KSP Update

Mick Seaman

This note includes: an annotated description of the proposed KSPDU format, describing the life and purpose of all PDU fields; object diagrams of the classes that represent the entities, state machines, and data maintained by a KSP entity; the more significant KSP state machines; and the most important procedures. All these reflect revision of KSP to use a contributory key agreement mechanism to determine each SAK, although the transport and basic communication mechanisms remain unchanged from the first proposal.

With the possible exception of the annotated KSPDU format, it is unlikely that the information contained in this note will prove illuminating or satisfactory to anyone who wasn't at the last meeting. It is being distributed in advance of the upcoming meeting because the essential diagrams and code contain too much information to be satisfactorily presented on an LCD projector, or perused on a laptop while in a meeting. Bring your own printed copies.

Further notes

KSPDU format and fields

KSP uses a single packet type and format (illustrated below). KSPDUs are transmit periodically and as needed subject to a leaky bucket rate limiter. The transmitter of the PDU is referred to as the ‘actor’, and other protocol participants as its ‘peers’.

Integrity protected frame in clear allows debug/ attack investigation by field operations personnel without need to disclose/ provide disclosure of CAK (integrity protecting master key).

Multicast address, confined by bridges to a single LAN.

Destination address integrity protected. Makes it hard to launch an attack from a distance as address will not pass through bridges, but cannot be changed on captured frames.

Use (or not) of Latest and Old Key fields below, if used the MACSec association number (AN) bound by the actor to each key, and whether receiving/transmitting using the key.

Identifies the CAK (secure Connectivity Association Key), i.e. the master key used to GMAC protect this KSPDU. MAC address based (EUI-48) so can be allocated by system managing master keys. Persists across power cycles/reboots/system resets, while all other recorded info apart from MAC Address/ SCI assumed lost.

Random IV, independently generated for each KSPDU transmitted, used by GMAC together with the CAK .

MAC address (EUI-48) based Secure Channel Identifier used when transmitting MACsec data frames. Receivers bind SCI,AN to selected SAKs (Secure Association Keys) for MACsec.

Random nonce, generated at reboot/system initialization. Also reselected if collision detected (station with other SCI using same nonce), or Message Number space exhausted.

Nonce, incrementing from 1 when new MI generated. Actor records values at intervals to support timeliness verification (see below). Good for 13+ years before new MI reqd.

Random nonce, generated at reboot. Reselected whenever MACsec data PN (packet number/nonce) space for selected data key (SAK) near exhaustion. Input to pseudo-random function using CAK to generate SAK.

XOR of all KCs currently input to SA. Probably uniquely identifies selected SAK but provides no info to attacker .Protocol converges even if collisions, may be data packet loss. SAK selected and receiving initiated when at least one LIve Peer, and no Potential Peers , or all Live Peers agree LKI. Transmit initiated when all Live Peers report receiving.

Old SAK used to transmit while latest being selected, retained after transmitting on new SAK to collect frames of differing priority and allow others to move to new SAK. Explicitly identified to ensure no problems if participant loses messages when LKI becomes OKI, and new LKI calculated soon after, and to clarify result of group merge while two LKIs in selection.

Bounds data transit delay, particularly where priorities/drop precedence mean no PN based data replay protection.

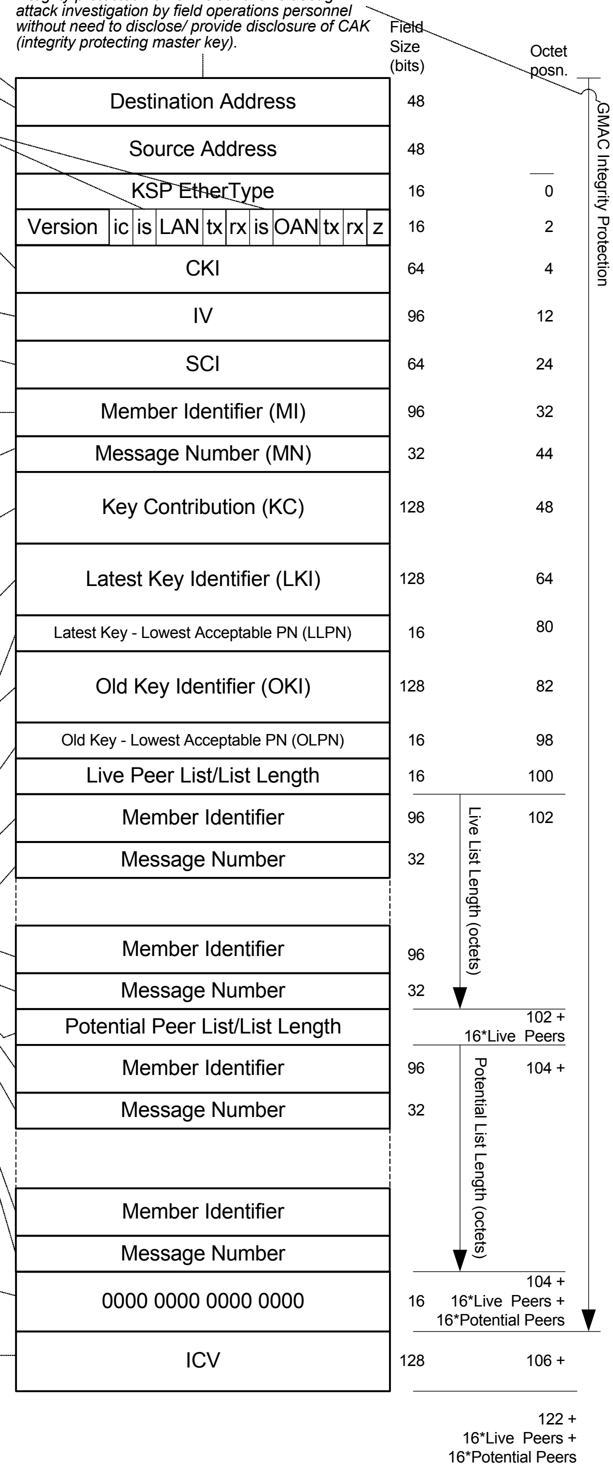
Reflecting received identifiers proves liveness to others.

Reflecting last received message number proves timeliness to others, defeats ‘delay frames’ attack. If no timely messages (max delay 2 - 10 secs) from participant, will be removed from Live Peer List and SAK calculation and reception stopped.

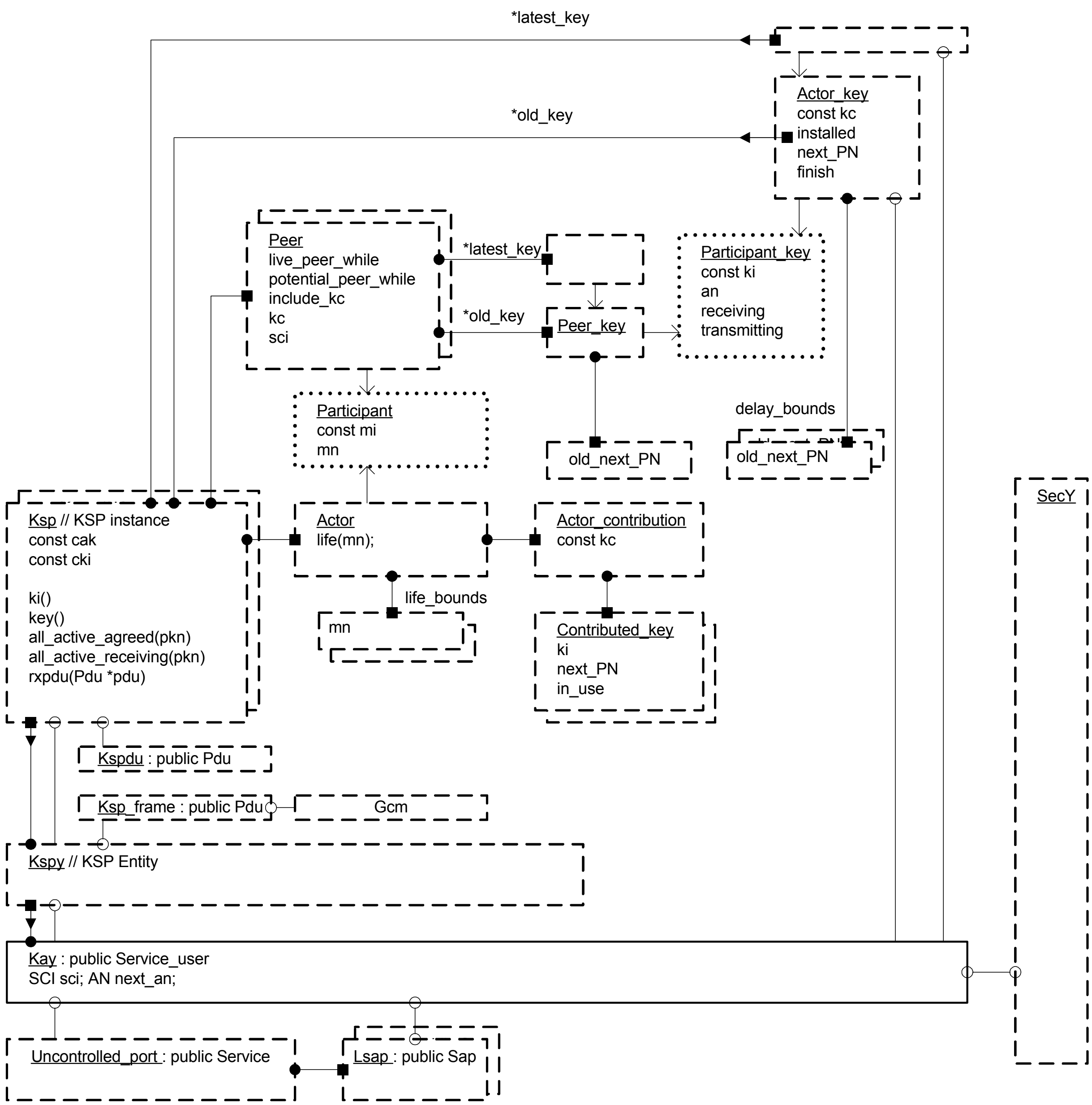
Separately identifying “Live Peers” i.e. participants that have proved liveness and timeliness to actor, from “Potential Peers” to which actor will respond to prove own liveness, allows participants quicker retransmit when apparent lost messages have defeated their proving liveness. Also allows Potential Peer List to be seeded from others Live Peer List (speeds convergence) without keeping old participants/Member Identifiers in circulation for ever.

Terminates PDU while allowing TLV extension for future revision.

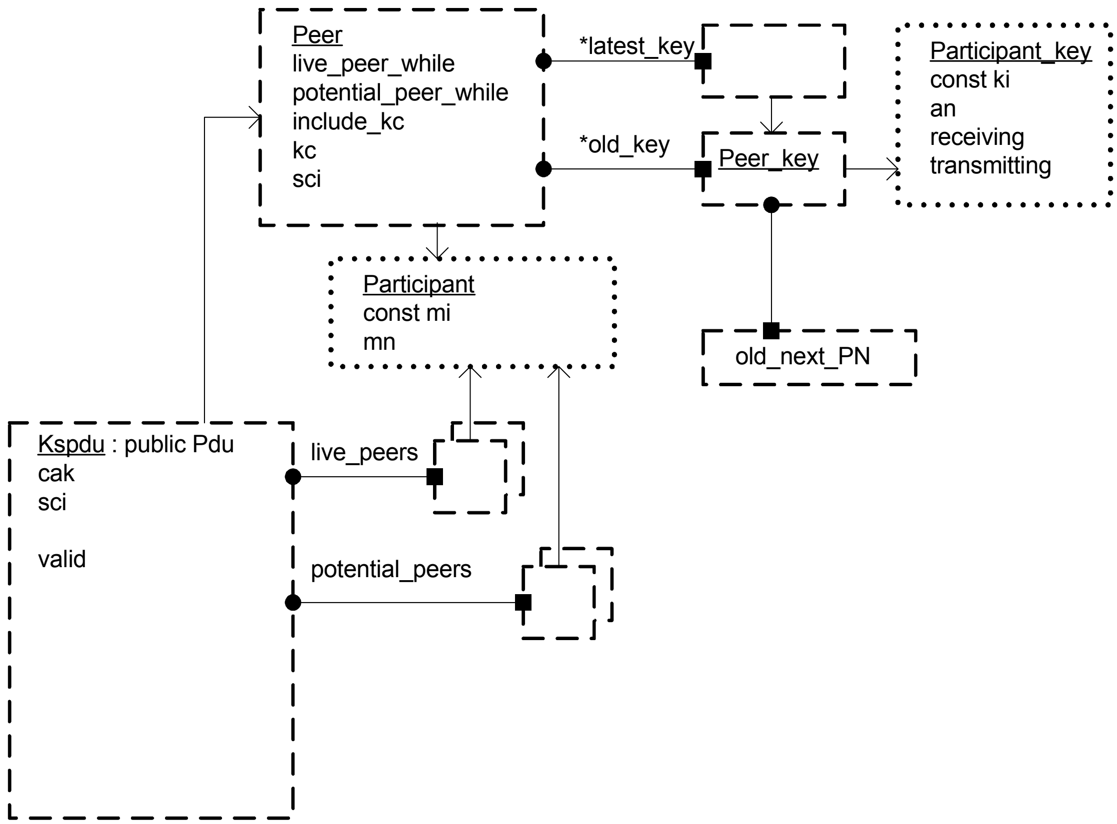
Integrity Check Value calculated using CAK (master key) and IV allows each participant to prove possession of the master key, and prevents message modification by attackers.



v0.2 9/16/2004 8:04 PM PST

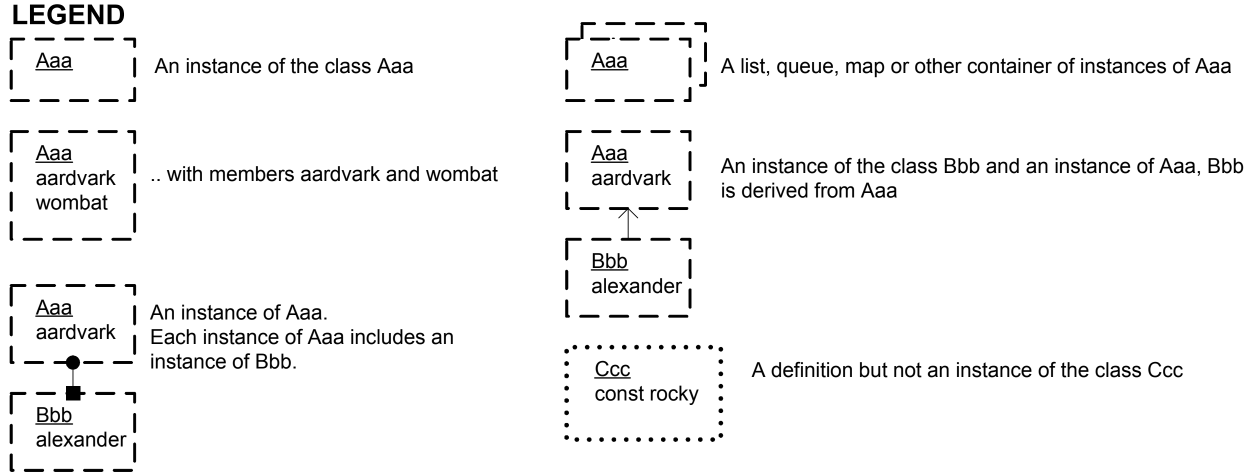


KSP Objects <<KSPO 0.1>>

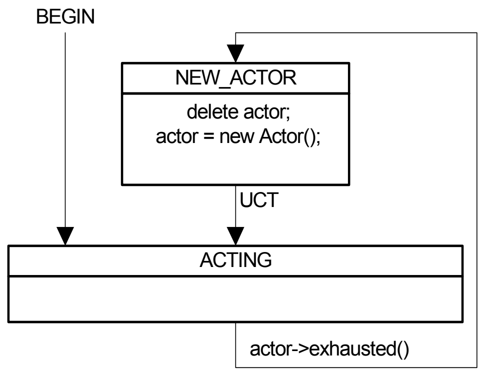


KSPDU Objects <<KSPO 0.1>>

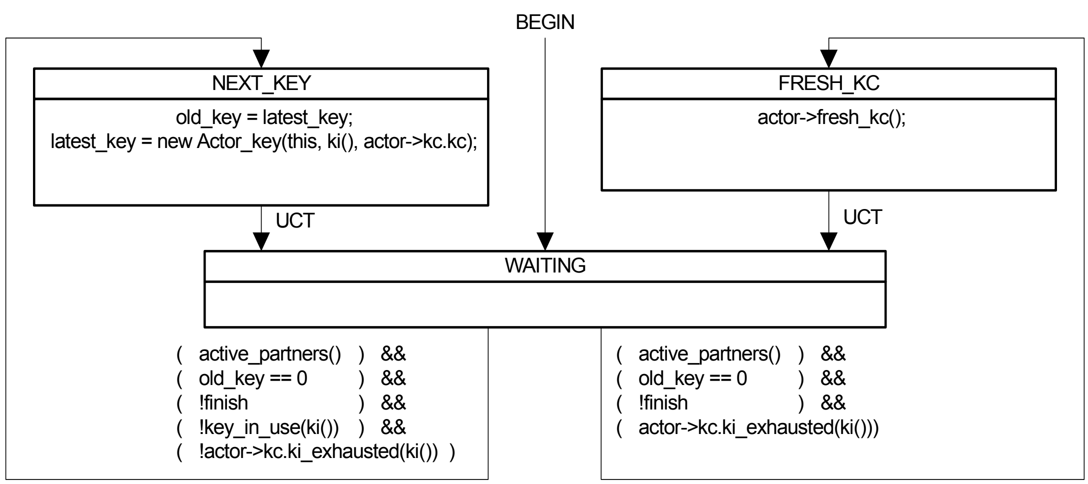
LEGEND



```
Ksp::Ksp(Kspy *p, const CAK ca_key, const CKI ca_key_id) : kspy(p), cak(ca_key), cki(ca_key_id)
{   actor = new Actor();
};
```



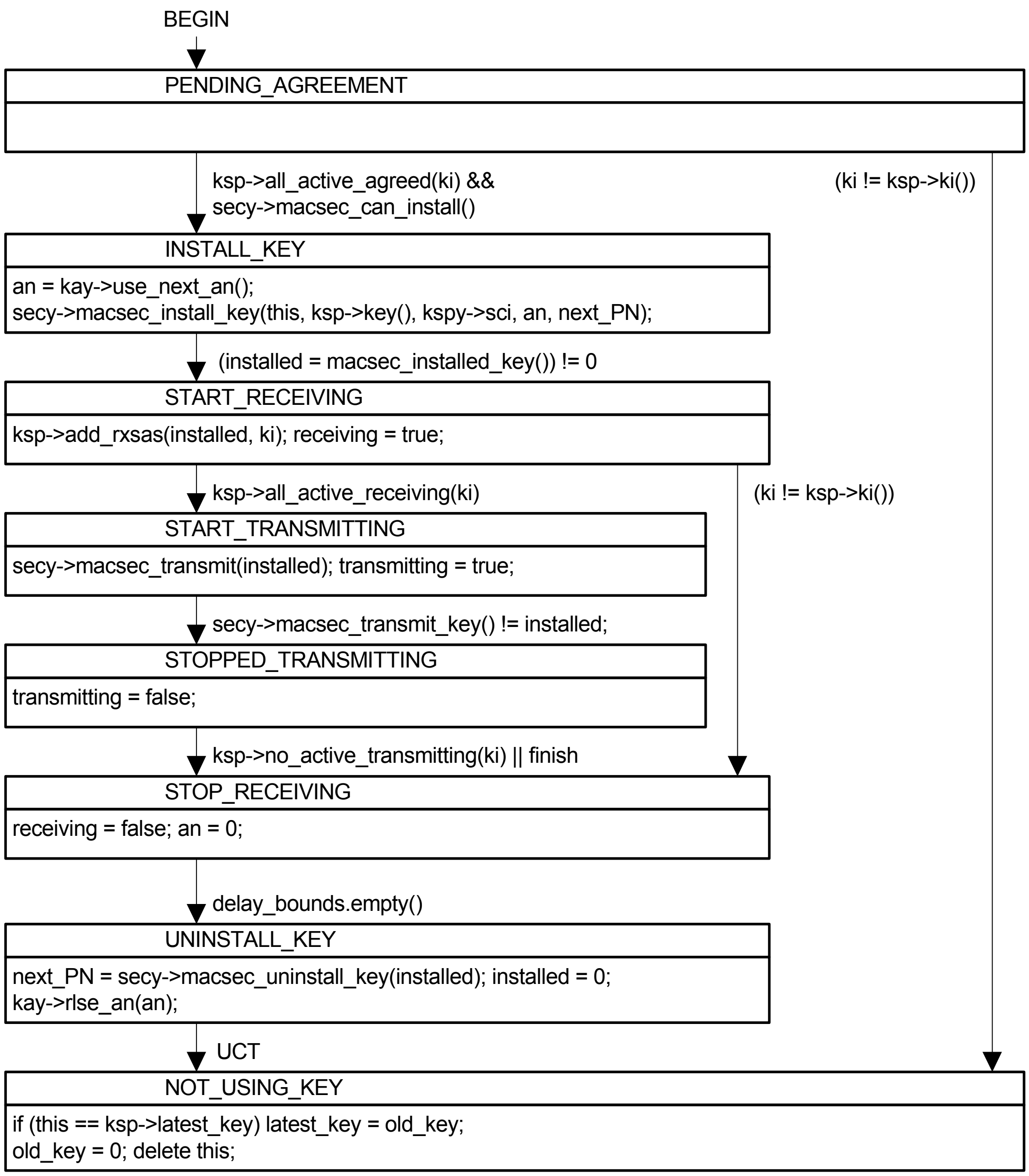
Ksp actor state machine <<AM 0.1>>



Ksp keys state machine <<KKM 0.1>>

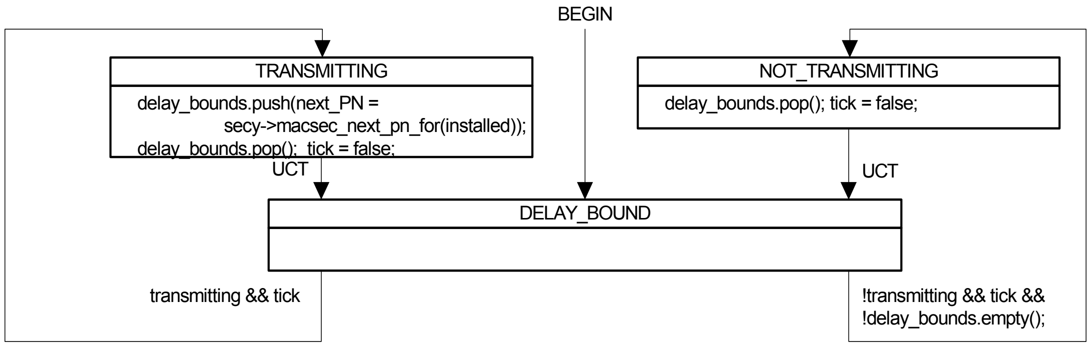
```
Actor_key::Actor_key(Ksp *p, KI key_id, KC key_contribution) : Participant(key_id), ksp(p), kc(key_contribution),
{
    receiving = transmitting = finish = false; installed = 0; an = 0;
   next_PN   = ksp->next_pn_for(key_contribution, key_id);
                                                                                                                    \
   for (int i = 0; i < ticks_to_record; i++) delay_bounds.push(next_PN);

   akm = PENDING_AGREEMENT;
   dbm = DELAY_BOUND;
};
```




```
KaY *   const kay    = ksp->kspy->kay;
SecY*   const secy   = kay->secy;
```

Actor key state machine <<AKM 0.2>>



Delay bound state machine <<DBM 0.1>>

```
void Ksp::rxpdu(Pdu *received_pdu)
{
   Kspdu  rcvd(received_pdu);

   if (!rcvd.valid )      {               rcv_event(Invalid_pdu)    return; };
   if ( rcvd.sci == sci ) {               rcv_event(Loopback_pdu)   return; };
   if ( rcvd.mi == actor.mi)
   {
      this.change_mi();                    rcv_event(Duplicate_mi)   return; };
   };                                      // broken psrng?

   Peer *peer = find_peer( rcvd->mi);

   if (peer != 0)
   {
      if (rcvd.mn  <  peer->mn) { rcv_event(Misordered_pdu) return; };
      if (rcvd.mn  == peer->mn) { rcv_event(Duplicate_pdu)  return; };

      if (rcvd.sci != peer->sci){ rcv_event(Peer_sci_changed);
                                   delete peer; peer = 0;            };
   }
   if (peer == 0)
   {
      peers.push_back(Peer( this, rcvd.mi, rcvd.sci));
      peer = &(*peers.last());
   };

   peer->potential_peer_while = potential_peer_life;

   peer->mn  =  rcvd->mn;

   Ticks life = actor->life(rcvd->find_me(actor->mi));
   if (life > peer->live_peer_while) peer->live_peer_while = life;

   if (peer->live_peer_while != 0)
   {
      if ((peer->include_kc != rcvd->include_kc) || (peer->kc != rcvd->kc))
         bool recalculate_key = true;

      peer->include_kc = rcvd->include_kc;
      peer->kc         = rcvd->kc;

      peer->rx_keys(&rcvd);

      add_potential_peers(*(rcvd->peers)); // from live peer's live list

      if (recalculate_key)
      {
         if (old_key != 0)    old_key->execute_akm();
         if (latest_key != 0) latest_key->execute_akm();
         execute_kkm();
      };
}; };
```