

UNIX and Pilots Part IV: The PADP Layer

Kevin L. Flynn
Kodachi Systems Group
flynn@kodachi.com

In the last issue of *Handheld Systems*, I described the Minimal HotSync, a simple test program that does a synchronization that doesn't actually transfer any data between the Pilot and the desktop. The Minimal HotSync was built from hand-constructed packets layered directly on top of SLP, which is useful for testing, but completely useless in the real world. This time, I return to the protocol stack and add a PADP layer complete with its own implementation of the Minimal HotSync.

PADP Revisited

It's been a while since I looked at PADP, so a brief introduction is in order before I take a closer look at how it acts on the wire. PADP is the Packet Assembly/Dissassembly Protocol, which rides immediately above SLP in the protocol stack. SLP is a best-effort datagram protocol that provides checksums and a port space. PADP adds reliability to the mix by specifying a positive acknowledgment mechanism with retransmissions and timeouts.

PADP also breaks large packets into smaller fragments, each of which must be acknowledged independently. This has the effect of improving reliability (and performance) by reducing the amount of data that must be retransmitted in the event of an error. Instead of retransmitting the entire packet, you only retransmit the fragment in error.

The fragmentation idea is central to PADP. It's a major part of its functionality and important enough to be reflected in the name of the protocol. Reflecting this underlying design concept, this series of articles uses the term packet to refer to the set of data originally handed to PADP by a higher-level protocol, and the term fragment to refer to the individual transmission units sent over the wire. Fragments are limited to 1024 bytes of user data. A PADP packet may fit into a single fragment or it may require multiple fragments.

PADP Structure

I described the basic structure of PADP fragments last issue, so this should look familiar until I get into the guts of the protocol. PADP fragments consist of a PADP header and some body data, both riding in the body of an SLP packet:

```
[ PADP header ][ Fragment body (user data) ]
```

At this level, PADP is really simpler than SLP. After all, it doesn't need to duplicate information or functionality already provided by SLP, so there's no PADP CRC, because SLP's CRC is enough to detect errors. Likewise, there are no port numbers or transaction IDs here.

There are only a few things in the PADP header:

```
typedef struct PadHeaderType {
    Byte type;           // Fragment type
    Byte flags;          // Flags
    Word sizeOrOffset;   // Size of packet, or offset of
                        // fragment
} PadHeaderType;
```

The `type` field defines the kind PADP fragment type:

```
#define padData 1    // User data
#define padAck 2     // ACKnowledgement
#define padTickle 4  // Tickle
```

(Type 3 used to be a negative acknowledgment, but it's no longer used.) `padData` fragments carry real user data. `padAck` fragments acknowledge successful receipt of `padData` fragments. `padTickle` fragments prevent timeouts during slow operations. The `flags` and

`sizeOrOffset` fields behave somewhat differently, depending on the fragment type.

In a `padData` fragment, the `flags` field is a bitmask with the following values:

```
#define padHdrFlagFirst 0x80 // first fragment
#define padHdrFlagLast 0x40 // last fragment
```

`padHdrFlagFirst` must be set on the first fragment of a given packet, to identify that the fragment starts a packet. `padHdrFlagLast` must be set on the last fragment of a given packet. If the packet fits into a single fragment, that fragment must have both `padHdrFlagFirst` and `padHdrFlagLast` set, since it's both the first fragment and the last fragment of the packet.

The `sizeOrOffset` field is overloaded to serve two functions:

- If `padHdrFlagFirst` is set, `sizeOrOffset` contains the size of the packet.
- If `padHdrFlagFirst` is not set, `sizeOrOffset` contains the offset of this fragment within the entire packet.

The first fragment contains the size of the entire packet, while subsequent fragments identify their location within the packet.

In a `padAck` fragment, `flags` and `sizeOrOffset` are normally set identical to those in the `padData` fragment being acknowledged. There is one exception – a new flag bit:

```
#define padHdrFlagErrMemory 0x20
// receiver out of memory
```

`padHdrFlagErrMemory` is set in the `padAck` to the first `padData` fragment of a packet if the receiver doesn't have enough memory to receive the whole packet. If the sender gets such a `padAck`, it cannot send the packet.

Note that although the `sizeOrOffset` field of a `padAck` fragment might indicate that some data should accompany the header, data is never sent with a `padAck`. The SLP packet size is sufficient to detect this unambiguously.

`padTickle` fragments are always marked as both the first and last segment, with a `sizeOrOffset` of zero. They never have any accompanying data and they are not acknowledged. Their only purpose is to prevent a timeout from occurring.

Timeouts and Other Issues

PADP is based on the idea that a correct fragment is acknowledged with a `padAck` fragment. What happens if a fragment is incorrect or corrupted in transit? There used to be a negative acknowledgment fragment type, but that was removed. Now, the system simply has to realize that the fragment wasn't acknowledged in time and retransmit the fragment in question.

Normally, this is straightforward. However, there are some situations that can confuse matters. The most problematic issue is that the Pilot can take a long time to carry out certain tasks for quite legitimate reasons. For example, the Pilot has to compact its heap to reduce fragmentation on occasion. This is important, but it can easily take longer than the inter-fragment timeout allows.

This is what `padTickle` fragments are for. When the Pilot is in the middle of a long-running operation, it periodically sends `padTickle` fragments to the desktop. The `padTickle` packets carry no data and are not acknowledged. However, they cause the desktop to reset its inter-fragment timer so its PADP receiver never times out.

Another issue is the possibility that a `padAck` fragment can get dropped due to transmission errors. In this situation, the fragment being acknowledged arrives successfully but the sender doesn't know it. This can cause problems at times.

From the desktop side, there are four distinct scenarios here:

1. You send a fragment of a `padData` packet. It's not the last fragment, and the `padAck` from the Pilot is lost. This isn't a

problem. Timeout, resend the fragment, and the Pilot sends a new padAck. If the line is so bad that this always fails, reach your retry limit and give up.

2. You send the last fragment of a padData packet, and the padAck from the Pilot is lost. Realizing that any padData packet sent to the Pilot is going to be a request of some kind, for which you expect some kind of response (remember, you're the client), there are two subcases:
 - 2a. The Pilot sends the first fragment of a response before the waiting-for-acknowledgment-timer expires. If this happens, you see a padData fragment carried in an SLP packet with a transaction ID that matches your request. You can treat that as an implicit acknowledgment of the last fragment of your request, so that's OK.
 - 2b. The Pilot doesn't send the first response fragment in time, and you timeout. This is not a problem. Resend the last fragment and the Pilot just ignores it. Eventually the Pilot sends a fragment back, and you land in case 2a, above.
3. You receive a padData packet from the Pilot, and your padAck to a fragment that isn't the last fragment is lost. This isn't a problem. The Pilot resends a fragment that you already have; just send a new padAck.
4. You receive the last fragment of a padData packet and your padAck is lost. This is also not a problem. The Pilot might send a duplicate last fragment, which you see as a fragment belonging to a transaction ID, and ignore it. When you send your next request, the Pilot treats that as implicitly acknowledging the previous response (like our case 2a, earlier).

Pseudocode

Putting it all together, we arrive at pseudocode that looks like this:

```
PAD_receive {
    try to receive a first fragment with the XID
    we're looking for;

    if ( timed out ) {
        abort, lost connection;
    }

    remember size;

    if ( we can't allocate space to hold the whole fragment
) {
        ack "memory error";
        return, no memory;
    }

    mark this fragment as having offset 0
    initialize expected offset to 0

    /* Fall through into stash/ack/next fragment loop */

    while ( 1 ) {
        if ( fragment has correct offset ) {
            stash fragment;
            bump expected offset;
        }

        ack;

        if ( padHdrFlagLast ) {
            break;
        }

        try to get another fragment with the XID
        we're looking for;
```

```
    if ( timed out ) {
        abort, lost connection;
    }
}

PAD_send {
    fragment_offset = 0;

    while ( fragment_offset < packet_size ) {
        determine fragment_size;
        construct fragment header & data;

        try to send the fragment;

        try to receive an acknowledgement;

        if ( it's a response fragment with our XID ) {
            push it back on the input stream and
            treat it as an ack;
        }

        if ( memory error ) {
            abort;
        }

        bump fragment_offset;
    }
}
```

UNIX Implementation

After all the hassles of building SLP, the implementation of PADP is almost simple. It's just a few more functions on top of the SLP layer with appropriate hooks into the psock_Send and psock_Recv functions. You also implement MinSyncPad, a version of MinSync that uses the new PADP layer to perform the Minimal HotSync. As usual, the source code can be found at <http://www.kodachi.com> and on the source code disk for this issue of *Handheld Systems*.

Next time, I move up the protocol stack into DLP.

Side Notes

Doubtless, you've heard of the new PalmPilots (*Handheld Systems* had an article about them in the last issue). I bought a PalmPilot Professional recently and have been using both it and my original Pilot 5000 to test the code I write here. The only difference between the two that the Minimal HotSync exposes is in the CMP Wakeup packet: the Pilot 5000 reports protocol version 1.0.0.0 (0x01000000), the PalmPilot reports version 1.1.0.0 (0x01010000). Minimal HotSync has been modified to handle this. I'll keep you posted on other differences as I find them.

On another note, in my previous article in *Handheld Systems* 5.3, I stated that SLP packets are limited to 1024 bytes of user data. As far as I can tell, this is an error. SLP packets have no such limit (though PADP fragments do.) Sorry about that. ✓