Specification for the Adaptive Transport Protocol

Benjamin Atkin

1 Overview of the protocol

ATP is a reliable, adaptive transport protocol intended for use in wireless networks [1]. Unlike TCP, it's a message-based protocol, not a stream-based protocol: it preserves boundaries between records. The idea is that an application wants to send data objects with varying degrees of importance and urgency. It knows what the current bandwidth is, but wants to be able to adapt to changing bandwidth. It encodes its assumptions about available bandwidth in deadlines for the objects it wishes to transmit. If an object cannot be delivered within its deadline, ATP will notify the application, which can decide to abort the transmission, defer it until later, or change the deadline to allow it to proceed.

ATP relies on a bandwidth estimation module to tell it what the current available bandwidth is. This isn't as easy as it sounds, because there is no reliable way of finding out available bandwidth, besides looking at the amount of traffic currently going through the network card. This is a good starting point, but unfortunately it may not be an upper limit.

2 Data types

Endpoints are ATP sockets. An application must open an endpoint for every machine/port pair it wishes to communicate with, just as in TCP, though in ATP there is no explicit connection step. Under the covers, the current implementation of ATP does no checking for port validity.

I/O vectors encapsulate application-level data units for transmission, similar to Unix-style I/O vectors. An I/O vector points to a variable number of memory regions, each with an accompanying length. A valid vector must be passed as an argument to a send operation, and an empty I/O vector must be filled in for a receive. In the kernel implementation, Unix I/O vectors should be used in place of ATP I/O vectors.

Since the ATP prototype runs in the same address space as the application, it's not necessary to copy data into the application address space. Kernel ATP will require the application to specify the buffers to copy received messages into explicitly. This introduces the possibility that the I/O vector the application passes doesn't have enough room for the message the kernel wants to deliver. Dealing with this problem is discussed in more detail in section 4.1.

Objects are what the application wants to send. ATP maintains objects to encapsulate a unit of application data and accompanying options.

Priorities are used to rank objects for transmission in a queue, using a strict highest-priority scheme. If an object of priority p is in a queue with an object of priority p', and $p' < p$, then the object of priority p will be transmitted first, in its entirety. Implementing priorities is not hard, but can be omitted from an initial kernel implementation.

Deadlines are used to decide if objects are deliverable. A deadline specifies the maximum number of milliseconds between the time when a send call is made and when ATP will stop trying to send the object. If an object's deadline expires, the send operation will terminate, executing a callback if appropriate.

Requests are handles returned by ATP operations to identify a send or receive. The application can use a request handle in subsequent operations: for instance, to abort a send operation, or to retrieve more data from an incoming message.

TICK_INTERVAL	50 ms	interval between "send periods"
TICK_FREQUENCY	20	number of "send periods" per second
MAX_PACKET_SIZE	1460	Ethernet MTU minus IP header
RDP_RETRANSMIT_TIMER	500 ms	time limit on unacknowledged RDP packets
RDP_DELAYED_ACK_TIMER	100 ms	interval to wait before sending delayed ack for packet
KERN_MAX_IFQUEUE_LEN	50	maximum number of datagrams in device driver send queue
MIN_BW_STALENESS	5 s	minimum seconds before bandwidth estimate is "stale"
BW_FILTER_SIZE	5	size of bandwidth averaging filter
BW_MAX_BACKLOG	9	maximum allowable backlock in device driver send queue
BW_MAX_PROBE_SIZE	36500 bytes	maximum speculative increase in bandwidth
BW_MAX_PACKET_PAIR_PERIOD	10	maximum interval between packet pair measurements

Table 1: ATP parameters. *With the exception of `KERN_MAX_IFQUEUE_LEN`, these are all configurable, but the values used in the experiments described in the paper are given. The maximum kernel interface queue length is for FreeBSD. `BW_MAX_PROBE_SIZE` is chosen to be 25 full-sized datagrams, i.e. half the maximum kernel queue length.*

Callbacks are user-supplied procedures which are invoked by ATP to alert the application to the expiry of an object's deadline. There are at least two ways of implementing callbacks: they could be via a dedicated, blocking thread, or with a signal, which doesn't work well with multithreading. Using a dedicated thread is probably easier. Callbacks refer to specific request handles.

Options for objects encapsulate priorities and deadlines, as well as specifying a callback to execute when an object's deadline expires.

Notifications are a special type of callback, which tell the application that bandwidth has strayed outside the bounds it set with a `bandwidth` call.

Queues are an additional layer between endpoints and the user; they allow object scheduling queues to be independent. Implementing queues is not critical (they are not fully implemented in the ATP prototype).

3 ATP in the kernel

At the moment, ATP is implemented at user-level over UDP, on FreeBSD, since this simplified the development considerably. Ideally, it should run in the kernel on Linux or FreeBSD. The important parts for an initial kernel implementation are bandwidth estimation, deadlines for messages, and message scheduling. There is a good fit between the ATP interface and the socket interface, with the exception of callbacks; this can be finessed by having a blocking thread receive callback operations. Details such as priorities, object queues and packet-pair estimation can be left out of the first kernel implementation. A further simplication is to assume that the ATP kernel implementation has at most one active endpoint at a time.

4 ATP pseudocode and detailed description

We discuss three important parts of ATP: the application interface, the reliable datagram protocol, and the ATP send code. A further component, the bandwidth estimator, is discussed in detail in the accompanying paper [1]. Consule the paper for details of the interface to ATP (or look at the prototype code, which is a little more mature). Table 1 gives sample parameters used in the ATP prototype.

4.1 ATP application interface

The ATP interface can be built on top of the BSD sockets interface: ATP has the same notions of hosts and ports as the IP protocols, and ATP endpoints are essentially the same as sockets, from the application's point of view. Send and receive calls can be implemented on top of the `sendmsg` and `recvmsg` calls, both of which take a header as an argument. This header can include a pointer to a protocol-specific data structure.

Figure 1 gives sample data structures for ATP on top of sockets. For the send call, this is where the priority and deadline, and request ID should be stored. Receiving messages is complicated by the fact that

```
enum {
  ATP_SYNC_SEND,
  ATP_ASYNC_SEND
};

struct atp_send_args_t {
  int type;
  int priority;
  int deadline;
  atp_req_t request;
};
```

```
enum {
  ATP_NEW_RECV,
  ATP_CONTINUE_RECV
};

struct atp_recv_args_t {
  int type;
  /* filled in if incomplete */
  atp_req_t request;
  int length;
};
```

Figure 1: Low-level data structures for the ATP interface. *The data structures are passed to the* `sendmsg` *and* `recvmsg` *calls to incorporate ATP-specific arguments.*

```
message_t* enqueue(endpoint_t* e, iov_t* iov, short src, short dest, opts_t* opts);

int send_data_packet(endpoint_t* e, message_t* msg);

int send_control_packet(endpoint_t* e);

int deliver_packet(endpoint_t* e, packet_t* pkt);
```

Figure 2: Reliable datagram interface. *ATP interacts with the reliable datagram protocol through messages, which are lists of one or more packets. Enqueueing a message creates the packets from the I/O vector provided; originating and destination ports for the message are supplied. When transmission of an incoming or outgoing message completes, the datagram protocol makes a call to ATP to notify it.* `Send_control_packet` *is called every* `TICK_INTERVAL` *milliseconds to allow for delayed acks and retransmissions.*

the buffer passed in with the receive call may be too small to fit the whole message. To deal with this, the data structure accompanying the call to `recvmsg` should incorporate three fields: a type field, a request ID field, and a length field. The type field indicates if the receive is for an existing message or, if it should continue retrieving a previous message. If a receive has a buffer which is too small to fit the whole message, the length field is set to indicate how much data still remains to be retrieved, and the request ID is set to allow the receiver identify the message on a subsequent receive call.

4.2 Reliable datagram protocol

ATP incorporates a reliable datagram protocol (we'll call it RDP for convenience). In the user-level prototype, this runs on top of UDP, but within the kernel, it could be a layer on top of IP. Figure 2 gives the most important RDP calls: these are used by ATP to send and receive messages. When a request to send a message completes, or the last packet of an incoming message arrives, RDP makes an upcall to ATP.

Each ATP endpoint (data structure for communicating with another host) runs a separate instance of the RDP protocol. The prototype RDP is a sliding window protocol with a fixed-size window. It uses a selective acknowledgement scheme for retransmissions [2], since this allows the receiver to give more information about what packets have been successfully received if a packet loss occurs. RDP incorporates retransmissions, delayed acknowledgements and duplicate suppression.

4.3 ATP send code

The main state kept by ATP is a queue of incoming and outgoing messages, ordered by deadline. If machine A is communicating with machine B, then the messages sent from A to be should be listed as being received in B's queue, and vice versa. This is necessary to ensure that A does not transmit when the first item in its queue is a message from B [1].

[1]This is likely to result in degraded performance: the message from B could be a short one, and bandwidth could be wasted while its final packet is "on the wire", during which A might be able to transmit its own message. Since A will know the length of B's

```
/* may also be decremented when packets are received */
int all_pkts = 0;

send_loop()
{
  int intervals = 0;
  int pkts_sent = 0;

  for (;;) {
    int pkts = 0;

    if (intervals == 0) {
      all_pkts = bandwidth_estimate() / MAX_PACKET_SIZE;
      intervals = TICK_FREQUENCY;
    }
    else
      all_pkts -= pkts_sent;

    pkts_sent = 0;
    if (all_pkts > 0) {
      int i;

      pkts = all_pkts / intervals;

      for (i=0; i<pkts; i++) {
        message_t* msg = scheduler_get_message();
        if (msg != NULL)
          send_data_packet(msg);
        else
          break;
      }
      pkts_sent = i;
    }

    intervals--;
    wait_for_next_interval();
  }
}
```

Figure 3: The ATP send loop. *This is a simplification of the routine in the ATP prototype. Some details are omitted, including handling endpoints and incoming requests.*

Pseudocode for the ATP send loop is given in Figure 3. Every second, ATP checks the available bandwidth reported by the bandwidth estimator and uses this to decide how many packets it can send or receive during the coming second. It then divides this number by the `TICK_INTERVAL` parameter to decide how many packets to send in each send interval (packets received during the interval also count against this number). The rationale is that sending too many packets at once (for instance, sending all packets at the start of the second) is likely to overflow the network device send queue.

References

[1] B. Atkin and K. P. Birman. Mobility-aware adaptation for transport protocols. Technical report, Department of Computer Science, Cornell University, Feb. 2002. Submitted for publication.

[2] R. Bruyeron, B. Hemon, and L. Zhang. Experimentations with TCP selective acknowledgement. *ACM Computer Communication Review*, 28(2), Apr. 1998.

message, we could conceivably do better.