference, contention from other hosts, or distance from a base station. Since this variability is more prevalent and more unpredictable than in wired networks, applications written for mobile hosts should be capable of adapting to changes in network behavior. For instance, an application could rely heavily on the network when connectivity is good, but scale back its communication if connectivity degrades. Adaptation of this sort requires a greater degree of information and control than is offered by the Transmission Control Protocol (TCP) and the BSD socket interface. The addition of deadlines to the transmission of data objects allows a transport protocol to inform the application if bandwidth is insufficient to deliver an object. Adding priorities allows the application to control how the transport protocol uses the available bandwidth. This paper explores alternatives for implementing priorities and deadlines in the transport protocol and compares their performance in a variety of scenarios.

We focus on message-oriented communication, since applications such as web browsers or remote filesystem clients commonly send and receive objects of various sizes, rather than streams of data. Three approaches are considered: two based on extending TCP at the application level, and one new transport protocol which runs over UDP. Though TCP is almost ubiquitous, application adaptation to network behavior over TCP can be awkward to implement, since TCP automatically and transparently adapts to bandwidth variations. Transmitting multiple data objects requires choosing how many TCP connections to use: a single connection forces serial transmission, whereas multiple connections can compete unpredictably for bandwidth. Our basic extension of TCP with deadlines uses a single connection, and the more advanced variant uses multiple connections dedicated to specific object priorities.

Our third protocol does not rely on TCP. The Adaptive Transport Protocol (ATP) is designed to fulfill the same role as TCP within a local area network, and incorporates additional semantics for mobility-awareness. An application using ATP is informed of the state of the network as it changes, and can use this information to decide which data to send, adjust priorities, and suspend, resume or cancel ongoing data transmissions in response to bandwidth changes. ATP autonomously decides how to schedule data objects for transmission, based on bandwidth availability, and notifies the application by an upcall if an object cannot be transmitted within the expected time. Our experiments show that ATP performs comparably to the TCP variants for many workloads. We have identified a class of workloads, including remote file access, where ATP significantly outperforms TCP.

The structure of the rest of this paper is as follows. Section 2 gives an overview of ATP, its semantics, and how it differs from TCP. Section 3 describes our implementation and the algorithms used by the protocol, as well as discussing some examples of execution. Section 4 compares the performance of ATP to TCP, and user-level extensions of TCP, in a series of experiments. Section 5 surveys related work, while section 6 concludes and describes our plans for future work.
2 Overview of ATP

ATP is a reliable, message-oriented transport protocol. It implements its own rate-based flow control scheme and retransmissions using selective acknowledgements. ATP incorporates bandwidth estimation through a combination of throughput readings from the network card and a packet-pair estimate.

The protocol is most suited for use by applications running on a mobile host, but communicating with remote hosts, in a way which permits trade-offs between communication and data quality. For instance, a web browser might request degraded images to conserve bandwidth, or a file system might reduce prefetching or defer writing back cached files. In the Odyssey system [21], these techniques are termed “data fidelity”, and applications change modes of operation based on the available bandwidth. ATP generalises this approach, allowing the application to choose the mode it runs in according to bandwidth, but also masking transient variability in bandwidth by doing its own, finer-grained adaptation.

ATP replaces TCP for communication within a local area network. Balakrishnan et al [3] have demonstrated that performance improvements in TCP can be achieved by augmenting the protocol with knowledge of the wireless link. We take a similar approach, but we change the transport protocol semantics, as well as specialising the implementation. While this prevents an ATP-enabled host from communicating with hosts on an outside WAN, the limitation can be rectified by interposing an intermediary ATP-to-TCP gateway.

To communicate with another host, an application creates a local “endpoint” to identify the destination, and then makes calls to send data objects. Sending an object can be either a blocking or non-blocking operation, so that an application can have multiple concurrent send operations outstanding at one time. To aid ATP in deciding the order to send messages in, sends can be tagged with a priority and an advisory deadline.

Data objects can be of arbitrary size. ATP handles fragmentation and reassembly of large objects.

Figure 1 illustrates ATP’s structure. ATP is implemented as a library which applications link with, running at user level over kernel UDP. A future version may be incorporated into the kernel for greater efficiency. We describe ATP’s interface and differences from TCP in detail in subsequent sections.

2.1 ATP interface

The basic ATP interface is quite similar to the BSD socket interface, as shown in Figure 2. Though its aims are different, the interface and general architecture of ATP also have considerable similarities to U-Net [29] and VIA [28], incorporating the same mechanisms of endpoints, message queues and asynchronous send operations. To initiate communication with another host, an application calls endpoint, supplying the host’s name or IP address, and receives an endpoint identifier. Alternatively, a server program can use accept to wait for new clients to begin communicating, which also returns an endpoint identifier.

Objects are sent by a blocking or non-blocking call, which takes an object and an options field as arguments. Options encapsulate the optional priority and deadline for delivery of an object. A priority is chosen from a finite range, and a deadline specifies the maximum number of milliseconds between receipt of the object by ATP and the time when the protocol should stop attempting to send it and report a failure. Each send operation is given a request identifier, which can be passed to reqctl in order to examine or modify the request while it is in progress. Permitted commands are to check the amount transferred, cancel the request, suspend the transfer, restart the transfer, or change its priority or deadline. The recv call blocks waiting for a data object to arrive at the endpoint.
These functions can be augmented by the addition of callbacks, which allow ATP to notify the application of exceptional conditions. The `callback` function registers a callback routine which ATP will invoke when the bandwidth strays outside the interval specified in the arguments to `callback`. The application-supplied routine can then make adjustments based on the new bandwidth level, and potentially re-register with new bandwidth bounds. ATP employs hysteresis to prevent a cascade of callback invocations when the bandwidth is unstable. Non-blocking send operations\(^1\) can also be accompanied by a callback function, which is invoked when the transfer completes, or on an exceptional condition (insufficient bandwidth to complete request, deadline expired, and so on). The addition of callbacks makes ATP object deadlines “advisory”: if an object cannot be delivered within its deadline, the application receives a callback, and can decide to terminate the transfer, or set a new deadline.

### 2.2 Priorities and deadlines

The interaction of priorities, deadlines and bandwidth is conceptually straightforward. Objects larger than the network’s Maximum Transmission Unit (MTU) are sent in fragments, and the next fragment to send is selected based on priorities and deadlines of objects. ATP always selects a fragment from among the objects with the highest priority. If there is more than one object with that priority, it will choose a fragment from the object with the earliest deadline. Objects are ranked by the order in which they will be sent, and a simple procedure determines whether delivery of an object is possible with the current bandwidth (we defer discussion of this algorithm until section 3). If the deadline for an object cannot be satisfied, its callback is invoked and its transmission is suspended. Using object priority to select fragments for transmission naturally raises the possibility of starvation, which is discussed in the next section.

The problem of choosing the order in which to send objects is similar to real-time CPU scheduling [19], and the scheme we have implemented is a refinement of the Earliest-Deadline First algorithm. We recognise that a case could be made for a more elaborate scheduling policy. However, the current ATP policy permits high-priority objects to block transmission of low-priority objects, whereas many priority-based schedulers opt to reserve a fraction of system resources for low-priority tasks. A greater degree of sophistication lies beyond the scope of our prototype, though the ATP scheduling module could easily be replaced with a different mechanism. One possible addition to the scheduler would be support for precedence constraints to refine the order in which objects are scheduled.

A simple application such as a web browser might choose to use ATP as follows: web pages are retrieved at a high priority, and embedded images are retrieved at a lower priority. As the available bandwidth decreases, the amount of bandwidth devoted to retrieving web pages will remain constant for as long as possible, while image bandwidth will decrease. Deadlines can be set for images to automatically prune an image request after a certain period of time has passed without the image being retrieved. Naturally, there is no reason to put all images at the same priority, headings or image maps might be put at a higher level than inessential images. While ATP does not provide a byte stream interface, streaming data can be mimicked by a suitable choice of deadlines for some division of the stream into objects, and relying on ATP’s internal deadline ordering.

### 2.3 Object queues

In practice, a scheme as simple as we have described is of restricted value. While it is useful for an isolated application, with multiple applications, it is possible for a single one to

\(^1\) Callbacks are superfluous for blocking send operations, since they can simply return with an error code.
3 Implementation

Having explained how ATP behaves, we now describe some of the implementation in more detail. Our initial implementation of ATP is on FreeBSD 4.2, and amounts to approximately seven thousand lines of C code. Three areas are of particular interest: the bandwidth estimation mechanism, the network scheduler for sending data, and how ATP reacts to changes in bandwidth.

3.1 Bandwidth estimation

A mechanism for 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. ATP incorporates a bandwidth estimator to determine how much data it should send each second. Much work has been devoted to the general problem of estimating bandwidth for flows in a wide-area network [16]. However, bandwidth estimation for ATP has some crucial differences from the traditional problem, which make it both easier and more difficult.

Bandwidth estimator requirements

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. In the prototype, we assume that traffic from protocols other than ATP constitutes a negligible fraction of the total traffic.

A further important difference is a side-effect of ATP semantics. Since ATP incorporates priorities for objects, an inaccurate estimate can cause a priority inversion. The difficulty lies in the network card’s device driver buffering 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 objects, these will hold up the transmission of datagrams from high-priority objects 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 mechanism to “back off” and reduce its estimate when it detects a backlog in the send queue.
The bandwidth estimation algorithm

ATP’s bandwidth estimator has similarities to TCP window size adaptation, though it derives a single estimate for all communication over the wireless link, and operates independently from the rest of the protocol. TCP adjusts the window size for a connection to match the bandwidth-delay product along the path, and relies on the rate at which acknowledgements arrive to trigger changes in the window size. The generality required of TCP’s window size adaptation mechanism precludes deriving a bandwidth estimate for the mobile host’s outgoing interface.

Estimation in ATP relies on three statistics: the observed bandwidth, the length of the network interface send queue, 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 ten packets; the maximum length allowed is fifty).

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 \( curbw \) value is the estimate of available bandwidth on the wireless link, which is used by ATP, as well as being supplied to the application. The \( available \) value is the amount of bandwidth which ATP should restrict itself to using over the next second. This is an internal ATP statistic, which may be lower than \( curbw \) if the network interface send queue is nonempty and there is a consequent risk of priority inversion. An averaging filter with a constant window size is used to smooth \( curbw \) in order to make it less sensitive to transient spikes.

The bandwidth estimator maintains its estimate by adjustments based on the count of bytes sent and received over the preceding second, and using a packet-pair measurement. Under normal circumstances it relies on the former, calling band-
width_estimate with the bandwidth usage and backlog (send queue length) reported by the kernel. If the backlog is nontrivial, curbw is reduced to the bandwidth reported, and available is reduced so as to clear the backlog. Otherwise, if the reported bandwidth exceeds curbw, then curbw is adjusted and available is speculatively increased by a constant in the expectation that the true bandwidth has not been attained. If the increase overshoots the true bandwidth, then the estimate will be rectified in the next second. The degree of the increase is limited by half the maximum send queue length times the MTU, but is reduced when the bandwidth estimate is very small.

Packet-pair estimation

The packet_pair routine is only used if the current estimate is known to be stale: this is equivalent to staleness exceeding a small threshold (five seconds in our implementation). In this case, the bandwidth estimator sends a pair of marked, MTU-size packets to a remote host, with no pause between the send calls. The receiving host measures the inter-arrival time of the packets and sends this value in a reply. The reciprocal of the inter-arrival time gives an order-of-magnitude estimate of the available bandwidth. The period between packet-pair estimates is adjusted using a simple heuristic: if a stale bandwidth estimate is causing objects to be rejected, then the period is decremented following every measurement; otherwise, it is incremented. While Figure 3 implies that send_packet_pair returns to the bandwidth estimator and execution resumes, in fact the bandwidth is only updated if and when the reply packet arrives.

Even though the inter-arrival time estimate can be quite inaccurate, it is useful in breaking a particular kind of deadlock which arose in an early version of ATP. The problem was as follows: if bandwidth fell to a low enough level, the network scheduler began to reject objects because their deadlines were too tight. As a result, ATP ceased transmitting objects and bandwidth_estimate was unable to recognise an increase in the available bandwidth, since the reported byte count was stuck at zero. The addition of packet-pair estimates solved this problem. An alternative approach would be to speculatively accept objects even when their deadlines appear to be unsatisfiable, anticipating that the true available bandwidth is sufficient to transmit them. This approach was rejected as too wasteful when the estimate was correct, and it is also antisocial, since it could interfere with other hosts on the network when conditions are poor. While our packet-pair estimate is less accurate, gross errors can be rapidly corrected, once the next byte count from the network card is obtained. The interaction of the bandwidth estimator and the object admission scheme is described in a following section.

3.2 Endpoints and the network scheduler

The network scheduler component of ATP deals with objects, not with individual packets. Each endpoint contains the data structures to handle transmission of objects to its corresponding remote host: this includes object data, and send and receive windows for packets. Our endpoint implementation uses a positive acknowledgement scheme, combined with selective acknowledgements to limit retransmissions when multiple packets are lost.

The operations of individual endpoints are transparent to the network scheduler, which just identifies the object which needs to be transmitted most urgently, and asks the endpoint managing that object to supply a packet. It is up to the endpoint to decide whether to retransmit a previous packet, or to supply a packet from the object the network scheduler has selected. When a packet arrives from the remote host, the network scheduler delivers it to the endpoint, which may adjust its send and receive windows accordingly. If a packet acknowledges the successful transmission of an object, then the endpoint informs the network scheduler. ATP then notifies the application that the object has been delivered to the receiver.

The scheduler uses the available value provided by the bandwidth estimator to decide how many packets to send over the course of a second. It divides the estimate by the network MTU to get a packet count, and then determines from this how many packets it can send during each 50 millisecond “send interval”. In this way, packet sends are staggered throughout the second to prevent the device driver queue discarding packets. Of course, if there are no objects available to be sent, the network scheduler is idle. It may also remain idle if the endpoints it polls refuse to supply packets to send, as they might if their send windows are full and they have no packets to retransmit. Under normal circumstances, this is an infrequent occurrence, since endpoints set their window sizes large enough to ensure that the bandwidth estimate, not the window size, is the limiting factor in the rate of transmission.

3.3 Controlling object admission

In the overview of ATP, the priority and deadline features were described from the application’s perspective. Under normal circumstances, the interaction of object priorities and deadlines is exactly as outlined: as long as bandwidth is adequate, priority is the most important criterion for deciding which object will be sent. Among objects of the same priority, the object with the closest deadline will be sent first. The effect is that high-priority objects are sent whenever possible, with lower-priority objects filling in the idle time when no high-priority objects are available.

Ensuring that objects are delivered within their deadlines requires some extra work. A notion of deadline safety is used to decide when objects should be tagged as undeliverable, and
the application notified. The sequence of objects is deadline-safe if a scan of the objects in their expected transmission order calculates that the bandwidth required is no more than the current estimate. The bandwidth required to deliver object $O_n$ is given by

$$\beta_n = \frac{\sum_{i=1}^{n} \text{size}(O_i)}{\text{deadline}(O_n)}$$

The $\beta_n$ values are calculated in sequence as each object is examined, and if a particular $\beta_n$ value is found to exceed $\text{curbw}$, then an “insufficient bandwidth” upcall for the corresponding object $O_n$ is made. It is up to the application to decide whether to abort the object transfer or to defer it until the bandwidth improves (this course might be preferable if the object has already been partially transferred).

Deadline safety is checked when the value of $\text{curbw}$ decreases. As an optimisation, the deadline check stores the final $\beta_n$ value for later comparison against $\text{curbw}$, which remains valid until a new object has been accepted for transmission. ATP also controls the admission of new objects by running the deadline-safety check as though a candidate object was already added. The new object is rejected if deadline safety requires that its addition would cause another object of the same priority, but a later deadline, to be discarded.

### 3.4 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 [25] (the bandwidth curves are explained in more detail in section 4).

Graph (a) shows the bandwidth curves when the system is saturated: every 0.25 seconds, a new 64 KB object enters the system, and must be delivered within a second. Between 120 and 220 seconds, almost no objects 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 an inaccurate packet-pair calculation; even though this causes some objects to be incorrectly admitted, it is quickly rectified. The accompanying spike, at which more data than the available bandwidth is reported as sent, appears to be an inaccuracy in Dummynet.

Graph (b) shows a mixed workload, with both high-priority and low-priority objects. Every 0.5 seconds, a high-priority, 4 KB object with a 1-second deadline enters the system; every 8 seconds, a low-priority, 256 KB object with a 16-second deadline enters. The spikes in the reported bandwidth curve indicate these larger, low-priority objects. The combination of smaller objects and looser deadlines 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 object 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 objects into the system and their delivery to the receiver. The small, high-priority objects arrive every 0.5 seconds, while a large object 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 object and its delivery lengthens as bandwidth decreases, and some small objects are dropped due to their deadlines expiring. It is worth noting that some arrival-delivery intervals are greater than one second, the nominal deadline, but this is due to the deadline only being enforced at the sender’s side, not at the receiver, in order to avoid a requirement for clock synchronisa-
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 objects into the system are denoted by “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 object commences at the sender and receipt occurs at the receiver. Objects after 148.5 seconds, and removal of objects due to deadlines, have been omitted – an object without an arrow was dropped due to a deadline.

While the bandwidth is technically sufficient for delivery of all the small objects shown on the timeline, a clamp-down on the available value after 138 seconds leads to some objects exceeding their deadlines. Insufficient bandwidth at 145 seconds causes a new 256 KB object to be rejected, and the 256 KB object admitted at 137 seconds is discarded due to insufficient bandwidth at 147 seconds, before its deadline 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 their purpose is to free buffer space at the receiver.

4 Experiments

We compared the performance of ATP to TCP in two sets of experiments: first, the performance for bulk data transfer was measured, without making use of ATP’s augmented semantics, and second, ATP and TCP were compared for a number of workloads incorporating deadlines and priorities. We describe the experimental setup and methodology before presenting the results of the experiments.

4.1 Experimental setup

We ran our experiments on an Aironet IEEE 802.11(b) wireless subnet with a single base station, which was attached to an Ethernet switch. A 200MHz Pentium Pro desktop computer running FreeBSD 4.2 served as the receiver. To minimise effects of contention on the wired network, it was attached directly to the switch. The sender was an 800MHz Pentium III laptop, also running FreeBSD 4.2, and communicating through an Aironet wireless Ethernet card. The advertised throughput of the Aironet card is 11 Mbps, though “in the field” we never saw more than 4.9 Mbps. Under controlled conditions, 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 [25]. Using Dummynet, we controlled the available bandwidth at the sender according to a trace file. Originally we used traces of real throughput measurements (obtaining results similar those presented by Noble et al [22]), but we found these to be too chaotic to permit a straightforward analysis of our results. Accordingly, we used a simplified, synthetic bandwidth trace to emulate network behavior, already shown in Figure 4.

4.2 Experimental methodology

Each experimental workload was in the form of a trace file of object descriptions (size, priority, deadline) and inter-arrival delays between objects. Regardless of the protocol used, each test program maintained an internal counter to determine when new objects should enter the system.

Comparing TCP and ATP using a workload that incorporates deadlines and priorities requires extra logic for TCP to capture some of ATP’s extended semantics. We used two schemes to extend TCP at the application level:

(i). TCP with deadlines (“TCP”). The sender opens a single, blocking connection to the receiver. When the con-
connection is idle, the sender waits for a new object, if none are queued. If there is a queue of new objects to send, the first one is sent if less than half of its deadline has expired, otherwise it is discarded (this heuristic was intended to protect against backlogs when bandwidth was very low). Discarded objects are logged. Send operations are irrevocable. Once a send completes, the time is compared against the deadline to see if it was delivered in time. The receiver sends a 1-byte user-level acknowledgement to the sender upon receipt of each packet, so as to conform to ATP semantics, which mandate that a send operation is only reported as completed once the object has been received. TCP allows a blocking send to complete once buffer space is available in the kernel to hold the data.

(ii). **Multi-stream TCP (“M-TCP”).** Multiple objects can be sent concurrently, by opening a configurable number of TCP connections to the receiver when a test starts. In our experiments, one connection was opened for low-priority objects, and the number of high-priority connections was varied. Behavior is otherwise the same as TCP with deadlines (in particular, sends cannot be aborted), though M-TCP uses the same interface as ATP.

### 4.3 Bulk data transfer

The raw performance of ATP was measured by a series of throughput tests, using object sizes starting at 1 KB, and increasing by powers of 2 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 objects were given a uniform deadline 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 TCP with a single connection.

The results are shown in Figure 6(a), which graphs the peak throughput for the protocols, so as to exclude the effects of slow start and bandwidth estimation. The line for TCP is roughly constant at 512 KB/s, the maximum available. For ATP there are two curves, “atp-sync” for ATP with synchronous send calls, and “atp-async” for ATP with asynchronous calls. Except at the largest object sizes, ATP with synchronous sends performs worse than TCP, and much worse for small objects, with a peak throughput of about 14 KB/s for 1 KB objects. However, for objects of at least 4 KB, ATP with asynchronous sends is competitive with TCP. A large part of the poor performance of synchronous ATP is due to context-switching overheads: two context switches are necessary for each blocking send operation, which are compounded with the base cost of the kernel calls to send the packets. In addition, there is a protocol overhead which manifests itself for both synchronous and asynchronous sends when the object size is below 4 KB. We conclude that ATP with asynchronous sends is the correct choice for high performance, provided that individual objects are not too small (as a point of reference, the typical UDP packet size used by NFS is 8 KB).

### 4.4 Priority and deadline workloads

To compare ATP and TCP, we used seven workloads, each mixing objects of different priorities and deadlines. When ATP and M-TCP were used, all objects were sent asynchronously. The characteristics of the objects for each test, grouped by priority, are shown in Table 1.

Most of the workloads had two classes of objects, a high-priority and a low-priority class. The objective of these tests was to measure how many high-priority objects each protocol could deliver. A secondary consideration was how many low-priority objects were delivered, since a trivial protocol could refuse to deliver all low-priority objects! The “uniform” test had only high-priority objects, while the “random” test used objects of low, medium and high priorities (205, 191 and 204 objects respectively). Priorities, sizes and deadlines for the random test were chosen randomly from the ranges listed, according to a uniform distribution. The “filesystem” workload was intended to model a mixture of large file chunk retrievals and validation calls, as might be encountered in a typical distributed file system. All the tests used the trace of bandwidth described earlier, which varied bandwidth between 0 and 200 KB/s, and lasted for five minutes.

Figures 6(b) and 6(c) plot the results of the first six tests, with 6(b) showing the proportion of high-priority objects delivered, and the proportion of low-priority objects in 6(c). In both graphs, the values plotted are normalised against the num-

<table>
<thead>
<tr>
<th>test name</th>
<th>priority</th>
<th>size</th>
<th>deadline</th>
<th>delay</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>uniform</td>
<td>low</td>
<td>4 KB</td>
<td>1 s</td>
<td>0.5 s</td>
<td>600</td>
</tr>
<tr>
<td>equal</td>
<td>high</td>
<td>4 KB</td>
<td>1 s</td>
<td>0.5 s</td>
<td>600</td>
</tr>
<tr>
<td></td>
<td>high</td>
<td>32 KB</td>
<td>4 s</td>
<td>2 s</td>
<td>150</td>
</tr>
<tr>
<td>unequal-1</td>
<td>high</td>
<td>4 KB</td>
<td>1 s</td>
<td>0.5 s</td>
<td>600</td>
</tr>
<tr>
<td></td>
<td>low</td>
<td>64 KB</td>
<td>16 s</td>
<td>8 s</td>
<td>38</td>
</tr>
<tr>
<td>unequal-2</td>
<td>high</td>
<td>4 KB</td>
<td>1 s</td>
<td>0.5 s</td>
<td>600</td>
</tr>
<tr>
<td></td>
<td>low</td>
<td>256 KB</td>
<td>16 s</td>
<td>8 s</td>
<td>38</td>
</tr>
<tr>
<td>reversed</td>
<td>high</td>
<td>64 KB</td>
<td>16 s</td>
<td>8 s</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>low</td>
<td>4 KB</td>
<td>1 s</td>
<td>0.5 s</td>
<td>600</td>
</tr>
<tr>
<td>filesystem</td>
<td>high</td>
<td>64 B</td>
<td>0.5 s</td>
<td>1 s</td>
<td>3000</td>
</tr>
<tr>
<td></td>
<td>low</td>
<td>64 KB</td>
<td>1 s</td>
<td>1 s</td>
<td>300</td>
</tr>
<tr>
<td>random</td>
<td>all</td>
<td>1-64 KB</td>
<td>0.5-4 s</td>
<td>0.5 s</td>
<td>600</td>
</tr>
</tbody>
</table>

Table 1: Parameters for the priority and deadline tests. Objects in each workload are divided by priority. The columns for “delay” and “n” give the inter-arrival spacing and the number of objects, respectively. Sizes and deadlines for the random test are distributed uniformly within the ranges indicated.
ber of objects of that priority present in the corresponding workload. For delivering high-priority objects, ATP performs comparably to TCP with deadlines, and significantly outperforms TCP for the unequal-2 and filesystem workloads. This is only to be expected, since TCP ignores priorities entirely. Figure 6(c) shows that this improvement is at the expense of low-priority objects, though the margin between TCP and ATP is mostly small. ATP performs the best in the tests where the difference in size between low-priority and high-priority objects is greatest. In the “reversed” test ATP delivers one high-priority object less than TCP, rejecting the extra object due to insufficient bandwidth, but delivers more low-priority objects.

Examining the relative performance of ATP and M-TCP shows a different pattern. Each experiment was repeated four times using M-TCP, with 1 low-priority stream, 1 medium-priority stream (where appropriate), and 1 to 4 high-priority TCP streams. Allowing M-TCP to exploit knowledge of priorities results in performance which approaches that of ATP for high-priority objects, and is as least as good for low-priority objects. Unsurprisingly, the best choice for the number of M-TCP streams depends on the characteristics of the workload, with no number dominating overall. However, the filesystem workload reveals the difference between ATP and M-TCP, with ATP performing 15-20% better for both priority levels. This is largely due to ATP avoiding the concurrent transmission of high-priority, short-deadline objects with low-priority objects, which M-TCP cannot avoid.

Figure 6(d) shows the “random” workload, in which ATP also outperforms TCP and M-TCP. It differs from the preceding experiments in having three rather than two priority levels, and the sizes and deadlines of all objects being selected according to the same probability distribution. Once again, ATP does better than regular TCP except at the lowest priority level. In comparison with the M-TCP configurations, ATP achieves 10-15% better performance at all priority levels. As in the filesystem test, M-TCP’s use of multiple streams results in multiple objects being transmitted at the same time. At some points this proves to be a significant liability, since it can result in all the concurrently-transmitted objects missing their deadlines. ATP chooses the transmission order autonomously, but transmits objects serially to avoid this phenomenon.

Further insight into the differences between the protocols can be gained by examining the final status of objects in some representative tests. Table 2 gives the status (successfully delivered, rejected on entry, or deadline exceeded) for high-priority objects in four of the tests.

Since TCP cannot admit objects asynchronously, it is very aggressive in rejecting objects. Among TCP tests in Table 2, the random test is the only one in which deadlines for high-
priority objects expired, since in this test most of the objects were large enough to make this likely. ATP and M-TCP reject fewer objects initially, but discard more later, as their deadlines expire when they are queued for transmission. Unlike M-TCP, ATP is able to discard objects early due to insufficient bandwidth, so it may recognise that an object’s deadline will expire without wasting bandwidth unnecessarily. As a result, ATP consumes less bandwidth than M-TCP in these tests (though more than TCP, which also delivers fewer objects).

To summarise, in most cases, both M-TCP and ATP represent an improvement over TCP in the proportion of high-priority objects delivered. However, ATP is able to outperform M-TCP by a factor of 10% or more for some workloads, including network communication typical of a distributed file system.

<table>
<thead>
<tr>
<th>test name</th>
<th>protocol</th>
<th>delivered</th>
<th>rejected</th>
<th>deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>equal</td>
<td>ATP</td>
<td>524</td>
<td>62</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>TCP</td>
<td>497</td>
<td>103</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>M-TCP2</td>
<td>506</td>
<td>69</td>
<td>25</td>
</tr>
<tr>
<td>unequal-2</td>
<td>ATP</td>
<td>522</td>
<td>58</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>TCP</td>
<td>360</td>
<td>240</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>M-TCP2</td>
<td>512</td>
<td>62</td>
<td>26</td>
</tr>
<tr>
<td>fs</td>
<td>ATP</td>
<td>2605</td>
<td>1</td>
<td>394</td>
</tr>
<tr>
<td></td>
<td>TCP</td>
<td>1319</td>
<td>1681</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>M-TCP2</td>
<td>1783</td>
<td>397</td>
<td>360</td>
</tr>
<tr>
<td>random</td>
<td>ATP</td>
<td>162</td>
<td>30</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>TCP</td>
<td>150</td>
<td>43</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>M-TCP2</td>
<td>128</td>
<td>36</td>
<td>37</td>
</tr>
</tbody>
</table>

Table 2: Final status of high-priority objects in four tests. For conciseness, only the configuration of M-TCP with two high-priority streams is shown.

5 Related work

Research in adapting a host to bandwidth variation and other characteristics of wireless networks can be divided into three groups: application adaptation, transport protocol enhancements, and link bandwidth allocation.

Adapting applications to bandwidth variations can be done in a number of ways. One possibility is in the file system, by adjusting to disconnections [14, 24] or low bandwidth availability [20]. Application-specific adaptations can be made if applications are able to change their mode of operation based on available bandwidth, in which case the amount of communication, or data quality, can be correspondingly varied [12, 17, 21]. Another alternative is to divide data accesses made by applications into a hierarchy of tasks or documents and allocate bandwidth among them [7, 10]. ATP is particularly similar to HATS [7], which divides documents into hierarchies of data objects, and allows policies for scheduling retrieval of particular object classes to be specified for the whole system. ATP differs in using lower-level mechanisms, and adding facilities for timely delivery of objects. The design of ATP borrows some ideas from Rover’s Queued RPC [12].

Considerable effort has also been made to adapt transport protocols to the peculiarities of wireless networks. Balakrishnan et al [3], Indirect-TCP [2] and M-TCP [6] (unrelated to our M-TCP) overcome high bit errors and hand-offs, and adjust TCP’s algorithms while attempting to still preserve TCP semantics. WebTP [11] is optimised for HTTP requests, while Mowgli [15] adds different connection modes for TCP connections, logical conditions on object transfers, and priorities for bandwidth allocation between flows. Mobiware [1] unifies the transport layer and application adaptation using modes of operation. Unlike ATP, it is a flow-based, not a message-based system.

Allowing quality of service guarantees for concurrent flows on a single link has been extensively studied [8]. Spring et al [26] control the bandwidth used by TCP streams at the receiver by manipulating kernel buffer sizes. Netnice [23] regulates application bandwidth usage in the kernel. RAMP [18] infers bandwidth requirements for real-time flows. Hierarchical link-scheduling [9, 27] shares bandwidth on a link among protocols or hierarchical classes of flows. Congestion Manager [4] adds unified congestion control between IP and the upper layers of the protocol stack to prevent oscillation of bandwidth allocations between flows. Like ATP, it has an asynchronous interface and callbacks, and schedules one packet at a time, but does not use deadlines or priorities for packets.

6 Conclusion

We have compared three mechanisms for providing adaptation through priorities and deadlines in the transport protocol: two user-level extensions of TCP, and one new protocol, ATP. We have also described the design and implementation of ATP and how it adjusts to changes in bandwidth. Our experiments show that both TCP and TCP with multiple streams give good performance, but that ATP generally performs comparably, and in some cases outperforms the TCP variants. In particular, for filesystem-like workloads, ATP achieves a considerable improvement. We intend to investigate this area further by implementing a filesystem for mobile computers which makes use of ATP’s extended semantics. Further work on ATP will investigate the performance of the protocol when there is contention among multiple hosts in the wireless network.

Acknowledgements

We would like to thank Alan Demers, Emin Gün Sirer, Robert van Renesse, Eva Tardos and Werner Vogels for advice and for pointing out mistakes. We also thank Venugopalan Ramasubramanian for commenting on the final draft, and Indranil Gupta and Rimon Barr for helpful discussions.
References


