# RTMFP Overview
for IETF77 TSV AREA

Matthew Kaufman

Sr. Computer Scientist, Project Lead

Adobe Systems

mkaufman@adobe.com

matthew@matthew.at

Yes, we’re crazy

- Designed and implemented a new transport protocol
  - Twice!

Adobe

Yes, we're crazy

- Designed and implemented a new transport protocol
  - Twice!
- But we successfully deployed this to almost every Internet host
- So…
  - Who are we?
  - Why did we do this?
  - And what is it?

Who are we?

- Matthew Kaufman and Michael Thornburgh
  - Background in building ISPs **and** writing software
  - Founded **amicima** in 2004
  - Designed and wrote MFP (Secure Media Flow Protocol), released as open source
  - Acquired by Adobe in 2006
  - Designed and wrote RTMFP, released in Flash Player 10.0
  - Designed and wrote RTMFP Groups, shipping in Flash Player 10.1

The other way to deploy a new transport protocol

Worldwide Ubiquity of Adobe Flash Player by Version - December 2009

	Flash Player 8 & below	Flash Player 9	Flash Player 10
Mature Markets	99.0%	98.9%	94.7%
US/Canada	99.0%	99.0%	94.2%
Europe	99.1%	98.8%	95.6%
Japan	98.0%	97.7%	93.4%
Australia/New Zealand	98.9%	98.6%	94.3%
Emerging Markets	98.2%	98.0%	92.7%

Overview of Flash Player Communications

- RTMFP is proprietary transport protocol
  - Ships in Flash Player 10.0 and later (client-server and peer-to-peer)
  - Used by RTMFP Groups (peer-to-peer overlay) in Flash Player 10.1 and later
    - Application-level Multicast, Posting, Directed Routing, Object Replication
- RTMP is a (now published) media transport over TCP
  - Variants: RTMPT, RTMPS, RTMPE
- Because RTMFP is presently proprietary I can share:
  - Things we've learned
  - Interesting design points (some covered by patent filings, see IPR disclosure)
  - How it coexists with other network protocols
- …but I cannot share:
  - Bit-level packet formats, details of cryptosystem

Why a new transport protocol?

- To securely deliver media flows over the Internet
- And do it “better” than existing choices
  - For our definition of “better”, of course
- To remember:
  - MFP designed in early 2004, open source release in July 2005
  - RTMFP designed in late 2006, learned from MFP
  - MFP predates DCCP (mostly), DTLS, HIP, ICE

Things we believe about the Internet

- Delivers datagrams (on a “best effort” basis)
- Lots of end-to-end delay (200km/msec, one way in glass)
- Usually lossless (copper 10^{-9}, glass 10^{-12}, wireless emulates)
- Usually in-order
- Possibly congested
  - Signaled by loss (and that loss can be bursty, and is usually near the endpoints)
- Likely path asymmetry
- Isn’t secure
  - Lots of scanning and denial-of-service attacks, untraceable address spoofing, some eavesdropping, some MITM
- Filled with NA(P)T and firewall devices, “end-to-end” gone
  - (but not forgotten)

Things we believe about Computers

- Fast and still getting faster
- Manipulating data is fast, moving data is slow
  - Especially slow to move data over the network (but even off-chip on-system matters)
- O(1) operations are better than O(n) operations
  - Especially when n is large
- Anything we want to do must be possible as a normal user on most popular operating systems
  - No raw sockets
  - No ECN
  - No way to force the source address when bound to INADDR_ANY
- Network availability can and will change as the computer moves
  - Even as simple as “unplugged Ethernet cable, let wireless card take over”

Things which therefore follow…

- The fewer the round trips, the better
- The shorter the packet, the better
  - Especially if you’re encrypting the packet
  - Even if it makes the ASCII drawings optimized for 32-bit aligned structures look bad
- The receiver should pick the data structure index when possible
- We must use UDP (NAT/firewall, Operating System, TCP won’t do)
- We must encrypt everything all the time
- We must not consume state (or even respond, if possible) until we are sure we want to talk to the other end
- We must not think we know our own IP address or UDP port number
- We must not believe that addresses do not change mid-session
- We must respect loss as congestion and respond appropriately

The Problem

- Real-time media delivery for Rich Internet Applications
  - Multiple streams of:
    - Audio
    - Video
    - Control
    - File transfer
  - Over the real Internet
    - Congestion
    - Packet loss
    - Insecure (tapping, active attacks, denial-of-service attacks)
  - With Peer-to-Peer capabilities
    - NAT and firewall traversal

The Solution

- Connectivity philosophy
  - “Just Works”
    - In real life, on today’s actual Internet
    - Dealing with NAT and firewall not an afterthought
  - Congestion control required
    - One congestion domain between a pair of endpoints
  - Avoid excess round-trip times
    - “Call setup” from a **cold start** needs to be fast
  - Avoid repeated work
  - Clean solution vs. “glue and tape” on existing protocols (or a deep set of new ones)
- Note: We believe in combined signaling and media
  - Avoids extra RTTs, repeated NAT traversal effort, key negotiation, etc.
  - Easy to separate if combined… reverse is not necessarily true

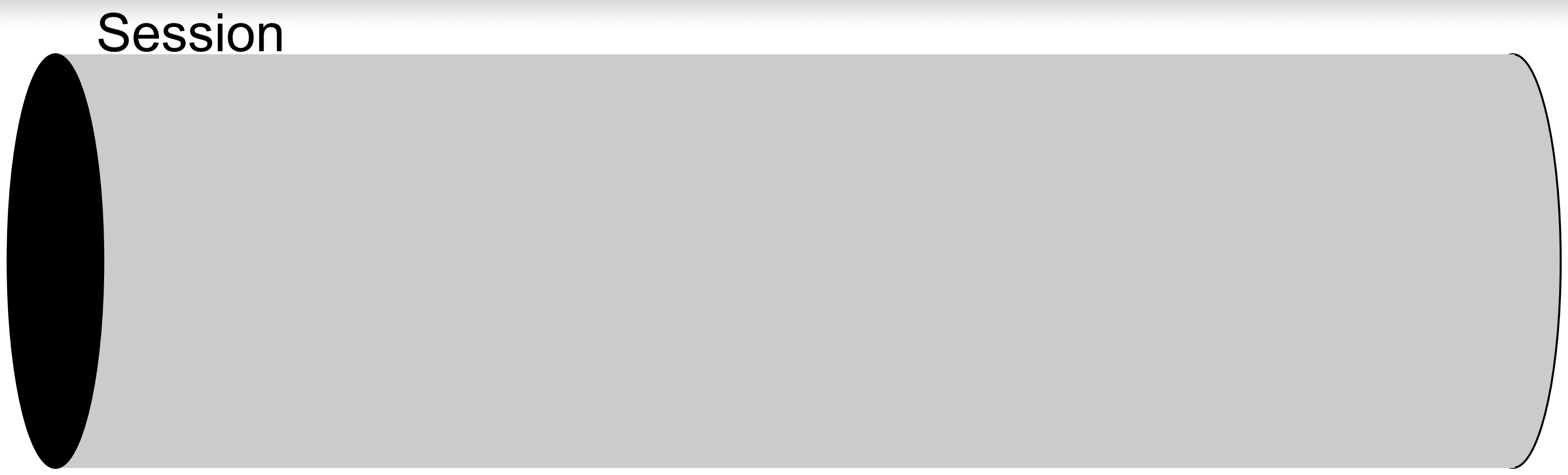
Layering

- Good:
  - Reuse existing solutions (TCP for reliable transfer, SSL for security)
  - Avoiding redesigning and rewriting (and the associated risks of doing so)

Layering

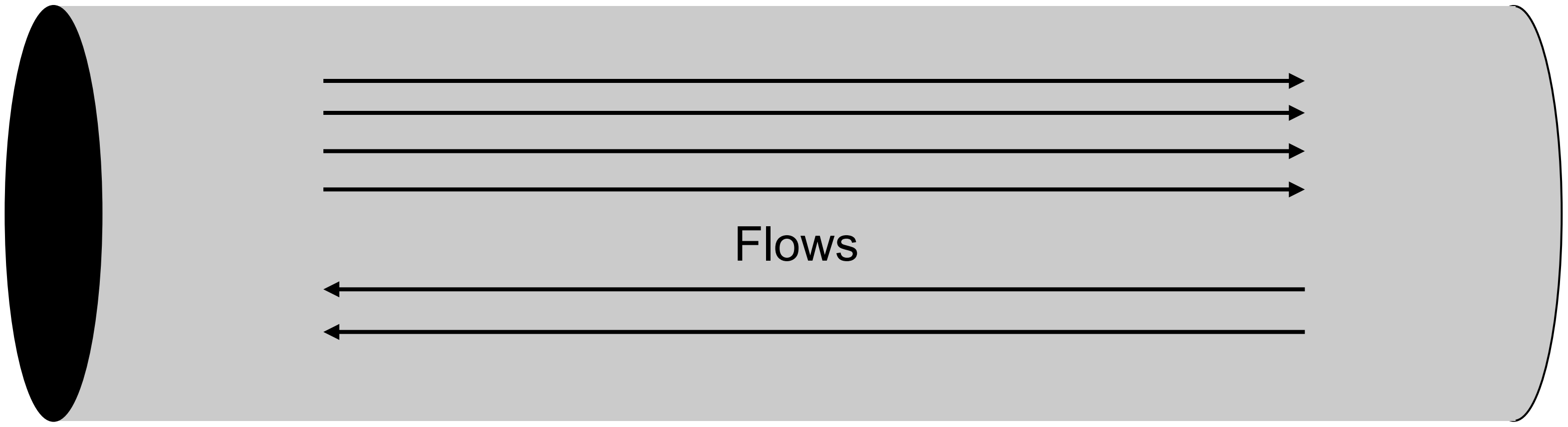
- Good:
  - Reuse existing solutions (TCP for reliable transfer, SSL for security
  - Avoiding redesigning and rewriting (and the associated risks of doing so)
- Bad:
  - Information can be lost (soft bit decisions on RF links not visible to higher layers)
  - Extra overhead (numerous protocols to add framing to TCP byte streams)
  - Extra round trips (set up TCP session, set up SSL session, signaling handshake,…)
  - Lower layer issues cannot always be solved by upper layers (SSL can’t fix SYN flood)

The Solution: Sessions



- Bidirectional, only one between any pair of endpoints
- Parallel Open (load balancing, NAT and firewall traversal)
- Secure (“always on”, set up in 2 RTT with anti-scan and anti-DOS)
- Congestion controlled (dynamic)
- IP address mobility, fast outage recovery

The Solution: Flows



- Many, no set-up time (“½ RTT”, no rekeying)
- Named
- Unidirectional
- Prioritized
- In- or out-of-order delivery
- Variable reliability
- Buffer management

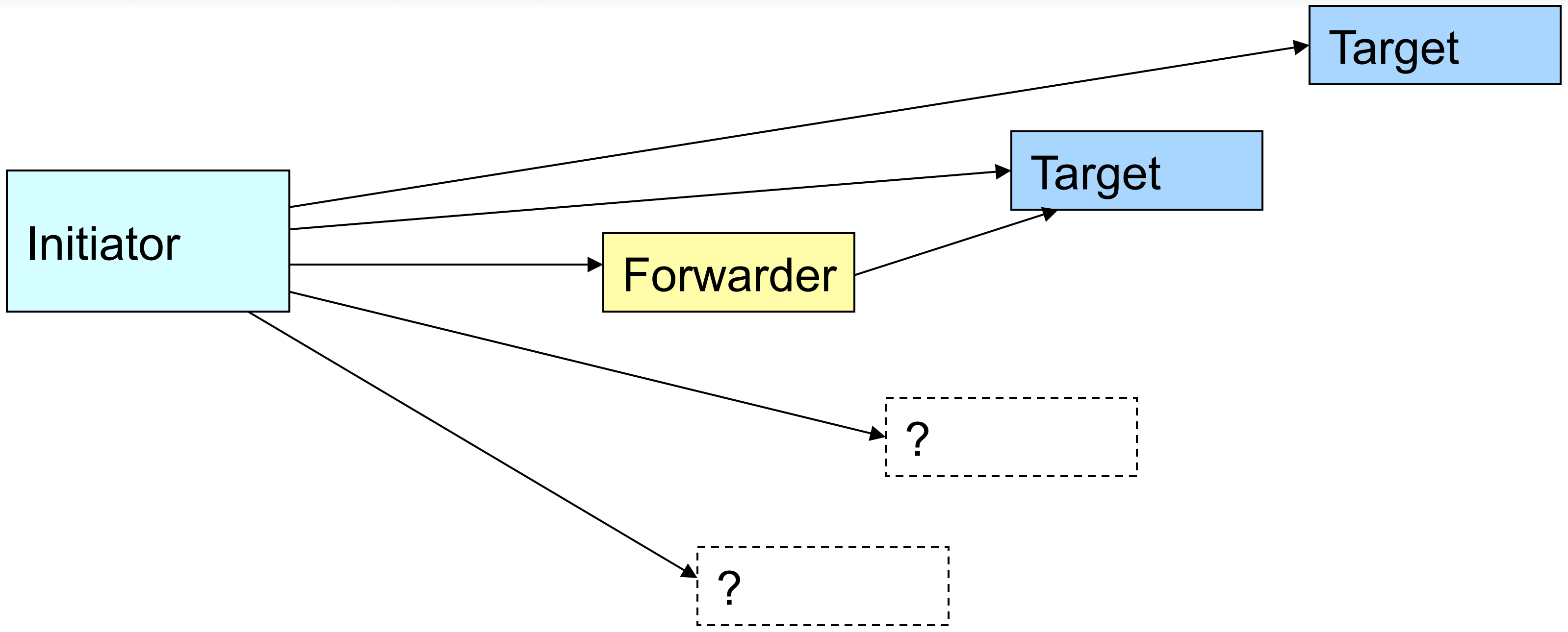
The Solution: How It Works

- Request for a Flow to an endpoint for which there is no open session
  - Session Established
    - Flow(s) Established
    - Flow Close or Exception
    - RTT measured as necessary
    - IP mobility handled as necessary
  - Eventually, Session Closed
    - No more flows for a period of time
    - Or session fails

The Solution: How It Works: Session Establishment

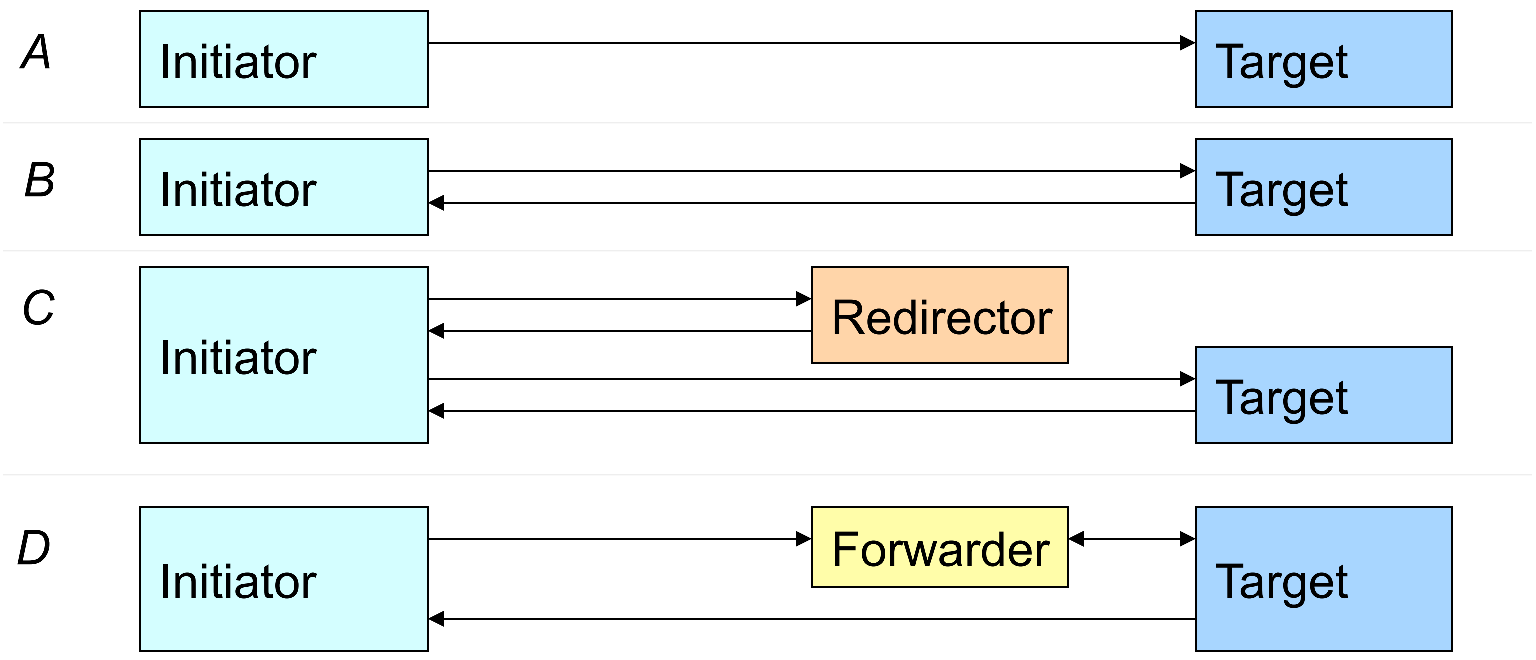
- Session Establishment
  - 4-way handshake (anti-DOS, anti-scanning)
  - Parallel Open
  - Security exchange embedded in handshake
  - Session IDs chosen (by the receiver who will need to demultiplex by these IDs)
  - Session nonce exchanged
    - Usable for later signature operations without another RTT to exchange nonces

Session Establishment: Parallel Open



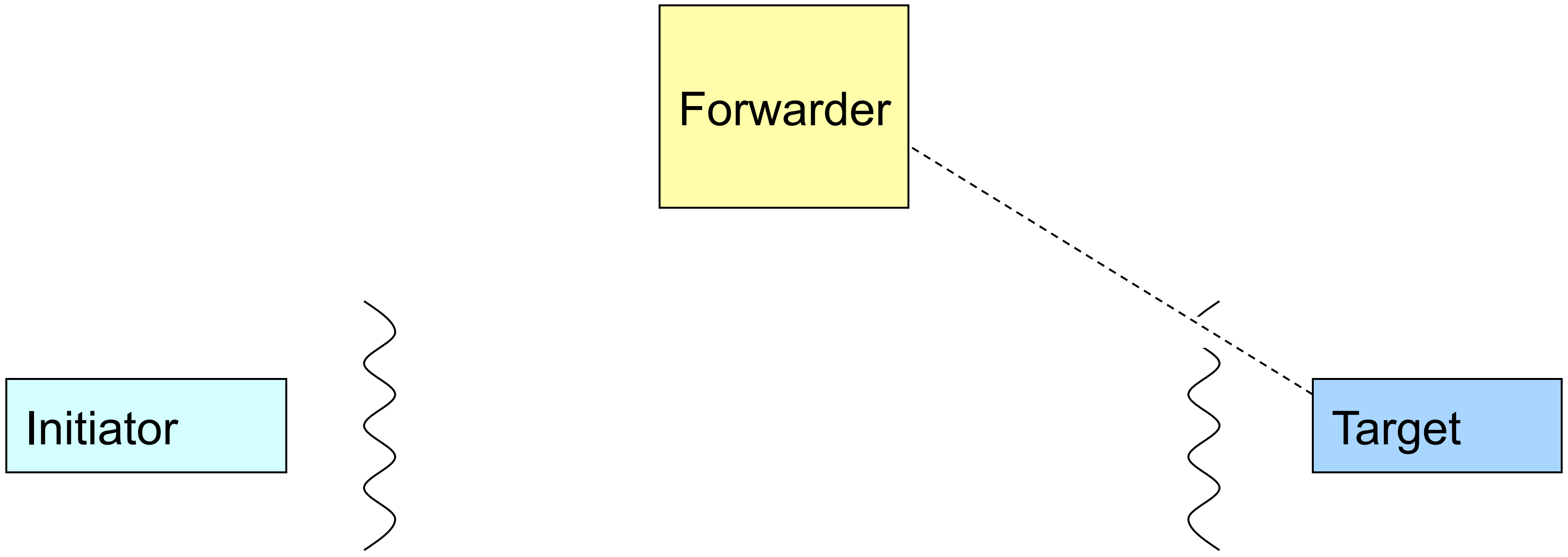
- Initiator Hello message is sent to all candidate addresses for target
  - Simultaneously or offset in time
- First to respond with correct certificate wins
  - Load balancing without hardware
  - Simultaneously open to private (behind NAT) and public address
  - Simultaneously open to a forwarding server (NAT traversal)

Session Establishment: Endpoint Discriminator



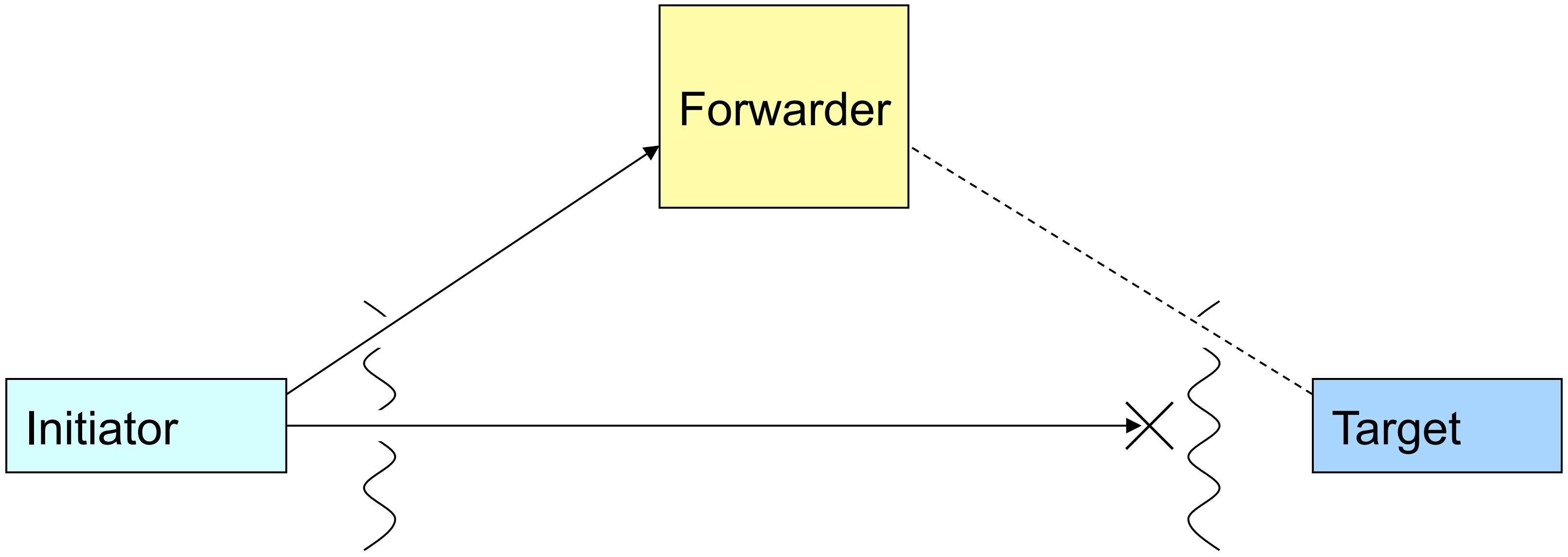
- Initiator Hello message contains Endpoint Discriminator (EPD)
- A node may Ignore, Respond, Redirect, or Forward
  - Ignore: Blocks port-scanning attempts, must know address **and** EPD
  - Respond: Normal case
  - Redirect: Supply an alternative (list of) IP address(es), results in more Initiator Hello messages being sent
  - Forward: Send the Initiator Hello onward to another (connected) node for NAT traversal

Session Establishment: NAT traversal



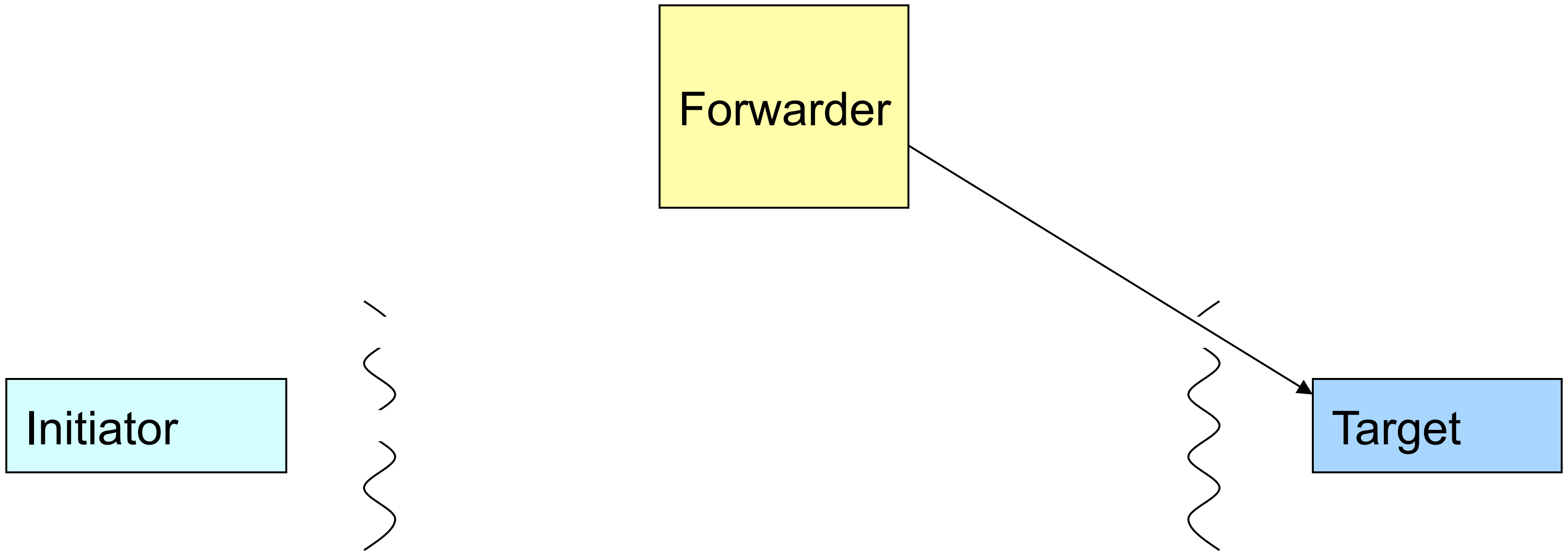
- Initiator and Target are both behind port-restricted (firewalling) NAT
- Target has an open Session to the Forwarder host

Session Establishment: NAT traversal



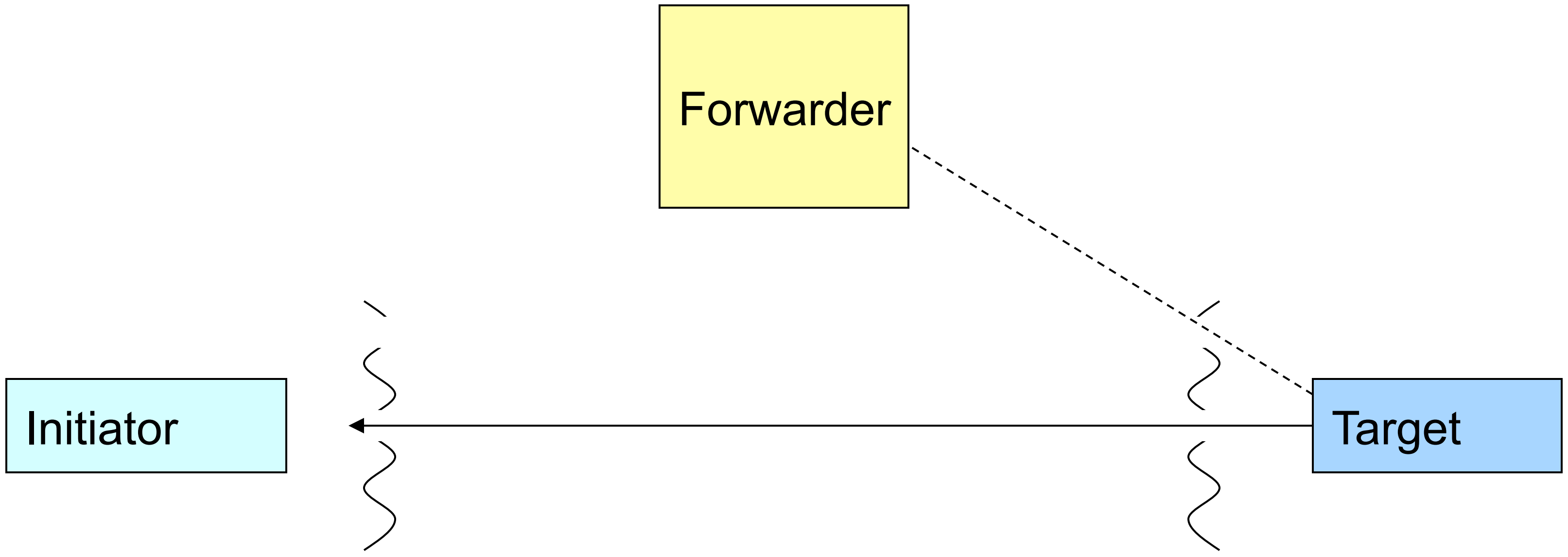
- Initiator tries to parallel-open to Target’s EPD at both Target’s address and Forwarder’s address
  - This opens “hole” at Initiator end
  - Direct message is blocked by NAT/firewall at Target’s end

Session Establishment: NAT traversal



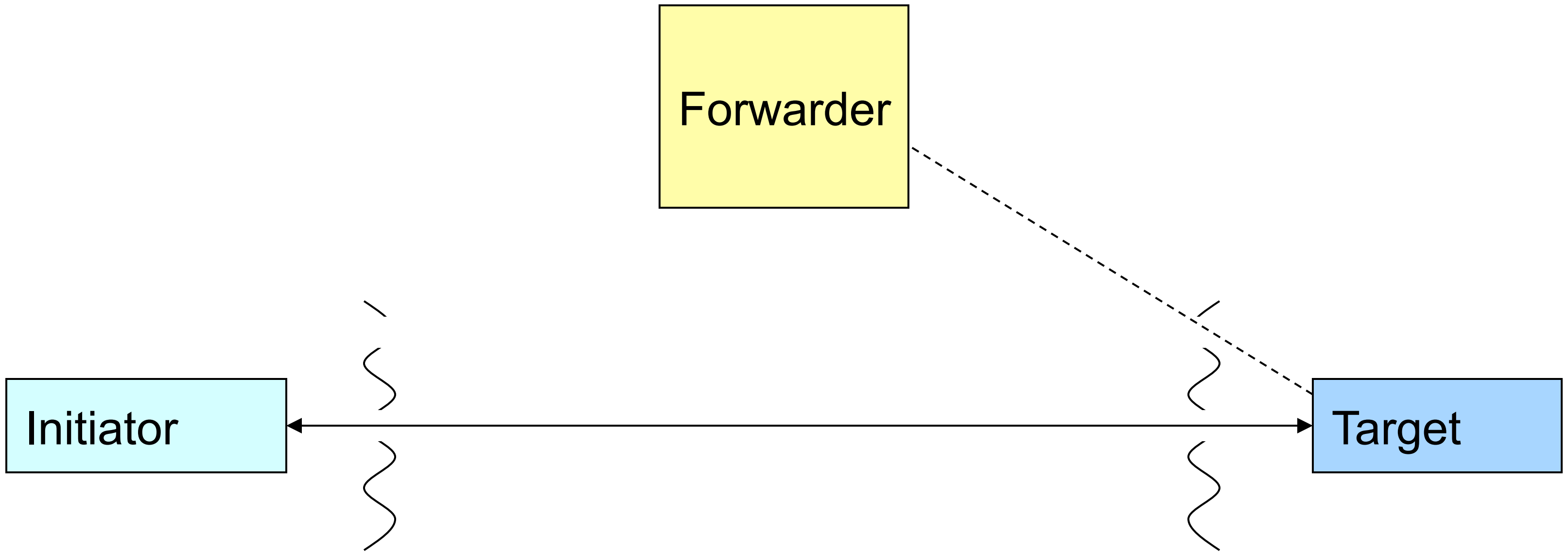
- Forwarder sends Initiator’s IHello (and derived address) to Target over existing session

Session Establishment: NAT traversal



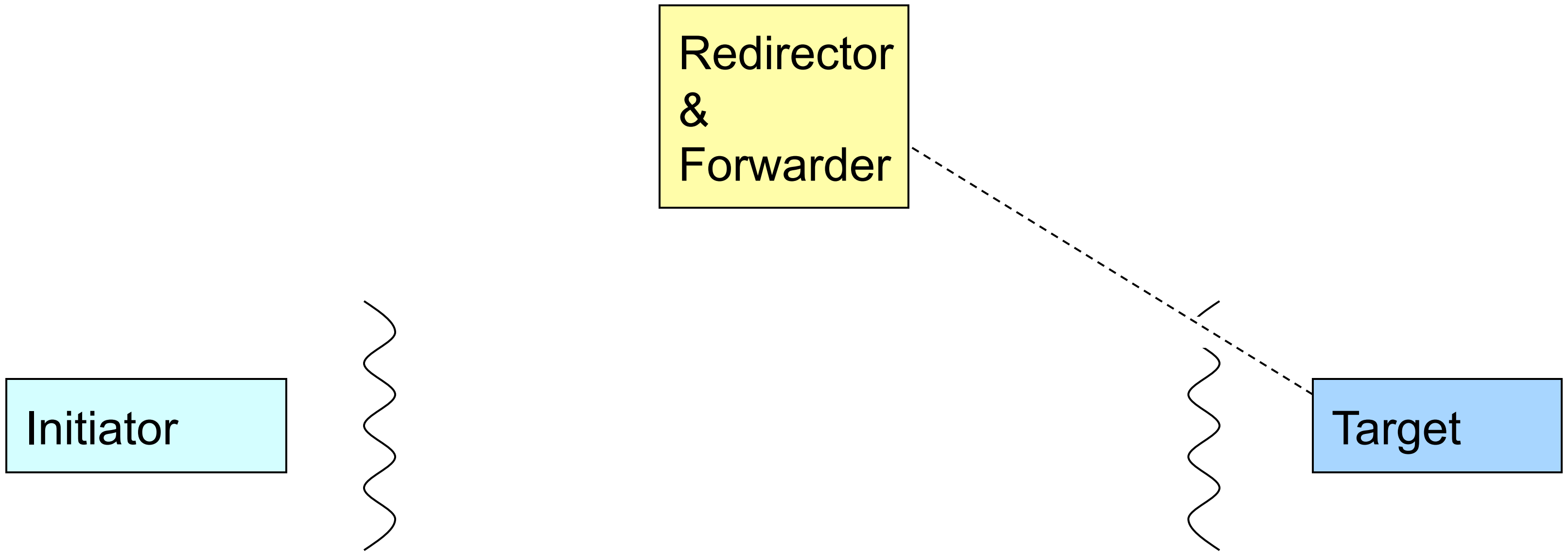
- Target receives forwarded IHello containing Initiator's address and sends RHello
  - This opens the "hole" necessary at the Target's end
  - At Initiator's end "hole" is still open

Session Establishment: NAT traversal



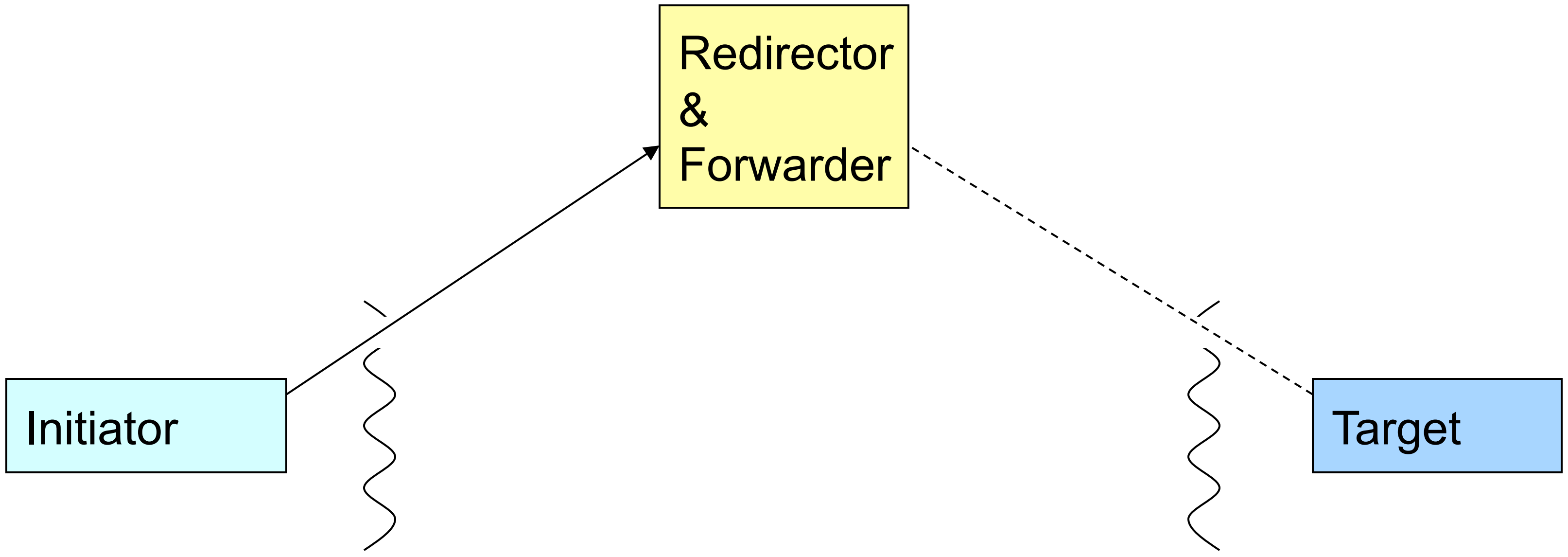
- Session establishment continues

Session Establishment: Lookup and NAT traversal



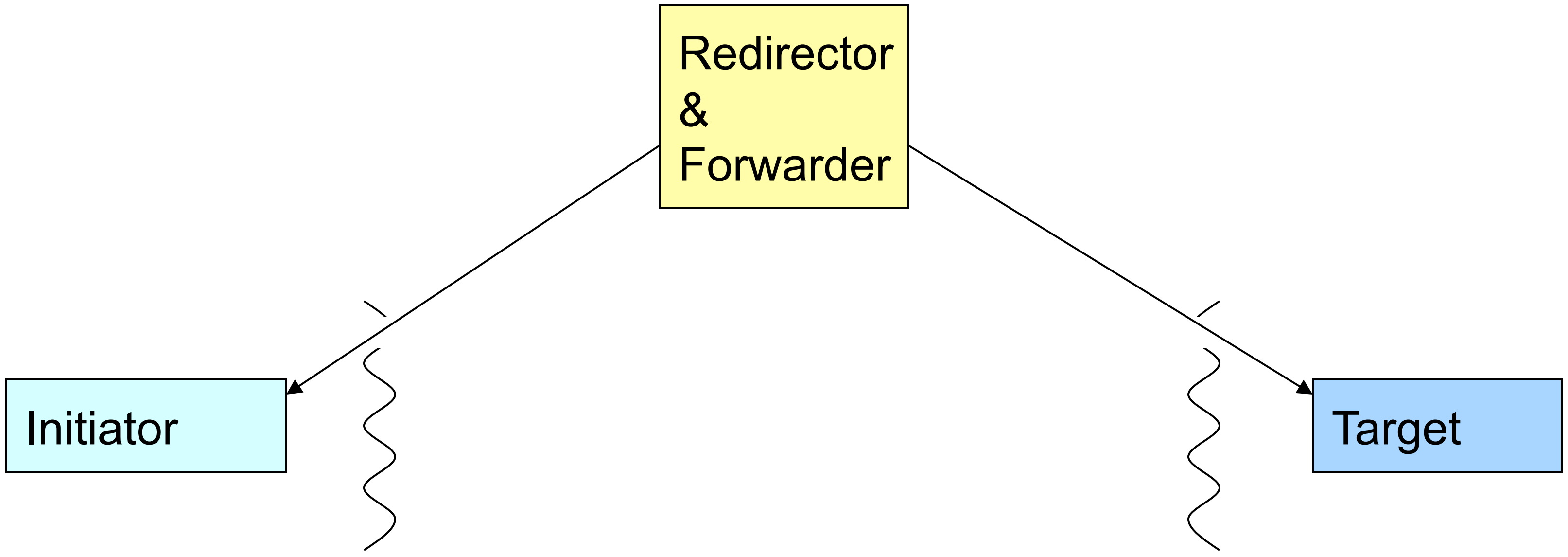
- Initiator and Target are both behind port-restricted (firewalling) NAT
- Target has an open Session to the Redirector-Forwarder host

Session Establishment: Lookup and NAT traversal



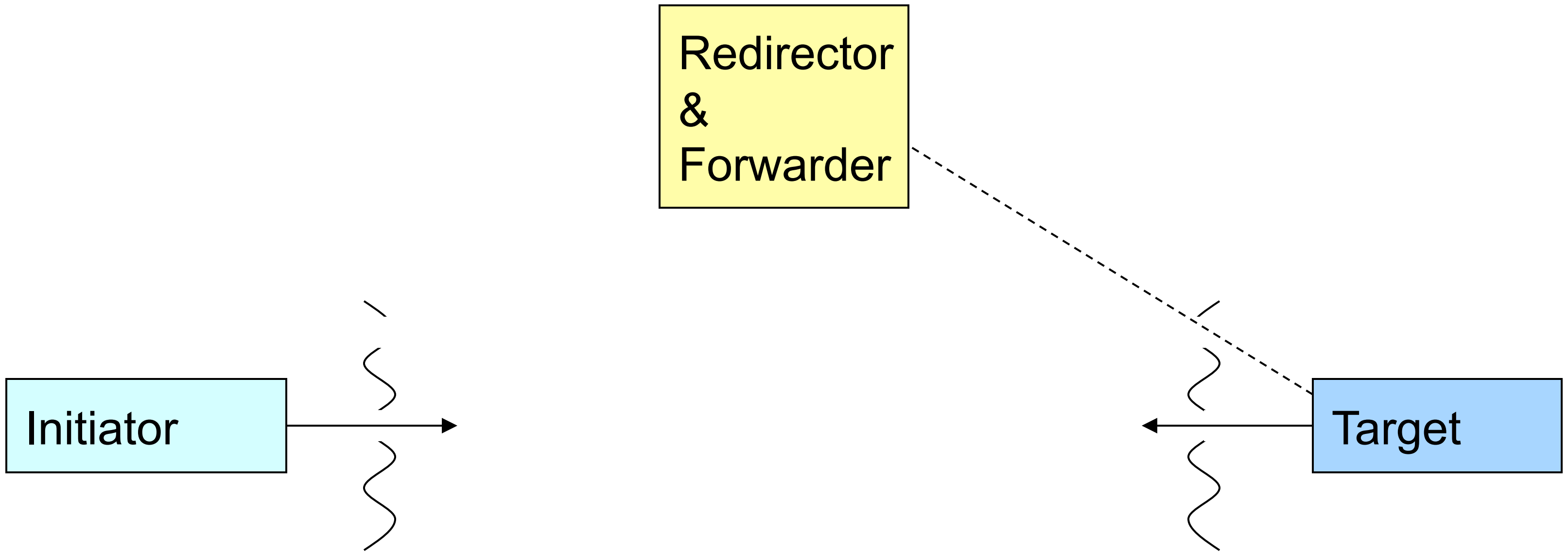
- Initiator tries to open to Target's EPD at Redirector-Forwarder's Address
  - Doesn't need to know Target's address

Session Establishment: Lookup and NAT traversal



- Redirector-Forwarder:
  - Sends Target’s IP address(es) to Initiator as a Redirect message
  - Forwards Initiator’s IHello message (with derived address) to Target over existing Session

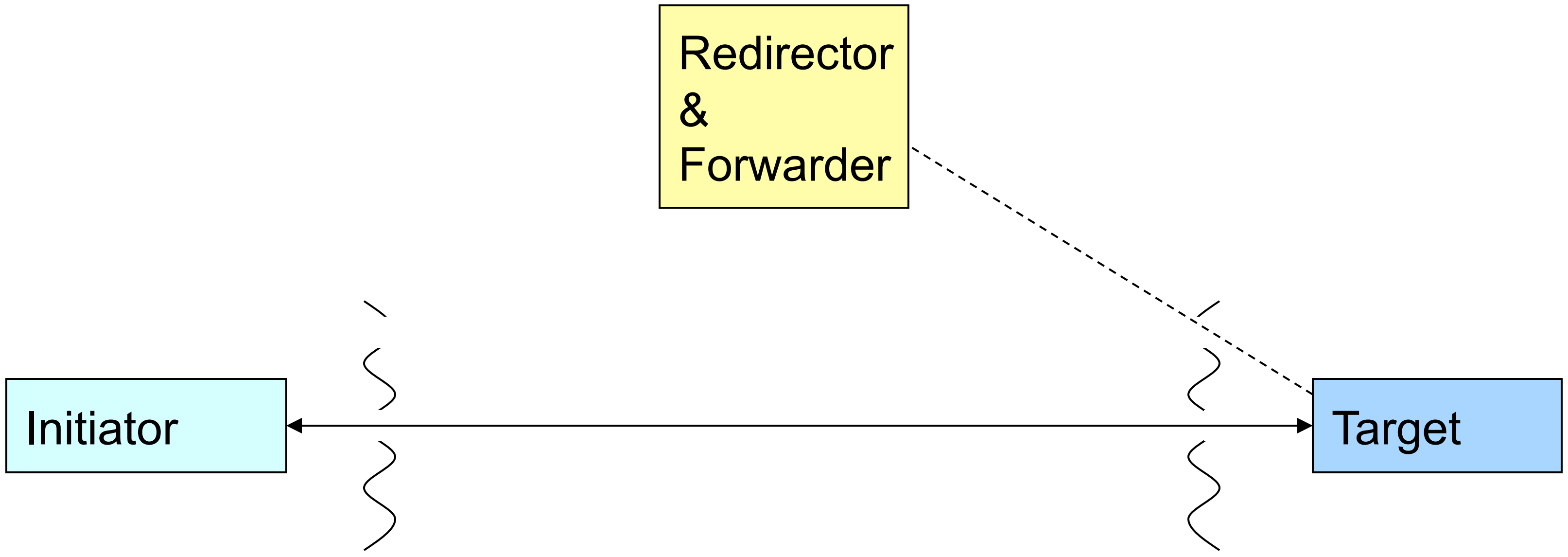
Session Establishment: Lookup and NAT traversal



- Initiator adds Target's address to candidate address list and sends IHello
  - This opens the "hole" necessary at Initiator's end
- Target receives forwarded IHello containing Initiator's address and sends RHello
  - This opens the "hole" necessary at the Target's end

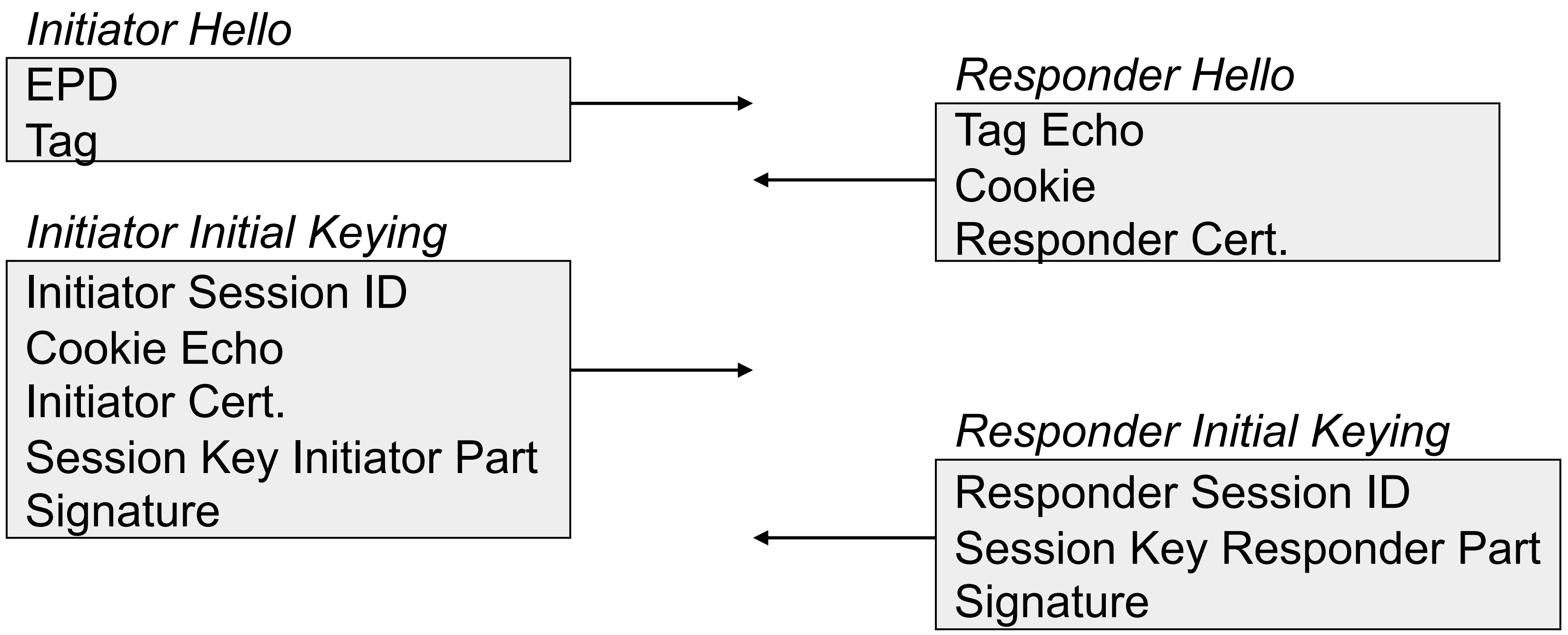
9

Session Establishment: Lookup and NAT traversal



- Session establishment continues

Session Establishment: 4-way handshake

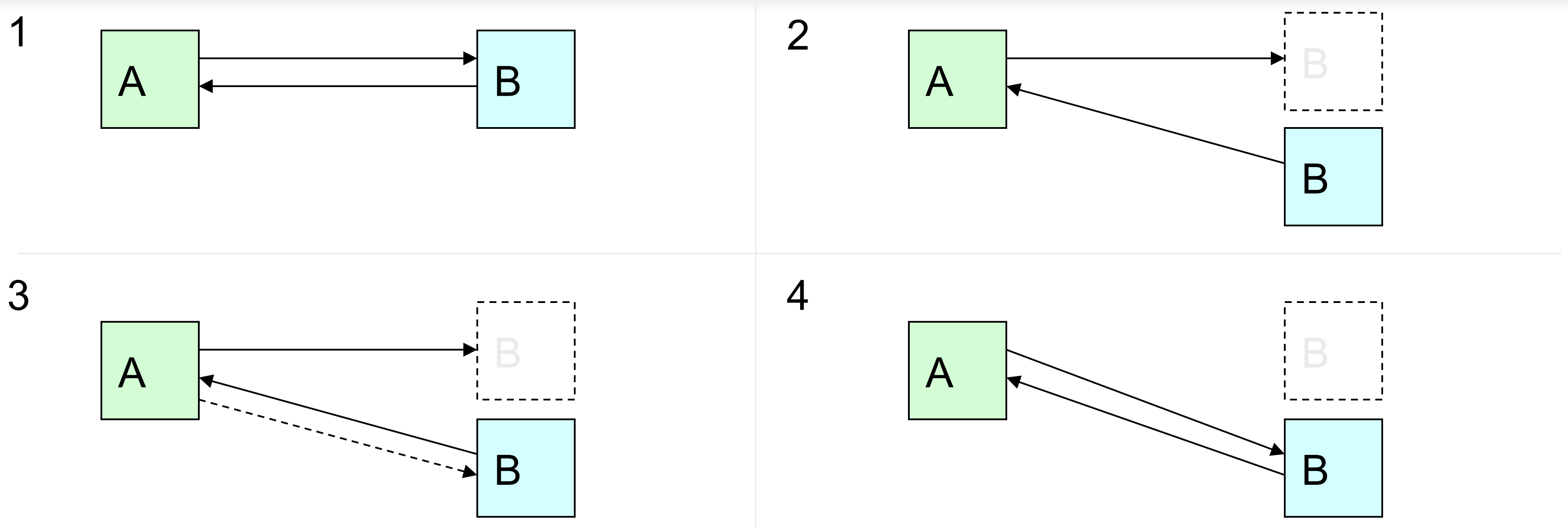


- Takes 2 round trip times

The Solution: How It Works: Sessions

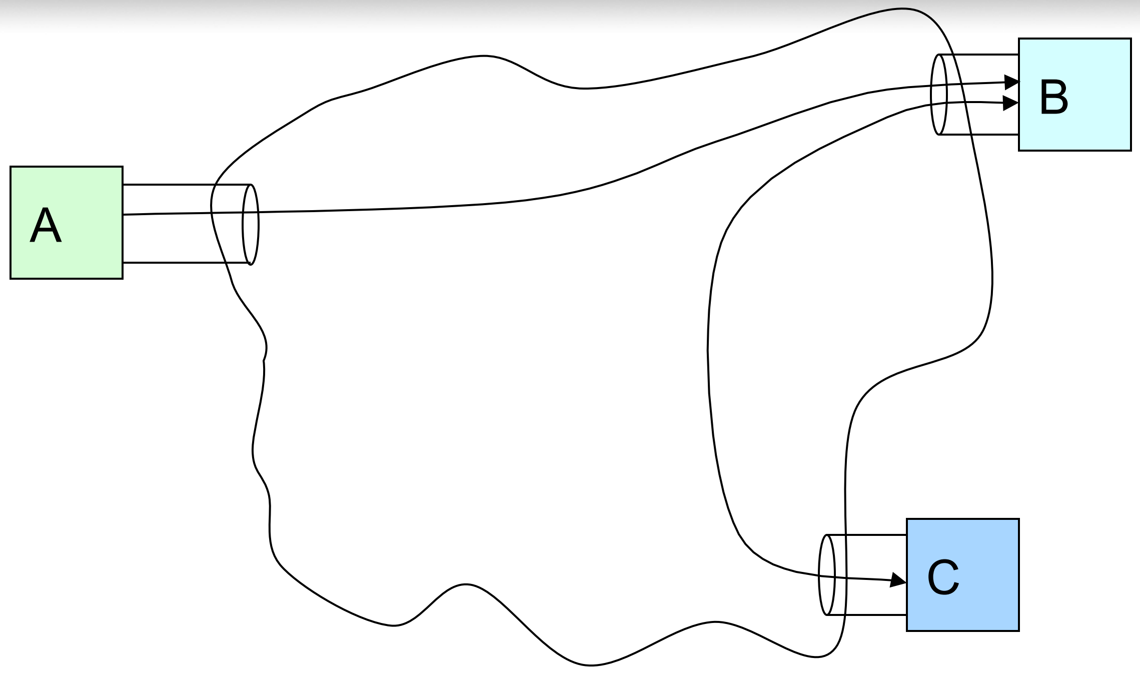
- Once a session is up:
  - RTT regularly measured
  - IP mobility handled as necessary
  - Congestion control
    - Dynamic response
      - Real-time priority traffic causes sender to more smoothly adjust
      - Knowledge that receiver is receiving real-time priority from 3rd party changes response
    - Congestion (packet loss) is determined from data acknowledgements in flows

Session: IP address mobility



- A can determine that B's address has changed (same Session ID, new IP address)
  - No data lost from B to A
- A continues sending data to B's old address, sends probes to B's new address
  - Prevents hijack (otherwise could replay B's data from a new address)
- When probe response is received from B's new address, A switches to sending to new address
  - Data from A to B is lost for one round-trip time to new address
- **Sometimes "mobility" is just "someone rebooted the NAT box"**

Session: Dynamic Congestion Management



- When A is sending “real-time” traffic it sets the “real-time” bit in its outgoing packets to B
  - And also changes its response to packet loss when sending to B to react more smoothly
- As a result B will set the “receiving real-time not from you” bit in its outgoing packets to C
- C will be more “timid” and will react more strongly to packet loss when sending to B
  - To leave bandwidth in B’s incoming network connection, since there’s a nearly 50% chance that is where the congestion is

The Solution: How It Works: Flows

- Flow Establishment
  - No handshake or need to pre-establish
- Sending
  - Framing
  - Each message can have a different reliability
    - None, Full, Partial
    - Forward sequence number
  - Fragmentation
  - Sequence numbers (no wrap)
- Receiving
  - As-received or sequence number ordering (even with partial/no reliability)
  - Acknowledgements sent
    - Notification of packet loss
    - Buffer flow control
- Either end can close (and receiver can reject immediately)
- Return associations

Variable-Length Values

- We use variable-length unsigned integers throughout
  - Moving bits in and out of the CPU or L1 cache is expensive
  - Twiddling bits is cheap
- We require (in most cases) **at least 64 bits** of range
- We never wrap or scale
- Example: sequence numbers
  - The number of bytes grows if there are many
  - But in order to have many, there must be enough bandwidth
  - Allows unambiguous selective acknowledgement no matter what the window size

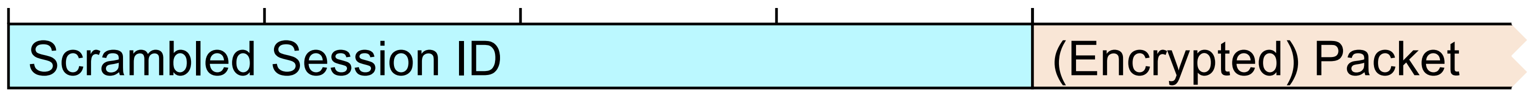
Congestion control

- Required
  - RFC 2914
  - RFC 3714
- Most real-time media doesn't do it
  - Might be ok if your application isn't popular
- Prioritization **requires** it
- Window-based works better than equation-based + fine rate control (e.g., token buckets)
  - Instability from increased time constants (including failure to stop when acks stop)
  - OS timing granularity
- Burst avoidance
  - Send no more than 6 packets per ack, no matter how big CWND is

Security

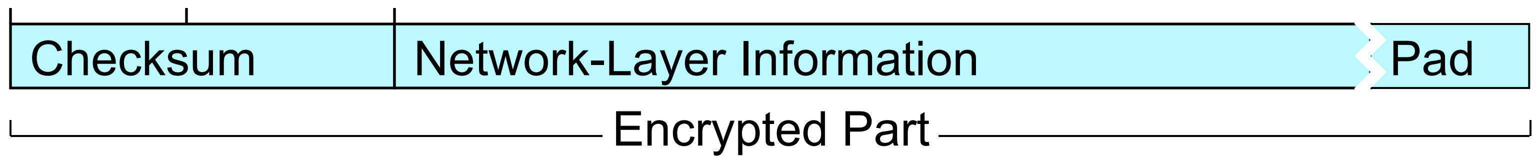
- Always on
- Details are external to protocol
  - But protocol defines how to exchange what is needed
- Protocol features can make security even more important
  - Example: IP address mobility without security is just “session hijacking support”
  - More subtle: Encrypting IP addresses hides them from NAT
    - And a false-positive is ok if the retransmission is encrypted differently

Session multiplexing



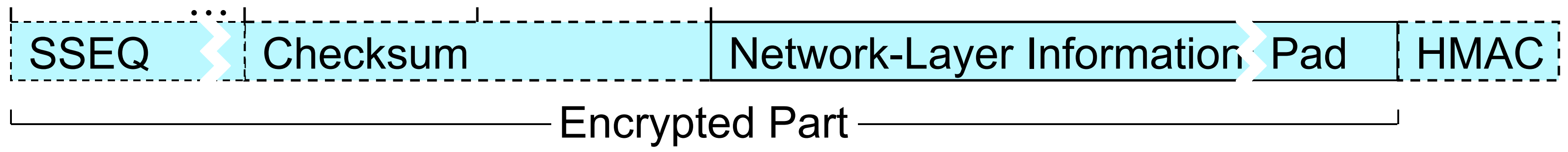
- Don’t rely on source/destination IP and port
- Explicit Session ID
  - But scrambled so that it looks more like noise
    - Avoids NAT false-positives
    - Annoys DPI boxes

Encapsulated Network-Layer Information



- The specific cryptosystem encapsulates the network layer
  - It defines how to encrypt and decrypt everything after the Session ID
  - For convenience, the network layer ignores padding
- The encapsulation is also responsible for data integrity

Encapsulated Network-Layer Information: Flash Player



- In the Flash Player cryptosystem:
  - Integrity is provided by a checksum **or** HMAC (negotiated at startup)
    - The HMAC is outside the encryption
      - Doesn't matter for our block cipher, but does matter for some stream ciphers
  - Session-level sequence numbers are optional (negotiated at startup)

Network-Layer information

- Flags
  - Time-critical Forward notification
  - Time-critical Reverse notification
  - Timestamp Present
  - Timestamp Echo Present
  - Initiator/Responder Mark
    - If a cryptosystem happens to use the same session key in each direction, this protects against reflecting packets back at a sender
- Timestamp (optional)
  - 4 millisecond clock, not opaque (so other end can advance before echoing if needed)
- Timestamp Echo (optional)
- One or more chunks

Chunks

- Tag-Length-Value (Fixed-size tag and length for fast parsing)
- Session Setup
  - Initiator Hello
  - Responder Hello
  - Initiator Initial Keying
  - Responder Initial keying
  - Responder Hello Cookie Change
  - Responder Redirect
- Control (In-session)
  - Ping, Ping Reply
  - Re-keying Initiate, Re-keying Response
  - Close, Close Acknowledge
  - Forwarded Initiator Hello
- Flows (In-session)
  - User Data, Next User Data
  - Buffer Probe
  - User Data Acknowledge (Bitmap), User Data Acknowledge (Ranges)
  - Flow Exception Report
- 0x00 and 0xff are reserved so that either may be used to pad

Chunks: Initiator Hello

- Sent to one or more candidate addresses for Responder
- Contains Endpoint Discriminator and Tag
- Endpoint Discriminator (EPD)
  - Opaque data, understood by cryptosystem
  - Cryptosystem sets up the EPD such that a Responder can tell if this is for them
    - Preferably via a one-way function, so eavesdroppers cannot tell desired identity unless they know it (then they can precompute EPD(s) to match against)
- Tag
  - Opaque data, understood by sender transport protocol implementation
  - Transport can match up a returned tag with an opening session
  - Chosen randomly
- A Forwarded Initiator Hello (from introducer) also contains a sockaddr

Chunks: Responder Hello

- Sent in response to Initiator Hello or Forwarded Initiator Hello with EPD of “this” endpoint
  - If the EPD is for “another” endpoint the Initiator Hello is ignored (no “port scan”)
  - Unless this is an introducer, in which case Responder Redirect is sent instead and/or Forwarded Initiator Hello sent onward
- Contains a Tag Echo, Cookie, Responder Cert
- Tag Echo
- Cookie
  - Generated statelessly, allows Responder to only accept next packet if this was actually received by Initiator. Eliminates “SYN flood” attacks.
- Responder Cert
  - Opaque data, understood by cryptosystem
    - Might have things like an identity, public key

Canonical EPDs

- Note that more than one EPD might map to a single endpoint
- And that endpoint will have a single Cert
- And we don’t want to have >1 session open to a single endpoint
  - If we do, we don’t share the congestion domain, so prioritization is lost
- So we have the concept of a “Canonical EPD”
  - A Cert can be turned into a Canonical EPD
  - When we receive a Responder Hello back at the Initiator we check to see if the computed Canonical EPD matches the Canonical EPD of any existing session
  - If so, we put the opening flows from the opening session onto the already-open session, and stop opening the new session
  - This is one of the reasons why the API is flows, not sessions
    - Another is to make handling Session glare easier… when we detect and resolve glare, we can move the flows to the winning session

Chunks: Initiator Initial Keying

- Contains: Cookie Echo, Initiator Session ID, Initiator Cert, Session Key Initiator Component, Signature
- Cookie Echo
  - Echo of the Cookie provided by Responder. Responder will only process if valid.
  - If Responder thinks it is “partially correct” can request a “cookie change” (e.g., source address changed since cookie generated, but rest of cookie looks good)
- Initiator Session ID
  - Session ID to use – picked by the receiver
- Initiator Cert
  - Opaque data known by cryptosystem, just like Responder cert
- Session Key Initiator Component
  - Opaque data known by cryptosystem. May contain multiple parts (e.g., to negotiate HMAC)
- Signature
  - Opaque data understood by cryptosystem. Cryptosystem asked to compute after all previous parts are computed and serialized.

Chunks: Responder Initial Keying

- Contains: Responder Session ID, Session Key Responder Component, Signature
- Once both end's cryptosystems reach this point:
  - Initiator Component and Responder Component can be combined
  - Block cipher key can be changed
- Transport level:
  - Switches to using newly chosen Session IDs
  - Can begin sending flow data on the newly created Session
- Entire handshake has retransmission rules to deal with lost packets
- Certs and signatures need to be kept compact, as some UDP paths won't fragment and have small MTUs

Chunks: User Data

- API for flows is “per write” not “per byte” framing
- Writes can be fragmented into multiple User Data chunks if needed
- User Data chunk contains:
  - Flags (fragmentation, options-present, abandon, final)
  - Flow ID (variable-length)
  - Sequence Number (variable-length) – each User Data chunk has one unique #
    - Sequence Number (and fragment count) exposed in receiver API
  - Forward Sequence Number Offset (variable-length) – for partially-reliable in-order delivery
  - An optional Option List (some of which may be mandatory-to-understand)
    - Metadata – instead of “well-known ports”, opaque to protocol itself
    - Flow association – to create full-duplex (or more) associations of flows
  - Data

Metadata

- Opaque to protocol
- Sent with each user data until acknowledged
  - Allows a flow to start by just sending data, not waiting for a round trip to “open a flow”
  - Allows flows to start with partially-reliable transmission

Chunks: Next User Data

- Compact form of User Data when multiple User Data are sent in same packet
  - Multiple small writes in same flow before packet dispatch
  - Small fragments when doing PLPMTUD
- Flow ID, Sequence Number, and FSN Offset are implicit, options are usually not required to be duplicated
- A single packet never contains data from more than one flow
  - Priority inversion
  - Head-of-line blocking

Chunks: Acknowledgements

- True selective acknowledgements
  - “No Take-backs”
- Two kinds
  - List of ranges
  - Bitmap
- Both include cumulative ack point and buffer advertisement
- Buffer Probe exists to force an acknowledgement in case transmitter finds that advertised buffer has gone to zero
- Flow Exception exists so that receiver can reject a flow or close it early

Sending Data: Transmit Priority and congestion control

- Multiple priority levels, some considered “real-time”
  - “Real-time” sets the appropriate header bit
- Different AIMD responses
- AIMD responses change if “real-time reverse notification” is being received
- There is “slow start”
- There is “Fast TCP”-style added growth for large delay*bandwidth
- Loss is derived from the selective acknowledgements
  - Duplicate missing ack detection
  - Ultimately there is RTO
    - RTO is capped at a reasonably short value
  - And eventually a session will die if acknowledgements stop
    - Or can be closed via a two-way handshake agreement (this is done when there are no flows on a session for a period of time), or an “emergency” close (stack being shut down)

Sending Data: Partial Reliability

- Each write into a flow can be fully reliable or partially reliable
- Partially reliable means that:
  - It is possible that the resulting User Data chunk(s) may never be sent
  - It is possible that the resulting User Data chunk(s) may be sent, but abandoned (not retransmitted even if not acknowledged) after a period of time
- In-order delivery is supported at the receiver by sending a Forward Sequence Number, advising of the lowest sequence number which might still be retransmitted
  - Sent as an offset from the Sequence Number to keep the variable-length encoding size smaller
- If data flow stops, Forward Sequence Number Update(s) (User Data chunk, but with no actual data) might need to be sent in order to release data at the receiver as data is abandoned

Implementation

- C++
  - ≈13000 lines
  - ≈ 100k
- Largely single-threaded
  - Slow cryptographic operations can be run in separate work-queue thread(s)
- Platform-independent
  - Tested on Win32, MacOS X (PPC, PPC64, x86, x86-64), Linux (x86 and ARM), etc.
- API hides Sessions
  - Application deals only with Flows
- Portable and flexible
  - Cryptosystem and Metadata plug-ins, Platform Adaptor, API Adaptor

RTMFP-based UDP/NAT connectivity test

- Runs in my garage, **not a production service**:
  - http://cc.rtmfp.net
- Establishes an RTMFP session
- Then (using some extra hooks at the server) does what STUN does, in reverse, using existing RTMFP protocol behavior
  - Probes you with IHello and Forwarded IHello and listens for replies
    - 4 UDP ports across 3 IP addresses on one server
  - Tests both mapping and filtering behavior and report results

Adobe®